A Passive State-Machine Approach for Accurate Analysis of TCP Out-of-Sequence Segments*

Sushant Rewaskar, Jasleen Kaur, and F. Donelson Smith
Department of Computer Science
University of North Carolina at Chapel Hill
rewaskar@cs.unc.edu, jasleen@cs.unc.edu, smithfd@cs.unc.edu

ABSTRACT

In this paper we describe a new tool being made available to the networking research community for passive analysis of TCP segment traces. The purpose of the tool is to provide more complete and accurate classification of out-of-sequence segments than those provided by prior tools. One of the crucial factors that limits the accuracy of prior tools is that these do not incorporate variations across TCP implementations (for different operating systems) that have different parameters (e.g., timer granularity, minimum RTO, duplicate ACK thresholds, etc.) or algorithms that influence what can be inferred about out-of-sequence segments. Our tool explicitly accounts for implementation-specific details in four prominent TCP stacks (Windows, Linux, FreeBSD/Mac OS-X, and Solaris). We validate our tool through several controlled experiments with instances of all four OS-specific implementations used in the analysis. We then run this tool on packet traces of 52 million Internet TCP connections collected from 5 different locations and present the results. We also include comparisons with results from running selected prior tools on the same traces.

Categories and Subject Descriptors

C.2.3 [**Network Operations**]: Network Monitoring; C.4 [**Performance of Systems**]: Measurement Techniques; D.4.m [**Operating Systems**]: Miscellaneous

General Terms

Measurement, Design

Keywords

TCP Analysis, Passive, State Machine, Loss

1. INTRODUCTION

In this paper we describe a new tool being made available to the networking research community for passive analysis of TCP segment traces. The purpose of the tool is to provide more complete and accurate results for identifying and characterizing out-of-sequence segments than those provided by prior tools such as tcpanaly, tcpflows, LEAST, and Mystery [9, 18, 19, 26].

Our methodology classifies each segment that appears out-of-sequence (OOS) in a packet trace into one of the following categories: network reordering or TCP retransmission triggered by one of—timeout, duplicate ACKs, partial ACKs, selective ACKs, or implicit recovery. Further, each retransmission is also assessed for whether it was needed or not.

One of the crucial factors that limits the accuracy of prior tools is that different TCP implementations (for different operating systems) have unique parameters (e.g., timer granularity, minimum RTO, duplicate ACK thresholds, etc.) or algorithms that influence what can be inferred about out-of-sequence segments. Our approach is to analyze each TCP segment trace from the perspective of each of four implementations (Linux, Windows, FreeBSD/Mac OS-X, and Solaris) and determine which specific implementation behavior best explains the out-of-sequence segments and timings observed in the trace.

We validate our tool through several controlled experiments with instances of all four OS-specific implementations used in the analysis. We then run this tool on packet traces of 52 million Internet TCP connections collected from 5 different locations and present the results including comparisons with results from running selected prior tools on the same traces.

Given that prior tools have been shown to provide reasonably good results, one might question whether the additional completeness and accuracy justifies creating a new tool. We believe that they do so for the following reasons. First, as we discuss in Sections 2 and 5 below, each of these prior tools has particular strengths and weaknesses for analyzing some aspect(s) of out-of-sequence segments but none deal with all aspects at the desired level of accuracy. Second, a number of potential uses for the analysis results are much enhanced when they are are accurate. For example, while the TCP loss detection and recovery mechanisms are quite mature and unlikely to undergo major design changes, there may still be opportunities for "fine-tuning" to improve certain cases. Prior studies have indicated (and our analysis in this paper has substantiated) that retransmissions are triggered much more frequently by timeouts than by duplicate ACKs, and that significant numbers of retransmissions are unnecessary. Having accurate data on issues such as these is necessary for quantifying the potential benefits of fine-tuning these TCP mechanisms. Another example where accurate results from analysis of out-of-sequence segments are needed is in validating and evaluating models of TCP performance; such models are based on the evolution of TCP's congestion window as it changes along with retransmissions, and according to how the need for a retransmission was detected (timeout or duplicate ACKs) [14, 15, 25]. An inaccurate classification of such retransmission can mislead such evaluations.

In the rest of this paper, we describe our passive analysis methodology in Section 2. We present tool validation in Section 3 and our analysis of Internet connections in Section 4. We summarize related work in Section 5 and our conclusions in Section 6.

*This research was supported in part by NSF CAREER grant CNS-0347814, a UNC Junior Faculty Development Award, and NSF RI grant EIA-0303590.

2. PASSIVE INFERENCE OF TCP LOSSES

A packet trace of a TCP connection is a time-ordered sequence of data segments and acknowledgments (ACKs) exchanged (and observed at the trace-collecting monitor) between the TCP sender and the TCP receiver. Our objective is to find out, given a packet trace, which TCP segments were lost in the network. Below, we first describe the sources of Internet packet traces used throughout this paper and then describe our passive loss inference methodology.

2.1 Data Sources

Table 1 describes the traces used in our analysis. These traces are collected from links with transmission capacity ranging from 155 Mbps to OC-48. The *abi* traces [5] are collected from a backbone link of the Internet-2 network (Abilene); the *jap* trace [6] is collected off a trans-Pacific link connecting Japan to the US by the MAWI working group; the *unc* trace is collected at the campus-to-Internet links of the University of North Carolina; and the *wls* and *wrd* traces are captured inside the UNC campus. The *wls* trace captures wireless TCP connections from over 600 wireless access points while the *wrd* trace captures just the wired network. The *lei* [4] traces are collected at the campus-to-Internet links of University of Leipzig; the *ibi* trace captures traffic served by a cluster of high-traffic web-servers (mirror for ibiblio.org). All traces except the one from the link to Japan were collected using Endace DAG cards [2]; the *jap* trace was collected using tcpdump [17]. The *abi* and *lei* traces are from the NLANR repository. The *unc*, *ibi*, and *jap* traces include TCP options as well. Our trace set is fairly diverse in its geographic location, proximity to TCP senders, as well as types of users represented.

For our analysis, we use only those connections that transmit at least 10 segments. Furthermore, since our objective is to study TCP retransmissions, we select only those connections in which at least one OOS segment is observed ("OOS" connections). Table 2 shows the impact of applying the latter filter. While less than 50% of connections that transmit at least 10 segments also have some OOS segments, these connections carry most of the bytes in this class. Furthermore, the traces vary significantly in the distribution of bytes transmitted per connection—this adds to the diversity of our results.

2.2 Passive Loss Inference Methodology

TCP uses a well-known combination of detection and recovery mechanisms to deal with packet losses—we refer the reader to [8, 12, 15, 16, 23, 31] for details of retransmission timeouts (RTOs), fast retransmit/recovery (FR/R), triple duplicate acks (TDA), partial acks (PAs), and selective acks (SACKs). Each of these mechanisms is used to detect and *retransmit* segments that are perceived to be lost. Below we consider several approaches that exploit the existence of these mechanisms, for reliably inferring packet losses from the packet trace of a TCP connection.

Why not consider all retransmissions?

Since TCP *retransmits* segments on detecting packet losses, the simplest (and common) approach for inferring segment loss is to simply look for the reappearance of some segments in the TCP packet trace and assume that the original transmission was lost somewhere between the monitor and the receiver [21]. However, this approach can lead to over-estimation of losses as illustrated in Fig 1, which depicts part of a TCP connection selected from the *unc* trace. Segment 3 is retransmitted during a post-timeout period, although the original transmission was successfully received

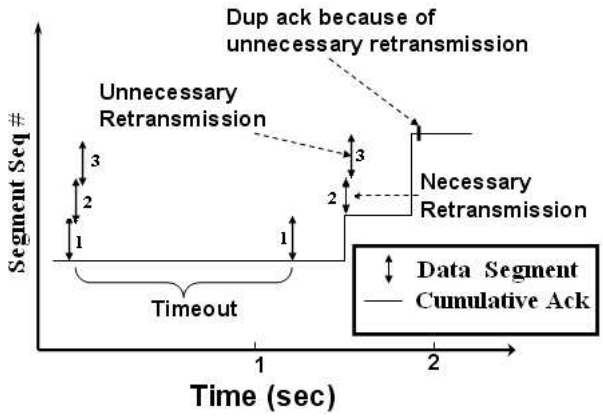

Figure 1: Implicit TCP Retransmission. Segment 1 is retransmitted due to a timeout. Segment 2 is a necessary implicit retransmission while segment 3 is an unnecessary implicit retransmission triggered simply due to TCP's recovery mechanism.

(as is confirmed by the subsequent duplicate ACK). In [9], Allman proposes an algorithm, *LEAST*, that accounts for such unneeded retransmissions in computing the true loss rate of a connection, by simply subtracting the count of duplicate ACKs that are received after timeouts. However, such an approach does not help identify *which* retransmissions were unneeded.

Note that while segment 3 was retransmitted, this was not the result of any *explicit* loss detection/recovery attempt by the TCP protocol. This example, thus, illustrates that in order to reliably infer packet losses from all segment retransmissions, it is important to *track the explicit triggering of TCP's loss detection mechanisms—namely, RTO, TDA, PA, and SACK*.

Why not simply track TCP sender state?

It turns out that even simply tracking the triggering of loss detection/recovery mechanisms in a TCP sender—as is done in [19]—is not sufficient for reliably inferring packet losses. This is because of two reasons related to TCP's inability to accurately infer packet losses:

Some losses do not trigger TCP's loss detection phases. For implementation efficiency, TCP senders maintain only a limited history about unsuccessful transmissions. In particular, if multiple packet losses are followed by a timeout, the sender explicitly discovers and recovers only from the first of those losses. As a result, the remaining packet losses may not get discovered by simply tracking the invocation of TCP's four loss detection mechanisms (RTO, TDA, PA, SACK). Fig 1 illustrates this for segment 2, which was unsuccessfully transmitted the first time. The segment gets retransmitted in the post-timeout period, but without explicitly triggering TCP's loss detection/recovery mechanisms. It is, thus, important to *identify* ***implicit*** *retransmissions that are* ***needed*** *for recovering from packet losses*.

Note that if history about all previously transmitted data packets is maintained, then the ACK stream can help identify such retransmissions (in Fig 1, the cumulative ACK received after retransmission of segment 1 indicates that segment 2, which was previously transmitted, was lost).

A TCP sender may incorrectly infer packet losses. TCP may retransmit a packet too early if its RTO computation is not conser-

Trace	Duration	Avg TCP Load	# Connections	# Bytes	# Packets
Abilene-OC48-2002 (abi)	2h	211.41 Mbps	7.1 M	190.3 G	160.1 M
Japan-155Mbps-2004 (jap)	4h	1.93 Mbps	0.3 M	3.5 G	3.7 M
UNC-wireless-2005 (wls)	178h	1.58 Mbps	20.2 M	126.9 G	157.6 M
UNC-wired-2005 (wrd)	178h	2.18Mbps	6.8M	175.1 G	217.5 M
Liepzig-1Gbps-2003 (lei)	2h 45m	9.53 Mbps	2.4 M	11.8 G	17.3 M
UNC-1Gbps-2005 (unc)	4h	74 Mbps	14.5 M	133.3 G	151.0 M
Ibiblio-1Gbps-2005 (ibi)	4h	90.64 Mbps	0.9 M	163.2 G	158.9 M

Table 1: General Characteristics of Packet Traces. The trace name indicates the location, link speed, the year data was collected and the acronym used for the trace. The remaining columns describe the duration of the trace, average load on the link, and the number of connections, bytes, and packets.

	All Connections			OOS Connections			All OOS Segments Explained		
Trace	# Conn	# Bytes	# Packets	# Conn	# Bytes	# Packets	# Conn	# Bytes	# Packets
abi	389.0 K	180.1 G	148.4 M	66.1 K	120.1 G	100.0 M	40.5 K	55.8 G	45.0 M
jap	58.0 K	5.0 G	4.8 M	29.8 K	4.2 G	4.1 M	23.1 K	1.3 G	1.5 M
wls	329.8 K	121.7 G	144.1 M	101.3 K	113.3 G	122.1 M	63.3 K	28.0 G	40.1 M
wrd	290.9 K	171.3 G	208.8 M	98.0 K	167.7 G	200.0 M	73.3 K	36.7 G	63.1 M
lei	75.4 K	10.5 G	12.6 M	14.0 K	7.8 G	9.7 M	10.7 K	3.1 G	4.4 M
unc	774.8 K	121.3 G	129.5 M	168.1 K	94.7 G	100.5 M	131.7K	46.0 G	49.1 M
ibi	287.5 K	161.8 G	157.2 M	78.5 K	135.6 G	129.5 M	59.8 K	57.4 G	64.9 M

Table 2: Characteristic of Connections That Transmit More Than 10 Segments. Connections that transmit atleast 10 data segments are described under "All Connections". Out of these, the connections with traces that contain atleast one OOS segment are described under "OOS Connection". The final set of columns describe the characteristics of the connections for which our tool was able to unambiguously explain and classify all OOS segments.

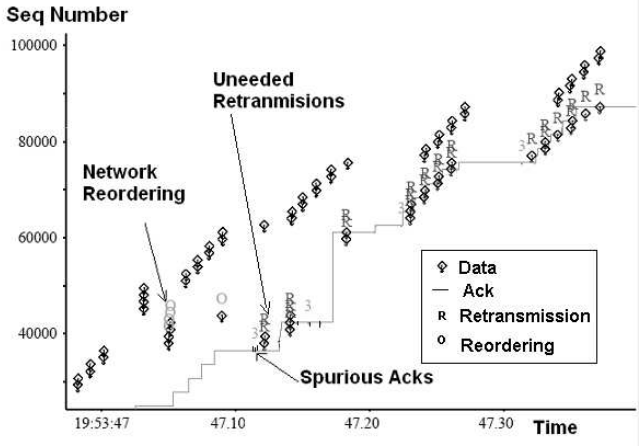

Figure 2: Unneeded Retransmission. This visualization of a real connection from the *unc* trace shows how a single occurrence of network reordering results in some spurious duplicate ACKs, that ultimately trigger 64 subsequent phases of unnecessary retransmissions.

vative. Furthermore, some packet re-ordering events may result in the receipt of TDAs, triggering a loss detection/recovery phase in TCP. In fact, Fig 2, which again depicts part of a TCP connection selected from the *unc* trace (and visualized using the *tcptrace* utility [3]), plots a connection in which a *single* packet reordering event resulted in the triggering of 64 subsequent phases of fast retransmit/recovery, that lasted for more than 5 seconds! It is, thus, important to *identify* ***explicit*** *retransmissions that are* ***not needed*** *for recovering from packet losses*.

Such unneeded explicit retransmissions are not identified by *LEAST* [9]—our analysis of Internet TCP connections in Section 4 shows that more than 90% of unneeded segment retransmissions in the Internet may occur due to explicit loss detection/recovery actions by TCP. Note that an explicit retransmission can be identified as unneeded if an ACK is received within a fraction of the connection's minimum RTT after the segment is retransmitted—we use a fraction of 0.75 in our analysis.

Basic Approach

As reasoned above, if the timing and history about all previously transmitted packets are maintained for each connection, then the ACK stream can help achieve each of the three goals outlined above. Based on this intuition, our basic approach for passive inference of TCP losses is to: **(i)** replicate partial state machine for a TCP sender that uses the data and ACK streams to track the triggering of loss detection/recovery mechanisms, and **(ii)** augment the state machine with extra state and logic about the transmission order and timing of *all* previously-transmitted packets, in order to classify retransmissions as needed or not. Using this basic approach, we can classify segment retransmissions as triggered by: (i) RTOs, (ii) TDAs, (iii) PAs, (iv) SACKs, and (v) implicit. Furthermore, each retransmission is also classified as *needed* or *unneeded*. Fig 3 depicts this classification taxonomy.

A similar approach is taken in [19] for developing a tool, *tcpflows*, for studying congestion window behavior of TCP connections. However, due to the different objective, *tcpflows* does not focus on accurately identifying and classifying segment losses. In particular, it classifies retransmissions into RTO-triggered, TDA-triggered, RTO-recovery, and FR/R-recovery. It does not analyze implicit retransmissions (RTO-recovery) to see if these are needed or not. In Section 4, we show that up to 30% of needed (and up to 40% of unneeded) segment retransmissions in the Internet occur during such an RTO-recovery phase.

2.3 Practical Challenges in Loss Inference

Three kinds of practical concerns complicate the implementation of the above approach. We describe these concerns and how we address them below.

Diverse and Non-documented TCP Stacks

The Challenge:

TCP implementations written by different operating system (OS) vendors may differ (sometimes significantly) in either their interpretations or their conformance to TCP specification/standards. Furthermore, a few aspects of TCP—such as how a sender responds to SACK blocks—are not standardized. As a result, the sender-side state machines can differ across OSes. This results in two main challenges in implementing our basic approach. First, the difference in implementations on different OSes necessitates that we implement different analysis tools to analyze connections originating from different sender-side OSes. More significantly, given the trace of a TCP connection, it is non-trivial to identify the corresponding sender-side OS and decide which OS-specific analysis program to use for analyzing the connection. Second, most OSes either have proprietary code or have insufficient documentation on their TCP implementations. Without detailed knowledge of the loss detection/recovery implementations, it is not trivial to replicate these mechanisms in our OS-specific analysis programs.

This challenge has not been addressed in *tcpflows* [19], which replicates only the TCP standards specification [8, 12, 16, 23, 27]. *tcpflows* has been validated only against connections with FreeBSD senders (that follow the standards closely). Our analysis of general Internet connections in Section 4 reveals that more than 80% of real-world connections involve either a Windows or Linux sender. More importantly, we find that analyzing such connections with a FreeBSD-based tool can introduce significant inaccuracy in identifying and classifying TCP losses.

Our Approach:

We consider and incorporate 4 prominent OS stacks in our analysis tools—namely, Windows XP, Linux2.4.2, FreeBSD 4.10, and Solaris. The TCP sender stack in MacOS is identical to the FreeBSD stack; hence this OS is also implicitly incorporated in our analysis. We used the popular passive fingerprinting tool, *p0f* [32], in order to identify the sender OS in three of our traces (*unc*, *ibi* and *jap*)—we found that nearly 90% of TCP connections originated from one of these 5 sender-side OSes.

We extract sufficient details about the implementation of loss detection/recovery in the above OS stacks using three different approaches: (i) by studying the source code when publicly available, (ii) through direct communication with OS Vendors, and (iii) by using an approach similar to the *TBIT* approach described in [24] (in order to infer non-public details). To extract OS information using *TBIT* we install all four above-mentioned OSes on experimental lab machines and run the Apache web-server on each machine. We then implement an application-level TCP receiver (by borrowing from the *TBIT* code base) that initiates TCP connections to each of the server machines and requests HTTP objects. Once the server machines start sending the objects, the receiver artificially generates different sequences in the ACK stream to trigger loss detection/recovery mechanisms on the sender-side stacks (including TDAs, RTOs, PAs, and SACKs). We then use the manner in which the server responds to the ACK stream for inferring several characteristics of the sender-side TCP implementation, including the computation of RTO, the number of duplicate ACKs that trigger FR/R, and the response to SACK blocks. Details of the extracted characteristics can be found in Table 3 and in [28]. We use these details in our implementation of four OS-specific trace analysis programs.

For each TCP connection to be analyzed, we run its packet trace against all four analysis programs. We then select the program that is able to explain and classify each retransmission event.

Delays and Losses Between Monitor and Sender

The Challenge:

Packet traces used in passive analysis are typically collected at links that aggregate traffic from a large and diverse population. As a result, there may be several network links on the path between a TCP sender and the trace monitoring point. Thus, the data packets transmitted by the sender may experience delays,[1] losses, duplication, or reordering before the monitor observes them; the same is true for ACK packets that traverse between the monitor and the sender. Consequently, the data and ACK streams observed at the monitor may differ from those seen at the TCP sender. In particular, if some of the TDAs observed at the monitor fail to reach the sender, the analysis programs may incorrectly conclude that the sender has entered FR/R. Similarly, if a data packet gets lost before it reaches the monitor, and subsequently gets retransmitted, the analysis programs may fail to infer that the packet has been *re*-transmitted. Thus, the programs may not be able to accurately track the sender-side state machine.

Our Approach:

In order to deal with this complication, we use a general approach in which loss indications in the ACK stream trigger only *tentative* state changes in the monitor state machine, which are *confirmed* only by subsequent retransmission behavior by the sender. In addition, we consider all *out-of-sequence* (OOS) segments (and not just retransmitted segments) as possible indicators of packet loss. Furthermore, we infer network reordering by either (i) detecting if an OOS segment appears within a fraction (0.75) of the connection's minimum RTT after the segment with the next higher sequence number, or (ii) detecting reordering in the *IP-id* field of packets seen from a given TCP source. Finally, we infer network duplication of packets by detecting repetition in the *IP-id* field of reoccurring segments seen from a given TCP source. We remove such duplicated OOS segments from further analysis.

Non-availability of SACK Blocks in Traces

The Challenge:

A large number of traces do not capture the TCP option field. SACK blocks are transmitted as TCP options and hence are not available

[1]The RTT measured at the monitor (monitor-receiver-monitor) is less than that measured at the sender (sender-receiver-sender). We address this issue (i) by estimating the monitor-sender-monitor delay during the initial three-way SYN/SYN+ACK handshake, and (ii) by adding this quantity to each estimate of the monitor-receiver-monitor delay, in order to obtain the sender-receiver-sender RTT. The initial sub-RTT obtained from the SYN/SYN+ACK exchange is a good approximation of the minimum monitor-sender-monitor delay [7]. If subsequent delays on this sub-path vary significantly, the RTO computed at the monitor may be smaller than that used by the sender. Fortunately, this discrepancy does not negatively impact our analysis—the RTO is used as a *minimum* threshold for the gap between the original transmission and retransmission of a lost segment. Therefore, a smaller-than-actual value of RTO would simply lower the threshold and still be able to correctly identify retransmissions that occur due to timeouts.

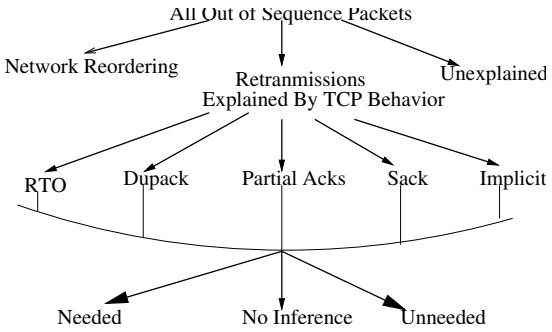

Figure 3: Classification Taxonomy.

for passive analysis of these traces. The sender may have used the SACK block information to retransmit certain packets. In absence of these blocks, the monitor will fail to accurately identify the cause of these retransmissions.

Our Approach:

To overcome this problem, we develop the following heuristic to identify whether a packet could have been triggered by incoming SACK information. We classify a segment retransmission as SACK-triggered if: (i) the connection is in FR/R, (ii) the retransmission is not explained by either RTO or a PA, and (iii) the sequence number of the retransmitted segment is less than the highest sequence number that was in flight when the connection entered FR/R. We evaluate this heuristic using the *unc* and *jap* traces. We first run our analysis tools with the SACK blocks available and log all OOS segments that were SACK-triggered. Then we remove the SACK blocks from these traces and run the tools with the above heuristic. The heuristic-based analysis identified all of the OOS segments identified as SACK-triggered by the analysis based on SACK blocks; however, it also marked 6.9% and 15.3% of the unexplained events as being SACK-triggered, in the *unc* and *jap* traces respectively. Our analysis of Internet TCP connections in Section 4 shows that only a small fraction (less than 7%) of all OOS segments are SACK-triggered—the possible overestimation introduced by the above heuristic, therefore, is not significant.

2.4 Summary of Our Methodology

Our methodology for reliably inferring and classifying TCP losses can be summarized as follows.

1. We first extract the implementation details of four prominent TCP stacks (Windows XP, Linux 2.4.2, FreeBSD 4.10 (MacOS), and Solaris) using the approaches described in Section 2.3. These details primarily include the initial RTO, the minimum RTO, the RTO estimation algorithm, the number of duplicate ACKs that trigger FR/R, and the responses to partial ACKs and SACKs. In addition, some OS-specific peculiarities are included—for instance, if a segment with options fields is to be retransmitted in FR/R, some versions of Windows transmit a small packet equal to the size of the options field.

2. We then replicate the loss detection/recovery mechanisms in four OS-specific analysis state machines—these state machines use the data and ACK streams as input. Loss indications in the ACK stream are used to only tentatively trigger state transitions, which are confirmed only by subsequent segment retransmission behavior. For instance, on detecting an RTO-based retransmission, the state machine will enter an "RTO-recovery" state. A new RTO is calculated, any pending RTT measurements are canceled and the SACK block, if present, is cleared. The machine exits this state on receiving an ACK for the highest packet that was in flight when RTO was detected.

3. We then augment these machines with extra logic and state about all previously-transmitted packets, in order to classify retransmissions as needed or unneeded and infer packet losses with accuracy greater than TCP.

4. We then run each connection trace against all four machines and use the results from the one that can explain and classify all of the observed OOS segments. In case more than one machine matches this criteria, we check if the classification of each OOS segment is the same in each machine. If not, we discard the connection. We also discard the connection in case none of the machines can explain each OOS segment.

Our methodology classifies all OOS segments that appear within the packet trace of a TCP connection, according to the taxonomy depicted in Fig 3.

We have implemented the above machines in the C programming language. All four implementations can analyze more than a million connections in a few minutes. Several details of our methodology and implementation have not been included in this section due to space constraints. These details can be found in [28]. The source code is available online via [1].

In the next two sections, we validate our methodology and compare its performance with past work.

3. VALIDATION

Our primary validation method is to compare the output from the analysis tools for TCP connections where the "ground truth" about the classification of each OOS segment is known. To do this, we modified the TCP Behavior Inference Tool (*TBIT*) [24] in order to observe the sender's responses under additional controlled conditions. We supplement this validation by comparing the determination made by the tools for identifying a specific OS implementation with the results from *p0f* [32] - a well-known passive fingerprinting tool.

3.1 Validation Against *TBIT* Controlled Conditions

TBIT emulates a TCP protocol stack for the receiver side of a unidirectional data transfer where the sender is a normal application (in our case a Web server) running over a real TCP implementation in a specific operating system. We modify *TBIT* to simulate different packet loss scenarios that would trigger sender responses by withholding ACKs, sending duplicate ACKs, and providing SACK blocks. Because the state-machine analysis critically depends on inferring the TCP sender's RTO to identify retransmissions triggered by timeouts, we use *TBIT* to delay ACKs thus simulating variable round-trip delays. For some of the validation scenarios described below we also use dummynet on the *TBIT* machine to create additional constant latency between the sender and receiver.

For each validation scenario we used two machines, one running *TBIT* and the other running a web server, connected over a switched 100/1000 Mbps Ethernet that is shared by users in the Computer Science department. *TBIT* established a TCP connection to the web server and sent a valid HTTP request for a very large file. *TBIT* then implemented the desired validation scenario with a specifically generated ACK stream. Unless stated otherwise, each

Parameter	Linux	Windows	FreeBSD	Solaris
Timer granularity	10ms	100ms	10ms	10ms
Initial RTO (s)	3	3	3	3.375
Min RTO (ms)	200	200	1200	400
RTO	srtt + vartt	srtt + 4*rttvar	srtt+ 4*rttvar	1.25*srtt + 4*rttvar
Dup-ACK threshold	3	2	3	3

Table 3: Values of key parameters in different TCP Stacks

validation scenario was repeated 100 times because not all sources of variation in timing could be controlled (e.g. OS scheduling, Ethernet switch delays, etc.). Separate estimates of these uncontrolled delays concluded that the majority were less than 1 millisecond and nearly all were less than 10 milliseconds.

The entire suite of validation scenarios was run with *TBIT* connecting to each of four different TCP implementations on the server machine – Windows XP, Solaris, Linux 2.4.2, and FreeBSD 4.10. Bidirectional tcpdumps of all packets were taken on these server machines and the traces were then used as input to our validation procedures. The procedures have two parts – (1) to verify that each TCP implementation responds in real operation as expected (thus establishing the "ground truth") , and (2) to verify that the state-machine analysis programs correctly emulate each implementation's responses. For part (1) we processed the tcpdump traces with *tcptrace* [3] and other tools to verify the implementations' responses by inspection. For part (2), we used the tcpdumps as input to the state-machine analysis programs and recorded their outputs. By comparing the results from the state-machine analysis with the known implementation responses, we could determine how correct the inferences about conditions at the sender were. We also used the tcpdumps as input to the analysis program, *tcpflows*, presented in [19] but report the results from this only when they differ substantially from ours. In addition, we implement the *LEAST* algorithm from [9] for identifying unneeded retransmissions.

3.1.1 RTO classification:

The first group of validation scenarios deal with how well the state-machine analysis can infer the sender's estimate of RTT and RTO which are critical in identifying retransmissions triggered by timeouts. In this group of validation scenarios, *TBIT* causes all retransmissions to be triggered by timeouts (by withholding ACKs). The analysis state machine for each implementation requires correct values for parameters defining the initial and minimum RTO, the timer granularity, and the equations used in computing RTO. These elements are verified as part of the validation results. Table 3 gives the values used in the state machine for each TCP implementation.[2] [3]

RTT estimation:

Dummynet was used in experiments with constant minimum RTTs—of 50, 100, 150, 200, 400, 1000, and 2000 ms—between the two machines. All RTTs estimated for segment/ACK pairs by our state machines were within +/- 10 milliseconds of the value set by dummynet (these differences are consistent with the inherent variable delays in the switches).

[2]Details about the RTO computation (srtt and rttvar) are taken from RFC 2988 [27]. Linux, however, uses a significantly different computation for the variance in RTT—we extract this from the Linux source code. The details can be found in [28].

[3]Some parameters for Windows are based on private communication with engineers at Microsoft Corp.

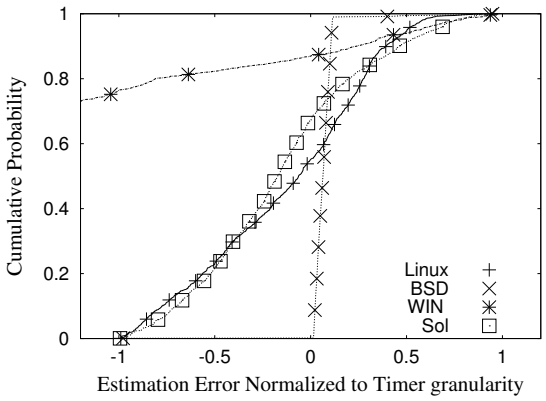

Figure 4: Error in RTO estimation for different OSes

Initial RTO setting:

The initial RTO parameter helps classify retransmissions of SYN or SYN+ACK segments at connection establishment. *TBIT* initiated a connection (sent SYN) but did not respond to the SYN+ACK sent by the server. This resulted in a retransmission of the SYN+ACK after the initial RTO interval. Our OS-specific state machines correctly identified the SYN+ACK retransmission as being triggered by RTO; further, the measured RTO was equal to the value expected +/- the timer granularity (also shown in Table 3).

Minimum RTO setting:

No delays were added to the actual RTT (typically 1 millisecond) over the switched Ethernet. *TBIT* received and ACKed a significant number of segments (typically 50 or more) so the sender's RTO calculation stabilized before withholding all ACKs to trigger an RTO retransmission. The extremely small RTT and the stabilization of the RTO before we simulate a dropped packet ensure that RTO should occur after an interval approximately equal to the minimum RTO. The OS-specific state machines correctly identified these retransmissions as triggered by RTO using these minimum values and timer granularities.

RTO Estimation

These validations were conducted with both near-constant and highly variable random delays. For the experiments with near-constant delays (varying by only 1-10 ms caused by switch delays), we used dummynet to set a target minimum RTT ranging from 10 to 1000 ms between the two machines. For experiments with highly variable delays, ACKs were delayed randomly by *TBIT* to vary the RTT from 0 to 400% above the dummynet minimum delays described above. In both sets of experiments *TBIT* triggered RTO retransmissions by withholding ACKs after a randomly selected packet.

Figure 4 summarizes the results of all the RTO experiments. It shows the CDF of the error between the actual RTO extracted from the tcpdump and the RTO value predicted by the state machines, normalized to the timer-granularity. We see that the errors fell well within the timer granularity for a particular OS except for Windows. Windows exhibits a strong instability in its RTO calculation. We contacted engineers at Microsoft who attributed our observations to a "rounding issue" with the OS, the details of which were not revealed due to copyright issues. However, our heuristics for timeout detection in the state machine for Windows are conservative enough to not be affected by the error. In terms of absolute numbers, the difference between the observed and state machine

Trace	# OOS Connections	p0f-identified Connections	% Linux		% Windows		% FreeBSD		% Solaris		% Other OSes
			Correct	Wrong	Correct	Wrong	Correct	Wrong	Correct	Wrong	
jap	23923	21260 (89%)	25.79	0.02	21.21	0.32	41.51	0	1.98	0.05	9.11
ibi	59713	59713 (100%)	99.80	0.20	0	0	0	0	0	0	0
unc	138214	136524 (99%)	7.26	0	78.08	0.69	5.02	0	8.52	0.17	0.25

Table 4: Validation using *p0f*. The third column lists the number (and percent) of connections for which p0f was able to identify the source OS. For each OS, we next list the percent of connections for which our estimation of sender OS was correct or wrong. The last column lists the percent of connections which did not belong to any of the OSes that we model. All percent values are with respect to the second column.

RTO was within $20ms$ for Linux and within $2ms$ for FreeBSD and Solaris.

These experiments also allowed us to estimate the percentage of RTO timeout events that would have been missed if we used only the RFC specifications in the analysis tools as is done for *tcpflows* [19]. We found that if only the RFC specification was used, we would miss 85% of RTO events in Linux TCP connections, 55% in Windows, and 100% in Solaris. This is perhaps the most important reason that OS-specific logic needs to be incorporated in the analysis tools.

3.1.2 FR/R classification:

The second group of *TBIT* validation scenarios deals with how well the state-machine analysis can infer the sender's response to duplicate ACKs, partial ACKs in Fast Recovery, and SACK blocks. In all cases, *TBIT* received and ACKed a randomly chosen number of segments before creating a specific loss scenario.

Number of duplicate ACKs to trigger retransmission:

To simulate this case, *TBIT* sent duplicate ACKs (without delays) in response to subsequent segments (thus simulating loss of a random segment). The number of duplicate ACKs was varied from 1 to 4. We repeated each of these experiments 4 times with different random number seeds. In the absence of enough duplicate ACKs, the sender times out and this is detected correctly by our OS-specific state machines. In the presence of enough duplicate ACKs, the retransmission was by a Fast retransmit and our OS-specific state machines also classified these events correctly. For a windows connection, the *tcpflows* tool failed to identify the retransmissions triggered by 2 duplicate ACKs. This is because it assumes that 3 duplicate ACKs are needed, as is recommended in the RFC specification.

Response to Partial ACKs in Fast Recovery:

TBIT triggered a retransmission by sending sufficient duplicate ACKs (as described above) and then sent partial ACKs for a randomly chosen segment from among those transmitted between the original and retransmission. We repeated this experiment 4 times with different random seeds. Our OS-specific state machines correctly identified these Partial ACK events. Note that Windows TCP does not retransmit on receiving a partial ACK during FR/R but instead retransmissions are triggered by RTO (does not implement newReno but does use SACK if present).

Response to SACK blocks

TBIT triggered a retransmission by sending sufficient duplicate ACKs and generated several different cases of SACK block contents indicating gaps in the received segments beyond the simulated loss. In all cases, our OS-specific state machines correctly classified such retransmissions. There are minor differences in the way Windows responds to SACK. This can cause a RTO-triggered retransmission even in presence of correct SACK blocks. These packets were correctly classified by our Windows-specific state machine. The *tcpflows* tool, which does not use SACK blocks, classified the above as simply retransmissions during "FR/R recovery".

Unneeded and Needed Retransmissions:

TBIT simulated instances of the implicit retransmission scenario of Fig 1. In a second set of scenarios, it sends spurious duplicate ACKs to trigger an unneeded retransmission (similar to Fig 2). Our OS-specific state machines correctly classified the corresponding retransmissions as *needed* or *unneeded*. These experiments also allowed us to compare our state-machine results with those we obtained by implementing the algorithm used in *LEAST* [9]. *LEAST* correctly identified the unneeded retransmissions in the first scenario but failed to identify them in the second case.

3.2 Validation Against Real TCP Connections

Next, we validate our tool-set against traces of real-world TCP connections. In this case, since we do not have access to either the TCP sender or the receiver for these connections, the ground truth about the classification of each OOS segment is not known. Consequently, we can not use the same validation tests as those used in Section 3.1. Instead, we use our tool to identify the sender OS (as the one corresponding to the state machine that is able to explain all OOS segments). Our validation evaluates how accurately does our tool-set identify the sender-OS (and hence, is able to accurately model the sender state machine and classify OOS segments).

For establishing the ground truth about the sender-OS, we rely on *p0f* [32]—a popular passive fingerprinting tool which uses the information present in the option fields of SYN, SYN+ACK, or Reset segments to identify the source OS for the packet. We use *p0f* to identify the sender-side OS for all OOS connections in the *jap*, *ibi*, and *unc* traces that were successfully classified by our tool-set. These traces include TCP option fields and, hence, can be analyzed by *p0f*.

We compare our estimate of the sender-OS to that reported by *p0f*. Table 4 reports the comparison results. The numbers listed under the OS-specific columns report the percent of *p0f*-identified connections for which our tool-set correctly or incorrectly identified the sender-OS. We observe that:

- *p0f* is able to identify the sender-OS for 89-100% of the connections. The relative mix of sender-OS is quite different across the three traces. This is to be expected; *ibi* represents connections to a cluster of web-servers, all of which run Linux; *unc* represents members of an academic and medical community, most of whom use Windows PCs; *jap* represents trans-continental connections made by a generic mix of users in Japan.

- Our estimate of sender-OS matches that of *p0f* for more than 99% of the connections—accuracy is high for all four OSes. We attribute this high level of success to two factors: (i) our in-depth modeling of sender-state as well as high granularity of analysis of OOS segments; and (ii) our conservative approach of filtering out connections with even a single OOS segment that is not robustly explained.

A natural question to ask is: *in practice, how important is it to correctly model the sender OS?* In particular, if an RFC-based analysis tool is used, how different would the results be. We investigate this and other issues in the next section.

4. IMPACT

We believe the reason for the high degree of accuracy of our tool is that we insist on unambiguously explaining and classifying all OOS segments that appear within a connection. In order to be able to do so, our tool encodes significant amount of state and logic and it incorporates much of the diversity across TCP implementations. It is natural to ask: *in practice, how much difference does this make?* In particular, if prior tools are used to analyze real-world OOS connections, how different would the classification results be? We investigate this issue by raising several questions below—we address each question by analyzing all of the seven Internet TCP trace-sets described in Section 2.1.

- ***How many OOS segments can we successfully classify?***

  Table 2 reports the number of OOS connections in which *all* OOS segments were unambiguously classified by our analysis. We find that in nearly 25-35% of OOS connections, at least one OOS segment could not be classified. Two main factors are responsible for the failure to completely classify a connection.

  - First, we specifically model only 5 sender OS versions. In order to study the prevalence of these OSes, we ran *p0f* against all connections (whether or not they had any OOS segments) that appear in the *jap*, *unc*, and *ibi* traces. While more than 80% of connections in each trace originated from a Windows or Linux machine, we found that nearly 10% of connections in each trace originated from an OS different from the above five—such connections, consequently, may not be successfully modeled by our state machines.
  - Second, recall that we apply a conservative filter for accepting a connection classification: (i) each OOS segment that appears in a connection trace must be explained, and (ii) the explanations must match if more than one state machine explains all such segments.

    More that 50% of the discarded connections are discarded only due to the second rule above. In Section 3.2 we saw that *p0f* can be used quite effectively for identifying the source OS of connections. This allows us to eliminate the second filtering rule—if more than one state machines explain all OOS segments of a connection, *p0f* can be used to identify the sender OS, and the corresponding OOS classification can be accepted. We are currently incorporating this feature in our tool-set.

  Figure 5 plots the distribution of the number of unexplained OOS segments in each OOS connection. We see that *all* segments are classified in 62-95% of the connections and these are accepted by our filters (as is also indicated in Table 2). More interestingly, the number of unexplained OOS segments are less than 5 in most of the remaining connections. Since the total number of unexplained segments is small, it may be worthwhile to include in our analysis the explained OOS segments even from the discarded connections. We study below how our classification results change if we do so. For the rest of the analysis in this section, however, we do not include results from such connections.

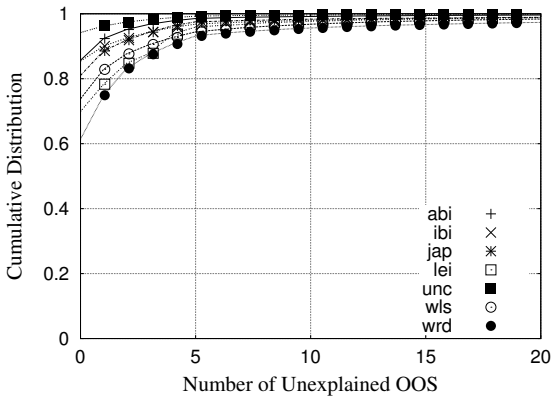

Figure 5: The distribution of the number of unexplained OOS segments in each OOS connection.

- ***How important is it to replicate TCP sender state?***

  Tables 5 and 7 report our classification for OOS segments in the seven traces, according to the taxonomy of Fig 3. Table 7 shows that the fraction of retransmissions that are unneeded (the original transmission of the segment was not lost) ranges from 3-19%. This suggests that, in practice, TCP loss rate would be significantly overestimated if every segment retransmission is taken as an indicator of packet loss—this underscores the importance of modeling and replicating TCP sender state.

  In order to study the effect of our connection filter—that requires that each OOS segment be unambiguously classified—we present in Table 6, our classification results when the explained OOS segments are included from all connections (including the connections discarded by our filter). We find that the number of unexplained segments are small (7-16%) in each trace. More importantly, Tables 5 and 6 are quite comparable in the distribution of elements within the classification tree.[4] This suggests that our filter does not bias our classification results significantly.

- ***How important is it to identify unneeded explicit retransmissions?***

  Table 7 also compares the classification of needed and unneeded retransmissions made by our tool-set to that made by *LEAST* [9]. We find that the number of unneeded retransmissions reported by *LEAST* is always lower (sometimes by more than 50%) than that reported by our tool-set. There are two reasons for this. First, as illustrated in Section 2 and as demonstrated in Section 3, *LEAST* does not identify some explicit retransmissions that are unneeded. Table 7 indicates

[4]We also find, although not reported here, that the values in Table 7 (including those reported for *LEAST*) do not change significantly if explained OOS segments from discarded connections are also included.

Trace	# OOS Segments	Our Tool-set							*tcpflows*				
		% Network Reorder	% Segment Retransmissions						% Network Reorder	% Segment Retransmissions			
			Total	RTO	Dupack	PA	SACK	Implicit		RTO	Dupack	RTO recovery	FR/R
abi	339.2 K	14.1	85.9	33.4	17.9	4.4	7.1	23.1	-	-	-	-	-
jap	121.7 K	4.2	95.8	46.9	13.9	5.8	2.4	26.9	-	-	-	-	-
wls	1119.5 K	5.8	94.2	52.9	10.7	6.1	2.4	22.1	-	-	-	-	-
wrd	1250.4 K	5.3	94.7	66.0	9.6	2.5	1.1	15.6	-	-	-	-	-
lei	110.5 K	0.2	99.8	53.5	9.9	2.8	5.0	28.6	0.8	55.2	7.5	34.7	1.8
unc	1327.4 K	12.9	87.1	40.3	13.2	6.1	6.5	20.9	13.8	39.5	7.0	36.0	3.6
ibi	787.4 K	0.2	99.8	32.8	17.3	8.7	0.4	40.8	0.27	26.5	21.2	29.7	22.3

Table 5: Classification of OOS segments by *tcpflows* and by our tool-set. These are from connections for which we were able to unambiguously explain and classify all OOS segments. *tcpflows* classifies an OOS segment as one of: network reordering, retransmission triggered by RTO, duplicate ACKs, or during FR/R or RTO-recovery.

Trace	# OOS Segments	% Network Reordered	% Segment Retransmissions						Unexplained
			Total	RTO	Dupack	PA	SACK	Implicit	
abi	1345.0 K	11.4	88.6	26.9	17.1	4.0	7.3	17.4	16.0
jap	340.8 K	6.3	93.7	35.6	14.6	5.0	4.2	22.1	12.2
wls	2927.5 K	7.7	92.3	39.9	11.4	5.9	2.7	22.3	10.0
wrd	4177.3 K	7.3	92.7	43.9	11.4	2.2	0.8	19.2	15.3
lei	294.5 K	0.4	75.6	40.6	11.6	3.1	5.6	24.0	14.7
unc	2752.9 K	12.6	87.4	32.9	12.7	5.7	6.0	20.1	10.0
ibi	2383.9 K	0.7	93.0	26.3	19.6	10.9	0.2	34.8	7.5

Table 6: Classification of all OOS segments (including unexplained events) by our tool-set. These are all connections irrespective of whether we were able to explain all events or not.

that a majority of unneeded retransmissions occur due to explicit TCP loss detection-recovery actions. Second, when duplicate ACKs generated by unneeded implicit retransmissions are lost in the network, *LEAST* fails to conclude that the retransmission was not needed. While this is true even for our tool-sets, our additional analysis of the timing between the retransmission and the ACK (ACK arrives within a fraction of the minimum RTT) for the segment helps us identify some of these retransmissions.

- ***How important is it to classify implicit retransmissions?***

  Implicit retransmissions are not analyzed for whether these are needed or not by *tcpflows* [19]. Table 5 indicates that the fraction of retransmissions that are sent implicitly by TCP is significant (16-40%). More importantly, Table 7 indicates that, in practice, up to 30% of needed (and up to 40% of unneeded) segment retransmissions occur implicitly. Classifying these is, therefore, important for any study of either TCP losses or the effectiveness of TCP mechanisms.

- ***How important is it for the analysis to be OS-sensitive?***

  tcpflows [19] is based on TCP standards specified in RFCs and does not incorporate variations that exist in TCP implementations across different OSes. In order to assess the impact of being OS-insensitive, we analyze using *tcpflows* all OOS connections that were explained by our tool-set in the *lei*, *unc*, and *ibi* traces (the other traces could not be processed by *tcpflows* due to incompatible trace formats). Table 5 includes the results—note that the "FR/R" classification of *tcpflows* is a combination of our PA- and SACK-triggered categories, and that "RTO-recovery" is captured by our implicit category. The classification differs significantly from that of our tool-set reported in the same table, underscoring the need for incorporating popular implementations.

  We also evaluate the need for OS-sensitive analysis using our tool-set. For this, we again consider all OOS connections in the above three traces that were explained by our OS-sensitive tool-set, and observe the classification results when only our FreeBSD-specific state machine (which follows the TCP standards fairly closely) is used on these. This state machine was unable to explain around 50% of all OOS segments in each trace!

- ***How important is it to incorporate delays and losses between the monitor and the sender?***

  Table 5 shows that significant number of OOS segments occur due to network reordering between the sender and the monitor. We have also observed that a significant fraction of losses occur between the sender and the monitor. It is, therefore, important to incorporate such network anomalies in the analysis.

  In the *abi* and *unc* traces,[5] nearly 13-14% of OOS events are classified as due to network packet reordering between the sender and the monitor—these numbers appear unusually high. To investigate these events further, in Fig 6, we plot the time gap (referred to as the *resequencing delay*) between each such OOS segment and the segment with the next higher sequence number. We find that most of the resequencing delays are within 5 ms—this indeed corresponds to timescales

[5]A known contributor of excessive reordering in the UNC trace is the presence of intrusion detection appliances that divert, from selected connections, a few IP packets from the fast data-path for deeper inspection.

Trace	# Total Retran	Our Tool-set							*LEAST* [9]	
		% Needed			% Unneeded			% No Inference		
		Total	Implicit	Explicit	Total	Implicit	Explicit		% Needed	% Unneeded
abi	291.9 K	79.1	13.1	66.0	12.0	4.9	7.1	8.8	89.6	10.4
jap	116.6 K	82.4	4.4	78.0	15.6	6.6	9.0	2.0	92.7	7.3
wls	1054.2 K	86.7	22.3	64.4	13.2	1.1	12.1	0.1	98	2.0
wrd	1184.2 K	96.2	15.9	80.3	3.7	0.5	3.2	0.1	97.7	2.3
lei	110.3 K	82.5	19.6	62.9	12.7	4.2	8.5	4.8	87.7	12.3
unc	1155.9 K	91.2	21.4	69.8	7.7	1.1	6.6	1.5	96.2	3.8
ibi	785.5 K	76.6	23.2	53.4	18.9	13.1	5.8	4.5	85.0	15.0

Table 7: Needed and Unneeded Retransmissions (for connections with all OOS segments unambiguously explained).

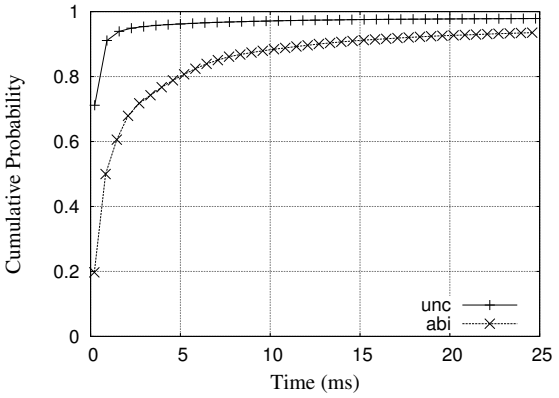

Figure 6: Resequencing delays for reordered segments

of network reordering and is much smaller than typical RTTs. The small fraction of OOS segments with large resequencing delays occur in connections with large minimum RTTs as well.

5. RELATED WORK

There already exists an extensive body of work on passive analysis of TCP connections. Tcptrace [3] is perhaps the most widely used among the many tools available for passive analysis. However, tcptrace, like many other tools, does not maintain enough state to accurately infer TCP losses. The most closely related work is *tcpflows* [19], a state-machine approach for analysis of TCP connections. This state-machine design is based on RFC specifications for congestion response, retransmissions, and RTO calculations. *tcpflows* can perform a passive analysis of traces taken at an arbitrary network link and attempts to characterize the causes of losses using inferences of RTO and the sender's congestion window. Study of a trace from a backbone link using *tcpflows* in [19] concluded that 9-19% of retransmissions were unneeded, and that 7-25% of out-of-sequence events were because of packet reordering. Though this tool was a significant advance in passive analysis, we found that this method has several practical limitations because of the differences in the various widely used TCP implementations. Furthermore, since the primary purpose of [19] was not to study packet losses in detail, their analysis tool is limited in the granularity with which OOS segments are classified. The results from their method (especially the RTT calculation and the subsequent RTO calculation) are dependent on the frequency with which RTT is measured (per packet vs. per flight), and the inferences necessary to track the sender's congestion window. All of these depend on the details of specific TCP implementations. To overcome these problems we have used OS-specific state machines to more robustly infer TCP sender state in passive analysis.

OS-specific analysis is not a new idea. In [26], Paxson implemented a stateful implementation-specific analysis in his tool "tcpanaly" for passive analysis of traces at the end system. The primary limitation is that it has not been extended to handle traces taken from an arbitrary link. Since the analysis were performed on end system traces, there was no need to address several practical challenges such as packets lost between the trace point and end system. Further, the analysis did not have to infer the specific TCP implementation characteristics because the end system OS was known in advance. Given the above reasons and the significant pace of changes to TCP implementations since the time the tool was developed, we believe that our tools represent a substantial advance.

Finally, there is work related to identification and classification of TCP losses. We do not consider active loss characterizations [13, 30, 29] as they are studying loss properties of network paths, rather than loss characteristics of TCP connections. In [11], the authors design a tool for actively measuring reordering on a network path. This tool exploits the relative sequence number spacing between segments transmitted and can not be easily adapted for passive monitoring of the network. TCP *mystery* [20] is another tool which identifies loss events and classify them as necessary or unnecessary. This tool uses a subset of the algorithms used in *tcpflows*—our comparison to *tcpflows* suggests that it needs to incorporate additional details in order to achieve accuracy similar to ours. In [9], the authors have presented the *LEAST* algorithm for passively estimating unneeded retransmissions that occur after a timeout (for Reno implementations) or using SACK blocks. We find that their method underestimates unneeded retransmissions in Reno implementations because they do not address additional retransmissions in Fast Retransmission/Fast Recovery. Further, the limited state maintained does not track unneeded retransmissions when duplicate ACKs are lost. Our tools maintain sufficient history about all packets, including those that are retransmitted, so a more robust identification of unneeded retransmission can be made.

There are other methods in the literature for identifying spurious retransmission due to timeouts [10, 22]. Both these methods deal only with timeout-triggered retransmissions. [10] relies on the time difference between the retransmitted segment and the ACK to identify spurious timeouts. [22] proposed the Eifel Algorithm which uses the timestamp option to actively detect spurious timeouts. This method requires end-system cooperation and is not suitable for passive analysis.

6. CONCLUDING REMARKS

The primary contribution of our work is the implementation and validation of a new suite of tools for passive analysis of TCP connections. These tools are freely available to the networking research community and we hope they will encourage others to contribute to our understanding of TCP behavior "in the wild" by analyzing larger and more diverse sets of traces. While many of the ideas used in these tools are not new (see the discussion of related work), we believe this is the first time all have been integrated into a single, carefully validated, analysis approach. Further, we have made significant advances by explicitly including TCP implementation-specific factors for those operating systems that are currently (and likely to be for the foreseeable future) the dominant end points for TCP connections (Windows, Linux, FreeBSD/MAC OS X, Solaris). We have also been careful to cover many of the "corner cases" and boundary conditions that are missing in prior work, choosing to rely on explicit sender-state tracking rather than approximations or heuristics where possible.

We believe the accuracy and high classification granularity of our tools will enable other networking researchers to address issues related to the efficacy of TCP's loss detection/recovery mechanisms, to develop new models for the underlying loss processes that TCP must deal with, and to better understand the impact of network congestion on real-world TCP performance. For example, the analysis of real-world TCP connections may suggest important implications for the refinement of analytic models of TCP throughput as a function of loss rates [14, 25].

While we have discussed only the analysis of TCP loss and retransmission characteristics, that is not the end of the story. We believe the TCP implementation-specific state machines are sufficiently detailed and robust that they can form the basis for passively tracking additional TCP and network states, specifically congestion windows, packets in flight, and end-system buffering. Additional research is in progress to address these issues.

Acknowledgments. The authors would like to thank Sharad Jaiswal for making available and helping install the code for *tcpflows*.

7. REFERENCES

[1] URL http://www.cs.unc.edu/∼jasleen/research/.

[2] The dag project,univ. of waikato, URL http://dag.cs.waikato.ac.nz/.

[3] tcptrace. URL http://jarok.cs.ohiou.edu/software/tcptrace/tcptrace.html.

[4] URL http://pma.nlanr.net/special/leip1.html.

[5] URL http://pma.nlanr.net/traces/long/ipls1.html.

[6] URL http://tracer.csl.sony.co.jp/mawi/.

[7] J. Aikat, J. Kaur, D. Smith, and K. Jeffay. Variability in TCP round-trip times. In *Proceedings of the ACM SIGCOMM Internet Measurement Conference*, October 2003.

[8] M. Allman, V. Paxson, and W. Stevens. RFC 2581: TCP congestion control, 1999.

[9] Mark Allman, Wesley M. Eddy, and Shawn Ostermann. Estimating loss rates with TCP. *SIGMETRICS Perform. Eval. Rev.*, 31(3), 2003.

[10] Mark Allman and Vern Paxson. On estimating end-to-end network path properties. In *SIGCOMM '99: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 263–274, New York, NY, USA, 1999. ACM Press.

[11] John Bellardo and Stefan Savage. Measuring packet reordering. In *IMW '02: Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, pages 97–105, New York, NY, USA, 2002. ACM Press.

[12] E. Blanton, M. Allman, K. Fall, and L. Wang. RFC 3517: A conservative selective acknowledgment (SACK)-based loss recovery algorithm for TCP, April 2003.

[13] J. Bolot. End-to-end packet delay and loss behavior in the Internet. In *SIGCOMM '93: Conference proceedings on Communications architectures, protocols and applications*, 1993.

[14] Neal Cardwell, Stefan Savage, and Thomas Anderson. Modeling TCP latency. In *INFOCOM (3)*, pages 1742–1751, 2000.

[15] K. Fall and S. Floyd. Simulation-based comparisons of tahoe, reno, and SACK TCP. *ACM Computer Communication Review*, 26(3), July 1996.

[16] S. Floyd, T. Henderson, and A. Gurtov. RFC 2582: The newreno modification to TCP's fast recovery algorithm, 2004.

[17] V. Jacobson, C. leres, and S. McCanne. tcpdump: URL http://www.tcpdump.org.

[18] S. Jaiswal, G. Iannaccone, C. Diot, J. Kurose, and D. Towsley. Measurement and classification of out-of-sequence packets in a tier-1 ip backbone. In *Proceedings of IEEE INFOCOM*, April 2003.

[19] S. Jaiswal, G. Iannaccone, C. Diot, J. Kurose, and D. Towsley. Inferring TCP connection characteristics through passive measurements. In *Proceedings of IEEE INFOCOM*, March 2004.

[20] Sachin Katti, Dina Katabi, Charles Blake, Eddie Kohler, and Jacob Strauss. Multiq: automated detection of multiple bottleneck capacities along a path. In *IMC '04: Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 245–250, New York, NY, USA, 2004. ACM Press.

[21] R Krishnan, M Allman, C Partridge, and J Sterbenz. Explicit transport error notification (ETEN) for error-prone wireless and satellite networks. *Technical Report TR-8333, BBN Technologies*, March 2002.

[22] Reiner Ludwig and Randy H. Katz. The Eifel algorithm: making TCP robust against spurious retransmissions. *SIGCOMM Comput. Commun. Rev.*, 30(1):30–36, 2000.

[23] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. RFC 2018: TCP selective acknowledgement options, 1996.

[24] J. Padhye and S. Floyd. On inferring TCP behavior. In *Proceedings of ACM SIGCOMM*, 2001.

[25] Jitendra Padhye, Victor Firoiu, Don Towsley, and Jim Kurose. Modeling TCP throughput: a simple model and its empirical validation. In *Proceedings of ACM SIGCOMM*, pages 303–314. ACM Press, 1998.

[26] V. Paxson. *Measurements and Analysis of End-to-End Internet Dynamics*. PhD dissertation, University of California, April 1997.

[27] V. Paxson and M. Allman. RFC 2988: Computing TCP's retransmission timer, November 2000.

[28] S. Rewaskar, J. Kaur, and D. Smith. Passive inference of TCP losses using a state-machine based approach. *Technical Report TR06-002, Department of Computer Science, University of North Carolina at Chapel Hill*, October 2005.

[29] Stefan Savage. Sting: A tcp-based network measurement tool. In *USENIX Symposium on Internet Technologies and Systems*, 1999.

[30] Joel Sommers, Paul Barford, Nick Duffield, and Amos Ron. Improving accuracy in end-to-end packet loss measurement. In *SIGCOMM '05: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, volume 35, pages 157–168, New York, NY, USA, October 2005. ACM Press.

[31] W. Richard Stevens. *TCP/IP illustrated (vol. 1): the protocols*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.

[32] M. Zalewski. Passive OS fingerprinting tool: URL http://lcamtuf.coredump.cx/p0f.shtml., 2006.