

# **1 – Title Page**

Project: Basic Packet Sniffer

Author: Cameron Bell (cjb15@hw.ac.uk)

Supervisor: Mike Just (m.just@hw.ac.uk)

Final Year Dissertation

Bachelor of Science in Computer Science

## **2 – Declaration**

I, Cameron Bell confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (e.g., ideas, equations, figures, text, tables, programs) are properly acknowledged at any point of their use. A list of the references employed is included.

Signed: 

Date: 26/04/2023

### **3 – Abstract**

Networks are an essential part of our modern, computer based, society. Whether they be a houses local network or the world wide web issues may arise so having tools to test them is essential. One method of this is packet sniffing which can be used to record the traffic being sent across a network. The intention of the project is the investigation of existing packet sniffing tools followed by the planning and development of a new packet sniffing tool based on a perceived gap not current fulfilled. With a plan for a packet sniffing tool focused on the processing and generating of the packet information as charts and analysis documents.

## **4 – Table of Contents**

1 – Title Page.....	1
2 – Declaration.....	2
3 – Abstract.....	3
4 – Table of Contents.....	4
5 – Introduction.....	8
5.1 – Context.....	8
5.2 – Objectives.....	8
5.3 – Problems.....	8
5.4 – Methods.....	9
5.5 – Achievements.....	10
6 – Background.....	11
6.1 – Important terms.....	11
6.2 – Packet Sniffers.....	11
6.3 – Tcpcap.....	12
6.4 – Wireshark.....	12
6.5 – PktMon.....	13
6.6 – Ettercap.....	13
6.7 – Network Miner.....	13
6.8 – PRTG Network Monitor.....	13
6.9 – Comparison.....	14
6.10 – Packet Sniffers Conclusion.....	15
6.11 – Tool Technicals.....	16
7 – User Guide.....	18
7.1 – Overview.....	18

7.2 – Setup.....	18
7.3 – Sniffing Mode.....	19
7.4 – Loading Mode.....	20
8 – Developer Guide.....	22
8.1 – Main.py.....	22
8.2 – sniffer_func.py.....	23
8.3 – log_loader_func.py.....	24
8.4 – output_gen_func.py.....	25
9 – Testing.....	29
9.1 – Sniffing and Printing test.....	29
9.2 – Sniffing and Chart Generation.....	30
9.3 – Filtered Sniffing.....	31
9.4 – Specific IP output.....	32
9.5 – Sniffing Without Administrator Privileges.....	33
9.6 – Setting invalid Max packets/entries.....	34
9.7 – Setting invalid file to load.....	35
9.8 – Load file with non-existent specified IP address.....	36
9.9 – Invalid Filter.....	36
9.10 – Setting valid Max packets/entries.....	37
9.11 – Sniffing with no internet connection.....	37
9.12 – Loading with no internet connection.....	38
10 – Conclusions.....	39
10.1 – Functional Requirements.....	39
10.2 – Non-Functional Requirements.....	40
10.3 – Requirements Reasoning.....	40
10.3 – Requirements Not Met.....	41

10.4 – Difficulties Encountered.....	42
10.5 – Project Going Forward.....	44
11 – References.....	47
12 – Appendices.....	49
12.1 – Figure 1.....	49
12.2 – Figure 2.....	50
12.3 – Figure 3.....	51
12.4 – Figure 4.....	52
12.5 – Figure 5.....	53
12.6 – Figure 6.....	54
12.7 – Figure 7.....	55
12.8 – Figure 8.....	56
12.9 – Figure 9.....	57
12.10 – Figure 10.....	58
12.11 – Figure 11.....	59
12.12 – Figure 12.....	60
12.13 – Figure 13.....	61
12.14 – Figure 14.....	62
12.15 – Figure 15.....	63
12.16 – Figure 16.....	64
12.17 – Figure 17.....	65
12.18 – Figure 18.....	66
12.19 – Figure 19.....	67
12.20 – Figure 20.....	68
12.21 – Figure 21.....	69
12.22 – Figure 22.....	70

12.23 – Figure 23.....	71
12.24 – Figure 24.....	72
12.25 – Figure 25.....	73
12.26 – Figure 26.....	74
12.27 – Figure 27.....	75
12.28 – Figure 28.....	76
12.29 – Figure 29.....	77
12.30 – Figure 30.....	78

## **5 – Introduction**

### **5.1 – Context**

Packet sniffers are incredibly useful tools when it comes to monitoring networks and diagnosing faults. As they allow for the monitoring and recording of packets of data sent across a network. They can be set to record data only for a specific device or all data sent for all devices across a network. They can record all packets or be filtered to only collect packets that meet specific conditions. The data collected from then can be analysed by network administrators to find analyse bandwidth usage and which devices use the most, test if network connections are functional or have gone down or even to monitor for suspicious activity. They could even be used by malicious user to find confidential data in the contents of the packets such as passwords, emails or images.

### **5.2 – Objectives**

The objective of this project was to analyse existing packet sniffing tools to create a packet sniffing program. Specifically it was decided to develop one with a focus on the processing of collected IP packet data into a medium that is easier for a user to understand via charts/graphs and analysis documents.

### **5.3 – Problems**

Problems that would requiring solving would be:

- Packet sniffing, the collecting of all packets received by a host machine across all



network interfaces, and producing of a log of all packets intercepted on a network.

- Compiling of this data into a versatile data structure, one that can be used to easily process that data and return the required data points.
- Extracting of useful data points from the data, using both internal conditional logic of the program to derive data points as well as external tools such as IP lookup services.
- Production of graphs, charts and documents utilising the returned data points to allow the data to be viewed and understood an easier and more visual way.

## 5.4 – Methods

The following methods were used to solve these problems:

- The program was created using python as the primary programming language.
- The Scapy package was used for the packet sniffing component allowing packets to be intercepted and a packet log generated.
- The Pandas package data frames were used as the main data structure to hold the data retrieved from the packet log.
- Useful data points were extracted using python conditional statements and iterative loops along with useful commands along with commands from pandas to retrieve specific data rows meeting specific requirements.
- The web service ipinfo.io was used to look up information regarding IP addresses.
- The Mathplotlib package can be used to generate graphs and maps using the data points extracted from the packet logs.
- The python-docx package can be used to generate text documents to hold the analysis report.

## 5.5 – Achievements

The program can be run in sniff mode to sniff packets over a network, intercepting all packets on all network interfaces including ones not intended for it. This sniffing can continue for as long as the user wants or until a specified packet limit is reached. The program can be set with a filter to only log specific packets and can be set to print each packet on interception. Once complete the packets will then be logged in a pcap file and saved. The program can then be run in load mode where a pcap file can then be loaded by the program which will then be processed into a pandas dataframe, with each packet as a new row. The program will then return data frame rows with specific values in specific columns and use ipinfo to return info on specified IP addresses. These values can then be returned to the user in the form of source/destination packet IP and port bar charts, location maps and an analysis document listing some of the IP addresses along with the other IP addresses and ports then sent and received packets from along with information about each IP address. The user also specify an IP to which all the generated charts will be specifically about such as the source IP chart counting only IP address that sent packets to the specified IP address. The user can also set how detailed these are by specifying the top however many IPs and ports to show in the charts/document.

This is a potentially useful tool to allow a user to scan a network, recording the packets of network traffic and allowing the visualisation of this data. Seeing visually which IPs have been sending and receiving the most packets across the network and where in the world could be more helpful than a raw list of all the packets. Additionally the tool is simple to use only requiring one command to begin the packet sniffing. After that only one additional command is required to generate all the charts, maps and document.

## **6 – Background**

### **6.1 – Important terms**

“LAN” – “Local Area Network”

“NIC” – “Network Interface Card”

“MAC” – “Media Access Control”

“Sniffer” – “Machine/Device running packet sniffer program”

“CLI” – “Command Line Interface”

“GUI” – “Graphical User Interface”

“IP” – “Internet Protocol”

“UI” – “User Interface”

“ARP” – “Area Resolution Protocol”

“BPF” – “Berkley Packet Filters”

### **6.2 – Packet Sniffers**

Packet sniffing is a method of passively intercepting packets of data across a network, a packet sniffer is simply a program that preforms this. These are often used by network administrators, or malicious users, to monitor the traffic on a network. This can be for any reason such as troubleshooting network problems, monitoring bandwidth usage or intercepting data packets holding critical data such as passwords. Typically the program will be run on a machine connected to a network, the sniffer, then whenever a data packet are sent or received by that NIC it will be logged by the program. If used on a network such as a LAN where each packet is sent to every other device on the network then any packets not intended for the machine running the packet sniffer can be logged. Under normal

circumstances any packets not intended for a machine would be discarded by the NIC, it would see that the destination MAC address of the packet does not match the MAC address of the machine. However, packet sniffers will often put their devices NIC into what's known as "promiscuous mode" where any packets not intended for them will still be accepted and logged silently without announcing it has been accepted. [1], [2]

There already exist many different packet sniffing tools, each with different features and strengths.

### **6.3 – Tcpdump**

Free, open source, CLI based packet sniffer known for being light weight and simple to use. One of the most used tools in the industry. When run it displays each sent or received packet on a new line showing the: timestamp, source, destination and other information to explain its purpose. An example can be seen in fig. X. Different options can be entered when running the 'Tcpdump' command to change the data shown or to filter which packets are shown. This tool is simple, allowing for the quick and easy viewing and logging of the packet data as that is its main purpose. See figure 28 in the appendix for a demonstration. [3]–[5]

### **6.4 – Wireshark**

A free, open source, GUI based packet sniffer (with a CLI option) and also a very popular packet sniffing tool. It can scan other other forms of networks such as Bluetooth. Its GUI interface allows for easy viewing of each packet recorded in a list box, allowing the user to select each one to view more information including the contents of the packets. It has

options to print and analyse data, viewing statistics. It has a very robust filter system. This tool is still fairly simple to use despite it's additional options. See figure 29 in the appendix for a demonstration. [4], [6]–[8]

## **6.5 – PktMon**

A CLI based packet sniffer, comes pre-packaged with the latest version of windows. Similar to Tcpdump in that it is simple light-weight, displays each packet as a new line in the CLI, basic filter etc... [9]

## **6.6 – Ettercap**

A free GUI and CLI based tool that can be used as a packet sniffer, however it is intended to be used in 'man in the middle attacks'. It allows for the intercepting of data from a specific host, where the data packets can then viewed or hijacked via an attack such as ARP posing. [8], [10], [11]

## **6.7 – Network Miner**

A free with paid option, open source, GUI based packet sniffer. Designed around digital forensics with features such as automatic extracting images, files and emails from the contents of the packets and not for viewing each individual packet like Tcpdump or Wireshark. [12], [13]

## **6.8 – PRTG Network Monitor**

A GUI based network monitoring tool, it is robust with many features and ways of collecting

network data including packet sniffing. It can be set up to monitor multiple different connections and then use the collected data to produce detailed report or various graphs. This software main version costs with the cheapest version being over £1000. When compared to previous tools this is significantly more complicated and would require the user to take significant time to learn it's systems. It also has many options unrelated to packet sniffing which while they could be useful, for a user that just wants the packet sniffing functionality it may be unnecessary confusion. See figure 30 in the appendix for a demonstration. [14]–[17]

## 6.9 – Comparison

A compare some of the previously mentioned points:

- Tcpdump, PktMon are CLI based. (Wireshark and Ettercap have CLI options).
- Wireshark, Network Miner, Ettercap, PRTG Network Monitor are GUI based.
- Tcpdump, Wireshark, Ettercap, Network Miner are all free and open source.
- PRTG Network Monitor is propriety and costs.
- PktMon is propriety and comes pre-packaged with windows.
- Tcpdump, Wireshark, PktMon, Network Miner are all designed around packet sniffing as their primary function.
- Ettercap is designed more around 'man in the middle attacks'.
- PRTG Network Monitor is designed for monitoring networks and systems via many different means.
- Tcpdump and PktMon are basic packet sniffers with basic filtering options.

- Wireshark is more advanced, allows for much more in-depth analysis of individual packets, view some statistics of the data as a whole as well as a robust filter system.
- Ettercap allows for basic viewing of packets and their contents from a specific host, as well allowing for an attack such as an ARP posing attack to hijack the packets.
- Network Miner allows for the automatic extraction of files, such as images, from packets but isn't as good for individual packet analysis.
- PRTG Network Monitor is very robust containing many features for passive network monitoring and analysis including packet sniffing. This makes the tool significantly more complicated to other tools.

## 6.10 – Packet Sniffers Conclusion

With this quick look over some packet sniffing tools some deductions can be made.

Tools such as Tcpdump/PktMon are very simple run one command to start the programs packet sniffing but are also very simple in terms of features. Wireshark poses more features such as a more user friendly UI and some data analysis tools. PRTG network monitor has lots of analysis options as well as options to creates graphs/charts visualising the data. However PRTG possesses lots of features unrelated to packet sniffing and is overall a lot more complicated requiring more work to set up, it also cost which would make it unacceptable to users not in a sufficiently sized company.

This would seem to leave space for a tool, that is simple to load, run and is focused on packet sniffing while also having the data analysis, report generation and graph/chart generation to visualise the data. All this while remaining free and open source.

Additionally while Network Miner has options to modify packets to preform man in the middle

attacks a similar thing could be employed for testing networks. The program could possess the ability generate and send packets, the response packets could then be logged to measure delay, packet-loss or even if a website goes down.

## 6.11 – Tool Technicals

Python has been chosen as the programming language to develop this tool as it is an easy to use yet versatile language with many already developed packages/libraries to use. [18] These existing libraries is particularly useful as it significantly cuts down on the work required.

Two machines communicate over a network via sockets, when a program wants to send or receive it must create a socket, which can then be used to create or receive the packets.

There are different ways the crating and use of sockets could be handled:

- 'socket', which is a library dedicated to low level networking that allows for the creation of sockets. This way would require the most work manually writing code to handle the sockets and other aspects of low level networking. [19], [20]
- 'libpcap', which is a python library based on the C language 'libpcap' language which is used by both Tcpdump and Wireshark. This allows for more high level programming as the libpcap functions will be able to handle that. The python libpcap doesn't seem to be too widely used as there isn't as much documentation. [5], [21]
- 'scapy', which is a python program designed to manipulate packets. Similar to libpcap it has high level functions to handle most of the low level networking. It is robust enough that it can be used on it's own and as a CLI packet sniffer. [22], [23]

While working with socket would allow for more direct and specific control over the sockets the easiest option would be scapy as using it's already existing functions could handle all the



low level networking would resulting in more time to work on the analysis aspect of the program.

Scapy also allows for easy filtering of packets during sniffing by accepting an expression in the BPF syntax, this will allow for the user to use an existing syntax which scapy will handle it's self. [24], [25]

Scapy runs natively on Linux without issue however for it to work on Windows the program Npcap will have to be installed. Npcap is a library that can be used for packet capture and sending that replaced WinPcap, it is similar to libpcap on Linux. [26], [27]

When compiling the packet log into data to be analysed by the user the pandas library could be used, as it allows for fast and easy handling and manipulation of data. [28]

'matplotlib' can be used to visualises this data, as it allows for the creation of graphs and charts of differing styles, layouts and formats. [29]

'tkinter' can be used to generate a GUI for the program allowing the user to setup how the program is to run as well as to display the charts generated by matplotlib and possibly display the packet log. [30]

'python-docx' could be used to generate docx text files, this could be used to create the analysis document. [31]

The web service 'ipinfo.io' and it's python package 'ipinfo' can be used to retrieve information about IP addresses which the program can then use. [32], [33]

## **7 – User Guide**

A copy of the user guide can be found in the README.md file in the project folder (BasicPacketSniffer).

### **7.1 – Overview**

This tool is used via a command line, by using a command to run the 'Main.py' python script.

In the following guide the given commands were written for a Windows machine command line running Python 3.10.2, which uses the 'python' command.

The commands for other systems such as Linux machines may differ, such as using 'python3' instead.

Administrator privileges are required for some features of this tool as such the command line should be launched in administrator mode or each command should be prefixed with sudo (if on Linux).

An internet connection is also required to use certain features.

### **7.2 – Setup**

First python must be installed as this program is a python script (<https://www.python.org/downloads/>).

Then if being set up on Windows then Npcap must be installed to allow it to work (<https://npcap.com/#download>), otherwise if on Linux this can be ignored.

Then to install all the required python packages the user should open the project folder in the command line and use the requirements.txt file to install every specified package. The requirements text file contains the name of every required package. An example of this being done with PIP is:

```
python -m pip install -r requirements.txt
```

Once this is done they can access the scripts in the 'src' folder

## 7.3 – Sniffing Mode

Sniffing mode allows the user to begin sniffing packets on the network, it must be run in administrator mode to function.

To launch the program in sniffing mode use the '-s' argument followed by the name of the pcap file to be saved afterwards:

```
python Main.py -s "Example"
```

This would begin sniffing until the user interrupts with 'Ctrl + C' keys at which point the sniffing would end and would save the intercepted packets as 'Example.pcap'.

The user can have the program print each pack intercepted in real time by including the '-p' argument:

```
python Main.py -s "Example" -p
```

The user can have the program sniff until a specified amount of packets have been intercepted by including the `-m` command followed by a positive integer:

```
python Main.py -s "Example" -m 20
```

This would sniff until 20 packets had been intercepted.

The user can filter the packets that are recorded by the sniffer with the `-f` argument followed by a string expression in the Berkeley Packet Filter (BPF) syntax format:

```
python Main.py -s "Example" -f "src host 51.11.122.226 and tcp"
```

This would only intercept packets that has '51.11.122.226' as it's source IP address and has tcp as its protocol.

## 7.4 – Loading Mode

Loading mode allows the user to load an existing pcap file to be processed.

To launch the program in loading mode use the `-l` argument followed by the name/location of the pcap file to be loaded afterwards:

```
sudo python3 Main.py -l "Example.pcap"
```

This will then load and process the selected pcap file, it will then create a folder called "Example.pcap outputs", here it will save some of the charts. During this process it will connect online to an IP lookup service, if there is no internet connection then this stage will be skipped, the IP maps and document will then be saved.

The user can set the maximum number of entries that will appear in the bar charts and document b using the `-m` argument followed by a positive integer:

```
python Main.py -l "Example.pcap" -m 20
```

In this example the returned charts and document will only show the top 20 Addresses and ports in the charts/document. If this is not set then it will default to 50 to ensure that the returned charts/document are legible.

The user can specify an IP address to generate the charts/document on. Instead of showing a chart of all source IP addresses it will only count cases where the source IP sent packets to the specified IP address, vice versa for only counting destination addresses that received packets from the specified IP address. It is used with a '-i' argument followed by an IP address:

```
python Main.py -l "Example.pcap" -i "10.0.2.4"
```

In this example the charts/map/document will only show information pertaining to the IP address "10.0.2.4".

## **8 – Developer Guide**

The main work carried out for this project was the planning and development of this tool, it consists of 4 python scripts.

### **8.1 – Main.py**

This is the main script and is the one run by the user.

When run the main() function is run parsing any arguments the user provided in the command.

Then using the 'getopt' package the arguments entered are processed and used to tell the program how to run.

If the user gave the '-h' argument then the script runs the 'printhelp()' function to return the help message and stops running.

There is then a conditional check for:

- If the user gave the '-s' argument then the program is run in sniffing mode, the script runs the 'packetsniff()' function in the sniffer\_func.py script and parsing the given filename, print during option, filter expression and max packet number.
- Else if the user gave the '-l' argument then the program is run in load mode, script runs the 'loadpcapfile()' function in the 'log\_loader\_func.py' script parsing the name/location of the file to be loaded, it then saves the returned pcap file. It then runs the 'loadpcapfile()' function in the 'log\_loader\_func.py' script parsing the saved pcap

file, it then saves the returned pandas dataframe.

- Else if neither was entered then then print the help message.

Additionally there are other arguments that can be given:

- The '-f' argument can be given after the '-s' argument and accepts a string to act as the filter expression, it is given in the Berkeley packet filters (BPF)
- The '-p' argument can be given after the '-s' argument and flips the 'printduring' variable to true, this is then sent to the 'packetsniff()' function in 'sniffer\_func.py' script and makes it print each packet as they're intercepted during the sniffing process.
- The '-i' argument can be given after the '-l' argument and is followed by a string, this string specifies an IP address which is saved in 'ipadress'. This is then sent to 'graphgen()' in 'output\_gen\_func.py' and the graphs/report outputed is specifically about that IP address instead of a generic report about the whole packet log.
- The '-m' argument is followed by a positive integer, it dose differing thing depending on the mode:
  - After the '-s' argument, this integer specifies the maximum number of packets to sniff, ones reached the sniffing process ends.
  - After the '-l' argument, this integer specifies the maximum number of entries to be displayed in the charts/doc, this is done as too many entries being shown can result in the charts/doc being unreadable. By default this is set to 50, from now on this number will be referred to as **N**.

## 8.2 – sniffer\_func.py

Then a python script for the sniffing process. This script has a function that can be called by

the main script.

The 'packetsniff()' function is parsed the name the output file is to be saved as, the filter expression to be used, if the sniffer is to print each packet while running. Then using scapy's 'sniff()' function it creates a new sniffer. This sniffer has it's iface left as None, this is it's default value and sets it to sniff on all network interfaces. Has it's filter set as the user inputted filterString, if the user gave no filter then it will be set as None resulting in no filtering. Has it's print message set to a lambda function that prints a summary of each packet if the user flagged it in print mode, if the user did not then the print message is made a blank string resulting in no printing. The count is then set as the maxpackets value the user gave, this means if the user gave 20 it will sniff until 20 packets are recorded, if the user gave no value it is then defaulted to 0 which means it will sniff infinity. This will continue until the user gives a keyboard interrupt such as 'Ctrl + C'. After this it will stop and save the packet log as a pcap file with the user specified name.

## 8.3 – log\_loader\_func.py

Then a python script for loading in an pcap file existing was created. This script has 2 functions that can be called by the main script.

The 'loadpcapfile()' function which can be call and parsed the name/location of a pcap file to load. Using the scapy function 'rdpcap()' which takes the pcap file as a parameter and returns a list of each packet. This then prints some basic details on the packet log, with the function then returning the packet list. If the file loaded is invalid it will stop the program with an error message.



The 'loadpacketdata()' function is another function which can be called to load the packet list into a pandas data frame. It first defines the 7 columns and creates the data frame based on them. It then starts a tqdm loading bar and loops through each packet in the list, each time creating a data frame new row with the packets source and destination IP, source and destination port, protocol type (TCP, UDP or other), time when intercepted and whether it is an IPv4, IPv6 or non IP packet. If the packet is IPv4 then it extracts the data from the IP section of the packet, if IPv6 then from the IPv6 section and if not an IP packet then assign it unknown values. This row is then added to the main data frame. If any data is unable to be extracted then an unknown value is placed in instead. Once this is done the function returns the data frame containing every packet.

## 8.4 – output\_gen\_func.py

Then a python script for generating the graphs/document. This script has a function that can be called by the main script.

The 'graphgen()' function is parsed the packet log data frame, the name/location of the file loaded, the IP addresses to be shown in detail and the max number of packets to show in the charts.

If a specific IP address was not given, then charts/document generated will be generic and for the whole data frame with all the packets will be used ('log\_df'). If one was given then charts/document will be about that IP address specifically with it assinging 2 new data frames, one for all packets where the specified IP address is the sender ('ip\_src\_df') and one for all packets where the specified IP address is the receiver ('ip\_dst\_df '). With these 2 being used instead of 'log\_df'.

It then creates a folder to hold all outputs naming it after the file name and the name of the specified IP address if applicable and only using the first 4 characters if the IP address is an IPv6 as colons can't be used in file names.

All of the following charts were generated with matplotlib.pyplot. They are generated by using the pandas 'value\_counts()' function to get each unique instance and count from a specific column. Then the pandas 'nLargest()' function return the top N rows, this is done as too many entries being shown can result in the graphs/doc being unreadable. Then the matplotlib.pyplot function 'plot()' is used to generate the chart, using the 'kind' argument 'barh' to generate a horizontal bar chart with the IP address as the bar name and it count as the bar's value. This chart is then labelled, saved in the created folder and then cleared.

It then calls 'sourceIPChart()' to generate a bar chart showing the top N source IP addresses, from either 'log\_df' or 'ip\_dst\_df', of packets and their number of occurrences. If a specific IP address was given, then it will only count source IPs that sent packets to the specified IP address.

It then calls 'destinationIPChart()' to generate a bar chart showing the top N destination IP addresses, from either 'log\_df' or 'ip\_src\_df', of packets and their number of occurrences. If a specific IP address was given, then it will only count destination IPs that received packets from the specified IP address.

It then calls 'sourcePortChart()' to generate a bar chart showing the top N source ports, from either 'log\_df' or 'ip\_dst\_df', of packets and their number of occurrences. If a specific IP address was given, then it will only count source ports that sent packets to the specified port.

It then calls 'destinationPortChart()' to generate a bar chart showing the top N destination ports, from either 'log\_df' or 'ip\_src\_df', of packets and their number of occurrences. If a specific IP address was given, then it will only count destination ports that received packets from the specified IP address.

Then using the ipinfo package a handler is created using a token. NOTE: This token is associated with a personal account created with the ipinfo.io service and has a limited number of uses, this token can be switched to a different one if the limit is reached and should be removed if the tool is to be made public. Using the pandas 'values' function the source and destination IP addresses are extracted from the 'log\_df', data frame containing all packets, and saved in a 2d array. This is then flattened into a 1D array. The using the pandas function 'unique()' and any duplicate IPs are dropped. This array is then cycles through each IP address requesting it's information from ipinfo.io which is then saved in a list. It then flags that the info lookup was successful, if it wasn't then any error will throw the try/except and it will be flagged as unsuccessful.

Next the IP location maps are generated if the IP lookup was successful, if not then this is skipped. It does this with matplotlib.pyplot again. By creating a base map, setting it's colours and then cycling though either the 'log\_df' or 'ip\_dst\_df' for the source IP map and 'log\_df' or 'ip\_src\_df' for the destination map. For the source map it gets each source IP address and for the destination map it get each destination IP address. Then it uses the pandas 'value\_counts()' function to get each unique instance and count from a specific column. It then uses the pandas 'index' and 'tolist()' to put the top IP addresses into a list. It then cycles through this list and finds each IP address in the ipinfo data list. If that IP address has a found location then it is added to the map as a marker, if it does not have a

location due to being an internal address then nothing is done. The map is then saved and cleared.

Finally a docx document is generated using the 'python-docx' package. If the program was run normally then the function 'documentGeneric()' is run where as if it was run with a specific IP address 'documentSpecific()' is instead run.

'documentGeneric()' creates a new document, adding a heading to the document. It then creates a heading on the next page, under which it then cycles through the top N source IP addresses listing each one. For each IP address it adds the IP's location (if applicable), the number of packets sent, then cycling through and adding the top 3 destination IP addresses and ports that the source IP address sent packets to. It then repeats this for the destination IPs; printing the location (if applicable), the top 3 source IP addresses and ports that sent packets to the destination address. Once it has added all the information the document is saved.

'documentSpecific()' creates a new document, adding a heading to the document. But instead cycles through and adds all IP addresses that sent packets to the user specified IP address. For each IP address it adds the IP's location (if applicable), the number of packets sent, then cycling through and adding the all the destination IP addresses and ports that the source IP address sent packets to. It then repeats this for the destination IPs; printing the location (if applicable), the top all the source IP addresses and ports that sent packets to the destination address. Once it has added all the information the document is saved.

The generic document provides less information for each IP address as it provides more of a general overview, only showing some of the IP addresses and some of their information, whereas the document for the user specified IP address provides more to provide a more specific view, showing all IPs and more information for each one. Once it has added all the information the document is saved.

## **9 – Testing**

All figures mentioned are viewable in the appendix at the end of the document.

The following test were preformed on a Windows machine with python 3.10.2 installed, with similar test being preformed on Linux to ensure the results are the same, it will be noted if they ever provide differing results. Other machines on the same network created additional network traffic by accessing websites, steaming and playing online games. The program was run through the Windows command line and in administrator mode unless specified otherwise.

### **9.1 – Sniffing and Printing test**

In this test the program is run in sniffing mode with it set to print each packet.

The expected result is that it should sniff and print each packet until the user stops it with the 'Ctrl + C' keys. After which it should save the packet log as a pcap file.

The Main.py script is run in sniffing mode with the output file name "Example" with the printing argument. This is done with the command:

```
python Main.py -s Example -p
```

This program is launched in sniffing mode, printing each packet as they're intercepted. This continues until it is interrupted with 'Ctrl + C' at which point it stops and each saves the packet log as "Example.pcap". The CLI showing this can be seen in the figure 1 and the saved pcap file can be viewed in wireshark in figure 2.

This test was successful.

## 9.2 – Sniffing and Chart Generation

In this test the program is run in sniffing mode, after that the program is run in load mode and the saved pcap file is loaded.

The expected result is that it should sniff and print each packet until the user stops and the packet log is saved as a pcap file. It should then load the pcap file and create a folder with a source IP address chart, destination IP address chart, source port chart, destination port chart, map showing the locations of the packet sources, map showing the locations of the packet destinations and an analysis document.

The Main.py script is run in sniffing mode with the output file name “Example”. This is done with the command:

```
python Main.py -s Example
```

Once this is done the program will begin sniffing run until the user presses “Ctrl + C”, saving the packet log as “Example.pcap”.

The Main.py script is then run this time in load mode with the input file name “Example.pcap”. This is done with the command:

```
python Main.py -l Example.pcap
```

This processes the packet log, creating a folder called “Example.pcap Output” and saving a source IP address chart (figure 4), destination IP address chart (figure 5), source port chart (figure 6), destination port chart (figure 7), map showing the locations of the packet sources (figure 8), map showing the locations of the packet destinations (figure 9). As well as a

document detailing the top 50 source IP addresses, along with the top 10 destination IP addresses and ports associated with that address, (excerpt at figure 10) and the top 50 destination IP addresses, along with the top 10 source IP addresses and ports associated with that address, (excerpt at figure 11). The CLI showing this can be seen in the appendix as figure 3.

This test was successful.

## 9.3 – Filtered Sniffing

In this test the program is run in sniffing mode with it set to print each packet and with a filter expression specifying a source IP address and the tcp protocol.

The expected result is that it should sniff and print only tcp packets sent from the specified IP address until the user stops it with the 'Ctrl + C' keys. After which it should save the packet log of the specified packets of as a pcap file.

The Main.py script is run in sniffing mode with the output file name "ExampleFilter". This includes the filter expression "src host 91.221.58.40 and tcp" and the print argument. This is done with the command:

```
python Main.py -s ExampleFilter -f "src host 91.221.58.40 and tcp" -p
```

This program is launched in sniffing mode, sniffing and printing only packets that were sent from "91.221.58.40" and are tcp protocol. This can be seen in figure 12.

This test was successful.

## 9.4 – Specific IP output

In this test the program is run in sniffing mode, after that the program is run in load mode twice and the saved pcap file is loaded. First time normally and second time with a specified IP address.

The expected result is that it should sniff and print each packet until the user stops and the packet log is saved as a pcap file. First it should load the pcap file and create a folder with all the charts and document as shown in test 8.2, secondly it should load the pcap file again create a folder with the charts/document however they should only relate to the specified IP address.

The Main.py script is run in sniffing mode with the output file name “ExampleSpecific”. This is done with the command:

```
python Main.py -s ExampleSpecific
```

Once this is done the program will begin sniffing run until the user presses “Ctrl + C”, saving the packet log as “ExampleSpecific.pcap”.

The Main.py script is then run in load mode loading the ExampleSpecific.pcap file. This is done with the command:

```
python Main.py -s ExampleSpecific
```

The Main.py script is then run in load mode again loading the ExampleSpecific.pcap file this time with the specified IP address “2q03:2880:f258:c4:face:b00c:0:32c2”. This is done with the command:



```
python Main.py -s ExampleSpecific -i
```

```
"2q03:2880:f258:c4:face:b00c:0:32c2"
```

The result is 2 separate folders, one called "ExampleSpecific.pcap Output" and one called "ExampleSpecific.pcap Output 2q03". The first one having all the charts as shown in test 8.2 with data for the whole pcap file used, see figures 14 and 16 for examples. However in the second folder the same charts/document were generated but only using packets related to the specified IP address such as the source chart only showing IP addresses that sent packets to the specified IP address and the document all listing the packets sent to and from the specified IP address, see figures 15 and 17. Screenshots of the CLI running these processes can be seen in figure 13.

This test was successful.

## 9.5 – Sniffing Without Administrator Privileges

In this test the program is launched in sniffing mode from a command line that does not have administrator privileges.

The expected results are that it will enter sniffing mode but will flag an error then closing, it should not crash the compiler. This is done with the command:

```
python Main.py -s NoAdmin
```

The result is the program launching and announcing that it's in sniffing mode, it however stops and prints the error "Error: Not in administrator mode, cannot sniff in this mode". This is not a compiler error but a print command in the script. After this the program ends. This can be seen in figure 17.

This test was a success.

## 9.6 – Setting invalid Max packets/entries

In this test the program is launched in sniffing mode and load mode with the '-m' argument to set the max packets to sniff/max entries to show in the charts/document, however, with an invalid value. A valid value is any positive integer over 0, this test will use the vales '0', '-5' and 'apple'.

The expected result is, no matter what the value is or if it's in sniff/load mode it will return an invalid max number. This is done with the commands:

```
python Main.py -s FilterTest -m 0
```

```
python Main.py -s FilterTest -m -5
```

```
python Main.py -s FilterTest -m Apple
```

```
python Main.py -l FilterTest -m 0
```

```
python Main.py -l FilterTest -m -5
```

```
python Main.py -l FilterTest -m apple
```

Each of these has the same results, printing the error "Invalid Max Number". This can be scene in figure 18.

Additionally it was tested with no max number value at all, the expected result is that it will print an error and print the help screen, this being the error message when the GetoptError is thrown. This is done with the command:

```
python Main.py -s FilterTest -m
```

The actual result was it printing “Command Error!” and printing the help scene. This can be seen in figure 18.

This test was successful.

## 9.7 – Setting invalid file to load

In this test the program is launched in load mode with the file following the ‘-l’ command being an invalid. The file names given are ‘NonExistantFile’, a file that does not exist, ‘test.txt’, a blank text file, and ‘:’, another non-existing file.

The expected results are that each file should return an invalid file error. This is done with the commands:

```
python Main.py -l NonExistantFile
```

```
python Main.py -l test.txt
```

```
python Main.py -l :
```

The actual result is the files beginning to load but stopping immediately to print an ‘Error: Invalid File loaded!’. This can be seen in figure 19.

Additionally it was tested with no file name value at all, the expected result is that it will print an error and print the help screen, this being the error message when the GetoptError is thrown. This is done with the command:

```
python Main.py -l
```

The actual result was it printing “Command Error!” and printing the help scene. This can be seen in figure 19.

This test was successful.

## 9.8 – Load file with non-existent specified IP address

In this test the program is launched in sniffing mode to create a pcap file, this pcap file is loaded with the specified IP address 'NotAnIpAddress' which is not an IP address and therefore doesn't appear in the pcap file.

The expected result is the file being loaded, after when the data generation begins it should flag that the IP never appears and ends early. This is done with the command:

```
python Main.py -l Iptest.pcap -i NotAnIpAddress
```

The result is that the file loads successfully but when the data generation begins it prints 'Error: Specified IP address doesn't occur.' at which point the program ends. This can be seen in figure 20.

This test was a success.

## 9.9 – Invalid Filter

In this test the program is launched in sniffing mode with the '-f' argument to apply a filter to the sniffing, with the filter applied being an invalid one. Namely 'NotValidFilter' which is not a valid filter in the BPF syntax.

The expected results are that it will flag that the filter is invalid and the program will sniff without any filter. This is done with the command:

```
python Main.py -s FilterTest -f "NotValidFilter"
```

The actual result is that the sniffing begins and it prints a message "Error: Could not compile filter expression NotValidFilter". After this it sniffs as it would normally with none of the packets being filtered. This can be seen in figure 21.

This test was successful.

## 9.10 – Setting valid Max packets/entries

In this test the program is launched in sniffing mode to create a pcap file. Then the pcap file is loaded twice, once normally and secondly with the '-m' argument to set the max number of packets shown as 5.

The expected result is that the first source chart should show all the source IPs where as the second should only show the top 5 IP addresses. This is done with the commands:

```
python Main.py -l MaxEntryTest.pcap
```

```
python Main.py -l MaxEntryTest.pcap -m 5
```

The actual result is that the source chart from the first command shows all the source IP addresses where as the second should only show the top 5 IP addresses. This can be seen in figures 22, 23 and 24.

## 9.11 – Sniffing with no internet connection

In this test the machine used for testing is disconnected from the internet and the program is launched in sniff mode with packets set to print.

The expected result is that the program start but nothing will ever be sniffed, with it doing

nothing until the user ends the sniffing. This is done with the command:

```
python Main.py -s NoInternetTest -p
```

The actual result is that the sniffing begins and nothing is ever printed, however, the user cannot stop the sniffing with the 'Ctrl + C' command resulting in the program soft-locked likely until a packet is sniffed. This can be seen in figure 25.

This test was partly unsuccessful, the idea of sniffing without an internet connection was never considered during development and while this is a niche issue it is an issue nonetheless.

## 9.12 – Loading with no internet connection

In this test the machine used for testing is disconnected from the internet and the program is launched in load mode with it loading a pcap file.

The expected result is that the program will generate the IP and port charts but wont generate the maps and wont include location details in the document. This is done with the command:

```
python Main.py -l NoInternetTest.pcap
```

The actual result is that the file is loaded with the source IP, destination IP, source port, destination port and document are generated. The document dose not specify any of the IP locations as this information is reliant on Ipinfo.io, similarly as the maps rely on this they are not generated. This can be seen in figure 26 and 27.

This test was a success.

## **10 – Conclusions**

These were the requirements set out at the beginning of this project.

Note: This uses the MoSCoW method for prioritisation. [34]

### **10.1 – Functional Requirements**

Number	Requirement	Priority	Requirement met?
F-1	It should log each packet received by the host machine and allow the user to view them.	Must	Yes
F-2	It should allow the host machine to receive packets not intended for it.	Must	Yes
F-3	It should compile and generate data from the packet log.	Must	Yes
F-4	It should generate visualisations based on the compiled data in the form of graphs/charts.	Should	Yes
F-5	It should have the option to save the packet log as a separate file.	Must	Yes
F-6	It should have the option to load a previously saved packet log file.	Should	Yes
F-7	It should have the option to filter which packets are logged by the program.	Should	Yes
F-8	It should be able to generate a report using the data and visualisation.	Could	Yes
F-9	It should have the option to create and send packets which it will log the responses.	Could	No
F-10	It should be usable from the CLI only.	Must	Yes
F-11	It should be usable from a GUI only.	Should	No
F-12	It should work on Linux	Must	Yes
F-13	It should work on Windows	Could	Yes
F-14	It should work on MacOS	Won't	No

## 10.2 – Non-Functional Requirements

Number	Requirement	Priority	Requirement met?
NF-1	It should put the computer into promiscuous mode to intercept packets not intended for the host.	Must	Yes
NF-2	It should be programmed in python.	Must	Yes
NF-3	It should use the 'pandas' library to handle the processing and compiling of data.	Should	Yes
NF-4	It should use the 'matplotlib' library to handle data visualisation such as graphs or charts.	Should	Yes
NF-5	It should use the 'tkinter' library to handle the GUI element of the program.	Should	No
NF-6	The program should be useable without an external guide.	Should	Partial

## 10.3 – Requirements Reasoning

These requirements were derived from the analysis of the existing tools previously with it being a basic packet sniffer tool that can intercept and view all packets, including those not intended for them, filtering the intercepted packets and then saving it as these are very basic features each of the aforementioned packet sniffers. The option to generate charts and report were inspired by the data visitations that something like PRTG Network Monitor has. With loading pcap files being necessary to do this. Having it be a command line program was inspired by similar tools such as Tcpdump that are much simpler to use than PRTG which has a complicated interface with many options unrelated to packet sniffing. Having a GUI was inspired by wireshark having a somewhat simple to use GUI, one with all the



functions of that would likely be out of the scope of this tool. Being able to generate packets was inspired by Ettercap allowing the user to tamper with packets, the idea was a tool that allowed the user to be more active but in a way that still related to network analysis. It should work on Windows as that is a commonly used OS as well as Linux which is more used by technical users. It won't be developed for mac due to a lack of devices to properly test with on the developers side as well as a perceived less usefulness due to mac users potentially being a smaller demographic.

## 10.3 – Requirements Not Met

Most of the requirements set out at the beginning of this project have been met, all of which have been shown earlier in this report. For the requirements that were not met:

- The program is not usable in a GUI, this feature was cut early on when it was decided that the program would not allow the user choose which charts were generated after running the program but instead to simply generate all of them each time. This was chosen as it resulted in a quicker and less finicky user experience. With the only user prompt during the program removed there wasn't much point in a GUI as it would be at most a couple of text boxes to ask for information that could be entered into the CLI on launching.
- The program has not been tested on MacOS, this does not mean it won't work as it might if python is able to be set up and the correct packages installed. Sniffing mode will also be unusable due to the administrator access check. It just hasn't been tested if this is possible as it is not in the scope of this project.
- The program has a help menu that attempts to explain how to operate it so it may be possible for a user to operate it without referencing the user guide. It may also be

possible that a user may not understand and as user testing was not decided at the beginning of this project this is marked a partial implemented.

- The program is not able to generate packets, this feature was de-prioritised early on due to concerns over it's usefulness. As a result it was outright cut later due to time constraints.

## 10.4 – Difficulties Encountered

One difficulty that was encountered was that on the windows command line, CMD, it is not possible to use a keyboardInterrupt. This was an issue at one point as the sniffer relied on a try/except that was triggered by a keyboardInterrupt, which itself was done as scapy's 'AsyncSniffer()' was used instead of 'sniff()'. Where as 'sniff()' can be cancelled by user input, 'AsyncSniffer()' cannot as it is intended to be run asynchronously from other functions. This was chosen despite being unnecessary as no functions are run asynchronously and was only included due to frank oversight. Originally to remedy this the ability to limit the number of packets intercepted was included to allow testing on a windows machine but was later fully fixed when 'sniff()' was correctly used and keyboardInterrupt was no-longer needed.

When ever the iFace parameter of 'sniff()' was specified as anything other than None it would crash on the windows machine used for testing citing: "OSError: Error opening adapter: The filename, directory name, or volume label syntax is incorrect. (123)".

Investigation revealed that this was likely caused by an issue on the windows testing machine. Due to this the feature to specify which interface to sniff on had to be cut as it couldn't be ensured that it wouldn't crash. This error could be confirmed if it could be tested on another windows machine with administrator access however due to time constraints this couldn't be put in place. In retrospect this could have been mitigated by putting the sniffing

function in a try/except and if the error is thrown instead run the version without the specified iFace.

When packet data is extracted if the requested data dose not exist then it causes an error, an issue if the source IP address from the IP section of a packet is extracted from a non IP packet then it will crash. At first this was fixed by dropping all non IP packets unaware that this cause all IPv6 packets to be dropped as well. This wasn't discovered for a bit but was fixed by checking for IPv6 packet and extracting the data from the IPv6 section of it. Additionally to fix the issue of certain IP packets having IP address but not port addresses, such as IGMPv3, try/except were added and if an error is thrown while trying to fill a specific value it will default to unknown.

Charts charts generated originally has too many entries that it made readability difficult, as a result a default of 50 had to be added to ensure they would be readable. Also when saved as images the edges of charts would be cut off which with some tinkering was fixed by setting the bbox\_inches parameter to 'tight'. Additionally the size of the maps needed to be tinkered with to ensure the markers weren't to large and that the map wasn't too pixelated when zoomed into, making it larger helped with this.

One issue that almost wasn't discovered was that if data outputs for a specific IP were generated it includes the IP in the folders name, this is an issue if the specific IP is an IPv6 address which includes colons. As windows cannot have folder/file names with colons in them this caused it to crash. To fix this if the IP is an IPv6 it will split and only use the first 4 characters before the first colon.

One difficulty that arose and is still an issue is the 'output\_gen\_func.py' scrip is fairly messy.

This made bug fixing and any changes much harder to implement. This can be seen especially in the 2 document generating functions, both of which could probably have been consolidated into one function to reduce duplicate code.

## 10.5 – Project Going Forward

As mentioned in section 5.5 (Achievements) and in the previous requirement sections most of the main requirements were met with the program working as a basic packet sniffer that can generate visual charts and a document to assist with the analysis of a networks packet traffic. With options to filter the sniffing, specify the amount of packets to sniff, if it should print each one or not, if it should generate the outputs for a specific IP address or all of them, how many entries should be shown in the bar charts/document. This program is however still relatively basic and could be expanded upon if development were to continue, such as:

- Allow user to specify which interfaces to sniff on instead of all ways sniffing on all interfaces, requires further investigating the previously mention crashing.
- Allow user to immediately generate the charts right after sniffing without having to re-load the packet capture.
- Look into ways to optimise the loading packet capture into pandas dataframe function as this can take a while with longer pcap files.
- Include option to allow users to specify more details about the generated charts to change their formatting or appearance. Possibly specify which graphs to generate if more are added.
- Include an option to blacklist specific IP address, ports, protocols or even countries/locations to not use any data from packets that include the specified information in the generated charts/document.
- Include an option to whitelist specific IP address, ports, protocols or even

countries/locations to only use any data from packets that include the specified information in the generated charts/document. Currently this can be one with one specific IP address however this concept could be expanded on.

- Both of the previous blacklist and whitelist could be done with BPF syntax, it should be investigated if scapy, or any other tool, could allow for filtering of existing pcap files during processing.
- Include statistics on packet protocol types, this is partly collected with packets being marked as 'TCP', 'UDP' and 'Other' but this currently goes unused. More detailed information could be logged and used in the document or to create additional charts.
- Include a chart, possibly a histogram, showing packets intercepted over time. Currently the time a packet was sniffed is collected but goes unused. This would need to be dynamic or allow users to specify the time divisions (I.E. bars representing packets intercepted in a specific second, minute, 5/10 minute block, hour, etc...).
- Log the purpose of each packet and surface this information to the user, such as specifying that packets sent to the network were as a file transfer and the packets sent back were as acknowledgement packets.
- Add the ability to set the program to process multiple pcap files automatically one after the other.
- Include a check so that sniffing cannot be performed without an internet connection.

The program as it stands is a basic tool with a narrow use case however it shows the potential for a graphical data analysis tool, with the suggested features above expanding the scope of it. This is a tool that with some work could be a versatile tool to assist in analysing network traffic. It could be useful to work with technical users in the future, either in an informal setting or with a proper user study, to derive what features would be perceived as


useful.

## **11 – References**

Note: Citations were handled using 'Zotero' with IEEE citation style. [35]

References that refer to [Linux] or [Windows] are referring to software used on those operating systems that can be downloaded from the accompanying URL.

The rdf file containing the citations can be provided on request.

- [1] S. Ansari, S. G. Rajeev, and H. S. Chandrashekar, 'Packet sniffing: a brief introduction', *IEEE Potentials*, vol. 21, no. 5, pp. 17–19, Dec. 2002, doi: 10.1109/MP.2002.1166620.
- [2] M. A. Qadeer, A. Iqbal, M. Zahid, and M. R. Siddiqui, 'Network Traffic Analysis and Intrusion Detection Using Packet Sniffer', in *2010 Second International Conference on Communication Software and Networks*, Singapore: IEEE, 2010, pp. 313–317. doi: 10.1109/ICCSN.2010.104.
- [3] 'Tcpdump'. in (Software). Jun. 09, 2021. [Linux]. Available: <https://www.tcpdump.org>
- [4] P. Goyal and A. Goyal, 'Comparative study of two most popular packet sniffing tools- Tcpdump and Wireshark', in *2017 9th International Conference on Computational Intelligence and Communication Networks (CICN)*, Girne: IEEE, Sep. 2017, pp. 77–81. doi: 10.1109/CICN.2017.8319360.
- [5] 'Home | TCPDUMP & LIBPCAP'. <https://www.tcpdump.org/> (accessed Dec. 01, 2022).
- [6] Wireshark, 'Wireshark · About'. <https://www.wireshark.org/about.html> (accessed Nov. 01, 2022).
- [7] 'Wireshark'. Oct. 26, 2022. [Windows]. Available: <https://www.wireshark.org/#download>
- [8] K. M. Majidha Fathima and N. Santhiyakumari, 'A Survey On Network Packet Inspection And ARP Poisoning Using Wireshark And Ettercap', in *2021 International Conference on Artificial Intelligence and Smart Systems (ICAIS)*, Coimbatore, India: IEEE, Mar. 2021, pp. 1136–1141. doi: 10.1109/ICAIS50930.2021.9395852.
- [9] 'Packet Monitor (Pktmon) | Microsoft Learn'. <https://learn.microsoft.com/en-us/windows-server/networking/technologies/pktmon/pktmon> (accessed Nov. 01, 2022).
- [10] 'About « Ettercap'. <https://www.ettercap-project.org/about.html> (accessed Nov. 03, 2022).
- [11] 'Ettercap'. Aug. 01, 2022. [Linux]. Available: <https://www.ettercap-project.org/downloads.html>
- [12] 'NetworkMiner'. Apr. 04, 2022. [Linux]. Available: <https://www.netresec.com/?page=NetworkMiner>
- [13] 'NetworkMiner - The NSM and Network Forensics Analysis Tool '. <https://www.netresec.com/?page=NetworkMiner> (accessed Nov. 02, 2022).
- [14] 'PRTG network monitoring tool: Network monitoring for professionals'. [https://www.paessler.com/network\\_monitoring\\_tool](https://www.paessler.com/network_monitoring_tool) (accessed Nov. 05, 2022).
- [15] 'PRTG Network Monitor » All-in-one network monitoring software'. <https://www.paessler.com/prtg/prtg-network-monitor> (accessed Nov. 05, 2022).
- [16] 'PRTG Network Monitor'. Nov. 07, 2022. [Windows]. Available: [https://www.paessler.com/network\\_monitoring\\_tool](https://www.paessler.com/network_monitoring_tool)
- [17] D. Mistry, P. Modi, K. Deokule, A. Patel, H. Patki, and O. Abuzagheh, 'Network traffic measurement and analysis', in *2016 IEEE Long Island Systems, Applications and*

- Technology Conference (LISAT)*, Farmingdale, NY, USA: IEEE, Apr. 2016, pp. 1–7. doi: 10.1109/LISAT.2016.7494141.
- [18] 'Python'. <https://www.python.org/> (accessed Nov. 26, 2022).
  - [19] 'socket — Low-level networking interface'. <https://docs.python.org/3/library/socket.html>
  - [20] J. Hunt, 'Sockets in Python', in *Advanced Guide to Python 3 Programming*, in Undergraduate Topics in Computer Science. Cham: Springer International Publishing, 2019, pp. 457–470. doi: 10.1007/978-3-030-25943-3\_39.
  - [21] 'python-libpcap'. <https://pypi.org/project/python-libpcap/> (accessed Nov. 19, 2022).
  - [22] 'Scapy'. <https://scapy.net/> (accessed Nov. 24, 2022).
  - [23] R. R. S, R. R, M. Moharir, and S. G, 'SCAPY- A powerful interactive packet manipulation program', in *2018 International Conference on Networking, Embedded and Wireless Systems (ICNEWS)*, Bangalore, India: IEEE, Dec. 2018, pp. 1–5. doi: 10.1109/ICNEWS.2018.8903954.
  - [24] J. Schulist, D. Borkmann, and A. Starovoitov, 'Linux Socket Filtering aka Berkeley Packet Filter (BPF)'. <https://www.kernel.org/doc/Documentation/networking/filter.txt> (accessed Apr. 01, 2023).
  - [25] S. Valentine, 'Berkeley Packet Filters with Scapy (and Friends)', *Linux J.*, vol. 2014, no. 242, pp. 84–95, Jun. 2014.
  - [26] 'Windows', *Scapy Download and Installation*. <https://scapy.readthedocs.io/en/latest/installation.html#windows> (accessed Apr. 04, 2023).
  - [27] 'Npcap', *Packet capture library for Windows*. <https://npcap.com/#download> (accessed Apr. 04, 2023).
  - [28] 'pandas'. <https://pandas.pydata.org/> (accessed Nov. 05, 2022).
  - [29] 'Matplotlib: Visualization with Python'. <https://matplotlib.org/> (accessed Nov. 07, 2022).
  - [30] 'tkinter — Python interface to Tcl/Tk'. <https://docs.python.org/3/library/tkinter.html> (accessed Nov. 05, 2022).
  - [31] 'python-docx'. <https://pypi.org/project/python-docx/> (accessed Apr. 01, 2023).
  - [32] 'ipinfo'. <https://pypi.org/project/ipinfo/>
  - [33] 'The trusted source for IP address data'. <https://ipinfo.io/>
  - [34] K. Brennan, 'MoSCoW Analysis (6.1.5.2)', in *A guide to the Business analysis body of knowledge (BABOK guide)*, Version 2.0. Toronto: International Institute of Business Analysis, 2009, p. 102. [Online]. Available: <https://books.google.co.uk/books?id=CFHw8jSEWwkC&printsec=frontcover#v=onepage&q&f=false>
  - [35] 'Zotero'. <https://www.zotero.org/> (accessed Nov. 29, 2022).
  - [36] 'Screenshots of the network monitor tool PRTG'. [https://hlassets.paessler.com/common/files/screenshots/bandwidth\\_check\\_header\\_sniffer.png](https://hlassets.paessler.com/common/files/screenshots/bandwidth_check_header_sniffer.png) (accessed Mar. 07, 2023).



## 12 – Appendices

### 12.1 – Figure 1

```
C:\Users\Ganymede\Desktop\aaa>python Main.py -s Example -p
Launched in sniffing mode.
Starting packet sniffing, press 'Ctrl + C' to end sniffing.
Ether / IP / UDP 192.168.1.138:54915 > 192.168.1.255:54915 / Raw
Ether / IPv6 / UDP fe80::700a:da13:2bec:123b:56981 > ff02::c:ws_discovery / Raw
Ether / IP / TCP 192.168.1.138:11669 > 192.168.1.78:8009 PA / Raw
Ether / IP / TCP 192.168.1.78:8009 > 192.168.1.138:11669 PA / Raw
Ether / IP / TCP 192.168.1.138:11669 > 192.168.1.78:8009 A
Ether / IP / UDP / DNS Qry "b'_oculusal_sp._tcp.local.'"
Ether / IPv6 / UDP / DNS Qry "b'_oculusal_sp._tcp.local.'"
Ether / IP / UDP / DNS Ans "b'Luke_-_::DESKTOP-QLB8RAR._oculusal_sp._tcp.local.'"
Ether / IPv6 / UDP / DNS Ans "b'Luke_-_::DESKTOP-QLB8RAR._oculusal_sp._tcp.local.'"
Ether / IP / UDP 192.168.1.138:54915 > 192.168.1.255:54915 / Raw
Packet sniffing stopped!
<Sniffed: TCP:3 UDP:7 ICMP:0 Other:0>
Saving packet log as Example.pcap

C:\Users\Ganymede\Desktop\aaa>
```

## 12.2 – Figure 2

No.	Source	Destination	Protocol	Length	Info
1	192.168.1.138	192.168.1.255	UDP	305	54915 → 54915 Len=263
2	fe80::700a:da13:2bec:123b	ff02::c	UDP	718	56981 → 3702 Len=656
3	192.168.1.138	192.168.1.78	TCP	164	11669 → 8009 [PSH, ACK] Seq=1 Ack=1 Win=1025 Le
4	192.168.1.78	192.168.1.138	TCP	164	8009 → 11669 [PSH, ACK] Seq=1 Ack=111 Win=1075
5	192.168.1.138	192.168.1.78	TCP	54	11669 → 8009 [ACK] Seq=111 Ack=111 Win=1024 Len
6	192.168.1.138	224.0.0.251	MDNS	83	Standard query 0x0000 PTR _oculusal_sp._tcp.loc
7	fe80::700a:da13:2bec:123b	ff02::fb	MDNS	103	Standard query 0x0000 PTR _oculusal_sp._tcp.loc
8	192.168.1.138	224.0.0.251	MDNS	361	Standard query response 0x0000 PTR Luke_-_::DES
9	fe80::700a:da13:2bec:123b	ff02::fb	MDNS	381	Standard query response 0x0000 PTR Luke_-_::DES
10	192.168.1.138	192.168.1.255	UDP	305	54915 → 54915 Len=263

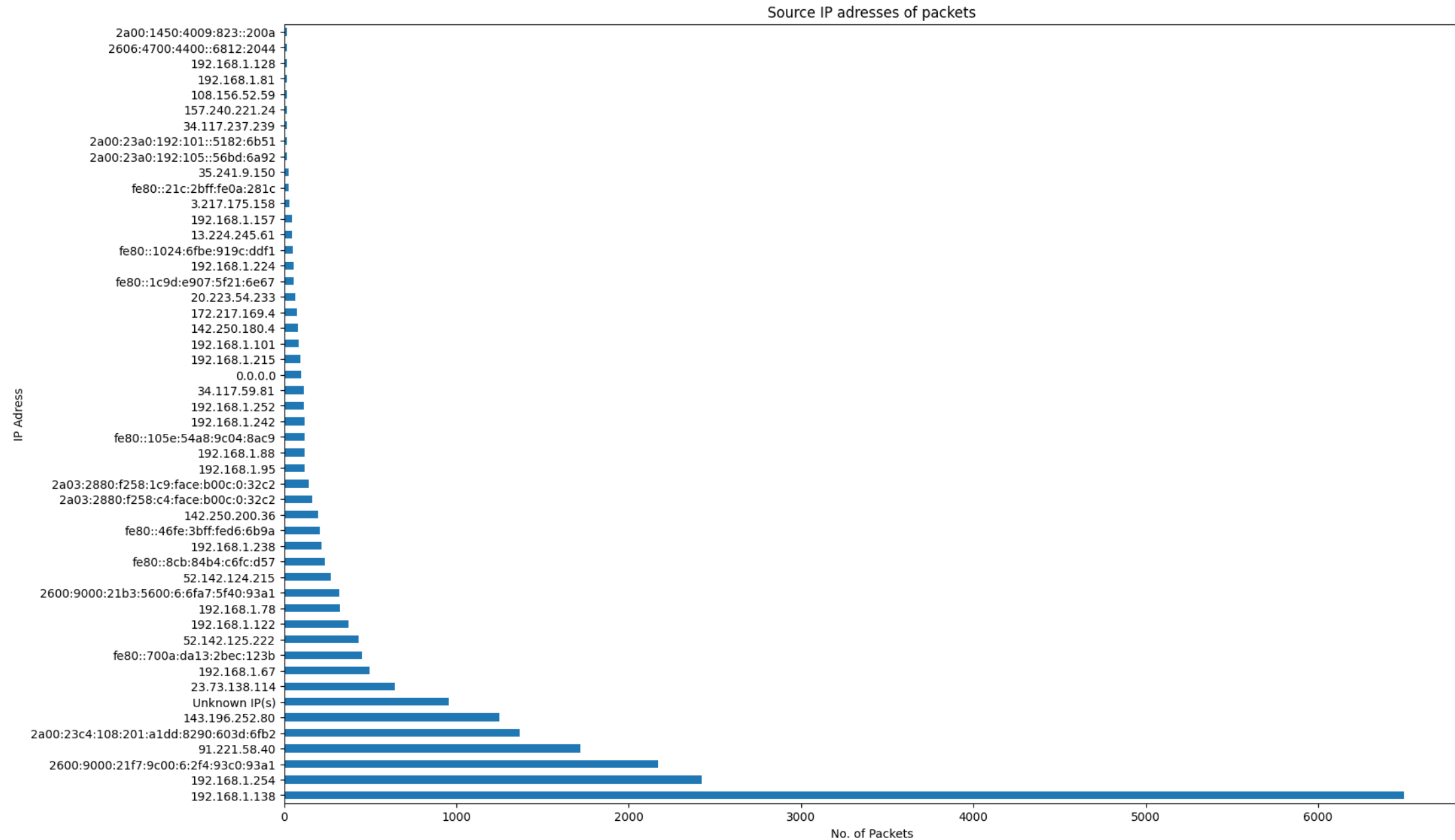
### 12.3 – Figure 3

```
C:\Users\Ganymede\Desktop\aaa>python Main.py -s Example
Launched in sniffing mode.
Starting packet sniffing, press 'Ctrl + C' to end sniffing.
Packet sniffing stopped!
<Sniffed: TCP:17335 UDP:2879 ICMP:1350 Other:1139>
Saving packet log as Example.pcap

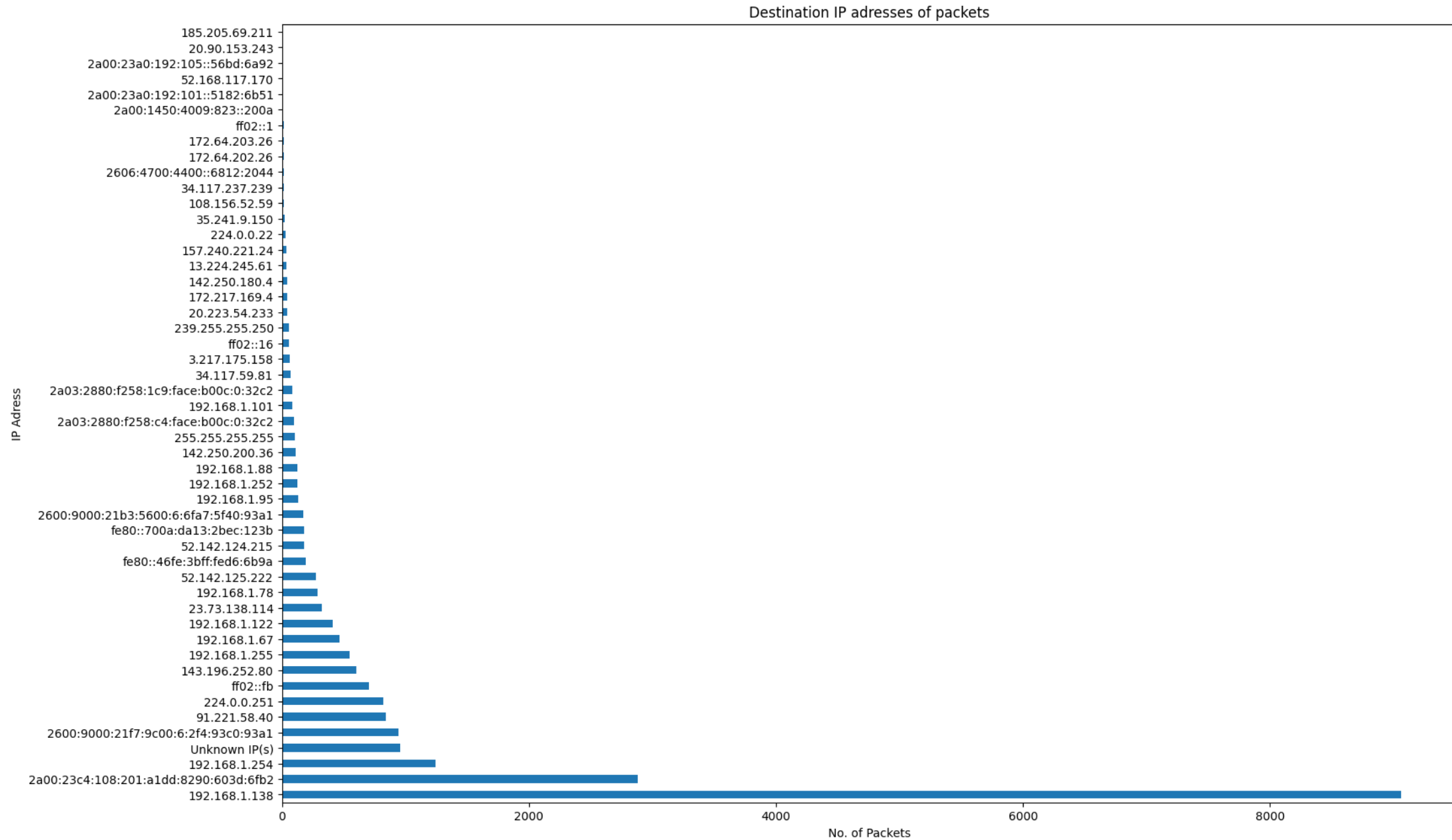
C:\Users\Ganymede\Desktop\aaa>
C:\Users\Ganymede\Desktop\aaa>python Main.py -l Example.pcap
Loading file Example.pcap...
<Example.pcap: TCP:17335 UDP:2879 ICMP:1350 Other:1139>
Loading packet data...
100%|██████████████████████████████████████████████████████████████████████████████| 22703/22703 [00:48<00:00, 470.57it/s]
Generating Output Data
Source IP Graph Generated
Destination IP Graph Generated
Source Port Graph Generated
Destination Port Graph Generated
IPinfo connection successful
Source IP map Generated
Destination IP map Generated
Document Generated
Outputs saved in the "./Example.pcap Output" folder

C:\Users\Ganymede\Desktop\aaa>
```

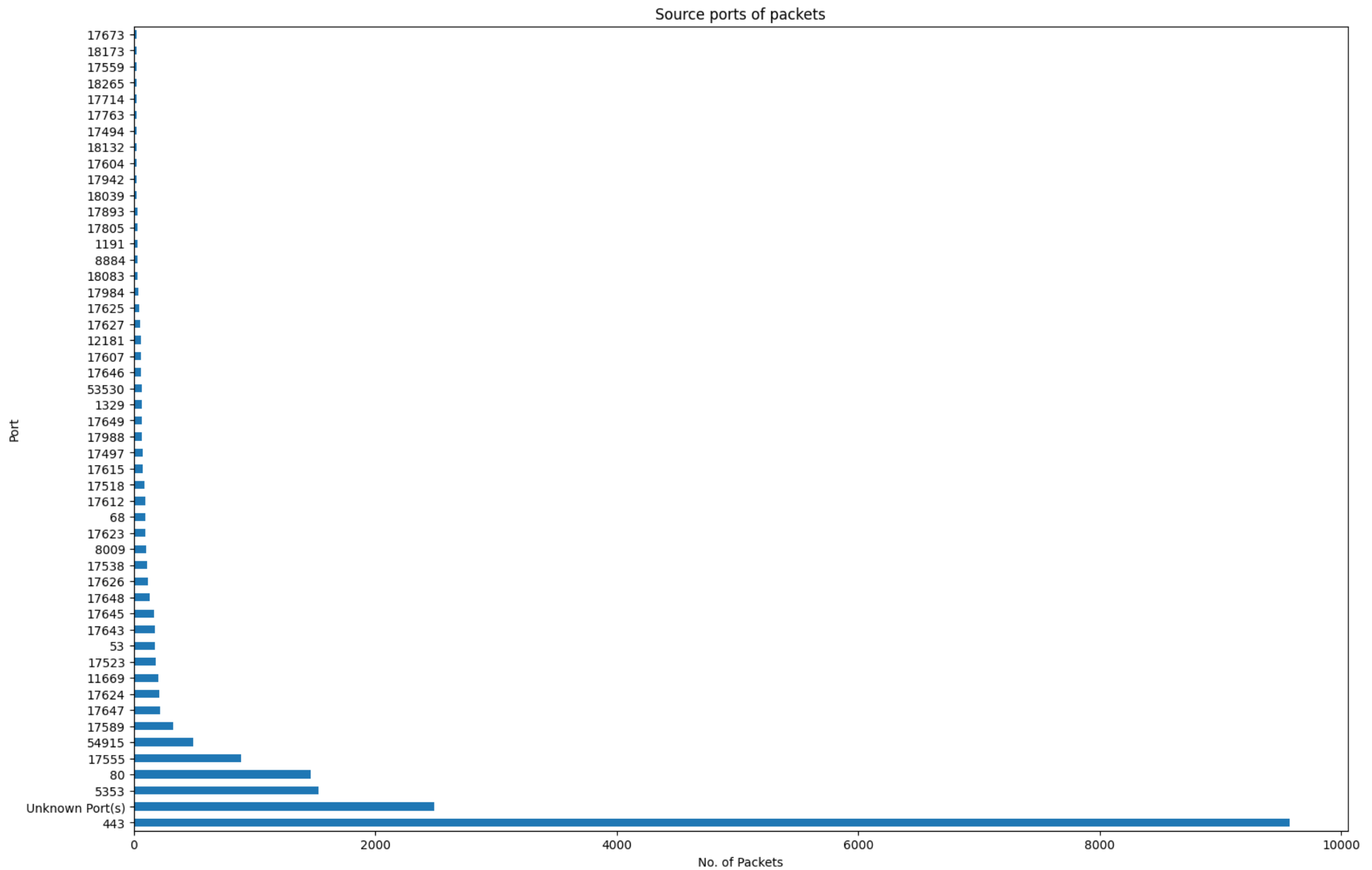
## 12.4 – Figure 4



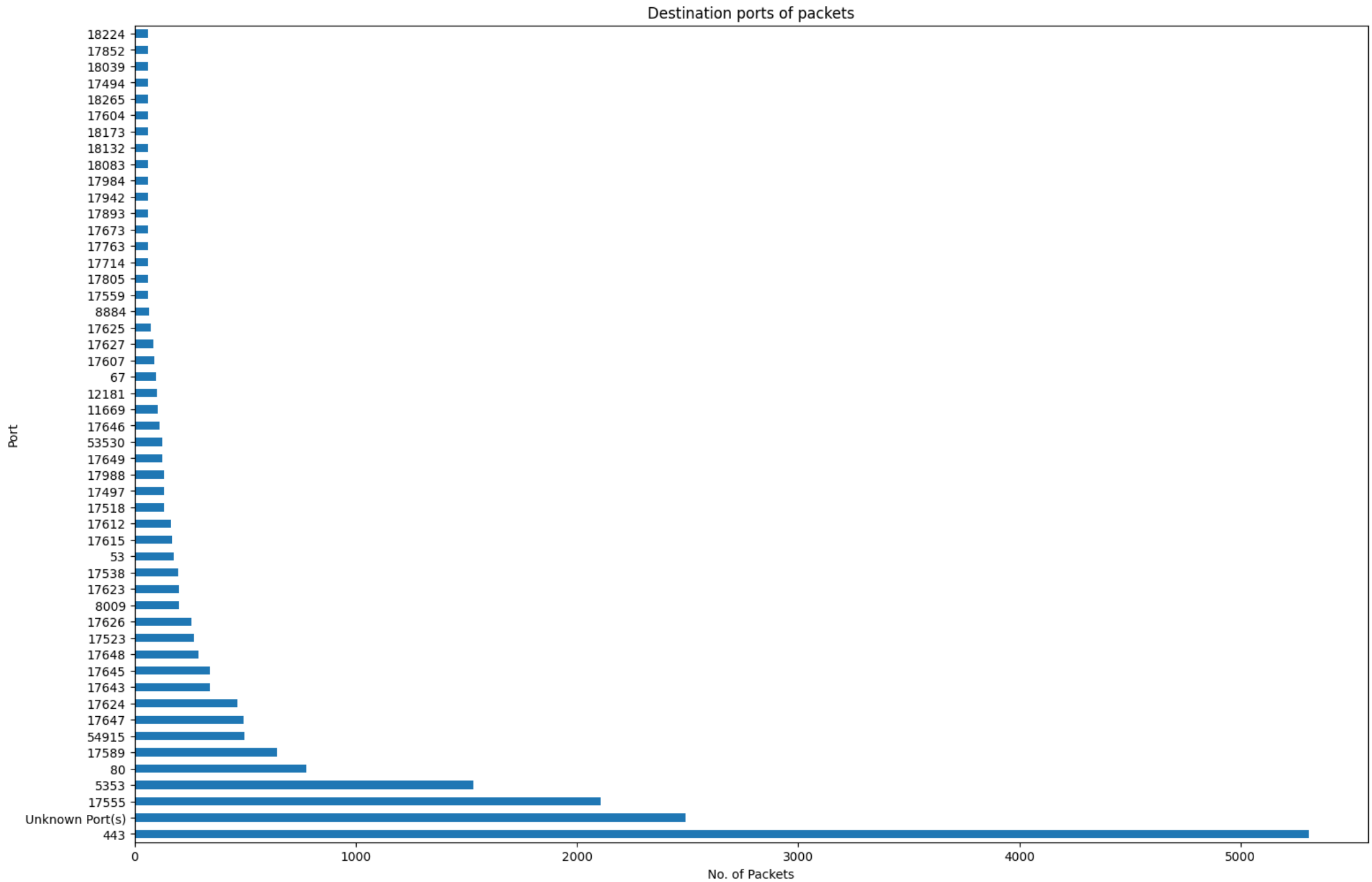
## 12.5 – Figure 5



## 12.6 – Figure 6



## 12.7 – Figure 7



## 12.8 – Figure 8

Sources of the packets (Orange)





## 12.9 – Figure 9

Destination of the packets (Blue)



## 12.10 – Figure 10

### 1. 192.168.1.138 - 6500 packet(s)

Location: None/Internal

Top 10 IP addresses that the packets were sent to:

192.168.1.254 - 1242 packet(s).  
91.221.58.40 - 840 packet(s).  
143.196.252.80 - 603 packet(s).  
192.168.1.255 - 497 packet(s).  
192.168.1.67 - 470 packet(s).  
192.168.1.122 - 414 packet(s).  
23.73.138.114 - 327 packet(s).  
192.168.1.78 - 287 packet(s).  
52.142.125.222 - 277 packet(s).  
224.0.0.251 - 264 packet(s).

Top 10 Ports that the packets were sent to:

443 - 3977 packet(s).  
80 - 755 packet(s).  
Unknown Port(s) - 686 packet(s).  
54915 - 496 packet(s).  
5353 - 264 packet(s).  
8009 - 202 packet(s).  
8884 - 66 packet(s).  
53 - 20 packet(s).  
1900 - 16 packet(s).  
25 - 10 packet(s).

### 2. 192.168.1.254 - 2424 packet(s)

Location: None/Internal

Top 10 IP addresses that the packets were sent to:

192.168.1.138 - 2420 packet(s).  
224.0.0.1 - 4 packet(s).

Top 10 Ports that the packets were sent to:

Unknown Port(s) - 89 packet(s).

17494 - 63 packet(s).  
17559 - 63 packet(s).  
17984 - 63 packet(s).  
17942 - 63 packet(s).  
17893 - 63 packet(s).  
18132 - 63 packet(s).  
17805 - 63 packet(s).  
17763 - 63 packet(s).  
17714 - 63 packet(s).

### 3. 2600:9000:21f7:9c00:6:2f4:93c0:93a1 - 2171 packet(s)

Location: Seattle, Washington, United States 🇺🇸

Top 10 IP addresses that the packets were sent to:

2a00:23c4:108:201:a1dd:8290:603d:6fb2 - 2171 packet(s).

Top 10 Ports that the packets were sent to:

17555 - 2108 packet(s).  
17551 - 13 packet(s).  
17550 - 13 packet(s).  
17552 - 13 packet(s).  
17554 - 12 packet(s).  
17553 - 12 packet(s).

### 4. 91.221.58.40 - 1718 packet(s)

Location: Köln, North Rhine-Westphalia, Germany 🇩🇪

Top 10 IP addresses that the packets were sent to:

192.168.1.138 - 1718 packet(s).

Top 10 Ports that the packets were sent to:

17647 - 492 packet(s).  
17643 - 342 packet(s).  
17645 - 341 packet(s).  
17648 - 291 packet(s).  
17649 - 127 packet(s).

## 12.11 – Figure 11

### 1. 192.168.1.138 - 9062 packet(s)

Location: None/Internal


Top 10 IP addresses that packets were sent from:

192.168.1.254 - 2420 packet(s).  
91.221.58.40 - 1718 packet(s).  
143.196.252.80 - 1249 packet(s).  
23.73.138.114 - 644 packet(s).  
192.168.1.67 - 497 packet(s).  
52.142.125.222 - 435 packet(s).  
192.168.1.122 - 373 packet(s).  
52.142.124.215 - 271 packet(s).  
192.168.1.78 - 223 packet(s).  
142.250.200.36 - 200 packet(s).

Top 10 Ports that the packets were sent from:

443 - 6723 packet(s).  
80 - 1444 packet(s).  
Unknown Port(s) - 675 packet(s).  
8009 - 106 packet(s).  
8884 - 33 packet(s).  
53 - 20 packet(s).  
49902 - 16 packet(s).  
1900 - 16 packet(s).  
25 - 13 packet(s).  
51973 - 1 packet(s).

### 2. 2a00:23c4:108:201:a1dd:8290:603d:6fb2 - 2884 packet(s)

Location: Falkirk, Scotland, United Kingdom 

Top 10 IP addresses that packets were sent from:

2600:9000:21f7:9c00:6:2f4:93c0:93a1 - 2171 packet(s).  
2600:9000:21b3:5600:6:6fa7:5f40:93a1 - 321 packet(s).  
2a03:2880:f258:c4:face:b00c:0:32c2 - 162 packet(s).  
2a03:2880:f258:1c9:face:b00c:0:32c2 - 145 packet(s).  
2a00:23a0:192:101::5182:6b51 - 18 packet(s).

2a00:23a0:192:105::56bd:6a92 - 18 packet(s).  
2606:4700:4400::6812:2044 - 15 packet(s).  
2a00:1450:4009:823::200a - 15 packet(s).  
fe80::46fe:3bff:fed6:6b9a - 13 packet(s).  
2a00:17f0:1300:3285::4 - 6 packet(s).

Top 10 Ports that the packets were sent from:

443 - 2850 packet(s).  
80 - 21 packet(s).  
Unknown Port(s) - 13 packet(s).

### 3. 192.168.1.254 - 1242 packet(s)

Location: None/Internal

Top 10 IP addresses that packets were sent from:

192.168.1.138 - 1242 packet(s).

Top 10 Ports that the packets were sent from:

Unknown Port(s) - 85 packet(s).  
17984 - 35 packet(s).  
18083 - 33 packet(s).  
17805 - 29 packet(s).  
17893 - 28 packet(s).  
17559 - 27 packet(s).  
18039 - 27 packet(s).  
17942 - 27 packet(s).  
18132 - 27 packet(s).  
17763 - 27 packet(s).

### 4. Unknown IP(s) - 957 packet(s)

Location: Unknown

Top 10 IP addresses that packets were sent from:

Unknown IP(s) - 957 packet(s).



## 12.13 – Figure 13

```
C:\Users\Ganymede\Desktop\aaa>python Main.py -s ExampleSpecific
Launched in sniffing mode.
Starting packet sniffing, press 'Ctrl + C' to end sniffing.
Packet sniffing stopped!
<Sniffed: TCP:22054 UDP:8589 ICMP:3896 Other:3627>
Saving packet log as ExampleSpecific.pcap
```

```
C:\Users\Ganymede\Desktop\aaa>python Main.py -l ExampleSpecific.pcap
Loading file ExampleSpecific.pcap...
<ExampleSpecific.pcap: TCP:22054 UDP:8589 ICMP:3896 Other:3627>
Loading packet data...
100%|████████████████████████████████████████████████████████████████████████████████| 38166/38166 [02:03<00:00, 308.14it/s]
Generating Output Data
Source IP Graph Generated
Destination IP Graph Generated
Source Port Graph Generated
Destination Port Graph Generated
IPinfo connection successful
Source IP map Generated
Destination IP map Generated
Document Generated
Outputs saved in the "./ExampleSpecific.pcap Output" folder
```



## 12.14 – Figure 14

### 12. 2a03:2880:f258:c4:face:b00c:0:32c2 – 513 packet(s)

Location: Menlo Park, California, United States us

Top 10 IP addresses that the packets were sent to:

2a00:23c4:108:201:a1dd:8290:603d:6fb2 – 513 packet(s).

Top 10 Ports that the packets were sent to:

35210 – 168 packet(s).

35676 – 149 packet(s).

36612 – 139 packet(s).

35211 – 16 packet(s).

35675 – 16 packet(s).

35212 – 13 packet(s).

36611 – 12 packet(s).

### 13. 192.168.1.215 – 485 packet(s)

Location: None/Internal

Top 10 IP addresses that the packets were sent to:

224.0.0.251 – 368 packet(s).

192.168.1.255 – 107 packet(s).

224.0.0.22 – 10 packet(s).

Top 10 Ports that the packets were sent to:

5353 – 368 packet(s).

40777 – 107 packet(s).

Unknown Port(s) – 10 packet(s).

### 14. 2606:4700::6812:710 – 436 packet(s)

Location: San Francisco, California, United States us

Top 10 IP addresses that the packets were sent to:

2a00:23c4:108:201:a1dd:8290:603d:6fb2 – 436 packet(s).

Top 10 Ports that the packets were sent to:

60875 – 436 packet(s).

### 17. 2a03:2880:f258:c4:face:b00c:0:32c2 – 276 packet(s)

Location: Menlo Park, California, United States us

Top 10 IP addresses that packets were sent from:

2a00:23c4:108:201:a1dd:8290:603d:6fb2 – 276 packet(s).

Top 10 Ports that the packets were sent from:

35210 – 82 packet(s).

35676 – 78 packet(s).

36612 – 73 packet(s).

35211 – 12 packet(s).

35675 – 12 packet(s).

35212 – 10 packet(s).

36611 – 9 packet(s).

### 18. 192.168.1.101 – 245 packet(s)

Location: None/Internal

Top 10 IP addresses that packets were sent from:

192.168.1.138 – 245 packet(s).

Top 10 Ports that the packets were sent from:

Unknown Port(s) – 245 packet(s).

### 19. 239.255.255.250 – 210 packet(s)

Location: None/Internal

Top 10 IP addresses that packets were sent from:

192.168.1.134 – 108 packet(s).

192.168.1.138 – 72 packet(s).

192.168.1.254 – 18 packet(s).

192.168.1.105 – 6 packet(s).

192.168.1.93 – 3 packet(s).

192.168.1.230 – 3 packet(s).

## 12.15 – Figure 15

IP addresses that 2a03:2880:f258:c4:face:b00c:0:32c2 sent packets to:

2a00:23c4:108:201:a1dd:8290:603d:6fb2 – 513 packet(s).

IP addresses that 2a03:2880:f258:c4:face:b00c:0:32c2 has recieved packets from:

2a00:23c4:108:201:a1dd:8290:603d:6fb2 – 513 packet(s).

Ports that 2a03:2880:f258:c4:face:b00c:0:32c2 sent packets to:

35210 – 168 packet(s).

35676 – 149 packet(s).

36612 – 139 packet(s).

35211 – 16 packet(s).

35675 – 16 packet(s).

35212 – 13 packet(s).

36611 – 12 packet(s).

Ports that 2a03:2880:f258:c4:face:b00c:0:32c2 has recieved packets from:

35210 – 168 packet(s).

35676 – 149 packet(s).

36612 – 139 packet(s).

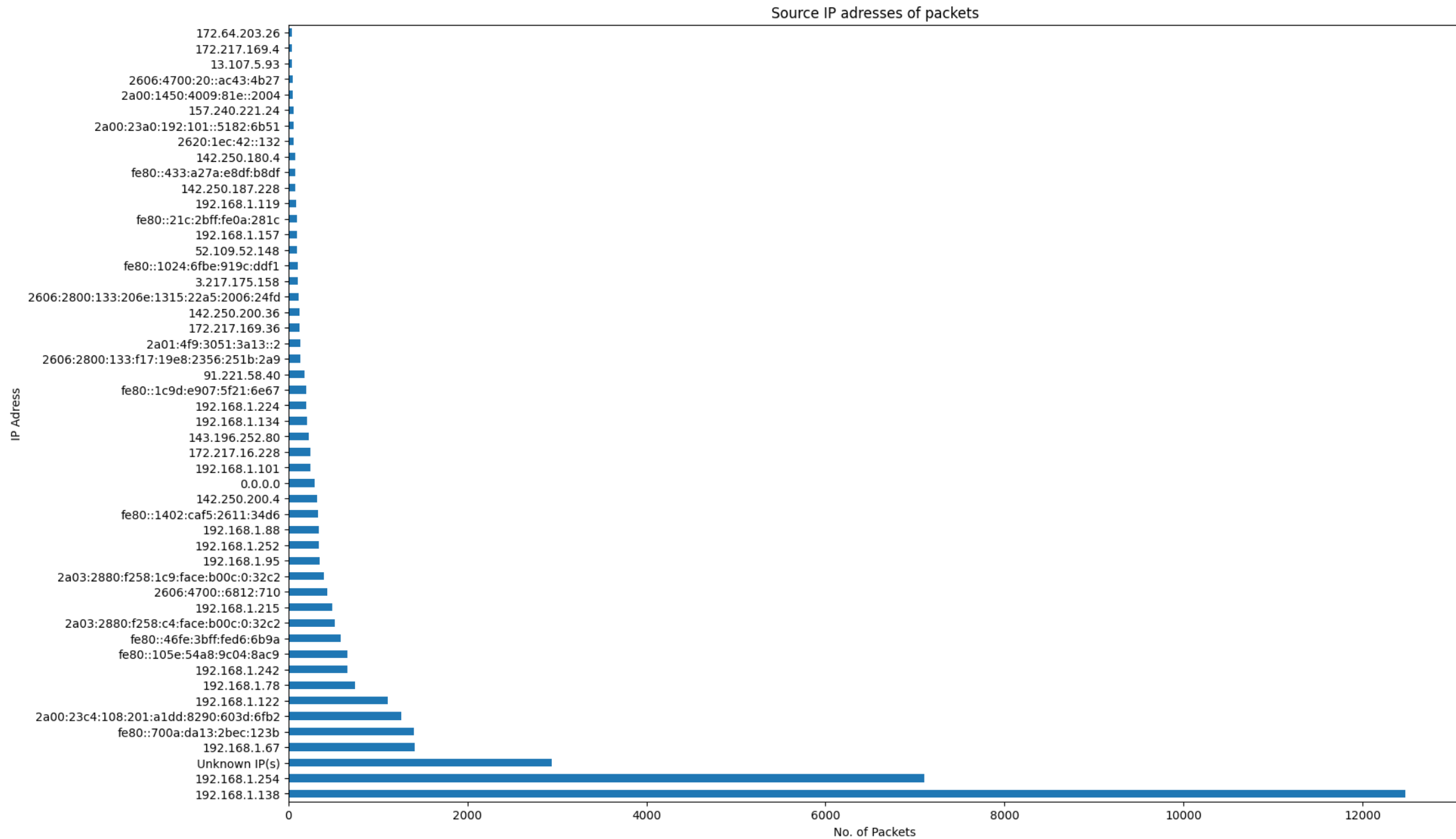
35211 – 16 packet(s).

35675 – 16 packet(s).

35212 – 13 packet(s).

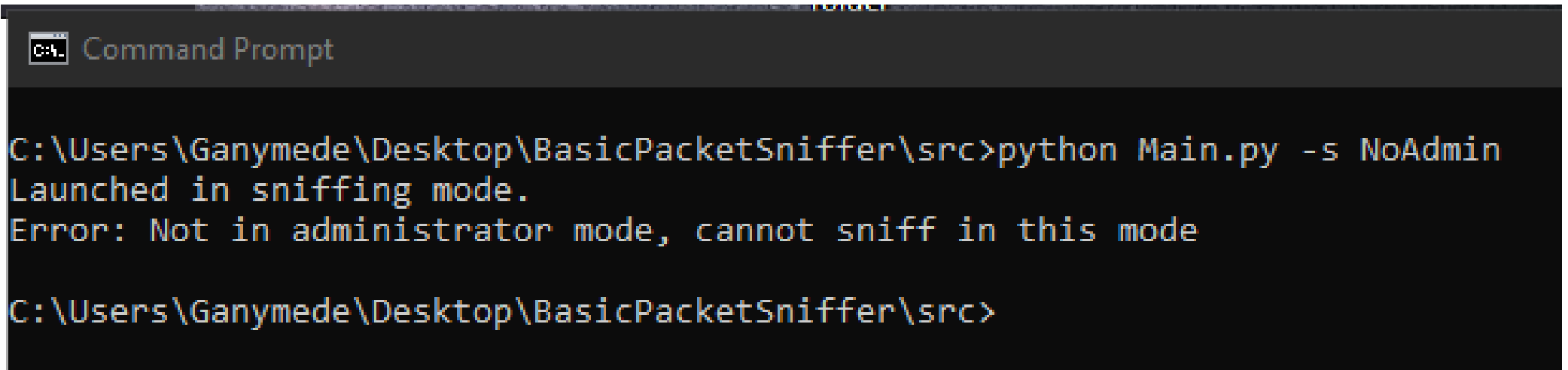
36611 – 12 packet(s).

## 12.16 – Figure 16





## 12.17 – Figure 17



```
Command Prompt

C:\Users\Ganymede\Desktop\BasicPacketSniffer\src>python Main.py -s NoAdmin
Launched in sniffing mode.
Error: Not in administrator mode, cannot sniff in this mode

C:\Users\Ganymede\Desktop\BasicPacketSniffer\src>
```

## 12.18 – Figure 18

Administrator: Command Prompt

```
C:\Users\Ganymede\Desktop\BasicPacketSniffer\src>python Main.py -s FilterTest -m 0
Invalid Max Number

C:\Users\Ganymede\Desktop\BasicPacketSniffer\src>python Main.py -s FilterTest -m -5
Invalid Max Number

C:\Users\Ganymede\Desktop\BasicPacketSniffer\src>python Main.py -s FilterTest -m Apple
Invalid Max Number
```

Command Prompt

```
C:\Users\Ganymede\Desktop\BasicPacketSniffer\src>python Main.py -s ErrorTesting -m 0
Invalid Max Number

C:\Users\Ganymede\Desktop\BasicPacketSniffer\src>python Main.py -s ErrorTesting -m -5
Invalid Max Number

C:\Users\Ganymede\Desktop\BasicPacketSniffer\src>python Main.py -s ErrorTesting -m apple
Invalid Max Number

C:\Users\Ganymede\Desktop\BasicPacketSniffer\src>python Main.py -s ErrorTesting -m
Command Error!
-h                Prints a list of all commands, not running the program.

-s "filename"     Starts in sniffing mode, saving the resulting packet log.
-l "filename"     Starts the program in packet analysis mode loading an existing .pcap file.
Use -s to sniff and save a .pcap file, then use -l to load and generate the packet analysis.

-f "filter"       Specifes the filters to be applied to the sniffing process, uses Berkeley Packet Filter syntax. (For -s)#TODO
-p               Flags to print the packets as they are intercepted. (For -s)

-i "IP adress"    Generates a report about only packets sent to or from this IP adress, giving more information than the normal report. (For -l)

-m "Number"       In sniff mode sets the max amopunt of packets to intercept before ending, no value sniffs infinitely. (For -s)
                  In load mode sets the maximum number of enteries in the charts and document, no value defaults to 50. (For -l)

C:\Users\Ganymede\Desktop\BasicPacketSniffer\src>
```

## 12.19 – Figure 19

```
C:\Users\Ganymede\Desktop\BasicPacketSniffer\src>python Main.py -l NonExistanFile
Loading file NonExistanFile...
Error: Invalid File loaded!

C:\Users\Ganymede\Desktop\BasicPacketSniffer\src>python Main.py -l test.txt
Loading file test.txt...
Error: Invalid File loaded!

C:\Users\Ganymede\Desktop\BasicPacketSniffer\src>python Main.py -l :
Loading file :...
Error: Invalid File loaded!

C:\Users\Ganymede\Desktop\BasicPacketSniffer\src>python Main.py -l
Command Error!
-h          Prints a list of all commands, not running the program.

-s "filename"  Starts in sniffing mode, saving the resulting packet log.
-l "filename"  Starts the program in packet analysis mode loading an existing .pcap file.
Use -s to sniff and save a .pcap file, then use -l to load and generate the packet analysis.

-f "filter"    Specifes the filters to be applied to the sniffing process, uses Berkeley Packet Filter syntax. (For -s)#TODO
-p            Flags to print the packets as they are intercepted. (For -s)

-i "IP adress" Generates a report about only packets sent to or from this IP adress, giving more information than the normal report. (For -l)

-m "Number"   In sniff mode sets the max amopunt of packets to intercept before ending, no value sniffs infinitely. (For -s)
              In load mode sets the maximum number of enteries in the charts and document, no value defaults to 50. (For -l)
```

```
C:\> Administrator: Command Prompt
Microsoft Windows [Version 10.0.19044.2846]
(c) Microsoft Corporation. All rights reserved.

C:\WINDOWS\system32>cd C:\Users\Ganymede\Desktop\BasicPacketSniffer\src

C:\Users\Ganymede\Desktop\BasicPacketSniffer\src>python Main.py -s IPtest
Launched in sniffing mode.
Starting packet sniffing, press 'Ctrl + C' to end sniffing.
Packet sniffing stopped!
<Sniffed: TCP:180 UDP:118 ICMP:50 Other:33>
Saving packet log as IPtest.pcap

C:\Users\Ganymede\Desktop\BasicPacketSniffer\src>python Main.py -l IPtest.pcap -i NotAnIpAddress
Loading file IPtest.pcap...
<IPtest.pcap: TCP:180 UDP:118 ICMP:50 Other:33>
Loading packet data...
100%|██████████████████████████████████████████████████████████████████████████████| 381/381 [00:00<00:00, 838.89it/s]
Generating Output Data for IP NotAnIpAddress
Error: Specified IP address doesn't occur.

C:\Users\Ganymede\Desktop\BasicPacketSniffer\src>
```

## 12.21 – Figure 21

C:\> Administrator: Command Prompt

```
C:\Users\Ganymede\Desktop\BasicPacketSniffer\src>python Main.py -s FilterTest -f "NotValidFilter"
Launched in sniffing mode.
Starting packet sniffing, press 'Ctrl + C' to end sniffing.
Filtering with BPF syntax expression "NotValidFilter".
ERROR: Could not compile filter expression NotValidFilter
Packet sniffing stopped!
<Sniffed: TCP:194 UDP:36 ICMP:20 Other:29>
Saving packet log as FilterTest.pcap

C:\Users\Ganymede\Desktop\BasicPacketSniffer\src>
```

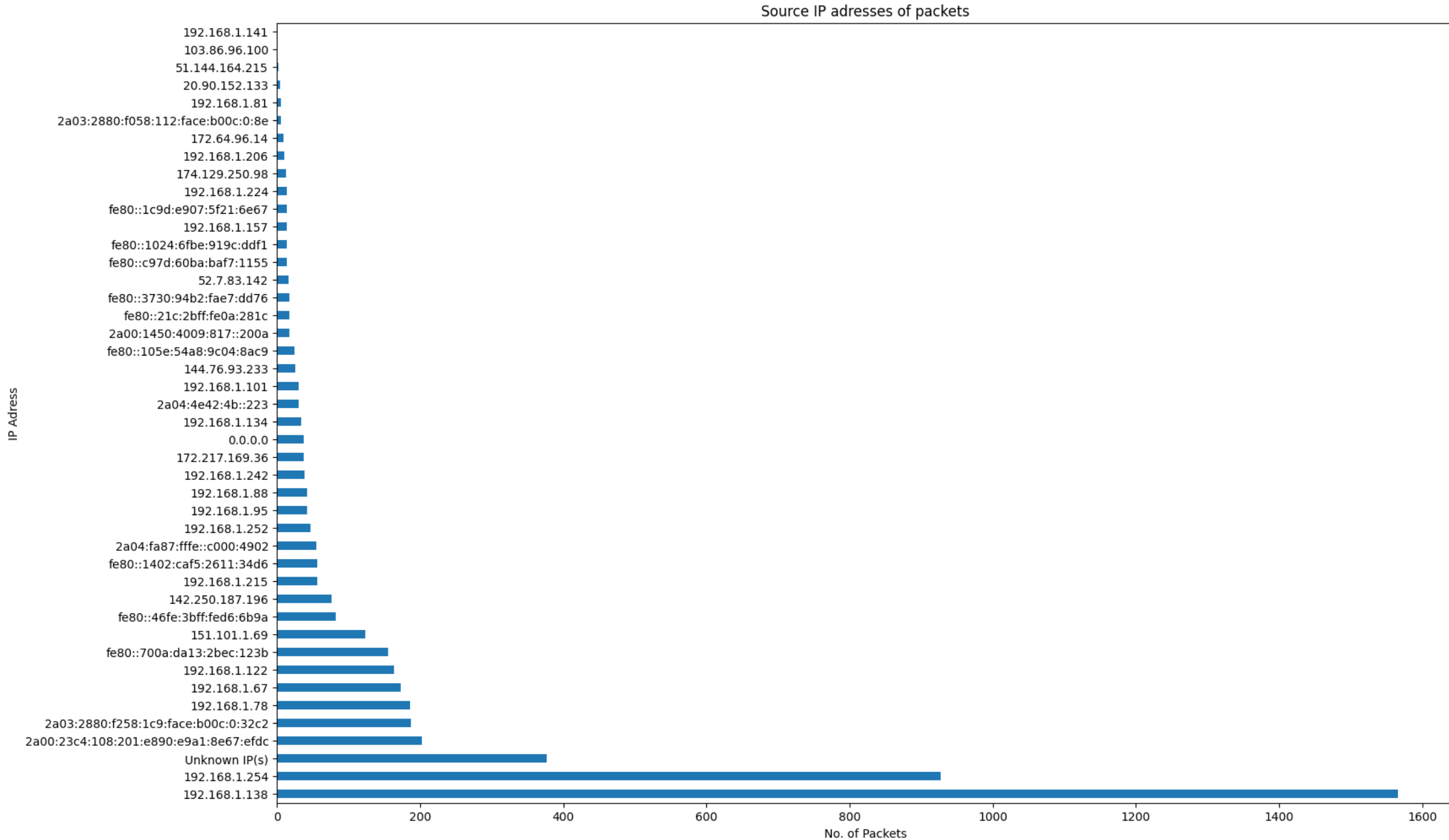
```
C:\Users\Ganymede\Desktop\BasicPacketSniffer>python Main.py -l MaxEntryTest.pcap
Launched in sniffing mode.
Starting packet sniffing, press 'Ctrl + C' to end sniffing.
Packet sniffing stopped!
<Sniffed: TCP:3096 UDP:893 ICMP:500 Other:478>
Saving packet log as MaxEntryTest.pcap

C:\Users\Ganymede\Desktop\BasicPacketSniffer\src>python Main.py -l MaxEntryTest.pcap
Loading file MaxEntryTest.pcap...
<MaxEntryTest.pcap: TCP:3096 UDP:893 ICMP:500 Other:478>
Loading packet data...
100% |██████████████████████████████████████████████████████████████████████████████| 4967/4967 [00:06<00:00, 811.89it/s]
Generating Output Data
Source IP Graph Generated
Destination IP Graph Generated
Source Port Graph Generated
Destination Port Graph Generated
IPinfo connection successful
Source IP map Generated
Destination IP map Generated
Document Generated
Outputs saved in the "./MaxEntryTest.pcap Output" folder

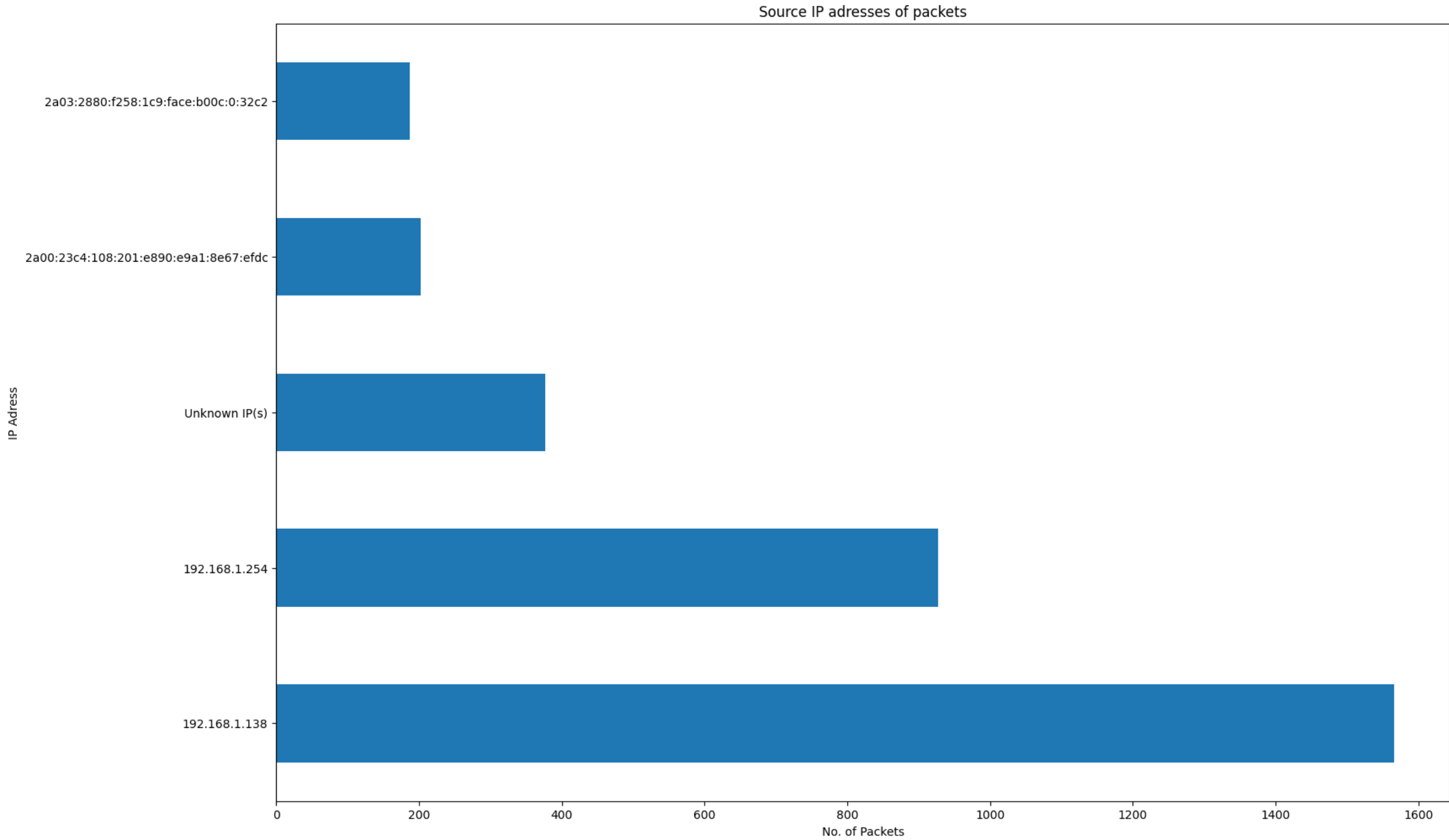
C:\Users\Ganymede\Desktop\BasicPacketSniffer\src>python Main.py -l MaxEntryTest.pcap -m 5
Loading file MaxEntryTest.pcap...
<MaxEntryTest.pcap: TCP:3096 UDP:893 ICMP:500 Other:478>
Loading packet data...
100% |██████████████████████████████████████████████████████████████████████████████| 4967/4967 [00:06<00:00, 795.83it/s]
Generating Output Data
Source IP Graph Generated
Destination IP Graph Generated
Source Port Graph Generated
Destination Port Graph Generated
IPinfo connection successful
Source IP map Generated
Destination IP map Generated
Document Generated
Outputs saved in the "./MaxEntryTest.pcap Output" folder

C:\Users\Ganymede\Desktop\BasicPacketSniffer\src>
```

## 12.23 – Figure 23



## 12.24 – Figure 24





## 12.25 – Figure 25

 Administrator: Command Prompt - python Main.py -s NoInternetTest -p

```
C:\Users\Ganymede\Desktop\BasicPacketSniffer\src>python Main.py -s NoInternetTest -p
Launched in sniffing mode.
Starting packet sniffing, press 'Ctrl + C' to end sniffing.
```

### 12.26 – Figure 26

```
C:\Users\Ganymede\Desktop\BasicPacketSniffer\src>python Main.py -l Test.pcap  
Loading file Test.pcap...  
<Test.pcap: TCP:3096 UDP:893 ICMP:500 Other:478>  
Loading packet data...  
100%|██████████████████████████████████████████████████| 4967/4967 [00:06<00:00, 765.81it/s]  
Generating Output Data  
Source IP Graph Generated  
Destination IP Graph Generated  
Source Port Graph Generated  
Destination Port Graph Generated  
IPinfo connection unsuccessful  
Can't generate IP location maps  
Document Generated  
Outputs saved in the "./Test.pcap Output" folder  
  
C:\Users\Ganymede\Desktop\BasicPacketSniffer\src>
```

## 12.27 – Figure 27

### Top 50 packet source IP addresses

#### 1. 192.168.1.138 – 1566 packet(s)

Location: Unknown

Top 10 IP addresses that the packets were sent to:

192.168.1.254 – 479 packet(s).  
192.168.1.255 – 186 packet(s).  
192.168.1.122 – 183 packet(s).  
192.168.1.67 – 168 packet(s).  
224.0.0.251 – 98 packet(s).  
192.168.1.78 – 81 packet(s).  
151.101.1.69 – 69 packet(s).  
192.168.1.252 – 50 packet(s).  
192.168.1.88 – 45 packet(s).  
192.168.1.95 – 45 packet(s).

Top 10 Ports that the packets were sent to:

443 – 677 packet(s).  
80 – 263 packet(s).  
Unknown Port(s) – 253 packet(s).  
54915 – 186 packet(s).  
5353 – 98 packet(s).  
8009 – 51 packet(s).  
8884 – 26 packet(s).  
1900 – 8 packet(s).  
53 – 4 packet(s).

#### 2. 192.168.1.254 – 928 packet(s)

Location: Unknown

Top 10 IP addresses that the packets were sent to:

192.168.1.138 – 927 packet(s).  
224.0.0.1 – 1 packet(s).

Top 10 Ports that the packets were sent to:

6958 – 63 packet(s).  
7112 – 63 packet(s).  
7260 – 63 packet(s).

7039 – 63 packet(s).  
7336 – 61 packet(s).  
7408 – 61 packet(s).  
7382 – 58 packet(s).  
7086 – 58 packet(s).  
7028 – 58 packet(s).

#### 3. Unknown IP(s) – 377 packet(s)

Location: Unknown

Top 10 IP addresses that the packets were sent to:

Unknown IP(s) – 377 packet(s).

Top 10 Ports that the packets were sent to:

Unknown Port(s) – 377 packet(s).

#### 4. 2a00:23c4:108:201:e890:e9a1:8e67:efdc – 203 packet(s)

Location: Unknown

Top 10 IP addresses that the packets were sent to:

2a03:2880:f258:1c9:face:b00c:0:32c2 – 100 packet(s).  
2a04:fa87:fffe::c000:4902 – 45 packet(s).  
2a04:4e42:4b::223 – 26 packet(s).  
2a00:1450:4009:817::200a – 16 packet(s).  
2a03:2880:f058:112:face:b00c:0:8e – 12 packet(s).  
fe80::46fe:3bff:fed6:6b9a – 4 packet(s).

Top 10 Ports that the packets were sent to:

443 – 199 packet(s).  
Unknown Port(s) – 4 packet(s).

#### 5. 2a03:2880:f258:1c9:face:b00c:0:32c2 – 187 packet(s)

Location: Unknown

Top 10 IP addresses that the packets were sent to:

2a00:23c4:108:201:e890:e9a1:8e67:efdc – 187 packet(s).

## 12.28 – Figure 28

```
[04/25/23]seed@VM:~$ sudo tcpdump
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on enp0s3, link-type EN10MB (Ethernet), capture size 262144 bytes
20:50:39.109879 IP s3-eu-west-1-r-w.amazonaws.com.https > VM.40014: Flags [P.], seq 12936:12967, ack 3656015386, win 31505, length 31
20:50:39.109920 IP VM.40014 > s3-eu-west-1-r-w.amazonaws.com.https: Flags [.], ack 31, win 62780, length 0
20:50:39.110238 IP VM.40014 > s3-eu-west-1-r-w.amazonaws.com.https: Flags [P.], seq 1:32, ack 31, win 62780, length 31
20:50:39.110267 IP VM.40014 > s3-eu-west-1-r-w.amazonaws.com.https: Flags [F.], seq 32, ack 31, win 62780, length 0
20:50:39.110519 IP s3-eu-west-1-r-w.amazonaws.com.https > VM.40014: Flags [.], ack 33, win 31473, length 0
20:50:39.111565 IP VM.45065 > 192.168.1.254.domain: 18155+ [1au] PTR? 4.2.0.10.in-addr.arpa. (50)
20:50:39.114743 IP 192.168.1.254.domain > VM.45065: 18155 NXDomain 0/0/1 (50)
20:50:39.114859 IP VM.45065 > 192.168.1.254.domain: 18155+ PTR? 4.2.0.10.in-addr.arpa. (39)
20:50:39.116950 IP 192.168.1.254.domain > VM.45065: 18155 NXDomain 0/0/0 (39)
20:50:39.117992 IP VM.43165 > 192.168.1.254.domain: 49898+ [1au] PTR? 254.1.168.192.in-addr.arpa. (55)
20:50:39.134363 IP 192.168.1.254.domain > VM.43165: 49898 NXDomain 0/0/1 (55)
20:50:39.134491 IP VM.43165 > 192.168.1.254.domain: 49898+ PTR? 254.1.168.192.in-addr.arpa. (44)
20:50:39.136771 IP s3-eu-west-1-r-w.amazonaws.com.https > VM.40014: Flags [F.], seq 31, ack 33, win 31473, length 0
20:50:39.136787 IP VM.40014 > s3-eu-west-1-r-w.amazonaws.com.https: Flags [.], ack 32, win 62780, length 0
20:50:39.152175 IP 192.168.1.254.domain > VM.43165: 49898 NXDomain 0/0/0 (44)
^Z
[3]+  Stopped                  sudo tcpdump
[04/25/23]seed@VM:~$
```

## 12.29 – Figure 29

\*4 interfaces

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

Apply a display filter ... <Ctrl-/>

No.	Source	Destination	Protocol	Length	Info
1	127.0.0.1	127.0.0.1	TCP	56	48015 → 6468 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
2	127.0.0.1	127.0.0.1	TCP	44	6468 → 48015 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
3	127.0.0.1	127.0.0.1	TCP	56	48013 → 6467 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
4	127.0.0.1	127.0.0.1	TCP	44	6467 → 48013 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
5	127.0.0.1	127.0.0.1	TCP	56	48016 → 6468 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
6	127.0.0.1	127.0.0.1	TCP	44	6468 → 48016 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
7	192.168.1.138	192.168.1.254	DNS	69	Standard query 0x5ea4 A localhost
8	192.168.1.138	192.168.1.95	ICMP	74	Echo (ping) request id=0x000a, seq=13878/13878, ttl=128 (reply in 12)
9	192.168.1.138	192.168.1.252	ICMP	74	Echo (ping) request id=0x000a, seq=13879/14134, ttl=128 (reply in 11)
10	192.168.1.254	192.168.1.138	DNS	85	Standard query response 0x5ea4 A localhost A 127.0.0.1
11	192.168.1.252	192.168.1.138	ICMP	74	Echo (ping) reply id=0x000a, seq=13879/14134, ttl=255 (request in 9)
12	192.168.1.95	192.168.1.138	ICMP	74	Echo (ping) reply id=0x000a, seq=13878/13878, ttl=255 (request in 8)
13	192.168.1.138	192.168.1.252	ICMP	74	Echo (ping) request id=0x000a, seq=13883/15158, ttl=128 (reply in 14)
14	192.168.1.252	192.168.1.138	ICMP	74	Echo (ping) reply id=0x000a, seq=13883/15158, ttl=255 (request in 13)
15	192.168.1.138	192.168.1.95	ICMP	74	Echo (ping) request id=0x000a, seq=13884/15414, ttl=128 (reply in 16)
16	192.168.1.95	192.168.1.138	ICMP	74	Echo (ping) reply id=0x000a, seq=13884/15414, ttl=255 (request in 15)
17	192.168.1.138	192.168.1.252	ICMP	74	Echo (ping) request id=0x000a, seq=13886/15926, ttl=128 (reply in 19)
18	192.168.1.138	192.168.1.95	ICMP	74	Echo (ping) request id=0x000a, seq=13887/16182, ttl=128 (reply in 27)
19	192.168.1.252	192.168.1.138	ICMP	74	Echo (ping) reply id=0x000a, seq=13886/15926, ttl=255 (request in 17)
20	192.168.56.1	192.168.56.1	ICMP	64	Echo (ping) request id=0x000a, seq=13877/13622, ttl=128 (reply in 21)
21	192.168.56.1	192.168.56.1	ICMP	64	Echo (ping) reply id=0x000a, seq=13877/13622, ttl=64 (request in 20)
22	127.0.0.1	127.0.0.1	TLSv1.2	191	Application Data
23	127.0.0.1	127.0.0.1	TCP	44	23560 → 1170 [ACK] Seq=1 Ack=148 Win=10054 Len=0

> Frame 1: 56 bytes on wire (448 bits), 56 bytes captured (448 bits) on 000002 00 00 00 45 00 00 34 72 a4 40 00 40 06 00 00 ...E..4 r.@.@...

wireshark\_4\_interfacesUJAR31.pcapng || Packets: 130 · Displayed: 130 (100.0%) || Profile: Default

12.30 –

Figure 30

