A Heuristic-Based Approach to Real-Time TCP State and Retransmission Analysis

A thesis presented to

the faculty of

the Russ College of Engineering and Technology of Ohio University

In partial fulfillment

of the requirements for the degree

Master of Science

James E. Swaro

December 2015

© 2015 James E. Swaro. All Rights Reserved.
This thesis titled
A Heuristic-Based Approach to Real-Time TCP State and Retransmission Analysis

by

JAMES E. SWARO

has been approved for
the Department of Electrical Engineering and Computer Science
and the Russ College of Engineering and Technology by

Shawn Ostermann
Associate Professor of Electrical Engineering and Computer Science

Dennis Irwin
Dean, Russ College of Engineering and Technology
ABSTRACT

SWARO, JAMES E., M.S., December 2015, Computer Science

A Heuristic-Based Approach to Real-Time TCP State and Retransmission Analysis (130 pp.)

Director of Thesis: Shawn Ostermann

This study focuses on understanding how to classify out-of-order network traffic sent using the Transport Control Protocol (TCP). Packets that arrive out of order are the result of network reordering or loss recovery. TCP initiates loss recovery in response to the perceived loss of data, decreasing the congestion window and throughput of the connection. When TCP reacts poorly to loss, throughput may drop, latency may increase, and congestion collapse may occur.

This thesis analyzes TCP traffic from an arbitrary observation point in a network, rather than at the TCP endpoint. Observing traffic at a TCP endpoint inhibits the inference of loss and detection of network reordering in one direction of the connection. Alternatively, observing traffic at an arbitrary point between two TCP endpoints allows inference of loss and detection of network reordering in both directions. Positioning the observation point at an arbitrary point can increase the diversity of observed connections, increasing the likelihood of detecting rare forms of aberrant behavior.

In this paper, several algorithms and heuristics for classification of out-of-order TCP traffic are analyzed and implemented in a new TCP traffic analyzer called tcprs. An in-depth analysis of each algorithm and heuristic is given and compared with the results from tcptrace and tcpcsm. It was found that tcprs achieves an improvement in classification accuracy as compared with tcptrace and tcpcsm.
ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Ostermann, for the constant stream of ideas and suggestions. I am thankful for my opportunity to work with him and the rest of the students in the IRG Lab. A special thanks to Dr. Kruse for his thoughts on my ideas about traffic analysis.

Thanks to Samuel Jero, Zack Sims, Gilbert Clark, and Joshua Schendel for all of the help in the lab. It was great to discuss research ideas and issues with so many talented people.

My thanks to the Bro-IDS team for the development of a flexible traffic analyzer framework. Thanks as well to Katrina LaCurts for the initial work done on the TCPSstats analyzer which served as an model for my early work on the tcprs traffic analyzer.

Thanks to Google for providing a trace file for analysis. The preliminary analysis of the trace file led to the initial investigation into retransmission analysis.

My thanks extend to my good friend, Michael Cleaver. He has dedicated many evenings listening to my design ideas and complaints without falter, something which I appreciate greatly.

I would thank John Dykstra, a friend and colleague, who has been incredibly helpful in many different ways. He is a patient man who listened to many of my inane ideas, and provided valuable and well-grounded feedback.

I would also like to thank my lovely girlfriend, Lif, who has listened to more tirades about TCP analysis than I care to admit to. Her patience and exterior perspective has helped in many ways to shape the final version of this thesis.
# Table of Contents

Abstract ........................................................................................................... 3

Acknowledgments .......................................................................................... 4

List of Tables ................................................................................................. 7

List of Figures ............................................................................................... 8

1 Introduction .................................................................................................. 10
   1.1 Transport Control Protocol .................................................................. 11
   1.2 Network Congestion .......................................................................... 12
   1.3 Congestion Control Analysis ............................................................... 13
   1.4 Research Goals .................................................................................. 14

2 Background .................................................................................................. 16
   2.1 TCP ...................................................................................................... 16
      2.1.1 Early History ............................................................................... 16
   2.2 Network Traffic Analysis Framework (Bro) ........................................ 18
   2.3 TCP Traffic Analysis ......................................................................... 18
      2.3.1 Difficulties .................................................................................. 18
         2.3.1.1 Packet Capture .................................................................... 19
         2.3.1.2 Vantage Point ..................................................................... 19
         2.3.1.3 Stack Fingerprinting .............................................................. 27
      2.3.2 Prior Work ................................................................................... 28
         2.3.2.1 Tcpanaly ............................................................................... 28
         2.3.2.2 Tcptrace ............................................................................... 29
         2.3.2.3 Tcpdebug .............................................................................. 30
         2.3.2.4 Tcpesm ................................................................................ 31

3 Motivation .................................................................................................... 32
   3.1 Analysis Methods ............................................................................... 32
   3.2 Protocol Efficiency ............................................................................ 32
   3.3 Loss Recovery .................................................................................... 33
   3.4 Network Reordering .......................................................................... 36

4 Method .......................................................................................................... 38
   4.1 Classification Taxonomy ..................................................................... 38
   4.2 Round-trip Time Estimation ............................................................... 40
   4.3 Retransmission/Reordering Detection ................................................ 42
4.3.1 Retransmission Detection ........................................... 42
4.3.2 Reordering Detection ........................................... 45
4.4 Retransmission Classification ........................................... 54

5 Experiments, Result and Analysis ........................................... 57
5.1 Datasets ........................................... 57
5.2 Traffic Analysis and Instrumentation ........................................... 58
5.3 Sampling Process ........................................... 59
5.4 Comparative Analysis ........................................... 61
  5.4.1 TCPTRACE ........................................... 63
  5.4.2 TCPCS ........................................... 69
    5.4.2.1 Reordering Event Detection ........................................... 70
    5.4.2.2 Retransmission Timeout Event Detection ........................................... 74
    5.4.2.3 Fast Retransmit-like Event Detection ........................................... 78
    5.4.2.4 Analyzer Design Errors ........................................... 80
    5.4.2.5 Retransmission Timeout Detection Errors ........................................... 83
    5.4.2.6 Reordering Detection Errors ........................................... 87
    5.4.2.7 Fast Retransmit-like Detection Errors ........................................... 88
  5.5 Results ........................................... 91

6 Conclusions ........................................... 99
  6.1 Future Work ........................................... 100
    6.1.1 OS/TCP Fingerprinting ........................................... 100
    6.1.2 Elimination of Retransmission Classifications Based on Empirical Evidence ........................................... 101
    6.1.3 Heuristics ........................................... 101
    6.1.4 Modified Reordering Taxonomy ........................................... 102

References ........................................... 103

Appendix A: Time Sequence Graphs ........................................... 108

Appendix B: TCPRS Annotations ........................................... 115

Appendix C: Comparative Analysis Scripts ........................................... 118

Appendix D: SourceRepositories ........................................... 124

Appendix E: Additional TCP Classification Examples ........................................... 125
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Dataset Information</td>
<td>58</td>
</tr>
<tr>
<td>5.2 Dataset Event Data</td>
<td>58</td>
</tr>
<tr>
<td>5.3 Feature Comparison</td>
<td>62</td>
</tr>
<tr>
<td>5.4 Tcptrace Events by Dataset and Type</td>
<td>68</td>
</tr>
<tr>
<td>5.5 Tcprs Events by Dataset and Type</td>
<td>68</td>
</tr>
<tr>
<td>5.6 Differences Between tcptrace and tcprs by Dataset and Event Type</td>
<td>68</td>
</tr>
<tr>
<td>5.7 Distribution of Sampled FRETX Events by Dataset with Estimated Accuracy</td>
<td>91</td>
</tr>
<tr>
<td>5.8 Distribution of Sampled RTO Events by Dataset with Estimated Accuracy</td>
<td>92</td>
</tr>
<tr>
<td>5.9 Distribution of Sampled REORDER Events by Dataset with Estimated Accuracy</td>
<td>93</td>
</tr>
<tr>
<td>5.10 Percentage of Events that had the Same Classification in tcpcs m and tcprs</td>
<td>93</td>
</tr>
<tr>
<td>5.11 Estimated Percentages of Correct Classifications Made by tcprs by Event Type</td>
<td>94</td>
</tr>
<tr>
<td>5.12 Estimated Percentages of Correct Classifications Made by tcpcs m by Event Type</td>
<td>95</td>
</tr>
<tr>
<td>5.13 Estimated Accuracy by Top-Level Comparison Event Type and Dataset for tcp rs</td>
<td>95</td>
</tr>
<tr>
<td>5.14 Estimated Precision by Top-Level Comparison Event Type and Dataset for tcp rs</td>
<td>96</td>
</tr>
<tr>
<td>5.15 Estimated Accuracy by Top-Level Comparison Event Type and Dataset for tcpcs m</td>
<td>96</td>
</tr>
<tr>
<td>5.16 Estimated Precision by Top-Level Comparison Event Type and Dataset for tcpcs m</td>
<td>97</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>VP Case 1: Two Vantage Points Located at Each TCP Endpoint</td>
<td>20</td>
</tr>
<tr>
<td>2.2</td>
<td>VP Case 2: Two Vantage Points, VP 1 Located at Endpoint, VP 2 Located Before Loss Occurs</td>
<td>21</td>
</tr>
<tr>
<td>2.3</td>
<td>VP Case 3: Two Vantage Points, VP 1 Located at Endpoint, VP 2 Located After Loss Occurs</td>
<td>21</td>
</tr>
<tr>
<td>2.4</td>
<td>VP Case 4: Two Vantage Points, Both VPs Exist Between TCP Endpoints</td>
<td>22</td>
</tr>
<tr>
<td>2.5</td>
<td>VP Case 5: One Vantage point Located at the Sender</td>
<td>23</td>
</tr>
<tr>
<td>2.6</td>
<td>VP Case 6: One Vantage Point Located at the Receiver</td>
<td>24</td>
</tr>
<tr>
<td>2.7</td>
<td>VP Case 7: One Vantage Point, Packet Lost After Observed</td>
<td>24</td>
</tr>
<tr>
<td>2.8</td>
<td>VP Case 8: One Vantage Point, Packet Lost Before Observed</td>
<td>25</td>
</tr>
<tr>
<td>2.9</td>
<td>Positioning: VP Exists at Endpoint</td>
<td>26</td>
</tr>
<tr>
<td>2.10</td>
<td>Positioning: VP exists Between Sets of Endpoints</td>
<td>26</td>
</tr>
<tr>
<td>4.1</td>
<td>Classification Taxonomy</td>
<td>39</td>
</tr>
<tr>
<td>4.2</td>
<td>Normal RTO</td>
<td>42</td>
</tr>
<tr>
<td>4.3</td>
<td>Spurious RTO</td>
<td>43</td>
</tr>
<tr>
<td>4.4</td>
<td>Retransmission Detection via Outstanding Data Overlap</td>
<td>44</td>
</tr>
<tr>
<td>4.5</td>
<td>Retransmission Detection via Timestamp Option</td>
<td>46</td>
</tr>
<tr>
<td>4.6</td>
<td>Retransmission Detection via ACK Analysis</td>
<td>48</td>
</tr>
<tr>
<td>4.7</td>
<td>Retransmission Detection via Gap Analysis</td>
<td>51</td>
</tr>
<tr>
<td>5.1</td>
<td>Multiple Connection Trace</td>
<td>65</td>
</tr>
<tr>
<td>5.2</td>
<td>Connection with a False Reordering</td>
<td>66</td>
</tr>
<tr>
<td>5.3</td>
<td>Another Connection with False Reordering</td>
<td>67</td>
</tr>
<tr>
<td>5.4</td>
<td>Case: TCP Header Timestamp Option Based Reordering Detection</td>
<td>72</td>
</tr>
<tr>
<td>5.5</td>
<td>Case: Sequence Gap Based Reordering Detection</td>
<td>73</td>
</tr>
<tr>
<td>5.6</td>
<td>Case: TCP Header Acknowledgment Based Reordering Detection</td>
<td>73</td>
</tr>
<tr>
<td>5.7</td>
<td>Case: Repeated Retransmissions via Retransmission Timeout</td>
<td>74</td>
</tr>
<tr>
<td>5.8</td>
<td>Case: Unsolicited Retransmission via Retransmission Timeout</td>
<td>75</td>
</tr>
<tr>
<td>5.9</td>
<td>Case: Retransmission Timeout Inferred from RTT Information</td>
<td>76</td>
</tr>
<tr>
<td>5.10</td>
<td>Case: Retransmission Timeout from Incorrect Link Layer Duplicate Classification</td>
<td>77</td>
</tr>
<tr>
<td>5.11</td>
<td>Case: Retransmission Timeout Detected Outside of Implemented Connection Boundaries</td>
<td>77</td>
</tr>
<tr>
<td>5.12</td>
<td>Case: Retransmission Based on Fast Retransmit-like Behavior Following a Closed Loss Recovery Window</td>
<td>78</td>
</tr>
<tr>
<td>5.13</td>
<td>Case: Retransmission Based on Fast Retransmit-like Behavior</td>
<td>79</td>
</tr>
<tr>
<td>5.14</td>
<td>Case: Retransmission Based on Fast Retransmit-like Behavior Involving Coalesced Packets</td>
<td>80</td>
</tr>
</tbody>
</table>
5.15 Case: Misclassification of Event Due to Failure to Properly Classify Preceding Events ................................................................. 81
5.16 Case: Misclassification of Event Due to Reordered Retransmissions at the Beginning of Loss Recovery ..................................................... 82
5.17 Case: Misclassification of Event Due to Data Discard Policy in tcprs . . . . 82
5.18 Case: Misclassification of Event Due to Event Labeling Policy in tcprs . . . 83
5.19 Case: Misclassification Due to Bad RTO Estimate ................................................. 84
5.20 Case: Misclassification Due to Bad RTO Estimate of False Link Layer Duplicate 85
5.21 Case: Misclassification Due to Unexpected Behavior from Non-RFC Compliant TCP Implementation .................................................... 86
5.22 Case: Misclassification Due to Failure to Consider Cumulative Acknowledgments .......................................................... 87
5.23 Case: Misclassification Due to Unexpected Secondary Retransmission Classification ........................................................... 88
5.24 Case: Misclassification Due to Unreceived Acknowledgment ............................... 89
5.25 Case: Misclassification Due to RTT changes ...................................................... 90
5.26 Case: Misclassification Due to Deviations from Expected Loss Recovery Window Behavior ......................................................... 90

A.1 Sample Time Sequence Graph ............................................................... 110
A.2 Zoom of Sample Time Sequence Graph .................................................. 111
A.3 Zoom of Sample Time Sequence Graph at the Beginning of a Connection ..... 111
A.4 Zoom of Sample Time Sequence Graph at the End of a Connection ............ 112
A.5 Time Sequence Graph Showing SACK Blocks ....................................... 112
A.6 Time Sequence Graph Showing PUSH Segments ..................................... 113
A.7 Time Sequence Graph Showing URG Segment ....................................... 113
A.8 Time Sequence Graph Showing Zero Window Advertisements .................... 114

E.1 Case: FREC Example - Incorrect 3 ......................................................... 125
E.2 Case: FREC Example - Correct 2 ........................................................ 125
E.3 Case: FRETX Example - Correct 2 ..................................................... 126
E.4 Case: LOSSREC Example - Correct 3 ................................................. 126
E.5 Case: MSRETX Example - Incorrect 2 .................................................. 127
E.6 Case: REORDER Example - Correct .................................................... 127
E.7 Case: SACKFRETX Example - Incorrect 2 ......................................... 128
E.8 Case: SACKFRETX Example - Correct 3 ............................................. 128
E.9 Case: LOSSREC Example - Correct 2 ............................................... 129
E.10 Case: REORDER Example - Correct 2 ............................................. 129
E.11 Case: REORDER Example - Incorrect 2 ............................................ 130
1 INTRODUCTION

Many applications use TCP to communicate information between peers on a network. TCP traffic constitutes the majority of internet traffic in terms of bytes, connections, and packets[Wil01, cai]. While new services and technologies emerging on the world wide web may use different protocols, it is likely that TCP will remain the dominant transport layer protocol for the immediate future[TMW97]. Considering this information, it is important that the protocol is as efficient as possible.

The effectiveness of loss recovery is one of many important metrics to measuring the efficiency of TCP. Loss recovery affects the effective throughput of a connection, a measure of how quickly and efficiently a protocol can send data in the face of network congestion. Network congestion is one of the primary causes of packet loss[Pax97c]. Reliable delivery protocols such as TCP must attempt to mitigate the effects of packet loss by using loss recovery algorithms to re-send lost packets[Pos81]. If the loss recovery algorithms of the protocol are inefficient, loss recovery can take a prolonged period of time, resulting in reduced effective throughput[Jac88].

Existing network traffic analysis tools provide statistics and information about TCP connections for measuring protocol performance and debugging. Some tools only provide basic statistics about loss recovery such as packet retransmission counts, or packet reordering counts. Fewer tools provide in-depth details about loss recovery, such as effective throughput, or packet classification by loss recovery method.

This study investigates the difficulties of TCP loss recovery analysis and details the implementation of a TCP traffic analyzer called "tcpr". The next sections will discuss TCP, loss recovery analysis, previous research and the goals of this study.
1.1 Transport Control Protocol

TCP is a reliable end-to-end transport layer level protocol used by the majority of IP network traffic [TMW97, Odl03, Wil01, caI]. A reliable transport layer protocol guarantees delivery of data submitted for transfer by an application to the protocol, retransmitting data when necessary. In addition to reliable delivery, TCP uses two types of algorithms to govern protocol response to packet loss – congestion and loss recovery. Congestion control and loss recovery algorithms are responsible for handling protocol behavior in response to loss.

Congestion control algorithms govern the injection of packets into a network to avoid congestion. TCP uses several different congestion control algorithms such as slow start [Jac88]. Slow start is a congestion control algorithm aimed at increasing the size of the congestion window of TCP to an equilibrium state for a network. This works by increasing the window in size with the receipt of acknowledgments from packets sent. The increase in window size allows TCP to transmit more packets until congestion occurs, eventually arriving at the equilibrium state. The use of congestion control algorithms helps to limit the amount of time spent in loss recovery.

Loss recovery algorithms govern the response of TCP in the event of loss. Fast retransmit is a loss recovery algorithm that allows TCP to initiate loss recovery after the receipt of three duplicate acknowledgments. Once the three duplicate acknowledgments are observed, TCP retransmits the first packet that it suspects to be lost. Once the first lost packet is retransmitted, TCP will then attempt to retransmit lost data upon receipt of new cumulative acknowledgments until all lost data is fully recovered. To cover a wide variety of network loss cases, TCP may use multiple loss recovery algorithms based on different triggering conditions.

Loss recovery algorithms directly affect the performance of TCP. If TCP takes too long to recover from loss, latency increases and throughput decreases. However, if loss
recovery algorithms are too aggressive, TCP may trigger spurious retransmissions of data it suspects to be lost, potentially introducing more congestion into the network and causing further loss of data. Loss cannot be detected with perfect accuracy given the information available at a single endpoint. TCP relies on rules, timers or heuristics such as transmission timeouts and duplicate acknowledgment counting to infer loss. If the loss inference methods fail, TCP responds to perceived loss with an inappropriate response, potentially introducing even more congestion. So if loss recovery is handled carefully and efficiently, then time spent in recovery and performance decreases are minimized.

1.2 Network Congestion

Network congestion occurs when network resources become oversubscribed to the point where router queues become filled and packets are dropped. Network congestion symptoms can range from mild queuing delays to full-scale congestion collapse. Congestion collapse is defined as a state of a network in which little or no useful communication can occur due to network congestion[Jac88]. As stated by Jacobson, “The ‘obvious’ ways to implement a window-based transport protocol can result in exactly the wrong behavior in response to network congestion”[Jac88].

In 1986, the first documented instances of congestion collapse were observed between LBL and UC Berkeley, where throughput dropped from 40 Kbps to 32bps[Jac88]. The massive reduction in throughput was attributed to poor protocol response at the transport layer. This event initiated one of the first investigations into TCP congestion control analysis. The result of the investigation lead to the creation of several key algorithms in modern TCP implementations such as fast retransmit, exponential retransmit timer back-off and slow start[Jac88]. Several of those algorithms are still analyzed and experimented upon today as part of the on-going process of understanding network congestion and how to avoid it.
1.3 Congestion Control Analysis

There are two types of congestion control analysis, passive and active. Passive analysis involves observation of one or more data streams without interaction between the vantage point[Pax97b] and the endpoints of a TCP connection. Passive analysis approaches benefit from little functional or infrastructure overhead but lack the ability to accurately identify potential protocol quirks such as differences in the number of duplicate acknowledgments necessary to trigger loss recovery. Each bit of information known about a TCP endpoint can lead to increased accuracy when performing congestion control analysis. Examples of passive analysis tools would include programs such as tcptrace[Ost] and tcpanaly[Pax97b].

Active analysis involves probing TCP endpoints for information such as protocol implementation quirks[CL, Bev04]. Tools such as TBIT[PF01] or CAAI[YLX+11] actively connect to web servers in an attempt to glean information about the TCP implementations on each server. These tools need more infrastructure and may not be able to operate on as large of a data set as a passive analysis approach could.

Passive analysis for congestion event detection would be most valuable on larger sets of data, since a vantage point can often be placed at the gateway of a large institution. The vantage point observes all packets that pass the vantage point and record them in packet trace files. These files can then be processed offline at a later time. Many institutions, such as WAND[Gro] and universities, offer large bulk traces files for use in research into congestion control analysis.

Tcptrace is a tool for use in TCP congestion control analysis. It provides a model for detecting retransmissions and network reordering that is correct and executes quickly. However, it does not provide a way of distinguishing what retransmission algorithm was responsible for the retransmission. TCPDEBUG [RS06] expands upon this by creating a congestion state machine for several popular TCP stack implementations. The models
used by tcpdebug are effective for the TCP implementations of the study but do not seem suited for use as a generic congestion state analysis model.

1.4 Research Goals

This research started as an analysis of data provided by Google to identify the prevalence of network reordering and specific types of retransmissions in modern networks. Several different TCP traffic analysis tools were examined during the initial investigation into loss recovery and network reordering detection. Each tool was critiqued by the level of detail given about loss recovery and network reordering events. Some tools, such as tcptrace, provided simple statistics but did not provide enough detail into loss recovery to be viable for the investigation. Other tools, such as tcpcsm or tcpdebug, provided detailed information but differed greatly in their interpretation of the same traces. With several TCP traffic analysis tools differing in terms of classification of events, it is prudent to understand why and how the differences arise. The information learned from the analysis should be applied to the design of a new TCP traffic analysis tool to demonstrate an improvement in loss event classification accuracy. The research goals of this thesis are to understand the differences in events observed by existing TCP analysis tools, and develop a new TCP traffic analyzer for detecting network reordering and retransmission events.

Some existing TCP traffic analysis tools, such as TCPDEBUG[RKS06] and tcpcsm[AN11] use a set of TCP state machines to classify loss events by loss recovery/congestion control algorithm. The TCP state machines are modeled to represent specific congestion control algorithms such as New Reno. The accuracy of analysis done using state machines is highly dependent upon accurate identification of the algorithms used by the endpoint[RKS06]. However, TCP and OS fingerprinting do not always correctly identify the target TCP implementation. While specific state machines may
provide more accurate answers when fingerprinting succeeds, accuracy may diminish when fingerprinting fails to identify the correct TCP implementation.

In this paper, the results of TCP analysis tools such as tcpcsm and tcptrace are compared with the results from tcprs. By feeding the same input data to the various analysis tools and comparing the results, it is possible to identify which analysis strategies were more accurate. The results of the analysis will be used to further the understanding of TCP traffic analysis and design of TCP traffic analyzers.
2 Background

This chapter provides background information relevant to the work done in this thesis. First, TCP is defined and a brief history of the work done on TCP with respect to congestion is given; this section will define many of the terms used throughout the thesis. Following the section on TCP is a description of the network traffic analysis framework used in the design of the analyzer that resulted from this work. Then, previous work in the field of network traffic analysis specific to TCP will be discussed. Finally, the heuristics that were used or created in the process of this thesis will be discussed.

2.1 TCP

TCP is a reliable, connection-oriented stream protocol[Pos81]. TCP and several other protocols were designed as part of the creation of ARPANET, a DARPA/ARPA project to connect networks[Com00] in the late 1970s. The intent was to connect research sites and computers so that data could be shared and transferred over the network. This was the beginning of a much larger network, now referred to as ‘The Internet’, a collection of networks that share data on global scale with thousands of networks and millions of hosts.

In the next few sections, I will describe a brief history of TCP, TCP analysis tools, and analysis difficulties that have been discovered.

2.1.1 Early History

By 1988, the first documented instances of congestion collapse were observed and investigated by Jacobson and Karels[Jac88]. The results of their investigation drove the development of several new algorithms for TCP, including slow-start and round-trip-time variance estimation. Many, if not all of the new algorithms proposed in their work are now implemented in every modern TCP implementation. Each algorithm proposed by
Jacobson and Karels addressed a fundamental problem with the initial design of TCP that resulted in poor congestion handling.

Congestion is the result of a net flow of packets along a network path for which the capacity to handle has been exceeded. The first level of defense against congestion is the queuing of packets along the path until they can be sent out again. However, once the queues on an intermediate node in the path are full, no more packets can be stored and thus must be dropped. One of the most common causes of loss on a network is congestion\cite{Pax97a}. Other causes of loss include corruption of data, hardware errors and more.

Many tools have been developed to help researchers understand the effects of undesirable network behaviors such as packet duplication, packet loss and network reordering. Programs such as tcptrace\cite{Ost} and tcpanaly\cite{Pax97b} are analysis tools for TCP connections that allow researchers to understand protocol behavior through interpreting the statistics provided by each. The interpretation of the statistics can help researchers to understand the behavior of the protocol in more detail.

Not all tools designed for understanding TCP have the same design. Some tools were not in the form of analyzers but rather emulators which provided a certain level of insight into a connection. TCP emulation has been performed by replaying data into an emulation environment to generate the expected values of internal TCP variables over the duration of a connection\cite{BPS+98}. Similar to replaying a connection, analyzers such as tcptrace and LEAST\cite{AEO03} replay the connection and emulate aspects of the connection as well as providing statistical data and reports of the information gathered. Other analyzers have used more mathematical approaches such as Markov models to model the behavior of TCP connections\cite{CM00}.
2.2 Network Traffic Analysis Framework (Bro)

Bro[Pax99] was the framework which was chosen to tackle this problem. Bro is an open source network traffic analysis framework originally developed by Vern Paxson\(^1\) for network security monitoring. It utilizes many widely accepted models for packet capture and traffic analysis, which would not have be duplicated. Bro uses a layered traffic analysis approach that splits the analysis of individual network layers into specific analyzers. Adding an additional analyzer to a specific layer such as the transport layer is a relatively trivial process due to the design of Bro as a network traffic analysis framework.

Bro also supports many different methods of encapsulation, increasing the number of different types of network traffic that the analyzer would be able to observe. The wider variety of connections is useful for catching rare behavior that might be prevalent in one form of encapsulation than in others. Bro handles reading the network traffic, whether in the form of packet capture saved to file or a live network trace.

2.3 TCP Traffic Analysis

This section discusses two essential aspects of TCP traffic analysis: analysis difficulties and the prior work done in the field of TCP traffic analysis.

2.3.1 Difficulties

There are many difficulties in analyzing protocol behavior, such as packet capture position, OS/stack fingerprinting, protocol responses that differ from specification and implementation errors. The position of the packet capture point can cause irregularities in the observed data and the observed response, particular in the expected response of the protocol. Fingerprinting provides information about operating system and TCP stack in use but it may be difficult to pinpoint which implementation is in use. TCP design

\(^1\) Vern Paxson also authored tcpanaly.
documents can be used to infer specific information about how a connection is expected to behave. However, the implementation of TCP at any endpoint may not follow the specification, usually in pursuit of a performance increase or a misinterpretation causing erratic or incorrect behavior in the face of congestion. Each of these issues provide a wide array of difficulties that an analysis tool must consider.

### 2.3.1.1 Packet Capture

Several analysis issues can arise when using a packet capture device, such as loss, duplication or configuration error[1]. It is critical that an analysis tool should attempt to detect and rectify packet capture errors when it makes sense to do so.

The most widely recognized(and often most common) form of packet filter error is that of *drops*, in which the trace produced by the filter fails to include all of the packets appearing on the network link that matched the filter pattern

– Paxson, 1997 [Pax97b]

Loss of packets by the capture device can be inferred to some degree, but the loss of data can have variety of different implications. Duplication of packets by the packet capture device is possible as well, appearing as a hardware level duplication during analysis. Hardware level duplication however can be dealt with by discarding the detected duplicate within the analysis tool. Configuration errors cannot be dealt with directly by an analysis tool, but configuration errors are usually easily correctable.

### 2.3.1.2 Vantage Point

The position where a packet capture occurs on a network, or vantage point, is an important aspect of TCP traffic analysis. A single vantage point offers a limited perspective on network traffic, whereas multiple vantage points can use data from each packet capture to infer loss or reordering between vantage points.
If the vantage point is located at the TCP endpoint, then receipt of the packet by the receiver can be assumed, eliminating the need to infer whether a packet has been received or not. However, if the vantage point exists between the endpoints, two possible packet loss conditions can occur; packet is lost before being observed or the packet lost after being observed.

To illustrate how vantage point can affect network traffic analysis, let us discuss eight different cases. Each case represents a unique traffic analysis condition that can occur. The first four cases represent environments with two or more vantage points, and the last four cases represent environments with only a single vantage point.

Figure 2.1: VP Case 1: Two Vantage Points Located at Each TCP Endpoint

In figure 2.1, the vantage points exist at each TCP endpoint. When using two or more endpoints, the data collected can be used to infer loss between vantage points. In this specific case, the loss of sequence three can be inferred by the observation of sequence three by vantage point one, and the lack of observation by vantage point two.

In figure 2.2, the case differs from figure 2.1 in that vantage two has been moved to a position in between the endpoints. This represents the possibility that a packet can be lost
Figure 2.2: VP Case 2: Two Vantage Points, VP 1 Located at Endpoint, VP 2 Located Before Loss Occurs

after being observed without the ability to use the vantage point data to infer the loss. Other methods of inference must be used to infer the loss, such as retransmission detection.

Figure 2.3: VP Case 3: Two Vantage Points, VP 1 Located at Endpoint, VP 2 Located After Loss Occurs
In figure 2.3, the case differs from figure 2.2 in that the ability to infer loss has been retained because the packet was lost in between the vantage points. This has similar properties to the case illustrated by figure 2.1, but still retains the possibility of packet receipt ambiguity if the packet were lost between vantage point two and the receiving endpoint.

![Diagram](image)

**Figure 2.4: VP Case 4: Two Vantage Points, Both VPs Exist Between TCP Endpoints**

In figure 2.4, this case suffers the most from packet receipt ambiguity. As long as the loss occurs between the two endpoints, the loss can be inferred from the packet traces themselves. However, if the packet is lost outside of the observation window, no inferences can be made strictly using the observations.

Although more possibilities exist, figures 2.1, 2.2, 2.3 and 2.4 demonstrate general cases of behavior for multi-vantage point conditions. Multiple vantage points can be useful for correlating information to determine packet loss without using less accurate methods of retransmission detection. However, using multiple vantage points is sometimes infeasible for research as it requires multiple points in the infrastructure where
a packet capture can be taken. It is possible that either the ability to capture packets at multiple points is either physically impossible or simply impractical.

The other cases consist of environments where only a single vantage point exists. Figure 2.5 illustrates the case where the vantage point exists at the sender. When the vantage point exists at a TCP endpoint, retransmissions from the endpoint are much easier to detect as there is no possibility of loss on the network between the endpoint and the vantage point. In case 5, it would impossible to detect the loss of the packet, but it would be very easy to detect the retransmission of the packet.

![Figure 2.5: VP Case 5: One Vantage point Located at the Sender](image)

Figure 2.6 shows the case where the the vantage point exists at the receiver. In this specific case, the packet lost before being observed by the vantage. It may impossible to detect whether the packet was lost because it was never observed in the first place. However, it will be trivial to observe the acknowledgments sent from the receiver.

Figure 2.7 illustrates one of two possible scenarios where the vantage point exists between the two endpoints of a TCP connection. In this specific instance, the loss occurs after the packet been observed. In this case, the loss must be inferred by observing
Figure 2.6: VP Case 6: One Vantage Point Located at the Receiver

Figure 2.7: VP Case 7: One Vantage Point, Packet Lost After Observed

subsequent retransmissions from the sender and acknowledgment information from the receiver. This case is very similar to the case represented by figure 2.3.

Finally, 2.8 shows the case where the packet could be lost before ever being observed. Inferring the loss of the packet must be done in a similar manner as the case shown in figure 2.7.
Figures 2.5, 2.6, 2.7 and 2.8 all show cases where only a single vantage point is present. The lack of another vantage point with which to share observations makes it necessary to use other methods to infer loss or reordering on a network. Even with multiple vantage points, it may not be possible to infer loss without having a vantage point at each TCP endpoint.

Additionally, the number of vantage points has an effect on the diversity of TCP connections that can be analyzed. When using multiple vantage points, correlating data from each packet capture requires that the connection must be present in the packet captures from two of the vantage points. If the packet capture data isn’t matched with data from other vantage points, then the vantage point model can be simplified into a set of single vantage point captures.

Furthermore, the positioning of the vantage point can further increase the diversity of TCP endpoints collected in the packet captures at the risk of increasing packet loss ambiguity. Figure 2.9 represents the case where the vantage point exists at the endpoint. In this specific case, the packet capture at the vantage point would contain any connections between A, B and C if ENDP is an endpoint in any of the connections.
However, the diversity of the connections could be increased by positioning the vantage point between two sets of endpoints as shown by figure 2.10. The set of connections observed by the vantage point would include all connections where $A$, $B$ or $C$ is one of the endpoints of a connection, and $D$, $E$ or $F$ is the other endpoint of the connection.

The case represented by figure 2.10 would give a much more diverse set of connections than the case given by figure 2.9 at the risk of packet loss ambiguity.
Multiple vantage points can be used to simplify the analysis process but limits the scope of connections that can be analyzed if the analysis requires correlating the data between captures. Additionally, multiple vantage points can provide more information for analysis than a single vantage point. However, single vantage point analysis is still viable as protocol behavior can be used to infer loss in the absence of additional data from another vantage point. A single vantage point can also have a more diverse set of connections to analyze.

2.3.1.3 Stack Fingerprinting

Fingerprinting of operating systems (OS) and TCP stack implementations has been used to assist researchers in detecting and classifying retransmissions. Given knowledge of how an implementation should react, retransmissions can be classified based on observed reactions from both endpoints of a connection. Tools such as p0f[Zal06], ettercap[OV05] and nmap[Mes07] are used to provide active and passive fingerprinting. Active fingerprinting involves querying the system in question and probing for information using known action-reaction behavioral patterns for some implementations of TCP. Passive fingerprinting involves observing the packet stream and identifying quirks in the header or responses that would indicate a specific implementation of TCP. While both methods can give additional information to help analyze the connection, it can be faulty as well. Fingerprinting can be prevented by using a fingerprint scrubber[SMJ00] or blocking unsolicited inbound connections.

Fingerprinting by itself does not guarantee that an “fingerprinted” implementation of TCP will exhibit the behavior that the fingerprint says it should. Modifications to default behaviors in the system can throw off the fingerprinting process to some degree or display behaviors inconsistent with the known model. Thus, if retransmission behavior is analyzed strictly according to the model, misclassification of behavior can occur.
However, fingerprinting can be used to provide more information about what an implementation of TCP is most likely to do.

2.3.2 Prior Work

Previous studies in TCP analysis have used a variety of different approaches. Some analytical methods require manual analysis of information, a process of running an experiment and hand-verifying the results. Comer and Lin used ‘Active probing’[CL], a method of treating a TCP implementation as a black box and sending specific packets to detect TCP responses specific to a system. Brakmo and Peterson used a code simulation approach which detected errors in the BSD 4.4 TCP implementation[BP95]. A wide variety of differing approaches exist that utilize manual analysis, but manual analysis is only one of two options.

Automated analysis of TCP implementations is another approach, one which involves either active or passive analysis. Active analysis involves sending information to a TCP implementation to elicit a specific type of response. Techniques such as Active probing and OS fingerprinting are examples of Active analysis. Passive analysis relies on observing the traffic of a TCP implementation and inferring properties of a connection or implementation. Passive analysis techniques utilize captured TCP traffic to infer behavior based on known models. As this research focuses on the use of passive analysis methods to infer TCP behavior, the next sections will discuss work in passive TCP analysis.

2.3.2.1 Tcpanaly

Tcpanaly was written by Vern Paxson as an attempt at a generic passive TCP analysis tool for inspecting packet traces[Pax97b]. It was designed as a single pass, generic TCP action recognition tool for detecting aberrant behavior. However, it failed according to the author due to difficulties involving the vantage point and generic TCP action recognition problems. The tool was written to handle several different aspects of
TCP performance, such as window violations, checksum validation, gratuitous ACK validation and more\cite{Pax97b}. This ultimately failed as a single-pass analyzer because of packet filtering errors\footnote{described in section 2.3.1.1}, vantage point ambiguities and wide variations on TCP behavioral patterns.

Packet filtering errors can included dropped packets, duplication and re-sequencing of packets. Additionally, filters can omit potentially interesting messages from other protocols such as ICMP that may give information regarding the state of a connection. Tcpanaly uses several different self-consistency checks to detect dropped packets, such as acknowledging data that hasn’t been seen\footnote{It is possible, but unlikely a connection may be re-routed in a manner that causes packets to take a path that does not include the vantage point. This would cause similar behavior to dropped packets without actually dropping the packets.}. Duplicated packets are not analyzed by tcpanaly, which helps to prevent spurious triggers of retransmission methods. Packet re-sequencing is detected when the capture packet order differs from what was observed on the network. Each of these different errors has an impact on the accuracy of a single pass analyzer, which are further discussed in section 2.3.1.1.

Paxson attempted to write tcpanaly so that it could recognize generic behaviors without being aware of specific TCP implementation exceptions. It was found that it was a difficult problem because not all TCP implementations follow TCP RFCs to specification. Diverging from the TCP specification may occur because of a performance increase or logical error in the implementation. This causes complications for writing a generic model for classifying TCP behavior as there will be exceptions to specified behavior without knowing how the exception may be formed.

2.3.2.2 Tcptrace

Tcptrace is a TCP analysis tool written by Shawn Ostermann to provide detailed statistics about TCP connections\cite{Ost}. It works by reading captured packets from either a
device or saved file and analyzing each connection independently. \texttt{tcptrace} provides statistics on packet information, network reordering, TCP options, throughput, round-trip time, congestion window and abnormal packet types. While \texttt{tcptrace} is well-rounded as an analysis program, it doesn’t attempt to provide any information as to what behavior caused a retransmission. Manual analysis of the graphs provided by \texttt{tcptrace} often provide the information necessary to infer the behavior that triggered the retransmission.

2.3.2.3 \texttt{Tcpdebug}

\texttt{Tcpdebug} is a TCP analysis tool developed to classify the loss recovery method associated with retransmissions. It is passive analysis tool that was designed to account for newer TCP implementations and behaviors. Rather than attempting to classify every retransmission, the objective of \texttt{tcpdebug} was to classify the retransmissions that initiated loss recovery. Its developer’s approach consisted of implementing several partial state-machines for different TCP implementations[RS06].

Several difficulties arose during the creation of \texttt{tcpdebug}. Similar to \texttt{tcpanaly}, \texttt{tcpdebug} also had issues with undocumented behavior of TCP implementations or divergence from standardized TCP behavior. The result of this was the creation of several different programs, each intended to test a specific TCP implementation. Packet delays, losses and vantage point issues also caused further problems. Each issue was dealt with as a tentative change to the state machine and confirmed by retransmission behavior from the sender[RS06]. The author did employ a novel notion of detecting packet reordering using estimates of the minimum round-trip time.

The authors also demonstrated that another TCP analysis tool called \texttt{tcpflows} to be far too rigid. In [RKS06], the authors show that several retransmission events were missed by \texttt{tcpflows} due to the fact it adhered too strictly to the TCP behavior defined in RFCs.
Unfortunately, tcpdebug does not appear to have been maintained and has difficulties compiling on recent releases of Linux.

2.3.2.4 Tcpcsm

Tcpcsm is another TCP analysis tool designed by Shane Alcock to classify retransmission behavior[AN11]. Tcpcsm focuses on the use of the congestion window as the primary parameter of analysis, yet still runs into the same problems as previous analysis tools. Vantage point problems and variations in TCP implementations are the leading cause of analytical difficulties in tcpcsm. It uses a single state machine approach, designed to facilitate updating the tool for newer TCP implementations. It appears to be less vulnerable to timing variations by using information from newer TCP options such as timestamps[JBB92] and SACK[MMF+96]. It also provides additional retransmission classifications for loss recovery methods that were not present in the past such as F-RTO[AN11].

The tool tcpcsm is an improvement over tcpdebug, as it accounts for newer TCP implementations and makes several refinements. It handles variations between implementations by using basic fingerprinting. In comparison to tcpdebug, tcpcsm has fewer unknown events because of the ability to handle variations between TCP implementations. Tcpdebug classifies RTO behavior by the negation of all detected retransmission methods. This has a side affect of potentially identifying unrecognized retransmissions as RTO based congestion events.
3 Motivation

There are several reasons why one might study the behavior of a protocol. The protocol might be behaving erratically which may cause deterioration of performance or reliability. The protocol may react unfavorably to unforeseen events such as congestion, delays or loss of data. Analyzing the responses of the protocol to the events it observes is key to understanding how the protocol reacts. Once it is understood how the protocol reacts, new algorithms that improve or optimize existing behavior can be developed. Existing algorithms can be improved by understanding how they behave in different network environments.

3.1 Analysis Methods

There are many methods to analyzing the behavior of a protocol and two types of analysis. Manual analysis of network protocol data often relies observations gathered through live traffic capture, debugging information and performance metrics. Automated analysis of protocols involves similar techniques which employs tools such as tcptrace to determine how a protocol behaved. The combination of the two types of analysis allow us to understand protocol behavior and develop new algorithms to govern new responses to unfavorable conditions.

3.2 Protocol Efficiency

Some protocols strive for reliability at the cost of performance, or vice versa. Others attempt a blend of the two or optimize for special network traffic patterns. Transport Control Protocol, or TCP, is a reliable, in-order delivery stream protocol that strives for reliable delivery which attempting to achieve optimal throughput in a variety of different environments. TCP attempts to provide reliable delivery of data. If data is lost in transit between the sender and receiver, retransmission of the data is attempted. Some attempts at
loss recovery have only served to exacerbate deteriorating conditions within the network [Jac88]. Given that some forms of loss recovery can decrease performance, it becomes important to fully understand how loss recovery can be done effectively and efficiently.

When recovering from losses, it is important to understand the network conditions and how the protocol should respond to them. As there are many different types of network conditions, there are a multitude of different responses for each condition. If the network is congested, limiting the packet injection rate may serve to decrease congestion for all connections that share a network path. If a network path is lossy, redundancy may serve to increase effective throughput. By understanding how the reactions we instill into a protocol affect the network state, both locally and globally, we can increase the effectiveness of the protocol itself.

3.3 Loss Recovery

The efficiency of loss recovery can be measured by several different metrics, such as retransmissions count, elapsed recovery time and effective throughput during loss recovery. Each metric has a different effect on the state of the network. Attempting to minimize the retransmission count during loss recovery may prolong the elapsed recovery time beyond a reasonable amount. However, attempting to minimize elapsed recovery time naively can result in catastrophic congestion along the network path. A balance between each metric is necessary to achieve an optimal solution to loss recovery. Not every metric takes network fairness into account.

Network fairness is the concept that every connection has an equal opportunity to use the capacity along the path. It quickly becomes difficult to maintain fairness while also attempting to minimize the effects of loss, but a balance is possible. An example of a protocol feature intended to promote fairness is the exponential retransmit timer back-off.
algorithm [Jac88] in TCP which prevents the protocol from injecting new packets. While the introduction of the back-off algorithm is widely viewed as necessary, some contend that the algorithm could be removed with minimal repercussions [MK08].

Loss recovery mechanisms can differ greatly in how they affect the protocol. Some mechanisms attempt to maintain network fairness\(^4\) and others attempt to govern the retransmission of lost data\(^5\). The former are important for ensuring that the injection rate of data is reasonable for a network path. The latter are important for understanding how the protocol responded to the loss of data. Many researchers have delved into understanding how TCP reacts to various forms of loss using various tools. Some of these tools only analyze the connection with enough detail to detect the presence of retransmission and reorder, but not enough to tell what kind of retransmission took place.

Knowing that a certain protocol behavior has taken place is just as important as knowing what type of response generated it. In modern TCP implementations, there are multitudes of different heuristics which govern retransmission behavior. A retransmission timeout (RTO) is a retransmission that is sent in response to the expiration of the retransmission timer. Additionally, a retransmission could also be in response to the receipt of a specific number of duplicate acknowledgments in the case of the fast retransmit algorithm. Each heuristic has a different network traffic pattern that will trigger it and cause a retransmission in response. Without knowing which loss recovery heuristic was responsible for the retransmission, it is likely impossible to know how the connection recovered from loss other than knowing it did retransmit the lost data.

Fewer analytical options are available when we lack the understanding to classify the retransmission behavior that is observed. Some analytical models can still be used without knowing which specific retransmission heuristic is present, such as throughput analysis and coarse grained retransmission analysis. Coarse grained retransmission analysis is

---

\(^4\) Exponential retransmit timer back-off

\(^5\) Fast retransmit [Jac88]. Early Retransmit [AAA+10] and more.
differentiating retransmission timeouts from other forms of loss recovery and can be done with duplicate acknowledgment counting. When counting duplicate acknowledgments, the retransmission classification heuristic would assume a constant value of duplicate acknowledgments observed before the retransmission. While this method may be true in the majority of cases, it lacks other details that are relevant to differentiating RTO-like behavior from non-RTO-like behavior. Coarse grained analysis would provide information as to whether a connection was using alternative forms of loss recovery more frequently, but not which type. This could be problematic because differing traffic patterns could trigger vastly different responses within each connection. The differing responses then add noise to the analysis that make it difficult to determine whether new behavior was the cause of diminishing RTO-like behavior or perhaps just a symptom.

Most modern TCP implementations use several different loss recovery heuristics, such as Fast retransmit, Early retransmit and Forward Acknowledgment based recovery\cite{AAA10, APB09, MM96}. Each variation has a different approach to loss recovery which is designed to respond to a specific pattern of network traffic. In the case of Early Retransmit, the size of the outstanding data and number of selectively acknowledged packets has a direct effect on when the heuristic will retransmit upon observing a duplicate acknowledgment. This information can be used to attempt to fit retransmissions to loss recovery heuristics based on observable information. Knowing which loss recovery heuristic is responsible for the retransmission can be helpful in designing new heuristics as the new models will typically account for conditions that the older heuristics did not account for.

\footnote{They do not necessarily implement each algorithm, but most implementations have one or more of the above mentioned loss recovery heuristics}
3.4 Network Reordering

The problem of classifying retransmissions becomes more difficult once network reordering has been introduced. Network reordering is an observable condition where a packet has arrived at the destination at some point later than a subsequently transmitted packet from the same connection. Reordering affects loss recovery because TCP generates a duplicate acknowledgment for each packet that is received out of correct order. Heuristics such as Fast Retransmit rely on duplicate acknowledgments to detect loss, and thus reordering can spuriously trigger loss recovery. Reordering also adversely affects retransmission classification as a retransmitted packet also shares the property of being a packet that has arrived out of order.

Detecting reordering is non-trivial without observing both sides of the connection at the hosts themselves. While it is possible to observe the connection at two points and determine loss and reordering in that fashion, it is impossible without cooperation of two separate entities. This limitation makes it difficult to use this model of analysis when only one observation point is possible. A single observation point can be used in lieu of two points, however inferences must be drawn for what can be observed.

For the single observation point approach, two different scenarios arise which add their own complications to analysis. The first scenario, endpoint positioned capture, is where the packets are captured at one of the endpoints of a connection. This limits the observable set of connections because one of the endpoints will always be the host where the endpoint is set. Since the packets are being captured directly on the host, further packet loss on the network is not possible. The second scenario, intermediate positioned capture, is the case where packets are being captured at some point in between two distinct sets of hosts. This has the benefit of expanding the number of observable connections in a shorter amount of time, increasing the traffic pattern diversity. However,
since the packets are captured at a point between the two connections, reordering and loss
can occur at any point after the packets have been observed.
4 Method

Bro-IDS is a traffic analysis framework originally developed by Vern Paxson for network security monitoring[Pax99]. Tcprs uses this framework to reconstruct the TCP flow for analysis of retransmission behavior. Bro-IDS is a real-time online and offline bulk traffic analysis tool, which breaks up each packet and analyzes each layer of the packet with an analyzer specific to each layer. Tcprs hooks into the Bro-IDS framework at the transport layer for analyzing each individual TCP flow that is registered and read by the Bro-IDS framework.

Bro-IDS as a framework makes generic analysis much more simple by abstracting away from layers that you do not intend to analyze. With other tools, the author must understand each and every sub-layer for the analyzer to function properly. With Bro-IDS, the list of understood transport, link and frame layer protocols is extremely extensive. Additionally, bro-ids supports several different methods of encapsulation. The result is that many more diverse trace files can be read by the Bro-IDS framework.

Tcprs analyzes each packet and uses complex heuristics to determine the nature of each packet in a given TCP flow. Round-trip time measurement, retransmission behavior classification, retransmission classification, reordering classification and congestion state determination are among the functions of the tcprs analyzer.

Tcprs makes the assumption that it exists between two nodes and captures data about each endpoint separately.

4.1 Classification Taxonomy

The taxonomy given in figure 4.1 depicts the classification hierarchy used in the design of tcprs. Previously given classification taxonomies partition the set of out-of-order segments into three distinct subsets; reordering, retransmission, and unknown[RKS06]. While being unable to classify an event as a reordering or
retransmission does make it ambiguous to the nature of the segment, it does not make it mutually exclusive to either event. As such, the taxonomy given here changes the distinction between “unknown” segments.

An out-of-order segment is either a reordering or a retransmission. If the segment is a retransmission, sufficient distinction has been made by the analyzer to classify the out-of-order segment as such. Otherwise, if the segment has been classified as a reordering, two potential cases exist. The first case is that sufficient information exists to classify the reordering based on acknowledgment sequence analysis, timestamp analysis or gap analysis. The second case is that insufficient information exists to classify the reordering and likely is due to a lack of granularity in observed acknowledgments or timestamps, or a lack of round-trip time samples. More detail on each of the reordering heuristics is in section 4.3.2.
If the segment is a retransmission, then it then falls under two distinct subsets of retransmissions, retransmission timeout based loss recovery (RTO) and feedback based loss recovery. RTO based recovery only occurs when a TCP sender lacks sufficient feedback from the receiver to make a more informed reaction in the face of perceived loss. In the case of an RTO, a complete or sufficient lack of information is necessary to prevent the use of other more efficient methods of loss recovery. In the case of feedback based loss recovery, the TCP sender has sufficient information to make a more informed response to loss rather than the use of a RTO. Feedback based refers to all retransmissions sent upon receipt of some information from the TCP receiver. Two subsets of feedback based exist, retransmissions sent upon initiation of loss recovery and retransmissions sent as a result of the continuation of loss recovery. For example, the first set refers to the retransmission sent upon the receipt of the triple duplicate acknowledgment from the TCP receiver, in the case of Fast Retransmit. The second set refers to subsequent retransmissions that occur during loss recovery as a result of new information from the TCP receiver.

4.2 Round-trip Time Estimation

Each packet read by tcprs is potentially used as a sample for the round-trip time estimate used by the analyzer. Whether the packet is used to create a sample of the round-trip time of the connection depends on whether the packet is a retransmission of previously sent data, or potentially reordered data.

Since tcprs makes the assumption that it exists between two nodes, it is necessary to have an estimate for the round-trip time between the observer and each of the endpoints of a TCP flow. This results in two distinct round-trip time samples, which together estimate the round-trip time between the two endpoints. It becomes necessary to restrict the types of packets used for round-trip time samples. Retransmitted packets become inappropriate for samples as it is impossible to determine to which packet (original or retransmission(s))
that the ACK is responding. Retransmitted packets can be used as valid samples if and only if the acknowledgment can disambiguate the packet response with the TCP timestamp option.

The round-trip time for a sample is measured as the interval from observation of a packet to observation of its acknowledgment.

\[
RTT_{sample} = T_{acknowledgment} - T_{packet}
\]

The sample is simply an estimate for the round-trip time between the observation point and the receiving endpoint to which the data packet is destined. It is not possible to do any better than an estimate without knowledge from the receiving endpoint. The round-trip time between the receiving endpoint and the observation point where the analyzer is located is updated with the smoothed RTT formula given in RFC 2988[PA00]. Nagle gave a round-trip time estimation formula that is very similar to the formula given in RFC 2988[Nag84], using a smoothed RTT estimate. The formula given by Nagle was improved upon by Karn et al[KP91], which is very similar to the formula used by tcprrs.

The round-trip time estimate between each endpoint is following:

\[
RTT_{endpoints} = RTT_{send} + RTT_{recv}
\]

\[
RTTVAR_{endpoints} = RTTVAR_{send} + RTTVAR_{recv}
\]

This ‘inferred’ round trip time of the connection is an estimate of the actual round trip time. Jaiswal et al[JID+07] discuss the method of inferring the round-trip time of a connection by using the independent samples of the round trip time between the vantage point and the TCP endpoint.
4.3 Retransmission/Reordering Detection

A reordered packet is a packet that invalidates the expectation of the receiving endpoint that all packets containing data will have a greater sequence number than the last observed packet carrying data with exception when sequence wrap occurs. Retransmitted packets are a subset of reordered packets such that the packet contains data which has been previously transmitted from the sender to the receiver previously.

4.3.1 Retransmission Detection

Retransmitted packets are TCP packets containing data which has been previously transmitted from the sender to the receiver.

There are many different heuristics to detecting whether a reordered packet is a retransmitted packet or not. The heuristics detailed within this section can only determine if a reordering is a retransmission of previously sent data.

The first method is based on simple set classification. If a segment is observed by the analyzer, it is held in a set consisting of segments that have been observed, but not yet acknowledged. This set of outstanding data segments provides the analyzer a simple way...
of determining if a observed TCP packet is a retransmission. If the TCP packet matches a sequence range contained within the set of outstanding data, then the suspect reordering is a retransmission. Otherwise, it remains as a suspect reordering as nothing can yet be determined. The suspect reordering likely contains a range that was removed from the set when an acknowledgment was observed by the analyzer, or contains a range that is not contained within the set as it encompasses one or more of the sequence ranges in the set of outstanding data.

The second method is based on the set of outstanding data. The following rules govern this method:

**If the suspect reordering sequence range falls within any of the following criteria:**

- The suspect reordering sequence range is completely encapsulated by some sequence range of outstanding data
- The suspect reordering sequence range contains some sequence range of outstanding data
The suspect reordering sequence range contains the beginning of some sequence range of outstanding data.

The suspect reordering sequence range contains the end of some sequence range of outstanding data.

Then the suspect reordering is a retransmission by definition.

This can be more generally stated such that if an out-of-order segment contains data that was previously transmitted by TCP, it is a retransmission by definition.

The third method is based on the observed acknowledgments sent by a TCP receiver. If the suspect reordering only contains data that has been acknowledged based on a previously observed acknowledgment, then the suspect reordering is a retransmission. This is considered to be a fully previously acknowledged retransmission.
The fourth method is also based on the observed acknowledgments sent by a TCP receiver. If the suspect reordering contains data that has been previously acknowledged as well as new data, it is by definition, a retransmission. This is considered to be a partially previously acknowledged retransmission.

If any of the above methods haven’t come to some determination as to what the suspect reordering is, the analyzer then uses reordering detection heuristics to attempt to further disambiguate reorderings from retransmissions.

4.3.2 Reordering Detection

Reordered packets are TCP packets which arrive at the observation point in a different order than they were transmitted. tcprs can disambiguate reordered TCP packets by analyzing the sequence ordering of a TCP flow.

There are three heuristics in tcprs to detect reordering, if possible, in a TCP flow. The first method evaluates the TCP option timestamp within the options of a suspect packet reordering. The TCP timestamp option, if present, is extracted from the options and compared against the last observed timestamp.

The following rules determine whether the reordered packet is a retransmission, or a reordered packet:

if the timestamp of the suspect reordering is chronologically later than the oldest timestamp observed by the analyzer:
   -- suspect reordering is a retransmission
else if the timestamp of the suspect reordering is chronologically earlier than the oldest timestamp observed by the analyzer:
   -- suspect reordering is a reordering
else:
   -- observed timestamps are not granular enough to tell,
further heuristics must be applied

Examine the following possible cases:

Case 1: Reordering

host A sends segment X with timestamp Y
host A sends segment X+1 with timestamp Y+1
segment X+1 is captured at the vantage point with timestamp Y+1
segment X is captured at the vantage point with timestamp Y

Case 2: Retransmission

host A sends segment X with timestamp Y
host A sends segment X+1 with timestamp Y+1
segment X is lost before the vantage point
segment X+1 is captured at the vantage point with timestamp Y+1
host A retransmits segment X with timestamp Y+2
segment X is captured at the vantage point with timestamp Y+2

Case 3: Ambiguous
host A sends segment X with timestamp Y
host A sends segment X+1 with timestamp Y
segment X+1 is captured at the vantage point with timestamp Y
segment X is captured at the vantage point with timestamp Y

In case 1, the segment is a reordering. This can be logically inferred from the fact that the timestamp option must monotonically increase \[JBB92\]. Since segment X is observed with an older timestamp, it can be inferred that segment X was sent before segment X+1. Thus segment X is a reordering.

In case 2, the segment is a retransmission. This can be logically inferred from the same facts as case 1. Segment X is observed after segment X+1, however segment X has a timestamp that is greater than the timestamp in segment X+1. Because the timestamp value given in the TCP header must be a monotonically increasing value, this implies that segment X must have been transmitted after segment X+1. However, TCP does not transmit segments out-of-order unless the segment in question is a retransmission of previously sent data. This implies that segment X is a retransmission.

In case 3, the segment is ambiguous from the perspective of timestamp option analysis. The timestamp in segment X is equivalent to the timestamp in segment X+1, offering no way to distinguish between the two using this method. Depending upon the implementation, RFC 1323 only necessitates that the timestamp increase once per $2^{32}$ bytes transmitted by a TCP sender (PAWS) \[JBB92\]. While it can increase more frequently than that, it isn’t mandated, which can negate the usefulness of this heuristic when the timestamp increments infrequently.

The second method is analysis of the acknowledgment field present in the suspect reordering packet. The following rules determine whether the reordered packet is a retransmission, or a reordered packet:

if the highest acknowledged sequence observed by the observation
point is less than the acknowledged sequence in suspect reordering:

-- suspect reordering is a retransmission due to the monotonically increasing property of acknowledgments

else if the highest acknowledged sequence observed by the observation point is greater than the acknowledged sequence in the suspect reordering:

-- suspect reordering is a reordering

else:

-- comparison of acknowledgment data is insufficient to draw a conclusion

Figure 4.6: Retransmission Detection via ACK Analysis

Examine the following possible cases:

Case 1: Reordering

host A sends segment X with acknowledgment Y
host A sends segment X+1 with acknowledgment Y+1
segment X+1 is captured at the vantage point with acknowledgment Y+1
segment X is captured at the vantage point with acknowledgment Y

Case 2: Retransmission
host A sends segment X with acknowledgment Y
host A sends segment X+1 with acknowledgment Y+1
segment X is lost before the vantage point
segment X+1 is captured at the vantage point with acknowledgment Y+1
host A retransmits segment X with acknowledgment Y+2
segment X is captured at the vantage point with acknowledgment Y+2

Case 3: Ambiguous

host A sends segment X with acknowledgment Y
host A sends segment X+1 with acknowledgment Y
segment X+1 is captured at the vantage point with acknowledgment Y
segment X is captured at the vantage point with acknowledgment Y

Before delving into each case, it should be noted that this heuristic is nearly identical to the timestamp analysis heuristic. The main difference lies in that the ACK sequence space is not explicitly declared to be monotonically increasing, yet is treated so with good reason. A TCP receiver acknowledges responsibility for the delivery of segments that it has acknowledged the positive receipt there of. The renege of the acceptance of responsibility by the TCP receiver would cause TCP to be a non-reliable protocol as the TCP sender may discard the sequence range of data covered by a segment upon the receipt of the positive acknowledgment of that segment. Additionally, it is documented that the acknowledgment field is cumulative[WM06], further reinforcing this argument. It is implied and assumed that the acknowledgment field is assumed to be a monotonically increasing field. Sequence wrap is accounted for in tcprqs by comparing the the highest acknowledgment value with the newest acknowledgement value to determine if wrap has occurred.
In case 1, the segment is a reordering. This can be logically inferred from the assumption that acknowledgment field is monotonically increasing. Since segment X is observed with an acknowledgment less than the acknowledgment in segment X+1, it can be inferred that segment X was sent before segment X+1. Thus segment X is a reordering.

In case 2, the segment is a retransmission. This can be logically inferred from the same fact from case 1. Segment X is observed after segment X+1, however segment X has an acknowledgment that is greater than the acknowledgment in segment X+1. This implies that segment X must have been transmitted after segment X+1. However, TCP does not transmit segments out-of-order unless the segment in question is a retransmission of previously sent data. This implies that segment X is a retransmission.

In case 3, the segment is ambiguous from the perspective of acknowledgment sequence analysis. The acknowledgment in segment X is equivalent to the acknowledgment in segment X+1, offering no way to distinguish between the two using this method. This heuristic works best when data is being sent in each direction of the TCP flow, providing a continuous stream of updates to the acknowledgment field.

The important aspect to the previous two heuristics is their ability to distinguish reorderings from retransmissions without an estimate of the round-trip time.

The last method is the analysis of the gap present in the sequence space observed by the analyzer. Each time a gap in the sequence space is detected by the analyzer, gap analysis begin as analyzer records the size of the gap in the sequence space and the duration of the gap. ‘Large’ durations can indicate a retransmission is taking place while a shorter duration such as one that is less than a round trip time can indicate a reordering has taken place. The following rules determine whether the reordering is a retransmission or a reordered packet:

if the suspect reordering is contained within the sequence gap observed by the analyzer
-- if the duration between the start of the sequence gap and
the suspect reordering is less than a round trip time:
-- suspect reordering is a reordering
-- else if the duration between the start of the sequence gap
and the suspect reordering is greater than a round trip time:
-- suspect reordering is likely a retransmission

Figure 4.7: Retransmission Detection via Gap Analysis

In order for this heuristic to work, tcpr s must have an estimate of the round-trip
time. The gap itself is defined as the time-delta between the last transmission of new data
and the out-of-order segment. In the general case, network reordering occurs within less
than a single round-trip time of the connection. It is possible to even more tightly restrict
reordering to several milliseconds to microseconds from the last in-order segment. The
retransmission timeout timer is updated with the transmission of every new segment of
data, thus it is reasonable to expect a reordering would occur at any point before the
estimated retransmission timeout. However, since the retransmission timeout can vary
depending on the implementation, a more conservative approach should taken instead of
comparing against the estimated RTO timer. The RTO timer can approach the RTT with
subsequent sampling and convergence if the RTT variance is sufficiently small. A minimal
system dependent constant defined as the clock granularity should always ensure that RTO never converges to the RTT. Ultimately, this heuristic may be unusable in connections with such high amounts of loss that an estimate is never achieved. Consider each case:

**Case 1: Reordering**

RTT between host A and host B is **200ms**

RTT variance between host A and host B is **50ms**

Estimated RTO is \( SRTT + \max(G, K \times \text{rttvar}) \);

Let \( K = 4 \) as defined by standard, \( G = 250\text{HZ} \), or 4ms as observed in Linux 3.9.1.4

Estimated RTO = \( 200 + \max(4, 4 \times 50) = 400\text{ms} \)

host A sends segment X

host A sends segment X+1

segment X+1 is observed by the vantage point at \( t=1.000\text{s} \)

segment X is observed by the vantage point at \( t=1.005\text{s} \)

In this case, the sequence gap is noted by the analyzer upon the receipt of an out-of-order segment. If the out-of-order segment occurs before an entire RTT has elapsed since the last transmission of new data, this segment is a reordering based on the following reasons. As the RTT variance of a connection grows, the RTO timer diverges from the estimated RTT. As the RTT variance of a connection decreases, the RTO timer converges to the \( SRTT + G \) as the variance approaches 0[PA00]. \( G \) is defined as the clock granularity of the host operating system. Due to the chaotic nature of a network, there will some amount of variance in a connection, even if only negligible. Given that most network reordering occurs within milliseconds of the transmission of an in-order segment, defining reordering as segments that occur within the minimum of a round-trip time is
sufficient to cover the general case. Given the conditions above, with $\Delta t = 5ms$, it is impossible that this retransmission could be a RTO. It is also highly unlikely that the segment is a retransmission based on the receipt of only a single duplicate acknowledgment within such a small window of time w.r.t. RTT.

Case 2: Retransmission

If the out-of-order segment occurs after an entire RTT has elapsed since the last transmission of new data, this segment is likely a retransmission. Segments represented in this set often occur in response to the receipt of the duplicate acknowledgments generated by the TCP receiver, or the expiration of the RTO timer. Given the RTO timer is always greater than the RTT, all RTOs will exist in this subset. Additionally, retransmits occurring from initiation of feedback-based loss recovery often occur on the receipt of a duplicate acknowledgment from the TCP receiver. It takes at minimum, one RTT for the duplicate acknowledgments for newly transmitted data to be received by the TCP sender. Thus, it is likely that the out-of-order segment is a retransmission.

Case 3: Ambiguous

This case exists only if there is not a valid RTT estimate available to use this heuristic. In the absence of a RTT estimate, the analyzer is unable to determine reordering from retransmission with this method.

Potentially, this heuristic can fail in the most obscure of cases. It is possible that in the aftermath of loss recovery, the duplicate acknowledgments generated by spurious retransmissions in the absence of SACK information could trigger Fast Retransmit/Fast Recovery within a round-trip time of the detected gap. The retransmission would be inaccurately classified as a reordering based on the behavior of the heuristic. This inaccuracy can be mitigated by examining the number of duplicate acknowledgments

---

7 This remains true unless there are few samples of the RTT and each sample is not necessarily indicative of the actual RTT. This can occur at the beginning of a connection where the up-stream throughput is limited and RTT increases with respect to window size.
observed at the time when the out-of-order segment was detected. Without prior knowledge of what loss recovery methods are implemented by the TCP sender, this case becomes slightly ambiguous as to the nature of the out-of-order segment. Due to the infrequency of this case, this heuristic works for the general case and is sufficient.

If any of the above mentioned heuristics are unable to determine the type of reordering, the suspect reordering is acknowledged as an ambiguous reordering. Ambiguous reorderings are not used in RTT samples as it was not possible to determine whether the reordering was a retransmission or not.

4.4 Retransmission Classification

There are three different types of retransmissions. The first type of retransmission, initial recover retransmissions, are retransmissions that are assumed to transition a TCP endpoint from a normal operating state to a loss recovery state. This type of retransmission requires classification as it is often in response to observable pattern-like behavior that identifies some sort of recovery method initiated by the TCP sender in response to loss.

The second type of retransmission, transitionary retransmissions, are retransmissions that transition a TCP endpoint from a specific loss recovery state to a different loss recovery state when behavior observed by the analyzer does not match expected behavior during loss recovery. An example of this is a retransmission timeout during a fast retransmit loss recovery phase. This type of retransmission also requires classification as it changes the operation of the TCP endpoint in response to observable receiver behavior.

The third type of retransmission, loss recovery retransmissions, are retransmissions that occur during loss recovery that are in response to an acknowledgment of data sent by a TCP receiver. These retransmissions do not denote a change in the loss recovery, rather they only serve to show that loss recovery is progressing as expected. Classification of these retransmissions is unnecessary.
The first retransmission that triggers a loss recovery determines what is known as a *loss recovery window*. A loss recovery window is the outstanding data at the time loss recovery is initiated by a TCP sender. The boundaries of the loss recovery window are governed by the highest acknowledgment from the TCP receiver and highest sequence sent by the TCP sender observed by the analyzer at the time loss recovery is initiated. This window can be changed as new acknowledgment data is observed. The window can also be changed when the second type of retransmission is observed, causing the loss recovery window to be expanded.

Classifying retransmissions requires a significant amount of state to be kept by the analyzer for accurate predictions of the retransmission classification. In other implementations, the use of machine learning algorithms to build classifiers may be possible but are infeasible for a real-time solution. The approach taken by *tcprs* is a heuristic guided method of classification. To classify a retransmission, the following details should be known by the analyzer:

- Current recovery state of the analyzer
- Sequence number denoting the end of a recovery window
- Retransmission count of the packet in question
- Estimated RTO timeout
- Sequence number of the last retransmitted packet
- Retransmission classification of the last retransmitted packet
- Duplicate acknowledgments observed within three round trip times
- Knowledge of the use of SACK between the origination and responding TCP endpoints

...
To classify a retransmission as caused by a retransmission timeout, it must have fulfilled one of the following criteria:

1. The endpoint retransmitting the data is contained within a recovery window initiated by an earlier detected retransmission window:

   (a) If the packet has been retransmitted before:

   i. The timestamp estimate for the retransmission timeout is suspected to have expired and the packet contains the SYN flag

   ii. The last sequence to be retransmitted is the same sequence as the current retransmission and the last sequence to be retransmitted was a retransmission timeout

   (b) If it is unknown whether the packet has been retransmitted before:

   i. The timestamp estimate for the retransmission timeout is suspected to have expired and the packet is not in response to a duplicate acknowledgment

   ii. The last sequence to be retransmitted is the same sequence as the current retransmission and the last sequence to be retransmitted was a retransmission timeout

2. The retransmission contains the SYN or FIN flags

3. The retransmission is not associated with a duplicate acknowledgment

4. The retransmission is suspected to have been transmitted in response to the expiration of an estimated retransmission timeout

5. The retransmission occurred within some expected variance of the round trip time in response to the expiration of an estimated retransmission timeout
5 EXPERIMENTS, RESULT AND ANALYSIS

This chapter details the experiments performed to measure the accuracy of tcprs. In order to determine the accuracy of tcprs, we compare the results given by tcprs with other TCP traffic analysis output. Tcptrace and tcpcsm are the TCP traffic analyzers that have been chosen for comparison. Tcptrace provides retransmission and reordering event detection that should be sufficient for verifying the accuracy of retransmission and reordering event detection of tcprs. Tcpcsm provides a more detailed analysis of retransmission events, including classification by suspected retransmission method. The comparison of tcpcsm and tcprs events will provide a comparison point for the retransmission event classifications. Each analyzer will be given the same input datasets and the results will be used to estimate the accuracy of tcprs in comparison with the other tools. Time sequence graphs will be used to show events generated by tcprs and a primer for interpreting the graphs will be given in appendix A. The time sequence graph annotations provided by tcprs will be detailed in appendix B.

5.1 Datasets

Three datasets are used in the comparative analysis of tcprs, tcpcsm, and tcptrace. The Google (G) dataset consists of a single tcpdump trace file provided by Google. The Dartmouth (D) and SIGCOMM '08 (S08) datasets consist of several tcpdump trace files that yield a much larger number of connections and packet counts. The statistics for each dataset are given in table 5.1. The total number of connections is given by reported connection counts from tcptrace. capinfos, a utility provided as part of the Wireshark package, was used to gather the packet count statistics.
Table 5.1: Dataset Information

<table>
<thead>
<tr>
<th>Dataset name and capture dates</th>
<th>Connections</th>
<th>Packets</th>
<th>Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G$ (09/10/10)</td>
<td>136K</td>
<td>4.4M</td>
<td>1</td>
</tr>
<tr>
<td>$D$ (01/01/04 - 01/31/04)</td>
<td>66.8M</td>
<td>2.2B</td>
<td>1288</td>
</tr>
<tr>
<td>$S_{08}$ (8/27/08 - 8/27/08)</td>
<td>513K</td>
<td>47M</td>
<td>45</td>
</tr>
</tbody>
</table>

Table 5.2: Dataset Event Data. Percentages are given as a percent of the total packets.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Packets</th>
<th>Events</th>
<th>RTO</th>
<th>FRETX</th>
<th>REORDER</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G$</td>
<td>4.4M</td>
<td>50.4K(1.1%)</td>
<td>41.9K(0.95%)</td>
<td>4916(0.11%)</td>
<td>3525(0.08%)</td>
</tr>
<tr>
<td>$D$</td>
<td>2.2B</td>
<td>11.3M(0.51%)</td>
<td>3.35M(0.15%)</td>
<td>1.26M(0.06%)</td>
<td>6.72M (0.31%)</td>
</tr>
<tr>
<td>$S_{08}$</td>
<td>47M</td>
<td>93220(0.2%)</td>
<td>50K(0.11%)</td>
<td>16K(0.03%)</td>
<td>27K(0.06%)</td>
</tr>
</tbody>
</table>

5.2 Traffic Analysis and Instrumentation

While designing tcprs, the decision was made to use traffic dump files from traffic repositories rather than kernel instrumentation. Performing traffic analysis on bulk traffic dump files allows the analyzer to observe a much larger variety of connections than kernel instrumentation. Solutions involving kernel instrumentation limit the observable set of endpoints to connections which must talk to the host with the instrumentation. Kernel instrumentation allows a direct mapping of packets to observable events without any doubt regarding the classification of the event. However, instrumentation does not scale well as each system must have the same instrumentation within the kernel. Bulk traffic analysis solutions can read traffic dump files that have been taken at an intermediate point in the network, which increases connection diversity. For these reasons, tcprs was designed to use bulk traffic analysis rather than instrumented analysis. As a side-effect of this choice, tcprs will only be able to provide a “best-guess” classification for each event.
5.3 Sampling Process

Since tcprs does not use kernel instrumentation, the observable set of events must be reduced in order to provide a subset of events of interest. In most of the datasets, there are millions of packets and some portion of those packets will have a retransmission or reordering event associated with the packet. Fortunately, the ratio of packets to events is very high, usually less than two percent of the total packets have an event associated with them. This makes manual analysis of the packets somewhat easier, but even two percent of forty million packets is far too many to inspect by hand. So to further whittle down the number of events that should be inspected, comparative analysis can be used.

The comparative analysis involves using a companion analyzer to find a subset of the identified events that differ. We assume that analyzers that have been developed independently are less likely to influence or bias the results of a comparative analysis. If the two analyzers agree on the classification of an event, we assume that it is likely that both analyzers are correct. We also assume that the likelihood that both analyzers are correct may increase if each analyzer was developed independently of the other.

To answer one of the initial questions posed by this research, it is ideal to find an analyzer that is developed entirely independently of tcprs. An independent analyzer with a high level of detail into retransmission and reordering classification would be useful in comparing to the events given by tcprs. It was for that reason that I choose to use tcpcsm. The tcpcsm traffic analyzer is a community-maintained analyzer, developed at the University of Waikato, that provides a high level of detail into retransmission and reordering events. It was for those reasons that tcpcsm was picked as the analyzer for the comparative analysis.

In the comparative analysis, the assumption is made that when both tcprs and tcpcsm are in agreement with the classification of an event, that the classification is correct. This serves as an ‘oracle’ for the comparative analysis and reduces the number of
events that need to be inspected manually. The events that differ between the two analyzers are particularly interesting because those events highlight one of the research questions: Why do analyzers disagree on the same data?

To further reduce the number of events that should be manually inspected, the assumption that the event is correct when the analyzers agree is used. That assumption is used to eliminate those events from the set of events to inspect. Since the number of events where the two analyzers differ is usually less than thirty percent, we can eliminate an even larger number of events to sample. As the number of events to sample drops, the process of manually inspecting enough events to give a reasonable sample of the event space becomes easier.

The two analyzers, tcpcsm and tcprs, are given the same datasets and then the output of each analyzer is compared with the output of the other. The first step was to run each analyzer against each tracefile in the three different datasets. Since there are not direct mappings to all of the events in both analyzers, the events from both analyzers are classified into sets of equivalent events; RTO, Fast Retransmission(FRETX) and reordering (REORDER) events. If tcprs and tcpcsm differ within a set of equivalent events, then we look more closely at those differences.

The differing events are counted by the tcpcsm event classification as a distribution of events within the set of equivalent events. From each set of equivalent events, a sample of 100 events is taken from each tcpcsm event classification that differed. The samples are randomly chosen from all events in the tcpcsm classification within the set of equivalent events based on a python script. Each of the sampled events were manually inspected to estimate the accuracy of the classification that tcprs attributed to the event. An estimate of the accuracy of a specific classification within a set of events can be built from the number of samples that were correctly classified by tcprs and the distribution of
those tcpcsm events within the set of equivalent events. From the aggregate accuracy of a set of equivalent events, the aggregate accuracy across all events can be estimated.

Each of the samples are manually inspected for accuracy. Though personal bias could be introduced, each sample was examined as objectively as possible. The comparative analysis only serves as an estimate of accuracy of the tcprs since the ‘ground truth’ or factual accuracy of the analyzer cannot be determined without instrumentation on both sides of a connection.

5.4 Comparative Analysis

Tcpcsm[AN11], tcpdebug[RS06], and tcptrace[Ost] are TCP traffic analysis tools that provide connection information about specific TCP connections. To validate the classification results given by tcprs, we compare the results of tcprs against each of these tools to understand where tcprs may give better results than what was previously available with each tool.
Table 5.3: Feature Comparison

<table>
<thead>
<tr>
<th>Generic Features</th>
<th>tcptrace</th>
<th>tcpdebug</th>
<th>tcpcsm</th>
<th>tcprs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Congestion State Detection</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Retransmission classification</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Reordering classification</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>DSACK</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>F-RTO</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Recovery Classifications</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Generic loss recovery</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>RTO</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Fast Retransmit</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Sack based recovery</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>F-ACK based recovery</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Early Retransmit</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

The classification done by tcprs should give similar results to these tools but improve in the detection of retransmissions versus reorderings, classification of retransmissions in the generic case and provide classification for newer methods of loss recovery such as Early Retransmit [A.A.A + 10]. TCP debug is omitted from the comparative analysis because the tool requires deprecated versions of tcpdump to function properly and requires extensive modifications to the source code to compile for an up-to-date version of gcc. In addition, event data for connections will be omitted if the three-way handshake was not observed or less than two RTT samples were obtained. Less than two RTT samples can lead to improper classification of retransmission events in the case where the samples were obtained during periods of high congestion where the RTT observed is not representative of the average RTT of the connection. The minimum of two
RTT samples was chosen due to the necessity to include one-way TCP traffic into analysis. A one-way TCP connection will transmit data in one direction only and receive only acknowledgments in the other direction. Thus, in a one-way connection, it is only possible to observe two RTT samples, one from the three-way handshake (SYN/SYN-ACK/ACK) and one from the FIN portion of the TCP state machine (FIN/FIN-ACK/ACK). While tcprs can infer the use of optional features such as SACK and timestamps, omission of these connections is recommended for the sake of accuracy. If the three-way handshake takes place without being observed, either traffic is being routed around the observation point, or the connection was in progress prior to the start of packet capture. The first case is problematic due to lack of information and potentially inaccurate classification of events if asymmetric routing persists. The second case is only problematic because it is impossible to distinguish it from the first case. If the three-way handshake has not been observed, the MSS can only be estimated, which may lead to inconsistent estimates of the window and inaccurate classification of Early Retransmit triggered loss recovery. Additionally, the SACK_PERMITTED and TIMESTAMP options can only be inferred from subsequent packets and may not be accepted in both directions of the connection.

In this section, tcprs results will be individually compared with tcptrace and tcpcsm.

5.4.1 TCPTRACE

tcptrace is accepted by the TCP community as an accurate analytical tool for TCP traffic. While tcptrace provides a wealth of information about various statistics of a connection, it does not provide classifications about retransmissions for comparison against tcprs. In the comparative analysis between tcprs and tcptrace, we focus on retransmission and reordering statistics given by each tool. In tcptrace, the only values
given are the retransmission count and reordering count, for both directions of a given TCP flow.

In the results from the comparative analysis, a discrepancy was noted in the number of connections detected by tcprs and tcptrace. tcptrace detects fewer connections than tcprs due to the way that TCP connection boundaries are determined. tcptrace uses a conservative approach to connection boundaries that reduces the number of TCP connection flows by carefully observing new flows and traffic patterns. The approach taken by tcptrace is not used in the Bro-IDS framework. The resulting difference of connection counts between the two analysis tools has a subtle impact upon the out-of-order count given by each.

An example of such impact upon out-of-order count can be observed in figure 5.1. In this trace, seven FIN/ACK packets are transmitted, with six of them as retransmits. tcptrace treats this connection as a single connection and counts each retransmission of the FIN/ACK correctly. However, Bro-IDS breaks the connection into six different TCP connections, and thus tcprs only counts a single retransmission. One connection contains two FIN/ACK segments and the other five connections only contain a single FIN/ACK segment. This difference accounts for much of the discrepancy in the total out-of-order count between the two analysis tools. In this example, tcptrace is correct since the retransmissions occur at expected intervals.

A relatively large discrepancy was also noted in the counts of detected network reordering between the two tools. tcptrace takes the approach that any segment that has not been previously observed and is out-of-order is considered to be reordered. While this statement can be true, it is also false in some cases. Consider the following case:

host A sends segment X
host A sends segment X+1
segment X lost before observation point
segment X+1 is observed by the observation point
host A retransmits segment X

In the above given case, segment X is considered to be a reordered segment by definition in tcptrace. However, we know that the segment itself was a retransmission.

An example of such a case is given below in figure 5.2. In this example, the round-trip time is sufficiently small, such that reordering should have been expected directly after the first observed data segment. However, several RTTs later, the lost segment is retransmitted via an RTO. It is highly likely that this segment is indeed a retransmission based on what is known about the RTT and retransmission timeout estimate. In this example, the RTT is approximately 49ms at the time the segment is retransmitted. The elapsed time since the last new segment of data is roughly 279ms. This segment is several RTTs out-of-order and very likely is not a network reordering. Based off the estimated RTO timer, tcpres classifies this retransmission as a RTO.
Figure 5.2: Connection with a False Reordering

Another example of this behavior is given in figure 5.3. In this figure, we are observing an instance of fast retransmit. The lost segments are classified by tcptrace as out-of-order segments. In tcprs, these same segments are classified as retransmissions.

As noted in section 4.3.2, there are several methods that can determine whether an out-of-order segment is a retransmission without having to observe the segment when it was first transmitted. This difference allows tcprs to be more accurate in cases such as this.

To find the cases in which the two analyzers differ, it was necessary to collect the statistics of each individual connection. After the initial collection was complete, the statistics were aggregated into host-port pair records, after which determining their difference would become trivial. The criteria for a differing record is solely based on whether the count of the out-of-order events (retransmission or network reordering) differ between records. In general, if a difference existed, there was a shift from count of
reorderings versus retransmissions between the two analyzers. In special cases, such as when `tcptrace` detected a continuation of a connection that `tcprs` determined to be a new connection, the total count of reorderings often differed by the difference in connection counts for a given host-port pair.

It was noticed in this analysis that it isn’t sufficient to classify an out-of-order packet based on initial observation alone. Examples can be found in the comparative analysis between `tcptrace` and `tcprs`, where in many cases, the round-trip time analysis performed by `tcprs` gives important information for distinguishing retransmissions from reorderings. The primary difference in event counts between these two analyzers comes from the application of RTT information as a heuristic for distinguishing potentially ambiguous reordering.
Table 5.4: Tcptrace Events by Dataset and Type. Additionally, connection counts are given to show differences in connection boundary detection.

<table>
<thead>
<tr>
<th>Event type</th>
<th>$G$</th>
<th>$D$</th>
<th>$S_{08}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Detected connections</td>
<td>135053</td>
<td>66776937</td>
<td>154535</td>
</tr>
<tr>
<td>Total events</td>
<td>119471</td>
<td>44701035</td>
<td>203250</td>
</tr>
<tr>
<td>Retransmissions</td>
<td>114956 (96.22%)</td>
<td>33695317 (75.38%)</td>
<td>140343(69.05%)</td>
</tr>
<tr>
<td>Reorderings</td>
<td>4515 (3.78%)</td>
<td>11005718 (24.62%)</td>
<td>62907(30.95%)</td>
</tr>
</tbody>
</table>

Table 5.5: Tcprs Events by Dataset and Type. Additionally, connection counts are given to show differences in connection boundary detection.

<table>
<thead>
<tr>
<th>Event type</th>
<th>$G$</th>
<th>$D$</th>
<th>$S_{08}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Detected connections</td>
<td>141318</td>
<td>136795322</td>
<td>171218</td>
</tr>
<tr>
<td>Total events</td>
<td>116782</td>
<td>33163786</td>
<td>169028</td>
</tr>
<tr>
<td>Retransmissions</td>
<td>112938 (96.56%)</td>
<td>25337044(76.40%)</td>
<td>143197(84.72%)</td>
</tr>
<tr>
<td>Reorderings</td>
<td>3844 (3.34%)</td>
<td>7826742(23.60%)</td>
<td>25831 (15.28%)</td>
</tr>
</tbody>
</table>

Table 5.6: Differences Between tcptrace and tcprs by Dataset and Event Type

<table>
<thead>
<tr>
<th>Event type</th>
<th>$G$</th>
<th>$D$</th>
<th>$S_{08}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Detected connections</td>
<td>6265 (4.43%)</td>
<td>70018385 (51.18%)</td>
<td>16683 (10.80%)</td>
</tr>
<tr>
<td>Total events</td>
<td>2689 (2.30%)</td>
<td>11522477 (34.75%)</td>
<td>34222(16.84%)</td>
</tr>
<tr>
<td>Retransmissions</td>
<td>2018 (1.78%)</td>
<td>8348547 (32.94%)</td>
<td>2854(2.03%)</td>
</tr>
<tr>
<td>Reorderings</td>
<td>671 (17.46%)</td>
<td>3173930 (40.55%)</td>
<td>37076(58.94%)</td>
</tr>
</tbody>
</table>
5.4.2 TCPCSM

When comparing the different classes of congestion events, some types will be omitted due to the ambiguity of classification that they represent by definition. An example of such an omission is the collection of events that strictly represent retransmissions and not a subset of loss recovery events such as tcpcsm’s UNNEEDED and RETX_NEW events. Retransmissions classified by such events do not provide insight as to what loss recovery method elicited the retransmission. Instead, we believe that they demonstrate an attribute of the retransmission. We only consider the following set of events as potential matches for loss recovery comparative analysis; RTO, FRETX, MS_FRETX, SACK_FRETX, BAD_FRETX, LOSS_REC, FREC, UNEXP_FREC, REORDER, LINK_DUP, and IPID_DUP.

Other events as defined by the authors of tcpcsm may not reflect an loss recovery event such as a retransmission via fast retransmit, retransmission timeout, early retransmit and other potential loss recovery heuristics. As such, those events are excluded from the comparative analysis.

We include LOSS_REC, FREC and UNEXP_FREC events as the tcpcsm and tcprs may differ with respect to events that begin and end loss recovery. LINK_DUP and IPID_DUP events are also included as differences between the two analyzers with respect to detecting duplicate packets may be of interest. If one of the analyzers classifies the suspect duplicate packet as initiating loss recovery, then it is likely an error if they differ in classification.

REORDERING events are compared against all other events that could be considered a retransmission for the purpose of creating an exclusive subset of out-of-order events that represent network reordering. Similarly, the same is done with RTO events to create a subset of out-of-order events that represent retransmissions sent in response to the expiration of the retransmission timeout timer. FRETX, MS_FRETX, SACK_FRETX, and
BAD_FRETX compose a set of out-of-order events that represent retransmissions sent in response to feedback in the form of cumulative, selective, and duplicate selective acknowledgments from the receiver.

RTO events consist of the set of congestion events that are triggered by a distinct lack of response or sufficient level of response from the receiver necessary to elicit a more informed loss recovery response.

Fast retransmission events consist of the set of congestion events that are triggered by the presence of sufficient levels of response from the receiver necessary for more informed loss recovery responses.

Additionally, for the purposes of comparing reordering events, RETX_OVERLAP, RETX_NEW, and UNNEEDED tcpcsm events will be included as they differentiate network reordering from retransmission events. They are not included in the RTO and Fast Retransmission analysis as they cannot be used to disambiguate loss recovery methods.

In the following sections, I will discuss the cases in which tcpcsm and tcprs differed. Each subsection will cover a set of cases in which the two analyzers differ. Each case within the subsection will discuss why the two analyzers gave different results, referring to the associated figure to provide visual reference to the case. The Dartmouth(D) dataset is used for generating comparative analysis samples as it is the largest dataset and should contain the most diverse set of observable retransmission and reordering events. Additionally, after generating the events, neither the Google dataset nor the SIGCOMM dataset contained enough events in the subclassifications to provide a reasonable sampling.

5.4.2.1 Reordering Event Detection

Tcprs uses three different methods for detecting network reordering events: TCP timestamp comparison, ACK comparison and sequence gap detections.
TCP timestamp comparison uses the TCP timestamp option to record the echo and echo-reply values observed in each TCP packet. If a packet is observed with an echo value older than a previously observed packet, the packet must be a reordering because the echo value is a monotonically increasing value [JBB92].

ACK comparison uses the ACK field of the TCP header to record the highest ACK observed so far. If a packet with an older ACK value is observed, it must be a reordering because a cumulative acknowledgment cannot be reneged [Pos81].

Sequence gap detection is a heuristic for determining whether an event is a retransmission or network reordering based on the RTT information of a connection. A TCP packet that is reordered by the network is typically observed within a fraction of an RTT of the packet it was reordered with. If an event matches that criteria, it is likely a network reordering event.

In figure 5.4, tcprts detected the event as a reordering due to the TCP header timestamp option. This event occurs in the past as the TCP timestamp value would imply that it was sent prior to previously observed segments that have a more recent TCP timestamp value. A ‘more recent’ timestamp means that the monotonically increasing timestamp value was larger than previously observed value with due consideration for sequence wrap. This implies that this segment is a reordering as it was sent before another segment but received later. It is clear that this data is also a retransmission of previously unseen data based on the difference in time between the segment and other segments that are close to this event in the sequence space.

Now looking at figure 5.5, the difference in time between the detected gap in the sequence space and the event is much shorter than the round trip time. This would imply that the segment is more likely to be a reordering than a retransmission. Additionally, since a fast retransmit of a segment is in response to an acknowledgment, the only
potential triggering acknowledgments are too close in proximity in terms of time to the event to be potential triggers.

In figure 5.6, another case of network reordering can be observed. The ACK field within the TCP header contains a sequence that was transmitted before another segment containing a ‘later’ ACK sequence. A ‘later’ ACK sequence is a sequence number relative to another sequence number that acknowledges data that occurs later in the stream. The prior knowledge implies that the segment that is observed in figure 5.6 is a network reordering rather than a retransmission.
Figure 5.5: Case: Sequence Gap Based Reordering Detection

Figure 5.6: Case: TCP Header Acknowledgment Based Reordering Detection
5.4.2.2 Retransmission Timeout Event Detection

In figure 5.7, a simple retransmission case is observed. In this case, we see that the suspect reordering has been classified as a ‘BRUTE_FORCE_RTO’. This means that the data observed is at the bottom of the window and that the last data retransmitted was also this same segment. An RTO sends the segment at the bottom of the window, so this event matches the profile of a RTO as well. Further, by looking at the round-trip time information, none of the observed acknowledgments could have triggered the event, so it is likely that this is a retransmission timeout.

Figure 5.7: Case: Repeated Retransmissions via Retransmission Timeout

In figure 5.8, we see that the suspect reordering has been classified as an RTO. This happened because the round-trip time data doesn’t allow the analyzer to match any of the acknowledgments to the suspect reordering event. This makes it more likely that the event was retransmitted in an unsolicited fashion, something common to a RTO. Without
matching it to an acknowledgment and the fact that the event already occurs within a loss recovery window, it is likely this event is a RTO.

Figure 5.8: Case: Unsolicited Retransmission via Retransmission Timeout

Figure 5.9 shows a simple RTO event. As observed in the figure, there are three duplicate acknowledgments. However, based on the RTT information, the suspect reordering event doesn’t appear to occur as a response to any of the acknowledgments. Furthermore, the suspect reordering event occurs several RTTs after the cumulative acknowledgment – something that is typical of RTO behavior.

Figure 5.10 shows a case that tcprs classifies as a RTO that was classified by tcpcsm as a link-layer duplicate. Link-layer duplicate packets have several properties which make detecting them somewhat easy. Link-layer duplicates often occur within a fraction of the RTT of the packet that it duplicates and the duplicate packet contains the exact same link-layer headers and payload data. The example shown in figure 5.10 shows both identifying features. The TCP header differs from the original packet’s TCP header.
and the event occurs several RTTs after the original packet. See section 6.1.3 for future work with the heuristics.

Figure 5.11 shows a case where the suspect reordering lies outside of the detected connection boundaries for `tcpcsm`. In this example, the FIN segment is a retransmitted FIN segment. Given that `tcpcsm` does not generate an event for this segment, it is likely that `tcpcsm` either ignores packets after the initial FIN, or does not generate events for this specific type of event. Regardless, the estimated RTO matches the elapsed time between the retransmission and the previous FIN segment, making it likely that this is a RTO.
Figure 5.10: Case: Retransmission Timeout from Incorrect Link Layer Duplicate Classification

Figure 5.11: Case: Retransmission Timeout Detected Outside of Implemented Connection Boundaries
5.4.2.3 Fast Retransmit-like Event Detection

In figure 5.12, tcpr classifies the reordering event as a fast retransmission. This is due to the fact that the analyzer determined the first loss recovery window has closed and is treating this new reordering event as the start of a new loss recovery window. Since this is the first retransmission in the window, and it appears to emulate the behavior typical of a fast retransmission, tcpr has classified it as such. tcpsm likely classified this as a fast recovery due to differences in how the loss recovery window is handled.

Figure 5.12: Case: Retransmission Based on Fast Retransmit-like Behavior Following a Closed Loss Recovery Window

Figure 5.12 shows a case where the suspect reordering is a retransmission due to loss recovery. The classification here by tcpr is correct based on the outstanding window at the time of the first loss. The loss recovery window is initialized to the outstanding data window at the time of the loss. Since the data was recovered prior to the reordering event, tcpr would assume that a loss recovery had exited and congestion avoidance would
resume. The reordering event here appears to follow the pattern expected of a typical fast-retransmit-based loss recovery – a retransmission after observing three duplicate acknowledgments.

![Sequence Number vs Time Graph]

**Figure 5.13:** Case: Retransmission Based on Fast Retransmit-like Behavior

In figure 5.14, `tcpr` detects this event but `tcpcsm` does not. Based on the available information about the connection’s RTT and the time the event occurred, the classification is likely to be correct. The event is the retransmission of data on sequence boundaries that haven’t been observed yet in the trace.
Figure 5.14: Case: Retransmission Based on Fast Retransmit-like Behavior Involving Coalesced Packets

5.4.2.4 Analyzer Design Errors

Figure 5.15 shows another case of odd behavior by the protocol that was not anticipated. The event in this example appears to be reordered if you look at the last data segment that was sent which wasn’t out of order. There are two other reordered segments preceding the event, the first of which is likely to be a retransmission timeout. The fact that the event is not a reordering could be inferred from the neighboring segments and events. Additionally, the difference in time between the events and neighboring acknowledgments would suggest that the event is in response to an acknowledgment. Since the event appears to be in response to an acknowledgment, it is more likely that the event is a retransmission via loss recovery.

In figure 5.16, the suspect reordering has been misclassified as an RTO. This is due to the way that information is retained by tcprs. Tcprs only retains the data for a few
Figure 5.15: Case: Misclassification of Event Due to Failure to Properly Classify Preceding Events

round-trip times after the data has been acknowledged. This is an attempt to prevent too much memory from being allocated during analysis. Unfortunately, due to that feature, the information necessary to show what type of event this was is lost. The event in this figure is most likely a retransmission via loss recovery rather than an RTO. The retransmission directly proceeding the event is more likely to be an RTO if the elapsed time between the initial transmission and the retransmission is considered.

Figure 5.17 shows a suspect reordering that was reordered with respect to another retransmission. The later segment is the original segment sent as an RTO and the event classified as an RTO is actually a simple retransmission via loss recovery that was reordered by the network.

Figure 5.18 shows a case where the loss recovery labeling heuristic fails. The figure shows a simple example of a network reordering, but tcpr has incorrectly classified the event. Tcpr does not rely on the transmission of the lowest sequence of a window to start
Figure 5.16: Case: Misclassification of Event Due to Reordered Retransmissions at the Beginning of Loss Recovery

Figure 5.17: Case: Misclassification of Event Due to Data Discard Policy in tcprS
loss recovery but instead labels the first event it finds that would be part of a loss recovery window. This is done with the expectation that if loss recovery begins and it didn't begin with the segment at the bottom of the loss recovery window, that segment was likely lost and it won’t be observed until later. However, by setting the classification based on the first observed segment in loss recovery, the most likely reason for loss recovery isn’t lost, even if the event itself is misclassified. See section 6.1.3 for future work with the heuristics.

![Time sequence graph](image)

**Figure 5.18:** Case: Misclassification of Event Due to Event Labeling Policy in tcprts

### 5.4.2.5 Retransmission Timeout Detection Errors

In figure 5.19, we see a case where the suspect reordering event was classified as an RTO but was classified incorrectly. The round-trip time estimate in this trace shows the round-trip time to be approximately 1.44 seconds in length. Although not displayed, two retransmission timeouts occurred in the same loss recovery window before the reordering event, causing the retransmission timeout estimate to grow twice. Retransmission timeout
estimation is based on the equation provided by Allman et al [APB09], which sets the estimate to a fraction of the round trip time multiplied by some constant. In this case, the retransmission timeout estimate was over 4 seconds in length, which is longer than the elapsed time between the cumulative acknowledgment and the suspect reordering event. In this case, it seems unlikely that this is an RTO, especially if the last observed acknowledgment is considered. The last observed acknowledgment is a selective acknowledgment that is very close to the round-trip time in length from the suspect reordering. This makes it more likely that the suspect reordering is a simple retransmission of data during loss recovery and not a retransmission timeout.

![Time sequence graph](image)

Figure 5.19: Case: Misclassification Due to Bad RTO Estimate

Figure 5.20 shows a case where `tcprs` fails to detect a link layer duplicate. In this specific case, the link layer duplicate check fails because the retransmitted link layer ID was not retained. If `tcprs` had inspected each prior observed IP ID, then it would have been trivial to detect that the event was a link layer duplicate. However, in this case, it
should not have been flagged as a retransmission timeout as the RTT information would have shown that the event occurred far too close in proximity to the previous transmission of the segment. This is further supported by the RTO estimate as well.

Figure 5.20: Case: Misclassification Due to Bad RTO Estimate of False Link Layer Duplicate

Figure 5.21 shows a case of unexpected protocol behavior. The suspect reordering which tcpr is classifies as a RTO occurs in response to the arrival of a selective acknowledgment. This implies that the event is more likely to be a fast retransmit-like behavior rather than a RTO. Tcpr is doesn’t classify this event properly because there was only a single duplicate acknowledgment preceding the event. Thus, without further information and the assumption that such behavior does not follow the RFC, tcpr is classifies the event as a RTO.

Figure 5.22 shows a case where it is important to consider the round-trip time estimates of a connection before blindly assuming that a lack of data means the event was a RTO. In this case, the event happens within a fraction of the round-trip time of the
cumulative acknowledgment. Looking closely at the trace, `tcprs` is observing the data at a point closest to the sender, meaning that the acknowledgment should occur within close proximity to any responses the acknowledgments will trigger. Based on that information, the round-trip time between the endpoint and the observation point shows that the event is in response to the cumulative acknowledgement. Retransmission timeouts are not triggered in response to the receipt of new information, so it becomes more unlikely that this is a RTO. However, the fast retransmit-based heuristics fail because there are no duplicate acknowledgments to associate with the retransmit, causing the analyzer to fall back to a RTO as most likely choice. This was misclassified because `tcprs` assumed that the lack of duplicate acknowledgments was sufficient to classify an event as an RTO without considering RTT information.
5.4.2.6 Reordering Detection Errors

When tcprs attempts to classify a suspect reordering, it only attempts to classify the reordering as either a "network reordering" or a "retransmission". This causes all reorderings to be put into two distinct sets. This is a oversimplification by the analyzer because although rare, some retransmissions can also be network reorderings. In the appendix E, several traces are provided for observation. Tcprs is designed to classify events by using an ordered set of heuristics, stopping when one heuristic positively identifies the event. This can cause events that are both a retransmission and a network reordering to only be partially classified. For these events, we would expect that they are both a reordering and a retransmission based on the information, but tcprs only produces a single event by design. The design of tcprs is biased toward detecting retransmissions more than network reorderings, thus there may be more events where the reordering classification was missed rather than the retransmission classification. An analysis of the

Figure 5.22: Case: Misclassification Due to Failure to Consider Cumulative Acknowledgments
missed events from the reordering classification amounted to approximately 7.875% of the reordering samples.

In figure 5.23, tcprs fails to classify the event properly due to a design flaw in the analyzer. Tcprs uses a sequence gap analysis heuristic for determining whether a segment is a retransmission or a network reordering when it is detected to be out-of-order. Each time a new gap in the sequence space is discovered, the old gap information is discarded. This causes the observed event to be detected as a reordering because the difference in time between the last detected gap and the new event is much less than a round trip time, which would imply the packet would be a reordering in the common case.

![Sequence number vs. time graph showing misclassification](image)

Figure 5.23: Case: Misclassification Due to Unexpected Secondary Retransmission Classification

### 5.4.2.7 Fast Retransmit-like Detection Errors

Figure 5.24 shows a case where tcprs incorrectly classifies the reordering event as a fast transmission. The classification is incorrect because the new recovery window is
instantiated based on the assumption that the sender has seen all of the acknowledgment data. The acknowledgment and retransmission information implies that not all of the acknowledgments have been received. The SACK information present in the acknowledgments suggests that part of the retransmitted data has already been acknowledged by the receiver.

![Time sequence graph](image)

Figure 5.24: Case: Misclassification Due to Unreceived Acknowledgment

Figure 5.25 shows an instance where tcpr is has incorrectly labeled the event due to unreceived data. The preceding segment was incorrectly classified as a reordering due to inaccurate RTT information. Prior to this event, the round trip time had changed significantly. However, the RTT changes were likely only short lived, which caused the RTT estimates to be inaccurate while classifying the event.

Similarly to the figure 5.26, tcpr is has incorrectly classified this event. The reordering event is classified as the start of a new fast retransmit based recovery window by tcpr is. The reordering event exists in a well defined loss recovery window, which
Figure 5.25: Case: Misclassification Due to RTT changes

would negate the possibility of the event being the start of a new loss recovery window. This is likely a mistake in the logic for determining the end of a loss recovery window.

Figure 5.26: Case: Misclassification Due to Deviations from Expected Loss Recovery Window Behavior
5.5 Results

Table 5.7: Distribution of Sampled FRETX Events by Dataset with Estimated Accuracy. Column labels represent the datasets from which the events came and the row labels represent the tcpcsm classification. Undetected events are those that were not detected by tcpcsm.

<table>
<thead>
<tr>
<th></th>
<th>Accuracy</th>
<th>$G$</th>
<th>$D$</th>
<th>$S_{08}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aggregate</td>
<td>93.8% ± 1.3%</td>
<td>4916</td>
<td>1261819</td>
<td>16142</td>
</tr>
<tr>
<td>FREC</td>
<td>37% ± 9.5%</td>
<td>8(0.16%)</td>
<td>6970(0.55%)</td>
<td>593(3.7%)</td>
</tr>
<tr>
<td>LINK_DUP</td>
<td>76% ± 8.4%</td>
<td>2780(57%)</td>
<td>5562(0.44%)</td>
<td>49(0.3%)</td>
</tr>
<tr>
<td>LOSS_REC</td>
<td>19% ± 7.7%</td>
<td>17(0.35%)</td>
<td>2406(0.19%)</td>
<td>67(0.42%)</td>
</tr>
<tr>
<td>REORDER</td>
<td>62% ± 9.5%</td>
<td>1(0.02%)</td>
<td>51809(4.1%)</td>
<td>125(0.77%)</td>
</tr>
<tr>
<td>RTO</td>
<td>53% ± 9.8%</td>
<td>72(1.5%)</td>
<td>91627(7.3%)</td>
<td>999(6.2%)</td>
</tr>
<tr>
<td>Undetected</td>
<td>75% ± 8.5%</td>
<td>12(0.24%)</td>
<td>17028(1.3%)</td>
<td>5(0.03%)</td>
</tr>
<tr>
<td>Undetected(TCPRS)</td>
<td>0%</td>
<td>1 (0.02%)</td>
<td>328(0.03%)</td>
<td>49(0.3%)</td>
</tr>
</tbody>
</table>

Table 5.4.1 shows that there are relatively few differences between tcptrace and tcprs in terms of event classification. This is ideal as this implies that both tools are in agreement as to what constitutes a retransmission and what does not. The differences between tcptrace and tcprs are shown in figures 5.1, 5.2 and 5.3. The difference of connection boundaries is a difference in implementation between the Bro-IDS framework and the maintainers of tcptrace. Tcprs and tcptrace differ the most in terms of classifying events as network reordering events. Figures 5.2 and 5.3 show cases where the RTT information would imply the event is not a reordering.

When comparing tcprs and tcpcsm, tables 5.7, 5.8 and 5.9 show that the tools differ to a larger extent than tcprs and tcptrace. The comparative analysis of tcprs
Table 5.8: Distribution of Sampled RTO Events by Dataset with Estimated Accuracy. Column labels represent the datasets from which the events came from and the row labels represent the \texttt{tcpcsm} classification. Undetected events are those that were not detected by \texttt{tcpcsm}.

<table>
<thead>
<tr>
<th></th>
<th>\textbf{Accuracy}</th>
<th>\textbf{G}</th>
<th>\textbf{D}</th>
<th>\textbf{S_{08}}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aggregate</td>
<td>97.9% ± 0.98%</td>
<td>41941</td>
<td>3346058</td>
<td>49962</td>
</tr>
<tr>
<td>BAD_FRETX</td>
<td>98% ± 2.0%</td>
<td>122(0.25%)</td>
<td>5990(0.02%)</td>
<td>14(0.03%)</td>
</tr>
<tr>
<td>FREC</td>
<td>94% ± 4.7%</td>
<td>43(0.09%)</td>
<td>47468(1.4%)</td>
<td>95(0.19%)</td>
</tr>
<tr>
<td>FRETX</td>
<td>76% ± 8.4%</td>
<td>154(0.32%)</td>
<td>111738(3.3%)</td>
<td>750(1.5%)</td>
</tr>
<tr>
<td>LINK_DUP</td>
<td>19% ± 7.7%</td>
<td>157(0.33%)</td>
<td>6166(0.18%)</td>
<td>63(0.13%)</td>
</tr>
<tr>
<td>LOSS_REC</td>
<td>65% ± 9.4%</td>
<td>197(0.42%)</td>
<td>19053(0.57%)</td>
<td>67(0.13%)</td>
</tr>
<tr>
<td>REORDER</td>
<td>74% ± 8.6%</td>
<td>8(0.01%)</td>
<td>27094(0.81%)</td>
<td>100(0.2%)</td>
</tr>
<tr>
<td>SACK_FRETX</td>
<td>48% ± 9.8%</td>
<td>85(0.18%)</td>
<td>18059(0.54%)</td>
<td>121(0.24%)</td>
</tr>
<tr>
<td>Undetected</td>
<td>98% ± 2.7%</td>
<td>19159(41%)</td>
<td>542575(16%)</td>
<td>25422(51%)</td>
</tr>
<tr>
<td>Undetected(TCPRS)</td>
<td>0%</td>
<td>1(0.01%)</td>
<td>941(0.02%)</td>
<td>8 (0.02%)</td>
</tr>
</tbody>
</table>

and \texttt{tcpcsm} focuses on differences in subclassification of events, so it is expected to see more differences here than in the comparison of \texttt{tcptrs} and \texttt{tcptrace}. It is expected that the tools should agree with each other in the majority of cases as a larger difference might imply the existence of an implementation error in one of the analyzers.

Table 5.10 shows that the percent of events that had an equivalent classification in \texttt{tcptrs} and \texttt{tcpcsm} could fluctuate between each dataset. The difference in percent of events can attributed to a difference in observed events between data sets. The Google(\textit{G}) dataset contains a substantial amount of FREETX events that were classified as LINK_DUP events by \texttt{tcpcsm}. The SIGCOMM(\textit{S_{08}}) dataset contains a substantial amount of
Table 5.9: Distribution of Sampled REORDER Events by Dataset with Estimated Accuracy. Column labels represent the datasets from which the events came from and the row labels represent the \texttt{tcpasm} classification. Undetected events are those that were not detected by \texttt{tcpasm}.

<table>
<thead>
<tr>
<th></th>
<th>Accuracy</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( G )</td>
<td>( D )</td>
<td>( S_{08} )</td>
<td></td>
</tr>
<tr>
<td>Aggregate</td>
<td>99.9% ± 0.08%</td>
<td>3525</td>
<td>6719482</td>
<td>27116</td>
</tr>
<tr>
<td>FREC</td>
<td>87% ± 6.6%</td>
<td>4(0.11%)</td>
<td>13383(0.2%)</td>
<td>1943(7.2%)</td>
</tr>
<tr>
<td>FRETX</td>
<td>92% ± 5.3%</td>
<td>3(0.09%)</td>
<td>14356(0.21%)</td>
<td>1616(6%)</td>
</tr>
<tr>
<td>LOSS_REC</td>
<td>95% ± 4.3%</td>
<td>4(0.11%)</td>
<td>9828(0.15%)</td>
<td>71(0.26%)</td>
</tr>
<tr>
<td>RTO</td>
<td>97% ± 3.4%</td>
<td>48(1.4%)</td>
<td>47041(0.7%)</td>
<td>383(1.4%)</td>
</tr>
<tr>
<td>SACK_FRETX</td>
<td>87% ± 6.6%</td>
<td>28(0.79%)</td>
<td>2012(0.03%)</td>
<td>339(1.3%)</td>
</tr>
<tr>
<td>Undetected</td>
<td>93% ± 5%</td>
<td>13((0.36%)</td>
<td>28570(0.43%)</td>
<td>75(0.28%)</td>
</tr>
<tr>
<td>Undetected(TCPRS)</td>
<td>0%</td>
<td>2(0.06%)</td>
<td>12943(0.19%)</td>
<td>49(0.19%)</td>
</tr>
</tbody>
</table>

Table 5.10: Percentage of Events that had the Same Classification in \texttt{tcpasm} and \texttt{tcprs}

<table>
<thead>
<tr>
<th></th>
<th>( G )</th>
<th>( D )</th>
<th>( S_{08} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTO</td>
<td>41.21%</td>
<td>86.10%</td>
<td>88.61%</td>
</tr>
<tr>
<td>FRETX</td>
<td>57.89%</td>
<td>77.74%</td>
<td>46.70%</td>
</tr>
<tr>
<td>REORDER</td>
<td>97.19%</td>
<td>98.28%</td>
<td>83.67%</td>
</tr>
</tbody>
</table>

REORDER events that were classified as either FREC or FRETX events by \texttt{tcpasm}.

Although the percentage of events with an equivalent classification differs from dataset to dataset, the impact of the difference is dependent on how the events that differ were classified by \texttt{tcpasm}.
Table 5.11: Estimated Percentage of Correct Classifications Made by tcprs by Event Type. Estimates are based on results of the comparative analysis on the random samples generated from each sub-type matching. The column headings represent the generic top level classification from the comparative analysis and the row labels represent the tcpcem event labels. "NA" entries did not have any events from tcpcem for the top-level classification.

<table>
<thead>
<tr>
<th></th>
<th>FRETX</th>
<th>RTO</th>
<th>REORDER</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAD_FRETX</td>
<td>NA</td>
<td>99%</td>
<td>NA</td>
</tr>
<tr>
<td>FREC</td>
<td>31%</td>
<td>94%</td>
<td>84%</td>
</tr>
<tr>
<td>FRETX</td>
<td>NA</td>
<td>76%</td>
<td>92%</td>
</tr>
<tr>
<td>LINK_DUP</td>
<td>37%</td>
<td>19%</td>
<td>NA</td>
</tr>
<tr>
<td>LOSS_REC</td>
<td>19%</td>
<td>65%</td>
<td>91%</td>
</tr>
<tr>
<td>REORDER</td>
<td>62%</td>
<td>74%</td>
<td>NA</td>
</tr>
<tr>
<td>RTO</td>
<td>53%</td>
<td>NA</td>
<td>95%</td>
</tr>
<tr>
<td>SACK_FRETX</td>
<td>NA</td>
<td>48%</td>
<td>88%</td>
</tr>
<tr>
<td>undetected</td>
<td>75%</td>
<td>98%</td>
<td>97%</td>
</tr>
</tbody>
</table>

Events that have the an equivalent classification in both analyzers are assumed to be correct. To support the assumption, a random sampling of each event type was performed and manually inspected for accuracy. In each case, the result of the random sampling show that classification was correct in every sample where tcpcem and tcprs gave the same equivalent classification.

To derive an estimate of the accuracy and precision of tcprs, the results of the comparative analysis are used to build a set of aggregate precision and accuracy measurements. The accuracy and precision estimates are limited to the top-level classifications; RTO, FRETX, and REORDER. The estimated percentage of events that had the correct classification as given by tcprs are given in table 5.11.
Table 5.12: Estimated Percentage of Correct Classifications Made by tcprs by Event Type. Estimates are based on results of the comparative analysis on the random samples generated from each sub-type matching. The column headings represent the generic top level classification from the comparative analysis and the row labels represent the tcpcsm event labels. "NA" entries did not have any events from tcpcsm for the top-level classification.

<table>
<thead>
<tr>
<th></th>
<th>FRETX</th>
<th>RTO</th>
<th>REORDER</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAD_FRETX</td>
<td>NA</td>
<td>0%</td>
<td>NA</td>
</tr>
<tr>
<td>FREC</td>
<td>55%</td>
<td>2%</td>
<td>63%</td>
</tr>
<tr>
<td>FRETX</td>
<td>NA</td>
<td>16%</td>
<td>39%</td>
</tr>
<tr>
<td>LINK_DUP</td>
<td>4%</td>
<td>63%</td>
<td>NA</td>
</tr>
<tr>
<td>LOSS_REC</td>
<td>72%</td>
<td>35%</td>
<td>85%</td>
</tr>
<tr>
<td>REORDER</td>
<td>16%</td>
<td>23%</td>
<td>NA</td>
</tr>
<tr>
<td>RTO</td>
<td>35%</td>
<td>NA</td>
<td>9%</td>
</tr>
<tr>
<td>SACK_FRETX</td>
<td>NA</td>
<td>0%</td>
<td>39%</td>
</tr>
<tr>
<td>undetected</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
</tbody>
</table>

Table 5.13: Estimated Accuracy by Top-Level Comparison Event Type and Dataset for tcprs.

<table>
<thead>
<tr>
<th></th>
<th>$G$</th>
<th>$D$</th>
<th>$S_{08}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>aggregate accuracy</td>
<td>95.47% ± 0.5%</td>
<td>97.98% ± 0.49%</td>
<td>95.27% ± 0.5%</td>
</tr>
<tr>
<td>RTO</td>
<td>98.52%</td>
<td>96.93%</td>
<td>97.56%</td>
</tr>
<tr>
<td>FRETX</td>
<td>63.39%</td>
<td>91.70%</td>
<td>83.50%</td>
</tr>
<tr>
<td>REORDER</td>
<td>99.65%</td>
<td>99.67%</td>
<td>98.05%</td>
</tr>
</tbody>
</table>
Table 5.14: Estimated Precision by Top-Level Comparison Event Type and Dataset for tcprs

<table>
<thead>
<tr>
<th>Event Type</th>
<th>$G$</th>
<th>$D$</th>
<th>$S_{08}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>aggregate precision</td>
<td>99.67%</td>
<td>98.98%</td>
<td>98.62%</td>
</tr>
<tr>
<td>RTO</td>
<td>99.93%</td>
<td>98.66%</td>
<td>99.03%</td>
</tr>
<tr>
<td>FRETX</td>
<td>97.10%</td>
<td>96.52%</td>
<td>95.49%</td>
</tr>
<tr>
<td>REORDER</td>
<td>99.93%</td>
<td>99.60%</td>
<td>99.72%</td>
</tr>
</tbody>
</table>

Table 5.15: Estimated Accuracy by Top-Level Comparison Event Type and Dataset for tcpcsm

<table>
<thead>
<tr>
<th>Event Type</th>
<th>$G$</th>
<th>$D$</th>
<th>$S_{08}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>aggregate accuracy</td>
<td>59.32% ± 0.33%</td>
<td>89.69% ± 0.3%</td>
<td>64.69% ± 0.31%</td>
</tr>
<tr>
<td>RTO</td>
<td>57.94%</td>
<td>76.38%</td>
<td>47.31%</td>
</tr>
<tr>
<td>FRETX</td>
<td>45.57%</td>
<td>81.25%</td>
<td>76.43%</td>
</tr>
<tr>
<td>REORDER</td>
<td>96.85%</td>
<td>97.89%</td>
<td>89.70%</td>
</tr>
</tbody>
</table>

\[
\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \quad (5.1)
\]

\[
\text{Precision} = \frac{TP}{TP + TN} \quad (5.2)
\]

Accuracy is defined by equation 5.1 and precision is defined by equation 5.2. $TP$ represents the number of events that were ‘True Positive’ indicators. $FP$ represents the number of events that were ‘False Positive’ indicators. $TN$ represents the number of
Table 5.16: Estimated Precision by Top-Level Comparison Event Type and Dataset for tcpcsm

<table>
<thead>
<tr>
<th></th>
<th>$G$</th>
<th>$D$</th>
<th>$S_{08}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>aggregate precision</td>
<td>98.25%</td>
<td>96.76%</td>
<td>94.65%</td>
</tr>
<tr>
<td>RTO</td>
<td>99.67%</td>
<td>96.21%</td>
<td>95.93%</td>
</tr>
<tr>
<td>FRETX</td>
<td>83.64%</td>
<td>86.07%</td>
<td>82.97%</td>
</tr>
<tr>
<td>REORDER</td>
<td>99.80%</td>
<td>97.76%</td>
<td>99.26%</td>
</tr>
</tbody>
</table>

events that were ‘True Negative’ indicators and $FN$ represents the number of events that were ‘False Negative’ indicators. The estimated accuracy and precision of the top-level classifications are calculated directly from the estimated correctness given by the random sampling results of the comparative analysis. The aggregate accuracy and precision is calculated from accuracy and precision values of the top-level classifications after applying weights to values as determined by the distribution of events given in tables 5.7, 5.8, 5.9, 5.10 and 5.11. The results of the accuracy and precision calculations can be observed in tables 5.13 and 5.14.

The accuracy and precision estimates are based wholly on the manual inspection of results from the comparative analysis using the Dartmouth dataset. Since any event can be both a retransmission and a network reordering, the design of tcpr is introduces two different biases. The classification bias in tcpr is based on the assumption that an event could be classified as a retransmission before it could be evaluated by network reordering detection methods. Additionally, if the event is classified as a reordering by the ACK sequence and TCP timestamp reordering detection methods, it is possible the retransmission aspect of the event could be missed if the event would be classified as a retransmission by the gap sequence detection method. Since the retransmission bias is
introduced before the reordering bias, it is likely that more network reordering exist than are reported by \texttt{tcprs}.

Care should be taken when interpreting the results for the accuracy and precision for \texttt{tcpcsm}. The accuracy and precision measurements for \texttt{tcpcsm} are heavily influenced by the amount of undetected events. The undetected events are primarily SYN and FIN events, the latter which does not influence loss recovery to great extent. The primary cause behind undetected events in \texttt{tcpcsm} is the difference in connection boundaries as an implementation choice. If measured without the undetected SYN and FIN events, there would less difference in precision and accuracy than there is currently.

Though there is bias, \texttt{tcprs} is generally more accurate than \texttt{tcpcsm} for the same datasets. \texttt{Tcpcsm} is better than \texttt{tcprs} in detecting link layer duplicate and simple loss recovery packets. This is mostly due to a design defect in \texttt{tcprs} in retaining data which is explained in section 5.4.2.4. Additionally, \texttt{tcpcsm} is better at detecting FRETX type events than \texttt{tcprs} in the general case, with exceptions in the case where \texttt{tcpcsm} classified the event as a REORDER or failed to detect the event. However, \texttt{tcprs} is better in most other cases, with an emphasis in reordering detection. On average, \texttt{tcprs} is also noticeably better at correctly classifying RTO events with exceptions in the cases where \texttt{tcpcsm} classified the event as a LINK_DUP or SACK_FRETX event. There is little distinction between \texttt{tcpcsm} and \texttt{tcprs} in the case of a FRETX classified as an RTO by \texttt{tcpcsm} or the case of an RTO classified as a SACK_FRETX by \texttt{tcpcsm}.

In general, \texttt{tcprs} would provide more accurate results for data sets which do not experience excessive numbers of link layer duplicate packets that arrive after the data has already been acknowledged, or mostly composed of unidirectional connections. The lack of RTT samples in both directions directly affects the accuracy of \texttt{tcprs}. 
6 Conclusions

In section 1.4, three different questions were raised about TCP traffic analysis; Why do different TCP traffic analyzers provide different results for the same input files? How do these different results affect congestion control analysis results? Lastly, is it possible to create a new TCP traffic analyzer that gives more accurate results?

TCP traffic analyzers differ in terms of the level of detail, interpretation of TCP standards, and implementation design choices. Some analyzers provide a high-level view of a connection by information on a per-connection basis, such as tcptrace, and others provide information on a per-packet basis such as tcpcsm, tcprs, and tcpdebug. Designers of TCP traffic analyzers might interpret standards differently, which may result in different results between analyzers. Our tool, tcprs, follows the standard for initializing and closing loss recovery windows, but this rigidity may cause analysis errors as some TCP implementations may continue loss recovery beyond the initial loss recovery window, as seen in Figure 5.12. Tcpcsm allows for a more flexible definition of the loss recovery window which allows it to classify events more accurately than tcprs in special cases. Lastly, implementation design choices for the traffic analyzers have side effects on the analysis that is not immediately apparent. The two common cases observed are differences in connection boundaries and the length of time that is data is retained. Tcpcsm, tcprs and tcptrace differ in terms of connection boundaries and length of time that data is held by the analyzer. Tcpcsm has difficulty detecting SYN and FIN retransmissions due to the definition of connection boundaries as chosen by the maintainers of tcpcsm. The definition of a connection and the boundaries of a connection for tcprs are defined by the maintainers of Bro-IDS, which is not always correct as can be seen in figure 5.1. The length of time that a traffic analyzer retains data has a direct impact on accuracy. Tcprs may misclassify link-layer duplicate packets if they arrive after the cumulative acknowledgment, a problem that doesn’t appear in tcpcsm or
tcptrace. If the data had been retained for a longer period of time, detecting the event as a link-layer duplicate would have been trivial.

The comparative analysis of tcpcsm, tcptrace and tcprs shows estimated accuracy and precision improvements in tcprs over both tcptrace and tcpcsm. There are cases in which tcprs fails to make the correct classification, but in general, the classification made by tcprs is correct more often than the classification made by the other two analyzers. Tcpcsm has better estimated accuracy in some cases where tcprs and tcpcsm differ, such as link layer duplicate detection. However, the aggregate accuracy of tcprs is better due to increased accuracy in the general cases. Tcprs is more accurate and precise in all data sets in the top-level classifications as well as the overall precision and accuracy as compared with same values from tcpcsm. The difference in accuracy between tcprs and tcpcsm ranges between less one percent up to 44%. The estimated overall accuracy of tcprs ranges from 95.27% (S₀₈) up to 97.98% (D). The estimated overall accuracy of tcpcsm ranges from 59.32% (G) up to 89.69% (D). In terms of fewer classification errors, tcprs makes between 5 to 9 times fewer errors than tcpcsm on the same data sets. Based on the estimations, it is fair to say that tcprs is a success in the attempt to make a TCP traffic analyzer that is more accurate.

6.1 Future Work

6.1.1 OS/TCP Fingerprinting

There is considerable potential in the use of OS fingerprinting to identify host TCP congestion control algorithms. Many operating systems use a specific congestion control algorithm by default⁸. This information can be used to model the changes to ssthresh and cwnd, such that the estimation of ssthresh is more accurate. As the estimation for ssthresh and cwnd becomes more accurate, determining the congestion state of connection

⁸ Windows Vista uses Compound TCP by default, and many flavors of Linux currently use Cubic
becomes easier. When the AI/MD behavior of the connection is known in advance, the analyzer should be able to model the expected changes in window size with respect to the expected congestion control algorithm. Thus, congestion states are more accurately modeled when this information can be taken from the OS fingerprint.

Of note, this type of modeling can fail when expectations of the operating system deviate from the default behavior. Care should be taken to not allow the default to completely dominate the ssthresh calculations and allow non-standard ssthresh calculations when behavior does not match expected behavior.

6.1.2 Elimination of Retransmission Classifications Based on Empirical Evidence

Experimental retransmission methods such as Early Retransmit[AAA+10] are not implemented widely. Many operating systems do not explicitly reveal what congestion avoidance or loss recovery methods are not implemented. Care should be taken to determine if an experimental behavior could have triggered and if it did not, then it is probably not implemented. If the previous case is true, then future attempts at categorizing retransmissions with that heuristic should be disabled or scored much lower by the analyzer to reduce the existence of false positives.

6.1.3 Heuristics

Many of the failed classifications in the results were due to a heuristic or set of heuristics that could not fully account for the behavior observed. Some new heuristics could be added or simple modifications to existing heuristics could be made to account for the failed classifications in the results. For example, figure 5.18 in section 5.4.2.4 displays a case where a simple modification to the heuristic to account for the elapsed time since the last cumulative acknowledgment would likely address the incorrect classification.
6.1.4 Modified Reordering Taxonomy

In section 5.4.2.6, it was noticed that a relatively small percentage of network reordering events could be additionally classified as retransmissions. An assumption was made in section 4.1 that network reorderings and retransmissions were distinct subsets of reordered segments. However, in the results, this was proven to be false as there was clear evidence of network reorderings that were also retransmissions. While the assumption was made as a simplification, it remains true that the original taxonomy doesn’t account for the notion that there is some overlap between network reorderings and retransmissions. The overlap, while relatively small, has some implications for counting events. Some events may be lost when treating the two subsets as distinct subsets, as they were treated in this thesis. Future iterations of this analyzer will need to take this into account so that potentially valuable retransmission events are not lost to overlap cases in the taxonomy used here.
REFERENCES


APPENDIX A: TIME SEQUENCE GRAPHS

Time sequence graphs are graphs generated by tcptrace that show the sequence progression of a connection over time with annotations to show specific events. A brief introduction to interpreting time sequence graphs and the annotations provided by tcptrace will be given in this appendix.

Time sequence graphs show the progression of the sequence space of a connection over time while displaying key events that occurred during the observation of the connection. The X-axis represents time and the Y-axis represents the sequence space of the connection. The time sequence graph is described by many different features, as given below:

Green line
- current acknowledgment point of the connection

Yellow line
- current receive window as advertised from other endpoint

Green tick under line
- duplicate acknowledgment event

Yellow tick above line
- duplicate window advertisement event

White arrow
- A sent segment. The up arrow corresponds to the last byte of the segment and the down arrow corresponds to the first byte of the segment

Red arrow with 'R' above

---

9 The size of the window can be calculated as the difference in the sequence space between the yellow line and green line at the same point in time.
a retransmitted segment. The up and down arrows have the same representation as the white arrows.

**SYN**

a SYN (synchronize) segment

**FIN**

a FIN (finished/finalize) segment

**RST_IN/RST_OUT**

a RST (reset) segment. RST_IN is the label used for RST segments in the time sequence graph for the opposite direction and RST_OUT is used for RST segments in the original direction

'x' or little cross

segments with 0 data payload

Purple line with S above

SACK blocks representing the sequence space that has been selectively acknowledged at the time of the event

White line with diamond at top and arrow at bottom

a segment with a PUSH flag

White line with red 'U' above

a segment with the URG flag carrying urgent data

**P**

a window probe

**Z**

zero window size advertisement
segment received out of order

segment is a hardware duplicate

acknowledgment is the third duplicate acknowledgement observed

SYN and FIN events occur at the time and sequence number where the event was observed. A SYN event can be observed in figure A.3. A FIN event can be observed in figure A.4. SACK blocks can be observed in figure A.5. All other events can be observed in the remaining figures within this section.

Figure A.1: Sample Time Sequence Graph[Ost]
Figure A.2: Zoom of Sample Time Sequence Graph[Ost]. Features of the tcptrace annotations can be more easily seen.

Figure A.3: Zoom of Sample Time Sequence Graph at the Beginning of a Connection[Ost]. In this figure, the annotations for segments, the SYN segment, acknowledgment line, window advertisement line, retransmissions and duplicate acknowledgments can be seen.
Figure A.4: Zoom of Sample Time Sequence Graph at the End of a Connection[Ost]. In this figure, the annotations for segments, the FIN segment, acknowledgment line, window advertisement line, retransmissions and RST_IN can be seen.

Figure A.5: Time Sequence Graph Showing SACK Blocks[Ost]
Figure A.6: Time Sequence Graph Showing PUSH Segments

Figure A.7: Time Sequence Graph Showing URG Segments
Figure A.8: Time Sequence Graph Showing Zero Window Advertisements[Ost]
APPENDIX B: TCPRS ANNOTATIONS

This appendix details the annotations specific to TCPRS and explains what each annotation means. The time sequence graphs in this thesis have been annotated using the retransmission types from the logs with a python script, ‘annotate’. The annotations specific to TCPRS are the following: ‘UNKNOWN’, ‘RTO’, ‘FAST_3DUP’, ‘FAST_SUSPECT’, ‘EARLY_REXMIT’, ‘REXMIT’, ‘TESTING’, ‘NO_RTT’, ‘NO_TS’, ‘SEGMENT_EARLY_REXMIT’, ‘BYTE_EARLY_REXMIT’, ‘SACK_SEGMENT_EARLY_REXMIT’, ‘SACK_BYTE_EARLY_REXMIT’, ‘SACK_BASED_RECOVERY’, ‘BRUTE_FORCE_RTO’, ‘RTO_NO_DUP_ACK’, ‘FACK_BASED_RECOVERY’, ‘reordering’, and ‘ambiguous’. The ‘TESTING’, ‘NO_RTT’, and ‘NO_TS’ events do not occur unless a development build has been enabled, so they will not be discussed here.

The description for each event is given below:

UNKNOWN

an event that TCPRS was unable to classify

RTO

a retransmission timeout event

FAST_3DUP

a retransmission event representing the initiation of loss recovery by detection of fast retransmit after observing three duplicate acknowledgments

FAST_SUSPECT

a retransmission event representing the initiation of loss recovery by detection of fast retransmit that recognizes non-triple duplicate acknowledgment cases

EARLY_REXMIT
a retransmission event representing the initiation of loss recovery by detection of early retransmit

REXMIT

a retransmission event in response to the receipt of acknowledgment that is part of loss recovery, but not the initiation of loss recovery

SEGMENT_EARLY_REXMIT

a retransmission event representing the initiation of loss recovery by detection of segment-based early retransmit

BYTE_EARLY_REXMIT

a retransmission event representing the initiation of loss recovery by detection of byte-based early retransmit

SACK_SEGMENT_EARLY_REXMIT

a retransmission event representing the initiation of loss recovery by detection of segment-based early retransmit using SACK information

SACK_BYTE_EARLY_REXMIT

a retransmission event representing the initiation of loss recovery by detection of byte-based early retransmit using SACK information

SACK_BASED_RECOVERY

a retransmission event representing the initiation of loss recovery by detection of SACK-based recovery

BRUTE_FORCE_RTO

a retransmission timeout event of a retransmission that has been previously retransmitted before by a retransmission timeout
RTO_NO_DUP_ACK
    a retransmission timeout event that had no preceding duplicate acknowledgments to
    indicate any other type of retransmission event

FACK_BASED_RECOVERY
    a retransmission event representing the initiation of loss recovery by detection of
    FACK-based recovery

reordering
    a reordering event

ambiguous
    a network reordering that could not be determined to be either a retransmission
    event or reordering event based on observed data
APPENDIX C: COMPARATIVE ANALYSIS SCRIPTS

C.1 tcptrace event equivalence scripts

To generate the tcptrace output for a given file, the following command is necessary:

```
tcptrace -nl [tracefile] > [tracesummary]
```

where `tracefile` is the name of the tcpdump trace to generate data for and `tracesummary` is the name of the summary file to save the output to.

The `tracesummary` file is given as the argument to `tcptrace_summary.sh` to process the summary data given by `tcptrace` and give the raw numbers for comparison.

The `tracesummary` file is also used to generate the connection equivalence lines used in comparing the results between the `tcptrace` and `tcprs`. The `tcptrace_records.py` script reads the `tracesummary` file in from standard in, allowing the output of tcptrace to be piped into the tool instead of saved to file if desired. The input of the script is based on `tcptrace` version 6.6.7 output for the `-l` flag. The output of the `tcptrace_records.py` script is a set of tuples of the following format:

```
# <src_h> <src_p> <dst_h> <dst_p> <src_rexmt> <dst_rexmit> <src_reord> <dst_reord>
```

The `tcprs_records.py` script is similar in that it outputs tuples of the exact same format as `tcptrace_records.py`. However, the expected input of the script is the `tcpreordering.log` and `tcpretransmissions.log` given by the `tcp_analysis.bro` policy file. The policy file is included with `tcprs` as part of the distribution of the analyzer in Bro.

It is recommended that when doing the comparisons, the output of each script should be sorted so as to reduce tuple reordering problems when using a diff tool. Additionally, as mentioned in section 5.4.1, `tcptrace` draws connection boundaries at different places
than the Bro-IDS framework. This will cause some reported connections to have a
different source-destination host/port pair. Ideally, to match these with the tcptrace output,
comparisons should be done to see if the source and destination host/port pairs can be
swapped to match an tuple in the tcprs tuple results or vice versa.

It should be noted that by the tcptrace record tuples may contain connection tuples
that report zero retransmissions and zero reorderings. This is different from the tcprs
record tuples in that only connections with retransmission or reorderings will be reported.
Care should be taken in the analysis to omit record tuples from tcptrace_records.py output
before comparing the differences between the tools.

Two cases exist if the tuple exists in the tcptrace records and not the tcprs
records. The first case is that record does not exist in tcprs due to difference in TCP
connection boundaries. The second case occurs when tcprs does not detect any
retransmissions or reorderings for a given connection, thus no record would exist in either
tcpretransmissions.log or tcpreordering.log. No tuple would ever be created by the
tcprs_records.py script in the second case.

An example of the above steps is given on the next page.
"example script to compare events and behavior"

tracefile = some_trace

# run each program

tcptrace -nl ${tracefile}.dmp > ${tracefile}.tcptrace_output
bro -Cr ${tracefile}.dmp policy/protocols/tcp/tcp_analysis

# process output to create tuples for comparison

tcptrace_records.py < ${tracefile}.tcptrace_output | sort -s > ${tracefile}.tcptrace_records
tcpsrs_records.py tcpretransmissions.log tcpreordering.log | sort -s > ${tracefile}.tcprs_records

# post-process host/port pairs for optimal matching and resort

## I recommend a python script for this step. A simple script that
## reads in each tool's tuples, stores them in a dictionary and
## attempts an optimal matching for non-matched host port pairs will
## be sufficient. For the traces used in this work, the host port
## pairs were segregated nicely into two distinct network prefixes
## such that it was sufficient to use awk to do the post-processing
## rather than a more elaborate method.

# This is an example from one of the traces used in the comparative
analysis

for recordfile in $tracefile.tcptrace_records $tracefile.tcprs_records
do
# if the connection tuple begins with the 10.* prefix, leave
# the record alone

grep "# 10." > $recordfile.notswapped

# else swap the src and dst field positions and record it to
# the *.swapped file

grep -v "# 10." | awk '{print $1,$4,$5,$2,$3,$7,$6,$9,$8}' > $recordfile.swapped

# merge and sort each file

sort -s $recordfile.notswapped $recordfile.swapped > $recordfile
done

Note: Without the -n option provided to tcptrace, connection analysis takes much longer.

C.2 tcpcsm event equivalence scripts

tcpcsm provides retransmission classification that does not exist in tcptrace. Given the previous, it becomes necessary to perform a deeper comparative analysis between tcpcsm, and tcprs than was necessary for tcptrace. tcpcsm provides classification for retransmission timeouts(RTO), and fast retransmit type loss recovery. There are various difference between tcpcsm, and tcprs that need to be addressed. tcpcsm and tcprs both categorize similar types of events but with labeling differences. To compare these analysis tools, it becomes necessary to create equivalence classes for events of the same or similar type. For example, MS_FRETX is a retransmission event
produced by tcpcsm in response to observing a retransmission that occurs after two
duplicate acknowledgments. **MS_FRETX** is an event tag specific to tcpcsm but such an
event tag does not exist in tcprs. **FAST_SPEC** is an event tag that exists in tcprs for
retransmissions that fit the behavior for fast retransmit, but do not appear to be
retransmitted in response to the third duplicate acknowledgment as specified by RFC
5681. Both **FAST_SPEC** and **MS_FRETX** are events that are similar enough to be
equivalent in terms of event type. With that in mind, several different event equivalence
classes have been established for the purposes of performing a comparative analysis
between the two analysis tools.

The equivalence classes are discussed more thoroughly in section 5.4.2.

To create the event diffs between tcprs and tcpcsm, it is necessary to run the
generate_raw_output.sh and compare_reordering.sh scripts.

Upon executing both scripts, the following directory structure is created:

```
${destination}
  -- ${destination}/results
     ---- "contains .info files, which contain summary info
           from each event type"
     ---- ${destination}/results/RTO
           ------ "contains results from the event differences
                      for retransmission timeout events"
     ---- ${destination}/results/FRETX
           ------ "contains results from the event differences
                      for fast retransmit events"
     ---- ${destination}/results/REORDER
           ------ "contains results from the event differences
                      for reordering events "
```
The destination is provided as an argument to the `generate_raw_output.sh` script.

This process requires that bash and python scripts from the “tools” and “scripts” repositories to be installed. The URL to each repository is provided in Appendix D.

The respective `*.info` file for each equivalence event type provides information as to the count and event distribution of each event.

The version of TCPCSM used for the comparative analysis is version 1.03, available at:

http://www.wand.net.nz/~salcock/tcpcsm/
APPENDIX D: SOURCE REPOSITORIES

The source code for TCPRS is publicly available from the Bro-Plugins repository.

https://github.com/bro/bro-plugins

The source code for the comparative analysis tools, analysis tools and auxiliary tools are provided in two separate repositories. Both repositories have public read-only access. The source code for the majority of the comparative analysis tools is provided at:

https://bitbucket.org/jswaro/scripts

Some of the comparative analysis tools depend on programs written in the “tools” repository, provided at:

https://bitbucket.org/jswaro/tools
APPENDIX E: ADDITIONAL TCP CLASSIFICATION EXAMPLES

Figure E.1: Case: FREC Example - Incorrect 3

Figure E.2: Case: FREC Example - Correct 2
Figure E.3: Case: FRETX Example - Correct 2

Figure E.4: Case: LOSSREC Example - Correct 3
Figure E.5: Case: MSFRETX Example - Incorrect 2

Figure E.6: Case: REORDER Example - Correct
Figure E.7: Case: SACKFRETX Example - Incorrect 2

Figure E.8: Case: SACKFRETX Example - Correct 3
Figure E.9: Case: LOSSREC Example - Correct 2

Figure E.10: Case: REORDER Example - Correct 2
Figure E.11: Case: REORDER Example - Incorrect 2