Internet Performance and Troubleshooting Lab

Nmap

Report 4 - Individual **Group 9**

January 7, 2024 Salvatore Giarracca s314701

1 Introduction

1.1 Environment setup

The client used to run the following experiments is an HP laptop running **Linux Ubuntu 23.10**. In particular, the WiFi manufacturer is Intel and its logical name is **wlo1**. The nmap scans were performed inside a private Lan in which the provider is **FASTWEB**: it is an FTTH plan up to 1 Gb/s and the modem supports both 2.4 and 5 GHz network typologies.

Network card specifications

Current speed (Bit Rate) of **wlo1**	866.7 Mb/s
Current duplex status	Half (Wifi)
Enabled offloading capabilities	rx–checksumming, tx–checksumming, sg, tso, gso, gro, highdma, netns–local

Network layer configuration

IP address	192.168.1.144
Netmask	255.255.255.0
Default Gateway	192.168.1.254

Table 1: Client Overview

2 Hosts Discovery

In the first part of the experiment, I performed a scan of the entire network, aiming to discover all the hosts that are active/up in that moment. Six devices were found online (output in the bottom of the page).

2.1 Scan without root privileges

```
nmap 192.168.1.0/24 -sn
```

When host discovery scan is performed without root privileges, nmap uses the **connect()** system call, requesting to the operating system to send a **[SYN]** message to port 80 and 443 for each possible host in the network. Those message are sent in ranges of ip addresses and are ordered within the range. If the current processed host is known (present inside the ARP table), the tcp message is sent immediately, otherwise an **ARP request** is sent before, wishing that an **ARP reply** will be received. If so, the operating system sends the message to the freshly discovered host.

2.2 Scan with root privileges

```
sudo nmap 192.168.1.0/24 -sn
```

When host discovery is performed with root privileges, nmap has the rights to use raw sockets to send and capture packets at layer 3. Doing so, it can forge raw packets as it wants and receive them without modifications of any kind by other transport protocols. In this case, to discover the hosts up in the lan, that is the same lan in which the pc running nmap belongs to, exploiting **ARP protocol** is enough. Thus, nmap builds an ARP request for each possible host in the network, waiting for an ARP reply. If the latter arrives, it means that the host is up, then its ip address is saved to be shown in the final output, along with the resolved name of the host (if any). Infact, when all the online hosts are discovered, a DNS query is sent to the default gateway. Since nmap is able to read ARP replies, now also the **MAC** address of the discovered host is reported.

No sudo output

```
Nmap scan report for orangepi4-lts.lan (192.168.1.62)
Host is up (0.0042s latency).
Nmap scan report for dev.lan (192.168.1.144)
Host is up (0.00056s latency).
Nmap scan report for Google-Home-Mini.lan (192.168.1.161)
Host is up (0.0052s latency).
Nmap scan report for POCO-F3.lan (192.168.1.191)
Host is up (0.019s latency).
Nmap scan report for Chromecast.lan (192.168.1.223)
Host is up (0.0041s latency).
Nmap scan report for myfastgate.lan (192.168.1.254)
Host is up (0.0016s latency).
Nmap done: 256 IP addresses (6 hosts up) scanned in 6.32 seconds
```

Sudo output

```
Nmap scan report for orangepi4-lts.lan (192.168.1.62)
Host is up (0.0064s latency).
MAC Address: 26:BB:73:F4:94:B9 (Unknown)
Nmap scan report for Google-Home-Mini.lan (192.168.1.161)
Host is up (0.18s latency).
MAC Address: 7C:D9:5C:26:D4:9D (Google)
Nmap scan report for POCO-F3.lan (192.168.1.191)
Host is up (0.15s latency).
MAC Address: 46:10:3E:8B:75:44 (Unknown)
Nmap scan report for Chromecast.lan (192.168.1.223)
Host is up (0.072s latency).
MAC Address: 20:1F:3B:1E:76:17 (Google)
Nmap scan report for myfastgate.lan (192.168.1.254)
Host is up (0.0028s latency).
MAC Address: 20:B0:01:F7:ED:A4 (Technicolor Delivery Technologies Belgium NV)
Nmap scan report for dev.lan (192.168.1.144)
Host is up.
Nmap done: 256 IP addresses (6 hosts up) scanned in 3.36 seconds
```

3 Port Scanning

I selected two hosts within my network and performed TCP/UDP port scanning being an unprivileged and privileged user; The first host is my server, an **Orange Pi 4 LTS** (similar to a Raspberry Pi) with ip address **192.168.1.62** because there are multiple services that runs on it (wireguard VPN, pihole, ecc); The second host is my **Google Chromecast** with ip address **192.168.1.223**. The two hosts have been scanned sequentially in order to generate a cleaner wireshark capture and plots. During the experiment i noticed that nmap did not scan all the possible ports, but only the most common 1000 ones, and in a **random** order. This is true either for TCP and UDP but, there was a substantial difference between **how many times** every port has been scanned. In particular, for TCP some ports were scanned 2 or 3 times (Figure 11); UDP scan, instead, probes ports up to 91 times (Figure 12).

TCP

open	an app is listening, send responses
closed	no apps are listening, send responses
filtered	cannot determine if open, no responses
unfiltered	cannot determine if open or closed, but accessible
open | filtered	cannot determine if open or filtered, no responses
closed | filtered	cannot determine if closed or filtered

UDP

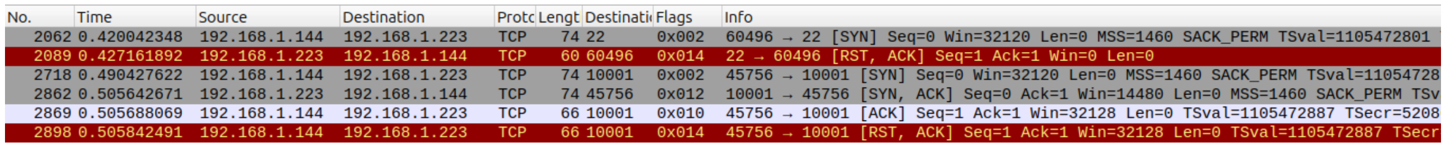
open	any UDP responses from a target port
open | filtered	no responses received
closed	ICMP port unreachable error (type 3, code 3)
filtered	ICMP unreachable error (type 3, code [1|2|9|10|13])

Table 2: Port Status Legend

3.1 TCP – scan without root privileges

```
nmap 192.168.1.62 > no_sudo_output ; nmap 192.168.1.223 >> no_sudo_output
```

Since nmap does not have root privileges, the default option when port scanning is performed is the **TCP Connect Scan** (–sT) (figure 7). This one and the TCP FTP Bounce Scan are the only ports scans that can be performed by unprivileged users. As previously mentioned in the unprivileged host discovery section, nmap make use of **connect()** syscall, offered by Berkeley Socket API, to initiate a 3-way handshake with the target host on that specific port. This API let the OS to allocate resources, necessary to accomodate the incoming data. Due to the fact that nmap does not have complete control over such system call, every connection to an open port ends with at least 4 messages exchanged, because after the **SYN** request, the port responds with a **SYN/ACK** and after that, the operating system is obliged to ACKnowledge such message. When the handshake is completed nmap close the connection with a **RST/ACK** message, without involving a **FIN** flag that would result in another handshake.



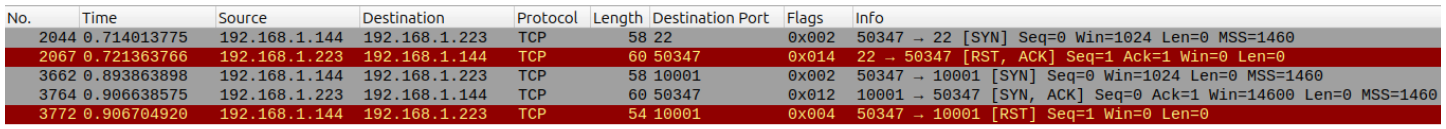
```
No.  Time        Source         Destination    Protc Lengt Destinati Flags Info
2062 0.420042348 192.168.1.144  192.168.1.223  TCP   74 22      0x002  60496 → 22 [SYN] Seq=0 Win=32120 Len=0 MSS=1460 SACK_PERM TSval=1105472801
2089 0.427161892 192.168.1.223  192.168.1.144  TCP   60 60496   0x014  22 → 60496 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
2718 0.490427622 192.168.1.144  192.168.1.223  TCP   74 10001   0x002  45756 → 10001 [SYN] Seq=0 Win=32120 Len=0 MSS=1460 SACK_PERM TSval=11054728
2862 0.505642671 192.168.1.223  192.168.1.144  TCP   74 45756   0x012  10001 → 45756 [SYN, ACK] Seq=0 Ack=1 Win=14480 Len=0 MSS=1460 SACK_PERM TSv
2869 0.505688069 192.168.1.144  192.168.1.223  TCP   66 10001   0x010  45756 → 10001 [ACK] Seq=1 Ack=1 Win=32128 Len=0 TSval=1105472887 TSecr=5208
2898 0.505842491 192.168.1.144  192.168.1.223  TCP   66 10001   0x014  45756 → 10001 [RST, ACK] Seq=1 Ack=1 Win=32128 Len=0 TSval=1105472887 TSecr
```

Figure 1: No sudo Open vs Closed port

3.2 TCP – scan with root privileges

```
sudo nmap 192.168.1.62 > sudo_output ; sudo nmap 192.168.1.223 >> sudo_output
```

Now that nmap has gained the privileges to use raw sockets, it performs a so called **TCP SYN (Stealth) Scan** (figure 8) by default (–sS). This way to operate is also called **half-open scanning** because every initiated connection is never completed. When a **SYN** message is sent to an open port, the host responds with a **SYN/ACK** because it want to complete the connection and exchange data. At this point the default behaviour should be to send back an **ACK** to the target host, but instead a **RST** message is sent, closing brutally the connection. This happens due to the ability of nmap to forge a raw SYN packet: without using **connect()**, the OS does not keep track of the previously opened connection, so that received message is unexpected. Thus, it sends a **RST** message (that is not an **ACK** this time) and close the connection. It is also possible to notice that now the **source ports** are not "random" but always the same. This is a different behaviour compared to the scan without root: here nmap decides the port to use, while the choice is based on the operating system policies in the unprivileged scan.



No.	Time	Source	Destination	Protocol	Length	Destination Port	Flags	Info
2044	0.714013775	192.168.1.144	192.168.1.223	TCP	58	22	0x002	50347 → 22 [SYN] Seq=0 Win=1024 Len=0 MSS=1460
2067	0.721363766	192.168.1.223	192.168.1.144	TCP	60	50347	0x014	22 → 50347 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
3662	0.893863898	192.168.1.144	192.168.1.223	TCP	58	10001	0x002	50347 → 10001 [SYN] Seq=0 Win=1024 Len=0 MSS=1460
3764	0.906638575	192.168.1.223	192.168.1.144	TCP	60	50347	0x012	10001 → 50347 [SYN, ACK] Seq=0 Ack=1 Win=14600 Len=0 MSS=1460
3772	0.906704920	192.168.1.144	192.168.1.223	TCP	54	10001	0x004	50347 → 10001 [RST] Seq=1 Win=0 Len=0

Figure 2: No sudo Open vs Closed port

3.3 UDP – scan without root privileges

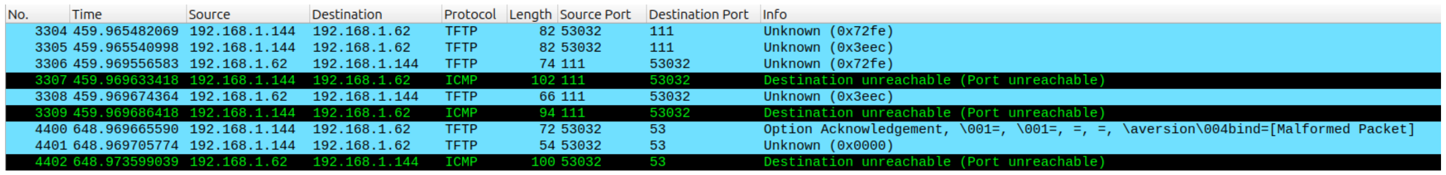
```
nmap 192.168.1.62 -sU > no_sudo_output ; nmap 192.168.1.223 -sU >> no_sudo_output
```

Unfortunately, the previous command does not perform any scan because an UDP scan requires root privileges. Due to the nature of UDP to be "best effort" protocol, it will not inform the sender if a packet gets dropped. Closed ports could decides to reply to an UDP probe with an **ICMP port unreachable** message, so nmap should have the privileges to read such messages otherwise the system will drop them.

3.4 UDP – scan with root privileges

```
sudo nmap 192.168.1.62 -sU > sudo_output ; sudo nmap 192.168.1.223 -sU >> sudo_output
```

When nmap performs an UDP scan (figure 9), it sends empty UDP probes (with an empty payload) to the most popular 1000 ports and wait for a response. A port is interpreted open if an UDP response come back from the target. On the contrary, if nmap does not detect any response, even after some retransmissions, it will label the port as **open|filtered**, maybe due to a network firewall in between. Closed and filtered port are recognized by exploiting ICMP errors sent from them as shown in Table 2. Those two are reason besides the long time required by perfoming an UDP scan. In particular this is due to the implementation used target-side. Infact most **UNIX based systems** limit ICMP port unreachable error rate to **1 every second**, causing the duration of the experiments lasts roughly 17 minutes using a default time template (–T3). It is possible to speed up the scan by using –T4 (aggressive) or even –T5 (insane) time templates given by nmap, but those options reduce drastically the **–max–rtt–timeout** (along with other parameters), leading the result of the scan to be less precise and reliable. In the figure 3 it is shown the packet trace of a **closed** port (**53**), that reply always with ICMP port unreachable, and an **open** port (**111**) that reply two times with an UDP message. One interesting thing could be the presence of ICMP destination unreachable errors after the UDP response from the target (figure 3). There could be two possible reasons for this behaviour: the first could be that that nmap send this message as he would have done with TCP with a **RST** packet; the second reason (imho more credited because UDP is connectionless) could be that the client OS does not expect a message from the (open) port **111**, so it replies with such message.



No.	Time	Source	Destination	Protocol	Length	Source Port	Destination Port	Info
3304	459.965482069	192.168.1.144	192.168.1.62	TFTP	82	53032	111	Unknown (0x72fe)
3305	459.965540998	192.168.1.144	192.168.1.62	TFTP	82	53032	111	Unknown (0x3eec)
3306	459.969556583	192.168.1.62	192.168.1.144	TFTP	74	111	53032	Unknown (0x72fe)
3307	459.969633418	192.168.1.144	192.168.1.62	ICMP	102	111	53032	Destination unreachable (Port unreachable)
3308	459.969674364	192.168.1.62	192.168.1.144	TFTP	66	111	53032	Unknown (0x3eec)
3309	459.969686418	192.168.1.144	192.168.1.62	ICMP	94	111	53032	Destination unreachable (Port unreachable)
4400	648.969665590	192.168.1.144	192.168.1.62	TFTP	72	53032	53	Option Acknowledgement, \001=, \001=, =, =, \aversion\004bind=[Malformed Packet]
4401	648.969705774	192.168.1.144	192.168.1.62	TFTP	54	53032	53	Unknown (0x0000)
4402	648.973599039	192.168.1.62	192.168.1.144	ICMP	100	53032	53	Destination unreachable (Port unreachable)

Figure 3: UDP Scan

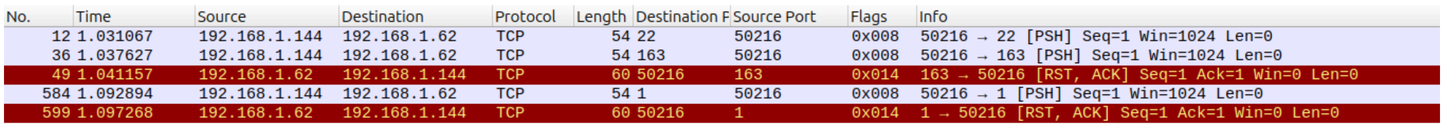
4 Evaluating TCP scan types

I selected my **Orange Pi 4 LTS** server to perform those types of scan. On both of them i voluntarely added the **–max–retries 0** option to make the plots appear less concentrated, in order to better appreciate the results. However this option make less reliable the results of port scanning, leading in to mark some of them into a status in which they aren't. Nmap output is shown in figure 10

4.1 Stealt Scan (–sS) with PSH flag set

```
sudo nmap 192.168.1.62 -sS --max-retries 0 --scanflags PSH -p 1-300
```

Here a custom scan was performed, more precisely a **PSH Scan** (figure 6a). I chose –sS option to make nmap interprets the results as Stealth scan, then with the help of –scanflag option, the **PSH** flag is set to each TCP probe sent to the target. This type of scan could determine only if a port is filtered or unfiltered, because nmap cannot do any assumptions due to the lack of response by the target.



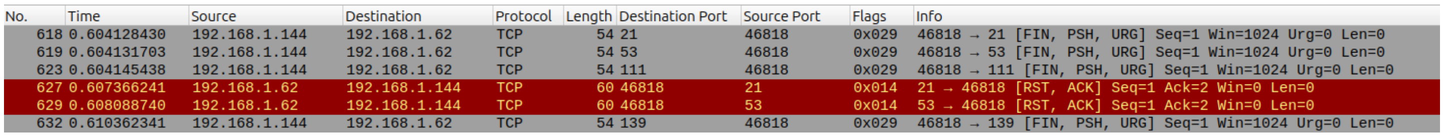
No.	Time	Source	Destination	Protocol	Length	Destination P	Source Port	Flags	Info
12	1.031067	192.168.1.144	192.168.1.62	TCP	54	22	50216	0x008	50216 → 22 [PSH] Seq=1 Win=1024 Len=0
36	1.037627	192.168.1.144	192.168.1.62	TCP	54	163	50216	0x008	50216 → 163 [PSH] Seq=1 Win=1024 Len=0
49	1.041157	192.168.1.62	192.168.1.144	TCP	60	50216	163	0x014	163 → 50216 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
584	1.092894	192.168.1.144	192.168.1.62	TCP	54	1	50216	0x008	50216 → 1 [PSH] Seq=1 Win=1024 Len=0
599	1.097268	192.168.1.62	192.168.1.144	TCP	60	50216	1	0x014	1 → 50216 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0

Figure 4: PSH Scan

4.2 Christmas Scan (–sX)

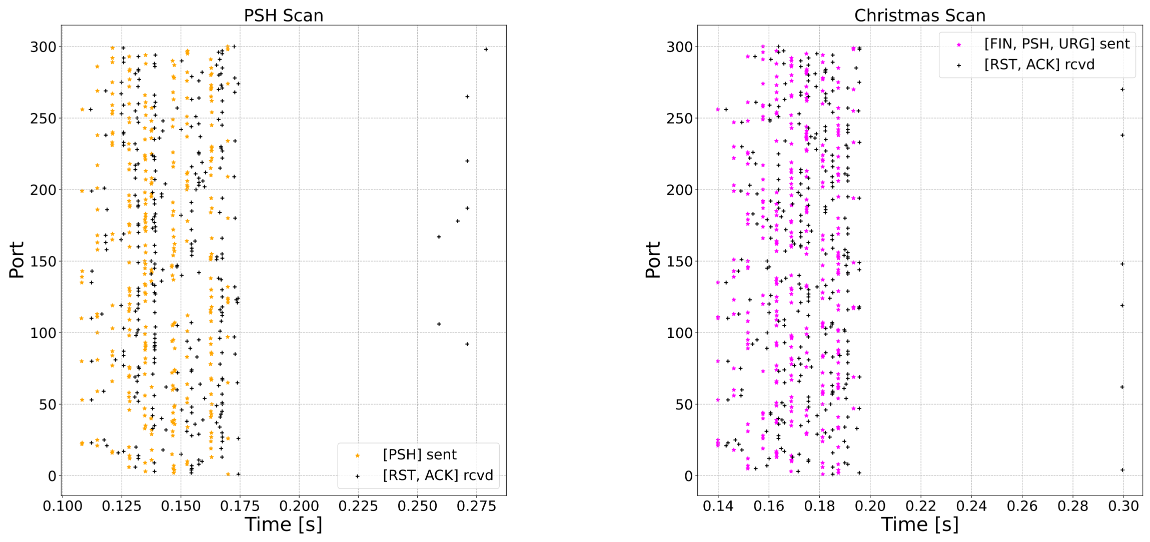
```
sudo nmap 192.168.1.62 -sX --max-retries 0 -p 1-300
```

A Christmas Scan (figure 6b) exploits some "rules" written in the TCP RFC to understand if a port is open or closed. According to RFC, a closed port would respond with a **RST** to a message sent with at least one between **SYN**, **ACK** and **RST** flag set, otherwise will not respond at all, meaning that the port is open. When an Xmas scan is performed, nmap sets all the **FIN**, **PSH** and **URG** flags (hence the name of the scan) of the probe packets.



No.	Time	Source	Destination	Protocol	Length	Destination Port	Source Port	Flags	Info
618	0.604128430	192.168.1.144	192.168.1.62	TCP	54	21	46818	0x029	46818 → 21 [FIN, PSH, URG] Seq=1 Win=1024 Urg=0 Len=0
619	0.604131703	192.168.1.144	192.168.1.62	TCP	54	53	46818	0x029	46818 → 53 [FIN, PSH, URG] Seq=1 Win=1024 Urg=0 Len=0
623	0.604145438	192.168.1.144	192.168.1.62	TCP	54	111	46818	0x029	46818 → 111 [FIN, PSH, URG] Seq=1 Win=1024 Urg=0 Len=0
627	0.607366241	192.168.1.62	192.168.1.144	TCP	60	46818	21	0x014	21 → 46818 [RST, ACK] Seq=1 Ack=2 Win=0 Len=0
629	0.608088740	192.168.1.62	192.168.1.144	TCP	60	46818	53	0x014	53 → 46818 [RST, ACK] Seq=1 Ack=2 Win=0 Len=0
632	0.610362341	192.168.1.144	192.168.1.62	TCP	54	139	46818	0x029	46818 → 139 [FIN, PSH, URG] Seq=1 Win=1024 Urg=0 Len=0

Figure 5: Xmas Scan



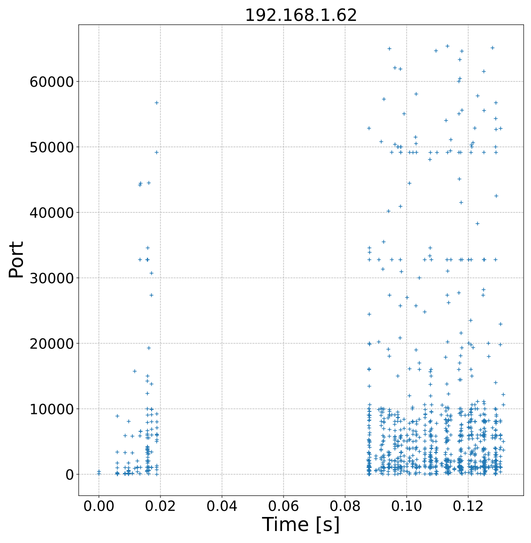
(a) TCP Push Scan

(b) TCP Christmas Scan

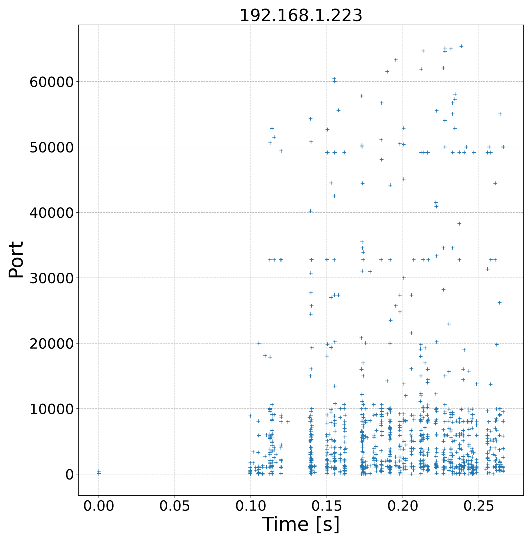
Figure 6: PSH and Xmas comparison

Appendix

TCP – scan without root privileges



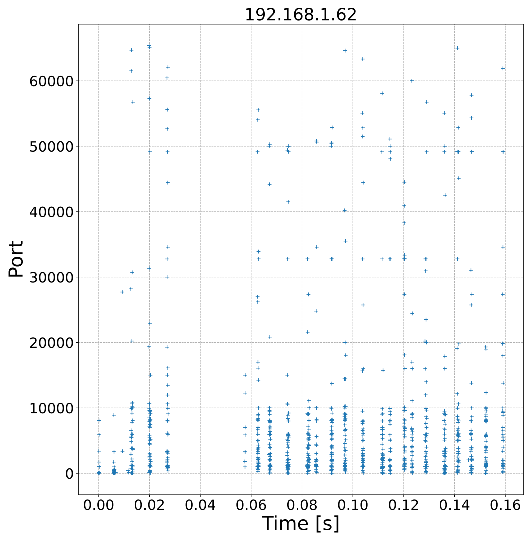
(a) Orange Pi 4 LTS



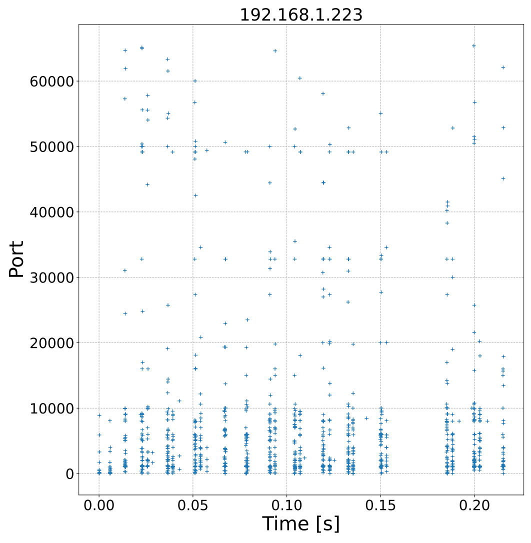
(b) Chromecast

Figure 7: TCP port scanning

TCP – scan with root privileges



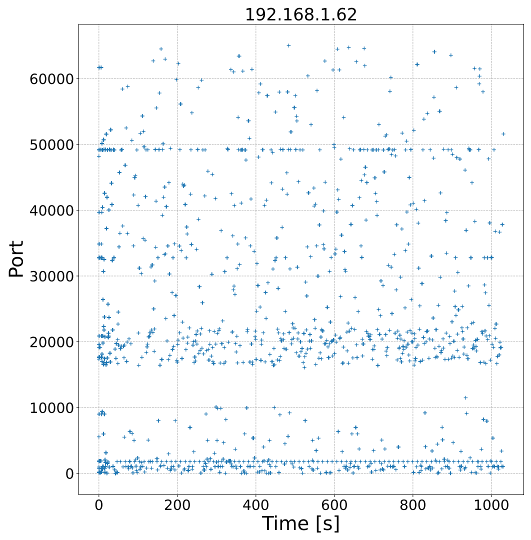
(a) Orange Pi 4 LTS



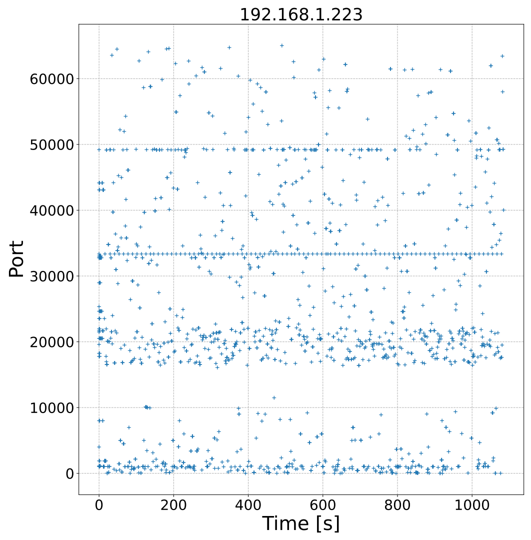
(b) Chromecast

Figure 8: TCP port scanning

UDP – scan without root privileges



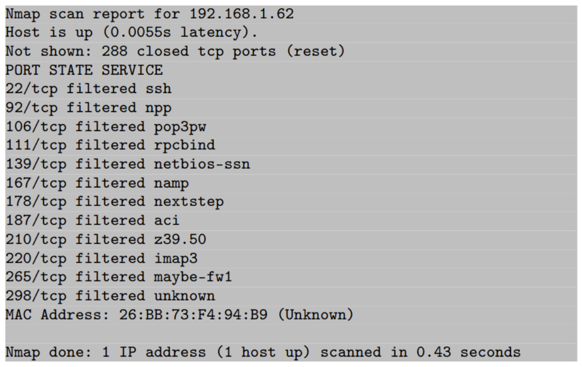
(a) Orange Pi 4 LTS



(b) Chromecast

Figure 9: UDP port scanning

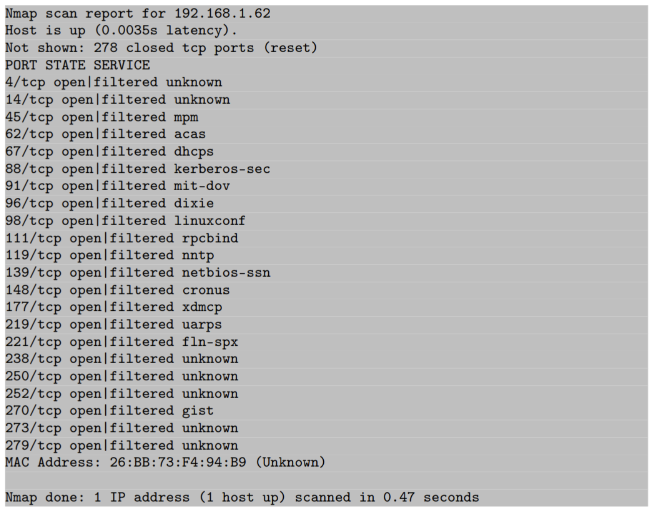
Custom scans output



```
Nmap scan report for 192.168.1.62
Host is up (0.0055s latency).
Not shown: 288 closed tcp ports (reset)
PORT STATE SERVICE
22/tcp filtered ssh
92/tcp filtered npp
106/tcp filtered pop3pw
111/tcp filtered rpcbind
139/tcp filtered netbios-ssn
167/tcp filtered namp
178/tcp filtered nextstep
187/tcp filtered aci
210/tcp filtered z39.50
220/tcp filtered imap3
265/tcp filtered maybe-fw1
298/tcp filtered unknown
MAC Address: 26:BB:73:F4:94:B9 (Unknown)

Nmap done: 1 IP address (1 host up) scanned in 0.43 seconds
```

(a) PSH Scan



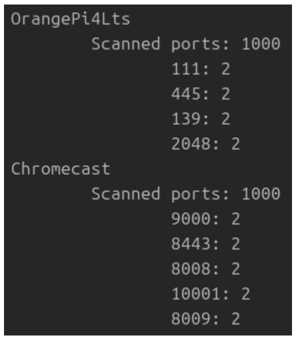
```
Nmap scan report for 192.168.1.62
Host is up (0.0035s latency).
Not shown: 278 closed tcp ports (reset)
PORT STATE SERVICE
4/tcp open|filtered unknown
14/tcp open|filtered unknown
45/tcp open|filtered mpm
62/tcp open|filtered acas
67/tcp open|filtered dhcps
88/tcp open|filtered kerberos-sec
91/tcp open|filtered mit-dov
96/tcp open|filtered dixie
98/tcp open|filtered linuxconf
111/tcp open|filtered rpcbind
119/tcp open|filtered nntp
139/tcp open|filtered netbios-ssn
148/tcp open|filtered cronus
177/tcp open|filtered xdmcp
219/tcp open|filtered uarps
221/tcp open|filtered fln-spx
238/tcp open|filtered unknown
250/tcp open|filtered unknown
252/tcp open|filtered unknown
270/tcp open|filtered gist
273/tcp open|filtered unknown
279/tcp open|filtered unknown
MAC Address: 26:BB:73:F4:94:B9 (Unknown)

Nmap done: 1 IP address (1 host up) scanned in 0.47 seconds
```

(b) Xmas Scan

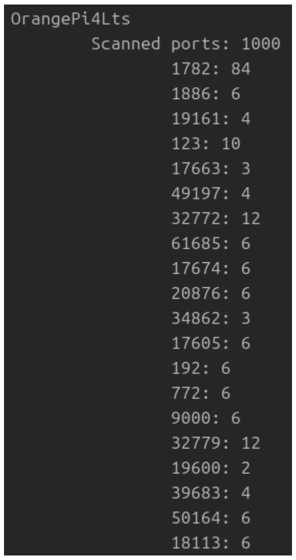
Figure 10: PSH and Xmas output

Ports scanned multiple times



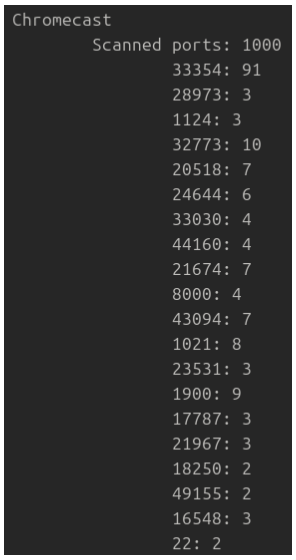
```
OrangePi4Lts
        Scanned ports: 1000
                111: 2
                445: 2
                139: 2
                2048: 2
Chromecast
        Scanned ports: 1000
                9000: 2
                8443: 2
                8008: 2
                10001: 2
                8009: 2
```

Figure 11: TCP scan



```
OrangePi4Lts
        Scanned ports: 1000
                1782: 84
                1886: 6
                19161: 4
                123: 10
                17663: 3
                49197: 4
                32772: 12
                61685: 6
                17674: 6
                20876: 6
                34862: 3
                17605: 6
                192: 6
                772: 6
                9000: 6
                32779: 12
                19600: 2
                39683: 4
                50164: 6
                18113: 6
```

(a) 192.168.1.62



```
Chromecast
        Scanned ports: 1000
                33354: 91
                28973: 3
                1124: 3
                32773: 10
                20518: 7
                24644: 6
                33030: 4
                44160: 4
                21674: 7
                8000: 4
                43094: 7
                1021: 8
                23531: 3
                1900: 9
                17787: 3
                21967: 3
                18250: 2
                49155: 2
                16548: 3
                22: 2
```

(b) 192.168.1.223

Figure 12: UDP scan

TCP script

```
import csv
from matplotlib import pyplot as plt
from collections import Counter
import sys
opi_csv =sys.argv[1]
chromecast_csv =sys.argv[2]

with open(opi_csv, "r", newline="") as opi_in, open(
    "sudo_opi_time-port", "w", newline=""
) as opi_out:
    reader =csv.reader(opi_in)
    writer =csv.writer(opi_out)
    for row in reader:
        writer.writerow([row[1], row[6]])
opi_in.close()
opi_out.close()
with open(chromecast_csv, "r", newline="") as chromecast_in, open(
    "sudo_chromecast_time-port", "w", newline=""
) as chromecast_out:
    reader =csv.reader(chromecast_in)
    writer =csv.writer(chromecast_out)
    for row in reader:
        writer.writerow([row[1], row[6]])
chromecast_in.close()
chromecast_out.close()

opi_distinct_ports ={}
with open("sudo_opi_time-port", "r") as file:
    data =file.readlines()
    x1 =[float(row.split(",")[0]) for row in data]
    y1 =[int(row.split(",")[1]) for row in data]
    opi_distinct_ports =dict(Counter(y1))
file.close()
chromecast_distinct_ports ={}
with open("sudo_chromecast_time-port", "r") as file:
    data =file.readlines()
    x2 =[float(row.split(",")[0]) for row in data]
    y2 =[int(row.split(",")[1]) for row in data]
    chromecast_distinct_ports =dict(Counter(y2))
file.close()

# statistics
with open("sudo_stats", "w") as stats:
    stats.write("OrangePi4Lts\n")
    stats.write(f"\tScanned ports: {len(opi_distinct_ports)}\n")
    for key, value in opi_distinct_ports.items():
        if value >1:
            stats.write("\t\t" +str(key) +": " +str(value) +"\n")
    stats.write("Chromecast\n")
    stats.write(f"\tScanned ports: {len(chromecast_distinct_ports)}\n")
    for key, value in chromecast_distinct_ports.items():
        if value >1:
            stats.write("\t\t" +str(key) +": " +str(value) +"\n")
plt.figure(1)
plt.rc("font", size=22)
plt.rc("axes", labelsize=30)
plt.plot(x1, y1, marker="+", linestyle="")
plt.xlabel("Time [s]")
plt.ylabel("Port")
plt.title("192.168.1.62")
plt.grid(linestyle="--")
plt.figure(2)
plt.plot(x2, y2, marker="+", linestyle="")
plt.xlabel("Time [s]")
plt.ylabel("Port")
plt.title("192.168.1.223")
plt.grid(linestyle="--")
plt.show()
```

UDP script

```
import csv
from matplotlib import pyplot as plt
from collections import Counter
import sys
opi_csv =sys.argv[1]
chromecast_csv =sys.argv[2]

with open(opi_csv, "r", newline="") as opi_in, open(
    "sudo_opi_time-port", "w", newline=""
) as opi_out:
    reader =csv.reader(opi_in)
    writer =csv.writer(opi_out)
    for row in reader:
        writer.writerow([row[1], row[6]])
opi_in.close()
opi_out.close()
with open(chromecast_csv, "r", newline="") as chromecast_in, open(
    "sudo_chromecast_time-port", "w", newline=""
) as chromecast_out:
    reader =csv.reader(chromecast_in)
    writer =csv.writer(chromecast_out)
    for row in reader:
        writer.writerow([row[1], row[6]])
chromecast_in.close()
chromecast_out.close()

opi_distinct_ports ={}
with open("sudo_opi_time-port", "r") as file:
    data =file.readlines()
    x1 =[float(row.split(",")[0]) for row in data]
    y1 =[int(row.split(",")[1]) for row in data]
    opi_distinct_ports =dict(Counter(y1))
file.close()
chromecast_distinct_ports ={}
with open("sudo_chromecast_time-port", "r") as file:
    data =file.readlines()
    x2 =[float(row.split(",")[0]) for row in data]
    y2 =[int(row.split(",")[1]) for row in data]
    chromecast_distinct_ports =dict(Counter(y2))
file.close()

# statistics
with open("sudo_stats", "w") as stats:
    stats.write("OrangePi4Lts\n")
    stats.write(f"\tScanned ports: {len(opi_distinct_ports)}\n")
    for key, value in opi_distinct_ports.items():
        if value >1:
            stats.write("\t\t" +str(key) +": " +str(value) +"\n")
    stats.write("Chromecast\n")
    stats.write(f"\tScanned ports: {len(chromecast_distinct_ports)}\n")
    for key, value in chromecast_distinct_ports.items():
        if value >1:
            stats.write("\t\t" +str(key) +": " +str(value) +"\n")
plt.figure(1)
plt.rc("font", size=22)
plt.rc("axes", labelsize=30)
plt.plot(x1, y1, marker="+", linestyle="")
plt.xlabel("Time [s]")
plt.ylabel("Port")
plt.title("192.168.1.62")
plt.grid(linestyle="--")
plt.figure(2)
plt.plot(x2, y2, marker="+", linestyle="")
plt.xlabel("Time [s]")
plt.ylabel("Port")
plt.title("192.168.1.223")
plt.grid(linestyle="--")
plt.show()
```

Custom scans script

```
import csv
from matplotlib import pyplot as plt
from collections import Counter
import sys
import pandas as pd

stealth_csv =sys.argv[1]
xmas_csv =sys.argv[2]

flags ={
    "FIN": 0x001,
    "SYN": 0x002,
    "RST": 0x004,
    "PSH": 0x008,
    "ACK": 0x010,
    "URG": 0x020,
}

flags_color ={
    0x002: "green",
    0x004: "red",
    0x008: "orange",
    0x012: "yellow",
    0x014: "black",
    0x029: "magenta",
}
def_color ="grey"

with open(stealth_csv, "r", newline="") as stealth_in, open(
    "stealth_time-port", "w", newline=""
) as stealth_out:
    reader =csv.reader(stealth_in)
    writer =csv.writer(stealth_out)
    for row in reader:
        direction ="sent"
        if row[2] =="192.168.1.62":
            direction ="rcvd"
            writer.writerow([row[1], row[7], direction, row[8]])
            continue
        writer.writerow([row[1], row[6], direction, row[8]])
stealth_in.close()
stealth_out.close()
stealth_distinct_ports ={}
with open("stealth_time-port", "r") as file:
    data =file.readlines()
    x1 =[float(row.split(",")[0]) for row in data]
    y1 =[int(row.split(",")[1]) for row in data]
    stealth_distinct_ports =dict(Counter(y1))
file.close()

with open(xmas_csv, "r", newline="") as xmas_in, open(
    "xmas_time-port", "w", newline=""
) as xmas_out:
    reader =csv.reader(xmas_in)
    writer =csv.writer(xmas_out)
    for row in reader:
        direction ="sent"
        if row[2] =="192.168.1.62":
            direction ="rcvd"
            writer.writerow([row[1], row[7], direction, row[8]])
            continue
        writer.writerow([row[1], row[6], direction, row[8]])
xmas_in.close()
xmas_out.close()
xmas_distinct_ports ={}
with open("xmas_time-port", "r") as file:
    data =file.readlines()
    x2 =[float(row.split(",")[0]) for row in data]
    y2 =[int(row.split(",")[1]) for row in data]
    xmas_distinct_ports =dict(Counter(y2))
file.close()
```

```python
# statistics
with open("stats", "w") as stats:
    stats.write("Stealth Scan\n")
    stats.write(f"\tScanned ports: {len(stealth_distinct_ports)}\n")
    for key, value in stealth_distinct_ports.items():
        if value >1:
            stats.write("\t\t" +str(key) +": " +str(value) +"\n")
    stats.write("Christmas Scan\n")
    stats.write(f"\tScanned ports: {len(xmas_distinct_ports)}\n")
    for key, value in xmas_distinct_ports.items():
        if value >1:
            stats.write("\t\t" +str(key) +": " +str(value) +"\n")

df1 =pd.read_csv(
    "stealth_time-port", header=None, names=["time", "port", "direction", "flags"]
)
df1["flags"] =df1["flags"].apply(lambda x: int(x, 16))
sent =df1[df1["direction"] =="sent"]
rcvd =df1[df1["direction"] =="rcvd"]
plt.figure(1)
plt.rc("font", size=22)
plt.rc("axes", labelsize=30)
plt.xlabel("Time [s]")
plt.ylabel("Port")
plt.title("PSH Scan")
plt.grid(linestyle="--")
plt.scatter(
    sent["time"],
    sent["port"],
    c=sent["flags"].map(flags_color).fillna(def_color),
    marker="*",
)
plt.scatter(
    rcvd["time"],
    rcvd["port"],
    c=rcvd["flags"].map(flags_color).fillna(def_color),
    marker="+",
)
plt.legend(["[PSH] sent", "[RST, ACK] rcvd"])

df2 =pd.read_csv(
    "xmas_time-port", header=None, names=["time", "port", "direction", "flags"]
)
df2["flags"] =df2["flags"].apply(lambda x: int(x, 16))
sent =df2[df2["direction"] =="sent"]
rcvd =df2[df2["direction"] =="rcvd"]
plt.figure(2)
plt.xlabel("Time [s]")
plt.ylabel("Port")
plt.title("Christmas Scan")
plt.grid(linestyle="--")
plt.scatter(
    sent["time"],
    sent["port"],
    c=sent["flags"].map(flags_color).fillna(def_color),
    marker="*",
)
plt.scatter(
    rcvd["time"],
    rcvd["port"],
    c=rcvd["flags"].map(flags_color).fillna(def_color),
    marker="+",
)
plt.legend(["[FIN, PSH, URG] sent", "[RST, ACK]"])
plt.show()
```