Final Project Report
50.012 Networks
SUTD 2021 ISTD

Group 2: Netiquette - Does It Pay to Be Selfish?
A study of congestion control abuse and countermeasures

Huang He (1004561)
Han Xing Yi (1004330)
Qiao Yingjie (1004514)
Zhang Peiyuan (1004539)
James Raphael Tiovalen (1004555)
Velusamy Sathiakumar Ragul Balaji (1004101)

Table of Contents

1. Introduction

TCP congestion control is implemented at the edge of the network, which implies that the responsibility lies in the hands of the client to carry out the algorithm in good faith. As such, there is a possibility that it can be abused.

If some users bypass the TCP congestion control or use UDP instead, how will the network behave under different settings? Will those selfish users benefit from it, or will this lead to a complete collapse of the network?

In this research, we would like to find the optimum degree of selfishness in a network that can pay off (i.e., how selfish can one get before congestion collapse occurs).

2. Related Works

2.1 Introduction to TCP Variants

Firstly, we will discuss the various TCP variants with different congestion control algorithms that we will consider in our research.

2.1.1 TCP Reno

Variables that affect congestion control in TCP Reno in a sequential server model:

1. Queue length
2. Packet size
3. Number of hosts
4. Service rate, i.e. how fast are the packets sent over

TCP Reno is known for its conservative congestion avoidance policy. Its response function (average congestion window size in terms of packet loss rate) is:

$$W_{reno} = \frac{1.22}{p^{0.5}}$$

which means TCP Reno places a serious constraint on the congestion window size, especially when the loss rate is low. As written in (Jama & Sultan, 2008, 30):

> *For example, for a TCP Reno connection with 1500-byte packets and 100ms RTT, achieving a steady-state throughput of 1Gbps would require an average congestion window of 8300 segments, and an average packet loss rate of 2×10^{-8}. This requirement is unrealistic in current networks. The congestion window takes more than 4000 RTT to recover after a loss event which prevents efficient use of the link bandwidth.*

2.1.2 CUBIC TCP

Under TCP Reno, the window size can only increase or decrease when expected or duplicate ACK is received, and the time taken by this process is determined by RTT. CUBIC TCP (Rhee and Xu, 2008), however, keeps the window growth rate independent of the RTT, making this protocol friendly to both short and long RTTs. Its window size is determined by the following equation:

$$W_{cubic} = C(t - K)^3 + W_{max}$$

$$K = (W_{max} B / C)^{1/3}$$

where B, C are constant factors, t is the time elapsed from the last window reduction, W_{max} is the window size just before the last reduction.

2.2 Existing Research on Selfish Network Behavior

We will now discuss the currently existing research that has been done on the topic of selfish network behavior.

2.2.1 A Mathematical Model for the TCP Tragedy of the Commons

The model proposed (López et al., 2005), based on Game Theory principles, features a simplified version of TCP that uses time-division multiplexing for its hosts to send packets into the network. Fair hosts send packets only during their time slot, while evil hosts send packets outside of their time slot with a probability of p. Their model proves that there exists a threshold whereby evil greediness will cause the game to take the form of “Tragedy of the Commons”. The model also shows that past this threshold, the performance of the network decreases proportionally to the number of evil hosts and the square of the number of fair hosts. They also found that as the tragedy becomes more significant, the incentive for users to be selfish tends to be lower.

2.2.2 Selfish Behavior of TCP Users Using Multiple Parallel Connections

A game-theoretic framework is used in this paper (Zhang et al., 2005) to evaluate the impact of users' greedy behavior when allowed to open multiple TCP connections to maximize their own utility. The authors found out that there is a Nash Equilibria and the overall network efficiency loss is bounded when given either a user's computation powers (the number of connections a user can open) is limited or the user is socially responsible (modelled by users being aware of the system-wide operating cost which is proportional to the sending rate of all connections). The authors also showed empirically that a user with greater computation power is able to obtain greater goodput at the Nash Equilibrium state.

2.2.3 TCP with Misbehaving Receiver

In this paper (Savage et al., 1999, 71-78), the authors explored 3 possible attacks that a receiver can exploit on the TCP Daytona congestion control behavior by abusing the frequency of ACKs the receiver sends back to the sender, which allows the receiver to drive a standard TCP sender arbitrarily fast, without losing end-to-end reliability, at the cost of other receivers. These exploits were then demonstrated to be exploitable on most OS-es back then. This paper shows the possibility of capitalizing on the congestion control algorithm that TCP adopts to suit a selfish agent’s needs, and hence, it is relevant for our study as it shows a possible technique to use in our study.

2.2.4 Existence of Tolls for Selfish Users in TCP Networks

In this paper (Fleischer et al., 2004, 9), the authors proved the existence of tolls when heterogeneous network users independently choose routes minimizing their own linear function of tolls versus latency to collectively form the traffic pattern of a minimum average latency flow. In simpler terms, it proved that it does pay to be selfish and that there is mathematical proof of optimum tolls in general congestion games.

3. Research Gap & Hypothesis

From the above literature review, we know that there is a certain threshold whereby the network suffers irrecoverably from congestion.

Research Gap:

1. For the paper “A Mathematical Model for the TCP Tragedy of the Commons”, López et al. only researched a simplified version of TCP that uses time-division multiplexing circuit switching techniques and assumes no loss of generality. It did not investigate whether the Tragedy of the Commons phenomenon would occur in a packet-switched network that is used widely today.
2. In the paper “TCP congestion control with a misbehaving receiver”, Savage et al. did not determine the best optimal parameter values for which the 3 attack techniques can be conducted. It also did not explore what would happen to the attacker’s throughput in a more realistic scenario, such as in a network with a star topology.

Hypothesis:

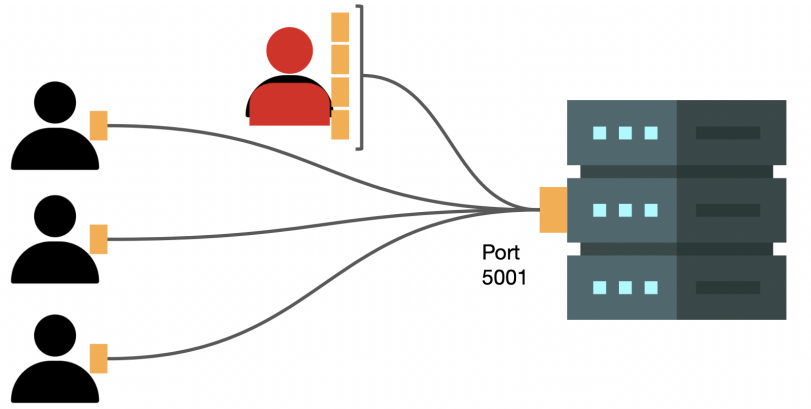
1. If a selfish user decides to bypass or modify the TCP congestion control mechanism in order to get higher throughput, as long as he is not asking for too much, other normal users in the network will only not be affected so badly and the selfish user can benefit from his behavior.
2. A single large flow of traffic in a TCP connection could result in other users getting little or no bandwidth at all. The network may become so congested that even the attacker is influenced. As a result, the selfish user may no longer benefit too much from his attack.
3. Combining 1 and 2, we speculate that there is an optimal degree of selfishness for a selfish user to benefit the most while others are less affected.

4. Experiments & Results

In this section, we will detail how we implemented the three attacks listed above, as well as showcase the results that we have obtained.

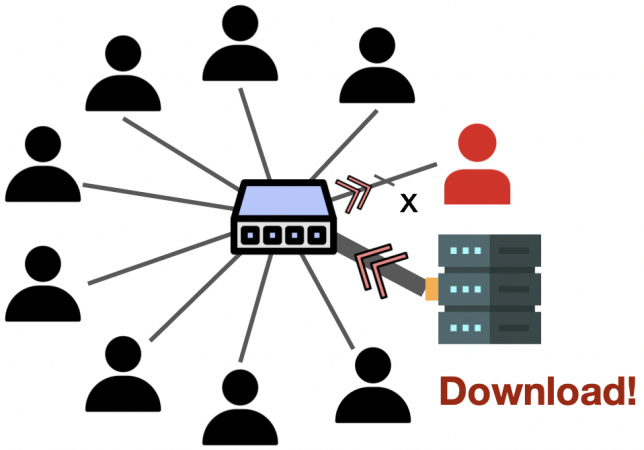
4.1 Opening Multiple TCP Parallel Connections

This attack involves opening up multiple ports on the attacker host and establishing many TCP connections with the server, in hopes of raising the average download throughput for the attacker.



4.1.1 Approach & Methodology

For this experiment, we first started out implementing a simple star topology, as shown in the figure below. Our network consists of 8 normal users, 1 selfish attacker and 1 server for a total of 10 hosts.

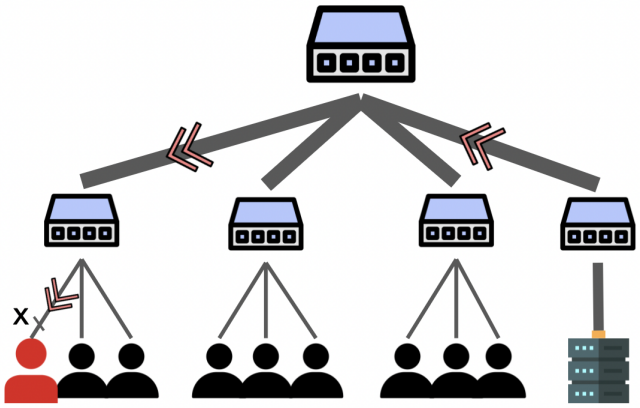


To open multiple parallel connections with the server, we made use of the `iperf` command that has the `-p` option to set the number of parallel connections to be established.

We then followed the following steps to achieve our results:

1. At the start of each test, we used `pingall()` to test if all the hosts in the network are reachable.
2. We then ran tests with varying parameters (described in section 4.1.2).
3. After collecting all the .csv output files produced by iperf, we used python to do data analysis and plot graphs to spot trends.
4. We also increased the number of parallel connections drastically so that the network would congest. This will be discussed in more detail in section 4.1.3.

Then, we implemented the Tree Topology, utilising the same methodology as described above:



The Tree Topology consists of individual switches to support 3 hosts each and an external switch to support the whole network. For this design, we set switch-to-switch bandwidth to be large enough so that it will not bottleneck the download throughput from server to host, in order to achieve the same network environment as that implemented in Star Topology.

4.1.2 Experiments Conducted

For the Star Topology, we had played around with several experiments first that we will not elaborate on in detail here (you may find the experiment results in our provided GitHub link in Section 5.4).

After running many small tests to observe and derive general trends, we ran a final large experiment and varied several parameters as shown in the table below with a standard test time of 5 seconds:

Parameter	**Values**
Consumer Link Bandwidth (for attacker and normal hosts)	[2, 4, 6] Mbps
Producer Link Bandwidth (for server)	[10, 16, 24] Mbps
Number of Parallel Connections	range(1, 250 + 1, 5)
TCP Congestion Control Algorithm	Reno, CUBIC, UDP (removed later)
Congestion Window (CWND) Size	[16, 64, 256, 512, 1024] KB

This experiment took around 12 hours to complete on an AMD Ryzen™ 9 5900X with 32GB of RAM. (It still only ran serially due to Mininet's limitations.)

For the Tree Topology, since we already had the results from prior experiments, we were able to quickly narrow down on the parameters that we wish to vary. To ensure consistency, we used the same parameters as the one listed in the table above, except we only ran it with consumer bandwidth = 2 Mbps due to time constraints. As the time taken to run pingall() for the tree topology was significantly larger, this experiment also took about 12 hours to complete.

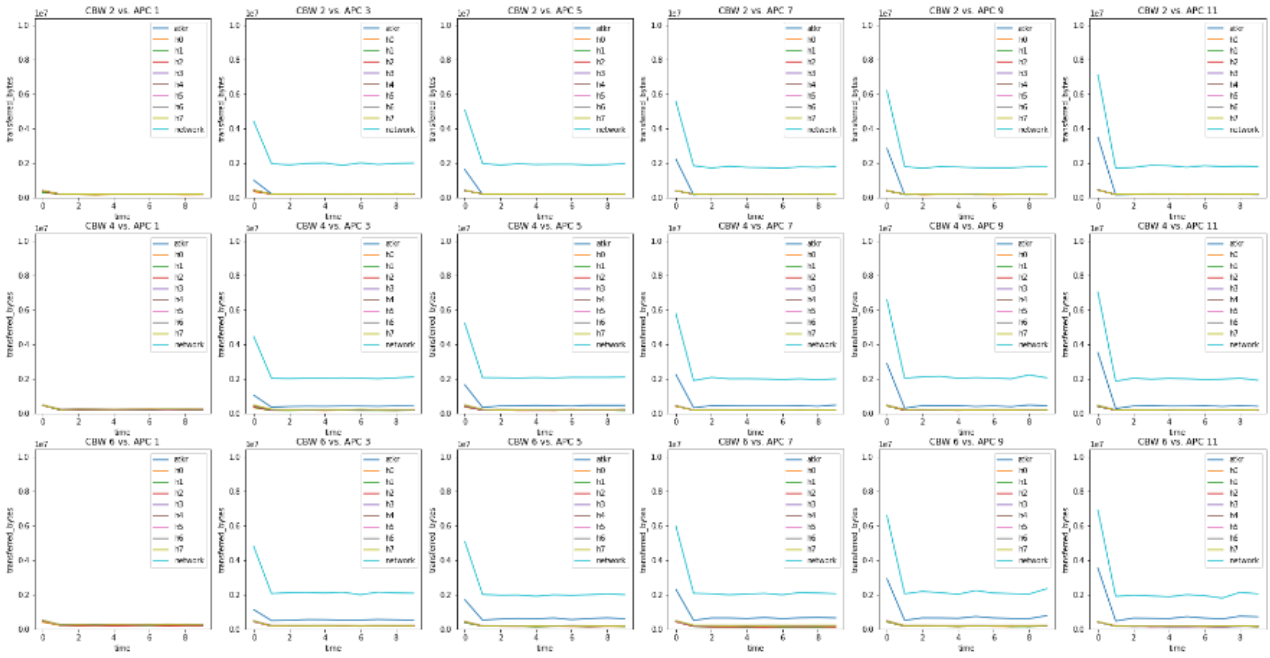
4.1.3 Results

For this section, other than Trivial Temporal Analysis, which was plotted as bytes transferred against time, all the values plotted were obtained bytes transferred against attacker parameter from the average of all iperf files that were generated by the same host for all the 9 hosts in the network (8 normal users, 1 attacker). This averaging was done to reduce random error by amortization over thousands of experiments.

For the following plots, the entire network throughput is represented by the 'cyan' line. The attacker's throughput is always represented with the 'dark blue' line, while the throughputs of the other hosts (except server) are represented by other colours.

4.1.3.1 Trivial Temporal Analysis

Firstly, we plotted the experiments individually to see each hosts' share of the bandwidth change over time. This resulted in the following graphs, which gave us unsurprising results which agreed with our prior expectations:



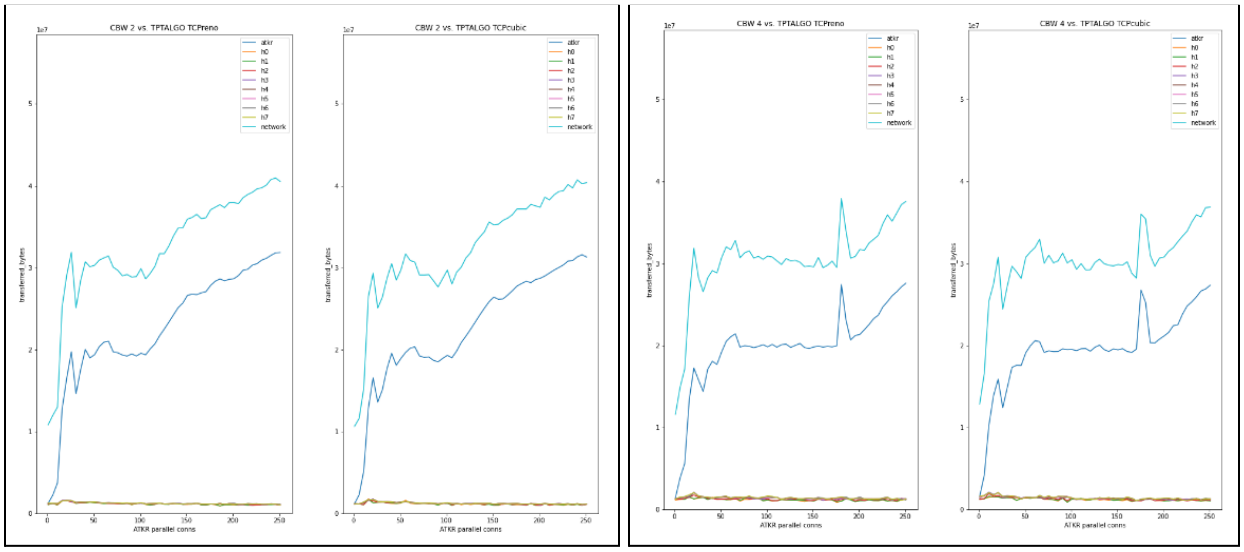
⟶ ⟶ ⟶ **Attacker Parallel Connections Increases** ⟶ ⟶ ⟶

We can clearly observe that when the attacker has only 1 connection to the server (control experiment), the throughput obtained by all hosts, except the server, are approximately the same. As the number of parallel connections increases, we see that while the attacker has a significant advantage in the first second or so, the attacker quickly converges with the rest of the network, with a final stable throughput proportional to his number of parallel connections.

One possible reason for this above observation would be because of TCP fairness, whereby multiplicative decreases enable all hosts in the network to achieve similar throughput in the long run.

4.1.3.2 Comparing Transport Congestion Control Algorithms

Next, we decided to look into how TCP congestion control algorithms affect the download throughput of the attacker (y-axis) against the number of parallel connections established (x-axis).



TCP Reno **TCP CUBIC** **TCP Reno** **TCP CUBIC**

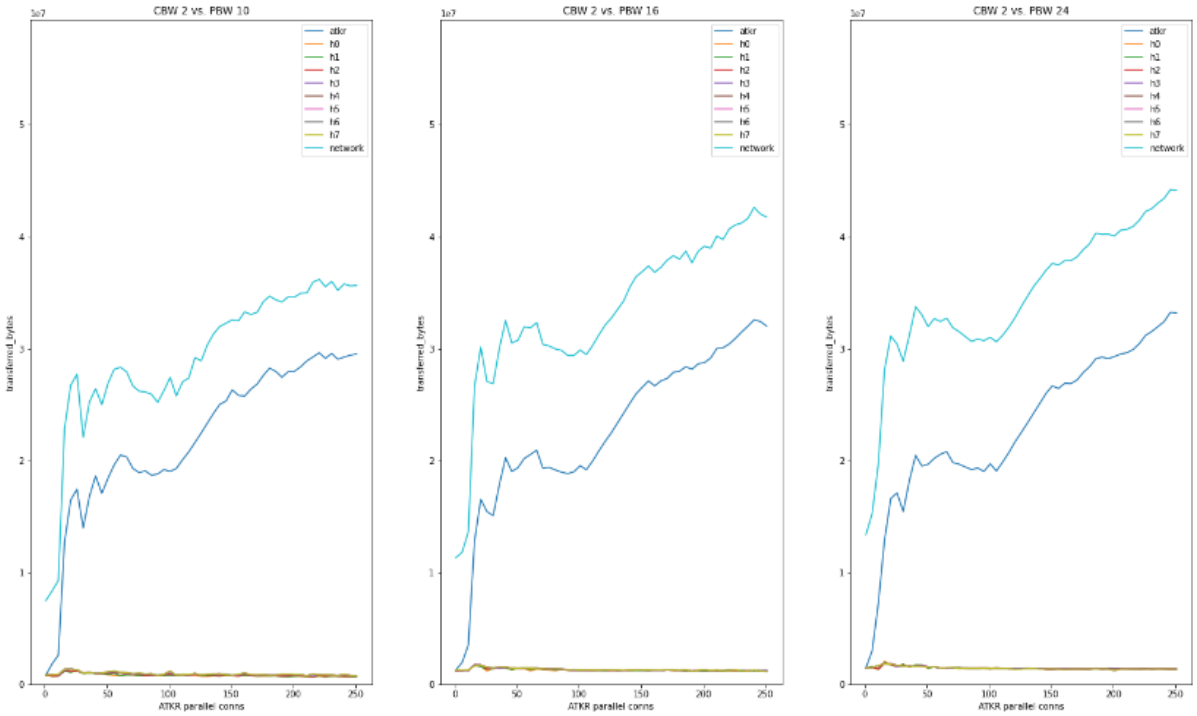
For each respective graph, TCP Reno is employed on the left-hand side, with TCP CUBIC on the right. We observe that while there are slight differences between TCP Reno and TCP CUBIC, the general trend for both graphs is very similar. This means that the congestion control

algorithm used does not cause a huge difference in the download throughput as the number of parallel connections increases.

4.1.3.3 Varying Bandwidths

Since we realised that the congestion control algorithm results in similar download throughput, we decided to explore further factors on what would affect the throughput. Thus, we decided to vary both the consumer and producer link bandwidths. We chose parameters in a way such that at higher consumer bandwidths and lower producer link bandwidths, the total possible theoretical consumer throughput would overwhelm the producer link bandwidth in an attempt to cause network collapse.

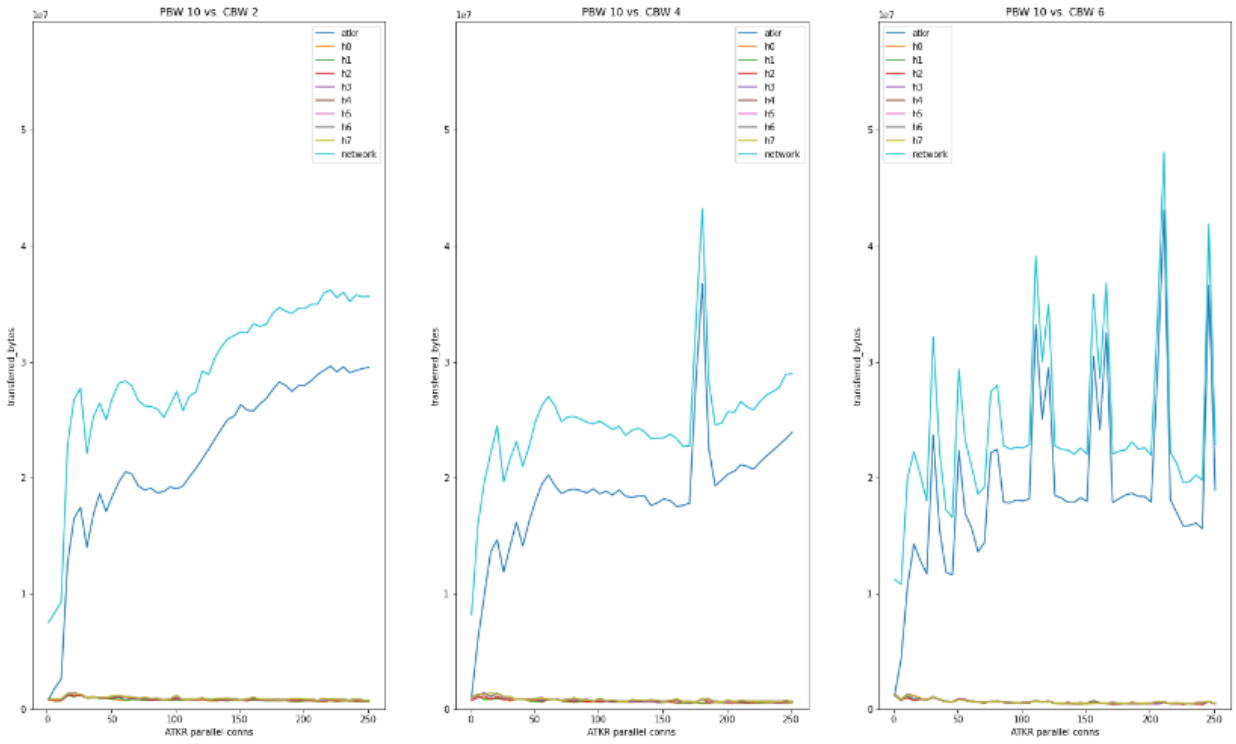
As you can see in the figures below, as the producer link bandwidth increases, the throughput for the attacker remains fairly similar across all three graphs. This suggests that the attacker is probably saturating his own link, which is constrained by the consumer link bandwidth.



⟶ ⟶ ⟶ PBW increases while CBW remains constant ⟶ ⟶ ⟶

On the other hand, we also varied the consumer link bandwidth while keeping the producer link bandwidth constant. The results are shown in the figure below. As we would expect, the attacker is now better able to monopolise on the server's available bandwidth as his own link bandwidth increases.

In the plots, we realise some strange phenomenon – the presence of sudden throughput spikes in the network. We hypothesise such a phenomenon might be due to the attacker flooding the network with packets. This causes all the buffers along the way to be filled with only his packets, while packets from other normal hosts are dropped. This results in a more extreme monopoly by the attacker on the server's service as other hosts have their packets dropped due to full buffers. We would need time to do this as future work to prove this hypothesis definitively.

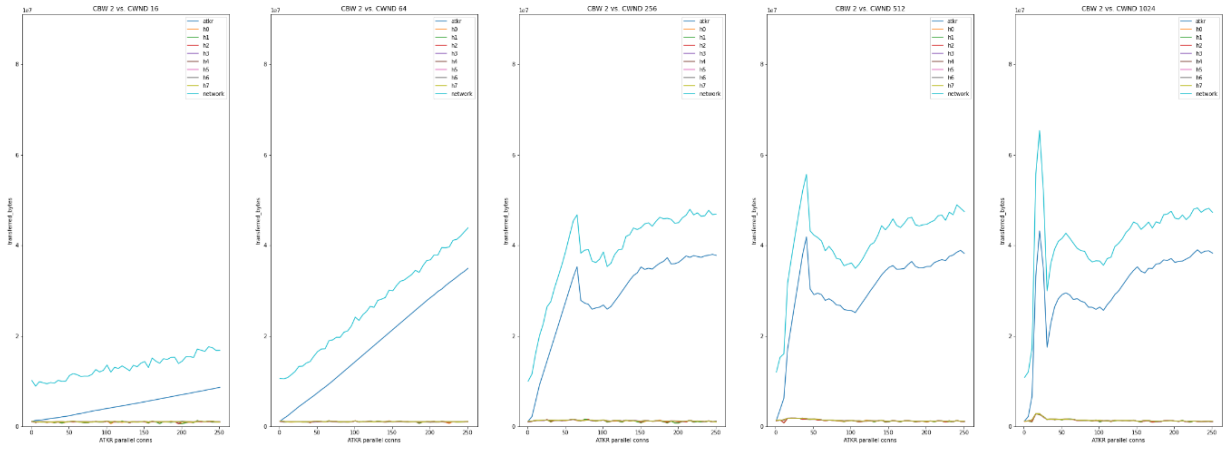


⟶ ⟶ ⟶ CBW increases while PBW remains constant ⟶ ⟶ ⟶

4.1.3.4 Changing Congestion Window (CWND) Size

We then investigated the effects of CWND on the amount of data transferred in the network to find some interesting results. We varied CWND from 16KB to 1024KB, as stated in section 4.1.2. When buffers are small, we can see that the TCP congestion control algorithms and TCP fairness makes it hard for the attacker to gain a massive advantage relative to the rest of the hosts. As these buffers grow larger, we see the emergence of what we think might be the bufferbloat problem (Cerf et al., 2012).

Bufferbloat is a phenomenon whereby network congestion causes long queueing delays to be experienced by packets in unnecessarily large buffers. This decreases the overall network throughput but ironically, such a phenomenon benefits the attacker disproportionately. He is now able to receive a larger share of the network's total throughput and total goodput, as bufferbloat slows down the effects of TCP fairness and congestion control. When the buffers are big, we also observe the emergence of a distinct threshold at which the attacker cannot simply keep increasing his parallel connections and expect the same performance gains. The point at which diminishing returns dominate indicates that this is a threshold at which an attacker should operate, in order to achieve optimum goodput and not waste system resources on opening more ports.

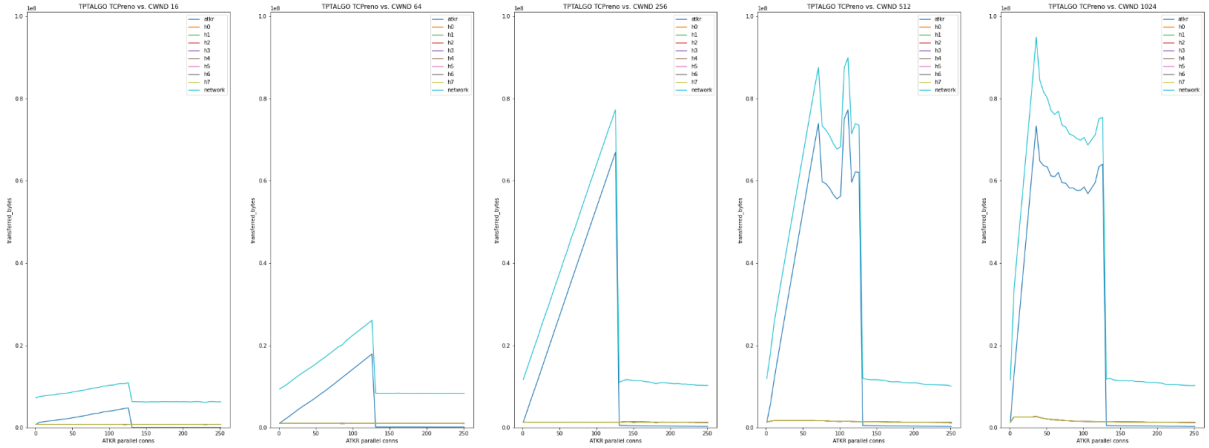


⟶ ⟶ ⟶ CWND increases, other factors constant ⟶ ⟶ ⟶

4.1.3.4 Tree Topology

We observed similar results for the tree topology, which suggest that our conclusions have no loss of generality across network topologies, although peer review is always welcome. We would also need to conduct more conclusive tests to prove such a claim.

A particularly interesting observation for the tree topology was that the network almost always completely collapsed once the attacker became too greedy (more aggressive in his attack by our definition), hence causing the network and the attacker to both lose out overall. This suggests that for realistic and complex topologies there also exists an upper bound on the threshold for the attacker's attack intensity so as to not cause a huge disruption to the network. See figure below for our evidence of how attacker throughput fluctuates majorly at higher attacker intensities in tree topologies:



⟶ ⟶ ⟶ CWND increases, other factors constant ⟶ ⟶ ⟶

Further experiments with more sophisticated tools might need to be employed in order to investigate such a result further. This is because, at around 130 parallel connections, Mininet is unable to simulate any more parallel connections. Hence, from 130 parallel connections onwards, we can notice a sudden drop in the network as the attacker throughput dips to 0.

(Note that such a situation also occurs for the Star Topology, but at around 250 parallel connections established with the server).

4.2 Aggressively Sending Packets into Network

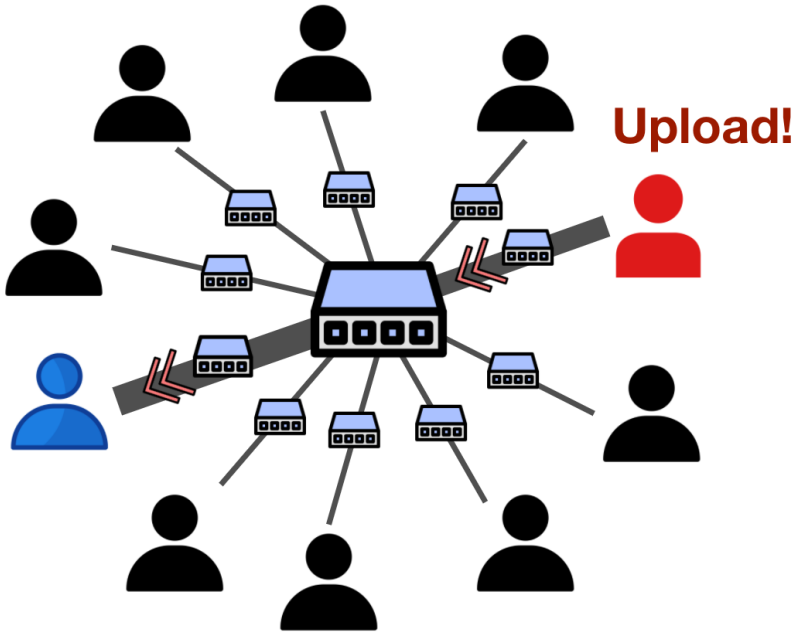
4.2.1 Approach & Methodology

In this approach, we aim to verify our 2nd hypothesis: A single large flow of traffic in a TCP connection could result in other users getting little or no bandwidth at all. To simulate this scenario, we decided to introduce an attacker that is aggressively sending packets into the network.

4.2.2 Experiments Conducted

We conducted experiments on 2 topologies: tree and star. In all experiments, there are 10 hosts in total, in which there is 1 attacking client, 1 receiving server, and 8 normal clients.

The star topology is designed as shown in the figure below:

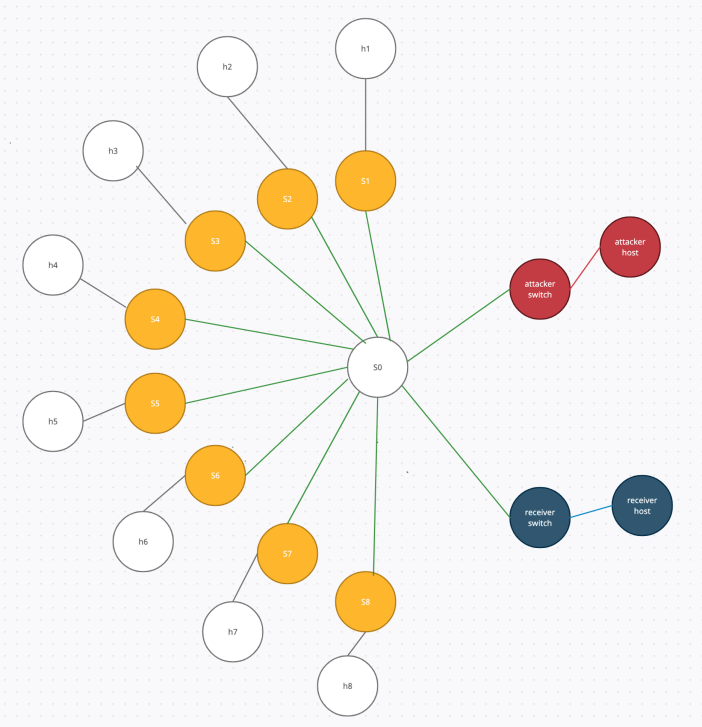


As shown in the illustration, The attacher (denoted in red) aggressively sends data to the server (denoted in blue), 8 other normal users (denoted in black) also try to send data to the server.

In the star topology, there is 1 switch between each host and the central switch. This is because each host is set to be sending data as fast as possible, and the link between (A) host and switch and (B)

switch and central switch is of different bandwidths, where the bandwidth of link A is smaller or equal to the bandwidth of link B. The link size and the switch to differentiate each link between each host and the central switch acts as flow control so that each host is sending/receiving data at a different rate.

To illustrate the idea of different link sizes, please see the figure below:



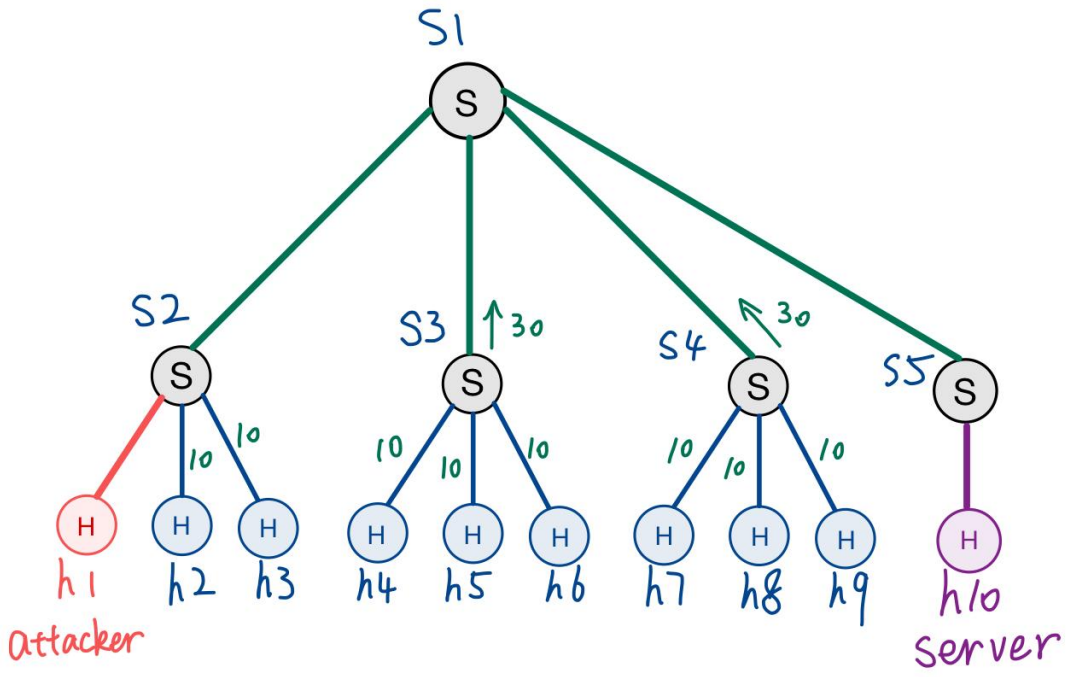
This represents the link-switch relationship more clearly, in which the red nodes are the attacker and its switch, the blue nodes are the receiver and its switch, and 8 white nodes as normal hosts and 8 yellow nodes as their switches.

The green link between each individual switch and central switch is defined as Infra Link: it has a very large link size so that it will not congest in any case. This is an important invariant in our experiments.

For the red attacker sender with its red attacking link, and the white normal sender with its black normal link, we will vary the red attack

link size to simulate different attacker sending rates, as in how aggressively the packets are being sent in order to abuse the network. The blue receiving link will also be varied in size to find out the network behaviors given different attacking intensity and the different sending relationship between attacker and normal years. This receiving link is where congestion is meant to happen.

For tree topology, we employ a similar design. As shown in the illustration below, we use a 3-layer tree with a fanout of 3 hosts on each node. Different from the star topology where each host has its direct link to the central switch, in tree topo, we expect the aggressive traffic could be regulated by the second-layer switches and multiple hosts will be fighting over the bandwidth of the inter-switch link.



We use multiple nested for-loops to vary the six-dimensional parameters, including a larger server-side link, which is illustrated in purple, to simulate a large content provider and a smaller server-side link to simulate a normal application server. We vary the bandwidth of the attacker-switch link to adjust the intensity of the attack.

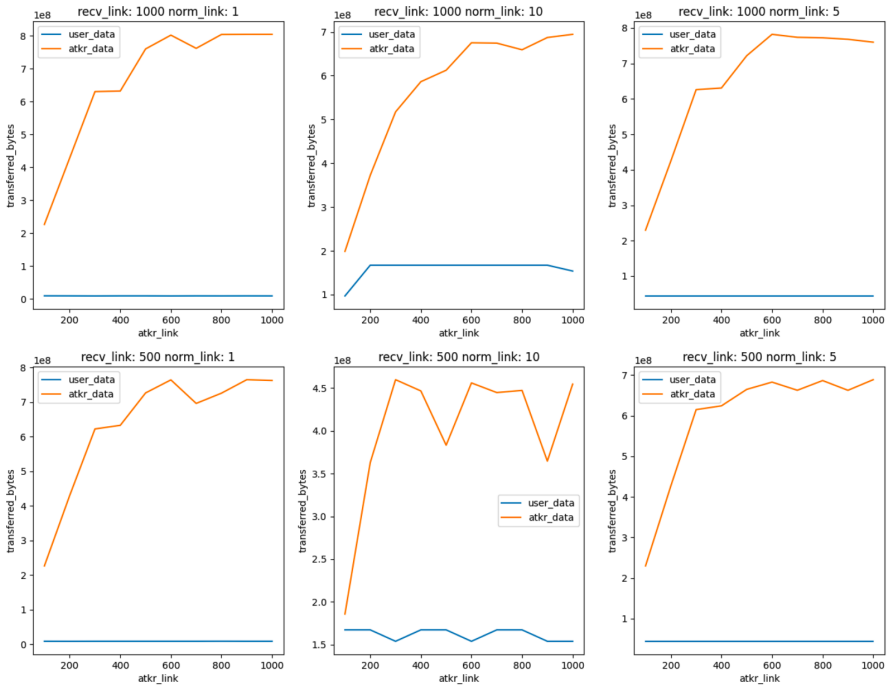
Parameter	Values
Server-Switch Bandwidth	[100, 500] Mbps
Attacker-Switch Bandwidth	[10, 20, 50, 200, 800] Mbps
Link Prop Delay	[2, 50] ms
TCP Congestion Control Algorithm	Reno, CUBIC
Congestion Window (CWND) Size	[16K, 1M]
Test Time	[10, 30] seconds

4.2.3 Results

4.2.3.1 Star Topologies

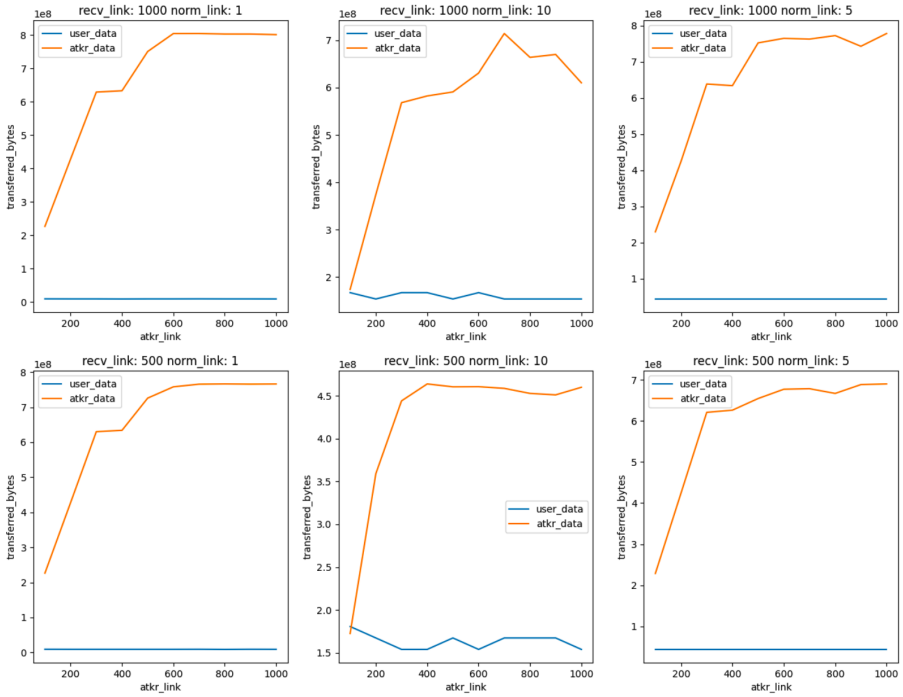
For each graph below, the receiver link and normal user link bandwidth are fixed, and the attacker link bandwidth varies.

TCP Reno, uncongested:



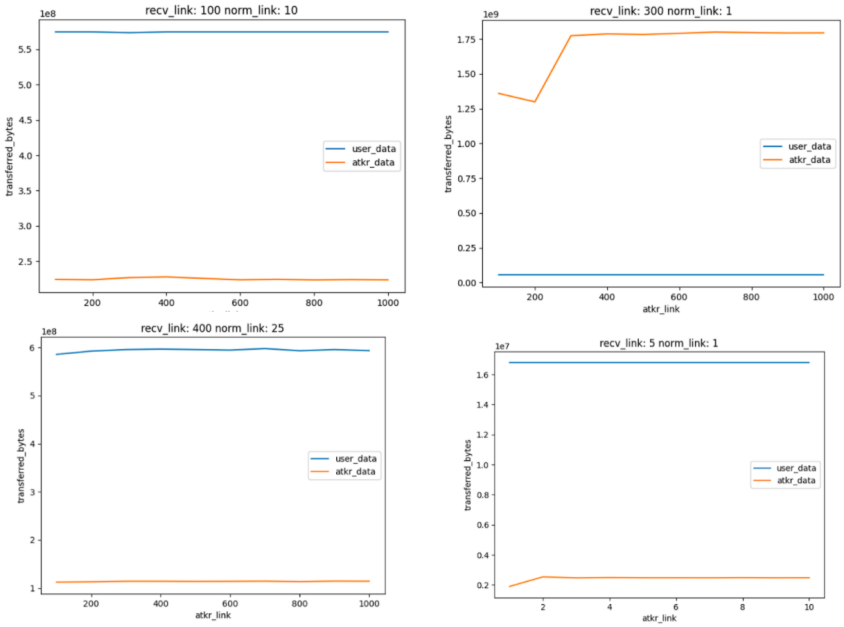
In this case, the receiver link is very large and it never congests. As a result, as the attacker sending rate increases, it is able to gain an advantage of being able to send more data to the receiver, while the total data transferred by normal hosts does not change much and stays at a stable level.

TCP CUBIC, uncongested:
The results observed using the TCP CUBIC protocol are similar to those of the TCP Reno protocol.

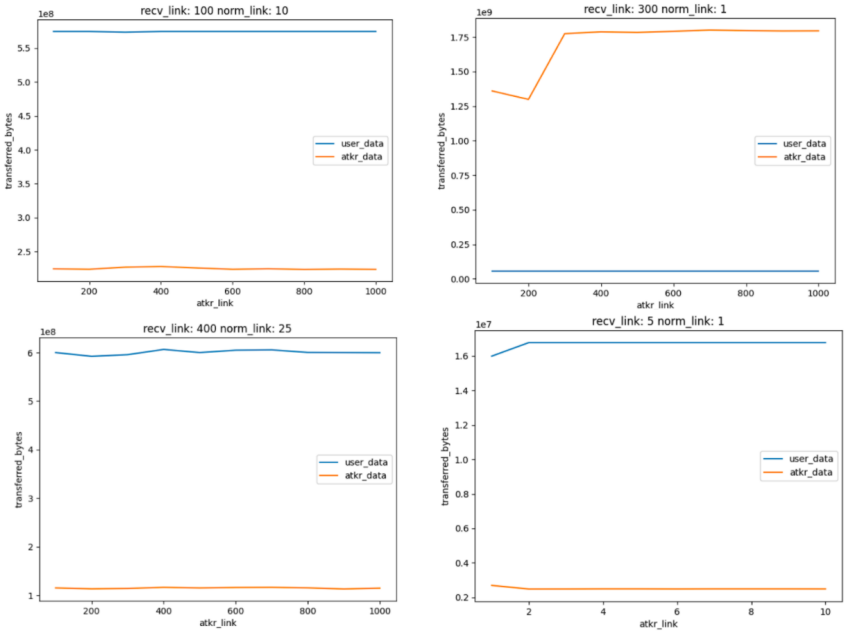


TCP Reno, congested:

When the receiver link is not large enough and congestion happens, it can be observed that the attacker is not able to gain an advantage by increasing the sending rate. Instead, the total data transferred stays at a plateau, which also happens to normal shots. Similar results are observed in the TCP CUBIC protocol as well.

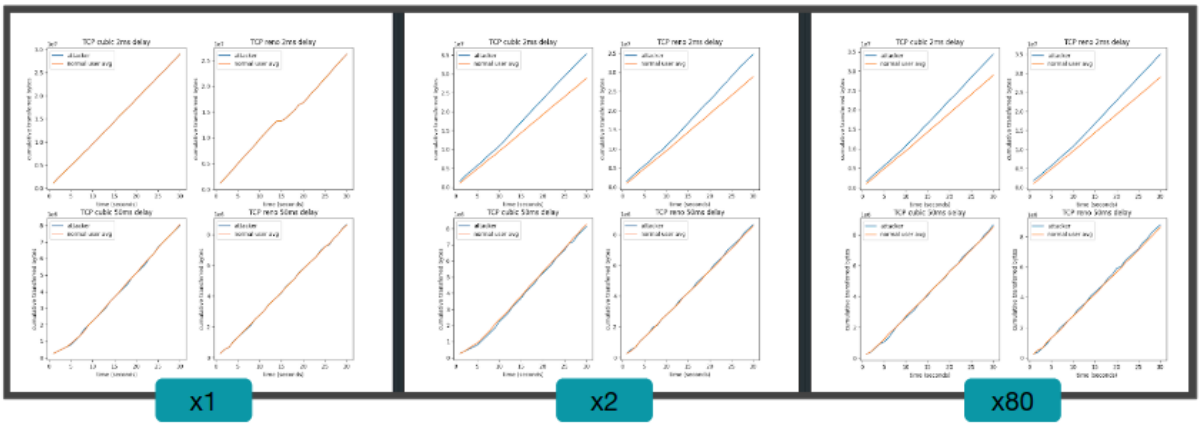


TCP CUBIC, congested:

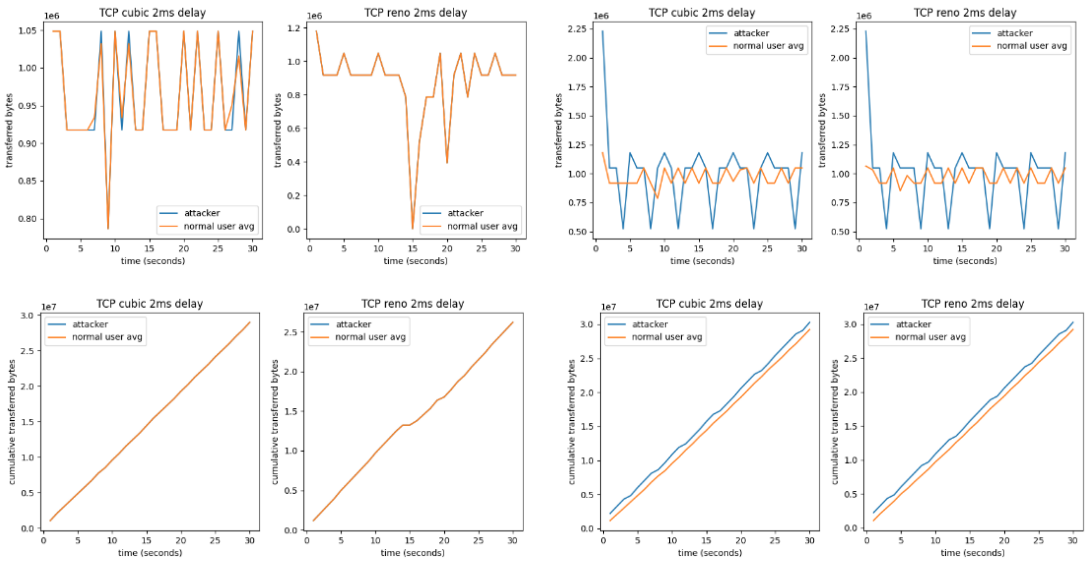


4.2.3.2 Tree Topologies

When we set the aggressive sender sending rate two times the normal user, we can see the attacker has a clear consistent advantage, but it is not two times throughput. When we set the attacker sending rate to 80 times bigger, the performance gain for the attacker did not grow at all. There is only a limited margin for exploitation and a single aggressive sender is bounded by the bandwidth of other links on the networks, combined with the TCP congestion control algorithm, the attacker did not manage to break the whole network down, and the impact on the normal users are very limited.

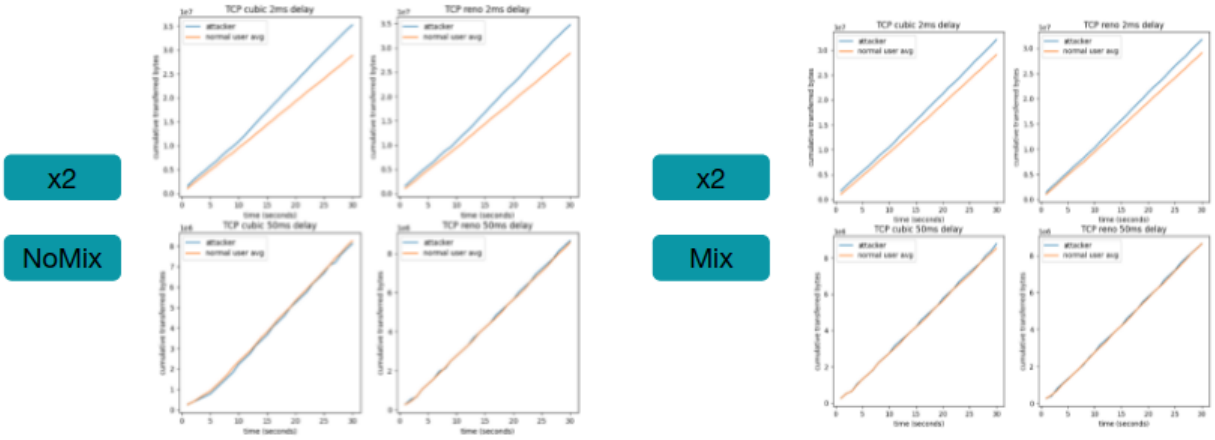


Another way for the attacker to send more packets than normal is to have an extremely large TCP window size at the beginning. There is an option on iperf that allows us to do that, we give the attacker a size 60 times bigger than normal users while keeping other parameters the same. Results are shown in the charts below, with the control group result on the left.



From the result, the attacker does get some initial advantage, but the throughput almost immediately drops after the initial stage when the attacker host experiences waves of packet loss events, and the average throughput quickly converges to be roughly the same as the normal users.

Lastly, we try to use a mixed protocol and see if it makes a difference. When we compare the result of an aggressive sender using two times bandwidth, on the left we have the attacker using the same algorithms with the reset of the users, on the right we have the attacker use the alternate algorithm. We can observe an interesting and consistent pattern: When everyone is using Reno but the attacker uses CUBIC, there is a decrease in the attacker's gain, and when in the reverse scenario, there is no noticeable difference. In both cases, the throughput of the normal users are not affected by this mixed use of protocols, and CUBIC is more effective in assuring fairness, as it actually should.



To replicate our experiments on aggressive senders or to read more experimental results, please refer to our [GitHub repository].

4.3 Strategic Abuse of TCP ACKs

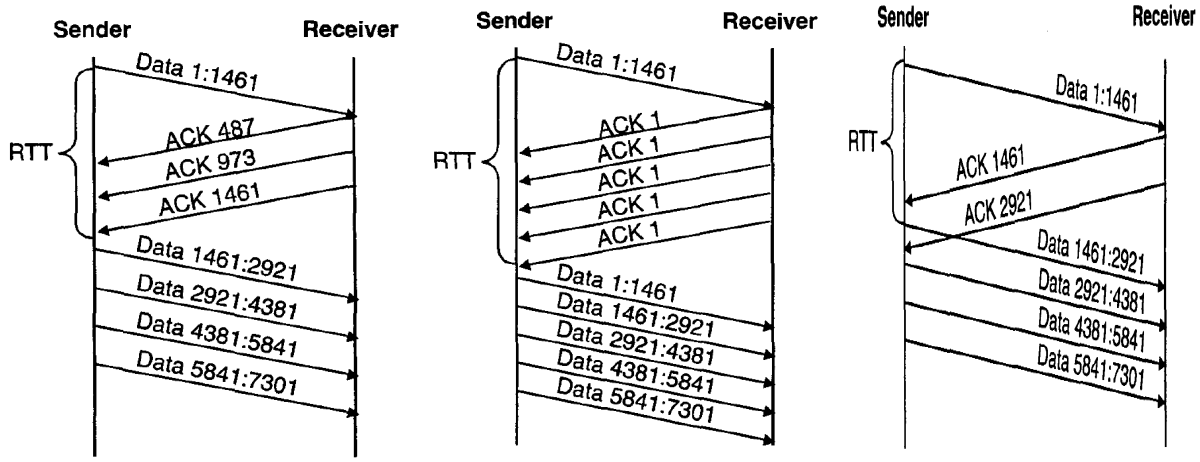
4.3.1 Approach & Methodology

The modern TCP mechanism implicitly relies on both the sender and receiver to cooperate in determining the proper data sending rate. TCP sockets located on the end users' machines are thus under the users' full control. Therefore, it is possible for a **misbehaving receiver** to get a faster download speed by tricking the TCP sender. In this section of our project, we explore three methods a misbehaving receiver can execute to exploit the **ACKs** sent from TCP receiver to sender.

4.3.1.1 ACK Division

This type of attack mainly takes advantage of the incongruence between TCP reliable data transfer protocol and congestion control protocol. In RDP, each TCP segment is assigned a sequence number and an acknowledgement field that is in **byte** offset to help ensure the data correctness and order. In congestion control, however, the sender will increase its *cwnd* for **each ACK packet** that acknowledges new data. This

mismatch between byte granularity and packet granularity leads to a potential vulnerability for attackers to exploit.



ACK Division *DupACK Spoofing* *Optimistic ACKing*

As shown in the first diagram, upon receiving a data segment containing N bytes, the receiver divides the resulting acknowledgement into M, where $M \leqslant N$, separate acknowledgements, each covering one of M distinct pieces of the received data segment. The sender will consider all of those M ACKs as new ACKs because they have different ACK numbers. Therefore, the sender's congestion window will grow M times faster than a normal TCP connection, and the misbehaving receiver would achieve a higher download speed as a result.

4.3.1.2 Duplicate ACK Spoofing

Critical readers may find out that in the TCP congestion control state diagram, apart from the slow start state and congestion avoidance state in which the congestion window can grow either exponentially or linearly, the *cwnd* size can also grow in the fast recovery state.

Consider the case when the TCP receiver sends N duplicate ACKs and N > 3. The first 3 duplicate ACKs will trigger the TCP sender to get into the fast recovery state and its *cwnd* is halved. Interestingly, any subsequent duplicate ACKs will then enable the TCP sender to grow its *cwnd* by one MSS and to send a new packet. If the N is very large, the

receiver can even use this mechanism to get a bigger *cwnd* and higher download speed accordingly.

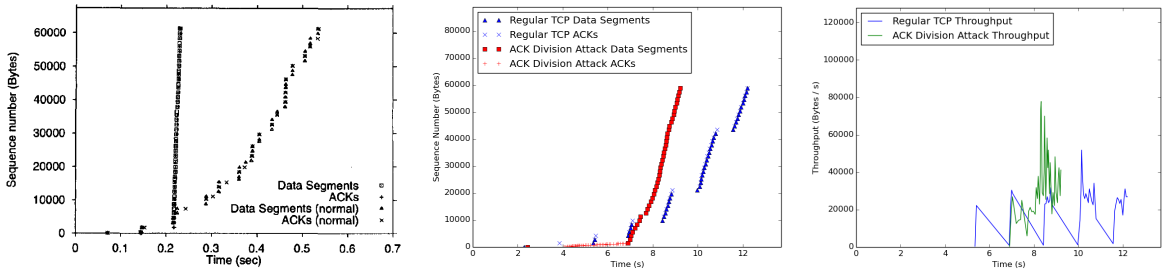
4.3.1.3 Optimistic ACKing

Upon receiving a data segment, the receiver sends a stream of acknowledgements anticipating data that will be sent by the sender. This would actually emulate a shorter RTT. However, since it can acknowledge data that is not actually received, this approach can not recover lost data and thus does not preserve end-to-end reliability. This method is the most dangerous and aggressive kind of attack. Since the congestion signal of loss events are “concealed” from the sender, the sender’s cwnd might increase indefinitely.

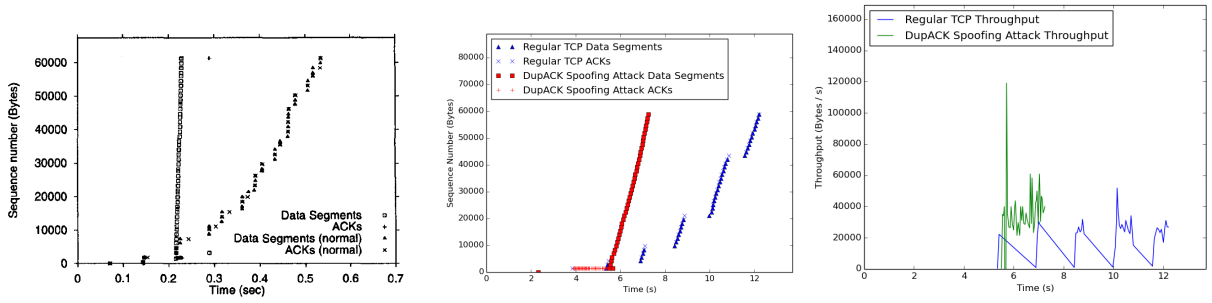
Careful readers may find out that the first two approaches of exploiting ACKs, i.e., ACK Division and Duplicate ACK Spoofing, preserve end-to-end reliability because they do not change the intrinsic way that ACKs work. It is a very important observation because this implies that those two attacks can be readily used by users who desire to get a better download speed while still reliably getting the data they want. Those two attacks have very minimal drawbacks for that single misbehaving user, so the incentive to use them can be quite high.

4.2.2 Experiments Conducted

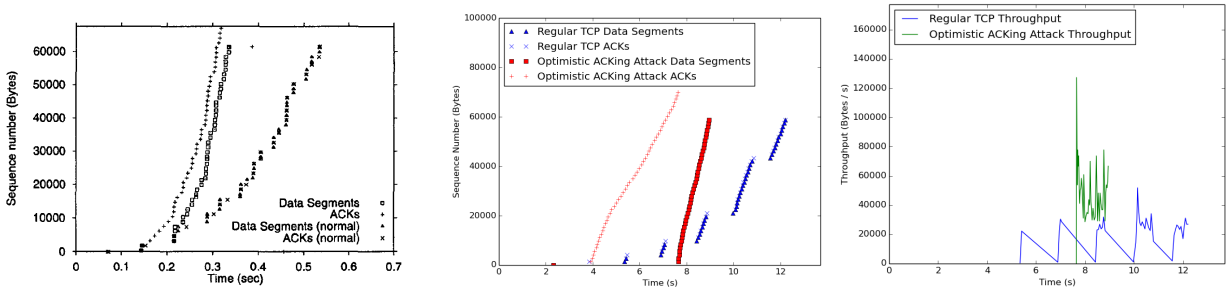
Firstly, we try to reproduce the plot shown in the original paper and successfully get similar results.



ACK Division



DupACK Spoofing



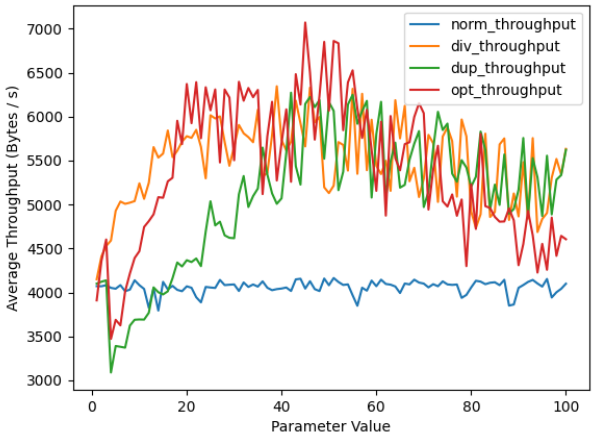
Optimistic ACKing

The plots are produced in an ideal network topology with one sender, one receiver, and one switch setup. The receiver can be either a regular TCP receiver or a misbehaving receiver from one of the three attacks.

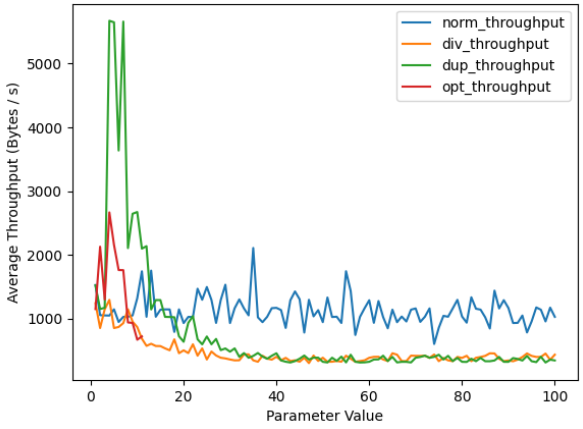
The left plots are the plots from the original paper. The plots in the middle are our reproduced work. The x-axis is time and the y-axis is the sequence number. To get a simpler comparison, we converted the metrics from sequence numbers to throughput, as shown in the right-hand-side plot. The AIMD mechanism can be clearly seen in the blue throughput plots, albeit with occasional fluctuations. These results clearly show that all three kinds of attackers would obtain a bigger throughput under this simple one-sender one-receiver setting.

Since our ultimate goal is to find the optimal degree of selfishness, we proceeded to define the selfishness in these three attacks (or the intensity of the attacks).

ACK Division	DupACK Spoofing	Optimistic ACKing
How many ACKs to divide into	How many duplicate ACKs to send	How many ACKs to optimistically acknowledge



Less congested network



More congested network

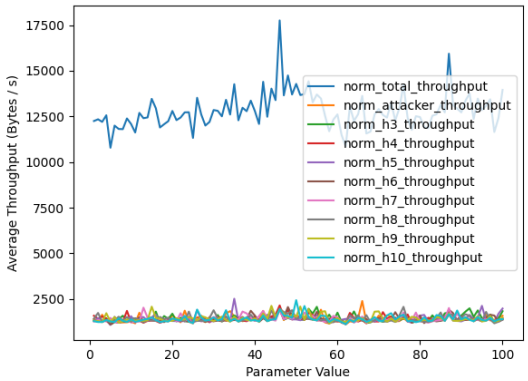
In this case, we are still using the ideal network topology with one sender and one receiver connected by a switch. The left-hand-side plot is the results of the experiment with bigger network bandwidth and queue size, and the right-hand-side is of one with limited bandwidth and queue size, mimicking a heavily congested network with low resources. The x-axis is the degree of selfishness as defined in the above table. The blue line representing the normal receiver only serves as a comparison, and as expected, it remains fairly horizontal. From the plots, we can deduce these findings:

1. Both plots show that there is a saturation point such that when the attacker is too selfish, they no longer benefit from the attack. The optimal parameter value for all 3 attacks for a less congested network seems to be around 50, while for a more congested network, it is around 3-5.
2. For the left plot, when the degree of selfishness is low, the attacker achieves a smaller throughput than the normal TCP receiver. We suspect that it is because the benefit of the attacks can not even counteract the overhead of executing said attacks when the attacks are less intense.
3. For the left plot, the Optimistic ACKing achieves the highest peak but also decreases faster across the x-axis after it crosses

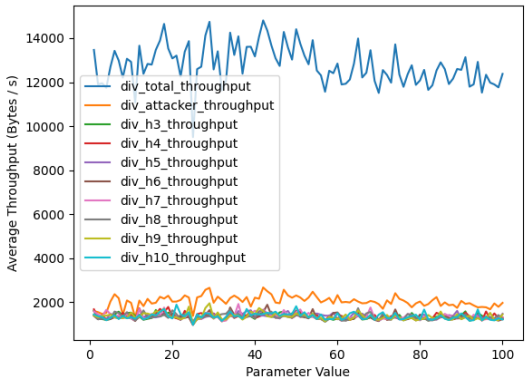
the peak. We guess that while Optimistic ACKing is the most aggressive type of attack (highest peak), it can not recover lost packets, and thus, when it attacks with too much intensity, many packets are lost (and hence decreases faster with a steeper gradient).

4. When we compare the left and right plots, we find out that the attacker benefits more in a less congested network. When the network is very congested and the degree of selfishness is high, the attacks may even be counterproductive, showing the necessity of having congestion control in modern networks.
5. The red line abruptly stops at around 10-11 for the plot on the more congested network. This was because it took too long for each experiment run to finish for the Optimistic ACKing attacker. Extrapolating, we can simply extend the red line to have a very low throughput value as the parameter value increases. We concluded that this was due to the fact that there were too many losses experienced by the attacker in a very heavily congested network such that it would not actually receive any more useful data at some point, which would lead to it halting.

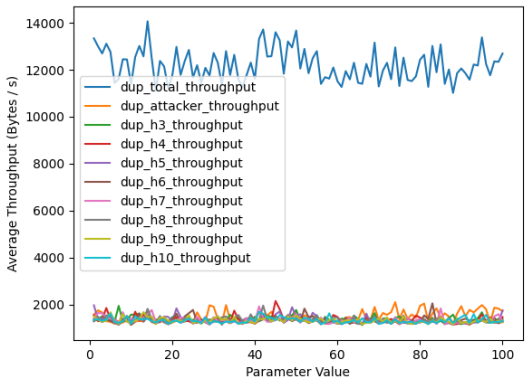
We also ran our experiment in a more realistic Star Topology to align our settings and be consistent with the previous 2 sections.



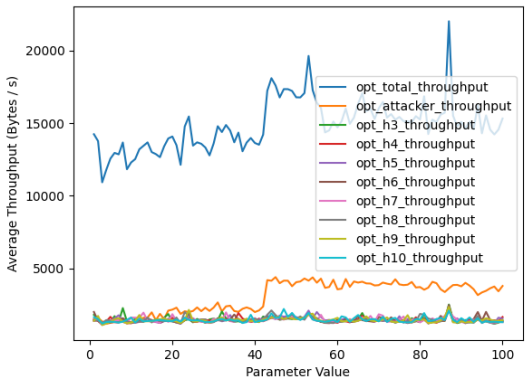
h2 = Normal Receiver



h2 = Div ACK Attacker



h2 = Dup ACK Attacker



h2 = Opt ACK Attacker

Our star topology has 10 hosts in total, where h1 is always the server and h3 to h10 are always normal TCP receivers. We then vary h2 in [normal receiver, Division ACK receiver, DupACK Spoofing receiver, Optimistic ACKing receiver] to obtain the above four plots. For the case whereby all receivers are normal, it is expected that their throughput graphs are about the same. However, for the other graphs, we can observe that ACK Division and Optimistic ACKing allow the attacker to gain a significant advantage over the normal receivers, while DupACK Spoofing does not seem to allow the attacker to gain a lot of advantage in this specific topology. For ACK Division, the advantage is smaller but more consistent, whereas for Optimistic ACKing, the advantage is higher, but it can only be experienced with larger parameter values.

5. Analysis & Discussion

5.1 Opening Multiple TCP Parallel Connections

From the results discussed in section 4.1, we achieve the following main observations:

1. TCP Fairness is very effective in ensuring that everyone receives a download throughput proportional to their number of connections to the server.
2. Differences in TCP Congestion Control Algorithm does not pose much of an impact on the overall throughput as the number of parallel connections increases.
3. The attacker's advantage is bottlenecked by his own link when his link bandwidth is much lower as compared to the link bandwidth of the server.
4. When the attacker's link is a relatively large fraction of the server link, we observe a strange phenomenon of spikes that is yet to be investigated (left for future work in lieu of project time constraints).
5. Bufferbloat problem occurs at higher CWND sizes and causes a threshold to appear where the increase of parallel connections results in diminishing returns.

Given the above observations, we can infer that there exists a threshold of selfishness that is the most optimal, but this threshold only occurs under specific network conditions and environments which might not be realised in real network situations. Nevertheless, the most promising threshold that we have discovered appears when the CWND size of the network routers are large (at least 256KB), and this causes the bufferbloat phenomenon to kick in. A rough estimate for such an optimum threshold under our specific network parameters would be around 60 to 70 parallel connections for small CWND sizes, and 20 to 30 parallel connections for large CWND sizes.

Furthermore, we see that network congestion and instability occurs particularly at large CWND sizes. This observation ties in nicely with our knowledge on the bufferbloat problem, and how unnecessarily large

buffers would result in a decrease in overall network throughput and increase latency and jitter.

To mitigate such undesirable situations of network bufferbloat, we propose that network device manufacturers produce devices with properly sized buffers (not too big, not too small) in place. If that is not possible due to less flexibility, network administrators can also choose to configure routers to use smaller buffers.

5.2 Aggressively Sending Packets into Network

Without deeper manipulation of the switch-host connection and modification of TCP protocol mechanism (e.g. how does the congestion window change, how is ACK responded, etc), TCP is able to maintain fairness facing simply larger traffic at a high rate.

- When it is not congested, the attacker will have advantages: more data gets sent to the server.
- When it is congested, TCP will ensure fairness and the data sent by the attacker (although at a higher sending rate) is fixed.

This is because this mechanism is less of an attack and more of a resource abuse - the attacker tries to take up as much bandwidth as possible - which could happen when it is uncongested and there are extra resources. But if there are limited resources, the TCP protocol is able to maintain fairness by giving resources to each incoming connection.

5.3 Strategic Abuse of TCP ACKs

Given the fact that ACK Division and DupACK Spoofing can be performed while still preserving end-to-end reliability, a normal Internet user has a huge incentive to perform such attacks to get a higher download speed. Moreover, a malicious hacker can perform Optimistic ACKing to force the server’s *cwnd* to grow indefinitely and finally result in

Denial of Service. Thus, we investigated if there are any countermeasures against these tactical abuses of ACKs.

It turns out that all three kinds of methods to abuse ACKs have their corresponding countermeasures, as gracefully elaborated in the original paper. For ACK Division, the sender just needs to increase its cwnd proportional to the ACK sequence number. Since an attacker utilizing ACK Division can only split the ACKs while keeping the absolute amount of data that it acknowledges as the same, this attack can no longer result in a faster cwnd growth speed at the sender side. Meanwhile, to fight against DupACK Spoofing and Optimistic ACKing, we can add two fields in the TCP header - Nonce and Nonce Reply. With these two fields, the TCP sender can differentiate valid duplicate ACKs that are triggered by a lost data packet from those that are maliciously generated by a DupACK attacker. For Optimistic ACKing, the sender can keep track of the cumulative sum of nonces to check if those ACKs are sent after the corresponding data packet has arrived.

We have managed to implement these possible defense mechanisms in code, which can be seen in our Github repository.

Since those counter-measures seem to be quite intuitive and relatively simple to implement, we then investigated if those measures are implemented in today's TCP protocol in the OS kernel. What we found is surprising, as almost all operating systems are still vulnerable to all of the 3 attacks! We thus draw the following speculations as to why no or very few countermeasures have been implemented so far:

1. There is no designated day for all Internet users and servers to simultaneously upgrade to the newer implementation at once, since businesses, industries, governments, and the entire world in general still need to be continuously operational. Backward compatibility also needs to be considered, as an attacker can still attack by declaring that he is still running on the old protocol. Without backward compatibility, many businesses would be set back by some time to perform system version upgrades, which might not be very desirable.

2. While point 1 explains the difficulty to upgrade the TCP protocol to prevent attacks, this attack prevention itself may be less important than we previously thought. Since these three types of attacks only attempt to get a higher throughput rather than attempting to break the network security, the damage that they can cause will be somewhat limited. As long as the majority of the users in the network still perform congestion control, then as demonstrated in our result plots, it will be hard for the very few attackers to cause the whole network to collapse. At the end of the day, all that the attackers would obtain is a relatively moderate increase in data transfer speed.

One possible limitation of our experiments in this section would be that we simply conducted our experiments in only TCP Reno. This is because based on our previous 2 sections, there is no noticeable difference between different TCP congestion control algorithms. This is actually a relatively assuring result since this implies that any congestion control algorithm performs relatively well in terms of preventing congestion in real life. Another limitation would be that our results are highly dependent on hardware and local network configuration, as indicated by some of our initial separate tests. In a different environment, the results that we would obtain might actually widely vary. This might slightly undermine the potential generalizability of our results. More experiments can be done in the future to ensure that these results are without much loss of generality.

5.4 Codebase

All the implementation code files of our experiments, diagrams and result datasets can be found in our GitHub repository: https://github.com/ragulbalaji/50.012Networks.

6. Conclusion & Future Work

In conclusion, we show that there exists a threshold indicating that it does pay to be selfish, but only under very specific network environments, depending on attack method and network configuration to varying degrees as specified in our analysis of our experiment results above.

For future work of this study, there are a few other areas we could enhance and explore:

1. We could conduct the experiments with finer granularity in terms of parameters and pattern observation, and seek to describe certain statistical insights with mathematical expressions and better precision.
2. We could conduct the research on more complex network topologies with more parameter variations and a variable number of attackers and normal users, as well as on different TCP variants (and possibly other network protocols as well) to further validate that our observations are indeed general.
3. We could seek alternative experiment methods and tools other than mininet and iPerf, or improve on the current experimentation scripts.

7. References

Ahamed, A. (2020). Traffic Analysis of a Congested and Uncongested Network in a Sequential Server Model. *Computer and Information Science*, *13*(1), 15. 10.5539/cis.v13n3p1

Cerf, V., Jacobson, V., Weaver, N., & Gettys, J. (2012, February). BufferBloat: What's Wrong with the Internet? *ACM Queue*, *55*(2), 40-47. 10.1145/2076450.2076464

Fleischer, L., Jain, K., & Mahdian, M. (2004). Tolls for heterogeneous selfish users in multicommodity networks and generalized congestion games. *Annual IEEE Symposium on Foundations of Computer Science*, *45*(1), 9. 10.1109/FOCS.2004.69

Jama, H., & Sultan, K. (2008). Performance Analysis of TCP Congestion Control Algorithms. *INTERNATIONAL JOURNAL OF COMPUTERS AND COMMUNICATIONS*, *2*(1), 30.

López, L., Almansa, G. d. R., Paquelet, S., & Fernandéz, A. (2005, October 10). A mathematical model for the TCP Tragedy of the Commons. *Theoretical Computer Science*, *343*(1-2), 4-26. 10.1016/j.tcs.2005.05.005

Savage, S., Cardwell, N., Wetherall, D. J., & Anderson, T. E. (1999, October 5). TCP congestion control with a misbehaving receiver. *ACM SIGCOMM Computer Communication Review*, *29*(5), 71-78. 10.1145/505696.505704

Zhang, H., Towsley, D., & Gong, W. (2005). TCP connection game: a study on the selfish behavior of TCP users. *13TH IEEE International Conference on Network Protocols (ICNP'05)*, 10 pp.-310. 10.1109/ICNP.2005.40