**COMP 7005**

**Computer Networks & Protocols**

*Assignment-02*

*Design*

##### Anmol Mittal A01397754

##### February 9th, 2025

##### Course Reference Number (CRN): 92522

[**Purpose** 3](#_Toc189357104)

[**Functions** 3](#_Toc189357105)

[Packet Sniffer (main.py) 3](#_Toc189357106)

[Packet Parser (packet\_parsers.py) 3](#_Toc189357107)

[**States** 4](#_Toc189357108)

[Packet Sniffer (main.py) 4](#_Toc189357109)

[**State Table** 4](#_Toc189357110)

[Packet Sniffer (main.py) 4](#_Toc189357111)

[**Pseudocode** 4](#_Toc189357112)

[Packet Sniffer (main.py) 4](#_Toc189357113)

[Main Execution Block 4](#_Toc189357114)

[capture\_on\_all\_interfaces: 5](#_Toc189357115)

[capture\_packets: 6](#_Toc189357116)

[packet\_callback: 6](#_Toc189357117)

[interface\_is\_loopback: 6](#_Toc189357118)

[has\_global\_ip: 7](#_Toc189357119)

[Packet Parser (packet\_parsers.py) 7](#_Toc189357120)

[parse\_ethernet\_header: 7](#_Toc189357121)

[parse\_arp\_header: 8](#_Toc189357122)

[parse\_ipv4\_header: 8](#_Toc189357123)

[parse\_udp\_header: 10](#_Toc189357124)

[parse\_tcp\_header: 10](#_Toc189357125)

[parse\_icmp\_header: 11](#_Toc189357126)

[parse\_ipv6\_header: 12](#_Toc189357127)

[parse\_icmp\_v6\_header: 13](#_Toc189357128)

[parse\_dns\_header: 14](#_Toc189357129)

# **Purpose**

The purpose of the program is to capture and analyze network traffic at the packet level using Python and Scapy. It will filter packets by protocol (Ethernet, IPv4, ICMP, TCP, UDP, DNS, IPv6, ICMPv6), convert raw packet data into hexadecimal dumps, and parse the packet headers to extract and display key fields such as source/destination MAC and IP addresses, protocol-specific details, and port numbers. The program aims to provide a clear, structured, and human-readable output of packet information.

The program accepts the command line argument as follows:

* sudo python3 main.py -i <interface> -f <filter> -c <count>

-i or --interface: Specifies the network interface to capture packets on. Use any to capture on all available interfaces.

-f or --filter: Specifies the BPF to apply. Common filters include tcp, udp, icmp, and arp. If no filter is provided, the program will capture all packets.

-c or --count: Specifies the number of packets to capture. Default is 1.

# **Functions**

## Packet Sniffer (main.py)

|  |  |
| --- | --- |
| **Function** | **Description** |
| main | Parses command-line arguments and starts the packet capture. |
| capture\_packets | Captures packets on a specific interface. |
| capture\_on\_all\_interfaces | Captures packets on all available interfaces. |
| packet\_callback | Processes each captured packet and extracts header information. |
| interface\_is\_loopback | Check if an interface is a loopback interface. |
| has\_global\_ip | Determines if an interface has a global IP address. |

## Packet Parser (packet\_parsers.py)

|  |  |
| --- | --- |
| **Function** | **Description** |
| parse\_ethernet\_header | Extracts and displays Ethernet header details. |
| parse\_arp\_header | Extracts and displays ARP header details. |
| parse\_ipv4\_header | Extracts and displays IPv4 header details. |
| parse\_udp\_header | Extracts and displays UDP header details. |
| parse\_tcp\_header | Extracts and displays TCP header details. |
| parse\_icmp\_header | Extracts and displays ICMP header details. |
| parse\_ipv6\_header | Extracts and displays IPv6 header details. |
| parse\_icmp\_v6\_header | Extracts and displays ICMPv6 header details. |
| parse\_dns\_header | Extracts and displays DNS header details. |

# **States**

## Packet Sniffer (main.py)

|  |  |
| --- | --- |
| **State** | **Description** |
| START | Initialize arguments and setup capture parameters. |
| SELECT | Choose an interface or capture on all interfaces. |
| CAPTURING | Capture and process packets. |
| STOPPED | Stop capture and processing after reaching the packet limit. |

# **State Table**

## Packet Sniffer (main.py)

|  |  |  |
| --- | --- | --- |
| **From State** | **To State** | **Function** |
| START | SELECT | main |
| SELECT | CAPTURING | capture\_packets |
| CAPTURING | STOPPED | packet\_callback |

# **Pseudocode**

## Packet Sniffer (main.py)

### Main Execution Block

#### Parameters

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Type** | **Description** |
| arguments | list | Command-line arguments passed to the program. |

#### Return

|  |  |
| --- | --- |
| **Value** | **Reason** |
| None | The function initializes and starts execution. |

#### Pseudo Code

parse\_arguments:

Add argument for interface with default "any"

Add argument for filter with no default

Add argument for count with default 1

Parse arguments

validate\_count:

IF count < 0 THEN:

DISPLAY "Error: The packet count (-c) cannot be negative."

EXIT

validate\_filter:

DEFINE allowed\_filters ← ["tcp", "icmp", "udp", "arp", "ip", "ip6", "icmp6", "dns"]

IF filter is not empty THEN:

SET filter\_lower ← lowercase(filter)

IF filter\_lower not in allowed\_filters THEN:

DISPLAY ERROR

EXIT

IF filter\_lower = "dns" THEN:

SET filter ← "udp port 53 or tcp port 53"

prompt\_for\_filter\_if\_needed:

IF filter is empty THEN:

DISPLAY prompt for filter input (tcp, icmp, udp, arp, ip, ip6, icmp6, dns, or all)

SET user\_input

IF user\_input in allowed\_filters THEN:

SET filter ← user\_input

ELSE

DISPLAY "Error: Invalid filter '{user\_input}'. Allowed filters: {allowed\_filters}"

EXIT

validate\_interface:

Professor Provided Function

capture\_on\_all\_interfaces:

#### Parameters

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Type** | **Description** |
| filter | string | The BPF filter to apply to the packet capture. |
| packet\_count | interger | The number of packets to capture before stopping. |

#### Return

|  |  |
| --- | --- |
| **Value** | **Reason** |
| None | Captures packets on all interfaces. |

#### Pseudo Code

Professor Provided Function

### capture\_packets**:**

#### Parameters

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Type** | **Description** |
| interface | String | The network interface to capture packets from. |
| filter | String | The BPF filter to apply to the packet capture. |

#### Return

|  |  |
| --- | --- |
| **Value** | **Reason** |
| None | Captures packets until the stop condition is met. |

#### Pseudo Code

Professor Provided Function

### packet\_callback**:**

#### Parameters

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Type** | **Description** |
| packet | Object | The captured packet object. |

#### Return

|  |  |
| --- | --- |
| **Value** | **Reason** |
| None | Processes a captured packet |

#### Pseudo Code

Professor Provided Function

### interface\_is\_loopback:

#### Parameters

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Type** | **Description** |
| interface | String | The name of the interface to check. |

#### Return

|  |  |
| --- | --- |
| **Value** | **Reason** |
| Boolean | Returns True if the interface is a loop back, else False. |

#### Pseudo Code

Professor Provided Function

### has\_global\_ip:

#### Parameters

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Type** | **Description** |
| interface | String | The name of the interface to check. |

#### Return

|  |  |
| --- | --- |
| **Value** | **Reason** |
| Boolean | Returns True if the interface has a global IP, else False. |

#### Pseudo Code

Professor Provided Function

## Packet Parser (packet\_parsers.py)

### parse\_ethernet\_header:

#### Parameters

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Type** | **Description** |
| hex\_data | String | The raw packet data in hexadecimal format. |

#### Return

|  |  |
| --- | --- |
| **Value** | **Reason** |
| None | Displays parsed Ethernet header details. |

#### Pseudo Code

parse\_header:

destination\_address ← Extract first 12 characters from packet\_data

source\_address ← Extract next 12 characters from packet\_data

ether\_type ← Extract next 4 characters from packet\_data

DISPLAY “Destination Address: "

DISPLAY “Source Address: "

DISPLAY “Protocol Identifier: "

payload\_data ← Extract remaining data from packet\_data

IF ether\_type corresponds to 0806:

CALL parse\_arp\_header and pass on the payload data

ELSE IF ether\_type = "0800" THEN:

CALL parse\_ipv4\_header and pass on the payload data

ELSE IF ether\_type = "08dd" THEN:

CALL parse\_ipv6\_header and pass on the payload data

ELSE:

ERROR

### parse\_arp\_header:

#### Parameters

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Type** | **Description** |
| hex\_data | String | The raw packet data in hexadecimal format. |

#### Return

|  |  |
| --- | --- |
| **Value** | **Reason** |
| None | Displays parsed ARP header details. |

#### Pseudo Code

parse\_arp\_header:

hardware\_type ← Convert first 4 characters of hex\_data to integer

protocol\_type ← Convert next 4 characters of hex\_data to integer

hardware\_size ← Convert next 2 characters of hex\_data to integer

protocol\_size ← Convert next 2 characters of hex\_data to integer

operation\_code ← Convert next 4 characters of hex\_data to integer

sender\_mac\_address ← Extract and format characters 16 to 28 as MAC address

sender\_ip\_address ← Extract and format characters 28 to 36 as IPv4 address

target\_mac\_address ← Extract and format characters 36 to 48 as MAC address

target\_ip\_address ← Extract and format characters 48 to 56 as IPv4 address

DISPLAY "ARP Header:"

DISPLAY “Hardware Type: "

DISPLAY “Protocol Type: "

DISPLAY “Hardware Size: "

DISPLAY “Protocol Size: "

DISPLAY “Operation: "

DISPLAY “Sender MAC: "

DISPLAY “Sender IP: "

DISPLAY “Target MAC: "

DISPLAY “Target IP: "

### parse\_ipv4\_header:

#### Parameters

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Type** | **Description** |
| hex\_data | String | The raw packet data in hexadecimal format. |

#### Return

|  |  |
| --- | --- |
| **Value** | **Reason** |
| None | Displays parsed IPv4 header details. |

#### Pseudo Code

parse\_ipv4\_header:

version ← Convert first character of hex\_data to integer

internet\_header\_length ← Convert second character of hex\_data to integer

type\_of\_service ← Convert next 2 characters of hex\_data to integer

total\_length ← Convert next 4 characters of hex\_data to integer

identification ← Convert next 4 characters of hex\_data to integer

flags\_and\_fragment\_offset ← Convert next 4 characters of hex\_data to integer

CONVERT flags\_and\_fragment\_offset to binary

reserved\_bit ← Extract first bit

df\_bit ← Extract second bit

mf\_bit ← Extract third bit

fragment\_offset ← Convert remaining 13 bits to integer

protocol ← Convert next 2 characters of hex\_data to integer

source\_ip ← Extract and format characters 24 to 32 as IPv4 address

destination\_ip ← Extract and format characters 32 to 40 as IPv4 address

DISPLAY “Version: "

DISPLAY “Header Length: "

DISPLAY "Total Length: "

DISPLAY "Identification: "

DISPLAY "Flags & Fragment Offset: "

DISPLAY "Reserved Bit: "

DISPLAY "DF (Do Not Fragment): "

DISPLAY "MF (More Fragments): "

DISPLAY "Fragment Offset: "

DISPLAY "Protocol: "

DISPLAY "Source IP: "

DISPLAY "Destination IP: "

IF protocol corresponds to UDP:

CALL parse\_udp\_header and pass remaining payload data

ELSE IF protocol corresponds to TCP:

CALL parse\_tcp\_header and pass remaining payload data

ELSE IF protocol corresponds to ICMP:

CALL parse\_icmp\_header and pass remaining payload data

ELSE:

ERROR

### parse\_udp\_header:

#### Parameters

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Type** | **Description** |
| hex\_data | String | The raw packet data in hexadecimal format. |

#### Return

|  |  |
| --- | --- |
| **Value** | **Reason** |
| None | Displays parsed UDP header details. |

#### Pseudo Code

parse\_udp\_header:

source\_port ← Convert first 4 characters of hex\_data to integer

destination\_port ← Convert next 4 characters of hex\_data to integer

length ← Convert next 4 characters of hex\_data to integer

checksum ← Convert next 4 characters of hex\_data to integer

payload ← Extract remaining data from hex\_data

DISPLAY "Parsing UDP Header"

DISPLAY "Source Port: "

DISPLAY "Destination Port: "

DISPLAY "Length: "

DISPLAY "Checksum: "

IF source\_port = 53 OR destination\_port = 53 THEN:

CALL parse\_dns\_header and pass payload

ELSE:

DISPLAY UDP Payload

### parse\_tcp\_header:

#### Parameters

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Type** | **Description** |
| hex\_data | String | The raw packet data in hexadecimal format. |

#### Return

|  |  |
| --- | --- |
| **Value** | **Reason** |
| None | Displays parsed TCP header details. |

#### Pseudo Code

parse\_tcp\_header:

source\_port ← Convert first 4 characters of hex\_data to integer

destination\_port ← Convert next 4 characters of hex\_data to integer

sequence\_number ← Convert next 8 characters of hex\_data to integer

ack\_number ← Convert next 8 characters of hex\_data to integer

data\_offset ← Convert next 1 character of hex\_data to integer

reserved ← Convert next 1 character of hex\_data to integer

flags ← Convert next 2 characters of hex\_data to integer

window ← Convert next 4 characters of hex\_data to integer

checksum ← Convert next 4 characters of hex\_data to integer

urgent\_pointer ← Convert next 4 characters of hex\_data to integer

payload\_offset ← data\_offset \* 4 # Convert from words to bytes

payload ← Extract remaining data starting from payload\_offset \* 2

DISPLAY "Parsing TCP Header"

DISPLAY "Source Port: "

DISPLAY "Destination Port: "

DISPLAY "Sequence Number: "

DISPLAY "Acknowledgment Number: "

DISPLAY "Data Offset: "

DISPLAY "Reserved: "

DISPLAY "Flags: " flags (binary representation)

DISPLAY " NS: "flags [3]

DISPLAY " CWR: "flags [4]

DISPLAY " ECE: "flags [5]

DISPLAY " URG: "flags [6]

DISPLAY " ACK: "flags [7]

DISPLAY " PSH: "flags [8]

DISPLAY " RST: "flags [9]

DISPLAY " SYN: "flags [10]

DISPLAY " FIN: "flags [11]

DISPLAY "Window: "

DISPLAY "Checksum: "

DISPLAY "Urgent Pointer: "

IF payload is not empty:

DISPLAY "TCP Payload: " payload

ELSE:

No TCP Payload

### parse\_icmp\_header:

#### Parameters

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Type** | **Description** |
| hex\_data | String | The raw packet data in hexadecimal format. |

#### Return

|  |  |
| --- | --- |
| **Value** | **Reason** |
| None | Displays parsed ICMP header details. |

#### Pseudo Code

parse\_icmp\_header:

icmp\_type ← Convert first 2 characters of hex\_data to integer

code ← Convert next 2 characters of hex\_data to integer

checksum ← Convert next 4 characters of hex\_data to integer

payload ← Extract remaining data from position 48 onward

DISPLAY "Parsing ICMP Header"

DISPLAY "Type: "

DISPLAY "Code: "

DISPLAY "Checksum: "

IF payload is not empty:

DISPLAY "ICMP Payload: " payload

ELSE:

DISPLAY "No ICMP Payload"

RETURN icmp\_type, code, checksum, payload

### parse\_ipv6\_header:

#### Parameters

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Type** | **Description** |
| hex\_data | String | The raw packet data in hexadecimal format. |

#### Return

|  |  |
| --- | --- |
| **Value** | **Reason** |
| None | Displays parsed IPv6 header details. |

#### Pseudo Code

parse\_ipv6\_header:

version ← Convert first character of hex\_data to integer (base 16)

traffic\_class ← Convert next 2 characters of hex\_data to integer (base 16)

flow\_label ← Convert next 5 characters of hex\_data to integer (base 16)

payload\_length ← Convert next 4 characters of hex\_data to integer (base 16)

next\_header ← Convert next 2 characters of hex\_data to integer (base 16)

hop\_limit ← Convert next 2 characters of hex\_data to integer (base 16)

source\_hex ← Extract next 32 characters from hex\_data

dest\_hex ← Extract next 32 characters from hex\_data

source\_ip ← Format source\_hex into colon-separated IPv6 address

dest\_ip ← Format dest\_hex into colon-separated IPv6 address

DISPLAY "IPv6 Header:"

DISPLAY "Version: "

DISPLAY "Traffic Class: "

DISPLAY "Flow Label: "

DISPLAY "Payload Length: "

DISPLAY "Next Header: "

DISPLAY "Hop Limit: "

DISPLAY "Source IP: "

DISPLAY "Destination IP: "

payload ← Extract remaining data from hex\_data

IF next\_header = 6 THEN:

CALL parse\_tcp\_header and pass payload

ELSE IF next\_header = 17 THEN:

CALL parse\_udp\_header and pass payload

ELSE IF next\_header = 58 THEN:

CALL parse\_icmp\_v6\_header and pass payload

ELSE:

ERROR

### parse\_icmp\_v6\_header:

#### Parameters

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Type** | **Description** |
| hex\_data | String | The raw packet data in hexadecimal format. |

#### Return

|  |  |
| --- | --- |
| **Value** | **Reason** |
| None | Displays parsed ICMPv6 header details. |

#### Pseudo Code

parse\_icmp\_v6\_header:

icmp\_v6\_type ← Convert first 2 characters of hex\_data to integer (base 16)

icmp\_v6\_code ← Convert next 2 characters of hex\_data to integer (base 16)

icmp\_v6\_checksum ← Convert next 4 characters of hex\_data to integer (base 16)

payload ← Extract remaining data from hex\_data

DISPLAY "ICMPv6 Header:"

DISPLAY "Type: "

DISPLAY "Code: "

DISPLAY "Checksum: "

DISPLAY "ICMPv6 Payload"

### parse\_dns\_header:

#### Parameters

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Type** | **Description** |
| hex\_data | String | The raw packet data in hexadecimal format. |

#### Return

|  |  |
| --- | --- |
| **Value** | **Reason** |
| None | Displays parsed DNS header details. |

#### Pseudo Code

parse\_dns\_header:

dns\_id ← Convert first 4 characters of hex\_data to integer (base 16)

dns\_flags ← Convert next 4 characters of hex\_data to integer (base 16)

dns\_question\_count ← Convert next 4 characters of hex\_data to integer (base 16)

dns\_answer\_count ← Convert next 4 characters of hex\_data to integer (base 16)

dns\_rr\_count ← Convert next 4 characters of hex\_data to integer (base 16)

dns\_additional\_rr\_count ← Convert next 4 characters of hex\_data to integer (base 16)

DISPLAY "DNS Header:"

DISPLAY "Transaction ID: "

DISPLAY "Flags: "

DISPLAY "Questions: "

DISPLAY "Answer RRs: "

DISPLAY "Authority RRs: "

DISPLAY "Additional RRs: "