IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received March 24, 2018, accepted April 30, 2018, date of publication May 4, 2018, date of current version June 19, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2833107

General TCP State Inference Model From Passive Measurements Using Machine Learning Techniques

DESTA HAILESELASSIE HAGOS[1,2], PAAL E. ENGELSTAD[1,2], ANIS YAZIDI[2], AND ØIVIND KURE[1,3]

[1]Autonomous Systems and Sensor Technologies Research Group, Department of Technology Systems, University of Oslo, 0315 Oslo, Norway
[2]Autonomous Systems and Networks Research Group, Department of Computer Science, Oslo Metropolitan University, 0167 Oslo, Norway
[3]Department of Telematics, Norwegian University of Science and Technology, 7491 Trondheim, Norway

Corresponding author: Desta Haileselassie Hagos (destahh@ifi.uio.no)

ABSTRACT Many applications in the Internet use the reliable end-to-end Transmission Control Protocol (TCP) as a transport protocol due to practical considerations. There are many different TCP variants widely in use, and each variant uses a specific congestion control algorithm to avoid congestion, while also attempting to share the underlying network capacity equally among the competing users. This paper shows how an intermediate node (e.g., a network operator) can identify the transmission state of the TCP client associated with a TCP flow by passively monitoring the TCP traffic. Here, we present a robust, scalable and generic machine learning-based method which may be of interest for network operators that experimentally infers *Congestion Window (cwnd)* and the underlying variant of *loss-based* TCP algorithms within a flow from passive traffic measurements collected at an intermediate node. The method can also be extended to predict other TCP transmission states of the client. We believe that our study also has a potential benefit and opportunity for researchers and scientists in the networking community from both academia and industry who want to assess the characteristics of TCP transmission states related to network congestion. We validate the robustness and scalability approach of our prediction model through a large number of controlled experiments. It turns out, surprisingly enough, that the learned prediction model performs reasonably well by leveraging knowledge from the emulated network when it is applied on a real-life scenario setting. Thus, our prediction model is *general* bearing similarity to the concept of transfer learning in the machine learning community. The accuracy of our experimental results both in an emulated network, realistic and combined scenario settings and across multiple TCP congestion control variants demonstrate that our model is reasonably effective and has considerable potential.

INDEX TERMS Network protocols, TCP, congestion control, passive measurement, machine learning, transfer learning, convolutional filtering, deep learning.

I. INTRODUCTION

Machine learning techniques have effectively advanced the *state-of-the-art* for many research domain problems in the computer networking community by creating real-world impacts. For example, they are being applied in the areas of *traffic classification* [26], [41], [42], *security monitoring* and *Intrusion Detection Systems (IDS)* [17], [35], *network scheduling* [24], and many other topics in computer networks. In this paper, we argue that employing machine learning-based techniques can also provide a potentially promising methodology for improving the accuracy of predicting TCP per-connection states from passive measurements. Much of the Internet's traffic is carried using the end-to-end TCP protocol [19] due to practical considerations that favored TCP over other transport protocols. To deal with network congestion, TCP uses congestion control algorithms to guide and regulate the network traffic on the Internet. This helps it to avoid sending more data that the underlying network is capable of transmitting which is maintained by the sender's *cwnd*. The global Internet highly relies on TCP congestion control algorithms and adaptive applications that adjust their data rate to achieve high performance while avoiding congestion on the network [4]. Congestion control is a fundamental problem in computer networks. One of the main

2169-3536 © 2018 IEEE. Translations and content mining are permitted for academic research only.
Personal use is also permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

parameters for TCP performance evaluation in a real-world setting is *cwnd*. The TCP congestion control algorithms that are widely deployed today perform the most important functionalities related to network congestion such as handling the *cwnd* from the sender-side. Therefore, it is very natural to ask:

- How well can we infer the most important TCP per-connection transmission states that determine a network condition (e.g., *cwnd*) from a passive traffic collected at an intermediate node of the network without having access to the sender?
- How can we track the underlying TCP variant that the TCP client is using from passive measurements?
- What percentage of network users are using either a *loss-based* or *delay-based* TCP variants?
- Which user is responsible for the majority of heavy flow traffic in the network?
- How do different implementations of TCP congestion control algorithms behave on the end-to-end variability of bandwidth, delay, different cross-traffic, Round-trip Time (RTT)?, etc.

Our work is mainly motivated by these important questions and therefore, in this paper, we investigate and explore these questions quantitatively as they apply to problems of network congestion.

The TCP congestion control itself has grown increasingly complex which in practice makes inferring TCP per-connection states from passive measurements a challenging task. Much of the existing research work on this problem rely on an *active* approach to measure the characteristics of TCP. The difference between *active* and *passive* measurement techniques will be explained later in detail in Section IV. A wide variety of approaches have been applied to the problem of congestion control characteristics. The work reported in [20] presented an approach to estimate TCP parameters at the sender-side based on packets captured at the monitoring point using a *Finite State Machine (FSM)*. The authors have pointed out that the estimation of *cwnd* may have potential errors primarily due to over-estimation of the RTT and estimation of incorrect window sizes [20]. Another limitation of this work, given the many existing variants of TCP, the use of a separate *state machine* for each variant is *unscalable* and that the constructed *replica* might not manage to reverse or backtrack the transitions taking the tremendous amount of data into consideration. In addition to this, the *replica* may also not observe the same sequence of packets as the sender and ACKs observed at the intermediate node may not also reach the sender. TCP implementations developed by different operating system vendors that have different parameters (e.g., minimum RTO, timer granularity, duplicate ACK thresholds, etc.) can also behave so differently [33]. For example, given the same ACK response from the receiver, there is a variation between a client using Linux TCP stack and Windows TCP stack [33]. Rewaskar *et al.* [33] addressed this problem by developing a separate state machine for each of the operating system vendors. The problem with this technique [33] is that it increases the amount of processing required per TCP connection when there is a change in operating system (e.g., when new operating systems are developed or old variants are changed) which again leads to the development of new state machines.

In moving towards a *generic* prediction approach, after we survey the existing works for monitoring of TCP transmission states from passive measurements, we believe there is very little work on a *robust*, *scalable* and *generic* method of predicting the *cwnd* and uniquely identifying the type of the underlying TCP congestion control algorithm from a passive traffic without the knowledge of the sender's *cwnd* for most of the widely used TCP variants in the Internet using machine learning. In this paper, we argue that the existing approaches for monitoring of TCP per-connection states from passive measurements do not adequately address the problem either due to being outdated or failing to recognize the difference between individual implementations of TCP variants [33]. Hence, compared to these previous studies, in this paper, we explore machine learning approaches based on the time series of *outstanding bytes* in flights to predict the per-connection state of a TCP *cwnd* of the sender by examining each cross-traffic of TCP flows of the endpoints passively collected at an intermediate node. We demonstrate how an intermediate node (e.g., a network operator) can identify the transmission states of the TCP client associated with a TCP flow related to network congestion from a traffic passively measured at an intermediate node using machine learning-based techniques. Our *general* prediction model handles multiple scenario settings and it can also work with different variants of TCP congestion control algorithms.

Our experimental results demonstrate the feasibility of our prediction model. We believe that our study has a potential opportunity and benefit for network operators in characterizing the operations of Internet service providers and a better understanding of the widely deployed implementations of TCP congestion control flavors in the Internet. It will also be potentially useful to researchers and scientists in the computer networking community who want to assess the characteristics of TCP transmission states related to network congestion from passive measurements.

OUR CONTRIBUTIONS

The summaries of our contribution in this paper are the following:

- We demonstrate how the intermediate node (e.g., a network operator) can identify the transmission state of the TCP client associated with a TCP flow and predict the *cwnd* size of the sender from passive measurements.
- We identify a set of methodological challenges involved in performing inference of TCP per-connection states from passive measurements.
- We explore the applicability of our *general* prediction model by presenting a *robust* and *scalable* methodology to uniquely identify the widely deployed underlying TCP variants that the TCP client is using.

- We show that the learned prediction model performs reasonably well by leveraging knowledge from the emulated network when it is applied on a real-life scenario setting. Thus our prediction model is *general* bearing similarity to the concept of transfer learning in the machine learning community [5], [30], [38]. This guarantees that our prediction model is able to discern the results to unforeseen scenarios.
- We validate the robustness and scalability approach of our prediction model extensively through a large number of controlled experiments and experimentally verified across an *emulated*, *realistic* and *combined* scenario settings and across multiple TCP variants.

II. MOTIVATION

TCP congestion control algorithms have a critical role in improving the performance of TCP and regulating the amount of network traffic on the Internet by preventing congestion collapse [9]. However, it is a challenging task to predict whether a complex network has a normal behavior or not and analyze network dynamics. One of the most important elements of TCP sender state that can help us study the characteristics of TCP per-connection states in the Internet is *cwnd*. For example, it can be used to determine the factors that limit the network throughput, to predict the underlying TCP variant and efficiently identify non-conforming TCP senders etc. However, when different variants of TCP algorithms coexist on a network, they can potentially influence the performance of each other. One approach to solve this issue is to control the TCP flows individually by uniquely identifying the underlying TCP variant. Here we can ask questions like:

- What is the reason someone needs to know which algorithm the TCP sender is using?
- Is there some action that someone would take based on knowing the information of the underlying TCP variant of the sender?

From an operational perspective, we argue that this information is useful for network operators to monitor if major content providers (e.g., *Google*, *Facebook*, *Netflix*, *Akamai* etc.) are manipulating their congestion windows in their servers to achieve more than their fair share of available bandwidth. Another scenario where network operators might find this information useful is if they have a path that they know is congested due to customer complaints, but the links using that path are not especially over-subscribed. In that case, details about the congestion window behavior of all the users on that path might be helpful in trying to diagnose the cause, i.e., *are there users that are using aggressive congestion control algorithms which are unfair and affecting other user's available bandwidth*?

From an ISP perspective, we believe knowledge about the TCP stack in use in the endpoints is useful for operators of big ISP networks that do much traffic engineering who need to move traffic from oversubscribed links. It can also be used to study the end-to-end characteristics of the TCP stack and *non-conformant* end-to-end traffic. In addition to this, researchers and scientists in the networking community from both academia and industry could use the information to evaluate and understand existing congestion control algorithms. It can also be used to diagnose TCP performance problems (e.g., to determine whether the sending application, the network or the receiving network stack are to blame for slow transmissions) in real-time. Another benefit might be to observe when large content providers implement their own custom congestion control behavior that does not match one of the known congestion control algorithms.

However, taking the nature of TCP, accurately predicting TCP per-connection states from passive measurement has a number of difficulties. One of the challenges is, for example, TCP packets can be lost between the sender and the intermediate monitor, or between the monitor and the receiver. If a TCP packet is lost before it reaches the intermediate node and is somehow retransmitted in order, there is no way we can determine whether a packet loss has occurred or not. Therefore, what the intermediate monitor sees may not be exactly what the sender or the receiver sees. This means what appears to be reordering from the intermediate node's perspective can actually be a retransmit (or vice versa). If a captured TCP packet at the intermediate node is lost before it reaches the destination, a *retransmission* will occur without sending an acknowledgment [20]. Acknowledgments can be lost between the sender and the intermediate monitor, or between the monitor and the receiver node.

If either the entire window of TCP packets are lost before the intermediate node or acknowledgments lost after the measuring point will lead to the overestimation of a *cwnd* [20]. In addition to this, end-to-end *delay* variations in the path preceding the intermediate monitor can also cause retransmissions that appear to be caused by an *Retransmission Timeout (RTO)* rather than a *fast retransmit* [21]. Because TCP packets are only halfway to their destination, the relative sequencing on the forward and reverse path can be confusing, e.g., retransmitted packets can be seen at the monitor shortly after acknowledgments that should have prevented their retransmission. This is possibly because the acknowledgments haven't yet reached their destination when they are observed at the monitoring point, so the receiver did not yet know that the packets were received before they decided to retransmit them. More on the location of the intermediate passive monitor and its effect on what we can infer from the passively collected measurements is found in [21]. In this paper, we advocate that machine learning-based approaches can give a better prediction accuracy of TCP sender connection states from passive measurements of traffic flows collected at an intermediate node by addressing the aforementioned practical challenges.

ROADMAP

The rest of the paper is organized as follows. Section III overviews the background of our study. In Section IV,

we review and give a detailed overview of the *state-of-the-art* and discuss closely related works on TCP variants research. In Section V, we describe our experimental setup for the evaluation. Section VI gives an overview of our methodology highlighting the machine learning techniques, performance measurement metrics used in our paper. Section VII presents detailed experimental results and the multiple scenario settings used to validate our prediction model. Finally, Section VIII concludes the paper and outlines directions of research for future extensions.

III. BACKGROUND

TCP congestion control is set to operate on the variability of bandwidth, different cross-traffic, RTT etc. Different TCP stacks come with a variety of features that will violate the assumptions we might make if we only look at one or two TCP implementations and for this very reason, the following are a list of the most widely used *loss-based* variations of TCP congestion control algorithms we consider in our work so as to cover the whole scope of the problem.

1) **TCP Reno**: Jacobson [19] is one of the most predominant implementations of TCP variant that implements the *Additive Increase and Multiplicative Decrease (AIMD)* scheme [6], which employs a conservative linear growth function for increasing the *cwnd* by one segment per RTT for each received ACK and multiplicative decrease function on encountering a packet loss per RTT. It includes the congestion control schemes of *slow start*, *congestion avoidance*, *fast retransmission*, *fast recovery*, and *timeout retransmission*. During a congestive collapse, Reno uses loss events as a back-off mechanism.
2) **TCP BIC**: BIC [39] is a predecessor of TCP CUBIC [15]. It is optimized for high speed networks with high *latency* and has been adopted as a default congestion control algorithm by Linux for many years replacing TCP-Reno [19]. It uses the concept of *binary search* algorithm along with the AIMD [6] in an attempt to find the maximum *cwnd* that will last longer period. BIC-TCP [39] stand out from other TCP algorithms in its *stability*, TCP *friendliness* and RTT *fairness*.
3) **TCP CUBIC**: CUBIC [15] is an enhanced version of BIC [39]. It is the default congestion control algorithm as part of the Linux kernel distribution configurations from version 2.6.19. CUBIC [15] is designed to modify the linear window growth function of existing TCP standards to be governed by a *cubic function* in order to improve the scalability of TCP over fast and long distance networks. It uses a similar window growth function as its predecessor (BIC [39]) and is designed to be less aggressive and fair to TCP in bandwidth usage than BIC [39] while maintaining the strengths of BIC [39] such as *stability*, window *scalability* and RTT *fairness*.

IV. RELATED WORK

Before delving into our methodologies and the experimental results of our paper, we believe it is important to better understand where to position our work compared to the previous related works. This section briefly discusses closely related studies on monitoring network traffic techniques and the per-connection characteristics of TCP congestion control algorithms from passive measurements. The techniques to monitor TCP per-connection characteristics are divided into *two* categories:

- *Active measurement*
- *Passive measurement*

While *active measurement* has received a lot of research attention, however, *passive measurement* remains still an under investigated research topic. Hence, in this paper, we try to bridge the gap and mainly focus on the *passive* measurement approach.

A. ACTIVE MEASUREMENT

This technique actively measures the TCP behaviors of Internet flows by injecting an artificial traffic into the network between at least two endpoints [25], [29]. It focuses mainly on active network monitoring and relies on the capability to inject specific traffic which is then monitored so as to measure service obtained from the network.

B. PASSIVE MEASUREMENT

In a passive measurement, passively collected packet traces are examined to measure TCP behaviors of Internet flows [13], [20], [31], [34], [43]. Passive measurement, unlike an active measurement, doesn't inject an artificial traffic into the network. It only measures the network without creating or modifying any real traffic on the network. Passive monitoring measurements are increasingly used by network operators and researchers in the networking community. Network operators can track the underlying TCP congestion control algorithms from passively collected traffic and analyze the traffic flows.

In the traditional methods of passive measurement, there has been much interest in the investigation of TCP connections aggregate properties and its characteristics in the global Internet. Another work of interest that is most closely related to our work is [20] which provides a passive measurement methodology to infer and keep track of the values of the sender variables: end-to-end RTT and *cwnd*. Their idea is to emulate a *state transition* by detecting RTO events at the sender and observing the ACKs which cause the sender to change the value of the *cwnd*. This work [20] considers only the predominant implementations of TCP (*Reno*, *NewReno* and *Tahoe*) and the basic idea is it constructs a *replica* of the TCP sender's state for each TCP connection observed at the intermediate node. The replica takes the form of a *finite state machine*. However, the use of a separate *state machine* for each variant is *unscalable* taking the many existing TCP variants into consideration. We also believe that the constructed *replica* [20] cannot manage to reverse

or backtrack the transitions taking the tremendous amount of data into consideration. Another limitation is that the *replica* may not observe the same sequence of packets as the sender and ACKs observed at the intermediate node may not also reach the sender.

As an extension of [20], the work in [21] presents a methodology to study the performance of TCP, classify *out-of-sequence* behavior of packets for retransmission so as to identify where congestion is occurring in the network, with the same measurement environment as in [20]. Similar to our work, Paxson [31] described a trace analyzer tool called *tcpanaly* that analyzes *tcpdump* traces, and reports on the differences in behavior of TCP implementations. The similarity between our work and [31] is that both works to infer and match the type of TCP flavor from a passive measurement. However, [31] mainly focuses on the differences between different TCP implementation stacks. Since our passive monitor, as shown in Figure 1, is located in between the sender and the receiver, it is a challenging task for us to perform a detailed case-by-case analysis and identify if a specific TCP sender behavior is due to events in the network or TCP protocol stack implementation problems of end systems. In this paper, our main goal is to estimate the *cwnd* size of a TCP client associated with a TCP flow and, as an extension of our previous work [16], predict the underlying TCP variant.

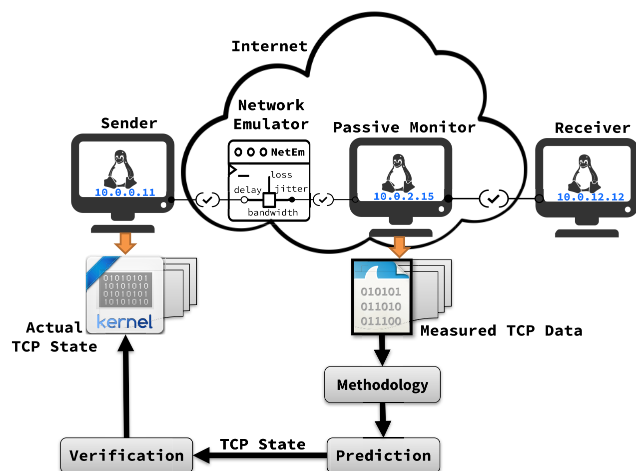

FIGURE 1. Experimental Testbed.

Rewaskar *et al.* [33] of the study developed a tool, called *tcpflows* that attempts to *passively* estimate the value of *cwnd* and identify TCP congestion control algorithms by analyzing the ACK stream to detect the occurrence of TCP congestion events. However, the *state machine* implemented with *tcpflows* is limited to old TCP variants and hence it cannot uniquely identify the newly deployed TCP congestion control algorithms. Oshio *et al.* [27] proposes a *cluster analysis-based* method that aims to identify between two versions TCP algorithms. This method was meant to be utilized in real-time applications to handle network traffic routing policies. It performs RTT and *cwnd* estimation in order to infer a group of traffic characteristics from the flow [27]. These characteristics are then clustered into two groups by applying a hierarchical clustering technique. Oshio *et al.* [27] show that only 2 out of 14 TCP congestion algorithms that are implemented in Linux can be identified based on their method. Most of the line of research work in the literature on the unique identification of the underlying variant of TCP congestion control algorithm from passive measurements focus on earlier flavors of TCP [20], [31]. Our work mainly differs from the previous research works in that our main goal is more fundamentally to develop a *robust*, *scalable* and *generic* prediction model for inferring TCP per-connection states for the most widely used *loss-based* congestion control algorithms including the newly deployed algorithms (e.g., BIC [39], CUBIC [15], Reno [19] etc.).

V. CONTROLLED EXPERIMENTS

In this section, we briefly explain the building blocks of our experimental test bed that we use to run controlled experiments that emulate the network.

A. EXPERIMENTAL SETUP

We describe our experimental procedure below. Figure 1 shows the experimental setup that we use for all of our experiments. We first created an emulated network and put a communication tunnel across the network and simultaneously push TCP cross-traffic to the network using an *iperf* traffic generator [12] so as to create a congestion. During a single TCP flow of our experiment, the parameters *bandwidth*, and *delay* are *constant* with a *uniform* distribution. However, since we have the *jitter* given as an average, its distribution is *normal*. We created an identical regular *tcpdump* of the TCP packets on the client node including information about the per-connection *states* so that we can match the *tcpdump* with the TCP *states*.

The passive monitor shown in Figure 1 is a separate Linux machine acting as a proxy. It is designed to do the *tcpdump* on all the interfaces available in the system and at the same time we want to predict what the per-connection state of a TCP packet was when it arrives in the monitor. It is important to remember that the traces we obtain from the *tcpdump* have no labels associated with them. Finally, we verified the predicted TCP states with the actual TCP kernel states directly logged from the Linux kernel of the sender used only for training whose data format output is shown in Table 1 and generate a new data for the learning model to predict on. One advantage of the sender is that it has a direct information about the outgoing packets and TCP states [36]. Once we finish with the verification of the TCP states, we run our learning model on the data and get the predictions.

TESTBED HARDWARE

We also validated our prediction model in an experimental test bed. Our experiments are performed using a cluster of machines based upon the GNU/Linux operating system running a modified version of the *4.4.0-75-generic*

TABLE 1. TCP Probe outputs from the sender-side kernel.

Column	Variable	Description
1	*tstamp*	Kernel Timestamps
2	*saddr:sport*	Sender Address:*port*
3	*daddr:dport*	Receiver Address:*port*
4	*length*	Packet Length (Bytes in packet)
5	*snd_nxt*	Next Send Sequence Number
6	*snd_una*	Unacknowledged Sequence Number
7	*snd_cwnd*	Congestion Window
8	*ssthresh*	Slow Start Threshold
9	*snd_wnd*	Send Window
10	*srtt*	Smoothed RTT
11	*tcp_ca_state*	Congestion Avoidance State

kernel release. We have performed our prediction experiment in two different environments based on the computational cost. The *GridSearchCV* for *Random Forest Regressor* model is performed on an NVIDIA Tesla K80 GPU accelerator computing with the following characteristics: Intel(R) Xeon(R) CPU E5-2670 v3 @2.30GHz, 64 CPU processors, 128GB RAM, 12 CPU cores running under Linux 64-bit. Whereas the *Gradient Boosting* model with a higher number of *boosting estimators* and *learning rates* that are used to scale the step length of the gradient descent procedure are performed on an HPC cluster with 700+ nodes where most nodes have 16 cores and 64 GiB memory of which 11,000+ cores and 52 TiB of memory are available in total as it needs more computational power for iterations. The CPUs in the computing cluster are 8-core 2.6 GHz Intel E5-2670. All nodes in the cluster are connected to a low latency 56 Gbit/s *Infiniband* network, *gigabit* Ethernet and have access to 600 TiB of *BeeGFS* parallel file system storage.

B. NETWORK EMULATION

TCP congestion control is set to operate on the variability of bandwidth, different cross-traffic, RTT, etc. Therefore, in order to create a realistic scenario, we have emulated the network in our setup as it is shown in Figure 1 by adding variability within a flow to the important network emulation parameters presented in Table 2. For the network emulation, we used the popular Linux-based network emulator, *Network Emulator (NetEm)* [18] on a separate node, that supports an end-to-end variability of *bandwidth*, *delay*, *jitter*, *packet loss*, *duplication* and more other parameters which the *cwnd* is influenced by to an outgoing packets of a selected network interface. The data traces for all our experiments are generated using the *iperf* [12] traffic generator on an emulated LAN link where we run each TCP variant with an end-to-end variation of the emulation parameters shown below where the *cwnd* is highly influenced by.

TABLE 2. Network emulation parameters.

	Bandwidth (*mbit*)	Delay (*ms*)	Jitter (*ms*)	Packet Loss (%)
1	10	1	0.001	0.01
2	100	2	0.1	0.05
3	300	3	0.2	0.1
4	500	5	0.5	1
5	700	7	1	1.5
6	1000	10	2	2
		[×6]	[×6]	[×6]

C. VERIFICATION OF THE EMULATOR

Given that the software emulator is not precise, can we trust the network emulator for all the end-to-end variations of *bandwidth*, *delay*, *jitter* and *packet loss* parameters that we change as shown in Table 2 for our evaluation irrespective of the measurement we get from TCP stream? As part of our study, we have also carefully investigated the precision of the network emulator, *NetEm* [18], we employed in this paper in order to use the tool with great care in an extremely well-contained environment. We created a filter that sets the parameter variation of each packet according to Table 2. As its precision cannot be measured from TCP stream, we setup a different experiment using *UDP* to evaluate and measure the precision where both the emulator and traffic generator create variations. We verified the raw performance by measuring the *bandwidth*, *delay*, *jitter* and *packet loss* variations created by the traffic generator and network emulator at the receiver side.

D. CROSS-TRAFFIC VARIABILITY

In our experimental setup of the emulator, we have carefully studied and validated our results in order to evaluate the impact of cross-traffic variability from the same TCP congestion protocol on our results by emulating other UDP traffic. *NetEm* [18] does lots of buffering and internally it has a buffer which is used to emulate a network by adding an end-to-end variability of *packet loss*, *delay*, *rate control* and other characteristics to packets outgoing from a selected network interface. Therefore, *NetEm* [18] (with a default *FIFO* queue) can also work in conjunction with other queuing disciplines (*qdisc*) by swapping the queue with another *qdisc*. It works well for traffic shaping and also supports a kernel level traffic shaping using the *Linux tc* utility. We ran *NetEm* [18] with variations in the data rate and the parameters presented in Table 2 between the client and the server and we found out that each variation run by *NetEm* [18] doesn't affect our results. We, therefore, believe that the variability of the cross-traffic in our current experimental setup will not impact our analysis. In general, when it comes to the *cwnd* variability, it will depend on the particular TCP congestion control in use. For example, TCP-Vegas [1] controls *cwnd* based on a queuing delay and *delay-based* congestion control algorithms thus may be affected by the variability of a cross traffic. We also believe the emulator may be impacted by network elements outside of its scope e.g., CPU load, busy devices, network card buffers, hardware architectural

factors etc. For example, cross-traffic in a real network is influenced by device resources that are used by both flows. Even if both flows are running on different interfaces and different line cards, there may be interaction due to buffer use and perhaps backplane occupancy.

E. TRAFFIC CAPTURES

The kernel might keep the TCP per-connection states of the packets in the buffer and waits for enough amount of packets before sending the TCP states to the userspace. TCP per-connection states might also get lost due to a slow process of TCP by the userspace process. Therefore, the first thing we did as a sanity check is to capture the packets at both the sender and the receiver for it helps us to know whether a packet was lost or just never sent as the ACKs from receiver to sender are just as important as the data packets for inferring packet loss. This way, it is possible to verify if the traffic captures are identical and there are no missing per-connection TCP states. The second thing we carried out in order to avoid missing of packets and capture exactly the same number of packets on the sender and the monitor is tuning the buffer size and flush the buffer to the userspace.

We carried out our experiment over a path that is jumbo-frame clean by disabling TCP segmentation offloading. Because we want to avoid packet sizes way over the regular legitimate Maximum Segment Size (MSS) and Maximum Transmission Unit (MTU) values. This is because, if we measure at a higher level and when packets are pushed down layer by layer on the protocol stack, the negotiated MSS will be violated. In order to avoid this violation, the TCP length must stay equal or below the MTU minus the IP and TCP header size. Every experiment of each TCP variant uses the same emulation setup parameters described in Table 2. Therefore, In all of our experiments, each TCP flow uses *1500-byte* data packets and an advertised window set by the operating system.

F. ASSUMPTIONS

In TCP, the *cwnd* is one of the main factors that determine the number of bytes that can be outstanding at any time. Hence, we assume that using the observed outstanding sequence of unacknowledged bytes on the network seen at any point in time in the lifetime of the connection as an estimate of the sending TCP's *cwnd* from *tcptrace* [28] when there is an end-to-end variability of *bandwidth*, *delay*, *loss* and *RTT* within a TCP connection is a better approach to estimate the *cwnd* and how fast the recovery is. Firstly, we assume that we don't know what TCP variant is running on the network and the per-connection state within the variant. Secondly, the results we present in this paper assume that the sender and receiver have the same *receiver window* in all of our measurements set by the operating system independent of the underlying TCP variant. Thirdly, in order to identify the TCP implementation of the client, we make use of the fact that the number of *outstanding bytes in flight* of the client cannot be more than its usable window size.

VI. METHODOLOGY

In this section, we describe the overall description of our approaches for experimentally inferring both the *cwnd* and uniquely identifying the underlying TCP variant from a passive measurement using a *general* machine learning-based techniques.

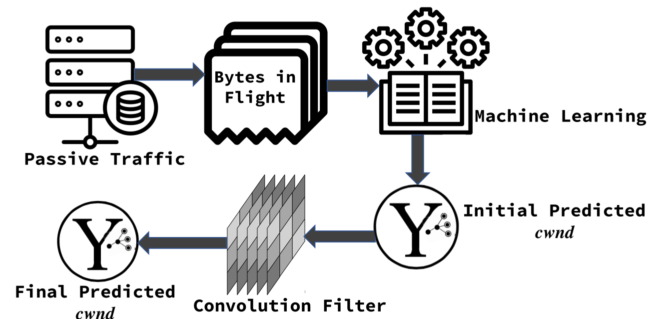

FIGURE 2. Methodology for *cwnd* prediction.

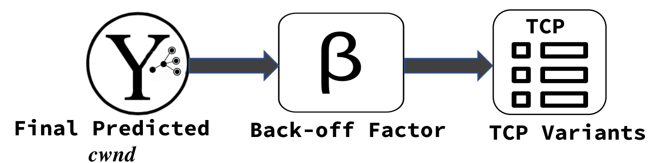

FIGURE 3. Methodology for *TCP Variant* prediction.

A. PASSIVE MONITORING OF BYTES_IN_FLIGHT

The measured passive traffic collected at the intermediate node as shown in Figure 1 is used for a training experiment of our model. The TCP implementation details and use of TCP options are not visible at the intermediate monitoring point. A TCP sender includes a *sequence number* to identify every unique data packets sent into the network. The TCP sender also keeps track of *outstanding bytes* by two variables in the kernel: *snd_nxt* (the sequence number of the next packet to be sent) and *snd_una* (the smallest unacknowledged sequence number, i.e., a record of the sequence number associated with the last ACK). This is because the TCP congestion control algorithms govern the TCP sender's sending rate by employing the *cwnd* that limits the number of cumulatively unacknowledged bytes that are allowed at any given time to do congestion avoidance [19]. From the passive traffic at the intermediate node, we can infer and manually analyze the number of bytes that have been sent but not yet cumulatively acknowledged on the network at a given point in time using *tcptrace* [28]. Figure 4 shows the comparison between the number of *outstanding bytes* from the intermediate node before running the *ensemble* model and applying the *convolutional filtering* techniques versus the actual *cwnd* tracked from the kernel of the sender-side.

Once we estimate the *cwnd* of the sender, we can infer the *multiplicative decrease* parameter (β) which is an important

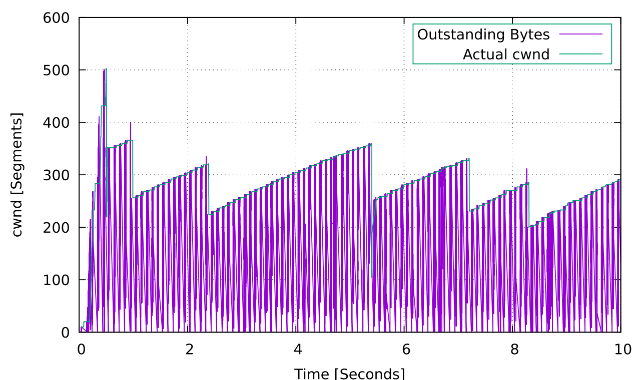

FIGURE 4. ***Outstanding bytes*** **calculated from the intermediate monitor using** ***tcptrace*** **[28] before applying** ***convolutional*** **filtering vs. the actual** ***cwnd*** **from the sender.**

feature for uniquely identifying TCP variants. This information is very useful in our experiment as it helps us match with the *cwnd* calculation of the particular TCP stack in use. Firstly, we run our *ensemble* model on the number of *outstanding bytes* which gives the *initial* predicted *cwnd* as it is shown in Figure 2. We then apply a *convolution filtering* technique, as it will be explained more in detail below in this Section, on the *initial* predicted *cwnd* which gives the *final* predicted *cwnd*.

Given that accurately inferring *cwnd* size from passive measurements is a challenging problem as it is not advertised, the most obvious approach is to try to use the observation of ACKs and retransmissions to predict whether the *cwnd* will increase or decrease. However, the effect of these events on the window will differ depending on the underlying TCP congestion control algorithm and the type of retransmission (e.g., fast retransmit versus a retransmit caused by a timeout). In order to estimate the *cwnd*, some research works assume that there is a congestion when the number of *bytes_in_flight* are below the *advertised window* by the receiver. However, if the number of *bytes_in_flights* are below the advertised window, it could also mean that the receiver has acknowledged packets before the advertised window was full. In this work, we are estimating *cwnd* from the calculated *bytes_in_flight* measured at the intermediate node calculated using *tcptrace* [28].

B. EXPERIMENTAL INFERENCE OF TCP CWND

The *cwnd* is a TCP per-connection state internal variable that represents the maximum amount of data a sender can potentially transmit at any given point in time based on the sender's network capacity and conditions. TCP [19] uses *cwnd* that determines the maximum number of *bytes* that can be *outstanding* without being acknowledged at any given time maintained independently by the sender to do congestion avoidance. TCP congestion control is set to operate on the variability of bandwidth, different cross-traffic, RTT etc. One initial approach we tried to estimate the *cwnd* was to process the packet headers of the flows in the *tcpdump* and calculate an aggregate TCP cross-traffic from the trace sets and add that as a feature. We, however, found out during our experiment that turns to be an insufficient detail for an accurate prediction. We have built a *convolutional* filtering technique in order to improve the accuracy of the prediction of TCP *cwnd* [16].

Another practical challenge of *cwnd* inference is when we place the passive monitor close to the receiver. If we try to measure the *cwnd* for the end-to-end path between the sender and the receiver basing our inference on the total amount of *outstanding* bytes, the further away from sender that our passive monitor is, the less likely it is that the packets that our monitor observes will match the packets that are used by the sending host to adjust its *cwnd*. For example, more hops between the sender and our passive monitor create more opportunities for packets to be lost, reordered or delayed. This means that the information we are using to infer congestion behavior (the packets observed at the passive monitor) is less reliable and introduces more opportunities for prediction algorithms to make false inferences. Because placing the monitor close to the receiver means, we will be seeing the ACKs before the sender does and so we may have more trouble estimating which of the data packets we capture were liberated by which of the ACKs we see. However, another technique we can try is to measure the size of the bursts of segments sent by the sender, where a burst is a series of segments that are sent back to back followed by a larger gap where no segments are sent. This is a lot trickier to perform – e.g., we need to be able to tell whether the timing gap between two data packets is a large inter-burst gap or just a slight delay between two packets in the same burst. But at least this allows us to mostly ignore the ACK stream from the receiver. We will address this approach in our next work.

In this work, we use the *python sklearn* library implementation [32] to build our *ensemble* machine learning prediction model using *Random Forest Regressor* algorithm [2] to estimate the *cwnd* where the entire *number of outstanding bytes in flight* is an input vector to the model. The size of the *Random Forest Regressor* model with the default parameters is $O(M * N * \log(N))$, where M is the number of trees and N is the number of samples. In order to further improve the performance of our *ensemble* prediction, we tuned the *Random Forest Regressor* optimal hyperparameters shown in Table 3 using a *GridSearchCV* that allows specifying only the ranges of values for optimal parameters by parallelization construction of the model fitting. In order to obtain an

TABLE 3. **Tuning parameters of the ensemble methods.**

n_estimators	*max_depth*	*min_samples_split*	*learning_rate*
10	1	2	0.1
100	2	5	0.2
300	3	10	0.3
500	5	20	0.5

optimal *cwnd* prediction model by minimizing the prediction function, we have also used *Gradient Boosting* algorithm [3] where the maximum number of features for the best split (*max_features*) is the same as *n_features*. We increased the variations of the tuning parameters in order to improve the initial TCP *cwnd* prediction fitting model by avoiding the risk of *overfitting* of the filters and fit the *ensemble* model by iteratively re-weighting the training outputs.

We trained our *ensemble* machine learning algorithm without the knowledge of the input features from the sender-side during the learning phase. We validated our methodology using the experimental test bed shown in Figure 1 over a LAN link. In order to train and test our prediction model, we employed every experiment with a ratio of 60% training, 40% testing split and a 5-fold *cross-validation* on all end-to-end variations of *bandwidth*, *delay*, *jitter* and *packet loss* into one robust and generic learning model. We learn the model from the training data and then finally predict the test labels from the testing instances on all variations of the emulation parameters. The *initial* prediction of TCP *cwnd* using a trained *ensemble* learning algorithm before optimizing the prediction performance using *convolution filtering* technique is shown in Figure 5.

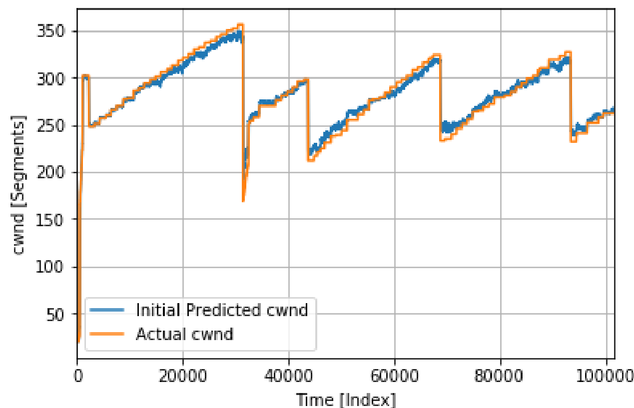

FIGURE 5. **Initial prediction of TCP *cwnd* versus the actual *cwnd* before applying the convolutional filtering technique.**

As it is shown in Table 5, we employ both the *Root Mean Square Error (RMSE)* and *Mean Absolute Percentage Error (MAPE)* metrics in order to evaluate our prediction model. The *MAPE* measures the absolute percentage error in our prediction model and is defined by the formula in Equation 1 where X is the actual input value to the model, Y is the target value and p is the learning model. For more information, we refer the interested readers to [8].

$$M = \frac{100}{n} \sum_{t=1}^{n} \left| \frac{p(X) - Y}{X} \right|, \quad X \neq 0 \tag{1}$$

C. CONVOLUTIONAL FILTERING

Convolutions are believed to have achieved an excellent performance in many applications (For example: [7], [10], [11], [23], etc.). Taking TCP packets dynamics and the complexity of accurately predicting *cwnd* from passive measurements, we have built a *convolutional* filtering technique in order to improve the accuracy of the *initial* prediction of TCP *cwnd* shown in Figure 5 and produce the *final* predicted value of *cwnd* shown in Figure 6 as per the methodology depicted in Figure 2. *Convolution* filtering technique is an operation on two complex-value functions f and g, which produces a third function that can be interpreted as a *filtered* version of f where the output is the full discrete linear convolution of the inputs. In Equation 2, g is the filter which in our case is the *final* predicted *cwnd* as shown in Figure 6.

$$f(x) * g(x) = \sum_{k=-\infty}^{\infty} f[k] \cdot g[x-k] \tag{2}$$

To perform the *final* prediction of TCP *cwnd*, we used *convolution filtering* to optimize the *initial* prediction accuracy of TCP *cwnd* obtained from tunning a *GridSearchCV* suite of parameters using a 5-fold *cross-validation* as shown in Table 3 and correctly recognize the patterns of the *cwnd* curves. As it is shown in Figure 6, the measured and actual *cwnd* match very well after we apply *convolution*. Our convolution method runs as a function taking the value of the *initial* predicted *cwnd*, a method to calculate the convolution, a mode which indicates the size of the output and a *standard deviation* of the fitting model as inputs to the function. We used a list comprehension to loop over the entire rows of the inputs from the *initial cwnd* prediction and pass the filtered data into an array for which the full convolution is computed. We have also *zero-pad* our convolution method in order to efficiently produce a full linear discrete result by preventing circular convolution. To calculate the convolution function for our evaluation of *cwnd* prediction, the *recommended* technique which automatically chooses either *Fast Fourier* or *direct* methods based on an estimate of which is faster is selected. In order to extract the valid part of the convolution which gives better smoothed sawtooth of the *cwnd* and detect the accurate pattern, we verified the equivalence of input and output sizes in every dimension through the parameter we pass to the *convolution* function. The *RMSE* and *MAPE* before optimizing the *initial* predicted value of TCP *cwnd* obtained from an *ensemble* model are 8.637 and 19.183% respectively. The *final* evaluation of TCP *cwnd* for the selected configurations after optimizing the initial predicted value of *cwnd* using *convolution* filtering technique are shown in Table 5.

D. PREDICTION OF TCP VARIANTS

Different end-to-end TCP algorithms widely in use behave differently under network congestion. Congestion control in any IP stack doesn't have much information available to drive its algorithm. It has to infer congestion from the history of packet loss and RTT. Our methodology for uniquely identifying the underlying TCP variant, by inferring the *multiplicative decrease* parameter (β) from the *final* predicted TCP *cwnd*, is shown in Figure 3. For the underlying TCP

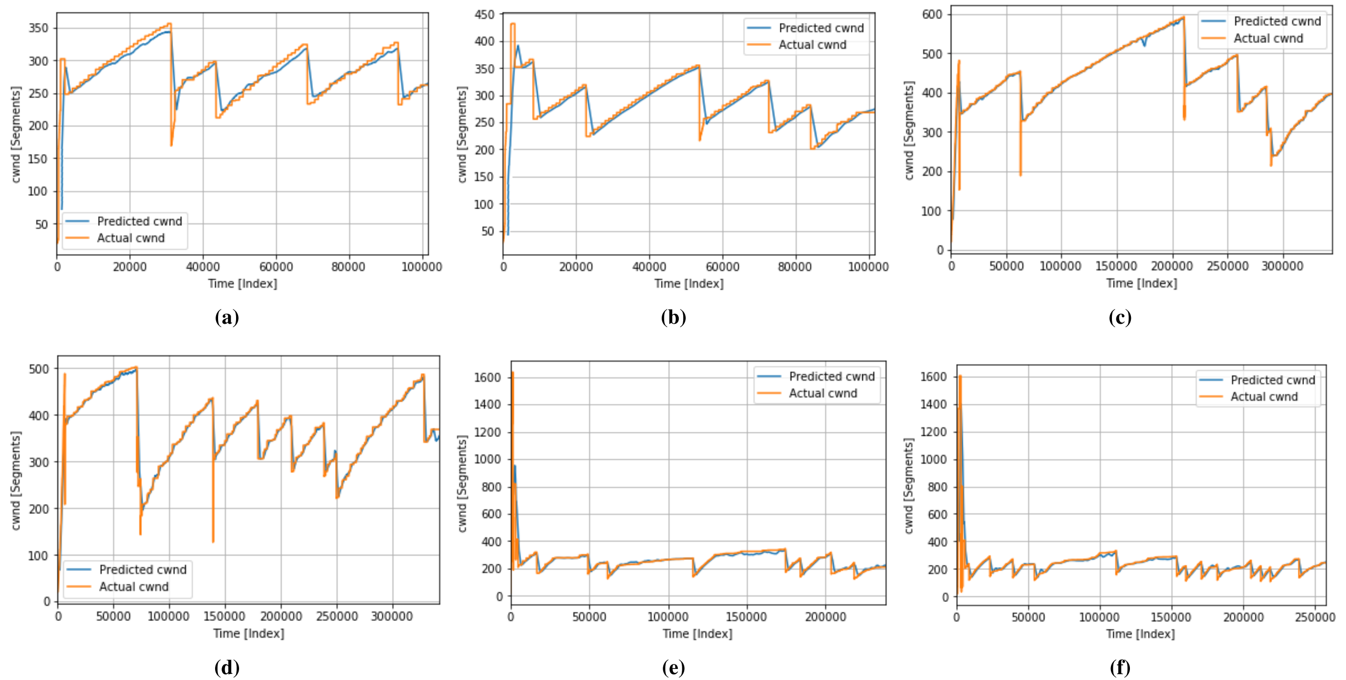

FIGURE 6. Final TCP *cwnd* prediction with different configurations of network emulation parameters for TCP CUBIC [15] and TCP Reno [19] after optimizing the *initial cwnd* prediction accuracy with *convolution* filtering technique in an emulated network. (a) CUBIC *final* predicted *cwnd* - Configuration C_1. (b) CUBIC *final* predicted *cwnd* - Configuration C_2. (c) CUBIC *final* predicted *cwnd* - Configuration C_3. (d) CUBIC *final* predicted *cwnd* - Configuration C_4. (e) Reno *final* predicted *cwnd* - Configuration C_1. (f) Reno *final* predicted *cwnd* - Configuration C_2.

variant prediction task, we consider only *loss-based* TCP congestion control algorithms that consider packet loss as an implicit indication of congestion by the network (e.g., CUBIC [15] BIC [39] and Reno [19]) for a proof of concept. As it is explained in Section II, since the global Internet is evolving from homogeneous to heterogeneous TCP congestion control algorithms, uniquely identifying the underlying TCP congestion control algorithm is a very important task. In practice; however, it is challenging to identify the TCP variant on the Internet taking the complexity and heterogeneity of congestion control algorithms into consideration [37]. One possibility would be to have a *state machine* model for each congestion control algorithm, and play the trace against the model to see if the trace is *consistent* with the model. However, there will again be some challenges, depending on where the trace is collected. Here we can ask questions:

- Do we see both directions of the traffic?
- Are we close to either endpoint, so we can hopefully estimate RTT accurately?
- How do we deal with the fact that some algorithms vary depending on past connections between the same pair of endpoints?
- How do we deal with the fact that sometimes a sender doesn't send a packet because of the congestion window but other times doesn't send because the application actually doesn't have any additional data in the send socket buffer?
- How do we deal with the varieties of old and modern operating system dependent TCP parameters?

As a solution to the aforementioned questions, in this paper we argue that training a classifier and *general* prediction model utilizing machine learning-based algorithms to uniquely identify the underlying TCP variant based on the *multiplicative decrease* window of the *cwnd* and the per-connection state within the variant from passive measurements collected at an intermediate node is very important. The standard TCP congestion algorithm employs an AIMD scheme that backs off in response to a single congestion indication [6]. A thorough analysis and evaluation of AIMD can be found in [6]. The AIMD has a linear growth function for *increasing* the *cwnd* at the receipt of an ACK packet and *multiplicative decrease* parameter, denoted by β, on encountering a TCP packet loss at the receipt of *triple duplicate ACKs* and it can be described as shown below in Function 3. This scheme adjusts the *cwnd* by the *increase-by-one decrease-to-half* strategy i.e., the TCP sending rate is controlled by a *cwnd* which is *halved* for every window of data containing a packet loss, and increased by one packet per window of segments are acknowledged.

$$\begin{aligned} Ack &: cwnd \leftarrow cwnd + \alpha \\ Loss &: cwnd \leftarrow \beta \times cwnd \end{aligned} \tag{3}$$

Most of the existing *loss-based* TCP congestion control algorithms implement AIMD scheme as it is proven to

converge [6]. It can generally be expressed as follows:

$$\uparrow_G: w_{t+R} \leftarrow w_t + \alpha; \alpha > 0$$
$$\downarrow_G: w_{t+\delta t} \leftarrow (1-\beta)w_t; 0 < \beta < 1, \quad (4)$$

Where $\uparrow_G$ refers to the increase in window as a result of the receipt of one window of acknowledgments in RTT and $\downarrow_G$ refers to the decrease in window on detection of network congestion by the sender, w_t is the window size at time t, R is the RTT of the flow and δ is a sampling rate. The AIMD algorithm is generalized by adding *two* variables, α and β that control the two aspects of AIMD: α indicates the increase in the window size if there is no packet loss in round-trip time and β indicates the fraction of the window size that it is decreased to when packet loss is detected [6]. Let $f(t)$ be the sending rate (e.g., the congestion window) during time slot t, $\alpha(\alpha>0)$, be the additive increase parameter, and $\beta(0<\beta<1)$ be the multiplicative decrease factor.

$$f(t+1) = \begin{cases} f(t)+\alpha, & \text{If congestion is detected} \\ f(t)\times\beta, & \text{If congestion is not detected} \end{cases} \quad (5)$$

In TCP, after *slow start*, the *additive increase* parameter α is typically one MSS every RTT, and the *multiplicative decrease* factor β on loss event is typically $\frac{1}{2}$ [6]. For example, CUBIC [15] decreases the *cwnd* whenever it detects that a segment was lost, either by using the TCP *Fast Retransmit* or *Fast Recovery* method of three duplicate ACK or when the *Retransmission Timeout* expires. And, it increases towards a target congestion window size (W) when in-order segments are acknowledged where W is defined by the following function:

$$W_{cubic}{}^{(t)} = |C(t-K)|^3 + W_{max} \quad (6)$$

Where W_{max} is the window size reached before the last packet loss event, C is a fixed scaling constant that determines the aggressiveness of window growth, t is the elapsed time from the last window reduction measured after the fast recovery, and where K is defined by the following function:

$$K = \sqrt[3]{\frac{W_{max}\beta}{C}} \quad (7)$$

Where β is a constant *multiplicative decrease* factor of CUBIC [15] applied for window reduction at the time of a TCP packet loss event (i.e., the window reduces to βW_{max} at the time of the last reduction) [15]. The β value of CUBIC [15] is 0.7, as shown in Table 4, which corresponds to reducing the window by 30% during a TCP packet loss event and can be calculated as per Equations 6 and 7.

TABLE 4. ***Loss-based*** **TCP variants β value.**

TCP Congestion Control Algorithm	β Value
BIC	0.8
CUBIC	0.7
Reno	0.5

The windows growth function of a TCP CUBIC [15] is a cubic function. TCP CUBIC [15] reduces its window by a factor of β after a loss event, the TCP-friendly rate per RTT would be $3((1-\beta)/(1+\beta))$ per RTT. Different congestion control algorithms have different window growth functions. However, when TCP BIC [39] detects a packet loss, it reduces its window by a multiplicative factor β. Its *cwnd* size just before the reduction is set to the *maximum* W_{max} (i.e., the window size just before the last fast recovery) and the window size just after the reduction is set to the current *minimum* W_{min} (i.e., $\beta \times W_{max}$). Then, BIC finally performs a binary search increase using these two parameters looking for the mid-point as shown in Equation 8.

$$\frac{W_{max}+W_{min}}{2} \quad (8)$$

The *multiplicative back-off* parameter, β, especially for *loss-based* congestion control algorithms is one of the most important TCP characteristics which determines important conditions of a network congestion like the *cwnd* and *Slow Start Threshold (ssthresh)* [40]. There are two approaches to measure the β value of a TCP congestion control algorithm: *(i)* using a packet loss event, and *(ii)* using a time out event. In the presence of a packet loss event, TCP sets both its *ssthresh* and the *cwnd* size to $\beta \times cwnd_loss$ where *cwnd_loss* is the *cwnd* size before a packet loss event or a time out occurs. When timeout occurs, TCP sets its *ssthresh* to $\beta \times cwnd_loss$ and its *cwnd* size to its initial congestion window (*init_cwnd*) size (1 or 2 segments depending on the TCP congestion control algorithm).

The *back-off* parameter along with other TCP characteristics (e.g., the rate at which the congestion window grows (α)) can be used to predict the underlying TCP congestion control algorithms. Hence, here we use the β value so as to uniquely predict the underlying TCP variant based on the multiplicative *back-off* factor of the selected *loss-based* TCP congestion control algorithms summarized in Table 4. Unlike *loss-based* algorithms, the β value of *delay-based* congestion control algorithms is not fixed. By design, *delay-based* TCP congestion control algorithms (e.g., TCP-Vegas [1], TCP-Westwood [14], etc.) have a variable β and the β value of these protocols will vary when there is variability in *delay* which makes it not easy to predict the variant from a passive traffic and we will address this in our next research work.

VII. EXPERIMENTAL SCENARIO SETTINGS RESULTS

Here, we explain in detail the experimental results of our main contributions: *(i)* Inferring TCP *cwnd* and *(ii)* Predicting the underlying TCP variants from passive measurements under multiple scenario settings. In the experimental evaluation, we choose a testing scenario configurations and present CUBIC [15], BIC [39] and Reno [19] in order to make our obtained evaluation results easily readable. We have experimented with several variations (36 configurations for each TCP variant, 216 in total as presented in Table 2). Due to space limitation in this paper, we can not present all the

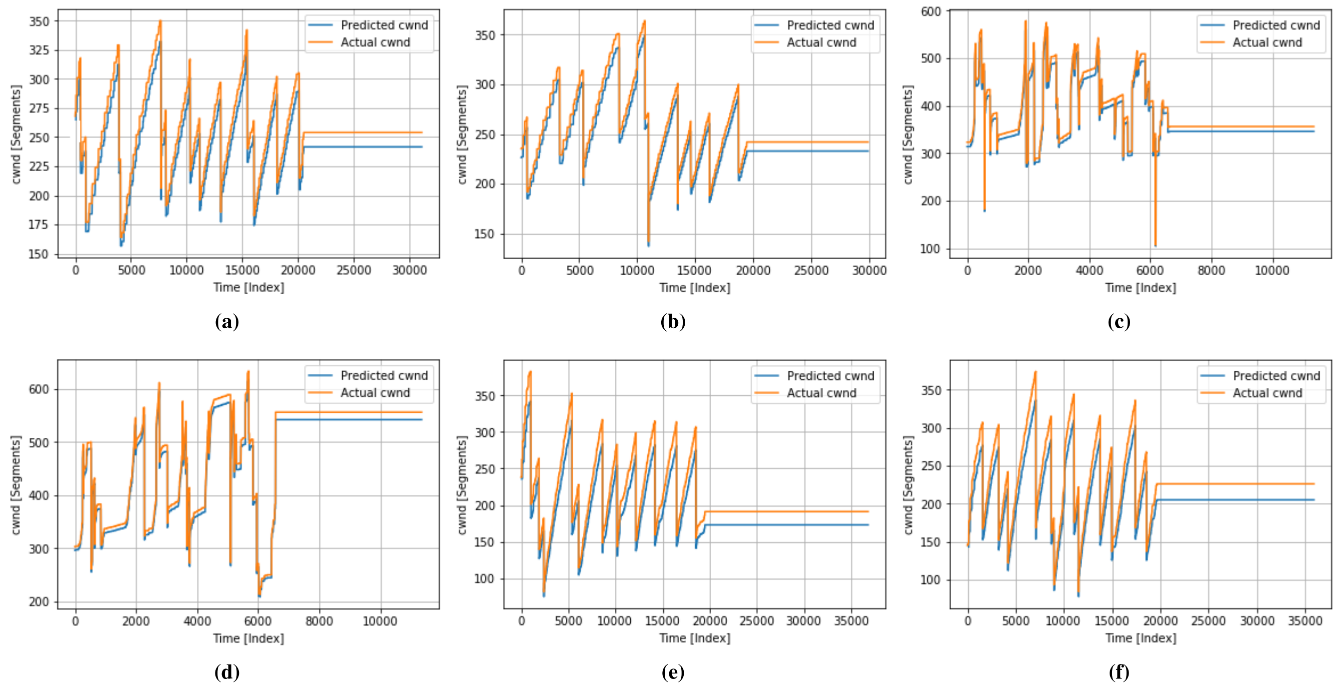

FIGURE 7. TCP *cwnd* prediction of TCP CUBIC [15], TCP BIC [39] and TCP Reno [19] from a *realistic* scenario on different *zones* of *Google Cloud* platform (East coast USA (North Carolina) and Northeast Asia (Tokyo, Japan) sites). (a) CUBIC *final* predicted *cwnd*, USA site. (b) CUBIC *final* predicted *cwnd*, Northeast Asia site. (c) BIC *final* predicted *cwnd*, USA site. (d) BIC *final* predicted *cwnd*, Northeast Asia site. (e) Reno *final* predicted *cwnd*, USA site. (f) Reno *final* predicted *cwnd*, Northeast Asia site.

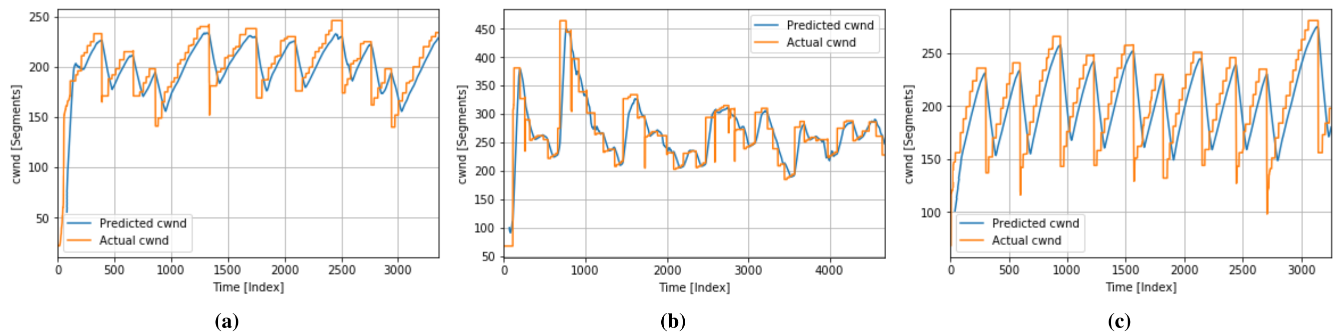

FIGURE 8. TCP *cwnd* prediction of TCP CUBIC [15], TCP BIC [39] and TCP Reno [19] from a *combined* scenario setting. (a) CUBIC *final* predicted *cwnd*, combined scenario. (b) BIC *final* predicted *cwnd*, Northeast Asia site. (c) Reno *final* predicted *cwnd*, USA site.

evaluation plots for a total of 216 configurations. Hence the results reported in this paper for all the scenario settings are for a subset of the selected configurations for a proof of concept as shown in Figures 6, 7 and 8 to verify the accuracy of our machine learning-based prediction model.

We evaluate our *final* TCP *cwnd* prediction model under different configurations of training and testing sample size ratios and the performance results are presented in Table 5. As it is shown in Figure 6, we found out the *convolutional* filtering we built for predicting *cwnd* captures the ratio of the *cwnd* drop very accurately. Figures 6*(a)* and *(b)* share the same bandwidth regardless of *delay*, *loss* and *jitter* configurations which cause the difference on the maximum number of segments over the course of the connection. For example, if we see on Figures 6*(c)* and *(d)*, Figure 6*(c)* has a *Bandwidth-Delay Product (BDP)* [22] of *700mb* 0.01s = 875,000 bytes*. At *1500 byte* segments, that's 583 segments and our emulation shows a maximum of 500-600 segments for *cwnd*. In all the plots we can see, once the timeout occurs, all the packet losses are handled with *fast recovery* in response to 3 *duplicate ACKs*. This is because the *cwnd* does not drop below half of its previous peak as it is shown in Figure 6. In the results, we can see there is a linear-increase phase followed by a packet loss event where the *cwnd* increases with new arriving ACK. This also demonstrates how the TCP congestion control algorithm responds to congestion events. We can see that the pattern of the *final* predicted *cwnd* generally matches the actual *cwnd*

quite well with a small prediction error. We matched both the increasing and decreasing parts of the sawtooth pattern using the precise *timestamp* obtained from the kernel.

A. EMULATED NETWORK SETUP

In Figure 6, the comparison of the *final* predicted TCP *cwnd* after optimizing the prediction performance using *convolution* filtering technique and the actual *cwnd* of the sender tracked from the *kernel* is presented. As it is shown in Figure 6, we found out the *convolutional* filtering we built for predicting *cwnd* captures the ratio of the *cwnd* drop very accurately. We evaluate our *final* TCP *cwnd* prediction model and the performance results are presented in Table 5. For the TCP variant prediction, we analyzed the β value by averaging out the window size of AIMD algorithm every time we have a peak so that we don't do the computation of the multiplicative decrease factor only on a *slow start* phase. The accuracy of uniquely identifying the underlying TCP variant prediction result in the *emulated* environment setting as presented in Table 7 is *93.51%*.

TABLE 5. TCP *final* predicted *cwnd* performance results of an *emulated network* setting with different configurations.

Algorithm	Configurations	RMSE	MAPE (%)
TCP CUBIC	*Final* predicted *cwnd* - C_1	5.839	6.953%
	Final predicted *cwnd* - C_2	3.075	3.725%
	Final predicted *cwnd* - C_3	2.209	2.857 %
	Final predicted *cwnd* - C_4	1.947	3.002%
TCP Reno	*Final* predicted *cwnd* - R_1	3.511	3.140%
	Final predicted *cwnd* - R_2	2.057	3.824%

TABLE 6. TCP variant prediction of an *emulated network* setting: confusion matrix.

	Predicted		
Actual	BIC	CUBIC	Reno
BIC	32	1	0
CUBIC	2	33	0
Reno	2	2	36

TABLE 7. TCP variant prediction of an *emulated network* setting: performance metrics.

	Precision	Recall	F1-Score	Support
BIC	0.89	0.97	0.93	33
CUBIC	0.92	0.94	0.93	35
Reno	1.00	0.90	0.95	40
Average/Total	0.94	0.94	0.94	108
Accuracy	**0.9351**			

B. REALISTIC SCENARIO SETUP

In order to demonstrate the *transferability* [5], [30], [38] approach of our proposed machine learning-based prediction model and further validate our results presented in Section VII by conducting a series of controlled experiments against other scenarios, we believe it is necessary to carefully test how well our model using an emulated network works with realistic scenarios by leveraging the knowledge of the emulated network. This guarantees that our prediction model is able to discern the results to unforeseen scenarios. Our experimental setup for this scenario setting is presented in Figure 9.

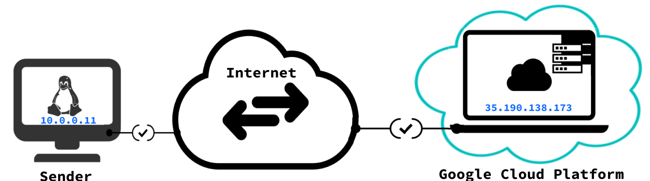

FIGURE 9. Realistic scenario setup.

From an experimental viewpoint, this helps us to justify and guarantee how our model could predict the development of a *cwnd* and the underlying TCP variant with other realistic network traffic scenarios captured from the Internet. To this end, we created a realistic test bed where we experiment from *Google Cloud* platform nodes by running our resources on the East coast of USA (South Carolina) and Northeast Asia (Tokyo, Japan) as shown in Figure 7. In order to create a realistic TCP session, we uploaded an *Ubuntu* image to *Google Cloud* platform sites so that we have a full control of the underlying TCP variant on the sender-side and at the same time run a *tcpdump* in the background and capture the whole TCP traffic flow for testing on the source node. We filtered out the host where we send the TCP traffic to. Finally, we calculated the number of *outstanding bytes* from the captured network traffic and run it through our learning model to predict the development of the TCP *cwnd* and variant. As it is shown in Figure 7, we confirm that our prediction model operates correctly and accurately recognizes the sawtooth pattern for realistic scenario settings across different *Google Cloud* platform *zones* as well. This shows that our prediction model is *general* bearing similarity to the concept of transfer learning in the machine learning community. The final *cwnd* prediction performance result of the realistic scenario setting across the *Google Cloud* platforms is presented in Table 8. As it is shown in Table 10, the accuracy of the TCP variant prediction for this scenario setting is *95%*.

TABLE 8. TCP *final* predicted *cwnd* performance results of a *realistic scenario* setting.

Congestion Algorithm	Google Cloud Zone	RMSE	MAPE (%)
TCP CUBIC	*USA* site	4.265	5.134%
	Japan site	3.522	4.738%
TCP BIC	*USA* site	2.952	3.809%
	Japan site	2.694	3.761%
TCP Reno	*USA* site	3.170	5.068%
	Japan site	3.396	5.197%

TABLE 9. TCP variant prediction of a *realistic scenario* setting: confusion matrix.

Actual	Predicted BIC	Predicted CUBIC	Predicted Reno
BIC	20	0	0
CUBIC	0	18	1
Reno	0	2	19

TABLE 10. TCP variant prediction of a *realistic scenario* setting: performance metrics.

	Precision	Recall	F1-Score	Support
BIC	1.00	1.00	1.00	20
CUBIC	0.90	0.95	0.92	19
Reno	0.95	0.90	0.93	21
Average/Total	0.95	0.95	0.95	60
Accuracy	**0.95**			

TABLE 11. TCP *final* predicted *cwnd* performance results of a *combined scenario* setting.

Algorithms	Per Configuration	RMSE	MAPE (%)
TCP CUBIC	*Configuration* C_1	5.704	8.053%
TCP BIC	*Configuration* B_1	5.193	7.831%
TCP Reno	*Configuration* R_1	4.752	5.739%

C. COMBINED SCENARIO SETTING

Real networks behave in more complex manner than emulated networks. The *loss* and *delay* of packets in TCP are both affected by, and affects, the TCP control loop. We believe, there are queue dynamics in the network which cause packet trains and other behaviors which software emulators like *NetEm* [18] can't reproduce well enough. In Section VII-B, we performed a realistic experiment when the random packet loss comes from the dynamics of multiple TCP connections sharing a link (congestion) rather than an injected packet loss. In this section, we address the scalability approach by conducting an experiment of our model under a broader range by combining the realistic and emulated scenario settings to justify the applicability and robustness of our prediction model. Our experimental setup for this scenario setting is presented in Figure 10.

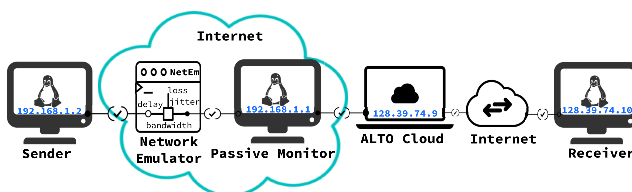

FIGURE 10. Combined scenario setup.

In this experiment, we combine the two scenario settings (one with an emulator and one with no emulator but Internet) where our intermediate node acts as a router. We get the traffic to the intermediate node, wrap and forward it to the network so that we can add more delay and the number of hops in the network on both sides. In this scenario, as it is shown in Figure 8, both the *increasing* and *decreasing* portions of the sawtooth pattern across different TCP variants is potentially accurate. The TCP variant prediction accuracy of the combined scenario setting, as it is presented in Table 13, is 91.66% and this justifies that our prediction model can handle multiple scenario settings.

TABLE 12. TCP variant prediction of a *combined scenario* setting: confusion matrix.

Actual	Predicted BIC	Predicted CUBIC	Predicted Reno
BIC	32	1	0
CUBIC	4	33	2
Reno	0	2	34

TABLE 13. TCP Variant Prediction of a *combined scenario* setting: performance metrics.

	Precision	Recall	F1-Score	Support
BIC	0.89	0.97	0.93	33
CUBIC	0.92	0.85	0.88	39
Reno	0.94	0.94	0.94	36
Average/Total	0.92	0.92	0.92	108
Accuracy	**0.9166**			

VIII. CONCLUSION AND FUTURE WORK

In this paper, we demonstrate how an intermediate node (e.g., a network operator) can identify the transmission state of the TCP client associated with a TCP flow by passively monitoring the TCP traffic. We presented a *robust*, *scalable* and *generic* machine learning-based prediction model that experimentally infers both TCP *cwnd* and the underlying variant of *loss-based* TCP congestion control algorithms within a flow from passive measurements collected at an intermediate node of the network. The significance of our paper is *two-fold*. First, it presents a prediction model for estimating TCP *cwnd* of the sender when there is variability within a flow. Our measurement results of the *cwnd* prediction show that we get a very good accuracy for both the *increasing* and *decreasing* portion of the sawtooth pattern. Second, this paper presents a *scalable* and *generic* learning model for predicting the widely deployed underlying TCP variants within a flow which may of interest for the network operators, researchers and scientists in the networking community from both academia and industry. In order to train and test our prediction model, we employed every experiment with a ratio of 60% training, 40% testing split and a 5-fold *cross-validation* on all end-to-end variations of *bandwidth*, *delay*, *jitter* and *packet loss* into one learning model. Our prediction model is tested under multiple scenario settings.

The experimental performance shows that the prediction model gives reasonably good performance on all the metrics

both in the *emulated*, *realistic* and *combined* scenario settings and across multiple TCP variants. We show that the learned prediction model performs reasonably well by leveraging knowledge from the emulated network when it is applied on a real-life scenario setting. Thus our prediction model is *general* bearing similarity to the concept of transfer learning in the machine learning community. The prediction accuracies of the underlying TCP variant for these scenario settings are 93.51%, 95%, and 91.66% respectively. To validate our evaluation of the prediction models, in addition to *accuracy*, we used multiple performance validation metrics such as *precision*, *recall*, *F1-Score* and *support*. Our evaluation across different scenario settings show that our model is effective and has considerable potential.

As a future work, there are many research avenues that can be explored. First, since now we are able to predict the *cwnd*, and the underlying TCP variant of *loss-based* congestion algorithms, we also think that we will be able to infer other TCP per-connection states. Second, it would be interesting to develop a *delay-based* method using both machine learning and deep learning techniques so as to verify how delay changes and look into how the TCP variants of *delay-based* congestion control algorithms can be predicted both from a passively measured traffic and real measurements over the Internet. Finally, we would like to design an approach based on machine learning techniques that is able to predict if a TCP packet loss is due to a buffer overflow in routers or wireless link in which two of them have different characteristics. Historically, TCP was designed for buffer overflow in routers and the action in TCP to back-off is based on the assumption that it is buffer overflow at a router as an implicit signal of network congestion. However, if we have another packet delay in the wireless link, the actions by TCP will not be necessarily the same because, in wireless networks, there might be a significant amount of packet loss due to corrupted packets as a result of interference. We plan to address these open issues and extend the approaches in our future work.

ACKNOWLEDGMENT

The authors gratefully acknowledge the anonymous reviewers for their thoughtful feedback and detailed comments. They would also like to thank the Research Infrastructure Services Group at the University of Oslo, Department of Informatics for the use of multicore cluster machines.

REFERENCES

[1] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson, *TCP Vegas: New Techniques for Congestion Detection and Avoidance*, vol. 24. New York, NY, USA: ACM, 1994.

[2] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001.

[3] P. Bühlmann and T. Hothorn, "Boosting algorithms: Regularization, prediction and model fitting," *Stat. Sci.*, vol. 22, no. 4, pp. 477–505, 2007.

[4] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "BBR: Congestion-based congestion control," *Commun. ACM*, vol. 60, no. 2, pp. 58–66, 2017.

[5] R. Caruana, "Multitask learning," in *Learning to Learn*. Boston, MA, USA: Springer, 1998, pp. 95–133.

[6] D.-M. Chiu and R. Jain, "Analysis of the increase and decrease algorithms for congestion avoidance in computer networks," *Comput. Netw. ISDN Syst.*, vol. 17, no. 1, pp. 1–14, 1989.

[7] Y. N. Dauphin, A. Fan, M. Auli, and D. Grangier. (2016). "Language modeling with gated convolutional networks." [Online]. Available: https://arxiv.org/abs/1612.08083

[8] A. de Myttenaere, B. Golden, B. Le Grand, and F. Rossi, "Mean absolute percentage error for regression models," *Neurocomputing*, vol. 192, pp. 38–48, Jun. 2016.

[9] N. Dukkipati, Y. Cheng, and A. Vahdat, "Research impacting the practice of congestion control," Assoc. Comput. Machinery, New York, NY, USA, 2016.

[10] V. Dumoulin and F. Visin. (2016). "A guide to convolution arithmetic for deep learning." [Online]. Available: https://arxiv.org/abs/1603.07285

[11] M. G. Elfeky, W. G. Aref, and A. K. Elmagarmid, "Periodicity detection in time series databases," *IEEE Trans. Knowl. Data Eng.*, vol. 17, no. 7, pp. 875–887, Jul. 2005.

[12] ESnet. (2017). *Iperf3*. [Online]. Available: https://iperf.fr/iperf-servers.php

[13] S. Gangam, J. Chandrashekar, Í. Cunha, and J. Kurose, "Estimating TCP latency approximately with passive measurements," in *Proc. Int. Conf. Passive Active Netw. Meas.*, Springer, 2013, pp. 83–93.

[14] M. Gerla, M. Y. Sanadidi, R. Wang, A. Zanella, C. Casetti, and S. Mascolo, "TCP Westwood: Congestion window control using bandwidth estimation," in *Proc. GLOBECOM*, Nov. 2001, pp. 1698–1702.

[15] S. Ha, I. Rhee, and L. Xu, "CUBIC: A new TCP-friendly high-speed TCP variant," *ACM SIGOPS Operat. Syst. Rev.*, vol. 42, no. 5, pp. 64–74, 2008.

[16] D. H. Hagos, P. E. Engelstad, A. Yazidi, and Ø. Kure, "A machine learning approach to TCP state monitoring from passive measurements," in *Proc. Wireless Days*, 2018, pp. 1–8.

[17] D. H. Hagos, A. Yazidi, Ø. Kure, and P. E. Engelstad, "Enhancing security attacks analysis using regularized machine learning techniques," in *Proc. IEEE 31st Int. Conf. Adv. Inf. Netw. Appl. (AINA)*, Mar. 2017, pp. 909–918.

[18] S. Hemminger, "Network emulation with NetEm," in *Proc. Linux Conf AU*, 2005, pp. 18–23.

[19] V. Jacobson, "Congestion avoidance and control," in *Proc. ACM SIGCOMM*, 1988, pp. 314–329.

[20] S. Jaiswal, G. Iannaccone, C. Diot, J. Kurose, and D. Towsley, "Inferring TCP connection characteristics through passive measurements," in *Proc. 23rd Annu. Joint Conf. IEEE Comput. Commun. Soc. (INFOCOM)*, vol. 3, 2004, pp. 1582–1592.

[21] S. Jaiswal, G. Iannaccone, C. Diot, J. Kurose, and D. Towsley, "Measurement and classification of out-of-sequence packets in a tier-1 IP backbone," *IEEE/ACM Trans. Netw.*, vol. 15, no. 1, pp. 54–66, Feb. 2007.

[22] D. Katabi, M. Handley, and C. Rohrs, "Congestion control for high bandwidth-delay product networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 32, no. 4, pp. 89–102, 2002.

[23] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2012, pp. 1097–1105.

[24] B. Mao *et al.*, "Routing or computing? The paradigm shift towards intelligent computer network packet transmission based on deep learning," *IEEE Trans. Comput.*, vol. 66, no. 11, pp. 1946–1960, 2017.

[25] A. Medina, M. Allman, and S. Floyd, "Measuring the evolution of transport protocols in the Internet," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 35, no. 2, pp. 37–52, 2005.

[26] T. T. Nguyen and G. Armitage, "A survey of techniques for Internet traffic classification using machine learning," *IEEE Commun. Surveys Tuts.*, vol. 10, no. 4, pp. 56–76, 4th Quart., 2008.

[27] J. Oshio, S. Ata, and I. Oka, "Identification of different TCP versions based on cluster analysis," in *Proc. 18th Int. Conf. Comput. Commun. Netw. (ICCCN)*, Aug. 2009, pp. 1–6.

[28] S. Ostermann. (2000). *Tcptrace*. [Online]. Available: http://www.tcptrace.org

[29] J. Pahdye and S. Floyd, "On inferring TCP behavior," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 31, no. 4, pp. 287–298, 2001.

[30] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE Trans. Knowl. Data Eng.*, vol. 22, no. 10, pp. 1345–1359, Oct. 2010.

[31] V. Paxson, "Automated packet trace analysis of TCP implementations," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 27, no. 4, pp. 167–179, 1997.

[32] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in Python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, Oct. 2011.

[33] S. Rewaskar, J. Kaur, and F. D. Smith, "A passive state-machine approach for accurate analysis of TCP out-of-sequence segments," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 36, no. 3, pp. 51–64, 2006.

[34] P. Romirer-Maierhofer, A. Coluccia, and T. Witek, "On the use of TCP passive measurements for anomaly detection: A case study from an operational 3G network," in *Proc. Int. Workshop Traffic Monit. Anal.*, Springer, 2010, pp. 183–197.

[35] R. Sommer and V. Paxson, "Outside the closed world: On using machine learning for network intrusion detection," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2010, pp. 305–316.

[36] S. Sundaresan, M. Allman, A. Dhamdhere, and K. Claffy, "TCP congestion signatures," in *Proc. Internet Meas. Conf.*, 2017, pp. 64–77.

[37] S. Sundaresan, X. Deng, Y. Feng, D. Lee, and A. Dhamdhere, "Challenges in inferring Internet congestion using throughput measurements," in *Proc. Internet Meas. Conf.*, 2017, pp. 43–56.

[38] L. Torrey and J. Shavlik, "Transfer learning," in *Handbook of Research on Machine Learning Applications and Trends: Algorithms, Methods, and Techniques*. Berlin, Germany: Springer, 2009.

[39] L. Xu, K. Harfoush, and I. Rhee, "Binary increase congestion control (BIC) for fast long-distance networks," in *Proc. 23rd Annu. Joint Conf. IEEE Comput. Commun. Soc.*, vol. 4, 2004, pp. 2514–2524.

[40] P. Yang, J. Shao, W. Luo, L. Xu, J. Deogun, and Y. Lu, "TCP congestion avoidance algorithm identification," *IEEE/ACM Trans. Netw.*, vol. 22, no. 4, pp. 1311–1324, Aug. 2014.

[41] J. Zhang, X. Chen, Y. Xiang, W. Zhou, and J. Wu, "Robust network traffic classification," *IEEE/ACM Trans. Netw.*, vol. 23, no. 4, pp. 1257–1270, Aug. 2015.

[42] J. Zhang, Y. Xiang, Y. Wang, W. Zhou, Y. Xiang, and Y. Guan, "Network traffic classification using correlation information," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 1, pp. 104–117, Jan. 2013.

[43] Y. Zhang, L. Breslau, V. Paxson, and S. Shenker, "On the characteristics and origins of Internet flow rates," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 32, no. 4, pp. 309–322, 2002.

DESTA HAILESELASSIE HAGOS received the B.Sc. degree in computer science from the Department of Computer Science, Mekelle University, and the M.Sc. degree in mobile systems from the Department of Computer Science Electrical and Space Engineering, Luleå University of Technology, Sweden, in 2012. He is currently a Ph.D. Research Fellow with the Autonomous Systems and Sensor Technologies Research Group, Department of Technology Systems, Faculty of Mathematics and Natural Sciences, University of Oslo. His current research interests include the areas of machine learning, deep learning, artificial intelligence, and computer networking.

PAAL E. ENGELSTAD received the bachelor's degree in physics from Norwegian University of Science and Technology (NTNU) in 1993, the master's degree (Hons.) in physics from NTNU/Kyoto University, Japan, in 1994, the bachelor's and Ph.D. degrees in computer science from University of Oslo in 2001 and 2005, respectively. He is currently a Full Professor with Oslo Metropolitan University. He is also a Research Scientist at Norwegian Defence Research Establishment and a Professor with the Autonomous Systems and Sensor Technologies Research Group, Department of Technology Systems, University of Oslo. He holds a number of patents and has been publishing a number of papers over the past years. His current research interests include fixed, wireless and ad hoc networking, cybersecurity, machine learning, and distributed and autonomous systems.

ANIS YAZIDI received the M.Sc. and Ph.D. degrees from University of Agder, Grimstad, Norway, in 2008 and 2012, respectively. He was a Researcher with Teknova AS, Norway. He is currently an Associate Professor with the Autonomous Systems and Networks Research Group, Department of Computer Science, Oslo Metropolitan University. His research interests include machine learning, learning automata, stochastic optimization, recommendation systems, pervasive computing, and the applications of these areas in industrial applications.

ØIVIND KURE received the Ph.D. degree from University of California, Berkeley, in 1988. He was a Senior Researcher with Telenor and the Research Manager with Telenor Research from 1989 to 2000. He is currently a Full Professor with the Center for Quantifiable Quality of Service in Communication Systems and the Department of Telematics, Norwegian University of Science and Technology, Trondheim, Norway, and the Autonomous Systems and Sensor Technologies Research Group, Department of Technology Systems, University of Oslo. His current research interests include the various aspects of QoS, data communication, performance analysis, and distributed operating systems.