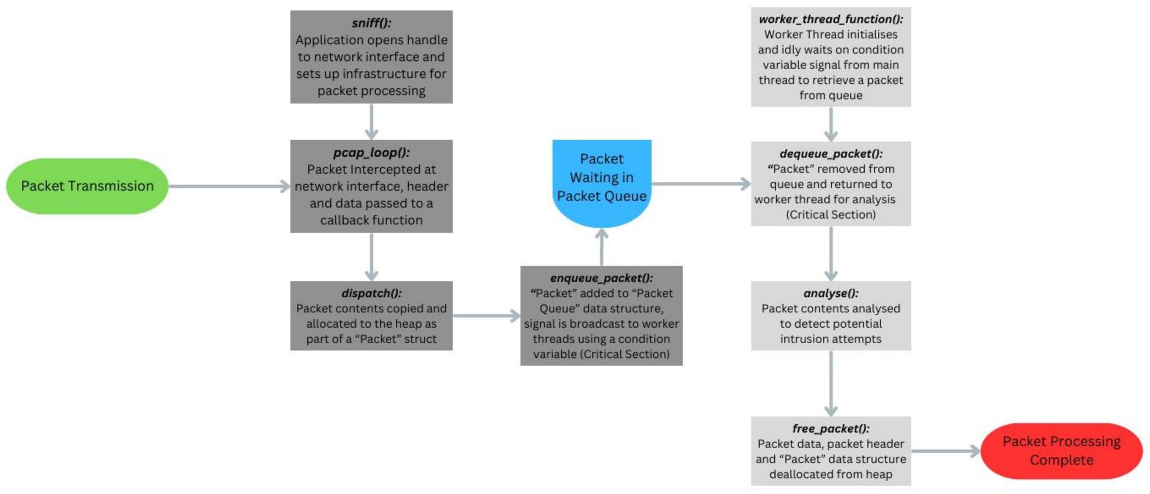
CS241 Coursework Report – 2008321

Contents:

1 – Implementation

1.1 – Design

The application's runtime is split into three main phases: setup, packet processing and cleanup. In the setup phase, data structures (a queue for packets, and a binary tree for unique Ipv4 addresses) are initialised and allocated onto the heap, and the pool of worker threads is created. The threads are initialised to wait idly on a condition variable signal from the main thread. During packet processing, the main thread continually executes *pcap_loop()* [1]. Packets are intercepted at the network interface, each triggering the callback function *dispatch()*. In a thread-safe manner, this adds a packet to the queue and signals the condition variable. A freshly awoken worker thread then dequeues and processes the packet. The final phase of runtime serves to deallocate all memory allocated to the heap, and to join all worker threads, collating the statistics from each to produce a final report to the user.



sniff(): Application opens handle to network interface and sets up infrastructure for packet processing
Packet Transmission
pcap_loop(): Packet Intercepted at network interface, header and data passed to a callback function
dispatch(): Packet contents copied and allocated to the heap as part of a "Packet" struct
enqueue_packet(): "Packet" added to "Packet Queue" data structure, signal is broadcast to worker threads using a condition variable (Critical Section)
Packet Waiting in Packet Queue
worker_thread_function(): Worker Thread initialises and idly waits on condition variable signal from main thread to retrieve a packet from queue
dequeue_packet(): "Packet" removed from queue and returned to worker thread for analysis (Critical Section)
analyse(): Packet contents analysed to detect potential intrusion attempts
free_packet(): Packet data, packet header and "Packet" data structure deallocated from heap
Packet Processing Complete

Figure 1: Packet Processing Flow

1.2 – Choice of Multithreading Model

Although a One Thread per X model is easier to implement, necessitating far less synchronisation, it presents two key issues. Firstly, if processing a large number of packets, the computational cost brought about by the continuous creation and destruction of threads grows to a problematic size, slowing application performance significantly. The other issue is the wasted CPU utilisation caused by constant context switches as the CPU must schedule time slices for a vast number of threads. The Threadpool Model avoids both these issues. It is far more scalable, enabling steady performance during the high-traffic bursts typical of internet traffic. The drawback, however, is the added implementation complexity due to the greater need for synchronisation and need for a queue data structure to manage packets. The flowchart below illustrates the workflow of a worker thread.

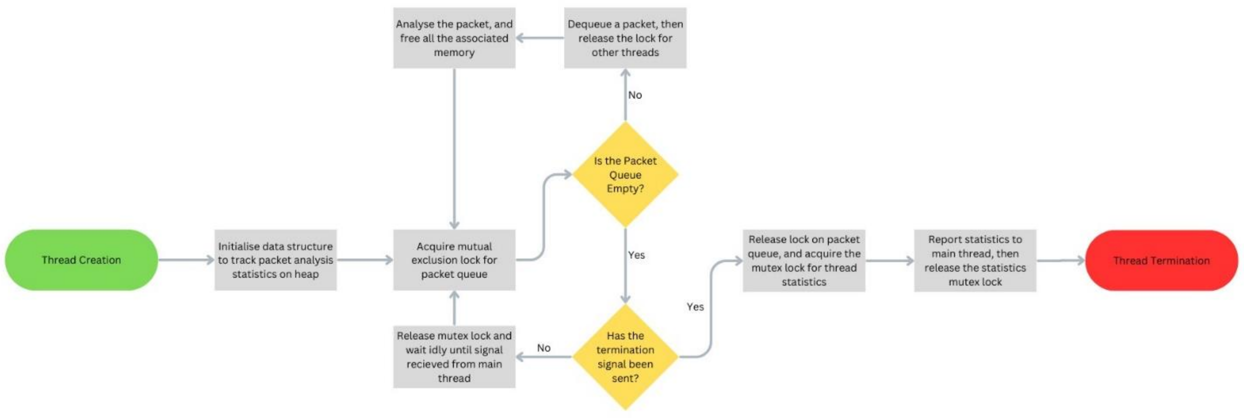

Thread Creation
Initialise data structure to track packet analysis statistics on heap
Acquire mutual exclusion lock for packet queue
Is the Packet Queue Empty?
No
Dequeue a packet, then release the lock for other threads
Analyse the packet, and free all the associated memory
Yes
Has the termination signal been sent?
No
Release mutex lock and wait idly until signal recieved from main thread
Yes
Release lock on packet queue, and acquire the mutex lock for thread statistics
Report statistics to main thread, then release the statistics mutex lock
Thread Termination

Figure 2: Worker Thread Execution Flow

1.3 – Synchronisation

A major design philosophy of the application was to minimise the need for synchronisation, as threads waiting on each other are likely to slow performance [2]. However, synchronisation primitives were implemented in four key places. Firstly, both a mutual exclusion lock and a condition variable are used for the packet queue. The mutex ensures packets are processed exactly once and are not lost while being added, avoiding potential memory leaks. The condition variable saves CPU resources by preventing busy waiting when the packet queue is empty. The second lock is used to synchronise printing to the terminal.

Another lock was used to synchronise search and insertion operations on the IP tree. A tree per thread design was considered, however, this would have necessitated the merging of many binary search trees into one while ensuring nodes were unique, a process which would have been both difficult to implement and computationally costly. Furthermore, each individual insertion operation has only logarithmic time complexity in average case, and analysis for most packets will not enter this critical section. For these reasons, a global tree was used. The final lock was used to safely report statistics to the main thread, preventing race conditions and consequent errors in the final report.

Deadlock is prevented by design, and therefore thread progress is ensured. The execution flow of the program ensures each thread is only waiting on, or allocated, at most one mutually excludable resource at any given time, preventing a circular wait condition and therefore deadlock from ever occurring. More formally, in terms of a resource allocation graph, the implementation ensures that all vertices denoting processes are either isolated or pendant (have degree 0 or 1 respectively), and therefore cannot be part of a cycle, in which all vertices must have degree at least two. It follows that no cycles are present in the graph. Bounded waiting is achieved by internal fairness mechanisms in the Linux implementation of the *pthreads* library [3], which attempt to grant a mutex lock to the process which has been waiting the longest.

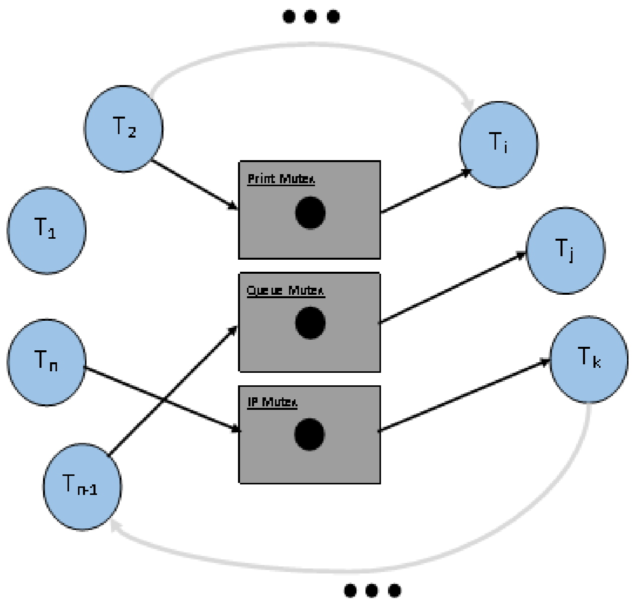


Figure 3: Resource allocation graph, with an arbitrary enumeration of vertices, representative of a typical state during packet processing

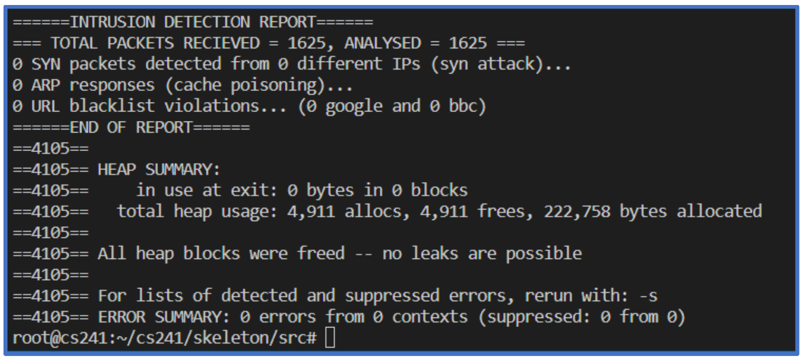
1.4 – Detecting Malicious Packets

The analysis of packets is done by using the layers of the OSI model successively to examine packet headers in detail. Firstly, the link layer ethernet header is inspected to determine the protocol for the network layer above. In the case of an ARP packet, the operation type is checked to spot and count ARP Replies. Alternatively, if the network layer protocol is IP and the transport layer protocol is TCP, the control flags in the TCP header are inspected to detect SYN packets. If one is found, a tree search/insert method is used to track unique IP addresses SYN packets have been sent from. After this, we check the destination port of the packet, to spot HTTP transmissions bound for port 80, and check the http request for attempts to access blacklisted URLs.

2 – Testing

2.1 – Correctness

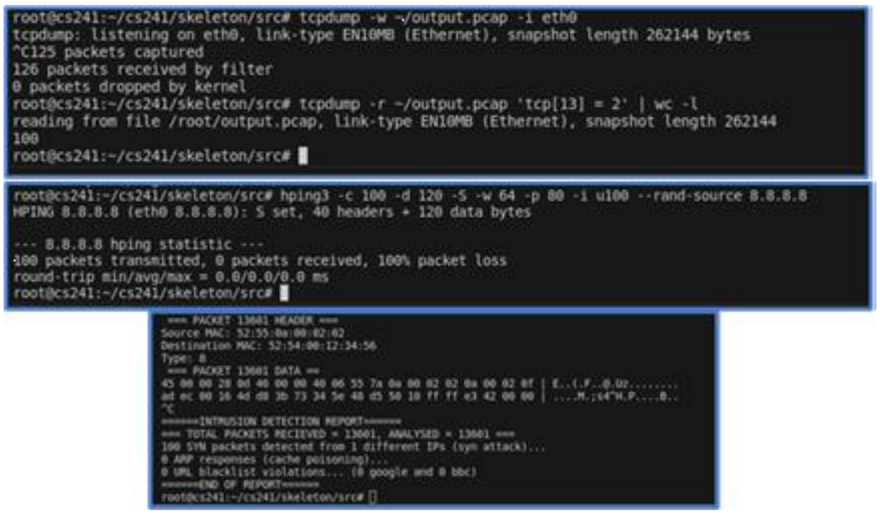
The first round of testing had the principal aim of ensuring application correctness. Firstly, The VSCode debug feature and *gdb* were used for debugging. Following this, *Valgrind* was used to check for memory leaks, with the addition of *Helgrind* [4] for multithreading errors.



```
======INTRUSION DETECTION REPORT======
=== TOTAL PACKETS RECIEVED = 1625, ANALYSED = 1625 ===
0 SYN packets detected from 0 different IPs (syn attack)...
0 ARP responses (cache poisoning)...
0 URL blacklist violations... (0 google and 0 bbc)
======END OF REPORT======
==4105==
==4105== HEAP SUMMARY:
==4105==     in use at exit: 0 bytes in 0 blocks
==4105==   total heap usage: 4,911 allocs, 4,911 frees, 222,758 bytes allocated
==4105==
==4105== All heap blocks were freed -- no leaks are possible
==4105==
==4105== For lists of detected and suppressed errors, rerun with: -s
==4105== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
root@cs241:~/cs241/skeleton/src# 
```

Figure 4: Valgrind report with helgrind enabled, after running packet processing for a short time on eth0

Terminal commands such as *hping* and *wget* were utilised to send packets over the *eth0* interface to initially test the software's capability to detect malicious packets. Another utilised tool was the packet analysis software *tcpdump*, which was used to confirm the absence of background SYN packets and validate the results of our own software.[5].



```
root@cs241:~/cs241/skeleton/src# tcpdump -w ~/output.pcap -i eth0
tcpdump: listening on eth0, link-type EN10MB (Ethernet), snapshot length 262144 bytes
^C125 packets captured
126 packets received by filter
0 packets dropped by kernel
root@cs241:~/cs241/skeleton/src# tcpdump -r ~/output.pcap 'tcp[13] = 2' | wc -l
reading from file /root/output.pcap, link-type EN10MB (Ethernet), snapshot length 262144
100
root@cs241:~/cs241/skeleton/src# 
```

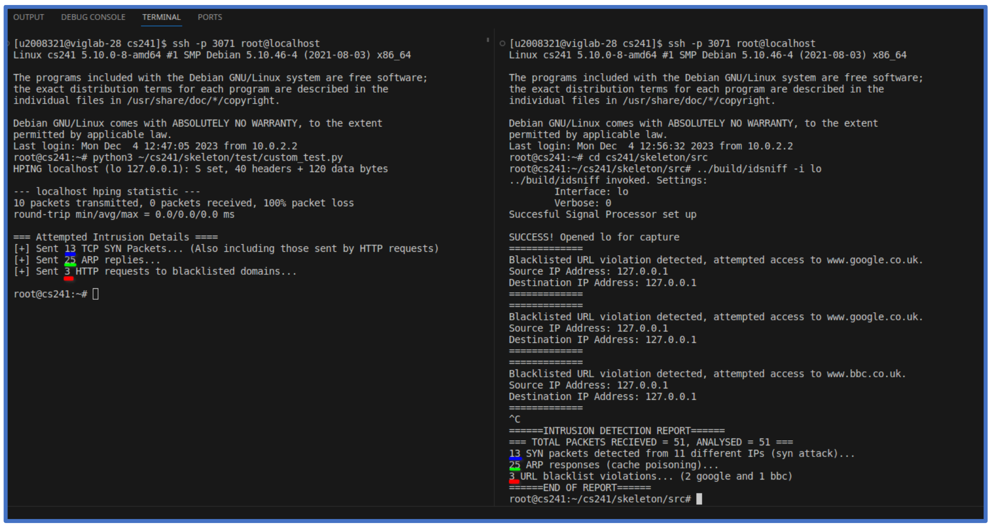
```
root@cs241:~/cs241/skeleton/src# hping3 -c 100 -d 120 -S -w 64 -p 80 -i u100 --rand-source 8.8.8.8
HPING 8.8.8.8 (eth0 8.8.8.8): S set, 40 headers + 120 data bytes

--- 8.8.8.8 hping statistic ---
100 packets transmitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms
root@cs241:~/cs241/skeleton/src# 
```

```
 === PACKET 13601 HEADER ===
Source MAC: 52:55:0a:00:02:02
Destination MAC: 52:54:00:12:34:56
Type: 8
 === PACKET 13601 DATA ==
45 00 00 28 0d 40 00 00 40 06 55 7a 0a 00 02 02 0a 00 02 0f | E..(.F..@.Uz........
ad ec 00 16 4d d8 3b 73 34 5e 48 d5 50 10 ff ff e3 42 00 00 | ....M.;s4^H.P....B..
^C
======INTRUSION DETECTION REPORT======
=== TOTAL PACKETS RECIEVED = 13601, ANALYSED = 13601 ===
100 SYN packets detected from 1 different IPs (syn attack)...
0 ARP responses (cache poisoning)...
0 URL blacklist violations... (0 google and 0 bbc)
======END OF REPORT======
root@cs241:~/cs241/skeleton/src# 
```

Figure 5: Terminal snippets using tcpdump to confirm correct SYN packet detection on eth0

For precise testing with no background packets, the loopback interface was utilised. To ensure all three attack types could be detected here, the etc/hosts file on the virtual machine was modified to redirect requests to the public IPs of the two blacklisted websites to 127.0.0.1 (localhost) [6]. While the application was being run concurrently, a custom python script was used to send a random barrage of intrusion attempts.



```
[u2008321@viglab-28 cs241]$ ssh -p 3071 root@localhost
Linux cs241 5.10.0-8-amd64 #1 SMP Debian 5.10.46-4 (2021-08-03) x86_64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Mon Dec  4 12:47:05 2023 from 10.0.2.2
root@cs241:~# python3 ~/cs241/skeleton/test/custom_test.py
HPING localhost (lo 127.0.0.1): S set, 40 headers + 120 data bytes

--- localhost hping statistic ---
10 packets transmitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms

=== Attempted Intrusion Details ====
[+] Sent 13 TCP SYN Packets... (Also including those sent by HTTP requests)
[+] Sent 25 ARP replies...
[+] Sent 3 HTTP requests to blacklisted domains...

root@cs241:~# 
```

```
[u2008321@viglab-28 cs241]$ ssh -p 3071 root@localhost
Linux cs241 5.10.0-8-amd64 #1 SMP Debian 5.10.46-4 (2021-08-03) x86_64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Mon Dec  4 12:56:32 2023 from 10.0.2.2
root@cs241:~# cd cs241/skeleton/src
root@cs241:~/cs241/skeleton/src# ../build/idsniff -i lo
../build/idsniff invoked. Settings:
        Interface: lo
        Verbose: 0
Succesful Signal Processor set up

SUCCESS! Opened lo for capture
=============
Blacklisted URL violation detected, attempted access to www.google.co.uk.
Source IP Address: 127.0.0.1
Destination IP Address: 127.0.0.1
=============
=============
Blacklisted URL violation detected, attempted access to www.google.co.uk.
Source IP Address: 127.0.0.1
Destination IP Address: 127.0.0.1
=============
=============
Blacklisted URL violation detected, attempted access to www.bbc.co.uk.
Source IP Address: 127.0.0.1
Destination IP Address: 127.0.0.1
=============
^C
======INTRUSION DETECTION REPORT======
=== TOTAL PACKETS RECIEVED = 51, ANALYSED = 51 ===
13 SYN packets detected from 11 different IPs (syn attack)...
25 ARP responses (cache poisoning)...
3 URL blacklist violations... (2 google and 1 bbc)
======END OF REPORT======
root@cs241:~/cs241/skeleton/src# 
```

Figure 6: Terminal output from testing with custom python script on loopback interface

2.2 – Optimisation

Secondly, care was taken to optimise the program with respect to runtime. An earlier design used many more mutual exclusion locks to update counts every time a packet was analysed, but it was discovered runtime could be saved by each thread having their own set of counts, which could then be collated after packet processing had concluded, where speed was less critical. Another question in this area related to the optimum size of the thread-pool [7]. By modifying one of the arguments of *pcap_loop()*, and using the *time.h* library, we were able to analyse the runtime of processing 100 packets while varying thread-pool size in increments of five. The results suggested an optimum threadpool size of 10.

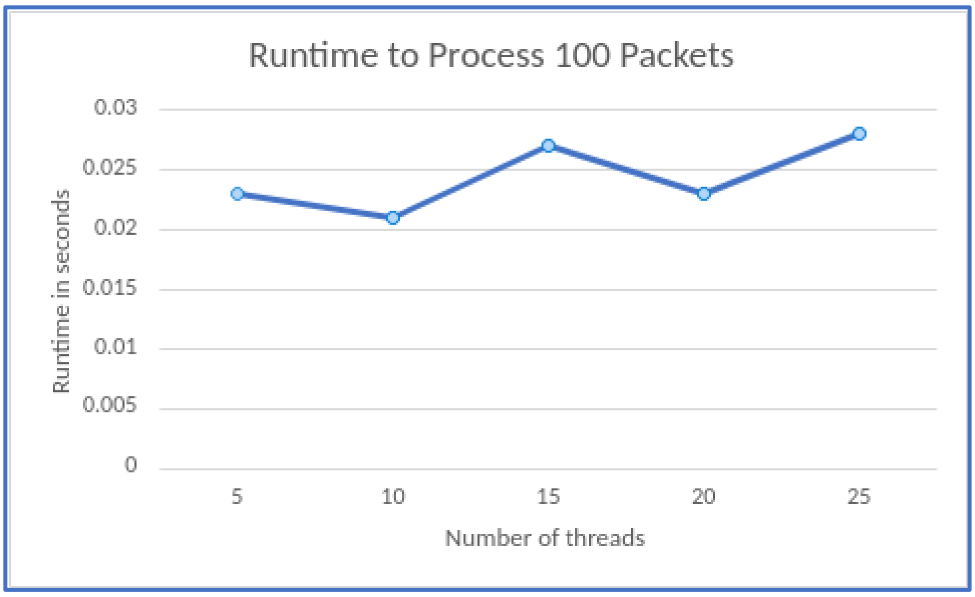
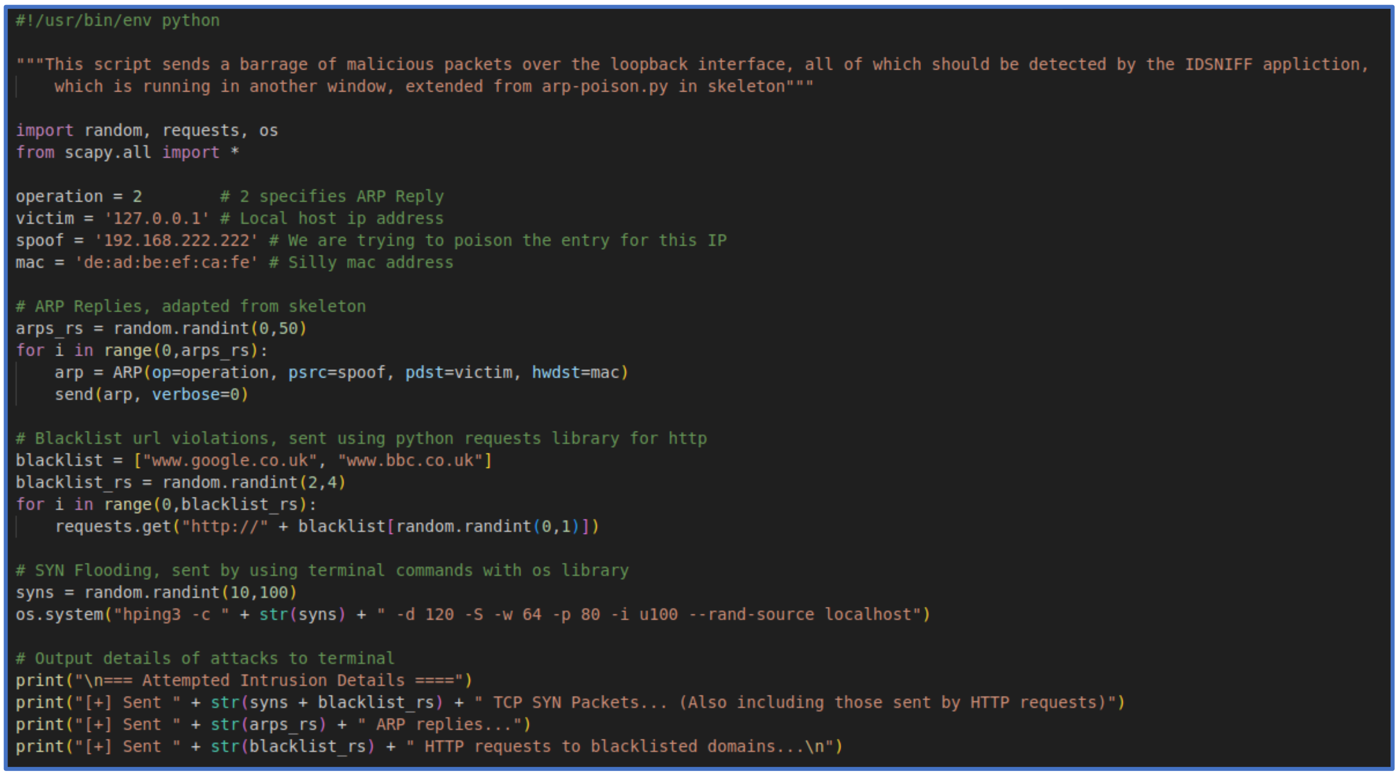


Figure 7: Experimental bivariate analysis on the effect of thread-pool size on runtime

Bibliography

- [1] The Linux Documentation Project 2023, 'pcap_loop(3) - Linux man page', die.net, Available at: https://linux.die.net/man/3/pcap_loop (Accessed: 4 December 2023).
- [2] Apple Inc. 2023, 'Thread Safety Summary', Apple Developer Documentation, Available at: https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/Multithreading/ThreadSafety/ThreadSafety.html (Accessed: 4 December 2023).
- [3] The Open Group 2017, 'POSIX Threads', The Open Group Base Specifications Issue 7, IEEE Std 1003.1, Available at: https://pubs.opengroup.org/onlinepubs/9699919799/ (Accessed: 4 December 2023).
- [4] Tcpdump Group 2023, 'Tcpdump - A powerful tool for network monitoring and data acquisition', Tcpdump, Available at: https://www.tcpdump.org/manpages/tcpdump.1.html (Accessed: 4 December 2023).
- [5] Valgrind Developers 2023, 'Valgrind User Manual', Valgrind Documentation, Available at: https://valgrind.org/docs/manual/hg-manual.html (Accessed: 4 December 2023).
- [6] Linux Documentation Project 2023, 'Chapter 9. Configuring the Kernel', The Linux Documentation Project, Available at: https://tldp.org/LDP/solrhe/Securing-Optimizing-Linux-RH-Edition-v1.3/chap9sec95.html (Accessed: 4 December 2023).
- [7] AgilePoint Documentation 2023, 'Sizing ThreadPool and Sizing Guidelines', AgilePoint Documentation, Available at: https://documentation.agilepoint.com/8010/admin/sizingThreadPoolSizingGuidelines.html (Accessed: 4 December 2023).

Appendix: Python Script for testing on loopback interface



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

"""This script sends a barrage of malicious packets over the loopback interface, all of which should be detected by the IDSNIFF appliction,
    which is running in another window, extended from arp-poison.py in skeleton"""

import random, requests, os
from scapy.all import *

operation = 2        # 2 specifies ARP Reply
victim = '127.0.0.1' # Local host ip address
spoof = '192.168.222.222' # We are trying to poison the entry for this IP
mac = 'de:ad:be:ef:ca:fe' # Silly mac address

# ARP Replies, adapted from skeleton
arps_rs = random.randint(0,50)
for i in range(0,arps_rs):
    arp = ARP(op=operation, psrc=spoof, pdst=victim, hwdst=mac)
    send(arp, verbose=0)

# Blacklist url violations, sent using python requests library for http
blacklist = ["www.google.co.uk", "www.bbc.co.uk"]
blacklist_rs = random.randint(2,4)
for i in range(0,blacklist_rs):
    requests.get("http://" + blacklist[random.randint(0,1)])

# SYN Flooding, sent by using terminal commands with os library
syns = random.randint(10,100)
os.system("hping3 -c " + str(syns) + " -d 120 -S -w 64 -p 80 -i u100 --rand-source localhost")

# Output details of attacks to terminal
print("\n=== Attempted Intrusion Details ====")
print("[+] Sent " + str(syns + blacklist_rs) + " TCP SYN Packets... (Also including those sent by HTTP requests)")
print("[+] Sent " + str(arps_rs) + " ARP replies...")
print("[+] Sent " + str(blacklist_rs) + " HTTP requests to blacklisted domains...\n")
```