UiO University of Oslo

Desta Haileselassie Hagos

# Discovering the Dynamic Complexity of TCP Using Machine Learning and Deep Learning Techniques

**Thesis submitted for the degree of Philosophiae Doctor**

Department of Informatics
Faculty of Mathematics and Natural Sciences

**2020**

*Series of dissertations submitted to the*
*Faculty of Mathematics and Natural Sciences, University of Oslo*
*No.* 2256

ISSN 1501-7710

Cover: Hanne Baadsgaard Utigard.
Print production: Reprosentralen, University of Oslo.

## Dedication

*This dissertation is dedicated to my mother Amlisha Mehari Tesfeu and my best friend Amha Tesfay Desta.*

"*All that I am or ever hope to be, I owe to my angel mother.*" [*Abraham Lincoln*]

# Preface

This doctoral dissertation is submitted to the Department of Informatics, Faculty of Mathematics and Natural Sciences, University of Oslo, in partial fulfillment of the requirements for the degree of *Philosophiae Doctor* (Ph.D.). The doctoral dissertation presented here is conducted under the main supervision of professor Paal E. Engelstad from the University of Oslo, Norwegian Defence Research Establishment (FFI), and Oslo Metropolitan University. My co-supervisors have been Professor Øivind Kure from the University of Oslo, and Professor Anis Yazidi from Oslo Metropolitan University. The research leading to this doctoral thesis has been carried out in the period from August 2015 to September 2019. As part of the Ph.D. program, one full-semester has been dedicated to course work of 30 credits from the University of Oslo, The Faculty of Mathematics and Natural Sciences, Department of Informatics. About 25% of the time has been dedicated to teaching and supervision of masters students and the remaining 75% was dedicated to research work. This research work has been fully supported for four years by the Research Council of Norway (RNC) under the Doctoral and post-doctoral research fellowships funding schemes for research and innovation.

## Acknowledgements

This dissertation wouldn't have been possible without the intellectual guidance, invaluable support and inspiration of all my supervisors to whom I wish to express my heartfelt gratitude.

Firstly, I would like to express my deepest gratitude to my main supervisor Professor Paal Engelstad for his incredible mentorship, critical inputs and suggestions, support of my research and for keeping me headed in the right direction since the beginning of my research work. His knowledge of computer networking, machine learning, deep learning and analytical way of interpreting experimental results is amazing and I have benefited from that greatly throughout my Ph.D. research work. Paal is a highly motivated and very intelligent supervisor who encourages his students to be bold in conducting scientific research and contribute something different to the research community. Besides, Paal is an encouraging professor who gives full research freedom to his students to work on what they like since the beginning in such a way that it builds their competence and confidence in conducting scientific research independently. Even more impressive, his sense of humor, simplicity, and friendliness are also equally amazing. Paal is very critical when it comes to the clarity of scientific writing and the rigorousness of argumentation. I could not, therefore, have imagined having the best and knowledgeable main supervisor for my Ph.D. than Professor Paal Engelstad. You are a great supervisor, Paal, and it is indeed a privilege to have had the opportunity of working with you and be supervised by you.

Secondly, I would like to thank my co-supervisor Professor Øivind Kure for reviewing the final drafts of my papers and providing valuable, insightful comments and feedback on every section of my papers. Professor Øivind's comments and questions are very detailed and critical. Øivind is full of new solid ideas and his critical feedback gives you a deep sense of research maturity.

Thirdly, Professor Anis Yazidi, as one of my co-supervisors, has been so generous with his time and kindly sharing his expertise on statistics, machine learning and deep learning with me. In my opinion taking all the positive energy, patience, kindness, and effort that he affords, Anis is the kind of supervisor any Ph.D. student would like to have. Professor Anis has always been willing to find some time out of his busy academic schedule for follow-up meetings, sometimes over the weekends and evenings, for a way forward discussions and to read the drafts of all of my papers with undivided attention. Anis is a very passionate and driven professor in cutting edge research with a lot of ideas and I have been extremely lucky to have him in the supervision committee of my research work.

Fourthly, this work has also greatly benefited from the incredibly valuable discussions with Professor Carsten Griwodz and I am grateful for his insightful inputs. Carsten is a very helpful and nice professor to work with who is deeply knowledgeable about TCP/IP protocols, the end-to-end performance of multimedia systems and operating systems. I would also sincerely like to gratefully acknowledge the Ph.D. full financial support provided to me by the Research Council of Norway (RNC) under the Doctoral and post-doctoral research fellowships funding schemes for research and innovation.

I am likewise truly indebted to my awesome and amazing colleagues: Ashish Rauniyar, Laura Andreína Marcano Canelones, Flavia Dias Casagrande, Ramtin Aryan, and Debesh Jha, without whom my Ph.D. life in Oslo would not have been so much fun. I am deeply grateful for all the fun, jokes and sometimes serious academic conversations over our lunch and dinner times. You all have been the source of my laughter and positive energy ever since we became colleagues. I have had so much fun along the way and my sincere thanks to you all buddies. There are still other very good friends, families, and colleagues, too many to list here, who I have had the opportunity to discuss ideas about my Ph.D. research work over the years to which I owe a great debt of heartfelt gratitude.

Last but not least, I would like to thank the two most important people in my life to whom this dissertation is dedicated: my brilliant and hard-working mother Amlisha Mehari Tesfeu; and my best friend Amha Tesfay Desta. Dear mom, your boundless support, unconditional love, courage, strength, resilience, constant source of encouragement in everything I do and your pro-education mentality is what has brought me this far, and I will forever be unbelievably grateful to you. For as long as I can remember, you have always believed in me and wholeheartedly encouraged and supported me to run after whatever it is that I passionately dream of pursuing in life. None of what I have achieved and all the miles I have traveled so far in my life would have been possible without you, mom. I owe everything I have ever achieved in my entire life to you, mom, and thank you for being my lifetime hero and my best friend.

Amish, you are one of the most thoughtful, loyal, dignified, compassionate and generous souls with a golden heart I have ever come in my life and it makes me extremely happy to proudly call you a dear brother and absolutely my best friend for life. Your positive attitude, profound knowledge, and understanding about a simple and ethical life, your true brotherhood and friendship have deeply influenced my life to which I am extraordinarily grateful. Most importantly, Amishey, you are one of the very few people who never gave up on me, no matter what and the best lifelong friend who has always been by my side through good and bad times. Of all your encouraging SMS messages I have received during this journey of my Ph.D. career, "The more I see you being strong and organized, I thank God! Always be on the right page of life and be grateful to God for everything you have.", stood out for me. In every step of my journey, I will try my level best to remain as strong and organized as possible and make you and my mom proud. Amishey, both you and my mother are constant reminders in my life of what it means to live with integrity and professionalism every day and this is something I will never ever forget as long as I live. Thank you!

**Desta Haileselassie Hagos**
Oslo, April 2020

# Abstract

Understanding the dynamic complexity of the internal states of TCP is a fundamental challenge, and particularly demanding due to the dynamics and complexity of modern networks. TCP is one of the key transport protocols of today's IP suite that supports most of the popular applications on the Internet. The main objective of this dissertation is to discover the dynamic complexity of TCP and obtain detailed knowledge about the end hosts from passive measurements using modern machine learning and deep learning techniques. Passive measurement has a clear advantage over active measurements since it doesn't generate traffic overhead to the underlying network. In the networking research community, there is an increasing interest in applying machine learning and deep learning techniques in different contexts. Machine learning approaches have effectively revolutionized and advanced the state-of-the-art for many research domain problems. In this dissertation, we study the applicability of state-of-the-art machine learning and deep learning approaches in computer networks by focusing on three main use cases: (*i*) TCP state monitoring from passive traffic measurements (*ii*) Network intrusion detection (*iii*) Passive operating system fingerprinting.

The main research questions around which this dissertation is centered are: (*i*) How can an intermediate node (e.g., a network operator) infer functionalities that determine a network condition from passive measurements? (*ii*) How can we enhance computer network security attack analysis using regularized machine learning techniques? (*iii*) Are we able to accurately classify the remote computer's operating system from passive measurements? Finally, this dissertation shows how an intermediate node can passively identify the transmission states of the TCP client associated with a TCP flow. We empirically demonstrate how the intermediate node can infer the cwnd size, predict at real-time the RTT between the sender and receiver, predict the underlying TCP variants of both loss-based and delay-based congestion control algorithms of the TCP client. Consequently, combining these contributions together, we built a deep learning-based universal tool for passive monitoring that can be applied to first estimate the cwnd, second predict the underlying TCP flavor and finally uses the predicted TCP variant as an input feature to passively fingerprint the remote computer's operating system. Our experimental results indicate the effectiveness of the proposed prediction models with reasonably high accuracy across different validation scenarios and multiple TCP variants. We believe that our work will be useful for the industry since passive measurements are becoming increasingly useful for network operators and Internet service providers to evaluate the communication performance of applications and services running on their networks.

"*Studying TCP is like studying natural processes. Even though the source code is easy to understand, we can no longer understand the effects of many TCPs in the wild..*"

*Professor Carsten Griwodz*

*University of Oslo, Simula Research Laboratory*

# Dissertation Structure

This dissertation is briefly organized as follows.

**Chapter 1** ***Introduction*** : provides a detailed overview of motivation, and the use cases that are relevant to this dissertation. It also describes the scientific methods used in this dissertation.

**Chapter 2** ***Summary and Contributions*** : presents a brief summary of the included papers in this dissertation and their scientific contributions which have been published in journals and international conferences.

**Chapter 3** ***Background*** : sketches some of the basic contextual background information that provides an overview of the background relevant to the reader and challenges that are dealt with in the research.

**Chapter 4** ***Related Work*** : presents a summary of the relevant state-of-the-art related works found in the literature for the three main use cases we presented in Chapter 1.

**Chapter 5** ***Conclusions*** : provides a summary of the research and highlights the main contributions of the dissertation. This chapter also provides considerations of the research and suggestions for promising future research directions.

**Part II** Contains the list of all the papers included in this dissertation.

# List of Papers

This Ph.D. dissertation is based on the following papers numbered from I to VIII. Since Paper III is a journal extension of Paper I and and Paper II, and Paper IX is a journal extension of Paper VIII, they are not included as part of this dissertation to avoid redundancy for the reader.

## Paper I

Desta Haileselassie Hagos, Paal E. Engelstad, Anis Yazidi, Øivind Kure. "A Machine Learning Approach to TCP State Monitoring from Passive Measurements".

***Published*** in the *2018 Wireless Days (WD)*, pp. 164–171. IEEE, 2018.
DOI: 10.1109/WD.2018.8361713.

## Paper II

Desta Haileselassie Hagos, Paal E. Engelstad, Anis Yazidi, Øivind Kure. "Towards a Robust and Scalable TCP Flavors Prediction Model from Passive Traffic".

***Published*** in the *27th International Conference on Computer Communication and Networks (ICCCN 2018)*, pp. 1–11. IEEE, 2018.
DOI: 10.1109/ICCCN.2018.8487396.

## Paper III

Desta Haileselassie Hagos, Paal E. Engelstad, Anis Yazidi, Øivind Kure. "General TCP State Inference Model From Passive Measurements Using Machine Learning Techniques".

***Published*** in *IEEE Access* 6 (2018): 28372–28387. IEEE, 2018.
DOI: 10.1109/ACCESS.2018.2833107.

## Paper IV

Desta Haileselassie Hagos, Paal E. Engelstad, Anis Yazidi, Øivind Kure. "Recurrent Neural Network-Based Prediction of TCP Transmission States from Passive Measurements".

***Published*** in the *17th IEEE International Symposium on Network Computing and Applications (NCA 2018)*, pp. 1–10. IEEE, 2018.
DOI: 10.1109/NCA.2018.8548064.

## Paper V

Desta Haileselassie Hagos, Paal E. Engelstad, Anis Yazidi, Carsten Griwodz. “A Deep Learning Approach to Dynamic Passive RTT Prediction Model for TCP”.
***Published*** in the *38th IEEE International Performance Computing and Communications Conference (IPCCC 2019)*. IEEE, 2019.
DOI: 10.1109/IPCCC47392.2019.8958727.

## Paper VI

Desta Haileselassie Hagos, Paal E. Engelstad, Anis Yazidi. “Classification of Delay-based TCP Algorithms From Passive Traffic Measurements”.
***Published*** in *the 18th IEEE International Symposium on Network Computing and Applications (NCA 2019)*. IEEE, 2019.
DOI: 10.1109/NCA.2019.8935063.

## Paper VII

Desta Haileselassie Hagos, Anis Yazidi, Øivind Kure, Paal E. Engelstad. “Enhancing Security Attacks Analysis Using Regularized Machine Learning Techniques”.
***Published*** in the *31st IEEE International Conference on Advanced Information Networking and Applications (AINA 2017)*, pp. 909–918. IEEE, 2017.
DOI: 10.1109/AINA.2017.19.

## Paper VIII

Desta Haileselassie Hagos, Martin Løland, Anis Yazidi, Øivind Kure, Paal E. Engelstad. “Advanced Passive Operating System Fingerprinting Using Machine Learning and Deep Learning”. ***Submitted for review***.

## Paper IX

Desta Haileselassie Hagos, Anis Yazidi, Øivind Kure, Paal E. Engelstad. “A Deep Learning-based Universal Tool for Operating Systems Fingerprinting from Passive Measurements”.
***Submitted for review***.

# Contents

# List of Figures

# List of Tables

# List of Acronyms

| | |
|---|---|
| **AIAD** | Additive Increase and Additive Decrease |
| **AIMD** | Additive Increase and Multiplicative Decrease |
| **BDP** | Bandwidth-Delay Product |
| **CDF** | cumulative distribution function |
| **cwnd** | Congestion Window |
| **DF** | Don't Fragment |
| **DNN** | Deep Neural Networks |
| **DoS** | Denial-of-Service |
| **ECDFs** | empirical cumulative distribution functions |
| **EWMA** | Exponential Weighted Moving Average |
| **ICMP** | Internet Control Message Protocol |
| **IDS** | Intrusion Detection Systems |
| **IoT** | Internet of Things |
| **IP** | Internet Protocol |
| **KNN** | K-Nearest Neighbors |
| **KS** | Kolmogorov-Smirnov |
| **LASSO** | Least Absolute Shrinkage and Selection Operator |
| **LSTM** | Long Short-Term Memory |
| **LVQ** | Learning Vector Quantization |
| **MAPE** | Mean Absolute Percentage Error |
| **ML** | Machine learning |
| **MLP** | Multilayer Perceptron |
| **MSS** | Maximum Segment Size |
| **MTU** | Maximum Transmission Unit |
| **NATs** | Network Address Translators |
| **NSD** | Norwegian center for research data |
| **OS** | Operating System |
| **OSes** | Operating Systems |

| | |
|---|---|
| **PDF** | probability distribution function |
| **RF** | Random Forest |
| **RBF** | Radial Basis Function |
| **ReLU** | Rectified Linear Unit |
| **R2L** | Remote-to-Local |
| **RFE** | Recursive Feature Elimination |
| **RMSE** | Root Mean Square Error |
| **RNN** | Recurrent Neural Networks |
| **RTO** | Retransmission Timeout |
| **RTT** | Round-Trip Time |
| **RTTVAR** | Round-Trip Time Variation |
| **SRTT** | Smoothed Round-Trip Time |
| **ssthresh** | Slow Start Threshold |
| **SVMs** | Support Vector Machines |
| **TCP** | Transmission Control Protocol |
| **TTL** | Time To Live |
| **U2R** | User-to-Root |
| **VMs** | Virtual Machines |

# Part I - Overview

Chapter 1

# General Context and Motivation

This chapter provides a detailed overview of the general context of the dissertation, motivation, and the use cases that are relevant to this Ph.D. dissertation.

## 1.1 Introduction

Inferring whether a complex and dynamic network operates under normal behavior is a fundamentally challenging problem especially when a few measurement points are monitored. In the networking research community, there is an increasing interest in applying state-of-the-art machine learning and deep learning techniques in computer networks in different contexts to infer the status of the network. In contrast to analytical models that require expert knowledge about the underlying network functioning modalities and rules, machine learning assumes no such prior knowledge. Machine learning constructs algorithms and models that can learn from data by unveiling an unforeseen pattern without prior human knowledge [22, 52, 55]. Deep learning, on the other hand, is characterized by a collection of computational neural network models that are composed of multiple processing layers capable of learning distributed representations of data with multiple levels of abstraction [5, 43]. For more detailed background information on machine learning and deep learning techniques, we refer our readers to Chapter 3.

Machine learning approaches have revolutionized and advanced the state-of-the-art for many research domain problems in the networking research community. For example, machine learning models are being actively applied and have been found effective in the areas of Internet traffic classification [56, 58], security monitoring and Intrusion Detection Systems (IDS) [4, 25, 74], flow clustering [53], fraud detection [51, 55], text classification [28, 72], face recognition [23], Spam detection [12, 34, 44, 62], image classification [45, 76], and many other fascinating topics in computer networks such as traffic anomaly detection [3]. In addition to this, legacy research works on Internet traffic classification using machine learning and statistical methods demonstrate that it is possible to characterize attributes of the data flow for a number of Transmission Control Protocol (TCP) applications [10, 16, 40, 41].

The TCP is one of the key protocols of today's Internet Protocol (IP) suite that supports most of the popular applications on the Internet today. It provides a connection-oriented communication by reliably sending data segments between the sender and the receiver. TCP handles network congestion by using a set of congestion control algorithms [17, 18]. One of the main responsibilities of congestion control is to ensure efficient and fair sharing of the network's limited resources among its users. The congestion control mechanism was added to TCP as a new feature in 1988 [29], after Van Jacobson observed the first Internet collapse in October 1986 due to congestion as shown in Figure 1.1. Starting with [29, 70], TCP congestion control and its performance analysis has maintained continuous research interest in the networking community [61]. Since the congestion control features of TCP has largely been responsible for the reliability and stability of the global Internet to date [17, 18], developing modern congestion control algorithms for TCP has been an active area of research. As the Internet continues to grow rapidly with the recent developments in high-speed networking capabilities supporting delay and loss-sensitive applications (e.g., video streaming), the implementation of advanced end-to-end congestion control algorithms is of increasing interest. In practice, however, this is fundamentally a challenging problem considering the complex behavior of networks.

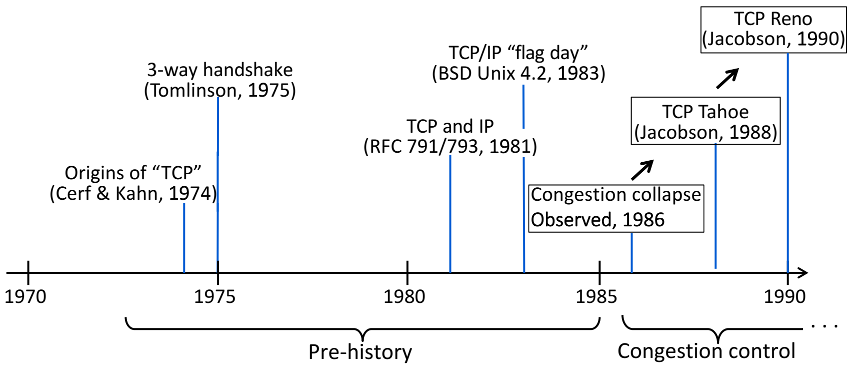

TCP Reno (Jacobson, 1990)
3-way handshake (Tomlinson, 1975)
TCP/IP "flag day" (BSD Unix 4.2, 1983)
TCP Tahoe (Jacobson, 1988)
TCP and IP (RFC 791/793, 1981)
Origins of "TCP" (Cerf & Kahn, 1974)
Congestion collapse Observed, 1986
1970
1975
1980
1985
1990
Pre-history
Congestion control
...

Figure 1.1: Historical background of TCP.

There are two main approaches to congestion control schemes: end-system (*source-based*) congestion control [2] and network-centric (*router-based*) congestion control [8, 37, 71]. End-system congestion control approaches are reactive, i.e., the source detects the congestion and reacts to it accordingly after getting implicit signals of congestion (e.g., packet loss or queueing delay). Based on the types of implicit congestion signals and other local information, the end-system congestion control techniques are further categorized into *loss-based* and *delay-based* variants as discussed in Chapter 3 in detail. On the other hand, the network-centric (*router-based*) congestion control approaches are proactive. This means since routers have more global information about the state of the underlying network infrastructure by continuously measuring the traffic load and queue length, they can proactively detect congestion and send a signal to the source node before the queue overflows. For more details about this approach, we refer the readers of this dissertation to [1, 69].

In this dissertation, we focus only on the end-system congestion control techniques. Adding congestion control management in the endpoints is critical and this is mainly because most of the intelligence on the Internet lies in the end hosts. In this dissertation, we employ state-of-the-art machine learning and deep learning techniques to monitor the internal states of the TCP client.

## 1.2 Motivation

The work presented in this dissertation aims at obtaining detailed knowledge about the end hosts by monitoring information of the packets that pass through the network, and by employing machine learning and deep learning-based techniques on the monitored network traffic. Since machine learning and deep learning methods are good at coping with complex tasks and massive amounts of data, they might play an important role to predict the TCP per-connection internal states. Understanding the dynamic complexity of the internal states of TCP is a fundamental challenge, and especially demanding due to the dynamics and complexity of modern networks. Even though this is the main objective of the dissertation, our work shows that related techniques can also be used to find other information about the hosts, such as their Operating System (OS) or TCP implementation or in a security perspective classify if the host's traffic is malicious or not.

The analyses of this dissertation focus mainly on TCP internal state monitoring from passive traffic measurements. We believe that our work will be useful for the industry since measurements are becoming increasingly useful for network operators and Internet Service Providers (ISPs) to evaluate the communication performance of applications and services running on their networks. Here, we summarize the benefits in the following three perspectives.

**Operational benefits**: Detailed knowledge about the underlying network by monitoring information of the packets that pass through the network is important for several reasons. For example, network operators can use this information to measure available bandwidth between endpoints, diagnose and troubleshoot network problems depending on the details of the information collected from the underlying network. We also believe that passively discovering the characteristics of TCP in an intermediate node has an operational advantage for network operators to monitor if major content providers (e.g., *Google*, *Facebook*, *Netflix*, *Akamai*, etc.) are manipulating their congestion control algorithms 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 there is 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 TCP Congestion Window (cwnd) 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*?

**Internet Service Provider (ISP) benefits**: Passively monitoring the TCP traffic at an intermediate node, allows the operators of big ISPs to assess the underlying network performance, which is crucial for their operation. We argue that detailed 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 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.

**Security ramifications**: We also believe passively observing the network-level characteristics found in TCP packets can give us more information about the remote computer's underlying OS. Hence, passively analyzing the internal states of the underlying TCP flavors is also useful for exploring security threats. This is because if we, for example, are able to infer the TCP variant, we can also make some guessing on the implementation of the underlying OS and search for security vulnerabilities. This can tell us about the encryption at the end-system that can be used to tailor-made attacks. We further believe that this will also help us to explore in detail the long-term characteristics of TCP traffic. The flip side of this is that the techniques presented in this dissertation also can be exploited by hackers. By knowing the OS, the hackers might also target known vulnerabilities of the detected OS. Thus, even though some of the intentions of our work are to provide tools to improve network analysis and security monitoring, the same technology might also be misused.

## 1.3 TCP Traffic Monitoring Techniques

In the networking community, there is a growing interest in observing Internet traffic characteristics at a given point of a network using end-to-end measurement techniques. Managing complex networks is extremely difficult due to the heterogeneity of communication networks. This represents an important challenge for network operators and ISPs. Many network operators would cope with this challenge by constantly monitoring the underlying network for analysis and further action e.g., in order to understand the state and the dynamic behavior of their network. As a result, there is a growing interest in observing Internet traffic characteristics at a given point of a network using end-to-end measurement techniques in the networking community.

The end-to-end measurement techniques to monitor the TCP per-connection characteristics are divided into two broad categories named active measurement and passive measurement. While active measurement has received a lot of research attention in the networking community, however, passive measurement remains still an under-investigated research topic. Hence, in this dissertation, we try to bridge the gap and mainly focus on passive measurement approaches.

### 1.3.1 Active Measurement

Much of the existing research work on traffic monitoring approaches rely on an active approach to measure the characteristics of TCP. This technique actively measures the TCP behaviors of Internet flows by injecting artificial traffic (e.g., probes) into the network between at least two endpoints [54, 60].

Active measurement techniques have several disadvantages. First, they introduce extra probing overhead traffic to the network. Second, often we have no control over either of the end-hosts of communication, so we cannot launch active measurements between the hosts. This is typical for a network operator that has not sufficient control over the equipment of the end customer. Finally, TCP probes or ping messages used in active measurements are often blocked by firewalls etc. Since it is common practice to disable probes by default, active measurement approaches are often prone to failure and their practical applicability is limited.

### 1.3.2 Passive Measurement

Passive measurement is at the heart of our work in this dissertation. In this technique, passively collected packet traces are examined to measure TCP behaviors of Internet flows [31, 63, 68]. Passive measurement doesn't inject artificial traffic into the network. It only measures the network without creating or modifying any real traffic on the network. In the traditional methods of passive measurement, there has been much interest in the investigation of TCP connections aggregate properties and their characteristics on the global Internet.

Passive measurement techniques of TCP flows have recently gained much attention in the networking research community lately [6, 11, 19, 35]. The main reason is that such measurements are becoming increasingly useful for network operators and ISPs to evaluate the communication performance of applications and services running on their network. Passively monitoring the traffic at an intermediate node, allows the ISPs to assess the underlying network performance, which is crucial for their operation. The main advantages of using passive measurements as compared to active measurements are that they do not put additional requirements on the configurations at the end hosts, they are not prone to failure due to firewalls etc., and they do not introduce additional traffic overhead. This dissertation focuses on using passive traffic measurements.

The TCP congestion control itself has grown increasingly complex which in practice makes inferring TCP per-connection internal states from passive traffic measurements collected at an intermediate node is a challenging task. Recently, the increasing practicality of leveraging state-of-the-art machine learning approaches has received considerable attention in overcoming critical challenges across many application domains some of which are presented above. However, the role of machine learning-driven models in computer network issues can be very broad. Hence, in this dissertation, we present our contributions by considering three main use cases of both machine learning and deep learning-based approaches in computer networks. These include: *(I)* TCP internal state monitoring from passive traffic measurements, *(II)* Security attack analysis based on passive traces, and *(III)* Passive OS fingerprinting.

## 1.4 Use Cases

In this dissertation, we study the applicability of state-of-the-art machine learning and deep learning techniques in computer networks by focusing on the following three main use cases.

### 1.4.1 Use Case 1: TCP State Monitoring from Passive Traffic Measurements

TCP is one of the dominant transport protocols that has played a great role in the exponential success of the Internet, network technologies and applications [29]. Many applications on the Internet use the reliable end-to-end TCP as a transport protocol due to practical considerations that favored TCP over other transport protocols [29]. TCP is a highly reliable end-to-end connection-oriented transport protocol designed to prevent excessive congestion on the Internet [29]. There are many different TCP variants in use, and each variant uses a specific end-to-end congestion control algorithm to avoid congestion, while also attempting to share the underlying network capacity equally among the competing users.

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 about inferring these functionalities that determine a network condition from passive traffic collected at an intermediate node of a network without having access to the sender?* In order to explore and answer this fundamental question, we first investigate evaluation methodologies for estimating cwnd in an intermediate node (e.g., network operator) from purely passive traffic measurements without the knowledge of the sender's cwnd for most of the widely used TCP variants in the Internet by leveraging both machine learning and deep learning-based techniques. We further expand our methodologies to predict the underlying TCP variants whose implicit signal of congestion are either packet loss or queueing delay across emulated and realistic settings.

The TCP congestion control is set to operate on the variability of bandwidth, different cross-traffic, RTT, etc. To deal with network congestion, as described above TCP uses congestion control algorithms to guide and regulate the network traffic on the Internet by avoiding sending more data than the underlying network is capable of transmitting which is maintained by the sender's cwnd. The global Internet highly relies on the TCP congestion control algorithms and adaptive applications that adjust its data rate to achieve high performance while avoiding congestion on the network [7]. One of the most important elements of the TCP sender state that can help us study the characteristics of the TCP per-connection states in a real-world setting 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, taking the nature of TCP, accurately inferring cwnd and its characteristics from passive traffic is a difficult problem. One of the difficulties 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). 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 [32]. 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, so the receiver did not yet know that the packets were received before they decided to retransmit them. More discussion on the location of the passive monitor and its effect on what we can infer from the measurements is found in [32]. In this dissertation, we argue that employing machine learning and deep learning-based techniques can also provide a potentially promising methodology for improving the accuracy of predicting TCP per-connection states from purely passive traffic measurements by addressing some of the practical challenges. For more detailed information on the methodologies and experimental results, we refer the reader to the included papers: Paper I, Paper II, Paper IV, Paper V, and Paper VI.

### 1.4.2 Use Case 2: Network Intrusion Detection

As the explosive rates of Internet growth and technological advancements continue, cyber defense is becoming an important and growing research area across wide ranges of application domains with direct commercial impact on public national enterprises, private organizations, and companies in every sector. The phenomenal growth of the global Internet has equally brought with it an undesirable increase in the number and variety of security attacks on Internet hosts. Any modern computer network needs to have robust and efficient mechanisms of detecting and deflecting any forms of security vulnerabilities. It also needs to be protected from any security violations, compromise of sensitive corporate data, computer abuse from unauthorized entry, etc. However, as computer and enterprise network systems have become more pervasive, dynamic and complex over the years, chances for attackers to compromise security flaws in these systems have also dramatically increased. A full list of security vulnerabilities for computer programs is presented in detail at [38]. Even though static computer network security mechanisms like a firewall can provide a fairly acceptable level of security, more modern and sophisticated IDS should be used in computer networks to automatically monitor the underlying traffic for any abnormal activities.

The role of IDS techniques is very crucial in monitoring computer network events for malicious activities, such as attacks against hosts and protecting computer systems and network infrastructures from a potential attack. From a security perspective, the problem with the evolution of network threats and attacks is that they are getting harder to detect and therefore it could be difficult to find out whether network traffic is normal or anomalous. Commercially available IDS tools are mainly signature-based that is designed to detect known malicious behaviors by using the precise signatures of those attacks. Such systems must be frequently updated with rule-sets and signature updates of the recent threat vectors, and are not capable of detecting potentially unknown attacks in network traffic.

Historically, several traditional IDS techniques use a signature-based approach in which events are detected and compared against a predefined database of precise signatures of known attacks that are provided by an administrator. The traditional approaches to IDS depend on experts or managers codifying rule-sets defining normal behavior and intrusions in a network [64, 73]. The two broad categories of IDS methods are: misuse and anomaly detection [15]. Misuse detection is a technique of searching for signatures based on patterns of known malicious behavior, either pre-configured by the system or set up manually by an administrator. This technique involves matching the signatures of known attacks in a network against events currently taking place in the system that should be considered as misuse [27, 36, 64]. We find this technique mostly used in operational settings. One of the main limitations of this approach is the failure of detecting and identifying either potentially unknown computer attacks that do not have explicit known signatures or slightly modified attacks whose precise signature is not included [27, 36, 46]. In this case, misuse detection is ineffective against such malicious behaviors because attacks can go undetected. Anomaly detection method, on the other hand, refers to the problem of finding patterns in data that do not comply with an expected notion of normal behavior in a system. Everything interpreted as a deviation from the profile of a normal system or user behavior is evidence of a malicious activity [14, 21, 39]. Anomaly detection, however, can detect previously unknown attacks but the problem with anomaly detection is that it has a higher false-positive rate mainly generated by the previously unseen behavior of the new attacks. Therefore, to effectively address this significant challenge, we argue it is important to use automated learning algorithms designed for large-scale anomaly detection.

Machine learning techniques have the potential of detecting unknown attacks in network traffic sharing features with other attacks by being trained on normal and abnormal types of traffic. However, one critical problem in machine learning is identifying and selecting the most relevant input features from which to construct an accurate model based on training data for a particular classification task. We, therefore, believe that it is important to do feature selection analysis to make it easier for network administrators to better understand the features that contribute to security attacks and consequently differentiate between normal and anomalous network behaviors. As a first use case in this dissertation work, we address the problem of an actual input feature selection for IDS to find security attack categories in a network through cross-validated regularized machine learning techniques and an artificial neural network feature ranking methods.

Selecting the most relevant actual features improves the detection quality for many algorithms that are based on learning techniques [26]. Feature selection helps to understand better which actual features are the most important ones to find attacks in a network. Therefore, in this use case, our focus is to analyze security attacks by exploring the contribution of the widely used actual input features and selecting the most contributory ones in effectively identifying anomalies in a network with respect to the attack categories. To that end, we have ranked the actual input features into strongly contributing, low contributory and irrelevant using a combination of feature selection filters and wrapper methods by carrying out comparisons with previous works. We investigate the most important features in identifying well-known security attacks by using Support Vector Machines (SVMs) and $\ell_1$-regularized method with Least Absolute Shrinkage and Selection Operator (LASSO). We use LASSO in particular for multiclass security attack classification to help us better understand which actual features shared by attacks in a network are the most important ones. LASSO is much more computationally effective and it provides coefficients that quantify how individual features affect the probability of specific security attack classes to occur. For more detailed experimental methodologies and evaluation results of this use case, we refer the reader to the included Paper VII.

### 1.4.3 Use Case 3: Passive Operating System Fingerprinting

Exploring the different implementations and characteristics of commonly used network protocols for security vulnerabilities is in the highest interest of network administrators. Consequently, taking the advantage of understanding the characteristics of the Transmission Control Protocol/Internet Protocol (TCP/IP) parameters, this further helps an administrator to remotely fingerprint the underlying OS without any application layer information for various reasons. OS fingerprinting is the process of carefully utilizing collected information of a machine that speaks TCP/IP in order to discover the underlying OS being run by a remote target device on the Internet without having physical access to the device [33]. As explained above, since the network infrastructures are rapidly growing in size, collecting detailed relevant knowledge about the dynamic characteristics and complexity of large heterogeneous networks is crucial for many purposes e.g., exploring network vulnerability assessment and monitoring. Developing advanced network security and monitoring techniques capable of a wide range of active and passive measurements are important for both the research and operational communities, as explained below.

Network scanning and accurate remote OS fingerprinting are the crucial steps for penetration testing in terms of security and privacy protection. Note that attackers can also embrace passive fingerprinting techniques to search for potential victims in a network. For example, by identifying the OS running on a remote computer and the list of services it runs, an attacker can target the device to eavesdrop on the communication between the endpoints without having physical access to the device. However, we argue that our work presented here is motivated by a number of practical applications that can be positively used by network administrators systems and networks.

Passively fingerprinting an OS by analyzing the packets it's generating and transmitted over a network is extremely important in the areas of network management and computer security for several reasons. For example, it is useful to explore a network for potential exploitations of security vulnerabilities which can be exploited by attackers, auditing, identifying critical attacks, revealing new information about a network user etc. In addition to this, it is also useful for network administrators to catalog a complete image and investigate the dynamic characteristics of large networks, to monitor unauthorized access and identifying rogue clients that may cause vulnerabilities in the network etc. Network administrators can, therefore, use this OS related information to maintain the security policy and reliability of their network by configuring a network-based IDS [48, 75]. Vulnerabilities and security threats in a network may result from rogue or unauthorized devices [77], unsecured internal nodes within the network and from external nodes [9]. Hence, passively fingerprinting an OS has a potential benefit in addressing these critical problems. This, from an academic point of view, is interesting and a topic that needs to be addressed from a network security research point of view.

Over the years, there has been a great deal of research work in the context of network management and cybersecurity on developing network security tools to fingerprint remote Operating Systems (OSes) [50, 57, 59, 78, 79]. There are many different implementations in fingerprinting of the most commonly used OSes based on the characteristics of its underlying TCP/IP network stack [33] and this, to a large extent, is due to variability in how the TCP/IP stack is traditionally implemented across different OSes [49]. One common approach, for example, is by collecting the TCP/IP stack basic parameters [47], e.g., IP initial Time To Live (TTL) default values [13], HTTP packets using the User-agent field [42], Internet Control Message Protocol (ICMP) requests [65], known open port patterns, TCP window size [30], TCP Maximum Segment Size (MSS) [67], IP Don't Fragment (DF) flag [66], a set of other specific TCP options, etc. However, in our work, we want to take this one step further by combining these basic features and with the underlying TCP variant as a distinguishing feature in our classification model due to the fact that different OSes have slightly different implementations of TCP. Some TCP congestion control algorithms, e.g., CUBIC [24], Reno [29], Veno [20], etc. quickly overshoot the size of the cwnd but we don't know why. Hence, we believe that knowing the implementation of the underlying OS may help us understand why they behave the way they do. It will also help us explore how to classify an OS when different OSes are implementing the same TCP congestion control algorithm.

Traditionally, most of the existing general OS fingerprinting techniques use manually generated signature matching from a database of heuristics which contains features of widely used OSes. This means, after comparing the generated signatures, the first set of responses match with the highest confidence against a database of fingerprints would be used to select the specific probable OS. However, manually updating a large number of signature and managing databases of new OSes adds a considerable amount of time and hence we may suffer from the consequences of the lack of recent signature updates of the known OSes. Consequently, newly developed computer and mobile OSes will not be recognized by these tools since they are not included in their fingerprint databases. Hence, we argue that it is important to consider making use of an up-to-date fingerprint database that contains variations of most currently used OSes and automating these tasks by employing learning algorithms capable of extracting all possible OS-specific features for discovering the underlying OSes. To explore this idea of applying learning algorithms, we present a robust classification approach to an advanced passive OS fingerprinting that leverages both machine learning and deep learning methods. We can determine what OS a remote computer on the Internet is running by either passively listening to traffic captured from a network or by actively sending packets to a target machine. Our fingerprinting technique is completely passive meaning that we only need to be able to observe network traffic from a target machine at any observation point in the network without injecting any traffic into the network. For more detailed experimental results and justifications, we refer the reader to the included Paper VIII.

## 1.5 Research Objectives and Methodology

The main objective of this dissertation is to explore passive monitoring tool of the internal states of a TCP session and provide some new deeper insights. The focus is on a network security engineering perspective and the tools are limited to state-of-the-art machine learning and deep learning techniques. To address the key objective of this dissertation, we consider the following three main research questions (one for each of the use cases discussed in detail in Section 1.4). For each of the main objectives, we identify a set of research questions that we attempt to answer in this dissertation.

1. How can an intermediate node (e.g., network operator) infer the per-connection internal transmission states of the TCP client associated with a TCP flow by passively monitoring the TCP traffic in an intermediate node of a network without having access to the kernel of the sender? This research question belongs to *Use Case 1*. It can be subdivided into four parts, detailing a different aspect of TCP and context for analysis.

   1.a) Paper I and Paper IV investigate how can an intermediate node passively predict the cwnd size of the TCP client using machine learning and deep learning techniques on both emulated and realistic settings? These papers consider only the underlying variants of loss-based TCP congestion control algorithms. We will employ a software emulator that supports an end-to-end variability of bandwidth, delay, jitter, packet loss, and other parameters that the cwnd is highly influenced by. Given that the software emulator is not precise, can we trust the network emulator for all the emulation parameters? The precision of the emulator for all the variations of bandwidth, delay, jitter and packet loss parameters and the impact of cross-traffic variability needs to be investigated.

1.b) Paper II investigates how can we experimentally infer the underlying variant of loss-based TCP algorithms within flow from passive traffic measurements collected at an intermediate node based on the total number of outstanding bytes?

1.c) Paper V investigates how can we dynamically predict at real-time the Round-trip Time (RTT) between the sender and receiver nodes based on passive measurements collected at an intermediate node?

1.d) There are TCP flavors that exploit queueing delay as a congestion signal and Paper VI discusses how can we classify the underlying variants of delay-based TCP congestion control algorithms from passive traffic measurements?

2. How can we understand the dynamics behind the security attacks classification models so that we create safe and human-interpretable systems? This research question belongs to *Use Case 2*. How can we enhance computer network security attack analysis using regularized machine learning techniques? Another aspect is how do we select the most important actual input features that are well understood within the networking community in identifying well-known security attacks? Paper VII addresses these research questions.

3. Is it feasible to classify the underlying OS of a remote computer from passive measurements when different OSes are implementing the same TCP variant? This research question belongs to *Use Case 3*. Paper VIII explores this issue.

Table 1.1: Linking the research objectives with our included papers.

| **Research Objectives** | **Included Papers** |
|---|---|
| 1.a | Paper I , Paper IV |
| 1.b | Paper II |
| 1.c | Paper V |
| 1.d | Paper VI |
| 2 | Paper VII |
| 3 | Paper VIII |

Table 1.1 sums up which research objective each of the papers addresses. The relationship between the papers is also detailed in Figure 2.1 where each paper's contributions are also included.

## 1.6 References

[1] R. Adams. Active queue management: A survey. *IEEE communications surveys & tutorials*, 15(3):1425–1476, 2012.

[2] A. Afanasyev, N. Tilley, P. Reiher, and L. Kleinrock. Host-to-host congestion control for TCP. *IEEE Communications surveys & tutorials*, 12(3):304–342, 2010.

[3] T. Ahmed, B. Oreshkin, and M. Coates. Machine learning approaches to network anomaly detection. In *Proceedings of the 2nd USENIX workshop on Tackling computer systems problems with machine learning techniques*, pages 1–6. USENIX Association, 2007.

[4] P. Barford and D. Plonka. Characteristics of network traffic flow anomalies. In *Internet Measurement Workshop*, pages 69–73. Citeseer, 2001.

[5] Y. Bengio, A. Courville, and P. Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013.

[6] P. Benko and A. Veres. A passive method for estimating end-to-end TCP packet loss. In *GLOBECOM'02*. IEEE, 2002.

[7] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, et al. BBR: congestion-based congestion control. *ACM*, 2017.

[8] G. Chatranon, M. A. Labrador, and S. Banerjee. A survey of TCP-friendly router-based AQM schemes. *Computer Communications*, 27(15):1424–1440, 2004.

[9] W. R. Cheswick, S. M. Bellovin, and A. D. Rubin. *Firewalls and Internet security: repelling the wily hacker*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[10] K. C. Claffy. *Internet traffic characterization*. PhD thesis, University of California, San Diego, Department of Computer Science . . . , 1994.

[11] J. Cleary, S. Donnelly, I. Graham, A. McGregor, and M. Pearson. Design principles for accurate passive measurement. In *Proceedings of PAM*, 2000.

[12] M. Crawford, T. M. Khoshgoftaar, J. D. Prusa, A. N. Richter, and H. Al Najada. Survey of review spam detection using machine learning techniques. *Journal of Big Data*, 2(1):23, 2015.

[13] N. Davids. Initial TTL values. http://noahdavids.org/self_published/TTL_values.html, 2011.

[14] D. E. Denning. An intrusion-detection model. *IEEE Transactions on software engineering*, (2):222–232, 1987.

[15] O. Depren, M. Topallar, E. Anarim, and M. K. Ciliz. An intelligent intrusion detection system (IDS) for anomaly and misuse detection in computer networks. *Expert systems with Applications*, 29(4):713–722, 2005.

[16] C. Dewes, A. Wichmann, and A. Feldmann. An analysis of internet chat systems. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, pages 51–64. ACM, 2003.

[17] M. Duke, E. Blanton, A. Zimmermann, R. Braden, and W. Eddy. A roadmap for transmission control protocol (TCP) specification documents. RFC 7414, 2015.

[18] K. R. Fall and W. R. Stevens. *TCP/IP illustrated, volume 1: The protocols.* addison-Wesley, 2011.

[19] C. Fraleigh, C. Diot, B. Lyles, S. Moon, P. Owezarski, D. Papagiannaki, and F. Tobagi. Design and deployment of a passive monitoring infrastructure. In *Thyrrhenian Internatinal Workshop on Digital Communications*, pages 556–575. Springer, 2001.

[20] C. P. Fu and S. C. Liew. TCP Veno: TCP enhancement for transmission over wireless access networks. *IEEE Journal on selected areas in communications*, 21(2):216–228, 2003.

[21] A. K. Ghosh, J. Wanken, and F. Charron. Detecting anomalous and unknown intrusions against programs. In *Proceedings 14th Annual Computer Security Applications Conference (Cat. No. 98EX217)*, pages 259–267. IEEE, 1998.

[22] I. Goodfellow, Y. Bengio, and A. Courville. *Deep learning.* MIT press, 2016.

[23] G. Guo, S. Z. Li, and K. Chan. Face recognition by support vector machines. In *Proceedings fourth IEEE international conference on automatic face and gesture recognition (cat. no. PR00580)*, pages 196–201. IEEE, 2000.

[24] S. Ha, I. Rhee, and L. Xu. CUBIC: a new TCP-friendly high-speed TCP variant. *ACM SIGOPS operating systems review*, 42(5):64–74, 2008.

[25] D. H. Hagos, A. Yazidi, Ø. Kure, and P. E. Engelstad. Enhancing security attacks analysis using regularized machine learning techniques. In *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*, pages 909–918. IEEE, 2017.

[26] F. Iglesias and T. Zseby. Analysis of network traffic features for anomaly detection. *Machine Learning*, 101(1-3):59–84, 2015.

[27] K. Ilgun, R. A. Kemmerer, and P. A. Porras. State transition analysis: A rule-based intrusion detection approach. *IEEE transactions on software engineering*, (3):181–199, 1995.

[28] D. J. Ittner, D. D. Lewis, and D. D. Ahn. Text categorization of low quality images. In *Symposium on Document Analysis and Information Retrieval*, pages 301–315. Citeseer, 1995.

[29] V. Jacobson. Congestion avoidance and control. In *ACM SIGCOMM computer communication review*, volume 18, pages 314–329. ACM, 1988.

[30] V. Jacobson, R. Braden, and D. Borman. TCP extensions for high performance. RFC 1323, 1992.

[31] S. Jaiswal, G. Iannaccone, C. Diot, J. Kurose, and D. Towsley. Inferring tcp connection characteristics through passive measurements. In *IEEE INFOCOM 2004*, volume 3, pages 1582–1592. IEEE, 2004.

[32] 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 Transactions on Networking (ToN)*, 15(1):54–66, 2007.

[33] T. Kohno, A. Broido, and K. C. Claffy. Remote physical device fingerprinting. *IEEE Transactions on Dependable and Secure Computing*, 2(2):93–108, 2005.

[34] P. Kolari, A. Java, T. Finin, T. Oates, A. Joshi, et al. Detecting spam blogs: A machine learning approach. In *Proceedings of the national conference on artificial intelligence*, volume 21, page 1351. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2006.

[35] M. Kühlewind, S. Neuner, and B. Trammell. On the state of ECN and TCP options on the Internet. In *International Conference on Passive and Active Network Measurement*, pages 135–144. Springer, 2013.

[36] S. Kumar and E. H. Spafford. A pattern matching model for misuse intrusion detection. 1994.

[37] M. A. Labrador and S. Banerjee. Packet dropping policies for ATM and IP networks. *IEEE Communications Surveys*, 2(3):2–14, 1999.

[38] C. E. Landwehr, A. R. Bull, J. P. McDermott, and W. S. Choi. A taxonomy of computer program security flaws. *ACM Computing Surveys (CSUR)*, 26(3):211–254, 1994.

[39] T. Lane and C. E. Brodley. Temporal sequence learning and data reduction for anomaly detection. *ACM Transactions on Information and System Security (TISSEC)*, 2(3):295–331, 1999.

[40] T. Lang, G. Armitage, P. Branch, and H.-Y. Choo. A synthetic traffic model for Half-Life. In *Australian Telecommunications Networks & Applications Conference*, volume 2003, 2003.

[41] T. Lang, P. Branch, and G. Armitage. A synthetic traffic model for Quake3. In *Proceedings of the 2004 ACM SIGCHI International Conference on Advances in computer entertainment technology*, pages 233–238. ACM, 2004.

[42] M. Lastovicka, T. Jirsik, P. Celeda, S. Spacek, and D. Filakovsky. Passive OS Fingerprinting Methods in the Jungle of Wireless Networks. In *NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium*, pages 1–9. IEEE, 2018.

[43] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[44] K. Lee, J. Caverlee, and S. Webb. Uncovering social spammers: social honeypots+ machine learning. In *Proceedings of the 33rd international ACM SIGIR conference on Research and development in information retrieval*, pages 435–442. ACM, 2010.

[45] J.-H. Lim. Learnable visual keywords for image classification. In *Proceedings of the fourth ACM conference on Digital libraries*, pages 139–145. ACM, 1999.

[46] U. Lindqvist and P. A. Porras. Detecting computer and network misuse through the production-based expert system toolset (P-BEST). In *Proceedings of the 1999 IEEE Symposium on Security and Privacy (Cat. No. 99CB36344)*, pages 146–161. IEEE, 1999.

[47] R. Lippmann, D. Fried, K. Piwowarski, and W. Streilein. Passive operating system identification from TCP/IP packet headers. In *Data Mining for Computer Security*. Citeseer, 2003.

[48] R. Lippmann, S. Webster, and D. Stetson. The effect of identifying vulnerabilities and patching software on the utility of network intrusion detection. In *International Workshop on Recent Advances in Intrusion Detection*, pages 307–326. Springer, 2002.

[49] G. F. Lyon. Remote OS detection via TCP/IP stack fingerprinting. *Phrack Magazine*, 8(54), 1998.

[50] G. F. Lyon. *Nmap network scanning: The official Nmap project guide to network discovery and security scanning.* Insecure, 2009.

[51] S. Maes, K. Tuyls, B. Vanschoenwinkel, and B. Manderick. Credit card fraud detection using Bayesian and neural networks. In *Proceedings of the 1st international naiso congress on neuro fuzzy technologies*, pages 261–270, 2002.

[52] J. McCarthy and E. A. Feigenbaum. In memoriam: Arthur samuel: Pioneer in machine learning. *AI Magazine*, 11(3):10–10, 1990.

[53] A. McGregor, M. Hall, P. Lorier, and J. Brunskill. Flow clustering using machine learning techniques. In *International workshop on passive and active network measurement*, pages 205–214. Springer, 2004.

[54] A. Medina, M. Allman, and S. Floyd. Measuring the evolution of transport protocols in the internet. *ACM SIGCOMM Computer Communication Review*, 35(2):37–52, 2005.

[55] T. M. Mitchell. Machine learning and data mining. *Communications of the ACM*, 42(11), 1999.

[56] A. W. Moore and D. Zuev. Internet traffic classification using bayesian analysis techniques. In *ACM SIGMETRICS Performance Evaluation Review*, volume 33, pages 50–60. ACM, 2005.

[57] Netresec. Networkminer. https://www.netresec.com/?page=NetworkMiner, 2007.

[58] T. T. Nguyen and G. J. Armitage. A survey of techniques for internet traffic classification using machine learning. *IEEE Communications Surveys and Tutorials*, 10(1-4):56–76, 2008.

[59] A. Ornaghi and M. Valleri. Ettercap. https://www.ettercap-project.org/, 2015.

[60] J. Pahdye and S. Floyd. On inferring TCP behavior. *ACM SIGCOMM Computer Comm. Review*, 31(4):287–298, 2001.

[61] M. Panda, H. L. Vu, M. Mandjes, and S. R. Pokhrel. Performance analysis of TCP NewReno over a cellular last-mile: Buffer and channel losses. *IEEE Transactions on Mobile Computing*, 14(8):1629–1643, 2014.

[62] P. Pantel, D. Lin, et al. Spamcop: A spam classification & organization program. In *Proceedings of AAAI-98 Workshop on Learning for Text Categorization*, pages 95–98, 1998.

[63] V. Paxson. Automated packet trace analysis of TCP implementations. *ACM SIGCOMM Computer Communication Review*, 27(4):167–179, 1997.

[64] V. Paxson. Bro: a system for detecting network intruders in real-time. *Computer networks*, 31(23-24):2435–2463, 1999.

[65] J. Postel. Internet control message protocol. RFC 792, 1981.

[66] J. Postel. Internet control message protocol. RFC 792, 1981.

[67] J. Postel. Transmission control protocol. RFC 793, 1981.

[68] F. Qian, A. Gerber, Z. M. Mao, S. Sen, O. Spatscheck, and W. Willinger. TCP revisited: a fresh look at TCP in the wild. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement*, pages 76–89. ACM, 2009.

[69] K. Ramakrishnan, S. Floyd, and D. Black. The addition of explicit congestion notification (ECN) to IP. RFC 3168, 2001.

[70] K. Ramakrishnan and R. Jain. A binary feedback scheme for congestion avoidance in computer networks. *ACM Transactions on Computer Systems (TOCS)*, 8(2):158–181, 1990.

[71] S. Ryu, C. Rump, and C. Qiao. Advances in internet congestion control. *IEEE Communications Surveys & Tutorials*, 5(1):28–39, 2003.

[72] F. Sebastiani. Machine learning in automated text categorization. *ACM computing surveys (CSUR)*, 34(1):1–47, 2002.

[73] M. M. Sebring. Expert systems in intrusion detection: A case study. In *Proc. 11th National Computer Security Conference, Baltimore, Maryland, Oct. 1988*, pages 74–81, 1988.

[74] R. Sommer and V. Paxson. Outside the closed world: On using machine learning for network intrusion detection. In *2010 IEEE symposium on security and privacy*, pages 305–316. IEEE, 2010.

[75] G. Taleck. Ambiguity resolution via passive OS fingerprinting. In *International Workshop on Recent Advances in Intrusion Detection*, pages 192–206. Springer, 2003.

[76] S. Tong and E. Chang. Support vector machine active learning for image retrieval. In *Proceedings of the ninth ACM international conference on Multimedia*, pages 107–118. ACM, 2001.

[77] W. Wei, K. Suh, B. Wang, Y. Gu, J. Kurose, and D. Towsley. Passive online rogue access point detection using sequential hypothesis testing with TCP ACK-pairs. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, pages 365–378. ACM, 2007.

[78] F. Yarochkin and O. Arkin. Xprobe2- A'Fuzzy'Approach to Remote Active Operating System Fingerprinting, 2002.

[79] M. Zalewski. p0f: Passive OS fingerprinting tool. *Online at http://lcamtuf.coredump.cx/p0f3*, 2017.

Chapter 2

# Summary of Included Papers and Contributions

In this chapter, we give a brief summary of the contributions of the included papers in the dissertation.

## 2.1 Introduction

The main contributions of this dissertation can be categorized under three cases that fall under the same umbrella of using machine learning to understand the complexity of TCP. Those contributions can be briefly summarized as follows.

- The first contribution is anchored within the field of of passive TCP traffic monitoring. We demonstrate an intermediate node can predict the internal states of the TCP client using reasonably effective state-of-the-art machine learning and deep learning techniques. Firstly, we use a generic machine learning-based prediction approach for inferring cwnd within a flow from passive traffic collected at an intermediate node basing our inference on the total number of outstanding bytes. Secondly, using the estimated cwnd as input feature with other TCP options, we extend our contribution to classify the underlying TCP variants for which the congestion signal is either loss or delay. We further validate that our Recurrent Neural Networks (RNN)-based model is applicable for Round-trip Time (RTT) prediction in relation to TCP from passive measurements collected at an intermediate node. We show that the learned prediction model performs reasonably well by leveraging trained knowledge from the emulated network. Furthermore, the trained classification models generalized well to realistic scenario settings which demonstrate the transfer learning capability of our models. These approaches are evaluated in detail across different scenarios and included in Paper I, Paper II, Paper IV, Paper V, and Paper VI.
- The second contribution falls in the area of network security and more precisely under machine learning-based intrusion detection systems (IDS). We introduce the use of standard regularized machine learning techniques to binary and multi-class attack classification of security attacks. Regularized machine learning techniques allows better interpretability of the complex decisions of the IDS. In our evaluation, we focus mainly on selecting the most important input features that are well understood within the networking community for identifying security attacks. We further provide deeper insight from a security engineering perspective on the importance

of those features identified by the regularized machine learning techniques. Our analysis is extensively performed on the benchmark intrusion detection dataset NSL-KDD [6]. The methodology of the evaluation and experimental results are briefly included in Paper VII.

- We advance the field of passive Operating System (OS) fingerprinting problem by introducing the underlying predicted TCP variant as a distinguishing feature in addition to the basic TCP/IP features that are the basis of OS fingerprinting. Our work is unique in that it employs the predicted TCP variant as a feature in two cases across different variations of OSes. Firstly, we use the default TCP flavor of the underlying OS. Secondly, we use the predicted TCP variant passively inferred from the famous sawtooth pattern behavior of TCP's estimated cwnd computed based on the outstanding bytes in flight. In terms of accuracy, we empirically demonstrate that accurately predicting the TCP variant has the potential to boost the evaluation performance across all our validation scenarios and different types of traffic sources. We further demonstrate the transferability approach of our passive OSes classification models by conducting a series of controlled experiments against other experimental scenarios. To the best of our knowledge, this is the first study that explores the potential for using the knowledge of the TCP variant to significantly boost the accuracy of passive OS fingerprinting. The employed techniques are evaluated in detail and presented in Paper VIII.

## 2.2 Included Papers Summary and Contributions

This dissertation consists of the following papers presented at peer-reviewed conferences and journals. The author of this dissertation has been the main responsible for the studies in the included papers where he is the first author. This includes proposing most of the research ideas, implementing the machine learning and deep learning models, setting-up and running the experiments across different scenarios, analyzing the experimental results and writing all the included papers in this dissertation. Figure 2.1 gives an illustration of the contribution by each paper grouped by the use cases, as well as showing the progression and quality of the research. Figure 2.2 shows the overall relationship of how the contribution of our papers is linked to each other.

**Paper I Summary:** Presents a general machine learning-based methodology to experimentally infer the internal Transmission Control Protocol (TCP) per-connection states of loss-based TCP flavors from passive measurements collected at an intermediate node when there is variability within a flow. By leveraging machine learning techniques, the paper demonstrates how an intermediate node (e.g., a network operator) can infer the transmission states of the TCP client associated with a TCP flow via passively monitoring the TCP traffic. More precisely, the paper demonstrates how the intermediate node can predict the Congestion Window (cwnd) size of the TCP client by examining each cross-traffic of TCP flows of the endpoints passively collected at an intermediate node.

**Paper I Contribution**: We explored machine learning-based techniques to monitor TCP per-connection states of loss-based TCP variants from passive measurements under varying network conditions. A study of interest that is most closely related to our work is [7] which provides a passive measurement methodology to infer and keep track of the values of the sender variables: end-to-end RTT and cwnd. The idea in [7] is to emulate a state transition by detecting Retransmission Timeout (RTO) events at the sender and observing the ACKs which cause the sender to change the value of the cwnd. The authors consider only the predominant implementations of TCP and the basic idea is to construct a replica of the TCP sender's state for each TCP connection observed at the intermediate node using 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 cannot manage to reverse or backtrack the transitions taking the tremendous amount of data into consideration. We argue that another limitation of this work is the fact 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. The work in [8] presents a methodology to study the performance of TCP, classify out-of-sequence behavior of packets for retransmission to identify where congestion is occurring in the network, with the same measurement environment as in [7]. The authors of the study presented in [17] have developed a tool called tcpflows that attempts to passively estimate the value of cwnd 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.

Our main goal in Paper I is to develop and evaluate a machine learning-based methodology for passively predicting cwnd of all loss-based TCP variants of the client by examining each cross-traffic of TCP flows of the endpoints. Our experimental results yield very good accuracy for both the increasing and decreasing portions of the sawtooth pattern across various validation scenarios and different loss-based TCP flavors.

**Paper II Summary:** Proposes a robust, scalable and generic machine learning-based model that experimentally infers both the cwnd and the underlying variants of loss-based TCP congestion control algorithms within flow from passive traffic measurements collected at an intermediate node. The significance of this paper is *two-fold*. By extending our work presented in Paper I, this paper presents a machine learning model for predicting the widely deployed underlying loss-based TCP variants within a flow. The scalability and robustness approach of the prediction model is validated across multiple controlled scenario settings. It turns out, surprisingly enough, that the learned prediction model performs reasonably well by leveraging knowledge from the emulated network when it is applied in a real-life scenario setting which bears similarity to the concept of transfer learning in the machine learning and deep learning communities.

**Paper II Contribution**: We passively control the TCP flows individually by uniquely identifying the underlying TCP variants of the clients. However, predicting TCP transmission states from passive measurement has a number of practical difficulties. One of the main challenges is, for example, TCP packets can be lost between the sender and the intermediate monitor, or between the monitor and the receiver [7, 8]. 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 RTO rather than a fast retransmit [8]. 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, so the receiver did not yet know that the packets were received before they decided to retransmit them.

In this paper, we advocate that machine learning-based approaches can give a better prediction accuracy of TCP sender connection states from passive measurements collected at an intermediate node. Hence, to address the aforementioned practical challenges, we present a robust and scalable machine learning-based methodology that passively estimates the cwnd and uniquely identifies the widely deployed underlying loss-based TCP variants that the client is using. We validate the robustness and scalability approach of our prediction model extensively across an emulated, realistic and combined scenario setting. The prediction accuracies of these scenario settings are 93.51%, 95%, and 91.66% respectively. 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 further show that our prediction model learned from emulated data generalizes to realistic scenarios bearing similarity to the concept of transfer learning in the machine learning community.

**Paper III** is a journal extension of Paper I and Paper II. Hence, this journal paper is not included as part of this dissertation to avoid redundancy.

**Paper IV Summary:** Explore the capability of Long Short-Term Memory (LSTM)-based RNN model to predict TCP transmission states performed by analyzing data collected through passive measurements. To the best of our knowledge, this paper is the first work that attempts to apply

LSTM for demonstrating how to identify the most important system-wide transmission states of a TCP client from passive traffic measured at an intermediate node of the network without having access to the sender. The main goal in this paper is to implement a learning predictive model that generates the pattern of cwnd from passive measurements using an LSTM architecture and finally justify if the previous machine learning-based experiments presented in Paper I and Paper II are valid. In addition to capturing the pattern of a TCP cwnd with small prediction errors, our LSTM-based prediction model is also applied to uniquely identify the underlying loss-based TCP variants based on the multiplicative decrease parameter of the cwnd and the per-connection states within the variant from passive measurements. The experimental results presented in this paper based on emulated and realistic settings show that the LSTM-based model outperforms the previous results presented in Paper I and Paper II by a reasonably significant margin. It shows that the learned prediction model by leveraging knowledge from the emulated network performs reasonably well when it is applied in a realistic scenario setting bearing similarity to the concept of transfer learning in the machine learning community. Through an extensive experimental evaluation on multiple scenarios, this paper demonstrates the scalability and robustness of the approach and its potential for monitoring TCP transmission states related to network congestion from passive measurements.

**Paper IV Contribution**: As in the previous two papers where we advocate the use of machine learning techniques in the context of TCP passive monitoring, in this paper, we are interested in the potential of RNN model based on emulated and realistic networks for estimating TCP cwnd as well as the underlying TCP variants within a flow. Hence, we have explored an LSTM architecture for RNN-based prediction approaches to monitor the most important TCP per-connection states from passive measurements related to network congestion. We demonstrate the capability of a deep neural network architecture based on a state-of-the-art learning LSTM recurrent predictive models to predict TCP transmission states by merely analyzing data collected through pure passive measurements from intermediate nodes. Our main goal in this work is to implement a learning predictive model that generates the pattern of cwnd from passive measurements using an LSTM architecture and finally justify if our previous machine learning based-based experiments to estimate cwnd and predict the underlying loss-based TCP variants presented in Paper I and Paper II are valid. We found out that our LSTM-based model outperforms our previous work presented in Paper II by a reasonably significant margin. The LSTM-based TCP variant prediction model achieves accuracies of 97.22%, 96.66% and 94.44% on the emulated, realistic and combined scenario settings, outperforming the standard machine learning-based which yields accuracies of 93.51%, 95% and 91.66% respectively.

**Paper V Summary:** Proposes and evaluates a novel deep learning-based model capable of dynamically predicting at real-time the RTT between the sender and receiver with high accuracy using TCP timestamps. This paper aims at improving the accuracy and timeliness of the RTT estimation by employing state-of-the-art LSTM-based prediction models to help network operators improving their analysis. It explores in detail a set of practical methodological challenges and considerations involved in performing inference of RTT reliably from passive measurements. Our prediction methodology is extensively validated across a controlled experimental testbed and in a realistic scenario on the Google Cloud platform using different TCP flavors and also dynamic changes in RTT. Hence, the primary contribution of the work presented in this paper is building an RTT prediction model that works well for transfer learning. Even though the RTT prediction model was trained on an emulated network, this paper demonstrates that it performs well also when applied to a realistic scenario setting by leveraging knowledge from the emulated network.

**Paper V Contribution**: We present a dynamic deep learning-based approach for RTT prediction in relation to TCP from passive traffic measurements. There are other previous research works who have examined and reported RTT estimation for TCP [1, 7, 9]. The approach presented in [9], for example, uses a unidirectional flow during the TCP handshake of a connection to estimate RTT using the time from SYN to SYN+ACK method. The estimation method proposed in [9] calculates one RTT sample per TCP connection associated either during the three-way handshake or during the slow-start phase. If we have captured the TCP three-way handshake as presented in [9], we can calculate the initial RTT (iRTT) by taking the time difference from the SYN packet to the ACK packet of the handshake. However, since the TCP handshake packets are processed by the kernel, the RTTs during the data transfer will probably be slightly larger than the iRTT. Hence, this approach may tend to underestimate the actual RTT. In addition to this, since TCP sets the initial retransmission timeout value to 3 seconds [15], this approach is not applicable in scenarios where the TCP connection setup takes longer which leads to long delays and packet losses introduced by the network.

Our deep learning-based approach to passively predict the continuous RTT measurement throughout the lifetime of a TCP session builds upon these classical approaches by taking advantage of the commonly used timestamps option. The main contribution of our work is building an LSTM-based RTT prediction model that works well for transfer learning from passive traffic measurements under emulated, realistic scenarios settings and different TCP variant configurations. Hence, we demonstrate that the learned prediction model performs reasonably well by leveraging trained knowledge from the emulated network when it is applied and transferred in a real-life scenario setting.

**Paper VI Summary:** Investigates and explores in greater detail on how an intermediate node can identify the transmission characteristics state of widely used delay-based TCP congestion control algorithms that exploit queueing delay as a congestion signal associated with a passively monitored TCP traffic. This paper presents an effective TCP variant identification methodology from traffic measured passively by utilizing $\beta$, the multiplicative back-off factor to decrease the cwnd on a loss event, and the queueing delay values. The paper addresses how $\beta$ varies as a function of queueing delay and how the TCP variants of delay-based congestion control algorithms can be predicted both from passively measured traffic and real measurements over the Internet.

Paper VI further employs a novel non-stationary time series approach from a stochastic nonparametric perspective using a two-sided Kolmogorov–Smirnov test to classify delay-based TCP algorithms based on the $\alpha$, the rate at which a TCP sender's side cwnd grows per window of acknowledged packets, parameter. To the best of our knowledge, this paper is the first to study how the variability of the $\beta$ parameter as a function of queueing delay and the $\alpha$ parameter can be used for passive delay-based TCP variant identification in real-time. Through extensive experiments on emulated and realistic scenarios, this paper demonstrates that the data-driven classification techniques based on probabilistic models and Bayesian inference for optimal identification of the underlying delay-based TCP congestion algorithms give promising and comparable results in terms of accuracy. It shows that the methods can also be applied equally well to loss-based TCP variants. As observed in Paper I, Paper II, Paper IV, and Paper V, this paper also shows that the learned prediction model performs reasonably well by leveraging trained knowledge from the emulated network when it is applied and transferred on a realistic scenario setting.

**Paper VI Contribution**: Identifying the underlying TCP variant from passive measurements is important for several reasons, e.g., exploring security ramifications, traffic engineering in the Internet, etc. There are many different TCP variants widely in use, and each variant uses a specific end-to-end congestion control algorithm to avoid congestion, while also attempting to share the underlying network capacity equally among the competing users. However, we believe that there is very little work on the identification of the underlying delay-based TCP congestion control algorithms from passive measurements. The work in [14] proposes a cluster analysis-based method that aims a router to identify between two versions of 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 [14]. These characteristics are then clustered into two groups by applying a hierarchical clustering technique. The authors show that only 2 out of 14 TCP congestion algorithms that are implemented in Linux can be identified based on their method [14]. Another related work [20] presents an active measurement technique to identify a diverse set of known congestion control algorithms. However, our work in this paper relies on a passive measurement technique and we are interested in investigating the delay characteristics of widely used TCP algorithms that exploit queueing delay as a congestion signal.

In Paper II, we presented a machine learning-based approach to identify the underlying traditional loss-based TCP variants which achieve a reasonably good accuracy on emulated and realistic scenarios. In this paper, we present an effective delay-based TCP variant identification methodology with both high and low queueing delay based on probabilistic models and Bayesian inference techniques. The prediction accuracies for the selected delay-based TCP variants on emulated and realistic scenarios with high and low queueing delay cases are 97.5%, 95%, 97.83%, and 95.46% respectively

**Paper VII Summary:** Introduces the use of standard regularized machine learning techniques for enhancing computer network security attack analysis of an effective benchmark intrusion detection dataset. It focuses mainly on the contribution of the actual input features that are well understood within the networking community to find what kinds of attacks in a network are the most significant. To that end, the actual input features studied in this paper are ranked into strongly contributing, low contributory and irrelevant using a combination of feature selection filters and wrapper methods by carefully carrying out comparisons with previous techniques. This paper adopts two well-known ranking distance measure metrics in the evaluation of how similar the ranking algorithms presented in the paper are in relative to other state-of-the-art methods.

Paper VII investigates the most important features in identifying well-known security attacks by using Support Vector Machines (SVMs) wrapped with Recursive Feature Elimination (RFE) algorithm and $\ell_1$-regularized method with Least Absolute Shrinkage and Selection Operator (LASSO) for robust regression both to binary and multiclass attack classification to give us an insight into features of different classes of security attacks. SVMs are one of the standards of machine learning classification techniques that give a reasonably good performance but with some drawbacks in terms of interpretability. On the other hand, LASSO is a regularized regression method often performing comparably well and it has extra compelling advantages of being very easily interpretable. Moreover, LASSO is much more computationally effective and provides coefficients that assess how individual features affect the probability of specific security attack classes. Hence, this paper uses LASSO in particular for multiclass classification in order to better understand the dynamics behind the classification model and get a better insight into which actual features shared by security attacks in a network are the most important ones for detecting and distinguishing between security attacks.

The analysis in Paper VII is performed using a benchmark intrusion detection public dataset where the data are labeled into either anomalous (denial-of-service (DoS), remote-to-local (R2L), user-to-root (U2R), and probe attack classes) or normal. Extensive experiments are performed where we compared feature ranking algorithms using both two-stage approaches with SVM and one-stage approach using LASSO. Total accuracy of 97% for binary classification is achieved for both the two-stage evaluation approach using SVM and a one-stage approach using LASSO. Compared to binary, a total multiclass classification accuracy of 95.90% is achieved using LASSO. This paper concludes that a one-stage approach using LASSO is simpler, computationally faster and gives us good performance with the most significant actual features.

**Paper VII Contribution**: The process of defining appropriate input features, performing feature selection, data normalization and the contribution of this with interpretable results on security attack classification and computational performance has not been thoroughly studied. Traditional approaches rely on expert knowledge or managers to define rule-sets defining normal behavior and intrusions in a network [16, 19]. Machine learning techniques have the potential of detecting unknown security attacks in network traffic sharing features with other attacks by being trained on normal and abnormal types of traffic. However, one critical problem in machine learning is identifying and selecting the most relevant input features to construct an accurate model based on training data for a particular classification task. As it is reported in the literature [4, 11, 18], employing machine learning techniques on the NSL-KDD [6] dataset gives a very low level of detection rate on some security attack categories within the misuse detection context. It is, therefore, important to perform an effective feature selection analysis to make it easier for network administrators to better understand the features that contribute to attacks. Finally, this paper provides a deeper insight from a security engineering perspective on why the features obtained by regularized machine learning techniques are so important in clearly identifying various security attacks in a network. We argue that the presented methodology may strengthen future research work in network intrusion detection settings by leveraging advanced state-of-the-art machine learning approaches.

**Paper VIII Summary:** Proposes and evaluates an advanced classification approach to passive OS fingerprinting by leveraging state-of-the-art classical machine learning and deep learning techniques. Our controlled experiments on benchmark data, emulated and realistic traffic is performed using two approaches. Through an Oracle-based machine learning approach, we found that the underlying TCP variant is an important feature for predicting the remote OS. Based on this observation, we develop a sophisticated tool for OS fingerprinting that first predicts the TCP flavor using passive traffic traces and then uses this prediction as an input feature for another machine learning algorithm for predicting the remote OS from passive measurements.

Paper VIII takes the passive fingerprinting problem one step further by introducing the underlying predicted TCP variant as a distinguishing feature in addition to the basic TCP/IP features that are the basis of OS fingerprinting. Using the TCP variant as a passive OS distinguishing feature remains largely unexplored and is not used by existing fingerprinting techniques. The reason why we concentrate on the implementations of the underlying TCP variant as a feature in our OSes classifier models is that due to the fact that different Operating Systems (OSes) are doing slightly different implementations of TCP. Hence, we strongly believe that passively observing the network-level characteristics found in TCP packets can give us more information about the remote computer's underlying OS. We further believe that this will also help us to explore in detail the practical implications and long-term characteristics of TCP traffic. This is one of the main contributions of Paper VIII. In terms of accuracy, we empirically demonstrate that accurately predicting the TCP variant has the potential to boosts the evaluation performance from 84.1% to 94.1% on average across all traffic types tested, and from 85.6% to 95.4% in an emulated setting. We also demonstrate a practical example of this potential, by increasing the performance to 91.3% on average using a tool for passive TCP variant prediction in an emulated setting. To the best of our knowledge, this is the first study that explores the potential of using the knowledge about the underlying TCP variant to significantly boost the accuracy of passive OS fingerprinting.

In this paper, we show that the presented machine learning and deep learning-based classification models for passive OS fingerprinting approaches perform consistently and reasonably well as compared to other existing state-of-the-art solutions. Furthermore, we also show that the OSes classification model works equally as well for transfer learning. In all our experiments, we made sure that both the training and validation accuracies are closer as a way of measuring the ability of the classification models to generalize on unforeseen scenarios.

Finally, Paper VIII highlights questions and technical challenges of interest to direct future research, such as: *what happens if an end-user (client) changes parameters that are the basis of passive OS fingerprinting*?, *What happens if we don't know the underlying OS?* We believe challenges like this would make OS fingerprinting from passive measurements potentially hard. Hence, investigating these key challenges is one possibility for our future work. It is known that TCP clock drift improves OS fingerprinting and hence measuring differences in the timing of how the IP stack works may allow us to predict the underlying OS with greater assurance in terms of accuracy. We, therefore, argue using other TCP options like Timestamps and queueing delay characteristics as an input feature vector for passive OSes fingerprinting model is also an interesting direction. Hence, as part of our future work, we plan to include these features and strengthen the research in OS fingerprinting from passive measurements. The method presented in this paper, where the TCP cwnd is first computed based on the outstanding bytes in flight, then the underlying TCP flavor is predicted from the estimated cwnd, is particularly efficient for loss-based TCP variants. In previous works, we have also developed a tool for the prediction of delay-based TCP flavors [5]. As future work, we plan to extend the method presented here to also cover delay-based TCP variants.

**Paper VIII Contribution**: An accurate passive OS fingerprinting plays a critical role in effective network management and cybersecurity protection. Traditionally, most of the existing general OS fingerprinting techniques resort to manually generated signature matching from a database of heuristics which contains features of widely used OSes. This means, after comparing the generated signatures, the first set of responses match with the highest confidence against a database of fingerprints would be used to select the specific probable OS. However, manually updating a large number of signature and managing databases of new OSes adds a considerable amount of time and hence we may suffer from the consequences of the lack of recent signature updates of the known OSes. Hence, we argue that it is important to consider making use of a fingerprint database that contains variations of most currently used OSes and automating these tasks by employing learning algorithms capable of extracting all possible OS-specific features for discovering the underlying OSes. To explore this idea of applying learning algorithms, we present a unified and robust classification approach to an advanced passive OS fingerprinting that leverages both machine learning and deep learning methods.

In the computer security community, there has been a great deal of work on remote OSes fingerprinting [2, 10, 12, 13]. A recent study that is most closely related to our work, and which has also given a comprehensive survey on passive fingerprinting methods, can be found in [10]. The average accuracy of OS classification using the TCP/IP parameters reported in [10] is 80.88%. Aksoy et al. [2] have employed genetic algorithms for identifying packet features suitable for OS classification based on the analysis of the network TCP/IP packets using machine learning algorithms. However, most of these previous works use the basic actual TCP/IP features for evaluating passive OS fingerprinting. Besides, we believe that these tools have the inability to extract all possible OS-specific features for passively fingerprinting the underlying OSes.

In contrast, what distinguishes our contribution in Paper VIII from the other previous related works is that our model supports a wider range of TCP/IP network stack features. The central goal of our work presented here is to combine these basic TCP/IP features that are the basis of

OS fingerprinting with the underlying TCP variant by leveraging both machine learning and deep learning techniques. This idea remains largely unexplored and is not used by existing fingerprinting techniques. Detecting the implementation of a TCP variant passively is a challenging task and this, we believe, is the reason why no previous works use it to passively fingerprint remote OSes. However, in our case, we already have a general solution for this difficulty presented in Paper I, Paper II, and Paper IV. The reason why we focus on the implementations of the underlying TCP variant as a feature in our OS classifier model is due to the fact that different OSes are doing slightly different implementations of TCP. We believe that passively observing the network-level characteristics found in TCP packets can give us more information about the remote computer's underlying OS. We further believe that this will also help us to explore in detail the long-term characteristics of TCP traffic. Hence, we propose and evaluate a novel approach that attempts to passively fingerprint the underlying remote OS by leveraging state-of-the-art machine learning and deep learning techniques.

In Paper VIII, we show that knowing the TCP variant has a great potential for boosting the classification performance of passive OS fingerprinting. However, in reality, we don't have an Oracle-given TCP variant and hence we don't know what exactly the underlying TCP flavor is. In this paper, we built a universal tool for passive monitoring that can be applied to first passively estimate the TCP cwnd computed based on the outstanding bytes in flight, second passively predict the underlying TCP flavor from the estimated cwnd and finally uses the predicted TCP variant as an input feature to detect the remote computer's OS in addition to the basic TCP/IP features that are the basis of OS fingerprinting. We demonstrate that our tool with the TCP variant prediction performs equally as well when compared to the Oracle-given TCP variant. The experimental results show that our classification models for passive OS fingerprinting perform highly consistently and reasonably well in terms of accuracy across different validation scenarios. To the best of our knowledge, this is the first study of passive fingerprinting OSes by applying machine learning and deep learning approaches combining the basic TCP/IP features and the predicted underlying TCP variant as input vectors.

**Paper IX** is a journal extension of Paper VIII and hence, this journal paper is not included as part of this dissertation to avoid redundancy for the reader.

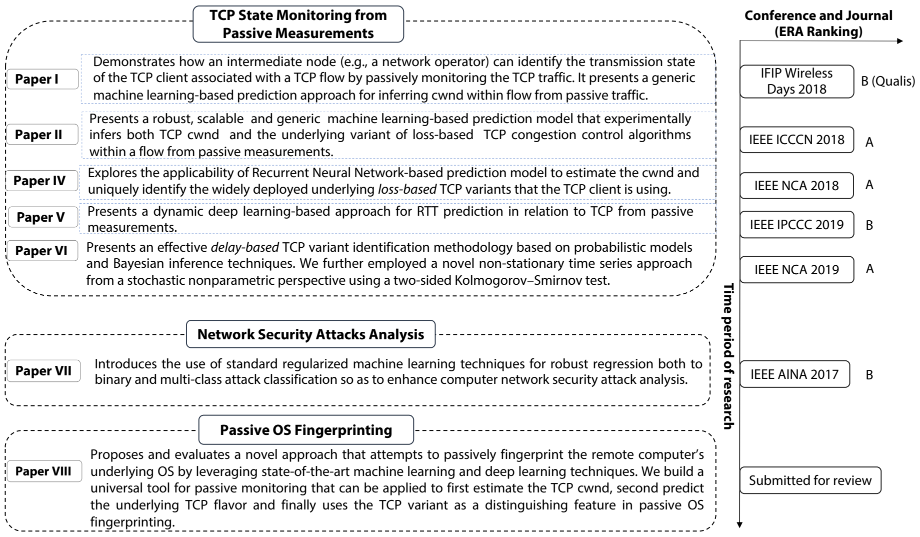

TCP State Monitoring from Passive Measurements
Paper I
Demonstrates how an intermediate node (e.g., a network operator) can identify the transmission state of the TCP client associated with a TCP flow by passively monitoring the TCP traffic. It presents a generic machine learning-based prediction approach for inferring cwnd within flow from passive traffic.
Paper II
Presents 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.
Paper IV
Explores the applicability of Recurrent Neural Network-based prediction model to estimate the cwnd and uniquely identify the widely deployed underlying *loss-based* TCP variants that the TCP client is using.
Paper V
Presents a dynamic deep learning-based approach for RTT prediction in relation to TCP from passive measurements.
Paper VI
Presents an effective *delay-based* TCP variant identification methodology based on probabilistic models and Bayesian inference techniques. We further employed a novel non-stationary time series approach from a stochastic nonparametric perspective using a two-sided Kolmogorov–Smirnov test.
Network Security Attacks Analysis
Paper VII
Introduces the use of standard regularized machine learning techniques for robust regression both to binary and multi-class attack classification so as to enhance computer network security attack analysis.
Passive OS Fingerprinting
Paper VIII
Proposes and evaluates a novel approach that attempts to passively fingerprint the remote computer's underlying OS by leveraging state-of-the-art machine learning and deep learning techniques. We build a universal tool for passive monitoring that can be applied to first estimate the TCP cwnd, second predict the underlying TCP flavor and finally uses the TCP variant as a distinguishing feature in passive OS fingerprinting.
Time period of research
Conference and Journal (ERA Ranking)
IFIP Wireless Days 2018
B (Qualis)
IEEE ICCCN 2018
A
IEEE NCA 2018
A
IEEE IPCCC 2019
B
IEEE NCA 2019
A
IEEE AINA 2017
B
Submitted for review

Figure 2.1: Overview of contributions in this Ph.D. dissertation. Conferences and journals with a ranking according to ERA [3] are given on the right side.

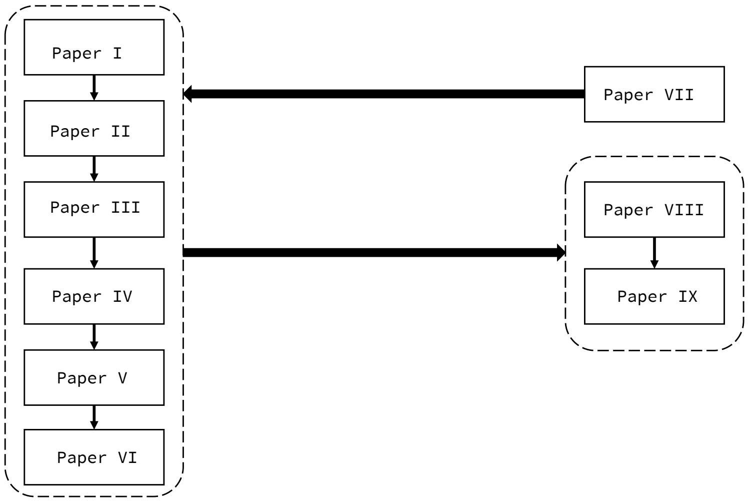


Figure 2.2: The overall relationship of how the contribution of each of our papers is linked to each other. For example, the contributions of Paper I are inputs to Paper II. Paper III is an extension of the contributions presented in Paper I and Paper II, etc. Combining all the contributions from Paper I - Paper VII are inputs to Paper VIII and Paper IX.

## 2.3 References

[1] J. Aikat, J. Kaur, F. D. Smith, and K. Jeffay. Variability in TCP round-trip times. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, pages 279–284. ACM, 2003.

[2] A. Aksoy, S. Louis, and M. H. Gunes. Operating system fingerprinting via automated network traffic analysis. In *2017 IEEE Congress on Evolutionary Computation (CEC)*, pages 2502–2509. IEEE, 2017.

[3] CORE. The ERA Conference. http://portal.core.edu.au/conf-ranks/, 2013.

[4] C. Elkan. Results of the KDD'99 classifier learning contest. In *Sponsored by the International Conference on Knowledge Discovery in Databases*, 1999.

[5] D. H. Hagos, P. E. Engelstad, and A. Yazidi. Classification of Delay-based TCP Algorithms From Passive Traffic Measurements. In *2019 IEEE 18th International Symposium on Network Computing and Applications (NCA)*. IEEE, 2019.

[6] ISCX. NSL-KDD Dataset. http://www.unb.ca/research/iscx/dataset/iscx-NSL-KDD-dataset.html, 2009.

[7] S. Jaiswal, G. Iannaccone, C. Diot, J. Kurose, and D. Towsley. Inferring tcp connection characteristics through passive measurements. In *IEEE INFOCOM 2004*, volume 3, pages 1582–1592. IEEE, 2004.

[8] 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 Transactions on Networking (ToN)*, 15(1):54–66, 2007.

[9] H. Jiang and C. Dovrolis. Passive estimation of TCP round-trip times. *ACM SIGCOMM Computer Communication Review*, 32(3):75–88, 2002.

[10] M. Lastovicka, T. Jirsik, P. Celeda, S. Spacek, and D. Filakovsky. Passive OS Fingerprinting Methods in the Jungle of Wireless Networks. In *NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium*, pages 1–9. IEEE, 2018.

[11] I. Levin. KDD-99 classifier learning contest: LLSoft's results overview. *SIGKDD explorations*, 1(2):67–75, 2000.

[12] R. Lippmann, D. Fried, K. Piwowarski, and W. Streilein. Passive operating system identification from TCP/IP packet headers. In *Data Mining for Computer Security*. Citeseer, 2003.

[13] G. F. Lyon. *Nmap network scanning: The official Nmap project guide to network discovery and security scanning*. Insecure, 2009.

[14] J. Oshio, S. Ata, and I. Oka. Identification of different TCP versions based on cluster analysis. In *2009 Proceedings of 18th International Conference on Computer Communications and Networks*, pages 1–6. IEEE, 2009.

[15] C. Paxson, M. Allman, J. Chu, and M. Sargent. Computing TCP's Retransmission Timer (RFC 6298), 2011.

[16] V. Paxson. Bro: a system for detecting network intruders in real-time. *Computer networks*, 31(23-24):2435–2463, 1999.

[17] S. Rewaskar, J. Kaur, and D. Smith. A Passive State-Machine Based Approach for Reliable Estimation of TCP Losses. 2006.

[18] M. Sabhnani and G. Serpen. Why machine learning algorithms fail in misuse detection on KDD intrusion detection data set. *Intelligent Data Analysis*, 8(4):403–415, 2004.

[19] M. M. Sebring. Expert systems in intrusion detection: A case study. In *Proc. 11th National Computer Security Conference, Baltimore, Maryland, Oct. 1988*, pages 74–81, 1988.

[20] P. Yang, J. Shao, W. Luo, L. Xu, J. Deogun, and Y. Lu. TCP congestion avoidance algorithm identification. *IEEE/Acm Transactions On Networking*, 22(4):1311–1324, 2014.

Chapter 3

# Background

In this chapter, we provide necessary background knowledge relevant to the reader of this doctoral dissertation in order to make the dissertation report self-contained.

## 3.1 Machine Learning

Machine learning, as a fundamental branch of Artificial Intelligence (AI), is a programming approach that endows computers with the ability to learn and improve from their past experiences driven by historical data and take a decision that is not explicitly programmed [54, 75]. Historically, machine learning is defined as a collection of powerful algorithms for data mining and knowledge discovery technologies useful for automated identification of patterns in large datasets [58, 63]. Machine learning is increasingly becoming so promising in wide ranges of active research areas and practical applications and services, ranging from smartphones and cameras, speech recognition, image recognition, etc., that are important in all aspects of modern society. In this dissertation, we focus on the state-of-the-art applicability of both machine learning and deep learning techniques from the perspective of computer networking.

**Categories of machine learning**: In practice, there are *two* broad categories of most widely used forms of machine learning [60]. Note that there is a third category of an active research area in the machine learning community known as reinforcement learning [79]. It is a useful computational approach to learning where an agent tries to learn how to act or behave by maximizing the total amount of occasional reward signal or punishment while interactive with a dynamic environment. However, reinforcement learning is beyond the scope of this dissertation. For more detailed information on reinforcement learning, we refer the interested readers to [41, 70, 78, 81, 91].

### 3.1.1 Supervised Learning

Supervised learning, sometimes called a predictive learning, is a powerful machine learning approach where the main task is learning a function, $f : x \to y$, that finds an underlying mapping from input space $x$ to output space $y$, given a labeled set of $n$ input-output pairs $\mathcal{D} = \{(x_1, y_1), \ldots, (x_n, y_n)\}$, $x_i \in \mathbb{R}^n, y_i \in \mathbb{R}$ where $\mathcal{D}$ is called the training set and $n$ is the number of training examples [60, 70]. Each training input space $x_i$ is a $D$-dimensional vector of numbers, representing, for example, the length and width of a rectangle. These characteristics are called features or attributes. Note that $x_i$ is a vector and $y_i$ is a discrete label in classification and real values in regression [60, 70]. As show in Figure 3.2,

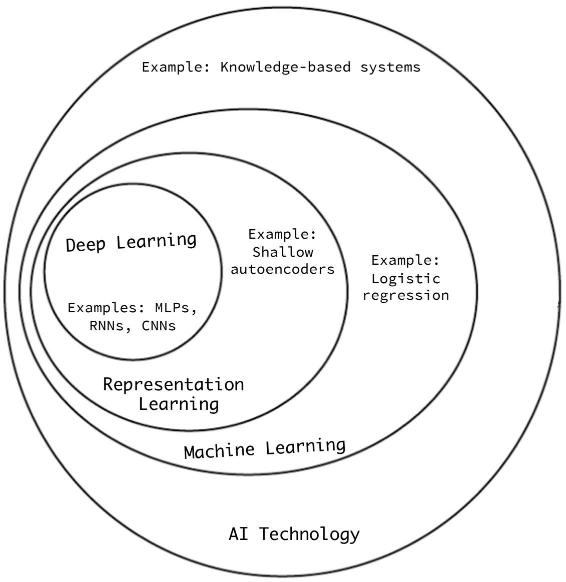

Example: Knowledge-based systems
Deep Learning
Examples: MLPs, RNNs, CNNs
Example: Shallow autoencoders
Example: Logistic regression
Representation Learning
Machine Learning
AI Technology

Figure 3.1: A Venn diagram showing how deep learning is a subfield of machine learning paradigm. Each section of the Venn diagram includes an example of an AI technology and it is inspired by [28].

the supervised learning model infers a function from labeled training data (e.g., image, document, text, time series data, etc.) with features consisting of a set of training examples [60].

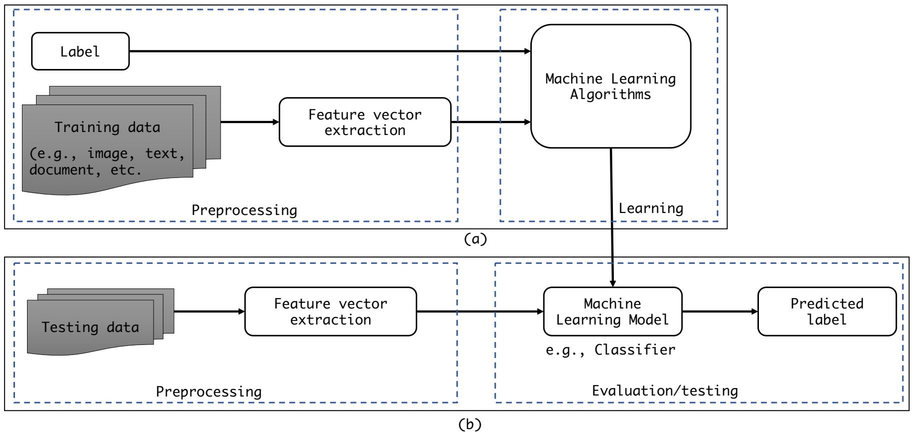


Figure 3.2: The supervised learning model process. (a) Training phase (b) Testing phase.

**Forms of supervised learning**: The two most common forms of supervised learning are: *classification* and *regression*.

#### 3.1.1.1 Classification

The main goal of classification task is to construct a function that maps from an input $x$ to a corresponding output $y$, i.e., $\{(x_i, y_i)\}_{i=1}^{n}$ where $y_i \in \{1, \ldots, n\}$, with $n$ being the number of different classes. There are two broad categories of classification: binary and multiclass classification. Binary classification is when $N = 2$, where $y_i \in \{+1, -1\}$. However, if $N > 2$, it is called multiclass classification. In general, if the possible set of output, $f : \mathcal{X} \rightarrow \mathbb{R}$, where $\mathcal{X}$ represents features of the underlying problem takes a finite set of discrete labels or categories (e.g., “yes” or “no”, “male” or “female”, image classification, handwriting recognition, facial detection and recognition, etc.), this indicates that the learning problem is a classification or pattern recognition. The supervised machine earning classification algorithms we employed in our papers are Support Vector Machines (SVMs) [10, 87], Naïve Bayes [20, 51], Random Forest [9], k-nearest neighbors (KNN) [19]. These are also some of the most popular supervised machine learning in use today.

#### 3.1.1.2 Regression

Unlike classification, if the possible set of response variables, $y_i$, takes continuous real values (e.g., predicting housing price, predict tomorrow's temperature, predict an employee's income, etc.), this indicates that the learning problem is a regression given as $\{(x_i, y_i)\}_{i=1}^{n}, x_i \in \mathcal{X}, y_i \in \mathbb{R}$. Regression approaches can be used to extract the underlying relationship between independent and dependent variables and identifying causal inference [38]. Some of the most common regression techniques include: linear regression [61, 73], logistic regression [43, 55], Least Absolute Shrinkage and Selection Operator (LASSO) regression [84, 85], etc. Among these, we have employed LASSO regression in our work.

### 3.1.2 Unsupervised Learning

Unsupervised learning is the second main category of machine learning whose main goal is to discover a particular hidden pattern in unlabeled data [35, 60]. This helps us to unveil the meaningful hidden relationships between the variables. In unsupervised learning we only have a training input without pre-existing labels, $\mathcal{D} = \{x_i\}_{i=1}^{n}$, where we observe only the features $x_1$, $x_2$, . . . . , $x_n$ without an associated response variable $y_i$ for a given $x$. Unlike supervised learning where we have the desired response variable provided to the learning model together with the input training data, with unsupervised learning we learn how to artificially reconstruct the natural structure of the input data using a representation [35]. In unsupervised learning, since we don't know the desired response variable, we formalize the learning task as unconditional density estimation by building learning models of the form $p(\mathbf{x}_i|\boldsymbol{\theta})$ [7, 62].

Unsupervised learning in recent years is becoming more widely applicable in a number of research fields since it does not require a human expert's knowledge about the data's attributes ahead of time [62]. One benefit of unsupervised learning is that it is often much easier to obtain unlabeled data than labeled data which can require additional human intervention to provide the correct labels. In addition to this, Bengio et al. has found out that unsupervised pre-training improves state-of-the-art learning algorithms for deep architectures such as Deep Belief Networks [22].

## 3.2 Deep Learning

Deep learning is an emerging and promising subfield of representation learning paradigm [4, 28]. Representation learning is a set of advanced techniques that seek to automatically discover the representations of large amounts of raw data fed into a machine for performing actions such as prediction, classification tasks or learning complex functions and relationships among data at multiple levels of abstraction [3, 48]. It is a new field in the machine learning community as depicted in Figure 3.1. The ability to automatically learn the depth and complexity of the large data representation at multiple levels is important as the amount of data and wide range of state-of-the-art applications to machine learning and deep learning techniques such as sound and speech recognition, object recognition, medical analysis, drug discovery, etc. continues to grow exponentially. For a more detailed explanation and historical background of representation learning, we refer the readers to the work presented in [28].

Deep learning methods are characterized by a collection of computational neural network models that are composed of multiple processing layers capable of learning distributed representations of data with multiple levels of abstraction [48]. Deep learning builds predictive models using large Artificial Neural Networks (ANN) as underlying techniques [4, 28]. These models have contributed remarkably well in advancing many research domains. Deep Neural Networks (DNN) [48, 72] are a feed-forward deep learning neural network architecture trained end-to-end using new machine learning methods that have shown advancements in a wide range of supervised and unsupervised machine intelligence tasks. Note that the computational effort of training the multiple processing layers of a fully connected recurrent network and finding the correct combination of weights from layer to layer and the parameters that change the input data becomes enormous and substantially harder when more complicated neural networks as described below are considered [69, 92]. To address this critical challenge, backpropagation [47, 49, 68, 88, 89, 90] as a learning technique to repeatedly update the change in weights in the neural network by comparing the networks actual output against the desired value in terms of the corresponding partial derivatives of the complex performance function computed using the chain rule with respect to a particular weight is introduced as of the form given in Equations 3.1 and 3.2.

$$\frac{\partial L(z,y)}{\partial w} = \frac{\partial L(z,y)}{\partial a} \times \frac{\partial a}{\partial z} \times \frac{\partial z}{\partial w} \tag{3.1}$$

where $L$, $z$, $y$, $w$, and $\alpha$ in Equation 3.2 denote the loss function, expected output, actual output, weight of the neural network, and the learning rate constant respectively. As a result, the weight of the neural network is updated as follows:

$$w \longleftarrow w - \alpha \frac{\partial L(z,y)}{\partial w} \tag{3.2}$$

In a neural network, weights of the neuron are simultaneously updated according to the following step by step procedures.

- Take a batch of training data.
- Perform forward propagation to obtain the corresponding loss.
- Backpropagate the loss to get the gradients.
- Use the gradients to update the weights of the network.

### 3.2.1 Recurrent Neural Network (RNNs)

As described above, the information in feed-forward neural networks moves in only one direction, i.e., from the input nodes towards the output nodes. One of the main benefits of the introduction of backpropagation is for efficiently and recursively training more powerful network models than feed-forward neural networks such as RNNs [48]. The backpropagation technique is capable of repeatedly updating the weights of the layers in the neural networks in order to minimize the measure of the difference between the actual and expected output vectors for a particular training example [47, 49, 68, 69, 88]. RNNs are widely applicable for tasks that involve the processing of sequential inputs such as speech recognition, computational biology, natural language processing (e.g., handwriting recognition) and other complex tasks. Unlike feed-forward neural networks, the connections between units in RNNs form a directed loop along a temporal sequence that allows the RNNs process an input sequence one element at a time using their internal state memory. The internal hidden memory maintains the information about the dynamic behavior and history of all the previous units of the sequence [68, 69]. Due to their internal memory cell capability, RNNs have, in recent years, become popular focus of research topic in the areas of DNN as diverse as, for example, automatic speech recognition [30, 31, 33, 56, 59, 71], music generation [15], text generation [76], image classification [5, 23, 34, 44, 80, 86], facial recognition [46, 82], sentiment classification [83], credit card fraud detection [26] and numerous other areas of major advancements. RNNs use input sequence data such as text and speech to solve both for prediction [18] as well as classification problems [12, 45, 52]. Even though RNNs are very successful and powerful dynamic systems used to map input sequences to output sequences, however, as the long-range dependency and duration of training RNNs arbitrarily increases, there are critical practical difficulties to be addressed as explained in detail below [6, 21, 36, 65].

### 3.2.2 Long Short-Term Memory (LSTM)

Properly training RNNs for arbitrarily long-range dependencies suffers from *two* widely known critical issues of the vanishing and exploding gradient problems [6]. While training RNNs, these practical difficulties are associated with the derivatives of the gradient-based methods, e.g., backpropagation, getting big or small at each time step [48]. As the duration of the long sequence dependencies during the course of backpropagation increases, the derivatives of the gradient over many time steps may eventually explode or vanish during training since the growing memory requirement is proportional to the length of the sequence [6, 48]. To significantly reduce this problem of RNNs, a variety of well-known approaches have been proposed [16, 21, 29, 36, 40, 53, 74, 77, 93]. Consequently, these advancements have made RNNs to become successful on a number of difficult machine learning tasks such as end-to-end speech recognition [11], text generation [76], word embedding extraction from a sequence [57], sequence mapping [14], neural machine translation [2, 14], and many other more complex tasks. LSTM [27, 29, 36] is one of the most popular implementations of Recurrent Neural Networks (RNN) state-of-the-art architectures that use special hidden units designed for a wide range of sequence modeling tasks and time series prediction models with long-range dependencies. The LSTM unit [27, 36] is a powerful and flexible RNN tool that has a memory cell that gives a previous hidden state containing connection information through the hidden layer activations from the past for a long period of time.

LSTM in its recurrent hidden layer has a special unit called memory blocks consisting of memory cell units that are responsible for remembering the temporal states of the network for arbitrary time intervals [27, 29, 36]. In each layer of the LSTM architecture [27, 29, 36], there is a forward propagation step which is a corresponding backward propagation through time step. In addition to this, there is a cache that passes information from one layer to another. This ability of LSTM [36] allows us to solve the vanishing gradient problem by dynamically controlling the information flow within the layers and capture the long-term dependencies of the connections in a sequence effectively. LSTM [27, 36] is used to address difficult sequence learning and prediction problems in machine learning and have subsequently achieved state-of-the-art results.

In recent years, LSTM networks have proved to be more effective by outperforming traditional models in certain applications, e.g., speech recognition, especially when they have multiple layers for each time step [31, 77]. One of the main benefits of using an LSTM model for challenges that involve time series data is to avoid the vanishing gradient problem. RNN model scans through the training data from left to right and the parameters it uses to govern the connection in the hidden layer for each time-step, learned features during the training are shared and this significantly improves the prediction. An LSTM model computes a mapping from an input feature vector $x = (x_{(1)}, x_{(2)}, x_{(3)}, ..., x_{(\mathrm{n})})$ where $x_{\mathrm{i}} \in \mathbb{R}^n$ to a corresponding output sequence $y = (y_{(1)}, y_{(2)}, y_{(3)}, ..., y_{(\mathrm{n})})$ where $y_{\mathrm{i}} \in \mathbb{R}^n$ by calculating the network unit activations of a weighted sum using the Equations 3.3-3.8 iteratively from *t = 1* to *n*. As it is shown in Equations 3.3, 3.4, and 3.6, LSTM [27, 36] uses *three* adaptive, an *input*, *forget* and *output*, gates shared by all cells in the LSTM block in order to learn long-term dependencies and control the flow of information. The output of these gates multiplicatively influences connections within the memory units. The *input* gate determines the flow of input activations into the memory cell whereas the *output* gate determines the output flow of cell activations into the rest of the network. The *forget* gate determines the extent to which the current value remains in the memory cell of the LSTM unit before it gets gradually discarded when its data is no longer needed.

$$i_t = \sigma(W_{ix}x_t + W_{im}m_{t-1} + W_{ic}c_{t-1} + b_i) \tag{3.3}$$

$$f_t = \sigma(W_{fx}x_t + W_{fm}m_{t-1} + W_{fc}c_{t-1} + b_f) \tag{3.4}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot g(W_{cx}x_t + W_{cm}m_{t-1} + b_c) \tag{3.5}$$

$$o_t = \sigma(W_{ox}x_t + W_{om}m_{t-1} + W_{oc}c_t + b_o) \tag{3.6}$$

$$m_t = o_t \odot h(c_t) \tag{3.7}$$

$$y_t = \phi(W_{ym}m_t + b_y) \tag{3.8}$$

where the $i$, $f$, $c$, $o$ are *input*, *forget*, *memory state*, and *output* gate activation vectors respectively at each time step $t$. $\sigma$ is the logistic sigmoid non-linearity while $\odot$, $g$ and $h$ are element-wise product of the vectors, the cell input and output non-linearity activation functions of the entire neural network, $ReLU$ in our case, applied to each layer of the deep network respectively. $W$ and $b$ represents a vector of weighted recurrent connections and the bias vector. $m_t$ is the hidden state output of the LSTM layer. Finally, $\phi$ is the activation function in the hidden layer applied to the network output. Figure 3.3 describes the basic unit of an LSTM network where the input sequence to the LSTM cell is carried over each *time step* of *t+1*, *t* and *t-1*. As shown in Figure 3.3, the hidden state, at time step *t*, is a function of the current input sequence $x_t$ at the same time step. $C_t$ and $C_{t-1}$ are the memory cell state activation vectors from the current and previous block at time *t* and *t-1* respectively.

### 3.2.3 Multilayer Perceptron (MLP)

MLP is one of the deep learning models we have employed in our papers. It is a feedforward artificial neural network consisting of multiple layers of neurons or hidden elements called perceptrons that interact using weighted connections [17, 37, 67]. As it has been widely discussed in the neural network literature, a typical MLP consists of an *input layer*, a *hidden layer* consisting of intermediate processing units and an *output layer* [37, 68]. As shown in Figure 3.4, a simple MLP neural network architecture consists of $n$ input feature vectors, $x = [x_1, x_2, x_3, x_4, ..., x_n]$, adjustable vector weights associated with the $i^{th}$ input vector, $w_i, i = 1, 2, 3, 4, ..., n$, a bias $b$, a non-linear activation function

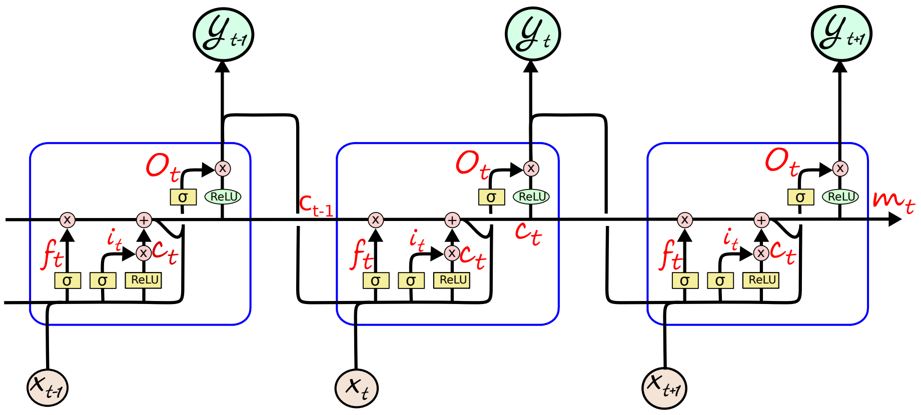


Figure 3.3: Simple LSTM Network Architecture. For more details, refer [64].

$\varphi$, $\mathbf{w}$ represents the vector of weights, and an expected output of the neuron $y_i$. The most classical case of MLP can be written mathematically as shown in Equation 3.9.

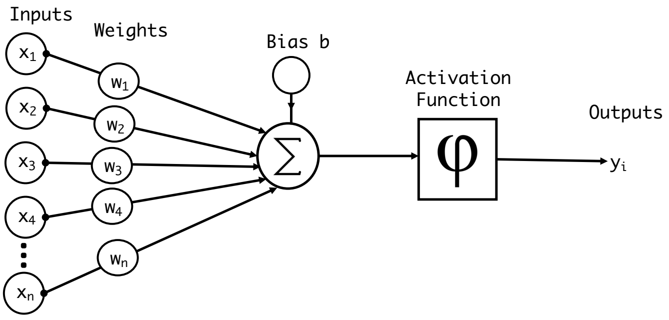


Figure 3.4: MLP neural network architecture.

$$y_i = \varphi\left(\sum_{i=1}^{n} w_i x_i + b\right) = \varphi\left(\mathbf{w}^T\mathbf{x} + b\right) \tag{3.9}$$

MLP is efficiently used for both feature selection and classification tasks [17]. Note that logistic regression is one special case of the MLP with no hidden elements [62]. Hence, MLP can be viewed as a collection of logistic regression classifier models where the final layer is being either another logistic regression or a linear regression model [62]. This depends depending on whether we are solving a classification or regression task. For example, if we are solving a regression task using two layers, the model has the forms presented in Equations 3.10 and 3.11 where $g$ is a non-linear activation (logistic) function, $\mathbf{z}(\mathbf{x}) = \phi(\mathbf{x}, \mathbf{V})$ is the hidden layer, $H$ is the number of hidden elements, $\mathbf{V}$ is the weight matrix from the inputs to the hidden nodes, and finally $\mathbf{w}$ is the weight vector from the hidden layer to the output layer.

$$p(y|\mathbf{x}, \boldsymbol{\theta}) = \mathcal{N}\left(y|\mathbf{w}^T\mathbf{z}(\mathbf{x}), \sigma^2\right) \tag{3.10}$$

$$\mathbf{z}(\mathbf{x}) = g(\mathbf{V}\mathbf{x}) = \left[g\left(\mathbf{v}_1^T\mathbf{x}\right), \ldots, g\left(\mathbf{v}_H^T\mathbf{x}\right)\right] \tag{3.11}$$

MLP has a universal approximation capability, that comes from the nonlinearities used in the nodes, for any continuous multivariate function [37]. This means that it can model any arbitrary function given enough hidden units. Hornik et al. has proved that a standard single hidden multilayer layer feedforward neural networks are capable of approximating any continuous function of interest, $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$, to any given degree of accuracy from one finite-dimensional space to another provided that adequate hidden units are available [37]. As shown in Equation 3.12, an MLP model is also designed to handle binary classification by passing the output through a sigmoid activation function. We can also extend this to predict multiple outputs that can be used for a multi-class classification as shown in Equation 3.13.

$$p(y|\mathbf{x}, \boldsymbol{\theta}) = \operatorname{Ber}\left(y|\operatorname{sigm}\left(\mathbf{w}^T\mathbf{z}(\mathbf{x})\right)\right) \tag{3.12}$$

$$p(y|\mathbf{x}, \boldsymbol{\theta}) = \operatorname{Cat}(y|\mathcal{S}(\mathbf{W}\mathbf{z}(\mathbf{x})) \tag{3.13}$$

Finally, it is good to remember that for different tasks, MLP uses different loss functions. For example, the loss function MLP uses for classification tasks is cross-entropy. However, for regression tasks, MLP uses the squared error loss function.

## 3.3 TCP Congestion Control

One of the main responsibilities of congestion control is to ensure efficient and fair sharing of the network's limited resources among its users. It ensures that sending nodes adjust their transmission rates based on the level of congestion in the network. Before congestion control was introduced into TCP, the sending rate was only limited by the receiver window (rwnd), maintained and advertised by the receiver to ensure appropriate flow control. With the introduction of congestion control, the sending rate is also limited by the Congestion Window (cwnd) maintained by the sender. The cwnd limits the maximum number of bytes that can be sent without being acknowledged at any time. The cwnd is changed dynamically according to the congestion control algorithm, and the maximum rate of transmission changes accordingly.

Over the years, TCP has adopted several end-to-end congestion control algorithms to address different requirements or to improve its general performance. Some of the congestion control algorithms introduced over the years are shown in Figure 3.5.

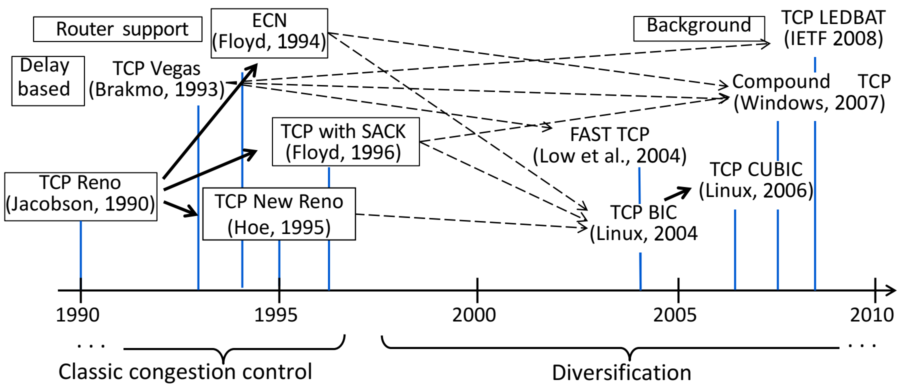


Figure 3.5: TCP historical background from 1990 - 2010.

## 3.3.1 Additive Increase and Multiplicative Decrease (AIMD)

The Additive Increase Multiplicative Decrease (AIMD) method is a basic building block for the congestion avoidance of many of the traditional TCP congestion control algorithms [13, 42]. With AIMD, the window size is first increased linearly by $\frac{\alpha}{cwnd}$ every time an ACK is received [1], until there is a congestion indication, such as when an ACK timeout is triggered.. Thus, the TCP sender will effectively increase the window linearly by roughly $\alpha$ segments for every Round-trip Time (RTT). However, when the congestion indication occurs, the window size is decreased multiplicatively by a factor $\beta$, i.e., the new reduced window size will be only a $\beta$-factor of the window size when the congestion indication occurred.

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, the AIMD control is illustrated by Equation 3.14.

$$f(t+1) = \begin{cases} f(t) + \alpha, & \text{If congestion is detected} \\ f(t) \times \beta, & \text{If congestion is not detected} \end{cases} \tag{3.14}$$

## 3.3.2 Phases of TCP implementations

TCP has the following phases:

- **Slow start**: Instead of the Additive Increase (AI) portion of the AIMD, the cwnd size increases exponentially, as shown in Figure 3.6. The window increases by one for each ACK received, resulting in a doubling of the cwnd size for every RTT until either a packet loss has occurred, the rwnd limit is reached, or if the given Slow Start Threshold (ssthresh) is reached [1, 39].

- **Congestion avoidance**: Traditionally TCP increases the transmission rate and then backs off when it sees signs of congestion. AIMD is an example of a congestion avoidance mechanism.
- **Fast retransmit**: The fast retransmit mechanism triggers the retransmission of a randomly dropped packet before the regular Retransmission Timeout (RTO) expires or before three duplicate ACKs are received [39]. Fast retransmit can fix a problem with a spuriously lost or reordered packet, without necessarily going into multiplicative decrease.
- **Fast recovery**: Fast recovery works by effectively avoiding the slow start phase even if there are still ACKs arriving in the pipe. When the TCP sender receives a duplicate ACK during fast recovery, instead of dropping its current cwnd all the way back to 0, the fast recovery algorithm simply drops it multiplicatively, e.g. to $\frac{cwnd}{2}$.

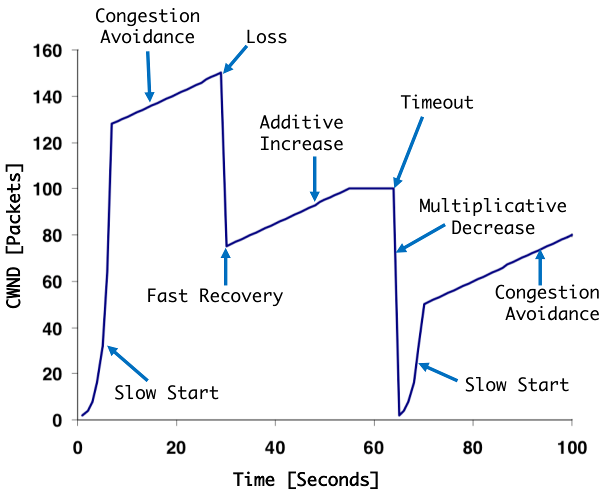


Figure 3.6: Phases in TCP cwnd.

## 3.3.3 Flavors of TCP congestion control algorithms

Transmission Control Protocol (TCP) is one of the dominant transport protocols that has significantly played a great role in the exponential success of the Internet, network technologies and applications [39, 66]. As explained above, the majority of all Internet traffic all over the world today use TCP due to practical considerations that favored TCP over other transport protocols [24]. The TCP congestion control strategies are broadly categorized into loss-based and delay-based variants. Detailed background on these two categories of TCP variants is presented as follows.

- **Loss-based TCP flavors**: One category of the widely deployed variants ranging from TCP CUBIC [32], Reno [39], BIC [94], etc. where packet loss probability is an implicit signal for congestion in the underlying network are called loss-based TCP congestion control algorithms. TCP variants of this kind aggressively fill up the actual network buffers in order to achieve better throughput by ignoring queueing delay and hence they tend to induce large queueing delays when the buffer sizes are large. However, this is challenging for the quality of latency-sensitive and bandwidth-intensive real-time media applications to achieve good performance when long-running flows also share large bottleneck link buffers. Therefore, to address this challenging problem, delay-based TCP schemes that adopt packet queueing delay rather than a loss as congestion signals are introduced.

- **Delay-based TCP flavors**: Unlike traditional loss-based approaches, delay-based TCP congestion control algorithms use the changes in queueing delay measurements as implicit feedback to congestion in the underlying network. Delay-based congestion control algorithms attempt to avoid network congestion by monitoring the trend of network path's RTT information contained in packets. In order to properly allocate, share the underlying network resources, and ensure network queueing delay stays low, delay-based congestion control algorithms require knowledge of an accurate estimate of the network path's base *smallest* possible RTT in the absence of congestion ($BaseRTT$) [50]. With delay-based congestion control algorithms, allocating network resources across competing users can be attained by supporting both high network utilization and low queuing delay even when the buffer sizes are large. Some of the end-to-end widely used delay-based congestion control algorithms on the Internet we use for our experimental evaluations include TCP Vegas [8] and TCP Veno [25].

### 3.3.4 Summary

To address the three use cases of the dissertation, we have used a number of state-of-the-art machine learning and deep learning techniques. Some of the classical machine learning methods we applied in Paper I, Paper II, Paper VII, and Paper VIII are: SVMs [10, 87], Naïve Bayes [20, 51], Random Forest [9], KNN [19], and LASSO regression [84, 85]. In Paper VI, we applied data-driven classification techniques based on probabilistic models and Bayesian inference by employing a novel non-stationary time series approach from a stochastic nonparametric perspective using a two-sided Kolmogorov–Smirnov test. The two most widely used RNN techniques we employed in Paper IV, Paper V, and Paper VIII are: MLP [17, 37, 67] and LSTM [27, 29, 36].

## 3.4 References

[1] M. Allman, V. Paxson, and E. Blanton. TCP congestion control. RFC 5681, 2009.

[2] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[3] Y. Bengio, A. Courville, and P. Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013.

[4] Y. Bengio et al. Learning deep architectures for AI. *Foundations and trends® in Machine Learning*, 2(1):1–127, 2009.

[5] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle. Greedy layer-wise training of deep networks. In *Advances in neural information processing systems*, pages 153–160, 2007.

[6] Y. Bengio, P. Simard, P. Frasconi, et al. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[7] C. M. Bishop. *Pattern recognition and machine learning.* springer, 2006.

[8] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. *TCP Vegas: New techniques for congestion detection and avoidance*, volume 24. ACM, 1994.

[9] L. Breiman. Random forests. *Machine learning*, 45(1), 2001.

[10] C. J. Burges. A tutorial on support vector machines for pattern recognition. *Data mining and knowledge discovery*, 2(2):121–167, 1998.

[11] W. Chan, N. Jaitly, Q. Le, and O. Vinyals. Listen, attend and spell: A neural network for large vocabulary conversational speech recognition. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4960–4964. IEEE, 2016.

[12] Z. Che, S. Purushotham, K. Cho, D. Sontag, and Y. Liu. Recurrent neural networks for multivariate time series with missing values. *Scientific reports*, 8(1):6085, 2018.

[13] D.-M. Chiu and R. Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Computer Networks and ISDN systems*, 17(1):1–14, 1989.

[14] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[15] K. Choi, G. Fazekas, M. Sandler, and K. Cho. Convolutional recurrent neural networks for music classification. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 2392–2396. IEEE, 2017.

[16] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[17] R. Collobert and S. Bengio. Links between perceptrons, MLPs and SVMs. In *Proceedings of the twenty-first international conference on Machine learning*, page 23. ACM, 2004.

[18] J. T. Connor, R. D. Martin, and L. E. Atlas. Recurrent neural networks and robust time series prediction. *IEEE transactions on neural networks*, 5(2):240–254, 1994.

[19] T. M. Cover, P. Hart, et al. Nearest neighbor pattern classification. *IEEE transactions on information theory*, 13(1):21–27, 1967.

[20] P. Domingos and M. Pazzani. On the optimality of the simple Bayesian classifier under zero-one loss. *Machine learning*, 29(2-3):103–130, 1997.

[21] S. El Hihi and Y. Bengio. Hierarchical recurrent neural networks for long-term dependencies. In *Advances in neural information processing systems*, pages 493–499, 1996.

[22] D. Erhan, Y. Bengio, A. Courville, P.-A. Manzagol, P. Vincent, and S. Bengio. Why does unsupervised pre-training help deep learning? *Journal of Machine Learning Research*, 11(Feb):625–660, 2010.

[23] C. Farabet, C. Couprie, L. Najman, and Y. LeCun. Learning hierarchical features for scene labeling. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1915–1929, 2012.

[24] M. Fomenkov, K. Keys, D. Moore, and K. Claffy. Longitudinal study of Internet traffic in 1998-2003. In *Proceedings of the winter international synposium on Information and communication technologies*, pages 1–6. Trinity College Dublin, 2004.

[25] C. P. Fu and S. C. Liew. TCP Veno: TCP enhancement for transmission over wireless access networks. *IEEE Journal on selected areas in communications*, 21(2):216–228, 2003.

[26] K. Fu, D. Cheng, Y. Tu, and L. Zhang. Credit card fraud detection using convolutional neural networks. In *International Conference on Neural Information Processing*, pages 483–490. Springer, 2016.

[27] F. A. Gers, J. Schmidhuber, and F. Cummins. Learning to forget: Continual prediction with LSTM. 1999.

[28] I. Goodfellow, Y. Bengio, and A. Courville. *Deep learning.* MIT press, 2016.

[29] A. Graves. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, 2013.

[30] A. Graves and N. Jaitly. Towards end-to-end speech recognition with recurrent neural networks. In *International conference on machine learning*, pages 1764–1772, 2014.

[31] A. Graves, A.-r. Mohamed, and G. Hinton. Speech recognition with deep recurrent neural networks. In *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on*, pages 6645–6649. IEEE, 2013.

[32] S. Ha, I. Rhee, and L. Xu. CUBIC: a new TCP-friendly high-speed TCP variant. *ACM SIGOPS operating systems review*, 42(5):64–74, 2008.

[33] G. Hinton, L. Deng, D. Yu, G. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, B. Kingsbury, et al. Deep neural networks for acoustic modeling in speech recognition. *IEEE Signal processing magazine*, 29, 2012.

[34] G. E. Hinton, S. Osindero, and Y.-W. Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

[35] G. E. Hinton, T. J. Sejnowski, and T. A. Poggio. *Unsupervised learning: foundations of neural computation.* MIT press, 1999.

[36] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[37] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[38] K. Imai, L. Keele, and T. Yamamoto. Identification, inference and sensitivity analysis for causal mediation effects. *Statistical science*, pages 51–71, 2010.

[39] V. Jacobson. Congestion avoidance and control. In *ACM SIGCOMM computer communication review*, volume 18, pages 314–329. ACM, 1988.

[40] M. Jaderberg, W. M. Czarnecki, S. Osindero, O. Vinyals, A. Graves, D. Silver, and K. Kavukcuoglu. Decoupled neural interfaces using synthetic gradients. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1627–1635. JMLR. org, 2017.

[41] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[42] D. Katabi, M. Handley, and C. Rohrs. Congestion control for high bandwidth-delay product networks. *ACM SIGCOMM computer communication review*, 32(4):89–102, 2002.

[43] D. G. Kleinbaum, K. Dietz, M. Gail, M. Klein, and M. Klein. *Logistic regression.* Springer, 2002.

[44] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[45] S. Lai, L. Xu, K. Liu, and J. Zhao. Recurrent convolutional neural networks for text classification. In *Twenty-ninth AAAI conference on artificial intelligence*, 2015.

[46] S. Lawrence, C. L. Giles, A. C. Tsoi, and A. D. Back. Face recognition: A convolutional neural-network approach. *IEEE transactions on neural networks*, 8(1):98–113, 1997.

[47] Y. Le Cun. Learning process in an asymmetric threshold network. In *Disordered systems and biological organization*, pages 233–240. Springer, 1986.

[48] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[49] Y. LeCun, D. Touresky, G. Hinton, and T. Sejnowski. A theoretical framework for back-propagation. In *Proceedings of the 1988 connectionist models summer school*, volume 1, pages 21–28. CMU, Pittsburgh, Pa: Morgan Kaufmann, 1988.

[50] D. J. Leith, R. N. Shorten, G. McCullagh, L. Dunn, and F. Baker. Making available base-RTT for use in congestion control applications. *IEEE Communications Letters*, 12(6):429–431, 2008.

[51] D. D. Lewis. Naive (bayes) at forty: The independence assumption in information retrieval. In *European conference on machine learning*, pages 4–15. Springer, 1998.

[52] P. Liu, X. Qiu, and X. Huang. Recurrent neural network for text classification with multi-task learning. *arXiv preprint arXiv:1605.05101*, 2016.

[53] J. Martens and I. Sutskever. Learning recurrent neural networks with hessian-free optimization. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 1033–1040. Citeseer, 2011.

[54] J. McCarthy and E. A. Feigenbaum. In memoriam: Arthur samuel: Pioneer in machine learning. *AI Magazine*, 11(3):10–10, 1990.

[55] S. Menard. *Applied logistic regression analysis*, volume 106. Sage, 2002.

[56] T. Mikolov, A. Deoras, D. Povey, L. Burget, and J. Černockỳ. Strategies for training large scale neural network language models. In *2011 IEEE Workshop on Automatic Speech Recognition & Understanding*, pages 196–201. IEEE, 2011.

[57] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[58] T. M. Mitchell. Machine learning and data mining. *Communications of the ACM*, 42(11), 1999.

[59] A.-r. Mohamed, G. E. Dahl, and G. Hinton. Acoustic modeling using deep belief networks. *IEEE transactions on audio, speech, and language processing*, 20(1):14–22, 2011.

[60] M. Mohri, A. Rostamizadeh, and A. Talwalkar. *Foundations of Machine Learning*. MIT press, 2012.

[61] D. C. Montgomery, E. A. Peck, and G. G. Vining. *Introduction to linear regression analysis*, volume 821. John Wiley & Sons, 2012.

[62] K. P. Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[63] T. T. Nguyen and G. J. Armitage. A survey of techniques for internet traffic classification using machine learning. *IEEE Communications Surveys and Tutorials*, 10(1-4):56–76, 2008.

[64] C. Olah. Understanding LSTM Networks. https://colah.github.io/posts/2015-08-Understanding-LSTMs, 2015.

[65] R. Pascanu, T. Mikolov, and Y. Bengio. On the difficulty of training recurrent neural networks. In *International conference on machine learning*, pages 1310–1318, 2013.

[66] J. Postel. Transmission control protocol. RFC 793, 1981.

[67] F. Rosenbaltt. The perceptron–a perciving and recognizing automation. *Report 85-460-1 Cornell Aeronautical Laboratory, Ithaca, Tech. Rep.*, 1957.

[68] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[69] D. E. Rumelhart, G. E. Hinton, R. J. Williams, et al. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1, 1988.

[70] S. J. Russell and P. Norvig. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,, 2016.

[71] T. N. Sainath, A.-r. Mohamed, B. Kingsbury, and B. Ramabhadran. Deep convolutional neural networks for LVCSR. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 8614–8618. IEEE, 2013.

[72] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[73] G. A. Seber and A. J. Lee. *Linear regression analysis*, volume 329. John Wiley & Sons, 2012.

[74] M. Seo, S. Min, A. Farhadi, and H. Hajishirzi. Neural speed reading via skim-rnn. *arXiv preprint arXiv:1711.02085*, 2017.

[75] P. Simon. *Too big to ignore: the business case for big data*, volume 72. John Wiley & Sons, 2013.

[76] I. Sutskever, J. Martens, and G. E. Hinton. Generating text with recurrent neural networks. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 1017–1024, 2011.

[77] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[78] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction.* MIT press, 2018.

[79] R. S. Sutton, A. G. Barto, et al. *Introduction to reinforcement learning*, volume 2. MIT press Cambridge, 1998.

[80] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[81] C. Szepesvári. Algorithms for reinforcement learning. *Synthesis lectures on artificial intelligence and machine learning*, 4(1):1–103, 2010.

[82] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf. Deepface: Closing the gap to human-level performance in face verification. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1701–1708, 2014.

[83] D. Tang, B. Qin, and T. Liu. Document modeling with gated recurrent neural network for sentiment classification. In *Proceedings of the 2015 conference on empirical methods in natural language processing*, pages 1422–1432, 2015.

[84] R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1):267–288, 1996.

[85] R. Tibshirani. Regression shrinkage and selection via the lasso: a retrospective. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 73(3):273–282, 2011.

[86] J. J. Tompson, A. Jain, Y. LeCun, and C. Bregler. Joint training of a convolutional network and a graphical model for human pose estimation. In *Advances in neural information processing systems*, pages 1799–1807, 2014.

[87] V. Vapnik. *The nature of statistical learning theory.* Springer, 1995.

[88] P. Werbos. Beyond Regression:" New Tools for Prediction and Analysis in the Behavioral Sciences. *Ph. D. dissertation, Harvard University*, 1974.

[89] P. J. Werbos. Applications of advances in nonlinear sensitivity analysis. In *System modeling and optimization*, pages 762–770. Springer, 1982.

[90] P. J. Werbos. Backwards differentiation in AD and neural nets: Past links and new opportunities. In *Automatic differentiation: Applications, theory, and implementations*, pages 15–34. Springer, 2006.

[91] M. Wiering and M. Van Otterlo. Reinforcement learning: State-of-the-art. *Adaptation, learning, and optimization*, 12:3, 2012.

[92] R. J. Williams and D. Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural computation*, 1(2):270–280, 1989.

[93] Y. Wu, S. Zhang, Y. Zhang, Y. Bengio, and R. R. Salakhutdinov. On multiplicative integration with recurrent neural networks. In *Advances in neural information processing systems*, pages 2856–2864, 2016.

[94] L. Xu, K. Harfoush, and I. Rhee. Binary increase congestion control (BIC) for fast long-distance networks. In *IEEE INFOCOM*, volume 4, pages 2514–2524. IEEE, 2004.

Chapter 4

# Related Work

This chapter briefly presents a summary of the relevant related works found in the literature for the three use cases we presented in Chapter 1.

**TCP State Monitoring from Passive Traffic Measurements**: Here we summarize the most important related works from our papers I through VI.

Much of the substantial literature on Transmission Control Protocol (TCP) state monitoring from passive traffic measurements are complementary to our approaches. In the traditional methods of end-to-end passive measurement, there has been much interest in the investigation of TCP connections aggregate properties and its characteristics on the global Internet. Starting with [32, 60], TCP congestion control has been an active area of research in the networking community. Our initial work in the first use case is particularly related to prior work that provides a passive measurement methodology to infer and keep track of the values of the sender variables: end-to-end Round-trip Time (RTT) and cwnd [34]. The idea is to emulate a state transition by detecting Retransmission Timeout (RTO) events at the sender and observing the ACKs which cause the sender to change the value of the cwnd. This work, [34], 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 [34] 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 [34], the work in [35], 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 [34]. Jitendra et al. [55] presents a tool called TBIT that characterizes the TCP behavior of a remote web server running over a real TCP implementation in a specific operating system by actively sending requests to web servers irrespective of the TCP variant.

In contrast, we propose to use a passive measurement approach which, as we motivate in Chapter 1, has a lot of benefits over active methods. The authors of the study [61] developed a tool called tcpflows that attempts to passively estimate the value of cwnd 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, we believe, it cannot uniquely identify the cwnd characteristics of newly deployed TCP variants. In Paper I, we show 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 demonstrate how the intermediate node can predict the Congestion Window (cwnd) size of the TCP client. Our method can also be extended to predict other TCP transmission states of the client. We use a generic machine learning-based prediction approach for inferring cwnd within a flow from passive traffic collected at an intermediate node. Our experimental results of Paper I indicate the effectiveness of our prediction model with reasonably good accuracy across different scenarios and multiple modern loss-based TCP variants by examining each cross-traffic of TCP flows of the endpoints passively collected at an intermediate node.

As an extension of this work, in Paper II we present a robust, scalable and generic machine learning-based model which may be of interest for network operators that experimentally infers the underlying variant of loss-based TCP algorithms within flow from passive traffic measurements collected at an intermediate node. Identifying the underlying TCP variant from passive measurements is important for several reasons, e.g., exploring security ramifications, traffic engineering in the Internet, etc. Oshio et al. [54] proposes a cluster analysis-based method that aims to identify between two versions of 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 [54]. These characteristics are then clustered into two groups by applying a hierarchical clustering technique. The authors show that only 2 out of 14 TCP congestion algorithms that are implemented in Linux can be identified based on their method [54].

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 [34, 58]. Our work mainly differs from the previous research works in that our main goal is 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 [72], CUBIC [25], Reno [32] etc.). Combining these two contributions, in Paper IV we have presented Long Short-Term Memory (LSTM)-based Recurrent Neural Networks (RNN) prediction approach for building a generic prediction model for TCP connection characteristics from passive measurements.

As a parallel contribution, in Paper VI, we investigated the delay characteristics of widely used TCP algorithms that exploit queueing delay as a congestion signal and as a result, we present an effective TCP variant identification methodology from traffic measured passively by analyzing $\beta$, the multiplicative back-off factor to decrease the cwnd on a loss event, and the queueing delay values. In Paper VI, we further employ a novel non-stationary time series approach from a stochastic nonparametric perspective using a two-sided Kolmogorov–Smirnov test to classify delay-based TCP algorithms based on the $\alpha$, the rate at which a TCP sender's side cwnd grows per window of acknowledged packets, parameter. Through extensive experiments on emulated and realistic scenarios, we demonstrate that the data-driven classification techniques based on probabilistic models and Bayesian inference for optimal identification of the underlying delay-based TCP congestion algorithms give promising results.

In Paper V, we propose and evaluate a novel deep learning-based model capable of dynamically predicting at real-time the RTT between the sender and receiver with high accuracy based on passive measurements collected at an intermediate node, taking advantage of the commonly used TCP timestamps. Measuring the network RTT has been widely acknowledged as one of the most crucial study findings in understanding the important characteristics of TCP connection on the public Internet. Hence, our contribution in Paper V benefits from a wide range of existing passive measurement-related research work in computer networking. TCP implements a retransmission strategy by setting the time-out interval to ensure data delivery in the absence of any acknowledgment for a particular segment from the receiver side [57]. The timer relies on the measurement of the network latency which TCP does by periodically estimating the current RTT of every active connection in order to determine the RTO when it sends data and receiving an acknowledgment for it.

Accurate measurement of RTO is crucial to TCP performance and it is determined by estimating the mean and variance of the estimated RTT [57]. When the timer RTO expires, the segment is retransmitted. To compute the current RTO, TCP sender keeps track of the Smoothed Round-Trip Time (SRTT) and the Round-Trip Time Variation (RTTVAR) state variables. When the first RTT measurement $R$ is made on the active connection, the host should compute the following Jacobson RTO Estimation algorithm [33]. After computing the RTO, if its value is less than *1 second*, then the RTO value should be rounded up to *1 second* [57]. However, the timeout can expire spuriously across low-bandwidth network paths and triggers unnecessary retransmissions when no packets have been lost [24]. Modern operating systems like Linux have a minimum value for RTO in order to avoid unnecessary high retransmission delays of an open active connection. The potential pitfall of choosing a low RTO, however, is that it may trigger retransmission of a packet even though the segment is received and an ACK is on its way. Setting a low value for RTO works better when there is a moderate background traffic [47].

To address the critical problem of spurious timeouts, a number of approaches have been proposed. For example, RTO estimators like [57] are based on the assumptions of older technologies. As described earlier, spurious timeouts lead to problems that cause several unnecessary retransmissions and congestion control back-off that affect the TCP throughput. In addition to this, estimating the RTT measurements are challenging in the presence of timeouts and packet loss in the end-to-end path. This is because of the receipt of an ACK after $R$ retransmissions, the sender cannot tell which one of the $R+1$ data sent is being acknowledged which again affects the measurement of SRTT. Wrongly computed SRTT values will eventually lead to wrong RTO values. If the value of RTO is too small, it will lead to unnecessary retransmission of data segments which again increases the load on the underlying network capacity. But if the value of RTO is too large, the sender waits too long before retransmitting lost segments which again increases delay and lowers the throughput for connections with packet loss. Making use of the TCP timestamps, the Eifel [24] algorithm has pointed out that it is possible to detect spurious TCP timeouts problems and recover by restoring a TCP sender's congestion control state saved before the timeout. There are other previous research works who have examined and reported RTT estimation for TCP [3, 34, 36]. The approach presented in [36] uses a unidirectional flow during the TCP handshake of a connection to estimate RTT using the time from SYN to SYN+ACK method. The approaches proposed in [36] calculates one RTT sample per TCP connection associated either during the three-way handshake or during the slow-start phase. If we have captured the TCP three-way handshake as presented in [36], we can calculate the initial RTT (*iRTT*) by taking the time difference from the SYN packet to the ACK packet of the handshake. However, since the TCP handshake packets are processed by the kernel, the RTTs during the data transfer will probably be slightly larger than the *iRTT*. Hence, this approach may tend to underestimate the actual RTT. In addition to this, since TCP sets the initial retransmission timeout value to *3 seconds* [57], therefore this approach is not applicable in scenarios where the TCP connection setup takes longer which leads to long delays and packet losses introduced by the network.

The study in [3] has reported a statistical characterization of RTT variability where the measurement point is closer to the sender. However, their study does not take delayed ACKs into account. The authors of [34] have introduced an approach for RTT measurements of TCP connections based on bidirectional traces captured at the monitoring point using a finite state machine that replicates the TCP sender states of observed ACKs depending on the underlying TCP flavor. The authors have pointed out that the estimation of the TCP parameters (e.g., cwnd) may have potential errors primarily due to over-estimation of the RTT and incorrect window sizes of a connection [34]. As stated above in detail, another limitation of this work, given differences of the many existing flavors of TCP stack implementations, the use of a separate state machine for each TCP variant is unscalable. 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. Our deep learning-based approach using LSTM to passively predict the continuous RTT measurement throughout the lifetime of a TCP session builds upon these classical approaches by avoiding the limitations taking advantage of the commonly used timestamp option.

**Network Intrusion Detection**: There has been much discussion in the computer security literature about the nature of Intrusion Detection Systems (IDS). An IDS is an active device or process that ultimately monitors, analyzes system and network policy violation for unauthorized entry or malicious activity [15, 71]. As computer and enterprise network systems have become more dynamic and complex over the years, chances for attackers to compromise security flaws in these systems have also dramatically increased. Even though static computer network security mechanisms like a firewall can provide a fairly acceptable level of security, more modern and sophisticated IDS that adapts to rapidly changing security threats and cybercrime should be used in computer networks. The role of IDS techniques in the computer security community are very crucial in monitoring computer network events for malicious activities, such as attacks against hosts and protecting computer systems and network infrastructures from a potential attack [7]. The problem with the evolution of network threats and attacks is that they are getting harder to detect and therefore it could be difficult to find out whether network traffic is normal or anomalous. Commercially available IDS techniques are mainly signature-based that are designed to detect known attacks by using the signatures of those attacks [40]. Some generic approaches to signature-based methods have been reported in the literature [65, 66, 70]. Such systems must be frequently updated with rule-sets and signature updates of the recent threat vectors, and are not capable of detecting unknown attacks in network traffic. Traditionally, several IDS methods use a signature-based approach in which events are detected and compared against a predefined database of signatures of known attacks that are provided by a network administrator.

The traditional approaches to IDS depend on experts codifying rule-sets defining normal behavior and intrusions in a network [59, 63]. The two broad categories of IDS methods in the intrusion detection literature are misuse and anomaly detection [9, 15, 17, 48]. Misuse detection is a technique based on rule-sets, either pre-configured by the system or set up manually by an administrator. This technique involves matching the signatures of known security attacks in a network against events currently taking place in the system that should be considered as misuse [30, 59]. One of the main limitations of this approach is the failure of detecting and identifying unknown computer attacks that do not have known signatures. Anomaly detection method, on the other hand, refers to the problem of finding patterns in data that do not comply with an expected notion of normal behavior in a dataset. In the context of computer security, everything interpreted as a deviation from the profile of a normal system or user behavior is evidence of a malicious activity [16, 22, 41]. Anomaly detection methods, however, can detect new attacks but the problem with anomaly detection is that it has a higher false-positive rate or misleading false alarms.

As described in detail above, modern machine learning techniques have effectively revolutionized the state-of-the-art for many research domain problems in the networking research community. For example, in the areas of security monitoring and IDS [10, 28, 67], fraud detection [51, 52], Spam detection [13, 38, 44, 56], and many other fascinating topics in computer networks such as traffic anomaly detection [2]. Hence, machine learning techniques have the potential of detecting unknown attacks in network traffic sharing features with other attacks by being trained on normal and abnormal types of traffic. However, one critical problem in machine learning is identifying and selecting the most relevant input features from which to construct an accurate model based on training data for a particular classification task. As it is reported in the literature [19, 45, 62, 64], employing machine learning techniques on the benchmark intrusion detection dataset, NSL-KDD [31], gives a very low level of detection rate on attack categories involving content features (i.e., user-to-root (U2R) and remote-to-local (R2L) attacks) within the misuse detection context. However, with the same set of 41 features, the detection rate for normal, denial-of-service (DoS) and probe are accurately high. We, therefore, believe it is important to do feature selection analysis to make it easier for network administrators to better understand the features that contribute to security attacks. Selecting the most relevant actual features improves the detection quality for many algorithms that are based on learning techniques [29]. Some previous works have addressed different techniques that help identify the important input features in building IDS [1, 6, 8, 12]. Feature selection helps to understand better which actual features are the most important ones to find attacks in a network. Note that, here we only discuss relevant previous works that have used the benchmark intrusion detection public dataset, NSL-KDD [31], for their performance benchmarking.

In Paper VII, we address the problem of an actual feature selection for IDS to find attack categories in a network by introducing cross-validated regularized machine learning techniques and an artificial neural network feature ranking methods. Paper VII focuses mainly on the contribution of the actual input features that are well understood within the networking community to find what kinds of attacks in a network are the most significant. To that end, the actual input features studied in this paper are ranked into strongly contributing, low contributory and irrelevant using a combination of feature selection filters and wrapper methods by carefully carrying out comparisons with previous techniques. Our paper investigates the most important features in identifying well-known security attacks by using Support Vector Machines (SVMs) wrapped with Recursive Feature Elimination (RFE) algorithm and $\ell_1$-regularized method with Least Absolute Shrinkage and Selection Operator (LASSO) for robust regression both to binary and multiclass attack classification to give us an insight into features of different classes of security attacks. SVMs are one of the standards of machine learning classification techniques that give a reasonably good performance but with some drawbacks in terms of interpretability. On the other hand, LASSO is a regularized regression method often performing comparably well and it has extra compelling advantages of being very easily interpretable. Moreover, LASSO is much more computationally effective and provides coefficients that contribute to how individual input features affect the probability of specific security attack classes to occur. Hence, Paper VII uses LASSO in particular for multiclass classification in order to better understand, from a security engineering perspective, the dynamics behind the classification model and get a better insight into which actual features shared by security attacks in a network are the most important ones.

**Passive Operating System (OS) Fingerprinting**: Remote Operating Systems (OSes) fingerprinting has a long history in the computer security community [5, 43, 46, 50]. Collecting detailed information about the underlying OS running on a remote computer is important for several reasons, e.g., detecting possible security vulnerabilities, defining OS-based access control security policies, configuring network-based IDS to classify and prioritize extraneous security alerts etc. A significant part of the literature on OS fingerprinting focuses on TCP/IP header information. This is mainly because TCP/IP header fingerprinting and any information related to application protocols are used to identify the underlying OS running on a remote host either actively or passively [49]. There are multiple existing tools for both the predominant active and passive OS fingerprinting approaches. Many of the existing popular OS fingerprinting tools depend on generating multiple active probes by introducing additional traffic to the network and analyzing the corresponding potentially identifying replies from the target hosts. For example, Nmap [50] is one of the most prominent open-source active fingerprinting tools. Nmap [50] exchanges multiple TCP SYN packets with the target hosts and then analyzes the SYN/ACK responses from the remote computer by examining the network behavior of known TCP/IP stack [68]. However, since active fingerprinting leads to longer scan times and injects additional traffic to the network by generating active probes, it may itself trigger false alarms and get blocked by firewall rules and Network address translators (NATs) [23]. The previous work presented in [69], SYNSCAN, works in a similar fashion to Nmap [50] but it performs the fingerprinting task by actively sending a small number of crafted network packets to a single TCP port. Xprobe2 [73] is another popular fingerprinting tool that relies primarily on Internet Control Message Protocol (ICMP) packets and it depends on how many changes we make to the default TCP/IP stack parameters. Since Xprobe2 does fuzzy fingerprinting with a signature matching algorithm as an alternative to Nmap, it means if we make a lot of changes to the default TCP/IP stack parameters, the underlying OS will not be detected. However, Xprobe2 is more robust to small fingerprint variations as compared to Nmap. As explained above the other fingerprinting tools, Ettercap [53] and p0f [74], have not been updated since 2011 and 2014 respectively to include variations of modern OSes.

For an effective passive OS fingerprinting, we believe a limitation of these fingerprinting approaches needs to be addressed. The work in [46] also demonstrates that the OS fingerprinting accuracy for Ettercap and p0f signature databases is low and proposed techniques to improve performance. It, hence, presents rule-based machine learning classifiers capable of identifying 75 classes of OSes from TCP/IP packet headers found in the Ettercap database. They proposed a classifier technique using k-nearest neighbors (KNN) that returns an approximate first match for an OS from a fingerprint database instead of avoiding hosts being classified as unknown if no exact match is found in the database [46]. However, their evaluation yielded poor experimental results, rejecting as much as 84% of the test packets, while 44% of the accepted patterns were wrongly classified [20]. The problems contributing to poor performance was believed to be caused by two main issues. First, substitution errors due to multiple OSes with exactly the same fingerprint feature values. The second reason for the poor performance is the high rejection rate caused by numerous unique feature values derived from the same OS. After combining the OS classes most often confused with each other, eliminating all the classes where the error could not be reduced by combining classes, the error percentage was reduced to 9.8% with no rejected packets. Beyond remote OS detection using TCP/IP network stacks, fingerprinting techniques have also been extended to be applied for remote device level fingerprinting [20]. This is one of the major drawbacks of active fingerprinting.

Recent related works have shown promising advances towards remote devices fingerprinting using different approaches [11, 14, 18, 37, 39]. Lastovicka et al. [43] has discussed a recent work of interest that is most closely related to our work, presented in Paper VIII, which has also given a comprehensive survey on passive fingerprinting methods. They have employed OS fingerprinting methods in the environment of wireless networks. Besides of using the three basic TCP/IP stacks (i.e., TTL, window size and initial SYN packet size), the authors suggested that a flow-based fingerprinting and classification using methods based on user-agents of HTTP request headers and communication with OS-specific domains can be usable in large dynamic networks [43]. A parallel line to their previous work, Lastovicka et al. [42] proposed system architecture to detect the underlying OS of every actively communicating device in the network using the methods presented in [43]. The average accuracy of OS classification using the TCP/IP parameters reported in [43] is 80.88%. Aksoy et al. [5] have employed genetic algorithms for identifying packet features suitable for OS classification based on the analysis of the network TCP/IP packets using three machine learning algorithms. They argued that combining automatic feature selection and machine learning algorithms enable for an adaptive OS classification. Aksoy et al. [4] has also recently applied the same techniques to select features that are most unique for the automatic identification of Internet of Things (IoT) devices. Zhang et al.'s paper on OS detection [75] utilizes only one machine learning technique using Support Vector Machine (SVM). However, the testing error rate of identifying some of the OSes e.g., Mac, Cisco, FreeBSD, and OpenBSD is 25.80%, 24.22%, 17.71%, and 15.85% respectively. Gagnon et al. [21], demonstrate the capabilities of a hybrid approach using diagnosis theory in addressing the fundamental limitations of both active and passive fingerprinting techniques.

However, most of these previous works use the basic actual TCP/IP features for evaluating passive OS fingerprinting. Besides, we believe that these tools have the inability to extract all possible OS-specific features that are the basis for passively fingerprinting the underlying OSes. In contrast, what separates our contribution in this paper from the other previous related works is that our tool supports a wider range of TCP/IP network stack features.

Unlike to the previous works, the main goal of our work presented in Paper VIII is to combine these basic TCP features that are the basis of OS fingerprinting and other settings with the underlying TCP variant by leveraging both machine learning and deep learning techniques. This contribution remains largely unexplored and is not used by existing fingerprinting techniques. For this contribution, we follow two approaches. Firstly, we use the default TCP variant of an OS as a feature along with the basic TCP/IP network stack features. Secondly, we believe it is natural to ask one valid question: *what happens if we don't know the underlying OS*? That means we don't know the implementation of the default TCP variant. This is where our previous works on

TCP sate prediction from passive traffic measurements [26, 27] come into play. In this contribution, we want to take the OSes fingerprinting problem one step further by combining the basic TCP/IP features and other settings with the TCP variant as a feature in our model. The reason why we concentrate on the implementations of the underlying TCP variant as a feature in our OSes classifier models is that due to the fact that different OSes are doing slightly different implementations of TCP. In the emulated scenario setting we used the predicted TCP variant passively inferred from the famous sawtooth pattern behavior of TCP's estimated cwnd computed based on the total number of outstanding bytes in flight [26, 27]. Hence, we strongly believe that passively observing the network-level characteristics found in TCP packets can give us more information about the remote computer's underlying OS. It can help us answer the question, "*are we able to accurately classify the underlying OS when different OSes are implementing the same TCP variant?*". We also believe coupling this with our previous works [26, 27] can help us explore the practical implications and long-term characteristics of TCP traffic. Besides we believe this will help us improve the classification of remote OSes form passive measurements since the values of some basic TCP/IP parameters are the same for multiple variations of OSes which may lead to inaccurate classifications of the underlying OSes. This is the core idea of our Paper VIII and to the best of our knowledge, this is the first study of fingerprinting OSes from passive measurements by applying RNN methods combining the basic TCP/IP features and the underlying TCP variant as input vectors. Paper VIII highlights questions and technical challenges of interest to direct future research, such as: What happens if an end-user (client) changes parameters that are the basis of OS fingerprinting? We believe challenges like this would make OS fingerprinting from passive measurements potentially hard. Hence, investigating these key challenges is one possibility we plan to address in our future work.

## 4.1 References

[1] M. H. Aghdam and P. Kabiri. Feature Selection for Intrusion Detection System Using Ant Colony Optimization. *IJ Network Security*, 18(3):420–432, 2016.

[2] T. Ahmed, B. Oreshkin, and M. Coates. Machine learning approaches to network anomaly detection. In *Proceedings of the 2nd USENIX workshop on Tackling computer systems problems with machine learning techniques*, pages 1–6. USENIX Association, 2007.

[3] J. Aikat, J. Kaur, F. D. Smith, and K. Jeffay. Variability in TCP round-trip times. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, pages 279–284. ACM, 2003.

[4] A. Aksoy and M. H. Gunes. Automated IoT Device Identification using Network Traffic. In *ICC 2019-2019 IEEE International Conference on Communications (ICC)*, pages 1–7. IEEE, 2019.

[5] A. Aksoy, S. Louis, and M. H. Gunes. Operating system fingerprinting via automated network traffic analysis. In *2017 IEEE Congress on Evolutionary Computation (CEC)*, pages 2502–2509. IEEE, 2017.

[6] S. Aljawarneh, M. Aldwairi, and M. B. Yassein. Anomaly-based intrusion detection system through feature selection analysis and building hybrid efficient model. *Journal of Computational Science*, 25:152–160, 2018.

[7] J. Allen, A. Christie, W. Fithen, J. McHugh, and J. Pickel. State of the practice of intrusion detection technologies. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 2000.

[8] M. A. Ambusaidi, X. He, P. Nanda, and Z. Tan. Building an intrusion detection system using a filter-based feature selection algorithm. *IEEE transactions on computers*, 65(10):2986–2998, 2016.

[9] S. Axelsson. Intrusion detection systems: A survey and taxonomy. Technical report, Technical report, 2000.

[10] P. Barford and D. Plonka. Characteristics of network traffic flow anomalies. In *Internet Measurement Workshop*, pages 69–73. Citeseer, 2001.

[11] H. Bojinov, Y. Michalevsky, G. Nakibly, and D. Boneh. Mobile device identification via sensor fingerprinting. *arXiv preprint arXiv:1408.1416*, 2014.

[12] S. Chebrolu, A. Abraham, and J. P. Thomas. Feature deduction and ensemble design of intrusion detection systems. *Computers & security*, 24(4):295–307, 2005.

[13] M. Crawford, T. M. Khoshgoftaar, J. D. Prusa, A. N. Richter, and H. Al Najada. Survey of review spam detection using machine learning techniques. *Journal of Big Data*, 2(1):23, 2015.

[14] A. Das, N. Borisov, and M. Caesar. Exploring ways to mitigate sensor-based smartphone fingerprinting. *arXiv preprint arXiv:1503.01874*, 2015.

[15] H. Debar, M. Dacier, and A. Wespi. Towards a taxonomy of intrusion-detection systems. *Computer Networks*, 31(8):805–822, 1999.

[16] D. E. Denning. An intrusion-detection model. *IEEE Transactions on software engineering*, (2):222–232, 1987.

[17] O. Depren, M. Topallar, E. Anarim, and M. K. Ciliz. An intelligent intrusion detection system (IDS) for anomaly and misuse detection in computer networks. *Expert systems with Applications*, 29(4):713–722, 2005.

[18] L. C. C. Desmond, C. C. Yuan, T. C. Pheng, and R. S. Lee. Identifying unique devices through wireless fingerprinting. In *Proceedings of the first ACM conference on Wireless network security*, pages 46–55. ACM, 2008.

[19] C. Elkan. Results of the KDD'99 classifier learning contest. In *Sponsored by the International Conference on Knowledge Discovery in Databases*, 1999.

[20] J. Franklin, D. McCoy, P. Tabriz, V. Neagoe, J. V. Randwyk, and D. Sicker. Passive Data Link Layer 802.11 Wireless Device Driver Fingerprinting. In *USENIX Security Symposium*, volume 3, pages 16–89, 2006.

[21] F. Gagnon and B. Esfandiari. A hybrid approach to operating system discovery based on diagnosis theory. In *2012 IEEE Network Operations and Management Symposium*, pages 860–865. IEEE, 2012.

[22] A. K. Ghosh, J. Wanken, and F. Charron. Detecting anomalous and unknown intrusions against programs. In *Proceedings 14th Annual Computer Security Applications Conference (Cat. No. 98EX217)*, pages 259–267. IEEE, 1998.

[23] L. G. Greenwald and T. J. Thomas. Toward Undetected Operating System Fingerprinting. *WOOT*, 7:1–10, 2007.

[24] A. Gurtov and R. Ludwig. Responding to spurious timeouts in TCP. In *IEEE INFOCOM 2003. Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE Cat. No. 03CH37428)*, volume 3, pages 2312–2322. IEEE, 2003.

[25] S. Ha, I. Rhee, and L. Xu. CUBIC: a new TCP-friendly high-speed TCP variant. *ACM SIGOPS operating systems review*, 42(5):64–74, 2008.

[26] D. H. Hagos, P. E. Engelstad, A. Yazidi, and Ø. Kure. A machine learning approach to TCP state monitoring from passive measurements. In *2018 Wireless Days (WD)*, pages 164–171. IEEE, 2018.

[27] D. H. Hagos, P. E. Engelstad, A. Yazidi, and Ø. Kure. Recurrent neural network-based prediction of tcp transmission states from passive measurements. In *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*, pages 1–10. IEEE, 2018.

[28] D. H. Hagos, A. Yazidi, Ø. Kure, and P. E. Engelstad. Enhancing security attacks analysis using regularized machine learning techniques. In *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*, pages 909–918. IEEE, 2017.

[29] F. Iglesias and T. Zseby. Analysis of network traffic features for anomaly detection. *Machine Learning*, 101(1-3):59–84, 2015.

[30] K. Ilgun, R. A. Kemmerer, and P. A. Porras. State transition analysis: A rule-based intrusion detection approach. *IEEE transactions on software engineering*, (3):181–199, 1995.

[31] ISCX. NSL-KDD Dataset. http://www.unb.ca/research/iscx/dataset/iscx-NSL-KDD-dataset.html, 2009.

[32] V. Jacobson. Congestion avoidance and control. In *ACM SIGCOMM computer communication review*, volume 18, pages 314–329. ACM, 1988.

[33] V. Jacobson. Congestion avoidance and control. *ACM SIGCOMM Computer Communication Review*, 25(1):157–187, 1995.

[34] S. Jaiswal, G. Iannaccone, C. Diot, J. Kurose, and D. Towsley. Inferring tcp connection characteristics through passive measurements. In *IEEE INFOCOM 2004*, volume 3, pages 1582–1592. IEEE, 2004.

[35] 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 Transactions on Networking (ToN)*, 15(1):54–66, 2007.

[36] H. Jiang and C. Dovrolis. Passive estimation of TCP round-trip times. *ACM SIGCOMM Computer Communication Review*, 32(3):75–88, 2002.

[37] T. Kohno, A. Broido, and K. C. Claffy. Remote physical device fingerprinting. *IEEE Transactions on Dependable and Secure Computing*, 2(2):93–108, 2005.

[38] P. Kolari, A. Java, T. Finin, T. Oates, A. Joshi, et al. Detecting spam blogs: A machine learning approach. In *Proceedings of the national conference on artificial intelligence*, volume 21, page 1351. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2006.

[39] A. Kurtz, H. Gascon, T. Becker, K. Rieck, and F. Freiling. Fingerprinting mobile devices using personalized configurations. *Proceedings on Privacy Enhancing Technologies*, 2016(1):4–19, 2016.

[40] H. Kvarnström. A survey of commercial tools for intrusion detection. *Goteborg, Sweden, Chalmers University of Technology*, 1999.

[41] T. Lane and C. E. Brodley. Temporal sequence learning and data reduction for anomaly detection. *ACM Transactions on Information and System Security (TISSEC)*, 2(3):295–331, 1999.

[42] M. Lastovicka and D. Filakovsky. Passive os fingerprinting prototype demonstration. In *NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium*, pages 1–2. IEEE, 2018.

[43] M. Lastovicka, T. Jirsik, P. Celeda, S. Spacek, and D. Filakovsky. Passive OS Fingerprinting Methods in the Jungle of Wireless Networks. In *NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium*, pages 1–9. IEEE, 2018.

[44] K. Lee, J. Caverlee, and S. Webb. Uncovering social spammers: social honeypots+ machine learning. In *Proceedings of the 33rd international ACM SIGIR conference on Research and development in information retrieval*, pages 435–442. ACM, 2010.

[45] I. Levin. KDD-99 classifier learning contest: LLSoft's results overview. *SIGKDD explorations*, 1(2):67–75, 2000.

[46] R. Lippmann, D. Fried, K. Piwowarski, and W. Streilein. Passive operating system identification from TCP/IP packet headers. In *Data Mining for Computer Security*. Citeseer, 2003.

[47] A. Loukili, A. L. Wijesinha, R. K. Karne, and A. K. Tsetse. TCP's Retransmission Timer and the Minimum RTO. In *2012 21st International Conference on Computer Communications and Networks (ICCCN)*, pages 1–5. IEEE, 2012.

[48] T. F. Lunt. A survey of intrusion detection techniques. *Computers & Security*, 12(4):405–418, 1993.

[49] G. F. Lyon. Remote OS detection via TCP/IP stack fingerprinting. *Phrack Magazine*, 8(54), 1998.

[50] G. F. Lyon. *Nmap network scanning: The official Nmap project guide to network discovery and security scanning*. Insecure, 2009.

[51] S. Maes, K. Tuyls, B. Vanschoenwinkel, and B. Manderick. Credit card fraud detection using Bayesian and neural networks. In *Proceedings of the 1st international naiso congress on neuro fuzzy technologies*, pages 261–270, 2002.

[52] T. M. Mitchell. Machine learning and data mining. *Communications of the ACM*, 42(11), 1999.

[53] A. Ornaghi and M. Valleri. Ettercap. https://www.ettercap-project.org/, 2015.

[54] J. Oshio, S. Ata, and I. Oka. Identification of different TCP versions based on cluster analysis. In *2009 Proceedings of 18th International Conference on Computer Communications and Networks*, pages 1–6. IEEE, 2009.

[55] J. Pahdye and S. Floyd. On inferring TCP behavior. *ACM SIGCOMM Computer Comm. Review*, 31(4):287–298, 2001.

[56] P. Pantel, D. Lin, et al. Spamcop: A spam classification & organization program. In *Proceedings of AAAI-98 Workshop on Learning for Text Categorization*, pages 95–98, 1998.

[57] C. Paxson, M. Allman, J. Chu, and M. Sargent. Computing TCP's Retransmission Timer (RFC 6298), 2011.

[58] V. Paxson. Automated packet trace analysis of TCP implementations. *ACM SIGCOMM Computer Communication Review*, 27(4):167–179, 1997.

[59] V. Paxson. Bro: a system for detecting network intruders in real-time. *Computer networks*, 31(23-24):2435–2463, 1999.

[60] K. Ramakrishnan and R. Jain. A binary feedback scheme for congestion avoidance in computer networks. *ACM Transactions on Computer Systems (TOCS)*, 8(2):158–181, 1990.

[61] S. Rewaskar, J. Kaur, and D. Smith. A Passive State-Machine Based Approach for Reliable Estimation of TCP Losses. 2006.

[62] M. Sabhnani and G. Serpen. Why machine learning algorithms fail in misuse detection on KDD intrusion detection data set. *Intelligent Data Analysis*, 8(4):403–415, 2004.

[63] M. M. Sebring. Expert systems in intrusion detection: A case study. In *Proc. 11th National Computer Security Conference, Baltimore, Maryland, Oct. 1988*, pages 74–81, 1988.

[64] S. A. R. Shah and B. Issac. Performance comparison of intrusion detection systems and application of machine learning to Snort system. *Future Generation Computer Systems*, 80:157–170, 2018.

[65] S. D. Shanklin, T. E. Bernhard, and G. S. Lathem. Intrusion detection signature analysis using regular expressions and logical operators, 2002. US Patent 6,487,666.

[66] S. Snapp, B. Mukherjee, and K. Levitt. Detecting intrusions through attack signature analysis. Technical report, Lawrence Livermore National Lab., CA (United States), 1991.

[67] R. Sommer and V. Paxson. Outside the closed world: On using machine learning for network intrusion detection. In *2010 IEEE symposium on security and privacy*, pages 305–316. IEEE, 2010.

[68] R. Spangler. Analysis of remote active operating system fingerprinting tools. *University of Wisconsin*, 2003.

[69] G. Taleck. Synscan: Towards complete tcp/ip fingerprinting. *CanSecWest, Vancouver BC, Canada*, pages 1–12, 2004.

[70] H. S. Teng and K. Chen. Adaptive real-time anomaly detection using inductively generated sequential patterns. page 278. IEEE, 1990.

[71] J. R. Vacca. *Computer and information security handbook*. Newnes, 2012.

[72] L. Xu, K. Harfoush, and I. Rhee. Binary increase congestion control (BIC) for fast long-distance networks. In *IEEE INFOCOM*, volume 4, pages 2514–2524. IEEE, 2004.

[73] F. Yarochkin and O. Arkin. Xprobe2- A'Fuzzy'Approach to Remote Active Operating System Fingerprinting, 2002.

[74] M. Zalewski. p0f: Passive OS fingerprinting tool. *Online at http://lcamtuf.coredump.cx/p0f3*, 2017.

[75] B. Zhang, T. Zou, Y. Wang, and B. Zhang. Remote operation system detection base on machine learning. In *2009 Fourth International Conference on Frontier of Computer Science and Technology*, pages 539–542. IEEE, 2009.

Chapter 5

# Conclusions

Modern machine learning and deep learning approaches have advanced the state-of-the-art for many research domain problems in the networking research community in different contexts, e.g., security monitoring, analyzing the complexity of networks, network traffic engineering, etc. In this dissertation, we catalogue that machine learning and deep learning-based techniques can provide a potentially promising methodology for improving the accuracy of predicting the internal states of TCP through passive traffic measurements. To demonstrate the suitability of our approaches, we highlighted the benefits of passively discovering the characteristics of TCP transmission states related to network congestion from different perspectives. Although there are already some legacy works that aspire to achieve the same objective, they are mainly based on engineered rules that require a deep understanding of the complexity of TCP protocols. The main advantage of using machine learning and deep learning-based techniques is that we do not need a domain expert knowledge about the TCP protocol. The work presented in this dissertation aims at obtaining detailed knowledge about the end hosts by passively monitoring and analyzing the TCP traffic using machine learning and deep learning-based techniques. Even though this is the main objective of the dissertation, our work shows that related techniques can also be used to find other important information about the hosts, such as their TCP implementation or in a security perspective classify if the host's traffic is malicious or not. In addition to this, our work also shows that these techniques can be extended to passively fingerprint the operating system of the end host.

The work presented in this dissertation advances the state-of-the-art for passive operating system fingerprinting problem one step further by combining the common TCP/IP features that are the basis of passive operating system fingerprinting with the underlying predicted TCP variant as a distinguishing input feature. We argue that passively observing the network-level characteristics found in TCP packets can give us more information about the remote computer's operating system. We further argue that this helps us to explore in detail the long-term characteristics of TCP traffic. In terms of accuracy, we empirically demonstrate that accurately inferring the TCP variant has a great potential to significantly boost the fingerprint performance across different types of traffic sources and evaluation settings.

For all our evaluations, we demonstrate the effectiveness of our proposed prediction models with reasonably good accuracies across different scenarios and multiple TCP variants. We also illustrated that the learned prediction models generalize well by leveraging knowledge from the emulated network and perform reasonably well when it is applied in a realistic scenario setting bearing similarity to the concept of transfer learning in the machine learning community.

Finally, we believe that our work presented in this dissertation opens new doors in the discovery of the dynamic complexity of TCP from passive measurements by leveraging advanced machine learning and deep learning-based techniques. We, further, believe that the study and results presented in this dissertation will be potentially useful to network operators, 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 from passive measurements.

## 5.1 Future Research Directions

At this juncture, we discuss the potential future directions exploiting the approaches we describe as well as encouraging more research to bring new advanced methods to bear for the different use cases presented in this dissertation.

- When it comes to the first use case presented in this dissertation, there are many research avenues that can be explored. For example, designing a general approach based on machine learning and deep learning techniques that are able to predict if a TCP packet loss is due to buffer overflow in routers or a wireless link in which two of them have different characteristics. Historically, TCP was designed for buffer overflow in routers and the TCP back-off action 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. It would, therefore, be interesting to investigate this as future research work.
- To infer the transmission states of TCP client from passive traffic measurements, we capture the traffic in an intermediate node. But what happens if we capture the traffic at both the sender and receiver endpoints and do the estimation of the internal states of TCP separately? There isn't a timing difference for predicting the underlying TCP variant since it simply measures the change in cwnd size and hence this will not complicate the inference of the cwnd. However, since there will be a difference in the timing of the received data, measuring at both endpoints will affect the passive Round-trip Time (RTT) estimation. It means this, to get a good measure of the raw one-way RTT for each direction, would require clock synchronization between the sender and receiver endpoints. We have no way to combine these two clocks with any strong guarantees unless the two endpoints are reasonably synchronized (e.g., by using GPS signals). Assessing this practical challenge is left for future work.
- The second use case of this dissertation focuses particularly on the four security attack classes namely Denial-of-Service (DoS), User-to-Root (U2R),

Remote-to-Local (R2L) and probe used in the labeled NSL-KDD public dataset. Even though the NSL-KDD benchmark dataset for intrusion detection we use in our analysis may not be an ideal representative of existing realistic networks, it does not suffer from any of the mentioned limitations in the other old public intrusion detection datasets. Because of the lack of public intrusion detection datasets in the networking research community, we believe it can be applied as an effective benchmark dataset for the general problem of network security analysis to help networking researchers work with different machine learning, deep learning and other sophisticated statistical techniques to perform intrusion detection. It is worth mentioning that generating a similarly labeled intrusion dataset of real-time network traffic with different classes of modern security attack distribution as in the NSL-KDD dataset is costly and significantly challenging. Besides, to the best of our knowledge, there is no thorough study that focuses on the investigation of the most important features in identifying well-known network security attacks. Hence, a deep investigation of a practical solution to this important problem and validating the findings presented in Paper VII using more recent realistic network traffic is one potential area for future work.

- For our emulated experiments of the TCP state monitoring, we employ a software-based emulator with great care in an extremely well-contained virtual environment for all the variations of bandwidth, delay, jitter, and packet loss parameters. However, as the precision of the emulator, given that a software emulator is not precise, cannot be measured from TCP streams, we set up 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. We 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. Hence, extending our emulated experiments using a hardware-based emulator would be a recommended focus for future work.
- In Paper VI, we present an effective TCP variant identification methodology from traffic measured passively by utilizing $\beta$, the multiplicative back-off factor to decrease the *cwnd* on a loss event, and the queueing delay values. By design, unlike loss-based algorithms, the multiplicative decrease parameter of delay-based congestion control algorithms is not fixed which makes it fundamentally challenging to predict the TCP variant from passive traffic when there is variability in delay. As future work, it would also be interesting to further develop a delay-based model using advanced methods from the field of Artificial Intelligence (AI) 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 passively measured traffic and real measurements over the Internet.

- As presented in the third use case of this dissertation, passively detecting the TCP variant is a challenging task which led to a two-step approach, where the TCP variant prediction of a deep learning-based universal tool is used as input to another machine learning method in the next step. However, by integrating the two machine learning approaches better, there should be potential for increasing the classification performance even further and get even closer to the idealistic results of using an Oracle presented in Paper VIII. Exploring such optimizations is left for future work.

- In the evaluation methods presented in Paper VIII, we adopted a multi-stage approach to passive OS fingerprinting where in the first stage, the cwnd behavior from the outstanding bytes in flight is used to predict the TCP variant. The predicted TCP variant is finally used as an input feature to the passive Operating Systems (OSes) fingerprinting process. We believe this approach requires more training time and it is computationally inefficient. To this end, dynamically addressing this problem with a one-stage approach guided by taking the whole TCP/IP header data as an input vector is left out for further future work.

- It is known that TCP clock drift improves OS fingerprinting and hence measuring differences in the timing of how the IP stack works may allow us to predict the underlying OS with greater assurance in terms of accuracy. We, therefore, argue for using other TCP options like timestamps and queueing delay characteristics as an input feature vector for passive OSes fingerprinting model as another interesting future work direction.

- For the passive OS fingerprinting experiments, we passively collected our realistic dataset from TCP traffic originated from the internal network of our university and destined to various hosts on the Internet. First, we passively collected data for fixed (non-mobile) desktop computers (typically using OSes like Windows, Linux, Unix, Mac OSx, etc.) by using an intermediate node. Then, we passively collected the data that covered mobile devices, like *android* and *iOS*. Our real traffic covers the communication to and from our university and hence all traffic whose source and destination IP addresses are within the subnets of our internal network. Hence the network administrator of our university has full control over the internal machines with real IP addresses that are not going to a NAT gateway, and therefore it is fairly possible to tell whether it is a laptop or a desktop PC by looking it up in the internal database owned by the university. However, since it is a dynamic network we do not have full control over external machines, because they can be anything behind an IP address that changes dynamically. This is because there is an endless number of machines spoofing scanning the network and they can appear as Linux-powered OSes but they could be Windows and vice versa and this happens because the user may have strongly tuned the TCP stack to look like something else. It is pretty hard to certainly say anything about the external computers because the communication can go through a NAT gateway possessing another OS type. For example, if a user is connected to a student wireless network, there is a chance that it may go to a Linux NAT gateway and hence from outside the user is seen as Linux NAT which makes it hard to predict whether the underlying OS is Linux, Mac or Windows. Therefore, fingerprinting devices behind NAT technology on a distributed network where a number of devices can hide behind a NAT is another critical challenge. It is, therefore, worth noting that establishing ground truth in dynamic networks at a larger scale remains a challenging problem. Further investigation and implementation to explore these difficulties is one possible future research direction.

- Finally, in addition to the difficulties of establishing ground truth at a larger scale on a dynamic network, there is a lot of other work to be done as an extension of our work presented in Paper VIII. For example, addressing answers to valid questions like: *What happens if an end-user (client) changes default parameters that are the basis of OS fingerprinting?* is one possibility for our future work. We expect that end-users don't change parameters often, while servers may do so if it helps improve performance. We believe this would make passive OS fingerprinting potentially hard and hence a further future work addressing these challenges using different advanced approaches is promising.

# Part II - Included Papers

Paper I

# A Machine Learning Approach to TCP State Monitoring from Passive Measurements

**Desta Haileselassie Hagos**[1], **Paal E. Engelstad**, **Anis Yazidi**, **Øivind Kure**

***Published*** in *IEEE Wireless Days (WD)*, 2018. IEEE, 2018.
DOI: 10.1109/WD.2018.8361713.

## 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 in use, and each variant uses a specific end-to-end 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. We demonstrate how the intermediate node can predict the *Congestion Window (cwnd)* size of the TCP client. The method can also be extended to predict other TCP transmission states of the client. We use a generic machine learning-based prediction approach for inferring *cwnd* within a flow from a passive traffic collected at an intermediate node. Our experimental results indicate the effectiveness of our prediction model with reasonably good accuracy across different scenarios and multiple TCP variants.

[1]University of Oslo, Department of Informatics, destahh@ifi.uio.no

**Keywords**: Network Protocols, TCP, Congestion Control, Passive Measurement, Machine Learning

## I.1 Introduction

Machine learning techniques have effectively advanced the state-of-the-art for many research domain problems in the computer networking community. For example, they are being applied in the areas of traffic classification [15], security monitoring and Intrusion Detection Systems (IDS) [8, 22], 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 to 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 [10] 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 by avoiding 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]. One of the main 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 about inferring these functionalities that determine a network condition from a passive traffic collected at an intermediate node of a network without having access to the sender?* This is the question we will explore and attempt to answer in our paper.

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 I.3. The work reported in [11] 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 Round-trip Time (RTT) and estimation of incorrect window sizes [11]. 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.

In moving towards a *generic* approach, we believe there is very little work on a *scalable* method of predicting the *cwnd* 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 [21]. Hence, compared to these previous studies, in this paper, we explore machine learning-based approaches 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. Our prediction model handles more scenarios and it can also work with different variants of TCP. 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 networking community who want to assess the characteristics of TCP states related to network congestion from passive measurements.

## I.2 Motivation

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, taking the nature of TCP, accurately inferring *cwnd* and its characteristics from passive traffic is a difficult problem. One of the difficulties 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). 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* [12]. 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, so the receiver did not yet know that the packets were received before they decided to retransmit them. More on the location of the passive monitor and its effect on what we can infer from the measurements is found in [12]. In this paper, we argue that machine learning-based approaches can give a better prediction accuracy of TCP sender connection states from passive measurements by addressing the aforementioned practical challenges.

The rest of the paper is organized as follows: In Section I.3, we review and give a detailed overview of the *state-of-the-art* and discuss closely related works on TCP variants research. In Section I.4, we describe our experimental setup for the evaluation and the assumptions we made during our experiment. Section I.5 gives an overview of our methodology highlighting the machine learning techniques and performance measure metrics used in our paper. Section I.6 presents detailed experimental results and evaluation of the emulated network for *cwnd* prediction. In Section I.7, we present a realistic setup which validates our prediction model with other scenarios. Finally, Section I.8 concludes the paper and outlines our future directions of research.

## I.3 Related Work

This section discusses closely related studies on monitoring network traffic techniques and the characteristics of TCP congestion control algorithms. The techniques to monitor TCP per-connection characteristics are divided into *two* categories: *active measurement* and *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.

### I.3.1 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 [14, 17].

### I.3.2 Passive Measurement

In this technique, passively collected packet traces are examined to measure TCP behaviors of Internet flows [11, 18, 19]. Passive 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. In the traditional methods of passive measurement, there has been much interest in the investigation of TCP connections aggregate properties and its characteristics on the global Internet.

Starting with [10, 20], TCP congestion control has been an active area of research in the networking community. A work of interest that is most closely related to our work is [11] 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 [11] 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* (FSM). 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* [11] 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 [11], the work in [12], 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 [11].

The authors of the study [21] developed a tool called *tcpflows* that attempts to *passively* estimate the value of *cwnd* 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 *cwnd* characteristics of newly deployed TCP variants. Our main goal in this study is more fundamental to develop a machine learning based methodology for predicting *cwnd* of all *loss-based* TCP variants by examining each cross-traffic of TCP flows of the endpoints passively collected at an intermediate node.

## I.4 Experimental Setup and Datasets

In this section, we provide an overview of our experimental testbed that has been designed and implemented in order to investigate the problem we are addressing in this work.

### I.4.1 Experimental Testbed

Figure I.1 shows the experimental setup that we use for all of our experiments in this paper. 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 [6] so as to create a congestion. 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* kernel release. The reason why we chose Linux is because we wanted to track the system-wide TCP *state* of every packet that is sent and received from the *kernel*. We carried out the experiment by capturing all sessions on the network when the client and server are sending TCP packets. 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 I.1 is a separate Linux machine acting as a proxy. We made sure its receiver window is much bigger than the *receiver window* of both the sender and the receiver in order to hinder the proxy from

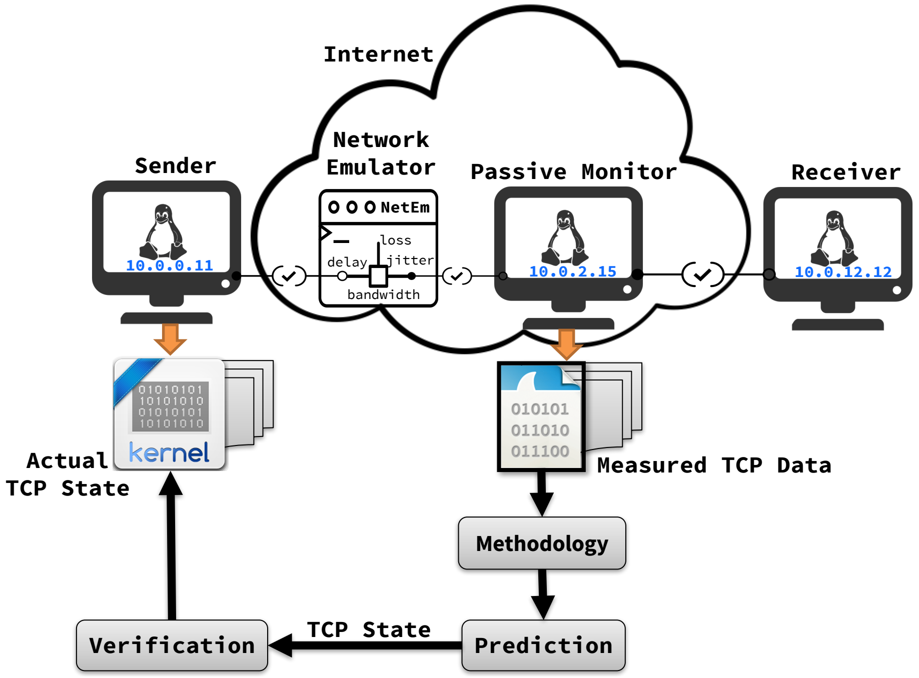


Figure I.1: Experimental Setup.

influencing on the network traffic apart from measuring all the TCP traffic sessions from the sender. 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. As it is shown in Figure I.1, we used a database to match and join the *measured* TCP data as an input to our methodology for a prediction of the TCP per-connection states. Finally, we verified the predicted TCP states with the actual TCP kernel states directly logged from the Linux kernel used only for training and generate a new data for the learning model to predict on. Once we finish with the verification, we run our learning model and get the predictions.

### Testbed hardware

We have performed our 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.

### I.4.2 Network Emulation

TCP congestion control is set to operate on the variability of bandwidth, different cross-traffic, RTT, etc. In order to create a realistic scenario, we have emulated the network in our setup as it is shown in Figure I.1 by adding an end-to-end variability within a flow to the important parameters shown in Table I.1. For the network emulation, we used the popular Linux-based network emulator, *Network Emulator (NetEm)* [9] on a separate node, that supports an end-to-end variability of *bandwidth*, *delay*, *jitter*, *packet loss*, *duplication*, *packet corruption* and more other parameters which the TCP *cwnd* is influenced by.

### I.4.3 Verification of the Emulator

Given that the software emulator is not precise, can we trust the network emulator for all the variations of *bandwidth*, *delay*, *jitter* and *packet loss* parameters that we change as shown in Table I.1 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* [9], 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 I.1. 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.

### I.4.4 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* [9] 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* [9] (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* [9] with variations in the data rate and the parameters presented in Table I.1 between the client and the server and we found out that each variation run by *NetEm* [9] 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.

### I.4.5 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 I.1. Therefore, In all of our experiments, each TCP flow uses *1500-byte* data packets and an advertised window set by the operating system.

### I.4.6 Network Emulation Parameters

The data traces for all our experiments are generated using the *iperf* [6] traffic generator on an emulated LAN link where we run each TCP variant with an end-to-end variation of the parameters *bandwidth*, *delay*, *jitter* and *packet loss* as shown below in Table I.1 where the *cwnd* is highly influenced by.

Table I.1: Network Emulation Parameters.

| | **Bandwidth** (*Mbit/s*) | **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 |

### I.4.7 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* [16] when there is an end-to-end variability of *bandwidth*, *delay*, *loss* and *RTT* is a better approach to estimate the *cwnd* and how fast the recovery is. Firstly, since we are estimating *cwnd* from *bytes_in_flight*, we have also considered that *cwnd* must be the limiting factor for the TCP sender. Secondly, we assume that we don't know what TCP variant is running in the network and the per-connection state within the variant. Lastly, 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.

## I.5 Methodology

In this section, we describe our methodology for experimentally inferring the *cwnd* from passive measurements.

In order to create the input data for the machine learning algorithm, *loss-based* TCP congestion control algorithms are used by emulating a background traffic using the end-to-end emulation of the parameters shown in Table I.1.

### I.5.1 Passive Monitoring of bytes_in_flight

The passive traffic (i.e., measured TCP data) collected at the intermediate node as shown in Figure I.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

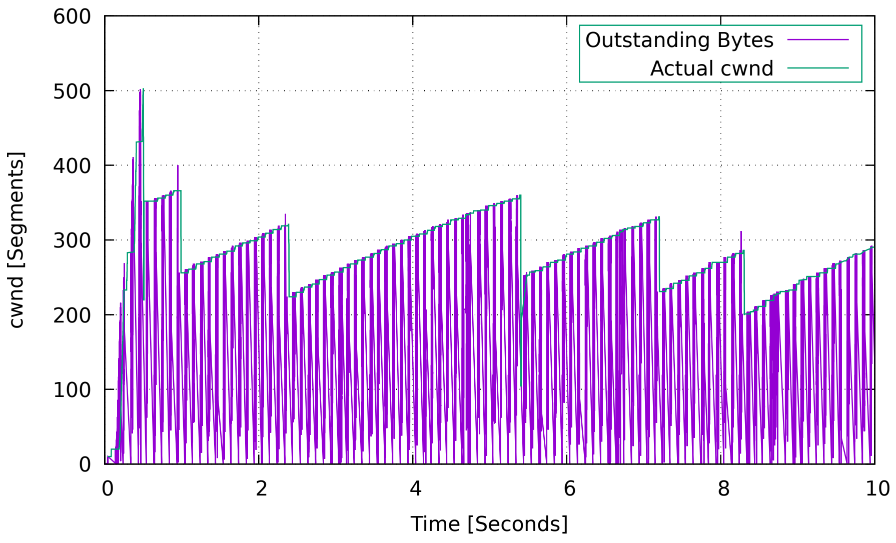


Figure I.2: *Outstanding bytes* calculated from the intermediate monitor using [16] before applying *convolutional* filtering vs. the actual *cwnd* from the sender.

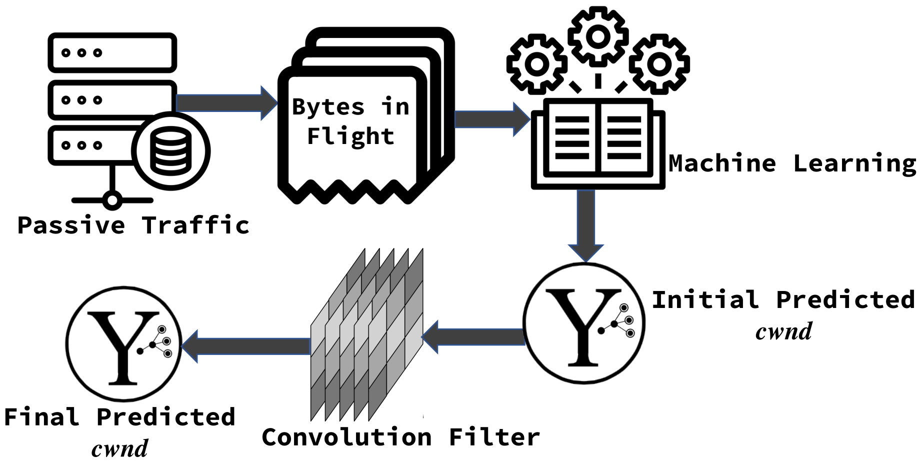


Figure I.3: Methodology for *cwnd* prediction.

TCP sender's sending rate by employing the *cwnd* that limits the number of cumulatively unacknowledged bytes that are allowed at any given time. 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* [16]. 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 I.3. 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 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 the number of *bytes_in_flight* 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.

### I.5.2 Prediction 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 [10] 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 that turns to be an insufficient detail for an accurate prediction.

We created an *ensemble* machine learning prediction model in *Python* where we apply a *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. Figure I.2 shows the comparison between the number of *outstanding bytes* from the intermediate node before running the *ensemble* model and applying the *filtering* techniques versus the actual *cwnd* tracked from the kernel of the sender-side. In order to further improve the performance of our *ensemble* prediction, we tuned the *Random Forest Regressor* optimal hyperparameters shown in Table I.2 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 optimal *cwnd* prediction model by minimizing the prediction function, we have also used *Gradient Boosting* algorithm [3]. 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.

Table I.2: Tuning parameters of the ensemble methods.

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

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 testbed shown in Figure I.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 I.4.

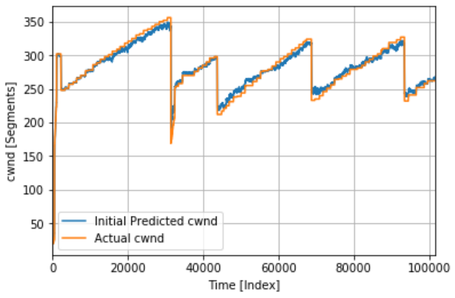


Figure I.4: Initial prediction of TCP *cwnd* versus the actual *cwnd* before applying the convolutional filtering technique.

As it is shown in Table I.3, 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 I.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 [5].

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

### I.5.3 Convolutional Filtering

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 I.4 and produce the *final* predicted value of *cwnd* shown in Figure I.5 as per the methodology depicted in Figure I.3. *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 I.2, $g$ is the filter which in our case is the *final* predicted *cwnd* as shown in Figure I.5.

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

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 I.2 and correctly recognize the patterns of the *cwnd* curves. As it is shown in Figure I.5, 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 I.3.

## I.6 Experimental Results

In this section, we present the experimental results of the emulated network for a TCP *cwnd* prediction of the sender. In Figure I.5, 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. Our methodology for inferring the TCP *cwnd* is shown in Figure I.3. We have experimented with several variations (36 configurations for each TCP variant). Due to space limitation in this paper, we will not present all the evaluation plots for all configurations as per Table I.1 and hence the results reported in this paper are for some of the selected configurations as shown in Figure I.5 to verify the accuracy of our machine learning-based prediction model.

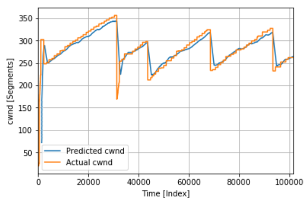
We evaluate our *final* TCP *cwnd* prediction model under different configurations of training and testing sample size ratios and the performance results are presented in Table I.3. As it is shown in Figure I.5, we found out the *convolutional* filtering we built for predicting *cwnd* captures the ratio of the *cwnd* drop very accurately. Figures I.5*(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 I.5*(c)* and *(d)*, Figure I.5*(c)* has a *Bandwidth-Delay Product (BDP)* [13] 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 I.5. In the results, 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.

Table I.3: TCP *final* predicted *cwnd* performance results of an *emulated network* with different configurations.

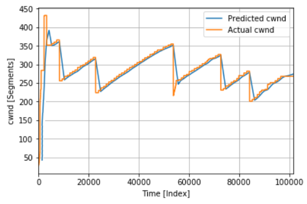
| TCP Algorithms | Configurations | RMSE | MAPE (%) |
|---|---|---|---|
| CUBIC | *Final* predicted *cwnd* - $C_1$ | 5.839 | 6.953% |
| | *Final* predicted *cwnd* - $C_2$ | 3.075 | 3.725% |
| | *Final* predicted *cwnd* - $C_3$ | 2.209 | 2.857 % |
| | *Final* predicted *cwnd* - $C_4$ | 1.947 | 3.002%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

## I.7 Realistic scenario setup

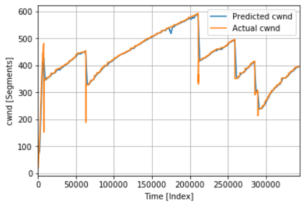
In order to further validate our results presented in Section I.6 against other scenarios, we believe it is necessary to carefully test how well our machine learning-based prediction model presented above using an emulated network works by conducting a series of controlled experiments in a realistic scenario setting. This helps us to justify and guarantee how our model could predict the



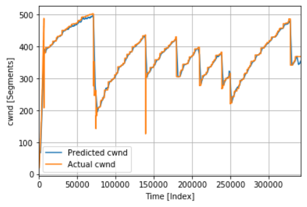
(a) CUBIC *final* predicted *cwnd* - Configuration $C_1$



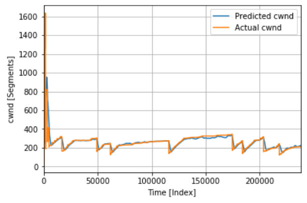
(b) CUBIC *final* predicted *cwnd* - Configuration $C_2$



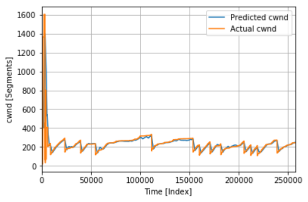
(c) CUBIC *final* predicted *cwnd* - Configuration $C_3$



(d) CUBIC *final* predicted *cwnd* - Configuration $C_4$

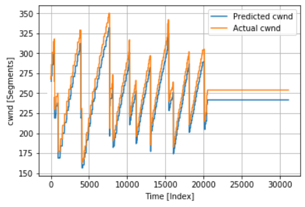


(e) Reno *final* predicted *cwnd* - Configuration $C_1$

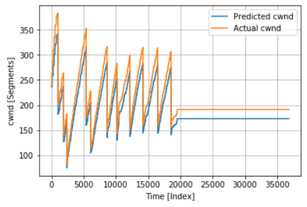


(f) Reno *final* predicted *cwnd* - Configuration $C_2$

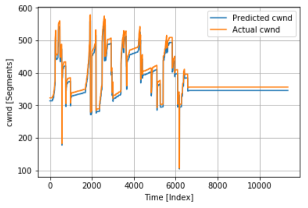
Figure I.5: Final TCP *cwnd* prediction with different configurations of network emulation parameters for TCP CUBIC [7] and TCP Reno [10] after optimizing the *initial cwnd* prediction accuracy with *convolution* filtering technique in an emulated network.



(a) CUBIC *final* predicted *cwnd*, USA site



(b) Reno *final* predicted *cwnd*, USA site



(c) BIC *final* predicted *cwnd*, USA site

Figure I.6: TCP *cwnd* prediction versus actual *cwnd* of TCP CUBIC [7], TCP BIC [23] and TCP Reno [10] from a realistic scenario on *Google Cloud* platform (East coast USA (North Carolina) site)

development of a *cwnd* pattern and the TCP variant used with other realistic network traffic scenarios captured from the Internet. To this end, we created a realistic testbed where we experiment from *Google Cloud* platform nodes by running our resources on the East coast of USA. In order to create a realistic TCP session, we uploaded an *Ubuntu* image to *Google Cloud* platform site so that we have a full control of the underlying TCP variant and at the same time run a *tcpdump* in the background and capture the traffic 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.* As it is shown in Figure I.7, we found out that our model could be performing very well with small prediction errors for realistic scenario settings too. The prediction performance evaluation result of the *final* predicted *cwnd* in the realistic scenario setting is presented in Table I.4.

Table I.4: TCP *final* predicted *cwnd* performance results of a *realistic scenario* setting.

| **TCP Algorithms** | **Google Cloud Platform** | **RMSE** | **MAPE (%)** |
|---|---|---|---|
| CUBIC | USA site | 4.265 | 5.134% |
| Reno | *USA* site | 3.170 | 5.068% |
| BIC | *USA* site | 2.952 | 3.809% |

## I.8 Conclusion and Future Work

In this paper, we have explored machine learning-based techniques to monitor TCP per-connection states of *loss-based* TCP variants from passive measurements when there is variability within a flow. Our paper presents a machine learning-based prediction model for experimentally inferring TCP *cwnd* of the sender by examining each cross-traffic of TCP flows of the endpoints passively collected at an intermediate node. Our measurement results show that we get a very good accuracy for both the *increasing* and *decreasing* portion of the sawtooth pattern across more scenarios and different TCP variants. 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 variations of *bandwidth*, *delay*, *jitter* and *packet loss* into one robust and generic learning model. In order to guarantee an optimal prediction model by minimizing the prediction function, we have also utilized *Gradient Boosting* algorithm. Our performance study shows that the prediction model gives a very good performance on all the metrics both in the emulated and realistic scenario settings.

As a future work, there are many research avenues that can be explored. First, since now we are able to predict the *cwnd*, we also think that we will be able to infer other TCP states, for example, predicting the underlying TCP variants of *loss-based* congestion control algorithms. Second, it would be interesting to develop a *delay-based* method so as to verify how delay changes and look into how the TCP variants of *delay-based* congestion control algorithms can be predicted from a passively measured traffic. 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 buffer overflow in routers or a 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 issues in our future work.

**Acknowledgements.** We gratefully acknowledge the reviewers for their helpful feedback and detailed comments. We would like to thank the Research Infrastructure Services Group at the University of Oslo for the use of multicore cluster machines.

## I.9 References

[1] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. *TCP Vegas: New techniques for congestion detection and avoidance*, volume 24. ACM, 1994.

[2] L. Breiman. Random forests. *Machine learning*, 45(1), 2001.

[3] P. Bühlmann, T. Hothorn, et al. Boosting algorithms: Regularization, prediction and model fitting. *Statistical Science*, 22(4):477–505, 2007.

[4] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, et al. BBR: congestion-based congestion control. *ACM*, 2017.

[5] A. De Myttenaere, B. Golden, B. Le Grand, and F. Rossi. Mean absolute percentage error for regression models. *Neurocomputing*, 192:38–48, 2016.

[6] ESnet. iperf3. https://iperf.fr/iperf-servers.php, 2017.

[7] S. Ha, I. Rhee, and L. Xu. CUBIC: a new TCP-friendly high-speed TCP variant. *ACM SIGOPS operating systems review*, 42(5):64–74, 2008.

[8] D. H. Hagos, A. Yazidi, Ø. Kure, and P. E. Engelstad. Enhancing security attacks analysis using regularized machine learning techniques. In *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*, pages 909–918. IEEE, 2017.

[9] S. Hemminger et al. Network emulation with NetEm. 2005.

[10] V. Jacobson. Congestion avoidance and control. In *ACM SIGCOMM computer communication review*, volume 18, pages 314–329. ACM, 1988.

[11] S. Jaiswal, G. Iannaccone, C. Diot, J. Kurose, and D. Towsley. Inferring tcp connection characteristics through passive measurements. In *IEEE INFOCOM 2004*, volume 3, pages 1582–1592. IEEE, 2004.

[12] 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 Transactions on Networking (ToN)*, 15(1):54–66, 2007.

[13] D. Katabi, M. Handley, and C. Rohrs. Congestion control for high bandwidth-delay product networks. *ACM SIGCOMM computer communication review*, 32(4):89–102, 2002.

[14] A. Medina, M. Allman, and S. Floyd. Measuring the evolution of transport protocols in the internet. *ACM SIGCOMM Computer Communication Review*, 35(2):37–52, 2005.

[15] T. T. Nguyen and G. J. Armitage. A survey of techniques for internet traffic classification using machine learning. *IEEE Communications Surveys and Tutorials*, 10(1-4):56–76, 2008.

[16] S. Ostermann. Tcptrace. http://www. tcptrace. org, 2000.

[17] J. Pahdye and S. Floyd. On inferring TCP behavior. *ACM SIGCOMM Computer Comm. Review*, 31(4):287–298, 2001.

[18] V. Paxson. Automated packet trace analysis of TCP implementations. *ACM SIGCOMM Computer Communication Review*, 27(4):167–179, 1997.

[19] F. Qian, A. Gerber, Z. M. Mao, S. Sen, O. Spatscheck, and W. Willinger. TCP revisited: a fresh look at TCP in the wild. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement*, pages 76–89. ACM, 2009.

[20] K. Ramakrishnan and R. Jain. A binary feedback scheme for congestion avoidance in computer networks. *ACM Transactions on Computer Systems (TOCS)*, 8(2):158–181, 1990.

[21] S. Rewaskar, J. Kaur, and D. Smith. A Passive State-Machine Based Approach for Reliable Estimation of TCP Losses. 2006.

[22] R. Sommer and V. Paxson. Outside the closed world: On using machine learning for network intrusion detection. In *2010 IEEE symposium on security and privacy*, pages 305–316. IEEE, 2010.

[23] L. Xu, K. Harfoush, and I. Rhee. Binary increase congestion control (BIC) for fast long-distance networks. In *IEEE INFOCOM*, volume 4, pages 2514–2524. IEEE, 2004.

Paper II

# Towards a Robust and Scalable TCP Flavors Prediction Model from Passive Traffic

**Desta Haileselassie Hagos**[1], **Paal E. Engelstad, Anis Yazidi, Øivind Kure**

***Published*** in the *27th International Conference on Computer Communication and Networks (ICCCN 2018)*, pp. 1–11. IEEE, 2018.
DOI: 10.1109/ICCCN.2018.8487396.

## Abstract

Different end-to-end Transmission Control Protocol (TCP) algorithms widely in use behave differently under network congestion. The TCP congestion control itself has grown increasingly complex which in practice makes predicting TCP per-connection states from passive measurements a challenging task. In this paper, we present a robust, scalable and generic machine learning-based model which may be of interest for network operators that experimentally infers the underlying variant of *loss-based* TCP algorithms within a flow from passive traffic measurements collected at an intermediate node. We believe that our study has also 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 several 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 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 variants demonstrate that our model is effective and has considerable potential.

[1]University of Oslo, Department of Informatics, destahh@ifi.uio.no

**Keywords**: Network Protocols, TCP, Congestion Control, Passive Measurement, Machine Learning, Deep Learning

## II.1 Introduction

Congestion control is a fundamental problem in computer networks. The TCP congestion control algorithms that are widely deployed today perform the most important functionalities related to congestion control such as handling the *cwnd* from the sender-side. In this paper, we investigate and explore questions quantitatively as they apply to problems of network congestion that include: *(i)* 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? *(ii)* How can we track the underlying TCP variant that the TCP client is using from passive measurements? *(iii)* What percentage of network users are using either a *loss-based* or *delay-based* TCP variants? *(iv)* Which user is responsible for the majority of heavy flow traffic in the network? *(iv)* 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.

Much of the Internet's traffic is carried using the dominant reliable end-to-end TCP protocol [14] 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 by avoiding sending more data that the underlying network is capable of transmitting which is maintained by the sender's *Congestion Window (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 [3]. One of the main parameters for TCP performance evaluation in a real-world setting is *cwnd*. Numerous existing research works on this problem that has been proposed rely on an *active* approach to measure the characteristics of TCP. In this paper, we focus mainly on the *passive* measurement approach[2].

A wide variety of approaches have been applied to the problem of congestion control characteristics. The work in [15] presented an approach to estimate TCP parameters at the sender-side based on packets captured at the monitoring point using a finite state machine. 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 [15]. Another limitation of this work, given the many existing variants of TCP, the use of a separate *state machine* for each TCP variant is *unscalable* and we also believe that the constructed *replica* may 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

[2]The difference between *active* and *passive* measurement techniques will be explained later in detail in Section II.4.

(e.g., minimum RTO, timer granularity, duplicate ACK thresholds, etc.) can also behave so differently [27]. 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 [27]. The authors in [27] addressed this problem by developing a separate state machine for each of the operating system vendors. The problem with this technique [27] 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.

Machine learning techniques, as they play important roles in other areas of research, are potentially useful in many areas within the computer networking research community. For example, *intrusion detection analysis and prediction* [12], *network scheduling* [18], *traffic classification* [20, 31], etc. However, 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. Hence, In this paper, 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 experimental results demonstrate the feasibility of our prediction model. We believe that our study will be potentially useful to network operators, 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 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 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. [4, 24, 28]. 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 several controlled experiments and experimentally verified across an *emulated*, *realistic* and *combined* scenario settings.

## II.2 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 [7]. 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 *(i)* What is the reason someone needs to know which algorithm the TCP sender is using? *(ii)* 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 behaviour 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, 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 [15]. Acknowledgments can be lost between the sender and the intermediate monitor, or between the monitor and the receiver. 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 over-estimation of a *cwnd* [15]. 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* [16]. 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, so the receiver did not yet know that the packets were received before they decided to retransmit them. More on the location of the passive monitor and its effect on what we can infer from the measurements is found in [16]. In this paper, we advocate that machine learning-based approaches can give a better prediction accuracy of TCP sender connection states from passive measurements collected at an intermediate node by addressing the aforementioned practical challenges.

The rest of the paper is organized as follows: Section II.3 overviews the background of our study. In Section II.4, we review and give a detailed overview of the *state-of-the-art* and discuss closely related works on TCP variants research. In Section II.5, we describe our experimental setup for the evaluation. Section II.6 gives an overview of our methodology highlighting the machine learning techniques, performance measurement metrics used in our paper. Section II.7 presents detailed experimental results and the multiple scenario settings used to validate our prediction model. Finally, Section II.8 concludes the paper and outlines directions of research for future extensions.

## II.3 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**: Reno [14] is one of the most predominant implementations of TCP variant that implements the *Additive Increase and Multiplicative*

*Decrease (AIMD)* scheme [5], 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 [29] is a predecessor of TCP CUBIC [10]. 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[14]. It uses the concept of *binary search* algorithm along with the AIMD [5] in an attempt to find the maximum *cwnd* that will last longer period. BIC-TCP [29] stand out from other TCP algorithms in its *stability*, TCP *friendliness* and RTT *fairness*.

3. **TCP CUBIC**: CUBIC [10] is an enhanced version of BIC [29]. It is the default congestion control algorithm as part of the Linux kernel distribution configurations from version 2.6.19. CUBIC [10] 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 [29]) and is designed to be less aggressive and fair to TCP in bandwidth usage than BIC [29] while maintaining the strengths of BIC [29] such as *stability*, window *scalability* and RTT *fairness*.

## II.4 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 research works on inferring TCP per-connection states related to network congestion from passive measurements. The techniques to monitor TCP per-connection characteristics are divided into *two* categories: *active measurement* and *passive measurement*.

### II.4.1 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 [19, 23]. 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.

### II.4.2 Passive Measurement

In a passive measurement, passively collected packet traces are examined to measure TCP behaviors of Internet flows [15, 25, 32]. 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 [15] 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 [15] 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* [15] 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 [15], the work in [16] 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 [15].

The authors of the study [27] 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. [21] proposes a *cluster analysis-based* method that aims to identify between two versions of 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 [21]. These characteristics are then clustered into two groups by applying a hierarchical clustering technique. The authors show that only 2 out of 14 TCP congestion algorithms that are implemented in Linux can be identified based on their method [21]. 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 [15, 25]. 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 [29], CUBIC [10], Reno [14] etc.).

## II.5 Controlled Experiments

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

### II.5.1 Experimental Setup

We describe our experimental procedure below. Figure II.1 shows the experimental setup that we use for all of our experiments.

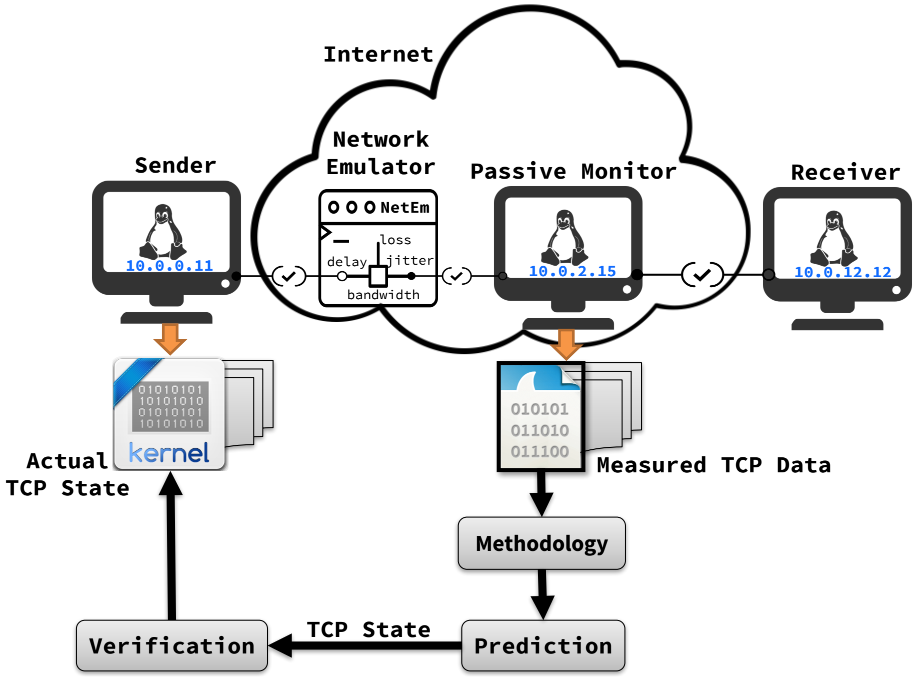


Figure II.1: Experimental Testbed.

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 [8] 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 II.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 used only for training whose data format output is shown in Table II.1 and generate a new data for the learning model to predict on. Once we finish with the verification, we run our learning model and get the predictions.

Table II.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 |

**Testbed hardware**

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* 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.

### II.5.2 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 II.1 by adding variability within a flow to the important network emulation parameters presented in Table II.2. For the network emulation, we used the popular Linux-based network emulator, *Network Emulator (NetEm)* [13] 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* [8] 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 II.2: Network Emulation Parameters.

| | **Bandwidth** (*Mbit/s*) | **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] |

### II.5.3 The Precision of the Emulator and Cross-traffic Variability

In order to use the network emulator, *NetEm* [13], with great care in an extremely well-contained environment for all the variations of *bandwidth*, *delay*, *jitter* and *packet loss* parameters, we created a filter that sets the parameter variation of each packet according to Table II.2. As the precision of the emulator, given that a software emulator is not precise, cannot be measured from TCP streams, we set up 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. In our experimental setup of the emulator, we have also carefully studied and validated the impact of cross-traffic variability from the same TCP congestion protocol on our results by emulating other UDP traffic. We ran *NetEm* [13] with variations in the data rate and the parameters presented in Table II.2 between the client and the server and we found out that each variation run by *NetEm* [13] 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.

In order to avoid packet sizes over the regular legitimate Maximum Segment Size (MSS) and Maximum Transmission Unit (MTU) values, we carried out our experiment on a path that is jumbo-frame clean by disabling TCP segmentation offloading. 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. Therefore, in all of our experiments, each TCP flow uses *1500*-byte data packets and an advertised window set by the operating system. 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.

### II.5.4 Assumptions

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.

## II.6 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 machine learning-based techniques.

### II.6.1 Passive Monitoring of outstanding bytes

The measured passive traffic collected at the intermediate node as shown in Figure II.1 is used for a training experiment of our model. A TCP sender includes a *sequence number* to identify every unique data packets sent into the

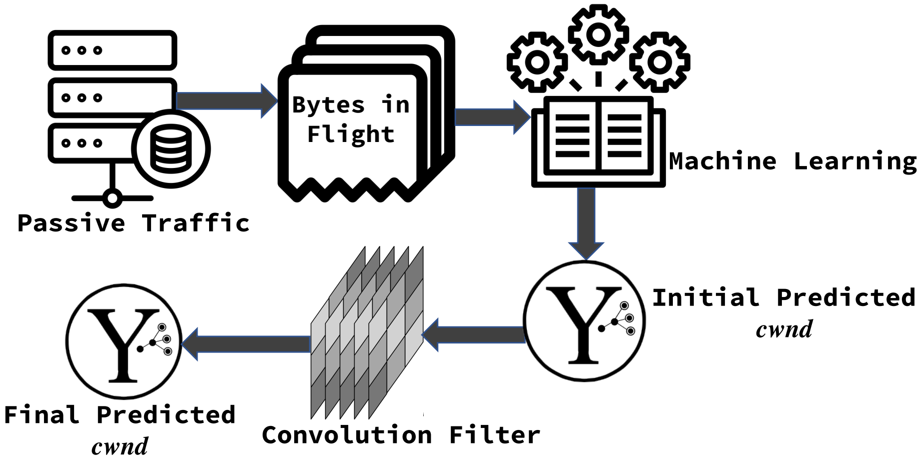


Figure II.2: Methodology for *cwnd* prediction.

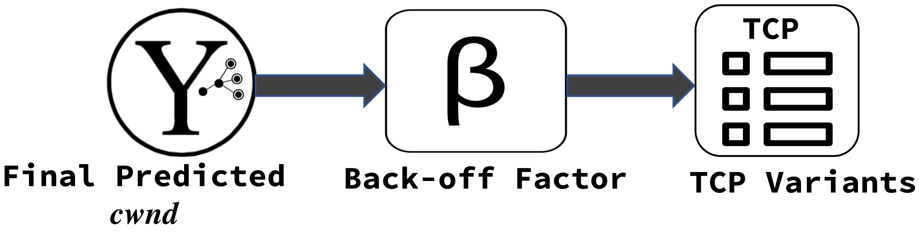


Figure II.3: Methodology for *TCP Variant* prediction.

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 to do congestion avoidance [14].

### II.6.2 Experimental inference of TCP cwnd from Passive Traffic

TCP's *cwnd* maintained independently by the sender controls the maximum number of packets a TCP flow may have in the network at any time maintained independently by the sender [14]. Taking the nature of TCP, accurately inferring *cwnd* of the sender by examining each cross-traffic of TCP flows of the endpoints passively collected at an intermediate node is a challenging task as it is not advertised. 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* [11]. The practical challenges with the experimental inference of TCP *cwnd* using a machine learning-based approach are explained thoroughly in [11].

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.

From the passive traffic at the intermediate node, we infer the number of bytes that have been *sent but not yet acknowledged* on the network at a given point in time using *tcptrace* [22]. This information is very useful in our experiment as it helps us match with the *cwnd* calculation of the particular TCP stack in use [11]. Once we estimate the *cwnd* of the sender, we can infer the *multiplicative decrease* parameter ($\beta$) which is an important feature for uniquely identifying TCP variants. We use the *python sklearn* library implementation [26] 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 tuning parameters of the ensemble methods are presented in detail in our previous work [11]. We trained our *ensemble* learning algorithm without the knowledge of the input features from the sender-side during the learning phase. We validated our methodology using the experimental testbed shown in Figure II.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 variations of *bandwidth*, *delay*, *jitter* and *packet loss* into one learning model. As it is shown below, 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 [6].

### II.6.3 Prediction of TCP Variants

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 ($\beta$) from the *final* predicted TCP *cwnd*, is shown in Figure II.3. For the underlying TCP 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 [10] BIC [29] and Reno [14]) for a proof of concept. As it is explained in Section II.2, 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. 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: *(i)* Do we see both directions of the traffic? *(ii)* Are we close to either endpoint, so we can hopefully estimate RTT accurately? *(iii)* How do we deal with the fact that some algorithms vary depending on past connections between the same pair of endpoints? *(iv)* 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? *(v)* 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 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 [5]. A thorough analysis and evaluation of AIMD can be found in [5]. The AIMD has a linear growth function for *increasing* the *cwnd* at the receipt of an ACK packet and *multiplicative decrease* parameter, denoted by $\beta$, on encountering a TCP packet loss at the receipt of *triple duplicate ACKs* and it can be described as shown below in Function II.1. 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.

$$\begin{aligned} \textbf{Ack} &: cwnd \leftarrow cwnd + \alpha \\ \textbf{Loss} &: cwnd \leftarrow \beta \times cwnd \end{aligned} \tag{II.1}$$

Most of the existing *loss-based* TCP congestion control algorithms implement

AIMD scheme as it is proven to converge [5]. It can generally be expressed as follows:

$$\begin{aligned} &\uparrow_G: w_{t+R} \leftarrow w_t + \alpha; \alpha > 0 \\ &\downarrow_G: w_{t+\delta t} \leftarrow (1-\beta) w_t; 0 < \beta < 1, \end{aligned} \tag{II.2}$$

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 $\delta$ is a sampling rate. The AIMD algorithm is generalized by adding *two* variables, $\alpha$ and $\beta$ that control the two aspects of AIMD: $\alpha$ indicates the increase in the window size if there is no packet loss in round-trip time and $\beta$ indicates the fraction of the window size that it is decreased to when packet loss is detected [5]. 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.

$$f(t+1) = \begin{cases} f(t) + \alpha, & \text{If congestion is detected} \\ f(t) \times \beta, & \text{If congestion is not detected} \end{cases} \tag{II.3}$$

In TCP, after *slow start*, the *additive increase* parameter $\alpha$ is typically one MSS every RTT, and the *multiplicative decrease* factor $\beta$ on loss event is typically $\frac{1}{2}$ [5]. For example, CUBIC [10] 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:

$$W_{cubic}{}^{(t)} = |C(t-K)|^3 + W_{max} \tag{II.4}$$

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:

$$K = \sqrt[3]{\frac{W_{max}\beta}{C}} \tag{II.5}$$

Where $\beta$ is a constant *multiplicative decrease* factor of CUBIC [10] applied for window reduction at the time of a TCP packet loss event (i.e., the window reduces to $\beta W_{max}$ at the time of the last reduction) [10]. The $\beta$ value of CUBIC [10] is 0.7, as shown in Table II.3, which corresponds to reducing the window by 30% during a TCP packet loss event and can be calculated as per Equations VI.6 and VI.7. The windows growth function of a TCP CUBIC [10] is a cubic function. TCP CUBIC [10] reduces its window by a factor of $\beta$ after a loss event, the TCP-friendly rate would be $3((1-\beta)/(1+\beta))$ per RTT. Different congestion control algorithms have different window growth functions. However, when TCP BIC [29] detects a packet loss, it reduces its window by a multiplicative factor $\beta$.

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 II.6.

$$\frac{W_{max} + W_{min}}{2} \tag{II.6}$$

The *multiplicative back-off* parameter, $\beta$, 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)* [30]. There are two approaches to measure the $\beta$ 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 ($\alpha$)) can be used to predict the underlying TCP congestion control algorithms. Hence, here we use the $\beta$ 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 II.3. Unlike *loss-based* algorithms, the $\beta$ 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 [9], etc.) have a variable $\beta$ and the $\beta$ 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.

Table II.3: *Loss-based* TCP Variants $\beta$ Value.

| **TCP Congestion Control Algorithm** | $\beta$ **Value** |
|---|---|
| BIC | 0.8 |
| CUBIC | 0.7 |
| Reno | 0.5 |

## II.7 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 [10], BIC [29] and Reno [14] 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 II.2). Due to space limitation in this paper, we can not present all the 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 II.4, II.5 and II.6 to verify the accuracy of our machine learning-based prediction model.

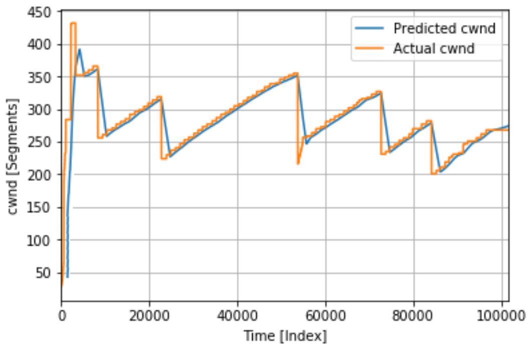
Our *final* TCP *cwnd* prediction model is evaluated under different configurations of training and testing sample size ratios. As it is shown in the plots below, we found out the *convolutional* filtering we built for predicting *cwnd* captures the ratio of the *cwnd* drop very accurately. Figures II.4*(a)* and *(b)* don't share the same *bandwidth*, *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 II.4*(b)*, it has a *Bandwidth-Delay Product (BDP)* [17] 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 shown below 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. In the results, 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.

### II.7.1 Emulated Network Setup

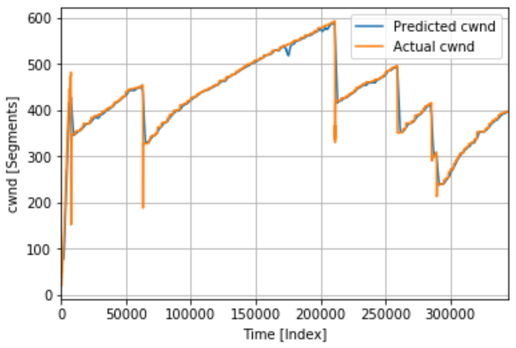
In Figure II.4, the comparison of the *final* predicted TCP *cwnd* after optimizing the prediction performance using *convolution* filtering technique and the actual *cwnd* of the sender tracked from the *kernel* is presented. As it is shown in Figure II.4, we found out the *convolutional* filtering we built for predicting *cwnd* captures the ratio of the *cwnd* drop very accurately. For a detailed explanation of the filtering technique refer to [11]. We evaluate our *final* TCP *cwnd* prediction model and the performance results are presented in Table II.6. For the TCP variant prediction, we analyzed the $\beta$ 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 as presented in Table II.5 is *93.51%*.

### II.7.2 Realistic Scenario Setup

In order to demonstrate the *transferability* [4, 24, 28] approach of our proposed machine learning-based prediction model and further validate our results presented in Section II.7 by conducting a series of controlled experiments against



(a) TCP CUBIC *final* predicted *cwnd* - emulated network



(b) TCP CUBIC *final* predicted *cwnd* - emulated network

Figure II.4: Final TCP *cwnd* prediction with different configurations of network emulation parameters for TCP CUBIC [10] after optimizing the *initial cwnd* prediction accuracy with *convolution* filtering technique in an *emulated* network. For more results with different configurations of an *emulated* network for TCP BIC [29] and TCP Reno [14] refer to our previous paper [11].

Table II.4: 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 II.5: 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** | | | |

Table II.6: TCP *final* predicted *cwnd* performance results of an *emulated network* setting with sample configurations.

| **Congestion Algorithm** | **Per Configuration** | **RMSE** | **MAPE (%)** |
|---|---|---|---|
| TCP CUBIC | Configuration $C_1$ | 5.839 | 6.953% |
| | Configuration $C_3$ | 2.209 | 2.857% |

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 II.7.

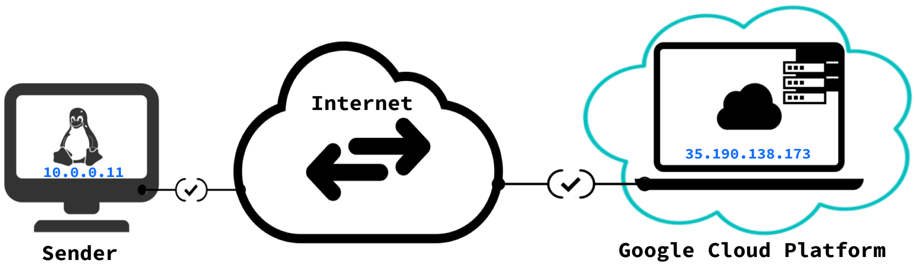
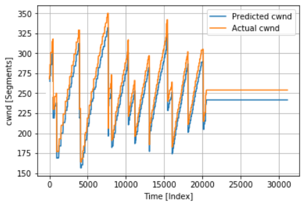
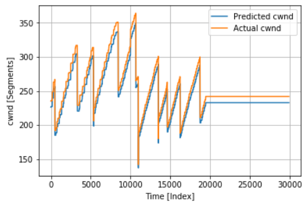


Figure II.7: Realistic scenario setup.

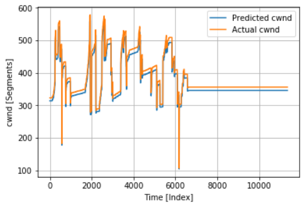
From an experimental view point, this helps us to justify and guarantee how our model could predict the development of a *cwnd* and the underlying TCP



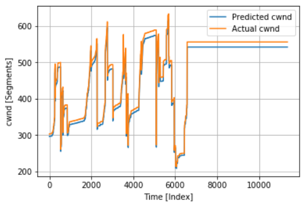
(a) CUBIC *final* predicted *cwnd*, USA site



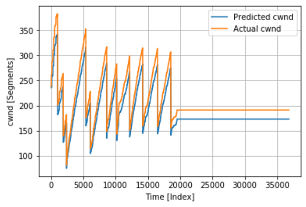
(b) CUBIC *final* predicted *cwnd*, Northeast Asia site



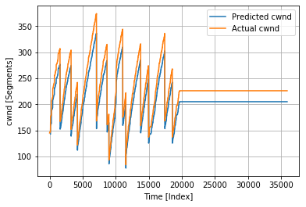
(c) BIC *final* predicted *cwnd*, USA site



(d) BIC *final* predicted *cwnd*, Northeast Asia site

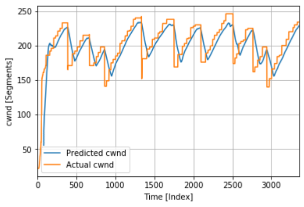


(e) Reno *final* predicted *cwnd*, USA site

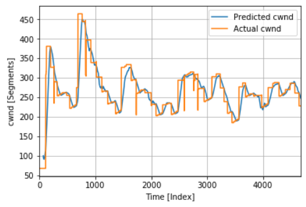


(f) Reno *final* predicted *cwnd*, Northeast Asia site

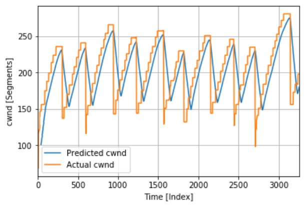
Figure II.5: TCP *cwnd* prediction of TCP CUBIC [10], TCP BIC [29] and TCP Reno [14] from a *realistic* scenario on different *zones* of *Google Cloud* platform (East coast USA (North Carolina) and Northeast Asia (Tokyo, Japan) sites).



(a) CUBIC *final* predicted *cwnd*, combined scenario



(b) BIC *final* predicted *cwnd*, Northeast Asia site



(c) Reno *final* predicted *cwnd*, USA site

Figure II.6: TCP *cwnd* prediction of TCP CUBIC [10], TCP BIC [29] and TCP Reno [14] from a *combined* scenario setting.

variant with other realistic network traffic scenarios captured from the Internet. To this end, we created a realistic testbed 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 II.5. 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 II.5, 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 II.7. As it is shown in Table II.9, the accuracy of the TCP variant prediction for this scenario setting is *95%*.

Table II.7: 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 II.8: TCP Variant Prediction of a *realistic scenario* setting: Confusion Matrix.

| | **Predicted** | | |
|---|---|---|---|
| **Actual** | BIC | CUBIC | Reno |
| BIC | 20 | 0 | 0 |
| CUBIC | 0 | 18 | 1 |
| Reno | 0 | 2 | 19 |

Table II.9: 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** | | | |

## II.7.3 Combined Scenario Setting

Real networks behave in more complex manner than emulated networks. The *loss* and *delay* of packets in TCP is 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* [13] can't reproduce well enough. In Section II.7.2, 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 II.8.

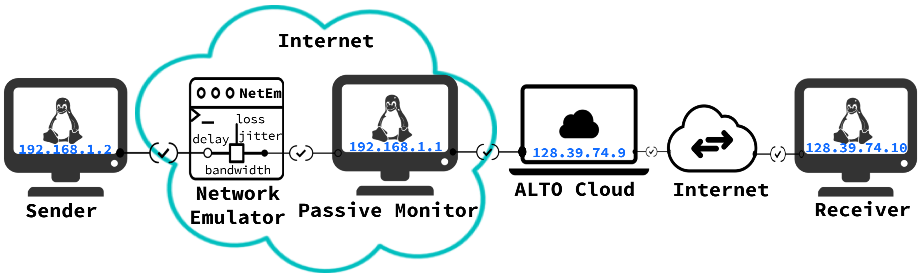


Figure II.8: Combined scenario setup.

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 II.6, 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 II.12, is 91.66% and this justifies that our prediction model can handle multiple scenario settings.

Table II.10: TCP *final* predicted *cwnd* performance results of a *combined scenario* setting.

| Congestion Algorithm | Per Configuration | RMSE | MAPE (%) |
|---|---|---|---|
| TCP CUBIC | *Sample Configuration* $C_1$ | 5.704 | 8.053% |
| TCP BIC | *Sample Configuration* $B_1$ | 5.193 | 7.831% |
| TCP Reno | *Sample Configuration* $R_1$ | 4.752 | 5.739% |

Table II.11: TCP Variant Prediction of a *combined scenario* setting: Confusion Matrix.

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

Table II.12: 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** | | | |

## II.8 Conclusion and Future Work

In this paper, 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 *robust*, *scalable* and *generic* learning model for predicting the widely deployed underlying TCP variants within a flow which may be 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 by leveraging knowledge from the emulated network performs reasonably well when it is applied on a real-life scenario setting 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, it would be interesting to develop a *delay-based* model 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. We plan to address these open issues and extend the approaches in our future work.

## Acknowledgment

We gratefully acknowledge the reviewers for their helpful feedback and detailed comments. We would like to thank the Research Infrastructure Services Group at the University of Oslo for the use of multicore cluster machines.

## II.9 References

[1] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. *TCP Vegas: New techniques for congestion detection and avoidance*, volume 24. ACM, 1994.

[2] L. Breiman. Random forests. *Machine learning*, 45(1), 2001.

[3] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, et al. BBR: congestion-based congestion control. *ACM*, 2017.

[4] R. Caruana. Multitask learning. In *Learning to learn*, pages 95–133. Springer, 1998.

[5] D.-M. Chiu and R. Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Computer Networks and ISDN systems*, 17(1):1–14, 1989.

[6] A. De Myttenaere, B. Golden, B. Le Grand, and F. Rossi. Mean absolute percentage error for regression models. *Neurocomputing*, 192:38–48, 2016.

[7] N. Dukkipati, Y. Cheng, and A. Vahdat. Research Impacting the Practice of Congestion Control, 2016.

[8] ESnet. iperf3. https://iperf.fr/iperf-servers.php, 2017.

[9] M. Gerla, M. Y. Sanadidi, R. Wang, A. Zanella, C. Casetti, and S. Mascolo. TCP Westwood: Congestion window control using bandwidth estimation. In *GLOBECOM'01. IEEE Global Telecommunications Conference (Cat. No. 01CH37270)*, volume 3, pages 1698–1702. IEEE, 2001.

[10] S. Ha, I. Rhee, and L. Xu. CUBIC: a new TCP-friendly high-speed TCP variant. *ACM SIGOPS operating systems review*, 42(5):64–74, 2008.

[11] D. H. Hagos, P. E. Engelstad, A. Yazidi, and Ø. Kure. A machine learning approach to TCP state monitoring from passive measurements. In *2018 Wireless Days (WD)*, pages 164–171. IEEE, 2018.

[12] D. H. Hagos, A. Yazidi, Ø. Kure, and P. E. Engelstad. Enhancing security attacks analysis using regularized machine learning techniques. In *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*, pages 909–918. IEEE, 2017.

[13] S. Hemminger et al. Network emulation with NetEm. 2005.

[14] V. Jacobson. Congestion avoidance and control. In *ACM SIGCOMM computer communication review*, volume 18, pages 314–329. ACM, 1988.

[15] S. Jaiswal, G. Iannaccone, C. Diot, J. Kurose, and D. Towsley. Inferring tcp connection characteristics through passive measurements. In *IEEE INFOCOM 2004*, volume 3, pages 1582–1592. IEEE, 2004.

[16] 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 Transactions on Networking (ToN)*, 15(1):54–66, 2007.

[17] D. Katabi, M. Handley, and C. Rohrs. Congestion control for high bandwidth-delay product networks. *ACM SIGCOMM computer communication review*, 32(4):89–102, 2002.

[18] B. Mao, Z. M. Fadlullah, F. Tang, N. Kato, O. Akashi, T. Inoue, and K. Mizutani. Routing or computing? the paradigm shift towards intelligent computer network packet transmission based on deep learning. *IEEE Transactions on Computers*, 66(11):1946–1960, 2017.

[19] A. Medina, M. Allman, and S. Floyd. Measuring the evolution of transport protocols in the internet. *ACM SIGCOMM Computer Communication Review*, 35(2):37–52, 2005.

[20] T. T. Nguyen and G. J. Armitage. A survey of techniques for internet traffic classification using machine learning. *IEEE Communications Surveys and Tutorials*, 10(1-4):56–76, 2008.

[21] J. Oshio, S. Ata, and I. Oka. Identification of different TCP versions based on cluster analysis. In *2009 Proceedings of 18th International Conference on Computer Communications and Networks*, pages 1–6. IEEE, 2009.

[22] S. Ostermann. Tcptrace. http://www. tcptrace. org, 2000.

[23] J. Pahdye and S. Floyd. On inferring TCP behavior. *ACM SIGCOMM Computer Comm. Review*, 31(4):287–298, 2001.

[24] S. J. Pan and Q. Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2010.

[25] V. Paxson. Automated packet trace analysis of TCP implementations. *ACM SIGCOMM Computer Communication Review*, 27(4):167–179, 1997.

[26] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in Python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

[27] S. Rewaskar, J. Kaur, and D. Smith. A Passive State-Machine Based Approach for Reliable Estimation of TCP Losses. 2006.

[28] L. Torrey and J. Shavlik. Transfer learning. *Handbook of Research on Machine Learning Applications and Trends: Algorithms, Methods, and Techniques*, 2009.

[29] L. Xu, K. Harfoush, and I. Rhee. Binary increase congestion control (BIC) for fast long-distance networks. In *IEEE INFOCOM*, volume 4, pages 2514–2524. IEEE, 2004.

[30] P. Yang, J. Shao, W. Luo, L. Xu, J. Deogun, and Y. Lu. TCP congestion avoidance algorithm identification. *IEEE/Acm Transactions On Networking*, 22(4):1311–1324, 2014.

[31] J. Zhang, X. Chen, Y. Xiang, W. Zhou, and J. Wu. Robust network traffic classification. *IEEE/ACM Transactions on Networking (TON)*, 23(4):1257–1270, 2015.

[32] Y. Zhang, L. Breslau, V. Paxson, and S. Shenker. On the characteristics and origins of internet flow rates. In *ACM SIGCOMM Computer Communication Review*, volume 32, pages 309–322. ACM, 2002.

Paper III

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

**Desta Haileselassie Hagos**[1], **Paal E. Engelstad, Anis Yazidi, Øivind Kure**

***Published*** in *IEEE Access* 6 (2018): 28372–28387. IEEE, 2018.
DOI: 10.1109/ACCESS.2018.2833107.

Since this paper is an IEEE journal extension of Paper I and and Paper II, it is not included as part of this dissertation to avoid redundancy for the reader. The publication is avaiblable at DOI: 10.1109/ACCESS.2018.2833107 for interested readers.

[1]University of Oslo, Department of Informatics, destahh@ifi.uio.no

Paper IV

# Recurrent Neural Network-Based Prediction of TCP Transmission States from Passive Measurements

**Desta Haileselassie Hagos**[1], **Paal E. Engelstad, Anis Yazidi, Øivind Kure**

***Published*** in the *17th IEEE International Symposium on Network Computing and Applications (NCA 2018)*, pp. 1–10. IEEE, 2018.
DOI: 10.1109/NCA.2018.8548064.

## Abstract

Long Short-Term Memory (LSTM) neural networks are a *state-of-the-art* techniques when it comes to sequence learning and time series prediction models. In this paper, we have used LSTM-based Recurrent Neural Networks (RNN) for building a generic prediction model for Transmission Control Protocol (TCP) connection characteristics from passive measurements. To the best of our knowledge, this is the first work that attempts to apply LSTM for demonstrating how a network operator can identify the most important system-wide TCP per-connection states of a TCP client that determine a network condition (e.g., *cwnd*) from passive traffic measured at an intermediate node of the network without having access to the sender. We found out that LSTM learners outperform the *state-of-the-art* classical machine learning prediction models. Through an extensive experimental evaluation on multiple scenarios, we demonstrate the scalability and robustness of our approach and its potential for monitoring TCP transmission states related to network congestion from passive measurements. Our results based on *emulated* and *realistic* settings suggest that *Deep Learning* is a promising tool for monitoring system-wide TCP states from passive measurements and we believe that the methodology presented in our paper may strengthen future research work in the computer networking community.

[1]University of Oslo, Department of Informatics, destahh@ifi.uio.no

**Keywords**: Deep Learning, Long Short-Term Memory, TCP Congestion Control, Passive Measurement, Recurrent Neural Networks

## IV.1 Introduction

Deep Neural Networks (DNN) [20, 28] are a deep learning architecture trained using new machine learning methods that have shown advancements in a wide range of supervised and unsupervised machine intelligence tasks. In recent years, Recurrent Neural Network (RNNs) have become popular focus of research topic in the areas of DNN as diverse as, for example, *speech recognition* [8], *music generation* [4], *text generation* [30], *sentiment classification* [31], and other areas of major advancements. RNNs use input sequences to solve both for prediction [5] as well as classification [2, 19, 21] problems. LSTM [14] is a special kind of RNN *state-of-the-art* architecture designed for a wide range of sequence modeling tasks and time series prediction models. The LSTM unit [14] is a powerful and flexible RNN tool that has a memory cell that gives a previous hidden state containing connection information through the hidden layer activations from the past for a long period of time. LSTM in its recurrent hidden layer has a special unit called *memory blocks* consisting of memory cell units that are responsible for remembering the temporal states of the network for an arbitrary time intervals [14]. In each layer of the LSTM architecture [14], there is a forward propagation step which is a corresponding backward propagation through time step. In addition to this, there is a *cache* that passes information from one layer to another. This ability of LSTM [14] allows us to solve the vanishing gradient problem by dynamically controlling the information flow within the layers and capture the long-term dependencies of the connections in a sequence effectively.

LSTM [14] is used to address difficult sequence learning and prediction problems in machine learning and have achieved *state-of-the-art* results. One of the main benefits of using an LSTM model for challenges that involve time series data is to avoid the *vanishing gradient* problem. RNN model scans through the training data from left to right and the parameters it uses to govern the connection in the hidden layer for each time-step, learned features during the training are shared and this significantly improves the prediction. An LSTM model computes a mapping from an input feature vector $x = (x_{(1)}, x_{(2)}, x_{(3)}, ..., x_{(\mathrm{n})})$ where $x_{\mathrm{i}} \in \mathbb{R}^n$ to an output sequence $y = (y_{(1)}, y_{(2)}, y_{(3)}, ..., y_{(\mathrm{n})})$ where $y_{\mathrm{i}} \in \mathbb{R}^n$ by calculating the network unit activations of a weighted sum using the Equations IV.1-IV.6 iteratively from $t = 1$ to $n$.

As it is shown in Equations IV.1, IV.2, and IV.4, LSTM [14] uses *three* adaptive, an *input*, *forget* and *output*, gates shared by all cells in the LSTM block in order to learn long-term dependencies and control the flow of information. The output of these gates multiplicatively influences connections within the memory units. The *input* gate determines the flow of input activations into the memory cell whereas the *output* gate determines the output flow of cell activations into the rest of the network. The *forget* gate determines the extent to which the current value remains in the memory cell of the LSTM unit before it gets gradually discarded when its data is no longer needed.

$$i_t = \sigma(W_{ix}x_t + W_{im}m_{t-1} + W_{ic}c_{t-1} + b_i) \tag{IV.1}$$
$$f_t = \sigma(W_{fx}x_t + W_{fm}m_{t-1} + W_{fc}c_{t-1} + b_f) \tag{IV.2}$$
$$c_t = f_t \odot c_{t-1} + i_t \odot g(W_{cx}x_t + W_{cm}m_{t-1} + b_c) \tag{IV.3}$$
$$o_t = \sigma(W_{ox}x_t + W_{om}m_{t-1} + W_{oc}c_t + b_o) \tag{IV.4}$$
$$m_t = o_t \odot h(c_t) \tag{IV.5}$$
$$y_t = \phi(W_{ym}m_t + b_y) \tag{IV.6}$$

Where the $i$, $f$, $c$, $o$ are *input*, *forget*, *memory state*, and *output* gate activation vectors respectively at each time step $t$. $\sigma$ is the logistic sigmoid non-linearity while $\odot$, $g$ and $h$ are element-wise product of the vectors, the cell input and output non-linearity activation functions of the entire neural network, $ReLU$ in our case, applied to each layer of the deep network respectively. $W$ and $b$ represents a vector of weighted recurrent connections and the bias vector. $m_t$ is the hidden state output of the LSTM layer. Finally, $\phi$ is the activation function in the hidden layer applied to the network output. Figure IV.1 describes the basic unit of an LSTM network where the input sequence to the LSTM cell is carried over each *time step* of *t+1*, *t* and *t-1*. As shown in Figure IV.1, the hidden state, at time step *t*, is a function of the current input sequence $x_t$ at the same time step. $C_t$ and $C_{t-1}$ are the memory cell state activation vectors from the current and previous block at time *t* and *t-1* respectively.

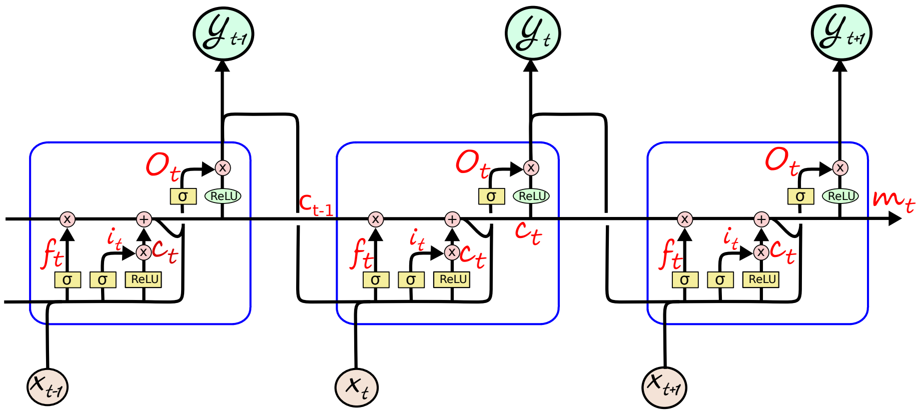


Figure IV.1: LSTM Networks. For more thorough details, refer [23].

In this paper, we are interested in the capability of RNN model based on emulated and realistic networks for estimating TCP *cwnd* as well as the underlying TCP variants within a flow. Hence, we have explored an LSTM architecture for RNN-based prediction approaches to monitor the most important TCP per-connection states from passive measurements related to network congestion. In our paper, we have demonstrated that LSTM can use its memory blocks and a series of gates to effectively capture the patterns of a TCP *cwnd* from passive measurements. Congestion control is a fundamental problem in computer networks. The TCP congestion control algorithms that are widely deployed today perform the most important functionalities related to congestion control such as handling the *cwnd* from the sender-side. In this paper, we investigate and explore questions quantitatively as they apply to problems of network congestion that include: *(i)* How well can we infer the most important TCP per-connection transmission states that determine a network condition from passive traffic collected at an intermediate node of the network? *(ii)* How can we uniquely track the underlying TCP variant that the TCP client is using from passive measurements? *(iii)* What is the motivation why we need to know which algorithm the TCP sender is using? *(iv)* Is there some action that we would take based on knowing the information of the underlying TCP variant of the sender? *(v)* Which user is responsible for the majority of heavy flow traffic in the network? etc.?

The work in [16] presented an approach to estimate TCP parameters at the sender-side based on packets captured at the monitoring point using a *finite state machine*. The authors have pointed out that the estimation of *cwnd* may have potential errors primarily due to an over-estimation of the Round-trip Time (RTT) and estimation of incorrect window sizes [16]. Another limitation of this work, given the many existing variants of TCP, is that the use of a separate *state machine* for each TCP variant is *unscalable* and we also believe that the constructed *replica* may 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. Within the computer networking research community, RNN techniques are potentially useful. After we extensively survey the existing works for monitoring of TCP transmission states from passive measurements, we believe there is not much work on a *scalable* method of predicting the *cwnd* and uniquely identifying the type of the underlying TCP control algorithm from passive traffic without the knowledge of the sender's *cwnd* for most of the widely used TCP variants using RNN-based techniques. Hence, in this paper, 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 LSTM [14]. Our experimental results demonstrate the feasibility of our prediction model. We believe that our study will be potentially useful to network operators, 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 from passive measurements. To the best of our knowledge, this is the first work that attempts to apply LSTM [14] for inferring the most important TCP per-connection states that determine a network condition from passive traffic collected at an intermediate node of the network without having access to the sender. Our prediction model has several benefits over other approaches as we demonstrate in our experimental results.

**Our Contributions**: The main contributions of our 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 *Congestion Window (cwnd)* size of the sender from passive measurements using an LSTM recurrent model.
- We explore the applicability of our LSTM-based 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 a trained knowledge from the *emulated network* when it is applied and transferred 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 [26].
- We validate the *robustness* and *scalability* approach of our prediction model extensively through several controlled experiments and experimentally verified across an *emulated*, *realistic* and *combined* scenario settings.

## IV.2 Motivation

Our work is mainly motivated by the questions presented on Section IV.1. Congestion control algorithms have a critical role in improving the performance of TCP on the Internet [6]. 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 predicting the *cwnd* and uniquely identifying the underlying TCP variant.

**Benefits:** From an *operational perspective*, 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 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. 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 and anomaly detection [12].

**Methodological Challenges:** In practice; however, 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. The set of methodological challenges we identify involved in performing inference of TCP per-connection states related to network congestion from passive measurements are presented more in detail in [10]. In this paper, we advocate that LSTM-based approaches can give a better prediction accuracy of TCP sender connection states from passive measurements collected at an intermediate node by addressing the aforementioned practical challenges.

**Roadmap**: The rest of the paper is organized as follows: In Section IV.3, we review and give a detailed overview of the closely related research works of TCP passive measurements considered as a *state-of-the-art*. In Section IV.4, we describe our experimental setup for the evaluation. Section IV.5 gives an overview of our methodology highlighting the machine learning techniques, performance measurement metrics used in our paper. Section IV.6 presents detailed experimental results and the multiple scenario settings used to validate our prediction model. Finally, Section IV.7 concludes the paper and outlines directions of research for future extensions.

## IV.3 Related Work

This section briefly discusses closely related research works on inferring TCP per-connection states related to network congestion from passive measurements. The techniques to monitor TCP per-connection characteristics are divided into *two* categories: *active* and *passive measurements*.

***Active Measurement***: Many existing research works that have been proposed rely on an *active* approach to measuring the characteristics of TCP. This technique actively measures the TCP behaviors of Internet flows by injecting an artificial traffic into the network between at least two endpoints [22, 25]. 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.

***Passive Measurement***: In a passive measurement, passively collected packet traces are examined to measure TCP behaviors of Internet flows [16]. 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. A work of interest that is most closely related to our work is [16] 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 Retransmission Timeout (RTO) events at the sender and observing the ACKs which cause the sender to change the value of the *cwnd*. This work [16] considers only the predominant implementations of TCP 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* [16] 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. The authors of the study [27] 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 new TCP congestion control algorithms.

Our work mainly differs from the previous works in that our main goal is more fundamentally to develop a *scalable* LSTM-based prediction model for inferring TCP per-connection states for the most widely used *loss-based* congestion algorithms. 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 variants. Hence, a list of the most widely used *loss-based* variations of TCP algorithms we consider in our work so as to cover the whole scope of the problem are BIC [32], CUBIC [9] and Reno [15].

## IV.4 Experimental Setup and Discussion

In this section, we provide a detailed overview of our experimental testbed.

### IV.4.1 Experimental Testbed

Figure IV.2 shows the experimental setup that we use for all of our experiments in this paper. In order to introduce congestion, 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 [7]. We carried out the experiment by capturing all sessions on the network when the client and server are sending TCP packets. 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*. As shown in Figure IV.2, we used the *measured* TCP data as an *input* to our methodology for a prediction of the TCP per-connection states. Finally, we verified the predicted TCP states with the actual TCP kernel states directly logged from the Linux kernel used only for training and generate a new data for the learning model to predict on. Once we finish with the verification, we run our learning model and get the predictions.

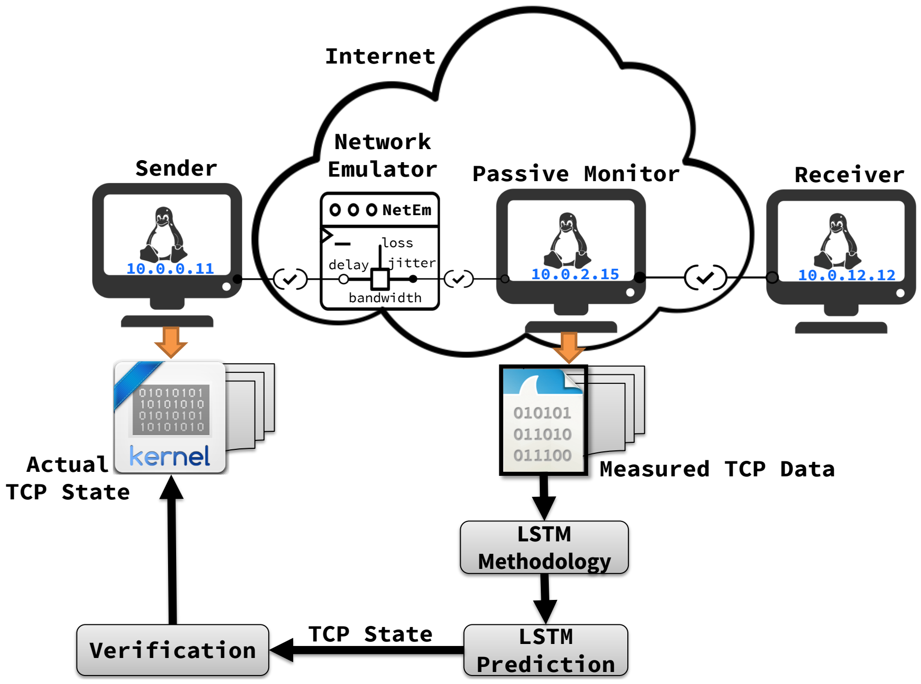


Figure IV.2: Experimental Setup.

### IV.4.2 Testbed Hardware

We have carried out our experiments using a cluster of HPC machines based upon the GNU/Linux operating system running a modified version of the *4.4.0-75-generic* kernel release. The prediction 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. All nodes in the cluster are connected to a low latency 56 Gbit/s *Infiniband*, *gigabit* Ethernet and have access to 600 TiB of *BeeGFS* parallel file system storage.

### IV.4.3 Network Emulation and Verification of the emulator

For the network emulation, we used the popular Linux-based network emulator, *Network Emulator (NetEm)* [13] on a separate node, that supports an end-to-end variability of *bandwidth*, *delay*, *jitter*, *packet loss*, and other parameters that the *cwnd* is highly influenced by to an outgoing packets of a selected network interface. Given that the software emulator is not precise, can we trust the network emulator for all the variations of *bandwidth*, *delay*, *jitter* and *packet loss* parameters that we change for our evaluation irrespective of the measurement we get from TCP stream? In order to use the network emulator with great care in an extremely well-contained environment for all the variations of the parameters, we created a filter that sets the parameter variation of each packet. As the precision of the emulator cannot be measured from TCP streams, we set up 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.

### IV.4.4 Impact of Cross-traffic Variability

We ran *NetEm* [13] with variations in the data rate and the emulation parameters between the client and the server. We have carefully studied and validated the impact of cross-traffic variability from the same TCP congestion protocol on our results by emulating other UDP traffic and we found out that each variation run by the emulator doesn't affect our results. We believe that the variability of the cross-traffic in our current 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. We also believe the emulator may be impacted by network elements outside of its scope e.g., CPU load, network card buffers, hardware architectural factors etc.

### IV.4.5 Network Traces

To evaluate our prediction model on both the emulated and realistic network conditions, we have generated our own dataset using *tcptrace* [24]. The data traces for all our experiments are generated using the *iperf* [7] traffic generator on an emulated LAN link where we run each TCP variant with variation of the parameters *bandwidth*, *delay*, *jitter* and *packet loss* as shown below in Table IV.1 where the *cwnd* is highly influenced by. However, 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 so that we can avoid packet sizes way over the regular legitimate size.

### IV.4.6 Network Emulation Parameters

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 IV.2 by adding variability within a flow to the important network emulation parameters presented in Table IV.1.

Table IV.1: Network Emulation Parameters.

| | **Bandwidth** (*Mbit/s*) | **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] |

### IV.4.7 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* [24] when there is variability of *bandwidth*, *delay*, *loss* and *RTT* is a better approach to estimate the *cwnd* and how fast the recovery is. Firstly, since we are estimating *cwnd* from *bytes in flight*, we have also considered that *cwnd* must be the limiting factor for the sender and it has to be less than the receiver side window. Secondly, we assume that we don't know what TCP variant is running in the network and the per-connection state within the variant. Lastly, the results we present in this paper assume that the endpoints have the same *receiver window* set by the operating system independent of the underlying TCP variant.

## IV.5 Methodology

This section explains the general methodology we have used to experimentally infer both the *cwnd* and uniquely identifying the underlying TCP variant from passive measurement using RNN-based techniques.

### IV.5.1 Passive Monitoring of bytes in flight

TCP congestion control algorithms govern the TCP sender's sending rate by employing the *cwnd* that limits the number of cumulatively *unacknowledged bytes*

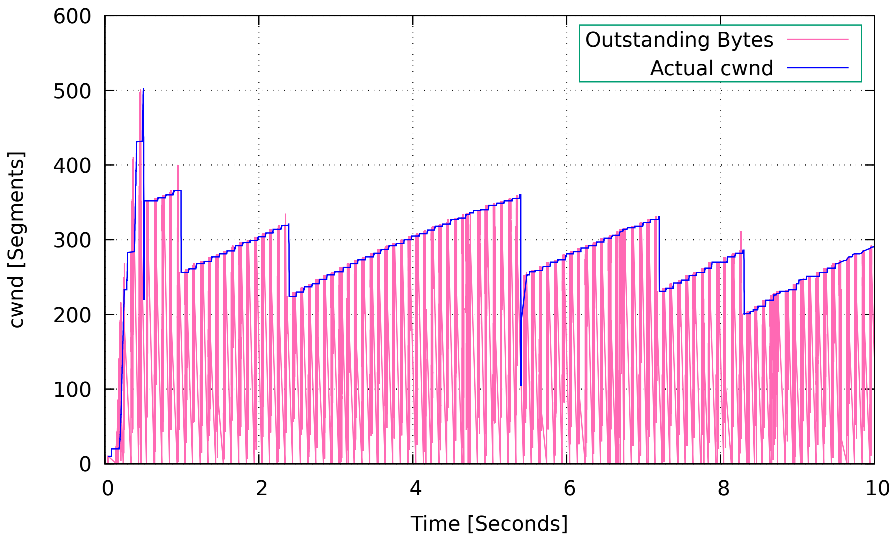


Figure IV.3: *Outstanding bytes* calculated from the monitor before applying LSTM technique vs. the actual *cwnd* from the sender.

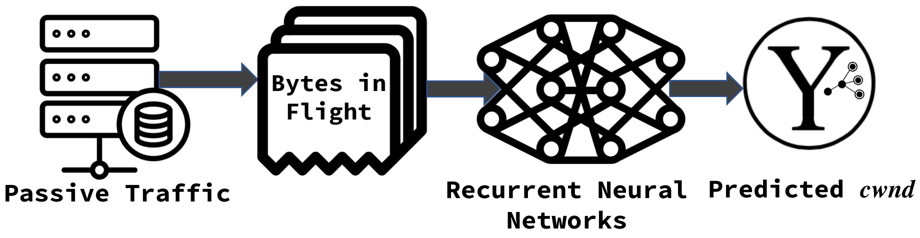


Figure IV.4: Methodology for *cwnd* prediction.

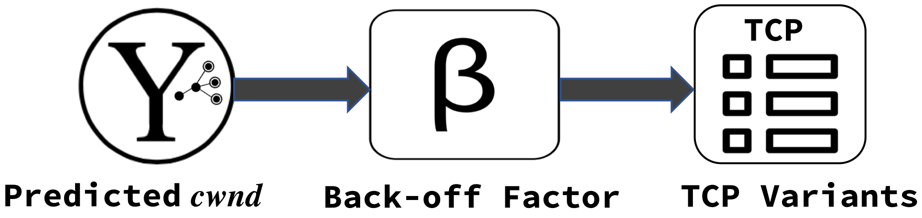


Figure IV.5: Methodology for *TCP Variant* prediction.

that are allowed at any given time. The measured passive TCP data collected at the intermediate node as shown in Figure IV.2 is used for a training experiment of our model. The TCP implementation details and use of TCP options are not visible at the monitoring point. 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).

### IV.5.2 Prediction of TCP cwnd from Passive Traffic

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 [15] uses *cwnd* that determine 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. Figure IV.3 shows the comparison between the number of *outstanding bytes* from the intermediate node before running the *neural* model and applying the LSTM techniques versus the actual *cwnd* tracked from the kernel of the sender-side with respect to time. Taking the nature of TCP, accurately inferring *cwnd* of the sender by examining each cross-traffic of TCP flows of the endpoints passively collected at an intermediate node is a challenging task as it is not advertised. 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 out to be an insufficient detail for an accurate prediction. In this paper, we argue that training a classifier and prediction model utilizing RNN-based algorithms to predict the *cwnd* from passive measurements is very important.

**Learning Context**: We built and trained a highly *robust* and *scalable* RNN-based prediction model in *Python* using the *Keras* deep learning framework with a *TensorFlow* backend [1] where we apply an LSTM-based architecture to estimate the *cwnd* trained over multiple epochs with a batch size of *32*. As shown in Figure IV.1, at each time-step of *t*, the LSTM model takes an entire array of *outstanding bytes in flight* as an input feature vector ($x$) indexed by *time stamp* obtained from the kernel. We propagate the input to the model through a multilayer LSTM cell followed by a dense layer of 15-dimensional hidden states with *ReLU* activation that generates an output of a sequence dimensional vector of predicted *cwnd* ($y$) of the same size indexed by *time stamp*.

Our LSTM network is trained using the *Truncated Back Propagation Through Time* (*TBPTT*) training algorithm for modern RNNs applied to sequence prediction problems [29]. We used this training algorithm to minimize LSTM's total prediction error between the expected output and the predicted output for a given input of the *bytes in flight*. We trained our LSTM-based learning algorithm without the knowledge of the input features from the sender-side during the learning phase. We validated our methodology using the experimental testbed shown in Figure IV.2 over a LAN link. In order to train and test our prediction model, we employed a single trained network that adapts to all experiments with variations of *bandwidth*, *delay*, *jitter* and *packet loss* into one learning model. We have trained our recurrent model on a GPU using the *Adam* stochastic optimization algorithm [18] with the default *learning rate* of *0.001*. We optimize the hyper-parameters (e.g., *Number of epochs*, *batch size*, *the number of time steps to unroll the LSTM during training*, *cell hidden state size* and *the number of LSTM layers*) related to the neural network topology so as to improve the performance of our prediction model. In order to boost our neural network implementation, we used the *ReLU* activation function for the hidden layer. 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. Finally, in order to evaluate and measure how well our LSTM-based prediction model performs in terms of capturing the *cwnd* pattern, all neural networks are trained, as it is shown in Section IV.6, by employing both the *Root Mean Square Error (RMSE)* and *Mean Absolute Percentage Error (MAPE)* loss functions.

### IV.5.3 Prediction of TCP Variants

Our methodology for uniquely identifying the underlying TCP variant from passive measurements by inferring the *multiplicative decrease* parameter, denoted by ($\beta$), from the predicted TCP *cwnd* is shown in Figure IV.5. The standard TCP congestion algorithm employs an Additive Increase and Multiplicative Decrease (AIMD) scheme that backs off in response to a single congestion indication [3]. The AIMD has a linear growth function for *increasing* the *cwnd* at the receipt of an ACK packet and $\beta$ on encountering a TCP packet loss at the receipt of *triple duplicate ACKs.* This scheme adjusts the *cwnd* by the *increase-by-one decrease-to-half* strategy. The aspect of the AIMD algorithm is generalized and controlled by adding *two* variables, $\alpha$ and $\beta$. $\alpha$ indicates the increase in the window size if there is no packet loss in round-trip time and $\beta$ indicates the fraction of the window size that it is decreased to when packet loss is detected [3]. 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.

$$f(t+1) = \begin{cases} f(t) + \alpha, & \text{If congestion is detected} \\ f(t) \times \beta, & \text{If congestion is not detected} \end{cases} \tag{IV.7}$$

For the underlying TCP variant prediction task, we consider only *loss-based* TCP congestion control algorithms (e.g., CUBIC [9] BIC [32], and Reno [15]) [11] that consider packet loss as an implicit indication of congestion by the network for a proof of concept. 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. The $\beta$ value 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 *ssthresh* [33]. There are two approaches to measure the $\beta$ value of a TCP congestion control algorithm: *(i)* using a packet loss event, and *(ii)* using a timeout 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 timeout 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. The *back-off* parameter along with other TCP characteristics can be used to predict the underlying TCP congestion control algorithms. Hence, here we use the $\beta$ value so as to uniquely predict the underlying TCP variant of the selected *loss-based* TCP congestion control algorithms summarized in Table IV.2.

Table IV.2: $\beta$ Values of *Loss-based* TCP Variants.

| **TCP Congestion Control Algorithm** | $\beta$ **Value** |
|---|---|
| BIC | 0.8 |
| CUBIC | 0.7 |
| Reno | 0.5 |

## IV.6 Experiments and Results

In this section, we summarize in detail the several experimental results that illustrate our main contributions under multiple scenarios using an LSTM-based RNN architecture. In the experimental evaluations, we choose a testing scenario configurations and present CUBIC [9], BIC [32] and Reno [15] 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 IV.1). Due to space limitation in this paper, we cannot present all the 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 IV.6, IV.7, IV.8, and IV.9 to verify the accuracy of our LSTM RNN-based prediction model.

The TCP *cwnd* pattern prediction model is evaluated under different configurations of training and testing sample size ratios. As it is shown in the plots below, we found out the *RNN-based* model we built for predicting *cwnd* captures the ratio of the *cwnd* drop very accurately. Figures IV.6*(a)* and *(b)* don't share the same *bandwidth*, *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 IV.6*(b)*, it has a *Bandwidth-Delay Product (BDP)* [17] 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 shown below 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. In the results, 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 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.

### IV.6.1 Emulated Network Setup

In Figure IV.6, the comparison of the *predicted* TCP *cwnd* and the actual *cwnd* of the sender in an *emulated* setup is presented. We found out our prediction model captures the ratio of the *cwnd* drop very accurately. We evaluate our TCP *cwnd* prediction model and the performance results with different configurations are presented in Table IV.3. For the TCP variant prediction, we analyzed the $\beta$ 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 as shown in Table IV.5 is *97.22%*.

Table IV.3: Prediction of *cwnd* on an *emulated* network.

| **TCP Algorithms** | **Sample Configurations** | **RMSE** | **MAPE (%)** |
|---|---|---|---|
| CUBIC | *Predicted cwnd* - $C_1$ | 2.181 | 2.846% |
| | *Predicted cwnd* - $C_2$ | 2.855 | 3.103% |
| Reno | *Predicted cwnd* - $R_1$ | 2.013 | 2.815% |

Table IV.4: TCP Variant Prediction of an *emulated network* setting: Confusion Matrix.

| | **Predicted** | | |
|---|---|---|---|
| **Actual** | BIC | CUBIC | Reno |
| BIC | 34 | 0 | 0 |
| CUBIC | 1 | 35 | 0 |
| Reno | 1 | 1 | 36 |

Table IV.5: TCP Variant Prediction of an *emulated network* setting: Performance metrics.

| | **Precision** | **Recall** | **F1-Score** | **Support** |
|---|---|---|---|---|
| BIC | 0.94 | 1.00 | 0.97 | 34 |
| CUBIC | 0.97 | 0.97 | 0.97 | 36 |
| Reno | 1.00 | 0.95 | 0.97 | 38 |
| Average/Total | 0.97 | 0.97 | 0.97 | 108 |
| **Accuracy** | **0.9722** | | | |

### IV.6.2 Realistic Scenario Setup

In order to demonstrate the *transferability* [26] approach of our proposed machine learning-based prediction model and further validate our results presented in Section IV.6 by conducting a series of controlled experiments against other

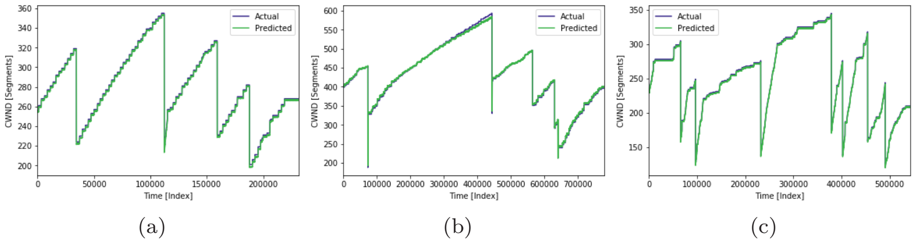


Figure IV.6: TCP *cwnd* prediction with different configurations in an *emulated network* setting. **(a)** CUBIC [9] Configuration $C_1$, **(b)** CUBIC [9] Configuration $C_2$, **(c)** Reno [15] Configuration $R_1$

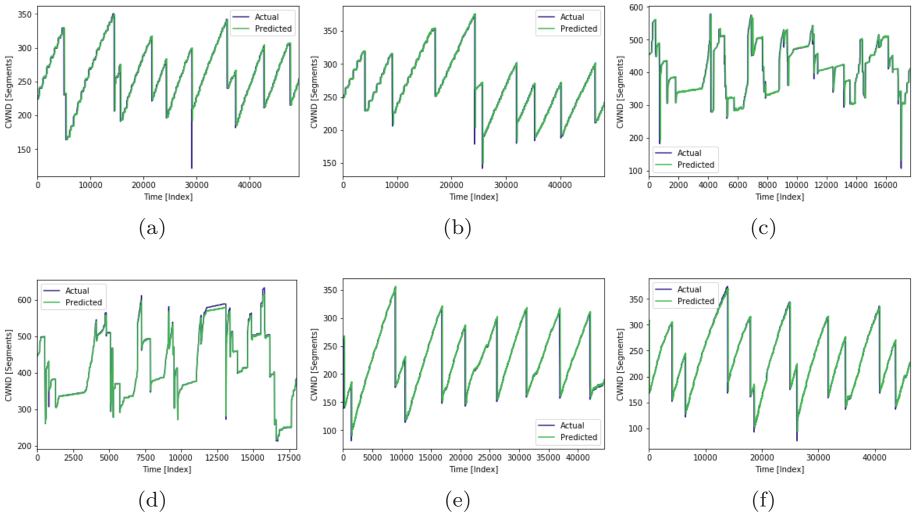
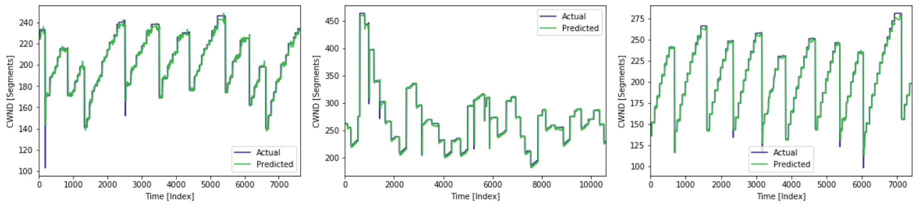


Figure IV.7: TCP *cwnd* prediction from a *realistic* scenario setting on different *zones* of *Google Cloud* platform (East coast USA (North Carolina) and Northeast Asia (Tokyo, Japan) sites). **(a)** CUBIC [9], USA site. **(b)** CUBIC [9], Japan site. **(c)** BIC [32], USA site. **(d)** BIC [32], Japan site. **(e)** Reno [15], USA site. **(f)** Reno [15], Japan site.

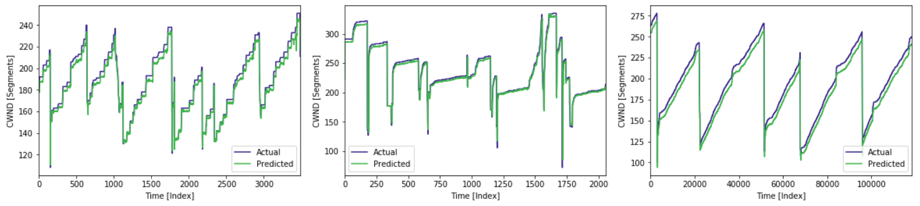
scenarios, we believe it is necessary to carefully test how well our model using an emulated network works with realistic scenarios by leveraging the knowledge of the emulated network. This guarantees that our prediction model is able to discern the results to unforeseen scenarios. In this experimental scenario, the prediction model is trained where the passive monitor is placed between the sender and the receiver.

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



(a) (b) (c)

Figure IV.8: TCP *cwnd* prediction with different configurations in a *combined network* setting. **(a)** CUBIC [9] Configuration $C_1$, **(b)** BIC [32] Configuration $B_1$, **(c)** Reno [15] Configuration $R_1$



(a) (b) (c)

Figure IV.9: TCP *cwnd* prediction across different Google Cloud settings where the *passive monitor* is closer to the *receiver*. **(a)** CUBIC [9] USA Zone, $C_{Receiver}$, **(b)** BIC [32] USA Zone $B_{Receiver}$, **(c)** Reno [15] USA Zone, $R_{Receiver}$.

variant with other realistic network traffic scenarios captured from the Internet. To this end, we created a realistic testbed where we experiment from *Google Cloud* platform nodes by running our resources on the East coast of the USA and Japan as shown in Figure IV.7. In order to create a realistic TCP session, we uploaded a big *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 IV.7, we confirm that our prediction model operates correctly and accurately recognizes the sawtooth pattern for realistic scenario settings across different *Google Cloud zones*. This shows that our prediction model is *general* bearing similarity to the concept of transfer learning in the machine learning community. The *cwnd* prediction performance result of the realistic scenario setting across the *Google Cloud* platforms is presented in Table IV.6. As it is shown in Table IV.8, the accuracy of the TCP variant prediction for this scenario setting is *96.66*%.

Table IV.6: Prediction of *cwnd* on a *realistic* scenario.

| **TCP Algorithms** | **Google Cloud Zone** | **RMSE** | **MAPE (%)** |
|---|---|---|---|
| CUBIC | *USA* Zone | 1.752 | 2.517% |
| | *Japan* Zone | 1.964 | 2.852% |
| BIC | *USA* Zone | 2.219 | 2.979% |
| | *Japan* Zone | 2.527 | 3.097% |
| Reno | *USA* Zone | 2.057 | 3.143% |
| | *Japan* Zone | 2.975 | 2.861% |

Table IV.7: TCP Variant Prediction of a *realistic scenario* setting: Confusion Matrix.

| | **Predicted** | | |
|---|---|---|---|
| **Actual** | BIC | CUBIC | Reno |
| BIC | 20 | 0 | 0 |
| CUBIC | 0 | 19 | 1 |
| Reno | 0 | 1 | 19 |

Table IV.8: TCP Variant Prediction of a *realistic scenario* setting: Performance metrics.

| | **Precision** | **Recall** | **F1-Score** | **Support** |
|---|---|---|---|---|
| BIC | 1.00 | 1.00 | 1.00 | 20 |
| CUBIC | 0.95 | 0.95 | 0.95 | 20 |
| Reno | 0.95 | 0.95 | 0.95 | 20 |
| Average/Total | 0.97 | 0.97 | 0.97 | 60 |
| **Accuracy** | **0.9666** | | | |

### IV.6.3 Intermediate Node Closer to the Receiver Scenario

Our experimental setup for this scenario setting across different *Google Cloud zones* is presented in Figure IV.10. It is fundamentally difficult to infer the sender's *cwnd* accurately from passive measurements collected 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 is less reliable and may introduce more opportunities for prediction algorithms to make false inferences. In this scenario, the number of hops are 18 with an average RTT of *137ms* whereas in the *emulated* scenario, the number hops are 3 with an average RTT of *1.8ms*. We believe the data wouldn't reveal what additional packets are in flight from the sender, or which ACKs from the receiver have been received. 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. As it is shown in Figure IV.9, we can see that our prediction model correctly recognizes the sawtooth pattern of the *cwnd*. However, as shown in Table IV.9, the prediction error is relatively higher as compared to the other scenario settings. This is because of the cases mentioned earlier. For predicting the underlying TCP variant, we can use the same evaluation methodology, applied on the other presented scenario settings, based on measuring the change in *cwnd* size.

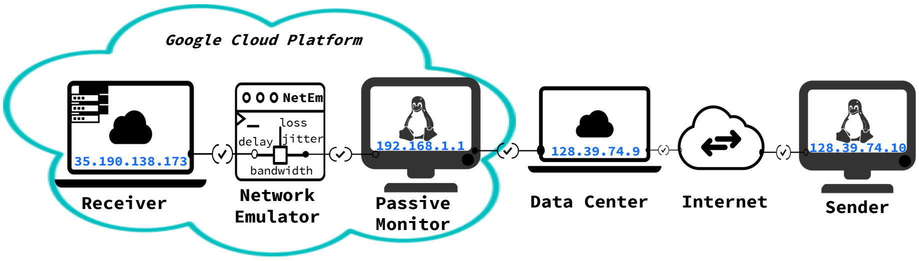


Figure IV.10: Intermediate node closer to the receiver scenario setup.

Table IV.9: Prediction of *cwnd* across different Google Cloud Zones when the monitor is closer to the receiver.

| TCP Algorithms | Google Cloud Zones | RMSE | MAPE (%) |
|---|---|---|---|
| CUBIC | $USA\ Zone,\ C_{Receiver}$ | 6.341 | 9.057% |
| BIC | $USA\ Zone,\ B_{Receiver}$ | 5.185 | 8.680% |
| Reno | $USA\ Zone,\ R_{Receiver}$ | 6.937 | 9.238% |

### IV.6.4 Combined Scenario Setting

Real networks behave in a more complex manner than emulated networks. The TCP control loop affects the *loss* and *delay* of packets. We believe, there are queue dynamics in the network which cause packet trains and other behaviors which software emulators like *NetEm* [13] can't reproduce well enough. In Section IV.6.2, 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 IV.11.

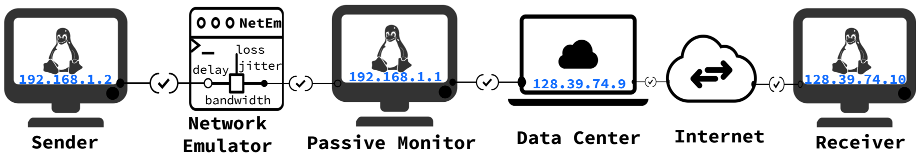


Figure IV.11: Combined scenario setup.

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 IV.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 IV.12, is *94.44*% and this justifies that our prediction model can handle multiple scenario settings.

Table IV.10: Prediction of *cwnd* on a *combined* setting.

| **TCP Algorithms** | **Per Configuration** | **RMSE** | **MAPE (%)** |
|---|---|---|---|
| CUBIC | *Sample Configuration* $C_1$ | 2.072 | 3.262% |
| BIC | *Sample Configuration* $B_1$ | 3.506 | 4.846% |
| Reno | *Sample Configuration* $R_1$ | 2.096 | 3.829% |

Table IV.11: TCP Variant Prediction of a *combined scenario* setting: Confusion Matrix.

| | **Predicted** | | |
|---|---|---|---|
| **Actual** | BIC | CUBIC | Reno |
| BIC | 33 | 0 | 0 |
| CUBIC | 2 | 33 | 0 |
| Reno | 1 | 3 | 36 |

Table IV.12: TCP Variant Prediction of a *combined scenario* setting: Performance metrics.

| | **Precision** | **Recall** | **F1-Score** | **Support** |
|---|---|---|---|---|
| BIC | 0.92 | 1.00 | 0.96 | 33 |
| CUBIC | 0.92 | 0.94 | 0.93 | 35 |
| Reno | 1.00 | 0.90 | 0.95 | 40 |
| Average/Total | 0.95 | 0.94 | 0.94 | 108 |
| **Accuracy** | **0.9444** | | | |

**Transfer Learning:** In our work, we are able to train in one scenario setting and apply it as a pre-training in another scenario setting. Therefore, we are able to show that the learned prediction model by leveraging a trained knowledge from the *emulated* network performs reasonably well as it is shown above when it is applied and transferred to a *realistic* scenario setting bearing similarity to the concept of *transfer learning* in the machine learning community [26].

**Optimality:** As it is shown in Tables IV.13 and IV.14, the experimental results show that our LSTM-based prediction model is able to outperform our previous approach using machine learning techniques [10]. Our LSTM-based TCP variant prediction model achieves accuracies of *97.22%*, *96.66%* and *94.44%* on the *emulated*, *realistic* and *combined* scenario settings, outperforming the standard ML-based which yields accuracies of *93.51%*, *95%* and *91.66%* respectively.

Table IV.13: TCP *cwnd* prediction comparison.

| **Scenario Settings** | **TCP Algorithms** | **Configuration** | **Techniques** | | | |
|---|---|---|---|---|---|---|
| | | | *Machine Learning* | | *LSTM* | |
| | | | RMSE | MAPE | RMSE | MAPE |
| *Emulated* | CUBIC | $C_1$ | 5.839 | 6.953% | 2.181 | 2.846% |
| | | $C_2$ | 3.075 | 3.725% | 2.855 | 3.103% |
| | Reno | $R_1$ | 3.511 | 3.140% | 2.013 | 2.815% |
| *Realistic* | CUBIC | USA | 4.265 | 5.134% | 1.752 | 2.517% |
| | | Japan | 3.522 | 4.738% | 1.964 | 2.852% |
| | BIC | USA | 2.952 | 3.809% | 2.219 | 2.979% |
| | | Japan | 2.694 | 3.761% | 2.527 | 3.097% |
| | Reno | USA | 3.170 | 5.068% | 2.057 | 3.143% |
| | | Japan | 3.396 | 5.197% | 2.975 | 2.861% |

Table IV.14: TCP variant prediction accuracy comparison.

| | **Scenario Settings** | | |
|---|---|---|---|
| **Techniques Accuracy** | Emulated | Realistic | Combined |
| Machine Learning-based | 93.51% | 95% | 91.66% |
| LSTM-based | 97.22% | 96.66% | 94.44% |

## IV.7 Conclusion and Future Work

In this paper, we have demonstrated the capability of a deep neural network architecture based on a learning LSTM recurrent predictive models to capture the pattern of a TCP *cwnd* with small prediction errors from passive traffic collected at an intermediate node. We have also uniquely identified the underlying TCP variants based on the *multiplicative decrease* window of the *cwnd* and the per-connection states within the variant from passive measurements. Our goal in this work was to implement a learning predictive model that generates the pattern of *cwnd* from passive measurements using an LSTM architecture and finally justify if our previous machine learning-based experiments are valid. The experimental results show the effectiveness of our LSTM-based prediction approach. We found out that our LSTM-based model outperforms our previous work carried out using the *state-of-the-art* machine learning-based prediction models by a reasonably significant margin. We show that the learned prediction model by leveraging knowledge from the *emulated network* performs reasonably well when it is applied on a *real-life scenario* setting bearing similarity to the concept of transfer learning in the machine learning community. Finally, we believe that our work can open up the path to a number of future research work directions in the computer networking community.

In this work, we consider only *loss-based* TCP congestion control algorithms that consider packet loss as an implicit indication of congestion by the network for a proof of concept. By design, unlike *loss-based* algorithms, the *multiplicative decrease* parameter of *delay-based* congestion control algorithms is not fixed which makes it fundamentally challenging to predict the TCP variant from passive traffic when there is variability in *delay*. As a future work, it would be interesting to develop a *delay-based* model 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 passively measured traffic and real measurements over the Internet. We plan to investigate these issues further and extend the approaches in our future work.

**Acknowledgements.** We greatly acknowledge the anonymous reviewers for their helpful feedback and detailed comments on our paper.

## IV.8 References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[2] Z. Che, S. Purushotham, K. Cho, D. Sontag, and Y. Liu. Recurrent neural networks for multivariate time series with missing values. *Scientific reports*, 8(1):6085, 2018.

[3] D.-M. Chiu and R. Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Computer Networks and ISDN systems*, 17(1):1–14, 1989.

[4] K. Choi, G. Fazekas, M. Sandler, and K. Cho. Convolutional recurrent neural networks for music classification. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 2392–2396. IEEE, 2017.

[5] J. T. Connor, R. D. Martin, and L. E. Atlas. Recurrent neural networks and robust time series prediction. *IEEE transactions on neural networks*, 5(2):240–254, 1994.

[6] N. Dukkipati, Y. Cheng, and A. Vahdat. Research Impacting the Practice of Congestion Control, 2016.

[7] ESnet. iperf3. https://iperf.fr/iperf-servers.php, 2017.

[8] A. Graves and N. Jaitly. Towards end-to-end speech recognition with recurrent neural networks. In *International conference on machine learning*, pages 1764–1772, 2014.

[9] S. Ha, I. Rhee, and L. Xu. CUBIC: a new TCP-friendly high-speed TCP variant. *ACM SIGOPS operating systems review*, 42(5):64–74, 2008.

[10] D. H. Hagos, P. E. Engelstad, A. Yazidi, and Ø. Kure. A machine learning approach to TCP state monitoring from passive measurements. In *2018 Wireless Days (WD)*, pages 164–171. IEEE, 2018.

[11] D. H. Hagos, P. E. Engelstad, A. Yazidi, and O. Kure. Towards a Robust and Scalable TCP Flavors Prediction Model from Passive Traffic. In *2018 27th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–11. IEEE, 2018.

[12] D. H. Hagos, A. Yazidi, Ø. Kure, and P. E. Engelstad. Enhancing security attacks analysis using regularized machine learning techniques. In *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*, pages 909–918. IEEE, 2017.

[13] S. Hemminger et al. Network emulation with NetEm. 2005.

[14] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[15] V. Jacobson. Congestion avoidance and control. In *ACM SIGCOMM computer communication review*, volume 18, pages 314–329. ACM, 1988.

[16] S. Jaiswal, G. Iannaccone, C. Diot, J. Kurose, and D. Towsley. Inferring tcp connection characteristics through passive measurements. In *IEEE INFOCOM 2004*, volume 3, pages 1582–1592. IEEE, 2004.

[17] D. Katabi, M. Handley, and C. Rohrs. Congestion control for high bandwidth-delay product networks. *ACM SIGCOMM computer communication review*, 32(4):89–102, 2002.

[18] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[19] S. Lai, L. Xu, K. Liu, and J. Zhao. Recurrent convolutional neural networks for text classification. In *Twenty-ninth AAAI conference on artificial intelligence*, 2015.

[20] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[21] P. Liu, X. Qiu, and X. Huang. Recurrent neural network for text classification with multi-task learning. *arXiv preprint arXiv:1605.05101*, 2016.

[22] A. Medina, M. Allman, and S. Floyd. Measuring the evolution of transport protocols in the internet. *ACM SIGCOMM Computer Communication Review*, 35(2):37–52, 2005.

[23] C. Olah. Understanding LSTM Networks. https://colah.github.io/posts/2015-08-Understanding-LSTMs, 2015.

[24] S. Ostermann. Tcptrace. http://www. tcptrace. org, 2000.

[25] J. Pahdye and S. Floyd. On inferring TCP behavior. *ACM SIGCOMM Computer Comm. Review*, 31(4):287–298, 2001.

[26] S. J. Pan and Q. Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2010.

[27] S. Rewaskar, J. Kaur, and D. Smith. A Passive State-Machine Based Approach for Reliable Estimation of TCP Losses. 2006.

[28] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[29] I. Sutskever. Training recurrent neural networks. *University of Toronto, Toronto, Ont., Canada*, 2013.

[30] I. Sutskever, J. Martens, and G. E. Hinton. Generating text with recurrent neural networks. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 1017–1024, 2011.

[31] D. Tang, B. Qin, and T. Liu. Document modeling with gated recurrent neural network for sentiment classification. In *Proceedings of the 2015 conference on empirical methods in natural language processing*, pages 1422–1432, 2015.

[32] L. Xu, K. Harfoush, and I. Rhee. Binary increase congestion control (BIC) for fast long-distance networks. In *IEEE INFOCOM*, volume 4, pages 2514–2524. IEEE, 2004.

[33] P. Yang, J. Shao, W. Luo, L. Xu, J. Deogun, and Y. Lu. TCP congestion avoidance algorithm identification. *IEEE/Acm Transactions On Networking*, 22(4):1311–1324, 2014.

Paper V

# A Deep Learning Approach to Dynamic Passive RTT Prediction Model for TCP

**Desta Haileselassie Hagos**[1], **Paal E. Engelstad, Anis Yazidi, Carsten Griwodz**

***Published*** in the *38th IEEE International Performance Computing and Communications Conference (IPCCC 2019)*. IEEE, 2019.
DOI: 10.1109/IPCCC47392.2019.8958727.

## Abstract

The Round-trip Time (RTT) is a property of the path between a sender and a receiver communicating with Transmission Control Protocol (TCP) over an IP network and over the public Internet. The end-to-end RTT value influences significantly the dynamics and performance of TCP, which is by far the most used communication protocol. Thus, in communication networks, RTT is an important network performance variable. By measuring the traffic at an intermediate node, a network operator or service provider can estimate the RTT and use the estimation to study and troubleshoot the per-connection characteristics and performance. This paper aims at improving the accuracy and timeliness of the RTT estimation, to help network operators improving their analysis. We propose and evaluate a novel deep learning-based model capable of dynamically predicting at real-time the RTT between the sender and receiver with high accuracy based on passive measurements collected at an intermediate node, taking advantage of the commonly used TCP *timestamps*. We validate extensively our prediction methodology in a controlled *experimental testbed* and in a *realistic* scenario on the *Google Cloud platform*. We show that our model, which is based on classical deep learning algorithms, gives reasonably effective *state-of-the-art* performance results across multiple TCP congestion control variants. We also show that the model works well for transfer learning. Even though the RTT prediction model was trained on an emulated network, it performs well also when applied to a realistic scenario setting, as demonstrated in our experimental evaluation.

[1]University of Oslo, Department of Informatics, destahh@ifi.uio.no

**Keywords**: TCP, RTT Prediction, LSTM, Passive Measurements

## V.1 Introduction and Motivation

Passive measurement techniques of TCP flows have gained much attention in the networking research community lately [3, 7, 9, 25]. The main reason is that such measurements are becoming increasingly useful for network operators and Internet Service Providers (ISPs) to evaluate the communication performance of applications and services running on their network. Monitoring the traffic at an intermediate node, allows the ISP to assess the underlying network performance, which is crucial for their operation. The RTT is one of the most important indicators of communication performance. The RTT is a TCP state variable that influences congestion control in many TCP variants, and the RTT has a huge influence on the performance of the end-to-end communication. In order for network access providers to determine and diagnose application performance issues on the public Internet, knowledge about the characteristics of the network is a very important factor. The ability to passively compute and dynamically predict the RTT is very crucial for a lot of reasons. For example, it allows network operators to measure and optimize the network performance of real-time applications and services, and it helps providers understand the responsiveness, availability of their network services, performance and predict the behavior of a TCP connection. Network operators and service providers care about RTT, and they even make RTT a Service Level Agreement (SLA) parameter in legal contracts with their customers [27]. Customers who demand better services can *passively* detect the occurrence of SLA violations and this ability would allow the network providers to quickly respond to Quality of Service (QoS) problems [27]. This is particularly important for the quality of *latency-sensitive* and *bandwidth-intensive* real-time media applications (such as video, audio, and application sharing), etc. [27]. RTT is the length of time it takes for an outgoing TCP client packet plus the minimal time spent for an acknowledgment of that segment from the server to be received by the client [23]. The RTT between the sending and receiving endpoints is typically a combination of a fixed *BaseRTT*, a fixed *propagation time*, and the amount of queueing that is experienced along the path. Thus, the changes in the RTT might give an indication of changes in queuing and the congestion in the network, and be a useful input to the TCP congestion control algorithm.

TCP is a highly reliable connection-oriented transport protocol capable of adding reliability and preventing excessive congestion on the Internet [17]. Note that congestion control in TCP was not part of the protocol initially until the first Internet congestion collapse was observed [16]. TCP controls congestion by also aiming for fair sharing of the available network resources by the competing flows, using strategies empowered by TCP [17]. TCP fairness means that if $N$ TCP sessions share the same bottleneck link of a bandwidth $B$, each session should ideally get an average rate of $\frac{B}{N}$ and $\frac{1}{N}$ of the available link capacity assuming that all the active TCP connections have the same increase of rates and similar RTTs. If the multiple TCP sessions have different RTTs but share the same bottleneck link, the flows with larger RTTs usually achieve lower throughput, while the flows having smaller RTT may utilize the bandwidth more aggressively than the others [13]. Indeed, the RTT directly influences the TCP throughput according to the following equation:

$$T_i \propto \frac{1}{\sqrt{p_i}.RTT_i} \tag{V.1}$$

where $T_i$ is the throughput, $p_i$ is the probability of a packet loss rate, and $RTT_i$ is RTT of a TCP flow $i$. Equation V.1 shows that the throughput ratio of individual TCP connections is inversely proportional to the RTT [29]. This means that RTT is one of the most important state variables that determine the aggressiveness of a TCP flow. This also means that passively predicting RTT is useful for the deployed TCP variants to optimize for high bandwidth by leveraging the TCP *timestamps* option carried in each TCP header. Evaluating the RTT, inflated by queueing across the network [29], may also give a more detailed view of the sender state than merely the throughput, as the RTT also influences the *Retransmission Timeout (RTO)* of an active TCP session and the *Congestion Window (cwnd)* size [19]. The *cwnd* is also one of the most important TCP per-connection state variables. The *cwnd* is a TCP per-connection state internal variable that represents the maximum amount of data a sender can potentially transmit per RTT at any given point in time based on the sender's network capacity and conditions. TCP decides the maximum number of *bytes* that can be *outstanding* without being acknowledged at any time maintained independently by the sender.

**Benefits**: It is very natural to ask: *why RTT prediction performed in an intermediate node from passive measurements is important*? In addition to the reasons we address above, there are myriad reasons we may want to use passive RTT measurements. Passive RTT prediction in an intermediate node is important, for example, when *(i)* We have no control over either end-host of communication so we can't launch active measurements from either host, but want to know the RTT between them. *(ii)* We want to know the RTTs of the actual communication occurring on the Internet, and not the RTT between a pair of hosts artificially picked. *(iii)* The TCP active probes used in active measurements (such as *ping* messages) are blocked by firewalls etc. For more details about the difference between *active* and *passive* measurement techniques, we refer the reader to our previous work [12].

**Recurrent Neural Networks (RNN) models**: In this paper, we are interested in the capabilities and potentials of *RNN* models for implementing our passive RTT prediction model for TCP using *timestamps* and *timestamp echoes* [18]. Hence, we have explored an approach to dynamically predict an end-to-end RTT for TCP from passive measurements using *Long Short-Term Memory (LSTM)*-based RNN architecture. As described in Section V.3, different approaches have been proposed to estimate RTT from passive measurements. However, we believe that no previous research works have applied *deep learning* models to estimate RTT in relation to TCP from passive measurements. To the best of our knowledge, this paper is the first to study the applicability of LSTM for passive RTT measurement schemes in real-time.

RNNs are powerful neural sequence models that achieve *state-of-the-art* performance on sequential, time-dependent prediction and classification tasks. However, when the input sequence is very long, RNNs have a significant limitation of *gradient vanishing*. LSTM [15] is a special kind of RNN introduced with the purpose of overcoming this shortcoming of RNNs. LSTM has the ability to solve the *vanishing gradient* problem by dynamically controlling the information flow within the layers through its *memory blocks* and capture the long-term dependencies of the connections in a sequence effectively [15]. In an LSTM model, we consider a time-series prediction task of length $n$ producing an output $y_t$ at each time-step t$\in\{1, 2, 3, ..., T\}$ by mapping a temporal input feature vector sequence $x_1^n = (x_{(1)}, x_{(2)}, x_{(3)}, ..., x_{(\text{n-1})}, x_{(\text{n})})$ where $x_\text{i} \in \mathbb{R}^n$ to a corresponding output vector sequence $y_1^n = (y_{(1)}, y_{(2)}, y_{(3)}, ..., y_{(\text{n-1})}, y_{(\text{n})})$ where $y_\text{i} \in \mathbb{R}^n$ by calculating the network unit activations of a weighted sum using the Equations V.2-V.7 iteratively from *t = 1* to *n*. As it is shown in Equations V.2, V.3, and V.5, LSTM [15] uses *three*, an *input*, *forget* and *output*, gates shared by all cells in the LSTM block in order to learn long-term dependencies and control the flow of information. The *input* gate determines the flow of input activation into the memory cell whereas the *output* gate determines the output flow of cell activation into the rest of the network. The *forget* gate determines the extent to which the current value remains in the memory cell of the LSTM unit before it gets gradually discarded when its data is no longer needed.

$$i_t = \sigma(W_{ix}x_t + W_{im}m_{t-1} + W_{ic}c_{t-1} + b_i) \tag{V.2}$$
$$f_t = \sigma(W_{fx}x_t + W_{fm}m_{t-1} + W_{fc}c_{t-1} + b_f) \tag{V.3}$$
$$c_t = f_t \odot c_{t-1} + i_t \odot g(W_{cx}x_t + W_{cm}m_{t-1} + b_c) \tag{V.4}$$
$$o_t = \sigma(W_{ox}x_t + W_{om}m_{t-1} + W_{oc}c_t + b_o) \tag{V.5}$$
$$m_t = o_t \odot h(c_t) \tag{V.6}$$
$$y_t = \phi(W_{ym}m_t + b_y) \tag{V.7}$$

where the $i$, $f$, $c$, $o$ are *input*, *forget*, *memory state*, and *output* gate activation vectors respectively at each time step $t$. $\sigma$ is the logistic sigmoid function while $\odot$, $g$ and $h$ are element-wise product of the vectors, the cell input and output non-linearity activation functions of the entire neural network applied to each layer of the deep network respectively. $W$ and $b$ represents a vector of weighted recurrent connections and the bias vector. $m_t$ is the hidden state output of the LSTM layer. Finally, $\phi$ is the activation function in the hidden layer applied to the network output. Figure V.1 describes the basic unit of an LSTM network where the input sequence to the LSTM cell is carried over each time-step of *t-1*, *t* and *t+1*. $C_t$ and $C_{t-1}$ are the memory cell state activation vectors from the current and previous blocks at time $t$ and *t-1* respectively.

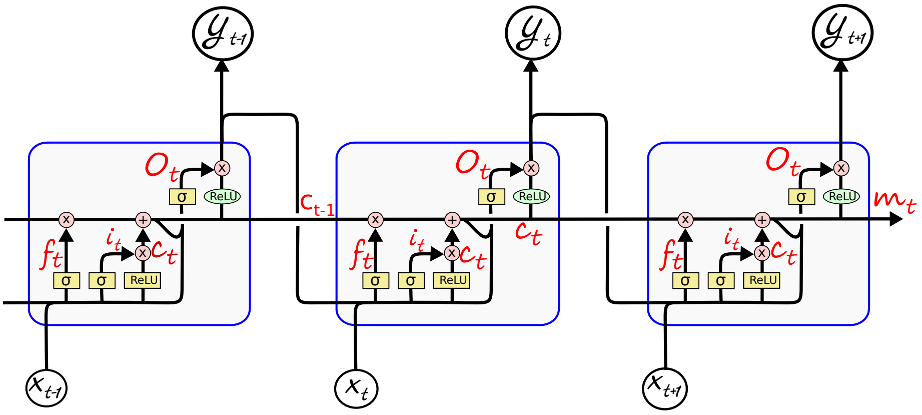


Figure V.1: LSTM Networks. For more thorough details, refer [28].

**Why did we use deep learning**? As explained above in Section V.1, both *cwnd* and RTT are TCP state variables relevant to congestion control. However, neither the value of *cwnd* nor the value of RTT is contained in the TCP header. The *cwnd* size is stored in the memory of the TCP sender, and the RTT is a product of the varying behavior of the network between the TCP sender and receiver. Therefore, trying to predict these values somewhere other than at the TCP sending node is challenging. Deep learning techniques have found a great success in multiple areas of research. In our case, let's consider a situation where a network model is trained for a specific intermediate node which has been trained for a specific bandwidth, background load, multiplexing rate, and a multitude of different router conditions, can predict well for exactly this node. Hence, we want a model that is able to train in one scenario setting and apply it as a pre-training on another setting by leveraging trained knowledge. As it is presented in Section V.6, this paper proofs that it makes sense in principle to use learning algorithms for TCP state predictions.

**Contributions**: We summarize our main contributions below.

- We present a dynamic *deep learning*-based approach for RTT prediction in relation to TCP from passive measurements collected at an intermediate node.
- We identify the main challenges in the *passive* estimation of RTT across a broad range of network conditions.
- We show that the learned prediction model performs reasonably well by leveraging trained knowledge from the *emulated network* when it is applied and transferred on a *real-life scenario* setting.
- We demonstrate the benefits and explore the applicability of our prediction model using an *LSTM* architecture.

- We experimentally validate our prediction model extensively through several controlled experiments across an *emulated* and *realistic* settings.

## V.2 Background

RTT measurements are used in congestion control algorithms to determine connection timeouts. *Delay-based* congestion control algorithms use the measured *RTT* as an implicit feedback to control congestion, and they adjust the *cwnd* size according to the queuing *packet delay* instead of packet loss [20]. These algorithms increase the *cwnd* size *quickly* when the queuing delay is low and decreases the *cwnd slowly* when the delay is high. Different *rate* and *delay-based* 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. Hence, the following are a list of the most widely used TCP variants we consider in our analysis to cover the whole scope of the problem.

1. **TCP Westwood**: Westwood [10] is a sender-side modification of the traditional TCP algorithm [17]. At the time of congestion triggered in response to RTO or *triple DupACKs*, TCP Westwood [10] estimates the available end-to-end per-connection bandwidth by monitoring the flow of returning ACK rates instead of packet loss and sets the *cwnd* size equal to the measured bandwidth which helps to avoid too much reduction of the *cwnd*.

2. **TCP-Vegas**: Vegas [5], instead of packet loss, uses a measured RTT as congestion feedback and hence it attempts to accurately tune the *cwnd* by using the measured *BaseRTT* of every segment sent and reacting to changes in it by altering the *cwnd*.

3. **BBR**: BBR is an emerging TCP delay-controlled congestion control algorithm from *Google* fully deployed across all *Google* TCP services and the B4 Wide Area Network (WAN) backbone connections [6]. Unlike the traditional congestion algorithms, BBR doesn't overreact to packet loss. Instead, it reacts to actual congestion and relies on maximizing the throughput with minimal queue by sequentially probing and periodically estimating the underlying available bottleneck bandwidth and minimum path RTT in a similar fashion as TCP Vegas [5].

**Roadmap**: The rest of this paper is organized as follows. Next, in Section V.3, we summarize the related work in the literature considered as a *state-of-the-art*. In Section V.4, we describe our controlled experimental setup for the evaluation. Section V.5 gives an overview of our methodology highlighting the practical challenges and considerations. Section V.6 presents the validation scenario settings of our prediction model. The experimental results and discussion are presented in detail in Section V.7. Finally, Section V.8 concludes the paper and outlines directions of research for future extensions.

## V.3 Related Work

Our work benefits from a wide range of existing passive measurement related research works in computer networking.

**TCP RTT Measurement**: TCP implements a retransmission strategy by setting the *time-out* interval to ensure data delivery in the absence of any acknowledgment for a particular segment from the receiver side [30]. The timer relies on the measurement of the network latency which TCP does by periodically estimating the current RTT of every active connection in order to determine the RTO when it sends data and receiving an acknowledgment for it. Accurate measurement of RTO is crucial to TCP performance and it is determined by estimating the *mean* and a *variance* of the estimated RTT [30]. When the timer RTO expires, the segment is retransmitted. To compute the current RTO, TCP sender keeps track of the *Smoothed Round-Trip Time (SRTT)* and the *Round-Trip Time Variation (RTTVAR)* state variables. When the first RTT measurement $R$ is made on the active connection, the host should compute the following *Jacobson RTO Estimation algorithm* [17].

$$\mathtt{SRTT} = \mathtt{R} \tag{V.8}$$

$$\mathtt{RTTVAR} = \mathtt{R}/2 \tag{V.9}$$

$$\mathtt{RTO} = \mathtt{SRTT} + \mathtt{max}(\mathtt{G}, \mathtt{K} * \mathtt{RTTVAR}) \tag{V.10}$$

However, when a subsequent RTT measurement $RTT'$ is made, the host should compute the following algorithm [17].

$$\mathtt{RTTVAR} = (1-\beta) * \mathtt{RTTVAR} + \beta * |\mathtt{SRTT} - \mathtt{RTT'}| \tag{V.11}$$

$$\mathtt{SRTT} = (1-\alpha) * \mathtt{SRTT} + \alpha * \mathtt{RTT'} \tag{V.12}$$

$$\mathtt{RTO} = \mathtt{SRTT} + \mathtt{max}(\mathtt{G}, \mathtt{K} * \mathtt{RTTVAR}) \tag{V.13}$$

**Understanding RTO**: In a typical implementation, TCP computes the RTT of an active connection using the *Exponential Weighted Moving Average (EWMA)* estimator [17]. As it is used in TCP RTT computation implementations, the *SRTT* is also updated using the *EWMA* estimator as it is shown in Equation V.12 where the *smoothing* factor $(\alpha) = \frac{1}{8}$ [30]. RTTVAR is also calculated using *EWMA* as shown in Equation V.11 where the smoothing gain of the samples $\beta$ (*variance factor*) $= \frac{1}{4}$. The new value of RTO is given in Equation V.13 as a function of *SRTT* and *RTTVAR* where $K$ is usually 4 and $G$ is a clock granularity in *seconds*. The TCP sender, dynamically adjusted based on the estimated RTT, keeps a timer which activates retransmission of packets that have not been acknowledged before the RTO expires [30]. After computing the RTO, if its value is less than *1 second*, then the RTO value should be rounded up to *1 second* [30]. However, the *timeout* can expire spuriously across low-bandwidth network paths and triggers unnecessary retransmissions when no packets have been lost [11]. Modern operating systems like *Linux* have a minimum value for RTO in order to avoid unnecessary high retransmission delays of an open active connection. The potential pitfall of choosing a low RTO, however, is that it may trigger retransmission of a packet even though the segment is received and an ACK is on its way. Setting a low value for RTO works better when there is a moderate background traffic [26]. For more technical descriptions and the rules governing the measurement of *SRTT*, *RTTVAR*, and *RTO* refer [30].

**Algorithms for Avoiding Spurious Timeouts**: To address this problem, a number of approaches have been proposed. For example, RTO estimators like [30] are based on the assumptions of older technologies. As described earlier, spurious timeouts lead to problems that cause several unnecessary retransmissions and congestion control back-off that affect the TCP throughput. In addition to this, estimating the RTT measurements are challenging in the presence of timeouts and packet loss in the end-to-end path. This is because on the receipt of an *ACK* after *R* retransmissions, the sender cannot tell which one of the *R+1* data sent is being acknowledged which again affects the measurement of *SRTT* shown in Equation V.12. Wrongly computed *SRTT* values will eventually lead to wrong *RTO* values. If the value of *RTO* is too *small*, it will lead to unnecessary retransmission of data segments which again increases the load on the underlying network capacity. But if the value of *RTO* is too *large*, the sender waits too long before retransmitting lost segments which again increases delay and lowers the throughput for connections with packet loss. Making use of the TCP *timestamps*, the *Eifel* [11] algorithm has pointed out that it is possible to detect spurious TCP timeouts problems and recover by restoring a TCP sender's congestion control state saved before the timeout.

There are other previous research works who have examined and reported RTT estimation for TCP [2, 21, 22]. The approach presented in [22] uses a *unidirectional* flow during the TCP *handshake* of a connection to estimate RTT using the time from *SYN* to *SYN+ACK* method. The approaches proposed in [22] calculates one RTT sample per TCP connection associated either during the *three-way handshake* or during the *slow-start* phase. If we have captured the TCP *three-way handshake* as presented in [22], we can calculate the *initial RTT* (*iRTT*) by taking the time difference from the *SYN* packet to the *ACK* packet of the handshake. However, since the TCP handshake packets are processed by the kernel, the RTTs during the data transfer will probably be slightly larger than the *iRTT*. Hence, this approach may tend to underestimate the actual RTT. In addition to this, since TCP sets the initial *retransmission timeout* value to *3 seconds* [30], therefore this approach is not applicable in scenarios where the TCP connection setup takes longer which leads to long delays and packet losses introduced by the network. The study in [2] has reported a statistical characterization of RTT variability where the measurement point is closer to the sender. However, their study does not take *delayed ACKs* into account. The authors of [21] have introduced an approach for RTT measurements of TCP connections based on *bidirectional* traces captured at the monitoring point using a *Finite State Machine (FSM)* that *replicates* the TCP sender states of observed

ACKs depending on the underlying TCP flavor. The authors have pointed out that the estimation of the TCP parameters (e.g., *cwnd*) may have potential errors primarily due to *over-estimation* of the RTT and incorrect window sizes of a connection [21]. Another limitation of this work, given differences of the many existing flavors of TCP stack implementations, the use of a separate state machine for each TCP variant is *unscalable* and we also believe that the constructed *replica* may not manage to reverse or backtrack the transitions taking the 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. Our *deep learning*-based approach to passively predict the continuous RTT measurement throughout the lifetime of a TCP session builds upon these classical approaches by avoiding the limitations taking advantage of the commonly used *timestamp* option as explained in Section V.5.

## V.4 Experimental Evaluation

### V.4.1 Testbed Hardware

We have carried out our experiments using a cluster of HPC machines based upon the GNU/Linux operating system running a modified version of the *4.15.0-39-generic* kernel release. The prediction 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, 128 GB RAM, 12 CPU cores running under Linux 64-bit. All nodes in the cluster are connected to a low latency 56 Gbit/s Infiniband, gigabit Ethernet and have access to 600 TiB of *BeeGFS* parallel file system storage.

### V.4.2 Passive RTT Monitoring Methodology and Trace Analysis

*Passive* measurement methodology is a technique of tracking the behavior and characteristics of packet streams where the network is not influenced by injecting extra traffic. More details on the two types of network measurement technique categories (i.e., *active* and *passive*) are briefly described in [12]. The RTT seen by a TCP segment is defined as the time a sender waits until it receives a corresponding *ACK* from the receiver before it sends more data packets. In this paper, we are interested in presenting a *deep learning-based* RTT prediction model using a packet statistics passively monitored between the sender and receiver endpoints of a network. In order to increase the RTT measurement precision, the *timestamps* option which every TCP segment carries in the header field is used in our methodology. When the server receives a data segment, it copies the *timestamps* into the ACK and this, in turn, enables the client to compute the RTT accurately for every acknowledged data segment.

**Trace Analysis**: To evaluate our prediction model and perform our analysis on both the *emulated* and *realistic* network conditions, we have generated our own dataset. In order to capture all sessions on the network when the client and server are sending TCP packets and measure the TCP data packets from both directions, we have used the fully controlled experimental setup shown in Figure V.2. The data passively collected at an intermediate node is fed into a model that can be trained in another context, e.g., an emulated scenario as discussed in Section V.6. The background traffic for all our experiments are generated using the *iperf* [8], an open source TCP streaming benchmark, traffic generator on an emulated LAN link where we run each TCP variant by adding a configurable variation of the emulation parameters *bandwidth (in Mbit/s)*, *delay (in ms)*, *jitter (in ms)* and *packet loss (%)* within a flow. The values of configuration parameters of the emulator for our samples collection are presented in Table V.1. The cross-traffic variability and verification of the popular Linux-based network emulator we used, *Network Emulator (NetEm)* [14], are thoroughly addressed in [12].

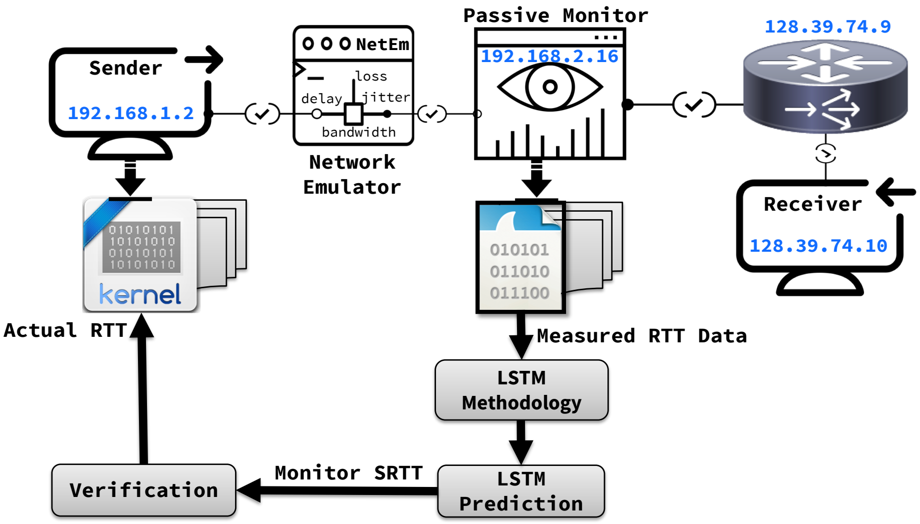


Figure V.2: Controlled Experimental Setup.

**Verification of the Emulator**: Given that the software emulator is not precise, we can ask: can we trust the network emulator for all the variations of *bandwidth*, *delay*, *jitter* and *packet loss* values introduced by the emulator irrespective of the measurement we get from TCP stream? As the precision of the emulator cannot be measured from TCP streams, we set up 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 and we found out that each variation run by the emulator doesn't affect our results. But it is good to remember that emulator experiments are always going to have some differences compared to a real network as different networks behave completely differently.

Table V.1: Network *emulation* parameters.

| **Bandwidth** (*Mbit/s*) | **Delay** (*ms*) | **Jitter** (*ms*) | **Packet loss** (%) |
|---|---|---|---|
| 10 | 1 | 0.001 | 0.01 |
| 100 | 2 | 0.1 | 0.05 |
| 300 | 3 | 0.2 | 0.1 |
| 500 | 5 | 0.5 | 1 |
| 700 | 7 | 1 | 1.5 |
| 1000 | 10 | 2 | 2 |

## V.5 Experimental Methodology

In this paper, we are exploring an approach to *dynamically* and *reliably* predict an end-to-end RTT for TCP from passive measurements using an *LSTM*-based RNN architecture. As illustrated in Figure V.3, there are different techniques to estimate passive RTT values from packet arrival times at an intermediate monitoring point. The *first* and *third* RTT computation techniques shown are the *initial three-way handshake* [22] and the *termination* of a connection phase that carries the *FIN* control flag. The *three-way handshake* method uses a TCP segments association during the initial handshake phase to compute the *minimum RTT* with a small packet burst (so less affected by propagation delay through intermediate devices) but it also has limitations as described in Section V.3. A similar estimation technique can also be applied during the connection termination phase. However, these *two* techniques do not consider continuously estimating RTT through the course of the connection and hence they are *statically* limited to the setup and termination of the TCP connection. In our paper, however, we are interested in the *second* estimation method where we have to account for the cases where there are large packet bursts by *continuously* measuring the TCP data segments sequence and their corresponding ACK for estimating RTTs when they carry data throughout the lifetime of the connection by associating the *timestamps* and *timestamp echoes*. This helps us to dynamically estimate the RTT between the sender and receiver from the perspective of the intermediate node by measuring the streams of RTT samples which can be added to get an end-to-end RTT. Let's simplify this more with an easy to understand example. When the sender, on Figure V.3, sends a TCP data segment, the receiver acknowledges the data segment with an ACK and echoes the sender's *timestamp*. The intermediate node recognizes the sender's *timestamp* in both data segments and associates the data segments with *timestamps* matching. When the sender receives an ACK, it sends more data packets by echoing the receiver *timestamps*. The intermediate node captures this data segment, it recognizes the receivers *timestamps* in both data segments and forms an association. Finally, with a *timestamp* matching of all these data segments, the intermediate node can observe a full estimated RTT. Hence, in order to reliably associate the data segments with its corresponding ACK that triggered it, compute accurate RTT and avoid the ambiguity between delayed and retransmitted segments, we have employed the TCP *timestamps* option.

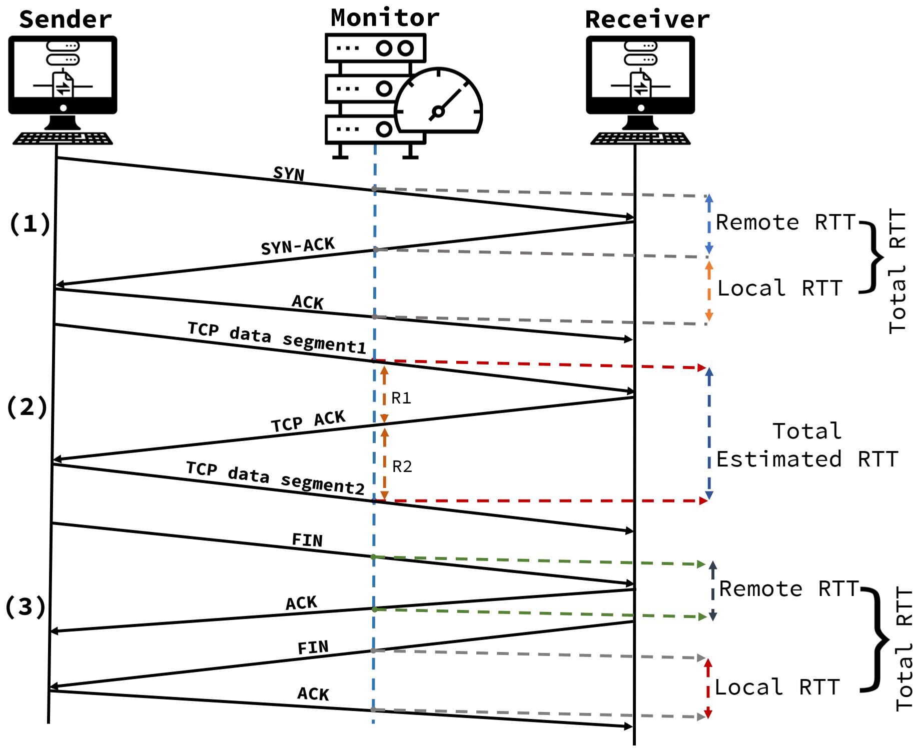


Figure V.3: Passive RTT estimation techniques.

**Timestamps option**: The reason why we used the *timestamp* option in our evaluation is to avoid the impact of incorrect (spurious) *timeouts* and get the *accurate* RTT measurement. Anytime TCP experiences spurious timeouts unnecessarily, it significantly suffers from unnecessary retransmissions and congestion control back-off. This, in turn, triggers TCP to drop the *Slow Start Threshold (ssthresh)* to half the current *cwnd* and reduces the value of *cwnd*. This is because when the retransmission of a lost packet is as a result of the RTO value expiration, TCP cannot infer anything about the *state* of the network. Unless TCP uses the *timestamp* option while sending the retransmitted packets, it cannot correctly measure the RTTs for those packets [23]. In addition to this, research studies like the *Eifel* mechanism have shown that the *timestamp* option substantially improves the overall TCP connection's performance over paths with a large *Bandwidth-delay product (BDP)* [11]. Since TCP is a symmetric protocol, allowing data segments to be sent and received at any given time in both directions, the *timestamp* options are also specified in a symmetrical manner [18] as they can be sent and echoed in both directions. This means the actual *Timestamp Value (TSval)* added by the sender is carried in both the ACK and data segments are echoed in *Timestamp Echo Reply (TSecr)* fields carried in the returning ACK or recently received data segments [18]. In this way, we can avoid *underestimating* the actual RTT measurement.

## V.5.1 Practical Challenges

The TCP congestion control algorithms that are widely deployed today perform the most important parameters used for TCP performance evaluation such as handling the *cwnd* and RTT from the sender-side. In this paper, we are interested in passively inferring an end-to-end RTT for TCP from packet arrival times collected at an intermediate monitoring point of a network without having access to the sender. However, there are some practical factors and concerns which complicate the implementation of a passive RTT measurement from an intermediate node. Here, we describe these practical challenges and the approaches how we address them in our evaluation.

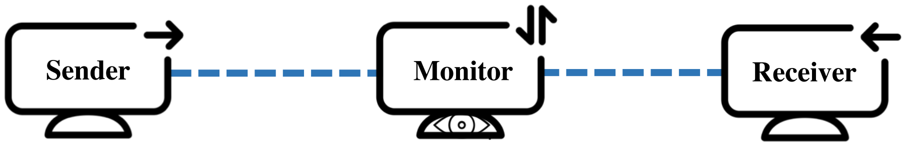


Figure V.4: RTT computation scenarios.

Let's suppose two end-points are exchanging traffic in both directions as shown in Figure V.4. If we measure at the intermediate point, one approach is to measure from the perspective of the sender in both directions. This is because if we measure the RTT at the receiver side, we cannot be sure when we send an ACK that it will trigger a new sent packet at the sender. In addition to this, the sender might not have any data to send, or the sender may also still have opportunities to send without receiving an ACK. If the sender sends data to the receiver, and the receiver sends a corresponding ACK, then the *monitor* could measure *monitor-to-receiver-to-monitor* part of the RTT by observing the data packet and the associated ACK. Similarly, if the receiver sends data to *sender*, and *sender* sends an ACK, then the *monitor* could measure the *monitor-to-sender-to-monitor* part of the RTT by observing the data segments and the associated ACK. Finally, these two values could be added together into an observation of an estimated RTT. However, there are a number of limitations that require consideration: *(i)* many connections send traffic mainly in one direction, rather than in both directions *(ii)* since Internet routing is not necessarily symmetric (i.e., the path from sender to receiver is not necessarily the reverse of the path from receiver to sender), the monitor might not be on the path in both directions between sender and receiver, and between receiver and sender. *(iii)* The RTT estimate is combined from two different data-ACK pairs at different times. In order to get a more reliable and accurate estimation, our passive RTT prediction model takes advantage of the TCP *timestamps* option (see Section V.5).

**Measuring at both endpoints**: What happens if we capture the traffic at both the sender and receiver endpoints and do the RTT estimation separately? There will be a difference in the timing of the received data which will affect the RTT estimation. It means this, to get a good measure of the raw one-way RTT for each direction, would require clock synchronization between the sender and receiver endpoints. We have no way to combine these two clocks with any strong guarantees unless the two endpoints are reasonably synchronized (e.g., by using GPS signals). However, it is important to remember that there isn't a timing difference for predicting the underlying TCP variant since it simply measures the change in *cwnd* size.

**Sender idle time**: *How do we technically handle the idle time (delay) of the sender when the buffer is not full and the sender waits for enough data to be pushed*? As explained above, since the queue is *one-way*, the idle time in the sender when there isn't enough data to send doesn't have an impact on the *network propagation time*, but it does for an application latency. Hence, this is both application and implementation dependent as there are many applications where the sender has nothing to send. We may have a transmitting delay when there is a lost packet that triggers a *dupPACK*. In the presence of a packet loss or *out-of-order* packet, the response will come right away. Basically, if the sender is application limited, measuring RTT on the *three-way handshake* is very reliable. However, we are interested in when a segment carries data. In order to passively estimate RTT, we measure all the sequence numbers of the data segments going in both directions at the intermediate node and their corresponding ACK only if they carry data. If the monitoring point is somewhere in the path, all we observe is packets flowing back and forth, and we can't tell the difference between *network* and *application* latency. Hence, we may treat them both the same way while data is being exchanged between the sending and receiving endpoints. The inclusion of TCP *timestamps* was supposed to help in these calculations independently by observing the *timestamps* sent and echoed in both directions and provide an improved RTT estimation.

**Multiple packets with the same sequence number**: The fact that TCP can send multiple packets with the same sequence number is a challenge. For example, if we send a packet with a sequence number and an ACK, *how do we know when the other packets have been sent if we see another packet with the same sequence number*? This is challenging and highly depends on whether the connection is using TCP *timestamps* or not. As explained in detail above, if TCP *timestamps* are not enabled, RTT samples cannot be safely computed due to the *retransmission ambiguity*, and thus *unreliable*. However, if TCP *timestamps* are enabled, then the sender can accurately compute an RTT sample even for retransmitted data using the *timestamp Echo Reply (TSecr)* [18]. This is one of the main reasons why we are using the *timestamps* options for our RTT measurement scheme. If more than one *Echo Reply* of a data packet is received before a reply segment is sent, our model estimates the RTT using the latest transmission time of most recently sent data packet with the oldest sequence number while ignoring the data packets with the earliest transmission time. This helps us to avoid spurious retransmissions.

### V.5.2 Considerations

Here is the list of TCP mechanism we consider in our analysis.

**Delayed ACKs**: During our RTT passive measurement scheme, we took regular *delayed ACKs* into account by leaving the *delayed ACK* enabled since major operating systems enable it by default for TCP even though the algorithms and constants are slightly different for each operating system. Most of the widely-deployed TCP variants nowadays will have the *delayed ACK* tag switch on by default to reduce network overhead. That means the TCP implementation would have to deal with whatever ACK algorithm the receiver is using and the receiver can wait up to *500ms* (common TCP implementations delay the ACK only up to *200ms*) before it sends an ACK in the hope that it can save the packet [4, 18]. So most of the TCP variants nowadays are configured in such a way that they are allowed to send an ACK for every second full-sized segment. In order to do that they receive data and wait for up to *200ms*. If nothing else comes, they send an ACK – if something else comes, they send a cumulative ACK for both. However, it is necessary to remember that the receiver's *delayed ACK* mechanism, noisy links, and other factors may introduce bias by causing systematic overestimation of RTT derived from Data-ACK matching (especially for slow-moving TCP connections like an interactive *telnet* or *ssh* session). In our analysis, to eliminate this bias when an ACK covers multiple packets, we used only the RTT from the latest data packet that is ACKed. It can also be done by filtering out the ACKs of unacknowledged data segments whose value is less than *2*MSS* since those are quite possibly *delayed ACKs* [4]. TCP is generally supposed to delay ACKs until either: *(i)* at least 2 full MSS of data has arrived, in which case the TCP receiver should send an immediate ACK, or *(ii)* the delayed ACK timer fires. If we get an ACK that is for >= *2*MSS* of data, then there is a very good chance that the ACK was triggered by *(i)*, in which case the latest data that arrived was probably ACKed immediately. If we get an ACK that is for less than *2*MSS* of data, it was probably triggered by *(ii)*, the delayed ACK timer, and should be filtered out if we want the RTT of the network path. This is precisely one of the reasons why we avoid packet sizes over the regular legitimate *MSS* in our experiments by disabling TCP segmentation offloading as described below.

***Maximum Segment Size (MSS)***: Our experiments are carried out over a path that is jumbo-frame clean by disabling TCP segmentation offloading in order to avoid packet sizes greater than the regular legitimate values. 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. This means when the data size is greater than the legitimate MSS, the messages will be split into several frames with a higher chance of unnecessary retransmissions which will introduce processing delays that affect the time it takes to send the data. Therefore, in order to avoid this violation, we made sure that each TCP flow uses a standard Maximum Transmission Unit (MTU) value of *1500-byte* data packets.

### V.5.3 Impact of the Underlying TCP Variants

We believe RTT is generally unaffected by the underlying TCP congestion algorithm that is being used, except *indirectly* due to ambiguous retransmissions which will probably make some RTTs seem longer when congestion is detected. TCP congestion avoidance algorithms specify: *(i)* How much should the current packets per burst be reduced when there is a packet loss, and *(ii)* How should that number be increased for each RTT. As described in Section V.2, RTT between two endpoints is typically a combination of a fixed *baseRTT*, the propagation time and whatever amount of queueing is experienced along the way. The propagation time is not a function of congestion control or even of TCP, it's the same for any IP packet. Therefore, we believe that there is no direct impact of the underlying TCP variant on a per-packet measured RTT.

## V.6 Validation Scenarios

Our model has been validated on the following settings.

**Emulated setting**: As illustrated in Figure V.2, we used the *measured* RTT data from the intermediate node as an *input* to our methodology for an inference of the RTT prediction. Finally, we verified the predicted RTT with the actual TCP *timestamps* directly logged from the Linux kernel used only for training and generate new data for the learning model to predict on. Once we finish with the verification, we run our learning model and get the predictions. We validated our methodology using the experimental testbed shown in Figure V.2 over a LAN link. In order to train and test our prediction model, we employed a single trained network that adapts to all experiments with variations of *bandwidth*, *delay*, *jitter* and *packet loss* into one learning model. We have demonstrated that our model can also be applicable in real networks.

**Realistic setting**: The ability to use embeddings of a model trained on an emulated environment to a realistic setting is a huge advantage in terms of scalability, applicability, and robustness. In this paper, we are able to train in one scenario setting and apply it as a pre-training in another scenario setting. Therefore, we are able to show that the learned passive RTT prediction model by leveraging a pre-trained knowledge of the LSTM network from the *emulated* network performs reasonably well as it is shown in the results when it is applied and transferred to a *realistic* scenario setting bearing similarity to the concept of

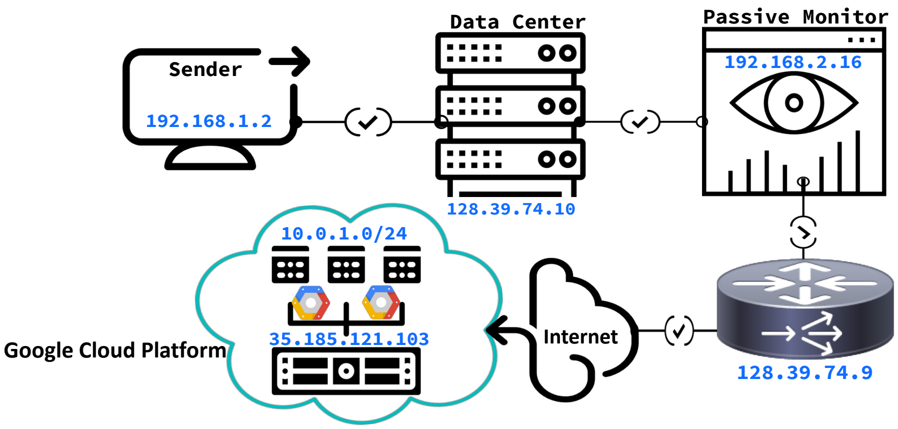


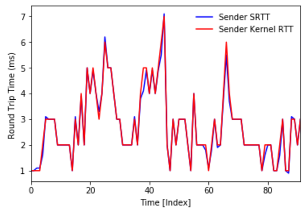
Figure V.5: Realistic Scenario Setup.

*transfer learning* in the machine learning community [32]. Here, we rely on passive measurements of real-world TCP network trace to evaluate the effectiveness of our model. This guarantees that our LSTM-based RTT prediction model is able to discern the results to unforeseen scenarios. As shown in Figure V.5, we performed a realistic experiment using Google cloud Virtual Machines (VMs) hosted across different regions. The experimental results of our realistic scenario are presented in Figure V.7.

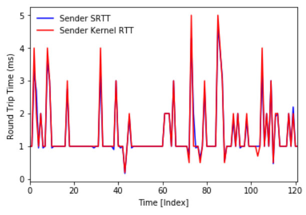
## V.7 Experimental Results and Discussion

On Section V.1, we have justified the choice of deep learning-based approaches in our paper. In this section, we will explain how the key features of deep learning and their implementations are being exploited.

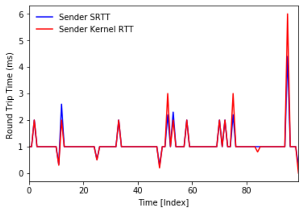
**Implementation details**: We implemented our RTT prediction model in *Python* using the *Keras* deep learning framework with *Google's TensorFlow* backend [1] running on NVIDIA Tesla K80 GPU where we apply an LSTM-based architecture to estimate the *RTT* trained over multiple epochs by taking the RTT samples as values in time-series. As shown in Figure V.1 at each time-step of $t$, as a learning process, the LSTM model takes an entire array of the Data-ACK matching based on *timestamps* captured on the monitoring point between the sender and receiver as an input feature vector ($x$) indexed by *timestamps* obtained from the kernel. We propagate the input to the model through a multilayer LSTM cell followed by a dense layer of 15-dimensional hidden states with *Rectified Linear Unit (ReLU)* activation function for the different layers that generates an output of a sequence dimensional vector of predicted *RTT* ($y_t$) of the same size indexed by *timestamps*. Our LSTM network is trained using the *Truncated Back Propagation Through Time* (*TBPTT*) training algorithm for modern RNNs applied to sequence prediction problems [31]. We used this training algorithm



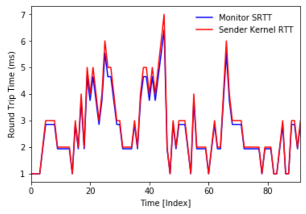
(a)



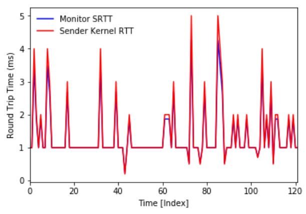
(b)



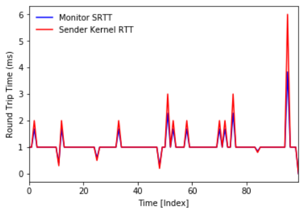
(c)



(d)

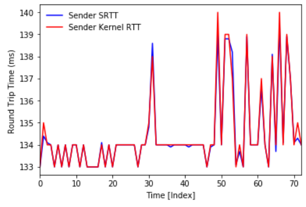


(e)

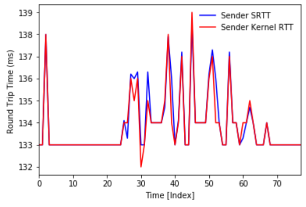


(f)

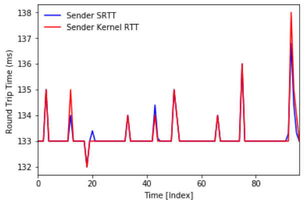
Figure V.6: RTT Prediction results comparison of an *emulated* setting between the TCP sending node and the intermediate node. **(a)-(d)**, TCP Westwood [10]. **(b)-(e)**, TCP Vegas [5]. **(c)-(f)**, TCP BBR [6].



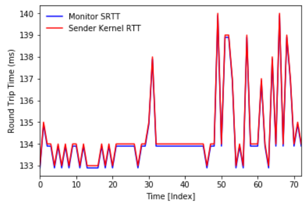
(a)



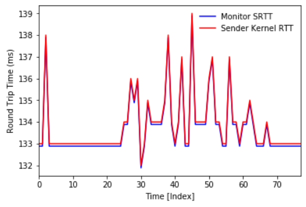
(b)



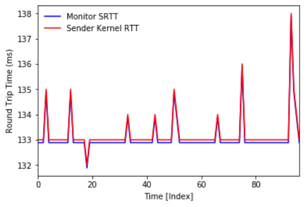
(c)



(d)



(e)



(f)

Figure V.7: RTT Prediction results comparison of a *realistic* setting between the TCP sending node and the intermediate node. **(a)-(d)**, TCP Westwood [10]. **(b)-(e)**, TCP Vegas [5]. **(c)-(f)**, TCP BBR [6].

to minimize LSTM's total prediction error between the expected output and the predicted output for a given input of the measured RTT time-series. We trained our LSTM-based learning algorithm without the knowledge of the input features from the TCP sender-side during the learning phase. 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. In order to train our prediction model more quickly, and get a more stable and robust to changes RTT estimation model, we have applied one of the most effective optimization algorithms in the deep learning community, the *Adam* stochastic algorithm [24] with an initial *learning rate* of *0.001* and *exponential decay rates* of the first ($\beta_1$) and second ($\beta_2$) moments set to 0.9 and 0.999 respectively. Totally, all of our configurations were trained for a maximum of 100 epochs with the mini-batch size of 256 samples. We further optimize a wide range of important hyper-parameters related to the neural network topology to improve the performance of our prediction model. 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 variations of *bandwidth*, *delay*, *jitter* and segment *loss* into one learning model.

**Evaluation metrics**: In order to evaluate and measure how well our LSTM-based prediction model performs in terms of capturing the time-series *RTT* patterns under different network conditions, all the neural networks are trained, as it is shown in Tables V.2 and V.3, by employing both the *Root Mean Square Error (RMSE)* and *Mean Absolute Percentage Error (MAPE)* performance metrics. The *RMSE* measures the root average squared error between the predicted and actual value, while MAPE measures the absolute deviation between the predicted and actual value as a percentage. The well-known *RMSE* and *MAPE* metrics are both means of estimating the point-wise errors in predictions and it is good to remember that these metrics don't depend on RTT sample sizes. Hence, the metrics values do not change for different numbers of samples in the output of the neural network. This is because the sum increases with the number of summed elements, but *mean* is sum divided by the number of elements, so it's "per element".

Table V.2: Prediction accuracy of an *emulated* setting.

| | *Kernel RTT vs. Sender SRTT* | | *Monitor SRTT vs. Kernel RTT* | |
|---|---|---|---|---|
| **TCP Algorithms** | **RMSE** | **MAPE (%)** | **RMSE** | **MAPE (%)** |
| Westwood [10] | 0.1587 | 0.8632 | 1.4916 | 1.7391 |
| Vegas [5] | 0.1854 | 0.6341 | 0.7289 | 0.6581 |
| BBR [6] | 0.2103 | 1.0217 | 0.5733 | 1.2812 |

**Discussion**: We start by exploring in detail the practical challenges in the dynamic inference of RTT for TCP connections in IP networks from passively monitored traffic. The plots presented in Figures V.6 and V.7 show the RTT prediction as a function of the elapsed time since a packet is sent until a

Table V.3: Prediction accuracy of a *realistic* setting.

| | *Kernel RTT vs. Sender SRTT* | | *Monitor SRTT vs. Kernel RTT* | |
|---|---|---|---|---|
| **TCP Algorithms** | **RMSE** | **MAPE (%)** | **RMSE** | **MAPE (%)** |
| Westwood [10] | 0.2504 | 1.3185 | 1.5277 | 1.8479 |
| Vegas [5] | 0.4155 | 2.3097 | 1.8327 | 2.5103 |
| BBR [6] | 0.2714 | 0.8730 | 1.7942 | 2.0715 |

corresponding ACK is received at the sender (*y-axis*) and index of *time* in *seconds* (*x-axis*) of various TCP variants under a wide range of network conditions and validation scenario settings. The general sawtooth patterns of the time-series RTT prediction plots we presented in Figures V.6 and V.7 are consistent with the behavior of each TCP variant considered. The minimum RTT during a given time window begins at around *1ms* in the *emulated* and *133ms* in the *realistic* setting but slowly ramp up to the maximum. We believe that this happens because the packets are being queued somewhere. When the queue gets filled, packets begin to be dropped and therefore, the RTTs level off. They level out because we see RTTs for packets that have been at the end of the queue. However, with TCP BBR as it is shown in Figure V.6 (**c**), (**f**) and Figure V.7 (**c**), and (**f**), the RTT can go down because when BBR notices the queue, it decides to send slower to drain it even if there are no packet drops [6]. Our measurement results show that we achieve high accuracy of the RTT pattern across different settings. We performed several experiments that illustrate our main approach under multiple scenarios settings and different configurations. However, due to lack of space, the experimental results presented in Figures V.6 and V.7 are a subset of the configurations for a proof of concept to show that our prediction model is applicable both in an *emulated* and *real-world* settings.

The experimental comparisons on both scenarios presented on Tables V.2 and V.3 are between the actual RTT values we obtained from Linux kernel of the TCP sending node against the SRTT of RTT samples collected on the sender. The second column on both tables compares the estimated SRTT of the intermediate node (passive monitor) against the actual RTT value of RTT obtained from Linux kernel of the TCP sending node. For a more detailed explanation and definition of SRTT, we refer the reader to Section V.3 of this paper. On Tables V.2 and V.3, *Kernel RTT* is the actual RTT value used by the Linux kernel of the TCP sending node. Whereas, *monitor*, as shown in Figure V.2, is the *intermediate* node between the sender and the receiver. Tables V.2 and V.3 show the performance of our model in an *emulated* and *realistic* scenarios, respectively, and we observe that our prediction model performs comparably well in both validation settings.

**Optimality**: The experimental results show that our deep learning-based RTT prediction model performs with high accuracy across different validation scenarios.

## V.8 Conclusion and Future Work

This work demonstrates how methods from the field of Artificial Intelligence (AI) can in principle aid in solving network-related complex problems. Under different variants of TCP, RTT is a property of the path between the sender and receiver whose value influences the dynamics of TCP. Hence, an accurate and dynamic estimation of RTT is crucial for TCP to maximize fair-share of the network resources. It provides useful information for network operators in investigating the critical factors that limit a flow rate and cause a congestive collapse in their networks. In this paper, we have proposed and evaluated a novel LSTM-based prediction model capable of dynamically predicting at real-time the RTT between the sender and receiver with high accuracy based on passive measurements collected at an intermediate node, taking advantage of the commonly used TCP timestamps. We explored in detail a set of practical methodological challenges and considerations involved in performing inference of RTT dynamically and reliably from passive measurements. The primary contribution of our work is building a prediction model that works well for *transfer learning*. We have demonstrated the efficiency of our model through extensive experiments both on a controlled *experimental testbed* network and in a *realistic* scenario setting on the *Google Cloud platform*. As future work, we would like to explore extensions in greater detail to the model we have presented across a broad range of different network conditions and multiple simultaneous TCP connections. By design, unlike *loss-based* algorithms, the *back-off* parameter of *delay-based* congestion control algorithms is not fixed which makes it fundamentally challenging to predict the TCP variant from passive traffic when there is variability in *delay*. Hence, now that we are able to predict RTT with a high accuracy, we believe extending our work in developing a *delay-based* pattern mining methodology that identifies the underlying *delay-based* TCP flavors from passive traffic and real measurements over the Internet using the RTT prediction model as an input vector is a promising direction for future research.

## Acknowledgment

We would like to sincerely thank Professor Øivind Kure for his comments on the final draft of our paper.

## V.9 References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[2] J. Aikat, J. Kaur, F. D. Smith, and K. Jeffay. Variability in TCP round-trip times. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, pages 279–284. ACM, 2003.

[3] P. Benko and A. Veres. A passive method for estimating end-to-end TCP packet loss. In *GLOBECOM'02*. IEEE, 2002.

[4] R. Braden. Requirements for Internet hosts-communication layers. RFC 1122, 1989.

[5] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. *TCP Vegas: New techniques for congestion detection and avoidance*, volume 24. ACM, 1994.

[6] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson. BBR: Congestion-based congestion control. 2016.

[7] J. Cleary, S. Donnelly, I. Graham, A. McGregor, and M. Pearson. Design principles for accurate passive measurement. In *Proceedings of PAM*, 2000.

[8] ESnet. iperf3. https://iperf.fr/iperf-servers.php, 2017.

[9] C. Fraleigh, C. Diot, B. Lyles, S. Moon, P. Owezarski, D. Papagiannaki, and F. Tobagi. Design and deployment of a passive monitoring infrastructure. In *Thyrrhenian Internatinal Workshop on Digital Communications*, pages 556–575. Springer, 2001.

[10] M. Gerla, M. Y. Sanadidi, R. Wang, A. Zanella, C. Casetti, and S. Mascolo. TCP Westwood: Congestion window control using bandwidth estimation. In *GLOBECOM'01. IEEE Global Telecommunications Conference (Cat. No. 01CH37270)*, volume 3, pages 1698–1702. IEEE, 2001.

[11] A. Gurtov and R. Ludwig. Responding to spurious timeouts in TCP. In *IEEE INFOCOM 2003. Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE Cat. No. 03CH37428)*, volume 3, pages 2312–2322. IEEE, 2003.

[12] D. H. Hagos, P. E. Engelstad, A. Yazidi, and Ø. Kure. General TCP State Inference Model From Passive Measurements Using Machine Learning Techniques. *IEEE Access*, 6:28372–28387, 2018.

[13] M. Hanai, S. Yamaguchi, and A. Kobayashi. TCP Fairness Evaluation with Modified Controlled Delay in the Practical Networks. In *Proceedings of the 12th International Conference on Ubiquitous Information Management and Communication*, page 77. ACM, 2018.

[14] S. Hemminger et al. Network emulation with NetEm. 2005.

[15] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[16] V. Jacobson. Congestion avoidance and control. In *ACM SIGCOMM computer communication review*, volume 18, pages 314–329. ACM, 1988.

[17] V. Jacobson. Congestion avoidance and control. *ACM SIGCOMM Computer Communication Review*, 25(1):157–187, 1995.

[18] V. Jacobson, R. Braden, and D. Borman. TCP extensions for high performance. RFC 1323, 1992.

[19] M. Jain and C. Dovrolis. End-to-end available bandwidth: measurement methodology, dynamics, and relation with TCP throughput. *IEEE/ACM Transactions on Networking (TON)*, 11(4):537–549, 2003.

[20] R. Jain. A delay-based approach for congestion avoidance in interconnected heterogeneous computer networks. *ACM SIGCOMM Computer Communication Review*, 19(5):56–71, 1989.

[21] S. Jaiswal, G. Iannaccone, C. Diot, J. Kurose, and D. Towsley. Inferring tcp connection characteristics through passive measurements. In *IEEE INFOCOM 2004*, volume 3, pages 1582–1592. IEEE, 2004.

[22] H. Jiang and C. Dovrolis. Passive estimation of TCP round-trip times. *ACM SIGCOMM Computer Communication Review*, 32(3):75–88, 2002.

[23] P. Karn and C. Partridge. Improving round-trip time estimates in reliable transport protocols. *ACM SIGCOMM Computer Communication Review*, 25(1):66–74, 1995.

[24] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[25] M. Kühlewind, S. Neuner, and B. Trammell. On the state of ECN and TCP options on the Internet. In *International Conference on Passive and Active Network Measurement*, pages 135–144. Springer, 2013.

[26] A. Loukili, A. L. Wijesinha, R. K. Karne, and A. K. Tsetse. TCP's Retransmission Timer and the Minimum RTO. In *2012 21st International Conference on Computer Communications and Networks (ICCCN)*, pages 1–5. IEEE, 2012.

[27] J. Martin and A. Nilsson. On service level agreements for IP networks. In *Proceedings. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 2, pages 855–863. IEEE, 2002.

[28] C. Olah. Understanding LSTM Networks. https://colah.github.io/posts/2015-08-Understanding-LSTMs, 2015.

[29] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP throughput: A simple model and its empirical validation. *ACM SIGCOMM Computer Communication Review*, 28(4):303–314, 1998.

[30] C. Paxson, M. Allman, J. Chu, and M. Sargent. Computing TCP's Retransmission Timer (RFC 6298), 2011.

[31] H. Tang and J. Glass. On Training Recurrent Networks with Truncated Backpropagation Through Time in Speech Recognition. In *2018 IEEE Spoken Language Technology Workshop (SLT)*, pages 48–55. IEEE, 2018.

[32] K. Weiss, T. M. Khoshgoftaar, and D. Wang. A survey of transfer learning. *Journal of Big data*, 3(1):9, 2016.

Paper VI

# Classification of Delay-based TCP Algorithms From Passive Traffic Measurements

**Desta Haileselassie Hagos**[1], **Paal E. Engelstad**, **Anis Yazidi**

***Published*** in *the 18th IEEE International Symposium on Network Computing and Applications (NCA 2019)*. IEEE, 2019.
DOI: 10.1109/NCA.2019.8935063.

## Abstract

Identifying the underlying TCP variant from passive measurements is important for several reasons, e.g., exploring security ramifications, traffic engineering in the Internet, etc. In this paper, we are interested in investigating the delay characteristics of widely used TCP algorithms that exploit queueing delay as a congestion signal. Hence, we present an effective TCP variant identification methodology from traffic measured passively by analyzing $\beta$, the multiplicative back-off factor to decrease the *cwnd* on a loss event, and the queueing delay values. We address how $\beta$ varies as a function of queueing delay and how the TCP variants of delay-based congestion control algorithms can be predicted both from passively measured traffic and real measurements over the Internet. We further employ a novel non-stationary time series approach from a stochastic nonparametric perspective using a two-sided Kolmogorov–Smirnov test to classify delay-based TCP algorithms based on the $\alpha$, the rate at which a TCP sender's side *cwnd* grows per window of acknowledged packets, parameter. Through extensive experiments on emulated and realistic scenarios, we demonstrate that the data-driven classification techniques based on probabilistic models and Bayesian inference for optimal identification of the underlying delay-based TCP congestion algorithms give promising results. We show that our method can also be applied equally well to loss-based TCP variants.

[1]University of Oslo, Department of Informatics, destahh@ifi.uio.no
**Keywords**: TCP, RTT Prediction, LSTM, Passive Measurements

## VI.1 Introduction and Motivation

Transmission Control Protocol (TCP) is one of the dominant transport protocols that has played a great role in the exponential success of the Internet, network technologies and applications [17]. The majority of all Internet traffic all over the world today uses TCP. TCP is a highly reliable end-to-end connection-oriented transport protocol designed to prevent excessive congestion on the Internet [17]. Note that congestion control in TCP was not part of the protocol initially until the first Internet congestion collapse was observed [17]. TCP controls congestion by aiming for fair sharing of the available network resources by the competing flows, using strategies empowered by TCP [17]. Congestion control algorithms provide a fundamental set of techniques critical for maintaining the robustness, efficiency, and stability of the global Internet. Since the specification of TCP [25], various end-to-end congestion control algorithms have been designed and implemented on a larger scale for the Internet with several enhancements. One category of the widely deployed variants ranging from TCP CUBIC [12], Reno [17], BIC [30], etc. where packet loss probability is an implicit signal for congestion in the underlying network are called loss-based TCP congestion control algorithms. Variants of this kind aggressively fill up the actual network buffers in order to achieve better throughput by ignoring queueing delay. However, this is challenging for the quality of latency-sensitive and bandwidth-intensive real-time media applications to achieve good performance when long-running flows also share large bottleneck link buffers. Therefore, to address this challenging problem, delay-based TCP schemes that adopt packet queueing delay rather than a loss as congestion signals are introduced. With delay-based congestion control algorithms, allocating network resources across competing users can be attained by supporting both high network utilization and low queuing delay even when the buffer sizes are large. Detailed background on these categories of TCP variants is presented in Section VI.2. In this paper, we are interested in investigating the delay characteristics of widely used TCP algorithms that exploit queueing delay as a congestion signal and demonstrating how an intermediate node can identify both loss-based and delay-based TCP variants from passively captured TCP traffic using *state-of-the-art* approaches. As explained below, inferring whether the underlying network is using *loss-based* or *delay-based* TCP congestion control algorithms has potential benefits for various reasons. Our work in this paper is mainly motivated by the following questions: *(i)* How well can we infer the most important TCP per-connection transmission states that determine a network condition (e.g., *Congestion Window (cwnd)*) from passive traffic collected at an intermediate node of the network without having access to the sender? *(ii)* How can we identify the underlying delay-based TCP variant that the TCP client is using from passive measurements? *(iii)* What percentage of network users are using *delay-based* TCP variants? *(iv)* 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.

**Potenial opportunities and benefits**: It is reasonable to ask: *Why is the identification of the underlying TCP flavors performed in an intermediate node from passive measurements important*? Some of the main reasons why passive estimation of TCP internal state variables in an intermediate node is important, for example, is when *(i)* We have no control over either end-host of communication so we can't launch active measurements from either host, *(ii)* The TCP active probes used in active measurements (such as *ping* messages) are blocked by firewalls etc. In addition to this passive measurements, unlike active measurements, do not introduce additional traffic into the network that can skew the results. For more details about the difference between *active* and *passive* measurement techniques, we refer the reader to our previous work [13]. There are myriad reasons we may want to use passive measurements to identify the underlying TCP flavors but in our paper, we will explore the above questions quantitatively from different perspectives as they apply to the problems of network congestion.

**Operational benefit**: We argue that uniquely inferring the underlying TCP congestion algorithm the client is using 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 cwnd behaviour 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*?

**ISP benefit**: 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 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 to one of the known congestion control algorithms.

**Security ramifications**: We believe it is also useful for exploring security threats. This is because if we are able to infer the TCP variant, we can also make some guessing on the implementation of the underlying operating system and search for vulnerabilities. This can tell us about the encryption at the end-system that can be used to tailor-made attacks.

**Contributions**. Our paper makes the following contributions.

- We identify the main challenges in investigating the delay characteristics of the widely used TCP algorithms.
- We demonstrate how the intermediate node (e.g., a network operator) can predict the *cwnd* size of delay-based TCP algorithms using *state-of-the-art* deep learning techniques.
- We examine a set of *state-of-the-art* techniques that are reasonably effective in classifying the underlying variants of delay-based TCP congestion control algorithms within flow from passive measurements based on the $\beta$ parameter.
- We employ a novel non-stationary time series approach from a stochastic nonparametric perspective using a two-sided Kolmogorov–Smirnov test to classify delay-based TCP algorithms based on the $\alpha$ parameter.
- We are able to identify the widely used TCP variants with high accuracy and explore security ramifications.
- We compare our delay-based classification approach with recent *state-of-the-art* loss-based identification techniques.
- Finally, we show that the learned prediction model performs reasonably well by leveraging trained knowledge from the *emulated* network when it is applied and transferred on a *realistic* scenario setting.

## VI.2 Background

TCP congestion control strategies are broadly categorized into loss and delay sensitive schemes. Loss-based TCP congestion control algorithms consider packet loss as an implicit indication of congestion by the network. TCP variants of this kind attempt to fill the network buffers and hence they tend to induce large queueing delays when the buffer sizes are large. Unlike traditional loss-based approaches, delay-based TCP congestion control algorithms use the changes in queueing delay measurements as implicit feedback to congestion in the network. Delay-based congestion control algorithms attempt to avoid network congestion by monitoring the trend of network path's RTT information contained in packets. It is believed that variants of this kind achieve better average throughput by not filling buffers and maintaining full path utilization with low queueing and fair allocation of rates to flows [4, 28]. In order to properly allocate, share the underlying network resources, and ensure network queueing delay stays low, delay-based congestion control algorithms require knowledge of an accurate estimate of the network path's $BaseRTT$ [21], usually defined as the smallest of all measured minimum RTTs of a segment in the absence of congestion. The following are the list of an end-to-end widely used delay-based congestion control algorithms on the Internet we use for our evaluation.

- **TCP Vegas** [4]: Vegas is a delay-based implementation of TCP congestion control algorithm motivated by the studies [18] and [27]. Vegas's congestion detection technique depends on the accurate estimation of $BaseRTT$ [4]. Hence, if the estimated value of $BaseRTT$ is too small, then it's throughput will stay below the available bandwidth; however, if the estimated value for $BaseRTT$ is too large, then it will overrun the connection [4]. As shown in Equation VI.1, Vegas computes the *expected* throughput of the connection as the ratio of the current window size and $BaseRTT$. The main idea of Vegas behind Equations VI.1 and VI.2 is that when the network is not congested, the actual flow rate will be close to the expected flow rate. However, if the network is congested, the expected flow rate will be greater than the actual flow rate.

$$Expected = \frac{WindowSize}{BaseRTT} \tag{VI.1}$$

$$Actual = \frac{WindowSize}{RTT} \tag{VI.2}$$

To estimate the available bandwidth and congestion state of the network, TCP Vegas compares the actual sending rate and evaluates the difference, *Diff*, between the *estimated* throughput and the current *actual* throughput computed as shown in Equation VI.3 and updates the *cwnd* accordingly.

$$Diff = Exptected{-}Actual \tag{VI.3}$$

By definition *Diff* is a non-negative since $Actual{>}Expected$ implies that we need to change $BaseRTT$ to the latest sampled RTT. To adjust the congestion window size, TCP Vegas uses *two* threshold values $\alpha$ and $\beta$ where $0{\leq}\alpha{<}\beta$ [4]. Depending on this difference as shown in Figure VI.1 and Equation VI.4, if $Diff{<}\alpha$, Vegas increases the *cwnd* size linearly until the next RTT, and when $Diff{>}\beta$, then Vegas reduces the *cwnd* linearly until the next RTT. However, Vegas leaves the *cwnd* size unchanged when $\alpha{<}Diff{<}\beta$.

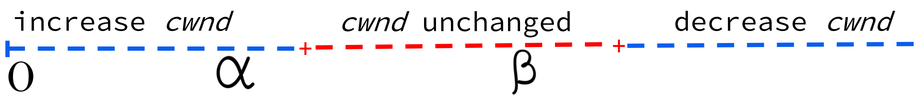


Figure VI.1: TCP Vegas throughput thresholds.

$$cwnd = \begin{cases} cwnd + 1 & \text{If } Diff < \alpha \\ cwnd - 1 & \text{If } Diff > \beta \\ cwnd & \text{Otherwise} \end{cases} \tag{VI.4}$$

- **TCP Veno** [11]: Veno adopts the same methodology as TCP Reno [17] in determining the congestion window size in the network. But Veno uses the delay information of TCP-Vegas [4] to further differentiate non-congestion packet losses when RTT varies greatly by estimating the backlogged packets in the buffer similar to TCP Vegas. If the number of backlogged packets in the buffer is below a certain threshold, it is a strong indication of random loss. However, if packet loss is detected when the connection is in the congestive state, TCP Veno uses the standard TCP Reno Additive Increase and Multiplicative Decrease (AIMD) scheme to reduce the *cwnd* during its congestion avoidance mode. TCP Veno sets $\beta$ factor to 0.8 when the queueing delay is small. However, when the queueing delay is high, TCP Veno sets $\beta$ to 0.5.

For comparison reasons we also consider the following most widely used loss-based TCP congestion control algorithms.

- **TCP Reno** [17]: Reno is one of the most predominant implementations of loss-based TCP variants which employs a conservative linear growth function for increasing the *cwnd* by one segment for each received ACK and multiplicative decrease function on encountering a packet loss per RTT [5]. Its $\beta$ value is 0.5 which corresponds to reducing the window by 50% during a loss event as shown in Equation VI.5.

$$cwnd = \begin{cases} cwnd + 1 & \text{Slow start phase} \\ cwnd + \frac{1}{cwnd} & \text{Congestion avoidance} \\ \frac{cwnd}{2} & \text{If packet is lost} \end{cases} \tag{VI.5}$$

- **TCP CUBIC** [12]: CUBIC is the default congestion control algorithm as part of the Linux kernel distribution configurations from version 2.6.19 onwards. It modifies 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. CUBIC decreases the *cwnd* by a factor of $\beta$ whenever it detects that a segment was lost. And, it increases towards a target congestion window size ($W$) when in-order segments are acknowledged where $W$ is defined as follows:

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

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 $K$ is defined by the following function:

$$K = \sqrt[3]{\frac{W_{max}\beta}{C}} \qquad \text{(VI.7)}$$

where $\beta$ is a constant back-off factor of CUBIC [12] applied for window reduction at the time of a TCP packet loss event. The $\beta$ value of CUBIC is 0.7 which corresponds to reducing the window by 30% during a TCP packet loss event [12] and can be calculated as per Equations VI.6 and VI.7.

**Roadmap**: The rest of this paper is organized as follows. Next, in Section VI.3, we discuss the related work in the literature considered as a *state-of-the-art*. In Section VI.4, we describe an overview of our controlled experimental setup for the evaluation. Section VI.5 presents approaches to our classification models in detail. The experimental results and discussion are presented in Section VI.6. Finally, Section VI.7 concludes the paper and outlines directions of research for future extensions.

## VI.3 Related work

TCP is one of the key protocols of today's Internet Protocol (IP) suite and its performance analysis has been extensively studied in the computer networking community [24]. Many research studies have also analyzed the underlying TCP congestion control algorithms as we have already discussed the most relevant works in Section VI.2. There are many different TCP variants widely in use, and each variant uses a specific end-to-end congestion control algorithm to avoid congestion, while also attempting to share the underlying network capacity equally among the competing users. However, we believe that there is very little work on the identification of the underlying delay-based TCP congestion control algorithms from passive measurements. The work in [22] proposes a *cluster analysis-based* method that aims a router to identify between two versions of 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 [22]. These characteristics are then clustered into two groups by applying a hierarchical clustering technique. The authors show that only 2 out of 14 TCP congestion algorithms that are implemented in Linux can be identified based on their method [22]. Another related work [31] presents an *active* measurement technique to identify a diverse set of known congestion control algorithms. However, our work in this paper relies on a *passive* measurement technique. In a closely related previous work [15], we presented a machine learning-based approach to identify the underlying traditional *loss-based* TCP variants which yield accuracies of

93.51% and 95% on emulated and realistic scenarios respectively. The *cwnd* prediction performance result of the loss-based variants across different scenario settings is presented in Table VI.1.

Table VI.1: cwnd prediction accuracy of *loss-based* TCP variants under an *emulated* and *realistic* settings [14, 15].

| | *Emulated Setting* | | *Realistic Setting* | |
|---|---|---|---|---|
| **TCP Algorithms** | **RMSE** | **MAPE** (%) | **RMSE** | **MAPE** (%) |
| CUBIC | 5.839 | 6.953 | 3.522 | 4.738 |
| Reno | 3.511 | 3.140 | 3.396 | 5.197 |

This was achieved by analyzing the $\beta$ value by averaging out the window size of *loss-based* algorithms every time a peak is detected so that the computation of the multiplicative decrease factor is not done only on a *slow start* phase. However, this work doesn't address how the $\beta$ as a function of queueing delay varies and how the TCP variants of delay-based congestion control algorithms can be predicted both from passively measured traffic and real measurements over the Internet. By design, unlike loss-based algorithms, the $\beta$ value of delay-based congestion control algorithms is not fixed which makes it fundamentally challenging to predict the TCP variant from passive traffic when there is variability in delay. Hence, in this paper, we want to substantially address this problem by building a two-dimensional space model and see if the $\beta$ is dependent on queueing delay or not. This helps us to expand our previous method [15] to address bigger cases covering both loss-based and delay-based TCP congestion control algorithms.

## VI.4 Evaluation Methodology

### VI.4.1 Experimental Setup

Our evaluation experiments are carried out using a cluster of HPC machines based upon the GNU/Linux operating system running a modified version of the *4.15.0-39-generic* kernel release. The prediction 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, 128 GB RAM, 12 CPU cores running under Linux 64-bit. All nodes in the cluster are connected to a low latency 56 Gbit/s Infiniband, gigabit Ethernet and have access to 600 TiB of BeeGFS parallel file system storage.

### VI.4.2 Validation Experiment

We have conducted a controlled experiment both on simulated environments and realistic scenario settings.

**Emulated Setting**: We construct an experimental setup shown in Figure VI.2 where we generate the training data and predict the *cwnd* from passively captured

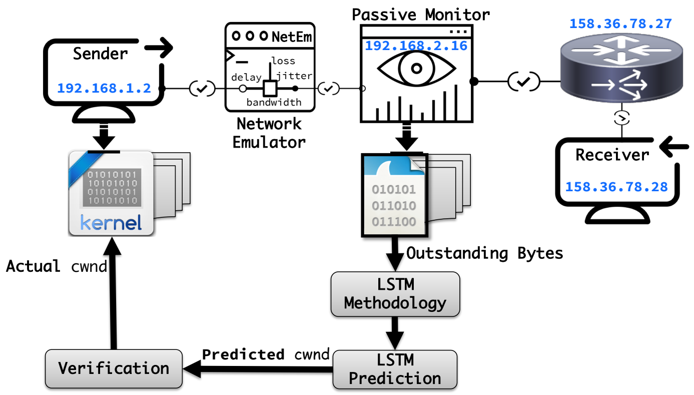


Figure VI.2: LSTM Methodology for *cwnd* prediction.

traffic using state-of-the-art deep learning approaches. To evaluate the prediction model and perform our analysis on both the emulated and realistic network conditions, we have generated our own training dataset. In order to capture all sessions on the network when the client and server are sending TCP packets and measure the TCP data packets from both directions, we have used the fully controlled experimental setup shown in Figure VI.2. The background TCP stream traffic for all our experiments are generated using the *iperf* [9] traffic generator on an emulated LAN link where we run each TCP variant by adding a variation of the emulation parameters *bandwidth (in Mbit)*, *delay (in ms)*, *jitter (in ms)* and *packet loss (%)* within a flow. 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*. The issues of cross-traffic variability and verification of the popular Linux-based network emulator we used, *Network Emulator (NetEm)* [16], are thoroughly addressed in our previous work [13].

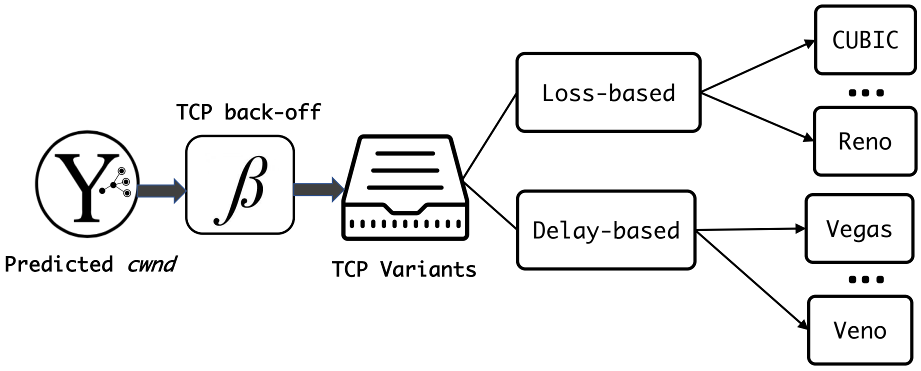


Figure VI.3: Methodology for TCP Variant classification.

**Realistic Setting**: In addition to the simulation validation described above, we have also evaluated our experiments over real-world Internet paths using the setup shown in Figure VI.4 so that we can further validate our results presented in Figure VI.9 against other scenario settings. This helps us to carefully test how well our deep learning-based cwnd prediction model using an emulated network works by conducting a series of controlled experiments in a realistic setting. In this way, we can justify and guarantee how our model could predict the development of a *cwnd* pattern and the TCP variant used with other realistic network traffic scenarios captured from the Internet. To this end, we created a realistic experimental testbed where we experiment by running our resources on Google Cloud platform nodes across the Internet as shown in Figure VI.4. In order to create a realistic TCP session, we uploaded a massive image file to Google Cloud platform site so that we have a full control of the underlying TCP variant on the sending node and at the same time run a *tcpdump* in the background and capture all sessions on the network when the client and server are sending TCP packets. Next, we filtered out the receiving host where we send the TCP traffic to. Finally, we calculated the number of outstanding bytes obtained from the captured network traffic and run it through our learning model to predict the development of the TCP *cwnd*. Since we have full control of the sending node, we can track the system-wide TCP state of every packet that is sent and received from the kernel to verify our model's prediction accuracy against the ground truth by matching with the actual sending TCP states using the methodology shown in Figure VI.2. As it is shown in Figure VI.10, we found out that our model could be performing very well with small prediction errors when we apply it to real-world scenario settings too. The final cwnd sawtooth pattern prediction performance comparison between the emulated and realistic settings is presented in Table VI.5.

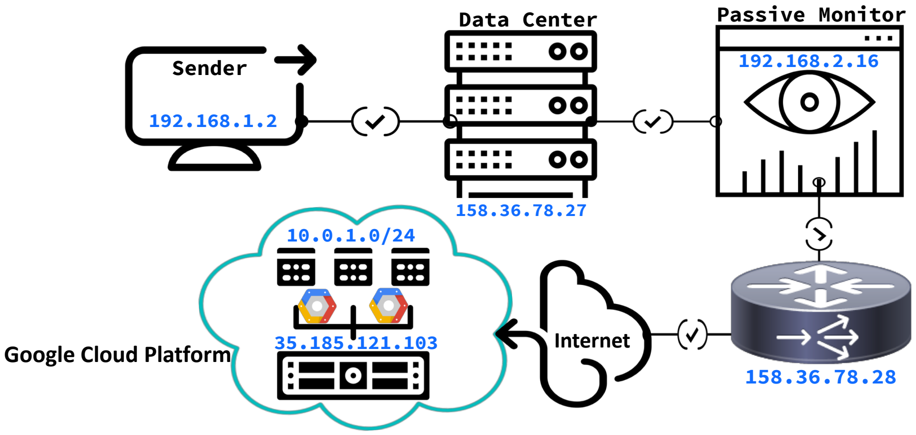


Figure VI.4: Realistic Scenario Setup.

## VI.5 Our Approaches

This section presents the concepts and approaches to the underlying TCP variant classification process.

### VI.5.1 Passive cwnd Prediction

For this task, we are interested in the capabilities and potentials of *Recurrent Neural Networks (RNN)* models for implementing our passive *cwnd* prediction model for TCP. Hence, we have explored an approach to investigate and explore in detail on how an intermediate node (e.g., a network operator) can identify the transmission state of both loss-based and delay-based TCP congestion control algorithms associated with a passively monitored TCP traffic using *Long Short-Term Memory (LSTM)*-based RNN architecture. 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* [23] when there is variability of *bandwidth*, *delay*, *jitter* and *loss* is a better approach to estimate the *cwnd* and how fast the recovery is. We 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 and run it through our learning model to predict the development of the TCP *cwnd* and it's variant.

**Implementation details**: Our methodology of the classification process is depicted in Figure VI.3. We implemented our passive *cwnd* prediction model in *Python* using the *Keras* deep learning framework with *Google's TensorFlow* backend [1] running on NVIDIA Tesla K80 GPU where we apply an LSTM-based architecture trained over multiple epochs by taking the *cwnd* samples as values in time-series. As shown in Figure VI.2 at each time-step of $t$, as a learning process, the LSTM model takes an entire array of the outstanding bytes matching based on *timestamps* captured on the intermediate monitoring point between the sender and receiver as an input feature vector indexed by *timestamps*. We propagate the input to the model through a multilayer LSTM cell followed by a dense layer of 15-dimensional hidden states with Rectified Linear Unit (ReLU) activation function for the different layers that generates an output of a sequence dimensional vector of predicted *cwnd* of the same size indexed by *timestamps*. Our LSTM network is trained using the Truncated Back Propagation Through Time (TBPTT) training algorithm for modern RNNs applied to sequence prediction problems [26]. We used this training algorithm to minimize LSTM's total prediction error between the expected output and the predicted output for a given input of the measured *cwnd* time-series. We trained our LSTM-based learning algorithm without the knowledge of the input features from the TCP sender-side during the learning phase. 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. In order to train our prediction model more quickly, and get a more stable and robust to changes *cwnd* estimation model, we have applied one of the most effective optimization algorithms in the deep learning community, the *Adam* stochastic algorithm [19] with an initial *learning rate* of *0.001* and *exponential decay rates* of the first ($\beta_1$) and second ($\beta_2$) moments set to 0.9 and 0.999 respectively. Totally, all of our configurations were trained for a maximum of 100 epochs with the mini-batch size of 256 samples. We further optimize a wide range of important optimal hyperparameters related to the neural network topology to improve the performance of our prediction model. 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 into one learning model.

**Why did we use RNN models**? As explained above, the *cwnd* is a TCP per-connection state internal variable, stored in the memory of the TCP sender, relevant to congestion control. However, since the value of the *cwnd* is not contained in the TCP header – trying to predict this value somewhere other than at the TCP sending node is fundamentally challenging. In our case, let's consider a situation where a network model is trained for a specific intermediate node which has been trained for a specific bandwidth, background load, multiplexing rate, and a multitude of different router conditions, can predict well for exactly this node. Hence, we want a model that is able to train in one scenario setting and apply it as a pre-training on another setting by leveraging trained knowledge. As it is presented in Section VI.4, this paper proofs that it makes sense in principle to use learning algorithms for TCP state predictions and this is the reason why we use RNN approach for the passive *cwnd* prediction.

## VI.5.2 TCP variant classification based on the $\beta$ parameter

**k-nearest neighbors (KNN)**: The first approach we used to identify the underlying TCP variants is a distance metrics using KNN machine learning classification algorithm [6]. Given an input feature vector of TCP protocols in an $n$-dimensional Euclidean space $\mathbb{R}^n$ with a set of back-off parameters and queueing delay instances, $\{\hat{\beta}_i, \widehat{diff}_i\} \in \mathbb{R}^n$, training samples of the form ($\{x_i, y_i\}$, $x_i \in \mathbb{R}^n$), we want to classify a new TCP protocol, $P$, by finding the value of $\{\hat{\beta}_i, \widehat{diff}_i\}$ that is nearest to $P$. For the estimation of $\widehat{diff}$ in our evaluations, we applied the formula proposed in TCP Vegas as shown in Equation VI.3.

As shown in Figure VI.5, our classifier model fits reasonably well with high accuracy. However, we believe that this approach has a limitation in classifying TCP Reno [17] and TCP Veno [11] when the queueing delay is high. For example, if we have many $\hat{\beta}$ points of TCP Veno in a two-dimensional space with low $\widehat{diff}$ that means we will have more $\hat{\beta}$ values with one cluster of 0.8. How do we ensure that the $\widehat{diff}$ is low and how do we tell the exact difference between TCP Veno and Reno? Hence, to avoid this shortcoming, we proposed the following methods.

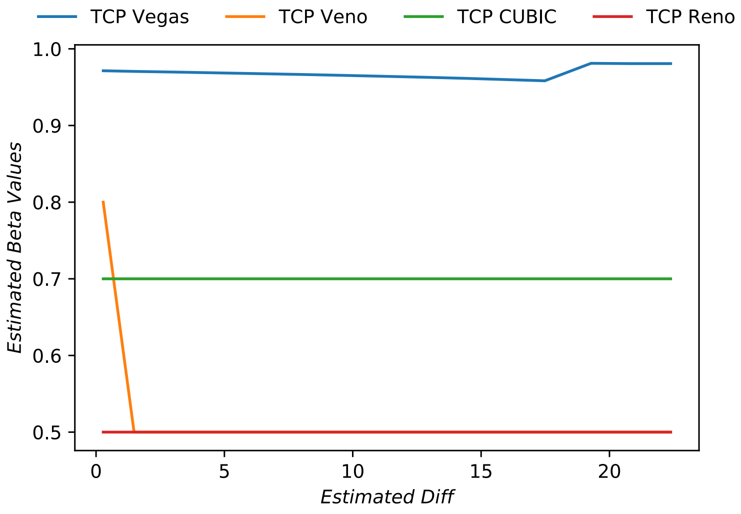


Figure VI.5: KNN Prediction of TCP Variants.

**Kullback-Leibler (KL) Divergence**: Before fitting our data into the beta distribution family, we wanted to answer the question: *How do we optimally choose the positive shape parameters, $\alpha$ and $\beta$, of beta distribution given in Equation VI.9?* Hence, we use the KL divergence [20] to find the fitting parameters in the beta distribution. KL, in statistics, information theory and pattern recognition, is a well-known distance measure between two probability distribution measures $p(y)$ and $q(y)$ defined as follows:

$$D(q\|p) := \mathbb{E}_{y\sim q(y)}\left[\log\frac{q(y)}{p(y)}\right] = \int q(y)\log\frac{q(y)}{p(y)}dy \qquad \text{(VI.8)}$$

In general, the KL divergence is only defined if $q(y) > 0$ for any value of $y$ such that $p(y) > 0$.

**Beta Distribution**: As shown in Figure VI.8, the beta distribution is the best fitting model for the problem we address in this paper for all the TCP protocols except TCP Veno [11]. Beta distribution, parametrized by two positive shape parameters, denoted by $\alpha$ and $\beta$, is appropriate for representing the uncertainty of a continuous probability distribution and is defined by:

$$f(K|\alpha,\beta) = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)}K^{\alpha-1}(1-K)^{\beta-1} \qquad \text{(VI.9)}$$

where $K \in [0, 1]$ and it represents the support of the probability distribution, $\Gamma(\cdot)$ is the gamma function defined as: $\Gamma(z) = \int_0^\infty t^{z-1}e^{-t}dt$.

For TCP Veno, as depicted in Figure VI.8(a), we used a *sigmoid*-based function and the reason why we applied *sigmoid* function for TCP Veno is because say that we have 1 million $\hat{\beta}$ points with high $\widehat{diff}$ value (i.e., when the connection is on a congestive state) and only 1 $\hat{\beta}$ point with low $\widehat{diff}$ value. If we compare this with TCP Reno [17] which has few points with a fixed $\hat{\beta}$ value of around 0.5, the inaccuracy might be a little higher when we measure it with other metrics. When we run our experiment on the Internet, we can't decide how many $\hat{\beta}$ points we get because it depends on the underlying network. As we can see it on Figures VI.6 and VI.7, we have one $\hat{\beta}$ point for Veno [11] and we believe this is completely realistic. Because this means we have one class (cluster) of 1 $\hat{\beta}$ point with low $\widehat{diff}$ and another class of many $\hat{\beta}$ points with high $\widehat{diff}$ and it is possible to classify the protocols based on these classes. We built our model in such a way that we don't want the sender or network change anything with the TCP parameter values (i.e., $\alpha$ and $\beta$) that control increase and decrease ratios of the *cwnd*. We simply want to observe things passively from an intermediate node between the sender and the receiver. Therefore, to tell the exact difference between the low $\widehat{diff}$ and high $\widehat{diff}$, we have to statistically measure how close we are to the border between the low and high $\widehat{diff}$. For example, when we are around a $\widehat{diff}$ threshold of 3, the $\widehat{diff}$ will not count much but the weight of the $\hat{\beta}$ value does. This way, we can tell the difference between the protocols TCP Veno and TCP Reno by running a *sigmoid*-based function on the border between low and high $\widehat{diff}$ values.

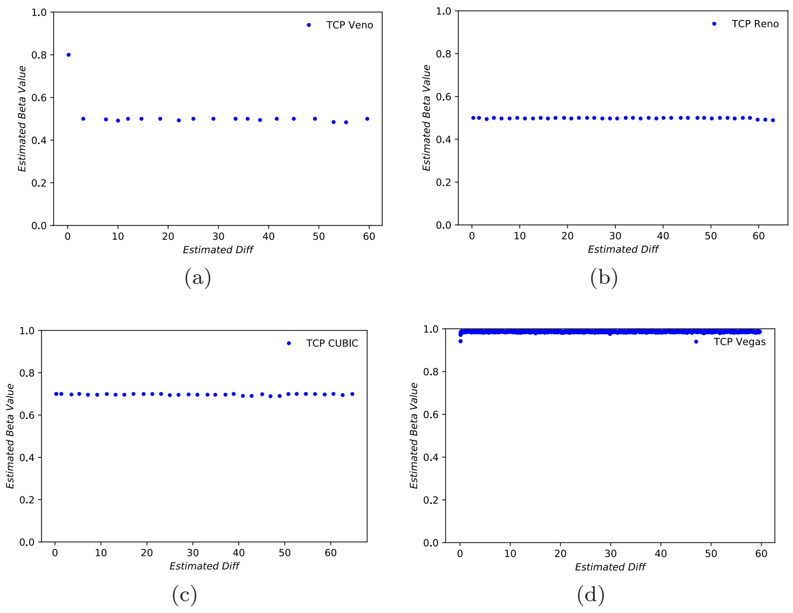
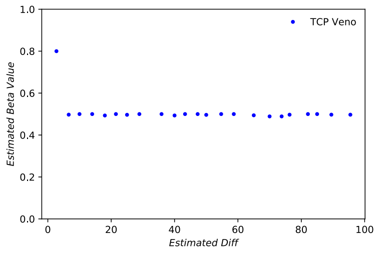
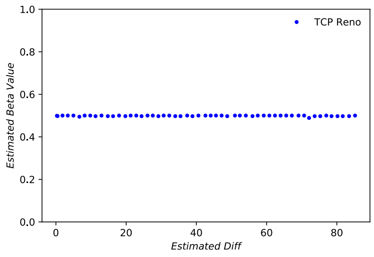


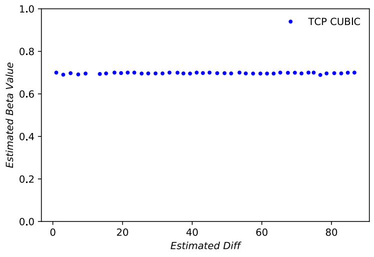
Figure VI.6: Beta analysis in an *emulated* setting. **(a)** Veno [11], **(b)** Reno [17], **(c)** CUBIC [12], **(d)** Vegas [4].



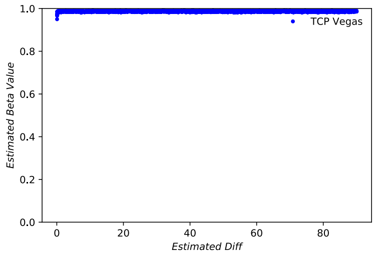
(a)



(b)

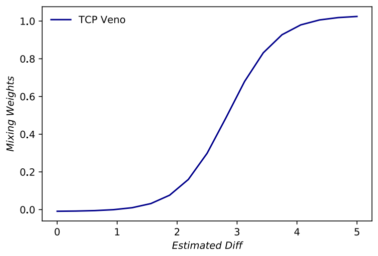


(c)

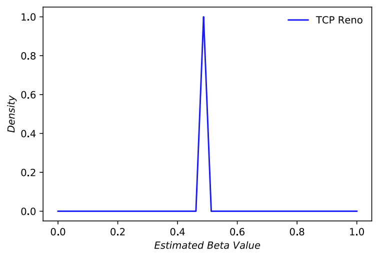


(d)

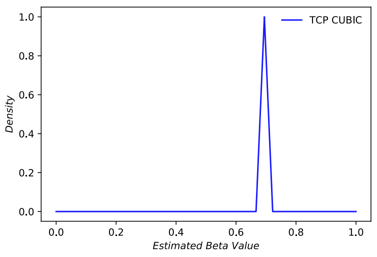
Figure VI.7: Beta analysis in a *realistic* setting. **(a)** Veno [11], **(b)** Reno [17], **(c)** CUBIC [12], **(d)** Vegas [4].



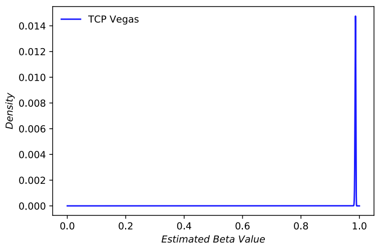
(a) Sigmoid analysis of TCP Veno [11]



(b)

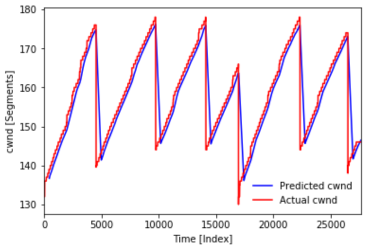


(c)

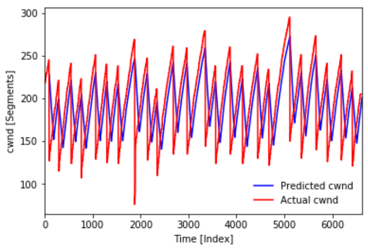


(d)

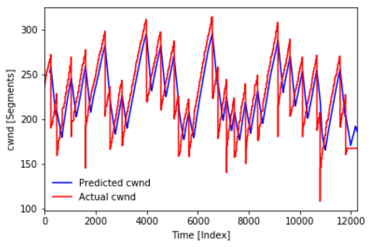
Figure VI.8: Sigmoid analysis and beta distributions. **(a)** Veno [11], **(b)** Reno [17], **(c)** CUBIC [12], **(d)** Vegas [4].



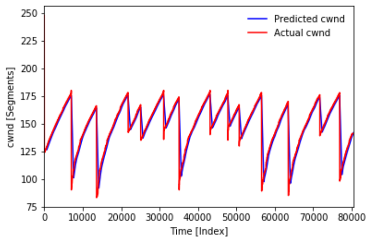
(a)



(b)

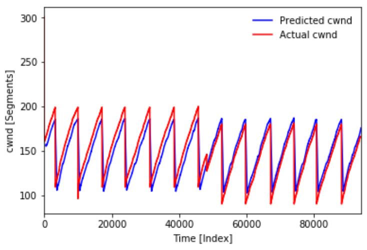


(c)

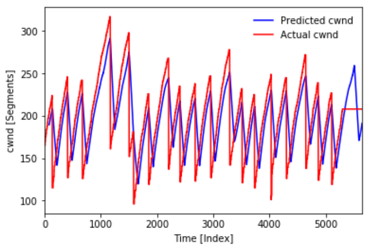


(d)

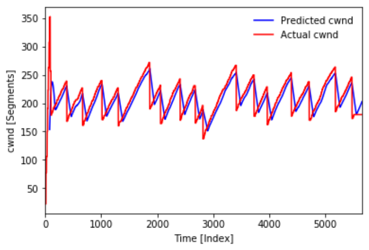
Figure VI.9: TCP *cwnd* Prediction results of an *emulated* setting. **(a)** Veno [11], **(b)** Reno [17], **(c)** CUBIC [12], **(d)** Vegas [4].



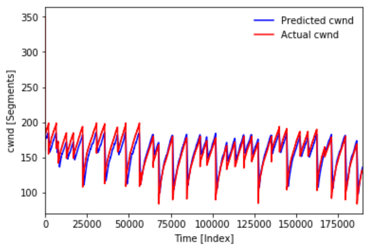
(a)



(b)



(c)



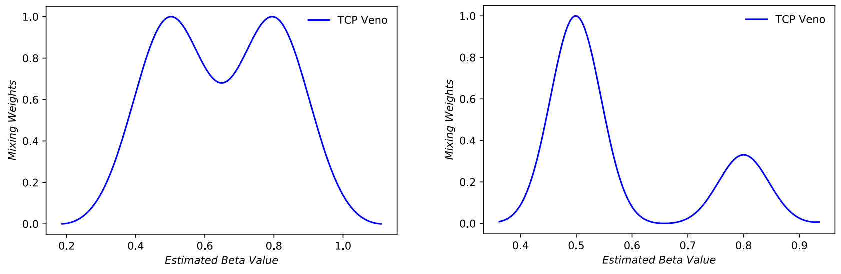
(d)

Figure VI.10: TCP *cwnd* Prediction results of a *realistic* setting. **(a)** Veno [11], **(b)** Reno [17], **(c)** CUBIC [12], **(d)** Vegas [4].

**Mixture Distributions**: For TCP Veno, we applied a beta mixture distribution model in a classification setting defined as follows:

$$f_1(\hat{\beta}_i, \widehat{diff}_i) = (1 - \lambda_1(\widehat{diff}))d_1(\hat{\beta}_i) + \lambda_2(\widehat{diff})d_2(\hat{\beta}_i) \tag{VI.10}$$

where $\lambda_i$ being the mixing weights (density) of the *sigmoid* function that depends on the value of $\widehat{diff}$, $\sum_i \lambda_i = 1$, $d_1$ and $d_2$ are two different distributions. First, we pick a distribution of TCP Veno with probabilities given by the mixing weight, $\lambda$ and $\widehat{diff}$, then we generate one observation according to the selected distribution as shown in Figure VI.11. Intuitively, the *sigmoid* of TCP Veno should be the weights of the two peaks of the $\hat{\beta}_i$ values, i.e., a beta distribution centered around 0.5 and a beta distribution centered around 0.8. For the other TCP protocols, we applied a beta distribution of the form $f_i(\hat{\beta}_i)$. We have experimented with different beta mixture distributions and finally, we have verified that our model yields reasonably good results. Sample beta mixture distributions of TCP Veno under different values are shown below in Figure VI.11.



(a) $\widehat{diff}$=3 (b) $\widehat{diff}$=4

Figure VI.11: Mixture distributions of TCP Veno [11].

**Bayesian Inference**: Using the data generated from the beta distribution, we built a Bayesian inference approach to machine learning by constructing a set of observations $O_{1:N} = \{O_1, O_2, O_3, \ldots, O_N\}$ in which each element $O_i$ represents a different set of observations of $\hat{\beta}_i$ and $\widehat{diff}_i$ of each TCP variant, $V$ that is obtained from the beta distribution model as $f_i(\hat{\beta}_i, \widehat{diff}_i)$. As shown in Equation VI.12, the normalization factor is the sum of the data.

$$P(V = V_i|O_i) \propto P(V = V_i) \prod_{i=1}^{N} P(O_i|V = V_i) \tag{VI.11}$$

From the law of probability theory, we know that:

$$\sum_{V_i} P(V = V_i | \{O_1, O_2, O_3, \ldots, O_N\} = 1$$
$$V = \{V_1, V_2, V_3, V_4, \ldots, V_N\} \tag{VI.12}$$

where $V_1 = Veno$, $V_2 = Reno$, $V_3 = CUBIC$ and $V_4 = Vegas$. For every $V_i$, the *argmax*() of these equations retrieves the index of the highest likelihood of the probability vector. In the absence of a priori detailed domain knowledge about the TCP protocols, from a Bayesian inference perspective we believe all TCP variants will have the same probability and hence, *P(Veno)* = *P(Reno)* = *P(CUBIC)* = *P(Vegas)*. Using Equations VI.11 and VI.12, we are able to perform the Bayesian inference and the results we obtained from both emulated and realistic scenario settings are presented as follows.

***Emulated setting***

- When the ground truth is TCP Veno, $P(V = Veno|O_1, O_2, O_3, O_4)$ gives a probability estimation vector of (**46.28**, 38.93, 14.34, 0.45) and from this *46.28* maximizes the probability that this is being classified as $V_1$ (Veno).
- When the ground truth is TCP Reno, $P(V = Reno|O_1, O_2, O_3, O_4)$ gives a probability estimation vector of (35.25, **49.81**, 14.57, 0.36) and from this *49.81* maximizes the probability that this is being classified as $V_2$ (Reno).
- When the ground truth is CUBIC, $P(V = CUBIC|O_1, O_2, O_3, O_4)$ gives an estimation vector of (10.13, 9.02, **71.83**, 9.02) and from this *71.83* maximizes the probability that this is being classified as $V_3$ (CUBIC).
- When the ground truth is Vegas, $P(V = Vegas|O_1, O_2, O_3, O_4)$ gives an estimation vector of (31.85, 0.28, 10.79, **57.08**) and from this *57.08* maximizes the probability that this is being classified as $V_4$ (Vegas).

***Realistic setting***

- When the ground truth is TCP Veno, $P(V = Veno|O_1, O_2, O_3, O_4)$ gives a probability estimation vector of (**46.83**, 39.4, 13.44, 0.34) and from this *46.83* maximizes the probability that this is being classified as $V_1$ (Veno).
- When the ground truth is TCP Reno, $P(V = Reno|O_1, O_2, O_3, O_4)$ gives a probability estimation vector of (30.99, **52.05**, 16.44, 0.52) and from this *52.05* maximizes the probability that this is being classified as $V_2$ (Reno).
- When the ground truth is CUBIC, $P(V = CUBIC|O_1, O_2, O_3, O_4)$ gives an estimation vector of (10.69, 9.53, **70.25**, 9.53) and from this *70.25* maximizes the probability that this is being classified as $V_3$ (CUBIC).

Table VI.2: $\widehat{diff}$ values performances

| **Emulated Setting** | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | *Low-*$\widehat{diff}$ | | | | *High-*$\widehat{diff}$ | | | |
| | Precision | Recall | F1-score | Support | Precision | Recall | F1-score | Support |
| Vegas | 1.00 | 0.92 | 0.96 | 13 | 1.00 | 0.91 | 0.95 | 11 |
| CUBIC | 1.00 | 1.00 | 1.00 | 10 | 0.92 | 1.00 | 0.96 | 11 |
| Reno | 0.90 | 1.00 | 0.95 | 9 | 0.90 | 1.00 | 0.95 | 9 |
| Veno | 1.00 | 1.00 | 1.00 | 8 | 1.00 | 0.89 | 0.94 | 9 |
| Avg/Total | 0.98 | 0.97 | 0.98 | 40 | 0.95 | 0.95 | 0.95 | 40 |
| *Accuracy* | *97.5%* | | | | *95%* | | | |
| **Realistic Setting** | | | | | | | | |
| | *Low-*$\widehat{diff}$ | | | | *High-*$\widehat{diff}$ | | | |
| | Precision | Recall | F1-score | Support | Precision | Recall | F1-score | Support |
| Vegas | 1.00 | 0.93 | 0.96 | 12 | 0.92 | 0.92 | 0.92 | 11 |
| CUBIC | 1.00 | 1.00 | 1.00 | 11 | 1.00 | 1.00 | 1.00 | 10 |
| Reno | 0.92 | 1.00 | 0.96 | 9 | 0.91 | 1.00 | 0.95 | 9 |
| Veno | 1.00 | 1.00 | 1.00 | 8 | 1.00 | 0.91 | 0.95 | 10 |
| Avg/Total | 0.98 | 0.98 | 0.98 | 40 | 0.96 | 0.95 | 0.95 | 40 |
| *Accuracy* | *97.83%* | | | | *95.46%* | | | |

Table VI.3: $\widehat{diff}$ values confusion matrix

| | | Predicted | | | |
|---|---|---|---|---|---|
| **Emulated** | Actual | CUBIC | Reno | Vegas | Veno |
| *Low-*$\widehat{diff}$ | CUBIC | 12 | 0 | 1 | 0 |
| | Reno | 0 | 10 | 0 | 0 |
| | Vegas | 0 | 0 | 9 | 0 |
| | Veno | 0 | 0 | 0 | 8 |
| *High-*$\widehat{diff}$ | CUBIC | 10 | 0 | 1 | 0 |
| | Reno | 0 | 11 | 0 | 0 |
| | Vegas | 0 | 0 | 9 | 0 |
| | Veno | 0 | 1 | 0 | 8 |
| **Realistic** | | | | | |
| *Low-*$\widehat{diff}$ | CUBIC | 13 | 0 | 1 | 0 |
| | Reno | 0 | 9 | 0 | 0 |
| | Vegas | 0 | 0 | 9 | 0 |
| | Veno | 0 | 0 | 0 | 8 |
| *High-*$\widehat{diff}$ | CUBIC | 10 | 0 | 1 | 0 |
| | Reno | 0 | 10 | 0 | 0 |
| | Vegas | 0 | 0 | 9 | 0 |
| | Veno | 1 | 0 | 0 | 9 |

- When the ground truth is Vegas, $P(V = Vegas|O_1, O_2, O_3, O_4)$ gives an estimation vector of (32.18, 0.39, 11.73, **55.7**) and from this *55.7* maximizes the probability that this is being classified as $V_4$ (Vegas).

### VI.5.3 TCP variant classification based on the $\alpha$ parameter

In our classification task of the underlying TCP algorithms, in contrast to the typical *increase-by-one decrease-to-half* scheme of TCP to adjust *cwnd* growth, we consider the $\beta$ and $\alpha$ parameters that control the increase and decrease ratios of *cwnd*. This means, the *cwnd* size is increased by $\alpha$ per window of acknowledged packets in the congestion avoidance state in response to every RTT and it is decreased to $\beta$ times its current value when there is congestion. Classifying the underlying TCP variant using the $\beta$ parameter with different approaches is discussed above in detail. Here, we will use the $\alpha$ parameter for the same task by employing a novel non-stationary time series approach from a stochastic nonparametric perspective. We believe this approach is appealing because the changing rate of the *cwnd* size can be modeled as a stochastic process [2, 3]. This method is ensuring to work properly because of the quasi-stationary properties that every TCP protocol has as it could be easily observed from Figure VI.12(a) and Figure VI.12(b) where the statistical behavior of the signal remains almost unaltered and note that the distribution in all of the scenarios is pretty consistent by maintaining the same property. We use the two-sided Kolmogorov-Smirnov (KS) test which is a nonparametric statistical test[2] for comparing two empirical cumulative distribution functions (ECDFs) [10]. The KS statistic for a given cumulative distribution function (CDF) *F(x)* is given as shown in Equation VI.13.

$$D_n(F_n, G_n) = \sup_x |F_n(x) - G_n(x)| \tag{VI.13}$$

where $\sup_x$ is the supremum of the set of distances over the given distributions, F and G are two ECDFs.

Our approach is to first estimate the probability distribution function (PDF) over categories in our classification task of each given TCP protocol as shown in Figure VI.12 and then estimate the 95% confidence interval for the distributions using bootstrap technique [7, 8] so that we can measure how certain we are about the predictions of the underlying TCP variant when its estimated $\alpha$ value changes frequently by comparing the uncertainty measure of the estimated PDF. We could also show the corresponding standard CDF for each value, but due to the limited space in this paper, here we present only the empirical PDF estimations. As we can see from Table VI.4, the stochastic confusion matrix has two values on both emulated and realistic settings. The first value compares the maximum difference between the ECDFs and chooses the protocol with the minimum distance that minimizes the probability of the log(*p-value*) as shown in Equation VI.14.

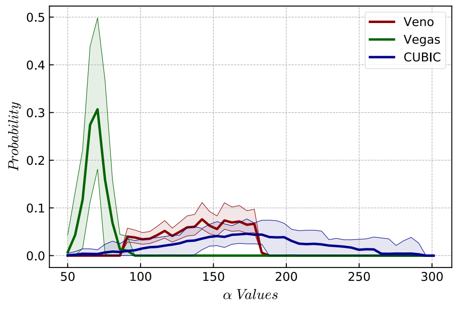
$$\gamma = \arg\min_{\gamma^\alpha} \int_\omega \left| \hat{P}_\gamma(\alpha) - \hat{P}_y(\alpha) \right| d\alpha \tag{VI.14}$$

[2] i.e., it does not assume a specific form of the distributions

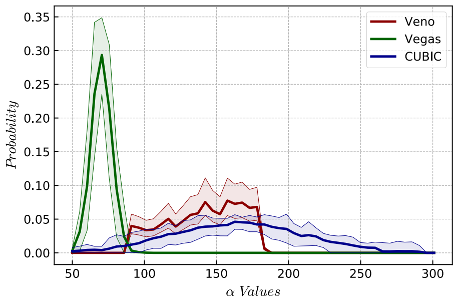
where $\gamma^{\alpha}$ represents the set of TCP protocols, $\omega$ represents all the possible values of $\alpha$, $\hat{P}_{\gamma}(\alpha)$ represents the empirical probability of a given TCP protocol ($\gamma$), $\hat{P}_{y}(\alpha)$ represents the estimated probability of $\alpha$ in dataset $y$. Whereas the second value compares the KS test values by maximizing the estimated PDF of each distribution using log($p$-$value$) of the bootstrap test for a given distribution as shown in Equation VI.15.

$$\gamma = \underset{\gamma \in \{V_i\}}{\arg\max} \log P[D_n(\gamma^C, y)], \quad i \in \{1, 2, 3\} \tag{VI.15}$$

where $y$ is another sampled time series of length $M$ used to classify the protocols so that we don't end up using the same time series and $\gamma^C$ are the candidate TCP protocols, $V_1 = Veno$, $V_2 = Vegas$, and $V_3 = CUBIC$, from which the PDF were initially estimated. Next, we calculate the distribution function using the raw time series data of each variant and compare it against the stochastic template of each TCP protocol using the KS test value whose result is presented in log($p$-$value$). Finally, we choose the underlying TCP protocol whose log($p$-$value$) bootstrap test is higher. In Figure VI.12, if the $\alpha$ interval between 100 and 160 have too high values, then the TCP variant is Veno. Otherwise, if the value is too low between these intervals, the protocol will be identified as CUBIC. Our method is robust enough to identify each underlying protocol without the prior domain knowledge of the internal characteristics of each TCP variant. As it is shown in Figure VI.12(a) and Figure VI.12(b), it is clear that the model performs well in terms of identifying the underlying TCP variants when applied both on an emulated and realistic scenario settings.



(a) Emulated Setting



(b) Realistic Setting

Figure VI.12: Empirical PDF estimations for the TCP protocols with a 95% confidence interval on emulated and realistic settings.

## VI.6 Experimental Results and Discussion

We have conducted several experiments over different scenario settings. In order to justify and guarantee how our learning model could predict the development

Table VI.4: Stochastic confusion matrix.

| | *Emulated Setting* | | | | | |
|---|---|---|---|---|---|---|
| | **Predicted** | | | | | |
| | Maximum difference | | | KS test | | |
| **Actual** | Veno | Vegas | Cubic | Veno | Vegas | Cubic |
| Veno | 99 | 0 | 1 | 100 | 0 | 0 |
| Vegas | 0 | 100 | 0 | 0 | 100 | 0 |
| Cubic | 2 | 0 | 98 | 5 | 0 | 95 |
| | *Realistic Setting* | | | | | |
| Veno | 100 | 0 | 0 | 100 | 0 | 0 |
| Vegas | 0 | 100 | 0 | 0 | 100 | 0 |
| Cubic | 1 | 0 | 99 | 3 | 0 | 97 |

of a *cwnd* sawtooth and the underlying TCP variant with other realistic network traffic scenarios captured from the Internet, we created a realistic testbed as shown in Figure VI.4 where we experiment by running our resources on Google Cloud platform nodes deployed across the globe. As shown in Figures VI.9 and VI.10, our passive *cwnd* prediction model works reasonably well when applied both on an emulated and realistic evaluation scenarios. We confirm that our model operates correctly and accurately recognizes the sawtooth pattern for realistic scenario settings across different Google Cloud platforms. This shows that our prediction model is *general* bearing similarity to the concept of transfer learning in the machine learning community [29].

**Evaluation metrics**: The passive *cwnd* prediction was evaluated for accuracy using the Root Mean Square Error (RMSE) and Mean Absolute Percentage Error (MAPE) metrics. The *cwnd* prediction performance result of both the emulated and realistic scenario settings across the Google Cloud platforms in terms of RMSE and MAPE is presented in Table VI.5. As stated in Section IV.5, the ground truth data for the realistic setting was collected from the kernel of the TCP sending node. The performance results on both metrics indicate that our model is able to achieve reasonably accurate passive predictions of the development of *cwnd* sawtooth pattern.

Table VI.5: cwnd prediction accuracy of *loss-based* and *delay-based* TCP variants under an *emulated* and *realistic* settings.

| | *Emulated Setting* | | *Realistic Setting* | |
|---|---|---|---|---|
| **TCP Algorithms** | **RMSE** | **MAPE (%)** | **RMSE** | **MAPE (%)** |
| Vegas [4] | 1.8225 | 2.7618 | 3.6536 | 4.8864 |
| Veno [11] | 3.1421 | 3.8644 | 3.9254 | 4.8705 |
| CUBIC [12] | 4.0775 | 5.2961 | 3.6370 | 4.2774 |
| Reno [17] | 4.2484 | 5.9947 | 4.7541 | 5.0322 |

In our two-dimensional space analysis, we evaluated how the $\hat{\beta}$ varies as a function of the estimated queueing delay ($\widehat{diff}$) for all TCP protocols basing our hypothesis on the approaches presented on Section VI.5. To see if the $\hat{\beta}$ is dependent on the queueing delay $\widehat{diff}$, let's consider TCP Veno [11]. Intuitively, If the value of $\widehat{diff}$ is *low* (i.e., $\widehat{diff}$<3), according to the standard specification Veno sets the $\beta$ to 0.8 and it means Veno decreases the *cwnd* upon packet loss only by 20%. However, if the delay is *high* (i.e., $\widehat{diff}$>3), Veno sets the $\beta$ to 0.5. In case of TCP Vegas [4], if the $\widehat{diff}$ threshold is high enough Vegas increases the *cwnd* and when the *cwnd* doesn't reach the pipe, it decreases by 1. However, if the *cwnd* is pretty large, it converges the $\beta$ towards 1 because of its Additive Increase and Additive Decrease (AIAD) strategy. In order to guarantee the accuracy of our TCP variant prediction model, we run our experiments where we ensure we have measurements with high $\widehat{diff}$ and low $\widehat{diff}$ values on different scenario settings. To this end, the prediction accuracies on emulated and realistic scenarios with these two measurement cases are 97.5%, 95%, 97.83%, and 95.46% respectively as shown in Table VI.2 and their corresponding confusion matrix is depicted in Table VI.3. As explained above, our stochastic approach is also robust enough in terms of classifying the TCP variants based on the $\alpha$ parameter without the prior domain knowledge of the internal characteristics of each variant as it is shown in Figure VI.12.

## VI.7 Conclusion and Future Work

In this paper, we investigate and explore in detail on how an intermediate node (e.g., a network operator) can identify the transmission state of delay-based TCP congestion control algorithms associated with a passively monitored TCP traffic. We present an effective TCP variant identification methodology from traffic measured passively by utilizing $\beta$, the multiplicative back-off factor to decrease the *cwnd* on a loss event, and the queueing delay values. We further employ a novel non-stationary time series approach from a stochastic nonparametric perspective using a two-sided Kolmogorov–Smirnov test to classify delay-based TCP algorithms based on the $\alpha$, the rate at which a TCP sender's side *cwnd* grows per window of acknowledged packets, parameter. Our model is built in such a way that we don't want the sender or network change anything with the TCP parameter values that control increase and decrease ratios of the *cwnd*. Through extensive experiments on emulated and realistic scenarios, we have demonstrated that the data-driven classification techniques based on probabilistic models and Bayesian inference for optimal identification of the underlying delay-based TCP congestion control algorithms give promising and comparable results in terms of accuracy. In conclusion, we show that the learned prediction model performs reasonably well by leveraging trained knowledge from the emulated network when it is applied and transferred in a realistic scenario setting. Finally, we have shown that our model can also be applied equally well to loss-based TCP variants using the presented approaches. To the best of our knowledge, this paper is the first to study how the variability of the $\beta$ parameter as a function of queueing delay and the $\alpha$ parameter can be used for passive TCP variant identification in real-time.

As part of our future work, we would like to substantially extend this work in terms of devising a generic learning model for operating system fingerprinting from passive measurements by combining the basic TCP/IP features and the underlying TCP variant as input vectors.

## VI.8 References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[2] A. A. Abouzeid and S. Roy. Stochastic modeling of TCP in networks with abrupt delay variations. *Wireless Networks*, 9(5):509–524, 2003.

[3] E. Altman, K. Avrachenkov, and C. Barakat. A stochastic model of TCP/IP with stationary random losses. *ACM SIGCOMM Computer Communication Review*, 30(4):231–242, 2000.

[4] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. *TCP Vegas: New techniques for congestion detection and avoidance*, volume 24. ACM, 1994.

[5] D.-M. Chiu and R. Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Computer Networks and ISDN systems*, 17(1):1–14, 1989.

[6] T. M. Cover, P. Hart, et al. Nearest neighbor pattern classification. *IEEE transactions on information theory*, 13(1):21–27, 1967.

[7] F. M. Dekking, C. Kraaikamp, H. P. Lopuhaä, and L. E. Meester. *A Modern Introduction to Probability and Statistics: Understanding why and how.* Springer, 2005.

[8] B. Efron and R. J. Tibshirani. *An introduction to the bootstrap.* CRC press, 1994.

[9] ESnet. iperf3. https://iperf.fr/iperf-servers.php, 2017.

[10] G. Fasano and A. Franceschini. A multidimensional version of the Kolmogorov–Smirnov test. *Monthly Notices of the Royal Astronomical Society*, 225(1):155–170, 1987.

[11] C. P. Fu and S. C. Liew. TCP Veno: TCP enhancement for transmission over wireless access networks. *IEEE Journal on selected areas in communications*, 21(2):216–228, 2003.

[12] S. Ha, I. Rhee, and L. Xu. CUBIC: a new TCP-friendly high-speed TCP variant. *ACM SIGOPS operating systems review*, 42(5):64–74, 2008.

[13] D. H. Hagos, P. E. Engelstad, A. Yazidi, and Ø. Kure. General TCP State Inference Model From Passive Measurements Using Machine Learning Techniques. *IEEE Access*, 6:28372–28387, 2018.

[14] D. H. Hagos, P. E. Engelstad, A. Yazidi, and Ø. Kure. Recurrent neural network-based prediction of tcp transmission states from passive measurements. In *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*, pages 1–10. IEEE, 2018.

[15] D. H. Hagos, P. E. Engelstad, A. Yazidi, and O. Kure. Towards a Robust and Scalable TCP Flavors Prediction Model from Passive Traffic. In *2018 27th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–11. IEEE, 2018.

[16] S. Hemminger et al. Network emulation with NetEm. 2005.

[17] V. Jacobson. Congestion avoidance and control. In *ACM SIGCOMM computer communication review*, volume 18, pages 314–329. ACM, 1988.

[18] R. Jain. A delay-based approach for congestion avoidance in interconnected heterogeneous computer networks. *ACM SIGCOMM Computer Communication Review*, 19(5):56–71, 1989.

[19] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[20] S. Kullback. *Information theory and statistics*. Courier Corporation, 1997.

[21] D. J. Leith, R. N. Shorten, G. McCullagh, L. Dunn, and F. Baker. Making available base-RTT for use in congestion control applications. *IEEE Communications Letters*, 12(6):429–431, 2008.

[22] J. Oshio, S. Ata, and I. Oka. Identification of different TCP versions based on cluster analysis. In *2009 Proceedings of 18th International Conference on Computer Communications and Networks*, pages 1–6. IEEE, 2009.

[23] S. Ostermann. Tcptrace. http://www. tcptrace. org, 2000.

[24] M. Panda, H. L. Vu, M. Mandjes, and S. R. Pokhrel. Performance analysis of TCP NewReno over a cellular last-mile: Buffer and channel losses. *IEEE Transactions on Mobile Computing*, 14(8):1629–1643, 2014.

[25] J. Postel. Transmission control protocol. RFC 793, 1981.

[26] H. Tang and J. Glass. On Training Recurrent Networks with Truncated Backpropagation Through Time in Speech Recognition. In *2018 IEEE Spoken Language Technology Workshop (SLT)*, pages 48–55. IEEE, 2018.

[27] Z. Wang and J. Crowcroft. A new congestion control scheme: Slow start and search (tri-s). *ACM SIGCOMM Computer Communication Review*, 21(1):32–43, 1991.

[28] D. X. Wei, C. Jin, S. H. Low, and S. Hegde. FAST TCP: motivation, architecture, algorithms, performance. *IEEE/ACM Transactions on Networking (ToN)*, 14(6):1246–1259, 2006.

[29] K. Weiss, T. M. Khoshgoftaar, and D. Wang. A survey of transfer learning. *Journal of Big data*, 3(1):9, 2016.

[30] L. Xu, K. Harfoush, and I. Rhee. Binary increase congestion control (BIC) for fast long-distance networks. In *IEEE INFOCOM*, volume 4, pages 2514–2524. IEEE, 2004.

[31] P. Yang, J. Shao, W. Luo, L. Xu, J. Deogun, and Y. Lu. TCP congestion avoidance algorithm identification. *IEEE/Acm Transactions On Networking*, 22(4):1311–1324, 2014.

Paper VII

# Enhancing Security Attacks Analysis Using Regularized Machine Learning Techniques

**Desta Haileselassie Hagos**[1], **Anis Yazidi, Øivind Kure, Paal E. Engelstad**

***Published*** in the *31st IEEE International Conference on Advanced Information Networking and Applications (AINA 2017)*, pp. 909–918. IEEE, 2017
DOI: 10.1109/AINA.2017.19.

## Abstract

With the increasing threats of security attacks, *Machine learning (ML)* has become a popular technique to detect those attacks. However, most of the *ML* approaches are black-box methods and their inner-workings are difficult to understand by human beings. In the case of network security, understanding the dynamics behind the classification model is a crucial element towards creating safe and human-friendly systems. In this article, we investigate the most important features in identifying well-known security attacks by using *Support Vector Machines (SVMs)* and $\ell_1$-regularized method with *Least Absolute Shrinkage and Selection Operator (LASSO)* for robust regression both to binary and multiclass attack classification. *SVMs* are one of the standards of *ML* classification techniques that give a reasonably good performance but with some drawbacks in terms of interpretability. On the other hand, *LASSO* is a regularized regression method often performing comparably well and it has extra compelling advantages of being very easily interpretable. *LASSO* provides coefficients that contribute how individual features affect the probability of specific security attack classes to occur.

Hence, we finally use *LASSO* in particular for multiclass classification to help us better understand which actual features shared by attacks in a network are the most important ones. To perform our analysis, we use the recent *NSL-KDD* intrusion detection public dataset where the data are labeled into either anomalous (*denial-of-service (DoS)*, *remote-to-local (R2L)*, *user-to-root (U2R)* and *probe* attack classes) or normal. Empirical

[1]University of Oslo, Department of Informatics, destahh@ifi.uio.no

**Keywords**: Machine Learning, Network Intrusion Detection, SVMs, LASSO, Feature Selection, Bayesian, Classification

results of the analysis and computational performance comparison over the competing methods used are also presented and discussed. We believe that the methodology presented in this paper may strengthen a future research in network intrusion detection settings.

## VII.1 Introduction

As computer and network systems have become more dynamic and complex over the years, chances for attackers to compromise security flaws in these systems have also increased. A full list of security vulnerabilities for computer programs is found at [26]. Even though, static computer network security mechanisms like a firewall can provide a fairly acceptable level of security, more modern and sophisticated Intrusion Detection Systems (IDS) should be used in computer networks. The role of IDS techniques is very crucial in monitoring computer network events for malicious activities, such as attacks against hosts and protecting computer systems and network infrastructures from a potential attack. The problem with the evolution of network threats and attacks is that they are getting harder to detect and therefore it could be difficult to find out whether network traffic is normal or anomalous. Commercially available IDS are mainly signature-based that are designed to detect known attacks by using the signatures of those attacks. Such systems must be frequently updated with rule-sets and signature updates of the recent threat vectors, and are not capable of detecting unknown attacks in network traffic. Examples of the computer attack classes mentioned in this paper are briefly explained in Section VII.2.

Several traditional IDS use a signature-based approach in which events are detected and compared against a predefined database of signatures of known attacks that are provided by an administrator. The traditional approaches to IDS depend on experts or managers codifying rule-sets defining normal behavior and intrusions in a network [36, 40]. The two broad categories of IDS methods are *misuse* and *anomaly detection* [6]. *Misuse detection* is a technique based on rule sets, either pre-configured by the system or setup manually by an administrator. This technique involves matching the signatures of known attacks in a network against events currently taking place in the system that should be considered as misuse [20, 36]. We find this technique mostly used in operational settings. One of the main limitations of this approach is the failure of detecting and identifying unknown computer attacks that do not have known signatures. *Anomaly detection* method, on the other hand, refers to the problem of finding patterns in data that do not comply with an expected notion of normal behavior in a dataset. Everything interpreted as a deviation from the profile of a normal system or user behavior is evidence of a malicious activity [5, 14, 27]. Anomaly detection, however, can detect new attacks but the problem with anomaly detection is that it has a higher false-positive rate.

*ML* techniques have the potential of detecting unknown attacks in network traffic sharing features with other attacks by being trained on normal and abnormal types of traffic. However, one critical problem in *ML* is identifying and selecting the most relevant input features from which to construct an accurate model based on training data for a particular classification task. As we have observed from our evaluation result in Section VII.7 and as it is also reported in the literature [11, 29, 38], employing *ML* techniques on the *NSL-KDD* [21] dataset gives a very low level of detection rate on attack categories involving content features (i.e., *user-to-root (U2R)* and *remote-to-local (R2L)* attacks) within the misuse detection context. However, with the same set of 41 features, the detection rate for *normal*, *denial-of-service (DoS)* and *probe* is accurately high. It is, therefore, important to do feature selection analysis to make it easier for network administrators to better understand the features that contribute to attacks.

In this paper, we address the problem of an actual feature selection for IDS to find attack categories in a network through cross-validated regularized *ML* techniques and an artificial neural network feature ranking methods. Selecting the most relevant actual features improves the detection quality for many algorithms that are based on learning techniques [19]. Feature selection helps to understand better which actual features are the most important ones to find attacks in a network. Therefore, in this paper, our focus is to analyze security attacks by exploring the contribution of the 41 widely used actual input features and selecting the most contributory ones in effectively identifying anomalies in a network with respect to the attack categories. To that end, we have ranked the actual input features into strongly contributing, low contributory and irrelevant using a combination of feature selection filters and wrapper methods by carrying out comparisons with previous works. We investigate the most important features in identifying well-know security attacks by using *SVMs* and $\ell_1$-regularized method with *LASSO*. We use *LASSO* in particular for multiclass classification to help us better understand which actual features shared by attacks in a network are the most important ones. *LASSO* is much more computationally effective and it provides coefficients that contribute to how individual features affect the probability of specific security attack classes to occur. This again allows us to be more specific in the analysis of the different classes of security attacks.

**Our Contributions**

Summaries of the main contributions of our paper are:

- We performed extensive simulation results where we compared feature ranking using both *two-stage*[2] approach using SVM and *one-stage*[3] approach using *LASSO*. We found that *LASSO* provides comparable results at a lower computational cost.
- Despite the simplicity of *LASSO*, we found that it yields a feature ranking that is similar to other well-established state-of-the-art *two-stages* based approaches. Such similarity between the ranking models was thoroughly tested using *Kendall's tau* and *Spearman's footrule* rank distance metrics.
- We use *LASSO* for multiclass classification to give us an insight into features of different classes of attacks.
- We provide a deeper insight from a security engineering perspective on why the features obtained by regularized *ML* techniques are so important in detecting various security attacks in a network.

[2] Apply classification technique first and then feature selection

[3] Feature selection and classification are intertwined using a penalization term

The rest of the paper is organized as follows: Section VII.2 gives an overview of the benchmarking public dataset used in our work. In Section VII.3, we review the state-of-the-art and related works of *SVM* and *LASSO*. Section VII.4 describes the feature selection and ranking techniques. Section VII.5 presents the ranking distance measure metrics. Section VII.6 presents the multiclass feature engineering of the actual network security attack scenarios. Section VII.7 describes the evaluation results. Finally, Section VII.8 concludes the paper with a discussion on future directions.

## VII.2 Methodology

In this paper, we investigate a methodology to examine security attack analysis using two well known *ML* techniques: *SVMs* and $\ell_1$-regularized method with *LASSO* for robust regression both to binary and multiclass attack classification trained on the *NSL-KDD* intrusion detection public dataset [21], which uses TCP/IP level information and embedded with domain-specific heuristics, to detect intrusions at the network level.

Table VII.1: List of full NSL-KDD dataset features.

| ID | Feature Name | Type | Description |
|---|---|---|---|
| | | | *Basic features of individual TCP connections* |
| f1 | *duration* | Integer | Duration of the connection in seconds |
| f2 | *protocol_type* | Nominal | Protocol type of the connection: TCP, UDP, ICMP |
| f3 | *service* | Nominal | Network service on the destination, e.g., *http*, *ftp*, *smtp*, *telnet*, etc. |
| f4 | *flag* | Nominal | Normal or error status of the connection: SF, S0, S1, S2, S3, OTH, REJ, RSTO, RSTOS0, SH, RSTRH, SHR |
| f5 | *src_bytes* | Integer | Number of data bytes sent in one connection from source to a destination |
| f6 | *dst_bytes* | Integer | Number of data bytes received in one connection from destination to a source |
| f7 | *land* | Binary | 1 if source and destination IP addresses and port numbers are equal; 0 otherwise |
| f8 | *wrong_fragment* | Integer | Number of bad checksum packets in a connection |
| f9 | *urgent* | Integer | Number of urgent packets in a connection |
| | | | *Content features within a connection suggested by domain knowledge* |
| f10 | *hot* | Integer | Number of hot actions (indicators) in a connection such as: entering a system directory, creating and executing programs |
| f11 | *num_failed_logins* | Integer | Number of failed (incorrect) login attempts in a connection |
| f12 | *logged_in* | Binary | 1 if successfully logged in; 0 otherwise |
| f13 | *num_compromised* | Integer | Number of compromised conditions in a connection |
| f14 | *root_shell* | Binary | 1 if the root gets the shell; 0 otherwise |
| f15 | *su_attempted* | Binary | 1 if the "*su*" command has been used; 0 otherwise |
| f16 | *num_root* | Integer | Number of root accesses (operations) in a connection |
| f17 | *num_file_creations* | Integer | Number of file creation operations in a connection |
| f18 | *num_shells* | Integer | Number of shell prompts (logins of normal users) |
| f19 | *num_access_files* | Integer | Number of operations in access control files in a connection |
| f20 | *num_outbounds_cmds* | Integer | Number of outbound commands in an ftp session |
| f21 | *is_hot_login* | Binary | 1 if the user is accessing as root or admin group; 0 otherwise |
| f22 | *is_guest_login* | Binary | 1 if the login is a "guest" login; 0 otherwise |
| | | | *Traffic features computed using a two-second time window* |
| f23 | *count* | Integer | Number of connections to the same destination IP address as the current connection in the past two seconds |
| f24 | *srv_count* | Integer | Number of connections whose service type is the same as the current connection in the past two seconds |
| f25 | *serror_rate* | Real | The percentage of connections that have "SYN" errors in the *count* feature (f23) |
| f26 | *srv_serror_rate* | Real | The percentage of connections that have "SYN" errors in the *srv_count* feature (f24) |
| f27 | *rerror_rate* | Real | The percentage of connections that have "REJ" errors in the *count* feature (f23) |
| f28 | *srv_rerror_rate* | Real | The percentage of connections that have "REJ" errors in the *count* feature (f23) |
| f29 | *same_srv_rate* | Real | The percentage of connections to the same service among the connections in the *count* feature (*f23*) |
| f30 | *diff_srv_rate* | Real | The percentage of connections to different services among the connections in the *count* feature (*f23*) |
| f31 | *srv_diff_host_rate* | Real | The percentage of connections to different hosts among the connections in the *srv_count* feature (*f24*) |
| f32 | *dst_host_count* | Integer | Number of connections to the same destination IP address |
| f33 | *dst_host_srv_count* | Integer | Number of connections to the same destination port number |
| f34 | *dst_host_same_srv_rate* | Real | The percentage of connections that were to the same service among the connections in the *dst_host_count* features (f32) |
| f35 | *dst_host_diff_srv_rate* | Real | The percentage of connections that were to the different services among the connections in the *dst_host_count* feature (*f32*) |
| f36 | *dst_host_same_src_port_rate* | Real | The % of connections that were to the same source port among the connections in the *dst_host_srv_count* feature (*f33*) |
| f37 | *dst_host_srv_diff_host_rate* | Real | The % of connections that were to different destination hosts among the connections in the *dst_host_srv_count* feature (*f33*) |
| f38 | *dst_host_serror_rate* | Real | The percentage of connections that have "SYN" errors in the *dst_host_count* feature (*f32*) |
| f39 | *dst_host_srv_serror_rate* | Real | The percentage of connections that have "SYN" errors in the *dst_host_srv_count* feature (*f33*) |
| f40 | *dst_host_rerror_rate* | Real | The percentage of connections that have "REJ" errors in the *dst_host_count* feature (*f32*) |
| f41 | *dst_host_srv_rerror_rate* | Real | The percentage of connections that have "REJ" errors in the *dst_host_srv_count* feature (*f33*) |

Table VII.2: Flag feature attribute values description.

| Flags | Description |
|---|---|
| *REJ* | Connection attempt rejected |
| *RSTO* | Connection established, originator aborted (sent a RST) |
| *RSTOS0* | Originator sent a SYN followed by an RST |
| *RSTR* | Connection established, responder aborted |
| *S0* | Connection attempt seen, no reply (only the first SYN packet is sent) |
| *S1* | Connection established, not terminated |
| *S2* | Connection established and close attempt by the initiator |
| *S3* | Connection established and closed by the responder |
| *SF* | Normal SYN/FIN completion |
| *SH* | A state 0 connection was closed before we ever saw the SYN ack |
| *OTH* | No SYN seen |

### VII.2.1 Overview of NSL-KDD Dataset

In this paper, we used the *NSL-KDD* [21] benchmarking public dataset. It is an improved version of the old *KDD* [4] and is suggested to solve some critical problems mentioned in [31, 43]. The KDD dataset [4] was created by processing the *tcpdump* portions of the 1999 DARPA evaluation dataset collected in a military network at MITś Lincoln Labs to study intrusion detection [30]. The *NSL-KDD* public dataset [21] covers attacks which fall into *4* main categories: *denial-of-service (DoS)*, *remote-to-local (R2L)*, *user-to-root (U2R)* and *probe*. For different examples of each of these attack categories, we refer the reader to Figure VII.1.

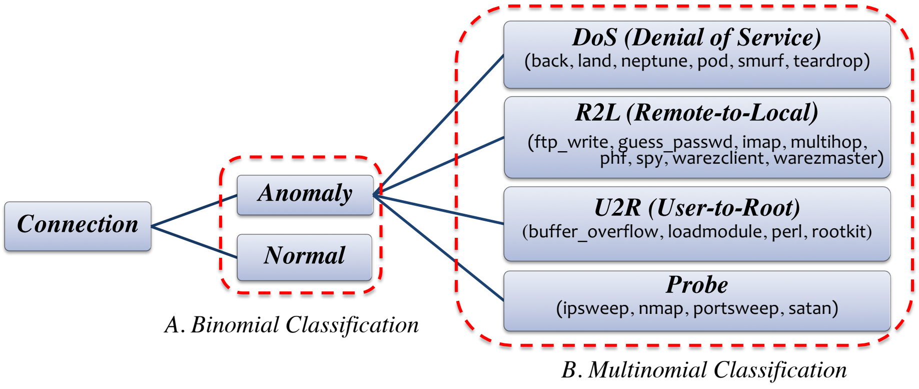

Connection
Anomaly
Normal
A. Binomial Classification
DoS (Denial of Service)
(back, land, neptune, pod, smurf, teardrop)
R2L (Remote-to-Local)
(ftp_write, guess_passwd, imap, multihop, phf, spy, warezclient, warezmaster)
U2R (User-to-Root)
(buffer_overflow, loadmodule, perl, rootkit)
Probe
(ipsweep, nmap, portsweep, satan)
B. Multinomial Classification

Figure VII.1: Attacks Classification.

1. ***Denial-of-Service (DoS) attacks***: When an attacker tries to make a network resource unavailable or too busy in order to prevent legitimate users from accessing information or using a service a system provides in a network e.g., *syn flood.*

2. ***Remote-to-Local (R2L) attacks***: These are a class of exploits in which the attacker does not have an account on the victim machine, hence exploits some vulnerability to gain local user access from a remote machine e.g., *guessing a password.*

3. ***User-to-Root (U2R) attacks***: This occurs when an attacker has local access to the victim machine and tries to gain root privileges e.g., various *"buffer overflow"* attacks.

4. ***Probing attacks***: these are a class of exploits that take place whenever the attacker tries to gather information (or find known potential vulnerabilities) about the target host(s) by automatically scanning a network of computers [13] e.g., *port scanning.*

Like *DARPA* [30] and *KDD* [4] datasets, the *NSL-KDD* public dataset [21] consists of a total of 41 features [28] for the analysis and one target predictor that indicates the attack category name. The set of features listed in Table VII.1 characterizing each connection are divided into the following *three* categories.

1. ***Basic features***: These groups of features encapsulate all the basic characteristics that can be derived from packet headers of an individual TCP/IP connection.

2. ***Content features***: These features rely on a connection suggested by domain knowledge to define suitable features for *R2L* and *U2R* attacks. Unlike most of the *DoS* and *probing*, the *R2L* and *U2R* attacks do not have any frequent pattern for intrusions. This is because *DoS* and *probing* attacks involve sending a lot of connections to some host(s) in a very short period of time. On the other hand, the *R2L* and *U2R* attacks are embedded in the data portions of the packets and normally involve only a single connection. In order to detect such kinds of attacks, we have to look at the content of the connection and therefore we need "content" features that indicate whether the data contents suggest suspicious behavior or not.

3. ***Traffic features***: These groups also called *connection-based* traffic features are computed using a time window interval. These are divided into *two* groups: *"same host"* and *"same service"* connection features (also called "time-based" traffic features of the connection records). The "*same host*" features examine only the connections computed in the past 2 seconds time window that has the same destination host as the current connection and calculate statistics related to *protocol behavior*, *service*, etc. The *same service* traffic features examine only the connections that

have the same service as the current connection and are computed using a window of *100* connections instead of 2 seconds time window. This is because there are slow probing attacks that scan the targeted host(s) using a much larger time window than 2 seconds. Therefore, in order for such attacks to produce an intrusion pattern, these features have to be re-calculated using a connection window of *100* connections.

## VII.3 Classification and Regression

### VII.3.1 Binary Classification Using SVMs

SVMs in *ML* are widely used supervised learning models for classification [3, 41], regression analysis [15, 48] and density estimation problems [32]. In classification, we are given $n$ labeled training data samples of the form $\{(x_i, y_i)|x_i \in \mathbb{R}^n, y \in \{-1, 1\}\}^n$ for $i = 1, 2, \ldots, n$ where $x_i \in \mathbb{R}^n$ represents an input feature of $n$-vectors of real-valued predictors for the $i^{th}$ observation that describe the training data point and $y_i \in \{\pm 1\}$ represent the class label of the $i^{th}$ data sample to which class the point $x_i$ belongs. In our evaluation, we used an *SVM* with a *Radial Basis Function (RBF)* kernel for classification. In order to perform an efficient SVM classification through cross-validation, we have applied *SVM tuning* using a model selection to find the best parameters $C$ and $\gamma$ which yield the least error and the best accuracy for non-linear SVM.

Before applying a regression model for linear analysis, since the *NSL-KDD* dataset is mixed with both *numerical* and *categorical* features, we applied a *Multiple Correspondence Analysis (MCA)* [1] so that each feature is represented as a vector of continuous values. We converted the categorical features (*protocol_type* with 3[4] attribute values, *service* with 71[5] attribute values, and *flag* with 11[6] attribute values) into continuous by creating binary features in every column. This means we expand the matrix and normalize the dataset per column and therefore the features are linearly interrelated. For example, `100` for *TCP*, `010` for *UDP* and `001` for *ICMP*. The resulting feature vectors used after binarization have *123*[7] dimensions out of which *92* normalized non-zero coefficient features for binary classification are presented in Table VII.4. The impact of this increased input dimensionality in the overall complexity of SVM is not that significant. However, the performance depends more on the convergence criteria inside the training algorithms and the sample size. The *negative* coefficients in the result identify as a normal and whereas the *positive* coefficients identify as an attack.

[4] *tcp, udp, icmp*

[5] *aol, auth, bgp, courier, csnet_ns, ctf, daytime, discard, domain, domain_u, echo, eco_i, ecr_i, efs, exec, finger, ftp, ftp_data, gopher, harvest, hostnames, http, http_2784, http_443, http_8001, r2l4, imap4, IRC, iso_tsap, klogin, kshell, ldap, link, login, mtp, name, netbios_dgm, netbios_ns, netbios_ssn, netstat, nnsp, nntp, ntp_u, other, pm_dump, pop_2, pop_3, printer, private, red_i, remote_job, rje, shell, smtp, sql_net, ssh, sunrpc, supdup, systat, telnet, tftp_u, tim_i, time, urh_i, urp_i, uucp, uucp_path, vmnet, whois, X11, Z39_50*

[6] *REJ, RSTO, RSTOS0, RSTR, S0, S1, S2, S3, SF, SH, OTH*

[7] $38 + 3 + 71 + 11 = 123$

Table VII.3: LASSO: Beta values for binary classification.

| **Number of Beta ($\beta$)Values** | **Categories** | $\lambda$ Values |
|---|---|---|
| Total Number of Beta Values | 92 | 7.937784e-05 |
| Non-Zero Beta Values | 84 | 0.0001670827 |
| Non-Zero Beta *Values_10* | 3 | 0.3130826 |
| Non-zero Beta *Values_25* | 11 | 0.1487396 |
| Non-Zero Beta *Values_30* | 12 | 0.135526 |
| Non-Zero Beta *Values_35* | 17 | 0.08511426 |
| Non-Zero Beta *Values_40* | 19 | 0.07066335 |
| Non-Zero Beta *Values_50* | 30 | 0.02539509 |
| Non-Zero Beta *Values_100* | 84 | 0.0001670827 |

## VII.3.2 LASSO with $\ell_1$-Regularization

In order to perform $\ell_1$-regularized feature selection, we recommend applying a one-stage approach using *LASSO* fitting a predictive model with a binary target. This again improves model performance as feature selection and classification analysis in *LASSO* are intertwined. While other approaches including SVM are *two-stages* which require more training time and they are computationally inefficient. The *LASSO* regression uses a shrinkage method which allows a variable to be partly included in the regression model to perform feature selection where the estimated coefficients are shrunken towards zero as $\lambda$ increases [44].

$$Minimize : \sum_{i=1}^{n}(Y_i - \sum_{j=1}^{p} X_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} |\beta_j| \tag{VII.1}$$

Where $\lambda \geq 0$ is a nonnegative tuning parameter controlling the degree of regularization. *LASSO* performs a model selection based on the shrinkage operator $\lambda$ which controls the size of the $\beta$ coefficients and the degree of sparsity in $\beta$. The feature selection in *LASSO* is performed by checking the coefficient vector $\beta$ which tells how relevant an actual feature is. Table VII.3 shows which $\lambda$ value gives a corresponding number of optimized $\beta$ values in the selection of normalized features for binary classification. In Table VII.4, we have a total of *84 non-zero* $\beta$ coefficients of the normalized features. The *92* optimized non-zero $\beta$ values and the corresponding $\lambda$ values shown in Table VII.3 are distributed into all the normalized features. For example, the first 10 $\beta$ values go into 3 categories, the next 25 $\beta$ values goes into 11 categories, etc. In our evaluation, the most significant actual features getting the highest $\beta$ value in a descending order are shown in Table VII.5.

## VII.4 Feature Selection and Ranking Methods

Feature selection in IDS is used to eliminate the irrelevant or redundant input features and utilize a few numbers of features. Feature selection has an important effect on IDS and some of its benefits [10, 24, 34] are: providing a better understanding of the most important features and the underlying process that generated the dataset, making classification models more efficient, reducing training times, providing cost-effective and faster predictors, avoiding overfitting, increasing prediction performance, etc. In the context of classification, there are *three* main categories of feature selection techniques [16, 39]: *filter methods* [37], *wrapper methods* [22] and *embedded methods* [39]. Because of space constraints, the basic concepts of feature selection techniques will not be discussed in detail here, as they are well-known and documented elsewhere. Our focus in this work is using *wrapper* feature selection techniques. Wrapper methods determine subsets of features according to their relevance to a given predictor [16].

In this paper, we have employed a cross-validated $\ell_1$-norm SVM-based feature selection algorithm called *Recursive Feature Elimination (RFE)* [17]. The RFE [17] algorithm constructs the feature ranking from the SVM training stage where the weights are assigned to the different entries of the classification problem. In order to compute the relevance of the features, RFE uses the separating hyperplane from the support vectors. RFE works based on a *greedy* algorithm called *backward feature selection*[8] [22]. The main objective is to select a subset of size $S$ among $f$ features ($S<f$) which maximizes the prediction accuracy. The feature selection process can be expressed as the following: given a feature set $X = (x_1, x_2, \ldots, x_n)$ where $x_i \in \mathbb{R}^n$ and a subset $Y = (y_1, y_2, \ldots, y_k)$ of $\boldsymbol{X}$ with $\boldsymbol{k<n}$, which optimizes an objective function $\boldsymbol{W(X)}$ by removing one weak[9] feature at a time. As presented in Algorithm 1, we keep on iterating until we observe a sharp drop in the predictive accuracy of the model.

The *For loop* function in Algorithm 1 (line 7) controls the contribution of each feature to the overall classification of the attacks. For a given class $X$, we determine the right number of features (line 14) and select the $N$ features that have the highest values. Each feature is ranked using its importance to the final model. According to [19], of all the *41* features presented in Table VII.1, 16 features (*f3, f4, f6, f8, f23, f25, f26, f28, f29, f32, f33, f34, f35, f38, f39, f40*) are selected to be of strong significance in the anomaly detection. However, *14* features (*f2, f5, f7, f10, f12, f13, f14, f15, f17, f18, f22, f24, f27, f41*) are of very little significance in the anomaly detection. The remaining *11* features (*f1, f9, f11, f16, f19, f20, f21, f30, f31, f36, f37*) are insignificant. In our evaluation, out of all the *17* most relevant features (*f2, f3, f4, f6, f8, f23, f25, f26, f28, f29, f32, f33, f34, f35, f38, f39, f40*), *16* of them are of strong significance in the anomaly detection as described earlier in this section. Our evaluation results show that the *SVM* and *LASSO* classifier accuracies of all the features including strong and low contributing ones are 79% and 83% respectively. However, the

[8]A search that starts with the full set of features and sequentially eliminates one feature at a time

[9]Feature with the least absolute coefficient in a linear model

**Algorithm 1** Feature selection algorithm using RFE

```
1:  procedure PRE-PROCESS THE DATA(e.g., normalization)
2:      for Each resampling iteration (10-fold cross-validation) do
3:          Partition data into training and test sets
4:          Train/tune the model using all the features
5:          Predict the held-back samples
6:          Calculate rankings to the model for each feature
7:          for Each subset size $X_i$, $i = 1 \dots X$ do
8:              Keep the $X_i$ most important features
9:              Train/tune the model using $X_i$ features
10:             Predict the held-back samples
11:         end for
12:         end for
13:         Find the accuracy over the $X_i$ using the samples
14:         Select the appropriate number of features
15:         Estimate the list of final features to keep in the final model
16:         Fit the model based on the optimal $X_i$ using the training set
17:     end procedure
```

*SVM* and *LASSO* accuracy of the 16 strongly contributing actual features is comparably high as shown in Section VII.7.

Table VII.4: Binary: Non-Zero Coefficient Features.

| ID | Selected Features | Coefficients | |
|---|---|---|---|
| ***f29*** | *same_srv_rate* | 9.6584405 | |
| ***f33*** | *dst_host_srv_count* | 8.8495808 | |
| ***f39*** | *dst _host_ srv_serror_rate* | 8.6792135 | |
| ***f34*** | *dst_host_same_srv_rate* | 8.6109899 | |
| ***f25*** | *serror_rate* | 7.9176348 | |
| ***f26*** | *srv_serror_rate* | 7.8804150 | |
| ***f38*** | *dst_host_serror_rate* | 7.1196058 | |
| ***f23*** | *count* | 6.9890928 | |
| ***f32*** | *dst_host_count* | 6.8792587 | |
| ***f40*** | *dst_host_rerror_rate* | 6.6997582 | |
| ***f35*** | *dst_host_diff_srv_rate* | 6.4034485 | |
| ***f28*** | *srv_rerror_rate* | 6.3596267 | |
| ***f4*** | *flagREJ* | 5.9594539 | |
| ***f3*** | *serviceeco_i* | 5.8488743 | |
| ***f3*** | *serviceecr_i* | 5.7939503 | |
| ***f3*** | *servicedomain_u* | 5.6628783 | |
| ***f3*** | *servicehttp* | 5.6548738 | |
| ***f3*** | *servicesmtp* | 5.5966379 | |
| ***f8*** | *wrong_fragment* | 5.5805339 | |
| ***f4*** | *flagSO* | 5.5728882 | |
| ***f4*** | *flagRSTO* | 5.5427097 | |
| ***f4*** | *flagRSTR* | 5.5210019 | |
| ***f4*** | *flagS1* | 5.4825985 | |
| ***f3*** | *servicenetstat* | 5.4398630 | |
| ***f3*** | *serviceremote_job* | 5.3662737 | |
| ***f3*** | *servicesystat* | 5.1609441 | |
| ***f3*** | *servicerje* | 5.0520126 | |
| ***f3*** | *serviceftp_data* | 4.5860699 | |
| ***f3*** | *serviceIRC* | 4.5360582 | |
| ***f3*** | *servicetelnet* | 4.5133403 | |
| ***f3*** | *serviceother* | 3.8991619 | |
| ***f2*** | *protocol_typeudp* | 3.8113397 | |
| ***f3*** | *servicefinger* | 3.7466049 | |
| ***f3*** | *servicetime* | 3.7452173 | |
| ***f3*** | *serviceurp_i* | 3.6329897 | |
| ***f17*** | *num_file_creations* | 3.5458035 | |
| ***f27*** | *rerror_ rate* | 3.2361194 | |
| ***f3*** | *servicepop_3* | 3.2291595 | |
| ***f16*** | *num_root* | 3.2152647 | |
| ***f3*** | *servicedomain* | 3.1977012 | |
| ***f3*** | *serviceX11* | 3.1686796 | |
| ***f3*** | *serviceftp* | 2.1324968 | |
| ***f4*** | *flagSF* | 2.1020335 | |
| ***f5*** | *src_bytes* | 2.0914915 | |
| ***f4*** | *flagS2* | 2.0727762 | |
| ***f14*** | *root_shell* | 2.0682542 | |
| ***f4*** | *flagSH* | 2.0604952 | |
| ***f4*** | *flagS3* | 2.0550322 | |
| ***f2*** | *protocol_typetcp* | 1.2548399 | |
| ***f2*** | *protocol_typeicmp* | 1.0515946 | |
| ***f1*** | *duration* | 0.0430454 | |
| ***f3*** | *servicessh* | 0.0078574 | |
| ***f3*** | *serviceimap4* | | -0.0288315 |
| ***f3*** | *servicesupdup* | | -0.0348969 |
| ***f15*** | *su_attempted* | | -0.0438128 |
| ***f3*** | *servicenetbios_dgm* | | -0.0521436 |
| ***f13*** | *num_compromised* | | -0.0587089 |
| ***f4*** | *flagSF* | | -0.0607704 |
| ***f4*** | *flagS1* | | -0.0711349 |
| ***f3*** | *servicevmnet* | | -0.0733044 |
| ***f3*** | *servicepop_2* | | -0.0748774 |
| ***f3*** | *servicekshell* | | -0.0816983 |
| ***f3*** | *servicemtp* | | -0.0857725 |
| ***f4*** | *flagREJ* | | -0.0975255 |
| ***f4*** | *flagS2* | | -1.0987939 |
| ***f3*** | *serviceklogin* | | -1.1033372 |
| ***f4*** | *flagS3* | | -1.1200202 |
| ***f3*** | *servicebgp* | | -1.1201005 |
| ***f3*** | *servicename* | | -1.1239379 |
| ***f3*** | *servicenntp* | | -1.1281446 |
| ***f3*** | *serviceefs* | | -1.1301224 |
| ***f3*** | *servicehostnames* | | -1.1415551 |
| ***f3*** | *serviceiso_tsap* | | -2.1419084 |
| ***f3*** | *serviceZ39_50* | | -2.1421581 |
| ***f3*** | *servicecsnet_ns* | | -2.1452788 |
| ***f3*** | *servicegopher* | | -2.1596109 |
| ***f3*** | *servicewhois* | | -3.1636177 |
| ***f3*** | *servicennsp* | | -3.1652713 |
| ***f3*** | *serviceuucp* | | -3.1669783 |
| ***f3*** | *servicecourier* | | -4.1687838 |
| ***f3*** | *servicehttp_443* | | -4.1695469 |
| ***f3*** | *servicectf* | | -5.1771981 |
| ***f3*** | *serviceprivate* | | -5.5384066 |
| ***f6*** | *dst_bytes* | | -6.8889978 |

### VII.4.1 Ranking models and Importance of Features

Since feature selection is important for selecting the relevant features during the data-preprocessing step to reduce space dimensionality, its accuracy should be evaluated after combining with feature ranking algorithms. In our evaluation analysis, an artificial neural network algorithm called *Learning Vector Quantization (LVQ)* and one-stage approach using *LASSO* are used for feature ranking. LVQ is a supervised prototype algorithm widely used for classification of vectorized data and feature ranking in the field of artificial neural networks [23]. It supports both binomial and multinomial classification problems. To understand how LVQ works, let us, for example, assume that a clustering of data into $K$ classes is to be learned and a set of n-dimensional data $\{(x_i, y_i) \subset \mathbb{R}^n \times \{1, 2, \ldots, K\} | i = 1, \ldots, n\}$ is given. The class labels or categories are given as $\{1, 2, \ldots, K\}$. The components of a vector $x \in \mathbb{R}^n$ are given as $x = (x_1, \ldots, x_n)$. LVQ chooses every class $K$ by a weight vector $W_j$ in $\mathbb{R}^n$. The distance between $x_i$ and $w_j$ is given by the weighted distance metric $dist(x_i, w_j, \lambda)$ applying the feature weights vector $\lambda = [\lambda_i, \ldots, \lambda_n]$, $\sum_{k=1}^{n} \lambda_k$. The weight vector $\lambda_k$ measures the importance of the $k^{th}$ of the input vector $x_i$. Rank of the $k^{th}$ feature at step $i$ of the LVQ training is an average of all $\lambda_k$, for $k = 1, \ldots, i$.

Table VII.5: Comparison of ranking techniques.

| | Related Work Techniques | | | Our Techniques | |
|---|---|---|---|---|---|
| Rank | WMR | SAM | MRMR | LVQ | LASSO |
| 1 | *f29* | *f29* | *f29* | *f29* | *f29* |
| 2 | *f33* | *f33* | *f33* | *f33* | *f33* |
| 3 | *f34* | *f34* | *f34* | *f34* | *f39* |
| 4 | *f39* | *f39* | *f39* | *f39* | *f34* |
| 5 | *f38* | *f38* | *f23* | *f26* | *f25* |
| 6 | *f25* | *f25* | *f25* | *f23* | *f26* |
| 7 | *f26* | *f26* | *f38* | *f25* | *f38* |
| 8 | *f23* | *f23* | *f26* | *f38* | *f23* |
| 9 | *f32* | *f32* | *f32* | *f32* | *f32* |
| 10 | *f40* | *f40* | *f35* | *f35* | *f40* |
| 11 | *f28* | *f35* | *f40* | *f40* | *f35* |
| 12 | *f35* | *f28* | *f28* | *f28* | *f28* |

## VII.5 Distance Between Ranking Models

Our analysis is not only reducing the number of actual features but also to understand their contribution to the classes of the attacks. As recommended by the authors in [16], we used filters to obtain and compare feature rankings and baseline results. The authors in [19] carry out feature rankings with filter algorithms based on *Weight by Maximum Relevance (WMR) [2]*, *Minimum*

*Redundancy Maximum Relevance (MRMR) [8]* and *Significance Analysis for Microarrays (SAM) [47]*. First, we normalize filters ranking for a coarse-tuning according to the maximum and minimum values and rearrange features in descending order. Finally, we used wrapper feature selection techniques for a fine-tuning. As shown in Table VII.5, the comparison of the diverse feature selection techniques showed strong positive agreement with an acceptable classification performance. All selection techniques agreed about finding traffic features as relevant and content features as irrelevant. We adopted *two* well-known ranking distance measure metrics in the evaluation of how similar our ranking algorithms are in relative to these methods as presented in Table VII.6 for the most relevant features using: *Kendall tau* and *Spearman's footrule distance*.

## VII.5.1 Kendall's tau ranking distance

*Kendall's tau* rank distance is a metric used to measure (count) the number of pairwise disagreements between two rankings on the same domain [25, 33]. Let $\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ denote a random sample of $n$ observation from a vector $(X, Y)$ of continuous random variables such that all the values of $(x_i)$ and $(y_i)$ are unique. For $n$ objects of observations in the sample, there are $\binom{n}{2}$ unique pairs to compare $(x_i,\ y_i)$ and $(x_j,\ y_j)$. *Kendall's tau ($\tau$)* [33] is given by:

$$K(\sigma) = \sum_{(i,j):i>j} [\sigma(i) < \sigma(j)]; with\ i \neq j \tag{VII.2}$$

This means, given a ranking sequence $\sigma(1), \ldots, \sigma(n)$, where $1, \ldots, n$ is the rankings of the $n$ objects of observations in the sample. For example, *Kendall's tau* ranking distance between two ranking predictions *R1* and *R2* is given by: $K(T_1, T_2) = |\{(i, j) : i < j, (T_1(i) < T_1(j) \wedge T_2(i) > T_2(j)) \vee (T_1(i) > T_1(j) \wedge T_2(i) < T_2(j))\}|$, where $T_1(i)$ and $T_2(i)$ are the rankings of the element $i$ in $R_1$ and $R_2$ respectively, thus $K(\tau)$ measures the total number of pairwise inversions [25]. In Table VII.6, $d_k$ represents the number of pairs whose values are in opposite order (also called "*discordant pairs*"). Since there are 7 discordant pairs between the algorithms WMR and LVQ, the *Kendall's tau* distance is 7. The *normalized Kendall's tau* distance (which lies in the interval [*0, 1*])[10] is defined by:

$$\tau = \frac{number\ of\ discordant\ pairs}{\frac{n(n-1)}{2}} \tag{VII.3}$$

A value of *0.10* indicates that 10% of pairs differ in ordering between the two ranking methods, i.e., they are 90% in agreement. The larger the normalized *Kendall's tau* rank distance, the more dissimilar the two ranking methods are.

[10] A value of 1 indicates maximum disagreement between the rankings

### VII.5.2 Spearman's footrule distance

The *Spearman's footrule distance* is the sum of the absolute values of the difference between the ranks [7, 25, 42]. The *Spearman's footrule* distance can be defined as:

$$K(\sigma) = \sum_i |i - \sigma(i)| \tag{VII.4}$$

As shown in Table VII.6, our results of the normalized *Spearman's footrule* distance ($\rho$) show that there is a strong positive agreement between the rankings. That means, our rankings are in agreement with the ranking methods presented in [19] to a large extent. The normalized *Spearman's footrule* distance ($\rho$) is defined by the following equation where $d_S$ is the difference between the ranks of each observation.

$$\rho = 1 - \frac{6\sum d_S^2}{n(n^2-1)} \tag{VII.5}$$

*Remark*: The higher the $\rho$ value is, the higher the similarity between the ranking techniques. This means, there is a linear association between the rankings in comparison.

Table VII.6: Distance between ranking algorithms: Comparison.

| | *Kendall's tau Distance* | | | | *Spearman's Footrule Distance* | | | |
|---|---|---|---|---|---|---|---|---|
| | LVQ | | LASSO | | LVQ | | LASSO | |
| | $d_k$ | *tau* ($\tau$) | $d_k$ | *tau* ($\tau$) | $d_S$ | $\rho$ | $d_S$ | $\rho$ |
| WMR | 7 | 0.10 | 4 | 0.06 | 12 | 0.916 | 8 | 0.965 |
| SAM | 6 | 0.09 | 3 | 0.045 | 10 | 0.961 | 6 | 0.972 |
| MRMR | 3 | 0.045 | 6 | 0.09 | 6 | 0.958 | 10 | 0.989 |

## VII.6 Multiclass Classification using LASSO

Binary classification has been intensively studied and in this paper, we go beyond that and study multiclass classification using *LASSO* with $\ell_1$-regularization technique which allows us to be more specific in the analysis of the different classes of attacks. In addition to the benefits explained above, the great thing about $\ell_1$-regression is that it is *non-continuous*. *LASSO* is very intuitive with regression and allows us to explore a lot of things in terms of features selection for attack analysis. Regression is such that with all the sets of features, we get the best accuracy. We applied *LASSO* for feature selection by selecting the optimal value of $\lambda$ from a number of $\lambda$ values that gives the best prediction accuracy. We increased the penalty on the regression so that we reduce the number of optimal non-zero $\beta$ values. This means the higher the penalty is, the more $\beta$ values will be pushed to *zero*. In multiclass *LASSO*, we get different $\beta$

values for each attack class. However, when we perform binary classification, we obtain one set of features and one bias.

The main benefit of *LASSO* for multiclass security attack classification is that it gives us a deeper understanding of the features of different attack categories. This helps to provide a deeper insight from a security engineering perspective on why the corresponding selected features are so important in clearly detecting a certain attack. There are `+ve` and `-ve` coefficient values which increase or decreases the probability of a certain attack as shown in Table VII.7. For example, the feature *wrong_fragment* with a coefficient of `12.271653598` means that this feature increases the probability that it is a *DoS* attack. Features *wrong_fragment* and *same_srv_rate* are among the most contributing to a *DoS* attack. This makes sense because TCP fragmentation (also called *Teardrop*) are *DoS* attacks that prevent reassembly protocols from putting together a fragmented UDP traffic packets sent across the network to the intended destination by rebooting the targeted host. *DoS* attack is sending a lot of traffic to the same service to block the communication channel and therefore *count, src_bytes, flagS0* etc. are among the contributing features for such an attack, e.g., *TCP SYN* flood attacks. The attacker may use many spoofed source IP addresses by sending a lot of TCP connections with *S0* flag to a port of the targeted host in a time window of *T* seconds. The *flag* features as shown in Table VII.2 are the *flags* in the TCP segment header and *flag* is an important feature in identifying attack patterns because it shows the summary information of the connection behavior with regard to the network protocol specification. Some contributing features of the *SYN* flood are percentage of connections that have the *"S0"* flag, *count*, *dst_host_count*, *srv_count* (connections that have the same service) etc. Depending on its good spread on the different services and in relationship to what is a positive contribution, the *dst_host_diff_srv_rate* feature discounts for a *DoS* attack. This means it will be an attack if there is no spread in the services. If the same service sends an *icmp* echo replies (*serviceecr_i*) to the same destination IP address, this may lead to a *smurf* attack (*count, srv_count*), i.e., a *DoS* where its effect is slowing down the network. If the source or destination IP address and port numbers are equal with TCP connection flag of either *"S0"*, *"REJ"* or *"RST0"*, then it could be a network attack of type *neptune* (*DoS*) that may eventually slow down the server response.

Out of all the 71 *service* feature attribute values, *serviceeco_i* used for port scanning is the most significant contributing to *ipsweep*: identifies active hosts, *portsweep*, and *nmap*: which lists open ports on a network, *satan* (*dst_host_diff_srv_rate, srv_rerror_rate*) i.e., *probe* attacks. *ICMP ping* is one of the most used tools in IP firewalls. Probe attacks are associated with the scanning of open ports and running services on a live host during an attack. For example, to find the list of running SSH services *servicessh* on a network, an attacker may probe many IP addresses on the default SSH listening port. If a response from the probe of the default SSH port is received, the attacker may launch a brute force attack on the target host. In order to launch a probe attack, the duration of a connection usually lasts longer than usual,

Table VII.7: Multiclass: Non-Zero Coefficient Features.

| Attack Class | ID | Selected Features | Coefficients | |
|---|---|---|---|---|
| DoS | f8 | wrong_fragment | 12.271653598 | |
| | f29 | same_srv_rate | 5.768419027 | |
| | f3 | serviceecr_i | 4.315135582 | |
| | f4 | flagS0 | 3.997512139 | |
| | f32 | dst_host_count | 3.902672852 | |
| | f5 | src_bytes | 3.894714431 | |
| | f23 | count | 3.831534986 | |
| | f25 | serror_rate | 3.617634819 | |
| | f33 | dst_host_srv_count | 2.007112749 | |
| | f24 | srv_count | 2.005255132 | |
| | f38 | dst_host_serror_rate | 1.024086592 | |
| | f40 | dst_host_rerror_rate | 0.413642565 | |
| | f39 | dst_host_srv_serror_rate | 0.339514109 | |
| | f2 | protocol_typetcp | 0.238965656 | |
| | f3 | serviceftp_data | | -0.002862796 |
| | f4 | flagSF | | -0.879086505 |
| | f35 | dst_host_diff_srv_rate | | -3.164487231 |
| Normal | f13 | num_compromised | 11.289718672 | |
| | f3 | serviceIRC | 9.672086382 | |
| | f3 | serviceurp_i | 9.277818694 | |
| | f3 | servicehttp | 8.949948108 | |
| | f4 | flagSF | 8.678467594 | |
| | f3 | servicedomain_u | 7.576078198 | |
| | f29 | same_srv_rate | 6.407448247 | |
| | f3 | servicesmtp | 6.405202769 | |
| | f3 | serviceX11 | 5.725198610 | |
| | f4 | flagS1 | 5.294657953 | |
| | f15 | su_attempted | 5.231055104 | |
| | f1 | duration | 4.928014017 | |
| | f4 | flagS2 | 4.319417825 | |
| | f4 | flagS3 | 3.134242962 | |
| | f4 | flagREJ | 2.780086083 | |
| | f3 | servicepop_3 | 1.004959401 | |
| | f2 | protocol_typeudp | 0.356947842 | |
| | f2 | protocol_typeicmp | 0.233562559 | |
| | f4 | flagRSTR | | -0.277986975 |
| | f40 | dst_host_rerror_rate | | -0.320941948 |
| | f24 | srv_count | | -3.015266290 |
| | f23 | count | | -7.145703431 |
| | f39 | dst_host_srv_serror_rate | | -9.054475025 |
| Probe | f3 | serviceeco_i | 9.68002664 | |
| | f3 | servicenetstat | 9.440867487 | |
| | f3 | servicessh | 8.412049192 | |
| | f3 | servicesystat | 8.413300736 | |
| | f3 | serviceremote_remote | 7.158598362 | |
| | f3 | servicerje | 7.032530367 | |
| | f35 | dst_host_diff_srv_rate | 6.884222244 | |
| | f5 | src_bytes | 6.288375361 | |
| | f4 | flagRSTR | 3.526688669 | |
| | f1 | duration | 1.33198478 | |
| | f3 | servicewhois | 0.790131328 | |
| | f28 | srv_rerror_rate | 0.23681525 | |
| | f3 | servicehttp | | -0.824674256 |
| | f3 | serviceftp_data | | -2.528426436 |
| | f12 | logged_in | | -2.704620781 |
| R2L | f3 | servicer2l4 | 6.924775328 | |
| | f22 | is_guest_login | 6.864848127 | |
| | f11 | num_failed_logins | 5.087381233 | |
| | f3 | serviceftp_data | 4.979039069 | |
| | f3 | serviceauth | 3.504325369 | |
| | f3 | flagRSTO | 3.307197817 | |
| | f15 | su_attempted | 2.651112950 | |
| | f3 | servicetelnet | 1.941753603 | |
| | f3 | serviceftp | 1.328600083 | |
| | f34 | dst_host_same_srv_rate | 1.165825106 | |
| | f32 | dst_host_count | 0.14562231 | |
| | f10 | hot | 0.073086650 | |
| U2R | f14 | root_shell | 4.82944013 | |
| | f16 | num_root | 2.356016890 | |
| | f17 | num_file_creations | 1.6585901 | |
| | f19 | num_access_files | 1.2985147 | |

and the number of data bytes in one connection sent by the attacker will be large. Hence, *src_bytes* and *duration* are among the contributing features for such attack. Other contributing features for *probe* attacks include: remote job (*serviceremote_job*), remote job entry (*servicerje*) etc.

On the other hand, *R2L* and *U2R* attacks are detected by looking more on the content of the connection. Because these attacks are embedded in the data portions of the packets and normally involve only a single connection. The content features as shown in Table VII.1 indicate whether the data contents suggest suspicious behavior or not, e.g., number of failed logins, if successfully logged in or not, whether a *su* command is attempted and succeeded, number of access to access control files (e.g., "*/etc/passwd*") etc. Some of the contributing features of *R2L* attacks are *servicer2l4*, *is_guest_login*, *su_attempted* etc. Another most contributing feature to *R2L* attacks is *serviceftp_data*. This is because if a large amount of traffic is sent from a source as compared to a destination during an FTP session, then either *warezmaster*, *warezclient* or *ftp_write* exploits, that are associated when an FTP server has mistakenly given write permission to guest users, where they could create hidden directories on the system, can be launched. Normally, guest users are never allowed write permissions on an FTP server. Another type of *R2L* attack contributed by the *num_failed_logins* feature is the *guess_passwd* attack where the attacker is trying to login to a machine where s/he is not authorized to use by guessing login information of a system. If a *root_shell* is obtained, then the telnet service (*servicetelnet*) connection can allow a remote attacker to launch a *buffer_overflow* attack, i.e., *U2R*. A *U2R* attack usually takes place when an attacker logs in as an administrator followed by creating a number of files and making a lot of changes to the access control files. The *num_root* is another most contributing feature for a *U2R* attack that provides the number of *root* accesses in a connection.

From a networking perspective, there is a strong correlation among some actual features. This means, one feature may be discarded due to redundancy in the presence of another relevant feature with which it is strongly correlated [16]. For example, let us consider features *f13* and *f16*. It is sufficient for a classification method to either resort to *f13* or *f16*. At the presence of two correlated features, *LASSO* [44, 45] tends to arbitrarily pick only one and drops the less relevant by reducing the $\beta$ coefficient to zero. However, which correlated feature is selected or dropped is "*random*" in LASSO that produces a highly unstable behavior. This is because it is not clear at the end as to which choice of the correlated feature $\beta$ values will highly contribute to the regression. This has led to other solution approaches and the problem of correlated features in *LASSO* penalty can be avoided by using either the *group LASSO* [49] designed to do feature selection at a group level which forces the model to assign a similar weight to correlated features [46] or $\ell_2$ (*ridge regression*) penalty technique [18] that push correlated features to the same value in the model by shrinking the larger coefficients. When there are correlated features, unlike LASSO, *ridge regression* has the tendency to share the coefficient value among the group of correlated predictors [35]. Taking model stability and interpretability into account, the *LASSO* penalty problem can also be avoided by a better regularization technique.

This technique is a combination of $\ell_1$ and $\ell_2$ penalties called *Elastic Net* [50] that pushes the values of the correlated features together and make them move in and out of the model as a group. The *Elastic Net* regularization penalty, similar to *LASSO*, does an automatic feature selection and continuous shrinkage, and it can select multiple features that are correlated as it is thoroughly discussed in [50]. The authors in [46] have shown that several state-of-the-art *ML* classification methods can generate randomly misleading feature rankings when the training datasets contain large groups of correlated features.

It is worth mentioning that *LASSO* operates with the implicit assumption that the features are independent which makes the analysis possible and maintains the computational simplicity. Such assumption is common in *ML* and is the basis for *Bayesian* classifier, which is known to be an optimal classifier when the features are independent. However, even in the presence of feature dependence, classification experimental results have shown that *Bayesian* classifiers yield accurate results [9]. Bayesian classifiers also give good performance in practice compared to the state-of-the-art supervised classification methods in *ML* such as Decision tree classifiers, k-Nearest Neighbor (kNN), etc. [12]. During the attack classification, we assume the notion of a *Bayesian* model where features are independent and we focus on the higher probability of the coefficient values for both binary and multiclass classification. As seen in Tables VII.4 and VII.7, the distribution of the features among the attack classes is based on the large $\beta$ coefficient values of the features.

## VII.7 Experiments and Results

In this paper, we set out to experimentally examine classifying security attack analysis using *SVMs* and $\ell_1$-regularized method with *LASSO*. To evaluate the prediction accuracy of the competing techniques and potential benefit of feature selection for IDS, we have performed an experimental evaluation using the *R* language on an NVIDIA Tesla K80 GPU accelerator computing with the following characteristics: Intel(R) Xeon(R) CPU E5-2670 v3 @2.30GHz, 48 CPU processors, 128GB RAM, 12 CPU cores running under Linux 64-bit. The execution time (*hh:mm:ss*) of *SVM* and *LASSO* classification is `05:49:14` and `02:27:04` respectively. The elapsed time for *SVM* is obviously longer. Table VII.8 shows the confusion matrix and performance metrics results for our approaches described in the earlier sections on a 10-fold cross-validation experiment of the complete *NSL-KDD* public dataset. We have ranked the actual input features into strongly contributing, low contributory and irrelevant using a combination of feature selection filters and wrapper methods. We have carefully performed our experiments by carrying out comparisons with previous approaches. To that end, our results showed that selecting the most important actual features has improved the overall performance of both methods. As it is shown in Table VII.8, the test sets used for the computing methods (i.e., *SVM* and *LASSO*) are of the same size. Our evaluation results show that *SVM* and *LASSO* classifier accuracies of all the 41 features including strong and low contributing features are 79% and 83% respectively. A total accuracy of 97% for binary classification is achieved for both *two-stage* evaluation approach using *SVM* and *one-stage* approach using *LASSO*. Compared to binary, a total multiclass classification accuracy of 95.90% is achieved for *DoS*, *U2R*, *R2L* and *probing* attack categories using *LASSO* at a lower computational cost. To validate our evaluation of the classification models, in addition to *accuracy*, we have assessed multiple performance validation metrics such as *precision*, *recall*, *specificity*, *F1-Score*, and *AUC*[11].

[11] Area under the curve

Table VII.8: Classification Performance Metrics.

(a) Binary: Confusion Matrix of SVM

| Prediction | Actual | |
|---|---|---|
| | Anomaly | Normal |
| Anomaly | 33551 | 441 |
| Normal | 966 | 40618 |
| Totals | 34517 | 41059 |

(b) Confusion Matrix of LASSO

| Prediction | Actual | |
|---|---|---|
| | Anomaly | Normal |
| Anomaly | 33699 | 1386 |
| Normal | 818 | 39673 |
| Totals | 34517 | 41059 |

(c) SVM Performance

| | |
|---|---|
| Precision | 98.08% |
| Recall | 97.20% |
| Specificity | 98.93% |
| F1-Score | 97.95% |
| AUC | 97.12% |
| Accuracy | 97.14% |

(d) LASSO Performance

| | |
|---|---|
| Precision | 99.95% |
| Recall | 100% |
| Specificity | 99.99% |
| F1-Score | 99.97% |
| AUC | 99.44% |
| Accuracy | 97.08% |

(e) Multiclass: Confusion Matrix of LASSO

| Prediction | Actual | | | | | |
|---|---|---|---|---|---|---|
| | DoS | Normal | Probe | R2L | U2R | Precision |
| DoS | 18048 | 202 | 249 | 71 | 27 | 97% |
| Normal | 62 | 26605 | 149 | 394 | 102 | 97% |
| Probe | 260 | 130 | 4264 | 34 | 16 | 90% |
| R2L | 48 | 63 | 41 | 204 | 53 | 50% |
| U2R | 57 | 109 | 15 | 25 | 195 | 49% |
| **Recall** | 97.7% | 98% | 90.4% | 28% | 50% | |
| **F1-Score** | 97.37% | 97.77% | 90.51% | 35.88% | 49.12% | |
| **Specificity** | 98.65% | 97.83% | 99% | 98.94% | 99.60% | |

## VII.8 Conclusion and future directions

The process of defining appropriate input features, performing feature selection, data normalization and the contribution of this with interpretable results on security attack classification and computational performance have not been very well-studied. In this paper, we introduce the use of regularized *ML* techniques so as to enhance computer network security attack analysis. In our work, an effective feature selection analysis for IDS is performed on the benchmarking *NSL-KDD* public intrusion detection dataset. We focused mainly on the contribution of the actual input features that are well understood within the networking community to find what kinds of attacks in a network are the most significant. To that end, we have ranked the actual input features into strongly contributing, low contributory and irrelevant using a combination of feature selection filters and wrapper methods by carefully carrying out comparisons with previous techniques.

We have examined *LASSO* for robust regression both to binary and multiclass security attack classification to give us an insight into features of different classes of security attacks. Extensive simulation results are performed where we compared feature ranking algorithms using both *two-stage* approaches with *SVM* and *one-stage* approach using *LASSO*. We adopted two well-known ranking distance measure metrics in the evaluation of how similar our ranking algorithms are in relative to other state-of-the-art methods. Using the same test set for the competing methods, we found that *LASSO* gives comparable results with *SVM*. However, *LASSO* is much more computationally effective. *LASSO* provides coefficients that contribute how individual features affect the probability of specific security attack classes to occur. Moreover, *LASSO* allows us to explore more in terms of feature contribution for security attack analysis and it gives a better insight into why specific features have been selected. We concluded that *one-stage* approach using *LASSO* is simpler, computationally faster and gives us good performance with the most significant actual features. Finally, we provide deeper insight from a security engineering perspective on why the features obtained by regularized *ML* techniques are so important in clearly identifying various security attacks. We believe that the methodology presented in this paper may strengthen a future research in network intrusion detection settings.

In this work, our experimental study focuses particularly on the four attack classes namely *DoS*, *U2R*, *R2L* and *probe* used in the labeled public dataset. Even though the *NSL-KDD* dataset we use in our analysis may not be an ideal representative of existing realistic networks, it does not suffer from any of the mentioned limitations in the other old public datasets. Because of the lack of public intrusion detection datasets in the computer networking research community, we believe it can be applied as an effective benchmark dataset for the general problem of security analysis to help networking researchers work with different *ML* techniques to perform intrusion detection. Generating a similarly labeled intrusion dataset of real-time network traffic with different classes of attack distribution as in *NSL-KDD* dataset is costly and challenging. Therefore, as part of our future work, we would like to deeply investigate this problem and validate our findings using more recent realistic network traffic.

## VII.9 References

[1] J.-P. Benzécri. *Correspondence analysis handbook.* Marcel Dekker, 1992.

[2] A. L. Blum and P. Langley. Selection of relevant features and examples in machine learning. *Artificial intelligence*, 97(1):245–271, 1997.

[3] B. E. Boser, I. M. Guyon, and V. N. Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on Computational learning theory*, pages 144–152. ACM, 1992.

[4] K. Cup. KDD Cup 1999 data. http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html, 1999.

[5] D. E. Denning. An intrusion-detection model. *IEEE Transactions on software engineering*, (2):222–232, 1987.

[6] O. Depren, M. Topallar, E. Anarim, and M. K. Ciliz. An intelligent intrusion detection system (IDS) for anomaly and misuse detection in computer networks. *Expert systems with Applications*, 29(4):713–722, 2005.

[7] P. Diaconis and R. L. Graham. Spearman's footrule as a measure of disarray. *Journal of the Royal Statistical Society: Series B (Methodological)*, 39(2):262–268, 1977.

[8] C. Ding and H. Peng. Minimum redundancy feature selection from microarray gene expression data. *Journal of bioinformatics and computational biology*, 3(02):185–205, 2005.

[9] P. Domingos and M. Pazzani. On the optimality of the simple Bayesian classifier under zero-one loss. *Machine learning*, 29(2-3):103–130, 1997.

[10] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern classification.* John Wiley & Sons, 2012.

[11] C. Elkan. Results of the KDD'99 classifier learning contest. In *Sponsored by the International Conference on Knowledge Discovery in Databases*, 1999.

[12] N. Friedman, D. Geiger, and M. Goldszmidt. Bayesian network classifiers. *Machine learning*, 29(2-3):131–163, 1997.

[13] S. Garfinkel, G. Spafford, and A. Schwartz. *Practical UNIX and Internet security.* " O'Reilly Media, Inc.", 2003.

[14] A. K. Ghosh, J. Wanken, and F. Charron. Detecting anomalous and unknown intrusions against programs. In *Proceedings 14th Annual Computer Security Applications Conference (Cat. No. 98EX217)*, pages 259–267. IEEE, 1998.

[15] S. R. Gunn et al. Support vector machines for classification and regression. *ISIS technical report*, 14(1):5–16, 1998.

[16] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *Journal of machine learning research*, 3(Mar):1157–1182, 2003.

[17] I. Guyon, J. Weston, S. Barnhill, and V. Vapnik. Gene selection for cancer classification using support vector machines. *Machine learning*, 46(1-3):389–422, 2002.

[18] A. E. Hoerl and R. W. Kennard. Ridge regression: Biased estimation for nonorthogonal problems. *Technometrics*, 12(1):55–67, 1970.

[19] F. Iglesias and T. Zseby. Analysis of network traffic features for anomaly detection. *Machine Learning*, 101(1-3):59–84, 2015.

[20] K. Ilgun, R. A. Kemmerer, and P. A. Porras. State transition analysis: A rule-based intrusion detection approach. *IEEE transactions on software engineering*, (3):181–199, 1995.

[21] ISCX. NSL-KDD Dataset. http://www.unb.ca/research/iscx/dataset/iscx-NSL-KDD-dataset.html, 2009.

[22] R. Kohavi and G. H. John. Wrappers for feature subset selection. *Artificial intelligence*, 97(1-2):273–324, 1997.

[23] T. Kohonen. Learning vector quantization. In *Self-Organizing Maps*, pages 175–189. Springer, 1995.

[24] D. Koller and M. Sahami. Toward optimal feature selection. 1996.

[25] R. Kumar and S. Vassilvitskii. Generalized distances between rankings. In *Proceedings of the 19th international conference on World wide web*, pages 571–580. ACM, 2010.

[26] C. E. Landwehr, A. R. Bull, J. P. McDermott, and W. S. Choi. A taxonomy of computer program security flaws. *ACM Computing Surveys (CSUR)*, 26(3):211–254, 1994.

[27] T. Lane and C. E. Brodley. Temporal sequence learning and data reduction for anomaly detection. *ACM Transactions on Information and System Security (TISSEC)*, 2(3):295–331, 1999.

[28] W. Lee, S. J. Stolfo, and K. W. Mok. Mining in a data-flow environment: Experience in network intrusion detection. In *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 1999.

[29] I. Levin. KDD-99 classifier learning contest: LLSoft's results overview. *SIGKDD explorations*, 1(2):67–75, 2000.

[30] R. Lippmann, J. W. Haines, D. J. Fried, J. Korba, and K. Das. The 1999 darpa off-line intrusion detection evaluation. *Computer networks*, 34(4):579–595, 2000.

[31] J. McHugh. Testing intrusion detection systems: a critique of the 1998 and 1999 darpa intrusion detection system evaluations as performed by lincoln laboratory. *ACM Transactions on Information and System Security (TISSEC)*, 3(4):262–294, 2000.

[32] S. Mukherjee and V. Vapnik. Support vector method for multivariate density estimation. *MIT. CBCL*, 1999.

[33] R. Nelsen. Kendall tau metric. *Springer*, 2001.

[34] A. Y. Ng. Feature selection, l1 vs. l2 regularization, and rotational invariance. In *Proceedings of the twenty-first international conference on Machine learning*, page 78. ACM, 2004.

[35] A. B. Owen. A robust hybrid of lasso and ridge regression. *Contemporary Mathematics*, 443(7):59–72, 2007.

[36] V. Paxson. Bro: a system for detecting network intruders in real-time. *Computer networks*, 31(23-24):2435–2463, 1999.

[37] M. Robnik-Šikonja and I. Kononenko. Theoretical and empirical analysis of ReliefF and RReliefF. *Machine learning*, 53(1-2):23–69, 2003.

[38] M. Sabhnani and G. Serpen. Why machine learning algorithms fail in misuse detection on KDD intrusion detection data set. *Intelligent Data Analysis*, 8(4):403–415, 2004.

[39] Y. Saeys, I. Inza, and P. Larrañaga. A review of feature selection techniques in bioinformatics. *bioinformatics*, 23(19):2507–2517, 2007.

[40] M. M. Sebring. Expert systems in intrusion detection: A case study. In *Proc. 11th National Computer Security Conference, Baltimore, Maryland, Oct. 1988*, pages 74–81, 1988.

[41] S.Mukherjee et al. Support vector machine classification of microarray data. *Artificial Intelligence Memo, MIT*, 1999.

[42] C. Spearman. The proof and measurement of association between two things. *The American journal of psychology*, 1904.

[43] M. Tavallaee, E. Bagheri, W. Lu, and A.-A. Ghorbani. A detailed analysis of the kdd cup 99 data set. In *Proceedings of the Second IEEE Symposium on Computational Intelligence for Security and Defence Applications*, 2009.

[44] R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1):267–288, 1996.

[45] R. Tibshirani. Regression shrinkage and selection via the lasso: a retrospective. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 73(3):273–282, 2011.

[46] L. Tolosi and T. Lengauer. Classification with correlated features: unreliability of feature ranking and solutions. *Bioinformatics*, 27(14):1986–1994, 2011.

[47] V. G. Tusher, R. Tibshirani, and G. Chu. Significance analysis of microarrays applied to the ionizing radiation response. *Proceedings of the National Academy of Sciences*, 98(9):5116–5121, 2001.

[48] V. N. Vapnik and V. Vapnik. *Statistical learning theory*, volume 1. Wiley New York, 1998.

[49] M. Yuan and Y. Lin. Model selection and estimation in regression with grouped variables. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 68(1):49–67, 2006.

[50] H. Zou and T. Hastie. Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 67(2):301–320, 2005.

Paper VIII

# Advanced Passive Operating System Fingerprinting Using Machine Learning and Deep Learning

**Desta Haileselassie Hagos[1], Martin Løland, Anis Yazidi, Øivind Kure, Paal E. Engelstad.**

***Submitted for review*.**

VIII

## Abstract

Securing and managing large, complex enterprise network infrastructure requires capturing and analyzing network traffic traces in real-time. An accurate passive Operating System (OS) fingerprinting plays a critical role in effective network management and cybersecurity protection. Passive fingerprinting doesn't send probes that introduce extra load to the network and hence it has a clear advantage over active fingerprinting since it also reduces the risk of triggering false alarms. This paper proposes and evaluates an advanced classification approach to passive OS fingerprinting by leveraging *state-of-the-art* classical machine learning and deep learning techniques. Our controlled experiments on benchmark data, emulated and realistic traffic is performed using two approaches. Through an Oracle-based machine learning approach, we found that the underlying TCP variant is an important feature for predicting the remote OS. Based on this observation, we develop a sophisticated tool for OS fingerprinting that first predicts the TCP flavor using passive traffic traces and then uses this prediction as an input feature for another machine learning algorithm for predicting the remote OS from passive measurements. This paper takes the passive fingerprinting problem one step further by introducing the underlying predicted TCP variant as a distinguishing feature. In terms of accuracy, we empirically demonstrate that accurately predicting the TCP variant has the potential to boost the evaluation performance from 84% to 94% on average across all our validation scenarios and across different

[1]University of Oslo, Department of Informatics, destahh@ifi.uio.no

**Keywords**: Operating System, Fingerprinting, Machine Learning, Deep Learning, Passive Measurements

types of traffic sources. We also demonstrate a practical example of this potential, by increasing the performance to 91.3% on average using a tool for TCP variant prediction in an emulated setting. To the best of our knowledge, this is the first study that explores the potential for using the knowledge of the TCP variant to significantly boost the accuracy of passive OS fingerprinting.

## VIII.1 Introduction and Motivation

As modern network infrastructures grow in size, collecting detailed relevant knowledge about the dynamic characteristics and complexity of large heterogeneous networks is crucial for many purposes e.g., network vulnerability assessment and monitoring, Spam detection, etc. Developing advanced network security and monitoring techniques are important for both the research and security practitioners. There has been a significant research work in the context of network management and cybersecurity on developing network security tools to fingerprint remote Operating Systems (OSes) [26, 27, 28, 44, 45]. OS fingerprinting is the process of inferring the OS of a machine operating with TCP/IP by a remote device connected on the Internet without having physical access to the device [20]. There are many different custom tools for fingerprinting of the most commonly used OSes based on the characteristics of its underlying TCP/IP network stack [20] and this, to a large extent, is due to variability in how the TCP/IP stack is traditionally implemented across different OSes [25]. One common approach, for example, is by collecting the TCP/IP stack basic parameters [23], e.g., IP initial Time To Live (TTL) default values [5], HTTP packets using the User-agent field [22], Internet Control Message Protocol (ICMP) requests [31], known open port patterns, TCP window size [18], TCP Maximum Segment Size (MSS) [33], IP Don't Fragment (DF) flag [32], a set of other specific TCP options to mention a few. However, in our work, we want to take this one step further by combining these basic features and other settings with the underlying TCP variant as a feature in our model due to the fact that different OSes are doing slightly different implementations of TCP. Some implementations of common TCP variants quickly overshoot the size of the Congestion Window (cwnd) because of differences in the variant implementations. Hence, we believe that knowing the implementation of the underlying OS may help us understand better their exact behavior. It can also help us explore how to classify an OS when different OSes are implementing the same TCP variant.

**Fingerprinting Techniques**: We can determine what OS a remote computer on the Internet is running by either passively listening to traffic captured from a network or by actively sending it packets. The most widely used complementary remote OS fingerprinting proven approaches that employ a variety of TCP/IP stack scanning are broadly categorized into classes of *active* and *passive* methods.

- ***Active Fingerprinting***: This technique is based on actively transmitting one or more specially crafted network packets with different packet settings or flags to a remote network device in order to analyze the corresponding potentially identifying replies [26, 44]. This method determines knowledge of the underlying OS according to the received responses from the target device by examining the network behavior of known TCP/IP stack [37]. However, since this approach injects additional traffic to the network by generating active probes, it may itself trigger alarms and get blocked by firewall rules and Network address translators (NATs) [9].

- ***Passive Fingerprinting***: This approach, on the other hand, inspects and analyzes packets traveling between end hosts without injecting any traffic into the network [27, 28, 45]. This technique with little resource simply analyzes a pattern of the OS-specific information that has already been sent in the network traffic and compares for a match with a predefined database that contains a list of known signatures of different OSes. Passive fingerprinting doesn't send probes and hence it has a clear advantage over active fingerprinting since it reduces the risk of triggering alarms [9].

OS fingerprinting can also be performed using classical techniques known as "*banner grabbing*". It is an approach used to gain detailed information about a remote computer system on a network and the associated services running on its opened ports [35]. Using techniques like this, some remote computers announce their underlying OS freely and running application services with their versions in use to anyone connecting to them as part of welcome banners or header information. Some of the widely used services that serve *banner grabbing* are: *Telnet*, *FTP*, *NetCat*, *SMTP*, etc. However, it is useful to remember that some of these basic services are effective against less secure networks.

**Potential benefits and applications**: Network scanning and accurate remote OS fingerprinting are the crucial steps for penetration testing in terms of security and privacy protection. Note that attackers can also embrace passive fingerprinting techniques to search for potential victims in a network. For example, by identifying the OS running on a remote computer and the list of services it runs, an attacker can target the device to eavesdrop on the communication between the endpoints without having physical access to the device. However, we argue that our work presented here is motivated by a number of practical applications that can be positively used by network and system administrators. Passively fingerprinting an OS by analyzing the packets it generates and transmits over a network is extremely important in the areas of network management and computer security for several reasons. For example, it is useful to explore a network for potential exploitations of security vulnerabilities which can be exploited by attackers, auditing, identify critical attacks, reveal new information about a network user etc. Network administrators can, therefore, use this OS related information to maintain the security policy and reliability of their network by configuring a network-based Intrusion Detection Systems (IDS) [24]. Vulnerabilities and security threats in a network may result from rogue or unauthorized devices [40], unsecured internal nodes within the network and from external nodes [4]. Hence, passively fingerprinting an OS has a potential benefit in addressing these critical problems. This, from an academic point of view, is interesting and something that needs to be addressed from a network security research point of view.

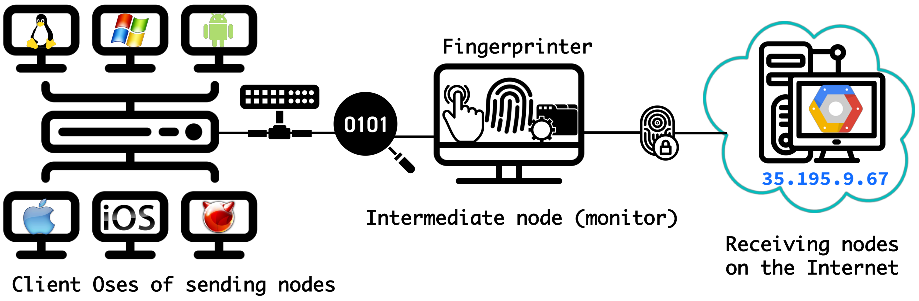

Fingerprinter
0101
35.195.9.67
Intermediate node (monitor)
Receiving nodes on the Internet
Client Oses of sending nodes

Figure VIII.1: Network architecture for passive OS fingerprinting by an intermediate node.

**Limitations of previous works**: Traditionally, most of the existing general OS fingerprinting techniques resort to manually generated signature matching from a database of heuristics which contains features of widely used OSes. This means, after comparing the generated signatures, the first set of responses match with the highest confidence against a database of fingerprints would be used to select the specific probable OS. However, manually updating a large number of signature and managing databases of new OSes adds a considerable amount of time and hence we may suffer from the consequences of the lack of recent signature updates of the known OSes. For example as reported in [22], the last updates of the fingerprint databases of *Ettercap* [28] and *p0f* [45] date to 2011 and 2014 respectively. Consequently, new OSes families like Android 4.4 and higher versions of Android, Windows 10 distributions, etc. will not be recognized by these tools since they are not included in their fingerprint databases. Hence, we argue that it is important to consider making use of a fingerprint database that contains variations of most currently used OSes and automating these tasks by employing learning algorithms capable of extracting all possible OS-specific features for discovering the underlying OSes. To explore this idea of applying learning algorithms, we present a unified and robust classification approach to an advanced passive OS fingerprinting that leverages both machine learning and deep learning methods. Our fingerprinting technique is completely passive meaning that we only need to be able to observe network traffic from a target machine at any observation point on the network without injecting any traffic into the network. Note that the TCP/IP header fields would not be impacted by SSL/TLS encryption of the TCP payload. Hence, since we utilize features that are readable even with encryption, our approach is independent of whether the flow is encrypted or not. Figure VIII.1 shows the architecture for implementing our fingerprinting methodology.

**Why machine learning approaches to OS fingerprinting**? There are several limitations imposed by classical fingerprinting techniques. Passive OS fingerprinting generally relies on recognizing the default values for various TCP/IP stack parameters. If a user changes these parameters, the task of OS fingerprinting becomes much more challenging. Most of the existing works on fingerprinting provide little capability to address this challenge. Motivated by this problem, we proposed a novel approach by leveraging both machine learning and deep learning-based techniques that consider the set of parameters as a whole, rather than individually so that our model caters for variations in TCP parameters. If a user changes the initial receive window size, for instance, we may still be able to recognize the OS from other parameters that have not been changed (TCP congestion control algorithm, initial cwnd size, etc.). Note that this depends entirely on the changes made by the user to the default TCP or OS stack parameters that are commonly used for signature based fingerprinting. The other reason why we create a model by employing learning techniques is to understand the complex patterns of the varying values in the TCP header and extract useful input features. Because machine learning offers new possibilities as it can extract pattern and general rules for classification. Machine learning can also be more robust to small variations in the input parameters. In addition to this, with the use of learning techniques, we argue that avoiding using manually updated static signature databases has two potential benefits. Firstly there is no tedious task of creating these unique fingerprints, all you need is a set of values or features. The second benefit comes from a known flaw in many of the existing fingerprinting tools, where a "first-match" policy is applied, meaning that if two fingerprints are equal the tool would always predict the first OS with that exact fingerprint. However, learning techniques, on the other hand, make calculated guesses of which of the classes with the same fingerprint that will be predicted.

**Contributions**: We summarize our main contributions below.

- We propose and evaluate a robust approach to OS fingerprinting from passive measurements by leveraging machine learning and deep learning techniques.
- We investigate the use of TCP congestion control variant as a distinguishing feature in passive OS fingerprinting.
- We explore variability in implementations of TCP variant by different OSes and its effect on classifying remote OS.
- We study the applicability of Recurrent Neural Networks (RNN)-based models for robust and advanced passive OS fingerprinting by combining the basic TCP/IP features and the predicted TCP variant as input vectors.
- We show that the TCP flavor has a great potential for boosting passive OS fingerprinting.

- We show that the learned OS fingerprinter model performs reasonably well by leveraging a trained knowledge from the emulated network when it is applied and transferred on a realistic network.
- We build a universal tool for passive monitoring that can be applied to first estimate the TCP cwnd, second predict the underlying TCP flavor and finally uses the TCP variant as an input feature to detect the remote computer's OS.

**Roadmap**: The rest of the paper is organized as follows. Section VIII.2 discusses related work, and Section VIII.3 presents the experimental datasets. Section VIII.4 presents the machine learning of the OS fingerprinter. The machine learning of the TCP variant prediction tool is presented in detail in Section VIII.5. Section VIII.6 presents the experimental results without a known TCP variant which will play the role of baseline. In order to assess the importance of knowing the TCP variant, experimental results of all the use cases with an Oracle-given TCP variant are presented in Section VIII.7. Section VIII.8 presents the experimental results with the predicted TCP variant. Section VIII.9 presents the transfer learning results. Finally, Section VIII.10 concludes our paper and suggests directions for future research work.

## VIII.2 Related Work

Remote OSes fingerprinting has a long history in the computer security community [2, 22, 23, 26]. TCP/IP header fingerprinting and any information related to application protocols are used to identify the underlying OS running on a remote host either actively or passively [25]. As we explained in Section VIII.1, there are multiple existing tools for both the predominant active and passive OS fingerprinting approaches, where *Nmap* [26] is one of the most prominent open-source active fingerprinting tools. The work presented in [38], *SYNSCAN*, works in a similar fashion to *Nmap*, but it performs the fingerprinting task by actively sending a small number of crafted network packets to a single TCP port. *Xprobe2* [44] is another popular fingerprinting tool, that relies primarily on ICMP packets, and it depends on how many changes we make to the default TCP/IP stack parameters. Since *Xprobe2* does fuzzy fingerprinting with a signature matching algorithm as an alternative to *Nmap*, it means that if we make a lot of changes to the default TCP/IP stack parameters, the underlying OS will not be detected. However, *Xprobe2* is more robust to small fingerprint variations as compared to *Nmap*. As explained above the other fingerprinting tools, *Ettercap* [28] and *p0f* [45], have not been updated since 2011 and 2014 respectively to include variations of most widely used modern OSes. For passive OS fingerprinting to be effective, we believe that the limitations of these fingerprinting tools need to be addressed. The work in [23] also demonstrates that the OS fingerprinting accuracy of the *Ettercap* and *p0f* signature databases is low and techniques to improve performance was proposed. Hence, the paper presents rule-based machine learning classifiers capable of identifying 75 classes of OSes from TCP/IP packet headers found in the *Ettercap* database. They proposed a classifier technique using k-nearest neighbors (KNN) that returns an approximate first match for an OS from a fingerprint database this counters the problem of classifying hosts as unknown if no exact match is found in the database [23]. However, their evaluation yielded poor experimental results, rejecting as much as 84% of the test packets, while 44% of the accepted patterns were wrongly classified [23]. The problems contributing to poor performance was believed to be caused by two main issues. The first reason is, substitution errors due to multiple OSes with exactly the same fingerprint feature values. The second reason for this poor performance is the high rejection rate caused by numerous unique feature values derived from the same OS. After combining the OS classes most often confused with each other, eliminating all the classes where the error could not be reduced by combining classes, the error percentage was reduced to 9.8% with no rejected packets. Beyond remote OS detection using TCP/IP network stacks, fingerprinting techniques have also been extended to be applied for remote device level fingerprinting [8].

A recent study that is most closely related to our work, and which has also given a comprehensive survey on passive fingerprinting methods, can be found in [22]. The authors have employed OS fingerprinting methods in the environment of wireless networks. Besides using the three basic TCP/IP stacks (i.e., TTL, window size and initial SYN packet size), the authors suggested also using the user-agent information in HTTP request headers and communication with OS-specific domains can be usable in large dynamic networks [22]. The average accuracy of OS classification using the TCP/IP parameters reported in [22] is 80.88%. Zhang et al.'s paper on OS detection [46] utilizes only one machine learning technique namely Support Vector Machine (SVM). However, the testing error rate of identifying some of the OSes e.g., Mac, Cisco, FreeBSD, and OpenBSD is 25.80%, 24.22%, 17.71%, and 15.85% respectively [46]. Aksoy et al. [2] have employed genetic algorithms for identifying packet features suitable for OS classification based on the analysis of the network TCP/IP packets using machine learning algorithms. However, most of these previous works use the basic actual TCP/IP features for evaluating passive OS fingerprinting. Besides, we believe that these tools have the inability to extract all possible OS-specific features for passively fingerprinting the underlying OSes. In contrast, what separates our contribution in this paper from the other previous related works is that out model supports a wider range of TCP/IP network stack features. As shown in Figure VIII.2, the main goal of our work presented here is to combine these basic TCP/IP features that are the basis of OS fingerprinting with the underlying TCP variant by leveraging both machine learning and deep learning techniques. This contribution remains largely unexplored and is not used by existing fingerprinting techniques. Detecting the implementation of a TCP variant passively is a challenging task and this, we believe, is the reason why no previous works use it to passively fingerprint remote OSes. However, in our case, we already have a general solution for this difficulty presented in our previous works [12, 13, 14]. The reason why we focus on the implementations of the underlying TCP variant as a feature in our OS classifier model is due to the fact that different OSes are doing slightly different implementations of TCP. Hence, we believe that passively observing the network-level characteristics found in TCP packets can give us more information about the remote computer's underlying OS. We further believe that this will also help us to explore in detail the long-term characteristics of TCP traffic. To the best of our knowledge, this is the first study of passive fingerprinting OSes by applying RNN methods combining the basic TCP/IP features and the underlying TCP variant as input vectors.

## VIII.3 Experimental Datasets

Our machine learning models for OS classification is developed and tested on three datasets, presented below.

### VIII.3.1 Benchmark Data

First, we utilize a large benchmark dataset that has been used for OS fingerprinting in a previous related work [22]. This dataset is closely aligned with our task, and it was collected from a university wireless network. The benchmark dataset was used in the previous work for OS fingerprinting based on the HTTP header, while the ambition of our paper is to do generic fingerprinting based only on the TCP packet fields. Since we aim at fingerprinting that is not application-specific, the TCP information in the dataset is useful for our purpose, while the HTTP User-agent information in our experiments is used only to establish ground truth about the OS that was used.

Table VIII.1: Statistics of the OSes and their market shares.

| Android | Windows | Mac OS | Linux | iOS | Unix | Other |
|---|---|---|---|---|---|---|
| 8.0 | 10 | Mojave | Ubuntu 16.04 | 12.1 | Solaris 11.4 | Unknown |
| 8.1 | 7 | High Sierra | Ubuntu 18.04 | 11.4 | FreeBSD 11.2 | |
| 6.0 | 8.1 | Sierra | Ubuntu 18.10 | 12.0 | | |
| 7.0 | 8.0 | El Capitan | Fedora 29 | 10.3 | | |
| 7.1 | XP | Yosemite | Debian 9 | 9.3 | | |
| 5.1 | Vista | Mavericks | CentOS 7.6 | 11.2 | | |
| | | | openSUSE 42.3 | | | |
| 36.5% | 35.99% | 6.37% | 0.79% | 13.99% | 1.58% | 4.78% |

The benchmark dataset contains 79087345 flows, activity of 21746 unique users, 253374 WiFi sessions, 25642 unique MAC addresses, and 6104 unique IP addresses, a fingerprint database of 2078 standard TCP/IP signatures of 51 known unique OSes with a total of 529 variations when considering major and minor versions [22]. The dataset consists of three basic TCP/IP network stack features, i.e., initial SYN packet size, TTL, and TCP window size [22]. After our first set of testing, we realized that the data was severely skewed and that only a few of the classes contained almost all of the entries, giving us artificially good classification results. We then removed most of the very seldom occurring classes and ended up with 33 reduced classes. We also removed all traffic that did not contain HTTP User-agent information, since we could not establish ground truth for this traffic. In addition, we created a new dataset where all the classes were bucketed into seven groups, consisting of the six most widely used major OS families: Android, Linux, Mac OS, Unix, Windows, iOS, and a seventh class called "Other" for OSes not suited for any of the other groups as shown in Table VIII.1. Finally, we ended up distributing all of the labels equally so that each OS class had the same number of occurrences. This is not necessarily the option that gives a model with the best classification accuracy, but it creates the most versatile model with balanced training data. This helps us improve the generalizability of our model with a unified approach that encompasses all variations of the most widely used OSes.

### VIII.3.2 Realistic Traffic

While benchmark traffic is useful to link our experiments to previous related work, we also wanted additional realistic traffic for which we have more control and that allows us to make our own assurances of the quality of the data. Thus, we passively collected our realistic dataset from TCP traffic originated from the internal network of the Oslo Metropolitan University and destined to various hosts on the Internet. First, we collected data for fixed (non-mobile) desktop computers (typically using OSes like Windows, Linux, Unix, Mac OSx, etc.) by using an intermediate node as shown in the network setup in Figure VIII.1. Then, we collected the data that covered mobile devices, like *android* and *iOS*. The latter was collected from the 5G 4IoT research lab [1, 36] of the Oslo Metropolitan University.

We spent a significant amount of effort in establishing ground truth, i.e., determining the actual OS that has been used for each traffic flow. To establish ground truth in the realistic dataset, we follow two approaches. The first approach was only applicable to the non-mobile desktops, while the second method was used for both mobile and non-mobile devices. With the first method, we leveraged the DHCP log messages associated with the non-mobile desktops to derive the ground truth from the DHCP server of the Oslo Metropolitan University network that logs the sessions by the MAC address and name of the device. Since we collect the real data from the internal network of our university, extracting the DHCP log messages can give us detailed information about the OSes. We could, for example, see information about the *vendor-specific prefixes* since most of the OS variants are identified based on their vendors. The list of device vendor prefixes is useful in revealing the specific implementation of an OS because most of the modern OSes from the same device vendor usually share the same OS kernel and similar network behaviors. For example, we found out that Apple products often share the same TCP/IP parameters. The second approach we used to identify the OS is getting the predefined browser strings that loosely tell the name of the underlying OS assigned by the vendor from Webserver. We believe changing the default device names by all users is not that common and sometimes discouraged by the vendors, e.g., Google and Apple OSes. However, the device name of Linux and Windows OSes could be changed easily by experienced users which would make passively identifying these devices hard. Since a number of computer vendors offer devices with a pre-installed OS and default device name and MAC address, we can use this information to derive the ground truth for OS fingerprinting. For example, Apple devices use a default string name of "<user>-iPhone", "<user>-iPad", Microsoft uses "Windows-Phone" for its mobile devices, and Android uses "android-<android_id>", etc.

Our real traffic covers the communication to and from our university and hence all traffic whose source and destination IP addresses are within the subnets of our internal network. Hence the network administrator of our university has full control over the internal machines with real IP addresses that are not going to a NAT gateway, and therefore it is fairly possible to tell whether it is a laptop or a desktop PC by looking it up in the internal database owned by the university. However, since it is a dynamic network we do not have full control over external machines, because they can be anything behind an IP address that changes dynamically. This is because there is an endless number of machines spoofing scanning the network and they can appear as Linux-powered OSes but they could be Windows and vice versa and this happens because the user may have strongly tuned the TCP stack to look like something else. It is pretty hard to certainly say anything about the external computers because the communication can go through a NAT gateway possessing another OS type. For example, if a user is connected to a student wireless network, there is a chance that it may go to a Linux NAT gateway and hence from outside the user is seen as Linux NAT which makes it hard to predict whether the underlying OS is Linux, Mac or Windows. Therefore, fingerprinting devices behind NAT technology on a distributed network where a number of devices can hide behind a NAT is another critical challenge. It is, therefore, worth noting that establishing ground truth in dynamic networks at a larger scale remains a challenging problem. Further investigation to explore these difficulties will be done in our future works. Finally, due to the privacy protection of possibly sensitive data, the payload of all the network packets collected was removed and anonymized with a prefix-preserving algorithm [7, 42]. Furthermore, we were only allowed to collect TCP headers of the traffic flows, while we could not collect complete traffic captures, due to privacy protection and legal reasons.

### VIII.3.3 Emulated Traffic

In a real scenario where the OS fingerprinting is going on continuously in an intermediate node of an enterprise or production network, the intermediate node will have more information available than only the TCP header, such as the traffic profile or the knowledge of congestion or the outstanding bytes-in-flight of a flow. In our experiments below, we show how this information can be very useful for OS fingerprinting. Since we do not have full traffic packet captures in our benchmark dataset or in our realistic dataset, we needed an additional dataset that we collected from an emulated network, where there would be no privacy protection or legal issues related to our dataset. The architecture of our emulated network is similar to the network setup shown in Figure VIII.1, except that all the nodes (the sender, the intermediate node and the receiver) are implemented in virtual machines. All background traffic of the OSes for our emulated scenario is generated using the *iperf* [6]. Establishing ground truth is straightforward, as we have full control of the OSes used when generating the traffic. In addition to establishing the ground truth, we also wanted to allow the intermediate node to establish a prediction of the TCP variant by monitoring the on-going traffic profile of the TCP flow between the sender and the receiver. As shown later in the paper, using definitive or predicted knowledge of the TCP variant as an additional input feature to the OS fingerprinting, might boost the fingerprinting accuracy significantly. How the machine learning model for prediction of the TCP variant in the emulated scenario is trained and how the TCP variant is subsequently predicted are presented in the following.

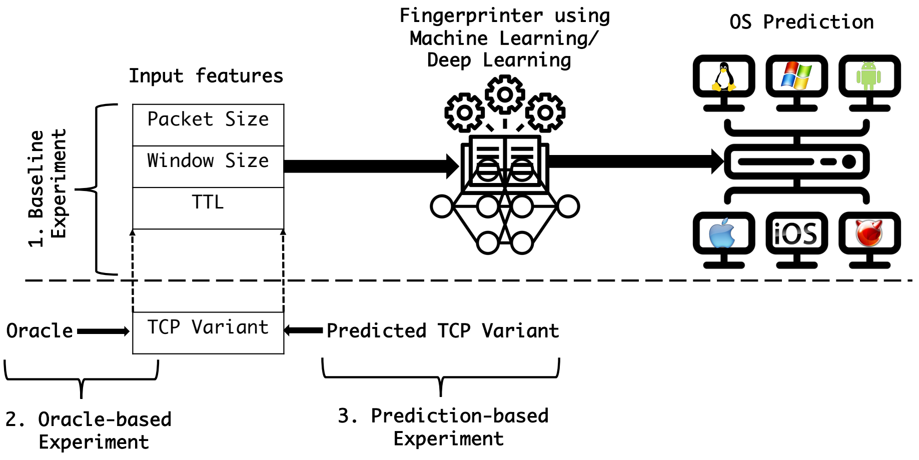


Figure VIII.2: The process implemented on the intermediate node for passive OS fingerprinting.

## VIII.4 Machine Learning of the OS Fingerprinter

### VIII.4.1 Classical Machine Learning Approaches

The OS fingerprinter takes various features as input parameters, and use machine learning to predict the OS as shown in Figure VIII.2. Many machine learning techniques could be used to implement a model for passive OS fingerprinting. In this paper, we have employed the following most commonly used classical machine learning methods suitable for our task. In order to train and test our classification models, we employed every experiment with a ratio of 60% training, 40% testing split and 5-fold cross-validation setting on all variations of the features into one learning model.

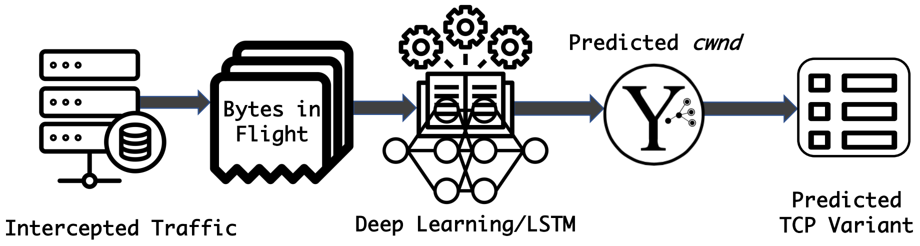


Figure VIII.3: The process implemented on the intermediate node for prediction of the TCP variant of the passively intercepted TCP traffic flow. An LSTM-based machine learning module predicts the cwnd from the outstanding bytes-in-flight. In the next step, the cwnd behavior is used to predict the TCP variant as explained in further detail in our previous works [12, 13, 14]. The predicted TCP variant is finally used as an input feature to the OS fingerprinting process (see bottom right part of Figure VIII.2).

**SVM**: In order to perform an efficient multi-class SVM classification through cross-validation, we tuned the SVM hyperparameters using a *GridSearchCV* that allows specifying only the ranges of values for optimal parameters by parallelization construction of the model fitting. Finally, in our evaluation, we found out that *SVM* with a Radial Basis Function (RBF) kernel for classification model yields a substantially better result.

**Random Forest (RF)**: We tuned the meta-estimator by varying the number of decision trees between 1 and 1000. We found out that increasing the number of trees more than 10 doesn't give much improvement in the classification accuracy.

**KNN**: We applied KNN by testing different values of $K$ ranging from 5 to 100 followed by a weight function for a total of 20 observations. The observations have been conducted in two ways. In the first experiment, we set the weight to *uniform*. In the second experiment, the points are weighted by the inverse of their distance, causing closer neighbors to have greater influence. Finally, we choose the model that has the highest accuracy for a given unseen instance.

### VIII.4.2 Deep Learning Approaches

To find the deeper characteristics of TCP variants implemented by respective OSes and exploit the extra OS-specific information, we apply the following two neural network architectures.

**Multilayer Perceptron (MLP)**: In our evaluation, MLP model with a single-layer feedforward neural network [16, 34] has been used to classify the different classes of OSes. After the hyperparameter tuning, we tested our MLP model with a different number of batch sizes, hidden layers and nodes (e.g., 0, 1, 2, 32, 64, 128) in each layer. Combining all of these, a total of 324 models were trained with and without the default TCP variant. We found out that the results for both with and without a known TCP variant were almost the same with an insignificant drop in the accuracy irrespective of which hyperparameters performed the best. Finally, 128 nodes of the network per dataset are trained for 150 epochs with a batch size of 500 by SGD with momentum of 0.9 and a constant learning rate of 0.01. However, we learned that SGD is sensitive in regards to the selection of the learning rate since it doesn't automatize the values and we also found that it suffers from premature convergence and is outperformed by *Adam*-based optimization methods. Hence, both *Adam* and *Nadam* gradient-based optimization algorithms fit for our purpose and that is because we wanted to use an optimization algorithm that adapts its learning rate dynamically in a way that doesn't affect the objective function and learning process of the model. Our experimental results show that the hyperparameter tuning baseline experiments by applying *tanh* as activation function and *Adam* optimization algorithm and training the model for 150 epochs, provides a substantial improvement in accuracy as compared to the other parameters.

**Long Short-Term Memory (LSTM) models**: We have explored an approach to classify the underlying OS from passive measurements using LSTM-based RNN architecture by combining the basic TCP/IP features and the underlying TCP variant shown in Table VIII.2 as input vectors. For more details about LSTM applied in the context of computer networks, we refer the reader to our previous paper [13]. We trained our LSTM model over 150 epochs of the training samples with a batch size of 32 as values in time-series. We propagate the input feature vector ($x$) to the model through a multilayer LSTM cell followed by a fully connected dense layer of 150 hidden nodes with Rectified Linear Unit (ReLU) activation function using the *hard_sigmoid* as recurrent activation for the different layers that generates an output of a sequence dimensional vector of predicted OSes ($y_t$).

We trained our LSTM-based learning algorithm without the knowledge of the input features from the true signatures of the OSes during the learning phase. We learn the model from the training data and then finally predict the test labels from the testing instances on all variations of the OS-specific parameters. In order to train our prediction model more quickly, and get a more stable and robust to changes OS classification model, we have applied one of the most effective optimization algorithms in the deep learning community, the *Adam* stochastic algorithm [19] with an initial *learning rate* of *0.001* and *exponential decay rates* of the first ($\beta_1$) and second ($\beta_2$) moments set to 0.9 and 0.999 respectively. We further optimize a wide range of important hyperparameters related to the neural network topology to improve the performance of our OS classification model.

### VIII.4.3 Experimental Hardware Setup

All our machine learning experiments are carried out using a cluster of HPC machines based upon the GNU/Linux operating system running a modified version of the *4.15.0-39-generic* kernel release. The prediction 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, 128 GB RAM, 12 CPU cores running under Linux 64-bit. All nodes in the cluster are connected to a low latency 56 Gbit/s Infiniband, gigabit Ethernet and have access to 600 TiB of BeeGFS parallel file system storage.

### VIII.4.4 Objectives of our Experiments

The aim of our experiments is to explore the effect of the TCP variant as an input feature when passively detecting the underlying OS. To investigate this, we divide our analysis into three different experiments. First, in the baseline experiment (Section VIII.6) we carry out the OS fingerprinting without using a known TCP variant as an input feature. This corresponds to the simplest state-of-the-art transport layer method, which is illustrated in the upper part of Figure VIII.2. Since there is a close connection between existing popular OSes and the TCP variants they use, our hypothesis was that the potential for improvement by using the TCP variant as an input feature would be significant. For example, CUBIC [10] is the default congestion control algorithm as part of the Linux kernel distribution configurations from version 2.6.19 onwards. Since Android devices are also Linux-powered, CUBIC remains to be the default TCP congestion control algorithm. Many Windows 7 distributions have been shipped with the default New Reno [15] and whereas Windows 8 families with CTCP [39]. Therefore, in the next Oracle-based experiment (Section VIII.7), we investigate the potential of knowing the TCP variant, and how much this knowledge might boost the fingerprinting accuracy. Here we assume that there is an Oracle that can identify and give the TCP variant used in the TCP flow that is fingerprinted. This is illustrated in the bottom left part of Figure VIII.2. However, in a real scenario, the intermediate node would not have access to definite knowledge of the TCP variant (e.g., given by an Oracle). Instead, the intermediate node might at best try to infer it from the monitored traffic. Thus, in the third prediction-based experiment (Section VIII.8), we first allow the intermediate node to predict the TCP variant passively. This is illustrated in the bottom right part of Figure VIII.2. The OS fingerprinter then uses that TCP variant prediction as an input feature to make the OS prediction illustrated in the upper part of Figure VIII.2. The TCP variant is predicted by analyzing the famous sawtooth pattern behavior of estimated cwnd of TCP, which is computed based on the outstanding bytes-in-flight [13, 14]. This is presented in more detail in the next section. Since the latter experiment requires TCP traffic details of outstanding bytes-in-flight, which is not available in our benchmark and realistic datasets, this experiment is only possible with our emulated dataset.

## VIII.5 Machine Learning of the TCP Variant Prediction Tool

The main goal of the experiments in the emulated network is to use the predicted TCP variant as an additional input feature to the OS fingerprinting. The TCP variant is predicted by the process illustrated in Figure VIII.3. As described in sufficient detail in our previous works [12, 13, 14], we used a database to match and join the intercepted TCP traffic on both the intermediate node and the sending node. The outstanding bytes-in-flight of the traffic (i.e., the number of bytes that have been sent but not yet acknowledged) is used as input to our machine learning model to predict the cwnd behaviour of the traffic. We use LSTM for the machine learning. We trained and verified the machine learning model by matching the predicted TCP states with the actual TCP kernel states directly logged from the Linux kernel. Since we have full control of the sending nodes, we can track the system-wide TCP state of every packet that is sent and received from the kernel to verify our model's prediction accuracy against the actual TCP variant by matching with the actual sending TCP states using the techniques presented in our previous works [12, 13, 14]. After the verification, we can run our learning model and get the cwnd predictions of the TCP stack in use.

Once we can estimate the cwnd of the sender, we can also infer the multiplicative back-off factor to decrease the cwnd on a loss event ($\beta$) which is an important feature for uniquely identifying the TCP variants. Finally, we combine the predicted TCP variant as the basis of OS fingerprinting with the basic TCP/IP features as shown in Figure VIII.2. Here, we consider only loss-based TCP congestion control algorithms, e.g., BIC [43], CUBIC [10], CTCP [39], Reno [17], and New Reno [15]. Our approach could also be useful to other TCP variants like Google's QUIC [21]. QUIC uses packet loss as an indicator of congestion and supports a number of different congestion control algorithms, including CUBIC [10] and BBR [3].

## VIII.6 Baseline Experiment: Results without Knowing the TCP Variant

Here we present the results of the machine learning and deep learning techniques under all the validation scenarios presented above without a known underlying TCP variant which will play the role of baseline for the other evaluations.

### VIII.6.1 Based on Benchmark Data from Previous Related Work

Looking at Tables VIII.2 and VIII.3, both machine learning and deep learning classification techniques have consistently achieved good levels of precision and recall for all general classes of OSes except iOS. Quantitatively, iOS and Mac OS devices were underrepresented in the benchmark data from previous related work. Besides, as it is shown in Figures VIII.4 and VIII.5, there is a slightly higher misclassification of iOS as unknown and this is why the precision and recall of iOS are comparably lower than the rest of OSes. We also believe that the limited TCP/IP stack basic features could contribute to the indistinguishability and misclassification of OS classes with the same kernel implementation. The false positives are easier to notice in the corresponding confusion matrices.

Table VIII.2: Benchmark data [22] experimental results without a known TCP variant using SVM, RF, and KNN.

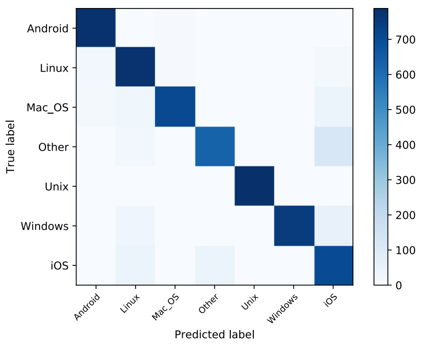
| | SVM | | RF | | KNN | |
|---|---|---|---|---|---|---|
| OS | Precission | Recall | Precision | Recall | Precision | Recall |
| Android | 0.74 | 0.88 | 0.87 | 0.91 | 0.87 | 0.91 |
| Linux | 0.85 | 0.85 | 0.91 | 0.90 | 0.91 | 0.90 |
| Mac OS | 0.65 | 0.77 | 0.61 | 0.83 | 0.58 | 0.88 |
| Other | 0.91 | 0.81 | 0.92 | 0.81 | 0.92 | 0.81 |
| Unix | 0.91 | 0.99 | 0.94 | 0.99 | 0.94 | 0.99 |
| Windows | 0.97 | 0.88 | 0.98 | 0.91 | 0.98 | 0.91 |
| iOS | 0.73 | 0.55 | 0.72 | 0.53 | 0.79 | 0.47 |
| *Average* | 0.83 | 0.82 | 0.85 | 0.84 | 0.86 | 0.84 |
| **Accuracy** | **81.96%** | | **84.07%** | | **83.95%** | |

Table VIII.3: Benchmark data [22] experimental results without a known TCP variant using MLP and LSTM.

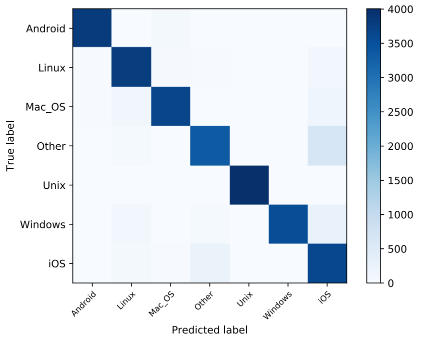
| | MLP | | LSTM | |
|---|---|---|---|---|
| OS | Precision | Recall | Precision | Recall |
| Android | 0.75 | 0.92 | 0.77 | 0.85 |
| Linux | 0.90 | 0.82 | 0.83 | 0.85 |
| Mac OS | 0.62 | 0.81 | 0.58 | 0.83 |
| Other | 1.00 | 0.74 | 0.91 | 0.81 |
| Unix | 0.94 | 0.99 | 0.94 | 0.99 |
| Windows | 0.97 | 0.91 | 0.97 | 0.86 |
| iOS | 0.67 | 0.57 | 0.79 | 0.48 |
| *Average* | 0.84 | 0.82 | 0.83 | 0.81 |
| **Accuracy** | **82.16%** | | **81.04%** | |

### VIII.6.2 Based on Realistic Traffic

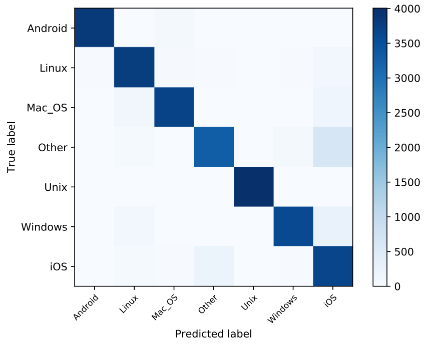
Our performance results of the realistic traffic without a known TCP variant using the machine learning and deep techniques are presented in Tables VIII.4 and VIII.5 respectively. The respective confusion matrix are presented in Figures VIII.6 and VIII.7.



(a) SVM

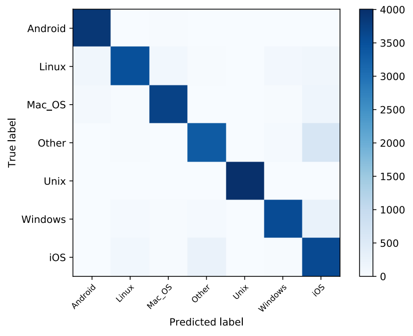


(b) KNN

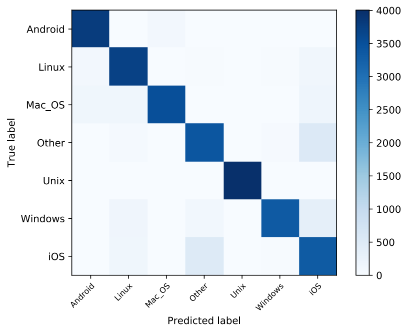


(c) RF

Figure VIII.4: Confusion matrix comparison of the machine learning techniques using the benchmark data from related work [22].

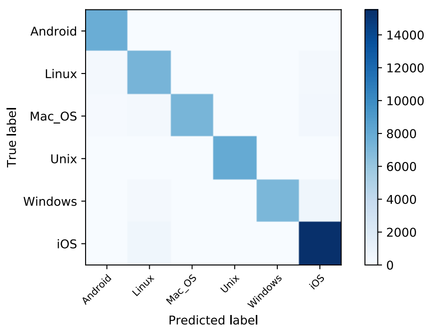


(a) *MLP*

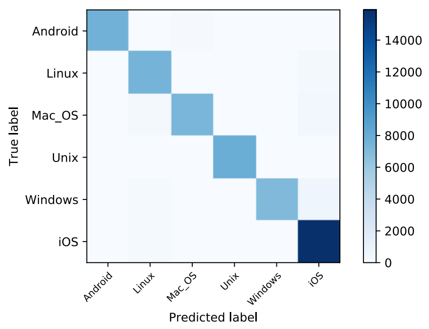


(b) *LSTM*

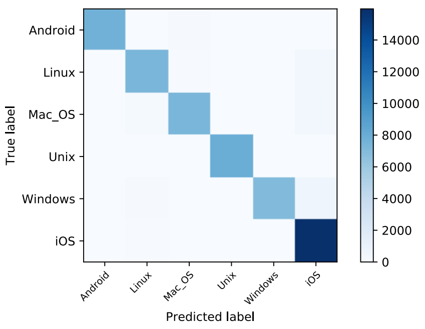
Figure VIII.5: Confusion matrix comparison of MLP and LSTM using the benchmark data from previous related work [22].



(a) SVM

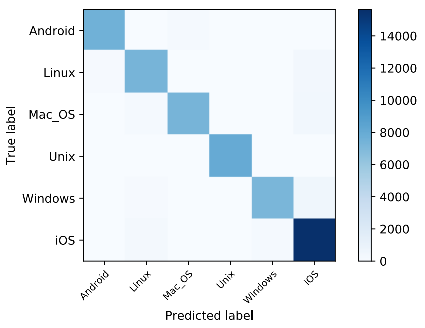


(b) KNN

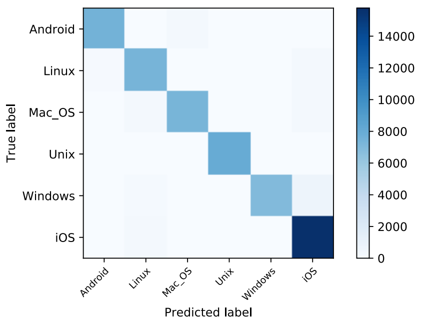


(c) RF

Figure VIII.6: Confusion matrix comparison of the classical machine learning techniques for OSes classification in a realistic traffic.



(a) *MLP*



(b) *LSTM*

Figure VIII.7: Confusion matrix comparison of MLP and LSTM using a realistic traffic.

Table VIII.4: Realistic traffic experimental results without a known TCP variant using SVM, RF, and KNN.

| | SVM | | RF | | KNN | |
|---|---|---|---|---|---|---|
| OS | Precision | Recall | Precision | Recall | Precision | Recall |
| Android | 0.75 | 0.89 | 0.86 | 0.90 | 0.84 | 0.93 |
| Linux | 0.89 | 0.82 | 0.94 | 0.89 | 0.93 | 0.88 |
| Mac OS | 0.63 | 0.81 | 0.61 | 0.82 | 0.61 | 0.82 |
| Unix | 0.94 | 0.99 | 0.94 | 0.99 | 0.94 | 0.99 |
| Windows | 0.97 | 0.89 | 0.98 | 0.89 | 0.98 | 0.89 |
| iOS | 0.88 | 0.72 | 0.86 | 0.73 | 0.88 | 0.72 |
| *Average* | 0.85 | 0.83 | 0.86 | 0.85 | 0.87 | 0.85 |
| **Accuracy** | **83.43%** | | **85%** | | **85.10%** | |

Table VIII.5: Realistic traffic experimental results without a known TCP variant using MLP and LSTM.

| | MLP | | LSTM | |
|---|---|---|---|---|
| OS | Precision | Recall | Precision | Recall |
| Android | 0.81 | 0.83 | 0.76 | 0.86 |
| Linux | 0.89 | 0.79 | 0.90 | 0.81 |
| Mac OS | 0.61 | 0.82 | 0.82 | 0.79 |
| Unix | 0.92 | 0.99 | 0.94 | 0.99 |
| Windows | 0.98 | 0.89 | 0.97 | 0.89 |
| iOS | 0.84 | 0.73 | 0.70 | 0.92 |
| *Average* | 0.84 | 0.83 | 0.83 | 0.84 |
| **Accuracy** | **83.91%** | | **83.27%** | |

### VIII.6.3 Based on Emulated Traffic

Our performance results of the emulated traffic without a known TCP variant as an input feature using both machine learning and deep learning techniques are presented in Tables VIII.6 and VIII.7 respectively. As we can see in the corresponding confusion matrices presented in Figures VIII.8 and VIII.9, there is slightly inaccurate classification of the Mac OS due to its underrepresentation. The precision and recall for the rest of the OSes using machine learning and deep learning techniques are reasonably good.

### VIII.6.4 Comparison of Results without Known TCP Variant

As shown in Tables VIII.2, VIII.3, VIII.4, VIII.5, VIII.6, and VIII.7, our experimental results are pretty consistent. Firstly, we can see that there is not much difference in performance across different machine learning and deep learning techniques. But more importantly, there are not many differences in performance between results from using different types of experimental data. This is intuitively correct, since the OS fingerprinting is based on the basic

Table VIII.6: Emulated traffic experimental results without a known TCP variant using SVM, RF and KNN.

| | SVM | | RF | | KNN | |
|---|---|---|---|---|---|---|
| OS | Precision | Recall | Precision | Recall | Precision | Recall |
| Android | 0.74 | 0.90 | 0.86 | 0.90 | 0.85 | 0.91 |
| Linux | 0.92 | 0.82 | 0.94 | 0.89 | 0.92 | 0.90 |
| Mac OS | 0.63 | 0.81 | 0.61 | 0.82 | 0.61 | 0.82 |
| Unix | 0.94 | 0.99 | 0.94 | 0.99 | 0.94 | 0.99 |
| Windows | 0.97 | 0.89 | 0.98 | 0.89 | 0.98 | 0.89 |
| iOS | 0.88 | 0.73 | 0.86 | 0.73 | 0.88 | 0.73 |
| *Average* | 0.85 | 0.84 | 0.86 | 0.85 | 0.87 | 0.85 |
| **Accuracy** | **84.67%** | | **85.73%** | | **85.27%** | |

Table VIII.7: Emulated traffic experimental results without a known TCP variant using MLP and LSTM.

| | MLP | | LSTM | |
|---|---|---|---|---|
| OS | Precision | Recall | Precision | Recall |
| Android | 0.75 | 0.88 | 0.91 | 0.85 |
| Linux | 0.93 | 0.78 | 0.92 | 0.74 |
| Mac OS | 0.62 | 0.81 | 0.86 | 0.88 |
| Unix | 0.92 | 0.99 | 0.94 | 1.00 |
| Windows | 0.93 | 0.91 | 0.98 | 0.73 |
| iOS | 0.88 | 0.73 | 0.82 | 1.00 |
| *Average* | 0.85 | 0.83 | 0.89 | 0.88 |
| **Accuracy** | **84.05%** | | **88.44%** | |

TCP/IP packet fields, and should not differ much between various types of data, whether we do evaluation using the benchmark data, real data or emulated data. Secondly, we believe accuracy in the range of 82-88% (average value) is perhaps not sufficient for a product in a real deployment. Our hypothesis is that this accuracy could be boosted considerably had we only known the implementation of the underlying TCP variant. We will explore this hypothesis in the next section.

## VIII.7 Oracle-based Experiment: Results using Oracle-given TCP Variant

Here we assume that we know exactly the underlying TCP variant, i.e., we assume it is given by an Oracle. We show that knowledge of the TCP variant has a great potential for boosting passive fingerprinting of OSes, and in this section, we will try to quantify this potential. In the next section, we will show that much of this potential can be harvested by using a tool that predicts the TCP variant.

### VIII.7.1 Based on Benchmark Data from Previous Related Work

Tables VIII.8 and VIII.9 show a significant performance gain across all classes of OSes when we assume prior knowledge of the underlying TCP variant, as compared to the results when the TCP variant is unknown presented in Tables VIII.2 and VIII.3.

Table VIII.8: Benchmark data [22] experimental results with Oracle-given TCP variant using SVM, RF and KNN.

| | SVM | | RF | | KNN | |
|---|---|---|---|---|---|---|
| OS | Precision | Recall | Precision | Recall | Precision | Recall |
| Android | 0.96 | 0.99 | 0.99 | 0.98 | 0.99 | 0.98 |
| Linux | 0.86 | 0.95 | 0.92 | 0.95 | 0.93 | 0.94 |
| Mac OS | 0.98 | 0.89 | 0.97 | 0.92 | 0.97 | 0.92 |
| Other | 0.93 | 0.81 | 0.93 | 0.81 | 0.90 | 0.83 |
| Unix | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| Windows | 0.99 | 0.89 | 0.97 | 0.92 | 0.99 | 0.91 |
| iOS | 0.75 | 0.89 | 0.75 | 0.91 | 0.76 | 0.91 |
| *Average* | 0.92 | 0.92 | 0.93 | 0.93 | 0.93 | 0.93 |
| **Accuracy** | **91.71%** | | **92.73%** | | **92.69%** | |

Table VIII.9: Benchmark data [22] experimental results with Oracle-given TCP variant using MLP and LSTM.

| | MLP | | LSTM | |
|---|---|---|---|---|
| OS | Precision | Recall | Precision | Recall |
| Android | 0.96 | 0.97 | 0.94 | 0.97 |
| Linux | 0.89 | 0.92 | 0.88 | 0.93 |
| Mac OS | 0.96 | 0.92 | 0.97 | 0.88 |
| Other | 0.93 | 0.81 | 0.84 | 0.84 |
| Unix | 1.00 | 1.00 | 1.00 | 1.00 |
| Windows | 0.96 | 0.92 | 0.98 | 0.84 |
| iOS | 0.76 | 0.89 | 0.73 | 0.83 |
| *Average* | 0.92 | 0.92 | 0.91 | 0.90 |
| **Accuracy** | **91.91%** | | **90.03%** | |

### VIII.7.2 Based on Realistic Traffic

The performance results of the realistic traffic with the Oracle-given TCP variant presented in Tables VIII.10 and VIII.11 show the potential of knowing TCP variant given by an Oracle for passive OS fingerprinting in a realistic scenario.

Table VIII.10: Realistic traffic experimental results with Oracle-given TCP variant using SVM, RF and KNN.

| | SVM | | RF | | KNN | |
|---|---|---|---|---|---|---|
| OS | Precision | Recall | Precision | Recall | Precision | Recall |
| Android | 0.95 | 1.00 | 0.99 | 0.98 | 0.99 | 0.98 |
| Linux | 0.86 | 0.91 | 0.94 | 0.93 | 0.92 | 0.94 |
| Mac OS | 0.99 | 0.90 | 0.96 | 0.92 | 0.97 | 0.92 |
| Unix | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| Windows | 0.99 | 0.89 | 0.99 | 0.89 | 0.99 | 0.89 |
| iOS | 0.93 | 0.96 | 0.91 | 0.99 | 0.92 | 0.98 |
| *Average* | 0.95 | 0.95 | 0.96 | 0.96 | 0.96 | 0.96 |
| **Accuracy** | **94.81%** | | **95.65%** | | **95.69%** | |

Table VIII.11: Realistic traffic experimental results with Oracle-given TCP variant using MLP and LSTM.

| | MLP | | LSTM | |
|---|---|---|---|---|
| OS | Precision | Recall | Precision | Recall |
| Android | 0.98 | 0.97 | 0.98 | 0.97 |
| Linux | 0.92 | 0.92 | 0.90 | 0.93 |
| Mac OS | 0.96 | 0.92 | 0.96 | 0.92 |
| Unix | 1.00 | 1.00 | 1.00 | 1.00 |
| Windows | 0.97 | 0.91 | 0.99 | 0.88 |
| iOS | 0.92 | 0.97 | 0.91 | 0.98 |
| *Average* | 0.95 | 0.95 | 0.95 | 0.95 |
| **Accuracy** | **94.98%** | | **94.89%** | |

### VIII.7.3 Based on Emulated Traffic

Our performance results of the emulated traffic with the Oracle-given TCP variant using both classical machine learning and deep learning techniques are presented in Tables VIII.12 and VIII.13. We can see that this shows a significant improvement in performance over the results without a known TCP variant presented in Tables VIII.6 and VIII.7. Both machine learning and deep learning techniques have comparable and consistent results in terms of accuracy.

### VIII.7.4 Comparison of Results with Oracle-given TCP Variant

Our accuracy results presented in Tables VIII.8, VIII.9, VIII.10, VIII.11, VIII.12, and VIII.13, demonstrate that by knowing the TCP variant we obtain a considerable performance boost in all our experimental results, compared to our previous results obtained without knowledge of the TCP flavor. With an Oracle-given TCP variant, we obtain a prediction accuracy of 94-96%, with an average value of 94.1% over all traffic classes and of 95.4% over only emulated

Table VIII.12: Emulated traffic experimental results with the Oracle-given TCP variant using SVM, RF, and KNN.

| | SVM | | RF | | KNN | |
|---|---|---|---|---|---|---|
| OS | Precision | Recall | Precision | Recall | Precision | Recall |
| Android | 0.97 | 0.98 | 0.99 | 0.98 | 0.99 | 0.98 |
| Linux | 0.90 | 0.91 | 0.95 | 0.93 | 0.92 | 0.95 |
| Mac OS | 0.99 | 0.90 | 0.97 | 0.92 | 0.97 | 0.92 |
| Unix | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| Windows | 0.99 | 0.89 | 0.97 | 0.91 | 0.97 | 0.91 |
| iOS | 0.91 | 0.98 | 0.92 | 0.98 | 0.93 | 0.97 |
| *Average* | 0.95 | 0.95 | 0.96 | 0.96 | 0.96 | 0.96 |
| **Accuracy** | **95.10%** | | **96.02%** | | **95.83%** | |

Table VIII.13: Emulated traffic experimental results with the Oracle-given TCP variant using MLP and LSTM.

| | MLP | | LSTM | |
|---|---|---|---|---|
| OS | Precision | Recall | Precision | Recall |
| Android | 0.98 | 0.97 | 0.98 | 0.96 |
| Linux | 0.92 | 0.91 | 0.92 | 0.91 |
| Mac OS | 0.96 | 0.92 | 0.95 | 0.92 |
| Unix | 1.00 | 1.00 | 1.00 | 1.00 |
| Windows | 0.99 | 0.89 | 0.97 | 0.91 |
| iOS | 0.91 | 0.98 | 0.91 | 0.97 |
| *Average* | 0.95 | 0.95 | 0.95 | 0.95 |
| **Accuracy** | **95.11%** | | **95.02%** | |

traffic. The accuracy results are pretty consistent across all scenarios. Comparing these results with our previous results that do not use the Oracle (84.1% on average for all traffic types and 85.6% only for emulated traffic), we observe a solid increase in the OS fingerprinting performance. This improvement would significantly boost the usefulness of a product to be implemented in a real enterprise network infrastructure.

As in the previous section, here again, we observe highly consistent performance results across different machine learning and deep learning techniques and also between the use of different types of experimental data. The latter is useful knowledge for the next section since it means that performance increases obtained over one traffic type is shown to be amenable to other traffic types as well. In the next section, we will have to base our evaluation on emulated data, since we do not have the TCP traffic patterns of the realistic data or benchmark data at hand. These traffic patterns are required to be able to passively infer the TCP variant in the experiments presented in the next section. In this section, the idealistic Oracle was used only to demonstrate the potential of knowing the TCP variant, but this is not a realistic assumption.

Thus, in the next section, we will instead base our evaluation on a TCP variant that is passively predicted by a deep learning-based tool that we developed and presented in our previous work [12, 13, 14]. Using this tool, we explore how close our performance will get to the ideal solution of having an Oracle.

## VIII.8 Prediction-based Experiment: Results Using TCP Variant Prediction

In Section VIII.7, we showed that Oracle-given knowledge of the TCP variant has a great potential for improving the passive OS fingerprinting. In reality, however, we don't have an Oracle-given TCP variant. Since passively detecting the TCP variant is a challenging task, this is where our tool from previous works on predicting the underlying TCP variant from passive measurements [12, 13, 14] comes into play. In this Section we use the TCP variant passively *predicted* by this tool as an input feature for the passive OS fingerprinting. The TCP variant is inferred from the famous Additive Increase and Multiplicative Decrease (AIMD) sawtooth pattern of TCP's estimated cwnd computed based on the outstanding bytes-in-flight. Since we don't have access to the actual cwnd of the senders in the benchmark data and realistic traffic, here we consider only the emulated traffic.

### VIII.8.1 Based on Emulated Traffic

In this section, we use a tool to predict the TCP variant from passive measurements of TCP traffic patterns, and this prediction is used as input to the passive OS fingerprinting method presented above. The experimental results of both techniques are presented in Tables VIII.14 and VIII.15.

Table VIII.14: Emulated traffic experimental results with predicted TCP variant using SVM, RF, and KNN.

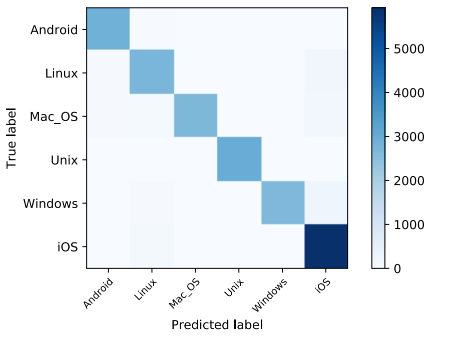
| | SVM | | RF | | KNN | |
|---|---|---|---|---|---|---|
| OS | Precision | Recall | Precision | Recall | Precision | Recall |
| Android | 0.92 | 0.96 | 0.92 | 0.97 | 1.00 | 0.97 |
| Linux | 0.79 | 0.85 | 0.94 | 0.82 | 0.92 | 0.94 |
| Mac OS | 0.96 | 0.88 | 0.97 | 0.87 | 0.85 | 0.94 |
| Unix | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| Windows | 0.92 | 0.78 | 0.85 | 0.80 | 0.88 | 0.91 |
| iOS | 0.85 | 0.94 | 0.86 | 0.96 | 0.93 | 0.87 |
| *Average* | 0.90 | 0.90 | 0.91 | 0.91 | 0.93 | 0.93 |
| **Accuracy** | **90.01%** | | **91.09%** | | **92.15%** | |

Table VIII.15: Emulated traffic experimental results with predicted TCP variant using MLP and LSTM.

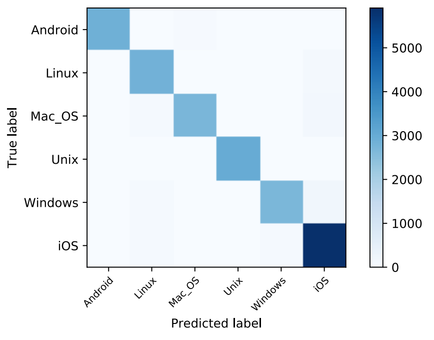
| | MLP | | LSTM | |
|---|---|---|---|---|
| OS | Precision | Recall | Precision | Recall |
| Android | 0.95 | 0.97 | 0.92 | 0.96 |
| Linux | 0.98 | 0.79 | 0.86 | 0.90 |
| Mac OS | 0.95 | 0.90 | 0.95 | 0.88 |
| Unix | 1.00 | 1.00 | 1.00 | 1.00 |
| Windows | 0.94 | 0.77 | 0.97 | 0.77 |
| iOS | 0.82 | 0.99 | 0.88 | 0.96 |
| *Average* | 0.92 | 0.91 | 0.92 | 0.92 |
| **Accuracy** | **91.45%** | | **91.93%** | |

### VIII.8.2 Comparison of Results with a Predicted TCP Variant

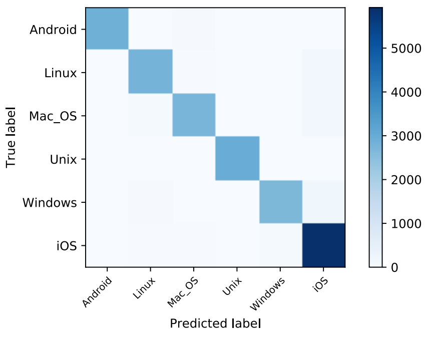
Results with emulated data and a passive prediction of the TCP variant (Tables VIII.14 and VIII.15) gives an accuracy of 91.3% on average, which comes pretty close to the accuracy of 95.4% obtained on emulated traffic with the TCP-variant given by the Oracle. Intuitively, when we do learning based on the TCP variant prediction, the accuracy must be lower than the Oracle-given TCP variant, but the question is how close we can get to the idealistic scenario of having an Oracle. Our results show that using our tool for TCP variant prediction gives reasonably good OS fingerprinting accuracies that come close to the results obtained by using Oracle-given TCP variant. Even though the performance results with the TCP variant passively predicted by our deep learning-based tool are slightly lower as compared to the TCP variant given by an idealistic Oracle, our performance results of using our tool are reasonably competitive.



(a) SVM

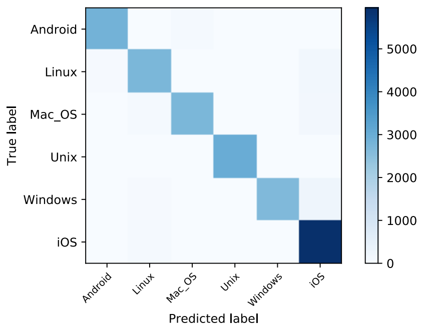


(b) KNN

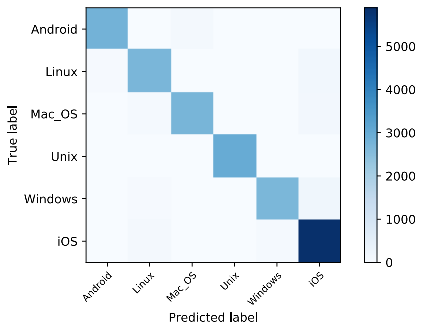


(c) RF

Figure VIII.8: Confusion matrix comparison of the classical machine learning techniques using an emulated traffic.



(a) *MLP*



(b) *LSTM*

Figure VIII.9: Confusion matrix comparison of MLP and LSTM using an emulated traffic.

## VIII.9 Transfer Learning Results

*Transfer learning* is the ability to take a model trained in one scenario and apply it for classification in a different scenario [29, 30, 41]. For example, in our case, that means we are able to train our model on a dataset created in an emulated network with an Oracle-given TCP variant and apply it for classification of our dataset from the realistic traffic. Results shown in Tables VIII.16 and VIII.17 shows that the learning of the OS fingerprinter transfers well into other scenarios. Good transfer learning results indicate that our passive OS fingerprinting model is able to discern the results of unforeseen scenarios and still perform reasonably well. In previous works, we have also demonstrated that the TCP variant predictor performs well in terms of transfer learning [12, 13, 14].

Table VIII.16: Transfer learning experimental results using SVM, RF, and KNN.

| | SVM | | RF | | KNN | |
|---|---|---|---|---|---|---|
| OS | Precision | Recall | Precision | Recall | Precision | Recall |
| Android | 0.95 | 1.00 | 0.98 | 0.98 | 0.99 | 0.98 |
| Linux | 0.86 | 0.91 | 0.90 | 0.95 | 0.92 | 0.95 |
| Mac OS | 0.99 | 0.90 | 0.98 | 0.92 | 0.97 | 0.92 |
| Unix | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| Windows | 0.99 | 0.89 | 0.98 | 0.90 | 0.97 | 0.91 |
| iOS | 0.93 | 0.96 | 0.93 | 0.97 | 0.93 | 0.97 |
| *Average* | 0.95 | 0.95 | 0.95 | 0.95 | 0.96 | 0.96 |
| **Accuracy** | **94.79%** | | **95.35%** | | **95.76%** | |

Table VIII.17: Transfer learning experimental results using MLP and LSTM.

| | MLP | | LSTM | |
|---|---|---|---|---|
| OS | Precision | Recall | Precision | Recall |
| Android | 0.97 | 0.98 | 0.97 | 0.96 |
| Linux | 0.95 | 0.85 | 0.91 | 0.91 |
| Mac OS | 0.94 | 0.94 | 0.96 | 0.90 |
| Unix | 1.00 | 1.00 | 1.00 | 1.00 |
| Windows | 0.99 | 0.89 | 0.98 | 0.87 |
| iOS | 0.90 | 0.98 | 0.90 | 0.98 |
| *Average* | 0.95 | 0.95 | 0.94 | 0.94 |
| **Accuracy** | **94.72%** | | **94.28%** | |

## VIII.10 Conclusion and Future Work

In this paper, we proposed and evaluated a novel approach that attempts to passively fingerprint the underlying remote OS by leveraging *state-of-the-art* machine learning and deep learning techniques under multiple controlled scenarios. We show that knowing the Oracle-given TCP variant has a great potential for boosting the classification performance of passive OS fingerprinting. In our setting, we demonstrate that using the idealistic Oracle has the potential to boost the prediction accuracy from 84.1% to 94.1% on average across all traffic types tested, and from 85.6% to 95.4% in an emulated setting. However, in reality, we don't have the Oracle-given TCP variant and hence we don't know exactly the underlying TCP flavor. To address this, we demonstrated a method for passive OS fingerprinting where the cwnd is first computed based on the outstanding bytes-in-flight, then the underlying TCP flavor is predicted from the estimated cwnd, and finally, the predicted TCP variant is used as an input feature to detect the remote computer's OS. This is an additional feature that is added to the basic TCP/IP features that are the basis of OS fingerprinting in previous works. We demonstrate that our method performs significantly better than not using the predicted TCP variant as an input feature, increasing the accuracy in our experiment from 85.6% to 91.3%. The results of this method come close to the accuracy of 95.4% obtained by using the idealistic Oracle. To the best of our knowledge, this is the first study that reports the potential of the underlying TCP feature in boosting significantly the accuracy of passive OS fingerprinting. We further validate and demonstrate the transferability approach of our OSes classification models by conducting a series of controlled experiments against other scenarios. Through comparing the experimental results between the benchmark dataset, realistic and emulated traffic in terms of accuracy and confusion matrix, it is clear that our passive OSes classification models are able to discern the results to unforeseen scenarios. Therefore, we are able to show that the learned passive OS fingerprinting model by leveraging a pre-trained knowledge of classification techniques from the emulated network performs reasonably well as it is shown in the experimental results when it is applied and transferred to a realistic scenario. Lastly, in all our experiments, we made sure that both the training and validation accuracies are closer which gives an idea about the ability of the OSes classification models to generalize on unforeseen scenarios.

The method presented in this paper, where the cwnd is first computed based on the outstanding bytes-in-flight, then the underlying TCP flavor is predicted from the estimated cwnd, is particularly efficient for loss-based TCP variants. In previous works, we have also developed a tool for the prediction of delay-based TCP flavors [11]. We plan to extend the method presented in this paper to also cover delay-based TCP variants and present it in a follow-on paper. Note that passively detecting the TCP variant is a challenging task, which led to a two-step approach, where the TCP variant prediction of a deep learning-based tool is used as input to another machine learning method in the next step. However, by integrating the two machine learning approaches better, there should be potential for increasing the performance even further and get even closer to the idealistic results of using an Oracle. Exploring such optimizations is also left for future work. It is known that TCP clock drift improves OS fingerprinting and hence measuring differences in the timing of how the IP stack works may allow us to predict the underlying OS with greater assurance in terms of accuracy. We, therefore, argue for using other TCP options like timestamps and queueing delay characteristics as an input feature vector for passive OSes fingerprinting model as another interesting direction. Finally, in addition to the difficulties of establishing ground truth (e.g., the TCP variant) at a larger scale on a dynamic network addressed in Section VIII.3, there is a lot of other work to be done as an extension of our work presented here. For example, addressing answers to valid questions like: *What happens if an end-user (client) changes default parameters that are the basis of OS fingerprinting?* is one possibility for our future work. We expect that end-users don't change parameters often, while servers may do so if it helps improve performance. We believe this would make OS fingerprinting potentially hard.

## Acknowledgment

We would like to thank the Norwegian center for research data (NSD) for granting us the legal permission to collect realistic experiment dataset that contains OS-specific information from the Oslo Metropolitan University. We would also like to thank the 5G 4IoT research lab at Oslo Metropolitan University for allowing us to collect the realistic data for android and iOS devices.

## VIII.11 References

[1] 5G4IoT. 5G4IoT. http://5g4iot.vlab.cs.hioa.no/, 2019.

[2] A. Aksoy, S. Louis, and M. H. Gunes. Operating system fingerprinting via automated network traffic analysis. In *2017 IEEE Congress on Evolutionary Computation (CEC)*, pages 2502–2509. IEEE, 2017.

[3] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson. BBR: Congestion-based congestion control. 2016.

[4] W. R. Cheswick, S. M. Bellovin, and A. D. Rubin. *Firewalls and Internet security: repelling the wily hacker.* Addison-Wesley Longman Publishing Co., Inc., 2003.

[5] N. Davids. Initial TTL values. http://noahdavids.org/self_published/TTL_values.html, 2011.

[6] ESnet. iperf3. https://iperf.fr/iperf-servers.php, 2017.

[7] J. Fan, J. Xu, M. H. Ammar, and S. B. Moon. Prefix-preserving IP address anonymization: measurement-based security evaluation and a new cryptography-based scheme. *Computer Networks*, 46(2):253–272, 2004.

[8] J. Franklin, D. McCoy, P. Tabriz, V. Neagoe, J. V. Randwyk, and D. Sicker. Passive Data Link Layer 802.11 Wireless Device Driver Fingerprinting. In *USENIX Security Symposium*, volume 3, pages 16–89, 2006.

[9] L. G. Greenwald and T. J. Thomas. Toward Undetected Operating System Fingerprinting. *WOOT*, 7:1–10, 2007.

[10] S. Ha, I. Rhee, and L. Xu. CUBIC: a new TCP-friendly high-speed TCP variant. *ACM SIGOPS operating systems review*, 42(5):64–74, 2008.

[11] D. H. Hagos, P. E. Engelstad, and A. Yazidi. Classification of Delay-based TCP Algorithms From Passive Traffic Measurements. In *2019 IEEE 18th International Symposium on Network Computing and Applications (NCA)*. IEEE, 2019.

[12] D. H. Hagos, P. E. Engelstad, A. Yazidi, and Ø. Kure. A machine learning approach to TCP state monitoring from passive measurements. In *2018 Wireless Days (WD)*, pages 164–171. IEEE, 2018.

[13] D. H. Hagos, P. E. Engelstad, A. Yazidi, and Ø. Kure. Recurrent neural network-based prediction of tcp transmission states from passive measurements. In *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*, pages 1–10. IEEE, 2018.

[14] D. H. Hagos, P. E. Engelstad, A. Yazidi, and O. Kure. Towards a Robust and Scalable TCP Flavors Prediction Model from Passive Traffic. In *2018 27th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–11. IEEE, 2018.

[15] T. Henderson, S. Floyd, A. Gurtov, and Y. Nishida. The NewReno modification to TCP's fast recovery algorithm. RFC 6582, 2012.

[16] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[17] V. Jacobson. Congestion avoidance and control. In *ACM SIGCOMM computer communication review*, volume 18, pages 314–329. ACM, 1988.

[18] V. Jacobson, R. Braden, and D. Borman. TCP extensions for high performance. RFC 1323, 1992.

[19] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[20] T. Kohno, A. Broido, and K. C. Claffy. Remote physical device fingerprinting. *IEEE Transactions on Dependable and Secure Computing*, 2(2):93–108, 2005.

[21] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, et al. The quic transport protocol: Design and internet-scale deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 183–196. ACM, 2017.

[22] M. Lastovicka, T. Jirsik, P. Celeda, S. Spacek, and D. Filakovsky. Passive OS Fingerprinting Methods in the Jungle of Wireless Networks. In *NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium*, pages 1–9. IEEE, 2018.

[23] R. Lippmann, D. Fried, K. Piwowarski, and W. Streilein. Passive operating system identification from TCP/IP packet headers. In *Data Mining for Computer Security*. Citeseer, 2003.

[24] R. Lippmann, S. Webster, and D. Stetson. The effect of identifying vulnerabilities and patching software on the utility of network intrusion detection. In *International Workshop on Recent Advances in Intrusion Detection*, pages 307–326. Springer, 2002.

[25] G. F. Lyon. Remote OS detection via TCP/IP stack fingerprinting. *Phrack Magazine*, 8(54), 1998.

[26] G. F. Lyon. *Nmap network scanning: The official Nmap project guide to network discovery and security scanning*. Insecure, 2009.

[27] Netresec. Networkminer. https://www.netresec.com/?page=NetworkMiner, 2007.

[28] A. Ornaghi and M. Valleri. Ettercap. https://www.ettercap-project.org/, 2015.

[29] S. J. Pan. Transfer Learning., 2014.

[30] S. J. Pan and Q. Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2010.

[31] J. Postel. Internet control message protocol. RFC 792, 1981.

[32] J. Postel. Internet control message protocol. RFC 792, 1981.

[33] J. Postel. Transmission control protocol. RFC 793, 1981.

[34] F. Rosenbaltt. The perceptron–a perciving and recognizing automation. *Report 85-460-1 Cornell Aeronautical Laboratory, Ithaca, Tech. Rep.*, 1957.

[35] J. Scambray, S. McClure, and G. Kurtz. *Hacking exposed.* McGraw-Hill Professional, 2000.

[36] SCOTT. European Leadership Joint Undertaking. https://scottproject.eu/, 2019.

[37] R. Spangler. Analysis of remote active operating system fingerprinting tools. *University of Wisconsin*, 2003.

[38] G. Taleck. Synscan: Towards complete tcp/ip fingerprinting. *CanSecWest, Vancouver BC, Canada*, pages 1–12, 2004.

[39] K. Tan, J. Song, Q. Zhang, and M. Sridharan. A compound TCP approach for high-speed and long distance networks. In *Proceedings-IEEE INFOCOM*, 2006.

[40] W. Wei, K. Suh, B. Wang, Y. Gu, J. Kurose, and D. Towsley. Passive online rogue access point detection using sequential hypothesis testing with TCP ACK-pairs. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, pages 365–378. ACM, 2007.

[41] K. Weiss, T. M. Khoshgoftaar, and D. Wang. A survey of transfer learning. *Journal of Big data*, 3(1):9, 2016.

[42] J. Xu, J. Fan, M. Ammar, and S. B. Moon. On the design and performance of prefix-preserving IP traffic trace anonymization. In *ACM SIGCOMM*, pages 263–266. ACM, 2001.

[43] L. Xu, K. Harfoush, and I. Rhee. Binary increase congestion control (BIC) for fast long-distance networks. In *IEEE INFOCOM*, volume 4, pages 2514–2524. IEEE, 2004.

[44] F. Yarochkin and O. Arkin. Xprobe2- A'Fuzzy'Approach to Remote Active Operating System Fingerprinting, 2002.

[45] M. Zalewski. p0f: Passive OS fingerprinting tool. *Online at http://lcamtuf.coredump.cx/p0f3*, 2017.

[46] B. Zhang, T. Zou, Y. Wang, and B. Zhang. Remote operation system detection base on machine learning. In *2009 Fourth International Conference on Frontier of Computer Science and Technology*, pages 539–542. IEEE, 2009.

Paper IX

# A Deep Learning-based Universal Tool for Operating Systems Fingerprinting from Passive Measurements

**Desta Haileselassie Hagos[1], Anis Yazidi, Øivind Kure, Paal E. Engelstad**

***Submitted for review*.**

Since this paper is a journal extension of Paper VIII, it is not included as part of this dissertation to avoid redundancy for the reader.

[1]University of Oslo, Department of Informatics, destahh@ifi.uio.no