Design Decisions

The analyse() procedure simply takes the memory location of the packet to analyse and extracts the appropriate information from each relevant header within the packet. Getting to a particular header in the packet can be done by adding the length of all previous headers to the memory location of the packet. The ethernet header is at the top of the stack of packets received, so it is handled first. It contains information about the protocol of the layer under it, which includes IP or ARP. If the next layer uses the ARP protocol, we must check if the packet is an ARP REPLY packet, which can be done by casting the pointer of the start of this header to a "netinet/ether_arp" struct and looking to see if the opcode stored there is equal to 2, then we must count it. Otherwise, the packet is an IP packet, and we must check for the other two conditions. The SYN flag is within the TCP header so we move to there and check that the SYN flag (and only that) has been set. If so, then count that packet and add the IP address of its sender to the array of addresses. The final check is ensuring that the host is not communicating with a blacklisted server which can be done by comparing the data in the HTTP header (header layer under the TCP layer if the packet is an HTTP header) with the names of the blacklisted servers.

Threading strategy and why it was chosen

The threading strategy used in the program is threadpooling. This describes a strategy in which a certain number of threads are initialised at the start of a program and each thread sleeps until a job is available for a thread to do. Another well-known threading strategy is the "one thread per request", where a thread is created whenever a job comes in and then joined back when the thread has completed execution. Creating a thread has a high overhead so creating many threads in a short timespan or in particularly bursty periods can slow down a system to a large degree. This is why threadpooling is deemed better than "one thread per request" (particularly for web servers, as those conditions occur frequently). It also allows for a bound on the number of threads that the program creates, however if the number of waiting jobs exceeds that bound (or every thread is busy) then a job will have to wait until a worker thread is available. This is why it is important to create a suitable number of threads on startup. Although threadpooling has downsides, the application must be able to handle all packets sent to and from the system without dropping any (to a reasonable degree) and the other model could lead to that happening.

Implementation of threading strategy

The threads are created and handled within "dispatch.c" using an initialise function which is called before pcap starts its capture loop. The number of threads is generally bounded below 10 as not much speedup came from deploying more. Each thread started and immediately went into waiting within "startThread()" on the condition that the job queue is empty. (CodeVault, 2021) The job queue is made up of references to different "queueItem" objects which held data as well as a reference to the memory location of the next queueItem. The data in these queueItems are structs of "tasks" defined as objects that held the pcap_pkthdr, the packet, as well as the verbosity of the application. Whenever a packet came in from pcap_loop(), the associated values are put

into a task struct object and sent to the “submitTask()” procedure, which is essentially the enqueuer or “producer”. The task is turned into a queue item (a struct which held the task data as well as a pointer to the next item in the queue) and then added to the tail of the queue. Any time the queue is handled, a mutex lock is also used to ensure that no two threads are writing to/reading from the queue concurrently. If this is allowed to occur, then the behaviour of the program is undefined and could lead to segmentation faults. The same is true for any global variable used by multiple threads, such as the SYN packet count variable in “analysis.c”. Each thread runs (either asleep or awake) until the user executes a keyboard interrupt (Arora, 2012) which the program then handles by breaking “pcap_loop()” and terminating each worker thread by breaking any sleeping thread out of their while loop. The application also then calculates how many unique IP addresses sent a SYN packet and outputs information according to the specification. Every function used in the signal handler was signal safe, except for printf(), however, we could guarantee that printf() is safe since it is called after the pcap_loop had broken and all child threads had completed.

Testing

Testing could be done using valgrind to check for memory leaks, as well as helgrind to check thread implementation. These were run using each of the tests given in the specification (SYN, ARP and Blacklist) to ensure there were no memory leaks or thread errors. There is an error in helgrind’s output as it displays an error every time a packet is added to the queue because it believes that the conditional lock signal is being sent when no thread holds that lock, but it that is not true otherwise the program wouldn’t run. The same is true for valgrind which contains 4 blocks of still reachable memory, however, the stack trace shows for each of these blocks that the is not being caused by any of the 4 files in src. The IP addresses output when a blacklisted URL is detected is incorrect and sometimes calculating the unique IP addresses can take too much time. These issues could not be fixed before submission. Otherwise, no other errors were encountered.