Reimplementation of TCP Congestion Control attacks

Team Members:
Albert Lin(albertl2@andrew.cmu.edu)
Run Wang(runw@andrew.cmu.edu)
Tristan Lin(tlin3@andrew.cmu.edu)

May 7, 2023

Abstract

During TCP connections, both the sender and receiver collaborate to determine the optimal transfer rate. The sender adjusts its data flow to prevent network congestion, while the receiver manages its buffer to prevent overflow. The TCP protocol also ensures that multiple data streams share the available bandwidth fairly.

Although it is well-known that a sender can intentionally send data faster than normal to increase its bandwidth share, the TCP Congestion Control [1] paper by Savage et al. shows that a receiver can also cause the sender to significantly increase its transfer rate. This could lead to a web client downloading content from a server much faster than other clients, even though the server intends to allocate resources equally. Our research project is based on an implementation of the paper "TCP Congestion Control with a Misbehaving Receiver" to re-implement certain misbehaving receivers and corresponding results using simple TCP/IP stack(LWIP) and evaluate possibilities of designing and implementing respective solutions to defend against optimistic attack besides using cumulative nonce.

1 Introduction

TCP-based end-to-end congestion control mechanisms play a crucial role in efficiently sharing limited bandwidth resources over the internet. The successful implementation of these mechanisms relies on the cooperation of both the client and server to agree on an appropriate data transmission rate. However, if the sender endpoint behaves improperly and does not follow the congestion control algorithms, it may send data faster than other well-behaved hosts. This can result in competing traffic being delayed or even dropped. Moreover, a misbehaving receiver can also cause similar issues, which is less commonly known.

The vulnerability of TCP congestion control has been mentioned in research "Known TCP Implementation Problems" [2]. However, the severity and ease of exploiting this vulnerability have not been fully recognized, as the large population of internet users presents a vast number of receivers who have both the motive and opportunity to exploit it, especially with the availability of open-source operating systems.

The author of "TCP Congestion Control with a Misbehaving Receive" investigates the impact of a misbehaving receiver on TCP congestion control and reveals three vulnerabilities that can be exploited to defeat it. The experiments conducted uses a modified TCP implementation, "TCP Daytona," confirming that many common TCP implementations possess these vulnerabilities. The interesting point motivating us to reproduce these vulnerability and defense mechanisms is that a group of researchers contacted Stefan Savage regarding re-implementation details about the paper but unfortunately, the code previously used was unavailable, which led to great difficulty in reproducing all three TCP congestion control related attacks. Furthermore, considering that the paper was published in 1999 and TCP's congestion control algorithm has undergone significant changes since then, it would be intriguing to investigate whether these attacks remain viable.

To avoid recompiling the kernel, we utilized a user-space TCP stack called lwIP, which is frequently used in embedded systems. However, it is also adaptable to Unix. In order to save time on validating the existence of vulnerability, we used Scapy to simulate the attack on the receiver end and implemented defense mechanisms by modifying the tcp_in.c file in lwIP stack. In our experiment, we created a Mininet topology comprising two hosts: a receiver and a sender, connected via a single link. The receiver utilizes lwIP as the TCP stack via a virtual TAP interface, while the tcpsink program listens for connections and records all received data. The sender employs a Python script to generate TCP traffic. For data analysis, we captured all outgoing data packets on the sender using tcpdump, then plotted the data by sequence numbers.

So far we have tried to replicate three different attacks. Based on our experimental design and evaluations, both the duplication attack and the division attack no longer seem to work on the linux kernel and lwIP. The only one that works as expected is the optimistic attack. Additionally, our team found evidence that the ACK Division and Duplicate Ack attack was resolved on lwIP and will include more details in next section.

2 Background

According to background researches we have done, lots of researchers have tried to replicate these three attacks in the past, but they have encountered more or less insurmountable problems in the process of doing so.

Andrew Haven and James Whitbeck [3] tried to reproduce three attacks. They discovered that ACK division still causes more data to be sent compared to a non-misbehaving receiver, but they were not able to reproduce the speed-up with ACK duplication. Finally, they discuss the challenges of implementing and testing the attacks using user-space TCP stacks and tools like Mininet and tcpdump.

Possible reasons that the Duplicate Ack attack and Ack division cannot be replicated were mentioned by Robert Gasparyan and Jongho Shin [4] and in RFC 5681 [5].

The recommendation made in the original paper that senders should keep a cumulative nonce to verify ACKs and prevent optimistic ACKing was standardized in RFC 3540 [6]. However, this new standardized feature is not introduced in the lwIP stack, which is one crucial factor that enabled us to could successfully conduct Optimistic attack as shown in the experiment section.

3 Design and Implementation

In order to simulate customized attacks and corresponding defenses, it is intuitive and mostly straightforward to come up with approaches to somehow modify TCP/IP implementations in terms of the network environment in which our experiment were to be performed.

The initial idea was to directly modify the implementation of the TCP/IP protocol stack and to re-compile the implementation on the kernel of the machine in our experiment. Undoubtedly, this would be the most acknowledged, convincing and also most close to reality. However, it was soon discovered that this way of re-implementation and direct compilation on the kernel would be extremely time consuming in numerous perspectives(e.g. debugging, kernel compiling etc). That being said, we attempted to explore other more effortless approaches to realize the task of modifying TCP/IP stacks.

The remainder of this section will cover the solution we eventually arrive, a thorough high level design of our experiment setup, along with a detailed implementation of the core customized attacks we intend to simulate.

3.1 Experiment setup

We begin by introducing the key tools applied to our experiment setup and based on the setup, continue to demonstrate the interactions between components of our experiment environment and the roles played by each of them.

3.1.1 lwIP

lwIP (lightweight IP) is a small, open-source TCP/IP protocol stack designed specifically for embedded systems. It is designed to be small in size, low in memory requirements, and efficient in performance. lwIP was originally developed by Adam Dunkels [7] at the Swedish Institute of Computer Science (SICS) and is now maintained by a group of developers around the world.

lwIP provides a set of network protocols, including IP, TCP, UDP, ICMP, DHCP, and DNS. It also includes support for Ethernet, PPP, and SLIP network interfaces. lwIP can be used in a variety of embedded systems, including microcontrollers, FPGAs, and DSPs. It is typically used in applications that require a small memory footprint, such as home automation, industrial automation, and IoT devices.

One of the key features of lwIP is its ability to run in a single thread, which makes it well-suited for resource-constrained systems. Another advantage that suits with our requirements perfectly is that lwIP is highly configurable, allowing developers to tailor and customize the protocol stack to meet the requirements of their specific application.

As a result, we have chosen lwIP as the layer to host a modified TCP/IP protocol stack based on the attacks applied to, given its flexibility, and the fact it allows us to avoid dealing with too trivial and complex issues regarding operating systems and kernel compilations.

3.1.2 mininet

It is also straightforward for us to come up with ways to create a network environment on our given machine in order to generate TCP traffic and observe relevant network information. We have chosen Mininet [8] to fulfill the task.

Mininet is a network emulator that creates a virtual network environment for testing and developing network software. It allows network researchers and developers to create complex network topologies, experiment with different network configurations, and evaluate the performance of network protocols and applications in a controlled environment.

Additionally, it creates a virtual network using lightweight virtualization technologies, such as Linux containers or virtual machines, and provides a command-line interface for interacting with the virtual network. Users can create custom network topologies by defining the number and type of hosts, switches, and links in the network, and then run different network protocols and applications to evaluate their performance.

Mininet is widely used in academia and industry for network research and education. It is an open-source project that is actively maintained by a community of developers around the world. Mininet is written in Python and supports a wide range of network technologies, including Ethernet, WiFi, and software-defined networking (SDN) protocols such as OpenFlow. It also provides plethora application programming interfaces in Python to support developers.

3.1.3 Scapy

Upon the premise in which the TCP/IP stack has been successfully modified upon our will and the network topology set up, we have to find a way to generate network traffic from one host to another. The tools we have chosen is a Python third party package Scapy [9].

Scapy is an open-source Python-based tool for network packet generation, manipulation, and analysis. It is a powerful and flexible packet manipulation tool that can be used for a wide range of network-related tasks, such as network testing, security auditing, and network protocol development.

Scapy allows users to create and manipulate packets at the network layer, transport layer, and application layer. It supports a wide range of network protocols, including Ethernet, IP, TCP, UDP, ICMP, DNS, DHCP, and many others. Users can craft custom packets with specific fields and values, inject packets into a network, and analyze network traffic to identify potential vulnerabilities or performance issues.

One of the key features of Scapy is its ability to be easily extended with custom modules and scripts. Users can write their own modules to support new protocols or add additional functionality to the tool. Scapy also includes a powerful interactive shell that allows users to experiment with packet crafting and analysis in real-time.

Scapy is widely used in network security, penetration testing, and network protocol development. It is an open-source project that is actively maintained by a community of developers around the world.

3.1.4 Network Topology

For the network topology, we follow the client/server model. Since we are simulating 3 different types of attacks, we are using the client side to perform certain attack behaviors, where the server performs corresponding defenses.

On the client side, we are using Scapy to generate simulated attacker's TCP traffic to the server side. On the server side, lwIP is ran on top of the server's network layer to allow the defense mechanisms applied to respective attacks onto the TCP stack are executed. All machines(client, server) are hosted under mininet.

```
# Network topology with one switch
# connecting a client and a server

client=self.addHost('client')
server=self.addHost('server')
switch=self.addSwitch('s0')

self.addLink(client,switch,bandwith,delay,
    max_queue_size)
self.addLink(server,switch,bandwith,delay,
    max_queue_size)
```

3.2 Implementation

3.2.1 TCP protocol review

TCP (Transmission Control Protocol) is a widely used transport layer protocol in computer networks. It provides reliable, ordered, and error-checked delivery of data between applications running on hosts connected to a network. TCP uses a combination of sequence numbers and acknowledgment numbers to provide reliable delivery of data.

TCP also includes a congestion control algorithm, which is used to avoid network congestion and ensure that the network is utilized efficiently. The TCP congestion control algorithm works by monitoring the network conditions and adjusting the rate at which data is sent based on the current state of the network.

The TCP congestion control algorithm uses a feedback mechanism to determine the optimal rate at which data can be sent without causing congestion. When the network is operating normally, TCP increases the sending rate, but when congestion is detected, TCP reduces the sending rate to prevent further congestion. The algorithm uses a variety of techniques to detect congestion, such as monitoring the number of lost packets and the round-trip time of packets.

The most commonly used congestion control algorithm in TCP is the "TCP Reno" algorithm. TCP Reno uses a "slow start" mechanism to gradually increase the sending rate, and then switches to a "congestion avoidance" phase once a certain threshold is reached. During congestion avoidance, TCP Reno reduces the sending rate in response to congestion, and then gradually increases the sending rate again as congestion subsides.

We now describe three types of attacks along with fragments of key pseudocode implementation on this congestion control process that exploits a sender's vulnerability to misbehaving receiver behaviors.

3.2.2 ACK Division

TCP utilizes an error control protocol that operates at the byte level, which means each TCP segment is identified by its sequence number and acknowledgment fields that are associated with byte offsets within a TCP data stream. Despite this byte granularity, TCP's congestion control algorithm is expressed in terms of segments rather than bytes.

During slow start, TCP increases the congestion window by one for every new ACK received. When a certain threshold is met, during congestion avoidance, the congestion window is increased by one full-sized segment per round-trip time (RTT). This disparity of granularity between two phases of the TCP congestion control procedure leads to a vulnerability.

The vulnerability can be exploited by dividing the acknowledgment of a data segment containing N bytes into M separate acknowledgments where M¡N, each of which ranges cumulatively among one of the M distinct pieces of the received data segment.

```
# DIVIDE_FACTOR:number of seperate ACKs
# INIT_SEQ_NO:initial sequence number

final_ack=data.seq+len(data.payload)+1
start_ack=data.seq

ack_interval=(final_ack-start_ack)/
    DIVIDE_FACTOR

ack_nos=range(start_ack+ack_interval,final_ack,
    ack_interval)

ack_nos.append(final_ack)

for ack_no in ack_nos:
  send(TCP(window,ack_no,seq=(INIT_SEQ_NO+1)))
```

3.2.3 Duplicate ACK

TCP has two mechanisms - fast retransmitt and fast recovery - that help to minimize the impact of packet loss. The fast retransmitt algorithm detects packet loss by analyzing three duplicate acknowledgments, and then quickly retransmitts what seems to be the missing packet. However, when a duplicate acknowledgment is received, it suggests that segments are leaving the network.

The fast recovery algorithm then takes advantage of this information by increasing the congestion window by the number of segments (three) that have left the network and that the receiver has buffered. For each subsequent duplicate ACK received, the congestion window is increased by a certain threshold specified to further inflate it and reflect the additional segment that has left the network.

```
# DUPLICATION_FACTOR: threshold to be detected
    as duplication
# INIT_SEQ_NO: initial sequence number

final_ack=data.seq+len(data.payload)+1

for i in range(DUPLICATION_FACTOR):
  send(TCP(ack=final_ack,seq=(INIT_SEQ_NO+1)))
```

3.2.4 Optimistic ACKing

TCP's algorithms rely on the premise that the time interval between a sent data segment and an acknowledgment for that segment is no less than one round-trip time. Additionally, TCP's congestion window expansion rate is proportional to the round-trip time, where it is an exponential growth rate during slow start and a linear growth rate during congestion avoidance.

Hence, shorter round-trip times indicates faster data transmission. However, the protocol does not employ any methods to enforce this assumption. This makes it possible for a receiver to simulate a shorter round-trip time by sending optimistic acknowledgments for data it has not yet received.

Practical implementations would require a prediction of sequence number of future TCP packets. An attacker can potentially leverage this vulnerability through an attack that involves sending a data segment, following anticipating ACKs that the data has yet to be sent by the sender.

```
# INIT_SEQ_NO: initial sequence number

"""After the first packet was received..."""

OPT_ACK_START=data.seq
PACKET_SIZE=len(data.payload)

for i in range(1,int(ACK_SPACING/PACKET_SIZE)):

    opt_ack=TCP(ack=(OPT_ACK_START+i*
    PACKET_SIZE),seq=(INIT_SEQ_NO+1))

    send(opt_ack)
```

4 Solution

According to the evaluation results discussed in the next section. The ACK Division and Duplicate ACK did not yield the predicted results. Therefore, this section will focus on the solution for the Optimistic ACKing approach.

Our proposed defense mechanism for the Optimistic ACKing attack is to add a segment boundary check algorithm in the lwip tcp_in.c file on the server. The algorithm basically verifies if the received ACK package sequence number falls in the unacknowledged segments' sequence number range. All the sent but unacknowledged segments are stored in the tcp_seg *unacked structure in the tcp protocol control block. The pseudocode for this defense approach is shown below.

```
# Come here when the ACK acknowledges new data
# Check if the ack falls in a segment boundary.
for each segment in pcb->unacked:
    if segment->tcphdr->seqno + TCP_TCPLEN(
    segment) == ACK_No:
  in_segment_boundary_flag = 1;
# Process if it falls in a segment boundary.
if in_segment_boundary_flag:
    # Process ACK
```

5 Evaluation

In order to visualize the feasibility of the previously mentioned attack vectors and defense mechanisms, we designed a evaluation procedure that intuitively shows the experimental results. The evaluation process is meant to reflect the effect of ACK Division, Duplicated ACKs and Optimistic ACKing attacks. It will also prove how the pro-

posed defense solution can help mitigate this congestion control vulnerability.

5.1 Methodology

Our methodology for showing the effects of the attacks and defense is through comparing the malicious/defended network flow against normally behaving receivers. The comparison is displayed through plotting the results with matplotlib. The four main evaluations that are significant for the results are listed below.

- ACK division attack on client and lwip on server against kernel on client and lwip on server.
- Duplicated ACKs on client and lwip on server against kernel on client and lwip on server.
- Optimistic ACKing on client and lwip on server against kernel on client and lwip on server.
- Optimistic ACKing on client and defended (segment boundary) lwip on server against kernel on client and lwip on server.

5.2 Implementation

To obtain the network traffic data needed for analysis, we used tcpdump on the client to capture network flow into a pcap file during the client-server TCP connection. The pcap file is then used to plot figures with the matplotlib library.

After capturing the network traffic on the client, we first organized the data into two classes: ACK packages sent by the client and data segments sent by the server. The time stamp and sequence numbers are then sorted according to the earliest recorded time and initial sequence number to make the graph concise.

For each graph generated, the x-axis represents the time stamp of the traffic and the y-axis represent the data bytes sent by that time. In each graph four data classes are plotted: Normal ACK flow, Normal data segment flow, Malicious ACK flow, and compromised data segment flow.

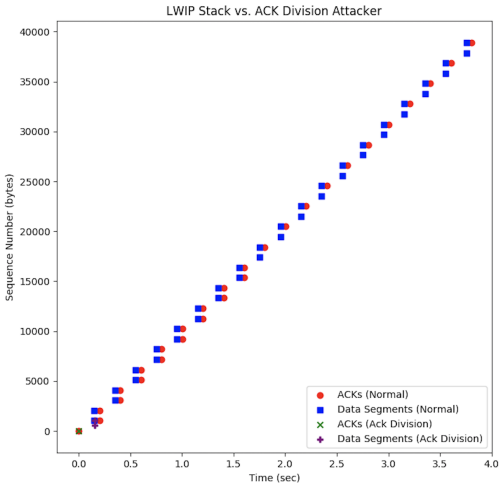

Figure 1: ACK division attack compared to normal behaving client

5.3 Key results

5.3.1 ACK Division

For the ACK Division attack the receiver is dividing the acknowledgment of a data segment an expecting to increase the congestion window to acheive higher throughput. However, as shown in Figure 1, the experiment results show that the server did not return the data segments as expected and the ACK division attack terminated due to the suspended response.

The results indicate that the ACK Division attack is outdated in the current lwip TCP/IP stack. The reason why this exploit is not working is mentioned in the RFC 5681 report [5]. During congestion avoidance, cwnd must not be increased by more than SMSS bytes per RTT. This method both allows TCPs to increase cwnd by one segment per RTT in the face of delayed ACKs and provides robustness against ACK Division attacks.

5.3.2 Duplicate ACK

For the Duplicate ACK attack the receiver is sending multiple identical ACK packages to increase the congestion window to acheive higher throughput. However, as shown in Figure 2, the results show that the misbehaving receiver is receiving data segments from the server at a lower speed

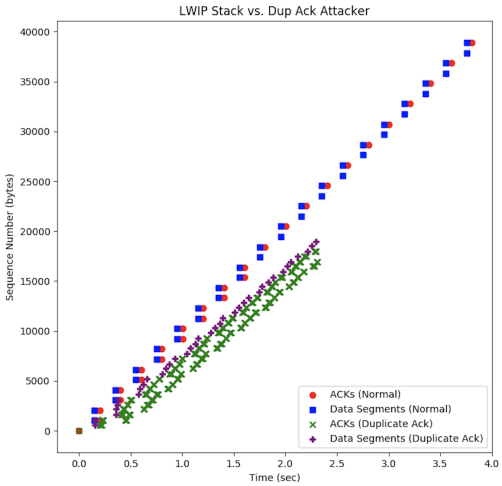

Figure 2: Duplicate ACK attack compared to normal behaving client

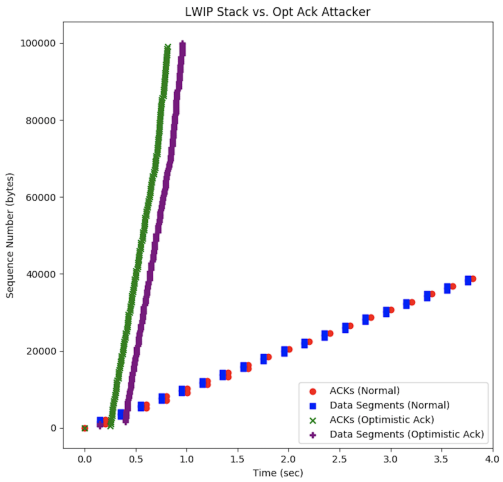

Figure 3: Optimistic ACKing attack compared to normal behaving client

than the normal performing receiver.

This is also an predictable result since the current fast retransmit and fast recovery algorithm may now limit the number of duplicate ACKs that artificially inflate cwnd during loss recovery to the number of segments outstanding to avoid the duplicate ACK spoofing attack [5].

In addition, the RFC 3465 [10] suggests "TCP Congestion Control with Appropriate Byte Counting (ABC)". This RFC suggests increasing cwnd based on the number of bytes being acknowledged by each arriving ACK, rather than by the number of ACKs that arrive. This can further protect the TCP protocol against Duplicate ACKing attacks.

5.3.3 Optimistic ACKing

For the Optimistic ACKing attack the receiver is predicting future sequence numbers and inflating the congestion window by acknowledging data segments that are not yet sent by the server. As shown in Figure 3, the results of this attack on the lwip TCP/IP stack is significant. The misbehaving receiver receives data segments at a much higher speed than the normal performing receiver.

5.3.4 Defense

In order to mitigate the Optimistic ACKing attack, we implemented segment boundary checks in the lwip TCP/IP stack. As shown in Figure 4, the defense approach is proven to be very effective since the misbehaving receiver no longer receives data segments faster than a normal receiver. In fact, the receiver does not receive any more data segment packages other than the ones sent at the beginning of the connection. This is because according to the defence implementation, the server simply discarded unexpected ACK packages and the receiver will not resend the Optimistic ACKing packages again. This leads to the situation where the receiver thinks that it already sent an ACK but the server did not process it, which will eventually result in the communication ending up in stalemate.

6 Conclusions and Limitations

We have successfully replicated the optimistic ACK attack, demonstrating that this vulnerability still exists in modern TCP. Additionally, we have implemented a defense and confirmed that, in theory, TCP can be protected against optimistic ACK attackers. Based on our experiments, we believe that the Dup ACK attack and ACK division attack are outdated on both Linux kernel and lwIP

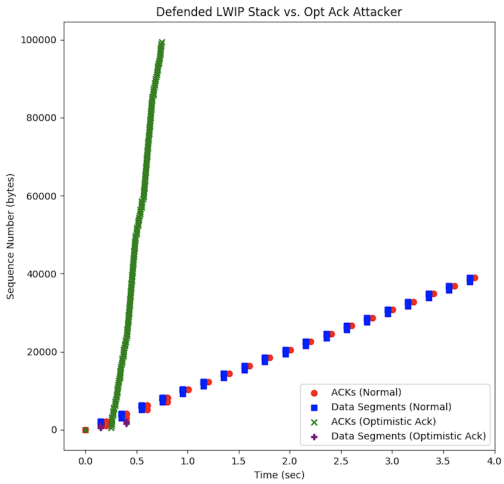

Figure 4: Optimistic ACKing defense compared to normal behaving client

stack. Andrew's [3] experimental results have troubled us for a long time because they succeeded in implementing an ACK division attack using a similar setup, but we could not find a reasonable explanation for our failure. Andrew's experiment cannot be reproduced from his source code because of some broken dependencies. For future research, we think we can spend more time understanding the underlying code logic of lwIP, and we hope that someone can provide more comprehensive documentation on lwIP to help us prove theoretically why Dup Ack attack and Ack division attack are no longer feasible.

7 References

[1] Savage, Stefan, et al. "TCP congestion control with a misbehaving receiver." Proceedings of the ACM SIGCOMM '99 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, ACM, 1999, pp. 137-146.

[2] Paxson, V., Allman, M., Dawson, S., Fenner, W., Griner, J., Heavens, I., Lahey, K., Semke, J., and B. Volz, "Known TCP Implementation Problems", RFC 2525, DOI 10.17487/RFC2525, March 1999.

[3] Andrew, James. (2012, August 8). TCP daytona: Congestion control with a misbehaving receiver. Reproducing Network Research. Retrieved April 12, 2023.

[4] R. (2014, June 3). CS244 '14: TCP Congestion Control with a Misbehaving Receiver. CS244 '14: TCP Congestion Control With a Misbehaving Receiver — Reproducing Network Research.

[5] Allman, M., Paxson, V., and E. Blanton. "TCP Congestion Control", RFC 5681, DOI 10.17487/RFC5681, September 2009.

[6] Spring, N., Wetherall, D., and D. Ely, "Robust Explicit Congestion Notification (ECN) Signaling with Nonces", RFC 3540, DOI 10.17487/RFC3540, June 2003.

[7] Dunkels, Adam. "Design and Implementation of the lwIP TCP/IP Stack." Swedish Institute of Computer Science 2.77 (2001).

[8] De Oliveira, Rogério Leão Santos, et al. "Using mininet for emulation and prototyping software-defined networks." 2014 IEEE Colombian conference on communications and computing (COLCOM). IEEE, 2014.

[9] Rohith, R., Minal Moharir, and G. Shobha. "SCAPY-A powerful interactive packet manipulation program." 2018 international conference on networking, embedded and wireless systems (ICNEWS). IEEE, 2018.

[10] Allman, M., "TCP Congestion Control with Appropriate Byte Counting (ABC)", RFC 3465, DOI 10.17487/RFC3465, February 2003

[11] Yan, Lisa, and Nick McKeown. "Learning networking by reproducing research results." ACM SIGCOMM Computer Communication Review 47.2 (2017): 19-26.