

A Level Computer Science Non-Examined Assessment (NEA)

Sam Leonard

Contents

1	Analysis	3
1.1	Identification and Background to the Problem	3
1.2	Analysis of problem	11
1.2.1	Successful connection over TCP	11
1.2.2	An attempted connection to a closed port	14
1.2.3	An attempted connection with a firewall rule to drop packets	15
1.2.4	Project aims and methods	16
1.3	Success Criteria	17
1.4	Description of existing solutions	18
1.5	Prospective Users	24
1.6	Estimating Runtime Memory Requirements	24
1.7	Data Flow Diagram	26
1.8	Description of Solution Details	27
1.9	Acceptable Limitations	39
1.10	Test Strategy	39
2	Design	41
2.1	Overall System Design (High Level Overview)	41
2.2	Design of User Interface	41
2.3	System Algorithms	43
2.4	Input data validation	48
2.5	Algorithm for complex structures	48
3	Technical Solution	50
4	Testing	52
4.1	Test Plan	52
4.2	Testing Evidence	52
4.2.1	Printing a usage message when run without parameters .	52
4.2.2	Printing a help message when passed -h	53
4.2.3	Printing a help message when passed -help	54
4.2.4	Scanning a subnet with ICMP echo request messages . . .	54
4.2.5	Translating a CIDR-specified subnet into a list of IP ad- dresses	55
4.2.6	Scanning without first checking whether hosts are up. . .	56
4.2.7	Detecting whether a TCP port is open	59
4.2.8	Detecting whether a TCP port is closed	60
4.2.9	Detecting whether a TCP port is filtered	61
4.2.10	Detecting whether a UDP port is open	62
4.2.11	Detecting whether a UDP port is closed	63
4.2.12	Detecting whether a UDP port is filtered	64
4.2.13	Detecting the operating system of another machine	64
4.2.14	Detecting the service and its version running behind a port	65
4.2.15	User enters invalid ip address	66

4.2.16	User enters invalid number of network bits	66
4.2.17	User enters an invalid port number to scan	67
4.2.18	User enters an invalid number of network bits and a bad IP address	67
4.3	Test Table	68
5	Evaluation	69
5.1	Reflection on final outcome	69
5.2	Evaluation against objectives	69
5.3	Potential improvements	71
A	Technical Solution	72
A.1	icmp_ping	72
A.2	ping_scanner	73
A.3	subnet_to_addresses	76
A.4	tcp_scan	77
	A.4.1 connect_scan	77
	A.4.2 syn_scan	78
A.5	udp_scan	81
A.6	version_detection	86
A.7	modules	91
A.8	examples	127
A.9	netscan	128
A.10	tests	133

1 Analysis

1.1 Identification and Background to the Problem

The problem my project tries to solve is how to look at devices on a network from a “black box” perspective and gain information about what services are running. Services are programs whose 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 a number of steps a device has to go through from when it is turned on until it can connect to the internet. There are many more steps than those listed below, but the most important ones are.

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

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. These can range from graphics drivers for games to use graphics cards, to networking drivers which interact with the Network Interface Card (NIC).

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.¹ For example, when I start my laptop, the following daemons start: upowerd (power management²), systemd (manages the creation of all processes³), dbus-daemon (manages inter-process communication⁴), iwd (manages my WiFi connections⁵) and finally Dynamic Host Configuration Protocol Client Daemon (DHCPD), which manages all interactions with the network around DHCP.⁶

Once the daemons are all started, the DHCP client can 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 255.255.255.255,⁷ which is the IP limited broadcast address,⁸ 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 that 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 (see Figure 1).

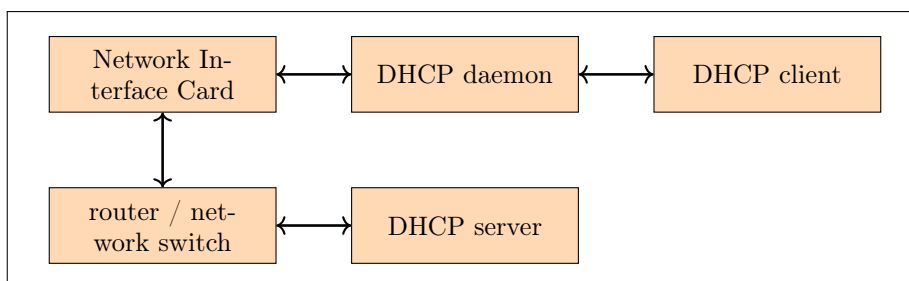


Figure 1: A block diagram showing the relationship between different elements of a DHCP negotiation.

Figure 2 shows a packet capture from my laptop where I turned WiFi off, started Wireshark listening and plugged in an Ethernet cable. Wireshark is a program which intercepts all the network communications on a single computer and records them to a file.⁹ It also displays them to the user, and is capable of performing an analysis and dissection of each 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. For a definition of packets see Page 6.

For the sake of clarity, the Figure displays Wireshark showing only the DHCP packets, so that the DHCP negotiation can be clearly seen, including the 255.255.255.255 limited broadcast destination address and the 0.0.0.0 unassigned address in the source column.⁸

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:¹⁰

7. Application: Applications Programming Interface (API)s, etc. . .
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.

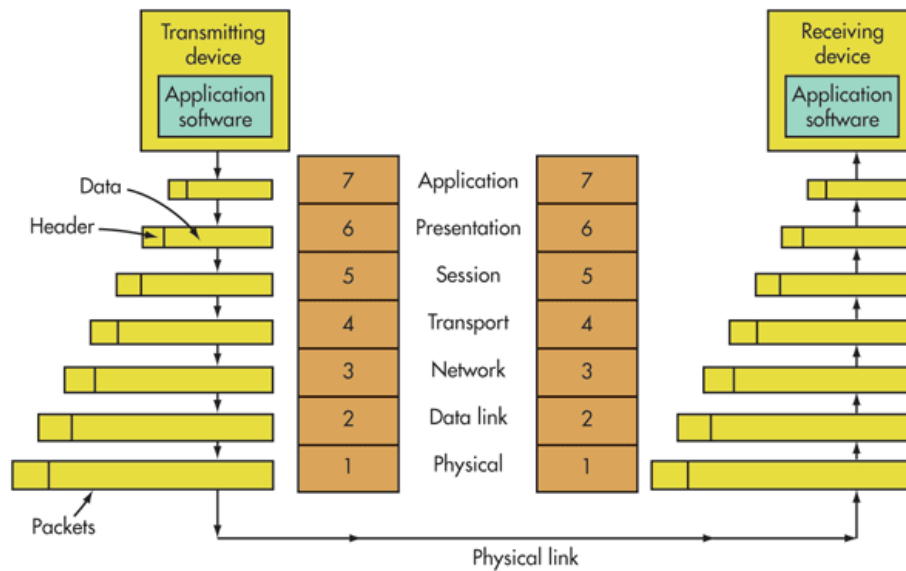


Figure 3: *OSI model diagram*, source: <https://www.electronicdesign.com>.

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; the innermost layer contains the data being transferred, and the other layers specify where the packet should go next. 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 anything. As an example, HyperText Transfer Protocol (HTTP) transfers websites using HyperText Markup Language (HTML) files and images. 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 were a hotel, the IP address would be the address 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.¹¹ There are this many ports because the port field in Transmission Control Protocol (TCP)¹² and User Datagram Protocol (UDP)¹³ is 16 bits long and the maximum value for a 16 bit unsigned integer is 65535. Both TCP¹⁴ and UDP¹⁵ 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. For this reason, UDP is used for purposes where speed is more important and missing some data is inconsequential, such as video streaming and playing online games.¹⁵

I would like to illustrate how ports and packets work using 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 interesting 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. The page is hosted using Python3’s `http.server` module, which is a quick and easy way to serve HTML pages and other content locally, without the hassle of setting up a full blown web server such as Nginx or Apache. Python’s `http.server` makes the current directory open on port 8000.¹⁶ From there, navigating to `/example.html` 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 in the TCP header is used to notify the client that

the server is ready to push the data (in this case example.html) to the client.¹⁴ The browser then sends back an acknowledgement and the server sends the page as shown in packets 7 and 8. Finally, the browser sends an 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.¹² Having received the FIN ACK packet, the server acknowledges this by sending an ACK packet back to the client, completing the graceful teardown. This process is repeated when the browser parses the HTML and identifies there is an image which it needs to get from the server as well. Because the image is a large file, it takes more packets to transfer the required data. In Figure 6 you can see a ladder diagram which shows the entire transaction symbolically. I have also colour coded Figure 6 with green arrow heads to the initial handshakes, blue for the HTTP protocol transactions and red for the TCP connection teardown packets.

The OSI model described in Figure 3 can be seen in action in Figures 4, 5 and 7. Figure 4 shows levels 6 and 7 of the OSI model. Levels 4 and 5 can be seen in figure 5 in the form of the TCP session negotiation and transferring the picture and example.html. Level 3/2/1 are shown in Figure 7 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.

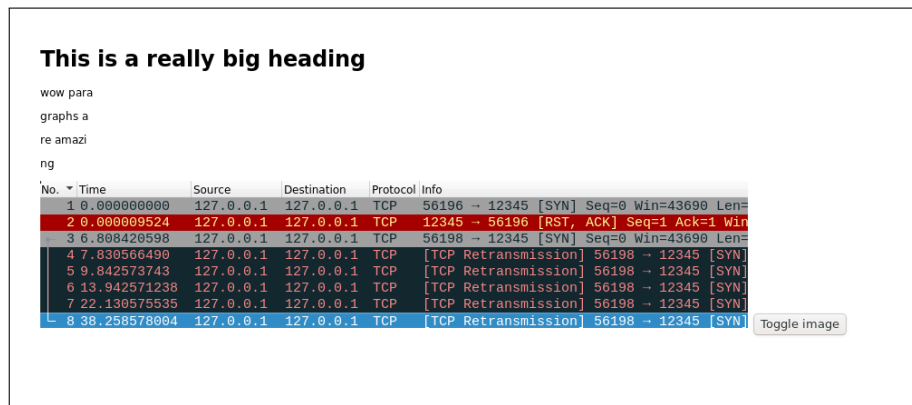


Figure 4: A basic static HTML webpage.

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help					
Apply a display filter ... <Ctrl-/> Expression... +					
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=0
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=0
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=
get_webpage_perfect.pcapng Packets: 38 · Displayed: 38 (100.0%) Profile: Default					

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

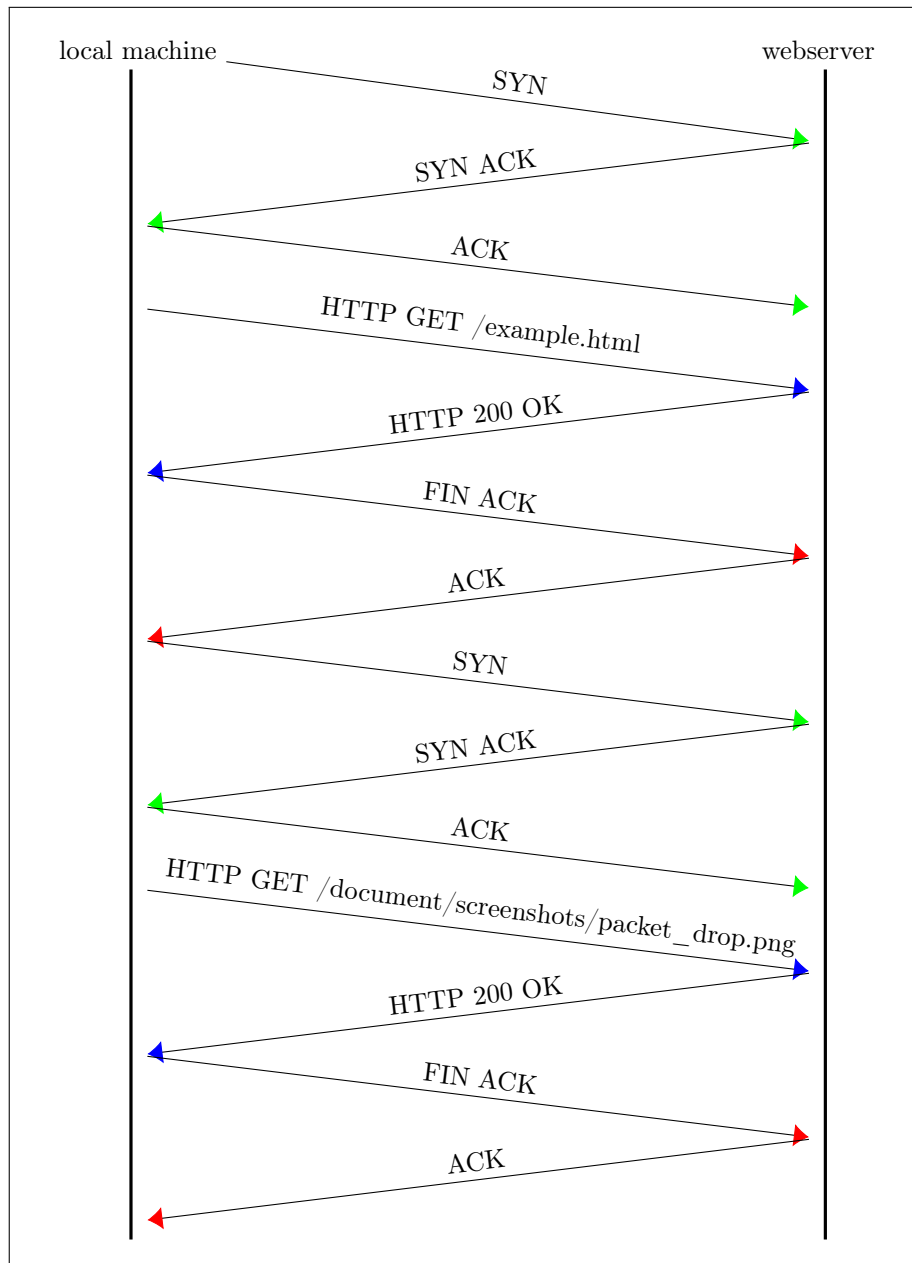


Figure 6: A ladder diagram showing the transaction in Figure 5.

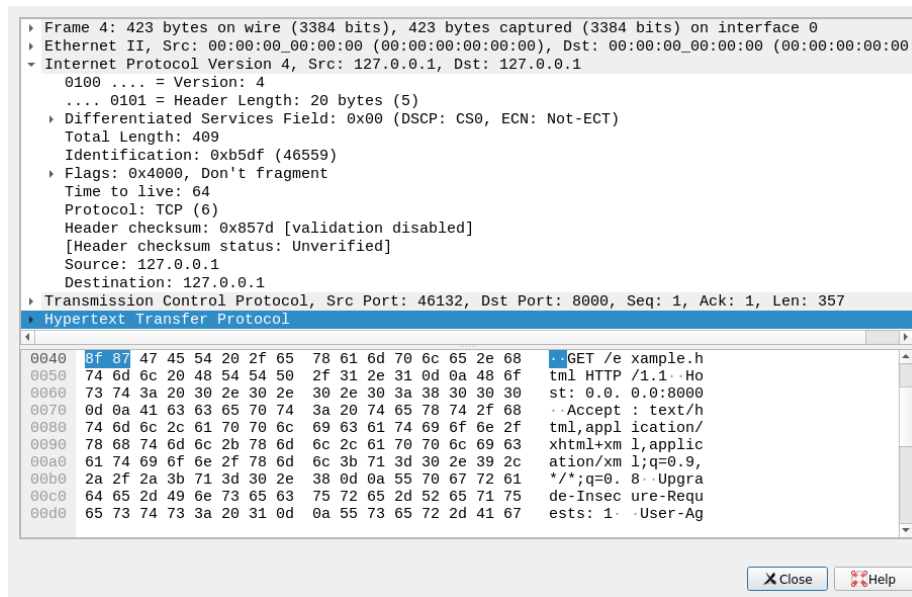


Figure 7: A look inside a TCP packet.

Listing 1: *example.html*

```

1  <!DOCTYPE html>
2  <html>
3  <head>
4  <title>Wow I can add titles</title>
5  </head>
6  <body>
7
8  <h1>This is a really big heading</h1>
9  <p>wow para</p>
10 <p>graphs a</p>
11 <p>re amazi</p>
12 <p>ng</p>
13 <script type="text/javascript">
14   function imgtog() {
15     if (document.getElementById("img").style.display == "none") {
16       document.getElementById("img").style = "block"
17     } else {
18       document.getElementById("img").style.display = "none"
19     }
20   }
21
22 </script>
23

```

```
24 
25
26 <button onclick="imgtog()">Toggle image</button>
27
28
29 </body>
30 </html>
```

1.2 Analysis of problem

To reiterate, the problem my project tries to solve is how to look at devices on a network from a “black box” perspective and gain information about what services are running. The difficulty with looking at a network from the outside is that the purpose of the network is to allow communication within the network, thus very little is exposed externally. This presents a challenge, as we want to know not only what machines are on the network, but also what services are running on each machine. This is not always possible, owing to the limited information that services reveal about themselves. Firewalls also play a large part in making network scanning difficult, as sometimes they simply drop packets instead of sending a TCP RST packet (reset connection packet). Dropping a packet means that when a packet is received, no response is sent back – as if the connection was just “dropped”. When firewalls drop packets, it becomes exponentially more difficult to determine the state of any port on the target machine, as you don’t know whether your packet was corrupted, or lost in transit, or if it 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.

1.2.1 Successful connection over TCP

For a TCP connection to be established there is a three-way handshake between the communicating machines. Initially, the machine trying to establish the connection sends a TCP SYN packet to the other machine. This packet holds a dual purpose:¹² 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 SYN, 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 8 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 11, where Wireshark allows us to take a look inside the packet to see the data being sent, along with the PSH and ACK flags being set.

The code I used to generate these packet captures is shown in Figures 9 and 10. Breaking the code down, Figure 10 shows me initialising a socket object,¹⁷ 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 to be looped back onto the current machine, hence its alternative name: the loopback address. The next line of code instructs the machine to listen for incoming connections. The program accepts the connection in Figure 9, 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 9 then sends some data which we then see printed to the screen in Figure 10, 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 8: *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 9: *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 10: *Receiving some basic text data over a TCP 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

▶	Frame 4: 366 bytes on wire (2928 bits), 366 bytes captured (2928 bits) on
▶	Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00
▶	Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▶	Transmission Control Protocol, Src Port: 47710, Dst Port: 12345, Seq: 1,
▶	Data (300 bytes)

0000	00 00 00 00 00 00 00 00	00 00 00 00 08 00 45 00E.
0010	01 60 70 81 40 00 40 06	cb 14 7f 00 00 01 7f 00	.`p@.@.....
0020	00 01 ba 5e 30 39 09 d1	70 b2 e9 c6 d7 ad 80 18	...^09..p.....
0030	01 56 ff 54 00 00 01 01	08 0a 1a 7c 9a 84 1a 7b	.V.T....{
0040	ca 01 68 69 20 49 27 6d	20 64 61 74 61 20 77 68	..hi I'm data wh
0050	61 74 27 73 20 79 6f 75	72 20 6e 61 6d 65 3f 20	at's you r name?
0060	68 69 20 49 27 6d 20 64	61 74 61 20 77 68 61 74	hi I'm d ata what
0070	27 73 20 79 6f 75 72 20	6e 61 6d 65 3f 20 68 69	's your name? hi
0080	20 49 27 6d 20 64 61 74	61 20 77 68 61 74 27 73	I'm dat a what's
0090	20 79 6f 75 72 20 6e 61	6d 65 3f 20 68 69 20 49	your na me? hi I
00a0	27 6d 20 64 61 74 61 20	77 68 61 74 27 73 20 79	'm data what's y
00b0	6f 75 72 20 6e 61 6d 65	3f 20 68 69 20 49 27 6d	our name ? hi I'm
00c0	20 64 61 74 61 20 77 68	61 74 27 73 20 79 6f 75	data wh at's you
00d0	72 20 6e 61 6d 65 3f 20	68 69 20 49 27 6d 20 64	r name? hi I'm d
00e0	61 74 61 20 77 68 61 74	27 73 20 79 6f 75 72 20	ata what 's your
00f0	6e 61 6d 65 3f 20 68 69	20 49 27 6d 20 64 61 74	name? hi I'm dat
0100	61 20 77 68 61 74 27 73	20 79 6f 75 72 20 6e 61	a what's your na
0110	6d 65 3f 20 68 69 20 49	27 6d 20 64 61 74 61 20	me? hi I 'm data
0120	77 68 61 74 27 73 20 79	6f 75 72 20 6e 61 6d 65	what's y our name
0130	3f 20 68 69 20 49 27 6d	20 64 61 74 61 20 77 68	? hi I'm data wh
0140	61 74 27 73 20 79 6f 75	72 20 6e 61 6d 65 3f 20	at's you r name?
0150	68 69 20 49 27 6d 20 64	61 74 61 20 77 68 61 74	hi I'm d ata what
0160	27 73 20 79 6f 75 72 20	6e 61 6d 65 3f 20	's your name?

Figure 11: *Highlighted packet carrying the data being transferred in Figure 9.*

1.2.2 An attempted connection to a closed port

In Figure 12 shows my program beginning by sending the same TCP SYN packet as we saw in the attempted connection to an open port discussed above. The difference comes in the next packet with the TCP RST flag being sent back. This flag resets the connection,¹² 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 12. The code used to generate this is shown in Figure 13; 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 12.

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 12: *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 13: *The code used to produce firewall packet dropping example in Figure 12.*

1.2.3 An attempted connection with a firewall rule to drop packets

I would like to commence this section by explaining a bit about firewalls and how they work. Firewalls are essentially the gatekeepers of the internet; they decide whether or not a packet is passed on. Firewalls work by a set of rules which decide what to do with a particular packet. Such a rule might be that it is coming from a certain IP address or has a particular destination port. The actions taken after the packet has had its 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 and lets the packet through; DROP simply 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). For demonstration purposes I will be using a Linux utility called iptables for implementing all firewall rules on my system. Packet number three in Figure 12 corresponds to line 4 of the code in Figure 13, the difference being that in Figure 13 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 directs that all packets arriving (-I INPUT) with source address 127.0.0.1 (-s 127.0.0.1) are dropped, with no response sent (-j DROP).¹⁸ With this firewall rule in place you can see in Figure 12, packet 3 receives no response, and as such Python assumes that the packet just got lost and tries to send the packet again repeatedly. This continued for more than 30 seconds before I stopped it, as shown by the time column in Figure 12, and the final `KeyboardInterrupt` in Figure 13. The amount of time that a system will continue trying to reconnect depends on the OS and other factors, but the minimum time is 100 seconds, as specified by RFC 1122.¹⁹ On most systems, the timeout is between 13 and 30 minutes, according to the Linux manual page on TCP reproduced below.²⁰

```
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.
```

1.2.4 Project aims and methods

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,²¹ which will be able to detect the state of ports on remote machines (as in whether the port is open/closed or filtered etc.); detect which hosts are up on a subnet; and 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 such as f-strings which are not in even fairly recent versions such as 3.5. F-strings allow for a clear and consistent string formatting syntax, which I will use extensively. I have chosen Python in particular, because it is very readable and has extensive low level bindings to kernel syscalls with the socket module allowing me to write code quickly that is easily understandable and has a clear purpose. Python enables me to use low level networking functions, and even change 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). 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, and 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 presents 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 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
- File Transfer Protocol (FTP): bounce scan

As well as supporting a vast array of scanning types, Nmap can also perform service and version detection, and operating system detection via custom probes. It 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 final port is allowed). It 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, and indeed there is an entire book on using nmap (<https://nmap.org/book/>). Figure 14 shows the logical structure behind nmap scanning a network.

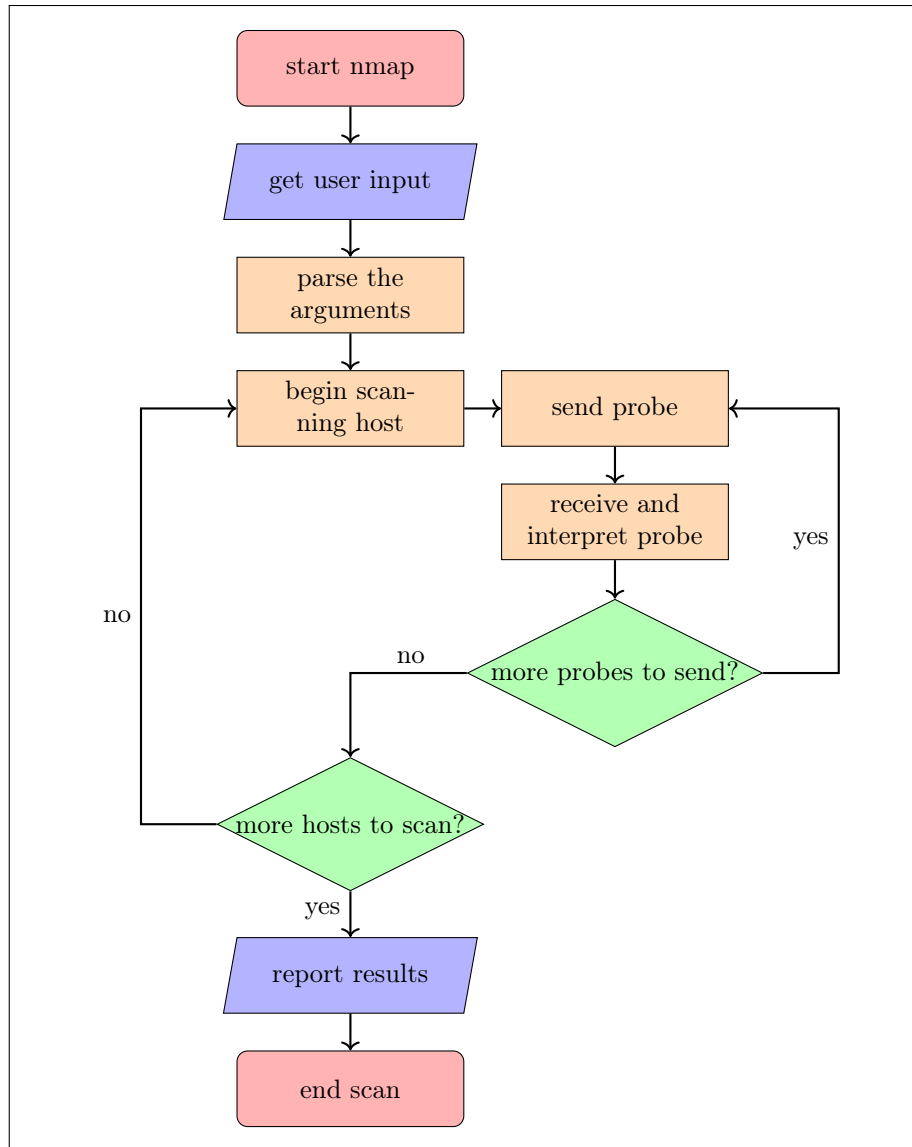


Figure 14: A flow chart showing how nmap does scanning.

The following paragraphs discuss an example nmap scan I did on my home network I did for this project. The command I used was:
`nmap -sC -sV -oA networkscan 192.168.1.0/24`. The purpose of the command line flags was 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**. The program outputs to three files: **networkscan.(nmap,gnmap,xml)**. Before I go into what each file contains, I will explain some terminology: something is greppable if it can be easily searched with the Linux utility **grep**.²² Grep stands for Globally search a Regular Expression and Print, which instructs the computer to search the specified file for lines that contain a certain word or pattern; for example, the command to find all lines with the word “hi” in them in the file “document” would be **grep ‘hi’ document**.

Returning to the files nmap created: **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$F: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(DNSStatusReq
uestTCP,E,"\0\x0c\0\0\x90\x04\0\0SF:\0\0\0\0\0\0");
Service Info: CPE: cpe:/o:asus:wrt_firmware
```

The above is the report for only one device on the network. In it, you can see information such as which ports are open, and what services are running

behind them. As this example device is network router, you can see port 8443, which nmap has recognised to be hosting the ASUS web admin, from which you can configure the router. There follows some other associated information extracted from the server. Most of this extra information is derived from the `-sC` flag, which enables 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. 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 explains why nmap is so good at recognising services: it has a lot of data to look at and learn from in regards to service fingerprinting.

`Networkscan.gnmap` contains exactly the same information as `networkscan.nmap`, but the output is formatted to enable easy searching with the `grep` utility. I reproduce part of the output file below:

```
# 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:
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)
```

As you can see above, all of the information is on a single line for each type of scan. This is useful: for example, if you want to scan a large number of hosts and just want to know which hosts are up you could use `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
```

This shows the hosts which are online and their host names. Other ways to use the greppable output format would be to search for 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, the `.gnmap` output is useful for when you want to use `grep` to filter results.

Finally, we have the eXtensible Markup Language (XML) format file, `networkscan.xml` part of which is reproduced below:

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE nmaprun>
3 <?xml-stylesheet href="file:///usr/bin/./share/nmap/nmap.xsl"
  type="text/xsl"?>
4 <!-- 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 -->
5 <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">
6 <verbose level="0"/>
7 <debugging level="0"/>
8 <host starttime="1554921379" endtime="1554923187"><status state="up"
  reason="syn-ack" reason_ttl="0"/>
9 <address addr="192.168.1.1" addrtype="ipv4"/>
10 <hostnames>
11 <hostname name="router.asus.com" type="PTR"/>
12 </hostnames>
13 <ports><extraports state="closed" count="995">
14 <extrareasons reason="conn-refused" count="995"/>
15 </extraports>
16 <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
17 -pc-Linux-gnu%r(DNSVersionBindReqTCP,20,&quot;\0\x1e\0\x06\x85\x85\0
18 \x01\0\0\0\0\0\0\x07version\x04bind\0\0\x10\0\x03&quot;)%r
19 (DNSStatusRequestTCP,E,&quot;\0\x0c\0\0\x90\x04\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>
20 </script></port>

```

It can be seen that this file is extremely verbose. It contains the reason why each port has the state it does, as well as a vast amount of other data that the other scans did not include. The consequence is that this output format is not very easy for humans to read, meaning that this format is available because it is easier for other programs to parse than the other formats. As well as this the extra information 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, it is an application at level 7 when the user interacts, with it but uses several libraries that interact at level 2, which it uses to get the raw headers of the packets being sent and thus gain

information from them (see Figure 15). Nmap has virtually no competitors in the Linux ecosystem, other than possibly Angry IP Scanner, which is another open source network scanner, except it has a much smaller user base.

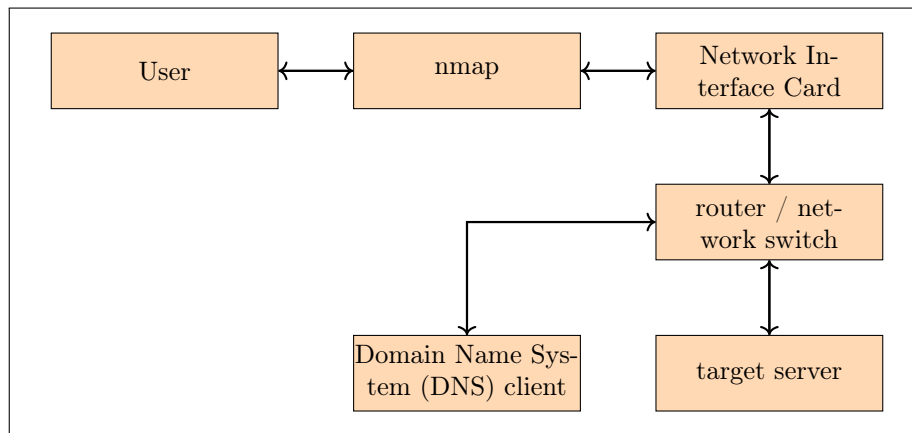


Figure 15: A block diagram showing how nmap sits in the software stack.

Before describing my program in detail I would like to explain some terminology I will use: “parse the arguments” means taking the string of text that the user enters after the program name i.e. `program <text>`. It is these texts that represent the arguments. Parsing the arguments means turning those strings into useful information that the program can use. For example, my program will allow people to enter the port number(s) they want to scan. I want them to be able to do this by specifying a range of ports. If the user specifies 10–20, this would mean ports 10, 11, . . . , 20. Thus an example of parsing would be the turning of 10–20 into the list of numbers from 10 to 20 as shown in Algorithm 1 below. “Probes” refer to the actual packets being sent to the server; I will refer to anything sent from my code to another machine as being a “probe”. I will use the term “hosts” to mean the other machines on the network that we are scanning.

Algorithm 1 *This is an example algorithm for parsing the port range argument I gave as an example above, extended by allowing for comma separated lists of ports intermixed with ranges.*

```

1: procedure PORT_PARSER
2:    $argument \leftarrow$  string after program name
3:    $chunks \leftarrow$  argument split on ‘,’
4:    $ports \leftarrow$  empty list
5:   for  $chunk$  in  $chunks$  do
6:     if  $chunk$  contains “-” then ▷ a range chunk
7:        $numbers \leftarrow chunk$  split on “-”
8:       for  $port \leftarrow numbers[0], numbers[1]$  do
9:         Append  $port$  to  $ports$ 
10:    else ▷ a single number chunk
11:      Append  $chunk$  to  $ports$ 
  return  $ports$ 

```

1.5 Prospective Users

The prospective users of my program would be system administrators, penetration testers or network engineers. In my particular case, prospective users would be my school’s system administrators. It would allow them to see an outsider’s 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. 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 that version of the program. 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.

I plan to use my school’s system administrators users in order to gain some feedback as to the usability and performance of my program.

1.6 Estimating Runtime Memory Requirements

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)
- The probes to be used for version detection

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 1000 ports of each machine. For the purpose of RAM estimation I will not consider version detection, as I am unsure of how I will implement it currently. To measure the size of an 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.*

```

1 >>> with open("hosts", "r") as f
2 ...     hosts = f.read().splitlines()
3 ...
4 >>> import sys
5 >>> sys.getsizeof(hosts)
6 2216
7 >>> ports = list(range(1000))
8 >>> sys.getsizeof(ports)
9 9112
10 >>> len(hosts)*sys.getsizeof(ports) / 2**10 # 2*10 is one kibibyte
11 2278.0
12 >>> sys.getsizeof(True)
13 28
14 >>> len(hosts)*(sys.getsizeof(True)) / 2**10
15 7.0
16 >>> pings[0]
17 '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'
18 >>> from binascii import unhexlify
19 >>> ping = unhexlify(pings[0].replace(" ", "")) # turn the string of
    numbers into a bytes object
20 >>> sys.getsizeof(ping)
21 117
22 >>> len(hosts)*sys.getsizeof(ping) / 2**10
23 29.25
24 >>> 2278.0 + 7.0 + 29.25 + 2.22
25 2316.47

```

As shown above 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 over than ninety eight percent of the total space used by the mock data I created. However, a total of 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 out from my application
2. Receiving packets back from the targets
3. Transferring data between functions

My program will only hold information in memory and provides no utility for saving the information from scans. This is because on the target systems (Linux/Unix based machines), the shell which is used to run commands has a very simple way of placing output in files by use of Unix “pipes”, which are how Unix-based operating systems handle interprocess communications. If nmap did not have a dedicated saving utility, the following command could be used to save the output of nmap to a text file called outputfile:

`nmap 192.168.1.0 > outputfile`. Thus, in my application, for the sake of simplicity, this is the method I will use to save output.

Figure 16 below shows the proposed data flow in my program.

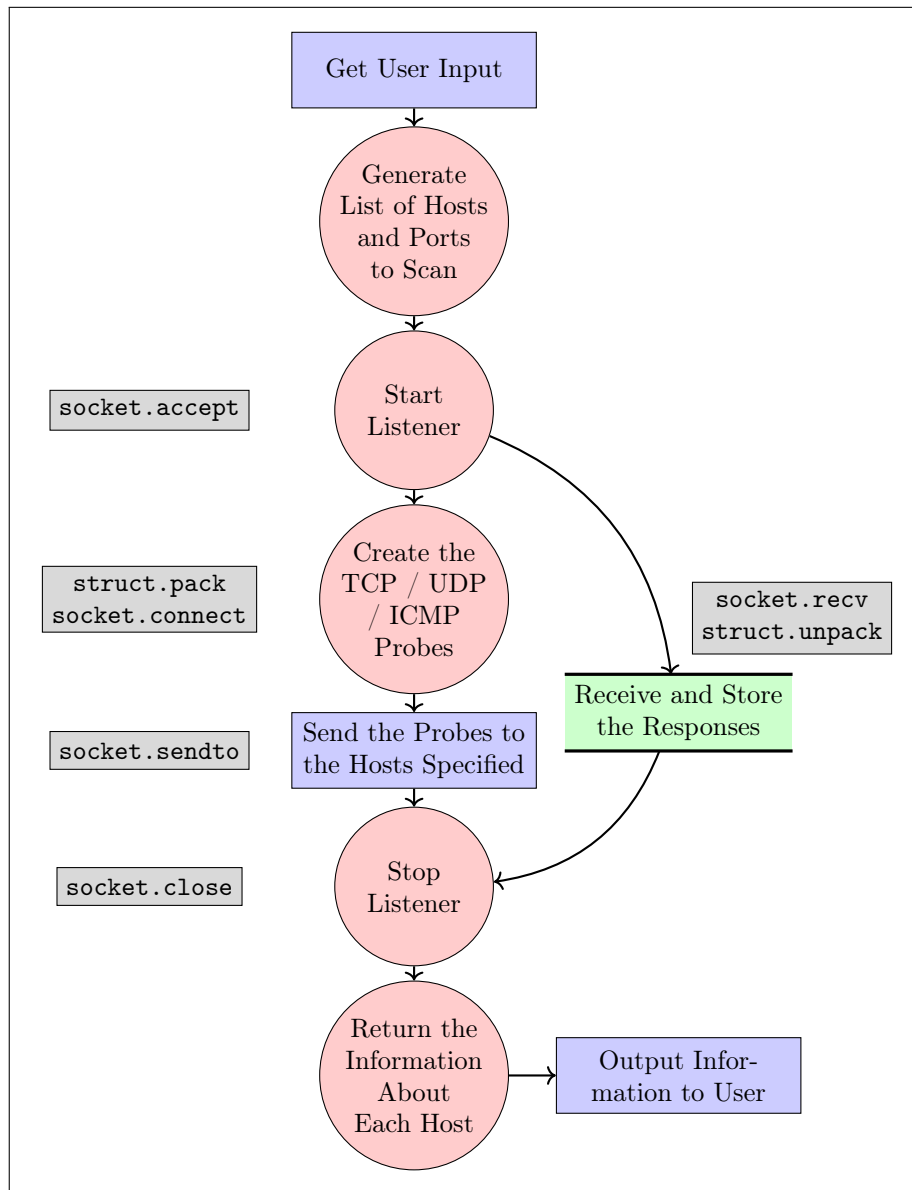


Figure 16: A data flow digram for information in my application.

1.8 Description of Solution Details

As already stated above on page 16, I will be using Python version 3.7.2 for my project because I am already familiar with Python's syntax and its socket

library has a very nice high level API for making system calls to the kernel's low level networking functions. This makes it ideal for a networking project like mine, as it allows me to prototype easily, and explore many ideas about how I could implement my solution in a time-efficient manner.

I decided to start my project by researching how to write code for receiving and sending ICMP **echo requests** (i.e. pings). ICMP sits at layer 3 of the OSI model. This means that it functions at a layer below that to which you are normally given access in the socket module. Sending an ICMP **echo request** requires a raw socket. A raw socket is one which will return everything contained by the ethernet/wifi frame, including the raw Internet Protocol (IP) headers. The bytes object received from a raw socket thus requires unpacking to extract relevant information. The struct module provides a convenient API for converting between packed values, because there is usually a difference in endianness between the network and the local machine which requires translation.

Interactions with the socket module are mainly through the pack and unpack functions. For each of these functions it is necessary to 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 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 an IP checksum function which I wrote (see Listing 5 below). In Figure 17, you can see the output when I run the command `ping 127.0.0.1` which the code in Listing 4 is listening for packets.

Algorithm 2 *The psuedocode representation of Listing 3.*

```

1: socket  $\leftarrow$  new ICMP socket
2: ID  $\leftarrow$  process ID & 0xFFFF
3: dummy header  $\leftarrow$  PACK("bbHHh", 8, 0, 0, ID, 1)
4: time  $\leftarrow$  PACK(TIME(now))
5: data  $\leftarrow$  time + "A"  $\times$  (192 - LENGTH(time))
6: checksum  $\leftarrow$  IPCHECKSUM(dummy header + data)
7: header  $\leftarrow$  PACK("bbHHh", 8, 0, checksum, ID, 1)
8: packet  $\leftarrow$  header + data
9: SOCKET.SEND(packet)

```

Listing 3: *A prototype for sending ICMP echo request packets.*

```

1 #!/usr/bin/Python3.7
2 import socket
3 import struct
4 import os
5 import time

```

```

6  import array
7
8  from os import getcwd, getpid
9  import sys
10 sys.path.append("../modules/")
11
12 import ip_utils
13
14
15 ICMP_ECHO_REQUEST = 8
16
17 # opens a raw socket for the ICMP protocol
18 ping_sock = socket.socket(socket.AF_INET, socket.SOCK_RAW,
19                             socket.IPPROTO_ICMP)
20 # allows manual IP header creation
21 # ping_sock.setsockopt(socket.SOL_IP, socket.IP_HDRINCL, 1)
22
23 ID = os.getpid() & 0xFFFF
24
25 # the two zeros are the code and the dummy checksum, the one is the
26 # sequence number
27 dummy_header = struct.pack("bbHHh", ICMP_ECHO_REQUEST, 0, 0, ID, 1)
28
29 data = struct.pack("d", time.time()) + bytes((192 -
30 struct.calcsize("d")) * "A", "ascii")
31
32 checksum = ip_utils.ip_checksum(dummy_header+data)
33
34 header = struct.pack("bbHHh", ICMP_ECHO_REQUEST, 0, checksum, ID, 1)
35
36 packet = header + data
37
38 ping_sock.sendto(packet, ("127.0.0.1", 1))

```

Algorithm 3 *Pseudocode for the code in Listing 4.*

- 1: *socket* \leftarrow new ICMP socket
 - 2: *packet* \leftarrow SOCKET.RECEIVE("one packet")
 - 3: *data* \leftarrow UNPACK(*packet*)
 - 4: PRINT(*data*)
-

Listing 4: *A prototype for receiving ICMP echo request packets.*

```

1  #!/usr/bin/Python3.7
2
3  import socket
4  import struct
5  import time

```

```

6 from typing import List
7
8 # socket object using an IPV4 address, using only raw socket access, set
  ICMP protocol
9 ping_sock = socket.socket(socket.AF_INET, socket.SOCK_RAW,
  socket.IPPROTO_ICMP)
10
11 packets: List[bytes] = []
12
13 while len(packets) < 1:
14     recPacket, addr = ping_sock.recvfrom(1024)
15     ip_header = recPacket[:20]
16     icmp_header = recPacket[20:28]
17
18     ip_hp_ip_v, ip_dscp_ip_ecn, ip_len, ip_id, ip_flg_ip_off, ip_ttl,
        ip_p, ip_sum, ip_src, ip_dst = struct.unpack('!BBHHBBI',
        ip_header)
19
20     hl_v = f"{ip_hp_ip_v:08b}"
21     ip_v = int(hl_v[:4], 2)
22     ip_hl = int(hl_v[4:], 2)
23     dscp_ecn = f"{ip_dscp_ip_ecn:08b}"
24     ip_dscp = int(dscp_ecn[:6], 2)
25     ip_ecn = int(dscp_ecn[6:], 2)
26     flgs_off = f"{ip_flg_ip_off:016b}"
27     ip_flg = int(flgs_off[:3], 2)
28     ip_off = int(flgs_off[3:], 2)
29     src_addr = socket.inet_ntoa(struct.pack('!I', ip_src))
30     dst_addr = socket.inet_ntoa(struct.pack('!I', ip_dst))
31
32     print("IP header:")
33     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_flg: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")
34
35     msg_type, code, checksum, p_id, sequence = struct.unpack('!bbHHh',
        icmp_header)
36     print("ICMP header:")
37     print(f"Type: [{msg_type}]\nCode: [{code}]\nChecksum:
        [{checksum:04x}]\nProcess ID: [{p_id:04x}]\nSequence:
        [{sequence}]\n")
38     packets.append(recPacket)
39 open("current_packet", "w").write("\n".join(" ".join(map(lambda x:
    "{x:02x}", map(int, i))) for i in packets))

```

```

1: function IP_CHECKSUM(data)
2:   if LENGTH(data) is odd then
3:     data.append(0)
4:   total  $\leftarrow$  0
5:   for i in 0,2,LENGTH(data) do
6:     total  $\leftarrow$  total + data[i] << 8
7:     total  $\leftarrow$  total + data[i+1]
8:   carried  $\leftarrow$  (total - (total & 0xFFFF)) >> 16
9:   total  $\leftarrow$  total & 0xFFFF
10:  total  $\leftarrow$  total + carried
11:  if total > 0xFFFF then
12:    total  $\leftarrow$  total & 0xFFFF
13:    total  $\leftarrow$  total + 1
14:  total  $\leftarrow$  INVERT(total) return total

```

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

```

1 def ip_checksum(packet: bytes) -> int:
2     """
3     ip_checksum function takes in a packet
4     and returns the checksum.
5     """
6     if len(packet) % 2 == 1:
7         # if the length of the packet is odd, add a NULL byte
8         # to the end as padding to make it even in length
9         packet += b"\0"
10
11     total = 0
12     for first, second in (
13         packet[i:i+2]
14         for i in range(0, len(packet), 2)
15     ):
16         total += (first << 8) + second
17
18     # calculate the number of times a
19     # carry bit was added and add it back on
20     carried = (total - (total & 0xFFFF)) >> 16
21     total &= 0xFFFF
22     total += carried
23
24     if total > 0xFFFF:
25         # adding the carries generated a carry
26         total &= 0xFFFF
27         total += 1
28
29     # invert the checksum and take the last 16 bits

```


30 `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 17: *Dissecting an ICMP echo request packet.*

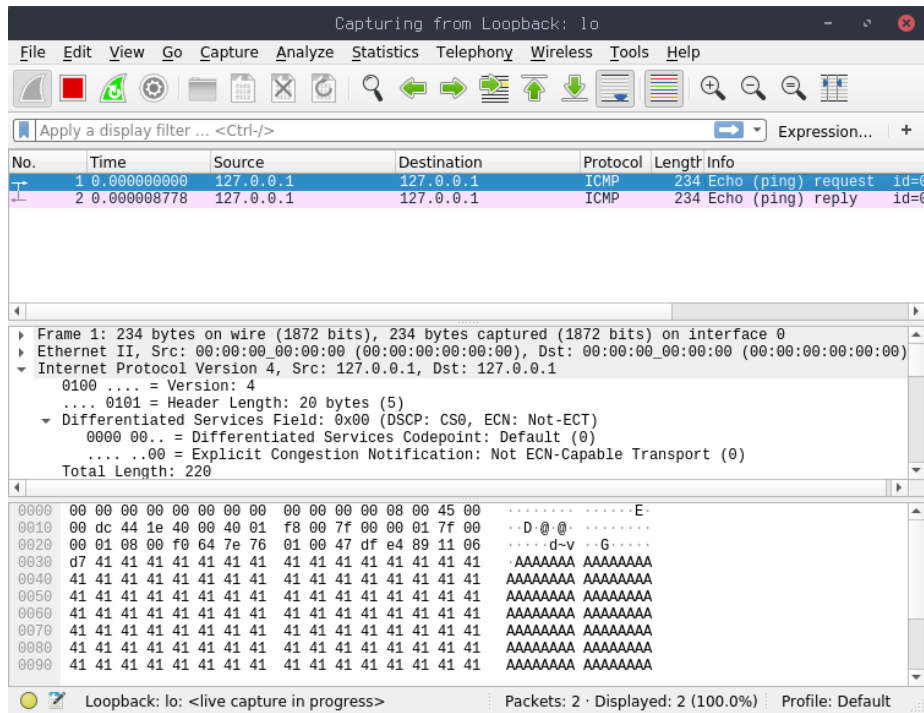


Figure 18: Screenshot of Wireshark showing a successful send of an ICMP echo request packet.

```

root@th0nkpad40:~/networkScanner/Code
1 #!/usr/bin/python
2
3 import socket
4 import struct
5 # socket object using an IPV4 address, using only raw socket access, set ICMP
6 ping_sock = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_ICMP
7 protocol
8 # this line sets the IP_HDRINCL attribute in SOL_IP to 1 allowing us to manual
9 ly create IP headers,
10 ping_sock.setsockopt(socket.SOL_IP, socket.IP_HDRINCL, 1)
11
12 while 1:
13     recPacket, addr = ping_sock.recvfrom(1024)
14     icmp_header = recPacket[20:28]
15     msg_type, code, checksum, p_id, sequence = struct.unpack('bbHh', icmp_header)
16     print("type: [" + str(msg_type) + "] code: [" + str(code) + "] checksum: ["
17 + str(checksum) + "] p_id: [" + str(p_id) + "] sequence: [" + str(sequence)
18 + "]\n")
19     print(" ".join(":%02h" % i for i in recPacket))
20
21 icmp-echo-rec.py 16, 34-37 All
22 "icmp-echo-rec.py" 16L, 775C written
23
24 root@th0nkpad40:~/networkScanner/Code
25
26 Ping 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
27 64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.076 ms
28 64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.093 ms
29 64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.098 ms
30
31 --- 127.0.0.1 ping statistics ---
32 3 packets transmitted, 3 received, 0% packet loss, time 25ms
33 rtt min/avg/max/mdev = 0.076/0.089/0.098/0.009 ms
34 [tr0tk@th0nkpad40 Code]$ ping 127.0.0.1
35 Ping 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
36 64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.075 ms
37 64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.093 ms
38 64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.098 ms
39
40 --- 127.0.0.1 ping statistics ---
41 3 packets transmitted, 2 received, 0% packet loss, time 17ms
42 rtt min/avg/max/mdev = 0.075/0.084/0.093/0.009 ms
43 [tr0tk@th0nkpad40 Code]$ ping 127.0.0.1
44 Ping 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
45 64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.084 ms
46 64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.104 ms
47 64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.098 ms
48
49 --- 127.0.0.1 ping statistics ---
50 3 packets transmitted, 3 received, 0% packet loss, time 43ms
51 rtt min/avg/max/mdev = 0.084/0.095/0.104/0.011 ms
52 [tr0tk@th0nkpad40 Code]$
53
54 no IPv6 | 27.6 GiB | v: down | E: down | BAT 38.57% 02:00:01 | 0.14 | 2018-12-10 07:58:05

```

Figure 19: Screenshot showing me first successfully dissecting an ICMP echo request packet.

Having written a prototype program which was capable of sending an ICMP echo request and another prototype which was capable of receiving and unpacking it. Having done these prototypes I identified that it would be best to abstract the code for dissecting all the headers i.e. ICMP, TCP and IP into classes where I can just pass the received packet into the class and have the class dissect it for me. This will also give me access to some of the benefits of classes, such as the `__repr__` method, which is called when you print classes out and allows control over what is printed out. Before embarking on the final program, I decided to write a prototype ping scanner, as this would allow me to get a feel for making a scanner, and to further exploring low level protocol interactions.

Listing 6: An attempt at making a ping scanner.

```

1 #!/usr/bin/Python3.7
2 from os import getcwd, getpid
3 import sys
4 sys.path.append("../modules/")
5

```

```

6 import ip_utils
7
8 import socket
9 from functools import partial
10 from itertools import repeat
11 from multiprocessing import Pool
12 from contextlib import closing
13 from math import log10, floor
14 from typing import List, Tuple
15 import struct
16 import time
17
18
19 def round_significant_figures(x: float, n: int) -> float:
20     """
21     rounds x to n significant figures.
22     round_significant_figures(1234, 2) = 1200.0
23     """
24     return round(x, n-(1+int(floor(log10(abs(x))))))
25
26
27 def recieved_ping_from_addresses(ID: int, timeout: float) ->
28     List[Tuple[str, float, int]]:
29     """
30     Takes in a process id and a timeout and returns the list of
31     addresses which sent
32     ICMP ECHO REPLY packets with the packed id matching ID in the time
33     given by timeout.
34     """
35     ping_sock = socket.socket(socket.AF_INET, socket.SOCK_RAW,
36                               socket.IPPROTO_ICMP)
37     time_remaining = timeout
38     addresses = []
39     while True:
40         time_waiting = ip_utils.wait_for_socket(ping_sock,
41                                                  time_remaining)
42         if time_waiting == -1:
43             break
44         time_recieved = time.time()
45         recPacket, addr = ping_sock.recvfrom(1024)
46         ip_header = recPacket[:20]
47         ip_hp_ip_v, ip_dscp_ip_ecn, ip_len, ip_id, ip_flg_ip_off,
48             ip_ttl, ip_p, ip_sum, ip_src, ip_dst =
49             struct.unpack('!BBHHHBBHII', ip_header)
50         icmp_header = recPacket[20:28]
51         msg_type, code, checksum, p_id, sequence =
52             struct.unpack('bbHHh', icmp_header)
53         time_remaining -= time_waiting
54         time_sent = struct.unpack("d",
55                                   recPacket[28:28+struct.calcsize("d")])[0]

```

```

47         time_taken = time_recieved - time_sent
48         if p_id == ID:
49             addresses.append((str(addr[0]), float(time_taken),
50                             int(ip_ttl)))
51         elif time_remaining <= 0:
52             break
53         else:
54             continue
55     return addresses
56
57 with closing(socket.socket(socket.AF_INET, socket.SOCK_RAW,
58                             socket.IPPROTO_ICMP)) as ping_sock:
59     addresses = ip_utils.ip_range("192.168.1.0/24")
60     local_ip = ip_utils.get_local_ip()
61     if addresses is not None:
62         addresses_to_scan = filter(lambda x: x!=local_ip, addresses)
63     else:
64         print("error with ip range specification")
65         exit()
66     p = Pool(1)
67     ID = getpid() & 0xFFFF
68     replied = p.apply_async(recieved_ping_from_addresses, (ID, 2))
69     for address in zip(addresses_to_scan, repeat(1)):
70         try:
71             packet = ip_utils.make_icmp_packet(ID)
72             ping_sock.sendto(packet, address)
73         except PermissionError:
74             pass
75     p.close()
76     p.join()
77     hosts_up = replied.get()
78     print("\n".join(map(lambda x: f"host: [{x[0]}}\tresponded to an ICMP
79                        ECHO REQUEST in {round_significant_figures(x[1], 2):<10}
80                        seconds, ttl: [{x[2]}}", hosts_up)))

```

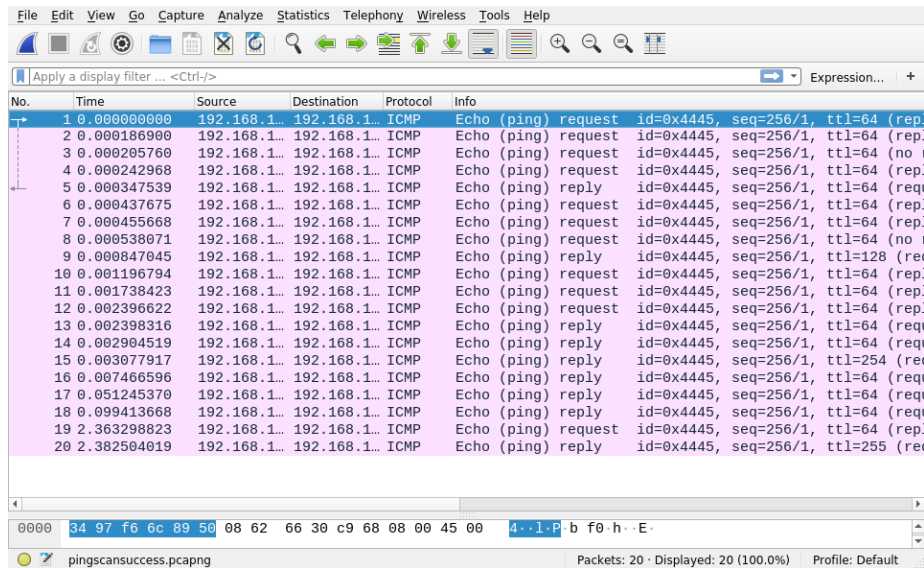


Figure 20: Screenshot of Wireshark showing a successful ping scan.

Listing 7: The output of from the ping scanner on the run which generated the PCAP file in figure 20

```

1 $ sudo ./ping_scan.py
2 host: [192.168.1.1] responded to an ICMP ECHO REQUEST in 0.00037
   seconds, ttl: [64]
3 host: [192.168.1.35] responded to an ICMP ECHO REQUEST in 0.00042
   seconds, ttl: [128]
4 host: [192.168.1.37] responded to an ICMP ECHO REQUEST in 0.002 seconds,
   ttl: [64]
5 host: [192.168.1.117] responded to an ICMP ECHO REQUEST in 0.0017
   seconds, ttl: [64]
6 host: [192.168.1.176] responded to an ICMP ECHO REQUEST in 0.0014
   seconds, ttl: [254]
7 host: [192.168.1.14] responded to an ICMP ECHO REQUEST in 0.0072
   seconds, ttl: [64]
8 host: [192.168.1.246] responded to an ICMP ECHO REQUEST in 0.049
   seconds, ttl: [64]
9 host: [192.168.1.8] responded to an ICMP ECHO REQUEST in 0.099 seconds,
   ttl: [64]

```

Completion of these prototypes has given me an understanding of 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 starts 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

My original concept included dedicated operating system detection as an option. However, if I find that time is short having implemented version detection, it will be an acceptable limitation to omit operating system detection. This is because version detection can provide some indication of the operating system in use, by virtue of the fact that some services are restricted to certain operating systems. For example, if the scanner detected that the service ActiveSync was in use, this would indicate that the system being scanned was a Windows system, which is reflected in the match directive and attached CPE information for ActiveSync:

```
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:ac
tivesync/ 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 will employ two separate testing strategies because they are good at different things, both of which I need in order to show that my project works. First, I will use unit testing to test some general purpose functions, which are pure functions (are independent of the current state of the machine).

I will use Wireshark to test those parts of the program which involve the use of impure functions and low level networking. 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 is on the network. Wireshark will allow me to see my scanners sending packets, and to check whether the parsers I have written for the various protocol are working. I can also check that the checksums in each of the various protocols are valid as Wireshark is capable of performing checksum verification for a wide variety of 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 have installed the following versions of

each of the programs I will be using to test my code: Wireshark 3.1.0, Python 3.7.2 and PyTest 4.3.1. I am also using pyenv version 1.2.9 to manage the version of Python in my Python environment. I plan not to use any 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 `socket.connect` it and sees whether it can connect or not. 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 are set in the received packet. Flags are parts of TCP packets that constitute a one byte long section, which store “flags” where each bit in the byte represents a different flag.

2.2 Design of User Interface

I am designing 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 has the advantage that anything learnt using either tool is applicable to both, which benefits everyone.

Based on this perception, I plan to use the same option flags as nmap, as well as similar help messages and an almost identical call signature (how the program is used on the command line).

Running `./netscan.py <options> <target specification>` should be almost identical to `nmap <options> <target specification>` in terms of which scan types will be run, which hosts will be scanned and which ports are scanned. Below, you can see a concept help message, for my program with all the arguments I plan to implement.

```
usage: netscan.py <options> <target specification>
```

```
required arguments:    target specification
```

```
optional arguments:    -h, --help -Pn, -sL, -sn, -sS,  
                        -sT, -sU, -sV, -p, --ports, -0  
                        --exclude_ports
```

The above shows clearly which are required arguments, and which are optional ones. It also shows that some arguments can be called with either a short format e.g. `-p` or with a more 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 should be easier to recall the function of the more verbose flags. If the user enters erroneous data, they should be greeted by a `ValueError`, which will explain exactly what the issue was with their input, and will print out the argument that caused the error.

2.3 System Algorithms

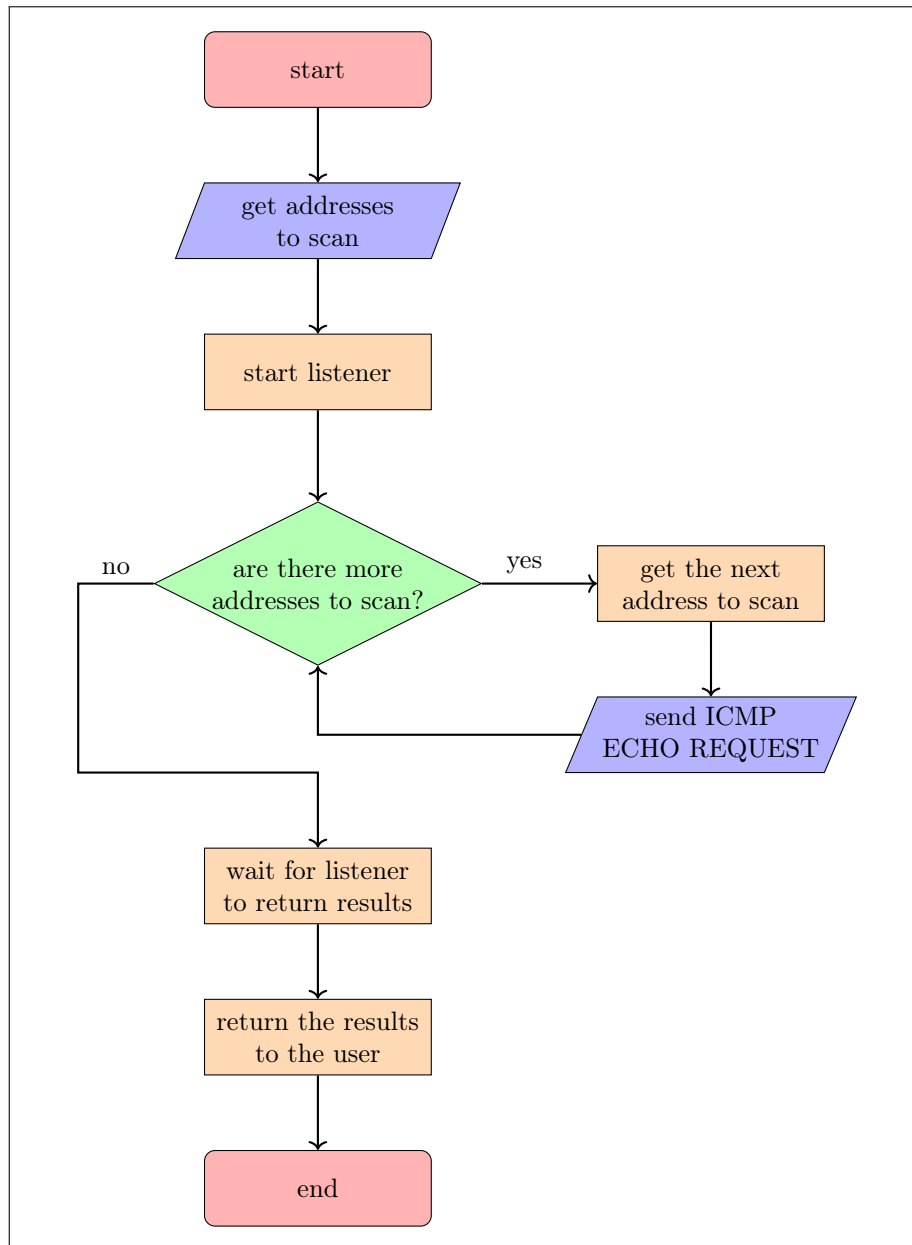


Figure 21: *The logic for how I will do Ping Scanning.*

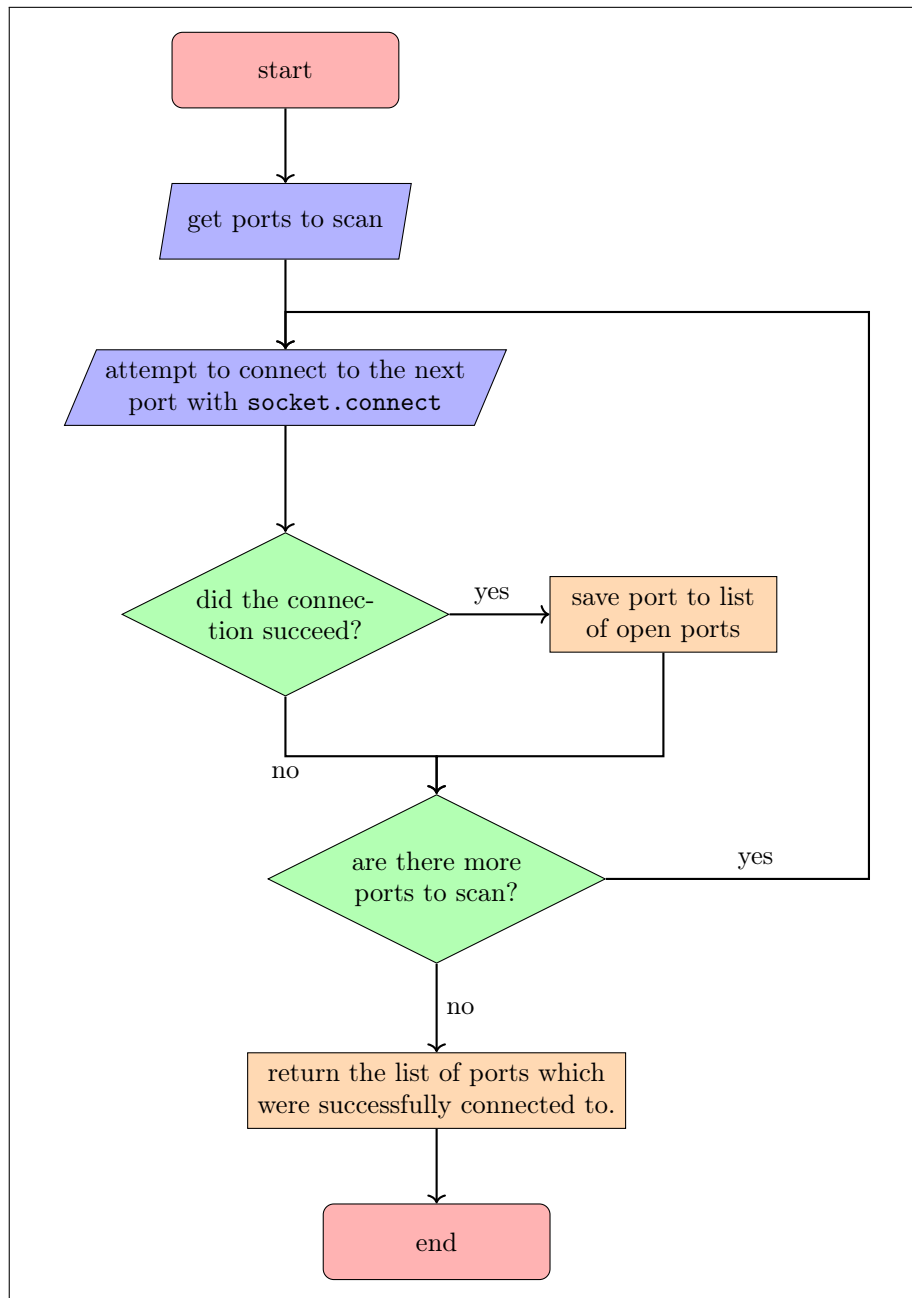


Figure 22: *The logic for how I will do TCP connect Scanning.*

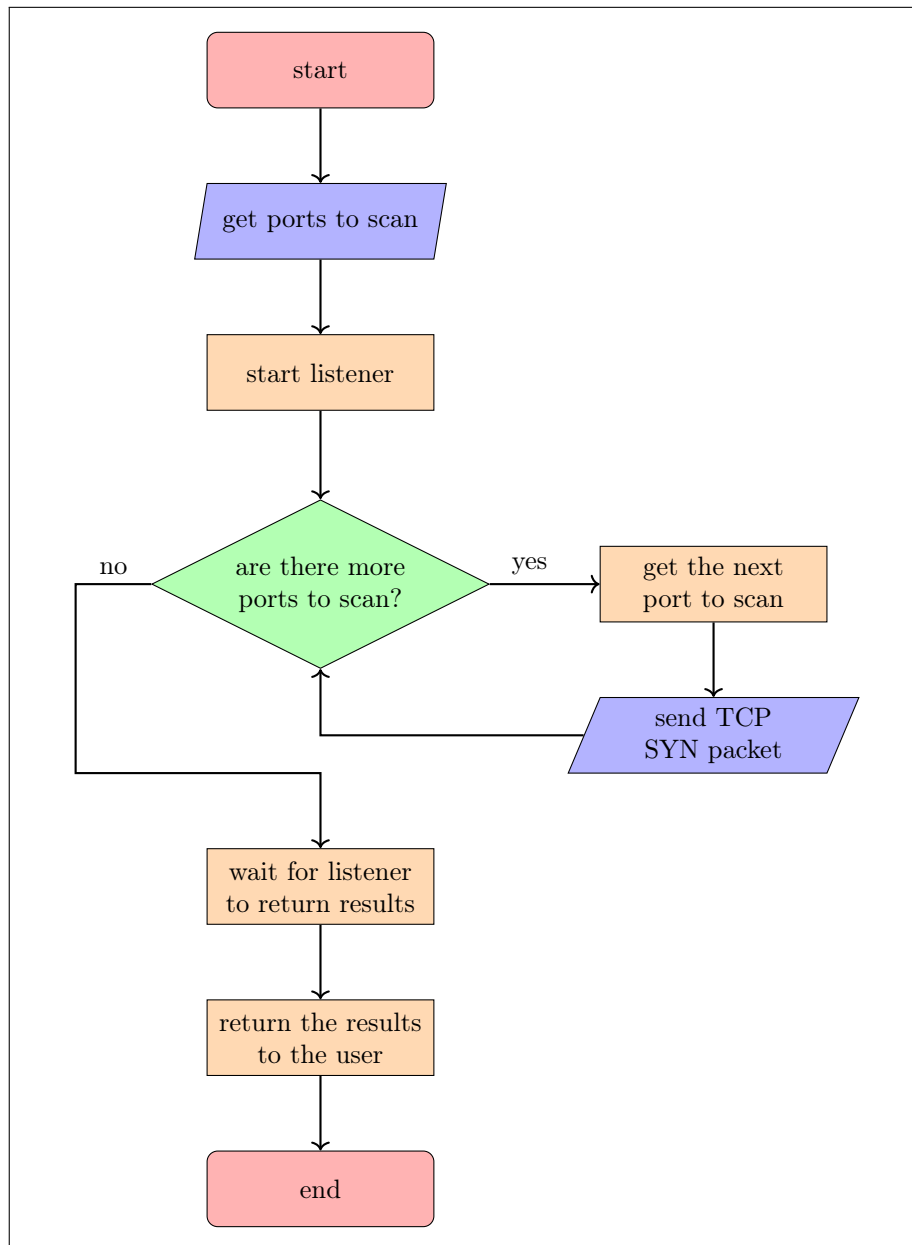


Figure 23: *The logic for how I will do TCP SYN scanning.*

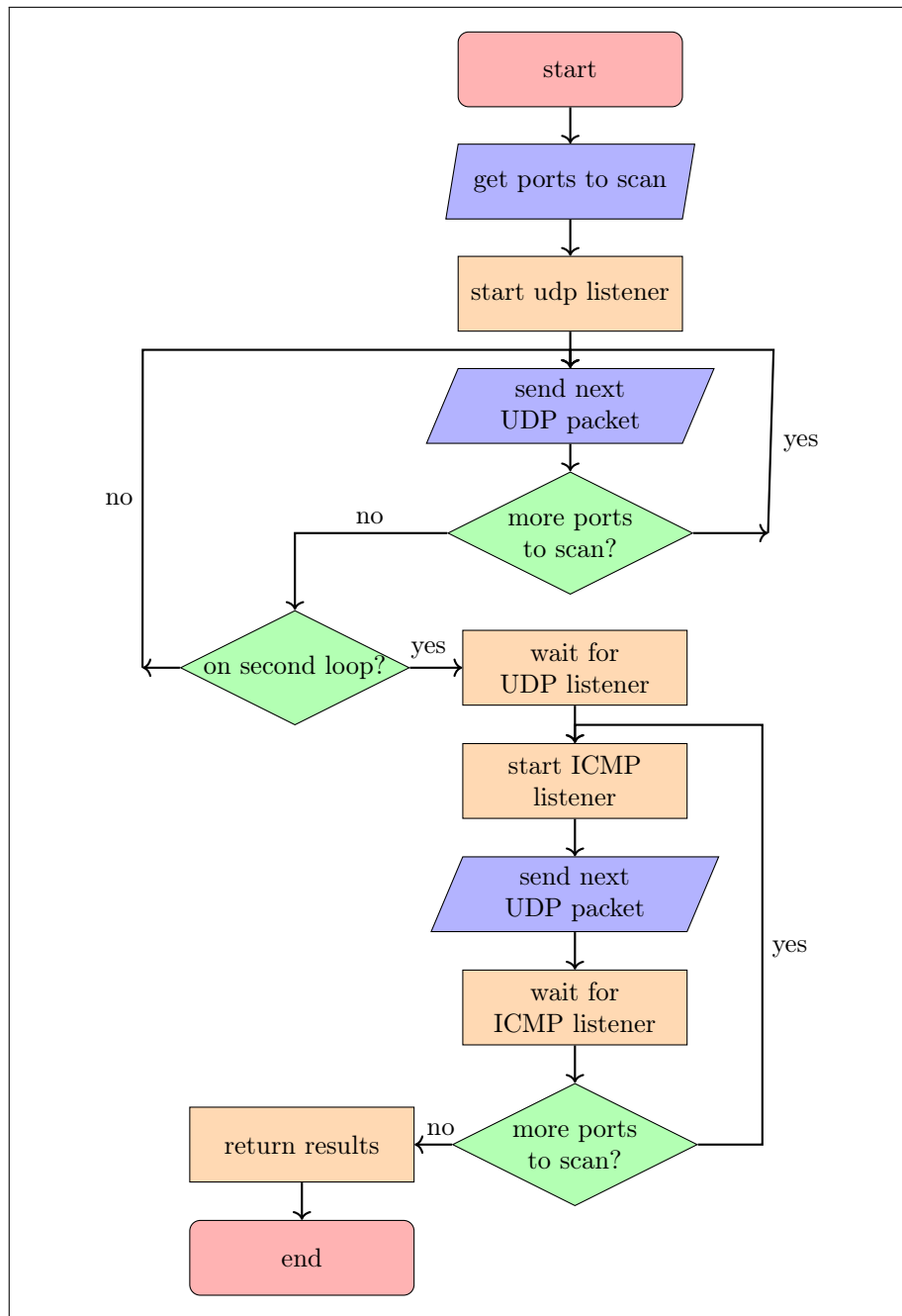


Figure 24: *The logic behind how UDP scanning works.*

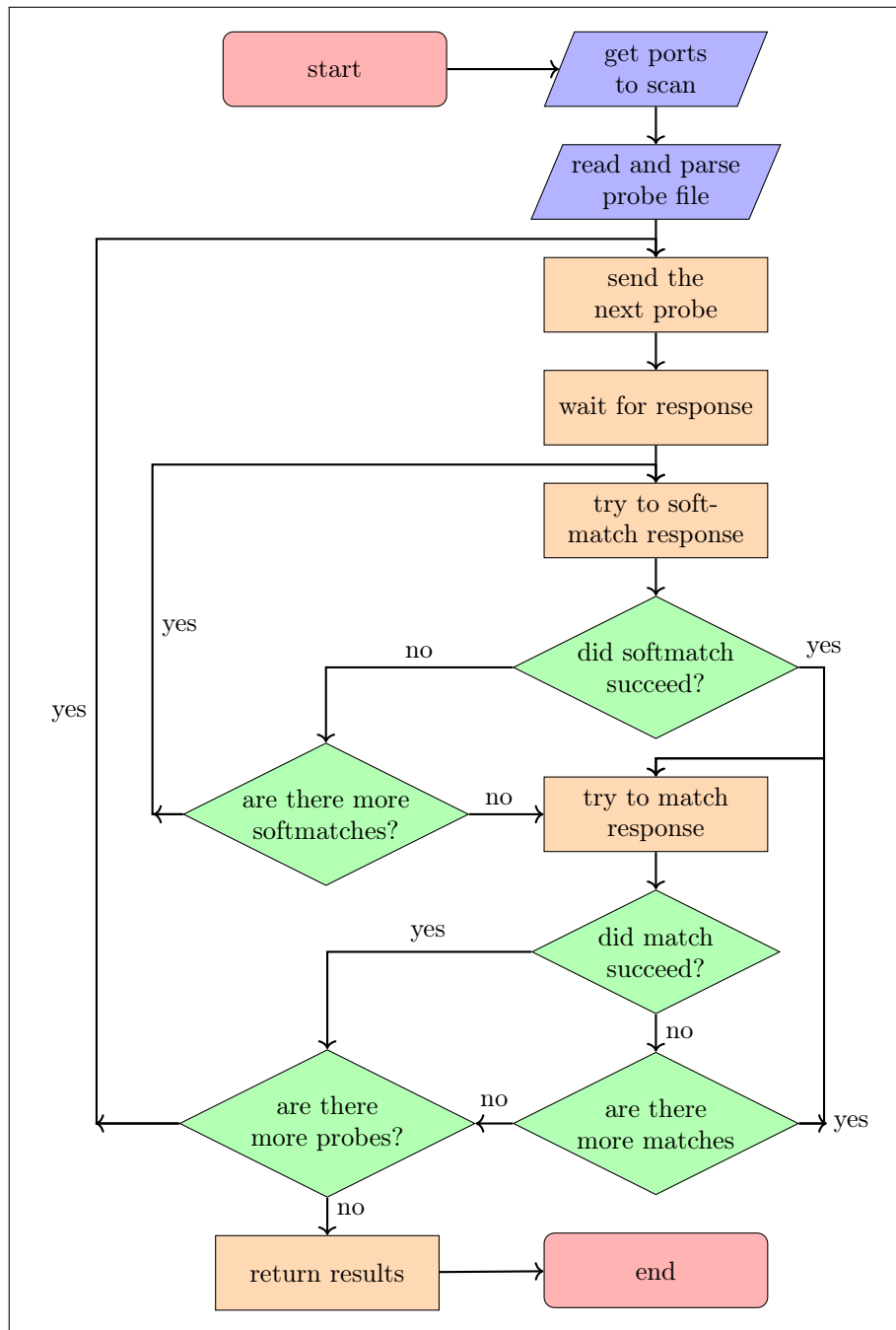


Figure 25: *The logic behind how version detection works.*

2.4 Input data validation

I plan to perform data validation in all of the functions in the fundamental modules which will hold the basic functionality for my project, such as the scanning functions. This is because my project will revolve heavily around these functions and they will need to be as error free as possible. Adding input validation to these core functions will enable me to find errors in my code earlier. For example, passing a function a list of strings instead of just a string might work in some cases, but the function will have a completely different result. These types of programming errors can be quite hard to debug, as although they may not generate errors very often but on occasions they will still break the application. Although data validation helps when programming, it will mainly be there to guide the user by showing them where in their arguments the problem is. This is far more useful than some programs, which simply exit with no information, beyond the fact that error occurred.

An example for a Python `ValueError` could be trying to turn the string "tacos" into an integer. This will result in the following error message: `ValueError: invalid literal for int() with base 10: 'tacos'` This informs you that you have tried to turn "tacos" into an integer with base 10, which is invalid. This is a clear and helpful error message, because it tells you what you tried to do that went wrong, and which argument was the one that caused the error.

2.5 Algorithm for complex structures

Algorithm 4 *My algorithm for turning a CIDR-specified subnet into a list of actual IP addresses*

```
1: procedure IP_RANGE
2:    $network\_bits \leftarrow$  number of network bits specified
3:    $ip \leftarrow$  base IP address
4:    $mask \leftarrow 0$ 
5:   for  $maskbit \leftarrow (32 - network\_bits), 31$  do
6:      $mask \leftarrow mask + 2^{maskbit}$ 
7:    $lower\_bound \leftarrow ip \text{ AND } mask$  ▷ zero the last 32-network_bits
8:    $upper\_bound \leftarrow ip \text{ OR } (mask \text{ XOR } 0xFFFFFFFF)$  ▷ turn the last
   32-network_bits to ones
9:    $addresses \leftarrow$  empty list
10:  for  $address \leftarrow lower\_bound, upper\_bound$  do
11:    append CONVERT_TO_DOT(address) to  $addresses$ 
  return  $addresses$ 
```

Algorithm 5 *My algorithm for pretty-printing a dictionary of lists of portnumbers such that ranges are specified as start-end instead of start, start+1, ..., end*

```

1: procedure COLLAPSE
2:   port_dictionary  $\leftarrow$  dictionary of lists of portnumbers
3:   key_results  $\leftarrow$  empty list  $\triangleright$  stores the formatted result for each key
4:   for key in port_dictionary do
5:     ports  $\leftarrow$  port_dict[key]
6:     result  $\leftarrow$  key + "{"
7:     if ports is empty then
8:       new_sequence  $\leftarrow$  FALSE
9:       for index  $\leftarrow$  1, (length of ports) - 1 do
10:        port = ports[index]
11:        if index = 0 then
12:          result  $\leftarrow$  result + ports[0]  $\triangleright$  append the first element
13:          if ports[index+1] = port + 1 then
14:            result  $\leftarrow$  result + "-"  $\triangleright$  begin a new sequence
15:          else
16:            result  $\leftarrow$  result + ","  $\triangleright$  not a sequence
17:          else if port + 1  $\neq$  ports[index+1] then  $\triangleright$  break in sequence
18:            result  $\leftarrow$  result + port + ","
19:            new_sequence  $\leftarrow$  TRUE
20:          else if port + 1 = ports[index+1] & new_sequence then
21:            result  $\leftarrow$  result + "-"
22:            new_sequence  $\leftarrow$  FALSE
23:          result  $\leftarrow$  result + ports[(length of ports)-1] + "}"
24:          append result to key_results
25:   return "{" + (key_results separated by ", ") + "}"

```

3 Technical Solution

I have placed all of my code in Appendix A. 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 the initial prototype of my ping scanner which uses ICMP `echo request` messages to detect hosts which are online on a given subnet. This is used to meet success criterion 4.

Appendix A.3 contains all the code which I wrote while writing a tool to translate a CIDR-specified subnet into the list of IP addresses for that subnet. It uses logic to exclude the broadcast address and host addresses for each subnet. This is used to meet success criterion 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 with creating my main application, described 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 owing to the directory structure everything has to be started from the root of the directory structure, otherwise errors occur. My solution for eliminating the errors was to run everything at the root of the directory structure, as this can call the `main()` function defined in each of the modules, and can also import all of the modules in the modules directory.

Appendix A.9 contains the code for my final application. This satisfies all of the success criteria apart from 12, which as previously discussed, is alternatively completed via version detection scanning.

Appendix A.10 contains all of the code for my unit tests which I run using `Python -m pytest`. It automatically runs each function, and delivers verbose information on each one. I have deliberately 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 26, where I have changed one of the tests so that it fails. You can see that the output of the test 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 “192.168.1.1” and in the right an extra element “192.168.1.0”. This is very helpful for preventing regressions in the code. Until I wrote these unit tests, I found that I would write a new feature and accidentally break another piece of functionality as a consequence.

```
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 26: 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 for 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 has accidental consequences that cause the application to break. I will use Wireshark to show the scanning portion of my code, and when external connections are made/custom packets created. Section 4.2 contains the tests using Wireshark; the unit tests are in Appendix A.10.

The Wireshark testing will require Wireshark and iptables. I will need to set up Wireshark in listen mode on the right interface so that it captures the packets that my program is sending. From there I will be able to inspect the sent packets and determine whether they fit what was expected in the test description or whether they don't match at all. For filtered packet tests I will need to run the command `iptables -I INPUT -s 127.0.0.1 -j DROP` and scan the localhost address, and after the test I will need to run the command `iptables -F` to flush all the iptables rules to prevent any confusion in future caused by an firewall rule that shouldn't be there.

To running the unit tests will require Python 3.7.2 and pytest 4.3.1. To run the tests I will need to run `python -m pytest` inside the Code directory. This will call pytest, which will find the tests inside the tests directory and run them. It will then display the number of tests that passed along with lots of information on the tests that failed, such as what the arguments were etc. Pytest does this via introspection of the comparison and assert commands. This means that it uses its own versions of those commands which allow it to get more information out about what went wrong, such as which element in a list was the one that caused the comparison to return false etc.

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 27 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 criterion 2.

```

networkScanner/Code on master [!?] venv:(net_scanner) pyenv:(net_scanner)
→ ./netscan.py
usage: netscan.py [-h] [-Pn] [-sL] [-sn] [-sS] [-sT] [-sU] [-sV] [-p PORTS]
                  [--exclude_ports EXCLUDE_PORTS]
                  target_spec
netscan.py: error: the following arguments are required: target_spec

```

Figure 27: 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 28 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 criterion 2.

```

networkScanner/Code on master [!?] venv:(net_scanner) pyenv:(net_scanner)
→ ./netscan.py -h
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

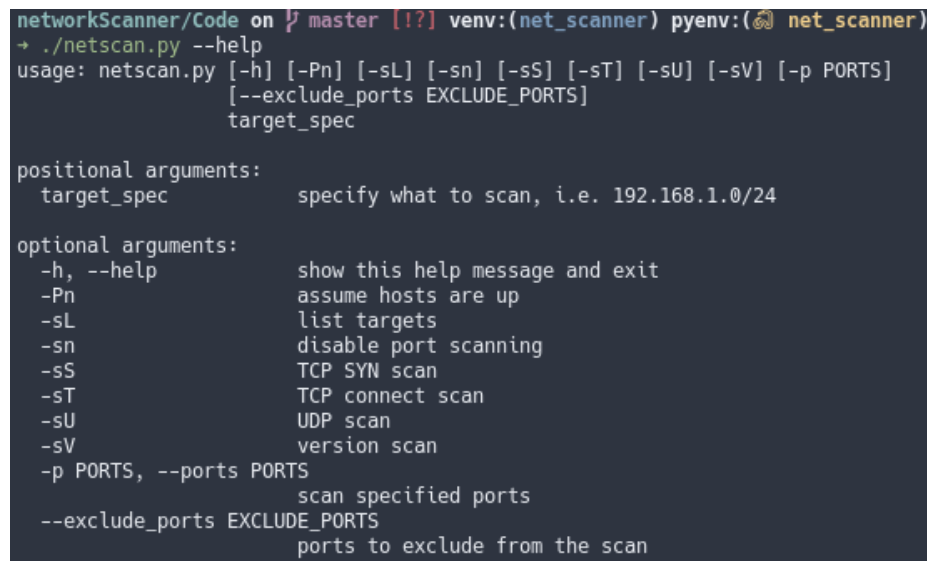
optional arguments:
  -h, --help          show this help message and exit
  -Pn                 assume hosts are up
  -sL                 list targets
  -sn                 disable port scanning
  -sS                 TCP SYN scan
  -sT                 TCP connect scan
  -sU                 UDP scan
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

```

Figure 28: 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 same output as with `-h`. This shows the same message as in the test of `-h`. To prove this, if I take the `shasum` 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 30. This shows success criterion 2.

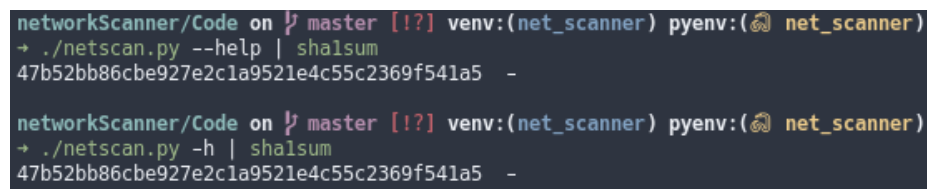


```
networkScanner/Code on master [!?] venv:(net_scanner) pyenv:(net_scanner)
→ ./netscan.py --help
usage: netscan.py [-h] [-Pn] [-sL] [-sn] [-sS] [-sT] [-sU] [-sV] [-p PORTS]
                  [--exclude_ports EXCLUDE_PORTS]
                  target_spec

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



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

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

Figure 30: 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 31 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 32, 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 meets 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 31: 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 32: 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 33, 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 criterion 3.

```
networkScanner/Code on  master [x1?] 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 33: 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 34: 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 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 the destination MAC address to send the packet to its destination as we are scanning a private IP range of my router. In Figure 35 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 36 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 criterion 5.

```
networkScanner/Code on master [x!?] venv:(net_scanner) pyenv:(net_scanner)
→ sudo ./netscan.py -Pn -p 12345 192.168.43.0/28 -sS

Scan report for: 192.168.43.11
Open ports:

Scan report for: 192.168.43.5
Open ports:

Scan report for: 192.168.43.6
Open ports:

Scan report for: 192.168.43.7
Open ports:

Scan report for: 192.168.43.13
Open ports:

Scan report for: 192.168.43.8
Open ports:

Scan report for: 192.168.43.9
Open ports:

Scan report for: 192.168.43.2
Open ports:

Scan report for: 192.168.43.14
Open ports:

Scan report for: 192.168.43.3
Open ports:

Scan report for: 192.168.43.4
Open ports:

Scan report for: 192.168.43.12
Open ports:

Scan report for: 192.168.43.10
Open ports:

Scan report for: 192.168.43.1
Open ports:
```

Figure 35: 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	30.261899876	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 36: 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. 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 39 you can see the expected TCP SYN-ACK handshake performed by my program and the script in Figure 37. You can see the output of my program in Figure 38; 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 37: Screenshot showing the script I ran to accept a connection on localhost port 12345.

```

networkScanner/Code on master [x1?] 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 38: Screenshot showing the output of my script when run with the specified flags and while the script in Figure 37 was running.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.0000000000	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 39: Screenshot showing the packet capture of the TCP SYN-ACK handshake performed by the scan in Figure 38 with the script in 37.

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 that 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. I will pass my program the same options as in the test for a TCP open port. In Figure 41, 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 40 with no open ports, showing success criteria 1 and 7.

```
networkScanner/Code on master [x1?] venv:(net_scanner) pyenv:(net_scanner) took 9s
→ sudo ./netscan.py 127.0.0.1 -p 12345 -sT
[sudo] password for tritoke:

Scan report for: 127.0.0.1
Open ports:
```

Figure 40: Screenshot showing the output of my program when run with the specified options.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	127.0.0.1	127.0.0.1	TCP	74	53892 → 12345 [SYN] Seq=
2	0.000006554	127.0.0.1	127.0.0.1	TCP	54	12345 → 53892 [RST, ACK]

Figure 41: Screenshot showing the packet capture of the TCP SYN-RST closed port indication caused by the scan in Figure 40.

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 that 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 instruct it to perform a TCP SYN scan on port 12345 and not to check that the host is up before beginning the scan. 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 42. It can be seen that port 12345 is displayed as filtered, and in the packet capture shown in Figure 43, 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.

```
networkScanner/Code on master [x1?] venv:(net_scanner) pyenv:(net_scanner) took 4s
→ sudo ./netscan.py 127.0.0.1 -p 12345 -sS -Pn

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

Figure 42: 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.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	127.0.0.1	127.0.0.1	TCP	58	38337 → 12345 [SYN]

Figure 43: Screenshot showing the packet capture of the scan in Figure 42

4.2.10 Detecting whether a UDP port is open

To show this I will perform a UDP scan on a script I had already written while developing UDP scanning. This 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 44, 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 45, you can see the packet capture of my program being run. However, the output was not precisely as expected, as I did not foresee the ICMP destination unreachable messages, which were sent by the kernel in response to the UDP probe. However, apart from those messages, the capture shows the program detecting the UDP port was open, as expected. This meets success criteria 1 and 9.

Listing 8: Script to open port 12345 to UDP.

```

1 import socket
2 from contextlib import closing
3
4 with closing(
5     socket.socket(
6         socket.AF_INET,
7         socket.SOCK_DGRAM
8     )
9 ) as s:
10     s.bind(("127.0.0.1", 12345))
11     print("opened port 12345 on localhost")
12     while True:
13         data, addr = s.recvfrom(1024)
14         s.sendto(bytes("Well hello there good sir.", "utf-8"), addr)

```

```
networkScanner/Code on P master [X!?] venv:(net_scanner) pyenv:(🐍 net_scanner)
→ sudo ./netscan.py 127.0.0.1 -p 12345 -sU -Pn

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

Figure 44: 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 45: screenshot showing the packet capture of the scan in Figure 44

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 46, you can see the output of my program when run with the options specified above. There are no ports displayed as either open or filtered. This shows that my program successfully identified the port as closed. This shows success criteria 1 and 10.

```
networkScanner/Code on P master [X!?] 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 46: 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 47: screenshot showing the packet capture of the scan in Figure 46

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 48 you can see my program correctly identifies the port as being filtered, and in Figure 49 you can see the packet capture of the scan which also as expected shows the three UDP packets with no reply packets. This shows the program meeting success criteria 1 and 11.

```
networkScanner/Code on master [x!?] 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 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	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 49: screenshot showing the packet capture of the scan in Figure 48

4.2.13 Detecting the operating system of another machine

I haven't directly added this as a feature to my project, largely because I discovered on implementing version scanning that this had the substantially the same effect. I found that in most instances it was possible to detect an operating system-dependent service and thereby deduce which operating system that machine was running. For example, if a machine is open on TCP port 22, and

Secure SHell (SSH) is detected to be running behind that port, then the machine is most likely running Linux, as Remote Desktop Protocol (RDP) is more commonly used for this purpose on Windows machines. It would become even more likely that the target machine runs Linux if the scan reveals some further information such as the Common Platform Enumeration (CPE). In Figure 50 you can see a scan of my machine where I have 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 that my machine is likely running Linux, which is the case. This partially completes success criterion 12.

```
networkScanner/Code on  master [x!?] venv:(net_scanner) pyenv:( net_scanner)
→ sudo ./netscan.py 127.0.0.1 -sV

Scan report for: 127.0.0.1
Open ports:
22/TCP      ssh
vendorproductname: OpenSSH
version: 7.9
info: protocol 2.0

CPE:
applications
vendor: openbsd
product: openssh
version: 7.9
update:
edition:
language:

Filtered ports:
```

Figure 50: *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 meets success criteria 1, 13 and 14.

```

networkScanner/Code on master [x!?] venv:(net_scanner) pyenv:(net_scanner)
→ sudo ./netscan.py 127.0.0.1 -sV

Scan report for: 127.0.0.1
Open ports:
22/TCP      ssh
vendorproductname: OpenSSH
version: 7.9
info: protocol 2.0

CPE:
applications
vendor: openbsd
product: openssh
version: 7.9
update:
edition:
language:

Filtered ports:

```

Figure 51: screenshot showing a version scan of my local machine running ssh.

4.2.15 User enters invalid ip address

To show this I will run my program with the `target_spec` option being `300.300.300.300` which is an invalid IPv4 address, because each of the octets is not between 0 and 255. I expect to see my program raise a Python `ValueError` saying that this is an invalid dot form IP address, and displaying `300.300.300.300` as the invalid IP address. In Figure 52 you can see my program's output for this invalid IP address. This shows a successful pass as it correctly identifies the invalid IP and displays the error and the argument that caused the error to the user.

```

networkScanner/Code on master [x!?] venv:(net_scanner) pyenv:(net_scanner)
→ ./netscan.py 300.300.300.300
Traceback (most recent call last):
  File "netscan.py", line 93, in <module>
    raise ValueError(f"invalid dot form IP address: [{base_addr}]")
ValueError: invalid dot form IP address: [300.300.300.300]

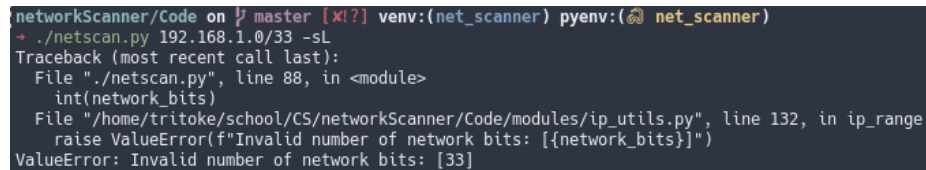
```

Figure 52: Screenshot showing the output from an invalid IP address being used.

4.2.16 User enters invalid number of network bits

To show this I will run my program and ask it to list the IP addresses specified by the subnet `192.168.1.0/33`. IP addresses are only 32 bits, long so specifying 33 network bits has no meaning, and thus is invalid data. I expect my program

to raise a `ValueError` and print out that it was an invalid number of network bits that caused the error, along with 33 being the network bits. In Figure 53 you can see that my program successfully identified the invalid number of network bits, raised the expected error, and printed the expected information.

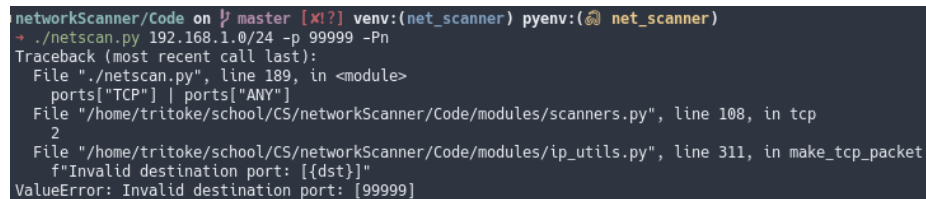


```
networkScanner/Code on master [x1?] venv:(net_scanner) pyenv:(net_scanner)
+ ./netscan.py 192.168.1.0/33 -sL
Traceback (most recent call last):
  File "./netscan.py", line 88, in <module>
    int(network_bits)
  File "/home/tritoke/school/CS/networkScanner/Code/modules/ip_utils.py", line 132, in ip_range
    raise ValueError(f"Invalid number of network bits: [{network_bits}]")
ValueError: Invalid number of network bits: [33]
```

Figure 53: Screenshot showing the output of my program when passed an invalid number of network bits.

4.2.17 User enters an invalid port number to scan

To show this I will run my program with the argument `-p 99999`. Because port numbers can only go up to 65535, this is erroneous data, and as such should generate an error message specifying that you have tried to scan an invalid destination port. In Figure 54 you can see that my program successfully identified 99999 as an invalid destination port and printed the correct error message accordingly.



```
networkScanner/Code on master [x1?] venv:(net_scanner) pyenv:(net_scanner)
+ ./netscan.py 192.168.1.0/24 -p 99999 -Pn
Traceback (most recent call last):
  File "./netscan.py", line 189, in <module>
    ports["TCP"] | ports["ANY"]
  File "/home/tritoke/school/CS/networkScanner/Code/modules/scanners.py", line 108, in tcp
    2
  File "/home/tritoke/school/CS/networkScanner/Code/modules/ip_utils.py", line 311, in make_tcp_packet
    f"Invalid destination port: [{dst}]"
ValueError: Invalid destination port: [99999]
```

Figure 54: Screenshot of my program showing the output from an invalid port number.

4.2.18 User enters an invalid number of network bits and a bad IP address

To show this I will run my program with both an invalid IP address and an invalid port number. I expect this to raise a `ValueError` for the invalid IP address. This is because the IP addresses are checked first, and thus an invalid IP address would be caught before an error could be raised for an invalid port number. To show this I will pass my program the following arguments: `192.168.1.a -p a`. In Figure 55 you can see that my program catches the

invalid IP address and raises the correct ValueError as expected, thus it passed the test.

```
networkScanner/Code on master [x!?] venv:(net_scanner) pyenv:(net_scanner)
+ ./netscan.py 192.168.1.a -p a
Traceback (most recent call last):
  File "./netscan.py", line 93, in <module>
    raise ValueError(f"invalid dot form IP address: [{base_addr}]")
ValueError: invalid dot form IP address: [192.168.1.a]
```

Figure 55: screenshot of my program showing the output from an invalid ip and an invalid port number.

4.3 Test Table

Test No.	Test Data	Expectation	Result	Fig	Success Criteria
1		usage message	Pass	27	2
2	-h	help message	Pass	28	2
3	--help	help message	Pass	29	2
4	-sL	print addresses	Pass	33	3
5	-sn	ping scan	Pass	31	4
6	-Pn	assume host up	Pass	35	5
7	-sS sT	TCP port open	Pass	38	6
8	-sS sT	TCP port closed	Pass	40	7
9	-sS	TCP port filtered	Pass	42	8
10	-sU	UDP port open	Pass	44	9
11	-sU	UDP port closed	Pass	46	10
12	-sU	UDP port filtered	Pass	48	11
13	-sV	OS detection	Partial	50	12
14	-sV	service detection	Pass	50	13
15	-sV	version detection	Pass	50	14
16		invalid IP	Pass	52	
17		invalid subnet	Pass	53	
18		invalid port & IP	Pass	54	

5 Evaluation

5.1 Reflection on final outcome

Overall, I am very happy with how my program has turned out. At the beginning of this project, I did not believe I could do anywhere near as much as I have done. My rationale for choosing this project was to learn more about low level networking, and networking in general, because these were areas where I felt my knowledge was deficient. I greatly enjoyed learning about various aspects of networking structure and protocols. For instance, I found it fascinating to discover that the order that bytes are packed into the packet matters enormously, and to learn how users are automatically assigned an IP address on joining a network.

There were many areas where I encountered difficulty while developing this program. The first problem was understanding what a packet is, and how I send one that I have made myself using Python. There seemed to be very little in the way of documentation, so I ended up reading many peoples' answers to questions on the Stack Overflow website, where they had encountered similar problems. Once I discovered that I could simply open a socket in raw mode and use the `socket.sendto` method to send any arbitrary bytes to an address and Python would handle the IP header, it became easier to make progress. The next major problem was getting the checksum correct for TCP packets, because they use a pseudo-header, which is sort of a header made from fields in the IP header that are not in the TCP header. They are used to calculate the checksum for the TCP header. Figuring out how to pack these properly, and in what endianness, turned out to be one of the biggest difficulties in getting TCP SYN scanning working. I had a completely different problem when it came to implementing UDP scanning. This is because when a UDP packet arrives and is destined for a closed port, an ICMP destination unreachable message is sent back, and on Linux systems these are time-limited to a maximum of one per second. Thus, I had to come up with a strategy that could deal with problem. In the end, I added a maximum wait time for each packet sent, and instructed the program to listen for that length of time to determine if an ICMP destination unreachable message was received. Implementing version scanning came with its own difficulties. Parsing the file which defines the probes and all the directives which are needed to interpret the service's response was challenging, as was matching the returned data using a regular expression that was parsed from a file.

5.2 Evaluation against objectives

The criteria identified at the start of the project for determining its success were the following:

1. Probe another computer's networking from a black box perspective.
2. To help the user with usage/help messages when prompted.
3. Translate 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.

Criterion 1, probing another computer's networking from a black box perspective. This criterion is satisfied by many of the scanning options my program supports, including `Connect()` scanning, SYN scanning, and version detection. This is met by the tests in sections 4.2.4, 4.2.7, 4.2.8 4.2.9, 4.2.10, 4.2.11, 4.2.12 and 4.2.14.

Criterion 2, sending usage and help messages when prompted. This criterion is satisfied by the tests in Sections 4.2.1, 4.2.2 and 4.2.3.

Criterion 3, translating CIDR specified subnets. This criterion is satisfied by the test in Section 4.2.5.

Criterion 4, send ICMP ECHO requests. This criterion is satisfied by the test in Section 4.2.4.

Criterion 5, perform any scan without first checking that the host is up. This criterion is satisfied by the test in Section 4.2.6.

Criterion 6, detect an open TCP port. This criterion is satisfied by the test in Section 4.2.7.

Criterion 7, detect a closed TCP port. This criterion is satisfied by the test in Section 4.2.8.

Criterion 8, detect a filtered TCP port. This criterion is satisfied by the test in Section 4.2.9.

Criterion 9, detect an open UDP port. This criterion is satisfied by the test in Section 4.2.10.

Criterion 10, detect a closed UDP port. This criterion is satisfied by the test in Section 4.2.11.

Criterion 11, detect a filtered UDP port. This criterion is satisfied by the test in Section 4.2.12.

Criterion 12, detect the Operating System running on a remote machine. This criterion is partially satisfied by the version detection scanning and an example of detecting the operating system via version/service detection is shown in Section 4.2.13.

Criteria 13 and 14 are both satisfied by the test in Section 4.2.14.

In conclusion, I have satisfied completely all but one of the success criteria, and have partially completed the remaining criterion. Having finished my application, I met with my school's head of network services Mr Blake. To discuss the application and to gather some end user feedback. He was satisfied with the application and noted that it did what it said on the tin. He especially liked that the application was hard to break by design in terms of the minimal user input and stringent data validation combined with helpful error messages.

5.3 Potential improvements

I believe that I could improve my program if it were to have a dedicated option for operating system detection. It is clear that as a single individual, it would not be feasible for me to collect the required number of operating system signatures myself. Therefore, in order to implement this feature, I would need use one of the files from nmap: `nmap-os-db`, which contains a database of TCP/IP stack fingerprints that show how the TCP and IP protocols are implemented in virtually all currently-used operating systems, as well as in different versions of Linux and different service packs and versions of Windows.

A Technical Solution

A.1 icmp_ping

Listing 9: *A prototype program for sending ICMP ECHO REQUEST packets*

```
1  #!/usr/bin/env python
2  import socket
3  import struct
4  import os
5  import time
6  from modules.ip_utils import ip_checksum
7
8
9  def main() -> None:
10     ICMP_ECHO_REQUEST = 8
11
12     # opens a raw socket for the ICMP protocol
13     ping_sock = socket.socket(
14         socket.AF_INET,
15         socket.SOCK_RAW,
16         socket.IPPROTO_ICMP
17     )
18     # allows manual IP header creation
19     # ping_sock.setsockopt(socket.SOL_IP, socket.IP_HDRINCL, 1)
20
21     ID = os.getpid() & 0xFFFF
22
23     # the two zeros are the code and the dummy checksum, the one is the
24     # sequence number
25     dummy_header = struct.pack("bbHHh", ICMP_ECHO_REQUEST, 0, 0, ID, 1)
26
27     data = struct.pack(
28         "d", time.time()
29     ) + bytes(
30         (192 - struct.calcsize("d")) * "A",
31         "ascii"
32     )
33     # the data to send in the packet
34     checksum = socket.htons(ip_checksum(dummy_header + data))
35     # calculates the checksum for the packet and psuedo header
36     header = struct.pack("bbHHh", ICMP_ECHO_REQUEST, 0, checksum, ID, 1)
37     # packs the packet header
38     packet = header + data
39     # concatonates the header and the data to form the final packet.
40     ping_sock.sendto(packet, ("127.0.0.1", 1))
41     # sends the packet to localhost
```

Listing 10: *A prototype program for receiving ICMP ECHO REQUEST packets*

```
1  #!/usr/bin/env python
2  from modules import headers
3  import socket
4  from typing import List
5
6
7  def main() -> None:
8      # socket object using an IPV4 address, using only raw socket access,
9      set
10     # ICMP protocol
11     ping_sock = socket.socket(
12         socket.AF_INET,
13         socket.SOCK_RAW,
14         socket.IPPROTO_ICMP
15     )
16
17     packets: List[bytes] = []
18
19     while len(packets) < 1:
20         recPacket, addr = ping_sock.recvfrom(1024)
21         ip = headers.ip(recPacket[:20])
22         icmp = headers.icmp(recPacket[20:28])
23
24         print(ip)
25         print()
26         print(icmp)
27         print("\\n")
28
29         packets.append(recPacket)
```

A.2 ping_scanner

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

```
1  #!/usr/bin/env python
2  from modules import headers
3  from modules import ip_utils
4  import socket
5  import struct
6  import time
7  from contextlib import closing
8  from itertools import repeat
9  from math import log10, floor
10 from multiprocessing import Pool
11 from os import getpid
12 from typing import Set, Tuple
13
```

```

14
15 def sig_figs(x: float, n: int) -> float:
16     """
17     rounds x to n significant figures.
18     sig_figs(1234, 2) = 1200.0
19     """
20     return round(x, n - (1 + int(floor(log10(abs(x))))))
21
22
23 def ping_listener(
24     ID: int,
25     timeout: float
26 ) -> Set[Tuple[str, float, headers.ip]]:
27     """
28     Takes in a process id and a timeout and returns
29     a list of addresses which sent ICMP ECHO REPLY
30     packets with the packed id matching ID in the time given by timeout.
31     """
32     ping_sock = socket.socket(
33         socket.AF_INET,
34         socket.SOCK_RAW,
35         socket.IPPROTO_ICMP
36     )
37     # opens a raw socket for sending ICMP protocol packets
38     time_remaining = timeout
39     addresses = set()
40     while True:
41         time_waiting = ip_utils.wait_for_socket(ping_sock,
42             time_remaining)
43         # time_waiting stores the time the socket took to become readable
44         # or returns minus one if it ran out of time
45
46         if time_waiting == -1:
47             break
48         time_recieved = time.time()
49         # store the time the packet was recieved
50         recPacket, addr = ping_sock.recvfrom(1024)
51         # recieve the packet
52         ip = headers.ip(recPacket[:20])
53         # unpack the IP header into its respective components
54         icmp = headers.icmp(recPacket[20:28])
55         # unpack the time from the packet.
56         time_sent = struct.unpack(
57             "d",
58             recPacket[28:28 + struct.calcsize("d")]
59         )[0]
60         # unpack the value for when the packet was sent
61         time_taken: float = time_recieved - time_sent
62         # calculate the round trip time taken for the packet
63         if icmp.id == ID:

```

```

63         # if the ping was sent from this machine then add it to the
        # list of
64         # responses
65         ip_address, port = addr
66         addresses.add((ip_address, time_taken, ip))
67     elif time_remaining <= 0:
68         break
69     else:
70         continue
71     # return a list of all the addresses that replied to our ICMP echo
    request.
72     return addresses
73
74
75 def main() -> None:
76     with closing(
77         socket.socket(
78             socket.AF_INET,
79             socket.SOCK_RAW,
80             socket.IPPROTO_ICMP
81         )
82     ) as ping_sock:
83         ip_addresses = ["127.0.0.1"] # ip_utils.ip_range("192.168.43.0",
            24)
84         # generate the range of IP addresses to scan.
85         # get the local ip address
86         addresses = [
87             ip
88             for ip in ip_addresses
89             if (
90                 not ip.endswith(".0")
91                 and not ip.endswith(".255")
92             )
93         ]
94
95         # initialise a process pool
96         p = Pool(1)
97         # get the local process id for use in creating packets.
98         ID = getpid() & 0xFFFF
99         # run the listeners.ping function asynchronously
100        replied = p.apply_async(ping_listener, (ID, 5))
101        time.sleep(0.01)
102        for address in zip(addresses, repeat(1)):
103            try:
104                packet = ip_utils.make_icmp_packet(ID)
105                ping_sock.sendto(packet, address)
106            except PermissionError:
107                ip_utils.eprint("raw sockets require root priveleges,
                    exiting")
108                exit()

```

```

109     p.close()
110     p.join()
111     # close and join the process pool to so that all the values
112     # have been returned and the pool closed
113     hosts_up = replied.get()
114     # get the list of addresses that replied to the echo request
        from the
115     # listener function
116     print("\n".join(
117         f"host: [{host}]\t" +
118         "responded to an ICMP ECHO REQUEST in " +
119         f"{str(sig_figs(taken, 2))+ 's':<10s}" +
120         f"ttl: [{ip_head.time_to_live}]"
121         for host, taken, ip_head in hosts_up
122     ))

```

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*

```

1  #!/usr/bin/env python
2  import re
3  from modules.ip_utils import ip_range, dot_to_long
4
5
6  if __name__ == '__main__':
7      from argparse import ArgumentParser
8      parser = ArgumentParser()
9      parser.add_argument(
10         "ip_subnet",
11         help="The CIDR form ip/subnet that you wish to print" +
12             "the IP addresses specified by."
13     )
14     args = parser.parse_args()
15     CIDR_regex = re.compile(r"(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}/\d+)")
16     search = CIDR_regex.search(args.ip_subnet)
17     if search:
18         ip, network_bits = search.group(1).split("/")
19         print("\n".join(
20             sorted(
21                 ip_range(ip, int(network_bits)),
22                 key=dot_to_long
23             )
24         ))

```

A.4 tcp_scan

A.4.1 connect_scan

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

```
1  #!/usr/bin/python3
2  from contextlib import closing
3  import socket
4  LOCAL_IP = "192.168.1.159"
5  PORT = 22
6
7  address = ("127.0.0.1", 22)
8
9  with closing(
10     socket.socket(
11         socket.AF_INET,
12         socket.SOCK_STREAM
13     )
14 ) as s:
15     try:
16         s.connect(address)
17         print(f"connection on port {PORT} succeeded")
18     except ConnectionRefusedError:
19         print(f"port {PORT} is closed")
```

Listing 14: *A program that performs TCP connect scanning*

```
1  #!/usr/bin/python3
2
3  from typing import List, Set
4
5
6  def connect_scan(address: str, ports: Set[int]) -> List[int]:
7      import socket
8      from contextlib import closing
9      open_ports: List[int] = []
10      for port in ports:
11          # loop through each port in the list of ports to scan
12          try:
13              with closing(
14                  socket.socket(
15                      socket.AF_INET,
16                      socket.SOCK_STREAM
17                  )
18              ) as s:
19                  # open an IPV4 TCP socket
20                  s.connect((address, port))
```

```

21         # attempt to connect the newly created socket to the
           target
22         # address and port
           open_ports.append(port)
23         # if the connection was successful then add the port to
           the
24         # list of open ports
           except ConnectionRefusedError:
25             pass
26     return open_ports
27
28
29
30
31 def main() -> None:
32     open_ports = connect_scan("192.168.43.225", set(range(65535)))
33     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)*

```

1  #!/usr/bin/python3.7
2  from contextlib import closing
3  import socket
4  import ip_utils
5
6  dest_port = 22
7  src_port = ip_utils.get_free_port()
8  local_ip = ip_utils.get_local_ip()
9  dest_ip = "192.168.1.159"
10 local_ip = dest_ip = "127.0.0.1"
11 loc_long = ip_utils.dot_to_long(local_ip)
12
13 SYN = 2
14 RST = 4
15
16
17
18 with closing(
19     socket.socket(
20         socket.AF_INET,
21         socket.SOCK_RAW,
22         socket.IPPROTO_TCP
23     )
24 ) as s:
25     tcp_packet = ip_utils.make_tcp_packet(
26         src_port,
27         dest_port,

```

```

28         local_ip,
29         dest_ip,
30         SYN
31     )
32     if tcp_packet is not None:
33         s.sendto(tcp_packet, (dest_ip, dest_port))
34     else:
35         print(f"Couldn't make TCP packet with supplied arguments:",
36               f"source port: [{src_port}]",
37               f"destination port: [{dest_port}]",
38               f"local ip: [{local_ip}]",
39               f"destination ip: [{dest_ip}]",
40               f"SYN flag: [{SYN}]",
41               sep="\n")

```

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

```

1  #!/usr/bin/python3.7
2  from modules import headers
3  from modules import ip_utils
4  import socket
5  from contextlib import closing
6  from multiprocessing import Pool
7  from typing import List, Set, Tuple
8
9
10 def syn_listener(address: Tuple[str, int], timeout: float) -> List[int]:
11     """
12     This function is run asynchronously and listens for
13     TCP ACK responses to the sent TCP SYN msg.
14     """
15     print(f"address: [{address}]\ntimeout: [{timeout}]")
16     open_ports: List[int] = []
17     with closing(
18         socket.socket(
19             socket.AF_INET,
20             socket.SOCK_RAW,
21             socket.IPPROTO_TCP
22         ) as s:
23         s.bind(address)
24         # bind the raw socket to the listening address
25         time_remaining = timeout
26         print("started listening")
27         while True:
28             time_taken = ip_utils.wait_for_socket(s, time_remaining)
29             # wait for the socket to become readable
30             if time_taken == -1:
31                 break
32             else:

```



```

33         time_remaining -= time_taken
34     packet = s.recv(1024)
35     # recieve the packet data
36     tcp = headers.tcp(packet[20:40])
37     if tcp.flags == 0b00010010: # syn ack
38         print(tcp)
39         open_ports.append(tcp.source)
40         # check that the header contained the TCP ACK flag and if
41         it
42         # did append it
43     else:
44         continue
45     print("finished listening")
46     return open_ports
47
48 def syn_scan(dest_ip: str, portlist: Set[int]) -> List[int]:
49     src_port = ip_utils.get_free_port()
50     # request a local port to connect from
51     local_ip = ip_utils.get_local_ip()
52     p = Pool(1)
53     listener = p.apply_async(syn_listener, ((local_ip, src_port), 5))
54     # start the TCP ACK listener in the background
55     print("starting scan")
56     for port in portlist:
57         packet = ip_utils.make_tcp_packet(src_port, port, local_ip,
58                                           dest_ip, 2)
59         # create a TCP packet with the syn flag
60         with closing(
61             socket.socket(
62                 socket.AF_INET,
63                 socket.SOCK_RAW,
64                 socket.IPPROTO_TCP
65             ) as s:
66             s.sendto(packet, (dest_ip, port))
67             # send the packet to its destination
68
69     print("finished scan")
70     p.close()
71     p.join()
72     open_ports = listener.get()
73     # collect the list of ports that responded to the TCP SYN message
74     print(open_ports)
75     return open_ports
76
77
78 def main() -> None:
79     dest_ip = "127.0.0.1"
80     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*

```
1  #!/usr/bin/ python
2  from contextlib import closing
3  import ip_utils
4  import socket
5
6  dest_ip = "192.168.1.1"
7  dest_port = 68
8  local_ip = ip_utils.get_local_ip()
9  local_port = ip_utils.get_free_port()
10
11 local_ip = dest_ip = "127.0.0.1"
12
13 address = (dest_ip, dest_port)
14
15 with closing(
16     socket.socket(
17         socket.AF_INET,
18         socket.SOCK_RAW,
19         socket.IPPROTO_UDP
20     )) as s:
21     try:
22         pkt = ip_utils.make_udp_packet(
23             local_port,
24             dest_port,
25             local_ip,
26             dest_ip
27         )
28         if pkt is not None:
29             packet = bytes(pkt)
30             s.sendto(packet, address)
31         else:
32             print(
33                 "Error making packet.",
34                 f"local port: [{local_port}]",
35                 f"destination port: [{dest_port}]",
36                 f"local ip: [{local_ip}]",
37                 f"destination ip: [{dest_ip}]",
38                 sep="\n"
39             )
40     except socket.error:
41         raise
```

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

```
1  #!/usr/bin/env python
2  from modules import headers
3  from modules import ip_utils
4  import socket
5  import time
6  from collections import defaultdict
7  from contextlib import closing
8  from multiprocessing import Pool
9  from typing import Set, DefaultDict
10
11
12  def udp_listener(dest_ip: str, timeout: float) -> Set[int]:
13      """
14          This listener detects UDP packets from dest_ip in the given timespan,
15          all ports that send direct replies are marked as being open.
16          Returns a list of open ports.
17      """
18
19      time_remaining = timeout
20      ports: Set[int] = set()
21      with socket.socket(
22              socket.AF_INET,
23              socket.SOCK_RAW,
24              socket.IPPROTO_UDP
25      ) as s:
26          while True:
27              time_taken = ip_utils.wait_for_socket(s, time_remaining)
28              if time_taken == -1:
29                  break
30              else:
31                  time_remaining -= time_taken
32                  packet = s.recv(1024)
33                  ip = headers.ip(packet[:20])
34                  udp = headers.udp(packet[20:28])
35                  # unpack the UDP header
36                  if dest_ip == ip.source and ip.protocol == 17:
37                      ports.add(udp.src)
38
39      return ports
40
41
42  def icmp_listener(src_ip: str, timeout: float = 2) -> int:
43      """
44          This listener detects ICMP destination unreachable
45          packets and returns the icmp code.
46          This is later used to mark them as either close, open|filtered,
47          filtered.
48      3 -> closed
```

```

48     0|1|2|9|10|13 -> filtered
49     -1 -> error with arguments
50     open|filtered means that they are either open or
51     filtered but return nothing.
52     """
53
54     ping_sock = socket.socket(
55         socket.AF_INET,
56         socket.SOCK_RAW,
57         socket.IPPROTO_ICMP
58     )
59     # open raw socket to listen for ICMP destination unreachable packets
60     time_remaining = timeout
61     code = -1
62     while True:
63         time_waiting = ip_utils.wait_for_socket(ping_sock,
64             time_remaining)
65         # wait for socket to be readable
66         if time_waiting == -1:
67             break
68         else:
69             time_remaining -= time_waiting
70             recPacket, addr = ping_sock.recvfrom(1024)
71             # receive the packet
72             ip = headers.ip(recPacket[:20])
73             icmp = headers.icmp(recPacket[20:28])
74             valid_codes = [0, 1, 2, 3, 9, 10, 13]
75             if (
76                 ip.source == src_ip
77                 and icmp.type == 3
78                 and icmp.code in valid_codes
79             ):
80                 code = icmp.code
81                 break
82             elif time_remaining <= 0:
83                 break
84             else:
85                 continue
86     ping_sock.close()
87     return code
88
89 def udp_scan(
90     dest_ip: str,
91     ports_to_scan: Set[int]
92 ) -> DefaultDict[str, Set[int]]:
93     """
94     Takes in a destination IP address in either dot or long form and
95     a list of ports to scan. Sends UDP packets to each port specified
96     in portlist and uses the listeners to mark them as open,

```

```

    open|filtered,
197 filtered, closed they are marked open|filtered if no response is
198 recieved at all.
199 """
200
201 local_ip = ip_utils.get_local_ip()
202 local_port = ip_utils.get_free_port()
203 # get local ip address and port number
204 ports: DefaultDict[str, Set[int]] = defaultdict(set)
205 ports["REMAINING"] = ports_to_scan
206 p = Pool(1)
207 udp_listen = p.apply_async(udp_listener, (dest_ip, 4))
208 # start the UDP listener
209 with closing(
210     socket.socket(
211         socket.AF_INET,
212         socket.SOCK_RAW,
213         socket.IPPROTO_UDP
214     )
215 ) as s:
216     for _ in range(2):
217         # repeat 3 times because UDP scanning comes
218         # with a high chance of packet loss
219         for dest_port in ports["REMAINING"]:
220             try:
221                 packet = ip_utils.make_udp_packet(
222                     local_port,
223                     dest_port,
224                     local_ip,
225                     dest_ip
226                 )
227                 # create the UDP packet to send
228                 s.sendto(packet, (dest_ip, dest_port))
229                 # send the packet to the currently scanning address
230             except socket.error:
231                 packet_bytes = " ".join(map(hex, packet))
232                 print(
233                     "The socket modules sendto method with the
234                     following",
235                     "argument resulting in a socket error.",
236                     f"\npacket: [{packet_bytes}]\n",
237                     "address: [{dest_ip, dest_port}]"
238                 )
239
240 p.close()
241 p.join()
242
243 ports["OPEN"].update(udp_listen.get())
244 ports["REMAINING"] -= ports["OPEN"]

```

```

145     # only scan the ports which we know are not open
146     with closing(
147         socket.socket(
148             socket.AF_INET,
149             socket.SOCK_RAW,
150             socket.IPPROTO_UDP
151         )
152     ) as s:
153         for dest_port in ports["REMAINING"]:
154             try:
155                 packet = ip_utils.make_udp_packet(
156                     local_port,
157                     dest_port,
158                     local_ip,
159                     dest_ip
160                 )
161                 # make a new UDP packet
162                 p = Pool(1)
163                 icmp_listen = p.apply_async(icmp_listener, (dest_ip,))
164                 # start the ICMP listener
165                 time.sleep(1)
166                 s.sendto(packet, (dest_ip, dest_port))
167                 # send packet
168                 p.close()
169                 p.join()
170                 icmp_code = icmp_listen.get()
171                 # recieve ICMP code from the ICMP listener
172                 if icmp_code in {0, 1, 2, 9, 10, 13}:
173                     ports["FILTERED"].add(dest_port)
174                 elif icmp_code == 3:
175                     ports["CLOSED"].add(dest_port)
176             except socket.error:
177                 packet_bytes = " ".join(map("{:02x}".format, packet))
178                 ip_utils.eprint(
179                     "The socket modules sendto method with the following",
180                     "argument resulting in a socket error.",
181                     f"\npacket: [{packet_bytes}]\n",
182                     "address: [{dest_ip, dest_port}]"
183                 )
184     # this creates a new set which contains all the elements that
185     # are in the list of ports to be scanned but have not yet
186     # been classified
187     ports["OPEN|FILTERED"] = (
188         ports["REMAINING"]
189         - ports["OPEN"]
190         - ports["FILTERED"]
191         - ports["CLOSED"]
192     )
193     # set comprehension to update the list of open filtered ports
194     return ports

```

```

195
196
197 def main() -> None:
198     ports = udp_scan("127.0.0.1", {22, 68, 53, 6969})
199     print(f"Open ports: {ports['OPEN']}")
200     print(f"Open or filtered ports: {ports['OPEN|FILTERED']}")
201     print(f"Filtered ports: {ports['FILTERED']}")
202     print(f"Closed ports: {ports['CLOSED']}")

```

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

```

1  #!/usr/bin/env python
2
3  import socket
4  from contextlib import closing
5
6  with closing(
7      socket.socket(
8          socket.AF_INET,
9          socket.SOCK_DGRAM
10     )
11 ) as s:
12     s.bind(("127.0.0.1", 12345))
13     print("opened port 12345 on localhost")
14     while True:
15         data, addr = s.recvfrom(1024)
16         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.*

```

1  #!/usr/bin/env python
2  from typing import Dict, Set, Pattern, Tuple, DefaultDict
3  from functools import reduce
4  from collections import defaultdict
5  from modules import directives
6  import re
7  import operator
8
9  # type annotaion for the container which
10 # holds the probes. I have abstracted it from
11 # the function definition because multiple functions
12 # depend on it and they weren't all getting updated
13 # if I needed to change the function signature.
14 PROBE_CONTAINER = DefaultDict[str, Dict[str, directives.Probe]]
15

```

```

16
17 def parse_ports(portstring: str) -> DefaultDict[str, Set[int]]:
18     """
19     This function takes in a port directive
20     and returns a set of the ports specified.
21     A set is used because it is O(1) for contains
22     operations as opposed for O(N) for lists.
23     """
24     # matches both the num-num port range format
25     # and the plain num port specification
26     # num-num form must come first otherwise it breaks.
27     proto_regex = re.compile(r"([ TU]):?([0-9,-]+)")
28     # THE SPACE IS IMPORTANT!!!
29     # it allows ports specified before TCP/UDP ports
30     # to be specified globally as in for all protocols.
31
32     pair_regex = re.compile(r"(\d+)-(\d+)")
33     single_regex = re.compile(r"(\d+)")
34     ports: DefaultDict[str, Set[int]] = defaultdict(set)
35     # searches contains the result of trying the pair_regex
36     # search against all of the command seperated
37     # port strings
38
39     for protocol, portstring in proto_regex.findall(portstring):
40         pairs = pair_regex.findall(portstring)
41         # for each pair of numbers in the pairs list
42         # seperate each number and cast them to int
43         # then generate the range of numbers from x[0]
44         # to x[1]+1 then cast this range to a list
45         # and "reduce" the list of lists by joining them
46         # with operator.ior (inclusive or) and then let
47         # ports be the set of all the ports in that list.
48         proto_map = {
49             " ": "ANY",
50             "U": "UDP",
51             "T": "TCP"
52         }
53         if pairs:
54             def pair_to_ports(pair: Tuple[int, int]) -> Set[int]:
55                 """
56                 a function to go from a port pair i.e. (80-85)
57                 to the set of specified ports: {80,81,82,83,84,85}
58                 """
59                 start, end = pair
60                 return set(range(start, end+1))
61             # ports contains the set of all ANY/TCP/UDP specified ports
62             ports[proto_map[protocol]] = set(reduce(
63                 operator.ior,
64                 map(pair_to_ports, pairs)
65             ))

```



```

66
67     singles = single_regex.findall(portstring)
68     # for each of the ports that are specified on their own
69     # cast them to int and update the set of all ports with
70     # that list.
71     ports[proto_map[protocol]].update(map(int, singles))
72
73     return ports
74
75
76 def parse_probes(probe_file: str) -> PROBE_CONTAINER:
77     """
78     Extracts all of the probe directives from the
79     file pointed to by probe_file.
80     """
81     # lines contains each line of the file which doesn't
82     # start with a # and is not empty.
83     lines = [
84         line
85         for line in open(probe_file).read().splitlines()
86         if line and not line.startswith("#")
87     ]
88
89     # list holding each of the probe directives.
90     probes: PROBE_CONTAINER = defaultdict(dict)
91
92     regexes: Dict[str, Pattern] = {
93         "probe": re.compile(r"Probe (TCP|UDP) (\S+) q\|(.*)\|"),
94         "match": re.compile(" ".join([
95             r"(?P<type>softmatch|match)",
96             r"(?P<service>\S+)",
97             r"m([@/%=|])(?P<regex>.+?)\3(?P<flags>[si]*)")
98         ])),
99         "rarity": re.compile(r"rarity (\d+)"),
100         "totalwaitms": re.compile(r"totalwaitms (\d+)"),
101         "tcpwrappedms": re.compile(r"tcpwrappedms (\d+)"),
102         "fallback": re.compile(r"fallback (\S+)"),
103         "ports": re.compile(r"ports (\S+)"),
104         "exclude": re.compile(r"Exclude T:(\S+)")
105     }
106
107     # parse the probes out from the file
108     for line in lines:
109         # add any ports to be excluded to the base probe class
110         if line.startswith("Exclude"):
111             search = regexes["exclude"].search(line)
112             if search:
113                 # parse the ports from the grouped output of
114                 # a search with the regex defined above.

```

```

115         for protocol, ports in
116             parse_ports(search.group(1)).items():
117                 directives.Probe.exclude[protocol].update(ports)
118     else:
119         print(line)
120         input()
121
122     # new probe directive
123     if line.startswith("Probe"):
124         # parse line into probe protocol, name and probestring
125         search = regexes["probe"].search(line)
126         if search:
127             try:
128                 proto, name, string = search.groups()
129             except ValueError:
130                 print(line)
131                 raise
132             probes[name][proto] = directives.Probe(proto, name,
133                 string)
134             # assign current_probe to the most recently added probe
135             current_probe = probes[name][proto]
136         else:
137             print(line)
138             input()
139
140     # new match directive
141     elif line.startswith("match") or line.startswith("softmatch"):
142         search = regexes["match"].search(line)
143         if search:
144             # the remainder of the string after the match
145             version_info = line[search.end()+1:]
146             # escape the curly braces so the regex engine doesn't
147             # consider them to be special characters
148             pattern = bytes(search.group("regex"), "utf-8")
149             # these replace the literal \n, \r and \t
150             # strings with their actual characters
151             # i.e. \n -> newline character
152             pattern = pattern.replace(b"\\n", b"\n")
153             pattern = pattern.replace(b"\\r", b"\r")
154             pattern = pattern.replace(b"\\t", b"\t")
155             matcher = directives.Match(
156                 search.group("service"),
157                 pattern,
158                 search.group("flags"),
159                 version_info
160             )
161             if search.group("type") == "match":
162                 current_probe.matches.add(matcher)
163             else:
164                 current_probe.softmatches.add(matcher)

```

```

163
164         else:
165             print(line)
166             input()
167
168     # new ports directive
169     elif line.startswith("ports"):
170         search = regexes["ports"].search(line)
171         if search:
172             for protocol, ports in
173                 parse_ports(search.group(1)).items():
174                 current_probe.ports[protocol].update(ports)
175         else:
176             print(line)
177             input()
178     # new totalwaitms directive
179     elif line.startswith("totalwaitms"):
180         search = regexes["totalwaitms"].search(line)
181         if search:
182             current_probe.totalwaitms = int(search.group(1))
183         else:
184             print(line)
185             input()
186
187     # new rarity directive
188     elif line.startswith("rarity"):
189         search = regexes["rarity"].search(line)
190         if search:
191             current_probe.rarity = int(search.group(1))
192         else:
193             print(line)
194             input()
195
196     # new fallback directive
197     elif line.startswith("fallback"):
198         search = regexes["fallback"].search(line)
199         if search:
200             current_probe.fallback = set(search.group(1).split(","))
201         else:
202             print(line)
203             input()
204
205     return probes
206
207 def version_detect_scan(
208     target: directives.Target,
209     probes: PROBE_CONTAINER
210 ) -> directives.Target:
211     for probe_dict in probes.values():
212         for proto in probe_dict:

```

```

212         target = probe_dict[proto].scan(target)
213     return target
214
215
216 def main() -> None:
217     print("reached here")
218     probes = parse_probes("./version_detection/nmap-service-probes")
219     open_ports: DefaultDict[str, Set[int]] = defaultdict(set)
220     open_filtered_ports: DefaultDict[str, Set[int]] = defaultdict(set)
221     open_filtered_ports["TCP"].add(22)
222     open_ports["TCP"].update([1, 2, 3, 4, 5, 6, 8, 65,
223                               20, 21, 23, 24, 25])
224
225     target = directives.Target(
226         "127.0.0.1",
227         open_ports,
228         open_filtered_ports
229     )
230     target.open_ports["TCP"].update([1, 2, 3])
231     print("BEFORE")
232     print(target)
233     scanned = version_detect_scan(target, probes)
234     print("AFTER")
235     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.

```

1  #!/usr/bin/env python
2  from collections import defaultdict
3  from contextlib import closing
4  from dataclasses import dataclass, field
5  from functools import reduce
6  from string import whitespace, printable
7  from typing import (
8      DefaultDict,
9      Dict,
10     Set,
11     List,
12     Pattern,
13     Match as RE_Match,
14     Tuple
15 )
16 from . import ip_utils
17 import operator
18 import re

```

```

19 import socket
20 import struct
21
22
23 class Match:
24     """
25     This is a class for both Matches and
26     Softmatches as they are actually the same
27     thing except that softmatches have less information.
28     """
29     options_to_flags = {
30         "i": re.IGNORECASE,
31         "s": re.DOTALL
32     }
33     letter_to_name = {
34         "p": "vendorproductname",
35         "v": "version",
36         "i": "info",
37         "h": "hostname",
38         "o": "operatingsystem",
39         "d": "devicetype"
40     }
41     cpe_part_map: Dict[str, str] = {
42         "a": "applications",
43         "h": "hardware platforms",
44         "o": "operating systems"
45     }
46     # look into match.expand when looking at the substring version info
47     # things.
48
49     def __init__(
50         self,
51         service: str,
52         pattern: bytes,
53         pattern_options: str,
54         version_info: str
55     ):
56         self.version_info: Dict[str, str] = dict()
57         self.cpes: Dict[str, Dict[str, str]] = dict()
58         self.service: str = service
59         # bitwise or is used to combine flags
60         # pattern options will never be anything but a
61         # combination of s and i.
62         # the default value of re.V1 is so that
63         # re uses the newer matching engine.
64         flags = reduce(
65             operator.ior,
66             [
67                 self.options_to_flags[opt]
68                 for opt in pattern_options

```

```

68         ],
69         0
70     )
71     try:
72         self.pattern: Pattern = re.compile(
73             pattern,
74             flags=flags
75         )
76     except Exception as e:
77         print("Regex failed to compile:")
78         print(e)
79         print(pattern)
80         input()
81
82     vinfo_regex = re.compile(r"([pvihod]|cpe:)([/|])(.+?)\2([a]*)")
83     cpe_regex = re.compile(
84         "?:?".join((
85             "(?P<part>[aho])",
86             "(?P<vendor>[~:]*)",
87             "(?P<product>[~:]*)",
88             "(?P<version>[~:]*)",
89             "(?P<update>[~:]*)",
90             "(?P<edition>[~:]*)",
91             "(?P<language>[~:]*)"
92         ))
93     )
94
95     for fieldname, _, val, opts in vinfo_regex.findall(version_info):
96         if fieldname == "cpe:":
97             search = cpe_regex.search(val)
98             if search:
99                 part = search.group("part")
100                 # this next bit is so that the bytes produced by the
101                 # regex
102                 # are turned to strings
103                 self.cpes[Match.cpe_part_map[part]] = {
104                     key: value
105                     for key, value
106                     in search.groupdict().items()
107                 }
108             else:
109                 self.version_info[
110                     Match.letter_to_name[fieldname]
111                 ] = val
112
113     def __repr__(self) -> str:
114         return "Match(" + ", ".join((
115             f"service={self.service}",
116             f"pattern={self.pattern}",
117             f"version_info={self.version_info}",

```

```

117         f"cpes={self.cpes}"
118     )) + ")"
119
120 def matches(self, string: bytes) -> bool:
121     def replace_groups(
122         string: str,
123         original_match: RE_Match
124     ) -> str:
125         """
126         This function takes in a string and the original
127         regex search performed on the data recieved and
128         replaces all of the $i, $SUBST, $I, $P occurrences
129         with the relavant formatted text that they produce.
130         """
131         def remove_unprintable(
132             group: int,
133             original_match: RE_Match
134         ) -> bytes:
135             """
136             Mirrors the P function from nmap which
137             is used to print only printable characters.
138             i.e. W\00\OR\OK\OG\OR\00\OU\OP -> WORKGROUP
139             """
140             return b"".join(
141                 i for i in original_match.group(group)
142                 if ord(i) in (
143                     set(printable)
144                     - set(whitespace)
145                     | {" "}
146                 )
147             )
148             # if i in the set of all printable characters,
149             # excluding those of which that are whitespace characters
150             # but including space.
151
152         def substitute(
153             group: int,
154             before: bytes,
155             after: bytes,
156             original_match: RE_Match
157         ) -> bytes:
158             """
159             Mirrors the SUBST function from nmap which is used to
160             format some information found by the regex.
161             by substituting all instances of 'before' with 'after'.
162             """
163             return original_match.group(group).replace(before, after)
164
165         def unpack_uint(
166             group: int,

```

```

167         endianness: str,
168         original_match: RE_Match
169     ) -> bytes:
170         """
171         Mirrors the I function from nmap which is used to
172         unpack an unsigned int from some bytes.
173         """
174         return bytes(struct.unpack(
175             endianness + "I",
176             original_match.group(group)
177         ))
178
179     text = bytes(string, "utf-8")
180     # fill in the version information from the regex match
181     # find all the dollar groups:
182     dollar_regex = re.compile(r"\$(\d)")
183     # find all the $i's in string
184     numbers = set(int(i) for i in dollar_regex.findall(string))
185     # for each $i found i
186     for group in numbers:
187         text = text.replace(
188             bytes(f"${group}", "utf-8"),
189             original_match.group(group)
190         )
191     # having replaced all of the groups we can now
192     # start doing the SUBST, P and I commands.
193     subst_regex = re.compile(rb"\$SUBST\((\d),(.+),(.+)\)")
194     # iterate over all of the matches found by the SUBST regex
195     for match in subst_regex.finditer(text):
196         num, before, after = match.groups()
197         # replace the full match (group 0)
198         # with the output of substitute
199         # with the specific arguments
200         text.replace(
201             match.group(0),
202             substitute(int(num), before, after, original_match)
203         )
204
205     p_regex = re.compile(rb"\$P\((\d)\)")
206     for match in p_regex.finditer(text):
207         num = match.group(1)
208         # replace the full match (group 0)
209         # with the output of remove_unprintable
210         # with the specific arguments
211         text.replace(
212             match.group(0),
213             remove_unprintable(int(num), original_match)
214         )
215
216     i_regex = re.compile(br"\$I\((\d),\"(\S)\\\"")

```



```

217         for match in i_regex.finditer(text):
218             num, endianness = match.groups()
219             # this means replace group 0 -> the whole match
220             # with the output of the unpack_uint
221             # with the specified arguments
222             text.replace(
223                 match.group(0),
224                 unpack_uint(
225                     int(num.decode()),
226                     endianness.decode(),
227                     original_match
228                 )
229             )
230
231         return text.decode()
232
233     search = self.pattern.search(string)
234     if search:
235         # the fields to replace are all the CPE groups,
236         # all of the version info fields.
237         self.version_info = {
238             key: replace_groups(value, search)
239             for key, value in self.version_info.items()
240         }
241         self.cpes = {
242             outer_key: {
243                 inner_key: replace_groups(value, search)
244                 for inner_key, value in outer_dict.items()
245             }
246             for outer_key, outer_dict in self.cpes.items()
247         }
248
249         return True
250     else:
251         return False
252
253
254 @dataclass
255 class Target:
256     """
257     This class holds data about targets to
258     scan. the dataclass decorator is simply
259     a way of python automatically writing some
260     of the basic methods a class for storing data
261     has, such as __repr__ for printing information
262     in the object etc.
263     """
264     address: str
265     open_ports: DefaultDict[str, Set[int]]
266     open_filtered_ports: DefaultDict[str, Set[int]]

```

```

267     services: Dict[int, Match] = field(default_factory=dict)
268
269     def __repr__(self) -> str:
270         def collapse(port_dict: DefaultDict) -> str:
271             """
272             Collapse a list of port numbers so that
273             only the unique ones and the start and end
274             of a sequence are displayed.
275             1,2,3,4,5,7,9,11,13,14,15,16,17 -> 1-5,7,9,11,13-17
276             """
277             store_results = list()
278             for key in port_dict:
279                 # items is a sorted list of a set of ports.
280                 items: List[int] = sorted(port_dict[key])
281                 key_result = f'"{key}":' + "{"
282                 # if its an empty list return now to avoid errors
283                 if len(items) != 0:
284                     new_sequence = False
285                     # enumerate up until the one before
286                     # the last to prevent index errors.
287                     for index, item in enumerate(items[:-1]):
288                         # if its the first one add it on
289                         if index == 0:
290                             key_result += f"{item}"
291                             # if its a sequence start one else put a comma
292                             if items[index+1] == item+1:
293                                 key_result += "-"
294                             else:
295                                 key_result += ","
296                             # if the sequence breaks then put a comma
297                             elif item+1 != items[index+1]:
298                                 key_result += f"{item},"
299                                 new_sequence = True
300                             # if its a new sequence the put the '-'s in
301                             elif item+1 == items[index+1] and new_sequence:
302                                 key_result += f"{item}-"
303                                 new_sequence = False
304                             # because we only iterate to the one before
305                             # the last element, add the last element on to the end.
306                             key_result += f"{items[-1]}" + "}"
307                             store_results.append(key_result)
308             # format the final result
309             result = "{" + ", ".join(store_results) + "}"
310             return result
311
312         open_ports = collapse(self.open_ports)
313         open_filtered_ports = collapse(self.open_filtered_ports)
314         return ", ".join((
315             f"Target(address=[{self.address}]",
316             f"open_ports=[{open_ports}]",

```

```

317         f"open_filtered_ports=[{open_filtered_ports}]",
318         f"services={self.services}")
319     ))
320
321
322 class Probe:
323     """
324     This class represents the Probe directive of the nmap-service-probes
325     file.
326     It holds information such as the protocol to use, the string to send,
327     the ports to scan, the time to wait for a null TCP to return a
328     banner,
329     the rarity of the probe (how often it will return a response) and the
330     probes to try if this one fails.
331     """
332     # a default dict is one which takes in a
333     # "default factory" which is called when
334     # a new key is introduced to the dict
335     # in this case the default factory is
336     # the set function meaning that when I
337     # do exclude[protocol].update(ports)
338     # but exclude[protocol] has not yet been defined
339     # it will be defined as an empty set
340     # allowing me to update it with ports.
341     exclude: DefaultDict[str, Set[int]] = defaultdict(set)
342     proto_to_socket_type: Dict[str, int] = {
343         "TCP": socket.SOCK_STREAM,
344         "UDP": socket.SOCK_DGRAM
345     }
346
347     def __init__(self, protocol: str, probename: str, probe: str):
348         """
349         This is the initial function that is called by the
350         constructor of the Probe class, it is used to define
351         the variables that are specific to each instance of
352         the class.
353         """
354         if protocol in {"TCP", "UDP"}:
355             self.protocol = protocol
356         else:
357             raise ValueError(
358                 f"Probe object must have protocol TCP or UDP not {protocol}.")
359         self.name: str = probename
360         self.string: str = probe
361         self.payload: bytes = bytes(probe, "utf-8")
362         self.matches: Set[Match] = set()
363         self.softmatches: Set[Match] = set()
364         self.ports: DefaultDict[str, Set[int]] = defaultdict(set)

```

```

364         self.totalwaitms: int = 6000
365         self.tcpwrappedms: int = 3000
366         self.rarity: int = -1
367         self.fallback: Set[str] = set()
368
369     def __repr__(self) -> str:
370         """
371         This is the function that is called when something
372         tries to print an instance of this class.
373         It is used to reveal information internal
374         to the class.
375         """
376         return ", ".join([
377             f"Probe({self.protocol}",
378             f"{self.name}",
379             f"\n{self.string}\n",
380             f"{len(self.matches)} matches",
381             f"{len(self.softmatches)} softmatches",
382             f"ports: {self.ports}",
383             f"rarity: {self.rarity}",
384             f"fallbacks: {self.fallback})"
385         ])
386
387     def scan(self, target: Target) -> Target:
388         """
389         scan takes in an object of class Target to
390         probe and attempts to detect the version of
391         any services running on the machine.
392         """
393         # this constructs the set of all ports,
394         # that are either open or open_filtered,
395         # and are in the set of ports to scan for
396         # this particular probe, this means that,
397         # we are only connecting to ports that we
398         # know are not closed and are not to be excluded.
399
400         ports_to_scan: Set[int] = (
401             (
402                 target.open_filtered_ports[self.protocol]
403                 | target.open_ports[self.protocol]
404             )
405             - Probe.exclude[self.protocol] - Probe.exclude["ANY"]
406         )
407         # if the probe defines a set of ports to scan
408         # then don't scan any that aren't defined for it
409         if self.ports[self.protocol] != set():
410             ports_to_scan -= self.ports[self.protocol]
411         for port in ports_to_scan:
412             # open a self closing IPV4 socket
413             # for the correct protocol for this probe.
414             with closing(

```

```

414         socket.socket(
415             socket.AF_INET,
416             self.proto_to_socket_type[self.protocol]
417         )
418     ) as sock:
419         # setup the connection to the target
420         try:
421             sock.connect((target.address, port))
422             # if the connection fails then continue scanning
423             # the next ports, this shouldn't really happen.
424         except ConnectionError:
425             continue
426         # send the payload to the target
427         sock.send(self.payload)
428         # wait for the target to send a response
429         time_taken = ip_utils.wait_for_socket(
430             sock,
431             self.totalwaitms/1000
432         )
433         # if the response didn't time out
434         if time_taken != -1:
435             # if the port was in open_filtered move it to open
436             if port in target.open_filtered_ports[self.protocol]:
437                 target.open_filtered_ports[
438                     self.protocol
439                 ].remove(port)
440                 target.open_ports[self.protocol].add(port)
441
442             # recieve the data and decode it to a string
443             data_recieved = sock.recv(4096)
444             # print("Recieved", data_recieved)
445             service = ""
446             # try and softmatch the service first
447             for softmatch in self.softmatches:
448                 if softmatch.matches(data_recieved):
449                     service = softmatch.service
450                     target.services[port] = softmatch
451                     break
452             # try and get a full match for the service
453             for match in self.matches:
454                 if service in match.service.lower():
455                     if match.matches(data_recieved):
456                         target.services[port] = match
457                         break
458         return target
459
460
461 PROBE_CONTAINER = DefaultDict[str, Dict[str, Probe]]
462
463

```

```

464 def parse_ports(portstring: str) -> DefaultDict[str, Set[int]]:
465     """
466     This function takes in a port directive
467     and returns a set of the ports specified.
468     A set is used because it is O(1) for contains
469     operations as opposed for O(N) for lists.
470     """
471     # matches both the num-num port range format
472     # and the plain num port specification
473     # num-num form must come first otherwise it breaks.
474     proto_regex = re.compile(r"([ TU]?):?([0-9,-]+)")
475     # THE SPACE IS IMPORTANT!!!
476     # it allows ports specified before TCP/UDP ports
477     # to be specified globally as in for all protocols.
478
479     pair_regex = re.compile(r"(\d+)-(\d+)")
480     single_regex = re.compile(r"(\d+)")
481     ports: DefaultDict[str, Set[int]] = defaultdict(set)
482     # searches contains the result of trying the pair_regex
483     # search against all of the command seperated
484     # port strings
485
486     for protocol, portstring in proto_regex.findall(portstring):
487         pairs = pair_regex.findall(portstring)
488         # for each pair of numbers in the pairs list
489         # seperate each number and cast them to int
490         # then generate the range of numbers from x[0]
491         # to x[1]+1 then cast this range to a list
492         # and "reduce" the list of lists by joining them
493         # with operator.ior (inclusive or) and then let
494         # ports be the set of all the ports in that list.
495         proto_map = {
496             "": "ANY",
497             " ": "ANY",
498             "U": "UDP",
499             "T": "TCP"
500         }
501         if pairs:
502             def pair_to_ports(pair: Tuple[str, str]) -> Set[int]:
503                 """
504                 a function to go from a port pair i.e. (80-85)
505                 to the set of specified ports: {80,81,82,83,84,85}
506                 """
507                 start, end = pair
508                 return set(range(
509                     int(start),
510                     int(end)+1
511                 ))
512             # ports contains the set of all ANY/TCP/UDP specified ports
513             ports[proto_map[protocol]] = set(reduce(

```

```

514         operator.ior,
515         map(pair_to_ports, pairs)
516     ))
517
518     singles = single_regex.findall(portstring)
519     # for each of the ports that are specified on their own
520     # cast them to int and update the set of all ports with
521     # that list.
522     ports[proto_map[protocol]].update(map(int, singles))
523
524     return ports
525
526
527 def parse_probes(probe_file: str) -> PROBE_CONTAINER:
528     """
529     Extracts all of the probe directives from the
530     file pointed to by probe_file.
531     """
532     # lines contains each line of the file which doesn't
533     # start with a # and is not empty.
534     lines = [
535         line
536         for line in open(probe_file).read().splitlines()
537         if line and not line.startswith("#")
538     ]
539
540     # list holding each of the probe directives.
541     probes: PROBE_CONTAINER = defaultdict(dict)
542
543     regexes: Dict[str, Pattern] = {
544         "probe": re.compile(r"Probe (TCP|UDP) (\S+) q\|(.*)\|"),
545         "match": re.compile(" ".join([
546             r"(?P<type>softmatch|match)",
547             r"(?P<service>\S+)",
548             r"m([\@/%=|])(?P<regex>.+?)\3(?P<flags>[si]*)"
549         ])),
550         "rarity": re.compile(r"rarity (\d+)"),
551         "totalwaitms": re.compile(r"totalwaitms (\d+)"),
552         "tcpwrappedms": re.compile(r"tcpwrappedms (\d+)"),
553         "fallback": re.compile(r"fallback (\S+)"),
554         "ports": re.compile(r"ports (\S+)"),
555         "exclude": re.compile(r"Exclude T: (\S+)")
556     }
557
558     # parse the probes out from the file
559     for line in lines:
560         # add any ports to be excluded to the base probe class
561         if line.startswith("Exclude"):
562             search = regexes["exclude"].search(line)
563             if search:

```

```

564         # parse the ports from the grouped output of
565         # a search with the regex defined above.
566         for protocol, ports in
            parse_ports(search.group(1)).items():
567             Probe.exclude[protocol].update(ports)
568     else:
569         print(line)
570         input()
571
572     # new probe directive
573     if line.startswith("Probe"):
574         # parse line into probe protocol, name and probestring
575         search = regexes["probe"].search(line)
576         if search:
577             try:
578                 proto, name, string = search.groups()
579             except ValueError:
580                 print(line)
581                 raise
582             probes[name][proto] = Probe(proto, name, string)
583             # assign current_probe to the most recently added probe
584             current_probe = probes[name][proto]
585         else:
586             print(line)
587             input()
588
589     # new match directive
590     elif line.startswith("match") or line.startswith("softmatch"):
591         search = regexes["match"].search(line)
592         if search:
593             # the remainder of the string after the match
594             version_info = line[search.end()+1:]
595             # escape the curly braces so the regex engine doesn't
596             # consider them to be special characters
597             pattern = bytes(search.group("regex"), "utf-8")
598             # these replace the literal \n, \r and \t
599             # strings with their actual characters
600             # i.e. \n -> newline character
601             pattern = pattern.replace(b"\\n", b"\n")
602             pattern = pattern.replace(b"\\r", b"\r")
603             pattern = pattern.replace(b"\\t", b"\t")
604             matcher = Match(
605                 search.group("service"),
606                 pattern,
607                 search.group("flags"),
608                 version_info
609             )
610             if search.group("type") == "match":
611                 current_probe.matches.add(matcher)
612         else:

```



```

613         current_probe.softmatches.add(matcher)
614
615     else:
616         print(line)
617         input()
618
619     # new ports directive
620     elif line.startswith("ports"):
621         search = regexes["ports"].search(line)
622         if search:
623             for protocol, ports in
624                 parse_ports(search.group(1)).items():
625                 current_probe.ports[protocol].update(ports)
626         else:
627             print(line)
628             input()
629     # new totalwaitms directive
630     elif line.startswith("totalwaitms"):
631         search = regexes["totalwaitms"].search(line)
632         if search:
633             current_probe.totalwaitms = int(search.group(1))
634         else:
635             print(line)
636             input()
637
638     # new rarity directive
639     elif line.startswith("rarity"):
640         search = regexes["rarity"].search(line)
641         if search:
642             current_probe.rarity = int(search.group(1))
643         else:
644             print(line)
645             input()
646
647     # new fallback directive
648     elif line.startswith("fallback"):
649         search = regexes["fallback"].search(line)
650         if search:
651             current_probe.fallback = set(search.group(1).split(","))
652         else:
653             print(line)
654             input()
655
656     return probes

```

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

```

1 import struct
2 import socket
3 from typing import Dict
4

```

```

5
6 class ip:
7     """
8     A class for parsing, storing and displaying
9     data from an IP header.
10    """
11    def __init__(self, header: bytes):
12        # first unpack the IP header
13        (
14            ip_hp_ip_v,
15            ip_dscp_ip_ecn,
16            ip_len,
17            ip_id,
18            ip_flg_ip_off,
19            ip_ttl,
20            ip_p,
21            ip_sum,
22            ip_src,
23            ip_dst
24        ) = struct.unpack('!BBHHHBBHII', header)
25        # now deal with the sub-byte sized components
26        hl_v = f"{ip_hp_ip_v:08b}"
27        ip_v = int(hl_v[:4], 2)
28        ip_hl = int(hl_v[4:], 2)
29        # splits hl_v in ip_v and ip_hl which store the IP version
30        # number and
31        # header length respectively
32        dscp_ecn = f"{ip_dscp_ip_ecn:08b}"
33        ip_dscp = int(dscp_ecn[:6], 2)
34        ip_ecn = int(dscp_ecn[6:], 2)
35        # splits dscp_ecn into ip_dscp and ip_ecn
36        # which are two of the components
37        # in an IP header
38        flgs_off = f"{ip_flg_ip_off:016b}"
39        ip_flg = int(flgs_off[:3], 2)
40        ip_off = int(flgs_off[3:], 2)
41        # splits flgs_off into ip_flg and ip_off which represent the ip
42        # header
43        # flags and the data offset
44        src_addr = socket.inet_ntoa(struct.pack('!I', ip_src))
45        dst_addr = socket.inet_ntoa(struct.pack('!I', ip_dst))
46        self.version: int = ip_v
47        self.header_length: int = ip_hl
48        self.dscp: int = ip_dscp
49        self.ecn: int = ip_ecn
50        self.len: int = ip_len
51        self.id: int = ip_id
52        self.flags: int = ip_flg
53        self.data_offset: int = ip_off
54        self.time_to_live: int = ip_ttl

```

```

53         self.protocol: int = ip_p
54         self.checksum: int = ip_sum
55         self.source: str = src_addr
56         self.destination: str = dst_addr
57
58     def __repr__(self) -> str:
59         return "\n\t".join((
60             "IP header:",
61             f"Version: [{self.version}]",
62             f"Internet Header Length: [{self.header_length}]",
63             f"Differentiated Services Point Code: [{self.dscp}]",
64             f"Explicit Congestion Notification: [{self.ecn}]",
65             f"Total Length: [{self.len}]",
66             f"Identification: [{self.id:04x}]",
67             f"Flags: [{self.flags:03b}]",
68             f"Fragment Offset: [{self.data_offset}]",
69             f"Time To Live: [{self.time_to_live}]",
70             f"Protocol: [{self.protocol}]",
71             f"Header Checksum: [{self.checksum:04x}]",
72             f"Source Address: [{self.source}]",
73             f"Destination Address: [{self.destination}]"
74         ))
75
76
77     class icmp:
78         """
79         A class for parsing, storing and displaying
80         data from an IP header.
81         """
82         # relates the type and code to the message
83         messages: Dict[int, Dict[int, str]] = {
84             0: {
85                 0: "Echo reply."
86             },
87             3: {
88                 0: "Destination network unreachable.",
89                 1: "Destination host unreachable",
90                 2: "Destination protocol unreachable",
91                 3: "Destination port unreachable",
92                 4: "Fragmentation required, and DF flag set.",
93                 5: "Source route failed.",
94                 6: "Destination network unknown.",
95                 7: "Destination host unknown.",
96                 8: "Source host isolated.",
97                 9: "Network administratively prohibited.",
98                 10: "Host administratively prohibited.",
99                 11: "Network unreachable for ToS.",
100                 12: "Host unreachable for ToS.",
101                 13: "Communication administratively prohibited.",
102                 14: "Host precedence violation.",

```

```

103         15: "Precedence cutoff in effect."
104     },
105     4: {
106         0: "Source quench."
107     },
108     5: {
109         0: "Redirect datagram for the network",
110         1: "Redirect datagram for the host.",
111         2: "Redirect datagram for the ToS & network.",
112         3: "Redirect datagram for the ToS & host."
113     },
114     8: {
115         0: "Echo request."
116     },
117     9: {
118         0: "Router advertisement"
119     },
120     10: {
121         0: "Router discovery/selection/solicitation."
122     },
123     11: {
124         0: "TTL expired in transit",
125         1: "Fragment reassembly time exceeded."
126     },
127     12: {
128         0: "Bad IP header: pointer indicates error.",
129         1: "Bad IP header: missing a required option.",
130         2: "Bad IP header: Bad length."
131     },
132     13: {
133         0: "Timestamp"
134     },
135     14: {
136         0: "Timestamp reply"
137     },
138     15: {
139         0: "Information request."
140     },
141     16: {
142         0: "Information reply."
143     },
144     17: {
145         0: "Address mask request."
146     },
147     18: {
148         0: "Address mask reply."
149     }
150 }
151
152 def __init__(self, header: bytes):

```

```

153         (
154             ICMP_type,
155             code,
156             csum,
157             remainder
158         ) = struct.unpack('!bbHI', header)
159
160         self.type: int = ICMP_type
161         self.code: int = code
162         self.checksum: int = csum
163
164         self.message: str
165         try:
166             self.message = icmp.messages[self.type][self.code]
167         except KeyError:
168             # if we can't assign a message then just set a description
169             # as to what caused the failure.
170             self.message = f"Failed to assign message:
171                             ({self.type/self.code})"
172
173         self.id: int
174         self.sequence: int
175         if self.type in {0, 8}:
176             self.id = socket.htons(remainder >> 16)
177             self.sequence = socket.htons(remainder & 0xFFFF)
178         else:
179             self.id = -1
180             self.sequence = -1
181
182     def __repr__(self) -> str:
183         return "\n\t".join((
184             "ICMP header:",
185             f"Message: [{self.message}]",
186             f"Type: [{self.type}]",
187             f"Code: [{self.code}]",
188             f"Checksum: [{self.checksum:04x}]",
189             f"ID: [{self.id}]",
190             f"Sequence: [{self.sequence}]"
191         ))
192
193 class tcp:
194     def __init__(self, header: bytes):
195         (
196             src_prt,
197             dst_prt,
198             seq,
199             ack,
200             data_offset,
201             flags,

```

```

202         window_size,
203         checksum,
204         urg
205     ) = struct.unpack("!HHIIBBHH", header)
206
207     self.source: int = src_prt
208     self.destination: int = dst_prt
209     self.seq: int = seq
210     self.ack: int = ack
211     self.data_offset: int = data_offset >> 4
212     self.flags: int = flags + ((data_offset & 0x01) << 8)
213     self.window_size: int = window_size
214     self.checksum: int = checksum
215     self.urg: int = urg
216
217     def __repr__(self) -> str:
218         return "\n\t".join((
219             "TCP header:",
220             f"Source port: [{self.source}]",
221             f"Destination port: [{self.destination}]",
222             f"Sequence number: [{self.seq}]",
223             f"Acknowledgement number: [{self.ack}]",
224             f>Data offset: [{self.data_offset}]",
225             f"Flags: [{self.flags:08b}]",
226             f"Window size: [{self.window_size}]",
227             f"Checksum: [{self.checksum:04x}]",
228             f"Urgent: [{self.urg}]"
229         ))
230
231
232     class udp:
233         def __init__(self, header: bytes):
234             # parse udp header
235             (
236                 src_port,
237                 dest_port,
238                 length,
239                 checksum
240             ) = struct.unpack("!HHHH", header)
241
242             self.src: int = src_port
243             self.dest: int = dest_port
244             self.length: int = length
245             self.checksum: int = checksum
246
247             def __repr__(self) -> str:
248                 return "\n\t".join((
249                     "UDP header:",
250                     f"Source port: {self.src}",
251                     f"Destination port: {self.dest}",

```

```

252         f"Length: {self.length}",
253         f"Checksum: {self.checksum:04x}"
254     ))

```

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

```

1  import array
2  import socket
3  import struct
4  import select
5  import time
6
7  from contextlib import closing
8  from functools import singledispatch
9  from itertools import islice, cycle
10 from sys import stderr
11 from typing import Set, Union
12
13
14 def eprint(*args: str, **kwargs: str) -> None:
15     """
16     Mirrors print exactly but prints to stderr
17     instead of stdout.
18     """
19     print(*args, file=stderr, **kwargs) # type: ignore
20
21
22 def long_to_dot(long: int) -> str:
23     """
24     Take in an IP address in packed 32 bit int form
25     and return that address in dot notation.
26     i.e. long_to_dot(0x7F000001) = 127.0.0.1
27     """
28     # these are long form values for 0.0.0.0
29     # and 255.255.255.255
30     if not 0 <= long <= 0xFFFFFFFF:
31         raise ValueError(f"Invalid long form IP address: [{long:08x}]")
32     else:
33         # shift the long form IP along 0, 8, 16, 24 bits
34         # take only the first 8 bits of the newly shifted number
35         # cast them to a string and join them with '.'s
36         return ".".join(
37             str(
38                 (long >> (8*(3-i))) & 0xFF
39             )
40             for i in range(4)
41         )
42

```

```

43
44 def dot_to_long(ip: str) -> int:
45     """
46     Take an ip address in dot notation and return the packed 32 bit int
47     version
48     i.e. dot_to_long("127.0.0.1") = 0x7F000001
49     """
50     # dot form ips: a.b.c.d must have each
51     # part (a,b,c,d) between 0 and 255,
52     # otherwise they are invalid
53
54     parts = [int(i) for i in ip.split(".")]
55
56     if not all(
57         0 <= i <= 255
58         for i in parts
59     ):
60         raise ValueError(f"Invalid dot form IP address: [{ip}]")
61
62     if len(parts) != 4:
63         raise ValueError(f"Invalid dot form IP address: [{ip}]")
64
65     else:
66         # for each part of the dotted IP address
67         # bit shift left each part by eight times
68         # three minus it's position. This puts the bits
69         # from each part in the right place in the final sum
70         # a.b.c.d -> a<<3*8 + b<<2*8 + c<<1*8 + d<<0*8
71         return sum(
72             part << ((3-i)*8)
73             for i, part in enumerate(parts)
74         )
75
76
77 @singledispatch
78 def is_valid_ip(ip: Union[str, int]) -> bool:
79     """
80     checks whether a given IP address is valid.
81     """
82
83
84 @is_valid_ip.register
85 def _(ip: int):
86     # this is the int overload variant of
87     # the is_valid_ip function.
88     try:
89         # try to turn the long form ip address
90         # to a dot form one, if it fails,
91         # then return False, else return True

```



```

92         long_to_dot(ip)
93         return True
94     except ValueError:
95         return False
96
97
98     # the type ignore comment is required to stop
99     # mypy exploding over the fact I have defined '_' twice.
100 @is_valid_ip.register # type: ignore
101 def _(ip: str):
102     # this is the string overload variant
103     # of the is_valid_ip function.
104     try:
105         # try to turn the dot form ip address
106         # to a long form one, if it fails,
107         # then return False, else return True
108         dot_to_long(ip)
109         return True
110     except ValueError:
111         return False
112
113
114 def is_valid_port_number(port_num: int) -> bool:
115     """
116     Checks whether the given port number is valid i.e. between 0 and
117     65536.
118     """
119     # port numbers must be between 0 and 65535(2^16 - 1)
120     if 0 <= port_num < 2**16:
121         return True
122     else:
123         return False
124
125 def ip_range(ip: str, network_bits: int) -> Set[str]:
126     """
127     Takes a Classless Inter Domain Routing(CIDR) address subnet
128     specification and returns the list of addresses specified
129     by the IP/network bits format.
130     If the number of network bits is not between 0 and 32 it raises an
131     error.
132     If the IP address is invalid according to is_valid_ip it raises an
133     error.
134     """
135     if not 0 <= network_bits <= 32:
136         raise ValueError(f"Invalid number of network bits:
137             [{network_bits}]")
138
139     if not is_valid_ip(ip):

```

```

138         raise ValueError(f"Invalid IP address: [{ip}]")
139     # get the ip as long form which is useful
140     # later on for using bitwise operators
141     # to isolate only the constant(network) bits
142     ip_long = dot_to_long(ip)
143
144     # generate the bit mask which specifies
145     # which bits to keep and which to discard
146     mask = int(
147         f"{1'*network_bits:0<32s}",
148         base=2
149     )
150     lower_bound = ip_long & mask
151     upper_bound = ip_long | (mask ^ 0xFFFFFFFF)
152
153     # turn all the long form IP addresses between
154     # the lower and upper bound into dot form
155     if network_bits <= 30:
156         return set(
157             long_to_dot(long_ip)
158             for long_ip in
159                 range(lower_bound+1, upper_bound)
160         )
161     else:
162         return set(
163             long_to_dot(long_ip)
164             for long_ip in
165                 range(lower_bound, upper_bound+1)
166         )
167
168
169
170 def get_local_ip() -> str:
171     """
172     Connects to the google.com with UDP and gets
173     the IP address used to connect(the local address).
174     """
175     with closing(
176         socket.socket(
177             socket.AF_INET,
178             socket.SOCK_DGRAM
179         )
180     ) as s:
181         try:
182             s.connect(("google.com", 80))
183             ip, _ = s.getsockname()
184         except:
185             ip = "127.0.0.1"
186     return ip
187

```

```

188
189 def get_free_port() -> int:
190     """
191     Attempts to bind to port 0 which assigns a free port number to the
192     socket,
193     the socket is then closed and the port number assigned is returned.
194     """
195     with closing(
196         socket.socket(
197             socket.AF_INET,
198             socket.SOCK_STREAM
199         )
200     ) as s:
201         s.bind(('', 0))
202         _, port = s.getsockname()
203     return port
204
205
206 def ip_checksum(packet: bytes) -> int:
207     """
208     ip_checksum function takes in a packet
209     and returns the checksum.
210     """
211     if len(packet) % 2 == 1:
212         # if the length of the packet is odd, add a NULL byte
213         # to the end as padding
214         packet += b"\0"
215
216     total = 0
217     for first, second in (
218         packet[i:i+2]
219         for i in range(0, len(packet), 2)
220     ):
221         total += (first << 8) + second
222
223     # calculate the number of times a
224     # carry bit was added and add it back on
225     carried = (total - (total & 0xFFFF)) >> 16
226     total &= 0xFFFF
227     total += carried
228
229     if total > 0xFFFF:
230         # adding the carries generated a carry
231         total &= 0xFFFF
232         total += 1
233
234     # invert the checksum and take the last 16 bits.
235     return (~total & 0xFFFF)
236

```

```

237
238 def make_icmp_packet(ID: int) -> bytes:
239     """
240     Takes an argument of the process ID of the calling process.
241     Returns an ICMP ECHO REQUEST packet created with this ID
242     """
243
244     ICMP_ECHO_REQUEST = 8
245     # pack the information for the dummy header needed
246     # for the IP checksum
247     dummy_header = struct.pack(
248         "bbHHh",
249         ICMP_ECHO_REQUEST,
250         0,
251         0,
252         ID,
253         1
254     )
255     # pack the current time into a double
256     time_bytes = struct.pack("d", time.time())
257     # define the bytes to repeat in the data section of the packet
258     # this makes the packets easily identifiable in packet captures.
259     bytes_to_repeat_in_data = map(ord, " y33t ")
260     # calculate the number of bytes left for data
261     data_bytes = (192 - struct.calcsize("d"))
262     # first pack the current time into the start of the data section
263     # the pack the identifiable data into the rest
264     data = (
265         time_bytes +
266         bytes(islice(cycle(bytes_to_repeat_in_data), data_bytes))
267     )
268     # get the IP checksum for the dummy header and data
269     # and switch the bytes into the order expected by the network
270     checksum = socket.htons(ip_checksum(dummy_header + data))
271     # pack the header with the correct checksum and information
272     header = struct.pack(
273         "bbHHh",
274         ICMP_ECHO_REQUEST,
275         0,
276         checksum,
277         ID,
278         1
279     )
280     # concatenate the header bytes and the data bytes
281     return header + data
282
283
284 def make_tcp_packet(
285     src: int,
286     dst: int,

```

```

287         from_address: str,
288         to_address: str,
289         flags: int) -> bytes:
290     """
291     Takes in the source and destination port/ip address
292     returns a tcp packet.
293     flags:
294     2 => SYN
295     18 => SYN:ACK
296     4 => RST
297     """
298     # validate that the information passed in is valid
299     if flags not in {2, 18, 4}:
300         raise ValueError(
301             f"Flags must be one of 2:SYN, 18:SYN:ACK, 4:RST. not:
302                 [{flags}]"
303         )
304     if not is_valid_ip(from_address):
305         raise ValueError(
306             f"Invalid source IP address: [{from_address}]"
307         )
308     if not is_valid_ip(to_address):
309         raise ValueError(
310             f"Invalid destination IP address: [{to_address}]"
311         )
312     if not is_valid_port_number(src):
313         raise ValueError(
314             f"Invalid source port: [{src}]"
315         )
316     if not is_valid_port_number(dst):
317         raise ValueError(
318             f"Invalid destination port: [{dst}]"
319         )
320     # turn the ip addresses into long form
321     src_addr = dot_to_long(from_address)
322     dst_addr = dot_to_long(to_address)
323
324     seq = ack = urg = 0
325     data_offset = 6 << 4
326     window_size = 1024
327     max_segment_size = (2, 4, 1460)
328     # pack the dummy header needed for the checksum calculation
329     dummy_header = struct.pack(
330         "!HHIIBBHHBBH",
331         src,
332         dst,
333         seq,
334         ack,
335         data_offset,
336         flags,

```

```

336         window_size,
337         0,
338         urg,
339         *max_segment_size
340     )
341     # pack the psuedo header that is also needed for the checksum
342     # just because TCP and why not
343     psuedo_header = struct.pack(
344         "!IIBBH",
345         src_addr,
346         dst_addr,
347         0,
348         6,
349         len(dummy_header)
350     )
351
352     checksum = ip_checksum(psuedo_header + dummy_header)
353     # pack the final TCP packet with the relevant data and checksum
354     return struct.pack(
355         "!HHIIBBHHHBBH",
356         src,
357         dst,
358         seq,
359         ack,
360         data_offset,
361         flags,
362         window_size,
363         checksum,
364         urg,
365         *max_segment_size
366     )
367
368
369     def make_udp_packet(
370         src: int,
371         dst: int
372     ) -> bytes:
373         """
374         Takes in: source IP address and port, destination IP address and
375             port.
376         Returns: a UDP packet with those properties.
377         the IP addresses are needed for calculating the checksum.
378         """
379         # validate data passed in
380         if not is_valid_port_number(src):
381             raise ValueError(
382                 f"Invalid source port: [{src}]"
383             )
384         if not is_valid_port_number(dst):
385             raise ValueError(

```

```

385         f"Invalid destination port: [{dst}]"
386     )
387     data = b"Most services don't respond to an empty data field"
388     # pack the data
389     # and return the packed bytes
390     # UDP checksum is optional over IPv4
391     return struct.pack(
392         "!HHHH",
393         src,
394         dst,
395         8+len(data),
396         0
397     ) + data
398
399
400 def wait_for_socket(sock: socket.socket, wait_time: float) -> float:
401     """
402     Wait for wait_time seconds or until the socket is readable.
403     If the socket is readable return a tuple of the socket and the time
404     taken
405     otherwise return None.
406     """
407     start = time.time()
408     is_socket_readable = select.select([sock], [], [], wait_time)
409     taken = time.time() - start
410     if is_socket_readable[0] == []:
411         return float(-1)
412     else:
413         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.

```

1  from modules import headers
2  from modules import ip_utils
3  import socket
4  import struct
5  import time
6  from collections import defaultdict
7  from contextlib import closing
8  from typing import Tuple, Set, DefaultDict
9
10
11  PORTS = DefaultDict[str, Set[int]]
12
13
14  def ping(
15      ID: int,
16      timeout: float

```

```

17 ) -> Set[Tuple[str, float, headers.ip]]:
18     """
19     Takes in a process id and a timeout and returns
20     a list of addresses which sent ICMP ECHO REPLY
21     packets with the packed id matching ID in the time given by timeout.
22     """
23     ping_sock = socket.socket(
24         socket.AF_INET,
25         socket.SOCK_RAW,
26         socket.IPPROTO_ICMP)
27     # opens a raw socket for sending ICMP protocol packets
28     time_remaining = timeout
29     addresses = set()
30     recieved_from = set()
31     while True:
32         time_waiting = ip_utils.wait_for_socket(ping_sock,
33             time_remaining)
34         # time_waiting stores the time the socket took to become readable
35         # or returns minus one if it ran out of time
36
37         if time_waiting == -1:
38             break
39         time_recieved = time.time()
40         # store the time the packet was recieved
41         recPacket, addr = ping_sock.recvfrom(1024)
42         # recieve the packet
43         ip = headers.ip(recPacket[:20])
44         # unpack the IP header into its respective components
45         icmp = headers.icmp(recPacket[20:28])
46         # unpack the time from the packet.
47         time_sent = struct.unpack(
48             "d",
49             recPacket[28:28 + struct.calcsize("d")])
50         # unpack the value for when the packet was sent
51         time_taken: float = time_recieved - time_sent
52         # calculate the round trip time taken for the packet
53         if icmp.id == ID:
54             # if the ping was sent from this machine then add it to the
55             # list of
56             # responses
57             ip_address, port = addr
58             # this is to prevent a bug where IPs were being added twice
59             if ip_address not in recieved_from:
60                 addresses.add((ip_address, time_taken, ip))
61                 recieved_from.add(ip_address)
62         elif time_remaining <= 0:
63             break
64         else:
65             continue

```



```

65     # return a list of all the addresses that replied to our ICMP echo
        request.
66     return addresses
67
68
69 def udp(dest_ip: str, timeout: float) -> Set[int]:
70     """
71     This listener detects UDP packets from dest_ip in the given timespan,
72     all ports that send direct replies are marked as being open.
73     Returns a list of open ports.
74     """
75
76     time_remaining = timeout
77     ports: Set[int] = set()
78     with socket.socket(
79         socket.AF_INET,
80         socket.SOCK_RAW,
81         socket.IPPROTO_UDP
82     ) as s:
83         while True:
84             time_taken = ip_utils.wait_for_socket(s, time_remaining)
85             if time_taken == -1:
86                 break
87             else:
88                 time_remaining -= time_taken
89                 packet = s.recv(1024)
90                 ip = headers.ip(packet[:20])
91                 udp = headers.udp(packet[20:28])
92                 if dest_ip == ip.source and ip.protocol == 17:
93                     ports.add(udp.src)
94
95     return ports
96
97
98 def icmp_unreachable(src_ip: str, timeout: float = 2) -> int:
99     """
100     This listener detects ICMP destination unreachable
101     packets and returns the icmp code.
102     This is later used to mark them as either close, open|filtered,
        filtered.
103     3 -> closed
104     0|1|2|9|10|13 -> filtered
105     -1 -> error with arguments
106     open|filtered means that they are either open or
107     filtered but return nothing.
108     """
109
110     ping_sock = socket.socket(
111         socket.AF_INET,
112         socket.SOCK_RAW,

```

```

113         socket.IPPROTO_ICMP
114     )
115     # open raw socket to listen for ICMP destination unreachable packets
116     time_remaining = timeout
117     code = -1
118     while True:
119         time_waiting = ip_utils.wait_for_socket(ping_sock,
120                                                 time_remaining)
121         # wait for socket to be readable
122         if time_waiting == -1:
123             break
124         else:
125             time_remaining -= time_waiting
126             recPacket, addr = ping_sock.recvfrom(1024)
127             # recieve the packet
128             ip = headers.ip(recPacket[20:])
129             icmp = headers.icmp(recPacket[20:28])
130             valid_codes = [0, 1, 2, 3, 9, 10, 13]
131             if (
132                 ip.source == src_ip
133                 and icmp.type == 3
134                 and icmp.code in valid_codes
135             ):
136                 code = icmp.code
137                 break
138             elif time_remaining <= 0:
139                 break
140             else:
141                 continue
142     ping_sock.close()
143     return code
144
145 def tcp(address: Tuple[str, int], timeout: float) -> PORTS:
146     """
147     This function is run asynchronously and listens for
148     TCP ACK responses to the sent TCP SYN msg.
149     """
150     ports: DefaultDict[str, Set[int]] = defaultdict(set)
151     with closing(
152         socket.socket(
153             socket.AF_INET,
154             socket.SOCK_RAW,
155             socket.IPPROTO_TCP
156         ) as s:
157         s.bind(address)
158         # bind the raw socket to the listening address
159         time_remaining = timeout
160         while True:
161             time_taken = ip_utils.wait_for_socket(s, time_remaining)

```

```

162         # wait for the socket to become readable
163         if time_taken == -1:
164             break
165         else:
166             time_remaining -= time_taken
167         packet = s.recv(1024)
168         # recieve the packet data
169         tcp = headers.tcp(packet[20:40])
170         if tcp.flags & 2: # syn flags set
171             ports["OPEN"].add(tcp.source)
172         elif tcp.flags & 4:
173             ports["CLOSED"].add(tcp.source)
174         else:
175             continue
176     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.

```

1  import socket
2  import time
3  from modules import directives
4  from modules import headers
5  from modules import ip_utils
6  from modules import listeners
7  from collections import defaultdict
8  from contextlib import closing
9  from itertools import repeat
10 from multiprocessing import Pool
11 from os import getpid
12 from typing import Set, Tuple
13
14
15 def ping(addresses: Set[str]) -> Set[Tuple[str, float, headers.ip]]:
16     """
17     Send an ICMP ECHO REQUEST to each address
18     in the set addresses. Then return a set which
19     contains all the addresses which replied and
20     which have the correct ID.
21     """
22     with closing(
23         socket.socket(
24             socket.AF_INET,
25             socket.SOCK_RAW,
26             socket.IPPROTO_ICMP
27         )
28     ) as ping_sock:
29         # get the local ip address
30         addresses = {
31             ip

```

```

32         for ip in addresses
33         if (
34             not ip.endswith(".0")
35             and not ip.endswith(".255")
36         )
37     }
38
39     # initialise a process pool
40     p = Pool(1)
41     # get the local process id for use in creating packets.
42     ID = getpid() & 0xFFFF
43     # run the listeners.ping function asynchronously
44     replied = p.apply_async(listeners.ping, (ID, 5))
45     time.sleep(0.01)
46     for address in zip(addresses, repeat(1)):
47         try:
48             packet = ip_utils.make_icmp_packet(ID)
49             ping_sock.sendto(packet, address)
50         except PermissionError:
51             ip_utils.eprint("raw sockets require root privileges,
52                             exiting")
53             exit()
54     p.close()
55     p.join()
56     # close and join the process pool to so that all the values
57     # have been returned and the pool closed
58     return replied.get()
59
60 def connect(address: str, ports: Set[int]) -> Set[int]:
61     """
62     This is the most basic kind of scan
63     it simply connects to every specified port
64     and identifies whether they are open.
65     """
66     import socket
67     from contextlib import closing
68     open_ports: Set[int] = set()
69     for port in ports:
70         # loop through each port in the list of ports to scan
71         try:
72             with closing(
73                 socket.socket(
74                     socket.AF_INET,
75                     socket.SOCK_STREAM
76                 )
77             ) as s:
78                 # open an IPV4 TCP socket
79                 s.connect((address, port))
80                 # 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

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

```

```

129 def udp(
130     dest_ip: str,
131     ports_to_scan: Set[int]
132 ) -> listeners.PORTS:
133     """
134     Takes in a destination IP address in either dot or long form and
135     a list of ports to scan. Sends UDP packets to each port specified
136     in portlist and uses the listeners to mark them as open,
137     open|filtered,
138     filtered, closed they are marked open|filtered if no response is
139     recieved at all.
140     """
141     local_port = ip_utils.get_free_port()
142     # get port number
143     ports: listeners.PORTS = defaultdict(set)
144     ports["REMAINING"] = ports_to_scan
145     p = Pool(1)
146     udp_listen = p.apply_async(listeners.udp, (dest_ip, 4))
147     time.sleep(0.01)
148     # start the UDP listener
149     with closing(
150         socket.socket(
151             socket.AF_INET,
152             socket.SOCK_RAW,
153             socket.IPPROTO_UDP
154         )
155     ) as s:
156         for _ in range(2):
157             # repeat 3 times because UDP scanning comes
158             # with a high chance of packet loss
159             for dest_port in ports["REMAINING"]:
160                 try:
161                     packet = ip_utils.make_udp_packet(
162                         local_port,
163                         dest_port
164                     )
165                     # create the UDP packet to send
166                     s.sendto(packet, (dest_ip, dest_port))
167                     # send the packet to the currently scanning address
168                 except socket.error:
169                     packet_bytes = " ".join(map(hex, packet))
170                     print(
171                         "The socket modules sendto method with the
172                         following",
173                         "argument resulting in a socket error.",
174                         f"\npacket: [{packet_bytes}]\n",
175                         "address: [{dest_ip, dest_port}]"
176                     )

```

```

177     p.close()
178     p.join()
179
180     ports["OPEN"].update(udp_listen.get())
181     # if we are on localhost remove the scanning port
182     if dest_ip == "127.0.0.1":
183         ports["OPEN"] -= set([local_port])
184     ports["REMAINING"] -= ports["OPEN"]
185     # only scan the ports which we know are not open
186     with closing(
187         socket.socket(
188             socket.AF_INET,
189             socket.SOCK_RAW,
190             socket.IPPROTO_UDP
191         )
192     ) as s:
193         for dest_port in ports["REMAINING"]:
194             try:
195                 packet = ip_utils.make_udp_packet(
196                     local_port,
197                     dest_port
198                 )
199                 # make a new UDP packet
200                 p = Pool(1)
201                 icmp_listen = p.apply_async(
202                     listeners.icmp_unreachable,
203                     (dest_ip,),
204                 )
205                 # start the ICMP listener
206                 time.sleep(0.01)
207                 s.sendto(packet, (dest_ip, dest_port))
208                 # send packet
209                 p.close()
210                 p.join()
211                 icmp_code = icmp_listen.get()
212                 # receive ICMP code from the ICMP listener
213                 if icmp_code in {0, 1, 2, 9, 10, 13}:
214                     ports["FILTERED"].add(dest_port)
215                 elif icmp_code == 3:
216                     ports["CLOSED"].add(dest_port)
217             except socket.error:
218                 packet_bytes = " ".join(map("{:02x}".format, packet))
219                 ip_utils.eprint(
220                     "The socket modules sendto method with the following",
221                     "argument resulting in a socket error.",
222                     f"\npacket: [{packet_bytes}]\n",
223                     "address: [{dest_ip, dest_port}]"
224                 )
225     # this creates a new set which contains all the elements that
226     # are in the list of ports to be scanned but have not yet

```

```

227     # been classified
228     ports["OPEN|FILTERED"] = (
229         ports["REMAINING"]
230         - ports["OPEN"]
231         - ports["FILTERED"]
232         - ports["CLOSED"]
233     )
234     del(ports["REMAINING"])
235     # set comprehension to update the list of open filtered ports
236     return ports
237
238
239 def version_detect_scan(
240     target: directives.Target,
241     probes: directives.PROBE_CONTAINER
242 ) -> directives.Target:
243     for probe_dict in probes.values():
244         for proto in probe_dict:
245             target = probe_dict[proto].scan(target)
246     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 Python's built in module structure instead.

```

1  #!/usr/bin/env python
2  from icmp_ping import icmp_echo_recv, icmp_echo_send
3  from ping_scanner import ping_scan
4  from tcp_scan.connect_scan import scan_port_list as connect_scan_list
5  from tcp_scan.syn_scan import scan_port_list as syn_scan_list
6  from udp_scan import scan_port_list as udp_scan_list
7  from version_detection import version_detection
8
9  examples = {
10     "icmp_echo_recv": icmp_echo_recv.main,
11     "icmp_echo_send": icmp_echo_send.main,
12     "ping_scanner": ping_scan.main,
13     "connect_scan": connect_scan_list.main,
14     "syn_scan": syn_scan_list.main,
15     "udp_scan": udp_scan_list.main,
16     "version_detection": version_detection.main,
17 }
18
19 print("\n\t".join(("Programs:", *examples)))
20
21 while True:

```



```

22     print()
23     program = input("Enter the name of the example program to run: ")
24     if program.lower() in {"quit", "q", "end", "exit"}:
25         break
26     found = False
27     for name in examples:
28         if name.startswith(program.lower()):
29             program = name
30             print(f"Running: {program}")
31             examples[program]()
32             found = True
33     if not found:
34         print(
35             "The program name must exactly match one of the following
36             examples"
37         )
38     print("\n".join(examples))

```

A.9 netscan

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

```

1  #!/usr/bin/env python
2  import re
3  from argparse import ArgumentParser
4  from collections import defaultdict
5  from math import floor, log10
6  from modules import (
7      scanners,
8      ip_utils,
9      directives,
10 )
11 from typing import (
12     DefaultDict,
13     Dict,
14 )
15
16 top_ports = directives.parse_ports(open("top_ports").read())
17 services: DefaultDict[str, Dict[int, str]] = defaultdict(dict)
18 for match in re.finditer(
19     r"(\S+)\s+(\d+)/(\S+)",
20     open("version_detection/nmap-services").read()
21 ):
22     service, portnum, protocol = match.groups()
23     services[protocol.upper()][int(portnum)] = service
24
25 parser = ArgumentParser()

```

```

26 parser.add_argument(
27     "target_spec",
28     help="specify what to scan, i.e. 192.168.1.0/24"
29 )
30 parser.add_argument(
31     "-Pn",
32     help="assume hosts are up",
33     action="store_true"
34 )
35 parser.add_argument(
36     "-sL",
37     help="list targets",
38     action="store_true"
39 )
40 parser.add_argument(
41     "-sn",
42     help="disable port scanning",
43     action="store_true"
44 )
45 parser.add_argument(
46     "-sS",
47     help="TCP SYN scan",
48     action="store_true"
49 )
50 parser.add_argument(
51     "-sT",
52     help="TCP connect scan",
53     action="store_true"
54 )
55 parser.add_argument(
56     "-sU",
57     help="UDP scan",
58     action="store_true"
59 )
60 parser.add_argument(
61     "-sV",
62     help="version scan",
63     action="store_true"
64 )
65 parser.add_argument(
66     "-p",
67     "--ports",
68     help="scan specified ports",
69     required=False,
70     default=top_ports
71 )
72 parser.add_argument(
73     "--exclude_ports",
74     help="ports to exclude from the scan",
75     required=False,

```

```

76     default=""
77 )
78
79 args = parser.parse_args()
80
81 # check whether the address spec is in CIDR form
82 CIDR_regex =
83     re.compile(r"(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})/(\d{1,2})")
84 search = CIDR_regex.search(args.target_spec)
85 if search:
86     base_addr, network_bits = search.groups()
87     addresses = ip_utils.ip_range(
88         base_addr,
89         int(network_bits)
90     )
91 else:
92     base_addr = args.target_spec
93     if not ip_utils.is_valid_ip(base_addr):
94         raise ValueError(f"invalid dot form IP address: [{base_addr}]")
95     addresses = {base_addr}
96
97 def error_exit(error_type: str, scan_type: str, scanning: str) -> bool:
98     messages = {
99         "permission": "\n".join((
100             "You have insufficient permissions to run this type of scan",
101             "EXITING!"
102         ))
103     }
104     print(f"You tried to scan {scanning} using scan type: {scan_type}")
105     try:
106         print(messages[error_type])
107     except KeyError:
108         print(f"ERROR MESSAGE NOT FOUND: {error_type}")
109     exit(-1)
110
111
112 if args.sL:
113     print("Targets:")
114     print("\n".join(sorted(addresses, key=ip_utils.dot_to_long)))
115 else:
116     if args.sn:
117         def sig_figs(x: float, n: int) -> float:
118             """
119             rounds x to n significant figures.
120             sig_figs(1234, 2) = 1200.0
121             """
122             return round(x, n - (1 + int(floor(log10(abs(x))))))
123
124     try:

```

```

125         print("\n".join(
126             f"host: [{host}]\t" +
127             "responded to an ICMP ECHO REQUEST in " +
128             f"{str(sig_figs(taken, 2))+ 's':<10s} " +
129             f"ttl: [{ip_head.time_to_live}]"
130             for host, taken, ip_head in scanners.ping(addresses)
131         ))
132     except PermissionError:
133         error_exit("permission", "ping scan", str(addresses))
134
135 else:
136     if args.Pn:
137         targets = [
138             directives.Target(
139                 addr,
140                 defaultdict(set),
141                 defaultdict(set)
142             )
143             for addr in addresses
144         ]
145     else:
146         try:
147             targets = [
148                 directives.Target(
149                     addr,
150                     defaultdict(set),
151                     defaultdict(set),
152                 )
153                 for addr, _, _ in scanners.ping(addresses)
154             ]
155         except PermissionError:
156             error_exit("permission", "ping_scan", str(addresses))
157     # define the ports to scan
158     if args.ports == "-":
159         # case they have specified all ports
160         ports = {
161             "UDP": set(range(1, 65536)),
162             "TCP": set(range(1, 65536)),
163         }
164     elif isinstance(args.ports, str):
165         # case they have specified ports
166         ports = directives.parse_ports(args.ports)
167     else:
168         # default
169         ports = args.ports
170
171     # exclude all the ports speified to be excluded
172     to_exclude = directives.parse_ports(args.exclude_ports)
173     ports["TCP"] -= to_exclude["TCP"]
174     ports["TCP"] -= to_exclude["ANY"]

```

```

175     ports["UDP"] -= to_exclude["UDP"]
176     ports["UDP"] -= to_exclude["ANY"]
177
178     # if version scanning is desired
179     if args.sV:
180         probes = directives.parse_probes(
181             "./version_detection/nmap-service-probes"
182         )
183
184     for target in targets:
185         if not args.sU and not args.sT or args.sS:
186             try:
187                 tcp_ports = scanners.tcp(
188                     target.address,
189                     ports["TCP"] | ports["ANY"]
190                 )
191             except PermissionError:
192                 error_exit("permission", "tcp_scan", target.address)
193             target.open_ports["TCP"].update(tcp_ports["OPEN"])
194             target.open_filtered_ports["TCP"].update(
195                 tcp_ports["FILTERED"]
196             )
197         if args.sT:
198             target.open_ports["TCP"].update(
199                 scanners.connect(
200                     target.address,
201                     ports["TCP"] | ports["ANY"]
202                 )
203             )
204         if args.sU:
205             try:
206                 udp_ports = scanners.udp(
207                     target.address,
208                     ports["UDP"] | ports["ANY"]
209                 )
210             except PermissionError:
211                 error_exit("permission", "udp_scan", target.address)
212
213             target.open_ports["UDP"].update(
214                 udp_ports["OPEN"]
215             )
216             target.open_filtered_ports["UDP"].update(
217                 udp_ports["FILTERED"]
218             )
219             target.open_filtered_ports["UDP"].update(
220                 udp_ports["OPEN|FILTERED"]
221             )
222         if args.sV:
223             target = scanners.version_detect_scan(target, probes)
224     # display scan info

```

```

225     print()
226     print(f"Scan report for: {target.address}")
227     # print(target)
228     print("Open ports:")
229     for proto, open_ports in target.open_ports.items():
230         for port in open_ports:
231             try:
232                 service_name = services[proto][port]
233             except KeyError:
234                 service_name = "unknown"
235         if port in target.services:
236             exact_match = target.services[port]
237             print(
238                 f"{port}/{proto}{exact_match.service:>8s}"
239             )
240             # print version information
241             for key, val in exact_match.version_info.items():
242                 print(f"{key}: {val}")
243             if exact_match.cpes:
244                 print()
245                 print("CPE:")
246                 for cpe_type, cpe_vals in
247                     exact_match.cpes.items():
248                     print(cpe_type)
249                     try:
250                         del(cpe_vals["part"])
251                     except KeyError:
252                         pass
253                     for key, val in cpe_vals.items():
254                         print(f"{key}: {val}")
255             print()
256         else:
257             print(f"{port} service: {service_name}?")
258
259     print("Filtered ports:")
260     for proto, filtered_ports in
261         target.open_filtered_ports.items():
262         for port in filtered_ports:
263             try:
264                 service_name = services[proto][port]
265             except KeyError:
266                 service_name = "unknown"
267             print(f"{port} service: {service_name}?")

```

A.10 tests

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

```

1  from modules.ip_utils import (
2      dot_to_long,
3      long_to_dot,
4      ip_range,
5      is_valid_ip,
6      is_valid_port_number,
7      ip_checksum,
8      make_tcp_packet,
9      make_udp_packet,
10     make_icmp_packet,
11 )
12 from binascii import unhexlify
13
14
15 def test_dot_to_long_private_ip() -> None:
16     assert(dot_to_long("192.168.1.0") == 0xC0A80100)
17
18
19 def test_long_to_dot_private_ip() -> None:
20     assert(long_to_dot(0xC0A80100) == "192.168.1.0")
21
22
23 def test_dot_to_long_localhost() -> None:
24     assert(dot_to_long("127.0.0.1") == 0x7F000001)
25
26
27 def test_long_to_dot_localhost() -> None:
28     assert(long_to_dot(0x7F000001) == "127.0.0.1")
29
30
31 def test_is_valid_ip_localhost_long() -> None:
32     assert is_valid_ip(0x7F000001)
33
34
35 def test_is_valid_ip_localhost() -> None:
36     assert is_valid_ip("127.0.0.1")
37
38
39 def test_is_not_valid_ip_5_zeros_dotted() -> None:
40     assert not is_valid_ip("0.0.0.0.0")
41
42
43 def test_is_not_valid_ip_5_255s_long() -> None:
44     assert not is_valid_ip(0xFF_FF_FF_FF_FF)
45
46
47 def test_is_valid_port_number_0() -> None:
48     assert is_valid_port_number(0)
49
50

```

```

51 def test_is_valid_port_number_65535() -> None:
52     assert is_valid_port_number(65535)
53
54
55 def test_is_not_valid_port_number_negative_one() -> None:
56     assert not is_valid_port_number(-1)
57
58
59 def test_is_not_valid_port_number_65536() -> None:
60     assert not is_valid_port_number(65536)
61
62
63 def test_ip_range() -> None:
64     assert(
65         ip_range("192.168.1.0", 28) == {
66             "192.168.1.1",
67             "192.168.1.2",
68             "192.168.1.3",
69             "192.168.1.4",
70             "192.168.1.5",
71             "192.168.1.6",
72             "192.168.1.7",
73             "192.168.1.8",
74             "192.168.1.9",
75             "192.168.1.10",
76             "192.168.1.11",
77             "192.168.1.12",
78             "192.168.1.13",
79             "192.168.1.14",
80         }
81     )
82
83
84 def test_ip_checksum_verify() -> None:
85     packet = unhexlify(
86         "45000073000040004011b861c0a80001c0a800c7"
87     )
88     assert ip_checksum(packet) == 0
89
90
91 def test_ip_checksum_generate() -> None:
92     packet = unhexlify(
93         "450000730000400040110000c0a80001c0a800c7"
94     )
95     assert ip_checksum(packet) == 0xB861
96
97
98 def test_make_tcp_packet() -> None:
99     correct = unhexlify(
100         "e54700500000000000000000600204002af50000020405b4"

```



```

101     )
102     info = 58695, 80, "192.168.1.45", "192.168.1.28", 2
103     assert correct == make_tcp_packet(*info)
104
105
106 def test_make_udp_packet() -> None:
107     correct = unhexlify(
108         "e5470050003a0000"
109     )
110     info = 58695, 80
111     # clipping the packet at 8 simply removes the data section
112     assert correct == make_udp_packet(*info)[:8]

```

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

```

1 from modules.directives import (
2     parse_ports
3 )
4 from collections import defaultdict
5 from typing import DefaultDict
6
7
8 def test_parse_probes_single() -> None:
9     portstring = "12345"
10    expected: DefaultDict[str, set] = defaultdict(set)
11    expected["ANY"] = set([12345])
12    assert expected == parse_ports(portstring)
13
14
15 def test_parse_probes_range() -> None:
16     portstring = "10-20"
17     expected: DefaultDict[str, set] = defaultdict(set)
18     expected["ANY"] = set(range(10, 21))
19     assert expected == parse_ports(portstring)
20
21
22 def test_parse_probes_single_and_range() -> None:
23     portstring = "1,2,3,10-20,6,7,8"
24     expected: DefaultDict[str, set] = defaultdict(set)
25     expected["ANY"] = set([1, 2, 3, *range(10, 21), 6, 7, 8])
26     assert expected == parse_ports(portstring)
27
28
29 def test_parse_probes_tcp_single() -> None:
30     portstring = "T:12345"
31     expected: DefaultDict[str, set] = defaultdict(set)
32     expected["TCP"] = set([12345])
33     assert expected == parse_ports(portstring)
34
35

```

```

36 def test_parse_probes_tcp_range() -> None:
37     portstring = "T:10-20"
38     expected: DefaultDict[str, set] = defaultdict(set)
39     expected["TCP"] = set(range(10, 21))
40     assert expected == parse_ports(portstring)
41
42
43 def test_parse_probes_tcp_single_and_range() -> None:
44     portstring = "T:1,2,3,10-20,6,7,8"
45     expected: DefaultDict[str, set] = defaultdict(set)
46     expected["TCP"] = set([1, 2, 3, *range(10, 21), 6, 7, 8])
47     assert expected == parse_ports(portstring)
48
49
50 def test_parse_probes_udp_single() -> None:
51     portstring = "U:12345"
52     expected: DefaultDict[str, set] = defaultdict(set)
53     expected["UDP"] = set([12345])
54     assert expected == parse_ports(portstring)
55
56
57 def test_parse_probes_udp_range() -> None:
58     portstring = "U:10-20"
59     expected: DefaultDict[str, set] = defaultdict(set)
60     expected["UDP"] = set(range(10, 21))
61     assert expected == parse_ports(portstring)
62
63
64 def test_parse_probes_udp_single_and_range() -> None:
65     portstring = "U:1,2,3,10-20,6,7,8"
66     expected: DefaultDict[str, set] = defaultdict(set)
67     expected["UDP"] = set([1, 2, 3, *range(10, 21), 6, 7, 8])
68     assert expected == parse_ports(portstring)
69
70
71 def test_parse_probes_any_and_tcp_single() -> None:
72     portstring = "12345 T:12345"
73     expected: DefaultDict[str, set] = defaultdict(set)
74     expected["TCP"] = set([12345])
75     expected["ANY"] = set([12345])
76     assert expected == parse_ports(portstring)
77
78
79 def test_parse_probes_any_and_tcp_range() -> None:
80     portstring = "10-20 T:10-20"
81     expected: DefaultDict[str, set] = defaultdict(set)
82     expected["TCP"] = set(range(10, 21))
83     expected["ANY"] = set(range(10, 21))
84     assert expected == parse_ports(portstring)
85

```

```

86
87 def test_parse_probes_any_and_tcp_single_and_range() -> None:
88     portstring = "1,2,3,10-20,6,7,8 T:1,2,3,10-20,6,7,8"
89     expected: DefaultDict[str, set] = defaultdict(set)
90     expected["TCP"] = set([1, 2, 3, *range(10, 21), 6, 7, 8])
91     expected["ANY"] = set([1, 2, 3, *range(10, 21), 6, 7, 8])
92     assert expected == parse_ports(portstring)
93
94
95 def test_parse_probes_any_and_udp_single() -> None:
96     portstring = "12345 U:12345"
97     expected: DefaultDict[str, set] = defaultdict(set)
98     expected["UDP"] = set([12345])
99     expected["ANY"] = set([12345])
100     assert expected == parse_ports(portstring)
101
102
103 def test_parse_probes_any_and_udp_range() -> None:
104     portstring = "10-20 U:10-20"
105     expected: DefaultDict[str, set] = defaultdict(set)
106     expected["UDP"] = set(range(10, 21))
107     expected["ANY"] = set(range(10, 21))
108     assert expected == parse_ports(portstring)
109
110
111 def test_parse_probes_any_and_udp_single_and_range() -> None:
112     portstring = "1,2,3,10-20,6,7,8 U:1,2,3,10-20,6,7,8"
113     expected: DefaultDict[str, set] = defaultdict(set)
114     expected["UDP"] = set([1, 2, 3, *range(10, 21), 6, 7, 8])
115     expected["ANY"] = set([1, 2, 3, *range(10, 21), 6, 7, 8])
116     assert expected == parse_ports(portstring)
117
118
119 def test_parse_probes_udp_and_tcp_single() -> None:
120     portstring = "U:12345 T:12345"
121     expected: DefaultDict[str, set] = defaultdict(set)
122     expected["TCP"] = set([12345])
123     expected["UDP"] = set([12345])
124     assert expected == parse_ports(portstring)
125
126
127 def test_parse_probes_udp_and_tcp_range() -> None:
128     portstring = "U:10-20 T:10-20"
129     expected: DefaultDict[str, set] = defaultdict(set)
130     expected["TCP"] = set(range(10, 21))
131     expected["UDP"] = set(range(10, 21))
132     assert expected == parse_ports(portstring)
133
134
135 def test_parse_probes_udp_and_tcp_single_and_range() -> None:

```

```

136     portstring = "U:1,2,3,10-20,6,7,8 T:1,2,3,10-20,6,7,8"
137     expected: DefaultDict[str, set] = defaultdict(set)
138     expected["TCP"] = set([1, 2, 3, *range(10, 21), 6, 7, 8])
139     expected["UDP"] = set([1, 2, 3, *range(10, 21), 6, 7, 8])
140     assert expected == parse_ports(portstring)
141
142
143     def test_parse_probes_all_single() -> None:
144         portstring = "12345 U:12345 T:12345"
145         expected: DefaultDict[str, set] = defaultdict(set)
146         expected["TCP"] = set([12345])
147         expected["UDP"] = set([12345])
148         expected["ANY"] = set([12345])
149         assert expected == parse_ports(portstring)
150
151
152     def test_parse_probes_all_range() -> None:
153         portstring = "10-20 U:10-20 T:10-20"
154         expected: DefaultDict[str, set] = defaultdict(set)
155         expected["TCP"] = set(range(10, 21))
156         expected["UDP"] = set(range(10, 21))
157         expected["ANY"] = set(range(10, 21))
158         assert expected == parse_ports(portstring)
159
160
161     def test_parse_probes_all_single_and_range() -> None:
162         portstring = "1,2,3,10-20,6,7,8 U:1,2,3,10-20,6,7,8
163                     T:1,2,3,10-20,6,7,8"
164         expected: DefaultDict[str, set] = defaultdict(set)
165         expected["TCP"] = set([1, 2, 3, *range(10, 21), 6, 7, 8])
166         expected["UDP"] = set([1, 2, 3, *range(10, 21), 6, 7, 8])
167         expected["ANY"] = set([1, 2, 3, *range(10, 21), 6, 7, 8])
168         assert expected == parse_ports(portstring)

```

References

- [1] Anonymous. Daemon (computing). [https://en.wikipedia.org/wiki/Daemon_\(computing\)](https://en.wikipedia.org/wiki/Daemon_(computing)), March 2019.
- [2] Anonymous. UPower. <https://en.wikipedia.org/wiki/UPower>, February 2019.
- [3] Anonymous. systemd. <https://en.wikipedia.org/wiki/Systemd>, April 2019.
- [4] Anonymous. D-Bus. <https://en.wikipedia.org/wiki/D-Bus>, April 2019.
- [5] Anonymous. iwd. <https://wiki.archlinux.org/index.php/Iwd>, April 2019.
- [6] Anonymous. dhcpcd. <https://wiki.archlinux.org/index.php/dhcpcd>, April 2019.
- [7] Anonymous. Dynamic Host Configuration Protocol. https://en.wikipedia.org/wiki/Dynamic_Host_Configuration_Protocol, April 2019.
- [8] Anonymous. IP address. https://en.wikipedia.org/wiki/IP_address, April 2019.
- [9] Wireshark Core Developers. Wireshark download page. <https://www.wireshark.org/download.html>.
- [10] Anonymous. OSI model. https://en.wikipedia.org/wiki/Osi_model, March 2019.
- [11] Anonymous. Port (computer networking). [https://en.wikipedia.org/wiki/Port_\(computer_networking\)](https://en.wikipedia.org/wiki/Port_(computer_networking)), March 2019.
- [12] Information Sciences Institute University of Southern California. Transmission Control Protocol. <https://tools.ietf.org/html/rfc793>, September 1981.
- [13] J. Postel ISI. User Datagram Protocol. <https://www.ietf.org/rfc/rfc768.txt>, August 1980.
- [14] anonymous. Transmission Control Protocol. https://en.wikipedia.org/wiki/Transmission_Control_Protocol, april 2019.
- [15] anonymous. User Datagram Protocol. https://en.wikipedia.org/wiki/User_Datagram_Protocol, april 2019.
- [16] Python Core Developers. http.server module documentation. <https://docs.python.org/3/library/http.server.html>, April 2019.

- [17] Python Core Developers. Socket module documentation. <https://docs.python.org/3/library/socket.html>, April 2019.
- [18] Linux Developers. iptables Man Pages. <https://linux.die.net/man/8/iptables>, March 2019.
- [19] Internet Engineering Task Force. Requirements for Internet Hosts – Communication Layers. <https://tools.ietf.org/html/rfc1122>, October 1989.
- [20] Linux Developers. TCP Man page. <https://linux.die.net/man/7/tcp>, March 2019.
- [21] Gordon “fyodor” lyon. Nmap download page. <https://nmap.org/download.html>.
- [22] Linux Developers. grep Man Pages. <https://linux.die.net/man/1/grep>, March 2019.
- [23] Anonymous. IPv4 header checksum. https://en.wikipedia.org/wiki/IPv4_header_checksum, April 2019.
- [24] Anonymous. Hypertext Transfer Protocol. https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol, April 2019.
- [25] Anonymous. Internet protocol suite. https://en.wikipedia.org/wiki/Internet_protocol_suite, March 2019.
- [26] Anonymous. Wireshark. <https://en.wikipedia.org/wiki/Wireshark>, April 2019.
- [27] Joe. send icmp echo request. <https://stackoverflow.com/questions/24575524/send-icmp-echo-request>, July 2014.
- [28] brice. Python raw sockets. <https://stackoverflow.com/questions/1117958/how-do-i-use-raw-socket-in-python>, June 2011.
- [29] Python Core Developers. defaultdict documentation. <https://docs.python.org/3/library/collections.html?highlight=collection#collections.defaultdict>, April 2019.
- [30] Python Core Developers. builtin data structures documentation. <https://docs.python.org/3/tutorial/datastructures.html#data-structures>, April 2019.
- [31] Python Core Developers. type hinting documentation. <https://docs.python.org/3/library/typing.html?highlight=typing#module-typing>, April 2019.
- [32] Python Core Developers. command line argument parsing documentation. <https://docs.python.org/3/library/argparse.html?highlight=typing>, April 2019.

- [33] Python Core Developers. stderr documentation. <https://docs.python.org/3/library/sys.html?highlight=stderr#sys.stderr>, April 2019.
- [34] Python Core Developers. multiprocessing documentation. <https://docs.python.org/3/library/multiprocessing.html>, April 2019.
- [35] Python Core Developers. struct documentation. <https://docs.python.org/3/library/struct.html>, April 2019.
- [36] Python Core Developers. operator documentation. <https://docs.python.org/3/library/operator.html>, April 2019.
- [37] Gordon 'Fyodor' Lyon. port scanning techniques. <https://nmap.org/book/man-port-scanning-techniques.html>, January 2001.
- [38] Gordon 'Fyodor' Lyon. service and version detection techniques. <https://nmap.org/book/man-version-detection.html>, January 2001.
- [39] Gordon 'Fyodor' Lyon. service and version detection techniques described. <https://nmap.org/book/vscan-technique.html>, January 2001.
- [40] Gordon 'Fyodor' Lyon. service and version detection file format. <https://nmap.org/book/vscan-fileformat.html>, January 2001.
- [41] Information Sciences Institute University of Southern California. Internet Protocol. <https://tools.ietf.org/html/rfc791>, September 1981.
- [42] J. Postel ISI. Internet Control Message Protocol. <https://tools.ietf.org/html/rfc792>, September 1981.
- [43] Bucknell University R. Droms. Dynamic Host Configuration Protocol. <https://www.ietf.org/rfc/rfc2131.txt>, March 1997.
- [44] et al. Fielding. HyperText Transfer Protocol. <https://tools.ietf.org/html/rfc2616>, 1999.
- [45] Berners-Lee & Connolly. HyperText Markup Language - 2.0. <https://tools.ietf.org/html/rfc2616>, November 1995.
- [46] Linux Developers. IP Man page. <https://linux.die.net/man/7/ip>, March 2019.
- [47] Linux Developers. UDP Man page. <https://linux.die.net/man/7/udp>, March 2019.
- [48] Linux Developers. ICMP Man page. <https://linux.die.net/man/7/icmp>, March 2019.
- [49] Linux Developers. DHCPD Man Pages. <https://linux.die.net/man/8/dhcpd>, March 2019.

Glossary

API Applications Programming Interface 5, 28

ARP Address Resolution Protocol 57

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, 11, 17, 70

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. 18, 39

CIDR Classless Inter-Domain Routing 17, 25, 48, 50, 70

CPE Common Platform Enumeration 65

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

DHCPD Dynamic Host Configuration Protocol Client Daemon 3

DNS Domain Name System 23

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 18

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 6, 15

HTTPS HyperText Transfer Protocol Secure 16

ICMP Internet Control Message Protocol 16, 17, 27, 28, 29, 33, 34, 35, 43, 46, 50, 55, 62, 63, 69, 70

IDS Intrusion Detection System 18

IP Internet Protocol 28, 35, 50, 66, 69, 71

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, 4, 6, 15, 48, 50, 66, 67, 68

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 57

NIC Network Interface Card 3, 5, 57

OSI model Open Systems Interconnection model 5, 7, 17, 28

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. 4, 5, 6, 7, 8, 10, 11, 15, 16, 18, 39, 41

PCAP Packet CAPture 38

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, 41, 49

port knocking Port knocking is where packets must be sent to a sequence of ports before access to the desired port is granted. 18

RDP Remote Desktop Protocol 65

SCTP Stream Control Transmission Protocol 18

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. 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, 11, 18, 41

SSH Secure SHell 65

subnet A subnet is simply the sub-network of every possible IP address that will be used for communication on a particular network. 4, 48

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

TCP Transmission Control Protocol 6, 7, 11, 12, 14, 16, 17, 18, 27, 35, 41, 45, 50, 57, 59, 60, 61, 64, 65, 69, 70, 71

UDP User Datagram Protocol 6, 16, 17, 18, 27, 46, 50, 62, 63, 64, 69, 71

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

XML eXtensible Markup Language 22