Misbehaving TCP Receivers Can Cause Internet-Wide Congestion Collapse
Technical Report: UMD-CS-TR-4737

Rob Sherwood Bobby Bhattacharjee Ryan Braud
{capveg,bobby}@cs.umd.edu rbraud@cs.ucsd.edu

Abstract

An *optimistic* acknowledgment (opt-ack) is an acknowledgment sent by a misbehaving client for a data segment that it has not received. Whereas previous work has focused on opt-ack as a means to greedily improve end-to-end performance, we study opt-ack exclusively as a denial of service attack. Specifically, an attacker sends optimistic acknowledgments to many victims in parallel, thereby amplifying its effective bandwidth by a factor of 30 million (worst case). Thus, even a relatively modest attacker can totally saturate the paths from many victims back to the attacker. Worse, a distributed network of compromised machines ("zombies") can exploit this attack in parallel to bring about wide-spread, sustained congestion collapse.

We implement this attack both in simulation and in a wide-area network, and show it severity both in terms of number of packets and total traffic generated. We engineer and implement a novel solution that does not require client or network modifications allowing for practical deployment. Additionally, we demonstrate the solution's efficiency on a real network.

1 Introduction

Savage et al. [29] present three techniques by which a misbehaving TCP receiver can manipulate the sender into providing better service at the cost of fairness to other nodes. With one such technique, optimistic acknowledgment ("opt-ack"), the receiver deceives the sender by sending acknowledgments (ACKs) for data segments before they have actually been received. In effect, the connection's round trip time is reduced and the total throughput increased. Savage et al. observe that a misbehaving receiver could use opt-ack to conceal data losses, thus improving end-to-end performance at the cost of data integrity. They further suggest that opt-ack could potentially be used for denial of service, but do not investigate this further.

In this paper, we consider a receiver whose *sole interest* is exploiting opt-ack to mount a distributed denial of service (DoS) attack against not just individual machines, but *entire networks*. In this paper, we:

1. Demonstrate a previously unrealized and significant danger from the opt-ack attack (one attacker, many victims) through analysis (Section 3) and both simulated and real world experiments.

2. Survey prevention techniques and present a novel, efficient, and *incrementally deployable* solution (Section 4.2) based on skipped segments, whereas previous solutions ignored practical deployment concerns.

3. Argue that the distributed opt-ack attack (many attackers, many victims) has potential to bring about sustained congestion collapse across large sections of the Internet, thus necessitating immediate action.

1.1 An Attack Based on Positive Feedback

Two significant components of transport protocols are the flow and congestion control algorithms. These algorithms, by necessity, rely on remote feedback to determine the rate at which packets should be sent. This feedback can come directly from the network [26, 19] or, more typically, from end hosts in the form of positive or negative acknowledgments. These algorithms implicitly assume that the remote entity generates correct feedback. This is typically a safe assumption because incorrect feedback rapidly deteriorates end-to-end performance [14]. *However, an attacker who does not care about data integrity could violate this assumption to induce the sender into injecting many packets into the network.* While not all of these packets may arrive at the receiver, they do serve to congest the sender's network and saturate the path from the sender to the receiver.

In this paper, we always assume that the attacker targets multiple victims, in order to maximize the damage that the attack can cause Because acknowledgment packets are relatively small (40 bytes), it is trivial for an attacker to target hundreds and even thousands of victims in parallel. In effect, not only are each victims' access links saturated, but, due to over-provisioning, higher bandwidth links in the upstream ISPs begin to suffer congestion collapse in aggregate as well. In Section 2.4, we argue that sufficiently many attackers can overwhelm backbone links in the core of the Internet, causing wide-area sustained congestion collapse.

1.2 Road map

The rest of the paper is structured as follows. Section 2 describes attack pseudo-code, implementation challenges, variants, and the distributed opt-ack attack. Section 3 discusses various bounds on the attacker's bandwidth amplification. In Section 4, we consider and evaluate possible solutions, propose one based on skipped segments, and describe its implementation. In Section 5, we present performance numbers of attacked machines with and without the proposed fix, in real world and simulated topologies. Next, we discuss related work in Section 6. We conclude with implications of the opt-ack attack and future work in Section 7. Appendix A describes the key observations required in a practical implementation of the opt-ack attack.

2 Attack Analysis

In this section we describe pseudo-code for the attack, a summary of implementation challenges, attack variants, and the details of the distributed version of the opt-ack attack. In Appendix A, we present the observations we made in implementing the attack and techniques for mitigating practical concerns.

2.1 The Opt-Ack Attack

Algorithm 1 shows how a single attacker can target many victims at once. Typically, the attacker would employ a compromised machine (a "zombie" [30]) rather than launch the attack directly.[1] Consider a set of victims, $v_1 \ldots v_n$, that serve files of various sizes. The attack connects to each victim, then sends an application level request, e.g., an HTTP GET. The attacker then starts to acknowledge data segments *regardless of whether they arrived or not* (Figure 1). This causes the victim to saturate its local links by responding faster and faster to the attackers opt-acks. To sustain the attack, the attacker repeatedly asks for the same files or iterates through a number of files.

The crux of the attack is that the attacker must produce a seemingly valid sequence of ACKs. For an ACK to be considered valid, it must not arrive before the victim has sent the corresponding packet. Thus, the attacker must estimate which packets are sent and when, based only on the stream of ACKs the attacker has already sent. At first this might seem a difficult challenge, but the victim's behavior on receiving an ACK is exactly prescribed by the TCP congestion control algorithm! The attack takes three parameters: a list of n victims, the maximum segment size (mss), and the window scaling ($wscale$) factor. In the algorithm, the attacker keeps track of each victim's estimated window (w_i) and sequence number to acknowledge (ack$_i$). The upper bound of w_i, $maxwindow$, is 65535 by default, but can be changed by the window scaling option (see Section 3). Note that the attacker can manipulate each victim's retransmission time out (RTO), because the RTO is a function of the round trip time, which is calculated by the ACK arrival rate. So, in other words, the attack can completely manipulate the victims in terms of how fast to send, how much to send, and when to time out.

Algorithm 1 –Attack($\{v_1 \ldots v_n\}$, mss, $wscale$)

maxwindow$\leftarrow 65535 \times 2^{wscale}$
$n \leftarrow |\{v_1, \ldots, v_n\}|$
for $i \leftarrow 1 \ldots n$ **do**
    connect(mss,$wscale$) to v_i , get isn$_i$
    ack$_i \leftarrow$ isn$_i + 1$
    $w_i \leftarrow$ mss
end for
for $i \leftarrow 1 \ldots n$ **do**
    send v_i data request { http get, ftp fetch, etc. . . }
end for
while true **do**
    for $i \leftarrow 1 \ldots n$ **do**
        ack$_i \leftarrow$ ack$_i + w_i$
        send ACK for ack$_i$ to v_i { entire window}
        if $w_i <$ maxwindow **then**
            $w_i \leftarrow w_i +$ mss
        end if
    end for
end while

There is a near arbitrary number of potential victims, given the pervasiveness of large files on the Internet. Any machine that is capable of streaming TCP data is a potential victim, including HTTP servers, FTP servers, content distribution networks (CDN), P2P file sharing peers (KaZaa[2], Gnutella[1]), NNTP servers, or even machines with the once common character generator ('chargen') service.

The attack stream is difficult to distinguish from legitimate traffic. To an external observer that is sufficiently close to the victim, such as a network intrusion detection system (IDS), this stream is in theory indistinguishable

[1] This attack can also be mounted if the attacker is able to spoof TCP connections, either by being on the path between the victim and the spoofed address, or from guessing the initial sequence number, but we do not further consider it.

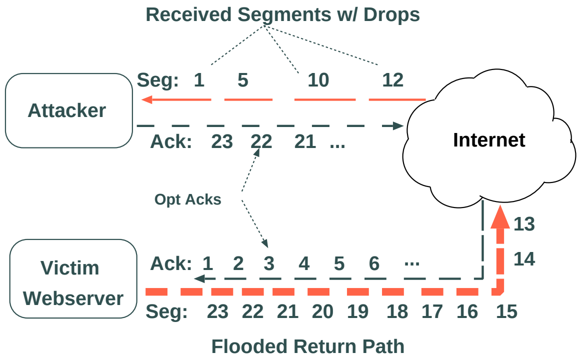

Figure 1: Opt-Ack Attack: Single Victim w/ Packet Loss (One of many victims)

from a completely valid high speed connection.[2] While it is common for IDSs to send out alerts if a large stream of packets enters the local network, the stream of ACKs from the attacker is comparatively small (see Section 3 for exact numbers). It is the stream of data *leaving* the network that is the problem.

Additionally, an attacker can further obscure the attack signature by sending acknowledgments to more victims less often, with the total amount of traffic generated staying constant. In other words, by generating less traffic per host and staying under the detection threshold, but increasing the total number of hosts *it is not locally obvious to the victims that they are participating in an DDoS attack*. As a result, short of a globally coordination, potentially through a distributed intrusion detection system, it is difficult for victims to locally determine if a given stream is malicious.

While Algorithm 1 works in theory, there are still challenges for the adversary to keep ACKs synchronized with the segments the victims actually send. We address these issues in the next section.

2.2 Implementation Challenges

The main challenge in implementing the attack is to accurately predict which segments the victim is sending and ensure that the corresponding ACKs arrive at the correct time. In Figure 1, the attacker injects ACKs into the network before the corresponding segments have even reached the attacker, so remaining synchronized with the victim can be non-trivial. Maintaining this synchronization of sequence numbers is crucial to the attack. If the attacker falls behind, i.e., it starts to acknowledge segments slower than they are sent, then the victim slows down, may time out, and the effect of the attack is reduced. Similarly, if the attacker gets ahead of the victim in the sequence space, i.e., the victim received ACKs for segments that are not yet sent, the victim ignores these ACKs and the stream stops making progress. We refer to this condition as *overrunning* the victim. Overruns can occur in three different ways: ACKs arriving too quickly, lost ACKs, and delays at the server. However, if an attacker does overrun the server, it is possible for the attacker to detect this condition and recover (Appendix A).

In accordance with RFC793 [4], Section 3.4, when the sender receives ACKs that are not in the window, it should not generate a RST, but instead an empty packet with the correct sequence number. One of the tenets of the Internet design philosophy is the robustness principle: “be conservative in what you send, and liberal in what you accept,” and it is this principle that opt-ack exploits.

There are many ways that an overrun condition may result, most common being the sending application stalls its output because it was preempted by another process. In general, there are a myriad of factors that affect the sender’s actual output rate, including: the victim’s load, application delay, the victim’s send buffer size, and the victim’s hardware buffer. However, these factors are mitigated when the number of victims is large. By sending ACKs to more victims, each individual victim receives ACKs less often. This provides more time for the victim to flush its buffers, place the sending application back into the run queue, etc.

It is worth noting that the implementation we developed is only a demonstration of the potential severity of opt-ack. It is by no means an optimal attack. There are a number of points where a more thorough attacker might be able to mount a more efficient attack. However, as we note in Section 5, the implementation is sufficiently devastating as to motivate immediate action.

In Appendix A, we discuss further strategies to mitigate and recover from overrunning the victim.

2.3 Lazy Opt-Ack

Lazy opt-ack is a variant of the standard opt-ack attack. Recall that the main difficulty in our implementation is in remaining synchronized with the sender’s sequence number. The synchronization issue can be totally avoided if the attacker ACKs any segment that it actually receives, independent of missing segments. This lazy variant is malicious in that the attacker is effectively concealing any packet loss, thereby creating a flow that does not decrease its sending rate when faced with congestion (i.e., a non-responsive flow). Since the attacker is using the actual RTT to the victim, it generates less traffic than the attack described in Algorithm 1. However, it is well known [12] that in a congested network, a non-responsive flow can cause compliant flows to back off, creating a DoS. Note

[2] Presumably, a monitoring system deployed closer to the attacker could detect the asynchrony between ACKs and data segments, but it is not practical to store per-flow state deep in the network.

that the lazy variant is different from the standard attack in that it is impossible for the attacker to overrun the victim. This observation is precisely what makes many existing solutions insufficient. The skipped segments solution we provide in Section 4.2 protects against both the lazy and standard attacks.

2.4 Distributed Opt-Ack Attack

In this section, we consider the distributed case where *multiple attackers* run the opt-ack attack in parallel, trivially, and with devastating effect. The only coordination required is that each attacker chooses a different set of victims. Because a single attacker can solicit an overwhelming number of packets (as we will see in Section 3) *a relatively small group of attackers can cause the Internet to suffer widespread and sustained congestion collapse*.

First, because opt-ack targets any TCP server, there are *millions* of potential victims on the Internet. Considering P2P file distribution networks alone, Kazaa and Gnutella have over 2 million[18, 17, 28] and 1.4 millions [20] users respectively that each host large multimedia files. While P2P nodes are typically low bandwidth home users, the popular content distributor Akamai runs over 14,000 [5] highly provisioned, geographically distributed servers.

It is not immediately clear how much traffic is necessary to adversely affect the wide-area Internet. One data point is the traffic generated from the Slammer/Sapphire worm. In [22], Moore et al. used sampling techniques to estimate the peak global worm traffic at approximately 80 million packets per second. At 404 bytes/packet, the worm generated approximately 31GB/s of global Internet traffic. Subsequent email exchanges by Internet operators [23] noted that many access links were at full capacity, and completely unusable. However, as noted in Table 1, it is theoretically possible for *a single attacker on a modem* to generate more than enough traffic to exceed this threshold using large $wscale$ values. If using large $wscale$ values were infeasible (for example, if packets containing the $wscale$ option were firewalled), then *five attackers* on T3 connections with more typical TCP options, i.e., $mss = 536$ and $wscale = 0$, would be sufficient to match the Slammer worm's traffic. If each attacker targeted sufficient number of victims, such that the load on no one victim was notably high, it would be difficult to locally distinguish malicious and valid data streams. So, unlike Slammer, there would be no clear local rule to apply to thwart the attack.

The traffic from the Slammer worm was not sufficient to push the core of the Internet into congestion collapse. Because of the inherent difficulty in modeling wide scale Internet phenomena, it is not clear how to estimate the number of opt-ack attackers required to induce such a collapse. However, a single attacker on a modem or a small number of other attackers can induce traffic loads equivalent to the Slammer worm. Recent studies[6] show that there exists networks of compromised machines("botnets") with over 200,000 nodes. Since each of these nodes represents a possible attacker, a large distributed opt-ack attack could easily be catastrophic.

3 Amplification Factor

While it is not surprising that a victim can be induced to send large amounts of data into the network, the actual opt-ack amplification factor is truly alarming. For example, an attacker on a 56Kbps modem can cause victims to push 71.2Mb/s of traffic into the network with standard TCP options. In the worst case, i.e., mss=88 and $wscale$=14, the same attacker can cause up to 1.6Tb/s of traffic to be generated. See Table 1 for other examples. While estimating these bounds is fairly simple (Section 3.1), our analysis includes the more sophisticated issues (Section 3.2) of maximum number of victims due to application time out, minimum victim bandwidths, and the time to grow the force of the attack.

3.1 Congestion Control Bounds

The upper bound on the traffic induced across all victims from a single attacker is a function of four items: the number of victims (n), and for each individual victim i, the rate at which ACKs arrive at each victim (α_i), the maximum segment size (mss_i), and the size of the victim's congestion window (w_i). Note that the attacker can use a single ACK to acknowledge an entire congestion window of packets. The number of packets from a single victim in the network at any one time is $\lfloor w_i/mss_i \rfloor$. If we assume a standard TCP/IP 40 byte header with no options and that the link layer is Ethernet (14 byte header), then the packet size is $54{+}mss_i$ bytes. The rate of attack traffic $\mathcal{T}$ in bytes/second is simply the sum across each victim of the product of the ACK arrival rate(α_i), the number of packets($\lfloor w_i/mss_i \rfloor$), and the size of each packet ($54 + mss_i$), or:

$$\mathcal{T} \;=\; \sum_{i=1}^{n} \alpha_i \times \left\lfloor \frac{w_i}{mss_i} \right\rfloor \times (54 + mss_i) \qquad (1)$$

To find the theoretic maximum possible flooding rate, $\mathcal{T}_{max}$, we have to consider the bandwidth the attacker *dedicates* to each victim (i.e., the thin dark line in Figure 1 from attacker to victim), which we denote β_i for the ith victim. If we assume β_i is measured in bytes/second, each ACK is 40 bytes, and again assume the link layer is Ethernet (14 byte header), then we find that $\beta_i = 54\alpha_i$ at maximum bandwidth. We use $\beta = \sum_{i=1}^{n} \beta_i$ to denote the attacker's total attack bandwidth to all victims, and for simplicity assume that each victim has the same mss and w_i,

Attacker Speed ($\mathcal{T}_{max}$)	$mss = 536$ $wscale = 0$	$mss = 88$ $wscale = 0$	$mss = 536$ $wscale = 14$	$mss = 88$ $wscale = 14$
Multiplier $\beta = 1$	1336 B/s	1958 B/s	20.9 MB/s	30.6 MB/s
Modem $\beta = 7000$	8.9 MB/s	13.1 MB/s	142.7 GB/s	209.2 GB/s
DSL $\beta = 16000$	20.4 MB/s	29.9 MB/s	326.1 GB/s	478.1 GB/s
T1 $\beta = 193000$	245.8 MB/s	360.5 MB/s	3.84 TB/s	5.63 TB/s
T3 $\beta = 5625000$	7.0 GB/s	10.3 GB/s	112.0 TB/s	164.1 TB/s

Table 1: Maximum theoretical flooding for various attacker speeds and options. MB/s refers to 2^{20} bytes/second, etc.

i.e., $\forall i; mss_i = mss$ and $\forall i; w_i =$ maxwindow. Thus, substituting β, mss, and maxwindow into (1) and rearranging produces:

$$\mathcal{T}_{max} = \left\lfloor \beta \times \text{maxwindow} \times \left(\frac{1}{mss} + \frac{1}{54} \right) \right\rfloor \quad (2)$$

As noted before, the maximum congestion window ($maxwindow$) is typically 65535. For a wide area connection, a typical value for mss would be 536.[3] Substituting these values into (2) produces $\mathcal{T}_{max} = 1336\,\beta$. Thus, using typical values, an attacker has an amplification factor of 1336. In real world terms, that means an attacker on a 56 Kilo-*bits*/s modem ($\beta = 7000$ B/s) can in theory generate 9,351,145 B/s or approximately 8.9MB/s of flooding summed across all victims. This value is more than the capacity of a T3 line, and close to the theoretical limit of a 100Mb Ethernet connection. See Table 1 for the amplification factor, and more examples.

For non-standard values of mss and $maxwindow$, the amplification factor of the opt-ack attack is significantly magnified. Recall from RFC 793 [4] that the mss is a 16 bit value set via TCP option by the receiver (the attacker) in the SYN packet. Looking at (2), decreasing mss makes packets smaller, but increases the number of packets sent, for a net increase in $\mathcal{T}_{max}$. While it is already well known [27] that transferring large files with a low mss value can create denial of service conditions, the damage is significantly amplified when coupled with the opt-ack attack. As noted by Reed, the minimum mss is highly system dependent with values varying from 1 to 128 for popular OSes. For example, Windows 2000 and Linux 2.4 have a minimum mss of 88, whereas Windows NT4's is 1. Reed also noted that at extremely low values of mss, the server can become CPU-bound because of high context switching from fielding too many interrupts.

In addition, RFC 1323 [15] defines the $wscale$ TCP option to increase $maxwindow$. The attacker can use the $wscale$ option to scale the congestion window by a factor of 2^{14}, increasing $maxwindow$ to 65535×2^{14} or approximately 10^9 bytes. As shown in Table 1, the effect of window scaling on $\mathcal{T}_{max}$ is dramatic. Specifically, a malicious connection with mss=88 and $wscale$=14 can reach a theoretical amplification factor of 32,085,228 or over *32 million*. With this level of amplification, it is possible for an attacker on a modem targeting many victims to induce more traffic than the Slammer worm (Section 2.4).

[3] Another typical value is mss=1460, but the effect on $\mathcal{T}_{max}$ is minimal

3.2 Application Timeouts and Growing the Congestion Window

Fortunately, there is a significant difference between the theoretical and practical effects of the opt-ack attack. First, there is a limit to the number of victims an attacker can target at once. From Algorithm 1, the attacker needs to connect to each victim, and retrieve the initial sequence number (ISN) for each connection *before* sending the application data request (Line 8), e.g., an http get or ftp file request. Note that it is very difficult for an attacker to learn new ISNs while attacking other hosts, because its incoming links are saturated. Thus, the attacker must connect to the entire victim set, learn their ISNs, and then launch the attack. However, if the attacker targets too many nodes, the loop in Algorithm 1 at line 3 will take too long, and the first victim will timeout at the application level before receiving its data request. If victims timeout before the request is sent, then the connection is dropped and the attack foiled. Note that the minimum time for the attacker to complete a TCP connection is the time to send the SYN packet and the time to send the ACK packet ($2 \times (40 + 14) = 108$ bytes). This assumes that the attacker efficiently interleaves SYNs and ACKs to multiple victims, such that each victim's time to respond with the SYN-ACK is not a limiting factor. Then, using the minimum connection time and the application timeout (ATO) value, we can calculate maximum possible number of victims as follows:

$$\text{max victims} = \text{ATO} \times \frac{\beta}{108} \quad (3)$$

Realistic ATO values are highly application dependent, and even within applications, the timeout value is tuned to the specific environment and workload. For example, a survey among an arbitrarily chosen set of popular websites showed ATO values for http ranged from 135 seconds (www.google.com) down to 15 seconds (www.cnn.com). As a further data point, the popular web server package Apache has a default timeout of 300 seconds. However, even with an ATO of 15 seconds, an attacker on a home DSL line (128Kbps up-link, $\beta = 16000$) can attack 2307 victims in parallel.

Another limitation is that it is not possible to create more flooding than the sum of the victims' up-links capacities. A direct implication of (3) is that each victim must have a minimum amount of bandwidth in order for an attack to reach the rates described in Table 1. To calculate the minimum bandwidth of each victim, we divide (2) by (3) and produce:

$$\text{min victim bandwidth} = \frac{\text{maxwindow}}{\text{ATO}} \times \left(\frac{108}{mss} + 2\right) \tag{4}$$

In other words, the same attacker on a home DSL line also needs each of the 2307 victims to have bandwidth in excess of 220MB/s in order to achieve 478.1GB/s in flooding as described in Table 1 (ATO=15, $mss = 88$, $wscale = 14$). Obviously, this is not immediately practical. However, if any victim is below the minimum bandwidth from (4), the result is simply that the sender saturates its outgoing link, which is sufficiently devastating to the victim's local network.

The last bound on the amplification is the time to grow the congestion window. The attacker must send sufficient number of ACKs to each victim in order to increase the congestion window to from its initial value (one mss) to its maximum value($maxwindow$). By Algorithm 1, as the number of victims increase, the time between ACKs sent to an individual node diminishes. Thus, we can calculate the minimum time required for the attack to reach maximum effect:

$$\text{min time} = \frac{54 \times \text{maxwindow} \times n}{\text{mss} \times \beta} \tag{5}$$

The values in Table 1 are upper bounds on $\mathcal{T}$ and may in fact never be achieved in practice. Other factors such as the victims' TCP send buffer size, outgoing bandwidth, and processing capacity affect the rate at which traffic is produced (as discussed in Appendix A). In Section 5, we show that our implementation achieves nearly 100% of $\mathcal{T}_{max}$ in simulation.

4 Defending against Opt-Ack

In this section, we present a simple framework for evaluating different defense mechanisms against the opt-ack attack, and evaluate potential solutions within that framework. Finally, we present one particular solution, randomly skipping segments, that efficiently and effectively defends against opt-ack. We also describe an implementation of randomly skipped segments in detail.

4.1 Solutions Overview

Any mechanism that defends against opt-ack should minimally possess the following qualities:

1. **Easy to Deploy** Due to the severity of the attack, any solution should be practically and quickly deployable in the global infrastructure. Minimally, the solution should allow incremental deployment, i.e., unmodified clients should be able to communicate with modified servers.

2. **Efficient** Compliant (i.e., non-attacking) TCP streams should suffer minimal penalty under the proposed solution. Also, low power embedded network devices do not have spare computational cycles or storage space. Because the problem is endemic to all implementations, the solution needs to be efficient on all devices that implement TCP.

3. **Robust** Any fix needs to defend against all variants (Section 2.3) of the opt-ack attack.

4. **Easy to Implement** This is a more pragmatic goal, leading from the observation that TCP and IP are pervasive, and run on an diverse range of devices. Any change in the TCP specification would affect hundreds (or thousands) of different implementations. As such, a simpler solution is more likely to be implemented.

In the rest of this section, we describe a number of possible defenses against opt-ack, and present a summary of solutions in Table 2.

4.1.1 Secure Nonces

One possible solution is to require that the client prove receipt of a segment by repeating an unguessable nonce. Assume each outgoing segment contains a random nonce which the corresponding ACK would have to return in order to be valid. Savage [29] et al. improve on this solution with *cumulative* nonces. In their system, the response nonce is a function of all of the packets being acknowledged, i.e., a cumulative response, ensuring that the client actually received the packets it claims to acknowledge.

Unfortunately, cumulative nonces are not practically deployable. They requires both the client and server to be modified, preventing incremental deployment. If deployment was attempted, updated servers would be required to maintain backward compatibility with non-nonce enabled clients, until all client software was updated. As a result, updated servers would have to chose between being vulnerable to attack or compatibility with unmodified clients. Additionally, nonces require additional processing and storage for the sender. Calling a secure pseudo-random generator once per packet could prove expensive for devices with limited power and CPU resources, violating our efficiency goal.

To aid deployment, one could consider implementing nonces in existing, unmodified clients via the TCP timestamp option. The send could replace high order bits of the timestamp with a random challenge, and any non-malicious client which implemented TCP timestamps would respond correctly with the challenge. If a client did not implement timestamps, the server could restrict throughput to something small, e.g, 4Kb/s. While this improves on the deployment of nonces, this solution still has problems. First, it loses the critical cumulative ACK property of Savage's solution. That is, an acknowledgment for a set of packets does not necessarily imply that all packets in the set were received, which opens itself to the lazy opt-ack attack. Second, as we discuss in Section 4.1.3 below, bandwidth caps are not effective.

4.1.2 Require ACK Alignment

One aspect of TCP that makes the opt-ack attack possible is the predictability of the ACK sequence. Furthermore, because communication is a stream, the client can in theory acknowledge the bytes anywhere in the sequence, not just along packet boundaries. Clark [11] cites the ability to retransmit one large packet when a number of smaller packets are lost as a main benefit of this. In practice, with large buffers and client-side window scaling, most implementations send only packet-aligned acknowledgments. We could use this insight to require clients to acknowledge only along packet boundaries, and then add a small, unpredictable amount of noise to the packet size. For example, with equal probability, the server could send a packet of size mss or of size $(mss - 1)$. In this way, a client that actually received the packet would get information that a opt-ack attacker does not have: the actual packet size. Only a client that actually receives all of the packets could continue to correctly ACK them probabilistically over time. The noise could be generated pseudo-randomly as a function of the sequence, so storing per-outstanding-packet state at the server could be avoided.

ACK alignment suffers from many of the same problems as non-cumulative secure nonces. Specifically, it is not secure against the lazy variant of opt-ack, and ACK alignment could be expensive for low powered devices. In addition, network devices, such as network address translation (NAT) box translating FTP or a firewall, could change the size of or split the packet in flight. Any such change in the packet size would result in false positives, which are unacceptable.

4.1.3 Bandwidth Caps

The obvious solution to an attacker consuming too many resources, as is the case with the opt-ack attack, is to limit resource consumption. Conceivably, this could be done at the server with a per IP address bandwidth cap, but unfortunately this is not sufficient. First, any restriction on bandwidth can simply be over come by increasing the number of victims. Suppose for example, that each victim sets the policy that no client can use more than a fraction $c \in (0, 1]$ of their bandwidth. Then the attacker need simply increase the number of victims by $1/c$ to maintain the same total attack traffic. Further, bandwidth caps interfere with legitimately fast clients, violating our efficiency goal.

4.1.4 Network Support

Since the opt-ack attack stream acts essentially as a non-responsive flow, one possible defense would be to implement fair queuing or "penalty boxes" in the network. As [12] notes, this is not a new problem, and there exists a wealth of research on the subject [8, 13, 9, 10]. A similar solution would be force flows that cause congestion to solve puzzles[32] in order to maintain their rate. However, these solutions are not currently widely deployed and the cost of doing so would seem prohibitive.

4.1.5 Disallow Out of Window ACKs

A straightforward solution is to change the TCP specification to disallow out of window ACKs. Recall from Section 2.2 that our implementation runs the risk overrunning the victim. If a victim sent a reset, terminating the connection, upon receipt of an out of window ACK, the opt-ack attack would be mitigated. However, this is not a viable solution as this opens non-malicious connections to a new DoS attack. A malicious third party could inject a forged out of window ACK into a connection, causing a reset [33]. Because the ACK is out of window, there would be no need to guess the sequence space. Also, compliant receivers can send out of window acknowledgments due to delays or packet reordering. For example, suppose ACKs for packets numbered 2 and 3 are sent but received in reverse order. The ACK for packet 3 would advance the window, and then the ACK for packet 2 would be and out of window ACK, causing a RST.

4.1.6 Random Pauses

As described in Section 2.2, the main difficulty in the implementation was to keep the attacker's sequence numbers synchronized with what the server was sending. Thus, one

Solution	Efficient	Robust	Deployable	Simple	Change TCP Spec.
Cumulative Secure Nonces	yes	yes	no	yes	client & server
Secure Nonces w/ timestamps	yes	no	yes	yes	server only
ACK Alignment	yes	no	yes	yes	server only
Bandwidth Caps	no	no	yes	yes	no
Network Support	yes	yes	no	no	no
Random Pauses	no	no	yes	yes	server only
Skipped Segments	yes	yes	yes	yes	server only

Table 2: Summary of Defenses to Opt-Ack Attack

way of thwarting the attacker would be for the server to randomly pause. A client correctly implementing the protocol will reciprocate by pausing with the server and waiting for more data. On the other hand, an attacker will continuously send ACKs for packets not yet sent, exposing the attack. This solution does not prevent against the lazy variant of the opt-ack attack. Also, if the server applied this pausing test too often, performance could suffer significantly. Our final proposed solution expands on the random pausing idea with additional robustness and minimal performance penalty.

4.2 Proposed Solution: Randomly Skipped Segments

The main problem with the random pause solution is the efficiency penalty to non-malicious clients. Instead of pausing, we propose the server *skip* sending the current segment, and instead send the rest of the current window. Note that this is equivalent to locally, *intentionally* dropping the packet. A client that actually gets all of the packets, save the skipped one, will start re-ACKing for the lost packet, thereby invoking the fast retransmit algorithm. However, an attacker, because it does not have a global view of the network, cannot tell where along the path a given packet was dropped, so it cannot tell the difference between an intentionally dropped packet and a packet dropped in the network by congestion. Thus, an attacker will ACK the skipped packet, alerting the server to the attack. Note that usually fast retransmission indicates network congestion, so the congestion window is correspondingly halved. However in this case, retransmission was not invoked due to congestion in the network, so the sender should not halve the congestion window/slow start threshold as it typically would. Given that most modern TCP stacks implement selective acknowledgments (SACK)[21], this solution is significantly more efficient than randomly pausing (see Section 5 for performance). *The only penalty applied to a conforming client is a single round trip time in delay.*

To determine how often to apply the skipped packet test, we maintain a counter of ACKs received. Once a threshold number of ACKs are received, the skip test is applied. It is important that the threshold be randomized, as the security of this system requires that the attack not predict which segment was be skipped. However, there is an obvious trade off in where to make the skipped packet threshold. If it is too low, the server will lose efficiency from skipping packets too often. Setting the threshold too high allows the attacker to do more damage before being caught (see Section 5 for an exploration of this trade-off). Our solution is to chose the threshold uniformly at random over a configurable range of values.

This simple skipped segment solution meets all of our goals. It is efficient: compliant clients suffer only one round trip time in delay, the computational costs consist of keeping only an extra counter, and the storage costs are trivial (5 bytes per connection, described below). The skipped packet solution is robust against the variations of the attack described in Section 2.3, because it inherently checks whether a client actually received the packets. This solution is a local computation, so it needs no additional coordination or infrastructure, i.e., the deployment requirements are met. Best of all, it is transparent to unmodified clients, allowing for incremental deployment.

4.3 Skipped Packet Implementation

We implemented the skipped packet solution for the Linux 2.4.24 kernel. The total patch is under 200 lines (including comments, prototypes, and headers), and was developed and tested in under one week's time by someone previously unfamiliar with the Linux kernel. We add two entries to the per connection state (struct tcp_opt): opt_ack_mode (1 byte) and opt_ack_data (4 bytes). Further, we add 3 global configuration variables: $sysctl\_tcp\_opt\_ack\_enabled$, $sysctl\_tcp\_opt\_ack\_min$, and $sysctl\_tcp\_opt\_ack\_max$. When a new connection is created, opt_ack_mode is initialized to $OPT\_ACK\_MODE\_COUNTDOWN$, and opt_ack_data is set to a number uniformly at random between $sysctl\_tcp\_opt\_ack\_min$ and $sysctl\_tcp\_opt\_ack\_max$, inclusive. With each successful ACK, opt_ack_data is decremented, until it reaches zero. Upon opt_ack_data reaching zero, we set opt_ack_mode to $OPT\_ACK\_MODE\_SKIP$, update the send head pointer to the next block (skipping the segment), and save the sequence number of the segment skipped into opt_ack_data. A compliant client will ACK the beginning of the hole (i.e. the sequence in opt_ack_data), where a malicious attacker will ACK a segment past the hole. If the client ACKs a segment before the hole, we leave the test in place until another ACK arrives. If the client ACKs past the hole, it fails the test: we reset the connection and log a message to the console. In implementation, we use parameters $sysctl\_tcp\_opt\_ack\_min = 100$ and $sysctl\_tcp\_opt\_ack\_max = 200$, as suggested by our evaluations in Section 5.3. Last, under Linux, the retransmission code automatically handles resending the skipped segment for clients that correctly ACK the beginning of the hole.

The description of the fix is complete, except for a few additional details. If a timeout occurs in the middle of the skip test, we need to reset the threshold countdown, and go back to mode $OPT\_ACK\_MODE\_COUNTDOWN$. The reasoning is this: if the segment before the hole is lost, and there are no segments after the hole (or they are all lost), then the client will not ACK the beginning of the hole, until after the retransmit. However when a timeout occurs, the retransmit code might resend the skipped segment, negating the test. Resetting the threshold counter and changing the mode in this obscure case solves this problem.

Also, to insure that the randomly skipped segments solution does not introduce a new DoS attack, we must ignore out of window ACKs during the skipped segments test. Otherwise, it might be possible for a malicious node to convince a server that a benevolent client was performing an opt-ackattack.

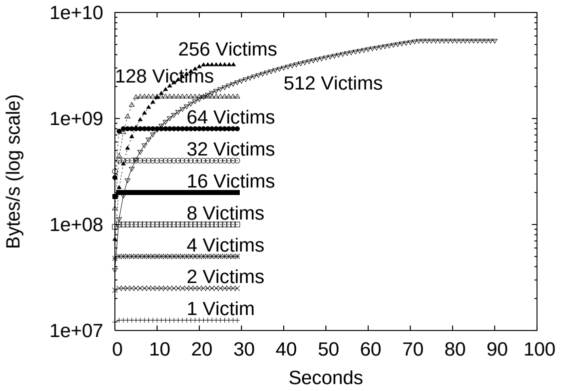

Figure 2: Maximum Traffic Induced Over Time; Attacker on T1 with mss=1460,$wscale$=4

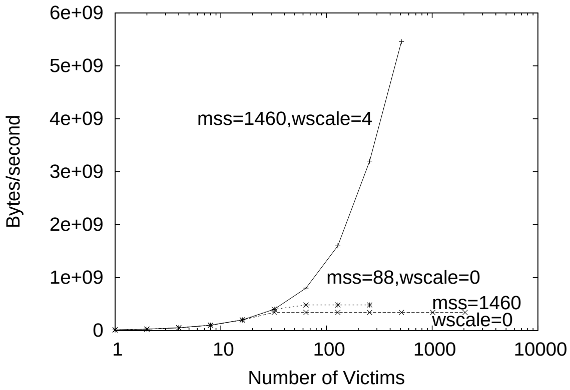

Figure 3: Maximum Traffic Induced By Number of Victims; Attacker on T1

5 Attack Evaluation

We evaluate the feasibility and effectiveness of the opt-ack attack in a series of simulated, local area, and wide area network experiments. In the first set of simulations, we determine the total amount of traffic induced by the opt-ack attacks. Next, we determine the effect of the attack on other (honest) clients trying to access the victim. We also present results for the amount of traffic (described in Section 3) our real world implementation actually achieves across a variety of platforms and across different file sizes. Finally, in Section 5.3, we evaluate the efficiency of our skipped segment solution.

5.1 Simulation Results

We have implemented the opt-ack attack in the popular packet level simulator ns2 and simulate the amount of traffic induced in various attack configurations. In each experiment, there is a single attacker and multiple victims connected in a star topology. Each victim has a link capacity of 100Mb/s, and all links have 10ms latency (the choice of delay is arbitrary because it does not affect the attack). We vary the number of victims, and the mss and $wscale$ of the connection. The attacker makes a TCP connection to each victim in turn, and only sends acknowledgments once all victims have been contacted. Victims are running the "Application/FTP" agent, which uses an infinite stream of data.

In Figure 2, we show the sum of the attack traffic generated over time with variable numbers of victims. In this experiment, the attacker is on a T1 (1.544Mbs) and uses connection parameters mss=1460 and $wscale$=4 ($maxwindow$=1048576). When the number of victims is less than 512, the amount of flooding is limited by the sum of the bandwidths of the victims, as predicted by Equation 4 in Section 3. The amount of traffic doubles as the number of victims double until 512 victims. The number of victim's increases, the attack takes longer to achieve full effect as predicted by Equation 5. The case with 512 vic-

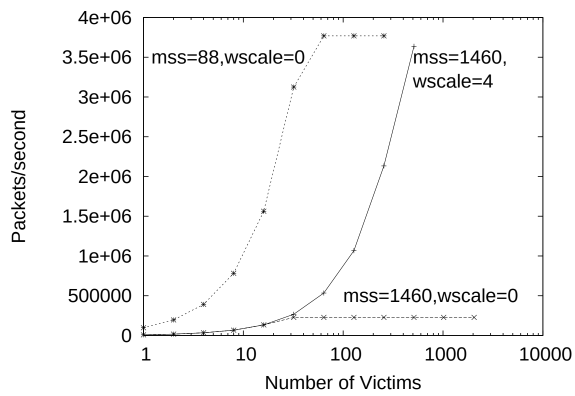

Figure 4: Maximum Packets Induced By Number of Victims; Attacker on T1

tims took 73 seconds to reach it peak attack rate, while all others did so in under 30 seconds. At 512 victims, the simulation achieves 99.9% of the traffic predicted by Equation 2.

As shown in Figure 2, once the attack's maximum effect is reached, it can be sustained indefinitely. In Figure 3 and Figure 4, we show the maximum traffic induced as we vary the number of victims, mss and $wscale$ for bytes/second and packets/second respectively. As predicted by Section 3, attackers with a lower mss produce more traffic than one with a higher value. Likewise, an increased $wscale$ has a dramatic increase in the total traffic generated.

Due to CPU and disk space limits, we were not able to simulate more than 512 victims for all parameters, or $wscale$ values above 4, despite the fact that our simulation machine was a dual processor 2.4Ghz Athlon-64 with 16GB ram and 300GB in disk.

5.2 Real World Implementation

In order to validate the attack, we implemented it C and experimented on real machines in a number of network settings. We measure the effect of the attack on a single victim and the actual bandwidth generated from a single victim running various popular operating systems.

It should be noted that we did not experiment with multiple attackers or multiple victims due to real world limitations of our test bed. Our experiments with a single attacker and single victim were sufficient to cause overwhelming traffic on our local networks. It would be irresponsible and potentially illegal to have tested the distributed attack on a wide-area test bed (e.g., PlanetLab[25]), and even our simple one attacker-one victim wide-area experiments caused network operators to block our experiments.[4]

5.2.1 Single Victim DoS Effect - Lan and Wan

Experiment	Average (sec)	Dev.	Increase
No Attack	89.11	0.007	1
LAN Attack	1552.03	141.76	17.42
WAN Attack	779.93	139.32	8.75

Figure 5: Average Times with Deviations for a Non-malicious Client to Download a 100MB File

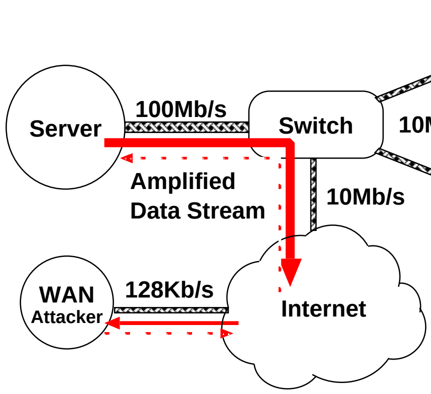

Figure 6: Topology for Experiments

This experiment measured the effect on a third party client's efficiency in downloading a 100MB file from a single victim during various attack conditions. We repeated this experiment with no attacker, with an attacker on the local area network, and with an attacker across the Internet (see Figure 6). The local area attacker was a dual processor Pentium III running Linux with a 10Mb Ethernet card, while the WAN attacker was a 100Mhz Pentium running GNU/Linux on an asymmetric 608/128 Kb/s downstream/upstream residential DSL line. The latency on the WAN link varied over time, with a average RTT of 13.5ms.

A typical web server runs on a fast local area network, which connects to a slower wide area network. In order to emulate this bottleneck, and also to safeguard against saturation of our production Internet connection, we connected our test web server to the world via a 10Mb connection on a Cisco Catalyst 3550 switch. Furthermore, both LAN and WAN attackers were configured to use $TargetBandwidth$ of 10^9 bytes/second, and $\beta = 16000$ bytes/s as their local bandwidth setting (see Appendix A for description). The intuition is that the LAN and WAN attackers should be equally capable with respect to their available bandwidth, but the WAN attacker must compensate for more end-to-end jitter and delay. Each run used mss=536 and $wscale$=0, i.e., typical values for Internet connections. Each experiment was repeated 10 times and the values averaged. The numbers were measured with a

[4] Incoming traffic to one author's home DSL IP address was temporarily blocked as a result of these experiments. This did not serve to stop the attack, as the outbound ACKs could still be sent. However, this served as evidence that we should cease the experiment.

command line web client (similar to *wget*) specially instrumented to measure bandwidth at 10 ms intervals. We present the results from these experiments in Table 5. The "Increase" column refers to the increase in time relative to the "No Attack" baseline.

The effect of the attack is significant. The 100MB file takes on average 17.42 and 8.75 times longer to download under LAN and WAN attack, respectively. We believe that the time difference between the WAN and LAN attacks is due to the increased jitter of the wide area Internet, and the increased standard deviation in the results supports this. This variability makes keeping synchronizing with the victim more difficult due to the buffered ACK problem, as described in Appendix A. However, more advanced attackers could target more victims (Section 2.2) or potentially employ more sophisticated segment prediction to increase the effectiveness of the attack.

We also re-ran the same set of experiments with a set of hubs in place of the switch, effectively removing queuing from the system. The times to download the 100MB file while under attack were reduced to 5 times and 4.5 times the baseline for LAN and WAN attackers, respectively. In other words, having queuing on the bottleneck link significantly *increased* the damage from the attack. We surmise this is because the opt-ack attacker used $mss = 536$ and the non-malicious client, since it was on local Ethernet, used $mss = 1448$. Once the queue was full, the switch could service two of the attack packets before there was room for a legitimate (i.e. destined to the non-malicious client) packet. Effectively, the higher rate of smaller packets caused the switch to drop more non-malicious/legitimate packets. Removing the queue from the system reduced the amount of dropped legitimate packets, therefore increasing non-malicious throughput.

5.2.2 Amplification Factors

To evaluate the potential effectiveness of the distributed opt-ack attack, we measure the amount of traffic that our implementation code can induce in a single victim. In this experiment, we use the LAN attacker, as above, to attack various operating systems including GNU/Linux 2.4.24, Solaris 5.8, Mac OS X 10.2.8, and Windows XP with service pack 1. For this experiment, instead of a web server, each victim ran a program that streamed data from memory. This was done to remove any potential application-level bottlenecks from the experiment. As above, the attacker used parameters $\beta = 16000$, $mss = 536$, and $wscale = 0$. We measured the bandwidth in one second intervals using a custom tool written with the libpcap library. Each experiment in Table 3 was run 10 times, averaged, and is shown as an amplification factor of the attacker's used local bandwidth.

OS	Avg. KB/s	Dev.	Amplification
Linux 2.4.24	3931.93	1102.38	251.6
Mac OSX	806.2	258.1	51.6
Solaris 5.8	3150.6	1301.1	201.6
Windows XP	640.62	378.85	41.0

Table 3: Average bytes/s of Induced Flooding, Standard Deviation, and Amplification Factor of Attacker's Bandwidth

File Size	Time(s)	Dev.	Factor Increase
No Attack	89.11	0.007	1
100MB File	1552.03	141.76	17.42
10 MB File	281.00	9.81	3.15
1 MB File	152.87	21.48	1.75
512 KB File	106.63	9.03	1.20

Table 4: Average Times for Client to Download a 100MB File, With Attacker Downloading Various-Sized Files

We believe that the variation in amount of flooding by OS is due to the lack of sophistication of our attack implementation. The amplification factor for Linux is 251.6 times the used bandwidth, which translates to $251.6/1336$ or approximately 18% of the theoretical maximum traffic, $\mathcal{T}_{max}$. This low number is in part because the implementation sends four ACKs per window (as described in Appendix A), which alone limits the attack to 25% of $\mathcal{T}_{max}$.

5.2.3 Smaller Files

In the first set of experiments, we assumed the victim served a 100MB file for the attacker to download. While there are files of this size and larger on the web (Windows XP service pack 2 is 272MB and heavily replicated), we repeated the experiment with smaller file sizes. The test bed is exactly as above (Figure 6) with the LAN attacker and queuing. Again, the non-malicious client downloaded a 100MB file from the victim. In this experiment, we vary the size of the file the attacker downloads. The results are presented in Table 4. As expected, smaller files are less useful for the attacker. However, even 10MB files cause the client to slow down by a factor of 3.15, so smaller files can still cause some damage.

Clearly, these results depend upon the attack implementation described in Appendix A, and there are inefficiencies in our implementation that can be improved upon. For example, the implementation code creates a new TCP stream each time a download is complete, and starts again in a loop. An easy optimization would have been to take advantage of HTTP's persistent connections and download multiple files on the same stream. However, the results presented here are sufficiently motivating. As above, each

Experiment	Time(s)	Deviation	%
Unfixed	89.136	0.007	100%
Fixed: 10-20	89.623	0.980	99.457%
Fixed: 1-200	89.158	0.0234	99.975 %
Fixed: 100-200	89.167	0.0256	99.965 %

Table 5: Time to Download a 100MB File for Various Fix Options - *SACK Enabled*

Experiment	Time(s)	Deviation	%
Unfixed	89.143	0.0152	100%
Fixed: 1-200	90.048	0.3960	98.994%
Fixed: 100-200	89.145	0.0111	99.998%

Table 6: Time to Download 100MB File for Various Fix Options - *SACK Disabled*

data point represents the average of 10 experiments. The "Factor Increase" column refers to the increase relative to the "No Attack" baseline in Figure 5.

5.3 Performance of Skipped Segments Solution

In the final experiment, we evaluate the efficiency of our proposed randomly skipped segments solution. Specifically, we measure the time for a non-malicious client on the LAN to download a 100MB file from the server with and without the fix, with and without selective acknowlegement (SACK) enabled on the client, and with various threshold values for the fix. The download times were measured with the UNIX *time* utility. Each experiment was run ten times, the results were averaged, and they are presented in Tables 5 and 6, with and without SACK respectively. The two numbers in the first column of each table refer to the threshold values used for $sysctl\_tcp\_opt\_ack\_min$ and $sysctl\_tcp\_opt\_ack\_max$ in each experiment.

The results show that the performance hit from the proposed fix is negligible for most parameters. Even when we chose the threshold to be intentionally inefficient, i.e., skipping a segment every 10 to 20 ACKs, the fix maintained 99.457% efficency. We found that varying $sysctl\_tcp\_opt\_ack\_min$ value had little effect when combined with SACK, but made a 1% difference without SACK. We believe the loss from skipping segments every 100-200 ACKs, i.e., less than 0.1% with or with SACK, is an acceptable price for defeating this attack.

6 Related Work

There is a long history of denial of service attacks against TCP, which we divide broadly into brute force attacks and more efficient attacks.

6.1 Brute Force DoS Attacks

The salient feature of brute force attacks is the fact that it is incumbent upon the attackers to provide the resource that ultimately overloads the victim. Example attacks include bandwidth flooding, connection flooding, and Syn flooding. The commonality among these attacks is that the attackers must be capable of draining more of a precious resource, be it bandwidth, file descriptors, or memory, than the victim's capacity. One possible defense against these attacks is to obtain more of the resource, i.e. buy more memory or lease more bandwidth.

The danger of the opt-ack attack is that the victim's own resources are being turned against them. If the victim adds more bandwidth capacity, then the attacker can then use the additional bandwidth to generate more traffic. While there is a bound, $\mathcal{T}_{max}$, to the traffic the attacker can induce (see Section 3), the victim is not necessarily safe if it secures more than $\mathcal{T}_{max}$ in capacity. Increasing the victim's local capacity pushes the bottleneck link further into the network, injecting more traffic into the Internet backbone.

6.2 Efficient Attacks

We refer to efficient attacks as those that require little resources from the attacker but result in victims introducing large amounts of resources to the network, essentially performing their attack for them.

6.2.1 Smurf Attack

A smurf attack [3] consists of forging a ping packet from the victim to the broadcast address of a large network. In this way, a single packet is amplified by the size of the network, and redirected at the victim. A variant of this attack is to forge the ping from the broadcast address of the victim, forcing the victim's switches to do more work in duplicating the packet.

The amplification aspects of this attack are similar to the opt-ack attack. However, the attack signature of smurf makes it easy to detect and defend against: simply block traffic from a broadcast address or rate limit $ICMP\ ECHO$ traffic at the border router. In contrast, opt-ack is not known to have an obvious attack signature, and most site policies would not allow blocking TCP traffic.

6.2.2 Shrew Attack

The attack most similar to opt-ack is the Shrew [16] attack, in that it also attempts to exploit of TCP congestion control. In Shrew, an attacker sends traffic directly to the receiver/victim in short bursts, trying to force a retransmission due to packet loss. If the bursts are timed correctly, the sender's RTO period can be abused such that each retransmission coincides with another burst, and thus a DoS condition is created. Analysis shows that a simple square wave pattern of bursts forces the sender's RTO period to synchronize with the bursts. Further, the bursts can be sufficiently infrequent such that the average rate would not alert a potential intrusion detection system.

Despite these similarities, the two attacks are quite different. In Shrew, it is the receiver who is attacked directly, where with opt-ack, it is the sender who is attacked which indirectly impacts all receivers. Also, Shrew assumes that the attacker has enough bandwidth to directly force packet loss. This is reasonable when the path from the attacker to the receiver includes the bottleneck link from sender to receiver, but this not always the case. In contrast, opt-ack makes no such assumptions. With opt-ack, it is the sender's first-hop link that is saturated, which thereby becomes the bottleneck for all connections (assuming a single homed sender). Further, even a relatively weak opt-ack adversary, such as an attacker on a modem, presents a serious threat to a comparatively high bandwidth server.

Also, it seems that the main advantage of the Shrew attack is that the average attack traffic rate is low. However, if the attack became popular, it seems intuitive that intrusion detection systems could easily adapt by examining the maximum traffic rate in addition to the average traffic.

Lastly, as we note in Section 5, elements of the two attacks can be combined. An intelligent opt-ack attacker can vary the rate of ACKs sent to cause the return stream to regularly burst like the Shrew attack. Using this method, it is apparent that more damage can be generated.

6.2.3 Misbehaving Receivers

As previously mentioned, Savage et al.[29] discovered the opt-ack attack as a method for misbehaving receivers to get better end-to-end performance. While they suggest that opt-ack can be used for denial of service, they did not investigate the magnitude of the amplification the attack can achieve. As a result, their cumulative nonce solution to the opt-ack attack does not consider global deployment as a goal. In this work, through analysis and implementation, we have shown that opt-ack is a serious threat. Further, we have engineered an efficient solution that does not require client-side modification, and thus is more readily deployable.

6.2.4 Reflector Attacks

In [24], Paxson discusses a number of attacks where the initiator can obscure its identity by "reflecting" the attack off non-malicious third parties. As a general solution, Paxson suggests upstream filtering based on the attack signature with the assumption that it is not possible to overwhelm the upstream filter with useless data. The work specifically mentions that if the attacker is able to guess the ISN of the third party, it is possible to mount a blind opt-ack attack against an arbitrary victim. No analysis is made of the amount of the amplification from the opt-ack attack, nor is it immediately clear what filter rules could be applied to arbitrary TCP data.

7 Discussion and Conclusion

We have described an analysis of the opt-ack attack on TCP and demonstrated that amplification from the attack makes it dangerous. We have also engineered an efficient skipped segments defense against attacks of this type that allows for incremental deployment. The opt-ack attack succeeds because it violates an underlying assumption made by the designers of TCP: that peers on the network will provide correct feedback. This assumption holds when clients are interested in receiving data, since false feedback will usually lead to worse end-to-end performance. However, the opt-ack attack shows that if malicious nodes do not care about data transfer integrity, they can cause widespread damage to other clients and to the stability of the network.

Since opt-ack violates an underlying assumption upon which TCP is based, we believe a proper solution for the opt-ack attack involves changing the TCP specification. Although new features can be added to TCP (e.g., cumulative nonces) to ensure the receiver TCP is in fact receiving all of the segments, this type of solution is difficult to deploy because it requires client modification. The skipped segment solution presented here requires modification of only high capacity servers, and is thus more readily deployable. In this paper, we have described different mechanisms that can be used to defend against opt-ack attacks. We recommend a specific change to the TCP specification that we have shown to be easy to implement, efficient for fast connections, and which does not burden resource-poor hosts.

References

[1] Gnutella Home Page. See `http://gnutella.wego.com`.

[2] See `www.kazaa.com`.

[3] Smurf Attack: See `http://www.cert.org/advisories/CA-1998-01.html`.

[4] RFC793: Transmitions Control Protocol, September 1981.

[5] http://www.akamai.com/en/html/technology/overview.html.

[6] T. H. P. R. Alliance. Know your enemy: Tracking botnets. http://www.honeynet.org/papers/bots/.

[7] M. Allman and V. Paxson. On Estimating End-to-End Network Path Properties. pages 263–274. SIGCOMM, 1999.

[8] B. Braden, D. Clark, and S. Shenker. Rfc1633: Integrated Services in the Internet Architecture: an Overview.

[9] W. chang Feng, D. Kandlur, D. Saha, and K. Shin. BLUE: A New Class of Active Queue Management Algorithms. Technical Report CSE-TR-387-99, 15, 1999.

[10] W. chang Feng, D. Kandlur, D. Saha, and K. Shin. Stochastic Fair Blue: A Queue Management Algorithm for Enforcing Fairness. pages 1520–1529. INFOCOM, 2001.

[11] D. D. Clark. The Design Philosophy of the DARPA Internet Protocols. pages 106–114, Stanford, CA, Aug. 1988. SIGCOMM.

[12] S. Floyd and K. Fall. Promoting the Use of End-to-End Congestion Control in the Internet. *IEEE/ACM Transactions on Networking*, 7(4):458–472, 1999.

[13] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, 1993.

[14] V. Jacobson. Congestion Avoidance and Control. *ACM Computer Communication Review; Proceedings of the Sigcomm '88 Symposium in Stanford, CA, August, 1988*, 18, 4:314–329, 1988.

[15] V. Jacobson, B. Braden, and D. Borman. RFC 1323: TCP Extensions for High Performance, May 1992.

[16] A. Kuzmanovic and E. Knightly. Low-Rate TCP-Targeted Denial of Service Attacks. volume 33, pages 75–86. SIGCOMM, October 2003.

[17] N. Leibowitz, M. Ripeaunu, and A. Wierzbicki. Deconstructing the kazaa network. In *3rd IEEE Workshop on Internet Application*. IEEE, 2003.

[18] J. Liang, R. Kumar, and K. W. Ross. The KaZaA Overlay: A Measurement Study. http://cis.poly.edu/~ross/papers/KazaaOverlay.pdf.

[19] M. Lichtenberg and J. Curless. DECnet Transport Architecture. *Digital Technical Journal*, 4(1), 1992.

[20] http://www.limewire.com/english/content/netsize.shtml.

[21] M. Mathis, J. Mahdavi, S. Floyd, and A. Ramanov. RFC2018: TCP Selective Acknowledgment Options, October 1996.

[22] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the Slammer Worm. See http://www.caida.org/outreach/papers/2003/sapphire2/.

[23] Nanog email: Dos? http://www.merit.edu/mail.archives/nanog/2003-01/msg00594.html.

[24] V. Paxson. An Analysis of Using Reflectors for Distributed Denial-of-service attacks. *ACM Computer Communications Review (CCR)*, 31(3), July 2001.

[25] Planet lab. http://www.planet-lab.org.

[26] K. Ramakrishnan, S. Floyd, and D. Black. RFC3168: The Addition of Explicit Congestion Notification (ECN) to IP.

[27] D. Reed. "Small TCP Packets == very large overhead == DoS?", July 2001. See http://www.securityfocus.com/archive/1/195457.

[28] S. Saroiu, K. Gummadi, and S. Gribble. Measuring and Analyzing the Characteristics of Naptster and Gnutella Hosts. volume 9, pages 170–184. Multimedia Systems, Springer-Verlag, 2003.

[29] S. Savage, N. Cardwell, D. Wetherall, and T. Anderson. TCP Congestion Control with a Misbehaving Receiver. *Computer Communication Review*, 29(5), 1999.

[30] S. Staniford, V. Paxson, and N. Weaver. How to 0wn the Internet in Your Spare Time. USENIX Security Symposium, 2002.

[31] W. R. Stevens. RFC2001: TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms, January 1997.

[32] X. Wang and M. K. Reiter. Mitigating bandwidth-exhaustion attacks using congestion puzzles. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 257–267, New York, NY, USA, 2004. ACM Press.

[33] P. Watson. Slipping In The Window: TCP Reset Attacks. See http://www.osvdb.org/reference/SlippingInTheWindow_v1.0.doc.

A Implementing Opt-Ack

In this section, we describe an actual implementation of opt-ack against TCP. There are three reasons we chose to implement the attack in addition to simulating it. First, it was clear that the opt-ack attack worked in theory, but we wanted to demonstrate that it was feasible in practice. Second, the implementation would allow us to test against deployed networks and gauge the effectiveness of the attack against real-world systems. Third, and most importantly, we hoped that in implementing the real attack, we would gather sufficient insight to design a viable solution. In the rest of the section, we describe our experience with implementing opt-ack, and highlight specific challenges present in real-world systems that we had to account for.

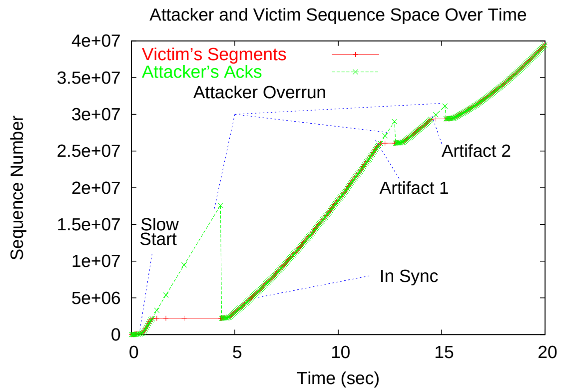

Figure 7: Attacker and Victim Sequence Space, Measured at Victim

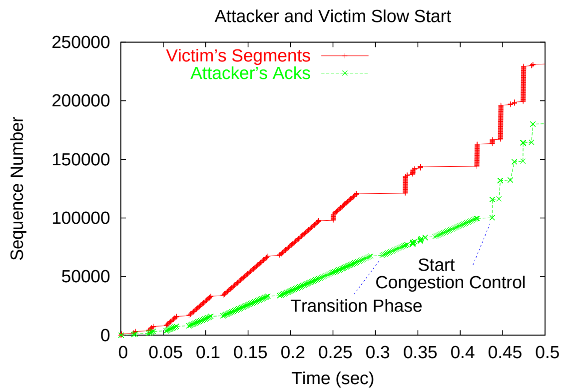

Figure 8: Detail: Attacker and Victim Slow Start

A.1 Recovery from Overruns

Compliant TCP streams are supposed to generate an empty segment upon receipt of an out of window ACK (Section 2.2). The attacker could use this empty segment to detect overruns, but the durring the attack incoming

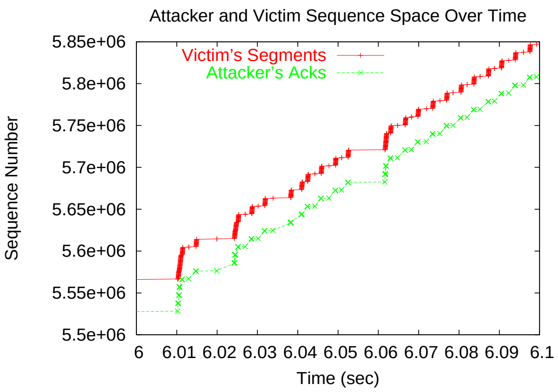

Figure 9: Detail: Attacker and Victim Synchronized

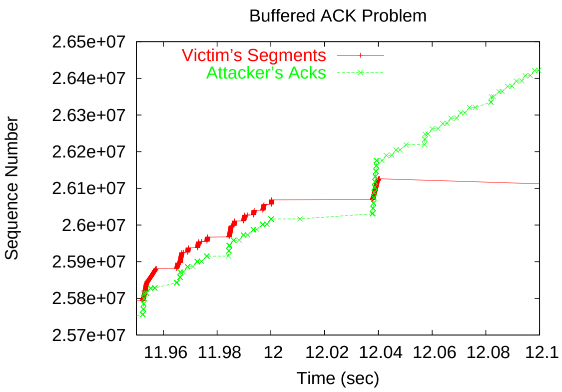

Figure 10: Artifact 1: Buffered ACKs

link is typically saturated, so the empty segment will be dropped. Additionally, Linux ignores an out of window ACK, times out on previous unACK'ed packets, and retransmits them. Other OSes, specifically MacOS X 10.2 and Windows 2000, correctly generate the empty packet. However, while a stream is making progress, the sequence numbers of packets received increase monotonically (barring packet reordering). Upon a retransmission, or when an empty packet is received, the sequence number is less than or equal to the previous packet, breaking monotonicity. So, by monitoring the sequence numbers of packets actually received, the attacker can detect overruns when the sequence numbers no longer increase. When an overrun is detected, the attacker can resume slow start on the last received packet. This is an expensive process, as it potentially requires waiting on the order of at least one second [7] for the server to timeout.

Figure 7 shows the life cycle of an attack against a GNU/Linux 2.4.20 victim, across a wide area network, as measured at the victim. The "attacker" data points show the ACKs at the time the victim received them, and the "victim" data points show the segments being sent by the victim. Note that for the majority of the time the two

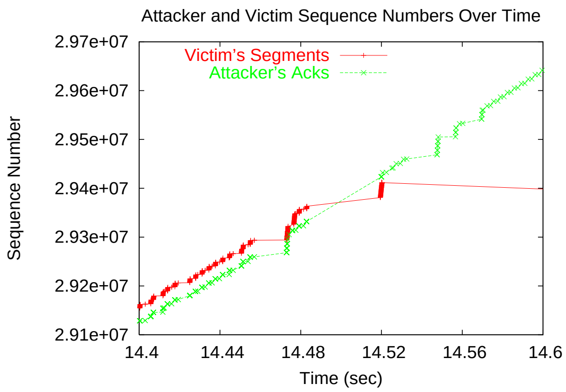

Figure 11: Artifact 2: Victim Delay and Buffered ACKs

lines are indistinguishable, i.e. the attacker is synchronized with the victim (Figure 9). However, on three occasions the attacker overruns the victim's sequence number, and is forced to recover, as described above. The attacker blindly continues sending ACKs that are ignored, as the victim stops making progress in sending the stream (as demonstrated by the flat line). In the first overrun, the victim actually retransmits three times before the attacker recovered, because the retransmitted packets were also lost. However, in the next two overruns, the attacker recovered faster, each on the order of one second.

Recovery code must track the victim's slowstart threshold ($ssthresh$) in addition to the estimated congestion window ($ecwnd$). The variable $ssthresh$ is initialized to the maximum window size, is set to half $ecwnd$ with every recovery, and grows with the congestion window, as prescribed by [31].

A.2 Victim's Processing Time

One of the most difficult challenges in keeping the attacker synchronized is estimating the time taken for the victim to send the packets, which we call the processing time. Obviously, an attacker should not ACK segments faster than a victim is capable of generating them. Through experimentation, we find that an upper bound on the processing time of a victim if 50ms (Section 5). However, this is a lose bound and in this section, we present techniques for more exactly determining it.

If the attacker knows the victim's processor speed, server load, operating system, and local bandwidth, it may be able to estimate the processing delay time. However, this information is difficult to determine, and underestimating the delay time leads to the attacker getting ahead of the server as well as significant performance degradation. To address this challenge, we introduce the $TargetBandwidth$ variable. With this variable, we can derive the processing delay:

$$processing\ delay \;=\; \frac{\lfloor cwnd/mss \rfloor \times (54 + mss)}{TargetBandwidth}$$

The $TargetBandwidth$ variable represents the rate of traffic the attacker is trying to induce the server to generate (in bytes/second). While the value of $TargetBandwidth$ can be determined adaptively based on how often the attacker is forced to recover, for the purposes of the implementation code, we specify it as a runtime parameter.

The processing time of an idle server is significantly shorter than that of a busy server. This implies that an attacker needs to estimate a server's load before attacking it. However, we noted that as the attacker's flow rate increases, the other connections are forced to back off, which in turn decreases the processing time of the server. Thus, we introduce the concept of adaptive delay. By overestimating the initial processing time and the delay between ACKs, i.e. sending ACKs slowly, and then progressively ramping up the ACK speed to the desired rate, third party streams are "pushed" out of the way with minimal overruns. How to do this effectively in an aggressive manner, without causing the attacker to overrun and restart, is an open question. However, in the implementation, we start arbitrarily at 10 times the estimated processing time, and then decrease down to the target processing time in steps of 500 μs per window.

Another variable affecting the processing time is the coarse grained time slice in the victim's scheduler. Periodically, the victim process is suspended for a number of time slices, which can cause a delay in sending if the kernel buffer is drained before the process can be rescheduled. An example of this is the second artifact (Figure 7, blown up as Figure 11), where the server actually pauses for 36 ms. Note, it is less obvious from Figure 11, but the server starts sending less than one millisecond before the buffered ACKs arrive. We do not have a technique to predict these delays, and rely on the recovery/restart mechanism.

A.3 Multiple ACKs Per Window and the Transition Phase

We noted that during congestion avoidance, the server rarely sent a full 64KB window, even when the congestion window would otherwise have allowed for it. The effect was that the number of segments in flight varied, and it became difficult for the attacker to ACK the correct number of segments. We speculate this is due to operating system buffering inefficiencies, and perhaps coarse grained time slices. Whatever the reason, we changed the attack algorithm to ACK half of the window at a time with the appropriate delay instead of the full window all at once. By ACKing half as much, twice as often, we were able to keep the amount of flooding high, reducing the chance the attacker gets ahead of the victim's sequence number. The downside is that by sending twice as many ACKs, we get only half of the performance listed in Section 3.

An additional benefit of sending two ACKs per window is resistance to lost ACKs. The basic algorithm assumes that each ACK successfully reaches the victim, which is obviously not true in general. To maximize this benefit in implementation, we send two ACKs slightly offset from each other twice per window for a total of four ACKs per window. The benefit here is two fold. First, the attacker can now lose three sequential ACKs in a row without overrunning the server. Second, with more ACKs the congestion window grows faster after recovery from overrun. The effect of sending four ACKs per window is we reduce the expected amplification by a factor of four.

It was difficult to track the exact state of the victim's congestion window and $ssthresh$, especially after recovering. It was common for the attacker to stay correctly synchronized with the victim through slow start and then get out of sync immediately when moving to the congestion avoidance algorithm. While we speculate there are many factors that cause this behavior, i.e. unpredictable server load, and the timing involved in the congestion avoidance phase may need to be more accurate than the slow start phase, it simply became easier to work around it. Thus, we introduce a "transition" phase for the attacker between slow start and congestion avoidance (see Figure 8). In this transition phase, we ACK every expected packet in turn for the full window. The effect of the transition phase is that it allows for a larger margin of error in estimating the victim's $ssthresh$ variable. In practice, we ACK two full windows in the transition phase before transitioning to the full congestion avoidance portion of the attack.

A.4 The Attacker's Local Bandwidth

Algorithm 1 does not take into account the attacker's local bandwidth. Given a local bandwidth of β in bytes per second, ACKs can be sent at at most $\alpha = \beta/54$ bytes/second. At speeds faster than α, and the ACKs get buffered or even dropped, which interferes with the timing of the attack. When ACKs are buffered (as shown in the first artifact of Figure 7, and Figure 10))they arrive at the victim all at once. The victim is not able to send fast enough to keep up with the sudden flood of ACKs and this creates an overrun. To fix this, we limit the rate of outgoing ACKs from the attacker as a function of the available local bandwidth, which is specified at runtime. The main effect of rate limiting the ACKs is to maintain even spacing when they arrive at the victim, despite network jitter and buffering.