Threading

Strategy and design

We decide to use a threadpool. Each thread runs the same function, which takes a sniffed packet from a global work queue and runs analysis on it, updating necessary variables. Each thread does this on an infinite loop until an interrupt signal is given, upon which each thread exits safely and rejoins the main thread. To separate out the analysing logic, thread logic, and main program logic, we implement the threadpool in the following way.

Implementation

1. We create the threads using `pthread_create()` before sniffing begins, with each threads' start routine being the `void* dispatch(void*)` function.

2. There is a global queue, `packetQueue`, which packets are added to via the `pcap_loop()` handler, `got_packet()`.

3. `void* dispatch(void*)` contains an infinite while loop in which each thread continually takes packets from the queue and runs `analyse()` on them. Since the queue is shared amongst threads, a mutex lock `qMutex` is used when dequeueing, and a condition `qCond` to signal when the queue is no longer empty.

4. In `analysis.c` are 3 mutex locks. After control flow to check what type of packet we have received, if any variables need updating, the respective mutex is used for the packet. (Note there is no overlap in the variables being updated across `analyseSYN`, `analyseHTTP` and `analyseARP`, so using separate locks is safe). Multiple locks allow multiple different types of packet to be analysed at any one time, improving efficiency.

5. Upon a `SIGINT` signal, a global variable `int interrupt` is set to 1 (after `pcap_breakloop()` is called in the signal handler), and `pCond` is broadcast to so that any thread waiting for the queue to be non-empty instead continues. Every thread will then unlock `qMutex` (so the next thread can follow) and call `pthread_exit(0)`. `pthread_join()` is then called in `sniff.c` for all threads.

Justification

In the context of sniffing packets, a threadpool has many benefits over other models.

Noting that internet traffic is bursty, meaning we can have a high influx of packets coming in during a short period of time:

1. There is a limit on the number of threads in a threadpool.
   (a) Thrashing is avoided, in which the CPU spends more time context switching or moving resources than performing execution.
2. Threads are not created dynamically in a threadpool.
   (a) Thread creation can be expensive for the CPU as system calls to the kernel are involved, and in models such as One-thread-per-X, a new thread would be created for every new packet received. A fixed number of threads can also lead to better system stability.

Testing

Optimal number of threads

To ensure the threadpool is actually more effective in processing a larger number of packets, and then to decide the optimal number of threads (balancing the overhead of thread creation and context switching with concurrent processing), we calculate the running time of the program in the following way:

1. We will flood the program with SYN packets, with the sniffer listening on the loopback `lo` interface.
   (a) `hping3 --flood -S --rand-source localhost`
2. We will stop program execution when the work queue fills up to a certain limit.
3. We time the program by finding the start and end time using `clock()` and dividing by `CLOCKS_PER_SEC`
4. Repeat, doubling the number of threads we use each time (starting with 1 thread)

```
#define NUMTHREADS 1
// ...
#include <time.h>
int limit = 100000;
int count = 0;
clock_t begin;
// ...
begin = clock();
// ...
if (count == limit)
{
  clock_t end = clock();
  double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
  fprintf(stderr, "Limit reached for queue in %f seconds \n",
      time_spent);

 exit(0);
}
```

Findings

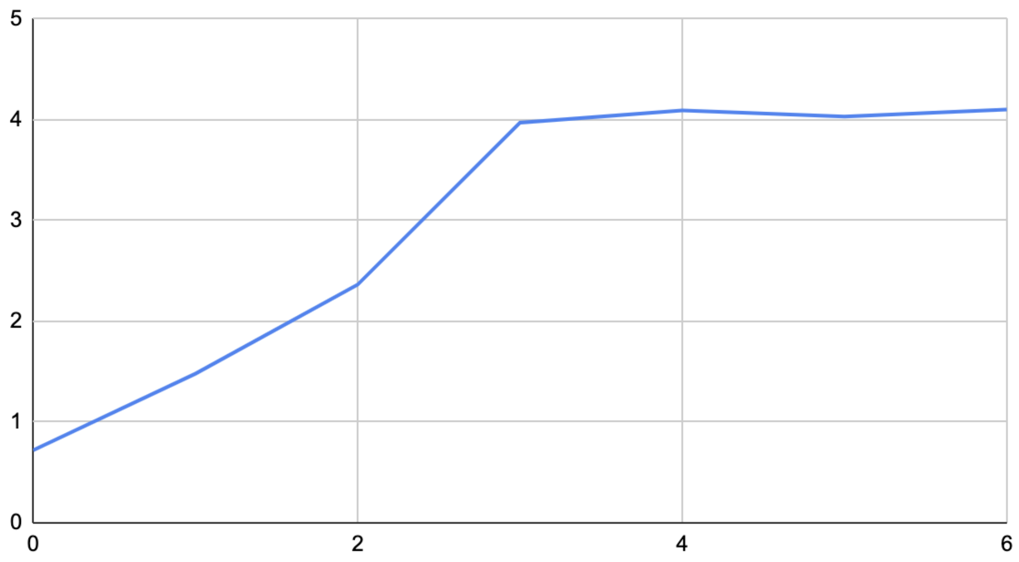

Figure 1: Time to Fill the Queue (seconds) vs. log_2 (Number of Threads)

Initially we have a roughly exponential increase in Time to Fill the Queue, flattening at around $2^3 = 8$ threads. Further testing on 9 and 10 threads confirms we have a flattening at this point.

```
8 3.972156s
9 4.050483s
10 4.042463s
```

I decided the optimal number of threads was 9.

Notes

It is important to note this is just testing with one type of packet, and if multiple packets were flooding the sniffer at any one time (say ARP and HTTP packets on `eth0`), more gains could be made by increasing thread count. This is because there are separate mutex locks for each different packet.

Correctness

To test the correctness of the solution, we:

1. start sniffing on `lo` and run

   (a) `python3 arp-poison.py`

   (b) `hping3 -c 100 -d 120 -S -w 64 -p 80 -i u100 --rand-source localhost`

   to send ARP reply and SYN request packets respectively to the loopback interface. Command **(b)** sends 100 SYN packets from spoofed IP addresses, so we could test if we were correctly counting distinct IP's.

2. start sniffing on `eth0` and run

   (a) `wget --no-hsts www.google.co.uk`

   (b) `wget --no-hsts www.facebook.com`

   to send HTTP request packets to the two blacklisted domains. Different domains were also used in the above commands to ensure we have no false positives being registered.

Memory leaks

We test for memory leaks using `valgrind --leak-check=full`, helping us remember to free memory we allocated on the heap and rejoin our threads.

We have bytes showing in the `still reachable` part of the valgrind analysis:

```
==1379== HEAP SUMMARY:
==1379==     in use at exit: 1,654 bytes in 4 blocks
==1379==   total heap usage: 28 allocs, 24 frees, 9,554 bytes allocated
==1379==
==1379== LEAK SUMMARY:
==1379==    definitely lost: 0 bytes in 0 blocks
==1379==    indirectly lost: 0 bytes in 0 blocks
==1379==      possibly lost: 0 bytes in 0 blocks
==1379==    still reachable: 1,654 bytes in 4 blocks
==1379==         suppressed: 0 bytes in 0 blocks
```

According to the valgrind documentation, `still reachable` involves one time memory allocations, references to which are kept for the entire process lifetime. When the process terminates, the OS will reclaim this memory. It is not a real memory leak and is seen as reasonable. This is most likely due to an error in the pcap library or due to threading.

945 words excluding code and output snippets.

References

- Valgrind
- Programming with pcap
- Linux man pages
- Stack overflow
- Terminating threads