T2 2024

SIT325 Task 4.3HD

A Peer Support Strategy Based Implementation of Mitigating DDoS Attacks in SDNs

Visal Dam
S223058093

Contents

Introduction

The revolutionization of networking caused by the introduction of SDNs cannot be overstated, though its young state does bring about challenges that must be tackled before it is fully adopted. In light of this however, its issues can be studied by researchers and academics alike to mitigate any vulnerabilities it may harbor. In particular, the case of dedicated denial-of-service (DDoS) attacks against a switch's flow table, known as a flowtable overflow/flooding attack, and form of mitigation, were presented in [1]. This report seeks to implement the technique described in [1]; namely, a strategy that utilizes the entire network of switch's flowtable entries to exhaust the resources of an attacker. This report presents the logic behind the implementation, the challenges, the successful experiment and key notes and observations. Additionally, my tutor Ali told me that I am free to write as much as I want, and I sincerely apologize for the large word count; however, this task cannot be contained within 1500 words alone.

Implementation

Peer-Support Strategy (PSS)

My peer support strategy (PSS) revolves around three main functions: 1) getting the flow details from each switch; 2) determining a list of idle and 'full' switches; and 3) offloading flow rules from a 'full' switch to an idle one. Using the ONOS controller allowed me to get the stats of each switch via its REST API, which was done in Python via the "requests" library. This allowed me to make HTTP requests to ONOS, which runs on the browser. This is seen in the form of using the HTTP 'get', 'post' and 'delete' functions on crafted urls that point to the ONOS IP address.

My current resources allow a reduced attack rate (around 200 requests/second) which is less than the attack rate of the paper at 500 requests/second. Hence, I decided to reduce the value for the maximum

capacity to make up for it. Whether or not a switch is considered 'full' is if the number of flow rules currently in it is equal to or exceeds the set threshold, set to 2000; we say that it is 'full' because this is the state a switch needs to be in for it to be subject to the PSS. Technically, a switch is not full unless the number of flow rules exceeds the set maximum capacity, set to 4000, as noted in [1]; there are checks in place in the monitoring section to detect when it does exceed this limit, which promptly ends all processes as the network is assumed to be down and has thus lost to the DDoS attack.

The number of current flow rules is returned by the get_switch_flow_count() function. The values were returned by the controller as json objects and have been processed.

The find_peers() function returns two lists, containing the device IDs of idle and 'full' switches. It does so by first asking the controller to return the device IDs of all switches in the network, then by invoking the get_switch_flow_count() for each switch specified by their IDs. If a switch's flow count is greater than the threshold, it is appended to the list of full switches, and to the list of idle switches if not. It also returns a dictionary which contains all flows of the idle switches; this highlights an important part in my implementation in that it minimizes the use of http requests, which slows it down. Hence, I ensure that I make the most of each function. For example, getting a switch's flow count involves getting a list of every flow, and this list will be used later on.

Finally, the traffic_guider() function is the main traffic logic of my implementation. It is invoked when a 'full' switch is detected. The function installs a permanent wildcard flow rule into the victim switch which has no criteria thereby any incoming packet will match it, which outputs it to a peer switch; this peer switch is determined by any other switch directly linked to the victim switch. This is determined via the get_ports() function, which returns the first found connected switch and the port that leads from the target to the peer. The catch-all rule however should not forward lldp packets, as ONOS uses this to validate link sessions. If it takes long for an lldp packet to be seen along a path, ONOS deems the link inaccurate, which may arise if such packets were forwarded away as well. This is solved by setting its priority lower than the default flow rule to match lldp traffic.

Finally, the del_catch_all() fucnton searches through the flow rules of an idle switch and deletes any wildcard rule installed by the traffic_guider() function, as the switch no longer needs it and can begin to process traffic like normal.

These functions work together within a never-ending while loop in one script, ran in a separate window.

Flowtable Flooding

To simulate a flowtable flooding attack, I created a script that sends spoofed TCP packets using the "scapy" library, with randomized source and destination ports, that runs on a host on one switch and sends to another host on another switch. I also used TCP with the 'reset' flag so there is no need for the destination host to respond back, which doubles the traffic and makes the results inaccurate. I have also configured the Reactive Forwarding application in ONOS to 1) create flow rules based on IPv4 address and TCP/UDP ports and 2) to set a default flow rule idle time of 30 seconds. This way, each packet forces the creation of a new flow rule. This function runs in separate threads in a while loop, set to stop after 50000 packets – this is for testing purposes.

Experimental Setup

Parameters	Values
Maximum Capacity	4000
Threshold Capacity	3000, 3500
DDoS Rule Timeout (seconds)	30
Attack rate (average packet sent per second)	200

The experiment is carried out in an Ubuntu (version 24.04) virtual machine, assigned 4 CPUs at 100% execution capacity. We say that the DDoS attack succeeds if any switch at any point has a flow rule count of 4000 and above. My current resources only permit an average attack rate of 200 requests per second, but I made up for this by decreasing the maximum capacity. The topology used is a mesh topology with ten switches and one host per switch. This is important as this ensures that every switch has a route to another switch.

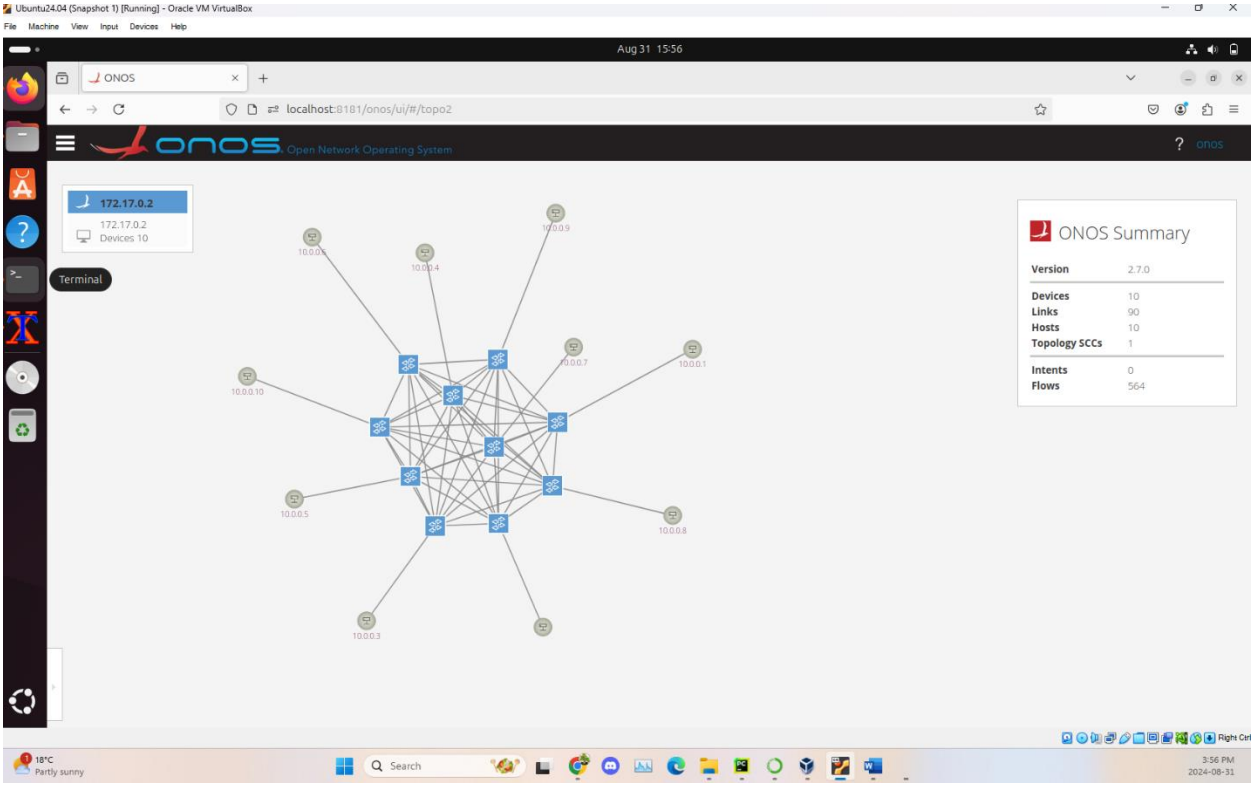

Figure 1: mesh topology used.

I also have another script set to prompt the ONOS controller every second (sleep=1) to query for switch statuses. This runs in a separate window for 300 seconds, after which it generates data to be graphed. Thus, there are three scripts running in total, and the impacts of this will be discussed. Furthermore, for experimental purposes I have set the maximum number of packets sent from the attacker to 50000. I believe that this is enough to demonstrate both the attack and mitigation scenarios.

We will assume that flow rules have a default 30s idle timeout, after which they are deleted naturally, achieved by changing the default Reactive Forwarding settings. We will also assume that each link remains valid throughout. Three main metrics will be used to prove the efficiency of the PSS: packet latency, packet loss and holding time. Here I define holding time as the time it takes for any switch to be

overpowered, i.e., reach beyond its maximum level. As both the attacking and defending parties share the same resource, I don't believe that CPU usage is a good metric, but I have included it regardless.

To measure for latency, specifically during an attack scenario with the peer support strategy turned on, I pinged the victim host from another non-attacking host. Similarly, I used Wireshark to track packet arrivals and measure packet loss. To measure CPU usage, I used the built-in Ubuntu system monitor tool, invoked via 'gnome-system-monitor'.

The theory behind the experiment is thus: a host h1 sends spoofed packets to another host h2 on the same network. These packets are processed by the switches s1 and s2 these hosts are connected to, and as they are spoofed the switches will create new flow rules. When s1 and s2 become 'full', the wildcard rule is installed in both switches that reflect traffic elsewhere to a different peer. A packet is sent from h1 to s1; s1 is full and the wildcard rule forwards the packet to s3; it cannot choose s2 as a peer because both s1 and s2 are in the full list, and it will only choose a peer from the idle list. The packet is received at s3, who makes a flow rule to process it. The packet has details that it needs to go to s2, so it directs it to s2; the mesh topology ensures that each switch has a link to every other. s2 is also 'full', the wildcard rule forwards it to another peer.

Say that during this scenario with the PSS, a legitimate packet is sent. The packet is thus forwarded around the network while s2 is processing old requests; also note that s2's flow level is not at its maximum capacity, but on average between the threshold and maximum (based on repeated observations). At any moment is s2 is below the threshold, no new wildcard rules are installed, and s2 begins to process all incoming flows, legitimate or otherwise. This looping of the packet is like a way of storing it within the network. Without the PSS, s1 and s2 reach their maximum limit and drop any and all incoming packets. Hence, with the PSS, latency is introduced rather than complete denial of service in the case without it, which is the goal of the PSS.

If the attack is completed and none of the switches reach their maximum capacity, then the implementation beats the attack entirely. If, however the attack goes on for much longer and with more resources (speed and number of attackers, say via a botnet), then the PSS at the current capacity increases the holding time of the network to a level greater than that without the PSS. This implies that by increasing the number of switches, and hence links, the holding time is increased further and further, to the point where an attacker would have to utilize greater and greater number of resources, which would be out of their power at a point for ordinary attackers.

However, it is important to note too that increasing the defending resources also increases the workload of the controller, which will reduce defending performance. One can mitigate this via means such as increasing controller capacity to reduce the workload, optimization of functions and leveraging new features of SDN; however, this is beyond the scope of this report and is only mentioned to show that any attack mitigation strategy has its costs and trade-offs.

Results

Without PSS

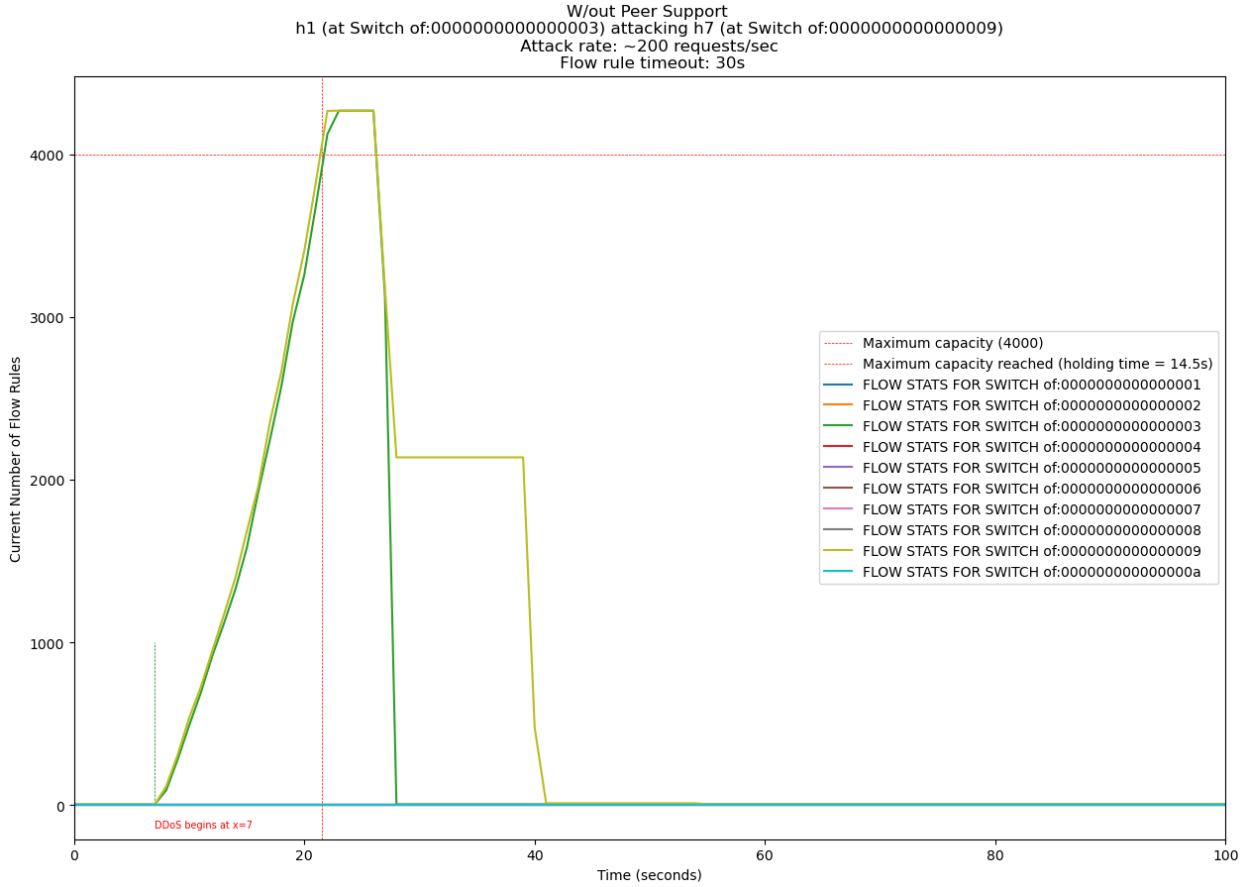

Figure 2: Flow rule count during an attack; note how it cuts off after reaching 4000.

The above graph shows the flow rule count for two switches under attack, switch of:0000000000000003 (green) and switch of:0000000000000009 (yellow). This is because h1 is sending spoofed packets at a fast rate to h7 and given that in the mesh topology there is a route from one switch to every other, only the link that links switch of:0000000000000002 and switch of:0000000000000008 is used. This is why there are only two colored lines shown, which is what we want.

As shown, the flow count reached and exceed the maximum level of 4000 at around the x=21.5s mark. The DDoS attack began at x=7s, meaning that the holding time, i.e., the time it takes for a switch to reach this limit, drop any incoming packets and thus destroying the network service, is 14.5 seconds. Note that in the graph above, not all 50000 unique flow rules are shown. To better mimic the effect of an overpowered switch, I have also implemented a function to install a flow rule (without any criteria for selection and instruction) to drop all incoming packets. This can be seen in the sharp drop after reaching the 4000 mark; all other packets were dropped.

Please note that in ONOS, the actual limit is very high, hence it is resistant to limited-resource flow table attacks. The correct interpretation of this result is that, when the switch has reached this limit, the flow level stays at that limit and any incoming packets gets dropped, until the previous flows die out and the new ones take their place. This effectively inhibits legitimate traffic, as processing packets, there are more packets coming from an attacker; thus, we say in this scenario that the switch is overpowered.

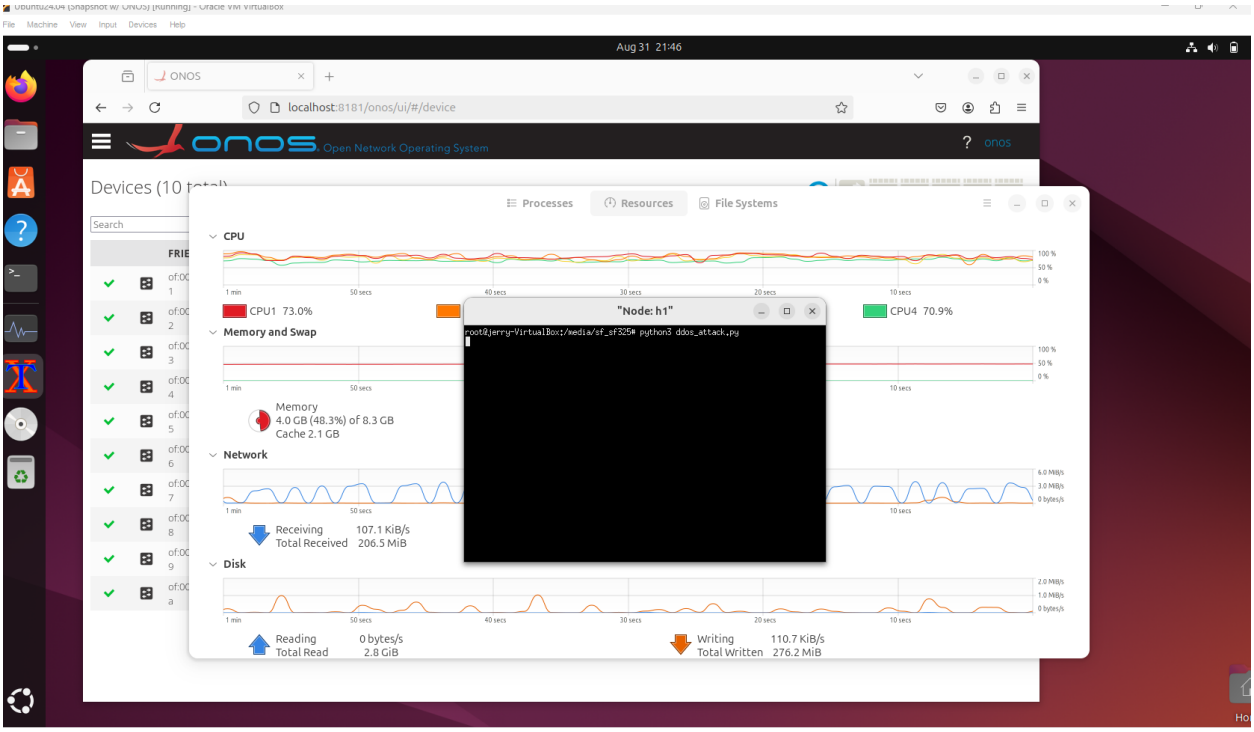

Figure 3: CPU usage during an attack.

The above figure highlights the high CPU usage during a DDoS attack. Note however that this is not an ideal metric given that both the attacking and defending parties are on the same machine.

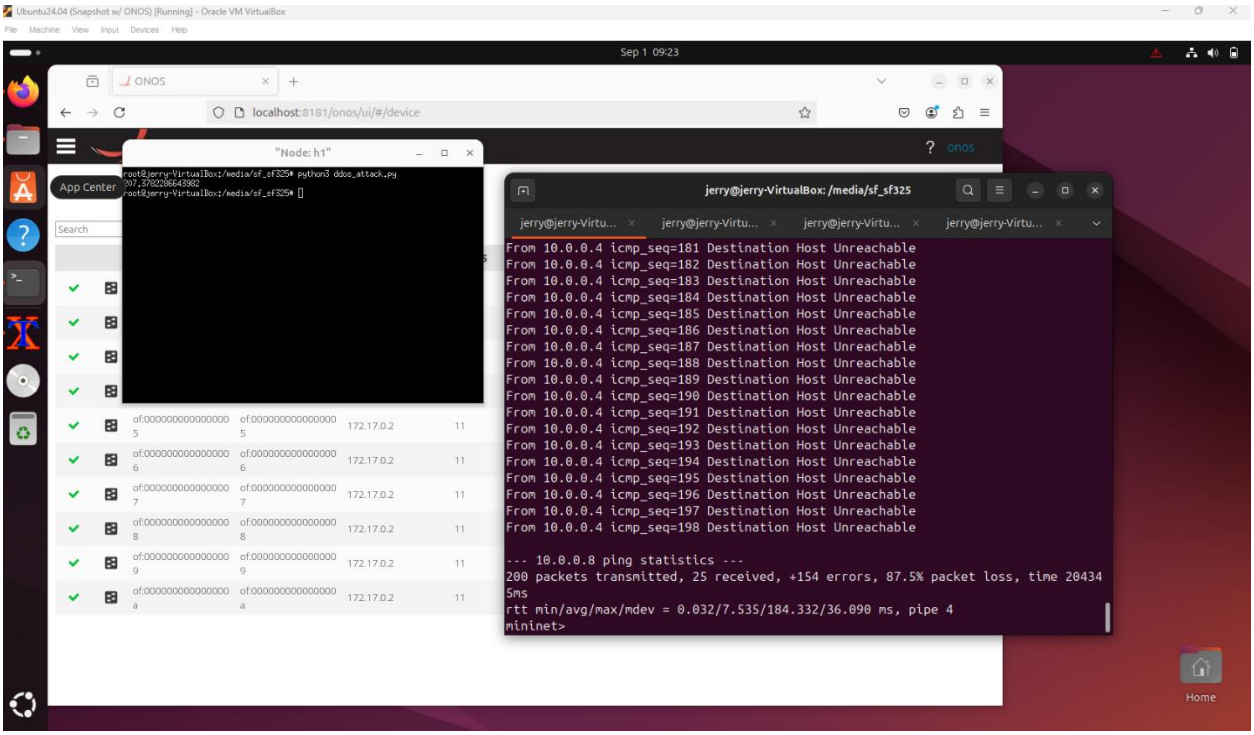

Figure 4: packet loss and latency during an attack without the PSS; the communication began 10s after the attack took place.

The above figure shows packet loss and latency during the attack. This was done by pinging h7 200 times, which is located at of:0000000000000009, from a non-attacking host (h3) and the ping began 10s after the DDoS attack started. All traffic was effectively cut off (because of our mimicking implementation), resulting in high packet loss at 87.5%. Any packets that may have arrived were transmitted before the maximum level was reached. In fact, communication was effectively cut off after packet sequence #25.

With PSS

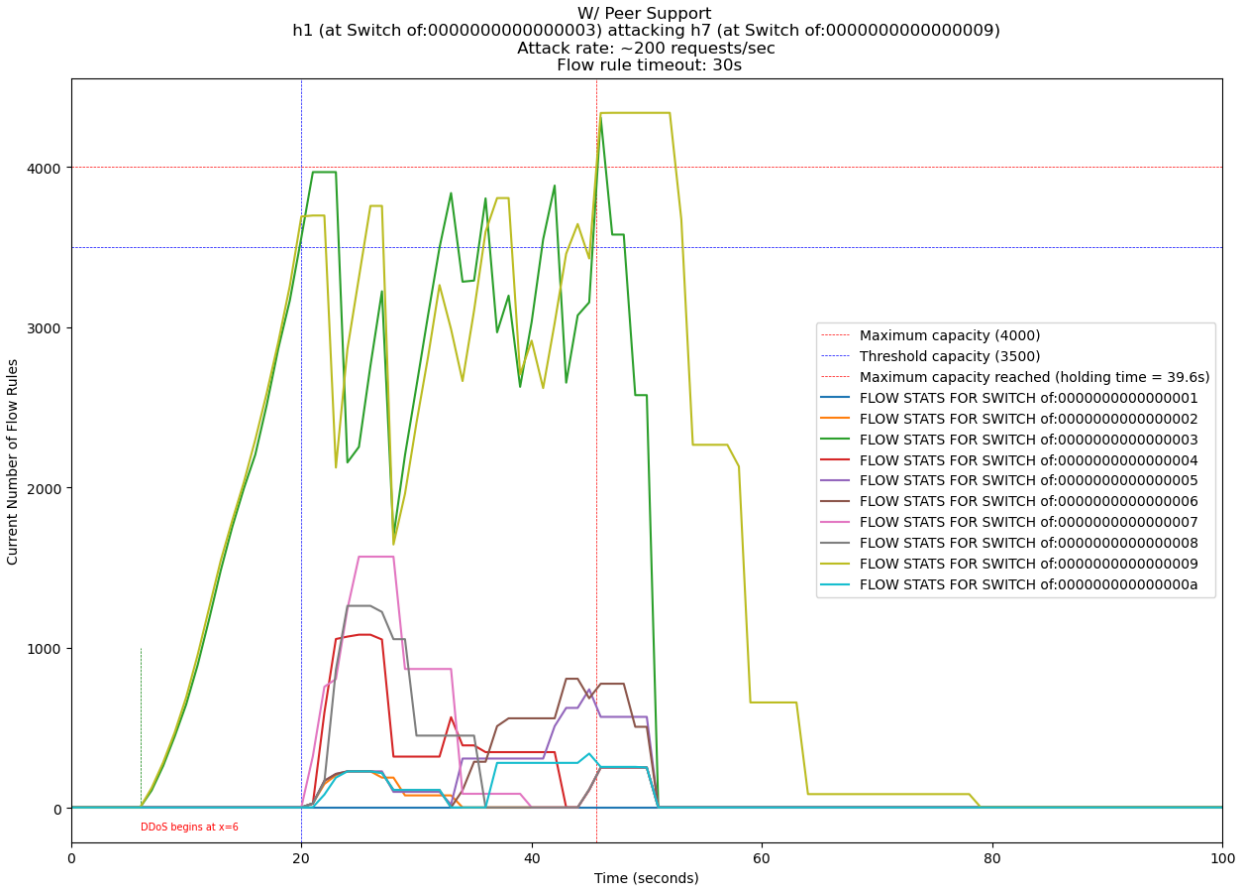

Figure 5: flow rule utilization across the switches.

The above graph shows the flow level during an attack with the PSS enabled. We see that after the threshold was reached the other switches had their flow tables utilized. Most importantly, we see that as the target switches reach back to normal numbers their flow tables increased; this is due to the previous packets that was 'stored' when it looped around the network. The maximum capacity was reached at around x=45.6; hence the holding time has been increased to around 39.6 seconds as the attack began at x=6. Note the dip and cut off when any of the switches, in this case, switch of:0000000000000009, goes up to and beyond 4000.

Note that this is when the threshold is at 3500. Let us investigate the case where the threshold is lower, hence giving the switches more leeway to respond.

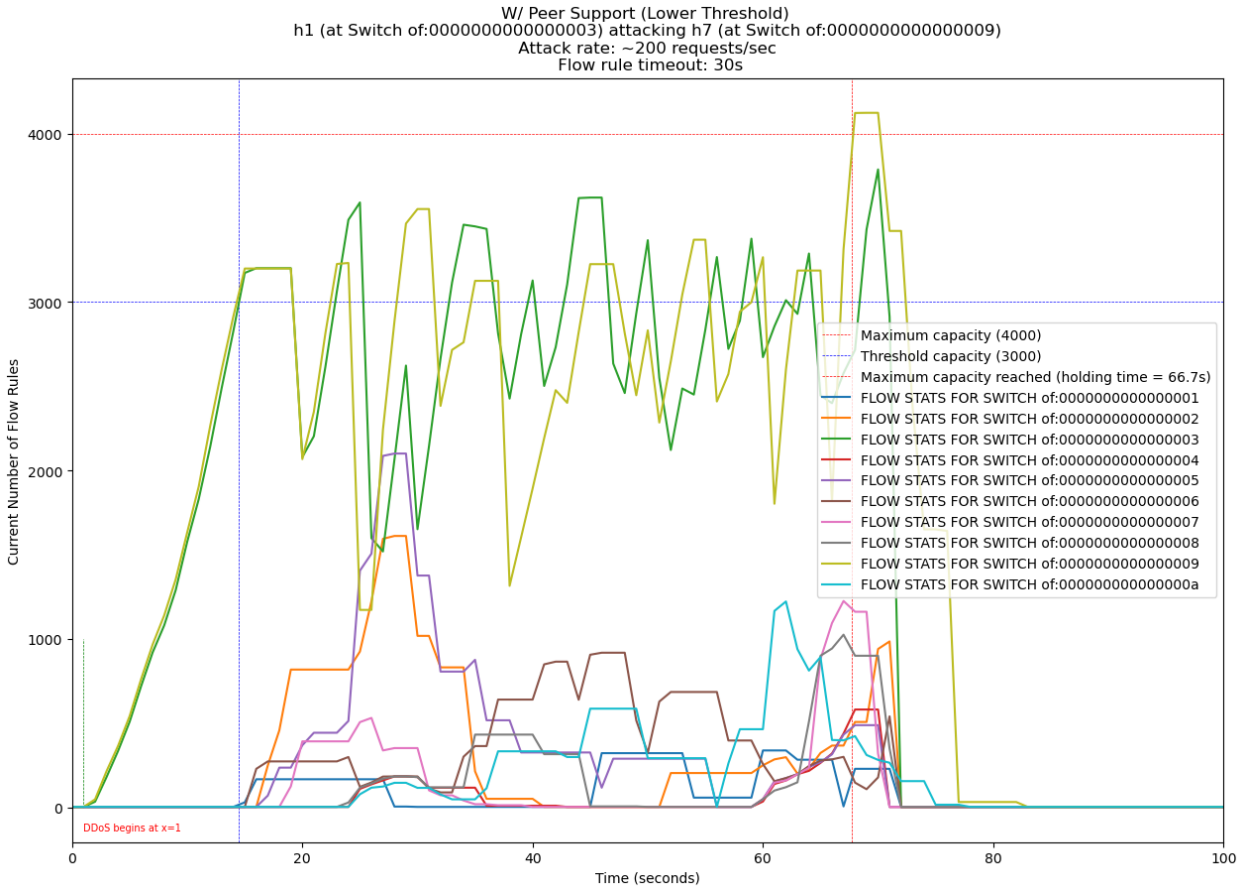

Figure 6: flow table utilization across the switches, at a lower threshold.

The above graph is the case where the threshold was lowered to 3000. This significantly increased the holding time to around 66.7s. However, this also means that essentially 1000 entries are wasted. A higher threshold would support legitimate traffic in the case of non-attack scenarios but would result in a lower holding time if an attack occurs. On the other hand, a lower threshold increases holding time but puts strain during normal operational time as the PSS introduces latency. This challenge can be mitigated by enabling the peer support strategy only when an attack is suspected; for example, logic can be implemented to measure the traffic statistics. If at any time flow enters the network at a highly unusual rate, the peer support strategy is enabled until traffic goes back to normal; this way, we get a balance of performance for both normal and attack scenarios.

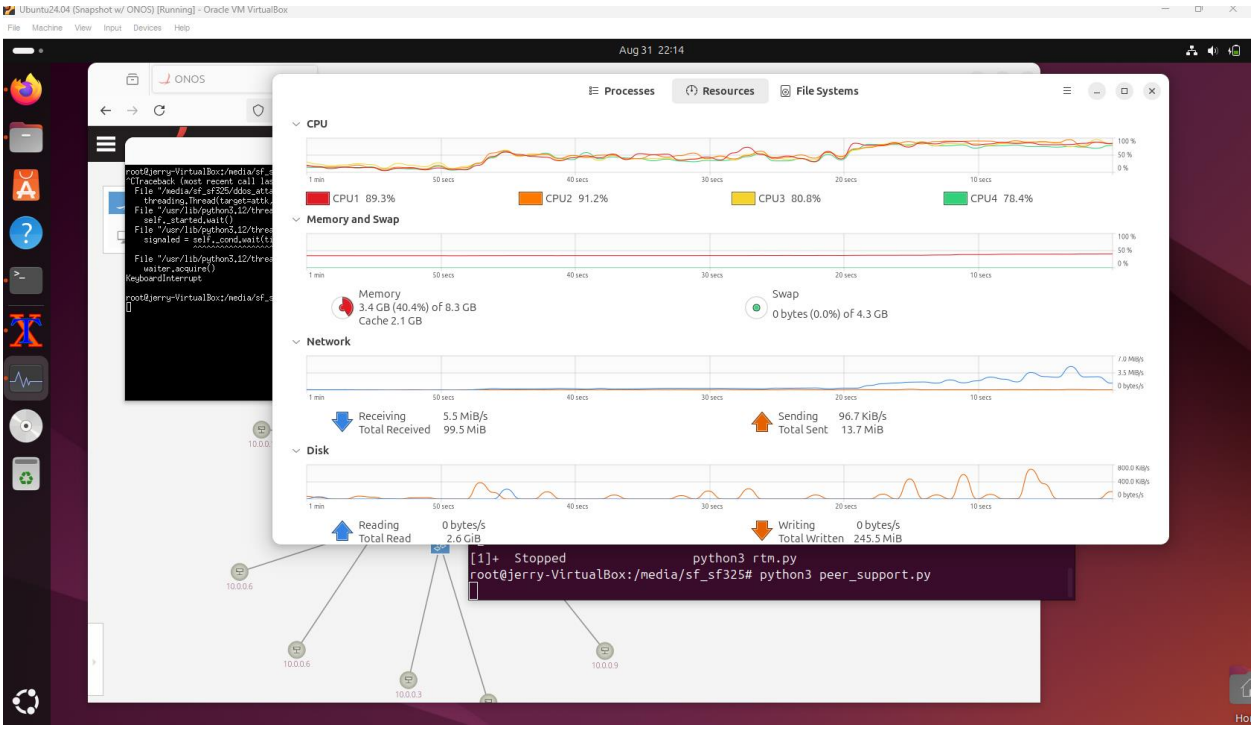

Figure 7: CPU usage during an attack scenario with the PSS enabled. Note the high CPU usage.

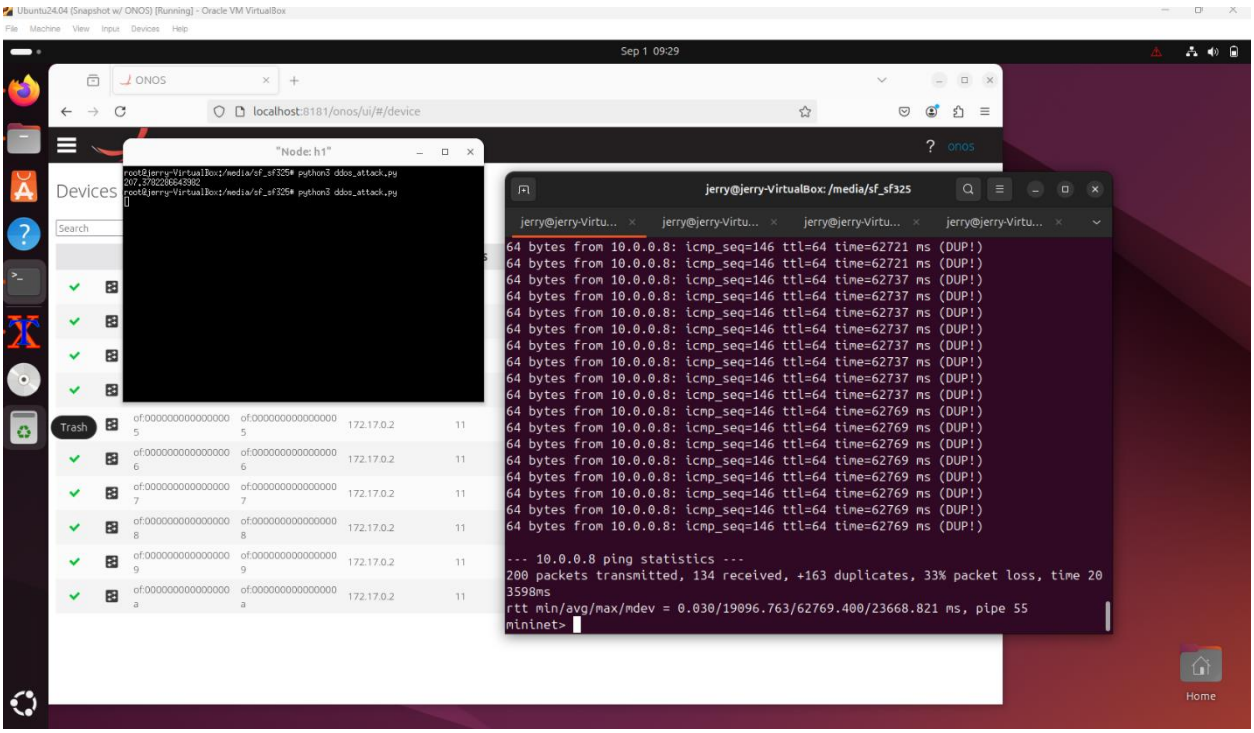

Figure 8: packet loss and latency during an attack with the PSS enabled.

The above figure is the same as Figure 4 but in the scenario where the PSS was enabled. There is much lower packet loss at around 33%. The threshold of the PSS enabled here was 3000. However, notice the high number of duplicates and extremely high latency. This is because of the latency and looping of

packets introduced by the PSS discussed earlier, which is still better than cutting communications entirely. This effectively showcases how legitimate packets are processed alongside the spoofed ones during an attack scenario with the PSS enabled, hence showing the effectiveness of the PSS.

Notes

It is important to note that all processes are run on the same machine. The attack, peer support, and monitoring implementation are carried out on the same VM which would muddy the results without a doubt. The peer support implementation is quite exhausting, so it would indeed interfere with the attack rate of the attacking host. I assume that the research paper carried their experiments via different machines, albeit connected to the same network.

Similarly, it is important to note that much of this experiment depends on the ONOS controller. It is prompted to get flow statistics for both the peer support strategy and to get the metrics, not to mention it also determines optimum routing paths by default, so it would be exhausted. However, this is consistent with real-world scenarios, so this factor is not impactful.

Finally comes the issue of scalability. Obviously, my limited resources limit the attack scenario to a lower attack rate. However, I believe that, in the ideal scenario where the attacking and defending parties use different resources, and that the controller is given more computational resources due to this, an attack of any scale by a normal adversary, can be fended off with acceptable performance. Note that increasing switch count also increases workload for the controller, as it has to check more switches. This can be mitigated by programming each switch to instead send SOS requests to the controller when its flow entries reach the threshold rather than having the controller constantly querying.

Link to Code Folder

The following links to my Deakin OneDrive folder with all the python scripts created for this task. This includes two main folders: 1) the final code; and 2) previous attempts.

LINK: https://deakin365-my.sharepoint.com/:f:/g/personal/s223058093_deakin_edu_au/EkrrvJU7Kl1BlK6fC8KN_HYBu1CFW8RcpMf-5O9mvLTWNQ?e=gcydND

Conclusion

This report aimed to show the successful implementation of a peer support strategy based on [1]. This was done using the ONOS controller, specifically its REST API. The methods to attack, mitigate, and monitor were coded in Python and ran as separate scripts. The experiment was done, and the metrics mentioned in the task sheet was presented for the system under both attack and attack-with-PSS conditions. The important tradeoffs, mainly computational performance, were discussed, mainly as the consequence of having both all functions sharing the same computational resource. Similarly, the impact of increasing switch count and the relationship this has with the overall strategy, in terms of both giving each peer more options to offload and increasing the workload of the controller, were presented in full. Furthermore, with advances in AI I believe a machine-learning based implementation to defend SDNs

against attacks is the more imminent strategy as opposed to this implementation which is costly and has many tradeoffs.

References

1. Yuan, Bin, Deqing Zou, Shui Yu, Hai Jin, Weizhong Qiang, and Jinan Shen. "Defending against flow table overloading attack in software-defined networks." IEEE Transactions on Services Computing 12, no. 2 (2016): 231-246.