# TCP Congestion Control with a Misbehaving Receiver

Stefan Savage, Neal Cardwell,
David Wetherall and Tom Anderson

Department of Computer Science and Engineering
University of Washington, Seattle

## The problem

- Bandwidth sharing on the Internet
  - Hosts **voluntarily** limit own data rate
  - Mechanism implemented in TCP
  - “Fair” rate determined by testing the network
  - Relies on **cooperation** between endpoints
- Why doesn’t everyone cheat?

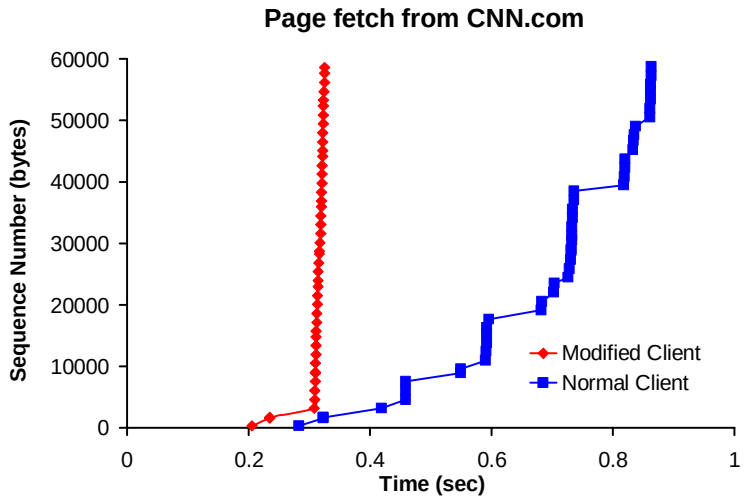
## One explanation

- Cheating requires motive **and** opportunity
- Senders (e.g., Web servers)
  - Have opportunity (could send too fast)
  - Limited motive (economic incentive to share)
- Receivers (e.g., Web clients)
  - Have competitive motive (faster Web surfing)
  - No opportunity (only receive data)… right?

## What if receivers misbehave?

- A client can **implicitly** control the data rate of a remote server
  - This is not an implementation error
  - It is a weakness in the TCP specification
  - TCP's design does not consider that senders and receivers might have disjoint interests
- The vulnerability is significant…

# Why the Web is faster in Seattle

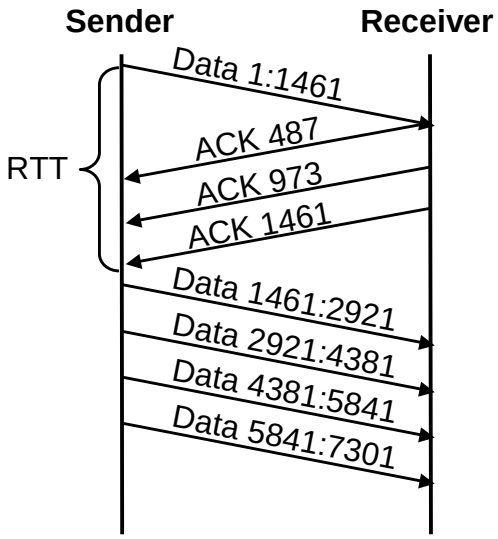



# Vulnerability 1: Bytes vs. Segments

- TCP: reliable byte stream w/ cum. ACKs
- Cwnd limits unacknowledged data
- TCP begins a session in *slow start*:

  *During slow start, TCP increments cwnd by at most SMSS bytes for each ACK received that acknowledges new data.*

# (1) ACK Division

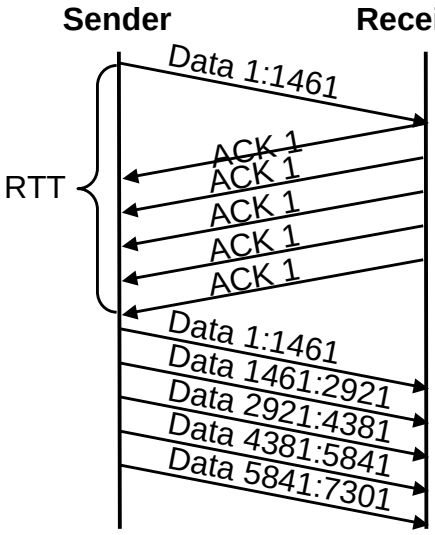



- Send M ACKs for one pkt
- Exponential growth factor proportional to M!
- Preserves end-to-end semantics

# Vulnerability 2: Fast Retransmit and Recovery

- Receive out-of-order segment => send dupack
- Sender receives 3 dupacks => fast retransmits, enters fast recovery
  - Cwnd = cwnd/2 + 3*SMSS
  - On a dupack, cwnd += SMSS

# (2) DupACK Spoofing

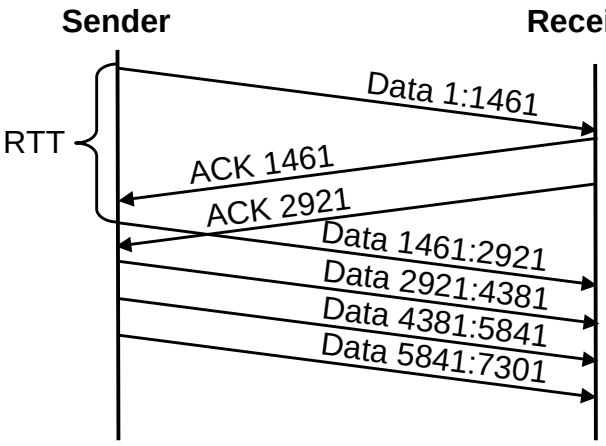



- Send extra duplicate ACKs
- Sender sends one pkt for each duplicate ACK
- Preserves end-to-end semantics

# Vulnerability 3

- When sender receives a new ACK, it increases cwnd
- *But how do you know the receiver got the data?*

# (3) Optimistic ACKing




- Send ACKs early
- Sender sends pkts in proportion to ACK rate
- Violates end-to-end semantics
- Lose reliability

# Implementation experience

- "TCP Daytona"
  - Easy to implement (<75 lines in Linux)
  - Works against all popular sender TCP stacks
    - Solaris, NT, Linux, FreeBSD, Tru64, IRIX, HPUX, AIX
      - Linux 2.2 immune to ACK division
      - NT 4 immune to DupACK spoofing
- Fetches most web pages in 2 RTT
  - We have the world's fastest Web browser!

## Simple Countermeasures

- Combating ACK Division:
  - Only increase cwnd when receiver ACKs >= 1 segment - Linux 2.2
  - Byte counting [Allman98, Allman99]
- Combating DupACK Spoofing:
  - Count outstanding segments
  - Ignore extra DupACKs
- Optimistic ACKing:
  - Randomize segment boundaries
  - Ignore ACKs unless they match a real boundary

## “Semantics”

- Meaning of message
  - Literal
  - Implied by assumptions about other party
- How message is acted upon
- Two levels:
  - TCP <-> TCP
  - TCP->application

## Principle 1

- *Every message should say what it means; the interpretation of the message should depend only on its contents*
- Ex: dupacks violate this
  - Meaning: all bytes up to X received
  - Interpreted as: another segment left net

## Principle 2

- *The conditions for a message to be acted upon should be clearly set out so that someone reviewing a design may see whether they are acceptable or not*
- Ex: when to increase cwnd in response to an ACK

## Principle 3

- *If the identity of a principal is essential to the meaning of a message, it is prudent to mention the principal's name explicitly in the message.*
- Ex: ACKs do not prove the identity of the data received

## A General Solution

- Problem: receiver lies about how many segments received
- Solution: make receiver *prove* it received each segment
- "Cumulative Nonce"
  - Each segment carries a random nonce
  - Receiver must echo sum of nonces
    - Only need *1-bit* nonce for probabilistic guarantee!
  - Receiver can only hurt itself by lying

## Deploying Cumulative Nonces

- Selfish receivers won't implement options that limit their performance
- How give receivers incentive to implement a TCP nonce *option*?
  - Combine ECN, nonce
  - Limit service

## Lessons

- "Security" only provides low-level primitives
  - Authentication, privacy, data integrity, access control, …
- Q: Is other endpoint sending semantically correct data?
  - Rational incentives
  - Malice
  - Bugs
- Q: Can you verify the semantics of the message?

## Applying the Lessons

- Systems using a similar approach:
  - Checking system call arguments
  - Proof-carrying code
  - DB consistency checks
- Other transport protocol feedback mechanisms that could use this approach:
  - Congestion Manager [BRS99]
  - RTCP
  - RAP [RHE99]

## Summary

- The Internet depends on cooperation
- But receivers have the motive and means to easily steal arbitrary capacity
  - Three attacks
  - Three deployable countermeasures
- General lesson for protocols, distributed sys.:
  - Even if “secure,” verify semantics of information
- General mechanism for acknowledgments:
  - Cumulative nonces

# Practical Network Support for IP Traceback

Stefan Savage
University of Washington/
University of California, San Diego

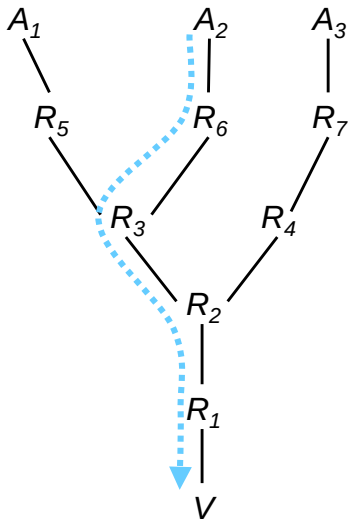
David Wetherall, Anna Karlin and Tom Anderson
University of Washington, Seattle

## Motivation

- Denial-of-service via packet flooding
  - Attackers overwhelm network and CPU resources with spurious requests
- Internet architecture permits anonymity
  - Use of IP source address is **voluntary**
  - Attackers use “false” source address
  - Network is stateless (no audit trails)
- Difficult to respond to attack without knowing what network path(s) it traverses

# Traceback problem

- Basic elements
  - Set of attackers $A_i$
  - Set of routers $R_i$
  - Victim $V$
- Attack path for $A_i$
  - Ordered list of routers between $A_i$ and V
  - e.g. {$R_6$, $R_3$, $R_2$, $R_1$}
- Goal
  - Determine attack path for each attacker




# State of the practice

- Router traffic stats + manual guesswork
- Input debugging
  - Install filter to detect attack traffic, determine upstream router, repeat
- Serious practical limitations
  - High management overhead, per-ISP only
  - Attack must be in progress

## Design constraints (self-imposed)

- **Assumptions about environment**
  - Attacker can generate any packet
  - Multiple attackers may conspire (DDoS)
  - Attackers send many packets
  - Routers resource limited (no *per-flow* state)
  - Routers not malicious
- **Goals**
  - Real-time and *post-mortem* trace capability
  - Incrementally deployable
  - Low management and network overhead

## Key idea

- **Routers store path state in packets**
- Naïve approach
  - Routers append their address to each packet
  - Can be expensive to implement
  - No assurance of “enough” space in packet
- Spread state across packets?

## Edge sampling: basic approach

- Augment packets with additional fields
  - **Edge**: two adjacent router addresses (start & end)
  - **Distance**: # edges traversed since marked
- Probabilistically mark packets in routers
  - Marked packets contain “sample” of path
- Victim, or victim’s ISP, reconstructs path after receiving enough packets

## Marking procedure at router

- Marking procedure for packets forwarded by *R*:

```
with probability p,
   write R into start field
   write 0 into distance field
else
   if distance == 0 then
      write R into end field
   increment distance field
```

## Path reconstruction at victim

- Extract identifiers from attack packets
- Build graph rooted at victim
  - Each (*start*,*end*,*distance*) tuple is an edge
  - Eliminate edges with inconsistent distance
  - Traverse edges from root to find attack paths
- # packets needed to reconstruct path

$$E(X) < \frac{\ln(d)}{p(1-p)^{d-1}}$$

*p*: marking probability
*d*: length of path

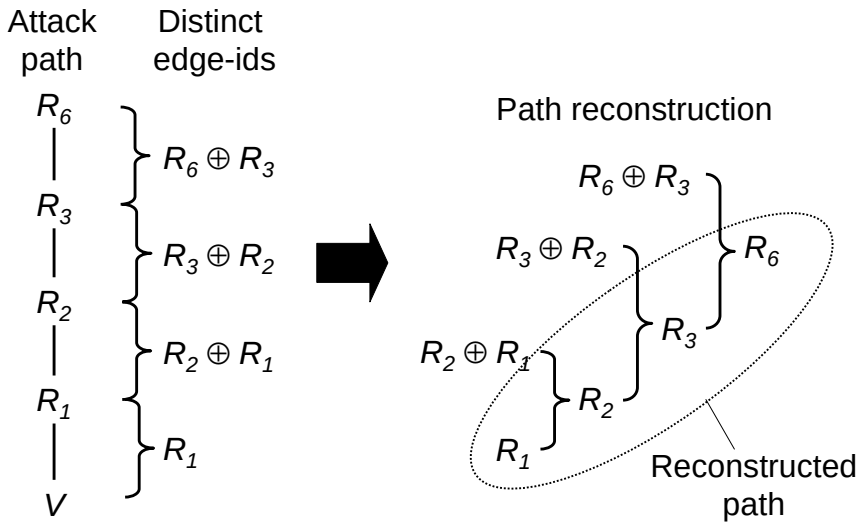
## Robustness

- Random marking decisions
  - Can't be anticipated by attacker
  - Can't be controlled by attacker
- Attackers can "lie" and invent edges
  - But not for *distances* less than their own
- Offline validation of "valid suffix"

# Practical problems

- Deployment issues
  - How to fit edge/distance data into IP header
  - Current scheme requires too much space
    - 32 bits for each router address, 8 for distance
    - Need lower per-packet overhead
- One hybrid approach
  - Trade space for reconstruction/robustness
  - Overload existing packet header field

# Compressing edges to edge-ids using XOR

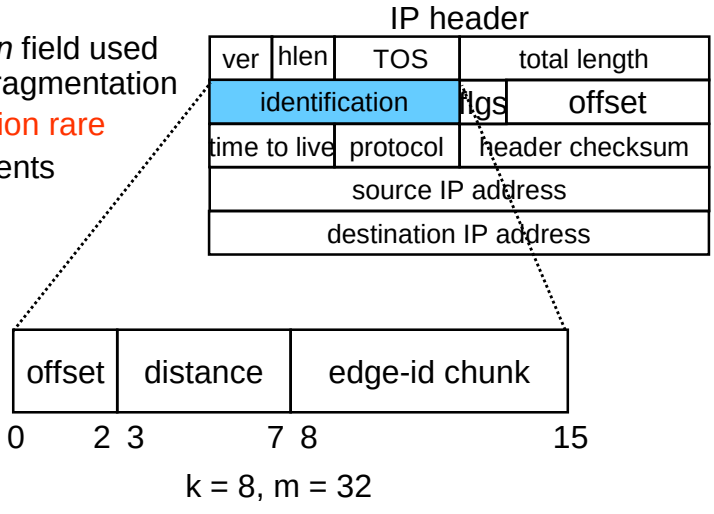


# Edge-id chunks

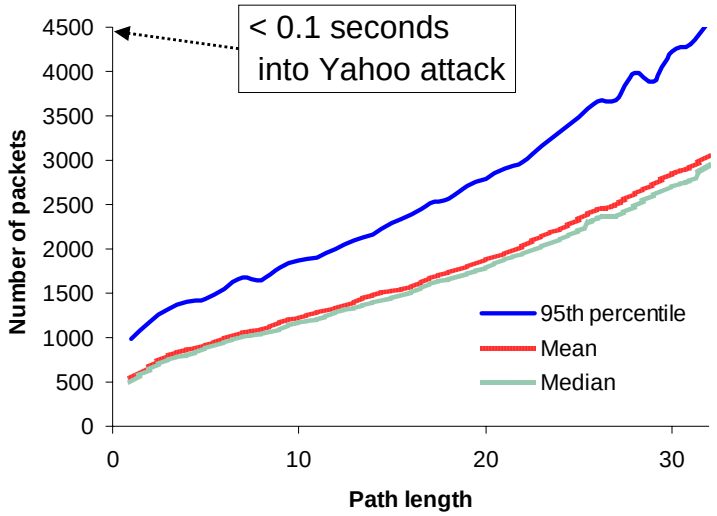
- Divide each edge-id into *k* chunks
- When marking packet
  - Pick edge-id chunk at random
  - Write chunk and its offset into packet
- Increases reconstruction time by *k* ln(*k*)
- Chunks may not be unique
  - Augment edge-id with hash of m bits
  - Validate chunk combinations at victim

# One opportunistic encoding

- *Identification* field used for packet fragmentation
- Fragmentation rare
- Mark fragments separately



# Experimental convergence time




# Weaknesses

- Path validation/authentication
- Robustness in highly distributed attacks
  - *Both addressed nicely in [Song&Perrig00]*
- Compatibility issues (IPsec AH, IPv6)
- Origin laundering (reflectors, tunnels, etc)

# Summary

- An efficient algorithm for tracing anonymous attacks: *edge sampling*
- Hybrid algorithm with reduced per-packet space overhead (at a cost)
- Potential encoding into current IP packet header