1 Threading model

A thread-pool model was chosen. The reasons for this are

1. Thread creation overhead is handled before reading packets, thus packets can be allocated to threads faster.
2. Threads are reused instead of creating new ones for each packet, avoiding additional memory operations which require CPU time.
3. If a large number of packets are read in a short time, thread-pools limit the number of threads created to prevent excessive memory and resource usage.

1.1 Implementation

To implement the thread-pool threading model, the following must be done.

1. Thread creation.
2. Thread to packet allocation.
3. Change thread behaviour to handle SIGINT signal.
4. Thread tear down.

1.1.1 Thread creation

Thread creation is handled once on program startup as follows.

```
signal(SIGINT, sig_handler);

work_queue = create_queue();
initstats();

int i;
for(i = 0; i < NUM_THREADS; i++) {
    pthread_create(&tid[i], NULL, runner, NULL);
}
```

We create the work queue which will be used to store packets in the order they were read from the chosen network interface. We then create a pre-defined constant number of threads for our thread-pool using a for loop. We set arguments 2 and 4 in `pthread_create` to be `null` because we want default thread creation and do not pass any user defined arguments to our threads.

1.1.2 Thread to packet allocation

When a packet is read from a network interface, it is added to the work queue in the dispatch function.

```
pthread_mutex_lock(&queue_mutex);
enqueue(work_queue, (unsigned char *) pkt, pkt->len);
pthread_cond_signal(&queue_cond);
pthread_mutex_unlock(&queue_mutex);
```

A mutex lock is needed to avoid the situation where a thread is dequeueing a packet at the same time as the packet is being enqueued. This situation is undesirable as if the head is equal to the tail, one thread may remove the head, while another thread tries to add a new element to the removed head. A signal is then sent to a thread currently waiting for work, if such a thread exists, using the `pthread_cond_signal` function.

1.1.3 Signal Handling

We want our program to implement the following behaviour upon receiving the SIGINT signal.

1. Stop reading packets from the interface.
2. Finish analysing packets in the work queue.
3. Output the analysis results.
4. Free memory and end gracefully.

We implement a signal handler to initiate these actions upon receiving the SIGINT signal.

```
void sig_handler(int signal) {
    pcap_breakloop(pcap_handle);
    run = 0;
}
```

`pcap_breakloop` stops the reading of packets on the chosen network interface. The `run` flag serves as a reference for the threads to adjust their behaviour based on whether the SIGINT signal has been received. The threads operate as follows.

```
while(run || !isempty(work_queue->ll)) {

    pthread_mutex_lock(&queue_mutex);
    while(run && isempty(work_queue->ll)) {
        pthread_cond_wait(&queue_cond, &queue_mutex);
    }

    struct packet *pkt = (struct packet *) dequeue(work_queue);
    pthread_mutex_unlock(&queue_mutex);

    if(pkt != NULL) {
```

```
        analyse(pkt);
    }
}
```

The outer while loop is responsible for keeping the thread from returning after a packet has been analysed. Before SIGINT (`run = 1`), this condition should always be true. After SIGINT (`run = 0`), threads should only keep trying to dequeue from the work queue if the queue is not empty, because packets are no longer being read from the network interface.

The inner while loop is responsible for controlling whether the thread should be waiting on a signal or not. If `run = 1`, the thread should wait if and only if the work queue is empty. If `run = 0`, no thread should be placed in the waiting state because if the work queue is not empty the thread should continue analysing; else the queue is empty and it is known that no new packets will enter the work queue, thus the thread can return.

The mutex lock around inner while loop is responsible for preventing two threads from trying to dequeue from the work queue at the same time. This is undesirable because a race condition may occur where one thread will dequeue first and thus the second thread will try to dequeue a node which no longer exists in the queue.

1.1.4 Thread tear down

To properly close the threads and free memory we execute the following at the end of program execution.

```
pthread_cond_broadcast(&queue_cond);
int i;
for(i = 0; i < NUM_THREADS; i++) {
    pthread_join(tid[i], NULL);
}

clear_mem();
free(work_queue);
```

The `pthread_cond_brodcast` function is called to unblock any threads that are in the waiting state before calling `pthread_join`, which waits for the threads to return and then handles freeing them from memory. We then free dynamically allocated memory such as the work queue.

1.1.5 Thread safety in analysis.c

Three mutex locks are used to ensure thread safety when updating the variables keeping track of analysis results: the ARP packet, SYN packet and HTTP mutex locks. These mutex locks ensure that only one thread can update the

same analysis variable at a time. Three mutex locks are used instead of one because there is no need to block all other threads from updating, say, the SYN packet count variable, if one thread is updating the ARP packet count variable since they are stored in different memory locations.

2 Testing

2.1 Requirements, test and rationale

Requirements	**Test**	**Rationale**
Count the number of ARP reply packets	100 ARP reply packets. 100 ARP request packet. 100 IP packets.	Tests that ARP reply packets are counted. Tests that non-ARP packets and ARP requests are not counted. Tests that the analysis outputs correctly if more than one of each packet is received. Tests that threads do not simultaneously modify the ARP packet counter variable because ARP reply packets are sent in succession.
Count the number of SYN packets, and of them, how many are from unique IP addresses.	100 SYN packets from 50 unique IP addresses. 100 ACK packets	Tests that SYN packets are counted. Tests that non-SYN packets are not counted. Tests that the analysis outputs correctly if more than one of each packet is received. Tests that counted number of unique IP addresses is correct by differentiating this number from the total number of IP packets sent. Tests that threads do not simultaneously modify the SYN packet counter variable and unique IP address set because SYN packets are sent in succession. Tests that thread-pool can handle high throughput without noticeably slowing down the host system.
Count the number of times HTTP traffic is received from a blacklisted URL.	10 HTTP GET request to non-blacklisted URL. 11 HTTP GET request to blacklisted URL.	Tests that blacklisted URLs are counted. Tests that non-blacklisted URLs are not counted. Tests that analysis outputs correctly if more than one of each HTTP requests is received. Tests that threads do not simultaneously modify the blacklisted URL counter variable because HTTP requests are made in succession.

2.2 Implementation

2.2.1 ARP cache poisoning

The provided `arp-poison.py` python script was used to send ARP replies and a slight modification of it to send ARP requests. The `hping3 -c 100 -d 120 -S -w 64 -p 80 -i u100 --rand-source localhost` command was used to generate the IP packets.

2.2.2 SYN flooding

A custom made bash script in the test folder called `syn-flood.sh` was used to send the 100 IP packets from 50 unique IP addresses and to generate these random IP addresses. To send the 100 ACK packets, the command `hping3 -c 100 -d 120 -A -w 64 -p 80 -i u100 --rand-source localhost` was used.

2.2.3 Blacklisted URLs

`wget www.google.co.uk` was used to make an HTTP GET request to a blacklisted URL, and `wget non-blacklisted-url-here` was used to make an HTTP GET request to a non-blacklisted URL.

References

DCS, University of Warwick (n.d.). *CS241 Coursework 2020-2021*. URL: `https://warwick.ac.uk/fac/sci/dcs/teaching/material/cs241/coursework20-21`.

Ethernet IEEE 802.3 Frame Format / Structure (n.d.). URL: `https://www.electronics-notes.com/articles/connectivity/ethernet-ieee-802-3/data-frames-structure-format.php`.

Information Sciences Institute University of Southern California, Jon Postel (n.d.[a]). *Internet Protocol*. URL: `https://tools.ietf.org/html/rfc791`.

– (n.d.[b]). *Transmission Control Protocol*. URL: `https://tools.ietf.org/html/rfc793`.

Sanfilippo, Salvatore (n.d.). *hping3(8) - Linux man page*. URL: `https://linux.die.net/man/8/hping3`.