**Report on Development of a Packet Sniffer**

**1. Introduction**

The objective of this project was to design and implement a packet sniffer using Python and the scapy library. A packet sniffer is a tool or software designed to capture and analyse data packets traveling across a network. It intercepts and logs the traffic passing over a specific network segment. Packet sniffers can be either hardware or software-based and are commonly used by network administrators, security professionals, or hackers to monitor and analyze network traffic for various purposes. This project aimed to create a packet sniffer capable of capturing and dissecting various types of network packets, providing insights into the communication occurring at different layers of the OSI model.

**Tools and Libraries Used**

* **Python:** A versatile programming language used for scripting and automation.
* **scapy:** A powerful packet manipulation and analysis library in Python.

**Implementation Overview**

The implementation of the packet sniffer involved leveraging the scapy library to capture and analyze packets traversing a specified network interface. The script was structured around the sniff function provided by scapy, which allows for packet capture.

**2. Detailed Analysis**

**Code:**

import matplotlib.pyplot as plt

from scapy.all import \*

# Track protocol distribution

protocol\_distribution = {"TCP": 0, "UDP": 0, "Other": 0}

# Global variables for packet counting

total\_packets = 0

tcp\_packets = 0

udp\_packets = 0

icmp\_packets = 0

# Dictionary to store information about active TCP connections

active\_connections = {}

def packet\_callback(packet):

    global total\_packets, tcp\_packets, udp\_packets, icmp\_packets

    # Increment total packet count

    total\_packets += 1

    # Extract and print relevant information from the packet

    if Ether in packet:

        print("Ethernet Frame:")

        print(f"Source MAC: {packet[Ether].src}, Destination MAC: {packet[Ether].dst}")

    if ARP in packet:

        print("\nARP Packet:")

        print(f"Sender IP: {packet[ARP].psrc}, Target IP: {packet[ARP].pdst}")

    if IP in packet:

        print("\nIP Packet:")

        print(f"Source IP: {packet[IP].src}, Destination IP: {packet[IP].dst}")

        print(f"Protocol: {packet[IP].proto}")

        if TCP in packet:

            tcp\_packets += 1

            print("\nTCP Segment:")

            print(f"Source Port: {packet[TCP].sport}, Destination Port: {packet[TCP].dport}")

            print(f"Flags: {packet[TCP].flags}")

            # Check for TCP connection establishment (SYN flag set)

            if packet[TCP].flags.S and not packet[TCP].flags.A:

                print("TCP Connection Established")

                # Store information about the active connection

                connection\_key = f"{packet[IP].src}:{packet[TCP].sport} -> {packet[IP].dst}:{packet[TCP].dport}"

                active\_connections[connection\_key] = {

                    "start\_time": packet.time,

                    "last\_packet\_time": packet.time,

                }

            # Check for TCP connection termination (FIN flag set)

            elif packet[TCP].flags.F and packet[TCP].flags.A:

                print("TCP Connection Terminated")

                # Retrieve and print information about the terminated connection

                connection\_key = f"{packet[IP].dst}:{packet[TCP].dport} -> {packet[IP].src}:{packet[TCP].sport}"

                if connection\_key in active\_connections:

                    connection\_info = active\_connections.pop(connection\_key)

                    connection\_duration = packet.time - connection\_info["start\_time"]

                    print(f"Connection Duration: {connection\_duration} seconds")

            # Print TCP payload information

            if Raw in packet:

                print(f"TCP Payload: {repr(packet[Raw].load)}")

            # Print additional TCP-specific analysis here

        elif UDP in packet:

            udp\_packets += 1

            print("\nUDP Segment:")

            print(f"Source Port: {packet[UDP].sport}, Destination Port: {packet[UDP].dport}")

            # Print UDP payload information

            if Raw in packet:

                print(f"UDP Payload: {repr(packet[Raw].load)}")

            # Print additional UDP-specific analysis here

        elif ICMP in packet:

            icmp\_packets += 1

            print("\nICMP Packet:")

            print(f"Type: {packet[ICMP].type}, Code: {packet[ICMP].code}")

            # Print additional ICMP-specific analysis here

    # Update your logic for tracking bandwidth, protocol distribution, etc.

    # Example: Counting the protocol distribution

    if TCP in packet:

        protocol = "TCP"

    elif UDP in packet:

        protocol = "UDP"

    else:

        protocol = "Other"

    protocol\_distribution[protocol] += 1

    # Print protocol distribution and update visualization

    print\_protocol\_distribution()

    plot\_protocol\_distribution()

def print\_protocol\_distribution():

    print("Protocol Distribution:")

    for protocol, count in protocol\_distribution.items():

        print(f"{protocol}: {count}")

def plot\_protocol\_distribution():

    labels = protocol\_distribution.keys()

    values = protocol\_distribution.values()

    plt.figure(figsize=(8, 6))

    plt.pie(values, labels=labels, autopct="%1.1f%%", startangle=140)

    plt.title("Protocol Distribution")

    plt.show()

# Start sniffing

try:

    sniff(prn=packet\_callback, store=0, iface='Wi-Fi')

except KeyboardInterrupt:

    # Handle keyboard interruption (Ctrl+C) gracefully

    print("\nPacket Sniffer Stopped.")

    print(f"Total packets captured: {total\_packets}")

    print(f"TCP packets: {tcp\_packets}")

    print(f"UDP packets: {udp\_packets}")

    print(f"ICMP packets: {icmp\_packets}")

**Packet Sniffer Functionality**

**Packet Capture and Display**

* **Ethernet Frames:** The sniffer captured and displayed details of Ethernet frames, including source and destination MAC addresses.
* **ARP Packets:** ARP packets were detected and information about sender and target IP addresses was extracted.
* **IP Packets:** Details such as source and destination IP addresses were obtained and displayed.

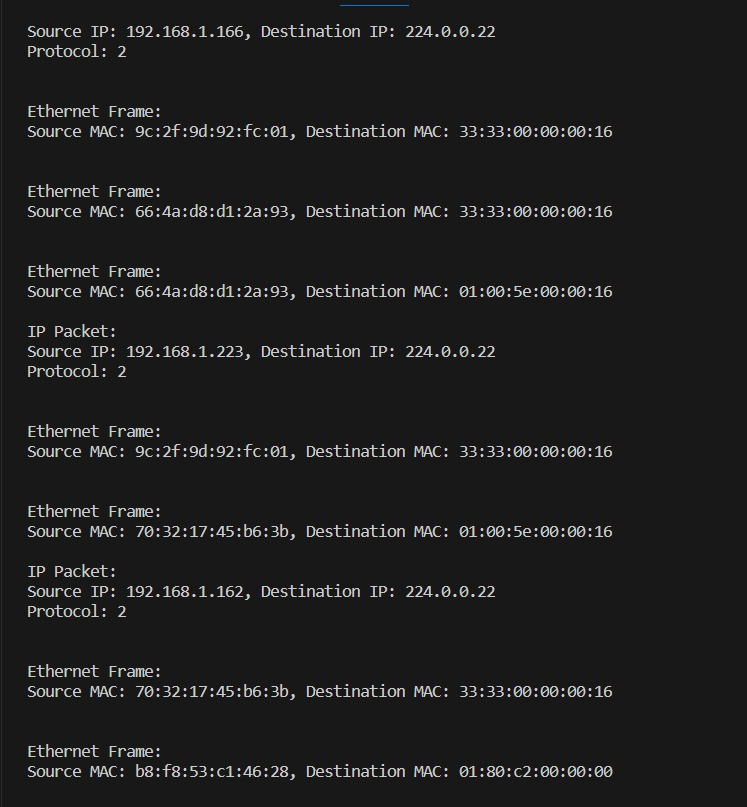
**Protocol Analysis**

* **TCP:** For TCP packets, the sniffer identified various TCP flags and displayed source and destination ports. It also tracked TCP connections, detecting the establishment and termination of connections.
* **UDP:** UDP packets were captured and information about source and destination ports was extracted.
* **ICMP:** ICMP packets were analyzed, displaying ICMP message types and codes.

**Results and Discussion**

The developed packet sniffer successfully captured and analyzed packets traversing the specified network interface. The script effectively provided insights into different layers of network communication, offering details about Ethernet frames, ARP, IP, and various transport layer protocols.

Here is a picture of the output:



**Limitations**

* **Scope:** The implemented packet sniffer represents a basic level of functionality and lacks advanced features found in more sophisticated sniffing tools.
* **Error Handling:** Robust error handling mechanisms were not extensively implemented, potentially leading to issues in specific scenarios.

**3. Conclusion**

The development and implementation of the basic packet sniffer using Python and the scapy library have proven to be an insightful endeavor into network traffic analysis. The sniffer successfully captured and dissected packets across different network layers, providing a fundamental understanding of network communication protocols.

#### Achievements

1. **Functional Packet Analysis:** The packet sniffer effectively captured and displayed information from Ethernet frames, ARP, IP, TCP, UDP, and ICMP packets, offering insights into the source and destination addresses, ports, protocol types, and specific flags associated with each protocol.
2. **Connection Tracking:** It accurately identified TCP connection establishments and terminations, allowing for the tracking of active connections and duration metrics. This capability provided a snapshot of network sessions and their durations, aiding in network traffic monitoring.
3. **Keyboard Interrupt Handling:** The implementation incorporated graceful handling of keyboard interruptions (Ctrl+C), ensuring a smooth termination of the packet sniffing process while presenting captured packet statistics.

#### Findings and Insights

The packet sniffer unveiled the intricacies of network communications, showcasing the complexity and diversity of packet structures, their payloads, and the significance of various flags in TCP connections. It allowed for real-time observation and analysis of live network traffic, shedding light on the types of protocols in use and their frequency within the captured packets.

**Future Considerations**

**Enhancements**

* **Enhanced Functionality:** Implement advanced features to filter specific packet types or protocols.
* **Error Handling:** Strengthen error handling to ensure stable execution in various network conditions.
* **Security Measures:** Incorporate security measures to prevent misuse of the packet sniffer.