Incrementally Deployable Prevention to TCP Attack with Misbehaving Receivers

Kun Gao and Chengwen Chris Wang
{kgao, chengwen}@cs.cmu.edu
Computer Science Department
Carnegie Mellon University

December 15, 2004

Abstract

In a TCP connection, selfish receiver has incentive to increase its own transmission rate at the expense of other receivers. Whether it is downloading a web page or transferring a file, most receivers prefer the server to focus only on their transmission and ignore those of other receivers. Unfortunately, the current TCP implementation can be easily exploited for a malicious receiver to gain an unfair amount of bandwidth.

This is a problem in TCP implementations because the only good estimate of congestion from a sender's perspective comes from a receiver's acknowledgment [FF99]. As a result, a receiver can mislead the sender's congestion control through spurious ACK packets. TCP is especially vulnerable because of it implicitly assumes that end users are well behaved. In fact, through spurious ACKs, a receiver can gain an arbitrary amount of bandwidth [SCWA99].

In our paper, we focus on the class of attacks that try to fool a sender's congestion control through spurious ACKs. We discuss related work in Section 2. We propose several different incrementally deployable solutions to this class of attacks and how our solution performs in subsequent sections. By incrementally deployable we mean that only modifications to the sender's TCP is needed.

1 Introduction

There are various attacks on TCP, such as those in [KK03] and [SCWA99]. In our paper, we focus on the class of attacks that try to fool a sender's congestion control through spurious ACKs. We first establish specifically the attack we would like to prevent. We then develop an incrementally deployable solution, give analysis of its performance compared to TCP Reno, our reference TCP protocol.

1.1 Related Work

Savage, Cardwell, Wetherall, and Anderson [SCWA99], analyzed the effect of sending spurious acknowledge. They developed the following three different attacks [SCWA99],

- ACK Division: When receiving N bytes of data, divide the data into M distinct segment and send an acknowledgments for each segment.
- DupACK Spoofing: The receiver sends a long stream of identical acknowledgments for the last sequence number received.
- Optimistic ACKing: The receiver sends a stream of acknowledgments for the anticipated arrival of packets that have yet to arrive.

Remark. Most TCP implementation ignore acknowledgment (ACK) for sequences outside the congestion window (this is a reasonable implementation, as more drastic action to invalid acknowledgments can lead to simple exploits to disrupt TCP connections of other receivers).

There is a simple solution to ACK division, and easy methods to mitigate effect of dupACK spoofing. (This can be solved by refusing to enter fast retransmit multiple times. [Flo95]) But for optimistic ACKing, the only proposed solution ([SCWA99]) is to include an additional field in the TCP header. This field is a randomly generated number from the sender. When the receiver sends back an acknowledgment, it has to send the corresponding random number back. As a result, receivers could no longer send acknowledgments for packets that have not yet arrived.

A simple solution to optimistic ACKing would be to vary the amount of bytes sent in each packet. If the receiver did not acknowledge the right length, it is assumed that the receivers misbehave. However, this solution makes the assumes that the receiver must ACK the exact number of bytes in a packet (for every packet sent). Indeed, this is not the case, as in TCP implementations, the receiver can under ACK. Specifically, if we vary the packet size between 1350 to 1400. The receiver could just send ACK for 1350, 2700, 4050, and so on. “In this example we can see that each acknowledgment is valid, in that it covers data that was sent and previously unacknowledged.”[SCWA99] Hence, this method does not prevent a misbehaving receiver from performing the attack.

1.2 Our Contribution: Overview and Motivation

The existing solution for optimistic ACKing has many drawbacks. In particular, it is not backward compatible. In other words, the existing solution requires both senders and receivers to change its implementation of TCP. We propose several different solutions that only require the sender to change its implementation, while keeping the same TCP implementation for the receiver. In addition, the existing analysis is only performed on HTTP 1.0 connection. In the current HTTP protocol, the receiver can not request any segment. (But in the newly purposed HTTP 1.1, the receiver can query any range. Thus, it is even more important to prevent this attack before HTTP 1.1 is deployed, as HTTP 1.1 allows the receiver to maintain better reliability while abusing the congestion feedbacks. We wish to extend the analysis to FTP and similar bulk file transfers which allow retransmission of individual segments. In these cases, the receiver could afford losses and just open a separate connection later to request the missing segments. Because FTP connections not only have a significantly more disastrous effect on the network performance, the ability for receivers to conceal losses (through subsequent requests to resend certain segments) enables them to be even more aggressive in sending false feedback.

1.3 Type of attack:

In order to discuss the usefulness of our result, we need to define specifically the type of attack we intend to prevent, and the assumptions we made.

Attacker: A single receiver of a single TCP connection. (We assume there is no collusion between different receivers, and a receiver can only open one connection at a time.)

Type of Attack: ACK before receiving data. Since the receiver still want to receiver data, it does not make sense to violate the TCP protocol during the hand-shake process or terminate the connection before all data is transmitted. Thus, the receiver must ACK for data, and we assume this is the only place the receiver can misbehave. In fact, there are several other attacks using spurious acknowledgment, but there are simple methods to mitigate the effects of other receiver attacks with small changes in the current sender's TCP implementation. [SCWA99] The only known prevention for acknowledgment before receiving data requires a change in the TCP header and implementation of both senders and receivers.

Goal of Attacker: Falsify to the sender so as to send data at a faster rate than the fair share. In the worse case (or best case for the receiver), the attacker wants to increase his congestion window quickly to the bottleneck bandwidth, and keep the congestion window there. As a result, the attacker can utilize almost 100% of the bottleneck, while other receivers get nothing.

Goal of our new TCP protocol:

- Prevent attack: In every RTT, if the receiver ACKs for q percent of packets before its arrival, then he will be caught at least p percentage of time. (The exact value for p and q depend on different methods.)
- Backward compatible: We only require the sender to update its TCP protocol. The receiver does not need (and is not expected) to change his own protocol. This is important in most HTTP transfer, because the server of most website want to share their webpages will as many honest receivers as possible.
- TCP Compatible: We want to make sure our protocol competes fairly with the original TCP protocol.

2 Simulation

Here, we discuss our experimental setup. We discuss this first, and then explain our experimental results when presenting our solutions. We use ns2 simulations of our implementation to determine the performance of our solution to TCP Reno. We wish to show that our methods can detect misbehaving receivers with only small or no penalty in performance. We use the test cases below, describing briefly the setup, as well as what the experiment is designed to measure.

Remark. When we compare our implementation to the standard TCP protocol, we use the same receiver protocol for both TCP sender.

Single Link: We would like to compare the throughput of our implementation against a standard TCP Reno. This experiment is designed to determine under no loss, if our implementation is affected by the bursty nature of a link. For this, we use two nodes connected by a single 1Mb link with 100ms latency, 25 queue size, DropTail queue, 500 byte packet size, FTP connection, and 10000 for receiver's congestion window and no loss.

Single Link with Loss Rate: In this simulation, we analyze how our implementation behaves under loss. We use a 1Mb link with 100ms latency (same parameters, application and topology as above) and vary the percent of packet loss.

Dumbbell Topology: This is probably a more reasonable network topology compare to the above two topologies. We analyze our implementation's behavior to random bursts created by many connections. We create two routers and a 1Mb link with 100ms latency (using the same parameters as above), and 10 senders connect to one router and 10 standard TCP receivers connect to the other with 100Mb, 5ms links.

Shared Dumbbell Topology: In case if only some servers uses standard TCP Reno, and others use our implementation, we analyze our implementation's nature when sharing a link with TCP Reno. We use the same topology as the above experiment except of the 10 senders, 5 use TCP Reno, and 5 use our implementation.

3 Methods to prevent Optimistic ACKing

We now describe several methods we have formulated to make TCP resistant to selfish behavior through *early ACK* attacks. Early ACK means acknowledge for sequence numbers that have yet to arrive.

The main reason receiver can ACK early is because the congestion window is predictable. Thus, the receiver can guess what to ACK before the packet even arrives. (The solution proposed in [SCWA99] is to add a field with random number. Whenever the receiver get the packet, he has to reply with some function of the random field. This effectively prevent the receiver from ACK early.) To prevent this attack without modifying receiver's protocol, we have to either make the congestion window unpredictable.

3.1 Uniform Random Congestion Window:

Main Idea: The sender keeps an *ExpectedCwnd*, which is exactly the same as the congestion windows size (cwnd) in the original TCP implementation. For every short fixed period of time, sender sets its congestion windows to a random number between 1 and twice the ExpectedCwnd minus one. (And keep the congestion window at this random number until the next update.) Whenever the sender receives acknowledgment not in the congestion window, an abuse is detected. (In fact, we only want to punish the receiver if the acknowledgment is for sequences slightly bigger than the last sequences send. Because if we punish all invalid acknowledgment, it might make our TCP implement vulnerable to other spoof IP attack.)

Possible Drawback: This above scheme is likely to create many short bursts, which might significantly reduce the effectiveness of TCP. [KK03]

Experimental Result: In our experiment, we randomly changed the congestion window based on ExpectedCwnd. The interval between changes are an exponential distributions with an average of 1 RRT.

It turns out that the above solution significantly reduces the performance of a TCP connection in our experimental topologies, because of the large variation in congestion window. Specifically, when the randomize congestion window is small, we under utilized the bandwidth. On the other hand, when the random number is large, we often send too many packages and lead to significant loss. Intuitively, this is due to the fact that we create a significant variance. Because we are using TCP Reno, a chain of loss often lead to time out.

Perhap a little unexpected, the method sent more packets than TCP Reno when there is a fix percentage of loss rate. This is partly because our loss model drops each packet with a fix percentage. As a result, Reno's congestion control is mislead to believe that its sending too much packets. On the other hand, a uniformly random congestion window protocol sends too many packets. Even though the same percentage is lost, more packets are still sent.

This method performs very well in a dumbbell topology, each connection sent approximately equal amount of packets, and almost completely utilized the bottleneck link. This is expected because each connection varies its connection window randomly. As a result, the changes in congestion window is highly asynchronous.

Unfortunately, a uniformly random congestion window method is not competitive with TCP Reno. It uses less than 20% of the bottleneck bandwidth. Because it times out frequently and artificially sends much less traffic when the congestion window is set to a small value, TCPReno can slowly take over most of the bottleneck bandwidth.

We do not show this in our figures, as the following approach is an approach that is more performance conscious.

Attack Detection: In our simulation, the interval between changes are exponential distributions because of its memory-less property. Thus, the fact that receiver knows congestion window has not change for a fix number of packet does not help him predict if congestion window is changed on the next packet.

It is not hard to see that in the above method, if the receiver cheats and sent p percentage more packets than the ExpectedCwnd in a RRT, then he will be caught with probability $p/2$.

3.2 Bi-modal Congestion Window: (Rand Reno)

Main Idea: A uniformly random congestion window method is sound in its idea. We can improve its performance by observing that we can reduce the variance while maintaining the same guarantee if we choose a different distribution. Original, it's Uniform(1, 2*cwnd-1). Instead, if we choose between Uniform(1, 2*cwnd-1) and 1.5*cwnd with equal probability, we have the same $E[C]$, and a lower $Var[C]$. But we can improve this further with the assumption below.

Since the attacker is only likely to cheat sporadically, we want to have a higher chance of catching an attack when receiver is conservatively creating false ACK, while penalize the chance of detection when attacker is grossly mis-reporting. (In addition, this assumption allow us to significantly reduces the variance.) When we solve for the distribution that guarantees a p percentage of catching an attacker, the distribution turn out to be a Bi-modal with probability p set cwnd to 1, and probability $(1-p)$ set cwnd to ExpectedCwnd$/(1-p)$.

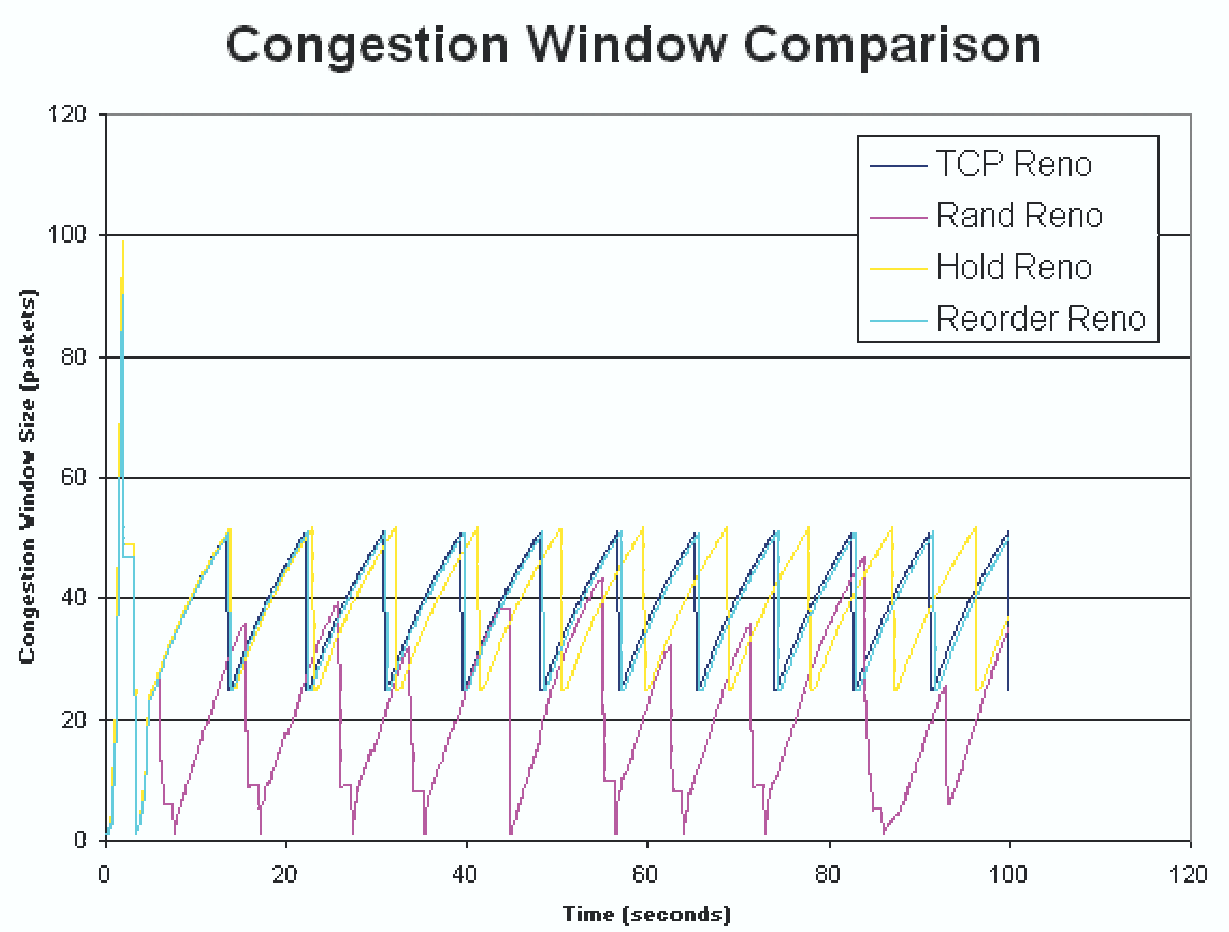

Figure 1: Congestion Window of TCP Alternatives. Bi-modal TCP's time out frequently (purple line). The line shows the ExpectedCwnd Packet withholding introduces a shift in the congestion window (yellow line) due to delay in congestion control caused by withholding a full round trip time. Packet reordering and TCP Reno has almost identical congestion window behavior (blue and teal lines).

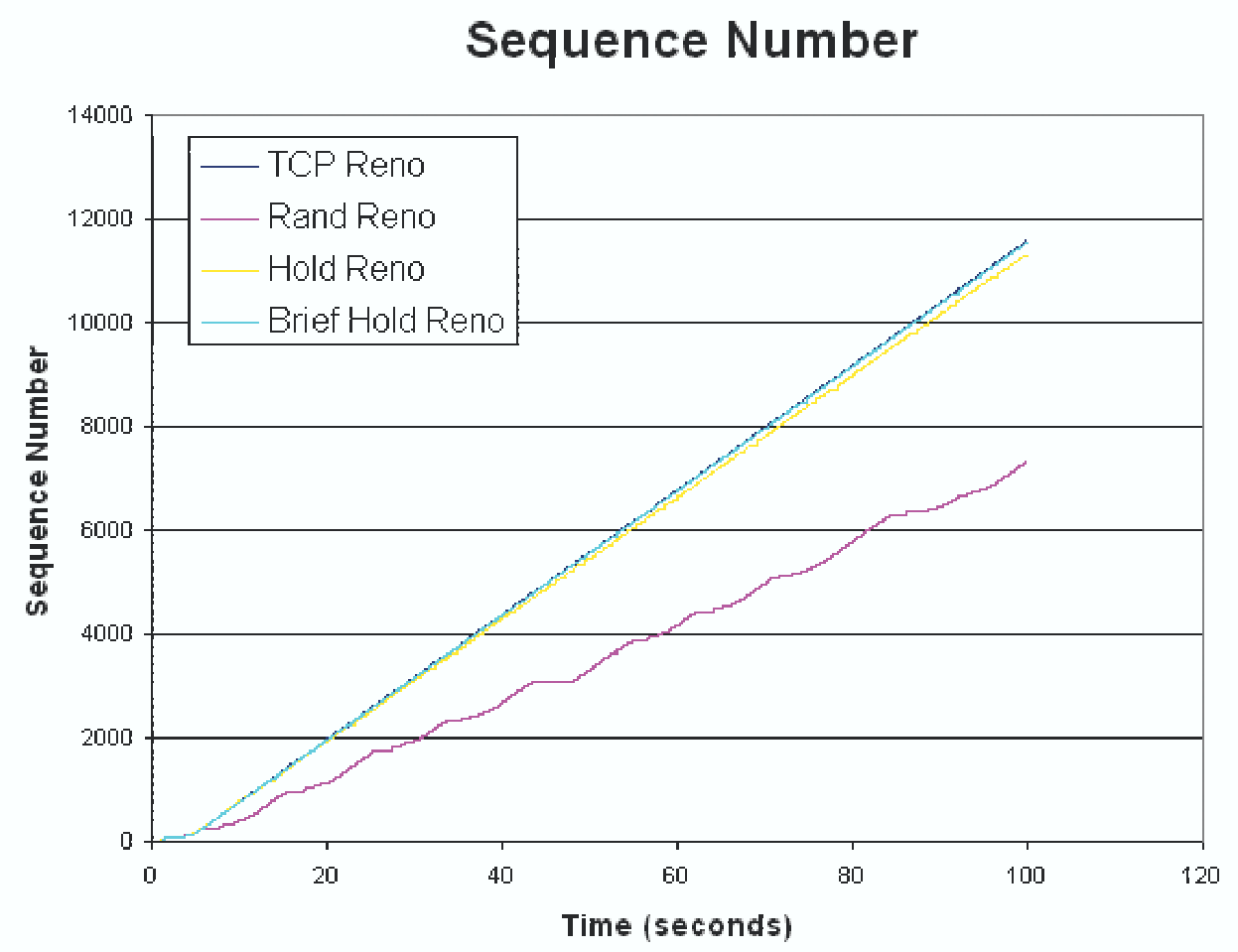

Figure 2: Throughput performance for TCP Alternatives. Packet Reordering approach performs almost identically to TCP Reno (blue and teal lines), while Packet Withholding does slightly worst (yellow line). A Bi-modal approach is over penalized for over estimation of congestion window (purple line).

Experimental Result: We choose p to be 10% in our experiment.

This method still perform poorly. The result is very similar to the uniformly random congestion window method. Specifically, a Bi-modal approach has only about 60-70% throughput in a single link simulation; perform slightly better than Reno in our loss model; shared and fully utilized the bottleneck bandwidth; and does not compete well against TCPReno. This is partly due to the fact we are using TCP Reno, and Reno times out when there are multiple loss in a single window. We labeled this method "Rand Reno" in our figures 1-4.

Attack Detection: If the receiver sends even a single ACK before receiving the packet, then he will have 10% chance of being caught. This detection percentage is independent of how much false early ACK he sent. (Unless receivers early ACK for more than $(1+1/p)*100\%$ of the congestion window, then he will definitely be caught.)

3.3 Packet Withholding: (Hold Reno)

Main Idea: The sender can randomly delay (or withhold) packets. Specifically, we can withheld byte number X and force the receivers to send DupACK for $X - 1$. We call those *expected DupACK*. Since there is no method for the receiver to distinguish if a missing packet is the result of congestion or withholding. Thus, if receiver conceal the loss and mis-report an acknowledgment for a sequence number slightly large than the delayed packet, the abuse can be detected.

There are some subtle implementation detail. Specifically, we continue to increase our congestion window when we get an expected DupACK. In addition, if a loss occur before the packet we withheld, we sent the withheld packet immediately.

In our implementation of withholding, when we are not withholding a packet, we withhold the next packet with 50% probability. However, if we currently withhold a packet, then we do not withhold a second packet.

Since a packet that is withheld will be propagated for a full congestion window, we set our withholding probability appropriately such that p percent of full congestion windows have a withheld packet.

Possible Drawback: While this scheme might not affect TCP SACKS much, it might reduce the performance of TCP Reno, especially when there are several losses immediately after the delayed packet. The sender will not realize the loss until significantly later. In addition, this scheme could be a problem for real-time application. (But majority of real-time application use UDP instead of TCP, so this is not as big an issue.)

Experimental Result: Packet withholding performs fairly well. It is only approximately 2% less than the standard TCP Reno in throughput tests. Whenever there is a loss right after the withheld packet, a packet withholding method does not notice it until a full RTT. As a result, packet withholding sometimes observe a congestion 1 RTT after the actual loss. Due to this slight error in estimation and delay in adjustment of cwnd, performs slightly worse than TCP Reno.

With respect to loss, packet withholding performs the same as TCP Reno. Because in our loss model, none of the those is due to congestion. As a result, the delay in congestion estimation did not penalize packet withholding.

Packet withholding loss is handled in the same way as TCP Reno. Thus, it has the same behavior with respect to synchronization. Since the only performance difference between packet withholding and TCP Reno is an over-estimation for congestion control, packet withholding shares competes fairly and equally with TCP Reno. We labeled this method “Hold Reno” in our figures 1-4.

Attack Detection: However, there is still a method for receiver to abuse our protocol. Whenever a packet is lost or withheld, the receiver could simply send a congestion window of DupACK without risk detection. Because we always withhold a packet for a RRT, and we have to increase the congestion window for the expected DupACK. (If we do not increase congestion window due to expected DupACK, the congestion window will increase very slowly because most ACK are expected DupACK.)

Thus, even though the receiver can not predict when a withholding occur, he can predict the number of DupACK whenever he perceived a loss. In our next method, we proposed another

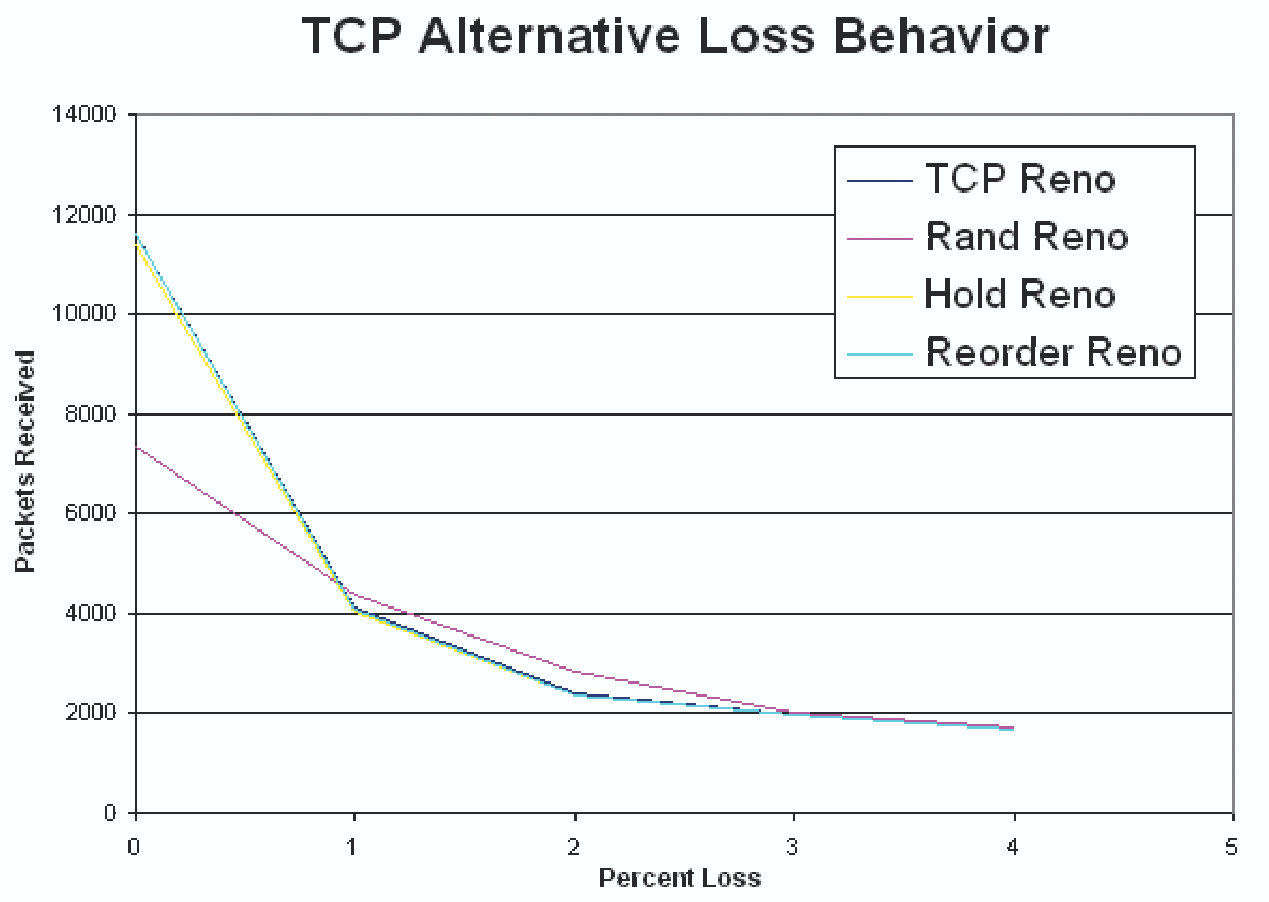

Figure 3: Behavior of TCP Alternatives under loss. Bi-modal TCP does worse with no loss (purple line), but slightly better due to over estimation than Packet Withholding and Packet Reordering methods (yellow and teal lines). TCP Reno is shown in blue.

solution that will prevent this attack.

3.4 Packet Reordering: (Reorder Reno)

Main Idea: Instead of always withheld a packet for a round trip time, we withheld each packet for a random short period of time (4 to 8 packets). In addition, we often withhold multiple packets. This reduces the delay in congestion control, because a packet is no longer withheld for a full RTT. This prevents an attack in that the sender knows exactly the number of expected DupACK behavior from the receiver. We choose at least 4 packets of reorder in order that our reordering does not coincide with what is expected as reordering by the link.

We set p appropriately such that the expected time we withhold a packet is p percent of a full congestion window size.

Experiment Result: Packet reordering perform almost identically to TCP Reno. Specifically, it has almost no penalty in throughput tests (which was a short coming of packet with-

holding). This is attributed to decreasing the amount of time we withhold packets so as to not be penalized severely when withholding a packet at the same time as there is a loss. Packet reordering also competes fairly and equally with TCP Reno. We labeled this method "Reorder Reno" in our figures 1-4.

Attack Detection: In this method, the receiver could no longer predict how many expected DupACK to send, because we reorder a packet by a random amount of time. (To be precise, the receiver might be able to sent 1 or 2 early DupACK here and there, because Reno allows up to 3 packet reordering. Moreover, if the receiver guessed the expected DupACK incorrectly, the sender are likely to assume there is a loss and reduce the congestion window.) In other words, even though receiver might be able to send a few early ACK here and there, his early expected DupACK ACK has a decent chance to decrease his congestion window. In fact, since congestion window is AIMD, a correct guess only increase congestion window slightly while a incorrectly guess can often reduce the congestion window by half. Thus, the large penalty of an incorrect guess is likely to prevent receiver from misbehave.

4 Conclusion

It is always in a TCP receiver's interest to fool a sender into increasing the congestion window of the connection. In fact, [SCWA99] showed that a receiver can increase a the congestion window arbitrarily with very few lines of code. We focus specifically on the problem of optimistic ACKing, in which a receiver will ACK in anticipation of arriving packets. While [SCWA99] proposed complex changes to both sender and receiver to carry hashed values in packets, we propose a much simpler solution. Our solution of brief packet reordering introduces the idea of artificial packet reordering at a sender's TCP level to introduce sender trackable behavior in the ACKs received. Brief packet reordering performs as well as TCP Reno in throughput, and under loss conditions. It is also TCP compatible. Furthermore, our solution is incrementally deployable (needs only to be deployed at the sender).

Future work will focus on analyzing deployment of on servers, as well as applying artificial reordering of packets to other similar attach problems.

References

[FF99] Sally Floyd and Kevin Fall. Promoting the use of end-to-end congestion control in the Internet. *IEEE/ACM Transactions on Networking*, 7(4):458–472, 1999.

[Flo95] Sally Floyd. TCP and successive fast retransmits. February 24, 1995.

[KK03] A. Kuzmanovic and E. Knightly. Low-rate tcp-targeted denial of service attacks (the shrew vs. the mice and elephants, 2003.

[SCWA99] Stefan Savage, Neal Cardwell, David Wetherall, and Tom Anderson. TCP congestion contorl with a misbehaving receiver. *Computer Communication Review*, 29(5), 1999.

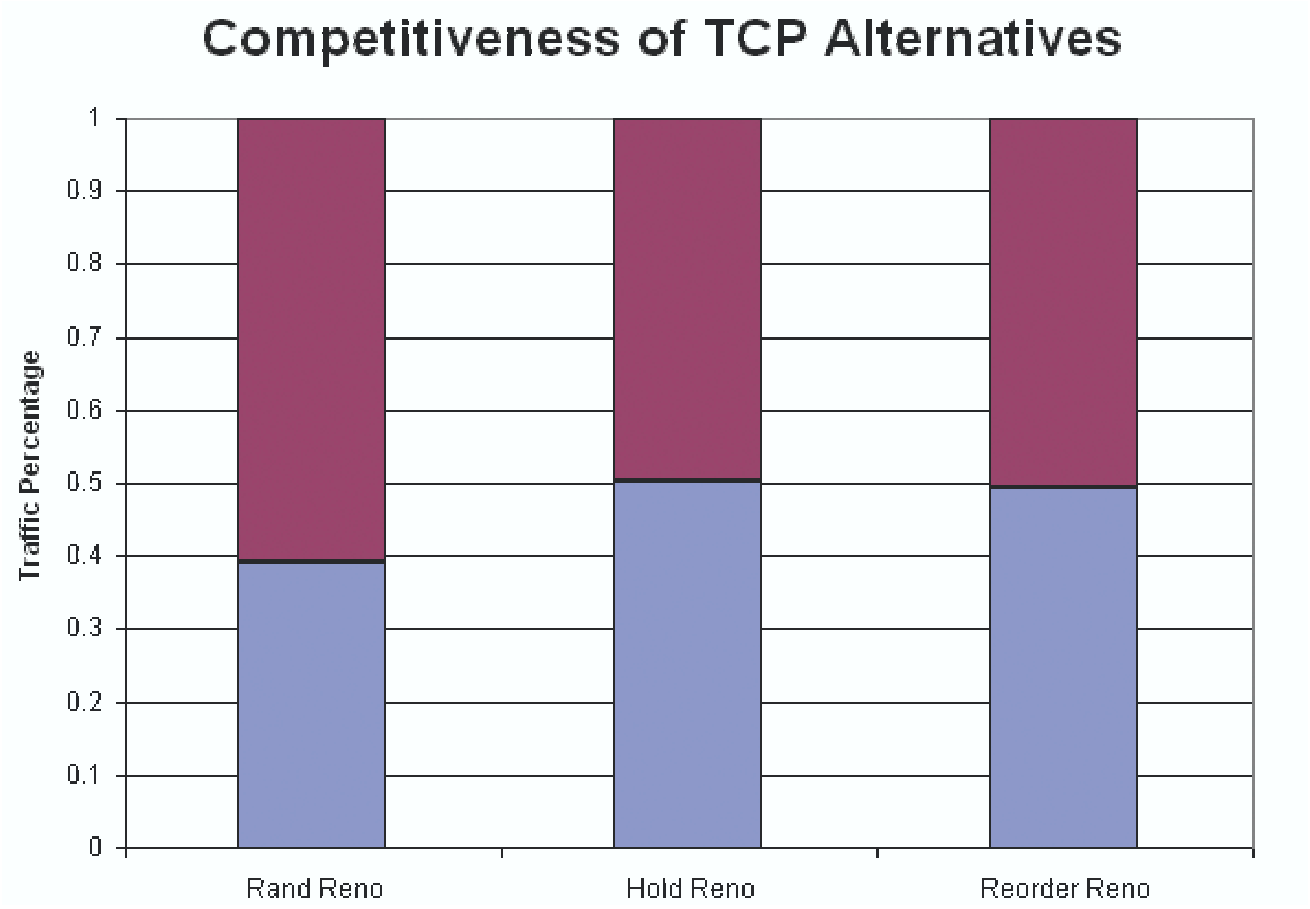

Figure 4: Competitiveness of TCP Alternatives. TCP Reno traffic is shown in purple, alternative TCP implementations shown in blue. This graph shows is generated by counting the percentage of packets sent through the bottleneck link in the shared dumbbell topology. (e.g. the first bar shows that 5 Rand Reno connections used about 39% of the bottleneck bandwidth, while the other 5 Reno used about 61% of the bandwidth.) Bi-modal TCP is less competitive since it backs off randomly. Both Packet withholding and Packet reordering methods compete fairly and fully with TCP Reno.