A Level Computer Science Non-Examined Assessment (NEA)

Sam Leonard

Contents

1 Analysis

1.1 Identification and Background to the Problem

The problem I am trying to solve with my project is how to look at devices on a network from a “black box” perspective and gain information about what services are running etc. Services are programs which their entire purpose is to provide a *service* to other programs, for example a server hosting a website would be running a service whose purpose is to send the webpage to people who try to connect to the website.
There are many steps in-between a device turning on to interacting with the internet.

1. load networking drivers
2. Starting Dynamic Host Configuration Protocol (DHCP) daemon
3. Broadcasting DHCP request for an IP address
4. Get assigned an IP address

There are many more steps than I have listed above but these are the most important ones. Starting from a linux computer being switched on the first step is that the kernel needs to load the networking drivers. The kernel is the basis for the operating system, it is what interacts with the hardware in the most fundamental way. drivers are small bits of code which the kernel can load in order to interact with certain hardware modules such as the Network Interface Card (NIC) which is essential for interfacing with the network, hence the name.

Next once the kernel has loaded the required drivers and the system has booted the networking ‘daemons’ must be started. In linux a daemon is a program that runs all the time in the background to serve a specific purpose or utility. For example when I start my laptop the following daemons start upowerd (power management), systemd (manages the creation of all processes), dbus-daemon (manages inter-process communication), iwd (manages my WiFi connections) and finally Dynamic Host Configuration Protocol Client Daemon (DHCPCD) which manages all interactions with the network around DHCP.

Once the daemons are all started the DHCP client can now take issue commands to the daemon for it to carry out. The DHCP client is simply a daemon that runs in the background to carry out any interactions between the current machine and the DHCP server. The DHCP server is normally the WiFi router or network switch for the local network and it manages a list of which computer has which IP address and negotiates with new computers trying to join a network to get them a free IP address. The DHCP client starts the DHCP address negotiation with the server by sending a discover message with the address 255.255.255.255 which is the IP limited broadcast address which means that whatever is listening at the other end will forward this packet on to everyone on the subnet. When the DHCP server (normally the router, sometimes a separate machine) on the subnet receives this message it reserves a free IP address for that client and then responds with a DHCP offer which contains the address the server is offering, the length of time the address is valid for and the subnet mask of the network. The client must then respond with a DHCP request message to request the offered address, this is in case of multiple DHCP servers offering addresses. Finally the DHCP server responds with a DHCP acknowledge message showing that it has received the request. Figure 2 shows a packet capture from my laptop where I turned WiFi off, started wireshark listening and plugged in an Ethernet cable, I have it showing only the DHCP packets so that it is clear to see the entire DHCP negotiation including the 255.255.255.255 limited broadcast destination address and the 0.0.0.0 unassigned address in the source column. I mention using wireshark to do packet capturing above without explaining what either packet capturing or wireshark are so I will do that here. Packets I define below and wireshark is simply a tool which intercepts all the network communications on a single computer and records them to a file as well as displaying them to the user as well as performing some analysis and dissecting each of the protocols used. This means that I can record the DHCP negotiation shown below and show it to you using wireshark to get all the information out of the packets being sent over the wire.

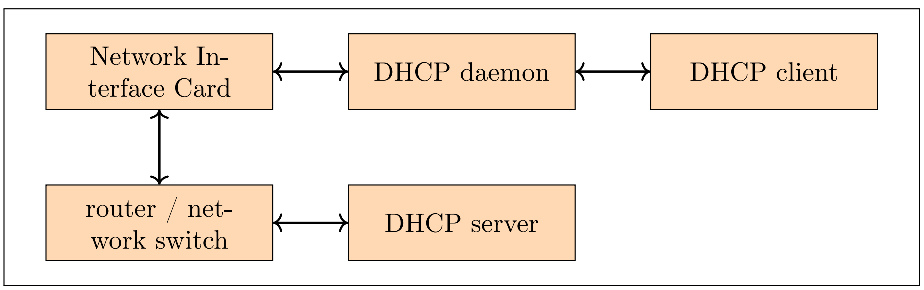


Figure 1: A block diagram showing the relationship between different elements of a DHCP negotiation.

No.	Time	Source	Destination	Protocol	Info
6	0.983737378	0.0.0.0	255.255.255.255	DHCP	DHCP Discover
32	4.239092378	192.168.1.1	192.168.1.47	DHCP	DHCP Offer
34	4.239420587	0.0.0.0	255.255.255.255	DHCP	DHCP Request
36	4.241743101	192.168.1.1	192.168.1.47	DHCP	DHCP ACK

Figure 2: DHCP address negotiation

All computer networking is encapsulated in the Open Systems Interconnection model (OSI model) which has 7 layers:

7. Application: Applications Programming Interface (API)s, Hypertext transfer Protocol (HTTP), File Transfer Protocol (FTP) among others.

6. Presentation: encryption/decryption, encoding/decoding, decompression etc...

5. Session: Managing sessions, PHP Hypertext Processor (PHP) session IDs etc...

4. Transport: TCP and UDP among others.

3. Network: ICMP and IP among others.

2. Data Link: MAC addressing, Ethernet protocol etc...

1. Physical: The physical Ethernet cabling/NIC.

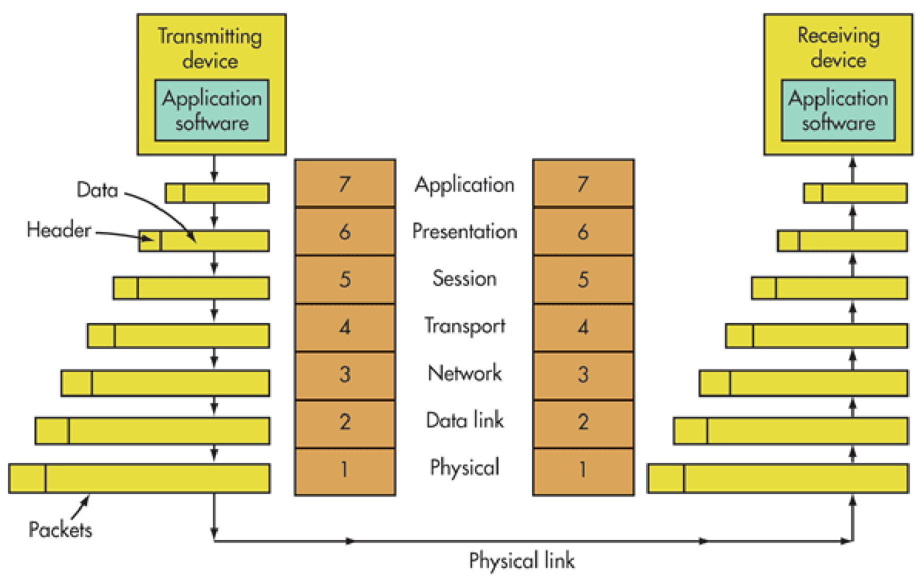


Figure 3: OSI model diagram, source: https://www.electronicdesign.com

Each of these layers is essential to the running of the internet but a single communication might not include all of the layers. These communications are all based on the most fundamental part of the internet: the packet. Packets are sequences of ones and zeros sent between computers which are used to transfer data as well as to control how networks function. They consist of different layers of information each specifying where the packet where should go next at a different level along with fundamentally the data/instructions contained in the innermost layer. When packets are sent between computers a certain number of layers are stripped off by each computer so that it knows where to send the packet next at which point it will add all the layers back again, this time with the instructions needed to go from the current computer to the next one on its route. Each of these layers actually consists of a number of fields at the start called a header some layers also append a footer to the end of the packet. The actual data being transferred in the packet can be quite literally anything, HTTP transfers websites so Hypertext Markup Language (HTML) files and images etc.... In particular there are two pieces of information stored in headers which together define the final destination of the packet: the IP address and the port number. The IP address defines the destination machine and the port number defines which "port" on the remote machine the packet should be sent to. Ports are essential entrances to a computer, for example if a computer was a hotel the IP address would be the address and location of the hotel and the port number would be the room inside the hotel. There are 65535 ports and 0 is a special reserved port. Both Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) use ports, TCP ports are mainly used for transferring data where reliability is a concern, as TCP has built in checks for packet loss whereas UDP does not and as such is used for purposes where speed is more important and missing some data is inconsequential, such as video streaming and playing games.

I'm going to use the example of getting a very simple static HTML page with an image inside. The code for the page is shown in listing 1. In figure 4 you can see how the page renders. However far more interestingly is how the browser retrieved the page, in figure 5 you can see the full sequence of packets that were exchanged for the browser to get the resources it needed to render the page. I am hosting the page using Python3's http.server module which is super convenient and just makes the current directory open on port 8000 from there I can just navigate to /example.html and it will render the page. Breaking figure 5 down packet one shows the browser receiving the request from the user to display `http://192.168.1.47:8000/example.html` and attempting to connect to 192.168.1.47 on port 8000. Packets two and three show the negotiation of this request through to the full connection being made. The browser now makes an HTTP GET request for the page example.html over the established TCP connection as shown in packet 4. The server then acknowledges the request and sends a packet with the PSH flag set as shown in packets 6 and 7. The PSH flag is a request to the browser to say that it is OK to received the buffered data, i.e. example.html. The browser then sends back an acknowledgement and the server sends the page as shown in packets 7 and 8. Finally the browser sends a final acknowledgement of having received the page before initiating a graceful session teardown by sending a FIN ACK packet which indicates the end of a session. Once the server responds to the FIN ACK with it's own the browser sends a final acknowledgement. This then repeats itself when the browser parses the HTML and realises theres an image which it needs to get from the server as well, except the image is a larger file and so takes a few more PSH packets. In figures 6 and 7 you can see a set of ladder diagrams which show the entire transaction symbolically. I have also colour coded figure 7 with green arrow heads to the initial handshakes, blue for the HTTP protocol transactions and red for the TCP connection teardown packets.

This shows clearly the interaction between each of the different layers in the OSI model, the browser at level 7: Application rendering the webpage. Level 6: Presentation is skipped as we have no files which need to be served compressed because they are so large. Level 5: Session is shown by the TCP session negotiation and graceful teardown of the TCP session. Level 4: Transport is shown when the image and webpage are transferred from the server to the browser. Level 3/2/1 are shown in figure 8 where you can see the IP layer information along with Ethernet II and finally frame 4 which is the bytes that went down the wire.

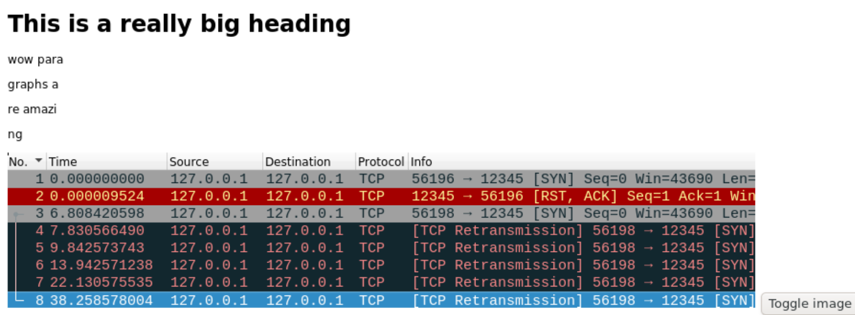
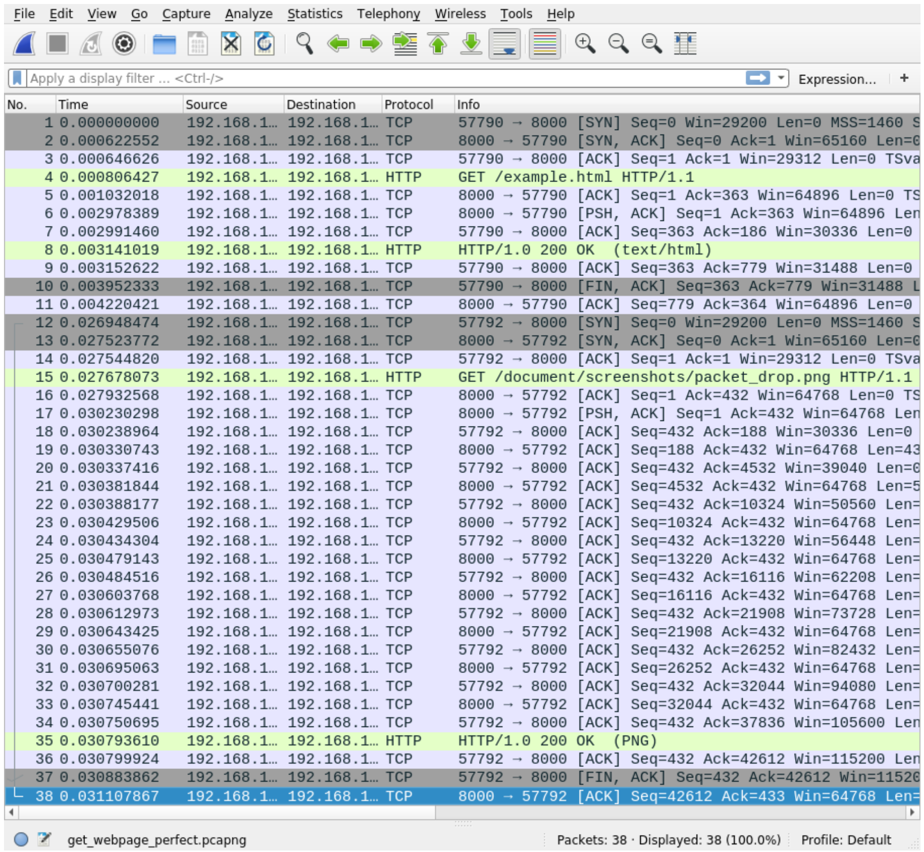


Figure 4: A basic static HTML webpage.



No.	Time	Source	Destination	Protocol	Info
1	0.000000000	192.168.1…	192.168.1…	TCP	57790 → 8000 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 S
2	0.000622552	192.168.1…	192.168.1…	TCP	8000 → 57790 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0
3	0.000646626	192.168.1…	192.168.1…	TCP	57790 → 8000 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSva
4	0.000806427	192.168.1…	192.168.1…	HTTP	GET /example.html HTTP/1.1
5	0.001032018	192.168.1…	192.168.1…	TCP	8000 → 57790 [ACK] Seq=1 Ack=363 Win=64896 Len=0 TS
6	0.002978389	192.168.1…	192.168.1…	TCP	8000 → 57790 [PSH, ACK] Seq=1 Ack=363 Win=64896 Len
7	0.002991460	192.168.1…	192.168.1…	TCP	57790 → 8000 [ACK] Seq=363 Ack=186 Win=30336 Len=0
8	0.003141019	192.168.1…	192.168.1…	HTTP	HTTP/1.0 200 OK (text/html)
9	0.003152622	192.168.1…	192.168.1…	TCP	57790 → 8000 [ACK] Seq=363 Ack=779 Win=31488 Len=0
10	0.003952333	192.168.1…	192.168.1…	TCP	57790 → 8000 [FIN, ACK] Seq=363 Ack=779 Win=31488 L
11	0.004220421	192.168.1…	192.168.1…	TCP	8000 → 57790 [ACK] Seq=779 Ack=364 Win=64896 Len=0
12	0.026948474	192.168.1…	192.168.1…	TCP	57792 → 8000 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 S
13	0.027523772	192.168.1…	192.168.1…	TCP	8000 → 57792 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0
14	0.027544820	192.168.1…	192.168.1…	TCP	57792 → 8000 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSva
15	0.027678073	192.168.1…	192.168.1…	HTTP	GET /document/screenshots/packet_drop.png HTTP/1.1
16	0.027932568	192.168.1…	192.168.1…	TCP	8000 → 57792 [ACK] Seq=1 Ack=432 Win=64768 Len=0 TS
17	0.030230298	192.168.1…	192.168.1…	TCP	8000 → 57792 [PSH, ACK] Seq=1 Ack=432 Win=64768 Len
18	0.030238964	192.168.1…	192.168.1…	TCP	57792 → 8000 [ACK] Seq=432 Ack=188 Win=30336 Len=0
19	0.030330743	192.168.1…	192.168.1…	TCP	8000 → 57792 [ACK] Seq=188 Ack=432 Win=64768 Len=43
20	0.030337416	192.168.1…	192.168.1…	TCP	57792 → 8000 [ACK] Seq=432 Ack=4532 Win=39040 Len=0
21	0.030381844	192.168.1…	192.168.1…	TCP	8000 → 57792 [ACK] Seq=4532 Ack=432 Win=64768 Len=5
22	0.030388177	192.168.1…	192.168.1…	TCP	57792 → 8000 [ACK] Seq=432 Ack=10324 Win=50560 Len=
23	0.030429506	192.168.1…	192.168.1…	TCP	8000 → 57792 [ACK] Seq=10324 Ack=432 Win=64768 Len=
24	0.030434304	192.168.1…	192.168.1…	TCP	57792 → 8000 [ACK] Seq=432 Ack=13220 Win=56448 Len=
25	0.030479143	192.168.1…	192.168.1…	TCP	8000 → 57792 [ACK] Seq=13220 Ack=432 Win=64768 Len=
26	0.030484516	192.168.1…	192.168.1…	TCP	57792 → 8000 [ACK] Seq=432 Ack=16116 Win=62208 Len=
27	0.030603768	192.168.1…	192.168.1…	TCP	8000 → 57792 [ACK] Seq=16116 Ack=432 Win=64768 Len=
28	0.030612973	192.168.1…	192.168.1…	TCP	57792 → 8000 [ACK] Seq=432 Ack=21908 Win=73728 Len=
29	0.030643425	192.168.1…	192.168.1…	TCP	8000 → 57792 [ACK] Seq=21908 Ack=432 Win=64768 Len=
30	0.030655076	192.168.1…	192.168.1…	TCP	57792 → 8000 [ACK] Seq=432 Ack=26252 Win=82432 Len=
31	0.030695063	192.168.1…	192.168.1…	TCP	8000 → 57792 [ACK] Seq=26252 Ack=432 Win=64768 Len=
32	0.030700281	192.168.1…	192.168.1…	TCP	57792 → 8000 [ACK] Seq=432 Ack=32044 Win=94080 Len=
33	0.030745441	192.168.1…	192.168.1…	TCP	8000 → 57792 [ACK] Seq=32044 Ack=432 Win=64768 Len=
34	0.030750695	192.168.1…	192.168.1…	TCP	57792 → 8000 [ACK] Seq=432 Ack=37836 Win=105600 Len
35	0.030793610	192.168.1…	192.168.1…	HTTP	HTTP/1.0 200 OK (PNG)
36	0.030799924	192.168.1…	192.168.1…	TCP	57792 → 8000 [ACK] Seq=432 Ack=42612 Win=115200 Len
37	0.030883862	192.168.1…	192.168.1…	TCP	57792 → 8000 [FIN, ACK] Seq=432 Ack=42612 Win=11520
38	0.031107867	192.168.1…	192.168.1…	TCP	8000 → 57792 [ACK] Seq=42612 Ack=433 Win=64768 Len=

Figure 5: A full chain of packets that shows retrieving a basic webpage from the server.

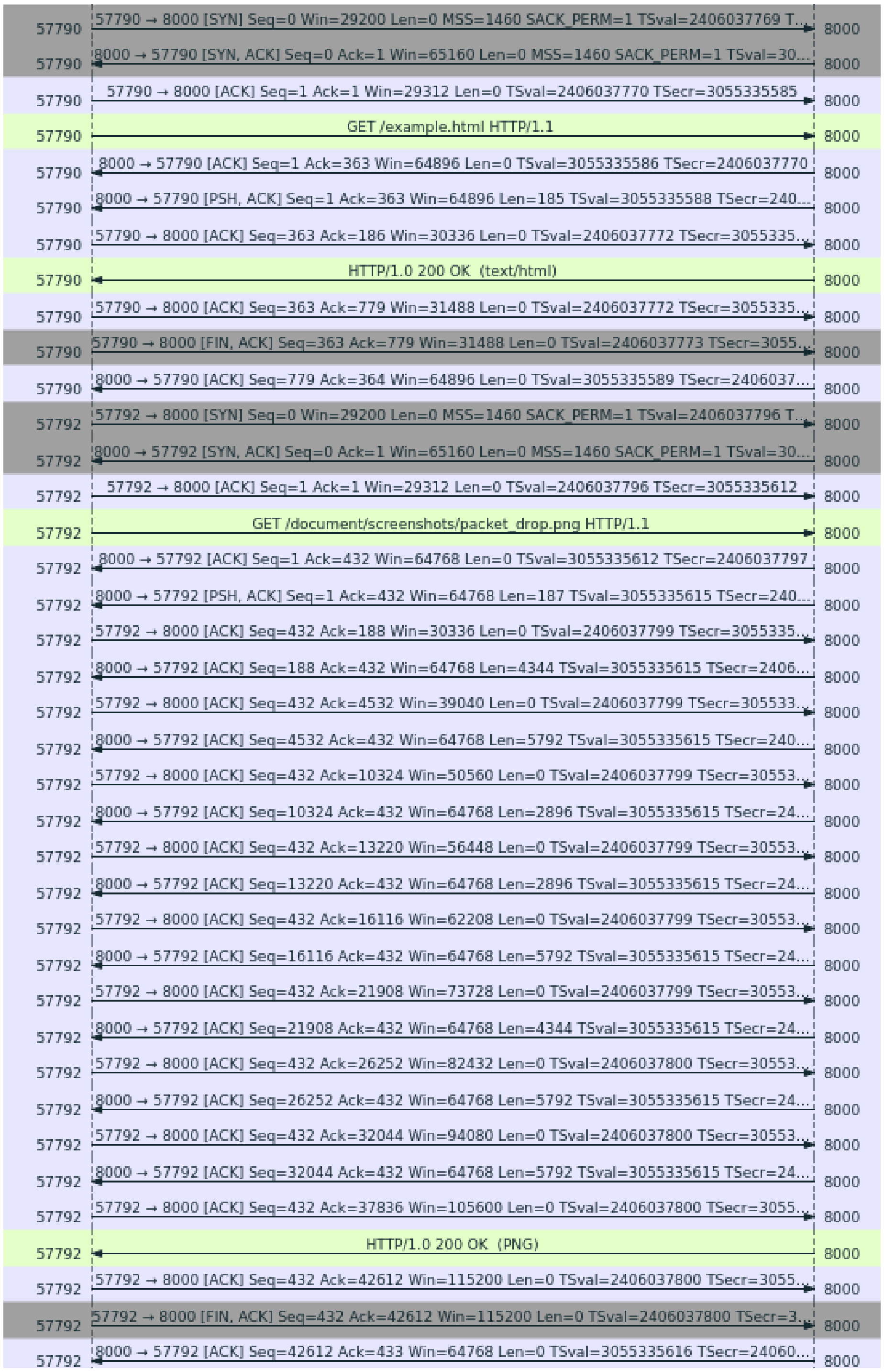


Figure 6: Ladder diagram of figure 5.

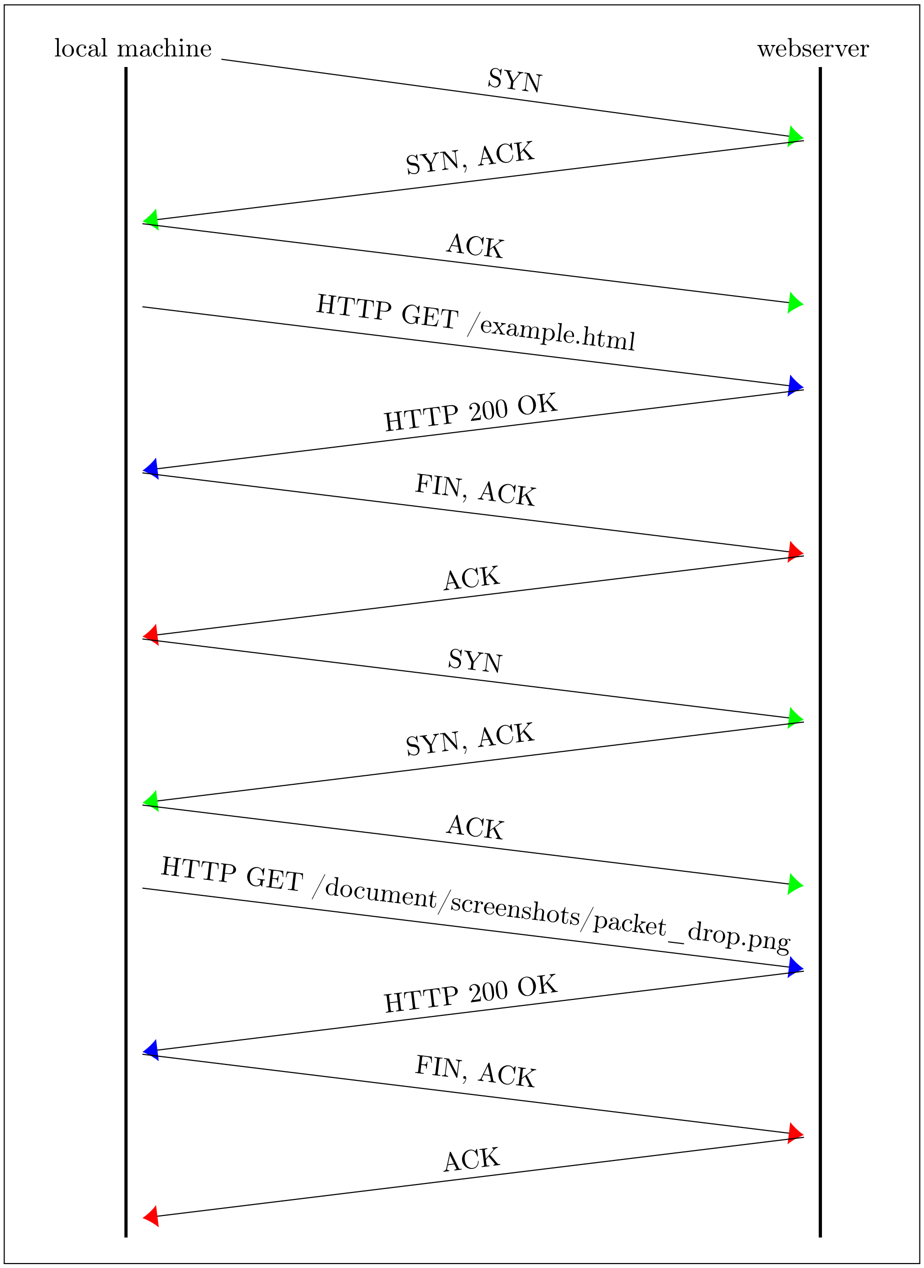


Figure 7: A simplified ladder diagram showing the transaction in figure 5

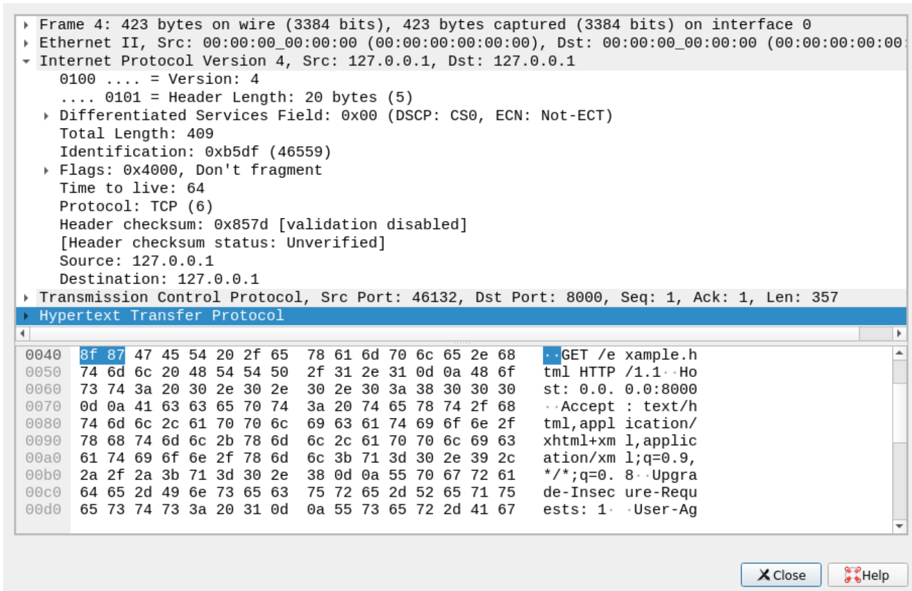


Figure 8: A look inside a TCP packet.

Listing 1: example.html

```
 <!DOCTYPE html>
<html>
<head>
<title>Wow I can add titles</title>
</head>
<body>

<h1>This is a really big heading</h1>
<p>wow para</p>
<p>graphs a</p>
<p>re amazi</p>
<p>ng</p>
  <script type="text/javascript">
    function imgtog() {
      if (document.getElementById("img").style.display == "none") {
        document.getElementById("img").style = "block"
      } else {
        document.getElementById("img").style.display = "none"
      }
    }

  </script>

```

```
<img id="img" src="document/screenshots/packet_drop.png">

<button onclick="imgtog()">Toggle image</button>


</body>
</html>
```

1.2 Analysis of problem

The problem with looking at a network from the outside is that the purpose of the network is to allow communication inside of the network, thus very little is exposed externally. This presents a challenge as we want to know what is on the network as well as what each of them is running which is not always possible due to the limited information that services will reveal about themselves. Firewalls also play large part in making scanning networks difficult as sometimes they simply drop packets instead of sending a TCP RST packet (reset connection packet). When firewalls drop packets it becomes exponentially more difficult as you don't know whether your packet was corrupted or lost in transit or if it was just dropped. Dropping a packet means that when a packet is received no response is sent back as if the connection was just "dropped".

To demonstrate this I will show three things:

1. A successful connection over TCP.
2. An attempted connection to a closed port.
3. An attempted connection with a firewall rule to drop packets.

Firstly A successful TCP connection. For a TCP connection to be established there is a three way handshake between the communicating machines. Firstly the machine trying to establish the connection sends a TCP SYN packet to the other machine, this packet holds a dual purpose, to ask for a connection and if it is accepted to SYNchronise the sequence numbers being used to detect whether packets have been lost in transport. The receiving machine then replies with a TCP SYN ACK which confirms the starting sequence number with the SYN part and ACKnowledges the connection request. The sending machine then acknowledges this by sending a final TCP ACK packet back. This connection initialisation is shown in figure 9 by packets one, two and three. Data transfer can then commence by sending a TCP packet with the PSH and ACK flags set along with the data in the data portion of the packet, this is shown in figure 12 where wireshark allows us to take a look inside the packet to see the data being sent in the packet along with the PSH and ACK flags being set. The code I used to generate these is shown in figures 10 and 11. Breaking the code down in figure 11 you can see me initialising a socket object then I bind it to localhost (127.0.0.1) port 12345 localhost is just an address which allows connections between programs running on the same computer as connections are looped back onto the current machine, hence its alternative name: the loopback address. I then tell it to listen for incoming connections, the one just means how many connections to keep as a backlog. I then accept the connection from the program in figure 10, line 3. I then tell the program to listen for up to 1024 bytes in the data part of any TCP packets sent. The program in figure 10 then sends some data which we then see printed to the screen in figure 11, both programs then close the connection.

No.	Time	Source	Destination	Protocol	Info
1	0.000000000	127.0.0.1	127.0.0.1	TCP	47710 → 12345 [SYN] Seq=0
2	0.000019294	127.0.0.1	127.0.0.1	TCP	12345 → 47710 [SYN, ACK]
3	0.000033431	127.0.0.1	127.0.0.1	TCP	47710 → 12345 [ACK] Seq=1
4	53.378941809	127.0.0.1	127.0.0.1	TCP	47710 → 12345 [PSH, ACK]
5	53.378958066	127.0.0.1	127.0.0.1	TCP	12345 → 47710 [ACK] Seq=1
6	65.928944995	127.0.0.1	127.0.0.1	TCP	12345 → 47710 [FIN, ACK]
7	65.936113471	127.0.0.1	127.0.0.1	TCP	47710 → 12345 [ACK] Seq=3
8	85.536923935	127.0.0.1	127.0.0.1	TCP	47710 → 12345 [FIN, ACK]
9	85.536940026	127.0.0.1	127.0.0.1	TCP	12345 → 47710 [ACK] Seq=2

Figure 9: Packets starting a TCP session, transferring some data then ending it.

```
In [1]: import socket

In [2]: sender = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

In [3]: sender.connect(("127.0.0.1", 12345))

In [4]: sender.send(b"hi I'm data what's your name? "*10)
Out[4]: 300

In [5]: sender.close()
```

Figure 10: Transferring some basic text data over a TCP connection.

```
In [1]: import socket

In [2]: receiver = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

In [3]: receiver.bind(("127.0.0.1", 12345))

In [4]: receiver.listen(1)

In [5]: connection, address = receiver.accept()

In [6]: connection.recv(1024)
Out[6]: b"hi I'm data what's your name? hi I'm data what's your name? hi I'm data what's your name? hi I'm
 data what's your name? hi I'm data what's your name? hi I'm data what's your name? hi I'm data what's you
r name? hi I'm data what's your name? hi I'm data what's your name? hi I'm data what's your name? "

In [7]: connection.close()
```

Figure 11: Receiving some basic text data over a TCP connection.

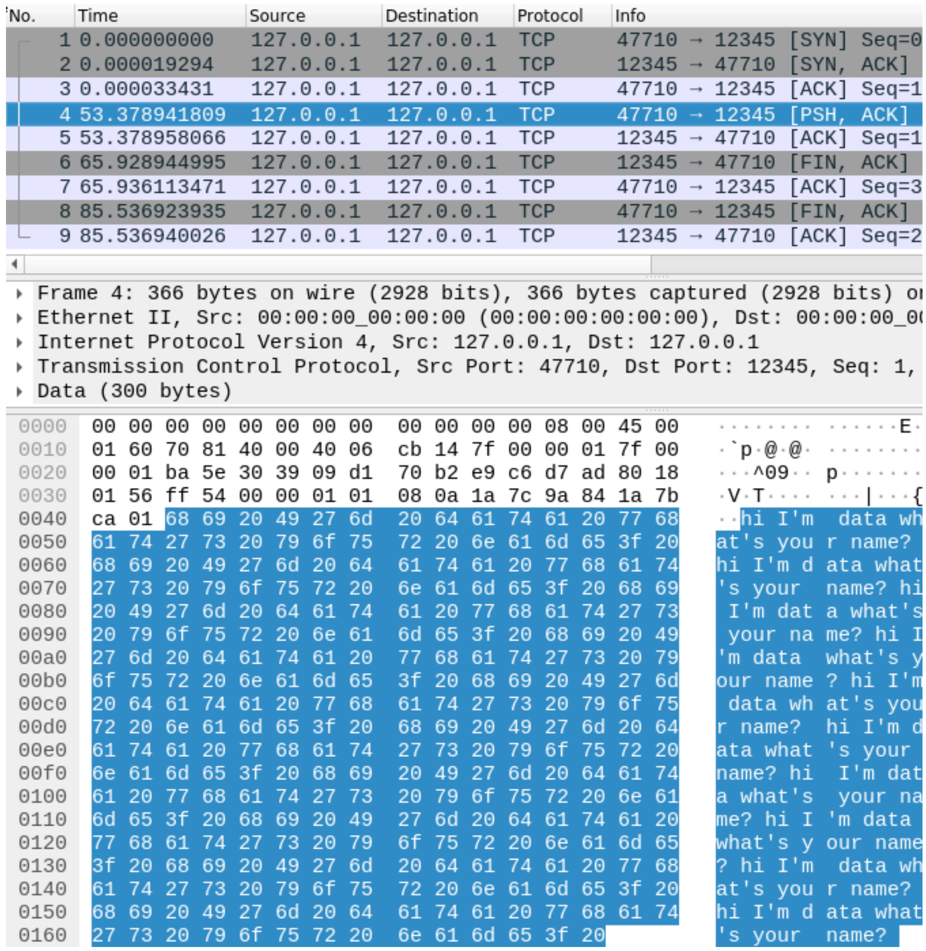


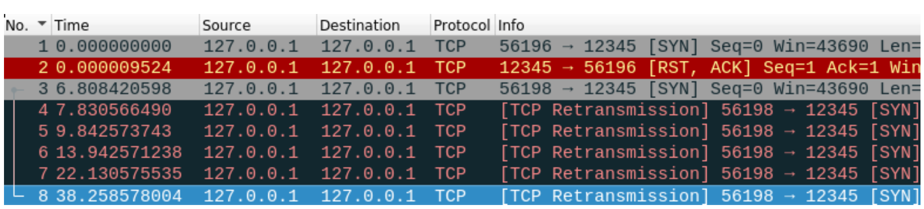
Figure 12: Highlighted packet carrying the data being transferred in figure 10.

Next an attempted connection to a closed port. In figure 13 packet one you can see the same TCP SYN packet as we saw in the attempted connection to an open port, as you would expect. The difference comes in the next packet with the TCP RST flag being sent back. This flag means to reset the connection, or if the connection is not yet established as in this case it means that the port is closed, hence why the packet is highlighted red in figure 13. The code used to generate this is shown in figure 14 line two shows the initialisation of a socket object. In line 3 the program tries to connect to port 12345 on localhost again, except this time we get a connection refused error back this shows us that the remote host sent a TCP RST packet back, which is reflected in figure 13.

Finally I will show a connection where the firewall is configured to drop the packet. However first I will explain a bit about firewalls and how they work.

Firewalls are essentially the gatekeepers of the internet they decide whether a packet gets to pass or whether they shall not pass. Firewalls work by a set of rules which decide what happens to it. A rule might be that it is coming from a certain IP address or has a certain destination port. The actions taken after the packet has had it's fate decided by the rules can be one of the following three (on iptables on linux): ACCEPT, DROP and RETURN, accept does exactly what you think it would an lets the packet through, drop quite literally just drops the packet and sends no reply whatsoever, return is more complicated and has no effect on how port scanning is done and as such we will ignore it. A common set of rules for something like a webserver would be to DROP all incoming packets and then allow exceptions for certain ports i.e. port 80 for HTTP or 443 for Hypertext transfer Protocol Secure (HTTPS). I will be using a linux utility called iptables for implementing all firewall rules on my system for demonstration purposes. Packet number three in figure 13 shows the connection request from line 4 of 14 except that I have enabled a firewall rule to drop all packets from the address 127.0.0.1, using the iptables command as so: `iptables -I INPUT -s 127.0.0.1 -j DROP`. This command reads as for all packets arriving (`-I INPUT`) with source address 127.0.0.1 (`-s 127.0.0.1`) drop them sending no response (`-j DROP`). With this firewall rule in place you can see in figure 13 packet 3 receives no response and as such Python assumes that the packet just got lost and as such tries to send the packet again repeatedly, this continued for more than 30 seconds before a stopped it as shown by the time column in figure 13 and the final `KeyboardInterrupt` in figure 14. The amount of time that a system will wait still trying to reconnect depends on the OS and a other factors but the minimum time is 100 seconds as specified by RFC 1122, on most systems it will be between 13 and 30 minutes according the linux manual page on TCP.

```
man 7 tcp:
tcp_retries2 (integer; default: 15; since Linux 2.2)
  The maximum number of times a TCP packet is retransmitted in
  established state before giving up. The default value is 15,
  which corresponds to a duration of approximately between 13 to 30
  minutes, depending on the retransmission timeout. The RFC 1122
  specified minimum limit of 100 seconds is typically deemed too short.
```



No.	Time	Source	Destination	Protocol	Info
1	0.000000000	127.0.0.1	127.0.0.1	TCP	56196 → 12345 [SYN] Seq=0 Win=43690 Len=
2	0.000009524	127.0.0.1	127.0.0.1	TCP	12345 → 56196 [RST, ACK] Seq=1 Ack=1 Win
3	6.808420598	127.0.0.1	127.0.0.1	TCP	56198 → 12345 [SYN] Seq=0 Win=43690 Len=
4	7.830566490	127.0.0.1	127.0.0.1	TCP	[TCP Retransmission] 56198 → 12345 [SYN]
5	9.842573743	127.0.0.1	127.0.0.1	TCP	[TCP Retransmission] 56198 → 12345 [SYN]
6	13.942571238	127.0.0.1	127.0.0.1	TCP	[TCP Retransmission] 56198 → 12345 [SYN]
7	22.130575535	127.0.0.1	127.0.0.1	TCP	[TCP Retransmission] 56198 → 12345 [SYN]
8	38.258578004	127.0.0.1	127.0.0.1	TCP	[TCP Retransmission] 56198 → 12345 [SYN]

Figure 13: Attempted connection to a closed port with and without firewall rule to drop packets.

```
In [1]: import socket

In [2]: a = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

In [3]: a.connect(("127.0.0.1", 12345))
---------------------------------------------------------------------------
ConnectionRefusedError                    Traceback (most recent call last)
<ipython-input-3-fbc96d60b5f2> in <module>
----> 1 a.connect(("127.0.0.1", 12345))

ConnectionRefusedError: [Errno 111] Connection refused

In [4]: a.connect(("127.0.0.1", 12345))
^C---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
```

Figure 14: The code used to produce firewall packet dropping example in figure 13

Having explained firewalls, how they affect port scanning and other things above I will now explain what I am actually trying to achieve with my project and how I am going to do it. I am trying to make a tool similar to nmap which will be able to detect the state (as in whether the port is open/closed or filtered etc) of ports on remote machines, detect which hosts are up on a subnet and finally I want to be able to try to detect what services are listening behind any of the ports. I am going to be writing in Python version 3.7.2 as it is the latest stable release of Python 3 and has many features which are not in even fairly recent versions such as 3.5, the biggest one of these being fstrings which are where I can put a single a 'f' before a string and then any formatting options I put inside using curly braces are expanded and formatted accordingly. This allows for a clear and consistent string formatting syntax which I will use extensively. I will be using Python in particular as a language because it is very readable and has extensive low level bindings to C networking functions with the socket module allowing me to write code quickly which is easily understandable and has a clear purpose and at the same time be able to use low level networking functions and even changing the behaviour at this low level with `socket.setsockopt`. As well as this the socket module allows me to open sockets that communicate using many different protocols such as TCP, UDP and Internet Control Message Protocol (ICMP) just to name a few. These features combine to make Python a great language for writing networking software with a high level of abstraction. In regards to the OSI model my code will sit with the user interface at level 7 specifying what to do at a high level then the actual scanning takes place at levels 3, 4 and 5 with host detection being at level 3. Port scanning will be taking place At level 4 for TCP SYN scanning and UDP scanning. Whereas `connect()` scanning and version detection will sit at level 5. Finally I will look at what is actually handling all of the networking on my machine. My machine runs linux and as such all networking is handled by system calls to the linux kernel. For example the `socket.connect` method is just a call to the underlying linux kernel's connect syscall but presenting a kinder call signature to the user as the Python socket library does some processing before the syscall is made.

1.3 Success Criteria

1. Probe another computer's networking from a black box perspective.
2. To help the user with usage/help messages when prompted.
3. Translate Classless Inter-Domain Routing (CIDR) specified subnets into a list of domains.
4. Send ICMP ECHO requests to determine whether a machine is active or not.
5. Perform any scan type without first checking whether the host is up.
6. Detect whether a TCP port is open (can be connected to).
7. Detect whether a TCP port is closed (will refuse connections).
8. Detect whether a TCP port is filtered (a firewall is preventing or monitoring access).
9. Detect whether a UDP port is open (can be connected to).
10. Detect whether a UDP port is closed (will refuse connections).
11. Detect whether a UDP port is filtered (a firewall is preventing or monitoring access).
12. Detect the operating system of another machine on the network solely from sending packets to the machine and interpreting the responses.
13. Detect what service is listening behind a port.
14. Detect the version of the service running behind a port.

1.4 Description of current system or existing solutions

Nmap is currently the most popular tool for doing port scanning and host enumeration. It supports the scanning types for determining information about remote hosts.

- TCP: SYN
- TCP: `Connect()`
- TCP: ACK
- TCP: Window

- TCP: Maimon
- TCP: Null
- TCP: FIN
- TCP: Xmas
- UDP
- Zombie host/idle
- Stream Control Transmission Protocol (SCTP): INIT
- SCTP: COOKIE-ECHO
- IP protocol scan
- FTP: bounce scan

As well as supporting a vast array of scanning types it also can do service version detection and operating system detection via custom probes. Nmap also has script scanning which allows the user to write a script specifying exactly how they want to scan e.g. to circumvent port knocking (where packets must be sent to a sequence of ports in order before access to the finalportis allowed). It also supports a plethora of options to avoid firewalls or Intrusion Detection System (IDS) such as sending packets with spoofed checksums/source addresses and sending decoy probes. Nmap can do many more things than I have listed above as is illustrated quite clearly by the fact there is an entire working on using nmap (https://nmap.org/book/). The following is an example nmap scan which I did on my home network: `nmap -sC -sV -oA networkscan 192.168.1.0/24`. Breaking it down this means to enable script scanning `-sc`, enable version detection `-sV` and then output all results in all the common formats: XML, nmap and greppable, using the base name `networkscan` which produces three files: `networkscan.(nmap,gnmap,xml)`. Before I go into what each file contains I will explain some terminology, greppable is anything which can be easily searched with the linux `grep` which stands for Globally search a Regular Expression and Print, which basically means look in files for lines that contain a certain word or pattern, for example finding all lines with the word "hi" in them in the file "document" `grep hi document`. Onto the files: `networkscan.nmap` contains what would usually be printed by nmap while the scan is being run, it looks like this:

```
# Nmap 7.70 scan initiated Wed Apr 10 19:36:18 2019 as:
    nmap -sC -sV -oA /home/tritoke/thing 192.168.1.0/24
Nmap scan report for router.asus.com (192.168.1.1)
Host is up (1.0s latency).
Not shown: 995 closed ports
PORT     STATE SERVICE    VERSION
```

```
53/tcp   open  domain     (generic dns response: NOTIMP)
| fingerprint-strings:
|   DNSVersionBindReqTCP:
|     version
|_    bind
80/tcp   open  http       ASUS WRT http admin
|_http-server-header: httpd/2.0
|_http-title: Site doesn't have a title (text/html).
515/tcp  open  printer
8443/tcp open  ssl/http   ASUS WRT http admin
|_http-server-header: httpd/2.0
|_http-title: Site doesn't have a title (text/html).
| ssl-cert: Subject: commonName=192.168.1.1/countryName=US
| Not valid before: 2018-05-05T05:05:17
|_Not valid after:  2028-05-05T05:05:17
9100/tcp open  jetdirect?
1 service unrecognized despite returning data. If you know the service/version,
please submit the following fingerprint at
https://nmap.org/cgi-bin/submit.cgi?new-service :
SF-Port53-TCP:V=7.70%I=7%D=4/10%Time=5CAE3DC5%P=x86_64-pc-linux-gnu%r(DNSV
SF:ersionBindReqTCP,20,"\0\x1e\0\x06\x85\x85\0\x01\0\0\0\0\0\0\x07version\
SF:x04bind\0\0\x10\0\x03")%r(DNSStatusRequestTCP,E,"\0\x0c\0\0\x90\x04\0\0
SF:\0\0\0\0\0\0");
Service Info: CPE: cpe:/o:asus:wrt_firmware
```

Above is just the report for one such device in the report as the full thing is over 200 lines lone. In it you can see information such as which ports are open and what services are running behind them as this is my router you can see port 8443 which nmap has recognised to be hosting the ASUS web admin from which you can configure the route. Then after than some other associated information extracted from the server. Most of this extra information is from the `-sC` flag which is script scanning and allows advanced interaction with running services specifically to gain more information by providing specialised probing per protocol. We can also see at the end an unrecognised service which nmap shows us the data it returned and asks us to submit a new service report at a given URL if we recognise the service. This system of submitting fingerprints of services is how nmap is so good at recognising services: it has a lot of data to look at and learn from in regards to service fingerprinting.

Next `networkscan.gnmap`:

```
# Nmap 7.70 scan initiated Wed Apr 10 19:36:18 2019 as:
    nmap -sC -sV -oA /home/tritoke/networkscan 192.168.1.0/24
Host: 192.168.1.1 (router.asus.com) Status: Up
Host: 192.168.1.1 (router.asus.com) Ports: 53/open/tcp//domain//
      (generic dns response: NOTIMP)/, 80/open/tcp//http//ASUS WRT http admin/,
      515/open/tcp//printer///, 8443/open/tcp//ssl|http//ASUS WRT http admin/,
      9100/open/tcp//jetdirect?///  Ignored State: closed (995)
```

```
Host: 192.168.1.8 (android-25a97e36c2e74456)  Status: Up
Host: 192.168.1.8 (android-25a97e36c2e74456)  Ports: 5060/filtered/tcp//sip///
      Ignored State: closed (999)
```

Again this is not all of the file as it is very large. As you can see above all of the information is on a single line for each type of scan, this is useful if you want to scan a large number of hosts and just want to know which hosts are up you can do `grep 'Status: Up' networkscan.gnmap` which outputs this:

```
$ grep 'Status: Up' networkscan.gnmap
Host: 192.168.1.1 (router.asus.com) Status: Up
Host: 192.168.1.8 (android-25a97e36c2e74456) Status: Up
Host: 192.168.1.10 (diskstation) Status: Up
Host: 192.168.1.88 () Status: Up
Host: 192.168.1.88 () Status: Up
Host: 192.168.1.117 () Status: Up
Host: 192.168.1.159 (groot) Status: Up
Host: 192.168.1.159 (groot) Status: Up
Host: 192.168.1.176 (ET0021B7C01F2E) Status: Up
```

Showing you clearly the hosts which are online and then their host names. Other ways to use this output format would be to find out which ports are open on only one machine, or which hosts have a webserver running on them or a vulnerable version of a mail server etc. In general it is useful for when you want to filter results.

Finally we have eXtensible Markup Language (XML) format:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE nmaprun>
<?xml-stylesheet href="file:///usr/bin/../share/nmap/nmap.xsl"
    type="text/xsl"?>
<!-- Nmap 7.70 scan initiated Wed Apr 10 19:36:18 2019 as: nmap -sC -sV
    -oA /home/tritoke/thing 192.168.1.0/24 -->
<nmaprun scanner="nmap" args="nmap -sC -sV -oA /home/tritoke/thing
    192.168.1.0/24" start="1554921378" startstr="Wed Apr 10 19:36:18
    2019" version="7.70" xmloutputversion="1.04">
<verbose level="0"/>
<debugging level="0"/>
<host starttime="1554921379" endtime="1554923187"><status state="up"
    reason="syn-ack" reason_ttl="0"/>
<address addr="192.168.1.1" addrtype="ipv4"/>
<hostnames>
<hostname name="router.asus.com" type="PTR"/>
</hostnames>
<ports><extraports state="closed" count="995">
<extrareasons reason="conn-refused" count="995"/>
</extraports>
<port protocol="tcp" portid="53"><state state="open" reason="syn-ack"
    reason_ttl="0"/><service name="domain" extrainfo="generic dns
```

```
    response: NOTIMP"
    servicefp="SF-Port53-TCP:V=7.70%I=7%D=4/10%Time=5CAE3DC5%P=x86_64
-pc-linux-gnu%r(DNSVersionBindReqTCP,20,&quot;\0\x1e\0\x06\x85\x85\0
\x01\0\0\0\0\0\0\x07version\x04bind\0\0\x10\0\x03&quot;)%r
(DNSStatusRequestTCP,E,&quot;\0\x0c\0\0\x90\x04\0\0\0\0\0\0\0\0&quot;);"
    method="probed" conf="10"/><script id="fingerprint-strings"
    output="&#xa; DNSVersionBindReqTCP: &#xa; version&#xa; bind"><elem
    key="DNSVersionBindReqTCP">&#xa; version&#xa; bind</elem>
</script></port>
```

It is verbose in the extreme contains the reason why each port has the state it does as well as a vast amount of other data that the other scans didn't include as well as this it is not very human readable meaning that this format is more likely available because it is easier for other programs to parse than the other formats. As well as this the verbosity can be good if you really need to dive into why a port was marked as closed etc or the exact bytes that a service replied with.

In terms of where nmap lives in the software stack is that it is an application at level 7 when the user interacts with it but it uses several libraries which interact at level 2 which it uses to get the raw headers of the packets being sent and thus gain information from them. Nmap has virtually no competitors other than possibly Angry IP Scanner which is another open source network scanner expect it has a much smaller user base.

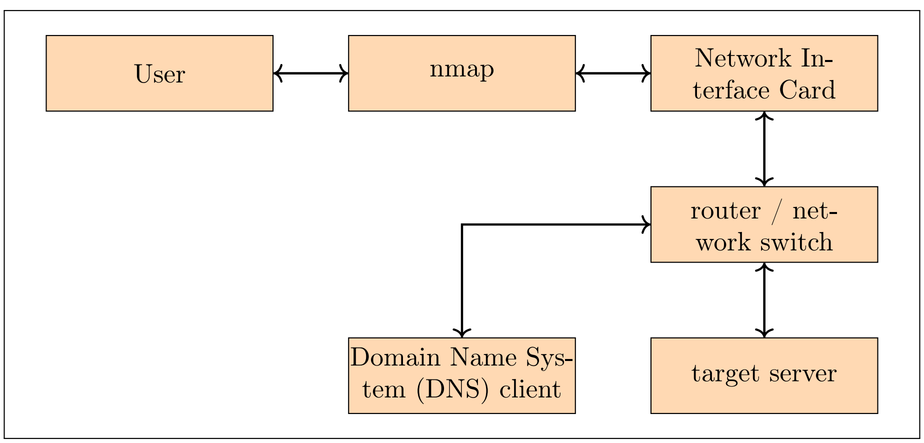


Figure 15: A block diagram showing how nmap sits in the software stack.

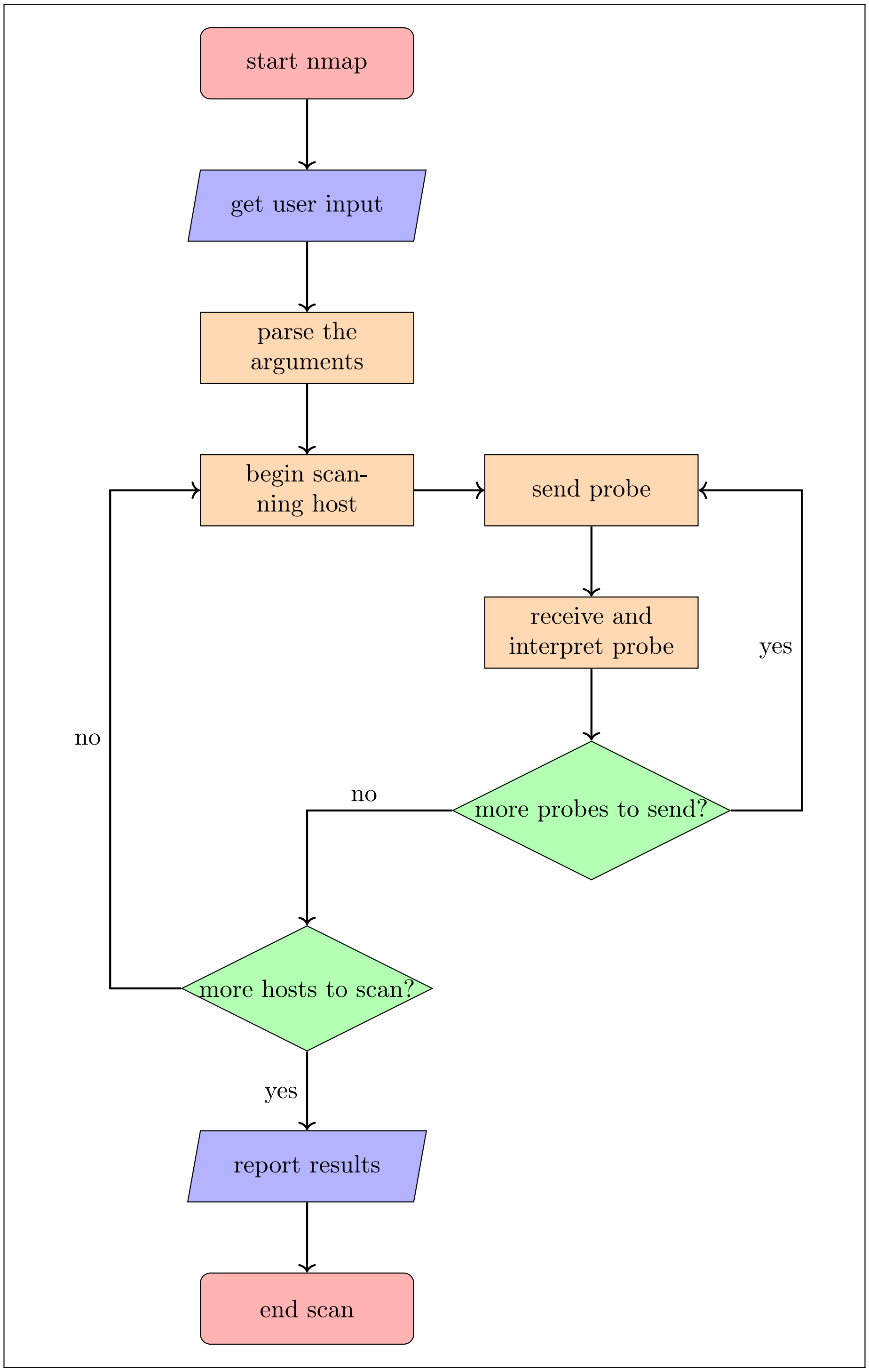


Figure 16: A flow chart showing how nmap does scanning.

1.5 Prospective Users

The prospective users of this system would be system administrators, penetration testers or network engineers. In my case my prospective users would be my school's system administrators and it would allow them to see an outsiders perspective on for example the server running the school's website page or to see if any of the programs on the servers were leaking information through banners etc. (most services send a banner with information like what protocol version they use and other information). Banners are short strings of text which a service or program will send to identify itself when it receives a new connection. They often contain information such as protocol version etc, which allows the connecting client to know how to communicate with the service. However they can also reveal too much information such as the version number of the service running, if the service version is old then it is likely that bugs will have been found in the program since then this information could allow an attacker to gain access to the server by exploiting the vulnerability in that service. This can obviously be prevented by keeping services up to date, however that is not always possible so as a best practice banners should reveal the minimum amount of information possible such that the client can interact with the service.

1.6 Data Dictionary

So while my program is running it will need to store many different things in memory:

- the list of hosts to scan
- the list of ports to scan on each host
- the state of each port we are scanning on each host
- the packet received by the listening socket (temporarily before processing)
- various counters and positional indicators are almost inevitable
- the probes to be used for version detection

So I am going to try to estimate the amount of RAM my program will use based on scanning a CIDR specified subnet of 192.168.1.0/24, and the most common ports 1000 ports of each machine I will not consider version detection as I am unsure of how I will implement it currently. To measure the size of object in python we can use the `getsizeof` function provided by the `sys` module, I also have a file called 'hosts' which contains the addresses specified by 192.168.1.0/24 and a file 'ping_bytes' which contains 4 captured packets from the ping command which I captured during an early exploratory testing phase.

Listing 2: some testing I did on the size of python objects

```
>>> with open("hosts", "r") as f:
...     hosts = f.read().splitlines()
```

```
...
>>> import sys
>>> sys.getsizeof(hosts)
2216
>>> ports = list(range(1000))
>>> sys.getsizeof(ports)
9112
>>> len(hosts)*sys.getsizeof(ports) / 2**10 # 2*10 is one kibibyte
2278.0
>>> sys.getsizeof(True)
28
>>> len(hosts)*(sys.getsizeof(True)) / 2**10
7.0
>>> pings[0]
'45 00 00 54 0f 82 40 00 40 01 2d 25 7f 00 00 01 7f 00 00 01 08 00 41 c5
    02 4f 00 01 cd ef 0f 5c de 9b 0d 00 08 09 0a 0b 0c 0d 0e 0f 10 11
    12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f 20 21 22 23 24 25 26 27
    28 29 2a 2b 2c 2d 2e 2f 30 31 32 33 34 35 36 37'
>>> from binascii import unhexlify
>>> ping = unhexlify(pings[0].replace(" ", "")) # turn the string of
    numbers into a bytes object
>>> sys.getsizeof(ping)
117
>>> len(hosts)*sys.getsizeof(ping) / 2**10
29.25
>>> 2278.0 + 7.0 + 29.25 + 2.22
2316.47
```

As shown in Listing 2 we can see that by far the most space intensive item stored by our program will be the port numbers for each host, making up just less that ninety six percent of the total space used by the mock data I created. However overall 2.3 mebibytes is not a huge amount of data by any means.

Holding	Data type	Space used /Kib	Percentage of total
ports	List[int]	2278	98.34
hosts	List[str]	2.22	0.1
port state	List[bool]	7	0.3
packets	List[bytes]	29.25	1.26

1.7 Data Flow Diagram

In my application there will be three way information flow:

1. sending packets (data) out from my application
2. receiving packets back from the targets
3. how my program sends data around between functions

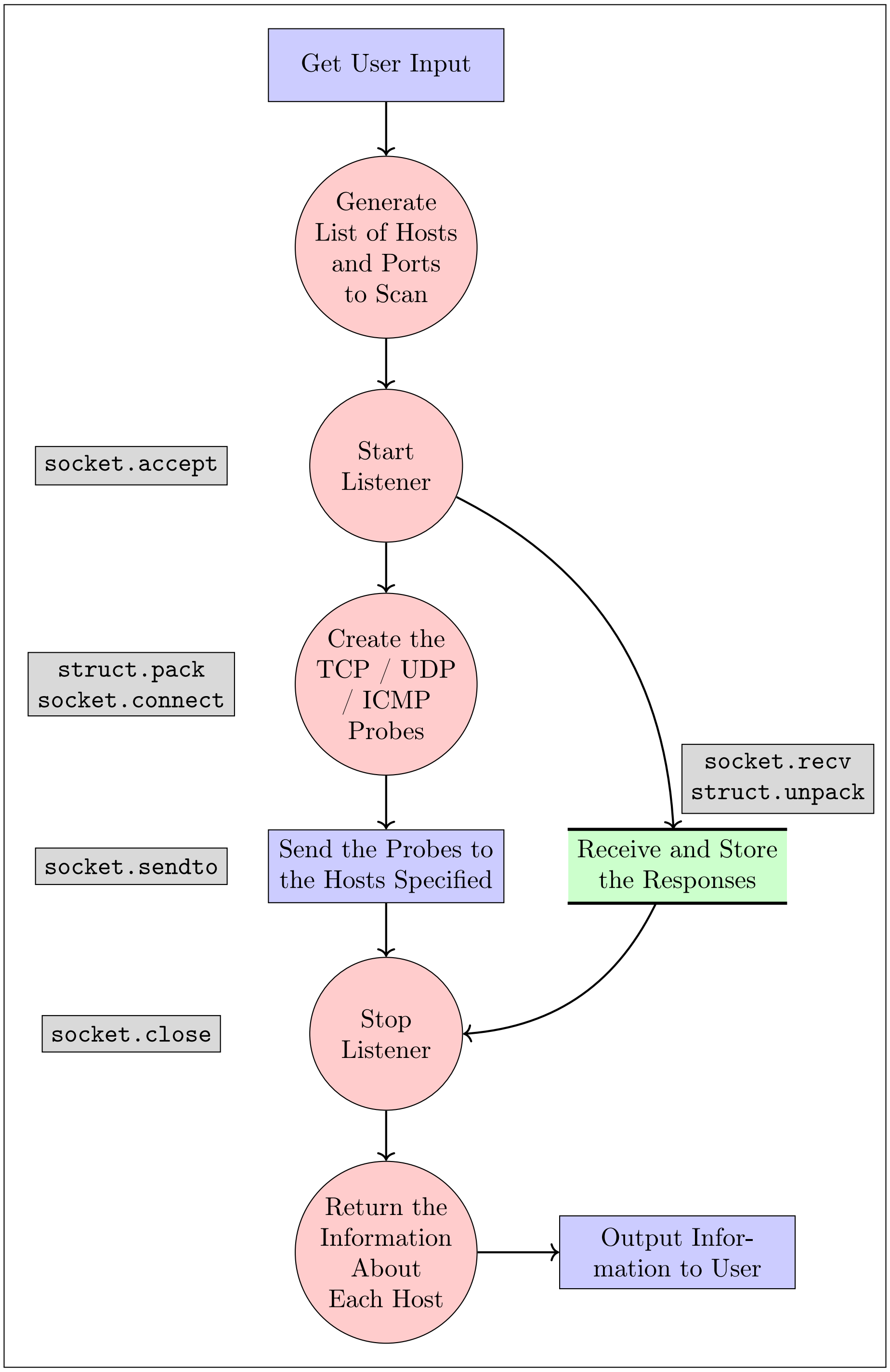


Figure 17: A data flow digram for information in my application.

1.8 Description of Solution Details

I will be using Python version 3.7.2 for my project because I am already familiar with Python's syntax and it's socket library has a very nice high level API for making system calls to the kernel's low level networking functions. This makes it very nice for a networking project like mine as it allows me to easily prototype and explore many ideas about how I could implement my solution without wasting vast amounts of time.

The first point of the success criteria that I wanted to get a feel for was receiving and sending ICMP ECHO requests aka pings. ICMP as a protocol sits at layer 3 of the OSI model this means it is a layer below what you are normally give access to in the socket module. This means instead of getting a bytes object with just the data from the header you instead get a bytes object which contains the entire packet and you have to dissect it yourself to get the information out of it, this can be quite difficult if it weren't for the struct module. The struct module provides a convenient API for converting between packed values i.e. packets in network endianness to unpacked values i.e. a double representing the current time in local endianness. Interactions with the socket module are mainly through the pack and unpack functions. For each of these functions you provide a format specifier defining how to unpack/pack the bytes/values. In Listing 3 you can see an example of me using the struct.pack function to pack the values which comprise an ICMP ECHO REQUEST into a packet and sending it the localhost address (127.0.0.1). This program is effectively the complement to the program listed in listing 4 which uses struct.unpack to unpack value from the received ICMP packet before printing the fields out to the terminal. Listing 3 makes use of the IP checksum function which I wrote. In figure 18 you can see the output when I run the command `ping 127.0.0.1` which the code in figure4 is listening for packets.

Listing 3: A prototype for sending ICMP ECHO REQUEST packets.

```python
#!/usr/bin/python3.7
import socket
import struct
import os
import time
import array

from os import getcwd, getpid
import sys
sys.path.append("../modules/")

import ip_utils


ICMP_ECHO_REQUEST = 8

# opens a raw socket for the ICMP protocol
```

```python
ping_sock = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_ICMP)
# allows manual IP header creation
# ping_sock.setsockopt(socket.SOL_IP, socket.IP_HDRINCL, 1)

ID = os.getpid() & 0xFFFF

# the two zeros are the code and the dummy checksum, the one is the sequence number
dummy_header = struct.pack("bbHHh", ICMP_ECHO_REQUEST, 0, 0, ID, 1)

data = struct.pack("d", time.time()) + bytes((192 - struct.calcsize("d")) * "A", "ascii")

checksum = ip_utils.ip_checksum(dummy_header+data)

header = struct.pack("bbHHh", ICMP_ECHO_REQUEST, 0, checksum, ID, 1)

packet = header + data

ping_sock.sendto(packet, ("127.0.0.1", 1))
```

Listing 4: A prototype for receiving ICMP ECHO REQUEST packets.

```python
#!/usr/bin/python3.7

import socket
import struct
import time
from typing import List

# socket object using an IPV4 address, using only raw socket access, set ICMP protocol
ping_sock = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_ICMP)

packets: List[bytes] = []

while len(packets) < 1:
    recPacket, addr = ping_sock.recvfrom(1024)
    ip_header = recPacket[:20]
    icmp_header = recPacket[20:28]

    ip_hp_ip_v, ip_dscp_ip_ecn, ip_len, ip_id, ip_flgs_ip_off, ip_ttl, ip_p, ip_sum, ip_src, ip_dst = struct.unpack('!BBHHHBBHII', ip_header)

    hl_v = f"{ip_hp_ip_v:08b}"
    ip_v = int(hl_v[:4], 2)
```

```python
    ip_hl = int(hl_v[4:], 2)
    dscp_ecn = f"{ip_dscp_ip_ecn:08b}"
    ip_dscp = int(dscp_ecn[:6], 2)
    ip_ecn = int(dscp_ecn[6:], 2)
    flgs_off = f"{ip_flgs_ip_off:016b}"
    ip_flgs = int(flgs_off[:3],2)
    ip_off = int(flgs_off[3:], 2)
    src_addr = socket.inet_ntoa(struct.pack('!I', ip_src))
    dst_addr = socket.inet_ntoa(struct.pack('!I', ip_dst))

    print("IP header:")
    print(f"Version: [{ip_v}]\nInternet Header Length: [{ip_hl}]\nDifferentiated Services Point Code: [{ip_dscp}]\nExplicit Congestion Notification: [{ip_ecn}]\nTotal Length: [{ip_len}]\nIdentification: [{ip_id:04x}]\nFlags: [{ip_flgs:03b}]\nFragment Offset: [{ip_off}]\nTime To Live: [{ip_ttl}]\nProtocol: [{ip_p}]\nHeader Checksum: [{ip_sum:04x}]\nSource Address: [{src_addr}]\nDestination Address: [{dst_addr}]\n")

    msg_type, code, checksum, p_id, sequence = struct.unpack('!bbHHh', icmp_header)
    print("ICMP header:")
    print(f"Type: [{msg_type}]\nCode: [{code}]\nChecksum: [{checksum:04x}]\nProcess ID: [{p_id:04x}]\nSequence: [{sequence}]"
    packets.append(recPacket)
open("current_packet", "w").write("\n".join(" ".join(map(lambda x: "{x:02x}", map(int, i))) for i in packets))
```

Listing 5: A function for calculating the IP checksum for a set of btyes.

```python
def ip_checksum(packet: bytes) -> int:
    """
    ip_checksum function takes in a packet
    and returns the checksum.
    """
    if len(packet) % 2 == 1:
        # if the length of the packet is even, add a NULL byte
        # to the end as padding
        packet += b"\0"

    total = 0
    for first, second in (
            packet[i:i+2]
            for i in range(0, len(packet), 2)
    ):
        total += (first << 8) + second

    # calculate the number of times a
```

```
    # carry bit was added and add it back on
    carried = (total - (total & 0xFFFF)) >> 16
    total &= 0xFFFF
    total += carried

    if total > 0xFFFF:
        # adding the carries generated a carry
        total &= 0xFFFF

    # invert the checksum and take the last 16 bits
    return (~total & 0xFFFF)
```

```
flags: [0]
fragment offset: [0]
ttl: [64]
prot: [1]
checksum: [28457]
source address: [127.0.0.1]
destination address: [127.0.0.1]

type: [0]
code: [0]
checksum: [9703]
p_id: [39682]
sequence: [256]


version: [4]
header length: [5]
dscp: [0]
ecn: [0]
total length: [21504]
identification: [21075]
flags: [0]
fragment offset: [64]
ttl: [64]
prot: [1]
checksum: [21737]
source address: [127.0.0.1]
destination address: [127.0.0.1]

type: [8]
code: [0]
checksum: [7566]
p_id: [39682]
sequence: [512]


version: [4]
header length: [5]
dscp: [0]
ecn: [0]
total length: [21504]
identification: [21331]
flags: [0]
fragment offset: [0]
ttl: [64]
prot: [1]
checksum: [21545]
source address: [127.0.0.1]
destination address: [127.0.0.1]

type: [0]
code: [0]
checksum: [7574]
p_id: [39682]
sequence: [512]
```

Figure 18: Dissecting an ICMP ECHO REQUEST packet.

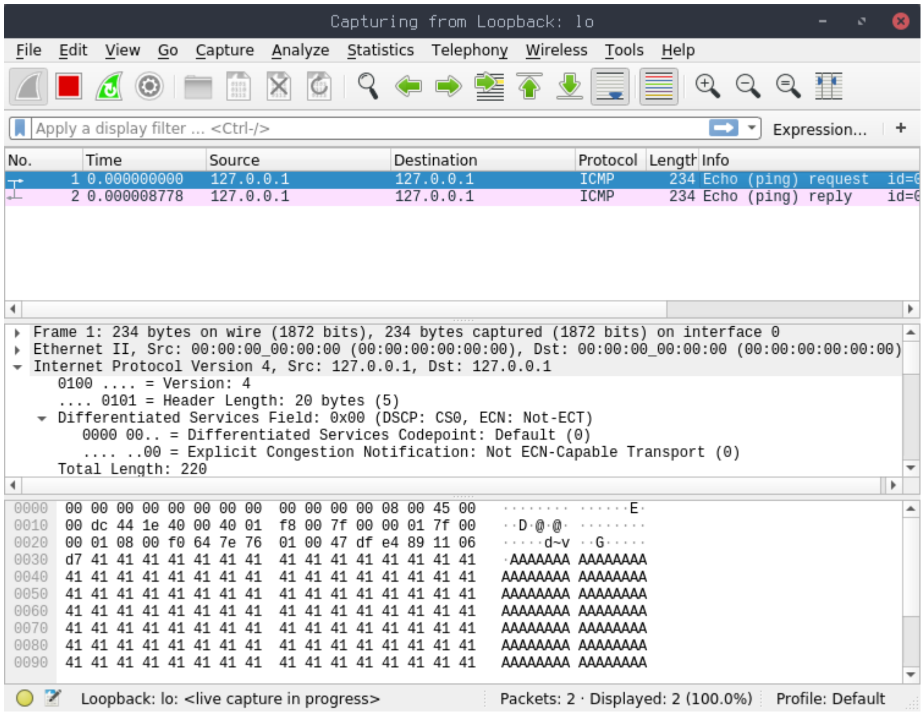


Figure 19: Screenshot of wireshark showing a successful send of an ICMP ECHO REQUEST packet.

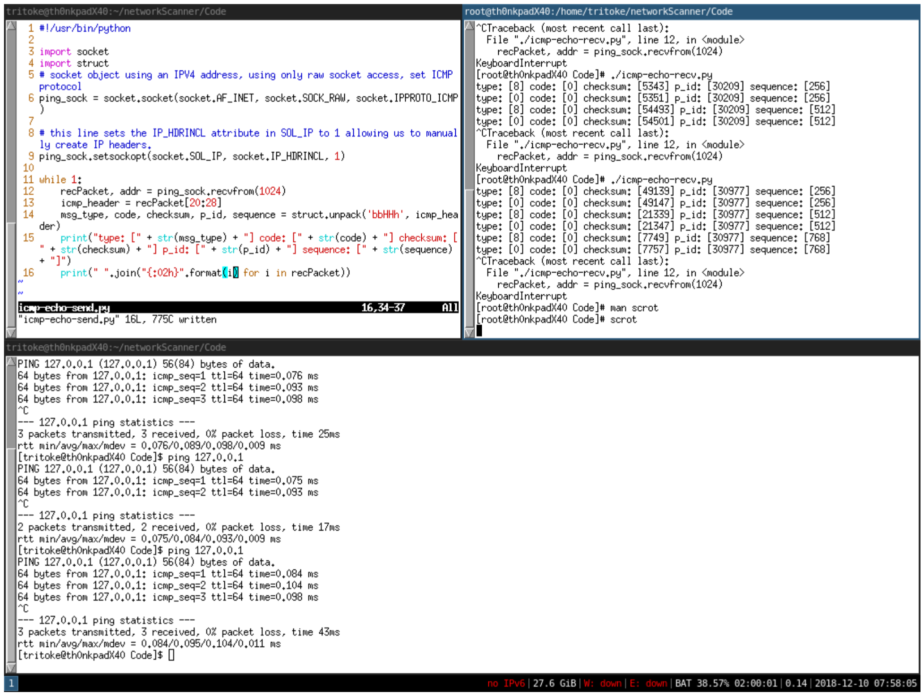


Figure 20: Screenshot showing me first successfully dissecting an ICMP ECHO REQUEST packet.

Having done these prototypes I have identified that it would probably be best to abstract the code for dissecting all the headers i.e. ICMP, TCP and Internet Protocol (IP) into classes where I can just pass the received packet into the class and have it dissect it for me and then I will also get access to some of the benefits of classes such as the `__repr__` method which is called when you print classes out and allows me to control what is printed out. Before I started to write the final piece I wanted to make a prototype ping scanner, as this would allow me to get a feel for making a scanner as well as further exploring low level protocol interactions.

Listing 6: An attempt at making a ping scanner.

```
#!/usr/bin/python3.7
from os import getcwd, getpid
import sys
sys.path.append("../modules/")

import ip_utils

import socket
```

```python
from functools import partial
from itertools import repeat
from multiprocessing import Pool
from contextlib import closing
from math import log10, floor
from typing import List, Tuple
import struct
import time


def round_significant_figures(x: float, n: int) -> float:
    """
    rounds x to n significant figures.
    round_significant_figures(1234, 2) = 1200.0
    """
    return round(x, n-(1+int(floor(log10(abs(x))))))


def recieved_ping_from_addresses(ID: int, timeout: float) -> List[Tuple[str, float, int]]:
    """
    Takes in a process id and a timeout and returns the list of addresses which sent
    ICMP ECHO REPLY packets with the packed id matching ID in the time given by timeout.
    """
    ping_sock = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_ICMP)
    time_remaining = timeout
    addresses = []
    while True:
        time_waiting = ip_utils.wait_for_socket(ping_sock, time_remaining)
        if time_waiting == -1:
            break
        time_recieved = time.time()
        recPacket, addr = ping_sock.recvfrom(1024)
        ip_header = recPacket[:20]
        ip_hp_ip_v, ip_dscp_ip_ecn, ip_len, ip_id, ip_flgs_ip_off, ip_ttl, ip_p, ip_sum, ip_src, ip_dst = struct.unpack('!BBHHHBBHII', ip_header)
        icmp_header = recPacket[20:28]
        msg_type, code, checksum, p_id, sequence = struct.unpack('bbHHh', icmp_header)
        time_remaining -= time_waiting
        time_sent = struct.unpack("d", recPacket[28:28+struct.calcsize("d")])[0]
        time_taken = time_recieved - time_sent
        if p_id == ID:
            addresses.append((str(addr[0]), float(time_taken),
```

```
                int(ip_ttl)))
        elif time_remaining <= 0:
            break
        else:
            continue
    return addresses


with closing(socket.socket(socket.AF_INET, socket.SOCK_RAW,
     socket.IPPROTO_ICMP)) as ping_sock:
    addresses = ip_utils.ip_range("192.168.1.0/24")
    local_ip = ip_utils.get_local_ip()
    if addresses is not None:
        addresses_to_scan = filter(lambda x: x!=local_ip, addresses)
    else:
        print("error with ip range specification")
        exit()
    p = Pool(1)
    ID = getpid()&0xFFFF
    replied = p.apply_async(recieved_ping_from_addresses, (ID, 2))
    for address in zip(addresses_to_scan, repeat(1)):
        try:
            packet = ip_utils.make_icmp_packet(ID)
            ping_sock.sendto(packet, address)
        except PermissionError:
            pass
    p.close()
    p.join()
    hosts_up = replied.get()
    print("\n".join(map(lambda x: f"host: [{x[0]}]\tresponded to an ICMP
        ECHO REQUEST in {round_significant_figures(x[1], 2):<10}
        seconds, ttl: [{x[2]}]", hosts_up)))
```

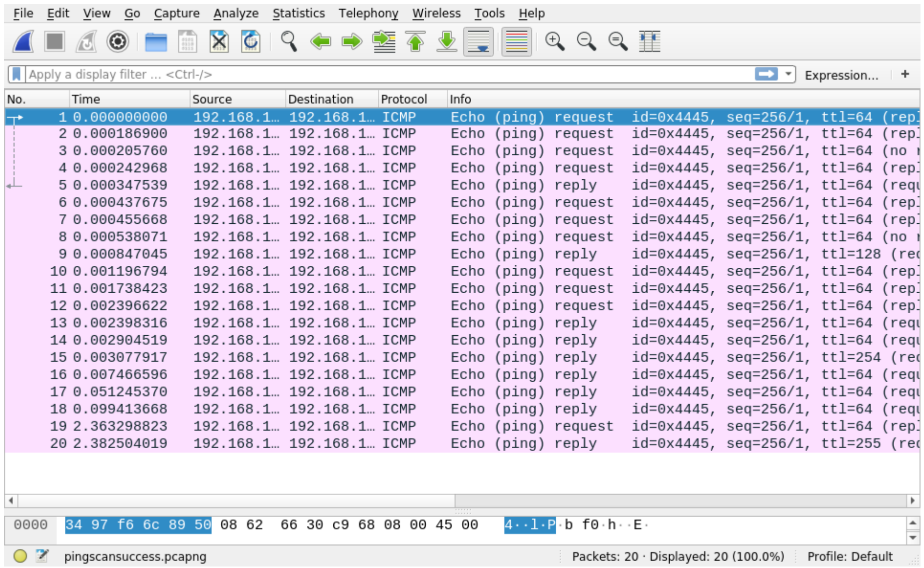


Figure 21: Screenshot of wireshark showing a successful ping scan.

Listing 7: The output of from the ping scanner on the run which generated the PCAP file in figure 21

```
$ sudo ./ping_scan.py
host: [192.168.1.1]  responded to an ICMP ECHO REQUEST in 0.00037
    seconds, ttl: [64]
host: [192.168.1.35] responded to an ICMP ECHO REQUEST in 0.00042
    seconds, ttl: [128]
host: [192.168.1.37] responded to an ICMP ECHO REQUEST in 0.002 seconds,
    ttl: [64]
host: [192.168.1.117] responded to an ICMP ECHO REQUEST in 0.0017
    seconds, ttl: [64]
host: [192.168.1.176] responded to an ICMP ECHO REQUEST in 0.0014
    seconds, ttl: [254]
host: [192.168.1.14] responded to an ICMP ECHO REQUEST in 0.0072
    seconds, ttl: [64]
host: [192.168.1.246] responded to an ICMP ECHO REQUEST in 0.049
    seconds, ttl: [64]
host: [192.168.1.8] responded to an ICMP ECHO REQUEST in 0.099 seconds,
    ttl: [64]
```

Now that I have done these prototypes I am fairly certain about how I will structure the rest of my scanners, how to interact with Python's socket programming interface and how I can use the struct module to make and dissect packets. My general plan for the scanners will be to start a process that listens for responses for a set amount of time and then start sending the packets in a different process before waiting for the listening process to get all the responses back and collecting the results from that process.

1.9 Acceptable Limitations

Originally I had planned to include dedicated operating system detection as an option however I ran out of time having implemented version detection. However it still does Operating system detection partially as some services are linux only and while doing service and version detection especially the Common Platform Enumeration (CPE) parts of the matched service/version will contain operating system information, such as microsoft ActiveSync would indicate that the system being scanned was a windows system which is reflected in the match directive and attached CPE information:

```
match activesync m|^.\0\x01\0[^\0]\0[^\0]\0[^\0]\0[^\0]\0[^\0]\0.*
\0\0\0$|s p/Microsoft ActiveSync/ o/Windows/ cpe:/a:microsoft:acti
vesync/ cpe:/o:microsoft:windows/a
```

1.10 Test Strategy

I am going to use two different methods to test my program:

1. Unit testing
2. Wireshark

I am using two separate testing strategies because they are both good at different things, both of which I need to show that my project works. Firstly I am using unit testing to test some general purpose functions which are pure functions (are independent of the current state of the machine) such as `ip_range()` and other functions which I can just check the returned value against what it should be.

Wireshark is useful for the other half of the program which uses impure functions and the low level networking e.g. `make_tcp_packet()`. Wireshark makes this easy by allowing capture of all the packets going over the wire, as well as this it has a vast array of packet decoders (2231 in my install) which it can use to dissect almost any packet that would be on the network. The main benefit of wireshark is that I can see my scanners sending packets and then check whether the parsers that I have written for the different protocols are working. I can also check that the checksums in each of the various protocols is valid as wireshark does checksum verification for various protocols.

I will be running these tests on my laptop which is a thinkpad T480 running arch linux with kernel version 5.0.7. I am using wireshark version 3.1.0, Python version 3.7.2 and pytest version 4.3.1. I am also using pyenv version 1.2.9 to manage the version of python in my python environment. I am using no modules outside of the python standard library so that my program is as portable as possible and its functionality is as reproducible as possible.

2 Design

2.1 Overall System Design (High Level Overview)

There are two types of scanning implemented for different scan types in my program.

- `Connect()`
- version
- listener / sender

`Connect()` scanning is the simplest in that it takes in a list of ports and simply calls the `socket.connect()` method on it and sees whether it can connect or not and the ports are marked accordingly as open or closed.

Version scanning is very similar to `Connect()` scanning in that it takes in a list of ports and connects to them, except it then sends a probe to the target to elicit a response and gain some information about the service running behind the port.

Listener / sender scanning does exactly what it says on the tin: it sets up a "listener" in another process to listen for responses from the host which the "sender" is sending packets to. It can then differentiate between open, open|filtered, filtered and closed ports based on whether it receives a packet back and what flags (part ofTCP packets are a one byte long section which store "flags" where each bit in the byte represents a different flag) are set in the received packet.

2.2 Design of User Interfaces HCI

I have designed my system to have a similar interface to the most common tool currently used: nmap this is because I believe that having a familiar interface will not only make it easier for someone who is familiar with nmap to use my tool it also makes it so that anything learnt using either tool is applicable to both which benefits everyone.

Based on this perception I have used the same option flags as nmap as well as similar help messages and an identical call signature (how the program is used on the command line). Running `./netscan.py <options> <target_spec>` is identical to `nmap <options> <target_spec>` in terms of which scan types will be run, which hosts will be scanned and which ports are scanned. Below you can see the help message generated by `./netscan.py --help`.

```
usage: netscan.py [-h] [-Pn] [-sL] [-sn] [-sS] [-sT] [-sU] [-sV] [-p PORTS]
                  [--exclude_ports EXCLUDE_PORTS]
                  target_spec

positional arguments:
  target_spec           specify what to scan, i.e. 192.168.1.0/24
```

```
optional arguments:
  -h, --help            show this help message and exit
  -Pn                   assume hosts are up
  -sL                   list targets
  -sn                   disable port scanning
  -sS                   TCP SYN scan
  -sT                   TCP connect scan
  -sU                   UDP scan
  -sV                   version scan
  -p PORTS, --ports PORTS
                        scan specified ports
  --exclude_ports EXCLUDE_PORTS
                        ports to exclude from the scan
```

It shows clearly which are required arguments and which are optional ones, as well as what each argument actually does. It also allows some some arguments to be called with either a short format e.g. `-p` and with a most verbose format `--ports` this allows the user to be clearer if they are using the tool as part of an automated script to perform scanning as it is more immediately obvious what the more verbose flags do.

2.3 System Algorithms

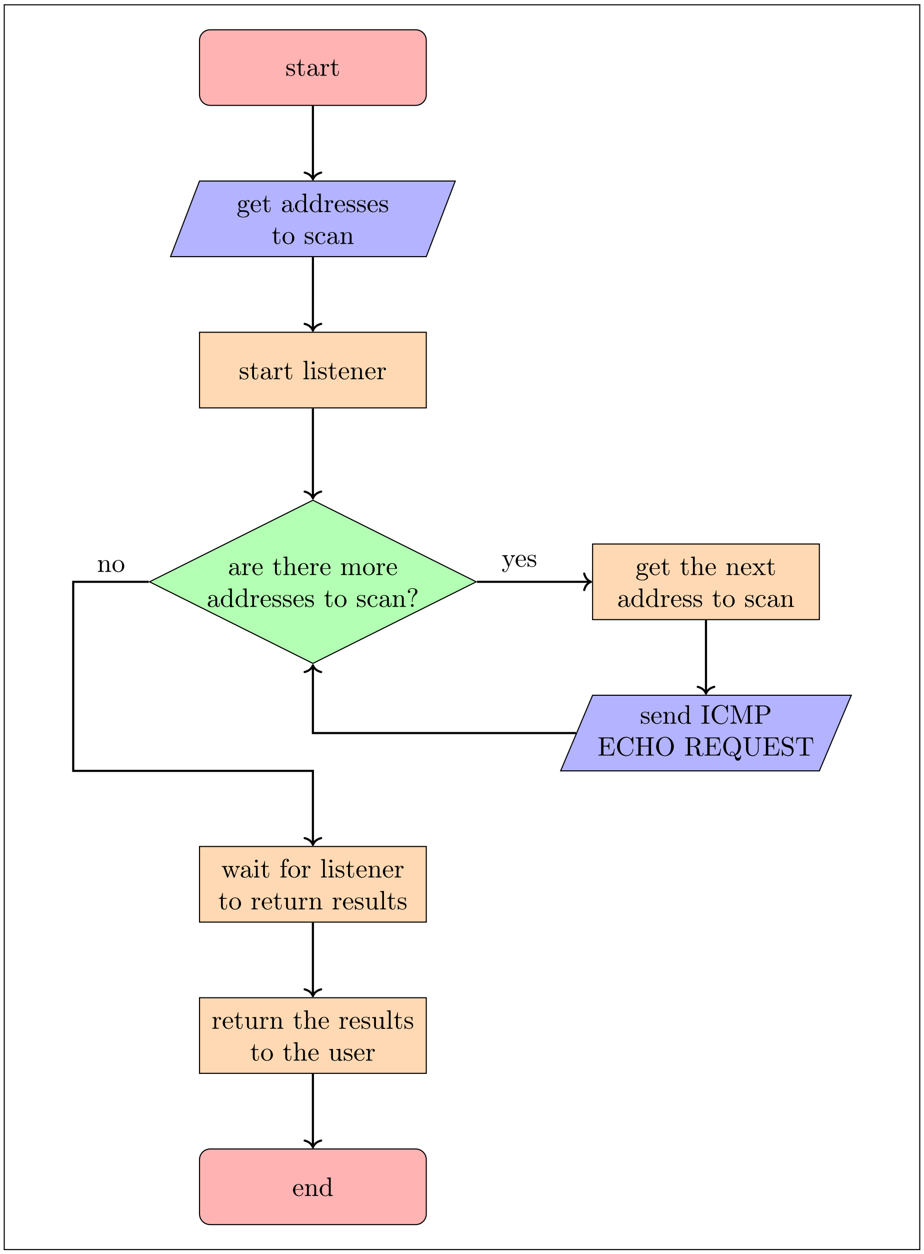


Figure 22: The logic for how I will do Ping Scanning.

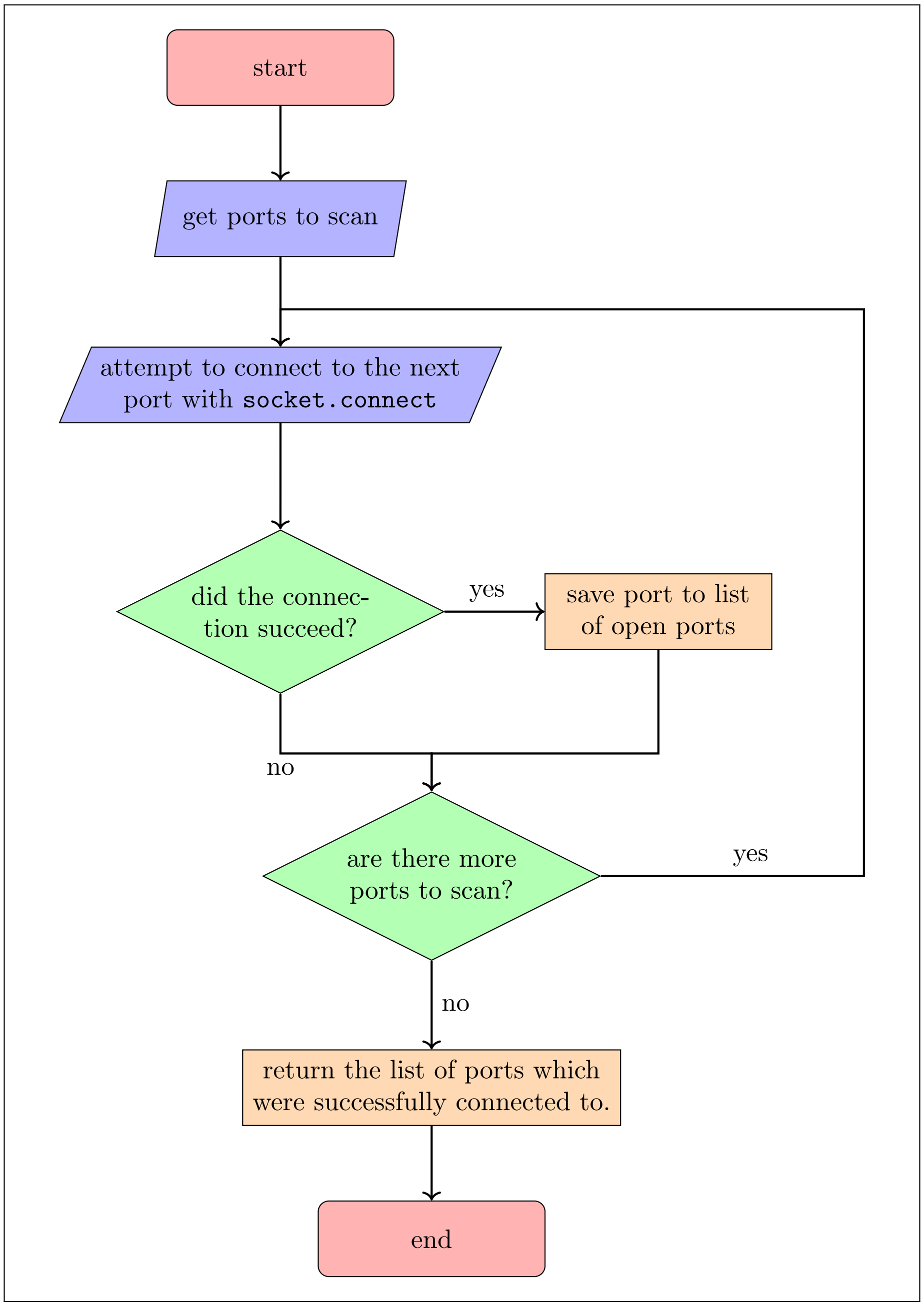


Figure 23: The logic for how I will do TCP connect Scanning.

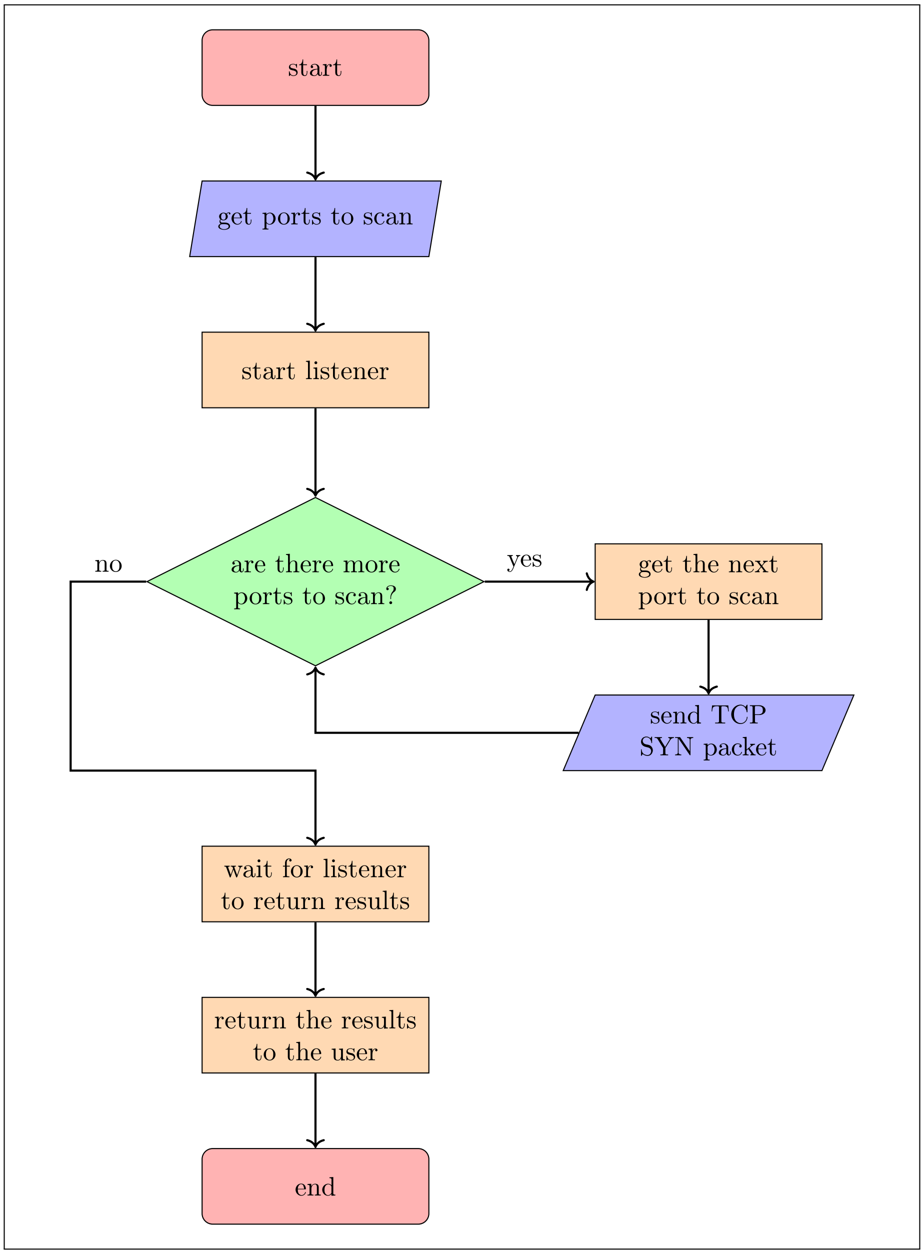


Figure 24: The logic for how I will do TCP SYN scanning.

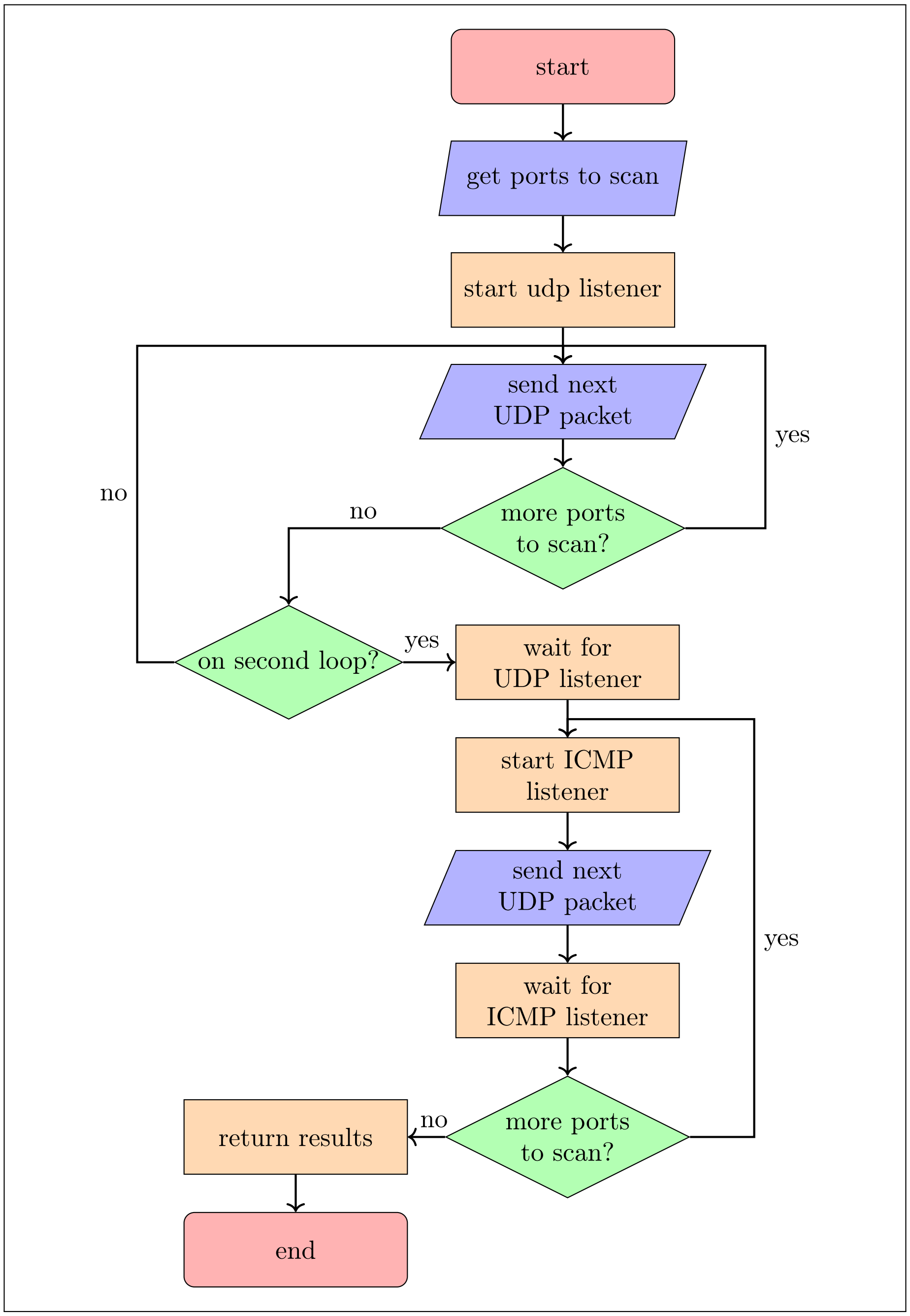


Figure 25: The logic behind how UDP scanning works.

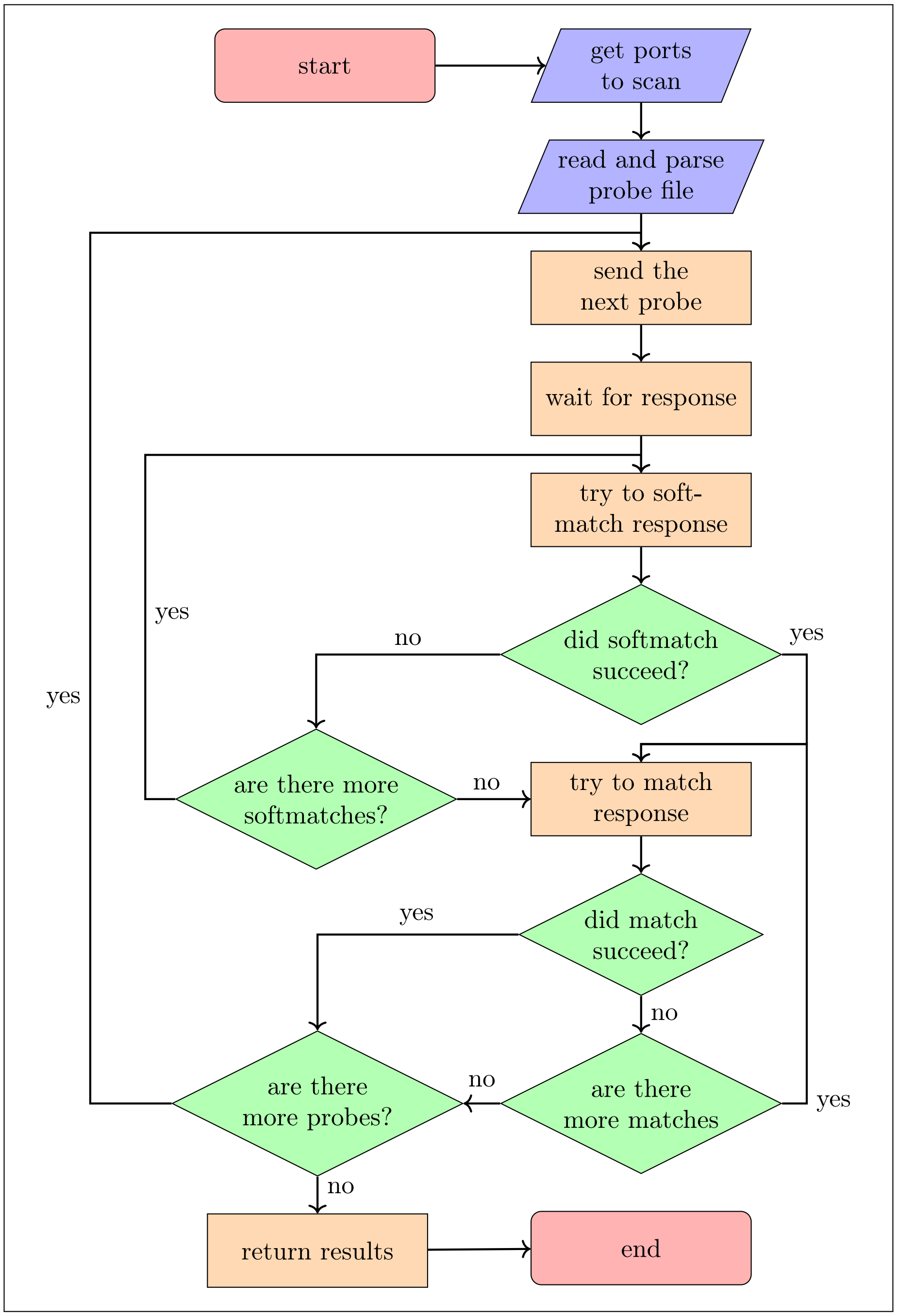


Figure 26: The logic behind how version detection works.

2.4 Input data Validation

I perform data validation in all of the functions in the fundamental modules which I have made. This is because a lot of the functionality of my project is based around the code in these modules so I need to know if I am passing bad input data to a function. So that I know there is a bug that I need to fix. This also helps when the program is interacting with the user if it detects invalid data it will raise a ValueError which is python's error message for when an error is raised because the data passed in has the incorrect value, for example trying to turn the string `"I love apples"` into an integer value will result in the error message: `ValueError: invalid literal for int() with base 10: 'I love apples'` This informs you that you have tried to turn `"I love apples"` into an integer with base 10 and it is invalid. An example from my code would be if I were to ask it to print out a subnet with 33 network bits it will raise a value error as network bits must be between 0 and 32. This is shown in figure 27. In general there are very few pieces of data that the user enters which can be erroneous, the user enters the IP address to scan, the ports to scan and the number of network bits in the case of a subnet. My program has functions which are used for verifying each of these, for the case of port numbers and IP addresses I have explicit functions which do this: `def is_valid_port_number(port_num: int) -> bool:` and `def is_valid_ip(ip: Union[str, int]) -> bool:`. For network bits they are validated inside of the function which generates the range of IP addresses for a subnet by a simple check on whether the number of network bits is between 0 and 32.

```
networkScanner/Code on master [!?] venv:(net_scanner) pyenv:( net_scanner)
→ ./netscan.py -sL 192.168.1.0/33
Traceback (most recent call last):
  File "./netscan.py", line 88, in <module>
    int(network_bits)
  File "/home/tritoke/school/CS/networkScanner/Code/modules/ip_utils.py", line 132, in ip_range
    raise ValueError(f"Invalid number of network bits: [{network_bits}]")
ValueError: Invalid number of network bits: [33]
```

Figure 27: A screenshot of the output of my code when asked to translate a subnet with 33 network bits.

2.5 Algorithm for complex structures

Algorithm 1 My algorithm for turning a CIDR specified subnet into a list of actual IP addresses

```
procedure IP_RANGE
    network_bits ← number of network bits specified
    ip ← base IP address
    mask ← 0
    for maskbit ← (32 − network_bits), 31 do
        mask ← mask + 2^maskbit
    lower_bound ← ip AND mask                    ▷ zero the last 32-network_bits
    upper_bound ← ip OR (mask XOR 0xFFFFFFFF)     ▷ turn the last 32-network_bits to ones
    addresses ← empty list
    for address ← lower_bound, upper_bound do
        append CONVERT_TO_DOT(address) to addresses
  return addresses
```

Algorithm 2 My algorithm for pretty-printing a dictionary of lists ofportnumbers such that ranges are specified as start-end instead of start,start+1,. . . ,end

```
procedure COLLAPSE
    port_dictionary ← dictionary of lists ofportnumbers
    key_results ← empty list          ▷ stores the formatted result for each key
    for key in port_dictionary do
        ports ← port_dict[key]
        result ← key + ":{"
        if ports is empty then
            new_sequence ← FALSE
            for index ← 1, (length of ports) − 1 do
                port = ports[index]
                if index = 0 then
                    result ← result + ports[0]        ▷ append the first element
                    if ports[index+1] = port + 1 then
                        result ← result + "-"         ▷ begin a new sequence
                    else
                        result ← result + ","         ▷ not a sequence
                else if port + 1 ≠ ports[index+1] then   ▷ break in sequence
                    result ← result + port + ","
                    new_sequence ← TRUE
                else if port + 1 = ports[index+1] & new_sequence then
                    result ← result + "-"
                    new_sequence ← FALSE
            result ← result + ports[(length of ports)-1] + "}"
            append result to key_results
  return "{" + (key_results separated by ", ") + "}"
```

3 Technical Solution

I have placed all of my code in Appendix A. This is because it takes up 60 odd pages and as such it would make it a pain to have to page through to get to the testing section. However I will be going through each of the items in this appendix and explaining what they do.

Appendix A.1 contains all the code which I wrote while in an early experimentation phase where I was testing out how I was planning to make and structure the project.

Appendix A.2 contains all the code which I wrote while writing my initial prototype of my ping scanner which uses ICMP ECHO REQUEST messages to detect hosts which are online on a given subnet. It is used to meet success criteria 4.

Appendix A.3 contains all the code which I wrote while writing a tool which can translate a CIDR specified subnet into the list of IP addresses for that subnet, it has logic to exclude the broadcast address and host addresses for each subnet. This is used to meet success criteria 3.

Appendix A.4 contains all of the prototypes for TCP based scanning which are contained in the sub Appendices A.4.1 and A.4.2. Appendix A.4.1 contains all of the code which I created whilst prototyping connect scanning. It satisfies success criteria 6 and 7. Appendix A.4.2 contains all of the code I wrote while prototyping TCP SYN scanning. It satisfies success criteria 6, 7 and 8.

Appendix A.5 contains all of the code I wrote while prototyping UDP scanning. It satisfies success criteria 9, 10 and 11.

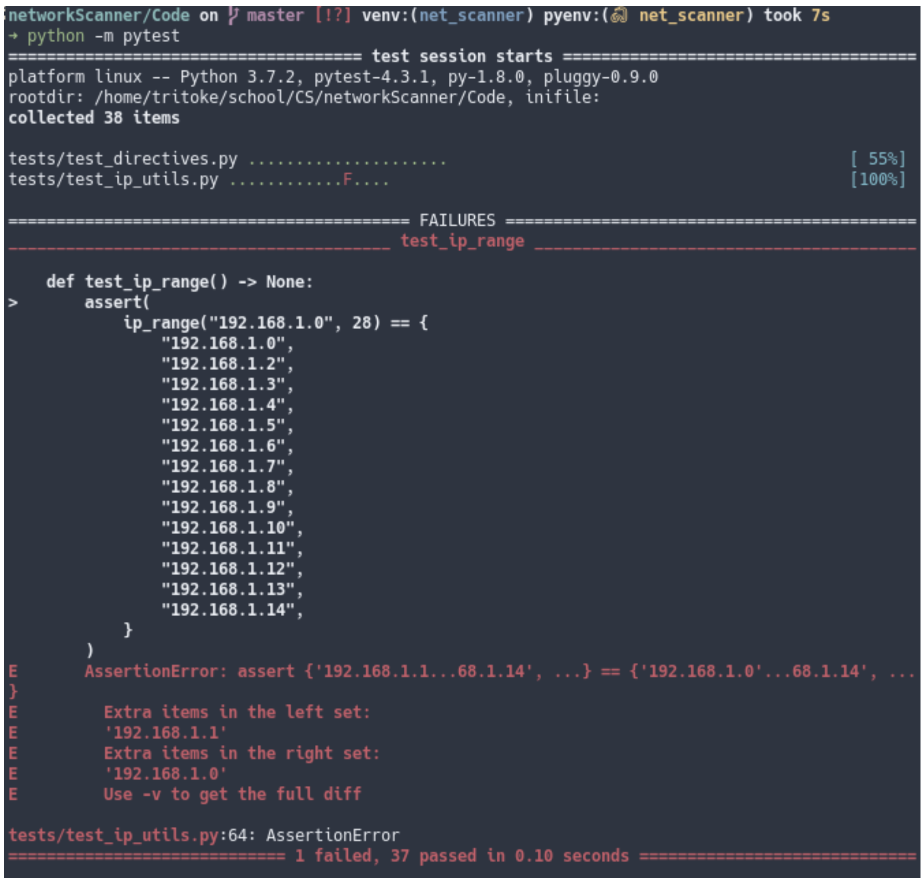
Appendix A.6 contains all of the code I wrote while prototyping version detection scanning. It satisfies success criteria 13 and 14.

Appendix A.7 contains all of the modules I wrote to help me make me with creating my main application which I will come on to later. These modules mainly contain code which I reuse often such as code to calculate an ip checksum or validate an IP address.

Appendix A.8 contains a script I wrote which will run each of the prototype applications I made. This doesn't satisfy any of the success criteria but was very useful for solving issues I had with importing python modules where due to the directory structure everything as to do started from the root of the directory structure otherwise everything goes a bit mad, and this was my solution for running everything at the root of the directory structure as this sits at the root and can call the `main()` function defined in each of the modules along with also being able to import all of the modules in the modules directory.

Appendix A.9 contains the code for my final application which satisfies all of the success criteria bar 12 which is partially completed via version detect scanning.

Appendix A.10 contains all of the for my unit tests which I run using `python -m pytest` and it automatically goes and runs each function and can give me verbose information on each one. I have named all of the test functions in a very verbose way and I only test one thing in each function. This means that it is much easier for me to read from the name of a failed test exactly what went wrong with what function and what argument caused it. An example of this can be seen in figure 28 where I have changed on of the tests so that it fails. You can see in that it shows me a clear difference between what was expected on one side of the assertion statement and then what actually happened on the other side. In this case it shows that in the left set there is an extra element of 192.168.1.1 and in the right an extra element of 192.168.1.0, this is very helpful for preventing regressions in the code where I would write feature and accidentally break another piece of functionality.



```
networkScanner/Code on master [!?] venv:(net_scanner) pyenv:( net_scanner) took 7s
→ python -m pytest
==================================== test session starts ====================================
platform linux -- Python 3.7.2, pytest-4.3.1, py-1.8.0, pluggy-0.9.0
rootdir: /home/tritoke/school/CS/networkScanner/Code, inifile:
collected 38 items

tests/test_directives.py .....................                                        [ 55%]
tests/test_ip_utils.py ............F....                                              [100%]

========================================== FAILURES ==========================================
_______________________________________ test_ip_range _______________________________________

    def test_ip_range() -> None:
>       assert(
            ip_range("192.168.1.0", 28) == {
                "192.168.1.0",
                "192.168.1.2",
                "192.168.1.3",
                "192.168.1.4",
                "192.168.1.5",
                "192.168.1.6",
                "192.168.1.7",
                "192.168.1.8",
                "192.168.1.9",
                "192.168.1.10",
                "192.168.1.11",
                "192.168.1.12",
                "192.168.1.13",
                "192.168.1.14",
            }
        )
E       AssertionError: assert {'192.168.1.1...68.1.14', ...} == {'192.168.1.0'...68.1.14', ...
}
E         Extra items in the left set:
E         '192.168.1.1'
E         Extra items in the right set:
E         '192.168.1.0'
E         Use -v to get the full diff

tests/test_ip_utils.py:64: AssertionError
============================ 1 failed, 37 passed in 0.10 seconds ============================
```

Figure 28: A screenshot of running pytest with a deliberately broken test.

4 Testing

4.1 Test Plan

I will be testing my application using a combination of unit tests and wireshark where applicable. Unit tests are more suitable to doing tests on specific functions to make sure that regressions don't occur while developing the application. A regression is a when a feature or change that was implemented into the program is by accident and would cause the application to break. Wireshark I will use to show the scanning portion of my code and where external connections are made/custom packets created. The following are the tests using wireshark, the unit tests are in Appendix A.10.

4.2 Testing Evidence

4.2.1 Printing a usage message when run without parameters

To show this I will run my program passing it no parameters. This should print out a message of the form: USAGE: ./<program> <required> <parameters> where everything in angle brackets should be replaced by what is necessary for my program. In figure 29 you can see me run ./netscan.py with no parameters and it prints out the required usage message telling me that I am missing the target_spec parameter, this shows that it passed this test. This shows success criteria 2.

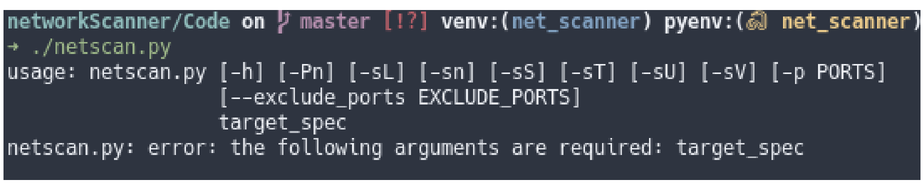


Figure 29: Screenshot showing my program being run without parameters.

4.2.2 Printing a help message when passed -h

To show this I will run my program with the -h flag. This should print out a message showing each of the options as well as what each of them do. It should also print out whether they are positional arguments or optional arguments and if an argument can have two forms then it should print out both forms of the flag, i.e. -p --ports. In figure 30 you can see me run my program with the -h flag and it proceeds to print of a help message with messages with what each option is for as well as short and long form of arguments, this shows my program passed this test. This shows success criteria 2.

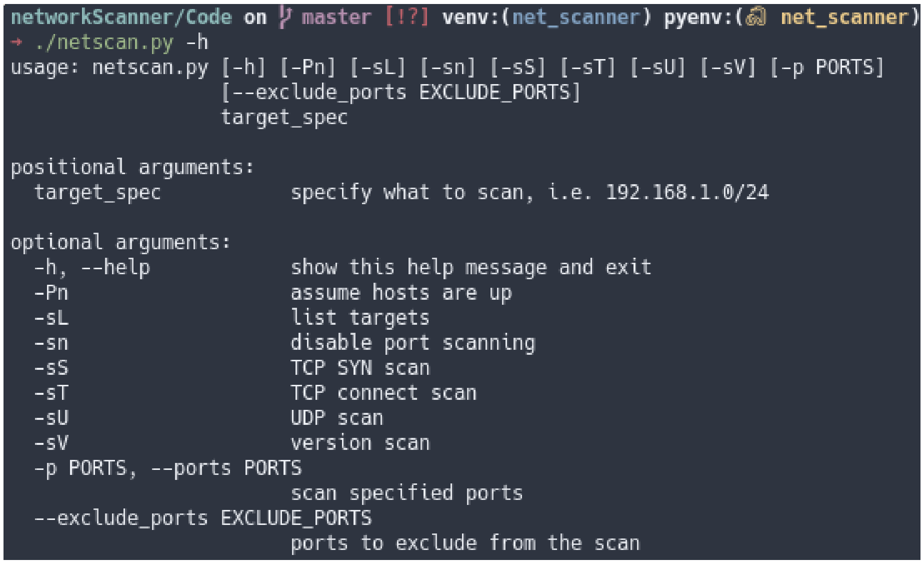


Figure 30: Screenshot showing my program being run with the -h flag.

4.2.3 Printing a help message when passed -help

To show this I will run my program with the `--help` flag. This should produce the exact same output as with `-h`. This shows the exact same message as in the test of `-h`. To prove this if I take the sha1sum of the output for both flags we can see that the hashes are identical and therefore the originals were also identical, this is shown in figure 32. This shows success criteria 2.

```
networkScanner/Code on  master [!?] venv:(net_scanner) pyenv:( net_scanner)
➜ ./netscan.py --help
usage: netscan.py [-h] [-Pn] [-sL] [-sn] [-sS] [-sT] [-sU] [-sV] [-p PORTS]
                  [--exclude_ports EXCLUDE_PORTS]
                  target_spec

positional arguments:
  target_spec           specify what to scan, i.e. 192.168.1.0/24

optional arguments:
  -h, --help            show this help message and exit
  -Pn                   assume hosts are up
  -sL                   list targets
  -sn                   disable port scanning
  -sS                   TCP SYN scan
  -sT                   TCP connect scan
  -sU                   UDP scan
  -sV                   version scan
  -p PORTS, --ports PORTS
                        scan specified ports
  --exclude_ports EXCLUDE_PORTS
                        ports to exclude from the scan
```

Figure 31: Screenshot showing my program being run with the help flag.

```
networkScanner/Code on  master [!?] venv:(net_scanner) pyenv:( net_scanner)
➜ ./netscan.py --help | sha1sum
47b52bb86cbe927e2c1a9521e4c55c2369f541a5  -

networkScanner/Code on  master [!?] venv:(net_scanner) pyenv:( net_scanner)
➜ ./netscan.py -h | sha1sum
47b52bb86cbe927e2c1a9521e4c55c2369f541a5  -
```

Figure 32: Screenshot showing the hashes of the two help messages.

4.2.4 Scanning a subnet with ICMP ECHO REQUEST messages

To show this I will run my program with the `-sn` flag and specify the subnet of my local network `192.168.178.0/24`. This should produce a list of all the hosts which are up on the network. In figure 33 you can see you can see my program's output showing that the hosts:

- 192.168.178.60
- 192.168.178.56
- 192.168.178.30
- 192.168.178.1

all responded with ICMP ECHO REPLY messages, this is reflected in a packet capture I took while performing the scan. A section of this scan is shown in figure 34 where you can see some of ICMP ECHO REQUEST messages my program sent, along with some of the requests to hosts that don't exist, note the different addresses in the source and destination fields and the Echo (ping) request vs reply in the info column. This successfully shows success criteria 1 and 4.

```
networkScanner/Code on  master [!?] venv:(net_scanner) pyenv:( net_scanner) took 13s
→ sudo ./netscan.py 192.168.178.0/24 -sn
host: [192.168.178.60]  responded to an ICMP ECHO REQUEST in 0.00011s   ttl: [64]
host: [192.168.178.56]  responded to an ICMP ECHO REQUEST in 0.28s      ttl: [64]
host: [192.168.178.30]  responded to an ICMP ECHO REQUEST in 0.027s     ttl: [64]
host: [192.168.178.1]   responded to an ICMP ECHO REQUEST in 0.031s     ttl: [64]
```

Figure 33: Screenshot showing the output of a scan of my local network.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	192.168.178.60	192.168.178.30	ICMP	234	Echo (ping) request
2	0.000749915	192.168.178.60	192.168.178.56	ICMP	234	Echo (ping) request
3	0.004504662	192.168.178.60	192.168.178.20	ICMP	234	Echo (ping) request
4	0.004830456	192.168.178.60	192.168.178.48	ICMP	234	Echo (ping) request
5	0.005289695	192.168.178.60	192.168.178.1	ICMP	234	Echo (ping) request
6	0.026946346	192.168.178.30	192.168.178.60	ICMP	234	Echo (ping) reply
7	0.036125893	192.168.178.1	192.168.178.60	ICMP	234	Echo (ping) reply
8	0.281829344	192.168.178.56	192.168.178.60	ICMP	234	Echo (ping) reply
9	0.282171289	192.168.178.60	192.168.178.51	ICMP	234	Echo (ping) request
10	2.329937472	192.168.178.60	192.168.178.21	ICMP	234	Echo (ping) request
11	2.330018351	192.168.178.60	192.168.178.35	ICMP	234	Echo (ping) request

Figure 34: Screenshot showing a selection of the packets being sent by this scan.

4.2.5 Translating a CIDR specified subnet into a list of IP addresses

To show this I will run my program with the `-sL` flag and I will specify a small subnet of `192.168.1.0/28` (I have chosen such a small subnet such that it will fit on my terminal and therefore in a screenshot). I expect the list of addresses to be `192.168.1.1 - 192.168.1.14`. To prove that my program works I will screenshot the output when run with the stated parameters and I will use a website to translate the same subnet and show that it displays the same addresses as my program. In figure 35 you can see that the output from my program matches the expected list of IP addresses from `192.168.1.1` to `192.168.1.14` which is also shown by the screen shot of the same subnet translated by the ipcalc utility on linux. This proves my program works and covers success criteria 3.

```
networkScanner/Code on  master [✘!?] venv:(net_scanner) pyenv:( net_scanner)
→ ./netscan.py -sL 192.168.1.0/28
Targets:
192.168.1.1
192.168.1.2
192.168.1.3
192.168.1.4
192.168.1.5
192.168.1.6
192.168.1.7
192.168.1.8
192.168.1.9
192.168.1.10
192.168.1.11
192.168.1.12
192.168.1.13
192.168.1.14
```

Figure 35: Screenshot showing the output of my program when asked to translate the subnet 192.168.1.0/28.

```
networkScanner/Code on  master [!?] venv:(net_scanner) pyenv:( net_scanner)
→ ipcalc 192.168.1.0/28
Address:   192.168.1.0          11000000.10101000.00000001.0000 0000
Netmask:   255.255.255.240 = 28 11111111.11111111.11111111.1111 0000
Wildcard:  0.0.0.15             00000000.00000000.00000000.0000 1111
=>
Network:   192.168.1.0/28       11000000.10101000.00000001.0000 0000
HostMin:   192.168.1.1          11000000.10101000.00000001.0000 0001
HostMax:   192.168.1.14         11000000.10101000.00000001.0000 1110
Broadcast: 192.168.1.15         11000000.10101000.00000001.0000 1111
Hosts/Net: 14                    Class C, Private Internet
```

Figure 36: Screenshot showing the range displayed by the ipcalc utility when asked to calculate the same subnet.

4.2.6 Scanning without first checking whether hosts are up.

To show this I will perform a TCP scan on a small subnet where I know there are no hosts and show that the scan continues despite there actually being no host on the other end. To do this I will pass the `-Pn` flag and I will specify the subnet `192.168.43.0/28` which I know has no has no hosts on it. I will also specify `-p 12345` to only scan port 12345 so that there are fewer requests in the packet capture. Finally I will specify `-sS` to do TCP SYN SCANNING. I expect to see a multiple of 14 Address Resolution Protocol (ARP) messages. This is because I don't know how many times my NIC will retry at getting the destination Media Access Control (MAC) address. It needs to destination MAC address to send the packet to its destination as we are scanning a private IP range of my router. In figure 37 you can see the output of my program when run with the specified flags, you can see that as expected it showed that there were no open ports on those machines as they don't exist. In figure 38 you can see the packet capture of the packets my code sent, however there are only ARP messages, this is because we are scanning in the private IP range of my router which was the only way I could guarantee that there was no machine at the other end. However this is as expected, as well as this we can see 42 ARP requests, which is 3×14 ARP requests, which would indicate each scan made three ARP requests before giving up. This shows my program can perform scans without first checking if the host is up, showing success criteria 5.

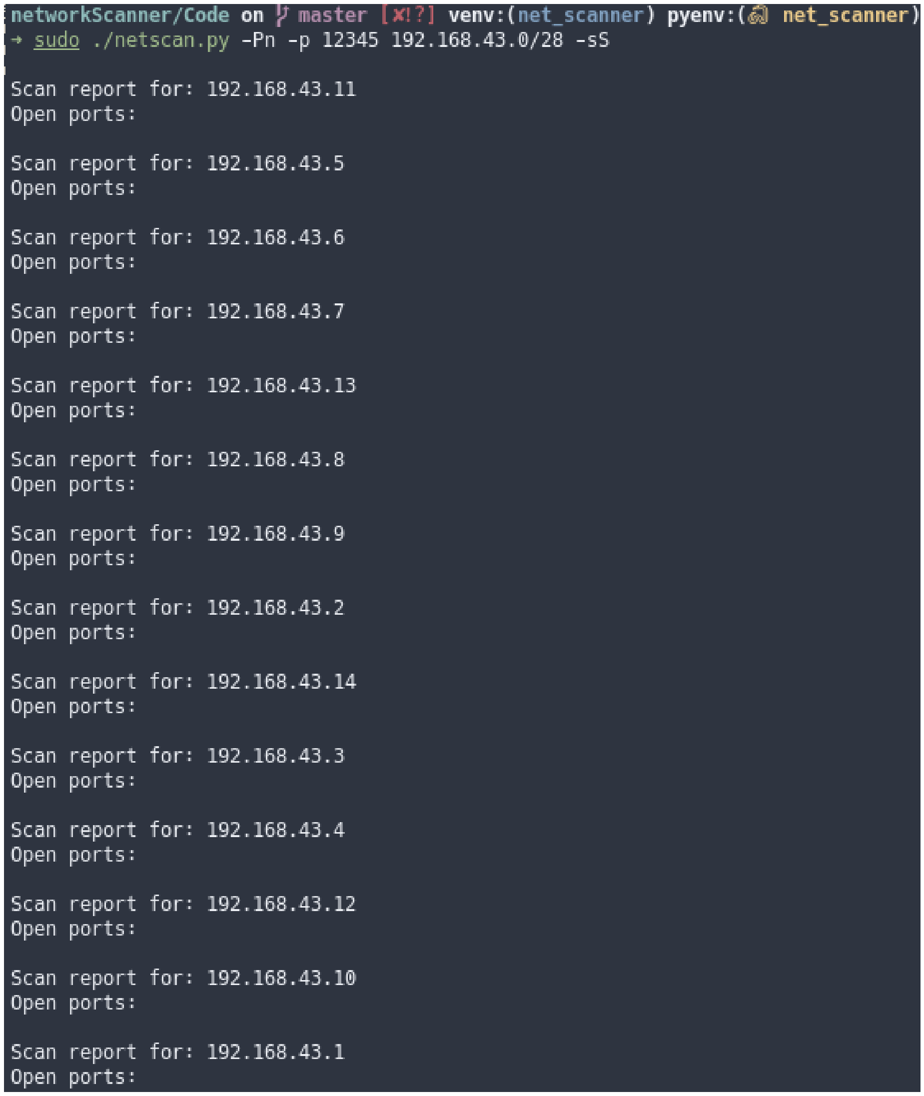
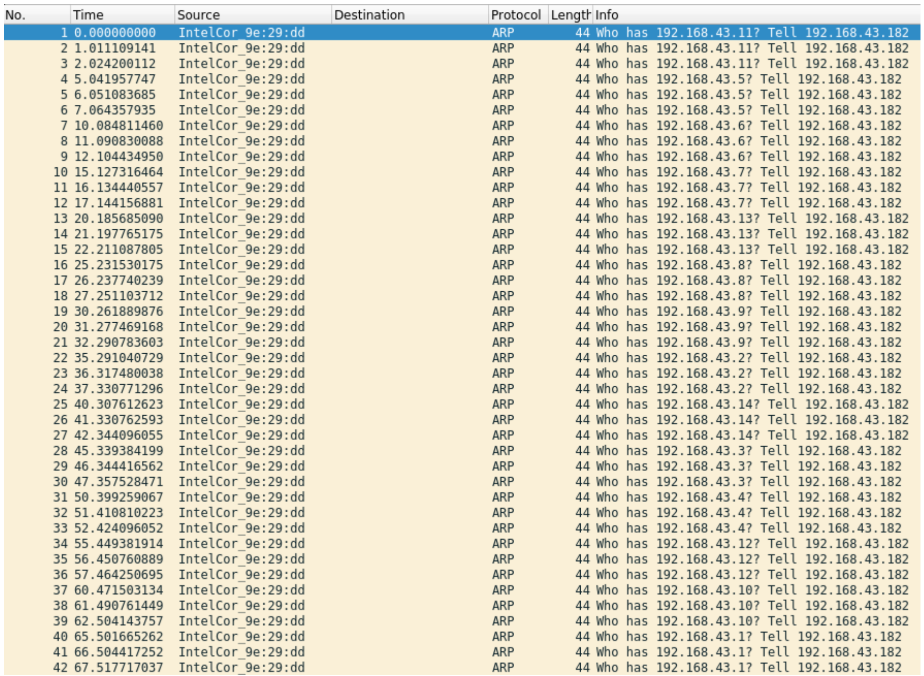


Figure 37: Screenshot showing the output from my code when asked to port scan a subnet with no machines behind the addresses.

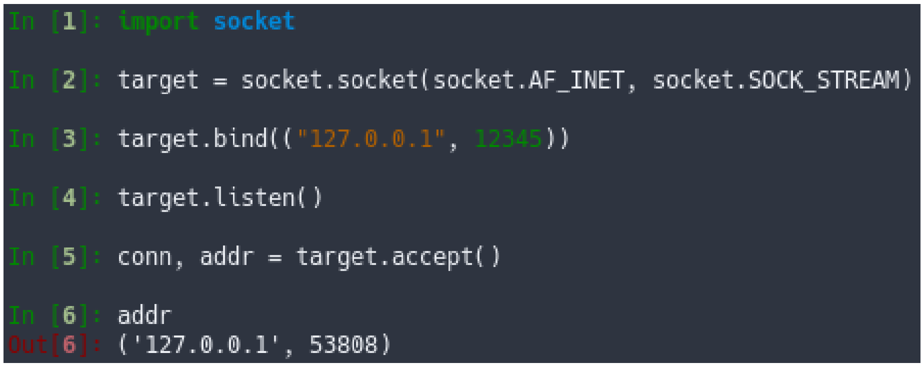


No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.11? Tell 192.168.43.182
2	1.011109141	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.11? Tell 192.168.43.182
3	2.024200112	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.11? Tell 192.168.43.182
4	5.041957747	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.5? Tell 192.168.43.182
5	6.051083685	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.5? Tell 192.168.43.182
6	7.064357935	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.5? Tell 192.168.43.182
7	10.084811460	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.6? Tell 192.168.43.182
8	11.090830088	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.6? Tell 192.168.43.182
9	12.104434950	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.6? Tell 192.168.43.182
10	15.127316464	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.7? Tell 192.168.43.182
11	16.134440557	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.7? Tell 192.168.43.182
12	17.144156881	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.7? Tell 192.168.43.182
13	20.185685090	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.13? Tell 192.168.43.182
14	21.197765175	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.13? Tell 192.168.43.182
15	22.211087805	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.13? Tell 192.168.43.182
16	25.231530175	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.8? Tell 192.168.43.182
17	26.237740239	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.8? Tell 192.168.43.182
18	27.251103712	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.8? Tell 192.168.43.182
19	30.261889876	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.9? Tell 192.168.43.182
20	31.277469168	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.9? Tell 192.168.43.182
21	32.290783603	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.9? Tell 192.168.43.182
22	35.291040729	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.2? Tell 192.168.43.182
23	36.317480038	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.2? Tell 192.168.43.182
24	37.330771296	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.2? Tell 192.168.43.182
25	40.307612623	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.14? Tell 192.168.43.182
26	41.330762593	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.14? Tell 192.168.43.182
27	42.344096055	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.14? Tell 192.168.43.182
28	45.339384199	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.3? Tell 192.168.43.182
29	46.344416562	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.3? Tell 192.168.43.182
30	47.357528471	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.3? Tell 192.168.43.182
31	50.399259067	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.4? Tell 192.168.43.182
32	51.410810223	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.4? Tell 192.168.43.182
33	52.424096052	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.4? Tell 192.168.43.182
34	55.449381914	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.12? Tell 192.168.43.182
35	56.450760889	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.12? Tell 192.168.43.182
36	57.464250695	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.12? Tell 192.168.43.182
37	60.471503134	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.10? Tell 192.168.43.182
38	61.490761449	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.10? Tell 192.168.43.182
39	62.504143757	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.10? Tell 192.168.43.182
40	65.501665262	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.1? Tell 192.168.43.182
41	66.504417252	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.1? Tell 192.168.43.182
42	67.517717037	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.1? Tell 192.168.43.182

Figure 38: Screenshot showing the ARP requests my NIC sent to attempt to determine where to send the attempted connection packets.

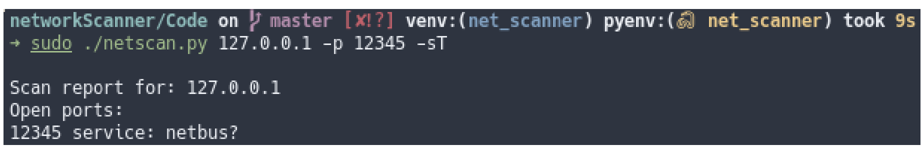
4.2.7 Detecting whether a TCP port is open

To show this I will perform a TCP `Connect()` scan on my local machine while running a script which will listen on port 12345 for any connections and send back a message. To do this I will pass my program the flags `-sT` and `-p 12345` as well as specifying localhost to scan (127.0.0.1). I expect to see a TCP SYN-ACK handshake between my program and the script and then my program to output that the port is open. In figure 41 you can see the expected TCP SYN-ACK handshake performed by my program and the script in figure 39. You can see the output of my program in figure 40, as expected it outputs that port 12345 is open. This shows success criteria 1 and 6.



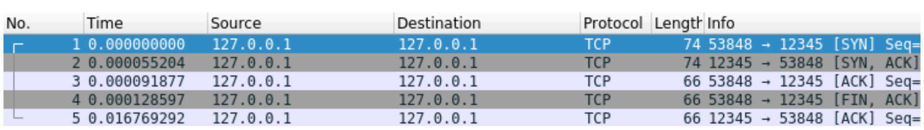
```
In [1]: import socket

In [2]: target = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

In [3]: target.bind(("127.0.0.1", 12345))

In [4]: target.listen()

In [5]: conn, addr = target.accept()

In [6]: addr
Out[6]: ('127.0.0.1', 53808)
```

Figure 39: Screenshot showing the script I ran to accept a connection on localhost port 12345.



```
networkScanner/Code on master [✘!?] venv:(net_scanner) pyenv:( net_scanner) took 9s
→ sudo ./netscan.py 127.0.0.1 -p 12345 -sT

Scan report for: 127.0.0.1
Open ports:
12345 service: netbus?
```

Figure 40: Screenshot showing the output of my script when run with the specified flags and while the script in figure 39 was running.



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	127.0.0.1	127.0.0.1	TCP	74	53848 → 12345 [SYN] Seq=
2	0.000055204	127.0.0.1	127.0.0.1	TCP	74	12345 → 53848 [SYN, ACK]
3	0.000091877	127.0.0.1	127.0.0.1	TCP	66	53848 → 12345 [ACK] Seq=
4	0.000128597	127.0.0.1	127.0.0.1	TCP	66	53848 → 12345 [FIN, ACK]
5	0.016769292	127.0.0.1	127.0.0.1	TCP	66	12345 → 53848 [ACK] Seq=

Figure 41: Screenshot showing the packet capture of the TCP SYN-ACK handshake performed by the scan in figure 40 with the script in 39.

4.2.8 Detecting whether a TCP port is closed

To show this I will perform a TCP `Connect()` scan on my local machine except instead of running a script to catch the request I will just let it try to connect to the closed port. I expect to see a TCP SYN packet sent to the port and then a RST, ACK packet sent back, my program should output no open ports. To do this I will pass my program the same options as in the test for a TCP open port. In figure 43 you can see the attempted connection to 127.0.0.1 port 12345 along with the RST, ACK packet afterwards indicating the port is closed. This is reflected in figure 42 with no open ports showing success criteria 1 and 7.

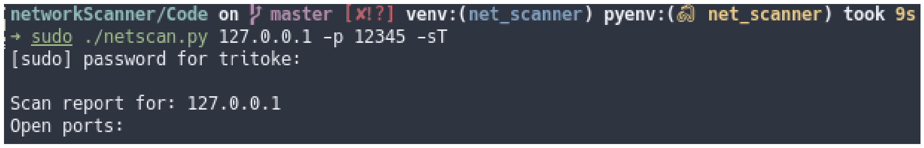


Figure 42: Screenshot showing the output of my program when run with the specified options.

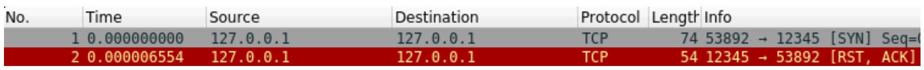


Figure 43: Screenshot showing the packet capture of the TCP SYN-RST closed port indication caused by the scan in figure 42.

4.2.9 Detecting whether a TCP port is filtered

To show this I will perform a TCP SYN scan on localhost port 12345 except I will also introduce a firewall rule to drop all requests to localhost. I expect this to produce no response to the initial SYN packet sent by my program and my program to output that port as filtered. To test this I will run my program with the flags `-sS,-p 12345,-Pn` this will cause it to not check whether the host is up, to perform a TCP SYN scan and only scan port 12345. I will also introduce a firewall rule using the linux iptables utility to drop all requests to localhost as so: `iptables -I INPUT -s 127.0.0.1 -j DROP`. The output of my program is shown in figure 44 you can see that port 12345 is displayed as filtered and in the packet capture shown in figure 45 you can see that there is no response to our initial packet which corresponds to what I thought would happen with an iptables rule in place to drop packets. This shows success criteria 1 and 8.

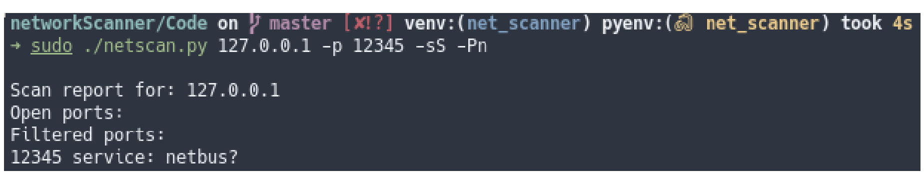


Figure 44: Screenshot showing the output of my program when run with the specified options and a firewall in place to drop all packets to 127.0.0.1.

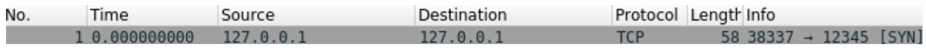


Figure 45: Screenshot showing the packet capture of the scan in figure 44

4.2.10 Detecting whether a UDP port is open

To show this I will perform a UDP scan on a script I have already written while developing UDP scanning which can be seen in listing 8. I expect to see my program output port 12345 as open and in the packet capture I expect to see two UDP packets followed by two response UDP packets from my listener program. I will test this using the following flags: `-Pn,-p 12345,-sU` these translate to scanning port 12345 over UDP and not checking the host is up beforehand. In figure 46 you can see the output of my program when run as specified and you can see that it correctly detects port 12345 as being open. In figure 47 you can see the packet capture of my program being run however this is not as I expected, I didn't foresee the ICMP destination unreachable messages, these are sent by the kernel in response to the UDP probe which it doesn't know what to do with, however apart from those the capture shows everything as expected. This shows success criteria 1 and 9.

Listing 8: Script to open port 12345 to UDP.

```
import socket
from contextlib import closing

with closing(
        socket.socket(
            socket.AF_INET,
            socket.SOCK_DGRAM
        )
) as s:
    s.bind(("127.0.0.1", 12345))
    print("opened port 12345 on localhost")
    while True:
        data, addr = s.recvfrom(1024)
        s.sendto(bytes("Well hello there good sir.", "utf-8"), addr)
```

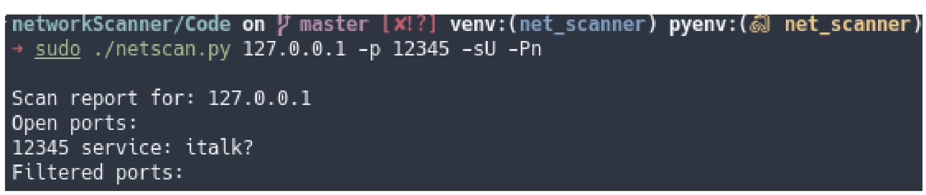
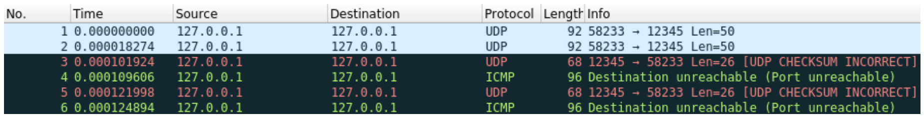


Figure 46: Screenshot showing the output of my program when run with the options specified above, and the script in listing 8 is running.

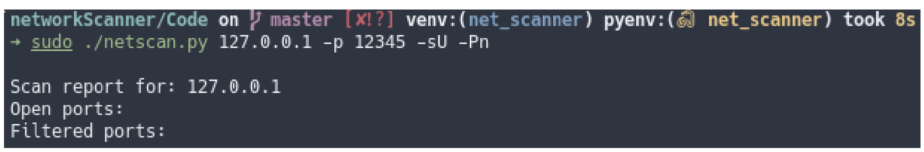


No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	127.0.0.1	127.0.0.1	UDP	92	58233 → 12345 Len=50
2	0.000018274	127.0.0.1	127.0.0.1	UDP	92	58233 → 12345 Len=50
3	0.000101924	127.0.0.1	127.0.0.1	UDP	68	12345 → 58233 Len=26 [UDP CHECKSUM INCORRECT]
4	0.000109606	127.0.0.1	127.0.0.1	ICMP	96	Destination unreachable (Port unreachable)
5	0.000121998	127.0.0.1	127.0.0.1	UDP	68	12345 → 58233 Len=26 [UDP CHECKSUM INCORRECT]
6	0.000124894	127.0.0.1	127.0.0.1	ICMP	96	Destination unreachable (Port unreachable)

Figure 47: screenshot showing the packet capture of the scan in figure 46

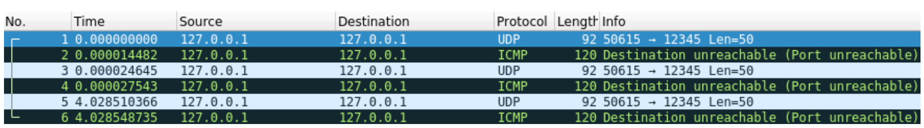
4.2.11 Detecting whether a UDP port is closed

To show this I will perform a UDP scan on a port which has no service listening behind it. I expect my program to print out no filtered ports and no open ports showing that the port was closed. In the packet capture I expect to see three UDP packets and three response ICMP packets. To test this I will use my program with the following flags: `-p 12345,-Pn,-sU` which perform a UDP port scan without first checking if the host is up. In figure 48 you can see the output of my program when run with the options specified above, you can see that there are no ports displayed as either open or filtered, this shows the my program successfully marked the port as closed. This shows success criteria 1 and 10.



```
networkScanner/Code on master [✘!?] venv:(net_scanner) pyenv:( net_scanner) took 8s
➜ sudo ./netscan.py 127.0.0.1 -p 12345 -sU -Pn

Scan report for: 127.0.0.1
Open ports:
Filtered ports:
```

Figure 48: screenshot showing the output of my program when scanning with the options specified above.



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	127.0.0.1	127.0.0.1	UDP	92	50615 → 12345 Len=50
2	0.000014482	127.0.0.1	127.0.0.1	ICMP	120	Destination unreachable (Port unreachable)
3	0.000024645	127.0.0.1	127.0.0.1	UDP	92	50615 → 12345 Len=50
4	0.000027543	127.0.0.1	127.0.0.1	ICMP	120	Destination unreachable (Port unreachable)
5	4.028510366	127.0.0.1	127.0.0.1	UDP	92	50615 → 12345 Len=50
6	4.028548735	127.0.0.1	127.0.0.1	ICMP	120	Destination unreachable (Port unreachable)

Figure 49: screenshot showing the packet capture of the scan in figure 48

4.2.12 Detecting whether a UDP port is filtered

To show this I will use my program to perform a UDP scan on my local machine with a firewall rule to drop any ports sent to the localhost address. I expect to see my program to output the port as filtered and in the packet capture I expect to see three UDP packets with no response to any of them. In figure 50 you can see my program correctly identifies the port as being filtered and in figure 51 you can see the packet capture of the scan which also as expected shows the three UDP packets with no reply packets. This shows success criteria 1 and 11.

```
networkScanner/Code on  master [✘!?] venv:(net_scanner) pyenv:( net_scanner) took 3s
➜ sudo ./netscan.py 127.0.0.1 -p 12345 -sU -Pn

Scan report for: 127.0.0.1
Open ports:
Filtered ports:
12345 service: italk?
```

Figure 50: screenshot showing the output of my program when scanning with the options specified above.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	127.0.0.1	127.0.0.1	UDP	92	41279 → 12345 Len=50
2	0.000008961	127.0.0.1	127.0.0.1	UDP	92	41279 → 12345 Len=50
3	4.026639713	127.0.0.1	127.0.0.1	UDP	92	41279 → 12345 Len=50

Figure 51: screenshot showing the packet capture of the scan in figure 50

4.2.13 Detecting the operating system of another machine

I haven't directly added this as a feature to my project partly because I didn't have time and also because it is partially achieved by version scanning in that if a particular service is detected and that service is OS dependent then you can be fairly certain that machine is running that OS. For example if a machine is open on TCP port 22 and SSH is detected to be running behind that port then they are likely to be running a linux machine. Even more likely if the scan reveals some further information such as the CPE. In figure 52 you can see a scan of my machine where I have Secure SHell (SSH) running, my program reveals that the version is 7.9 and the vendor is openbsd which is a unix like operating system, this shows that my ssh version is unix based and therefore I am likely to be running on linux, which is the case. So although it is not directly a feature in a round a bout way. This partially completes success criteria 12.

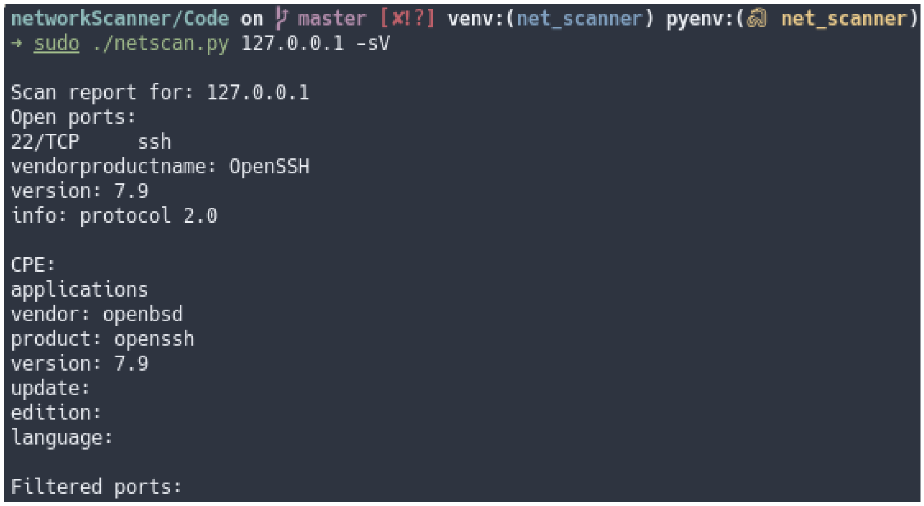


Figure 52: screenshot showing a version scan of my local machine.

4.2.14 Detecting the service and its version running behind a port

To show this I will use my program to perform a version detection scan on my local machine while I am running SSH. I expect to see my program identify that SSH is running on TCP port 22 and that it detects it as OpenSSH version 7.9. To test this I will run my program with the `-sV` flag to indicate version detection and I will run it against the localhost address. In figure 14 you can see that my program successfully identified SSH as running on TCP port 22 as well as the expected identification of OpenSSH version 7.9 operating on protocol version 2. It also identified some CPE information such as OpenSSH coming from the openbsd distribution. This shows success criteria 1, 13 and 14.

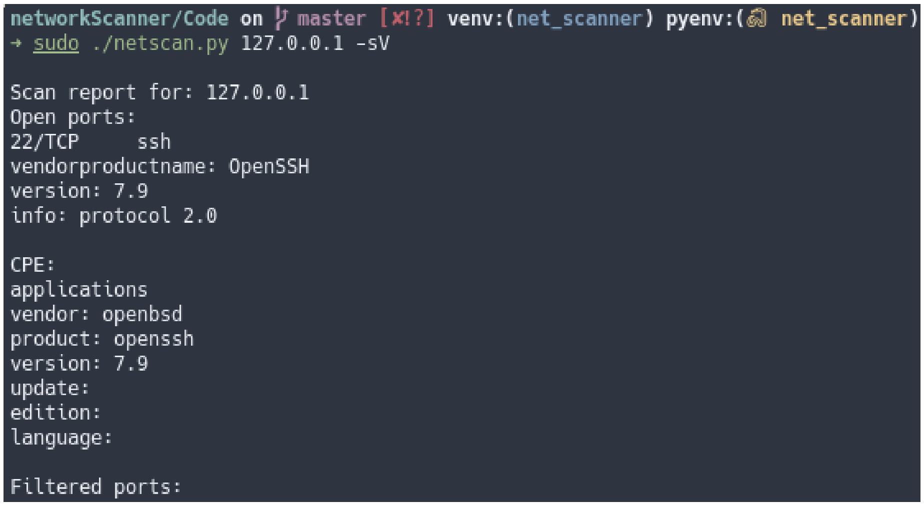


Figure 53: screenshot showing a version scan of my local machine running ssh.

4.3 Test Table

Test No.	Test Data	Expectation	Result	Fig	Success Criteria
1		usage message	Pass	29	2
2	`-h`	help message	Pass	30	2
3	`--help`	help message	Pass	31	2
4	`-sL`	print addresses	Pass	35	3
5	`-sn`	ping scan	Pass	33	4
6	`-Pn`	assume host up	Pass	37	5
7	`-sS|sT`	TCP port open	Pass	40	6
8	`-sS|sT`	TCP port closed	Pass	42	7
9	`-sS`	TCP port filtered	Pass	44	8
10	`-sU`	UDP port open	Pass	46	9
11	`-sU`	UDP port closed	Pass	48	10
12	`-sU`	UDP port filtered	Pass	50	11
13	`-sV`	OS detection	Partial	52	12
14	`-sV`	service detection	Pass	52	13
15	`-sV`	version detection	Pass	52	14

5 Evaluation

5.1 Reflection on final outcome

5.2 Evaluation against objectives, end user feedback

5.3 Potential improvements

A Technical Solution

A.1 icmp_ping

Listing 9: A prototype program for sending ICMP ECHO REQEST packets

```python
#!/usr/bin/env python
import socket
import struct
import os
import time
from modules.ip_utils import ip_checksum


def main() -> None:
    ICMP_ECHO_REQUEST = 8

    # opens a raw socket for the ICMP protocol
    ping_sock = socket.socket(
        socket.AF_INET,
        socket.SOCK_RAW,
        socket.IPPROTO_ICMP
    )
    # allows manual IP header creation
    # ping_sock.setsockopt(socket.SOL_IP, socket.IP_HDRINCL, 1)

    ID = os.getpid() & 0xFFFF

    # the two zeros are the code and the dummy checksum, the one is the
    # sequence number
    dummy_header = struct.pack("bbHHh", ICMP_ECHO_REQUEST, 0, 0, ID, 1)

    data = struct.pack(
        "d", time.time()
    ) + bytes(
        (192 - struct.calcsize("d")) * "A",
        "ascii"
    )
    # the data to send in the packet
    checksum = socket.htons(ip_checksum(dummy_header + data))
    # calculates the checksum for the packet and psuedo header
    header = struct.pack("bbHHh", ICMP_ECHO_REQUEST, 0, checksum, ID, 1)
```

```python
    # packs the packet header
    packet = header + data
    # concatonates the header and the data to form the final packet.
    ping_sock.sendto(packet, ("127.0.0.1", 1))
    # sends the packet to localhost
```

Listing 10: A prototype program for receiving ICMP ECHO REQEST packets

```python
#!/usr/bin/env python
from modules import headers
import socket
from typing import List


def main() -> None:
    # socket object using an IPV4 address, using only raw socket access,
        set
    # ICMP protocol
    ping_sock = socket.socket(
        socket.AF_INET,
        socket.SOCK_RAW,
        socket.IPPROTO_ICMP
    )

    packets: List[bytes] = []

    while len(packets) < 1:
        recPacket, addr = ping_sock.recvfrom(1024)
        ip = headers.ip(recPacket[:20])
        icmp = headers.icmp(recPacket[20:28])

        print(ip)
        print()
        print(icmp)
        print("\n")

        packets.append(recPacket)
```

A.2 ping_scanner

Listing 11: A prototype program for performing ‘ping’ scans

```python
#!/usr/bin/env python
from modules import headers
from modules import ip_utils
import socket
import struct
import time
from contextlib import closing
```

```python
from itertools import repeat
from math import log10, floor
from multiprocessing import Pool
from os import getpid
from typing import Set, Tuple


def sig_figs(x: float, n: int) -> float:
    """
    rounds x to n significant figures.
    sig_figs(1234, 2) = 1200.0
    """
    return round(x, n - (1 + int(floor(log10(abs(x))))))


def ping_listener(
        ID: int,
        timeout: float
) -> Set[Tuple[str, float, headers.ip]]:
    """
    Takes in a process id and a timeout and returns
    a list of addresses which sent ICMP ECHO REPLY
    packets with the packed id matching ID in the time given by timeout.
    """
    ping_sock = socket.socket(
        socket.AF_INET,
        socket.SOCK_RAW,
        socket.IPPROTO_ICMP
    )
    # opens a raw socket for sending ICMP protocol packets
    time_remaining = timeout
    addresses = set()
    while True:
        time_waiting = ip_utils.wait_for_socket(ping_sock, time_remaining)
        # time_waiting stores the time the socket took to become readable
    # or returns minus one if it ran out of time

        if time_waiting == -1:
            break
        time_recieved = time.time()
        # store the time the packet was recieved
        recPacket, addr = ping_sock.recvfrom(1024)
        # recieve the packet
        ip = headers.ip(recPacket[:20])
        # unpack the IP header into its respective components
        icmp = headers.icmp(recPacket[20:28])
        # unpack the time from the packet.
        time_sent = struct.unpack(
            "d",
```

```
            recPacket[28:28 + struct.calcsize("d")]
        )[0]
        # unpack the value for when the packet was sent
        time_taken: float = time_recieved - time_sent
        # calculate the round trip time taken for the packet
        if icmp.id == ID:
            # if the ping was sent from this machine then add it to the list of
            # responses
            ip_address, port = addr
            addresses.add((ip_address, time_taken, ip))
        elif time_remaining <= 0:
            break
        else:
            continue
    # return a list of all the addesses that replied to our ICMP echo request.
    return addresses


def main() -> None:
    with closing(
            socket.socket(
                socket.AF_INET,
                socket.SOCK_RAW,
                socket.IPPROTO_ICMP
            )
    ) as ping_sock:
        ip_addresses = ["127.0.0.1"] # ip_utils.ip_range("192.168.43.0", 24)
        # generate the range of IP addresses to scan.
        # get the local ip address
        addresses = [
            ip
            for ip in ip_addresses
            if (
                not ip.endswith(".0")
                and not ip.endswith(".255")
            )
        ]

        # initialise a process pool
        p = Pool(1)
        # get the local process id for use in creating packets.
        ID = getpid() & 0xFFFF
        # run the listeners.ping function asynchronously
        replied = p.apply_async(ping_listener, (ID, 5))
        time.sleep(0.01)
        for address in zip(addresses, repeat(1)):
            try:
```

```
                packet = ip_utils.make_icmp_packet(ID)
                ping_sock.sendto(packet, address)
            except PermissionError:
                ip_utils.eprint("raw sockets require root priveleges,
                    exiting")
                exit()
        p.close()
        p.join()
        # close and join the process pool to so that all the values
        # have been returned and the pool closed
        hosts_up = replied.get()
        # get the list of addresses that replied to the echo request
            from the
        # listener function
        print("\n".join(
            f"host: [{host}]\t" +
            "responded to an ICMP ECHO REQUEST in " +
            f"{str(sig_figs(taken, 2))+'s':<10s} " +
            f"ttl: [{ip_head.time_to_live}]"
            for host, taken, ip_head in hosts_up
        ))
```

A.3 subnet_to_addresses

Listing 12: A program which translates a CIDR specified subnet into a list of addresses and prints them out in sorted order

```
#!/usr/bin/env python
import re
from modules.ip_utils import ip_range, dot_to_long


if __name__ == '__main__':
    from argparse import ArgumentParser
    parser = ArgumentParser()
    parser.add_argument(
        "ip_subnet",
        help="The CIDR form ip/subnet that you wish to print" +
            "the IP addresses specified by."
    )
    args = parser.parse_args()
    CIDR_regex = re.compile(r"(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}/\d+)")
    search = CIDR_regex.search(args.ip_subnet)
    if search:
        ip, network_bits = search.group(1).split("/")
        print("\n".join(
            sorted(
                ip_range(ip, int(network_bits)),
                key=dot_to_long
```

```
        )
    ))
```

A.4 tcp_scan

A.4.1 connect_scan

Listing 13: prototype TCP connect scanner only attempting to detect the state of port 22

```
#!/usr/bin/python3
from contextlib import closing
import socket
LOCAL_IP = "192.168.1.159"
PORT = 22

address = ("127.0.0.1", 22)

with closing(
        socket.socket(
            socket.AF_INET,
            socket.SOCK_STREAM
        )
) as s:
    try:
        s.connect(address)
        print(f"connection on port {PORT} succedded")
    except ConnectionRefusedError:
        print(f"port {PORT} is closed")
```

Listing 14: A program that performs TCP connect scanning

```
#!/usr/bin/python3

from typing import List, Set


def connect_scan(address: str, ports: Set[int]) -> List[int]:
    import socket
    from contextlib import closing
    open_ports: List[int] = []
    for port in ports:
        # loop through each port in the list of ports to scan
        try:
            with closing(
                    socket.socket(
                        socket.AF_INET,
                        socket.SOCK_STREAM
                    )
```

```
            ) as s:
                # open an IPV4 TCP socket
                s.connect((address, port))
                # attempt to connect the newly created socket to the target
                # address and port
                open_ports.append(port)
                # if the connection was successful then add the port to the
                # list of open ports
        except ConnectionRefusedError:
            pass
    return open_ports


def main() -> None:
    open_ports = connect_scan("192.168.43.225", set(range(65535)))
    print("\n".join(map(lambda x: f"port: [{x}]\tis open", open_ports)))
```

A.4.2 syn_scan

Listing 15: A prototype program that tries to detect the state of port 22 via TCP SYN scanning (aka half open scanning)

```
#!/usr/bin/python3.7
from contextlib import closing
import socket
import ip_utils

dest_port = 22
src_port = ip_utils.get_free_port()
local_ip = ip_utils.get_local_ip()
dest_ip = "192.168.1.159"
local_ip = dest_ip = "127.0.0.1"
loc_long = ip_utils.dot_to_long(local_ip)

SYN = 2
RST = 4



with closing(
        socket.socket(
            socket.AF_INET,
            socket.SOCK_RAW,
            socket.IPPROTO_TCP
        )
) as s:
    tcp_packet = ip_utils.make_tcp_packet(
```

```python
        src_port,
        dest_port,
        local_ip,
        dest_ip,
        SYN
    )
    if tcp_packet is not None:
        s.sendto(tcp_packet, (dest_ip, dest_port))
    else:
        print(f"Couldn't make TCP packet with supplied arguments:",
              f"source port: [{src_port}]",
              f"destination port: [{dest_port}]",
              f"local ip: [{local_ip}]",
              f"destination ip: [{dest_ip}]",
              f"SYN flag: [{SYN}]",
              sep="\n")
```

Listing 16: A program that performs TCP SYN scanning (aka half open scanning)

```python
#!/usr/bin/python3.7
from modules import headers
from modules import ip_utils
import socket
from contextlib import closing
from multiprocessing import Pool
from typing import List, Set, Tuple


def syn_listener(address: Tuple[str, int], timeout: float) -> List[int]:
    """
    This function is run asynchronously and listens for
    TCP ACK responses to the sent TCP SYN msg.
    """
    print(f"address: [{address}]\ntimeout: [{timeout}]")
    open_ports: List[int] = []
    with closing(
            socket.socket(
                socket.AF_INET,
                socket.SOCK_RAW,
                socket.IPPROTO_TCP
            )) as s:
        s.bind(address)
        # bind the raw socket to the listening address
        time_remaining = timeout
        print("started listening")
        while True:
            time_taken = ip_utils.wait_for_socket(s, time_remaining)
            # wait for the socket to become readable
            if time_taken == -1:
```

```python
                break
            else:
                time_remaining -= time_taken
            packet = s.recv(1024)
            # recieve the packet data
            tcp = headers.tcp(packet[20:40])
            if tcp.flags == 0b00010010: # syn ack
                print(tcp)
                open_ports.append(tcp.source)
                # check that the header contained the TCP ACK flag and if
                    it
                # did append it
            else:
                continue
        print("finished listening")
    return open_ports


def syn_scan(dest_ip: str, portlist: Set[int]) -> List[int]:
    src_port = ip_utils.get_free_port()
    # request a local port to connect from
    local_ip = ip_utils.get_local_ip()
    p = Pool(1)
    listener = p.apply_async(syn_listener, ((local_ip, src_port), 5))
    # start the TCP ACK listener in the background
    print("starting scan")
    for port in portlist:
        packet = ip_utils.make_tcp_packet(src_port, port, local_ip,
            dest_ip, 2)
        # create a TCP packet with the syn flag
        with closing(
                socket.socket(
                    socket.AF_INET,
                    socket.SOCK_RAW,
                    socket.IPPROTO_TCP
                )
        ) as s:
            s.sendto(packet, (dest_ip, port))
            # send the packet to its destination

    print("finished scan")
    p.close()
    p.join()
    open_ports = listener.get()
    # collect the list of ports that responded to the TCP SYN message
    print(open_ports)
    return open_ports


def main() -> None:
```

```
    dest_ip = "127.0.0.1"
    syn_scan(dest_ip, set(range(2**16)))
```

A.5 udp_scan

Listing 17: A prototype program to detect whether UDP port 53 is open on a target machine

```
#!/usr/bin/ python
from contextlib import closing
import ip_utils
import socket

dest_ip = "192.168.1.1"
dest_port = 68
local_ip = ip_utils.get_local_ip()
local_port = ip_utils.get_free_port()

local_ip = dest_ip = "127.0.0.1"

address = (dest_ip, dest_port)

with closing(
        socket.socket(
            socket.AF_INET,
            socket.SOCK_RAW,
            socket.IPPROTO_UDP
        )) as s:
    try:
        pkt = ip_utils.make_udp_packet(
            local_port,
            dest_port,
            local_ip,
            dest_ip
        )
        if pkt is not None:
            packet = bytes(pkt)
            s.sendto(packet, address)
        else:
            print(
                "Error making packet.",
                f"local port: [{local_port}]",
                f"destination port: [{dest_port}]",
                f"local ip: [{local_ip}]",
                f"destination ip: [{dest_ip}]",
                sep="\n"
            )
    except socket.error:
        raise
```

Listing 18: A program for performing scans on UDP ports.

```python
#!/usr/bin/env python
from modules import headers
from modules import ip_utils
import socket
import time
from collections import defaultdict
from contextlib import closing
from multiprocessing import Pool
from typing import Set, DefaultDict


def udp_listener(dest_ip: str, timeout: float) -> Set[int]:
    """
    This listener detects UDP packets from dest_ip in the given timespan,
    all ports that send direct replies are marked as being open.
    Returns a list of open ports.
    """

    time_remaining = timeout
    ports: Set[int] = set()
    with socket.socket(
            socket.AF_INET,
            socket.SOCK_RAW,
            socket.IPPROTO_UDP
    ) as s:
        while True:
            time_taken = ip_utils.wait_for_socket(s, time_remaining)
            if time_taken == -1:
                break
            else:
                time_remaining -= time_taken
            packet = s.recv(1024)
            ip = headers.ip(packet[:20])
            udp = headers.udp(packet[20:28])
            # unpack the UDP header
            if dest_ip == ip.source and ip.protocol == 17:
                ports.add(udp.src)

    return ports


def icmp_listener(src_ip: str, timeout: float = 2) -> int:
    """
    This listener detects ICMP destination unreachable
    packets and returns the icmp code.
    This is later used to mark them as either close, open|filtered,
```

```
        filtered.
    3 -> closed
    0|1|2|9|10|13 -> filtered
    -1 -> error with arguments
    open|filtered means that they are either open or
    filtered but return nothing.
    """

    ping_sock = socket.socket(
        socket.AF_INET,
        socket.SOCK_RAW,
        socket.IPPROTO_ICMP
    )
    # open raw socket to listen for ICMP destination unrechable packets
    time_remaining = timeout
    code = -1
    while True:
        time_waiting = ip_utils.wait_for_socket(ping_sock,
             time_remaining)
        # wait for socket to be readable
        if time_waiting == -1:
            break
        else:
            time_remaining -= time_waiting
        recPacket, addr = ping_sock.recvfrom(1024)
        # recieve the packet
        ip = headers.ip(recPacket[:20])
        icmp = headers.icmp(recPacket[20:28])
        valid_codes = [0, 1, 2, 3, 9, 10, 13]
        if (
                ip.source == src_ip
                and icmp.type == 3
                and icmp.code in valid_codes
        ):
            code = icmp.code
            break
        elif time_remaining <= 0:
            break
        else:
            continue
    ping_sock.close()
    return code


def udp_scan(
        dest_ip: str,
        ports_to_scan: Set[int]
) -> DefaultDict[str, Set[int]]:
    """
    Takes in a destination IP address in either dot or long form and
```

```
    a list of ports to scan. Sends UDP packets to each port specified
    in portlist and uses the listeners to mark them as open, open|filtered,
    filtered, closed they are marked open|filtered if no response is
    recieved at all.
    """

    local_ip = ip_utils.get_local_ip()
    local_port = ip_utils.get_free_port()
    # get local ip address and port number
    ports: DefaultDict[str, Set[int]] = defaultdict(set)
    ports["REMAINING"] = ports_to_scan
    p = Pool(1)
    udp_listen = p.apply_async(udp_listener, (dest_ip, 4))
    # start the UDP listener
    with closing(
            socket.socket(
                socket.AF_INET,
                socket.SOCK_RAW,
                socket.IPPROTO_UDP
            )
    ) as s:
        for _ in range(2):
            # repeat 3 times because UDP scanning comes
            # with a high chance of packet loss
            for dest_port in ports["REMAINING"]:
                try:
                    packet = ip_utils.make_udp_packet(
                        local_port,
                        dest_port,
                        local_ip,
                        dest_ip
                    )
                    # create the UDP packet to send
                    s.sendto(packet, (dest_ip, dest_port))
                    # send the packet to the currently scanning address
                except socket.error:
                    packet_bytes = " ".join(map(hex, packet))
                    print(
                        "The socket modules sendto method with the following",
                        "argument resulting in a socket error.",
                        f"\npacket: [{packet_bytes}]\n",
                        "address: [{dest_ip, dest_port}])"
                    )

    p.close()
    p.join()

    ports["OPEN"].update(udp_listen.get())
```

```
    ports["REMAINING"] -= ports["OPEN"]
    # only scan the ports which we know are not open
    with closing(
            socket.socket(
                socket.AF_INET,
                socket.SOCK_RAW,
                socket.IPPROTO_UDP
            )
    ) as s:
        for dest_port in ports["REMAINING"]:
            try:
                packet = ip_utils.make_udp_packet(
                    local_port,
                    dest_port,
                    local_ip,
                    dest_ip
                )
                # make a new UDP packet
                p = Pool(1)
                icmp_listen = p.apply_async(icmp_listener, (dest_ip,))
                # start the ICMP listener
                time.sleep(1)
                s.sendto(packet, (dest_ip, dest_port))
                # send packet
                p.close()
                p.join()
                icmp_code = icmp_listen.get()
                # recieve ICMP code from the ICMP listener
                if icmp_code in {0, 1, 2, 9, 10, 13}:
                    ports["FILTERED"].add(dest_port)
                elif icmp_code == 3:
                    ports["CLOSED"].add(dest_port)
            except socket.error:
                packet_bytes = " ".join(map("{:02x}".format, packet))
                ip_utils.eprint(
                    "The socket modules sendto method with the following",
                    "argument resulting in a socket error.",
                    f"\npacket: [{packet_bytes}]\n",
                    "address: [{dest_ip, dest_port}])"
                )
    # this creates a new set which contains all the elements that
    # are in the list of ports to be scanned but have not yet
    # been classified
    ports["OPEN|FILTERED"] = (
        ports["REMAINING"]
        - ports["OPEN"]
        - ports["FILTERED"]
        - ports["CLOSED"]
    )
```

```
    # set comprehension to update the list of open filtered ports
    return ports


def main() -> None:
    ports = udp_scan("127.0.0.1", {22, 68, 53, 6969})
    print(f"Open ports: {ports['OPEN']}")
    print(f"Open or filtered ports: {ports['OPEN|FILTERED']}")
    print(f"Filtered ports: {ports['FILTERED']}")
    print(f"Closed ports: {ports['CLOSED']}")
```

Listing 19: A program I made to open a port via UDP for testing my UDP scanner.

```
#!/usr/bin/env python

import socket
from contextlib import closing

with closing(
        socket.socket(
            socket.AF_INET,
            socket.SOCK_DGRAM
        )
) as s:
    s.bind(("127.0.0.1", 12345))
    print("opened port 12345 on localhost")
    while True:
        data, addr = s.recvfrom(1024)
        s.sendto(bytes("Well hello there good sir.", "utf-8"), addr)
```

A.6 version_detection

Listing 20: A program which does version detection on services.

```
#!/usr/bin/env python
from typing import Dict, Set, Pattern, Tuple, DefaultDict
from functools import reduce
from collections import defaultdict
from modules import directives
import re
import operator

# type annotaion for the container which
# holds the probes. I have abstracted it from
# the function definition because multiple functions
# depend on it and they weren't all getting updated
# if I needed to change the function signature.
PROBE_CONTAINER = DefaultDict[str, Dict[str, directives.Probe]]
```

```python


def parse_ports(portstring: str) -> DefaultDict[str, Set[int]]:
    """
    This function takes in a port directive
    and returns a set of the ports specified.
    A set is used because it is O(1) for contains
    operations as opposed for O(N) for lists.
    """
    # matches both the num-num port range format
    # and the plain num port specification
    # num-num form must come first otherwise it breaks.
    proto_regex = re.compile(r"([ TU]):?([0-9,-]+)")
    # THE SPACE IS IMPORTANT!!!
    # it allows ports specified before TCP/UDP ports
    # to be specified globally as in for all protocols.

    pair_regex = re.compile(r"(\d+)-(\d+)")
    single_regex = re.compile(r"(\d+)")
    ports: DefaultDict[str, Set[int]] = defaultdict(set)
    # searches contains the result of trying the pair_regex
    # search against all of the command seperated
    # port strings

    for protocol, portstring in proto_regex.findall(portstring):
        pairs = pair_regex.findall(portstring)
        # for each pair of numbers in the pairs list
        # seperate each number and cast them to int
        # then generate the range of numbers from x[0]
        # to x[1]+1 then cast this range to a list
        # and "reduce" the list of lists by joining them
        # with operator.ior (inclusive or) and then let
        # ports be the set of all the ports in that list.
        proto_map = {
            " ": "ANY",
            "U": "UDP",
            "T": "TCP"
        }
        if pairs:
            def pair_to_ports(pair: Tuple[int, int]) -> Set[int]:
                """
                a function to go from a port pair i.e. (80-85)
                to the set of specified ports: {80,81,82,83,84,85}
                """
                start, end = pair
                return set(range(start, end+1))
            # ports contains the set of all ANY/TCP/UDP specified ports
            ports[proto_map[protocol]] = set(reduce(
                operator.ior,
                map(pair_to_ports, pairs)
```

```
            ))

        singles = single_regex.findall(portstring)
        # for each of the ports that are specified on their own
        # cast them to int and update the set of all ports with
        # that list.
        ports[proto_map[protocol]].update(map(int, singles))

    return ports


def parse_probes(probe_file: str) -> PROBE_CONTAINER:
    """
    Extracts all of the probe directives from the
    file pointed to by probe_file.
    """
    # lines contains each line of the file which doesn't
    # start with a # and is not empty.
    lines = [
        line
        for line in open(probe_file).read().splitlines()
        if line and not line.startswith("#")
    ]

    # list holding each of the probe directives.
    probes: PROBE_CONTAINER = defaultdict(dict)

    regexes: Dict[str, Pattern] = {
        "probe":       re.compile(r"Probe (TCP|UDP) (\S+) q\|(.*)\|"),
        "match":       re.compile(" ".join([
            r"(?P<type>softmatch|match)",
            r"(?P<service>\S+)",
            r"m([@/%=|])(?P<regex>.+?)\3(?P<flags>[si]*)"
        ])),
        "rarity":      re.compile(r"rarity (\d+)"),
        "totalwaitms": re.compile(r"totalwaitms (\d+)"),
        "tcpwrappedms": re.compile(r"tcpwrappedms (\d+)"),
        "fallback":    re.compile(r"fallback (\S+)"),
        "ports":       re.compile(r"ports (\S+)"),
        "exclude":     re.compile(r"Exclude T:(\S+)")
    }

    # parse the probes out from the file
    for line in lines:
        # add any ports to be excluded to the base probe class
        if line.startswith("Exclude"):
            search = regexes["exclude"].search(line)
            if search:
                # parse the ports from the grouped output of
                # a search with the regex defined above.
```

```
                for protocol, ports in parse_ports(search.group(1)).items():
                    directives.Probe.exclude[protocol].update(ports)
            else:
                print(line)
                input()

        # new probe directive
        if line.startswith("Probe"):
            # parse line into probe protocol, name and probestring
            search = regexes["probe"].search(line)
            if search:
                try:
                    proto, name, string = search.groups()
                except ValueError:
                    print(line)
                    raise
                probes[name][proto] = directives.Probe(proto, name, string)
                # assign current_probe to the most recently added probe
                current_probe = probes[name][proto]
            else:
                print(line)
                input()

        # new match directive
        elif line.startswith("match") or line.startswith("softmatch"):
            search = regexes["match"].search(line)
            if search:
                # the remainder of the string after the match
                version_info = line[search.end()+1:]
                # escape the curly braces so the regex engine doesn't
                # consider them to be special characters
                pattern = bytes(search.group("regex"), "utf-8")
                # these replace the literal \n, \r and \t
                # strings with their actual characters
                # i.e. \n -> newline character
                pattern = pattern.replace(b"\\n", b"\n")
                pattern = pattern.replace(b"\\r", b"\r")
                pattern = pattern.replace(b"\\t", b"\t")
                matcher = directives.Match(
                    search.group("service"),
                    pattern,
                    search.group("flags"),
                    version_info
                )
                if search.group("type") == "match":
                    current_probe.matches.add(matcher)
                else:
                    current_probe.softmatches.add(matcher)
```

```python

            else:
                print(line)
                input()

        # new ports directive
        elif line.startswith("ports"):
            search = regexes["ports"].search(line)
            if search:
                for protocol, ports in
                    parse_ports(search.group(1)).items():
                    current_probe.ports[protocol].update(ports)
            else:
                print(line)
                input()
        # new totalwaitms directive
        elif line.startswith("totalwaitms"):
            search = regexes["totalwaitms"].search(line)
            if search:
                current_probe.totalwaitms = int(search.group(1))
            else:
                print(line)
                input()

        # new rarity directive
        elif line.startswith("rarity"):
            search = regexes["rarity"].search(line)
            if search:
                current_probe.rarity = int(search.group(1))
            else:
                print(line)
                input()

        # new fallback directive
        elif line.startswith("fallback"):
            search = regexes["fallback"].search(line)
            if search:
                current_probe.fallback = set(search.group(1).split(","))
            else:
                print(line)
                input()
    return probes


def version_detect_scan(
        target: directives.Target,
        probes: PROBE_CONTAINER
) -> directives.Target:
    for probe_dict in probes.values():
        for proto in probe_dict:
```

```python
            target = probe_dict[proto].scan(target)
    return target


def main() -> None:
    print("reached here")
    probes = parse_probes("./version_detection/nmap-service-probes")
    open_ports: DefaultDict[str, Set[int]] = defaultdict(set)
    open_filtered_ports: DefaultDict[str, Set[int]] = defaultdict(set)
    open_filtered_ports["TCP"].add(22)
    open_ports["TCP"].update([1, 2, 3, 4, 5, 6, 8, 65,
                              20, 21, 23, 24, 25])

    target = directives.Target(
        "127.0.0.1",
        open_ports,
        open_filtered_ports
    )
    target.open_ports["TCP"].update([1, 2, 3])
    print("BEFORE")
    print(target)
    scanned = version_detect_scan(target, probes)
    print("AFTER")
    print(scanned)
```

A.7 modules

Listing 21: A python module I wrote for parsing and holding the version detection probes from the nmap_service_probes file.

```python
#!/usr/bin/env python
from collections import defaultdict
from contextlib import closing
from dataclasses import dataclass, field
from functools import reduce
from string import whitespace, printable
from typing import (
    DefaultDict,
    Dict,
    Set,
    List,
    Pattern,
    Match as RE_Match,
    Tuple
)
from . import ip_utils
import operator
import re
import socket
```

```python
import struct


class Match:
    """
    This is a class for both Matches and
    Softmatches as they are actually the same
    thing except that softmatches have less information.
    """
    options_to_flags = {
        "i": re.IGNORECASE,
        "s": re.DOTALL
    }
    letter_to_name = {
        "p": "vendorproductname",
        "v": "version",
        "i": "info",
        "h": "hostname",
        "o": "operatingsystem",
        "d": "devicetype"
    }
    cpe_part_map: Dict[str, str] = {
        "a": "applications",
        "h": "hardware platforms",
        "o": "operating systems"
    }
    # look into match.expand when looking at the substring version info
        things.

    def __init__(
            self,
            service: str,
            pattern: bytes,
            pattern_options: str,
            version_info: str
    ):
        self.version_info: Dict[str, str] = dict()
        self.cpes: Dict[str, Dict[str, str]] = dict()
        self.service: str = service
        # bitwise or is used to combine flags
        # pattern options will never be anything but a
        # combination of s and i.
        # the default value of re.V1 is so that
        # re uses the newer matching engine.
        flags = reduce(
            operator.ior,
            [
                self.options_to_flags[opt]
                for opt in pattern_options
            ],
```

```
            0
        )
        try:
            self.pattern: Pattern = re.compile(
                pattern,
                flags=flags
            )
        except Exception as e:
            print("Regex failed to compile:")
            print(e)
            print(pattern)
            input()

        vinfo_regex = re.compile(r"([pvihod]|cpe:)([/|])(.+?)\2([a]*)")
        cpe_regex = re.compile(
            ":?".join((
                "(?P<part>[aho])",
                "(?P<vendor>[^:]*)",
                "(?P<product>[^:]*)",
                "(?P<version>[^:]*)",
                "(?P<update>[^:]*)",
                "(?P<edition>[^:]*)",
                "(?P<language>[^:]*)"
            ))
        )

        for fieldname, _, val, opts in vinfo_regex.findall(version_info):
            if fieldname == "cpe:":
                search = cpe_regex.search(val)
                if search:
                    part = search.group("part")
                    # this next bit is so that the bytes produced by the regex
                    # are turned to strings
                    self.cpes[Match.cpe_part_map[part]] = {
                        key: value
                        for key, value
                        in search.groupdict().items()
                    }
            else:
                self.version_info[
                    Match.letter_to_name[fieldname]
                ] = val

    def __repr__(self) -> str:
        return "Match(" + ", ".join((
                f"service={self.service}",
                f"pattern={self.pattern}",
                f"version_info={self.version_info}",
                f"cpes={self.cpes}"
```

```
            )) + ")"

    def matches(self, string: bytes) -> bool:
        def replace_groups(
                string: str,
                original_match: RE_Match
        ) -> str:
            """
            This function takes in a string and the original
            regex search performed on the data recieved and
            replaces all of the $i, $SUBST, $I, $P occurances
            with the relavant formatted text that they produce.
            """
            def remove_unprintable(
                    group: int,
                    original_match: RE_Match
            ) -> bytes:
                """
                Mirrors the P function from nmap which
                is used to print only printable characters.
                i.e. W\0O\0R\0K\0G\0R\0O\0U\0P -> WORKGROUP
                """
                return b"".join(
                    i for i in original_match.group(group)
                    if ord(i) in (
                        set(printable)
                        - set(whitespace)
                        | {" "}
                    )
                )
                # if i in the set of all printable characters,
                # excluding those of which that are whitespace characters
                # but including space.

            def substitute(
                group: int,
                before: bytes,
                after: bytes,
                original_match: RE_Match
            ) -> bytes:
                """
                Mirrors the SUBST function from nmap which is used to
                format some information found by the regex.
                by substituting all instances of `before` with `after`.
                """
                return original_match.group(group).replace(before, after)

            def unpack_uint(
                    group: int,
                    endianness: str,
```

```
                original_match: RE_Match
        ) -> bytes:
            """
            Mirrors the I function from nmap which is used to
            unpack an unsigned int from some bytes.
            """
            return bytes(struct.unpack(
                endianness + "I",
                original_match.group(group)
            ))

        text = bytes(string, "utf-8")
        # fill in the version information from the regex match
        # find all the dollar groups:
        dollar_regex = re.compile(r"\$(\d)")
        # find all the $i's in string
        numbers = set(int(i) for i in dollar_regex.findall(string))
        # for each $i found i
        for group in numbers:
            text = text.replace(
                bytes(f"${group}", "utf-8"),
                original_match.group(group)
            )
        # having replaced all of the groups we can now
        # start doing the SUBST, P and I commands.
        subst_regex = re.compile(rb"\$SUBST\((\d),(.+),(.+)\)")
        # iterate over all of the matches found by the SUBST regex
        for match in subst_regex.finditer(text):
            num, before, after = match.groups()
            # replace the full match (group 0)
            # with the output of substitute
            # with the specific arguments
            text.replace(
                match.group(0),
                substitute(int(num), before, after, original_match)
            )

        p_regex = re.compile(rb"\$P\((\d)\)")
        for match in p_regex.finditer(text):
            num = match.group(1)
            # replace the full match (group 0)
            # with the output of remove_unprintable
            # with the specific arguments
            text.replace(
                match.group(0),
                remove_unprintable(int(num), original_match)
            )

        i_regex = re.compile(br"\$I\((\d),\"(\S)\"\)")
        for match in i_regex.finditer(text):
```

```
                num, endianness = match.groups()
                # this means replace group 0 -> the whole match
                # with the output of the unpack_uint
                # with the specified arguments
                text.replace(
                    match.group(0),
                    unpack_uint(
                        int(num.decode()),
                        endianness.decode(),
                        original_match
                    )
                )

            return text.decode()

        search = self.pattern.search(string)
        if search:
            # the fields to replace are all the CPE groups,
            # all of the version info fields.
            self.version_info = {
                key: replace_groups(value, search)
                for key, value in self.version_info.items()
            }
            self.cpes = {
                outer_key: {
                    inner_key: replace_groups(value, search)
                    for inner_key, value in outer_dict.items()
                }
                for outer_key, outer_dict in self.cpes.items()
            }

            return True
        else:
            return False


@dataclass
class Target:
    """
    This class holds data about targets to
    scan. the dataclass decorator is simply
    a way of python automatically writing some
    of the basic methods a class for storing data
    has, such as __repr__ for printing information
    in the object etc.
    """
    address: str
    open_ports: DefaultDict[str, Set[int]]
    open_filtered_ports: DefaultDict[str, Set[int]]
    services: Dict[int, Match] = field(default_factory=dict)
```

```python

    def __repr__(self) -> str:
        def collapse(port_dict: DefaultDict) -> str:
            """
            Collapse a list of port numbers so that
            only the unique ones and the start and end
            of a sequence are displayed.
            1,2,3,4,5,7,9,11,13,14,15,16,17 -> 1-5,7,9,11,13-17
            """
            store_results = list()
            for key in port_dict:
                # items is a sorted list of a set of ports.
                items: List[int] = sorted(port_dict[key])
                key_result = f'"{key}":' + "{"
                # if its an empty list return now to avoid errors
                if len(items) != 0:
                    new_sequence = False
                    # enumerate up until the one before
                    # the last to prevent index errors.
                    for index, item in enumerate(items[:-1]):
                        # if its the first one add it on
                        if index == 0:
                            key_result += f"{item}"
                            # if its a sequence start one else put a comma
                            if items[index+1] == item+1:
                                key_result += "-"
                            else:
                                key_result += ","
                        # if the sequence breaks then put a comma
                        elif item+1 != items[index+1]:
                            key_result += f"{item},"
                            new_sequence = True
                        # if its a new sequence the put the '-'s in
                        elif item+1 == items[index+1] and new_sequence:
                            key_result += f"{item}-"
                            new_sequence = False
                    # because we only iterate to the one before
                    # the last element, add the last element on to the end.
                    key_result += f"{items[-1]}" + "}"
                    store_results.append(key_result)
            # format the final result
            result = "{" + ", ".join(store_results) + "}"
            return result

        open_ports = collapse(self.open_ports)
        open_filtered_ports = collapse(self.open_filtered_ports)
        return ", ".join((
            f"Target(address=[{self.address}]",
            f"open_ports=[{open_ports}]",
            f"open_filtered_ports=[{open_filtered_ports}]",
```

```python
            f"services={self.services})"
        ))


class Probe:
    """
    This class represents the Probe directive of the nmap-service-probes file.
    It holds information such as the protocol to use, the string to send,
    the ports to scan, the time to wait for a null TCP to return a banner,
    the rarity of the probe (how often it will return a response) and the
    probes to try if this one fails.
    """

    # a default dict is one which takes in a
    # "default factory" which is called when
    # a new key is introduced to the dict
    # in this case the default factory is
    # the set function meaning that when I
    # do exclude[protocol].update(ports)
    # but exclude[protocol] has not yet been defined
    # it will be defined as an empty set
    # allowing me to update it with ports.
    exclude: DefaultDict[str, Set[int]] = defaultdict(set)
    proto_to_socket_type: Dict[str, int] = {
        "TCP": socket.SOCK_STREAM,
        "UDP": socket.SOCK_DGRAM
    }

    def __init__(self, protocol: str, probename: str, probe: str):
        """
        This is the initial function that is called by the
        constructor of the Probe class, it is used to define
        the variables that are specific to each instance of
        the class.
        """
        if protocol in {"TCP", "UDP"}:
            self.protocol = protocol
        else:
            raise ValueError(
                f"Probe object must have protocol TCP or UDP not {protocol}.")
        self.name: str = probename
        self.string: str = probe
        self.payload: bytes = bytes(probe, "utf-8")
        self.matches: Set[Match] = set()
        self.softmatches: Set[Match] = set()
        self.ports: DefaultDict[str, Set[int]] = defaultdict(set)
        self.totalwaitms: int = 6000
```

```python
        self.tcpwrappedms: int = 3000
        self.rarity: int = -1
        self.fallback: Set[str] = set()

    def __repr__(self) -> str:
        """
        This is the function that is called when something
        tries to print an instance of this class.
        It is used to reveal information internal
        to the class.
        """
        return ", ".join([
            f"Probe({self.protocol}",
            f"{self.name}",
            f"\"{self.string}\"",
            f"{len(self.matches)} matches",
            f"{len(self.softmatches)} softmatches",
            f"ports: {self.ports}",
            f"rarity: {self.rarity}",
            f"fallbacks: {self.fallback})"
        ])

    def scan(self, target: Target) -> Target:
        """
        scan takes in an object of class Target to
        probe and attempts to detect the version of
        any services running on the machine.
        """
        # this constructs the set of all ports,
        # that are either open or open_filtered,
        # and are in the set of ports to scan for
        # this particular probe, this means that,
        # we are only connecting to ports that we
        # know are not closed and are not to be excluded.

        ports_to_scan: Set[int] = (
            (
                target.open_filtered_ports[self.protocol]
                | target.open_ports[self.protocol]
            )
        ) - Probe.exclude[self.protocol] - Probe.exclude["ANY"]
        # if the probe defines a set of ports to scan
        # then don't scan any that aren't defined for it
        if self.ports[self.protocol] != set():
            ports_to_scan &= self.ports[self.protocol]
        for port in ports_to_scan:
            # open a self closing IPV4 socket
            # for the correct protocol for this probe.
            with closing(
                    socket.socket(
```

```python
                        socket.AF_INET,
                        self.proto_to_socket_type[self.protocol]
                    )
            ) as sock:
                # setup the connection to the target
                try:
                    sock.connect((target.address, port))
                    # if the connection fails then continue scanning
                    # the next ports, this shouldn't really happen.
                except ConnectionError:
                    continue
                # send the payload to the target
                sock.send(self.payload)
                # wait for the target to send a response
                time_taken = ip_utils.wait_for_socket(
                    sock,
                    self.totalwaitms/1000
                )
                # if the response didn't time out
                if time_taken != -1:
                    # if the port was in open_filtered move it to open
                    if port in target.open_filtered_ports[self.protocol]:
                        target.open_filtered_ports[self.protocol].remove(port)
                        target.open_ports[self.protocol].add(port)

                    # recieve the data and decode it to a string
                    data_recieved = sock.recv(4096)
                    #  print("Recieved", data_recieved)
                    service = ""
                    # try and softmatch the service first
                    for softmatch in self.softmatches:
                        if softmatch.matches(data_recieved):
                            service = softmatch.service
                            target.services[port] = softmatch
                            break
                    # try and get a full match for the service
                    for match in self.matches:
                        if service in match.service.lower():
                            if match.matches(data_recieved):
                                target.services[port] = match
                                break
        return target


PROBE_CONTAINER = DefaultDict[str, Dict[str, Probe]]


def parse_ports(portstring: str) -> DefaultDict[str, Set[int]]:
    """
    This function takes in a port directive
```

```python
    and returns a set of the ports specified.
    A set is used because it is O(1) for contains
    operations as opposed for O(N) for lists.
    """
    # matches both the num-num port range format
    # and the plain num port specification
    # num-num form must come first otherwise it breaks.
    proto_regex = re.compile(r"([ TU]?):?([0-9,-]+)")
    # THE SPACE IS IMPORTANT!!!
    # it allows ports specified before TCP/UDP ports
    # to be specified globally as in for all protocols.

    pair_regex = re.compile(r"(\d+)-(\d+)")
    single_regex = re.compile(r"(\d+)")
    ports: DefaultDict[str, Set[int]] = defaultdict(set)
    # searches contains the result of trying the pair_regex
    # search against all of the command seperated
    # port strings

    for protocol, portstring in proto_regex.findall(portstring):
        pairs = pair_regex.findall(portstring)
        # for each pair of numbers in the pairs list
        # seperate each number and cast them to int
        # then generate the range of numbers from x[0]
        # to x[1]+1 then cast this range to a list
        # and "reduce" the list of lists by joining them
        # with operator.ior (inclusive or) and then let
        # ports be the set of all the ports in that list.
        proto_map = {
            "":  "ANY",
            " ": "ANY",
            "U": "UDP",
            "T": "TCP"
        }
        if pairs:
            def pair_to_ports(pair: Tuple[str, str]) -> Set[int]:
                """
                a function to go from a port pair i.e. (80-85)
                to the set of specified ports: {80,81,82,83,84,85}
                """
                start, end = pair
                return set(range(
                    int(start),
                    int(end)+1
                ))
            # ports contains the set of all ANY/TCP/UDP specified ports
            ports[proto_map[protocol]] = set(reduce(
                operator.ior,
                map(pair_to_ports, pairs)
            ))
```

```python

        singles = single_regex.findall(portstring)
        # for each of the ports that are specified on their own
        # cast them to int and update the set of all ports with
        # that list.
        ports[proto_map[protocol]].update(map(int, singles))

    return ports


def parse_probes(probe_file: str) -> PROBE_CONTAINER:
    """
    Extracts all of the probe directives from the
    file pointed to by probe_file.
    """
    # lines contains each line of the file which doesn't
    # start with a # and is not empty.
    lines = [
        line
        for line in open(probe_file).read().splitlines()
        if line and not line.startswith("#")
    ]

    # list holding each of the probe directives.
    probes: PROBE_CONTAINER = defaultdict(dict)

    regexes: Dict[str, Pattern] = {
        "probe":        re.compile(r"Probe (TCP|UDP) (\S+) q\|(.*)\|"),
        "match":        re.compile(" ".join([
            r"(?P<type>softmatch|match)",
            r"(?P<service>\S+)",
            r"m([@/%=|])(?P<regex>.+?)\3(?P<flags>[si]*)"
        ])),
        "rarity":       re.compile(r"rarity (\d+)"),
        "totalwaitms": re.compile(r"totalwaitms (\d+)"),
        "tcpwrappedms": re.compile(r"tcpwrappedms (\d+)"),
        "fallback":     re.compile(r"fallback (\S+)"),
        "ports":        re.compile(r"ports (\S+)"),
        "exclude":      re.compile(r"Exclude T:(\S+)")
    }

    # parse the probes out from the file
    for line in lines:
        # add any ports to be excluded to the base probe class
        if line.startswith("Exclude"):
            search = regexes["exclude"].search(line)
            if search:
                # parse the ports from the grouped output of
                # a search with the regex defined above.
```

```
                for protocol, ports in parse_ports(search.group(1)).items():
                    Probe.exclude[protocol].update(ports)
            else:
                print(line)
                input()

        # new probe directive
        if line.startswith("Probe"):
            # parse line into probe protocol, name and probestring
            search = regexes["probe"].search(line)
            if search:
                try:
                    proto, name, string = search.groups()
                except ValueError:
                    print(line)
                    raise
                probes[name][proto] = Probe(proto, name, string)
                # assign current_probe to the most recently added probe
                current_probe = probes[name][proto]
            else:
                print(line)
                input()

        # new match directive
        elif line.startswith("match") or line.startswith("softmatch"):
            search = regexes["match"].search(line)
            if search:
                # the remainder of the string after the match
                version_info = line[search.end()+1:]
                # escape the curly braces so the regex engine doesn't
                # consider them to be special characters
                pattern = bytes(search.group("regex"), "utf-8")
                # these replace the literal \n, \r and \t
                # strings with their actual characters
                # i.e. \n -> newline character
                pattern = pattern.replace(b"\\n", b"\n")
                pattern = pattern.replace(b"\\r", b"\r")
                pattern = pattern.replace(b"\\t", b"\t")
                matcher = Match(
                    search.group("service"),
                    pattern,
                    search.group("flags"),
                    version_info
                )
                if search.group("type") == "match":
                    current_probe.matches.add(matcher)
                else:
                    current_probe.softmatches.add(matcher)

```

```
            else:
                print(line)
                input()

        # new ports directive
        elif line.startswith("ports"):
            search = regexes["ports"].search(line)
            if search:
                for protocol, ports in
                    parse_ports(search.group(1)).items():
                    current_probe.ports[protocol].update(ports)
            else:
                print(line)
                input()
        # new totalwaitms directive
        elif line.startswith("totalwaitms"):
            search = regexes["totalwaitms"].search(line)
            if search:
                current_probe.totalwaitms = int(search.group(1))
            else:
                print(line)
                input()

        # new rarity directive
        elif line.startswith("rarity"):
            search = regexes["rarity"].search(line)
            if search:
                current_probe.rarity = int(search.group(1))
            else:
                print(line)
                input()

        # new fallback directive
        elif line.startswith("fallback"):
            search = regexes["fallback"].search(line)
            if search:
                current_probe.fallback = set(search.group(1).split(","))
            else:
                print(line)
                input()
    return probes
```

Listing 22: A python module I made to dissect and hold protocol headers.

```
import struct
import socket
from typing import Dict


class ip:
```

```python
    """
    A class for parsing, storing and displaying
    data from an IP header.
    """
    def __init__(self, header: bytes):
        # first unpack the IP header
        (
            ip_hp_ip_v,
            ip_dscp_ip_ecn,
            ip_len,
            ip_id,
            ip_flgs_ip_off,
            ip_ttl,
            ip_p,
            ip_sum,
            ip_src,
            ip_dst
        ) = struct.unpack('!BBHHHBBHII', header)
        # now deal with the sub-byte sized components
        hl_v = f"{ip_hp_ip_v:08b}"
        ip_v = int(hl_v[:4], 2)
        ip_hl = int(hl_v[4:], 2)
        # splits hl_v in ip_v and ip_hl which store the IP version number and
        # header length respectively
        dscp_ecn = f"{ip_dscp_ip_ecn:08b}"
        ip_dscp = int(dscp_ecn[:6], 2)
        ip_ecn = int(dscp_ecn[6:], 2)
        # splits dscp_ecn into ip_dscp and ip_ecn
        # which are two of the compenents
        # in an IP header
        flgs_off = f"{ip_flgs_ip_off:016b}"
        ip_flgs = int(flgs_off[:3], 2)
        ip_off = int(flgs_off[3:], 2)
        # splits flgs_off into ip_flgs and ip_off which represent the ip header
        # flags and the data offset
        src_addr = socket.inet_ntoa(struct.pack('!I', ip_src))
        dst_addr = socket.inet_ntoa(struct.pack('!I', ip_dst))
        self.version: int = ip_v
        self.header_length: int = ip_hl
        self.dscp: int = ip_dscp
        self.ecn: int = ip_ecn
        self.len: int = ip_len
        self.id: int = ip_id
        self.flags: int = ip_flgs
        self.data_offset: int = ip_off
        self.time_to_live: int = ip_ttl
        self.protocol: int = ip_p
        self.checksum: int = ip_sum
```

```python
        self.source: str = src_addr
        self.destination: str = dst_addr

    def __repr__(self) -> str:
        return "\n\t".join((
            "IP header:",
            f"Version: [{self.version}]",
            f"Internet Header Length: [{self.header_length}]",
            f"Differentiated Services Point Code: [{self.dscp}]",
            f"Explicit Congestion Notification: [{self.ecn}]",
            f"Total Length: [{self.len}]",
            f"Identification: [{self.id:04x}]",
            f"Flags: [{self.flags:03b}]",
            f"Fragment Offset: [{self.data_offset}]",
            f"Time To Live: [{self.time_to_live}]",
            f"Protocol: [{self.protocol}]",
            f"Header Checksum: [{self.checksum:04x}]",
            f"Source Address: [{self.source}]",
            f"Destination Address: [{self.destination}]"
        ))


class icmp:
    """
    A class for parsing, storing and displaying
    data from an IP header.
    """
    # relates the type and code to the message
    messages: Dict[int, Dict[int, str]] = {
        0: {
            0: "Echo reply."
        },
        3: {
            0: "Destination network unreachable.",
            1: "Destination host unreachable",
            2: "Destination protocol unreachable",
            3: "Destination port unreachable",
            4: "Fragmentation required, and DF flag set.",
            5: "Source route failed.",
            6: "Destination network unknown.",
            7: "Destination host unknown.",
            8: "Source host isolated.",
            9: "Network administratively prohibited.",
            10: "Host administratively prohibited.",
            11: "Network unreachable for ToS.",
            12: "Host unreachable for ToS.",
            13: "Communication administratively prohibited.",
            14: "Host precedence violation.",
            15: "Precedence cutoff in effect."
        },
```

```python
            4: {
                0: "Source quench."
            },
            5: {
                0: "Redirect datagram for the network",
                1: "Redirect datagram for the host.",
                2: "Redirect datagram for the ToS & network.",
                3: "Redirect datagram for the ToS & host."
            },
            8: {
                0: "Echo request."
            },
            9: {
                0: "Router advertisment"
            },
            10: {
                0: "Router discovery/selection/solicitation."
            },
            11: {
                0: "TTL expired in transit",
                1: "Fragment reassembly time exceeded."
            },
            12: {
                0: "Bad IP header: pointer indicates error.",
                1: "Bad IP header: missing a required option.",
                2: "Bad IP header: Bad length."
            },
            13: {
                0: "Timestamp"
            },
            14: {
                0: "Timestamp reply"
            },
            15: {
                0: "Information request."
            },
            16: {
                0: "Information reply."
            },
            17: {
                0: "Address mask request."
            },
            18: {
                0: "Address mask reply."
            }
        }

        def __init__(self, header: bytes):
            (
                ICMP_type,
```

```
            code,
            csum,
            remainder
        ) = struct.unpack('!bbHI', header)

        self.type: int = ICMP_type
        self.code: int = code
        self.checksum: int = csum

        self.message: str
        try:
            self.message = icmp.messages[self.type][self.code]
        except KeyError:
            # if we can't assign a message then just set a description
            # as to what caused the failure.
            self.message = f"Failed to assign message: ({self.type/self.code})"

        self.id: int
        self.sequence: int
        if self.type in {0, 8}:
            self.id = socket.htons(remainder >> 16)
            self.sequence = socket.htons(remainder & 0xFFFF)
        else:
            self.id = -1
            self.sequence = -1

    def __repr__(self) -> str:
        return "\n\t".join((
            "ICMP header:",
            f"Message: [{self.message}]",
            f"Type: [{self.type}]",
            f"Code: [{self.code}]",
            f"Checksum: [{self.checksum:04x}]",
            f"ID: [{self.id}]",
            f"Sequence: [{self.sequence}]"
        ))


class tcp:
    def __init__(self, header: bytes):
        (
            src_prt,
            dst_prt,
            seq,
            ack,
            data_offset,
            flags,
            window_size,
            checksum,
```

```python
            urg
        ) = struct.unpack("!HHIIBBHHH", header)

        self.source: int = src_prt
        self.destination: int = dst_prt
        self.seq: int = seq
        self.ack: int = ack
        self.data_offset: int = data_offset >> 4
        self.flags: int = flags + ((data_offset & 0x01) << 8)
        self.window_size: int = window_size
        self.checksum: int = checksum
        self.urg: int = urg

    def __repr__(self) -> str:
        return "\n\t".join((
            "TCP header:",
            f"Source port: [{self.source}]",
            f"Destination port: [{self.destination}]",
            f"Sequence number: [{self.seq}]",
            f"Acknowledgement number: [{self.ack}]",
            f"Data offset: [{self.data_offset}]",
            f"Flags: [{self.flags:08b}]",
            f"Window size: [{self.window_size}]",
            f"Checksum: [{self.checksum:04x}]",
            f"Urgent: [{self.urg}]"
        ))


class udp:
    def __init__(self, header: bytes):
        # parse udp header
        (
            src_port,
            dest_port,
            length,
            checksum
        ) = struct.unpack("!HHHH", header)

        self.src: int = src_port
        self.dest: int = dest_port
        self.length: int = length
        self.checksum: int = checksum

    def __repr__(self) -> str:
        return "\n\t".join((
            "UDP header:",
            f"Source port: {self.src}",
            f"Destination port: {self.dest}",
            f"Length: {self.length}",
            f"Checksum: {self.checksum:04x}"
```

```
        ))
```

Listing 23: A python module I wrote to contain lots of useful functions which I found I was declaring in multiple places and makign changes so I decided to keep an up to date central one.

```python
import array
import socket
import struct
import select
import time

from contextlib import closing
from functools import singledispatch
from itertools import islice, cycle
from sys import stderr
from typing import Set, Union


def eprint(*args: str, **kwargs: str) -> None:
    """
    Mirrors print exactly but prints to stderr
    instead of stdout.
    """
    print(*args, file=stderr, **kwargs) # type: ignore


def long_to_dot(long: int) -> str:
    """
    Take in an IP address in packed 32 bit int form
    and return that address in dot notation.
    i.e. long_to_dot(0x7F000001) = 127.0.0.1
    """
    # these are long form values for 0.0.0.0
    # and 255.255.255.255
    if not 0 <= long <= 0xFFFFFFFF:
        raise ValueError(f"Invalid long form IP address: [{long:08x}]")
    else:
        # shift the long form IP along 0, 8, 16, 24 bits
        # take only the first 8 bits of the newly shifted number
        # cast them to a string and join them with '.'s
        return ".".join(
            str(
                (long >> (8*(3-i))) & 0xFF
            )
            for i in range(4)
        )


def dot_to_long(ip: str) -> int:
```

```
    """
    Take an ip address in dot notation and return the packed 32 bit int version
    i.e. dot_to_long("127.0.0.1") = 0x7F000001
    """

    # dot form ips: a.b.c.d must have each
    # part (a,b,c,d) between 0 and 255,
    # otherwise they are invalid

    parts = [int(i) for i in ip.split(".")]

    if not all(
            0 <= i <= 255
            for i in parts
    ):
        raise ValueError(f"Invalid dot form IP address: [{ip}]")

    else:
        # for each part of the dotted IP address
        # bit shift left each part by eight times
        # three minus it’s position. This puts the bits
        # from each part in the right place in the final sum
        # a.b.c.d -> a<<3*8 + b<<2*8 + c<<1*8 + d<<0*8
        return sum(
            part << ((3-i)*8)
            for i, part in enumerate(parts)
        )


@singledispatch
def is_valid_ip(ip: Union[str, int]) -> bool:
    """
    checks whether a given IP address is valid.
    """


@is_valid_ip.register
def _(ip: int):
    # this is the int overload variant of
    # the is_valid_ip function.
    try:
        # try to turn the long form ip address
        # to a dot form one, if it fails,
        # then return False, else return True
        long_to_dot(ip)
        return True
    except ValueError:
        return False

```

```python

# the type ignore comment is required to stop
# mypy exploding over the fact I have defined ‘_‘ twice.
@is_valid_ip.register # type: ignore
def _(ip: str):
    # this is the string overload variant
    # of the is_valid_ip function.
    try:
        # try to turn the dot form ip address
        # to a long form one, if it fails,
        # then return False, else return True
        dot_to_long(ip)
        return True
    except ValueError:
        return False


def is_valid_port_number(port_num: int) -> bool:
    """
    Checks whether the given port number is valid i.e. between 0 and 65536.
    """
    # port numbers must be between 0 and 65535(2^16 - 1)
    if 0 <= port_num < 2**16:
        return True
    else:
        return False


def ip_range(ip: str, network_bits: int) -> Set[str]:
    """
    Takes a Classless Inter Domain Routing(CIDR) address subnet
    specification and returns the list of addresses specified
    by the IP/network bits format.
    If the number of network bits is not between 0 and 32 it raises an error.
    If the IP address is invalid according to is_valid_ip it raises an error.
    """

    if not 0 <= network_bits <= 32:
        raise ValueError(f"Invalid number of network bits: [{network_bits}]")

    if not is_valid_ip(ip):
        raise ValueError(f"Invalid IP address: [{ip}]")
    # get the ip as long form which is useful
    # later on for using bitwise operators
    # to isolate only the constant(network) bits
    ip_long = dot_to_long(ip)
```

```python

    # generate the bit mask which specifies
    # which bits to keep and which to discard
    mask = int(
        f"{'1'*network_bits:0<32s}",
        base=2
    )
    lower_bound = ip_long & mask
    upper_bound = ip_long | (mask ^ 0xFFFFFFFF)

    # turn all the long form IP addresses between
    # the lower and upper bound into dot form
    if network_bits <= 30:
        return set(
            long_to_dot(long_ip)
            for long_ip in
            range(lower_bound+1, upper_bound)
        )
    else:
        return set(
            long_to_dot(long_ip)
            for long_ip in
            range(lower_bound, upper_bound+1)
        )



def get_local_ip() -> str:
    """
    Connects to the google.com with UDP and gets
    the IP address used to connect(the local address).
    """
    with closing(
            socket.socket(
                socket.AF_INET,
                socket.SOCK_DGRAM
            )
    ) as s:
        s.connect(("google.com", 80))
        ip, _ = s.getsockname()
    return ip


def get_free_port() -> int:
    """
    Attempts to bind to port 0 which assigns a free port number to the socket,
    the socket is then closed and the port number assigned is returned.
    """

```

```python
    with closing(
            socket.socket(
                socket.AF_INET,
                socket.SOCK_STREAM
            )
    ) as s:
        s.bind(('', 0))
        _, port = s.getsockname()
    return port


def ip_checksum(packet: bytes) -> int:
    """
    ip_checksum function takes in a packet
    and returns the checksum.
    """
    if len(packet) % 2 == 1:
        # if the length of the packet is even, add a NULL byte
        # to the end as padding
        packet += b"\0"

    total = 0
    for first, second in (
            packet[i:i+2]
            for i in range(0, len(packet), 2)
    ):
        total += (first << 8) + second

    # calculate the number of times a
    # carry bit was added and add it back on
    carried = (total - (total & 0xFFFF)) >> 16
    total &= 0xFFFF
    total += carried

    if total > 0xFFFF:
        # adding the carries generated a carry
        total &= 0xFFFF

    # invert the checksum and take the last 16 bits.
    return (~total & 0xFFFF)


def make_icmp_packet(ID: int) -> bytes:
    """
    Takes an argument of the process ID of the calling process.
    Returns an ICMP ECHO REQUEST packet created with this ID
    """

    ICMP_ECHO_REQUEST = 8
    # pack the information for the dummy header needed
```

```python
    # for the IP checksum
    dummy_header = struct.pack(
        "bbHHh",
        ICMP_ECHO_REQUEST,
        0,
        0,
        ID,
        1
    )
    # pack the current time into a double
    time_bytes = struct.pack("d", time.time())
    # define the bytes to repeat in the data section of the packet
    # this makes the packets easily identifiable in packet captures.
    bytes_to_repeat_in_data = map(ord, " y33t ")
    # calculate the number of bytes left for data
    data_bytes = (192 - struct.calcsize("d"))
    # first pack the current time into the start of the data section
    # the pack the identifiable data into the rest
    data = (
        time_bytes +
        bytes(islice(cycle(bytes_to_repeat_in_data), data_bytes))
    )
    # get the IP checksum for the dummy header and data
    # and switch the bytes into the order expected by the network
    checksum = socket.htons(ip_checksum(dummy_header + data))
    # pack the header with the correct checksum and information
    header = struct.pack(
        "bbHHh",
        ICMP_ECHO_REQUEST,
        0,
        checksum,
        ID,
        1
    )
    # concatonate the header bytes and the data bytes
    return header + data


def make_tcp_packet(
        src: int,
        dst: int,
        from_address: str,
        to_address: str,
        flags: int) -> bytes:
    """
    Takes in the source and destination port/ip address
    returns a tcp packet.
    flags:
    2 => SYN
    18 => SYN:ACK
```

```python
    4 => RST
    """
    # validate that the information passed in is valid
    if flags not in {2, 18, 4}:
        raise ValueError(
            f"Flags must be one of 2:SYN, 18:SYN,ACK, 4:RST. not:
                [{flags}]"
        )
    if not is_valid_ip(from_address):
        raise ValueError(
            f"Invalid source IP address: [{from_address}]"
        )
    if not is_valid_ip(to_address):
        raise ValueError(
            f"Invalid destination IP address: [{to_address}]"
        )
    if not is_valid_port_number(src):
        raise ValueError(
            f"Invalid source port: [{src}]"
        )
    if not is_valid_port_number(dst):
        raise ValueError(
            f"Invalid destination port: [{dst}]"
        )
    # turn the ip addresses into long form
    src_addr = dot_to_long(from_address)
    dst_addr = dot_to_long(to_address)

    seq = ack = urg = 0
    data_offset = 6 << 4
    window_size = 1024
    max_segment_size = (2, 4, 1460)
    # pack the dummy header needed for the checksum calculation
    dummy_header = struct.pack(
        "!HHIIBBHHHBBH",
        src,
        dst,
        seq,
        ack,
        data_offset,
        flags,
        window_size,
        0,
        urg,
        *max_segment_size
    )
    # pack the psuedo header that is also needed for the checksum
    # just because TCP and why not
    psuedo_header = struct.pack(
        "!IIBBH",
```

```
        src_addr,
        dst_addr,
        0,
        6,
        len(dummy_header)
    )

    checksum = ip_checksum(psuedo_header + dummy_header)
    # pack the final TCP packet with the relevant data and checksum
    return struct.pack(
        "!HHIIBBHHHBBH",
        src,
        dst,
        seq,
        ack,
        data_offset,
        flags,
        window_size,
        checksum,
        urg,
        *max_segment_size
    )


def make_udp_packet(
        src: int,
        dst: int
) -> bytes:
    """
    Takes in: source IP address and port, destination IP address and port.
    Returns: a UDP packet with those properties.
    the IP addresses are needed for calculating the checksum.
    """
    # validate data passed in
    if not is_valid_port_number(src):
        raise ValueError(
            f"Invalid source port: [{src}]"
        )
    if not is_valid_port_number(dst):
        raise ValueError(
            f"Invalid destination port: [{dst}]"
        )
    data = b"Most services don't respond to an empty data field"
    # pack the data
    # and return the packed bytes
    # UDP checksum is optional over IPv4
    return struct.pack(
        "!HHHH",
        src,
```

```
        dst,
        8+len(data),
        0
    ) + data


def wait_for_socket(sock: socket.socket, wait_time: float) -> float:
    """
    Wait for wait_time seconds or until the socket is readable.
    If the socket is readable return a tuple of the socket and the time
        taken
    otherwise return None.
    """

    start = time.time()
    is_socket_readable = select.select([sock], [], [], wait_time)
    taken = time.time() - start
    if is_socket_readable[0] == []:
        return float(-1)
    else:
        return taken
```

Listing 24: A python module I made to hold all of the listeners I had made for each of the different scanning types.

```
from modules import headers
from modules import ip_utils
import socket
import struct
import time
from collections import defaultdict
from contextlib import closing
from typing import Tuple, Set, DefaultDict


PORTS = DefaultDict[str, Set[int]]


def ping(
        ID: int,
        timeout: float
) -> Set[Tuple[str, float, headers.ip]]:
    """
    Takes in a process id and a timeout and returns
    a list of addresses which sent ICMP ECHO REPLY
    packets with the packed id matching ID in the time given by timeout.
    """
    ping_sock = socket.socket(
        socket.AF_INET,
        socket.SOCK_RAW,
```

```python
        socket.IPPROTO_ICMP)
    # opens a raw socket for sending ICMP protocol packets
    time_remaining = timeout
    addresses = set()
    recieved_from = set()
    while True:
        time_waiting = ip_utils.wait_for_socket(ping_sock, time_remaining)
        # time_waiting stores the time the socket took to become readable
    # or returns minus one if it ran out of time

        if time_waiting == -1:
            break
        time_recieved = time.time()
        # store the time the packet was recieved
        recPacket, addr = ping_sock.recvfrom(1024)
        # recieve the packet
        ip = headers.ip(recPacket[:20])
        # unpack the IP header into its respective components
        icmp = headers.icmp(recPacket[20:28])
        # unpack the time from the packet.
        time_sent = struct.unpack(
            "d",
            recPacket[28:28 + struct.calcsize("d")]
        )[0]
        # unpack the value for when the packet was sent
        time_taken: float = time_recieved - time_sent
        # calculate the round trip time taken for the packet
        if icmp.id == ID:
            # if the ping was sent from this machine then add it to the list of
            # responses
            ip_address, port = addr
            # this is to prevent a bug where IPs were being added twice
            if ip_address not in recieved_from:
                addresses.add((ip_address, time_taken, ip))
                recieved_from.add(ip_address)
        elif time_remaining <= 0:
            break
        else:
            continue
    # return a list of all the addesses that replied to our ICMP echo request.
    return addresses


def udp(dest_ip: str, timeout: float) -> Set[int]:
    """
    This listener detects UDP packets from dest_ip in the given timespan,
    all ports that send direct replies are marked as being open.
```

```
    Returns a list of open ports.
    """

    time_remaining = timeout
    ports: Set[int] = set()
    with socket.socket(
            socket.AF_INET,
            socket.SOCK_RAW,
            socket.IPPROTO_UDP
    ) as s:
        while True:
            time_taken = ip_utils.wait_for_socket(s, time_remaining)
            if time_taken == -1:
                break
            else:
                time_remaining -= time_taken
            packet = s.recv(1024)
            ip = headers.ip(packet[:20])
            udp = headers.udp(packet[20:28])
            if dest_ip == ip.source and ip.protocol == 17:
                ports.add(udp.src)

    return ports


def icmp_unreachable(src_ip: str, timeout: float = 2) -> int:
    """
    This listener detects ICMP destination unreachable
    packets and returns the icmp code.
    This is later used to mark them as either close, open|filtered, filtered.
    3 -> closed
    0|1|2|9|10|13 -> filtered
    -1 -> error with arguments
    open|filtered means that they are either open or
    filtered but return nothing.
    """

    ping_sock = socket.socket(
        socket.AF_INET,
        socket.SOCK_RAW,
        socket.IPPROTO_ICMP
    )
    # open raw socket to listen for ICMP destination unrechable packets
    time_remaining = timeout
    code = -1
    while True:
        time_waiting = ip_utils.wait_for_socket(ping_sock, time_remaining)
        # wait for socket to be readable
```

```
        if time_waiting == -1:
            break
        else:
            time_remaining -= time_waiting
        recPacket, addr = ping_sock.recvfrom(1024)
        # recieve the packet
        ip = headers.ip(recPacket[:20])
        icmp = headers.icmp(recPacket[20:28])
        valid_codes = [0, 1, 2, 3, 9, 10, 13]
        if (
                ip.source == src_ip
                and icmp.type == 3
                and icmp.code in valid_codes
        ):
            code = icmp.code
            break
        elif time_remaining <= 0:
            break
        else:
            continue
    ping_sock.close()
    return code


def tcp(address: Tuple[str, int], timeout: float) -> PORTS:
    """
    This function is run asynchronously and listens for
    TCP ACK responses to the sent TCP SYN msg.
    """
    ports: DefaultDict[str, Set[int]] = defaultdict(set)
    with closing(
            socket.socket(
                socket.AF_INET,
                socket.SOCK_RAW,
                socket.IPPROTO_TCP
            )) as s:
        s.bind(address)
        # bind the raw socket to the listening address
        time_remaining = timeout
        while True:
            time_taken = ip_utils.wait_for_socket(s, time_remaining)
            # wait for the socket to become readable
            if time_taken == -1:
                break
            else:
                time_remaining -= time_taken
            packet = s.recv(1024)
            # recieve the packet data
            tcp = headers.tcp(packet[20:40])
            if tcp.flags & 2: # syn flags set
```

```
                ports["OPEN"].add(tcp.source)
            elif tcp.flags & 4:
                ports["CLOSED"].add(tcp.source)
            else:
                continue
    return ports
```

Listing 25: A python module I made to hold all of the scanners I had made for each of the different scanning types.

```
import socket
import time
from modules import directives
from modules import headers
from modules import ip_utils
from modules import listeners
from collections import defaultdict
from contextlib import closing
from itertools import repeat
from multiprocessing import Pool
from os import getpid
from typing import Set, Tuple


def ping(addresses: Set[str]) -> Set[Tuple[str, float, headers.ip]]:
    """
    Send an ICMP ECHO REQUEST to each address
    in the set addresses. Then return a set which
    contains all the addresses which replied and
    which have the correct ID.
    """
    with closing(
            socket.socket(
                socket.AF_INET,
                socket.SOCK_RAW,
                socket.IPPROTO_ICMP
            )
    ) as ping_sock:
        # get the local ip address
        addresses = {
            ip
            for ip in addresses
            if (
                not ip.endswith(".0")
                and not ip.endswith(".255")
            )
        }

        # initialise a process pool
        p = Pool(1)
```

```python
        # get the local process id for use in creating packets.
        ID = getpid() & 0xFFFF
        # run the listeners.ping function asynchronously
        replied = p.apply_async(listeners.ping, (ID, 5))
        time.sleep(0.01)
        for address in zip(addresses, repeat(1)):
            try:
                packet = ip_utils.make_icmp_packet(ID)
                ping_sock.sendto(packet, address)
            except PermissionError:
                ip_utils.eprint("raw sockets require root priveleges, exiting")
                exit()
        p.close()
        p.join()
        # close and join the process pool to so that all the values
        # have been returned and the pool closed
        return replied.get()


def connect(address: str, ports: Set[int]) -> Set[int]:
    """
    This is the most basic kind of scan
    it simply connects to every specifed port
    and identifies whether they are open.
    """
    import socket
    from contextlib import closing
    open_ports: Set[int] = set()
    for port in ports:
        # loop through each port in the list of ports to scan
        try:
            with closing(
                    socket.socket(
                        socket.AF_INET,
                        socket.SOCK_STREAM
                    )
            ) as s:
                # open an IPV4 TCP socket
                s.connect((address, port))
                # attempt to connect the newly created socket to the target
                # address and port
                open_ports.add(port)
                # if the connection was successful then add the port to the
                # list of open ports
        except (ConnectionRefusedError, OSError) as e:
            pass
    return open_ports
```

```python


def tcp(dest_ip: str, portlist: Set[int]) -> listeners.PORTS:
    src_port = ip_utils.get_free_port()
    # request a local port to connect from
    if "127.0.0.1" == dest_ip:
        local_ip = "127.0.0.1"
    else:
        local_ip = ip_utils.get_local_ip()
    p = Pool(1)
    listener = p.apply_async(listeners.tcp, ((local_ip, src_port), 5))
    time.sleep(0.01)
    # start the TCP ACK listener in the background
    for port in portlist:
        # flag = 2 for syn scan
        packet = ip_utils.make_tcp_packet(
            src_port,
            port,
            local_ip,
            dest_ip,
            2
        )
        with closing(
                socket.socket(
                    socket.AF_INET,
                    socket.SOCK_RAW,
                    socket.IPPROTO_TCP
                )
        ) as s:
            s.sendto(packet, (dest_ip, port))
            # send the packet to its destination
    p.close()
    p.join()
    ports = listener.get()
    ports["FILTERED"] = portlist - ports["OPEN"] - ports["CLOSED"]
    if local_ip == "127.0.0.1":
        ports["OPEN"] -= set([src_port])

    return ports


def udp(
        dest_ip: str,
        ports_to_scan: Set[int]
) -> listeners.PORTS:
    """
    Takes in a destination IP address in either dot or long form and
    a list of ports to scan. Sends UDP packets to each port specified
    in portlist and uses the listeners to mark them as open,
        open|filtered,
```

```python
    filtered, closed they are marked open|filtered if no response is
    recieved at all.
    """

    local_port = ip_utils.get_free_port()
    # get port number
    ports: listeners.PORTS = defaultdict(set)
    ports["REMAINING"] = ports_to_scan
    p = Pool(1)
    udp_listen = p.apply_async(listeners.udp, (dest_ip, 4))
    time.sleep(0.01)
    # start the UDP listener
    with closing(
            socket.socket(
                socket.AF_INET,
                socket.SOCK_RAW,
                socket.IPPROTO_UDP
            )
    ) as s:
        for _ in range(2):
            # repeat 3 times because UDP scanning comes
            # with a high chance of packet loss
            for dest_port in ports["REMAINING"]:
                try:
                    packet = ip_utils.make_udp_packet(
                        local_port,
                        dest_port
                    )
                    # create the UDP packet to send
                    s.sendto(packet, (dest_ip, dest_port))
                    # send the packet to the currently scanning address
                except socket.error:
                    packet_bytes = " ".join(map(hex, packet))
                    print(
                        "The socket modules sendto method with the following",
                        "argument resulting in a socket error.",
                        f"\npacket: [{packet_bytes}]\n",
                        "address: [{dest_ip, dest_port}])"
                    )

    p.close()
    p.join()

    ports["OPEN"].update(udp_listen.get())
    # if we are on localhost remove the scanning port
    if dest_ip == "127.0.0.1":
        ports["OPEN"] -= set([local_port])
    ports["REMAINING"] -= ports["OPEN"]
    # only scan the ports which we know are not open
```

```python
    with closing(
            socket.socket(
                socket.AF_INET,
                socket.SOCK_RAW,
                socket.IPPROTO_UDP
            )
    ) as s:
        for dest_port in ports["REMAINING"]:
            try:
                packet = ip_utils.make_udp_packet(
                    local_port,
                    dest_port
                )
                # make a new UDP packet
                p = Pool(1)
                icmp_listen = p.apply_async(
                    listeners.icmp_unreachable,
                    (dest_ip,),
                )
                # start the ICMP listener
                time.sleep(0.01)
                s.sendto(packet, (dest_ip, dest_port))
                # send packet
                p.close()
                p.join()
                icmp_code = icmp_listen.get()
                # receive ICMP code from the ICMP listener
                if icmp_code in {0, 1, 2, 9, 10, 13}:
                    ports["FILTERED"].add(dest_port)
                elif icmp_code == 3:
                    ports["CLOSED"].add(dest_port)
            except socket.error:
                packet_bytes = " ".join(map("{:02x}".format, packet))
                ip_utils.eprint(
                    "The socket modules sendto method with the following",
                    "argument resulting in a socket error.",
                    f"\npacket: [{packet_bytes}]\n",
                    "address: [{dest_ip, dest_port}])"
                )
    # this creates a new set which contains all the elements that
    # are in the list of ports to be scanned but have not yet
    # been classified
    ports["OPEN|FILTERED"] = (
        ports["REMAINING"]
        - ports["OPEN"]
        - ports["FILTERED"]
        - ports["CLOSED"]
    )
    del(ports["REMAINING"])
    # set comprehension to update the list of open filtered ports
```

```python
    return ports


def version_detect_scan(
        target: directives.Target,
        probes: directives.PROBE_CONTAINER
) -> directives.Target:
    for probe_dict in probes.values():
        for proto in probe_dict:
            target = probe_dict[proto].scan(target)
    return target
```

A.8 examples

Listing 26: A program I wrote to run all of the example scripts I made from one main script to solve the issue of the PATH being used for determining import when I could use Pythons built in module structure instead.

```python
#!/usr/bin/env python
from icmp_ping import icmp_echo_recv, icmp_echo_send
from ping_scanner import ping_scan
from tcp_scan.connect_scan import scan_port_list as connect_scan_list
from tcp_scan.syn_scan import scan_port_list as syn_scan_list
from udp_scan import scan_port_list as udp_scan_list
from version_detection import version_detection

examples = {
    "icmp_echo_recv": icmp_echo_recv.main,
    "icmp_echo_send": icmp_echo_send.main,
    "ping_scanner": ping_scan.main,
    "connect_scan": connect_scan_list.main,
    "syn_scan": syn_scan_list.main,
    "udp_scan": udp_scan_list.main,
    "version_detection": version_detection.main,
}

print("\n\t".join(("Programs:", *examples)))

while True:
    print()
    program = input("Enter the name of the example program to run: ")
    if program.lower() in {"quit", "q", "end", "exit"}:
        break
    found = False
    for name in examples:
        if name.startswith(program.lower()):
            program = name
            print(f"Running: {program}")
            examples[program]()
```

```python
            found = True
    if not found:
        print(
            "The program name must exactly match one of the following
                examples"
        )
        print("\n".join(examples))
```

A.9 netscan

Listing 27: The program which provides the command line user interface for my projects functionality.

```python
#!/usr/bin/env python
import re
from argparse import ArgumentParser
from collections import defaultdict
from math import floor, log10
from modules import (
    scanners,
    ip_utils,
    directives,
)
from typing import (
    DefaultDict,
    Dict,
)

top_ports = directives.parse_ports(open("top_ports").read())
services: DefaultDict[str, Dict[int, str]] = defaultdict(dict)
for match in re.finditer(
        r"(\S+)\s+(\d+)/(\S+)",
        open("version_detection/nmap-services").read()
):
    service, portnum, protocol = match.groups()
    services[protocol.upper()][int(portnum)] = service

parser = ArgumentParser()
parser.add_argument(
    "target_spec",
    help="specify what to scan, i.e. 192.168.1.0/24"
)
parser.add_argument(
    "-Pn",
    help="assume hosts are up",
    action="store_true"
)
parser.add_argument(
    "-sL",
```

```
    help="list targets",
    action="store_true"
)
parser.add_argument(
    "-sn",
    help="disable port scanning",
    action="store_true"
)
parser.add_argument(
    "-sS",
    help="TCP SYN scan",
    action="store_true"
)
parser.add_argument(
    "-sT",
    help="TCP connect scan",
    action="store_true"
)
parser.add_argument(
    "-sU",
    help="UDP scan",
    action="store_true"
)
parser.add_argument(
    "-sV",
    help="version scan",
    action="store_true"
)
parser.add_argument(
    "-p",
    "--ports",
    help="scan specified ports",
    required=False,
    default=top_ports
)
parser.add_argument(
    "--exclude_ports",
    help="ports to exclude from the scan",
    required=False,
    default=""
)

args = parser.parse_args()

# check whether the address spec is in CIDR form
CIDR_regex =
    re.compile(r"(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})/(\d{1,2})")
search = CIDR_regex.search(args.target_spec)
if search:
    base_addr, network_bits = search.groups()
```

```python
    addresses = ip_utils.ip_range(
        base_addr,
        int(network_bits)
    )
else:
    base_addr = args.target_spec
    addresses = {base_addr}


def error_exit(error_type: str, scan_type: str, scanning: str) -> bool:
    messages = {
        "permission": "\n".join((
            "You have insufficient permissions to run this type of scan",
            "EXITING!"
        ))
    }
    print(f"You tried to scan {scanning} using scan type: {scan_type}")
    try:
        print(messages[error_type])
    except KeyError:
        print(f"ERROR MESSAGE NOT FOUND: {error_type}")
    exit(-1)


if args.sL:
    print("Targets:")
    print("\n".join(sorted(addresses, key=ip_utils.dot_to_long)))
else:
    if args.sn:
        def sig_figs(x: float, n: int) -> float:
            """
            rounds x to n significant figures.
            sig_figs(1234, 2) = 1200.0
            """
            return round(x, n - (1 + int(floor(log10(abs(x))))))

        try:
            print("\n".join(
                f"host: [{host}]\t" +
                "responded to an ICMP ECHO REQUEST in " +
                f"{str(sig_figs(taken, 2))+'s':<10s} " +
                f"ttl: [{ip_head.time_to_live}]"
                for host, taken, ip_head in scanners.ping(addresses)
            ))
        except PermissionError:
            error_exit("permission", "ping scan", str(addresses))

    else:
        if args.Pn:
            targets = [
```

```
                directives.Target(
                    addr,
                    defaultdict(set),
                    defaultdict(set)
                )
                for addr in addresses
            ]
        else:
            try:
                targets = [
                    directives.Target(
                        addr,
                        defaultdict(set),
                        defaultdict(set),
                    )
                    for addr, _, _ in scanners.ping(addresses)
                ]
            except PermissionError:
                error_exit("permission", "ping_scan", str(addresses))
        # define the ports to scan
        if args.ports == "-":
            # case they have specified all ports
            ports = {
                "UDP": set(range(1, 65536)),
                "TCP": set(range(1, 65536)),
            }
        elif isinstance(args.ports, str):
            # case they have specifed ports
            ports = directives.parse_ports(args.ports)
        else:
            # default
            ports = args.ports

        # exclude all the ports speified to be excluded
        to_exclude = directives.parse_ports(args.exclude_ports)
        ports["TCP"] -= to_exclude["TCP"]
        ports["TCP"] -= to_exclude["ANY"]
        ports["UDP"] -= to_exclude["UDP"]
        ports["UDP"] -= to_exclude["ANY"]

        # if version scanning is desired
        if args.sV:
            probes = directives.parse_probes(
                "./version_detection/nmap-service-probes"
            )

        for target in targets:
            if not args.sU and not args.sT or args.sS:
                try:
                    tcp_ports = scanners.tcp(
```

```
                    target.address,
                    ports["TCP"] | ports["ANY"]
                )
            except PermissionError:
                error_exit("permission", "tcp_scan", target.address)
            target.open_ports["TCP"].update(tcp_ports["OPEN"])
            target.open_filtered_ports["TCP"].update(tcp_ports["FILTERED"])
        if args.sT:
            target.open_ports["TCP"].update(
                scanners.connect(
                    target.address,
                    ports["TCP"] | ports["ANY"]
                )
            )
        if args.sU:
            try:
                udp_ports = scanners.udp(
                    target.address,
                    ports["UDP"] | ports["ANY"]
                )
            except PermissionError:
                error_exit("permission", "udp_scan", target.address)

            target.open_ports["UDP"].update(
                udp_ports["OPEN"]
            )
            target.open_filtered_ports["UDP"].update(
                udp_ports["FILTERED"]
            )
            target.open_filtered_ports["UDP"].update(
                udp_ports["OPEN|FILTERED"]
            )
        if args.sV:
            target = scanners.version_detect_scan(target, probes)
        # display scan info
        print()
        print(f"Scan report for: {target.address}")
        #  print(target)
        print("Open ports:")
        for proto, open_ports in target.open_ports.items():
            for port in open_ports:
                try:
                    service_name = services[proto][port]
                except KeyError:
                    service_name = "unknown"
                if port in target.services:
                    exact_match = target.services[port]
                    print(
                        f"{port}/{proto}{exact_match.service:>8s}"
                    )
```

```
                        # print version information
                        for key, val in exact_match.version_info.items():
                            print(f"{key}: {val}")
                        if exact_match.cpes:
                            print()
                            print("CPE:")
                            for cpe_type, cpe_vals in exact_match.cpes.items():
                                print(cpe_type)
                                try:
                                    del(cpe_vals["part"])
                                except KeyError:
                                    pass
                                for key, val in cpe_vals.items():
                                    print(f"{key}: {val}")
                        print()
                    else:
                        print(f"{port} service: {service_name}?")

            print("Filtered ports:")
            for proto, filtered_ports in target.open_filtered_ports.items():
                for port in filtered_ports:
                    try:
                        service_name = services[proto][port]
                    except KeyError:
                        service_name = "unknown"
                    print(f"{port} service: {service_name}?")
```

A.10 tests

Listing 28: Unit tests I wrote for the ip_utils module.

```
from modules.ip_utils import (
    dot_to_long,
    long_to_dot,
    ip_range,
    is_valid_ip,
    is_valid_port_number,
    ip_checksum,
    make_tcp_packet,
    make_udp_packet,
    make_icmp_packet,
)
from binascii import unhexlify


def test_dot_to_long_private_ip() -> None:
    assert(dot_to_long("192.168.1.0") == 0xC0A80100)
```

```python


def test_long_to_dot_private_ip() -> None:
    assert(long_to_dot(0xC0A80100) == "192.168.1.0")


def test_dot_to_long_localhost() -> None:
    assert(dot_to_long("127.0.0.1") == 0x7F000001)


def test_long_to_dot_localhost() -> None:
    assert(long_to_dot(0x7F000001) == "127.0.0.1")


def test_is_valid_ip_localhost_long() -> None:
    assert is_valid_ip(0x7F000001)


def test_is_valid_ip_localhost() -> None:
    assert is_valid_ip("127.0.0.1")


def test_is_not_valid_ip_5_zeros_dotted() -> None:
    assert not is_valid_ip("0.0.0.0.0")


def test_is_not_valid_ip_5_255s_long() -> None:
    assert not is_valid_ip(0xFF_FF_FF_FF_FF)


def test_is_valid_port_number_0() -> None:
    assert is_valid_port_number(0)


def test_is_valid_port_number_65535() -> None:
    assert is_valid_port_number(65535)


def test_is_not_valid_port_number_negative_one() -> None:
    assert not is_valid_port_number(-1)


def test_is_not_valid_port_number_65536() -> None:
    assert not is_valid_port_number(65536)


def test_ip_range() -> None:
    assert(
        ip_range("192.168.1.0", 28) == {
            "192.168.1.1",
```

```python
            "192.168.1.2",
            "192.168.1.3",
            "192.168.1.4",
            "192.168.1.5",
            "192.168.1.6",
            "192.168.1.7",
            "192.168.1.8",
            "192.168.1.9",
            "192.168.1.10",
            "192.168.1.11",
            "192.168.1.12",
            "192.168.1.13",
            "192.168.1.14",
        }
    )


def test_ip_checksum_verify() -> None:
    packet = unhexlify(
        "45000073000040004011b861c0a80001c0a800c7"
    )
    assert ip_checksum(packet) == 0


def test_ip_checksum_generate() -> None:
    packet = unhexlify(
        "450000730000400040110000c0a80001c0a800c7"
    )
    assert ip_checksum(packet) == 0xB861


def test_make_tcp_packet() -> None:
    correct = unhexlify(
        "e54700500000000000000000600204002af50000020405b4"
    )
    info = 58695, 80, "192.168.1.45", "192.168.1.28", 2
    assert correct == make_tcp_packet(*info)


def test_make_udp_packet() -> None:
    correct = unhexlify(
        "e5470050003a0000"
    )
    info = 58695, 80
    # clipping the packet at 8 simply removes the data section
    assert correct == make_udp_packet(*info)[:8]
```

Listing 29: Unit tests I wrote for the directives module.

```python
from modules.directives import (
```

```python
    parse_ports
)
from collections import defaultdict
from typing import DefaultDict


def test_parse_probes_single() -> None:
    portstring = "12345"
    expected: DefaultDict[str, set] = defaultdict(set)
    expected["ANY"] = set([12345])
    assert expected == parse_ports(portstring)


def test_parse_probes_range() -> None:
    portstring = "10-20"
    expected: DefaultDict[str, set] = defaultdict(set)
    expected["ANY"] = set(range(10, 21))
    assert expected == parse_ports(portstring)


def test_parse_probes_single_and_range() -> None:
    portstring = "1,2,3,10-20,6,7,8"
    expected: DefaultDict[str, set] = defaultdict(set)
    expected["ANY"] = set([1, 2, 3, *range(10, 21), 6, 7, 8])
    assert expected == parse_ports(portstring)


def test_parse_probes_tcp_single() -> None:
    portstring = "T:12345"
    expected: DefaultDict[str, set] = defaultdict(set)
    expected["TCP"] = set([12345])
    assert expected == parse_ports(portstring)


def test_parse_probes_tcp_range() -> None:
    portstring = "T:10-20"
    expected: DefaultDict[str, set] = defaultdict(set)
    expected["TCP"] = set(range(10, 21))
    assert expected == parse_ports(portstring)


def test_parse_probes_tcp_single_and_range() -> None:
    portstring = "T:1,2,3,10-20,6,7,8"
    expected: DefaultDict[str, set] = defaultdict(set)
    expected["TCP"] = set([1, 2, 3, *range(10, 21), 6, 7, 8])
    assert expected == parse_ports(portstring)


def test_parse_probes_udp_single() -> None:
    portstring = "U:12345"
```

```python
    expected: DefaultDict[str, set] = defaultdict(set)
    expected["UDP"] = set([12345])
    assert expected == parse_ports(portstring)


def test_parse_probes_udp_range() -> None:
    portstring = "U:10-20"
    expected: DefaultDict[str, set] = defaultdict(set)
    expected["UDP"] = set(range(10, 21))
    assert expected == parse_ports(portstring)


def test_parse_probes_udp_single_and_range() -> None:
    portstring = "U:1,2,3,10-20,6,7,8"
    expected: DefaultDict[str, set] = defaultdict(set)
    expected["UDP"] = set([1, 2, 3, *range(10, 21), 6, 7, 8])
    assert expected == parse_ports(portstring)


def test_parse_probes_any_and_tcp_single() -> None:
    portstring = "12345 T:12345"
    expected: DefaultDict[str, set] = defaultdict(set)
    expected["TCP"] = set([12345])
    expected["ANY"] = set([12345])
    assert expected == parse_ports(portstring)


def test_parse_probes_any_and_tcp_range() -> None:
    portstring = "10-20 T:10-20"
    expected: DefaultDict[str, set] = defaultdict(set)
    expected["TCP"] = set(range(10, 21))
    expected["ANY"] = set(range(10, 21))
    assert expected == parse_ports(portstring)


def test_parse_probes_any_and_tcp_single_and_range() -> None:
    portstring = "1,2,3,10-20,6,7,8 T:1,2,3,10-20,6,7,8"
    expected: DefaultDict[str, set] = defaultdict(set)
    expected["TCP"] = set([1, 2, 3, *range(10, 21), 6, 7, 8])
    expected["ANY"] = set([1, 2, 3, *range(10, 21), 6, 7, 8])
    assert expected == parse_ports(portstring)


def test_parse_probes_any_and_udp_single() -> None:
    portstring = "12345 U:12345"
    expected: DefaultDict[str, set] = defaultdict(set)
    expected["UDP"] = set([12345])
    expected["ANY"] = set([12345])
    assert expected == parse_ports(portstring)

```

```python
def test_parse_probes_any_and_udp_range() -> None:
    portstring = "10-20 U:10-20"
    expected: DefaultDict[str, set] = defaultdict(set)
    expected["UDP"] = set(range(10, 21))
    expected["ANY"] = set(range(10, 21))
    assert expected == parse_ports(portstring)


def test_parse_probes_any_and_udp_single_and_range() -> None:
    portstring = "1,2,3,10-20,6,7,8 U:1,2,3,10-20,6,7,8"
    expected: DefaultDict[str, set] = defaultdict(set)
    expected["UDP"] = set([1, 2, 3, *range(10, 21), 6, 7, 8])
    expected["ANY"] = set([1, 2, 3, *range(10, 21), 6, 7, 8])
    assert expected == parse_ports(portstring)


def test_parse_probes_udp_and_tcp_single() -> None:
    portstring = "U:12345 T:12345"
    expected: DefaultDict[str, set] = defaultdict(set)
    expected["TCP"] = set([12345])
    expected["UDP"] = set([12345])
    assert expected == parse_ports(portstring)


def test_parse_probes_udp_and_tcp_range() -> None:
    portstring = "U:10-20 T:10-20"
    expected: DefaultDict[str, set] = defaultdict(set)
    expected["TCP"] = set(range(10, 21))
    expected["UDP"] = set(range(10, 21))
    assert expected == parse_ports(portstring)


def test_parse_probes_udp_and_tcp_single_and_range() -> None:
    portstring = "U:1,2,3,10-20,6,7,8 T:1,2,3,10-20,6,7,8"
    expected: DefaultDict[str, set] = defaultdict(set)
    expected["TCP"] = set([1, 2, 3, *range(10, 21), 6, 7, 8])
    expected["UDP"] = set([1, 2, 3, *range(10, 21), 6, 7, 8])
    assert expected == parse_ports(portstring)


def test_parse_probes_all_single() -> None:
    portstring = "12345 U:12345 T:12345"
    expected: DefaultDict[str, set] = defaultdict(set)
    expected["TCP"] = set([12345])
    expected["UDP"] = set([12345])
    expected["ANY"] = set([12345])
    assert expected == parse_ports(portstring)
```

```python
def test_parse_probes_all_range() -> None:
    portstring = "10-20 U:10-20 T:10-20"
    expected: DefaultDict[str, set] = defaultdict(set)
    expected["TCP"] = set(range(10, 21))
    expected["UDP"] = set(range(10, 21))
    expected["ANY"] = set(range(10, 21))
    assert expected == parse_ports(portstring)


def test_parse_probes_all_single_and_range() -> None:
    portstring = "1,2,3,10-20,6,7,8 U:1,2,3,10-20,6,7,8
        T:1,2,3,10-20,6,7,8"
    expected: DefaultDict[str, set] = defaultdict(set)
    expected["TCP"] = set([1, 2, 3, *range(10, 21), 6, 7, 8])
    expected["UDP"] = set([1, 2, 3, *range(10, 21), 6, 7, 8])
    expected["ANY"] = set([1, 2, 3, *range(10, 21), 6, 7, 8])
    assert expected == parse_ports(portstring)
```

References

[1] Anonymous. IPv4 header checksum. https://en.wikipedia.org/wiki/IPv4_header_checksum, April 2019.

[2] brice. Python raw sockets. https://stackoverflow.com/questions/1117958/how-do-i-use-raw-socket-in-python, June 2011.

[3] Linux Developers. ICMP Man page. https://linux.die.net/man/7/icmp, March 2019.

[4] Linux Developers. IP Man page. https://linux.die.net/man/7/ip, March 2019.

[5] Linux Developers. TCP Man page. https://linux.die.net/man/7/tcp, March 2019.

[6] Linux Developers. UDP Man page. https://linux.die.net/man/7/udp, March 2019.

[7] Python Core Developers. builtin data structures documentation. https://docs.python.org/3/tutorial/datastructures.html#data-structures, April 2019.

[8] Python Core Developers. command line argument parsing documentation. https://docs.python.org/3/library/argparse.html?highlight=typing, April 2019.

[9] Python Core Developers. defaultdict documentation. https://docs.python.org/3/library/collections.html?highlight=collection#collections.defaultdict, April 2019.

[10] Python Core Developers. multiprocessing documentation. https://docs.python.org/3/library/multiprocessing.html, April 2019.

[11] Python Core Developers. operator documentation. https://docs.python.org/3/library/operator.html, April 2019.

[12] Python Core Developers. Socket module documentation. https://docs.python.org/3/library/socket.html, April 2019.

[13] Python Core Developers. stderr documentation. https://docs.python.org/3/library/sys.html?highlight=stderr#sys.stderr, April 2019.

[14] Python Core Developers. struct documentation. https://docs.python.org/3/library/struct.html, April 2019.

[15] Python Core Developers. type hinting documentation. https://docs.python.org/3/library/typing.html?highlight=typing#module-typing, April 2019.

[16] J. Postel ISI. User Datagram Protocol. https://www.ietf.org/rfc/rfc768.txt, August 1980.

[17] J. Postel ISI. Internet Control Message Protocol. https://tools.ietf.org/html/rfc792, September 1981.

[18] Joe. send icmp echo request. https://stackoverflow.com/questions/24575524/send-icmp-echo-request, July 2014.

[19] Gordon 'Fyodor' Lyon. port scanning techniques. https://nmap.org/book/man-port-scanning-techniques.html, January 2001.

[20] Gordon 'Fyodor' Lyon. service and version detection file format. https://nmap.org/book/vscan-fileformat.html, January 2001.

[21] Gordon 'Fyodor' Lyon. service and version detection techniques. https://nmap.org/book/man-version-detection.html, January 2001.

[22] Gordon 'Fyodor' Lyon. service and version detection techniques described. https://nmap.org/book/vscan-technique.html, January 2001.

[23] Information Sciences Institute University of Southern California. Internet Protocol. https://tools.ietf.org/html/rfc791, September 1981.

[24] Information Sciences Institute University of Southern California. Transmission Control Protocol. https://tools.ietf.org/html/rfc793, September 1981.

Glossary

API Applications Programming Interface 4, 27

ARP Address Resolution Protocol 53, 54

banner A short piece of text which a service with send to identify itself when it receives a connection request. Often contains information such as version number etc... 24

black box Looking at something from an outsider's perspective knowing nothing about how it works internally. 3, 18

checksum A checksum is a value calculated from a mathematical algorithm which is sent with the packet to its destination to allow the recipient to check whether the packet was corrupted on the way. 19, 37

CIDR Classless Inter-Domain Routing 18, 24, 46, 47

CPE Common Platform Enumeration 37, 61

daemon A process that runs forever in the background to facilitate other programs. 3

dbus-daemon A daemon which enable a common interface for inter-process communication. 3

DHCP Dynamic Host Configuration Protocol 3, 4

DHCPCD Dynamic Host Configuration Protocol Client Daemon 3

DNS Domain Name System 22

driver A tiny software module which is loaded into the kernel when the computer boots up, They mainly interface with hardware and are often very specific for each piece of hardware. 3

FTP File Transfer Protocol 4, 19

header A header is the first few bytes at the start of a packet often consisting of information on where to send the packet next, can also contain information though. 6

HTML Hypertext Markup Language 6, 7

HTTP Hypertext transfer Protocol 4, 6, 16

HTTPS Hypertext transfer Protocol Secure 16

ICMP Internet Control Message Protocol 17, 18, 26, 27, 28, 31, 32, 33, 40, 43, 47, 52, 59, 60

IDS Intrusion Detection System 19

IP Internet Protocol 33, 47

IP address Every computer on a network has a unique IP address assigned to them, which is used to identify where exactly message sent by computers are meant to go. 3, 6, 16, 46, 47

kernel The kernel is the foundation of an operating system and it serves as the main interface between the software running on the system and the underlying hardware it performs task such as processor scheduling and managing input/output operations. 3

MAC Media Access Control 53

NIC Network Interface Card 3, 5, 53

OSI model Open Systems Interconnection model 4, 27

packet Packets are simply a list of bytes which contains packed values such as to and from address and they are the basis for almost all inter-computer communications. 3, 4, 5, 6, 8, 11, 12, 16, 17, 19, 37, 38

PCAP Packet CAPture 36

PHP PHP Hypertext Processor 5

port Computers have "ports" for each protocol which can be connected to separately, this makes up part of a "socket" connection. 6, 18, 19, 38, 46

port knocking Port knocking is where packets must be sent to a sequence of ports before access to the desired port is granted. 19

SCTP Stream Control Transmission Protocol 19

server A server is any computer which it's purpose is to provide resources to others, either humans or other computers for purposes from hosting website or just as a resource of large computational power. 3, 4, 24

service A service is something running on a machine that offers a service to either other programs on the computer or to people on the internet. 3, 12, 19, 24, 37, 38

SSH Secure SHell 61, 62

subnet A subnet is simply the sub-network of every possible IP address that will be used for communication on a particular network. 3, 4, 46

systemd A daemon for controlling what is run when the system starts. 3

TCP Transmission Control Protocol 6, 12, 15, 16, 17, 18, 19, 26, 33, 38, 42, 47, 53, 56, 57, 58, 61, 62

UDP User Datagram Protocol 6, 17, 19, 26, 43, 47, 59, 60, 61

upowerd Manages the power supplied to the system: charging, battery usage etc... 3

XML eXtensible Markup Language 21