HW5 Report

Task 1: PCAP Analysis

Task 1: run tshark to perform PCAPC Analysis

a. typed `man tshark`

```
TSHARK(1)                         The Wireshark Network Analyzer                         TSHARK(1)

NAME
       tshark - Dump and analyze network traffic

SYNOPSIS
       tshark [ -2 ] [ -a <capture autostop condition> ] ...
       [ -b <capture ring buffer option>] ...  [ -B <capture buffer size> ]
       [ -c <capture packet count> ] [ -C <configuration profile> ]
       [ -d <layer type>==<selector>,<decode-as protocol> ] [ -D ] [ -e <field> ]
       [ -E <field print option> ] [ -f <capture filter> ] [ -F <file format> ] [ -g ] [ -h ]
       [ -H <input hosts file> ] [ -i <capture interface>|- ] [ -I ] [ -K <keytab> ] [ -l ]
       [ -L ] [ -n ] [ -N <name resolving flags> ] [ -o <preference setting> ] ...
       [ -O <protocols> ] [ -p ] [ -P ] [ -q ] [ -Q ] [ -r <infile> ] [ -R <Read filter> ]
       [ -s <capture snaplen> ] [ -S <separator> ] [ -t a|ad|adoy|d|dd|e|r|u|ud|udoy ]
       [ -T fields|pdml|ps|psml|text ] [ -u <seconds type>] [ -v ] [ -V ] [ -w <outfile>|- ]
       [ -W <file format option>] [ -x ] [ -X <eXtension option>] [ -y <capture link type> ]
       [ -Y <displaY filter> ] [ -z <statistics> ] [ --capture-comment <comment> ]
       [ <capture filter> ]

       tshark -G [ <report type> ]

DESCRIPTION
       TShark is a network protocol analyzer.  It lets you capture packet data from a live
       network, or read packets from a previously saved capture file, either printing a decoded
       form of those packets to the standard output or writing the packets to a file.  TShark's
       native capture file format is pcap format, which is also the format used by tcpdump and
       various other tools.
```

- read through the manual

b. typed `tshark -T fields -e frame.number -e frame.time -e telnet.data -r telnet.pcap`

```
158     Sep 15, 2017 16:41:52.152987000 UTC     Ubuntu 16.04.1 LTS

159     Sep 15, 2017 16:41:52.192950000 UTC
160     Sep 15, 2017 16:41:52.192976000 UTC     server login:
161     Sep 15, 2017 16:41:52.192995000 UTC
162     Sep 15, 2017 16:41:55.238745000 UTC     admin

163     Sep 15, 2017 16:41:55.239814000 UTC     admin

164     Sep 15, 2017 16:41:55.239861000 UTC
165     Sep 15, 2017 16:41:55.242365000 UTC     Password:
166     Sep 15, 2017 16:41:55.242401000 UTC
167     Sep 15, 2017 16:42:00.963166000 UTC     admin-password

168     Sep 15, 2017 16:42:00.963971000 UTC

169     Sep 15, 2017 16:42:00.963993000 UTC
170     Sep 15, 2017 16:42:03.820267000 UTC

171     Sep 15, 2017 16:42:03.820306000 UTC
172     Sep 15, 2017 16:42:03.820469000 UTC     Login incorrect

173     Sep 15, 2017 16:42:03.820484000 UTC
174     Sep 15, 2017 16:42:03.821778000 UTC     server login:
175     Sep 15, 2017 16:42:03.821797000 UTC
176     Sep 15, 2017 16:42:05.907777000 UTC     john

177     Sep 15, 2017 16:42:05.908174000 UTC     john

178     Sep 15, 2017 16:42:05.908196000 UTC
179     Sep 15, 2017 16:42:05.909525000 UTC     Password:
180     Sep 15, 2017 16:42:05.909542000 UTC
181     Sep 15, 2017 16:42:13.756176000 UTC     john-password

182     Sep 15, 2017 16:42:13.757236000 UTC

183     Sep 15, 2017 16:42:13.757286000 UTC
184     Sep 15, 2017 16:42:16.378455000 UTC
,Login incorrect

185     Sep 15, 2017 16:42:16.378503000 UTC
```

- got an output of this format, there’re fields from 1 - 157 (skipped as just frame number → frame time) and 185 - 198 (skipped as just frame number → frame time)

Task 2: Display the single packet containing invalid “admin” password

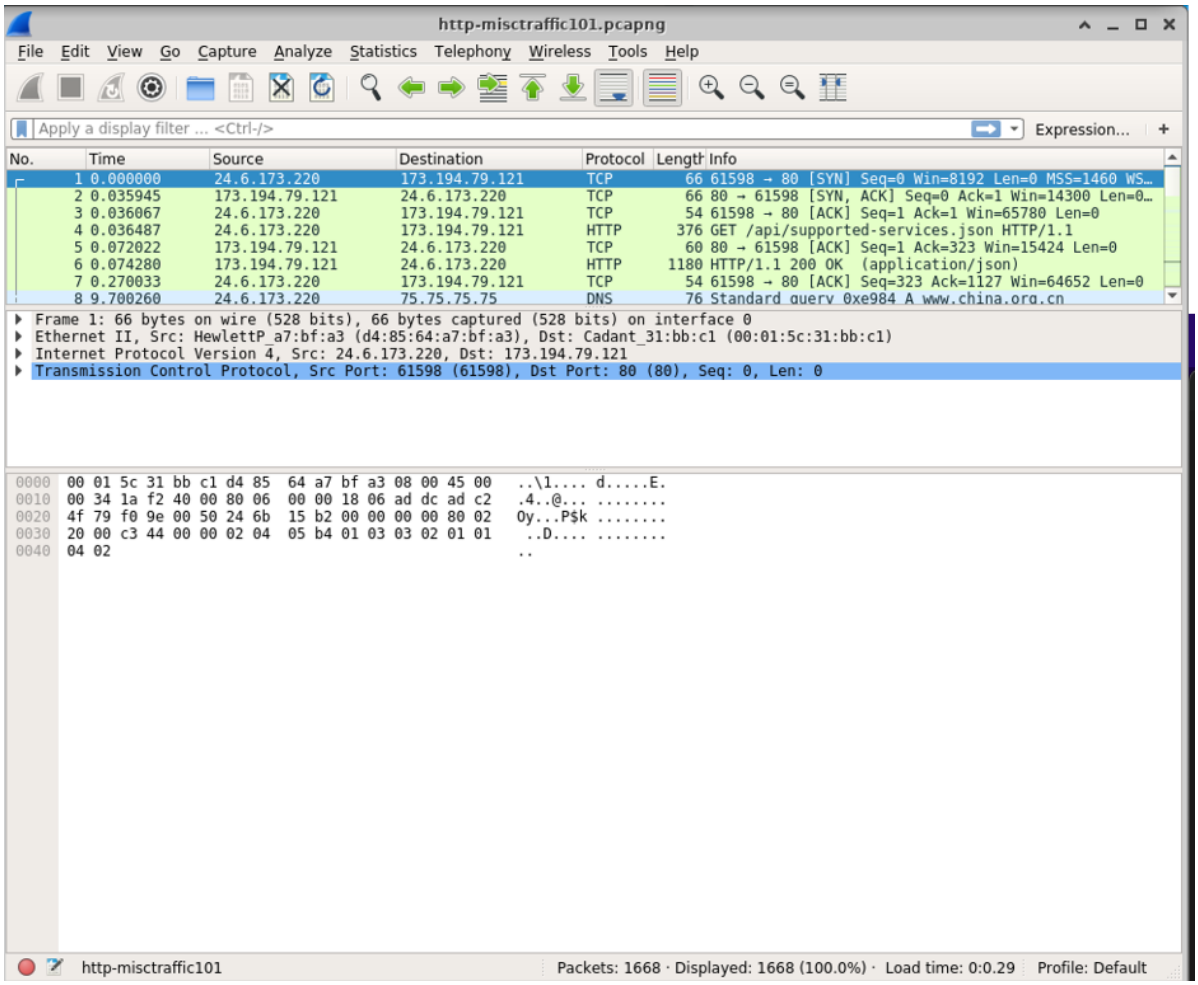
1. from previous image, we notice that the frame containing the incorrect login of admin user was frame 167
2. So we can just append `-Y frame.number==167` to the previous command
3. so `tshark -T fields -e frame.number -e frame.time -e telnet.data -r telnet.pcap -Y frame.number==167`
4. obtain the following result

```
ubuntu@pcapanalysis:~$ tshark -T fields -e frame.number -e frame.time -e telnet.data -r telnet.pcap -Y frame.number==167
167     Sep 15, 2017 16:42:00.963166000 UTC     admin-password
```

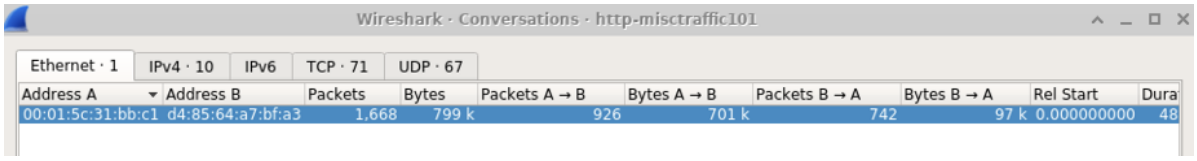
Task 2: Packet Capturing

Task 3.1: Find Most Active TCP flow

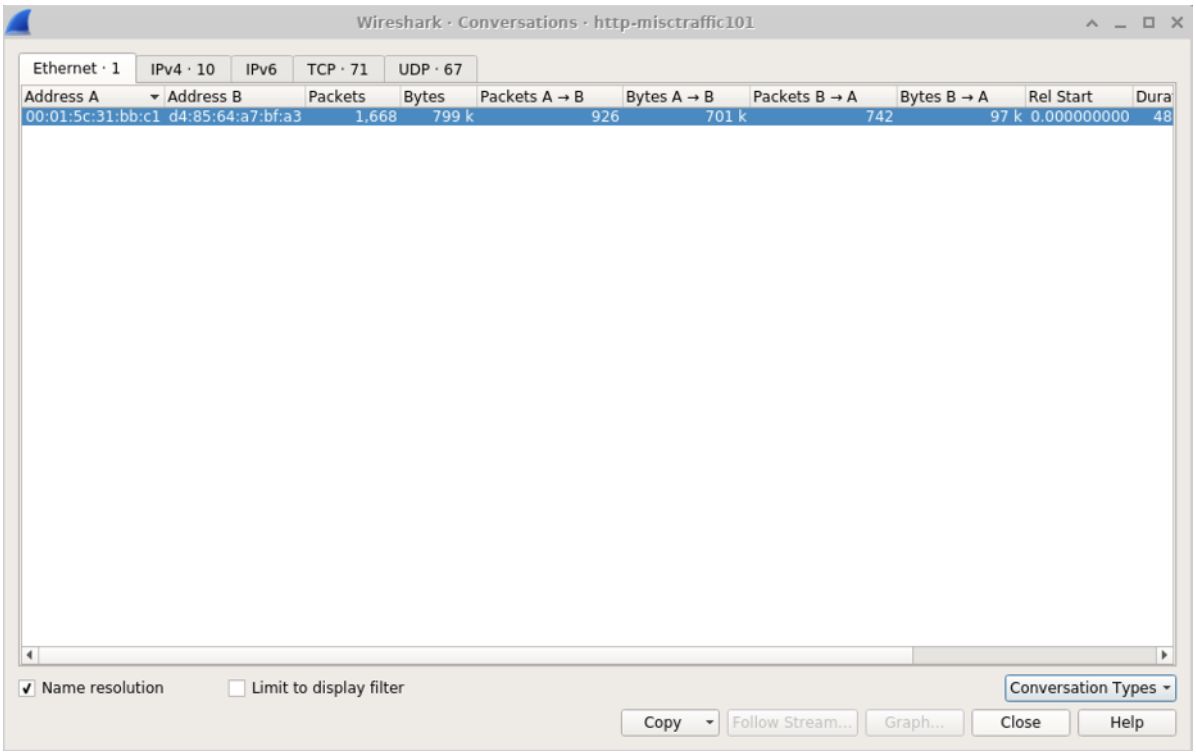
1. Opened the file in wireshark



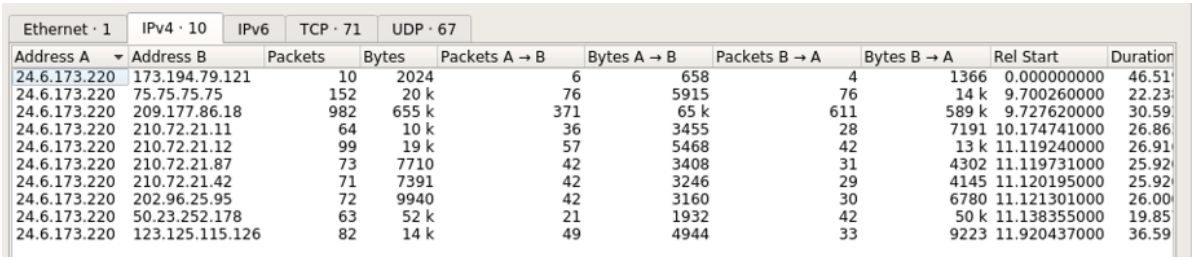
2. Selected Statistics and then Conversation, saw that there's only 1 pair of hosts communicating on the local network



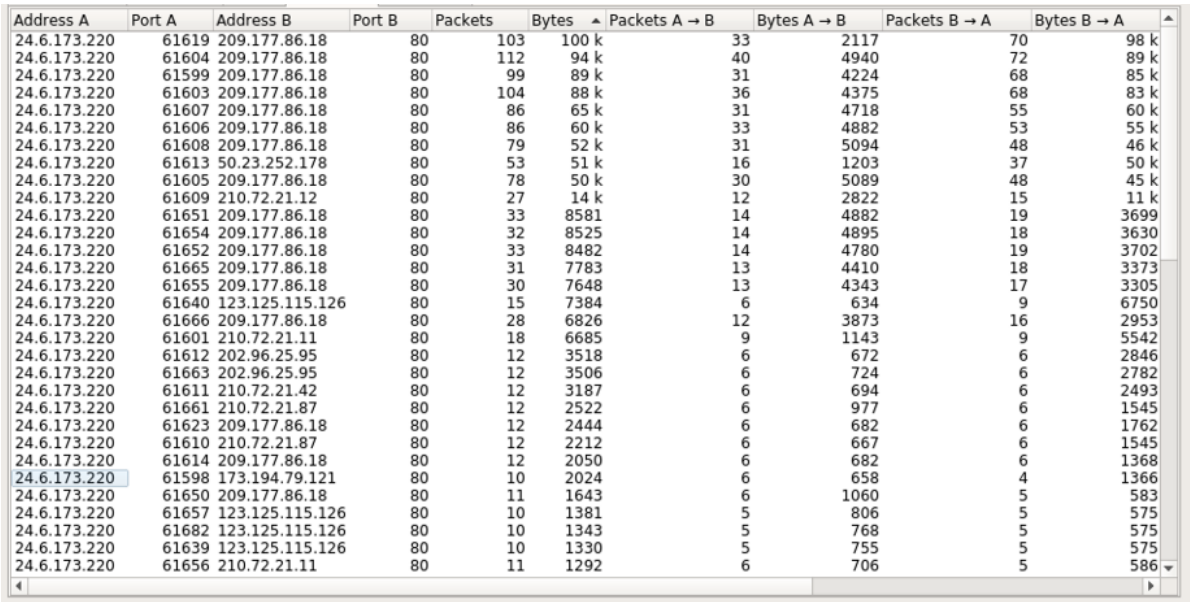
3. checked the name resolution box but nothing changed which is different from the lab instruction that "The MAC address listed as Cadant is the local router. The Flextron host is the client from which the traffic was captured."



4. Clicked on the IPv4 tab and saw this

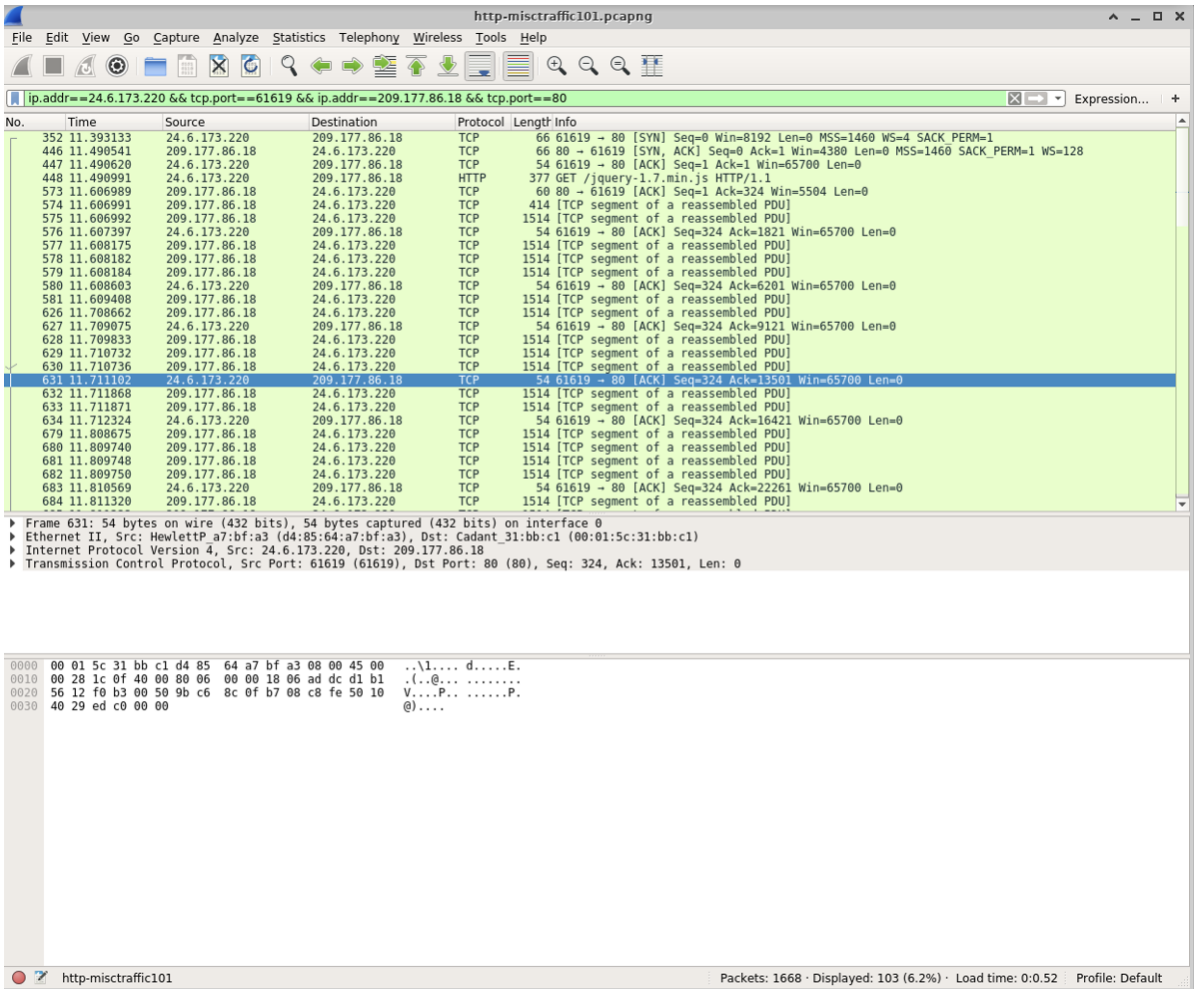


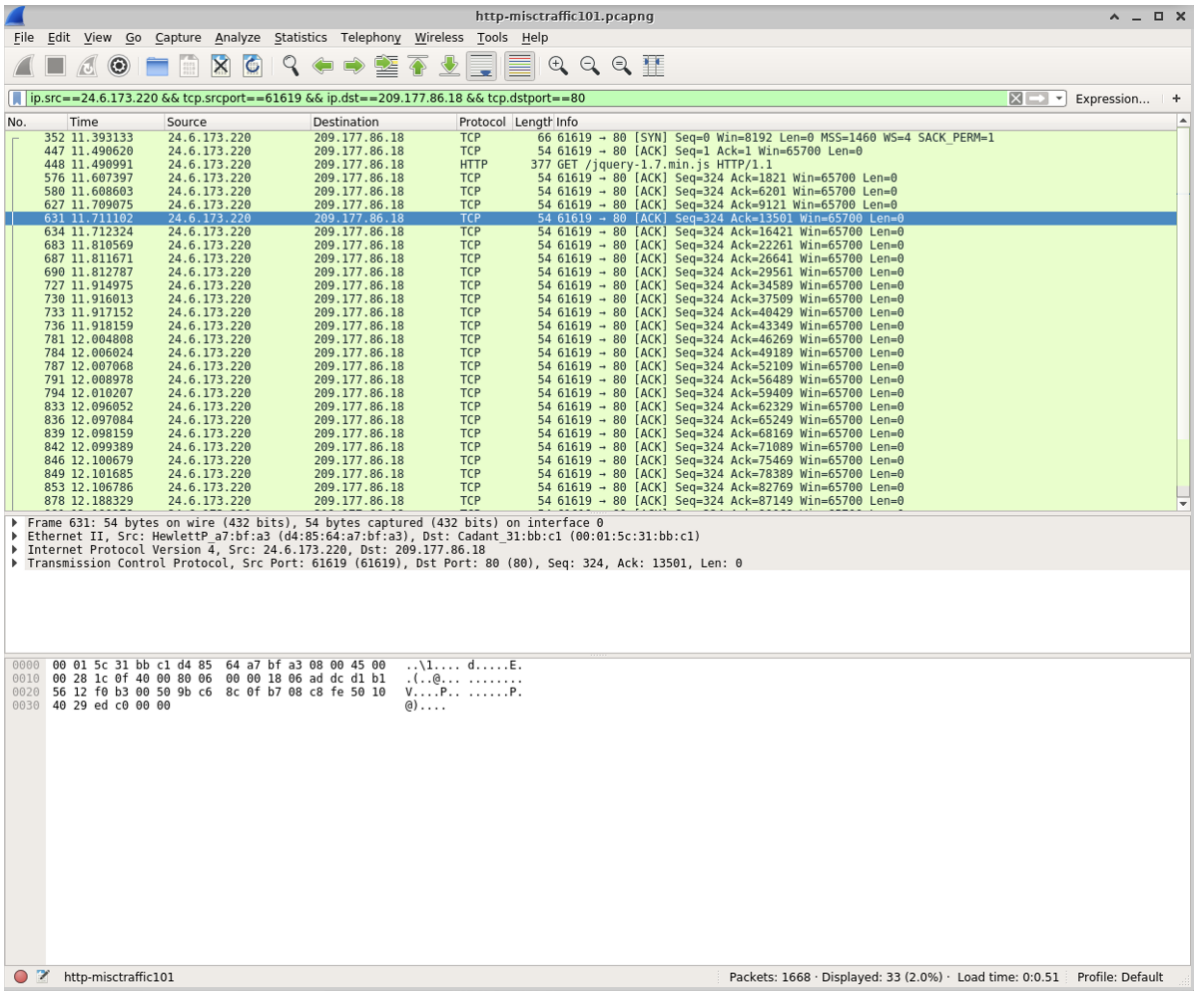
5. based on the byte count here, IP address: `209.177.86.18` participate in the most active IPv4 conversation which has a byte count of `655k`
6. Clicked on the TCP tab and saw this (sorted by byte count)



Address A	Port A	Address B	Port B	Packets	Bytes	Packets A → B	Bytes A → B	Packets B → A	Bytes B → A
24.6.173.220	61619	209.177.86.18	80	103	100 k	33	2117	70	98 k
24.6.173.220	61604	209.177.86.18	80	112	94 k	40	4940	72	89 k
24.6.173.220	61599	209.177.86.18	80	99	89 k	31	4224	68	85 k
24.6.173.220	61603	209.177.86.18	80	104	88 k	36	4375	68	83 k
24.6.173.220	61607	209.177.86.18	80	86	65 k	31	4718	55	60 k
24.6.173.220	61606	209.177.86.18	80	86	60 k	33	4882	53	55 k
24.6.173.220	61608	209.177.86.18	80	79	52 k	31	5094	48	46 k
24.6.173.220	61613	50.23.252.178	80	53	51 k	16	1203	37	50 k
24.6.173.220	61605	209.177.86.18	80	78	50 k	30	5089	48	45 k
24.6.173.220	61609	210.72.21.12	80	27	14 k	12	2822	15	11 k
24.6.173.220	61651	209.177.86.18	80	33	8581	14	4882	19	3699
24.6.173.220	61654	209.177.86.18	80	32	8525	14	4895	18	3630
24.6.173.220	61652	209.177.86.18	80	33	8482	14	4780	19	3702
24.6.173.220	61665	209.177.86.18	80	31	7783	13	4410	18	3373
24.6.173.220	61655	209.177.86.18	80	30	7648	13	4343	17	3305
24.6.173.220	61640	123.125.115.126	80	15	7384	6	634	9	6750
24.6.173.220	61666	209.177.86.18	80	28	6826	12	3873	16	2953
24.6.173.220	61601	210.72.21.11	80	18	6685	9	1143	9	5542
24.6.173.220	61612	202.96.25.95	80	12	3518	6	672	6	2846
24.6.173.220	61663	202.96.25.95	80	12	3506	6	724	6	2782
24.6.173.220	61611	210.72.21.42	80	12	3187	6	694	6	2493
24.6.173.220	61661	210.72.21.87	80	12	2522	6	977	6	1545
24.6.173.220	61623	209.177.86.18	80	12	2444	6	682	6	1762
24.6.173.220	61610	210.72.21.87	80	12	2212	6	667	6	1545
24.6.173.220	61614	209.177.86.18	80	12	2050	6	682	6	1368
24.6.173.220	61598	173.194.79.121	80	10	2024	6	658	4	1366
24.6.173.220	61650	209.177.86.18	80	11	1643	6	1060	5	583
24.6.173.220	61657	123.125.115.126	80	10	1381	5	806	5	575
24.6.173.220	61682	123.125.115.126	80	10	1343	5	768	5	575
24.6.173.220	61639	123.125.115.126	80	10	1330	5	755	5	575
24.6.173.220	61656	210.72.21.11	80	11	1292	6	706	5	586

7. based on the byte count here, the most active connection is between `24.6.173.220:61619` to `209.177.86.18:80` which has a byte count of `100k`
8. Do not see `107.6.133.250` using port `80` and `25.6.181.160` using port `1266` on wireshark. But rather host `24.6.173.220` using a incremental port number to sending network packets to various IP addresses and also communicating with those process through their IP address and port `80`
9. If we right click on the most active TCP conversation and select apply as a filter → select → $A \leftrightarrow B$ and then look back into wireshark, I obtained this “Displayed” `103` which I would assume there’s `103` for the $A \leftrightarrow B$ option, If we swtich to $A \rightarrow B$ only, the option dropped to `32`


http-misctraffic101.pcapng
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
ip.addr==24.6.173.220 && tcp.port==61619 && ip.addr==209.177.86.18 && tcp.port==80
Expression...
No. Time Source Destination Protocol Length Info
352 11.393133 24.6.173.220 209.177.86.18 TCP 66 61619 → 80 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=4 SACK_PERM=1
446 11.490541 209.177.86.18 24.6.173.220 TCP 66 80 → 61619 [SYN, ACK] Seq=0 Ack=1 Win=4380 Len=0 MSS=1460 SACK_PERM=1 WS=128
447 11.490620 24.6.173.220 209.177.86.18 TCP 54 61619 → 80 [ACK] Seq=1 Ack=1 Win=65700 Len=0
448 11.490991 24.6.173.220 209.177.86.18 HTTP 377 GET /jquery-1.7.min.js HTTP/1.1
573 11.606989 209.177.86.18 24.6.173.220 TCP 60 80 → 61619 [ACK] Seq=1 Ack=324 Win=5504 Len=0
574 11.606991 209.177.86.18 24.6.173.220 TCP 414 [TCP segment of a reassembled PDU]
575 11.606992 209.177.86.18 24.6.173.220 TCP 1514 [TCP segment of a reassembled PDU]
576 11.607397 24.6.173.220 209.177.86.18 TCP 54 61619 → 80 [ACK] Seq=324 Ack=1821 Win=65700 Len=0
577 11.608175 209.177.86.18 24.6.173.220 TCP 1514 [TCP segment of a reassembled PDU]
578 11.608182 209.177.86.18 24.6.173.220 TCP 1514 [TCP segment of a reassembled PDU]
579 11.608184 209.177.86.18 24.6.173.220 TCP 1514 [TCP segment of a reassembled PDU]
580 11.608603 24.6.173.220 209.177.86.18 TCP 54 61619 → 80 [ACK] Seq=324 Ack=6201 Win=65700 Len=0
581 11.609408 209.177.86.18 24.6.173.220 TCP 1514 [TCP segment of a reassembled PDU]
626 11.708662 209.177.86.18 24.6.173.220 TCP 1514 [TCP segment of a reassembled PDU]
627 11.709075 24.6.173.220 209.177.86.18 TCP 54 61619 → 80 [ACK] Seq=324 Ack=9121 Win=65700 Len=0
628 11.709833 209.177.86.18 24.6.173.220 TCP 1514 [TCP segment of a reassembled PDU]
629 11.710732 209.177.86.18 24.6.173.220 TCP 1514 [TCP segment of a reassembled PDU]
630 11.710736 209.177.86.18 24.6.173.220 TCP 1514 [TCP segment of a reassembled PDU]
631 11.711102 24.6.173.220 209.177.86.18 TCP 54 61619 → 80 [ACK] Seq=324 Ack=13501 Win=65700 Len=0
632 11.711868 209.177.86.18 24.6.173.220 TCP 1514 [TCP segment of a reassembled PDU]
633 11.711871 209.177.86.18 24.6.173.220 TCP 1514 [TCP segment of a reassembled PDU]
634 11.712324 24.6.173.220 209.177.86.18 TCP 54 61619 → 80 [ACK] Seq=324 Ack=16421 Win=65700 Len=0
679 11.808675 209.177.86.18 24.6.173.220 TCP 1514 [TCP segment of a reassembled PDU]
680 11.809740 209.177.86.18 24.6.173.220 TCP 1514 [TCP segment of a reassembled PDU]
681 11.809748 209.177.86.18 24.6.173.220 TCP 1514 [TCP segment of a reassembled PDU]
682 11.809750 209.177.86.18 24.6.173.220 TCP 1514 [TCP segment of a reassembled PDU]
683 11.810569 24.6.173.220 209.177.86.18 TCP 54 61619 → 80 [ACK] Seq=324 Ack=22261 Win=65700 Len=0
684 11.811320 209.177.86.18 24.6.173.220 TCP 1514 [TCP segment of a reassembled PDU]
Frame 631: 54 bytes on wire (432 bits), 54 bytes captured (432 bits) on interface 0
Ethernet II, Src: HewlettP_a7:bf:a3 (d4:85:64:a7:bf:a3), Dst: Cadant_31:bb:c1 (00:01:5c:31:bb:c1)
Internet Protocol Version 4, Src: 24.6.173.220, Dst: 209.177.86.18
Transmission Control Protocol, Src Port: 61619 (61619), Dst Port: 80 (80), Seq: 324, Ack: 13501, Len: 0
0000 00 01 5c 31 bb c1 d4 85 64 a7 bf a3 08 00 45 00 ..\1.... d.....E.
0010 00 28 1c 0f 40 00 80 06 00 00 18 06 ad dc d1 b1 .(..@... ....
0020 56 12 f0 b3 00 50 9b c6 8c 0f b7 08 c8 fe 50 10 V....P.. ......P.
0030 40 29 ed c0 00 00 @)....
http-misctraffic101
Packets: 1668 · Displayed: 103 (6.2%) · Load time: 0:0.52
Profile: Default



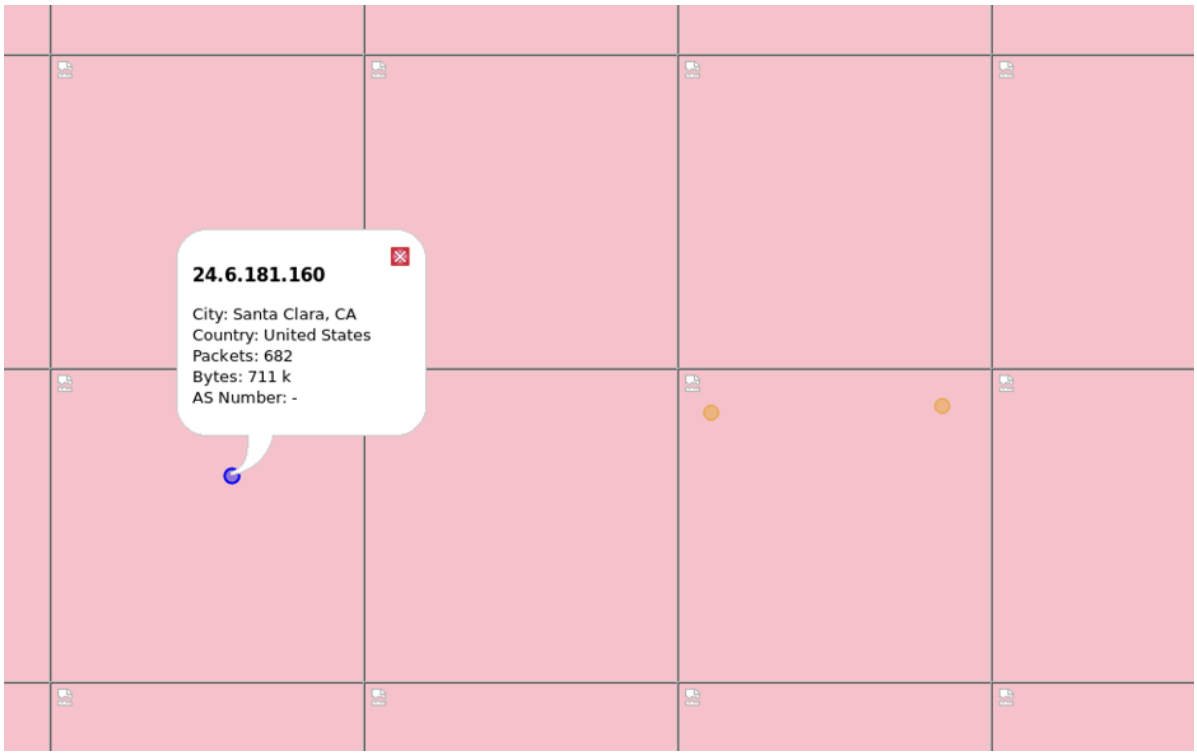
10. Part 1 cleaned up

Task 3.2: Geolocating IP addresses

1. Opened `pcaps/http-browse101c.pcapng` in wireshark
2. Clicked on `Edit`, then `Preference`, `Name`, Then clicked on the `+` icon and added the path that points to `maxmind` directory, then click okay on all window until I exit to wireshark
3. Selected statistics - Endpoints and clicked on the IPv4 tab and then saw information regards to country, city, latitude and longitude columns

Country	City	AS Number	Latitude	Longitude
United States	Santa Clara, CA	AS7922 Comcast Cable Communications, LLC	37.350101	-121.985397
United States	Chicago, IL	AS32475 SingleHop LLC	41.877602	-87.627197
United States	Boston, MA	AS27552 TowardEX Technologies International, Inc.	42.358398	-71.059799

4. Clicked on the Map button, the map is just broken but I can see 3 points correspond to those cities

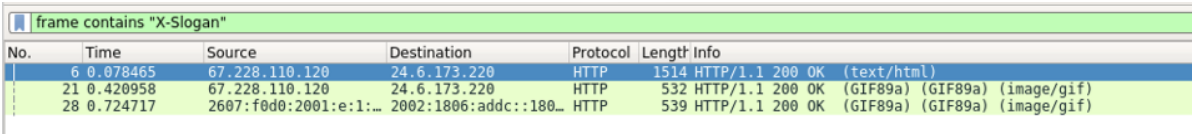


5. There's no traffic going to Milpitas, CA or I should say Milpitas was not the CA location I've got. Santa Clara is close enough to Milpitas but it is not Milpitas. So either the lab instruction is wrong or I'm missing something. If it is Santa Clara then its IP address is `24.6.181.160` which if we do a filtering on wireshark with `ip.src==24.6.181.160 || ip.dst==24.6.181.160`, we noticed all the entires return. Meaning 100% of the tracked traffics is either going to or going from Santa Clara.
6. Part 2 cleaned up

Task 3.3: Reassemble text from TCP stream

1. Opened `pcaps/http-wiresharkdownload101.pcapng` in wireshark
2. Saw the first 3 frame (packet) is a TCP handshake
3. Right clicked on frame 4 and select follow - TCP stream and got this

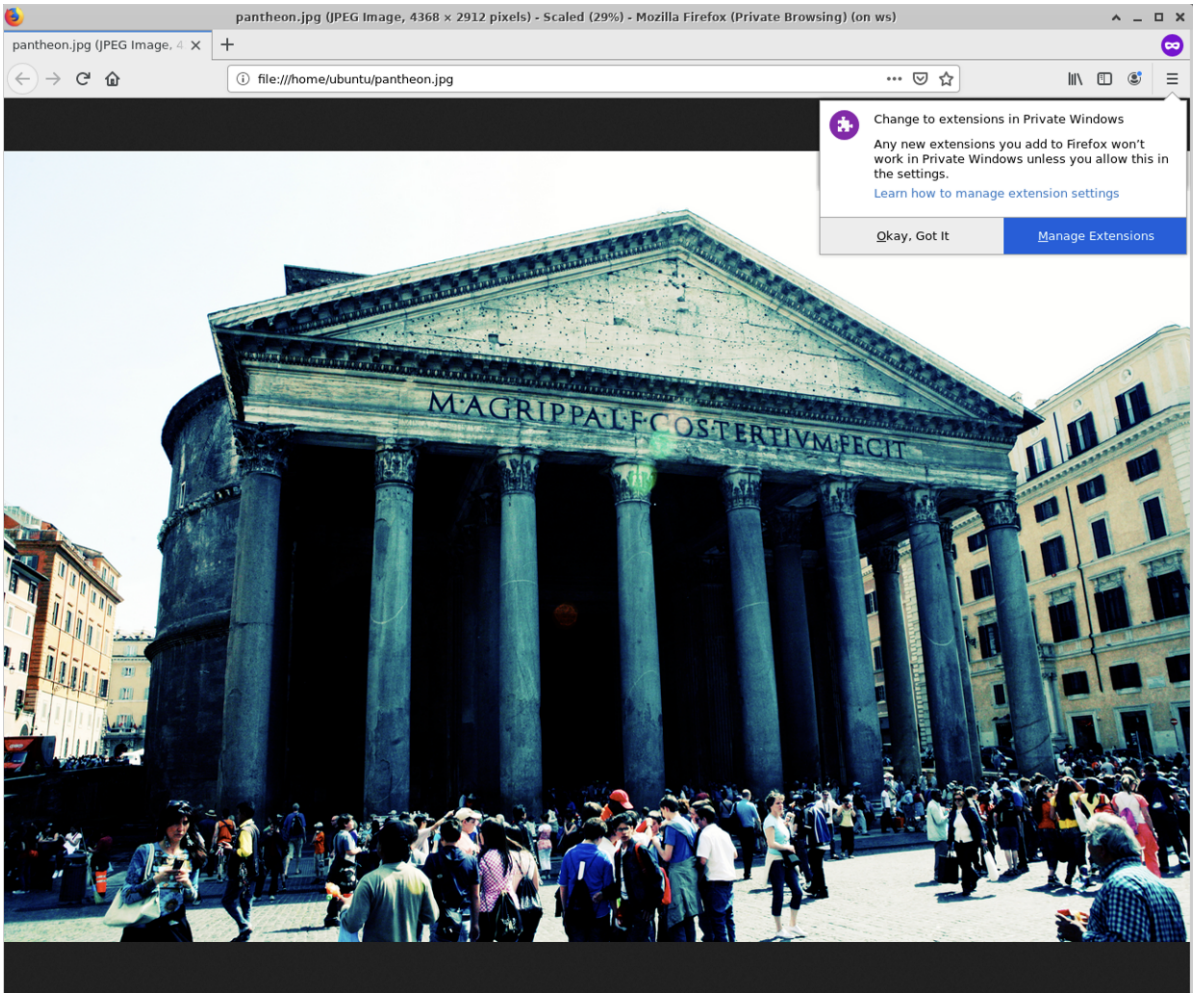
4. The X-Slogan is `Sniffing the glue that holds the Internet together.`
5. To find more X-Slogan, we can apply a filter of `frame contains "X-Slogan"` in Wireshark



6. The X-Slogan for the 2nd option here is the same X-Slogan as the first `Sniffing the glue that holds the Internet together.`
7. The X-Slogan for the 3rd option here is different, it is `Sniff free or die.`
8. Part 3 cleaned up

Task 3.4: Extract binary file from FTP session

1. Opened `pcaps/ftp-clientside101.pcapng` in wireshark
2. Followed the stream on the first frame, pressed the `Hide this stream` which removed the command channel stream
3. The file I see is `pantheon.jpg`
4. Pressed the `Hide this stream` on the directory list stream
5. followed the file transfer stream, saw some sort of meta data of an image file which I saved the data (using the RAW) option to a jpeg
6. This is the jpeg (saved as `pantheon.jpg` at `/home/ubuntu` and opened with `xdg-open pantheon.jpg`)



Task 3: PCAP Programming

Task 3.1: Unknown trace

```
import dpkt

with open("trace2.pcap", "rb") as f:
    pcap_reader = dpkt.pcap.Reader(f)
    snaplen = pcap_reader.snaplen
    linktype = pcap_reader.datalink()
```

```
print("Link-layer type: " + str(linktype))
print("Snap length: " + str(snaplen) + " bytes")
```

1. From the above script and ran it on the trace2.pcap, the Link-layer type is `1` which is ethernet connection
2. From the above script, the snaplen is 65535 bytes, this is the maximum amount of bytes captured per packet in the pcap file. This help us skipping large payloads and only kepy the relevant parts of the packet during capture
3. PcapNG is an improved verison of the pcap file, it is more flexible which supports multiple interfaces and encapsulation types in a single file, it also allows us to store richer metadata. However, there might be times where pcap must be used which we can change PcapNG to pcap with `dumpcap -P` command

Task 3.2: Basic traffic stats

```
import dpkt

with open("trace2.pcap", "rb") as f:
    pcap_reader = dpkt.pcap.Reader(f)
    first_timestamp = None
    last_timestamp = None
    ipv4_count = 0
    non_ipv4_count = 0
    total_packets = 0

    for timestamp, buf in pcap_reader:
        total_packets += 1

        if first_timestamp is None:
            first_timestamp = timestamp

        last_timestamp = timestamp

        eth = dpkt.ethernet.Ethernet(buf)
        if eth.type == dpkt.ethernet.ETH_TYPE_IP:
```

```
            ip = eth.data
            if ip.v == 4:
                ipv4_count += 1
            else:
                non_ipv4_count += 1
        else:
            non_ipv4_count += 1

        # print(timestamp, total_packets, ipv4_count, non_ipv4_c

capture_duration = (
    last_timestamp - first_timestamp if last_timestamp and first
)
avg_packet_rate = float(total_packets) / capture_duration if cap

print("Count IPv4: " + str(ipv4_count))
print("count Non-IPv4: " + str(non_ipv4_count))
print("First timestamp: %.2f" % first_timestamp)
print("Average packet rate: %.2f packets/second" % avg_packet_ra
```

1. From the result of the script (which ran for like 15 mins), there's `30611000` IPv4 packet (which is all the packets)
2. From the script, there's `0` non-IPv4 packet
3. From the script, the first timestamp is `1474265898.92`
4. From the script, (where the average packet rate is calculate by the total amount of packet divide by the last_timestamp subtracting the first timestamp) is `708.60 packets/second`

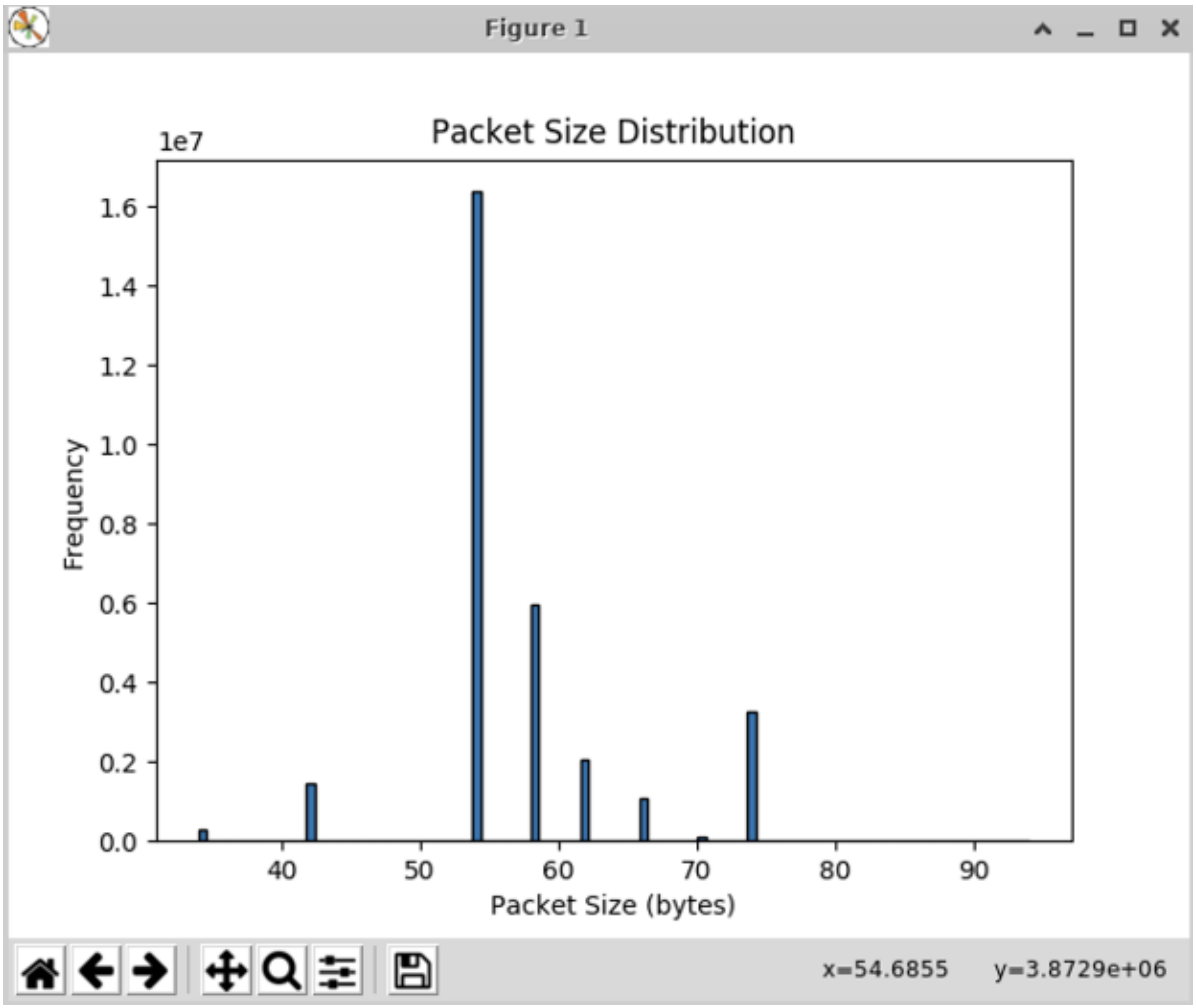
```
import dpkt
import socket
import numpy as np
import matplotlib.pyplot as plt

with open("trace2.pcap", "rb") as f:
```

```
    pcap_reader = dpkt.pcap.Reader(f)

    ipv4_count = 0  # 1
    non_ipv4_count = 0  # 2
    first_timestamp = None  # 3, 4
    last_timestamp = None  # 4
    total_packets = 0  # 4

    protocols = {}  # 5
    sizes = []  # 6
    sources = set()  # 7
    destinations = set()  # 8
    total_bytes = 0  # 9
    sources_byte = {}  # 9

    max_byte_source = 0  # 10
    source_mb = ""  # 10

    source_packet = {}  # 11
    max_packets_source = 0  # 11
    source_mp = ""  # 11

    for timestamp, buf in pcap_reader:
        total_packets += 1

        if first_timestamp is None:
            first_timestamp = timestamp

        last_timestamp = timestamp

        eth = dpkt.ethernet.Ethernet(buf)

        protocols[eth.type] = protocols.get(eth.type, 0) + 1

        sizes.append(len(buf))
```

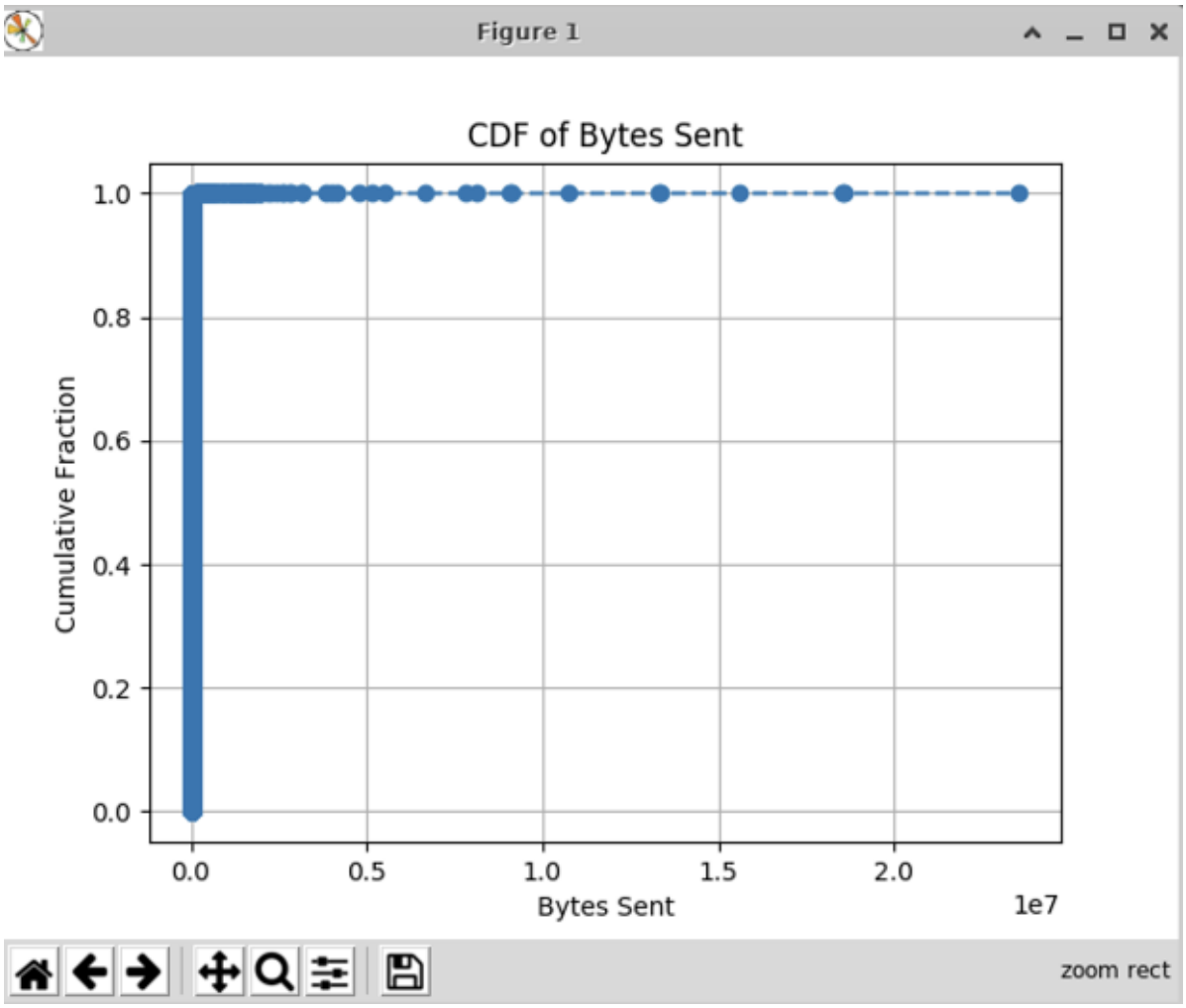
```
        if eth.type == dpkt.ethernet.ETH_TYPE_IP:
            ip = eth.data
            if ip.v == 4:
                ipv4_count += 1
                sources.add(ip.src)
                sources_byte[ip.src] = sources_byte.get(ip.src,
                source_packet[ip.src] = source_packet.get(ip.src
                total_bytes += len(buf)
                destinations.add(ip.dst)

                if sources_byte[ip.src] > max_byte_source:
                    source_mb = ip.src
                    max_byte_source = sources_byte[ip.src]

                if source_packet[ip.src] > max_packets_source:
                    source_mp = ip.src
                    max_packets_source = source_packet[ip.src]
            else:
                non_ipv4_count += 1
        else:
            non_ipv4_count += 1

capture_duration = (
    last_timestamp - first_timestamp if last_timestamp and first
)
avg_packet_rate = float(total_packets) / capture_duration
    if capture_duration > 0 else 0

print("Count IPv4: " + str(ipv4_count))
print("Count Non-IPv4: " + str(non_ipv4_count))
print("First timestamp: %.2f" % first_timestamp)
print("Avg packet rate: %.2f packets/second" % avg_packet_rate)
print("packet protocol distribution: " + str(protocols))

plt.hist(sizes, bins=100, edgecolor="black")
plt.xlabel("Packet Size (bytes)")
```

```
plt.ylabel("Frequency")
plt.title("Packet Size Distribution")
plt.show()

print("Unique sources: " + str(len(sources)))
print("Unique destinations: " + str(len(destinations)))

sorted_bytes = sorted(sources_byte.values())
cdf = np.arange(1, len(sorted_bytes) + 1) / float(len(sorted_byt

plt.plot(sorted_bytes, cdf, marker="o", linestyle="--")
plt.xlabel("Bytes Sent")
plt.ylabel("Cumulative Fraction")
plt.title("CDF of Bytes Sent")
plt.grid(True)
plt.show()

print("Source sending most bytes: " + socket.inet_ntoa(source_mb
print("Source sending most packets: " + socket.inet_ntoa(source_
```

5. All of them used the IPv4 Protocol, `packet protocol distribution: {2048: 30611000}`, the `2048` == `x0800` which stands for IPv4 Protocol
6. The histogram (side note, to make mathplot works, need to `sudo apt-get install python-pk` which was not mention in the document)



7. Unique sources: 1018015
8. Unique destinations: 32771
9. Below is the CDF



10. Source sending most bytes: 58.51.150.96
11. Source sending most packets: 58.51.150.96
12. All of the packets is using the IPv4 protocol, then many of the packets are some specific size (54-56 range is rather high) follow by (58 bytes I believe). Finally, all the packets does not contain a payload as seen by the Bytes sent CDF. They are almost all 0 bytes with other with only a little bit of payload. There's also a lot more unique sources then unique destinations
13. I believe that this represent activity from a network infected with some sort of pinging activity or potential DDoS attack. Given that there's high volumn of IPv4 traffic some large list of unique sources with small set of unique destinations with also how the packets contain no payload other then the header. But further investigation would be required to fully understand what's going on.

- Special Note, did check work and noticed ipv4_count and src_pckts not little up but clearly I've output those. It would be unfair to trust the autograde when I've physically output those and shown in code above that it would output those.

Task 4: Netflow

Task 3.1: Understanding IPFIX

1. Under what conditions is a flow expired, i.e. when are the statistics for a particular flow exported as an IPFIX record?

   A flow is inactive when no packets belonging to the Flow have been observed at the Observation Point for a given timeout. Which then become expired when

   1. If the Exporter detected the end of a Flow
   2. If the Flow has been inactive for a certain period of time (flow time out setting).
   3. If the exporter experiences internal constraints, a Flow MAY be forced to expire prematurely

   (Claise)

2. What transport protocol does IPFIX use to export records to remote IPFIX collection servers?

   The netflow export packets are encapsulated into UDP, but can also operate over congestion-aware protocols like SCTP

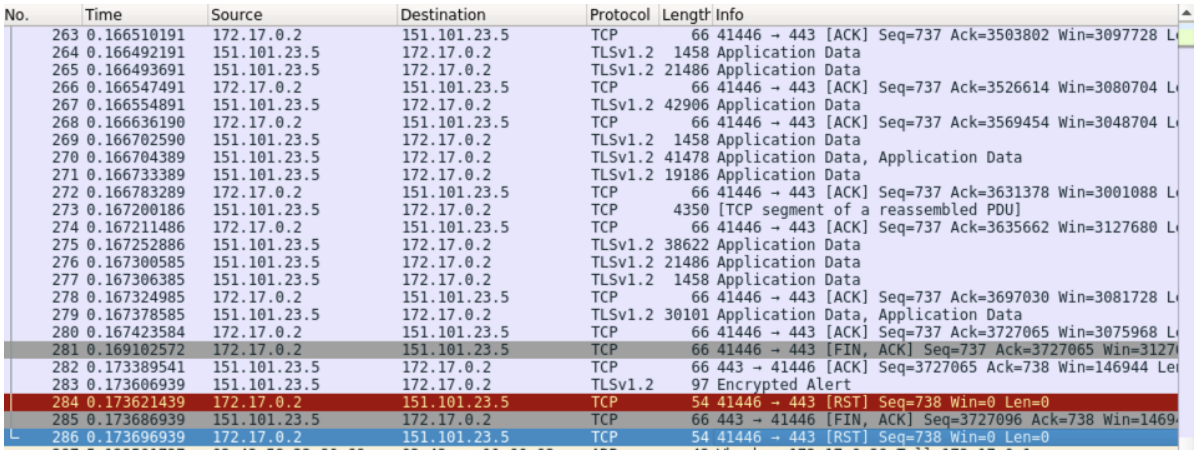
   (Claise)

3. Do IPFIX records represent bidirectional or unidirectional captured traffic? Explain your answer.

   IPFIX records represent unidirectional traffic, as a flow is defined as a unidirectional sequence of packet, which a record of a flow would also be unidirectional with a single direction of traffic, defined by its source IP, destination IP, protocol and etc

4. Visit a popular web site, e.g. http://www.cnn.com/ and capture the packets as a pcap file (using Wireshark or tcpdump). How many flows are generated as part of your session with the web site?

   Since browser is not install on this docker container, I used `wget www.cnn.com` instead



No.	Time	Source	Destination	Protocol	Length	Info
263	0.166510191	172.17.0.2	151.101.23.5	TCP	66	41446 → 443 [ACK] Seq=737 Ack=3503802 Win=3097728 L
264	0.166492191	151.101.23.5	172.17.0.2	TLSv1.2	1458	Application Data
265	0.166493691	151.101.23.5	172.17.0.2	TLSv1.2	21486	Application Data
266	0.166547491	172.17.0.2	151.101.23.5	TCP	66	41446 → 443 [ACK] Seq=737 Ack=3526614 Win=3080704 L
267	0.166554891	151.101.23.5	172.17.0.2	TLSv1.2	42906	Application Data
268	0.166636190	172.17.0.2	151.101.23.5	TCP	66	41446 → 443 [ACK] Seq=737 Ack=3569454 Win=3048704 L
269	0.166702590	151.101.23.5	172.17.0.2	TLSv1.2	1458	Application Data
270	0.166704389	151.101.23.5	172.17.0.2	TLSv1.2	41478	Application Data, Application Data
271	0.166733389	151.101.23.5	172.17.0.2	TLSv1.2	19186	Application Data
272	0.166783289	172.17.0.2	151.101.23.5	TCP	66	41446 → 443 [ACK] Seq=737 Ack=3631378 Win=3001088 L
273	0.167200186	151.101.23.5	172.17.0.2	TCP	4350	[TCP segment of a reassembled PDU]
274	0.167211486	172.17.0.2	151.101.23.5	TCP	66	41446 → 443 [ACK] Seq=737 Ack=3635662 Win=3127680 L
275	0.167252886	151.101.23.5	172.17.0.2	TLSv1.2	38622	Application Data
276	0.167300585	151.101.23.5	172.17.0.2	TLSv1.2	21486	Application Data
277	0.167306385	151.101.23.5	172.17.0.2	TLSv1.2	1458	Application Data
278	0.167324985	172.17.0.2	151.101.23.5	TCP	66	41446 → 443 [ACK] Seq=737 Ack=3697030 Win=3081728 L
279	0.167378585	151.101.23.5	172.17.0.2	TLSv1.2	30101	Application Data, Application Data
280	0.167423584	172.17.0.2	151.101.23.5	TCP	66	41446 → 443 [ACK] Seq=737 Ack=3727065 Win=3075968 L
281	0.169102572	172.17.0.2	151.101.23.5	TCP	66	41446 → 443 [FIN, ACK] Seq=737 Ack=3727065 Win=3127
282	0.173389541	151.101.23.5	172.17.0.2	TCP	66	443 → 41446 [ACK] Seq=3727065 Ack=738 Win=146944 Le
283	0.173606939	151.101.23.5	172.17.0.2	TLSv1.2	97	Encrypted Alert
284	0.173621439	172.17.0.2	151.101.23.5	TCP	54	41446 → 443 [RST] Seq=738 Win=0 Len=0
285	0.173686939	151.101.23.5	172.17.0.2	TCP	66	443 → 41446 [FIN, ACK] Seq=3727096 Ack=738 Win=1469
286	0.173696939	172.17.0.2	151.101.23.5	TCP	54	41446 → 443 [RST] Seq=738 Win=0 Len=0

   - From wireshark, it generated 286 flows - 2 flows (which was some other activity not related to this)

5. Provide an example of an attack that cannot be detected with IPFIX records, but can be detected by capturing packets (pcap)

   IPFIX only capture high level information like source/destination IP addresses, and other high level informations, but not the payload of the packets directly, If the malicious thing is being transported in the payload. The pcap will be easily identify which packet is the malicious one while IPFIX cannot.

Consider a hypothetical 1Gbps Ethernet link on which you would like to monitor traffic. Assume that the link is 75

6. What is the minimum size of a one-hour pcap capture on the link? State any assumptions necessary.

   Given that the link utilization is 75%, → 1Gbps → 750 Mbps

   1 hour has 60 * 60 seconds which is 3600

   Data = $750Mbps * 3600s = 2700000Mb$

   $2700000Mb * \frac{1MB}{8Mb} = 337500MB = 337.5GB$

Thus, the minimum size is approximately $337.5GB$

7. Assume that there are an average of 50 packets per flow. Assume that each IPFIX flow record is a fixed 64 bytes. What is the minimum size of a one-hour IPFIX capture on the same link? State any assumptions necessary.

   An assumption we would have to make it the size of 1 singular package on average on the internet, for simplicity sake, we will state that it is 750 bytes here (in reality, it is strong mode on 40 bytes, 1300 bytes and 1500 bytes but it is not easy to draw a conclusion).

   From last question we know that an hour can generate $337.5GB$ data in an hour

   A flow is $750B * 50 = 37.5KB$

   Amount of flow $= 337.5GB/37.5KB = 9000000$

   Since each flow contains a 64 bytes fixed record, then $9000000 * 64 = 576000000B = 576MB$

   So, there will be $\approx 576MB$ of IPFIX flow records being recorded.

Even with flow aggregation, the number of records may be very large. In addition, the router hardware may not be able to maintain state over all flows or packets transiting the device. As a result, many routers implement flow sampling. If a router is running 1:1000 sampling, then only every 1000th packet is considered for IPFIX processing.

8. Provide an example of a traffic monitoring task that cannot be successfully completed with sampled IPFIX. Explain your answer.

   Given it is only sampling 1 in 1000 packets, any important packets contain malicious code can be done in small amount of packets will be missed by this algorithm. Or Denial of service attack but with only 200 / 1000 packets in each rotation. Which this sampling method maybe difficult to pick up on

9. Provide an example of a traffic monitoring task that can be successfully completed with sam- pled IPFIX. Explain your answer.

It will succeed in monitoring the large-scale traffic trends, such as which application use the most bandwidth in general and able to identify performance issues of the network at the moment. Task that does not require looking into every packet in general basically. Since this will gives you a trend and a more effective strategy if your network is large which it is infeasible to analysis every single flow / packet which will have additional performance penalty on your system.

10. No a question so skipped, but ideally you want to be randomly sampling the packets in a specific range of packets.
11. Used command `rwaddrcount --print-stat trace2.silk` which showed `1010327` as the amount of `sIP_Uniq`
12. Used command `rwaddrcount --use-dest --print-stat trace2.silk` which showed `32771` as the amount of `dIP_Uniq`
13. The previous lab result was Unique sources: `1018015` and Unique destinations: `32771`, So there's a slight difference for source IP address but no difference in destination IP address. The difference may've been arise from the different data representation which might've cause the slight difference in amount of unique source IP address but it is still relatively closed.
14. Used command `rwstats --fields=sip --values=packets --count=10 trace2.silk` which showed this

sIP	Packets	%Packets	cumul_%
58.51.150.96	380111	1.252492	1.252492
115.231.159.14	299677	0.987456	2.239948
218.12.231.35	298382	0.983189	3.223137
60.191.186.135	251190	0.827688	4.050825
222.163.201.165	214618	0.707181	4.758006
1.179.247.113	180243	0.593913	5.351919
122.228.91.55	173561	0.571895	5.923815
85.232.241.219	168928	0.556629	6.480444
183.131.170.61	146317	0.482125	6.962569
185.93.185.235	144709	0.476826	7.439395

    - which on the last column, showed the CDF which the top 10 sIP took up around 7.44% of all the packets count
15. Used command `rwstats --fields=dport --values=packets --count=5 trace2.silk`

dPort	Packets	%Packets	cumul_%
23	20836256	68.656890	68.656890
2323	2130000	7.018496	75.675386
22	491828	1.620607	77.295993
3389	460034	1.515844	78.811837
53413	440112	1.450199	80.262036

16. Used command `rwstats --fields=sport --values=packets --count=5 trace2.silk`

sPort	Packets	%Packets	cumul_%
80	1916936	6.316435	6.316435
8370	251435	0.828496	7.144931
6000	208736	0.687799	7.832730
49722	169384	0.558132	8.390862
44509	145065	0.477999	8.868861

17. The file of `trace2.silk` has a size around 1.1GB by using the command `ls -lh`, which the `trace2.pcap` has a size around 2.1GB by using the same command. This make sense as `pcap` records the entire packet while `IPFIX` file only records the header of some file and dropping the payloads.
18. `rwaddrcount --print-stat peering2.silk` showed that there's `57872` unique source IP address (command same as Q11)
19. `rwaddrcount --use-dest --print-stat peering2.silk` showed that there's `423224` unique destination IP address (command same as Q12)
20. used command `rwstats --overall-stats peering2.silk`

    Min: 28 Bytes

    Median: 270.09612 Bytes

    Max: 7626300 Bytes
21. used the same command as previous

    Min: 1 packet

    Median: 2.40847 packets

    Max: 60488 packets
22. To find the specific protocol and their byte count, I used the command `rwuniq --fields=proto --values=bytes peering2.silk`

pro	Bytes
103	22632
50	75370970
6	5894576189
51	23084
94	2259335
41	714802
54	204
17	269982826
89	96672
47	24396317

- which TCP is protocol `6` and therefore contributed `5894576189` bytes

Then used another command `rwaddrcount --print-stat peering2.silk` which includes the total bytes as one of the output field

	sIP_Uniq	Bytes	Packets	Records
Total	57872	6267443031	30074407	3197786

So $\frac{5894576189}{6267443031} = 0.940507342443 = 94\%$ of all the bytes

23. Using the command `rwfilter --dcidr=18.128.0.0/9 --pass=stdout peering2.silk | rwaddrcount --print-stat`, I've obtain this result

	sIP_Uniq	Bytes	Packets	Records
Total	49857	2548325217	13857959	1342722

This indicated `2548325217` bytes of data is currently going to AT&T ISP (given this is a HJN to AT&T silk file) then to the RPN network, this is ~41% of all data in bytes being transfer in the singular hour as $\frac{2548325217}{6267443031} = 0.406597268519 = 41\%$

24. If there's cost of using AT&T for each money is being exchanged, we can reduce our cost charges / upgrades required for the AT&T connection given that once we peered with RPN, we can offload 40% of traffic for free and improve latency on 40% of our network traffics. This also frees up space on the AT&T side which allow us to handle more data. Hence a Peering with RPN should be beneficial.

Appendix

Claise, Benoît. “RFC 3954: Cisco Systems NetFlow Services Export Version 9.” *IETF Datatracker*, datatracker.ietf.org/doc/html/rfc3954.