A Level Computer Science Non-Examined Assessment (NEA)

Sam Leonard

Contents

1 Analysis

1.1 Identification and Background to the Problem

The problem I am trying to solve with my project is how to look at devices on a network from a "black box" perspective and gain information about what

services are running etc. Services are programs which their entire purpose is to provide a *service* to other programs, for example a server hosting a website would be running a service whose purpose is to send the webpage to people who try to connect to the website.
There are many steps in-between a device turning on to interacting with the internet.

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

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

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

Once the daemons are all started the DHCP client can now take issue commands to the daemon for it to carry out. The DHCP client is simply a daemon that runs in the background to carry out any interactions between the current machine and the DHCP server. The DHCP server is normally the WiFi router or network switch for the local network and it manages a list of which computer has which IP address and negotiates with new computers trying to join a network to get them a free IP address. The DHCP client starts the DHCP address negotiation with the server by sending a discover message with the address 255.255.255.255 which is the IP limited broadcast address which means that whatever is listening at the other end will forward this packet on to everyone on the subnet. When the DHCP server (normally the router, sometimes a separate machine) on the subnet receives this message it reserves a free IP address for that client and then responds with a DHCP offer which contains the address the server is offering, the length of time the address is valid for and the subnet mask of the network. The client must then respond with a DHCP request message to request the offered address, this is in case of multiple DHCP servers offering addresses. Finally the DHCP server responds with a DHCP acknowledge message showing that it has received the request. Figure 2 shows a packet

capture from my laptop where I turned WiFi off, started wireshark listening and plugged in an Ethernet cable, I have it showing only the DHCP packets so that it is clear to see the entire DHCP negotiation including the 255.255.255.255 limited broadcast destination address and the 0.0.0.0 unassigned address in the source column. I mention using wireshark to do packet capturing above without explaining what either packet capturing or wireshark are so I will do that here. Packets I define below and wireshark is simply a tool which intercepts all the network communications on a single computer and records them to a file as well as displaying them to the user as well as performing some analysis and dissecting each of the protocols used. This means that I can record the DHCP negotiation shown below and show it to you using wireshark to get all the information out of the packets being sent over the wire.

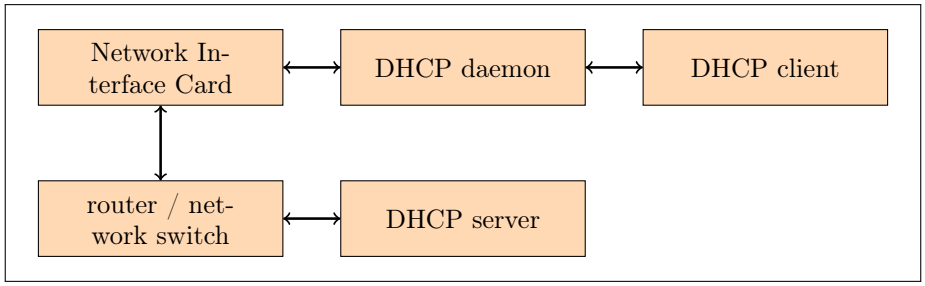

Figure 1: A block diagram showing the relationship between different elements of a DHCP negotiation.

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

Figure 2: DHCP address negotiation

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

7. Application: Applications Programming Interface (API)s, Hypertext transfer Protocol (HTTP), File Transfer Protocol (FTP) among others.
6. Presentation: encryption/decryption, encoding/decoding, decompression etc...
5. Session: Managing sessions, PHP Hypertext Processor (PHP) session IDs etc...
4. Transport: TCP and UDP among others.

3. Network: ICMP and IP among others.

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

1. Physical: The physical Ethernet cabling/NIC.

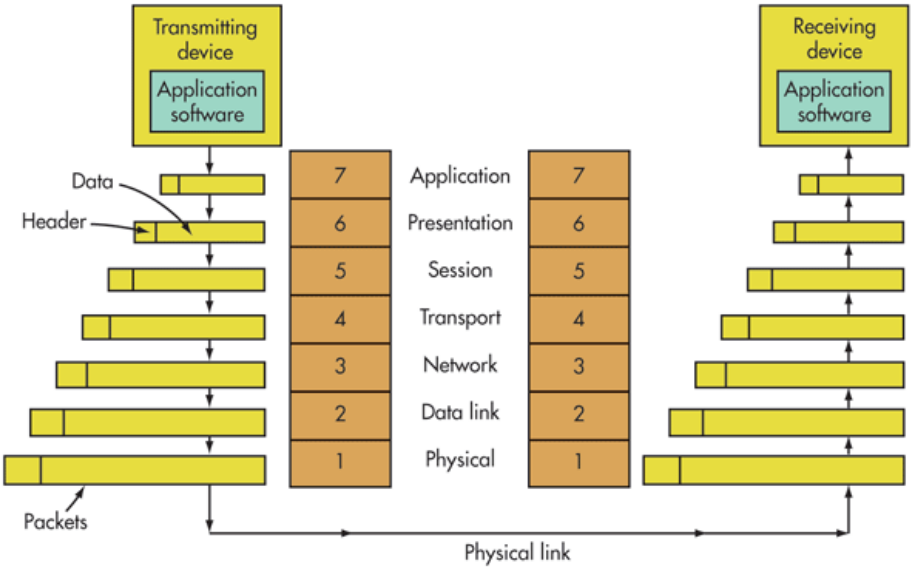

Figure 3: OSI model diagram, source: https://www.electronicdesign.com

Each of these layers is essential to the running of the internet but a single communication might not include all of the layers. These communications are all based on the most fundamental part of the internet: the packet. Packets are sequences of ones and zeros sent between computers which are used to transfer data as well as to control how networks function. They consist of different layers of information each specifying where the packet where should go next at a different level along with fundamentally the data/instructions contained in the innermost layer. When packets are sent between computers a certain number of layers are stripped off by each computer so that it knows where to send the packet next at which point it will add all the layers back again, this time with the instructions needed to go from the current computer to the next one on its route. Each of these layers actually consists of a number of fields at the start called a header some layers also append a footer to the end of the packet. The actual data being transferred in the packet can be quite literally anything, HTTP transfers websites so Hypertext Markup Language (HTML) files and images etc.... In particular there are two pieces of information stored in headers which together define the final destination of the packet: the IP

address and the port number. The IP address defines the destination machine and the port number defines which "port" on the remote machine the packet should be sent to. Ports are essential entrances to a computer, for example if a computer was a hotel the IP address would be the address and location of the hotel and the port number would be the room inside the hotel. There are 65535 ports and 0 is a special reserved port. Both Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) use ports, TCP ports are mainly used for transferring data where reliability is a concern, as TCP has built in checks for packet loss whereas UDP does not and as such is used for purposes where speed is more important and missing some data is inconsequential, such as video streaming and playing games.

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

This shows clearly the interaction between each of the different layers in the OSI model, the browser at level 7: Application rendering the webpage. Level 6: Presentation is skipped as we have no files which need to be served compressed because they are so large. Level 5: Session is shown by the TCP session negotiation and graceful teardown of the TCP session. Level 4: Transport is shown when the image and webpage are transferred from the server to the browser. Level 3/2/1 are shown in figure 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.

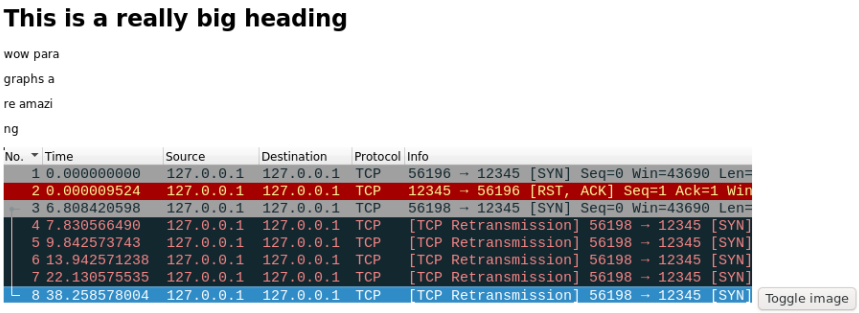

Figure 4: A basic static HTML webpage.

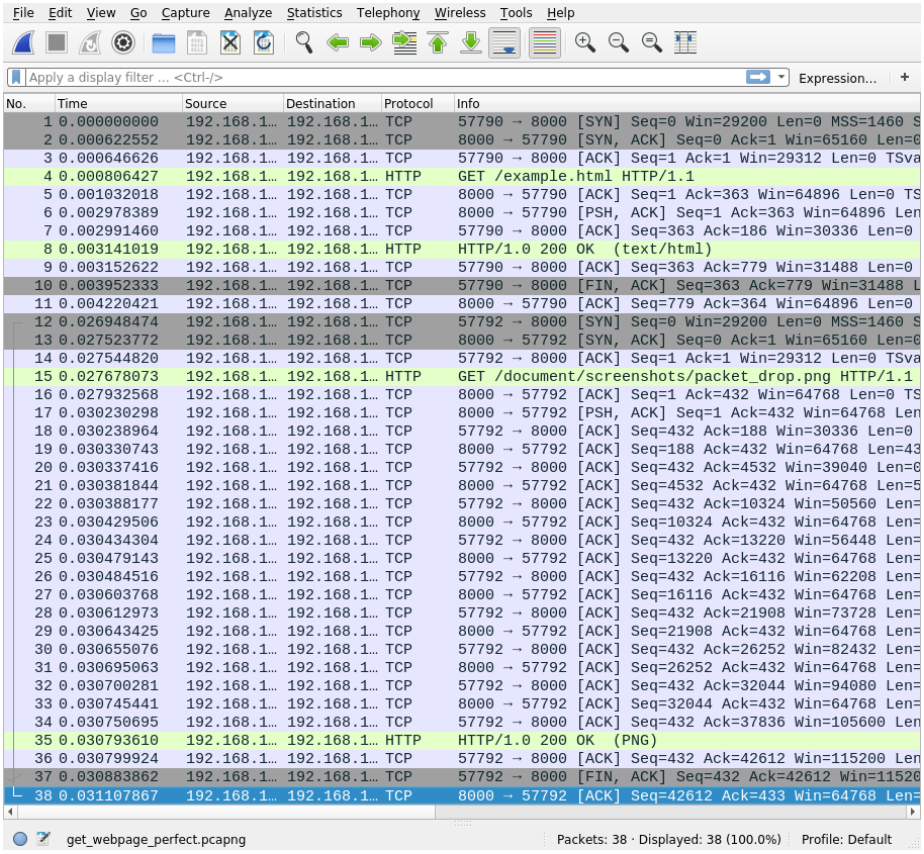

Figure 5: A full chain of packets that shows retrieving a basic webpage from the server.

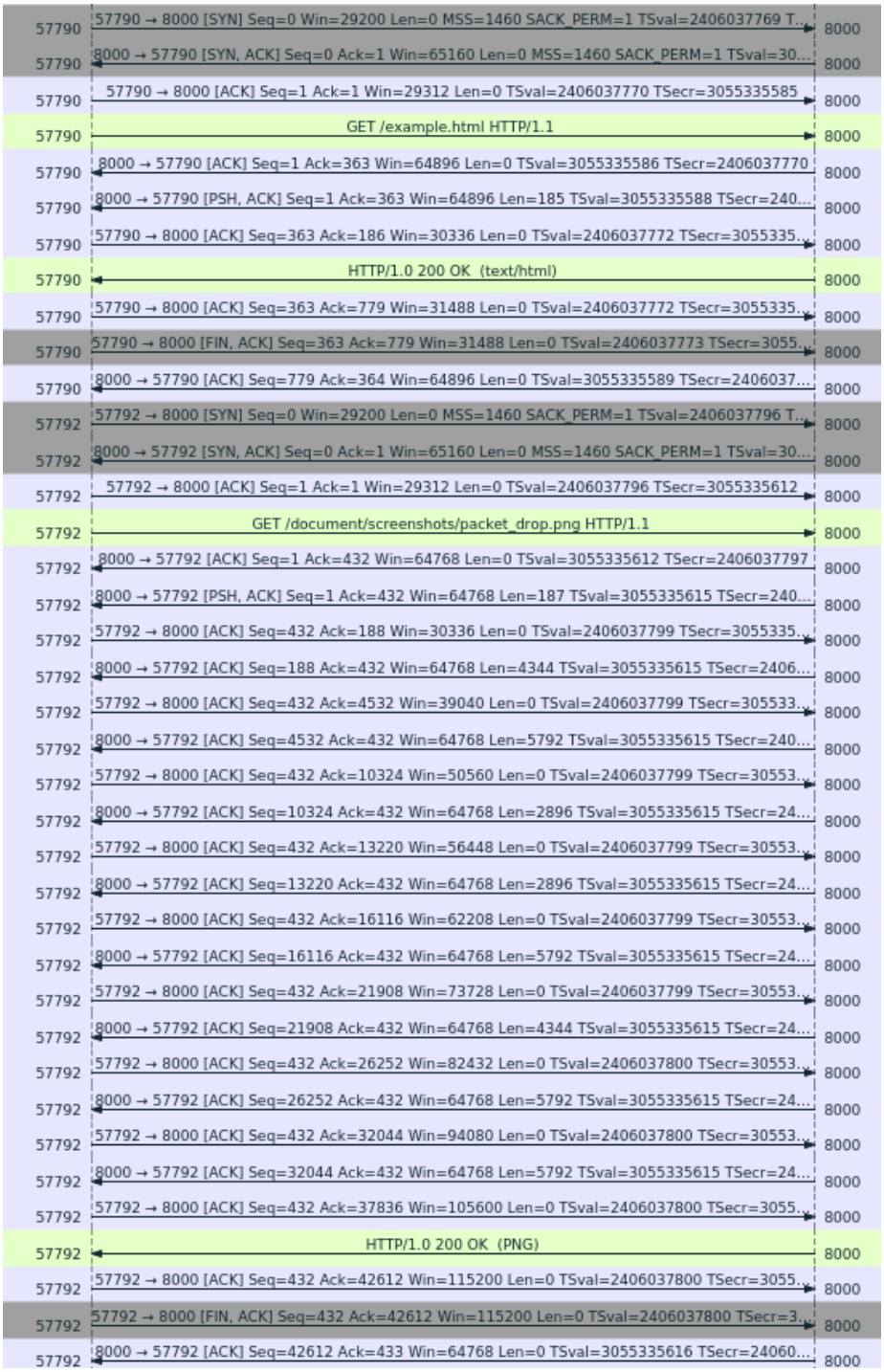

Figure 6: Ladder diagram of figure 5.

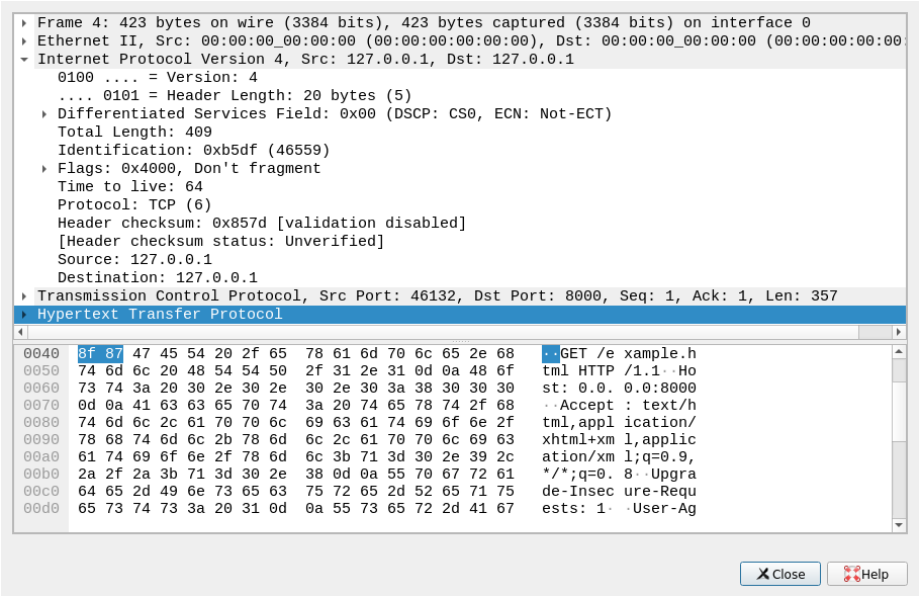

Figure 7: A look inside a TCP packet.

Listing 1: example.html

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

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

  </script>

```

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

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


</body>
</html>
```

1.2 Analysis of problem

The problem with looking at a network from the outside is that the purpose of the network is to allow communication inside of the network, thus very little is exposed externally. This presents a challenge as we want to know what is on the network as well as what each of them is running which is not always possible due to the limited information that services will reveal about themselves. Firewalls also play large part in making scanning networks difficult as sometimes they simply drop packets instead of sending a TCP RST packet (reset connection packet). When firewalls drop packets it becomes exponentially more difficult as you don't know whether your packet was corrupted or lost in transit or if it was just dropped.

To demonstrate this I will show three things:

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

Firstly A successful TCP connection. For a TCP connection to be established there is a three way handshake between the communicating machines. Firstly the machine trying to establish the connection sends a TCP SYN packet to the other machine, this packet holds a dual purpose, to ask for a connection and if it is accepted to SYNchronise the sequence numbers being used to detect whether packets have been lost in transport. The receiving machine then replies with a TCP SYN ACK which confirms the starting sequence number with the SYN part and ACKnowledges the connection request. The sending machine then acknowledges this by sending a final TCP ACK packet back. This connection initialisation is shown in figure 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 in the packet along with the PSH and ACK flags being set. The code I used to generate these is shown in figures 9 and 10. Breaking the code down in figure 10 you can see me initialising a socket object then I bind it to localhost (127.0.0.1) port 12345 localhost is just an address which allows connections between programs running on the same computer as connections are

looped back onto the current machine, hence its alternative name: the loopback address. I then tell it to listen for incoming connections, the one just means how many connections to keep as a backlog. I then accept the connection from the program in figure 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.

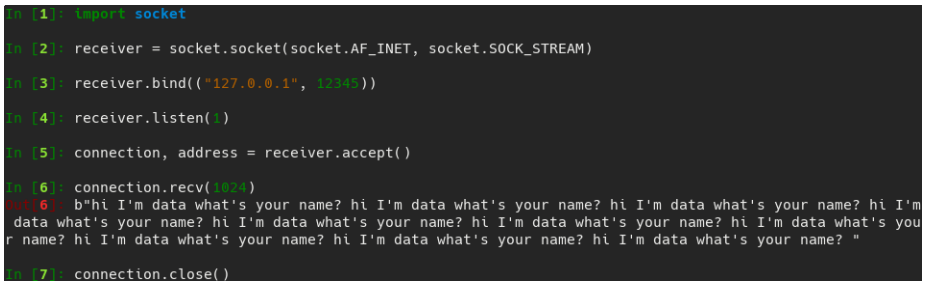

Figure 10: Receiving some basic text data over a TCP connection.

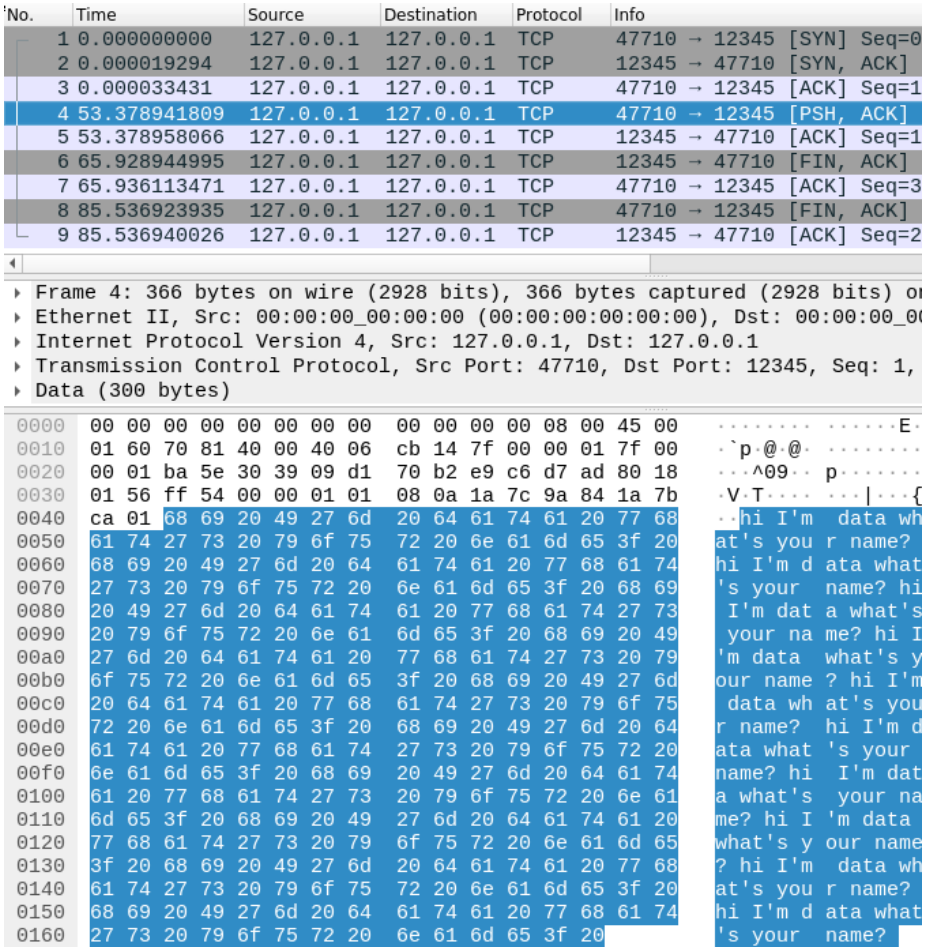

Figure 11: Highlighted packet carrying the data being transferred in figure 9.

Next an attempted connection to a closed port. In figure 12 packet one you can see the same TCP SYN packet as we saw in the attempted connection to an open port, as you would expect. The difference comes in the next packet with the TCP RST flag being sent back. This flag means to reset the connection, or if the connection is not yet established as in this case it means that the port is closed, hence why the packet is highlighted red in figure 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.

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

Firewalls are essentially the gatekeepers of the internet they decide whether a packet gets to pass or whether they shall not pass. Firewalls work by a set of rules which decide what happens to it. A rule might be that it is coming from a certain IP address or has a certain destination port. The actions taken after the packet has had it's fate decided by the rules can be one of the following three (on iptables on linux): ACCEPT, DROP and RETURN, accept does exactly what you think it would an lets the packet through, drop quite literally just drops the packet and sends no reply whatsoever, return is more complicated and has no effect on how port scanning is done and as such we will ignore it. A common set of rules for something like a webserver would be to DROP all incoming packets and then allow exceptions for certain ports i.e. port 80 for HTTP or 443 for Hypertext transfer Protocol Secure (HTTPS). I will be using a linux utility called iptables for implementing all firewall rules on my system for demonstration purposes. Packet number three in figure 12 shows the connection request from line 4 of 13 except that I have enabled a firewall rule to drop all packets from the address 127.0.0.1, using the iptables command as so: `iptables -I INPUT -s 127.0.0.1 -j DROP`. This command reads as for all packets arriving (`-I INPUT`) with source address 127.0.0.1 (`-s 127.0.0.1`) drop them sending no response (`-j DROP`). With this firewall rule in place you can see in figure 12 packet 3 receives no response and as such Python assumes that the packet just got lost and as such tries to send the packet again repeatedly, this continued for more than 30 seconds before a stopped it as shown by the time column in figure 12 and the final `KeyboardInterrupt` in figure 13. The amount of time that a system will wait still trying to reconnect depends on the OS and a other factors but the minimum time is 100 seconds as specified by RFC 1122, on most systems it will be between 13 and 30 minutes according the linux manual page on TCP.

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

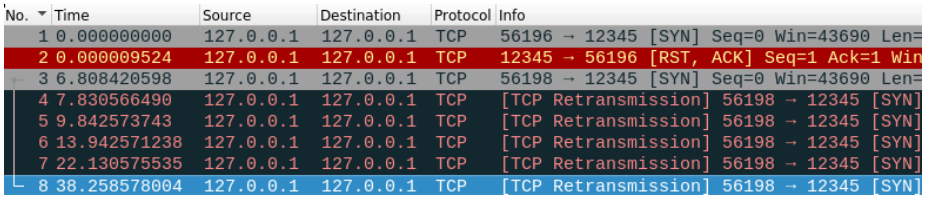

Figure 12: Attempted connection to a closed port with and without firewall rule to drop packets.

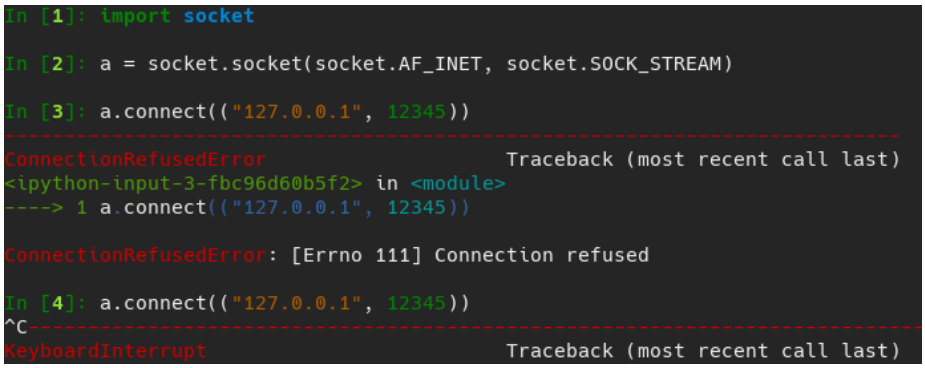

Figure 13: The code used to produce firewall packet dropping example in figure 12

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

a call to the underlying linux kernel's connect syscall but presenting a kinder call signature to the user as the Python socket library does some processing before the syscall is made.

1.3 Success Criteria

1. Probe another computer's networking from a black box perspective.
2. Send ICMP ECHO requests to determine whether a machine is active or not.
3. Translate Classless Inter-Domain Routing (CIDR) specified subnets into a list of domains.
4. Detect whether a TCP port is open (can be connected to).
5. Detect whether a TCP port is closed (will refuse connections).
6. Detect whether a TCP port is filtered (a firewall is preventing or monitoring access).
7. Detect whether a UDP port is open (can be connected to).
8. Detect whether a UDP port is closed (will refuse connections).
9. Detect whether a UDP port is filtered (a firewall is preventing or monitoring access).
10. Detect the operating system of another machine on the network solely from sending packets to the machine and interpreting the responses.
11. Detect what service is listening behind a port.
12. Detect the version of the service running behind a port.

1.4 Description of current system or existing solutions

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

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

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

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

```
# Nmap 7.70 scan initiated Wed Apr 10 19:36:18 2019 as:
    nmap -sC -sV -oA /home/tritoke/thing 192.168.1.0/24
Nmap scan report for router.asus.com (192.168.1.1)
Host is up (1.0s latency).
Not shown: 995 closed ports
PORT     STATE SERVICE    VERSION
53/tcp   open  domain     (generic dns response: NOTIMP)
| fingerprint-strings:
|   DNSVersionBindReqTCP:
|     version
```

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

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

Next `networkscan.gnmap`:

```
# Nmap 7.70 scan initiated Wed Apr 10 19:36:18 2019 as:
    nmap -sC -sV -oA /home/tritoke/networkscan 192.168.1.0/24
Host: 192.168.1.1 (router.asus.com) Status: Up
Host: 192.168.1.1 (router.asus.com) Ports: 53/open/tcp//domain//
      (generic dns response: NOTIMP)/, 80/open/tcp//http//ASUS WRT http admin/,
      515/open/tcp//printer///, 8443/open/tcp//ssl|http//ASUS WRT http admin/,
      9100/open/tcp//jetdirect?///  Ignored State: closed (995)
Host: 192.168.1.8 (android-25a97e36c2e74456)  Status: Up
Host: 192.168.1.8 (android-25a97e36c2e74456)  Ports: 5060/filtered/tcp//sip///
      Ignored State: closed (999)
```

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

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

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

Finally we have eXtensible Markup Language (XML) format:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE nmaprun>
<?xml-stylesheet href="file:///usr/bin/../share/nmap/nmap.xsl" type="text/xsl"?>
<!-- Nmap 7.70 scan initiated Wed Apr 10 19:36:18 2019 as: nmap -sC -sV -oA /home/tritoke/thing 192.168.1.0/24 -->
<nmaprun scanner="nmap" args="nmap -sC -sV -oA /home/tritoke/thing 192.168.1.0/24" start="1554921378" startstr="Wed Apr 10 19:36:18 2019" version="7.70" xmloutputversion="1.04">
<verbose level="0"/>
<debugging level="0"/>
<host starttime="1554921379" endtime="1554923187"><status state="up" reason="syn-ack" reason_ttl="0"/>
<address addr="192.168.1.1" addrtype="ipv4"/>
<hostnames>
<hostname name="router.asus.com" type="PTR"/>
</hostnames>
<ports><extraports state="closed" count="995">
<extrareasons reason="conn-refused" count="995"/>
</extraports>
<port protocol="tcp" portid="53"><state state="open" reason="syn-ack" reason_ttl="0"/><service name="domain" extrainfo="generic dns response: NOTIMP" servicefp="SF-Port53-TCP:V=7.70%I=7%D=4/10%Time=5CAE3DC5%P=x86_64
-pc-linux-gnu%r(DNSVersionBindReqTCP,20,&amp;quot;\0\x1e\0\x06\x85\x85\0
\x01\0\0\0\0\0\0\x07version\x04bind\0\0\x10\0\x03&amp;quot;)%r
```

```
19 (DNSStatusRequestTCP,E,&quot;\0\x0c\0\0\x90\x04\0\0\0\0\0\0\0\0&quot;);"
       method="probed" conf="10"/><script id="fingerprint-strings"
       output="&#xa; DNSVersionBindReqTCP: &#xa; version&#xa; bind"><elem
       key="DNSVersionBindReqTCP">&#xa; version&#xa; bind</elem>
20 </script></port>
```

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

In terms of where nmap lives in the software stack is that it is an application at level 7 when the user interacts with it but it uses several libraries which interact at level 2 which it uses to get the raw headers of the packets being sent and thus gain information from them.

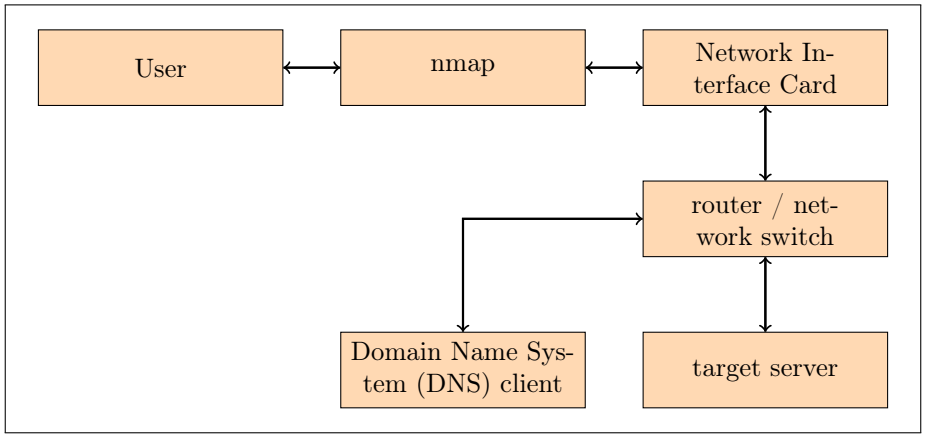

Figure 14: A block diagram showing how nmap sits in the software stack.

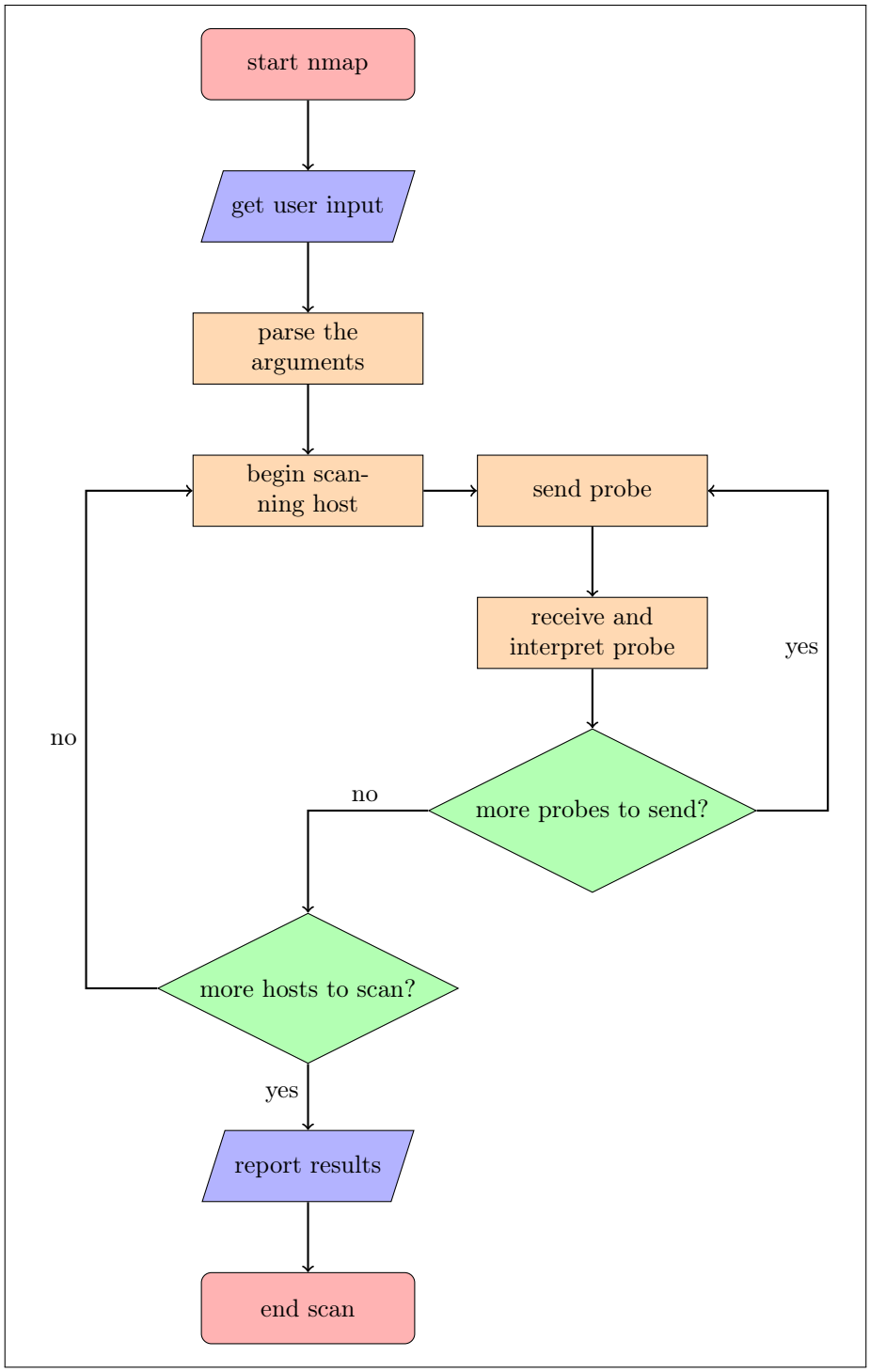

Figure 15: A flow chart showing how nmap does scanning.

1.5 Prospective Users

The prospective users of this system would be system administrators, penetration testers or network engineers. In my case my prospective users would be my school's system administrators and it would allow them to see an outsiders perspective on for example the server running the school's website page or to see if any of the programs on the servers were leaking information through banners etc. (most services send a banner with information like what protocol version they use and other information)

1.6 Data Dictionary

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

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

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

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

```
>>> with open("hosts", "r") as f:
...     hosts = f.read().splitlines()
...
>>> import sys
>>> sys.getsizeof(hosts)
2216
>>> ports = list(range(1000))
>>> sys.getsizeof(ports)
9112
>>> len(hosts)*sys.getsizeof(ports) / 2**10 # 2*10 is one kibibyte
2278.0
>>> sys.getsizeof(True)
28
```

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

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

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

1.7 Data Flow Diagram

This seems to be fairly relevant and to do with how data goes through my program i.e. going from the network to my port scanner into a target object and other scanners before version detection and finally displaying to the user. Make a flowchart for this.

https://en.wikipedia.org/wiki/Data-flow_diagram

1.8 Description of Solution Details

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

The first point of the success criteria that I wanted to get a feel for was receiving and sending ICMP ECHO requests aka pings. ICMP as a protocol sits at layer 3 of the OSI model this means it is a layer below what you are normally

give access to in the socket module. This means instead of getting a bytes object with just the data from the header you instead get a bytes object which contains the entire packet and you have to dissect it yourself to get the information out of it, this can be quite difficult if it weren't for the struct module. The struct module provides a convenient API for converting between packed values i.e. packets in network endianness to unpacked values i.e. a double representing the current time in local endianness. Interactions with the socket module are mainly through the pack and unpack functions. For each of these functions you provide a format specifier defining how to unpack/pack the bytes/values. In Listing 3 you can see an example of me using the struct.pack function to pack the values which comprise an ICMP ECHO REQUEST into a packet and sending it the localhost address (127.0.0.1). This program is effectively the complement to the program listed in listing 4 which uses struct.unpack to unpack value from the received ICMP packet before printing the fields out to the terminal. Listing 3 makes use of the IP checksum function which I wrote. In figure 16 you can see the output when I run the command `ping 127.0.0.1` which the code in figure4 is listening for packets.

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

```
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,
       socket.IPPROTO_ICMP)
19 # allows manual IP header creation
20 # ping_sock.setsockopt(socket.SOL_IP, socket.IP_HDRINCL, 1)
21
22 ID = os.getpid() & 0xFFFF
23
24 # the two zeros are the code and the dummy checksum, the one is the
       sequence number
25 dummy_header = struct.pack("bbHHh", ICMP_ECHO_REQUEST, 0, 0, ID, 1)
26
```

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

checksum = ip_utils.ip_checksum(dummy_header+data)

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

packet = header + data

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

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

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

import socket
import struct
import time
from typing import List

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

packets: List[bytes] = []

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

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

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

    print("IP header:")
```

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

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

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

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

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

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

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

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

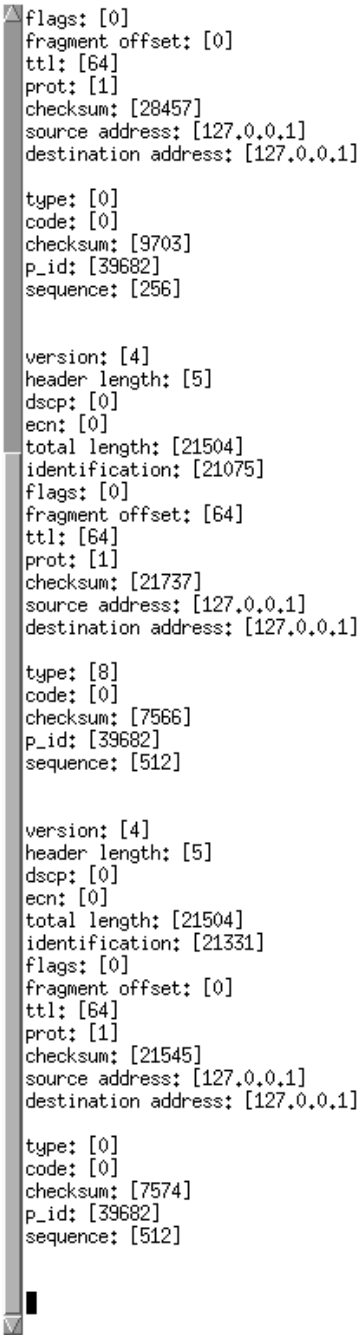

Figure 16: Dissecting an ICMP ECHO REQUEST packet.

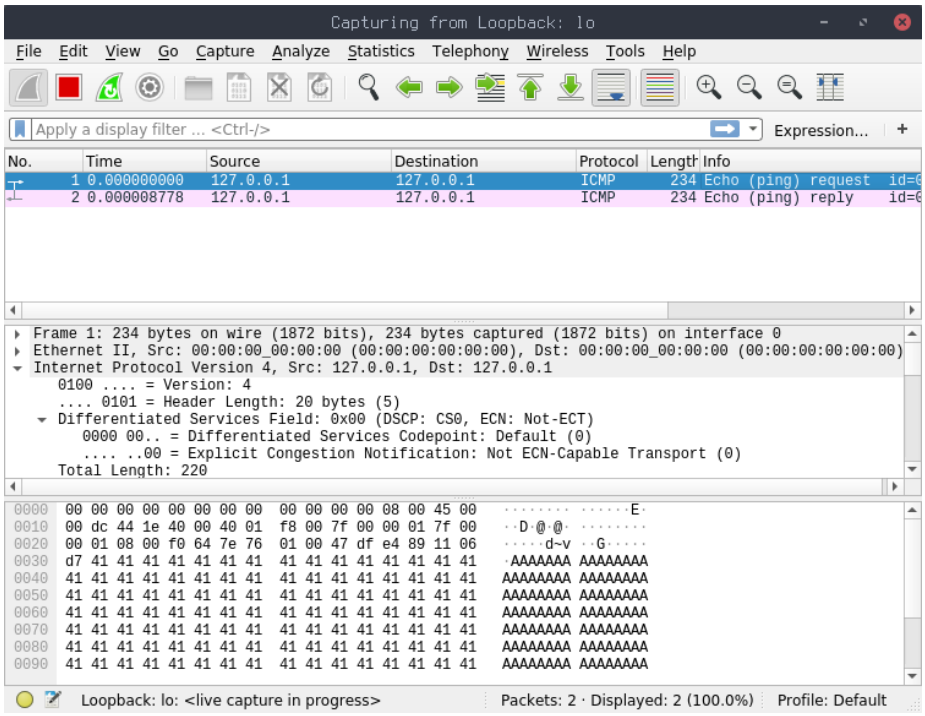

Figure 17: Screenshot of wireshark showing a successful send of an ICMP ECHO REQUEST packet.

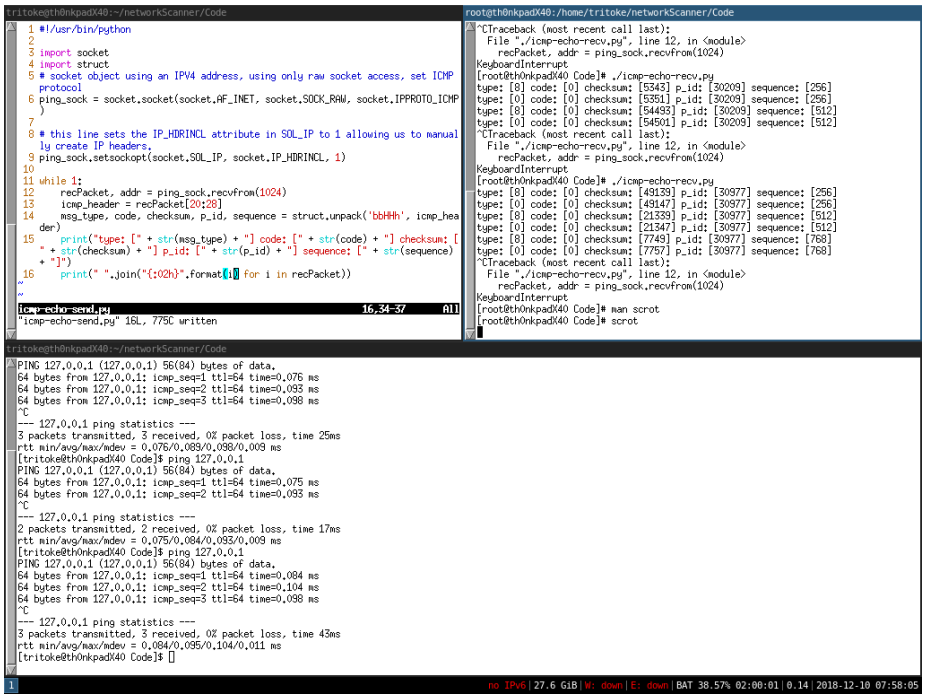

Figure 18: Screenshot showing me first successfully dissecting an ICMP ECHO REQUEST packet.

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

Listing 6: An attempt at making a ping scanner.

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

import ip_utils

import socket
```

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


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


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

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


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

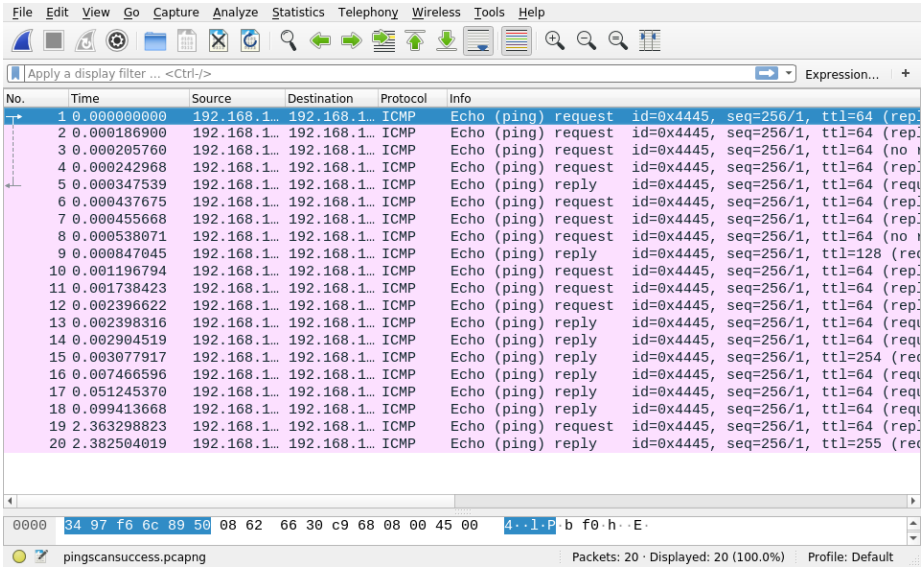

Figure 19: Screenshot of wireshark showing a successful ping scan.

Listing 7: The output of from the ping scanner on the run which generated the PCAP file in figure 19

```
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]
```

1.9 Acceptable Limitations

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

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

1.10 Test Strategy

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

1. Unit testing
2. Wireshark

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

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

2 Design

2.1 Overall System Design (High Level Overview)

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

- `Connect()`

- version
- listener / sender

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

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

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

2.2 Design of User Interfaces HCI

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

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

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

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

optional arguments:
  -h, --help            show this help message and exit
  -Pn                   assume hosts are up
  -sL                   list targets
  -sn                   disable port scanning
  -sS                   TCP SYN scan
  -sT                   TCP connect scan
```

```
-sU                   UDP scan
-sV                   version scan
-p PORTS, --ports PORTS
                      scan specified ports
--exclude_ports EXCLUDE_PORTS
                      ports to exclude from the scan
```

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

2.3 System Algorithms (Flowcharts)

When I have finished the first draft of the text bits I will add pictures / flowcharts

2.4 Input data Validation

My program takes very little input from the user which means that there is a very low chance of the program crashing due to user input error as the errors are detected All data which is entered is either parsed using a regular expression with the case of the ports directive (`-p`) or is run through checking functions like `ip_utils.is_valid_ip`. As well as using these checking functions whenever anIP addressis converted between "long form" and "dot form" which is used in every type of scanning.

2.5 Proposed Algorithms for complex structures (flow charts or Pseudo Code)

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

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

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

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

2.6 Design Data Dictionary

I have no idea what this means. All I can find is that it relates to database structure???

3 Technical Solution

3.1 Program Listing

3.2 Comments (Core)

3.3 Overview to direct the examiner to areas of complexity and explain design evidence

4 Testing

4.1 Test Plan

4.2 Test Table / Testing Evidence (Core: lots of screenshots)

5 Evaluation

5.1 Reflection on final outcome

5.2 Evaluation against objectives, end user feedback

5.3 Potential improvements

6 Appendices

You may show you program listing here
User feedback and survey data

Glossary

API Applications Programming Interface 3, 23

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... 22

black box Looking at something from an outsider's perspective knowing nothing about how it works internally. 1, 16

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. 17, 33

CIDR Classless Inter-Domain Routing 16, 22, 36

CPE Common Platform Enumeration 33

daemon A process that runs forever in the background to facilitate other programs. 2

dbus-daemon A daemon which enable a common interface for inter-process communication. 2

DHCP Dynamic Host Configuration Protocol 2, 3

DHCPCD Dynamic Host Configuration Protocol Client Daemon 2

DNS Domain Name System 20

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. 2

FTP File Transfer Protocol 3, 17

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. 4

HTML Hypertext Markup Language 4, 6

HTTP Hypertext transfer Protocol 3, 4, 5, 14

HTTPS Hypertext transfer Protocol Secure 14

ICMP Internet Control Message Protocol 15, 16, 23, 24, 25, 27, 28, 29

IDS Intrusion Detection System 17

IP Internet Protocol 29

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. 2, 4, 5, 14, 35, 36

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. 2

NIC Network Interface Card 2, 4

OSI model Open Systems Interconnection model 3, 23

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. 2, 3, 4, 5, 7, 9, 10, 14, 15, 17, 33, 34

PCAP Packet CAPture 32

PHP PHP Hypertext Processor 3

port Computers have "ports" for each protocol which can be connected to separately, this makes up part of a "socket" connection. 5, 16, 17, 34, 35, 36

port knocking Port knocking is where packets must be sent to a sequence of ports before access to the desired port is granted. 17

SCTP Stream Control Transmission Protocol 17

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. 2, 22

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. 2, 10, 17, 22, 33, 34

subnet A subnet is simply the sub-network of every possible IP address that will be used for communication on a particular network. 2, 36

systemd A daemon for controlling what is run when the system starts. 2

TCP Transmission Control Protocol 5, 10, 13, 14, 15, 16, 17, 29, 34

UDP User Datagram Protocol 5, 15, 17

upowerd Manages the power supplied to the system: charging, battery usage etc... 2

XML eXtensible Markup Language 19