Report on Threading Strategy, Design Decisions, and Testing

Threading strategy and program overview

This program demonstrates a multi-threaded packet sniffing application that captures packets from a specified network interface using `libpcap` and `libpthreads`. The program utilizes a thread pool design to distribute packet analysis work among a fixed number of worker threads.

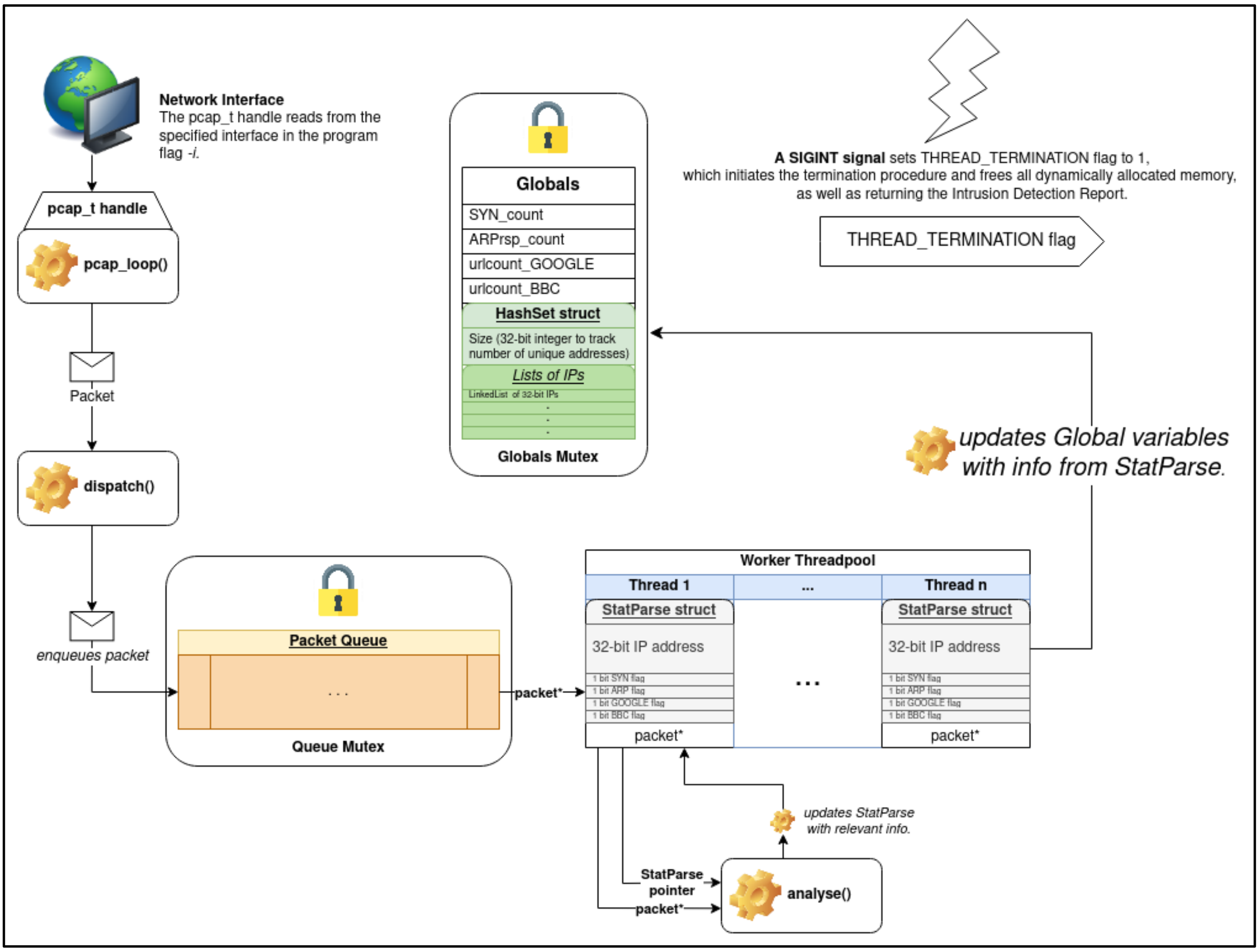


Figure 1

Figure 1 provides a diagrammatical overview of the programs execution, starting from the network interface in the upper left, and terminating upon detection of a SIGINT signal as represented in the upper right.

Program Overview:

1. **Packet Handling:**
   - The `pcap_loop()` function captures packets continuously, invoking the `dispatch()` function for each packet, which enqueues the packet's header in binary format to a FIFO work queue. This is performed under lock of the *queue*

mutex. Once enqueued, the `pthread_cond_broadcast()` function is invoked which wakes all sleeping threads.

2. **Thread work and packet analysis:**
   - Worker threads are responsible for packet analysis. They either wait for packets in the queue, or sleep until there is a packet in the queue, and then analyse the packet data using the `analyse()` function. After analysis, the thread updates global variables after obtaining a lock on the *globals mutex*.
   - A `StatParse` struct in each thread tracks detected attacks and relevant information for each packet and is reset at the beginning of each analysis.
   - A termination signal (`SIGINT`) triggers the termination process, signaling worker threads to complete their tasks and clean up resources.
3. **Data Structures:**
   - `Queue` in `queue.c` implements a basic FIFO queue used to store packet information for worker threads.
   - `HashMap` in `hashmap.c` implements a simple hash set to store unique IP addresses involved in SYN attacks.

Justification for the Thread-pool model

The chosen threading model involves a thread pool with a fixed number of worker threads (`NUMTHREADS`). This approach was selected for several reasons:

1. **Efficiency:**

   Due to the very large number of packet throughput on any given interface, thread creation for each packet would cause excessive overhead, and therefore a fixed pool of threads can more efficiently handle packet analysis.

2. **Resource Utilization:**

   Resource exhaustion is limited by the number of concurrent threads to a manageable level, based on a system's capabilities. The threads also share resources from the process stack, such as the packet queue, the global variables, and the hashset of unique IP addresses.

3. **Concurrency:**

   Multi-threading allows simultaneous processing of packets, maximizing the utilization of multi-core systems. Thread switching is also largely faster than context switching, and as such a multithreaded model is preferred to multi-process.

4. **Scalability:**

   The number of threads can also be adjusted based on system resources and performance requirements. Since the DCS virtual machine only offers 1 core, 4 threads were decided as an estimate on the number of cores for an actual system running the program.

Design Decisions: the thread stack and program

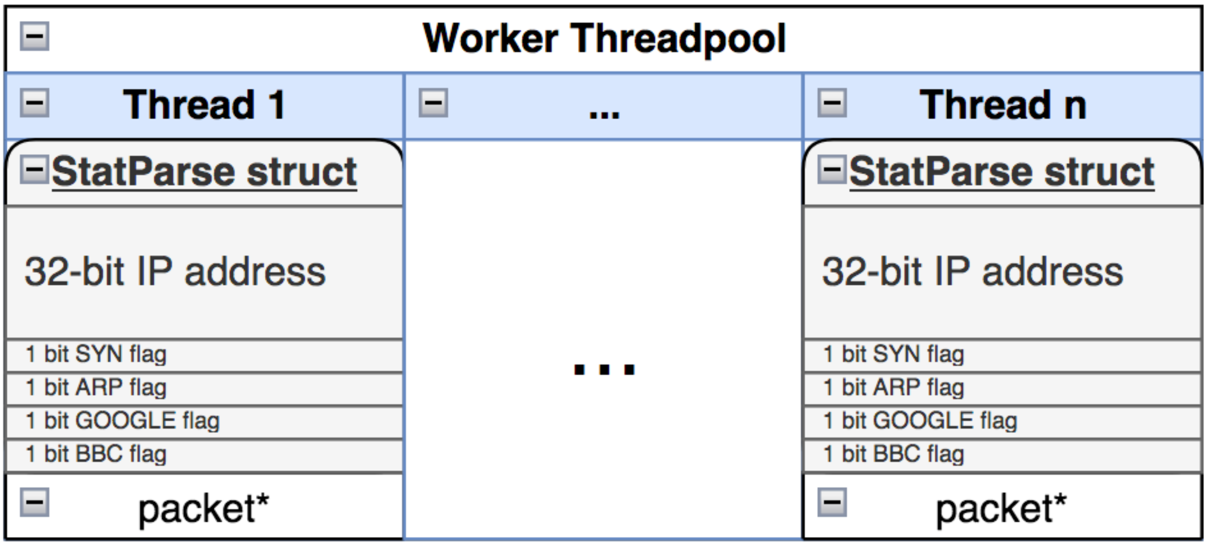


Figure 2: The thread-pool and thread stack

The stack of each thread is represented diagrammatically in Figure 2. Each thread has a `StatParse` struct, defined in analysis.h. This struct allocates memory for an IP as a 32 bit unsigned integer, and 4 x 1-bit flags of type uint8_t (or char). It also contains a pointer to the most recently dequeued packet data. The `StatParse` struct utilizes 1-bit flag fields to not only reduce the amount of memory that the program interacts with, but also allows for the program to perform bit setting efficiently using bitwise operations. The need for each thread to perform it's packet analysis with computational efficiency is noticed when observing the number of packets per second possibly present in modern day ethernet routers. The IEEE 802 Standards Committee recently addressed for example, 100 and up to 400 Gbps ethernet speeds[1], which can mean millions of packets per second. As such, reducing the overhead of operations performed in the threads' analysis program are vital.

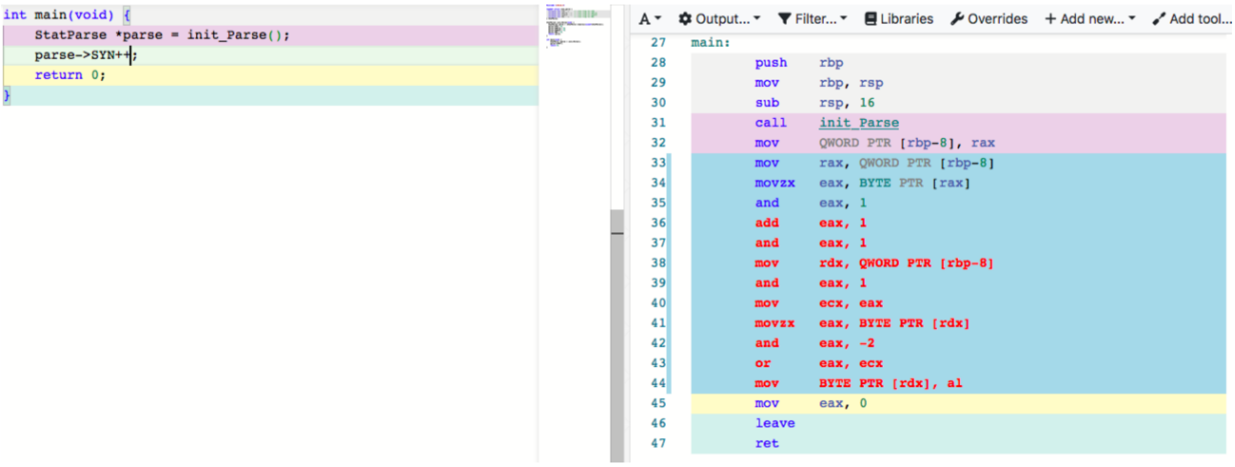
The following code snippet exhibits the utilisation of bitwise operations to improve computational speeds:

```
uint8_t flags = tcp_header->th_flags;

if ((flags & TH_SYN) && !(flags & TH_ACK)) {

     parse->SYN |= 1;

}
```

[1] [3]

As a SYN attack is comprised via the flooding of packets with the SYN TCP flag ON, and the ACK flag OFF, these must be checked by the program. To do this efficiently, the BITWISE AND operation is performed on the flag field to ensure that this is true.

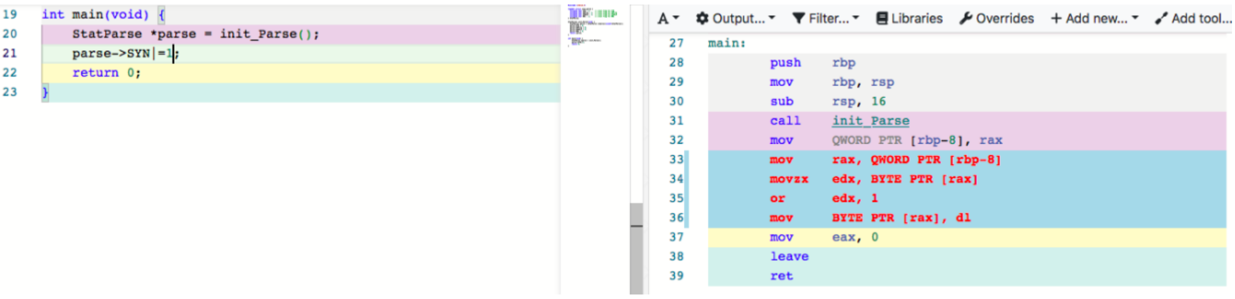
To set the SYN bit in the `StatParse` struct to 1, instead of performing `SYN++`, `SYN+=1` or `SYN=1`, `SYN|=1` is performed instead. Figure 3 and Figure 4[2] compare the x86 assembly instructions of `SYN++` and `SYN|=1`.



```
int main(void) {
    StatParse *parse = init_Parse();
    parse->SYN++;
    return 0;
}
```

```
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        call    init_Parse
        mov     QWORD PTR [rbp-8], rax
        mov     rax, QWORD PTR [rbp-8]
        movzx   eax, BYTE PTR [rax]
        and     eax, 1
        add     eax, 1
        and     eax, 1
        mov     rdx, QWORD PTR [rbp-8]
        and     eax, 1
        mov     ecx, eax
        movzx   eax, BYTE PTR [rdx]
        and     eax, -2
        or      eax, ecx
        mov     BYTE PTR [rdx], al
        mov     eax, 0
        leave
        ret
```

Figure 3: x86-64 assembly instructions for `SYN++` *compiled in gcc 13.2 highlighted in red*



```
int main(void) {
    StatParse *parse = init_Parse();
    parse->SYN|=1;
    return 0;
}
```

```
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        call    init_Parse
        mov     QWORD PTR [rbp-8], rax
        mov     rax, QWORD PTR [rbp-8]
        movzx   edx, BYTE PTR [rax]
        or      edx, 1
        mov     BYTE PTR [rax], dl
        mov     eax, 0
        leave
        ret
```

Figure 4: x86-64 assembly instructions for `SYN|=1` *compiled in gcc 13.2 highlighted in red*

As we can see, `SYN++` requires 12 assembly instructions, whereas `SYN|=1` requires only 4. Furthermore, `SYN=1` compiles to the exact same instructions as `SYN|=1`, performing the same assembly operation: `or [REG], 1` in order to set the bit. Using `SYN|=1` ensures that if a user was using a different compiler, they would still be performing the same 4 instructions whereas `SYN=1` may give different results.

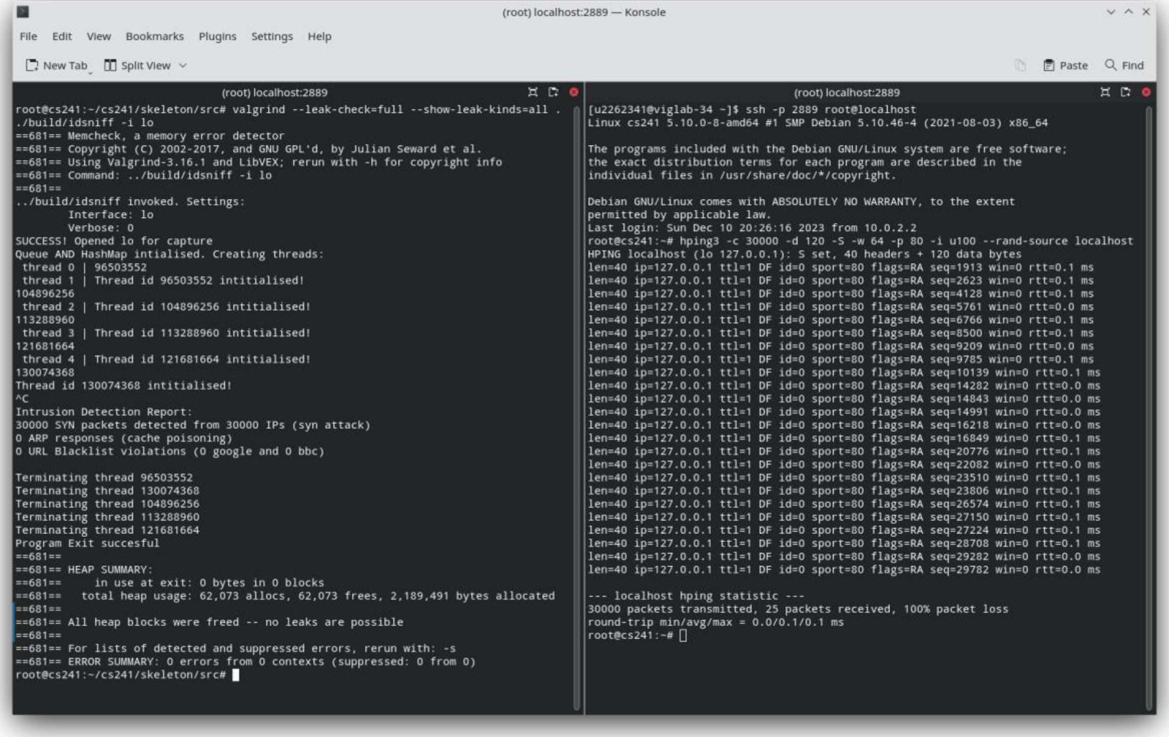
A hash set was used to store addresses as it would be largely space inefficient to store use a dynamic array in a very high throughput network, such as a datacentre. It is therefore optimal to implement a hashing function which both achieves low computational cost and even

[2] [1]

distribution of hash results. I used a 32-bit hash function created by Thomas Mueller of H2 Database which was shown to have an avalanche effect better than MurmurHash3, which uses computationally cheap *xorshift-multiply-xorshift* operations, as well as using the same multiplicative constant twice[3], for the best performance.

Testing

To test the program, numerous tests were performed. On top of successfully passing all tests provided within the coursework specification[4], SYN packet stress tests were performed on quantities of 30000, 100000 and 1000000 packets. All tests successfully detected all SYN packets and unique IP addresses with no memory leaks on program termination as displayed through `valgrind --leak-check=full`.

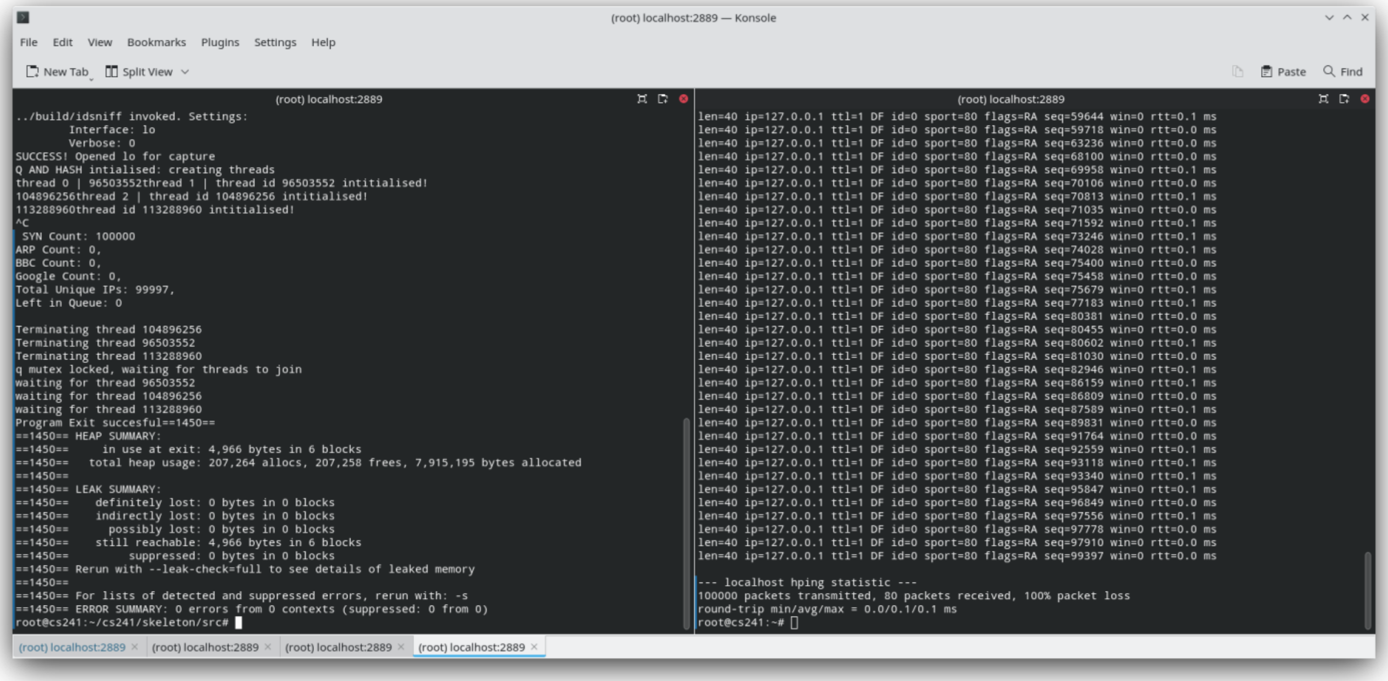


```
root@cs241:~/cs241/skeleton/src# valgrind --leak-check=full --show-leak-kinds=all .
./build/idsniff -i lo
==681== Memcheck, a memory error detector
==681== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==681== Using Valgrind-3.16.1 and LibVEX; rerun with -h for copyright info
==681== Command: ../build/idsniff -i lo
==681==
../build/idsniff invoked. Settings:
        Interface: lo
        Verbose: 0
SUCCESS! Opened lo for capture
Queue AND HashMap intialised. Creating threads:
 thread 0 | 96503552
 thread 1 | Thread id 96503552 intitialised!
104896256
 thread 2 | Thread id 104896256 intitialised!
113288960
 thread 3 | Thread id 113288960 intitialised!
121681664
 thread 4 | Thread id 121681664 intitialised!
130074368
Thread id 130074368 intitialised!
^C
Intrusion Detection Report:
30000 SYN packets detected from 30000 IPs (syn attack)
0 ARP responses (cache poisoning)
0 URL Blacklist violations (0 google and 0 bbc)

Terminating thread 96503552
Terminating thread 130074368
Terminating thread 104896256
Terminating thread 113288960
Terminating thread 121681664
Program Exit succesful
==681==
==681== HEAP SUMMARY:
==681==     in use at exit: 0 bytes in 0 blocks
==681==   total heap usage: 62,073 allocs, 62,073 frees, 2,189,491 bytes allocated
==681==
==681== All heap blocks were freed -- no leaks are possible
==681==
==681== For lists of detected and suppressed errors, rerun with: -s
==681== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
root@cs241:~/cs241/skeleton/src# █
```

```
[u2262341@viglab-34 ~]$ ssh -p 2889 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: Sun Dec 10 20:26:16 2023 from 10.0.2.2
root@cs241:~# hping3 -c 30000 -d 120 -S -w 64 -p 80 -i u100 --rand-source localhost
HPING localhost (lo 127.0.0.1): S set, 40 headers + 120 data bytes
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=1913 win=0 rtt=0.1 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=2623 win=0 rtt=0.1 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=4128 win=0 rtt=0.1 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=5761 win=0 rtt=0.0 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=6766 win=0 rtt=0.1 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=8500 win=0 rtt=0.1 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=9209 win=0 rtt=0.0 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=9785 win=0 rtt=0.1 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=10139 win=0 rtt=0.1 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=14282 win=0 rtt=0.0 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=14843 win=0 rtt=0.0 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=14991 win=0 rtt=0.0 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=16218 win=0 rtt=0.0 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=16849 win=0 rtt=0.1 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=20776 win=0 rtt=0.1 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=22082 win=0 rtt=0.0 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=23510 win=0 rtt=0.1 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=23806 win=0 rtt=0.1 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=26574 win=0 rtt=0.1 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=27150 win=0 rtt=0.1 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=27224 win=0 rtt=0.1 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=28708 win=0 rtt=0.1 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=29282 win=0 rtt=0.0 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=29782 win=0 rtt=0.0 ms

--- localhost hping statistic ---
30000 packets transmitted, 25 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.1/0.1 ms
root@cs241:~# ▯
```

Figure 5: 30000 packet stress test with no memory leaks possible.

[3] [4]
[4] [2]



```
../build/idsniff invoked. Settings:
        Interface: lo
        Verbose: 0
SUCCESS! Opened lo for capture
Q AND HASH intialised: creating threads
thread 0 | 96503552thread 1 | thread id 96503552 intitialised!
104896256thread 2 | thread id 104896256 intitialised!
113288960thread id 113288960 intitialised!
^C
 SYN Count: 100000
ARP Count: 0,
BBC Count: 0,
Google Count: 0,
Total Unique IPs: 99997,
Left in Queue: 0

Terminating thread 104896256
Terminating thread 96503552
Terminating thread 113288960
q mutex locked, waiting for threads to join
waiting for thread 96503552
waiting for thread 104896256
waiting for thread 113288960
Program Exit succesful==1450==
==1450== HEAP SUMMARY:
==1450==     in use at exit: 4,966 bytes in 6 blocks
==1450==   total heap usage: 207,264 allocs, 207,258 frees, 7,915,195 bytes allocated
==1450==
==1450== LEAK SUMMARY:
==1450==    definitely lost: 0 bytes in 0 blocks
==1450==    indirectly lost: 0 bytes in 0 blocks
==1450==      possibly lost: 0 bytes in 0 blocks
==1450==    still reachable: 4,966 bytes in 6 blocks
==1450==         suppressed: 0 bytes in 0 blocks
==1450== Rerun with --leak-check=full to see details of leaked memory
==1450==
==1450== For lists of detected and suppressed errors, rerun with: -s
==1450== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
root@cs241:~/cs241/skeleton/src#
```

```
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=59644 win=0 rtt=0.1 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=59718 win=0 rtt=0.0 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=63236 win=0 rtt=0.0 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=68100 win=0 rtt=0.0 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=69958 win=0 rtt=0.1 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=70106 win=0 rtt=0.0 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=70813 win=0 rtt=0.1 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=71035 win=0 rtt=0.0 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=71592 win=0 rtt=0.1 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=73246 win=0 rtt=0.1 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=74028 win=0 rtt=0.1 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=75400 win=0 rtt=0.0 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=75458 win=0 rtt=0.0 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=75679 win=0 rtt=0.1 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=77183 win=0 rtt=0.1 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=80381 win=0 rtt=0.0 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=80455 win=0 rtt=0.0 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=80602 win=0 rtt=0.1 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=81030 win=0 rtt=0.0 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=82946 win=0 rtt=0.1 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=86159 win=0 rtt=0.1 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=86809 win=0 rtt=0.0 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=87589 win=0 rtt=0.1 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=89831 win=0 rtt=0.0 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=91764 win=0 rtt=0.0 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=92559 win=0 rtt=0.1 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=93118 win=0 rtt=0.1 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=93340 win=0 rtt=0.1 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=95847 win=0 rtt=0.1 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=96849 win=0 rtt=0.0 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=97556 win=0 rtt=0.1 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=97778 win=0 rtt=0.0 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=97910 win=0 rtt=0.0 ms
len=40 ip=127.0.0.1 ttl=1 DF id=0 sport=80 flags=RA seq=99397 win=0 rtt=0.0 ms

--- localhost hping statistic ---
100000 packets transmitted, 80 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.1/0.1 ms
root@cs241:~# 
```

Figure 6: 1000000 packet stress test with no memory leaks

Bibliography

[1] "Compiler Explorer," 2023. [Online]. Available: https://godbolt.org/.

[2] "CS241 Coursework 2023-2024," December 2023. [Online]. Available: https://warwick.ac.uk/fac/sci/dcs/teaching/material/cs241/coursework23-24/.

[3] IEEE, "IEEE 802.3ck-2022: IEEE Standard for Ethernet Amendment 4: Physical Layer Specifications and Management Parameters for 100 Gb/s, 200 Gb/s, and 400 Gb/s Electrical Interfaces Based on 100 Gb/s Signaling," *IEEE 802 Standards*, p. 1, 2022.

[4] T. Mueller, "Open-source H2Database," 21 Oct 2012. [Online]. Available: https://stackoverflow.com/questions/664014/what-integer-hash-function-are-good-that-accepts-an-integer-hash-key/12996028#12996028.