Efficient and Robust TCP Stream Normalization

Mythili Vutukuru and Hari Balakrishnan
MIT CSAIL
{mythili,hari}@csail.mit.edu

Vern Paxson
ICSI & UC Berkeley
vern@icsi.berkeley.edu

Abstract

Network intrusion detection and prevention systems are vulnerable to evasion *by attackers who craft ambiguous traffic to breach the defense of such systems. A* normalizer *is an inline network element that thwarts evasion attempts by removing ambiguities in network traffic. A particularly challenging step in normalization is the sound detection of* inconsistent TCP retransmissions*, wherein an attacker sends TCP segments with different payloads for the same sequence number space to present a network monitor with ambiguous analysis. Normalizers that buffer all unacknowledged data to verify the consistency of subsequent retransmissions consume inordinate amounts of memory on high-speed links. On the other hand, normalizers that buffer only the hashes of unacknowledged segments cannot verify the consistency of 20–30% of retransmissions that, according to our traces, do not align with the original transmissions. This paper presents the design of RoboNorm, a normalizer that buffers only the hashes of unacknowledged segments, and yet can detect* all *inconsistent retransmissions in any TCP byte stream. RoboNorm consumes 1–2 orders of magnitude less memory than normalizers that buffers all unacknowledged data, and is amenable to a high-speed implementation. RoboNorm is also robust to attacks that attempt to compromise its operation or exhaust its resources.*

1. Introduction

Network intrusion detection and prevention systems (IDS/IPS) are now widely used to improve the security of networks run by providers, enterprises, and even home users. Such monitors usually operate on the path between the protected network and the rest of the Internet, observing all traffic coming in and out of the network and flagging (IDS) or blocking (IPS) activity deemed likely malicious. While historically some of these systems operated in a stateless, per-packet fashion [1], modern systems employ detailed protocol parsing in order to analyze the traffic at higher semantic levels [2] and require in-order reconstruction of TCP byte streams as the receivers would see them. However, these reconstructed byte streams have inherent *ambiguities* largely because the monitor does not know what traffic the receiver actually receives and accepts. Starting with the work of Ptacek and Newsham [3], the possibility of a wily adversary exploiting these ambiguities to mount an *evasion attack* and confound a monitor has been recognized in the research literature [2–6]. Moreover, tools that facilitate the automatic generation of evasive traffic are readily available for use by attackers today [7–9], making evasion attacks a real threat to intrusion detection systems.

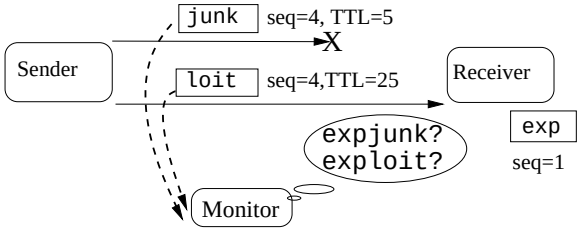

Figure 1. Evading a network monitor using inconsistent TCP retransmissions.

One important class of evasion attacks is attacks that employ *inconsistent TCP retransmissions* (i.e., TCP segments that contain different data for the same sequence number space) to confuse a network monitor's parsing. In such attacks, attackers send inconsistent TCP segments that all make it past the monitor, but which don't all reach the receiver, say, by having insufficient TTL hop-counts in their IP headers. Absent information about which segments the receiver will eventually receive and accept, the monitor cannot accurately infer if an attack is in progress. Figure 1 shows an example where the attacker sends two different TCP segments starting at sequence number 4, one of which completes a malicious string "`exploit`" at the receiver. The monitor, however, cannot unambiguously reconstruct the string that the receiver sees. The monitor cannot simply analyze all possible interpretations of such ambiguous traffic, since the combinations grow exponentially large [5]. Note that in this example, the receiver never sees the seg-

ment carrying "`junk`" because it lacks sufficient hop-count (TTL) to make it all the way to its purported address. Alternatively, the attacker can also rely on knowledge of whether the receiver's particular operating system accepts only the first instance of a segment, or overwrites the contents of the segment with any later transmission; network stacks differ in their treatment of this corner case [3].

Previous work by Malan et al. [4] and Handley et al. [5] developed the notion of traffic *normalization*, by which an in-line network element removes such ambiguities from a traffic stream prior to presenting it to a monitor for security analysis, thus thwarting evasion attacks. Detecting inconsistent retransmissions is one of the hardest steps in normalization because it cannot be performed in a simple stateless fashion, unlike most other steps [5]. In recent work, Varghese and colleagues pursued a more restricted problem of detecting inconsistent retransmissions in the context of byte-level signature detection alone [6]. Our goal in this work is to design an in-line network element that detects and blocks inconsistent retransmissions in *any* TCP byte stream, in a manner that is both memory-efficient and resistant to attacker-induced stress.

The brute-force approach to detecting inconsistent retransmissions, used by some existing intrusion detection systems (e.g., Bro [2]) and normalizers [4, 5], is to buffer all the unacknowledged bytes for each active connection and to compare any retransmission against the stored bytes for consistency. However, the brute-force ("full-content") normalizer needs to be provisioned with a significant amount of memory—almost a bandwidth-delay product's worth—rendering such normalizers impractical and expensive on high-speed links.

An alternative approach, potentially requiring significantly less memory, is to instead store hashes over the content of unacknowledged segments rather than a full copy of the contents, and compare the hashes of retransmissions to stored hashes. Such an approach cannot verify the consistency of retransmissions that are packetized differently from the original segments. In practice, however, retransmissions not infrequently occur misaligned with the original segment boundaries, overlapping in unexpected ways—from our analysis of five real-world TCP packet traces, we find that 20–30% of all retransmissions were not aligned along original segment boundaries (§2). While we know from discussions with implementors that some commercial systems also use the approach of storing hashes, the literature does not provide any analysis of how well such an approach actually works in practice.

Thus, existing normalizer designs fail to meet the full set of goals important for a practical normalizer:

1. **Memory-efficiency.** The normalizer must use memory frugally to store most of the data on fast (but scarce and expensive) on-chip memory in order to process packets at line speed. Memory efficiency becomes increasingly significant as link speeds improve and connections have greater amounts of data in-flight, and has major implications for cost and power consumption.
2. **Correctness.** The normalizer must *always* identify and block inconsistent TCP byte streams, irrespective of how the bytes are packetized into segments by the senders. Existing hash-based approaches *do not* provide actual protection from adversaries, since in practice such normalizers will fairly frequently encounter retransmissions for which they cannot verify payload consistency.
3. **Adversarial-resistance.** The normalizer must be robust to an adversary mounting attacks to degrade the normalizer's operation. Such attacks might seek to exhaust the memory or processing resources of the normalizer, preventing it either from functioning correctly or causing it to deny service to other, benign TCP connections. For example, an adversary can easily exhaust the memory of the full-content normalizer by having large windows of unacknowledged data over multiple connections.

In this paper we describe RoboNorm, a robust normalizer design that aims to meet all of these design goals. RoboNorm maintains a content hash for every unacknowledged segment of every connection, and (with careful design, per §3) verifies the consistency of all retransmissions including those that are misaligned with original segment boundaries. RoboNorm requires around 2.5 MB of memory on a typical Gbps access link, 1–2 orders of magnitude less than that required by the full-content approach (§4), making it amenable to an inexpensive yet high-speed implementation (§5). In addition, RoboNorm thwarts a variety of state-exhaustion attacks by using robust policies to evict connections when under stress; the policies require the adversary to summon major resources to impair the system's operation, and rarely inflict collateral damage on benign connections (§6). While sound and robust operation of RoboNorm necessitates occasional alteration of traffic and end-to-end semantics (e.g., ACK rewriting), we demonstrate that such alteration occurs exceedingly rarely in practice.

2. Assumptions and Challenges

This section investigates the challenges to designing an efficient and robust normalizer using a number of real-world packet traces. We begin with the assumptions and terminology used in this paper.

Assumptions. The normalizer is an in-line network element deployed at the access link of a network one wishes to protect, most likely in conjunction with an IDS/IPS. We assume that the normalizer always sees packets in both direc-

#	Trace Characteristics	$\mathbf{Univ}_1$	$\mathbf{Univ}_2$	$\mathbf{Lab}_1$	$\mathbf{Lab}_2$	Super
1	Total # half-connections with data	648K	15.3M	1.21M	601K	32.5K
2	% of above with retransmits	5.6	5.16	4.07	4.83	2.39
3	Total # TCP packets	31M	435M	127M	40.5M	30.3M
4	% of above that are retransmits	0.58	0.32	0.06	0.19	0.04
5	% retransmits not aligned with originals	29.0	25.2	31.5	20.6	18.1

Table 1. Basic statistics of the traces used in the paper.

tions (i.e., data and acknowledgments) of any TCP connection it processes.[1] We also assume that the normalizer can actively alter the traffic passing through it, say by holding onto some packets without forwarding or rewriting some fields of the packet headers. It can terminate connections that it suspects of conducting malicious activity. When under stress, the normalizer fails on the safe side by terminating suspicious connections to relieve stress instead of letting traffic through without inspection.

Terminology. TCP is a byte stream protocol for which a TCP *segment* is the unit of transmission. In our discussions, we represent segments by the sequence number ranges of the bytes they contain.[2] We term a TCP segment as *new* if none of its sequence numbers have previously appeared at the normalizer. Otherwise, we term the segment *retransmitted*. New segments that we later compare retransmissions against are at that point termed *original* segments. Note that a retransmitted segment can contain both sequence numbers previously seen and new sequence numbers. We also define a *hole* as a range of sequence numbers for which the normalizer has not seen the corresponding bytes.

Traces. We use five packet traces to understand the challenges in designing a normalizer, and to validate our design in the rest of the paper. These traces, referred to as Univ_1, Univ_2, Lab_1, Lab_2, and Super, were collected at the Gbps access links of four large sites: two large university environments, with about 45,000 hosts (Univ_1) and 30,000 hosts (Univ_2), respectively; a research laboratory with about 6,000 hosts (Lab_1 and Lab_2); and a supercomputer center with 3,000 hosts (Super). All traces were captured during afternoon working hours. Although we cannot claim that these traces are broadly representative, they do span a spectrum from many hosts making small connections (the primary flavor of university sites, Univ_1 and Univ_2) to a few hosts making large, fast connections (the supercomputing site, Super). Appendix A details the trace collection method.

Table 1 presents some aggregate statistics of the traces. Row 1 gives the number of TCP data-transfer paths ("half-connections") we analyzed. Each TCP connection potentially gives rise to two of these, one in each direction that actually transfers data; we analyze each direction independently. We see in row 2 that a significant fraction (2.4–5.6%) of half-connections undergo retransmission at some point, even though the next two rows show that a far lower fraction (0.5%) of the total *packets* are themselves retransmitted. The last row of the table shows that *20–30% of retransmitted segments are not aligned with the corresponding original segments.*

Challenges. The design of a TCP normalizer must overcome three challenges, as mentioned in §1: memory-efficiency, correctness in the face of complex TCP retransmission behavior, and attack-resilience. The normalizer must use memory sparingly: storing all unacknowledged bytes consumes an excessive amount of memory, especially on high link speeds. To reduce memory consumption, some systems store content-hashes of TCP segments, simply comparing retransmissions against the stored hashes. Unfortunately, as we noted above, a significant fraction of normal TCP connections today do not retransmit along the same segment boundaries as the original transmissions. In response, such designs must either terminate the connection, with significant collateral damage, or let misaligned data through, which allows an attacker who crafts such traffic to evade detection.

Thus, the second challenge is to guarantee correctness in the face of TCP retransmission vagaries. TCP's complex retransmission behavior arises because the TCP specification allows latitude in terms of how retransmitted segments correspond to original segments—a sender can repacketize the data during retransmission, with the result that the retransmitted segments may not match the original segments in size or sequence range. Moreover, while the specification states that a newly received segment that overlaps with an existing segment should be trimmed to only the new data (p. 53 of [10]), in reality different TCP implementations be-

[1] If the normalizer can only see one side of a connection, then it cannot safely reclaim state associated with acknowledged data, nor can it correctly execute the mechanisms we develop to handle retransmissions that are not aligned with original segment boundaries (§3).

[2] When not ambiguous, we will sometimes refer to "bytes" as a shorter term for "sequence numbers."

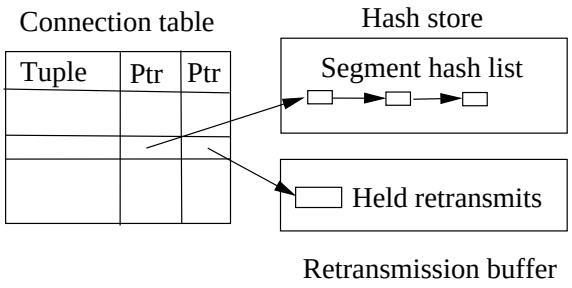

Figure 2. Architecture of RoboNorm.

have differently in this regard [3]. As a result, a network monitor might not be able to tell whether a given segment ultimately makes it all the way to the receiver—or, if it does, whether the receiver will use its contents—and thus whether the receiver will treat a subsequent retransmission as "new" or "overlapping."

The third challenge for a normalizer is to resist malicious adversaries. In general, malicious nodes can send arbitrary streams of packets to attempt to exhaust the computational capacity and memory of the normalizer, or to undermine its correct operation. Hosts inside the protected network can collude with adversaries outside the network. Attackers not on the path between communicating TCP hosts can spoof source IP addresses to disrupt TCP connections. Irrespective of the attacker's strategy, we must ensure that all inconsistent retransmissions are detected, and that no additional vulnerabilities are introduced with respect to other attacks.

3. Design of RoboNorm

This section describes the design and packet processing algorithms of RoboNorm. For ease of exposition, we assume that only data entering a protected network is being normalized; it is straight-forward to extend the scheme to normalize data in both directions.

3.1. System Overview

Figure 2 shows the various components of RoboNorm. For every TCP connection, RoboNorm maintains a *segment hash* for each unacknowledged segment, computed by hashing the segment's contents.[3] The segment hashes of each connection are stored as a linked list, called the *segment hash list*, that is sorted by the starting sequence number of the corresponding segments. Segment hashes in the list cover non-overlapping sequence number ranges. The collection of all segment hash lists in RoboNorm is referred to as its *hash store*.

[3] See §6.4 for a discussion on the choice of hash functions.

When a TCP segment (data or ACK) arrives, RoboNorm locates the connection's segment hash list by looking up the connection tuple in a hash table called the *connection table*. The connection table maps connection tuples to per-connection state that includes a pointer to the connection's segment hash list. If the arriving segment is a new data segment, RoboNorm creates a corresponding new segment hash and forwards the segment. If the segment is a retransmission, RoboNorm tries to verify the consistency of the segment by comparing its hash to existing segment hashes over the segment's sequence number range. Segments whose consistency cannot immediately be checked (e.g., segments which do not exactly overlap with existing segment hashes) are buffered without forwarding in the *retransmission buffer* of RoboNorm; handling them requires additional mechanism as described in §3.2. Handling ACKs involves clearing segment hashes over acknowledged data, and some subtleties to handle special cases (§3.3).

RoboNorm initializes state in the connection table on seeing the first data segment of the connection, not the first SYN segment, to prevent an easy state exhaustion attack caused by SYN flooding. Upon seeing a FIN or RST segment, RoboNorm marks the corresponding entry in the connection table for clearing, and completely clears the entry when all of the connection's pending data has been acknowledged. These simple state initialization and termination policies make RoboNorm vulnerable to a variety of attacks that aim to exhaust space in its connection table; we later describe the attacks and suitably augment RoboNorm's design to defend against them (§6.2).

3.2. TCP Data Segment Processing

When a TCP data segment arrives, RoboNorm retrieves the connection's segment hash list and checks if its sequence range has been seen before. If not, RoboNorm creates a new segment hash, inserts it into the segment hash list at the sorted position, and forwards the segment. Otherwise, it breaks up the segment's sequence range into portions that overlap *exactly* or *partially* with those already in the hash list, and into maximal new ranges (filling in one or more holes), with these latter treated as if they were new segments by creating new segment hashes for them. Note that we do not store segment hashes for sequence number ranges that have already been acknowledged. Figure 3(i) illustrates the process of splitting a retransmitted segment into new, exactly overlapping and partially overlapping ranges, shown as segments B, A, and C in the figure respectively.

For each range that exactly overlaps with a stored segment hash, RoboNorm computes the hash over the corresponding contents and compares it with the stored segment hash. If the hashes match, it forwards the segment. If the hashes do *not* match, it has found an inconsistent retrans-

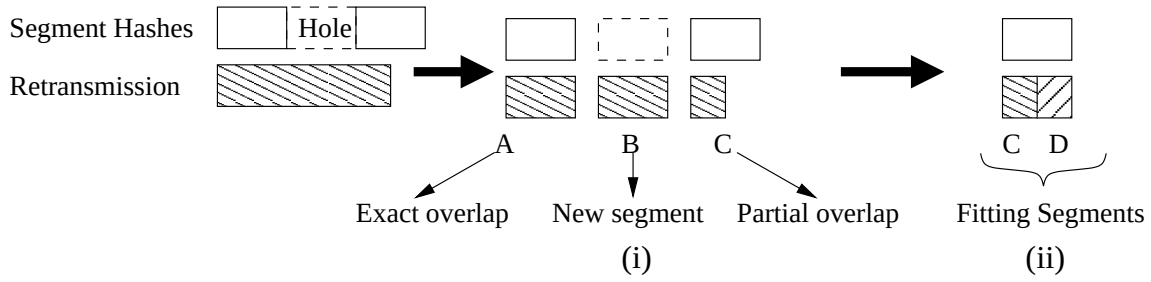

Figure 3. Retransmission terminology.

mission. A natural action for it to take at this point is to reset the connection, since even if the inconsistency is due to benign causes, the connection is in serious trouble in terms of its proper semantics. The actual action taken, however, is left to the policy of the network administrator.

Handling sequence number ranges that partially overlap with stored segment hashes is the only tricky case. Since partial overlaps can only occur at the beginning or end of an arriving segment, there can be at most two such ranges per retransmitted segment. For example, the retransmitted segment in Figure 3(i) contains one partial overlap (segment C) at its end. Table 2 shows that 12–20% of retransmitted segments (row 1) and 0.1–0.5% of half-connections (row 2) have partially overlapping ranges in our traces.

RoboNorm cannot verify the consistency of partially overlapping ranges because the original content-hashes were created over larger ranges. As a result, the system must *hold on* to the partially overlapping portions in RoboNorm's retransmission buffer—*without* forwarding them—until one or more partially overlapping segments that "fit together" to span an entire segment hash arrive. We use the term *fitting segments* to refer to partially overlapping segments that form an exactly overlapping segment when concatenated together. For example, segments C and D in Figure 3(ii) are fitting segments. Once all the fitting segments arrive, RoboNorm can then compute a hash over the concatenation of those segments and compare it with the corresponding segment hash value in the hash list for consistency, forwarding the segments upon a verified match.

One may wonder if holding on to each fitting segment without forwarding it will guarantee forward progress, i.e., will this approach always ensure that the remaining fitting segments eventually arrive, allowing the normalizer to determine whether the retransmission is consistent or not? If the partial overlap does not include the left edge of a stored segment hash (see Figure 4(i)) then eventually the *earlier* portion will have to arrive, perhaps after a TCP timeout at the sender, since the receiver will not otherwise send an ACK for it. On the other hand, if the fitting segment overlaps with the left edge of a stored segment hash, but does not extend all the way to the end of the stored segment, then our holding back the partial overlap without forwarding may cause the scheme to stall, as shown in Figure 4(ii). This is because the sender might continue retransmitting the partially overlapping segment and never decide to send any subsequent fitting segments. Coping with this possibility (which our traces indicate can indeed occur in practice) requires an additional mechanism to manipulate TCP ACKs in the opposite direction in order to *elicit* fitting segments. We describe this technique in §3.3, where we also analyze our traces to estimate how frequently this situation arises.

3.3. TCP ACK Processing

When a TCP ACK arrives, RoboNorm deletes segment hashes acknowledged by the ACK. There are two kinds of ACKs to consider: those that are aligned with an existing *segment hash boundary* (i.e., the start or end sequence number of a segment hash), and those that acknowledge data in the middle of an existing segment hash.[4] RoboNorm forwards every ACK it inspects, but, as discussed below, sometimes it must first modify the information in the ACK.

ACK on existing segment hash boundary. Upon seeing an aligned ACK, RoboNorm deletes all segment hashes in the connection's hash list that lie at or below the ACK. In addition, it also discards from the retransmission buffer any buffered segments that the ACK covers. Row 3 of Table 2 shows that 50–70% of all partially overlapping segments are acknowledged before the corresponding fitting segments appear in our traces. This situation occurs because the bytes corresponding to the remaining fitting segments were *actually already at the receiver*, and the retransmission of a *different* segment enabled the receiver to acknowledge the whole set. Thus, in this case it was unnecessary for RoboNorm to buffer the misaligned data, but it also did no harm.

[4] A third type, which acknowledges unsent data or a sequence number inside a hole, clearly represents some sort of significant failure, either in the end system or in the normalizer itself. The response in this case is a policy decision.

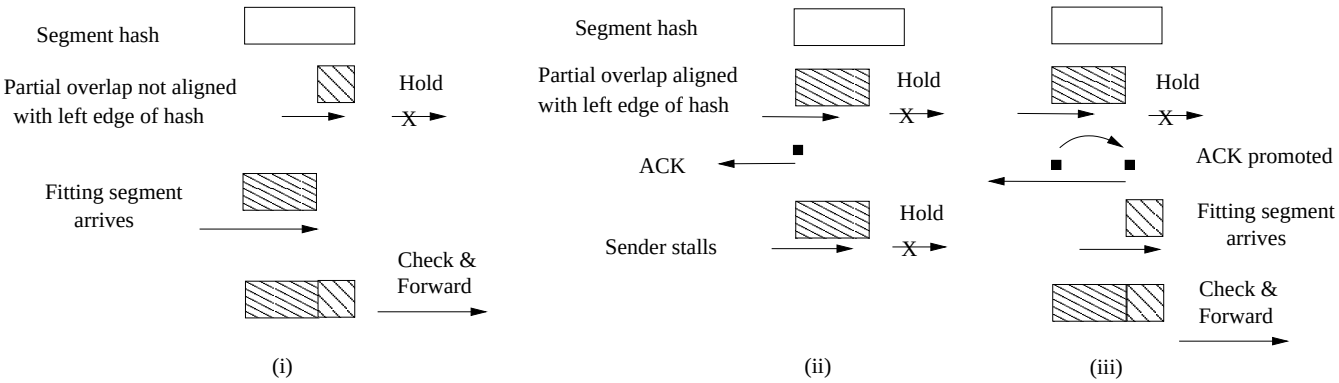

Figure 4. An illustration of (i) handling partial overlaps that are not aligned with the left edge of the segment hash, (ii) stalling of connections with partial overlaps that align with the left edge in the absence of additional mechanism, and (iii) the ACK promotion mechanism to elicit fitting segments and avoid stalling.

#	Prevalence of partially overlapping segments	Univ_1	Univ_2	Lab_1	Lab_2	Super
1	% of all retransmitted segments	17.8	18.3	21.0	12.3	12.4
2	% of all half-connections	0.4798	0.0954	0.1318	0.0891	0.0706
Partially overlapping segments cleared by ACKs						
3	% ACKed before fitting segments arrive	51.19	54.2	67.7	74.9	55.6
Frequency of ACK promotion						
4	% fitting segment pairs with ACK in between	22.1	20.1	45.7	14.2	9.6
5	% all half-conns with ACK between fitting segments	0.0976	0.0221	0.0181	0.0176	0.0092
Frequency of ACK demotion						
6	Number of ACKs not on segment hash boundaries	278	2169	1020	18	13
7	% of all half-conns with such ACKs	0.024	0.0072	0.0134	0.0019	0.003

Table 2. Trace characteristics pertaining to partially overlapping segments.

When receiving an aligned ACK, RoboNorm also checks to see if the connection has any partially overlapping segments in its retransmission buffer that *start* at the sequence number of the ACK. If such a segment exists, RoboNorm rewrites the sequence number of the ACK and shifts it upwards to the starting sequence number of the expected fitting segment (i.e., the end of the partially overlapping segment), as shown in Figure 4(iii). We refer to this mechanism as *ACK promotion*. As we can see from the figure, ACK promotion enables RoboNorm to check the consistency of the partially overlapping segment without stalling connections.

This situation might seem a bit precarious, as it requires RoboNorm to generate an ACK for data that has not in fact reached the receiver yet, playing fast-and-loose with TCP's end-to-end and fate-sharing semantics. However, given that the connection may simply stall and keep retransmitting the same segment (which will not be forwarded since each time there still isn't enough information to check for consistency), this step is required. Although RoboNorm must now "take responsibility" for the delivery of this segment to the receiver, doing so does not require RoboNorm to implement any portion of the TCP state machine or any additional timers. As long as ACKs come in from the receiver, RoboNorm can promote them, and as long as ACKs reach the TCP sender, the sender will transmit additional segments (e.g., a segment starting at the first unacknowledged byte) to make a set of fitting segments. Once the fitting segments have all arrived, RoboNorm will forward all of the held data, since it can now verify consistency with the corresponding original segment. This approach always guarantees forward progress, and requires no additional mechanisms in RoboNorm other than the ability to rewrite ACKs to promote them. Under the assumption that RoboNorm sees all packets of both directions of a connection, the packet processing algorithms ensure that *a connection never stalls indefinitely due to a pending consistency*

```
procedure HANDLEDATA(dataSgmt)
  tuple ← GETCONNECTIONTUPLE(dataSgmt)
  hashList ← FETCHHASHLIST(tuple)
  if (hashList = NULL)
    hashList ← INITHASHLIST(tuple)
  SegList ← SPLIT(dataSgmt)
     // Split at boundaries of previous segments
  for each sgmt ∈ SegList  do
   if (sgmt is new segment)
    HANDLENEWSEGMENT(sgmt)
  else if (sgmt completely overlaps some segHash)
    HANDLEEXACTOVERLAP(sgmt, segHash)
  else     // sgmt partially overlaps some segHash
    HANDLEPARTIALOVERLAP(sgmt, segHash)


procedure HANDLENEWSEGMENT(sgmt)
  hashList ← hashList ∪ SEGMENTHASH(sgmt)
  FORWARD(sgmt)

procedure HANDLEEXACTOVERLAP(sgmt, segHash)
  if (SEGMENTHASH(sgmt) = segHash)
   FORWARD(sgmt)
  else Flag inconsistent retransmission

procedure HANDLEPARTIALOVERLAP(sgmt, segHash)
  BufSegs ← {sgmt} ∪ BUFFEREDSEGMENTS(segHash)
  if (FITTINGSEGMENTS(BufSegs) = True)
   concat ← CONCATENATE(BufSegs)
   HANDLEEXACTOVERLAP(concat, segHash)
   Keep segments for which ACK promoted
   Clear rest of the fitting segments from buffer

procedure HANDLEACK(Ack)
  if (Ack not a segment hash boundary)
   DEMOTEACK()
  TRIMHASHLIST(Ack)
  if (Buffered segments starting at Ack)
    Ack ← Start of next fitting segment     // Promote
    Mark buffered segments as ACK promoted
  FORWARD(Ack)
```

Figure 5. RoboNorm's algorithms.

check of retransmitted segments.

From our traces we can estimate how often such ACK promotion is required in practice. Because the traces were collected without a normalizer in the forwarding path, it is impossible to tell for sure what would have happened had there been a normalizer that did not forward partial overlaps. Instead, we use a heuristic: we compute the number of times we observe the following sequence: an original segment $S = [s, e)$; a partial overlap $[s, e')$, where $e' < e$; an ACK for e'; followed at some later point in time by a segment starting with e'. That is, the retransmitted segment $[s, e')$ and its successor that started at e' were "split" in the trace by an ACK e'. The intuition here is that the ACK e' was in fact necessary to elicit the segment starting at e'. In such a case, holding the segment $[s, e')$ in a buffer without forwarding it could have prevented the ACK e' from arriving at all; ergo, if we don't promote ACKs, in this case the connection could stall.

Table 2 shows that about 20–50% of fitting segment pairs are split by an ACK between them (row 4), and that about 0.01% of all connections have such fitting segments (row 5), across all traces. These figures are low, but certainly not negligible: for sites that see millions of connections per day (as do all of the sites in our study), such a rate would result in 100s to 1000s of broken connections each day without the ACK promotion mechanism.

ACK not on segment hash boundary. While ACKs not on an existing segment hash boundary might strike us as highly peculiar (just what drove the receiver to select the particular sequence number to acknowledge?), our traces show that these do occur occasionally in real traffic (rows 6 and 7 of Table 2). In this case, RoboNorm first *demotes* the ACK to the segment hash boundary closest to and below its sequence number, after which it handles it like a normal ACK on a segment hash boundary. To see why we must demote such ACKs, observe that if we forwarded such an ACK to the TCP sender without demotion, its arrival may trigger a partially overlapping segment starting at the sequence number of the ACK. Moreover, the fitting segments of this triggered, partially overlapping segment would belong to the sequence space that has *already been acknowledged by the forwarded ACK* and hence will never be retransmitted. Thus, demoting ACKs avoids accumulating partially overlapping segments whose consistency can never be verified by RoboNorm.

The complete pseudocode of RoboNorm's operations is given in Figure 5.

4. Memory Savings With RoboNorm

In this section, we compute the amount of memory a typical RoboNorm deployment would consume. Under the assumption that RoboNorm stores an 8-byte hash of contents for each unacknowledged TCP segment,[5] each entry of the segment hash list—composed of the hash itself, a sequence number range, and a 3-byte pointer to the next seg-

[5] We argue in §6.4 that an 8-byte hash provides acceptable security guarantees.

#	Provisioning the connection table	$\mathbf{Univ}_1$	$\mathbf{Univ}_2$	$\mathbf{Lab}_1$	$\mathbf{Lab}_2$	Super
1	Peak concurrent connections	10,647	33,932	4,010	1,927	295
2	Avg. concurrent connections	7,616	23,686	3,098	1,556	203
Provisioning the retransmission buffer						
3	Avg. concurrent bytes per connection	639	579	665	594	566
4	Peak total concurrent bytes	13,213	116,937	87,118	12,411	2,256

Table 3. Measurements used to provision RoboNorm.

ment hash in the list—can all be made to fit in 15 bytes (Appendix B). Each connection table entry, consisting of a connection tuple and pointers into the hash store and retransmission buffer, consumes around 48 bytes (Appendix C). With these estimates, we find that RoboNorm deployed on a Gbps access link of a typical network needs to be provisioned with as little as 2.5 MB of on-chip memory, while a normalizer that buffers all unacknowledged data would need 10 times as much (§4.1). Given the high cost of fast on-chip memory, this memory gain is significant. Moreover, the actual memory that would have been consumed (had RoboNorm been deployed at the sites we collected the traces from) was found to be much smaller than the provisioned amount in most cases, and *up to two orders of magnitude smaller* than the actual memory consumed by the full-content normalizer (§4.2).

4.1. Savings in Provisioning

Hash Store. Each new segment hash in RoboNorm occupies space in the hash store and remains there until cleared by an ACK. Suppose segments arrive at a rate of λ per second, and that the average time before clearing is δ seconds across all connections. Then, by Little's Law, on average the system has to store $\lambda\delta$ segment hashes in it. In general, δ is roughly equal to the average connection round-trip time (RTT),[6] and λ is roughly equal to C/s, where C is the rate of traffic entering the system in bytes per second, and s is the average packet size in bytes. Thus, the number of segment hashes in the system at any time is roughly $\delta C/s$.

To estimate an upper bound on δ, we compute the largest segment clearing time observed during the lifetime of a connection for every connection in our traces, and compute the mean of this value across all connections in all traces. We found this value to be around 150 ms. We also found that the average non-empty segment is at least 1 KB across all traces. So, picking $\delta = 200$ ms and $s = 1$ KB gives us a bound of 25,000 hashes when provisioning for these $C = 1$ Gbit/s links. This translates to 375 KB of memory, assuming 15 bytes per segment hash. On the other hand, the full-content normalizer would require $\delta C = 25$ MB of memory to buffer all unacknowledged data.

Connection Table. We need to size the connection table according to the maximum number of concurrent (established) connections expected. Row 1 of Table 3 shows that the maximum value we find in our traces is about 34,000 connections. The next row of the table also gives the average value, which runs about 1/3 lower. Assuming each connection table slot consumes 48 bytes, and we use a hash table with 80% bucket utilization, we can accommodate 34,000 concurrent connections in about 2 MB. It is reasonable to assume that the full-content normalizer would also need comparable amounts of memory to store per-connection state.

Retransmission Buffer. Table 3 (rows 3 and 4) gives two different sets of statistics regarding the amount of buffer needed to hold partially overlapping retransmitted segments (*cf.* the "Hold" elements in Figure 4). Row 3 lists the average buffer space required for each connection that includes at least one such held retransmission. Row 4 lists the aggregate peak buffer space required for such held retransmissions across all connections. We see that a few 10 KBs suffice across all of our datasets, a number small enough that we ignore it for our subsequent comparisons.

In summary, we find that RoboNorm requires about 2.5 MB of memory on our 1 Gbit/s links, while the full-content normalizer requires about 27 MB, giving us a significant *provisioning gain* of a factor of 10 between the two designs.

4.2. Savings in Observed Memory Consumption

The *actual* memory consumed by a normalizer in real deployments will of course vary compared to the provisioned amount. We now estimate the actual memory consumption for each trace, as shown in Table 4. We consider both directions of every connection. Row 1 of the table gives the maximum number of concurrent hashes (that RoboNorm would

[6] Actually, δ is less than the average connection RTT, since what matters is the time that elapses between the monitor seeing a data packet and then seeing the corresponding ACK. In the absence of loss, this will be less than RTT; possibly a great deal, if the monitor is near the receiver.

#	Memory consumed in practice	Univ_1	Univ_2	Lab_1	Lab_2	Super
1	Peak concurrent hashes	18,417	9,000	2,124	1,469	2,118
2	Peak concurrent bytes when holding full data	16,417 KB	5,236 KB	2,709 KB	1,836 KB	3,029 KB
3	Peak total memory by RoboNorm	787 KB	1,764 KB	224 KB	115 KB	46 KB
4	Peak total memory by full-content norm.	16,928 KB	6,865 KB	2,901 KB	1,928 KB	3,043 KB
Factor of memory savings in practice						
5	Savings in total memory	21.5	3.9	13	16.8	66.3
6	Savings in unacknowledged data	59.4	38.8	85.0	83.3	95.3

Table 4. Memory savings of RoboNorm compared to the full-content normalizer.

have had to store if deployed) and row 2 gives the maximum number of concurrent bytes buffered by the system (had we deployed a normalizer that buffered all unacknowledged bytes) across all traces. We can then approximate the actual peak memory consumption of the trace as the memory required to store the peak number of concurrent connections and the peak number of concurrent bytes or hashes in the trace, as the case may be. Rows 3 and 4 of the table show the total maximum memory consumed by RoboNorm and the full-content normalizer respectively.

Row 5 of the table computes the ratio of the total memory consumed by the full-content normalizer to that consumed by RoboNorm. We find that the memory savings are considerable in practice too, generally 1–2 orders of magnitude. Note that these values include the connection table, which for our scheme heavily dominates total memory consumption (but not for the full-content normalizer). If we exclude the connection table, the savings are about two orders of magnitude (row 6). Thus any technique that compresses the per-connection hash table (e.g., connection compressors [11]) will improve the relative gain of RoboNorm over the full-content normalizer.

5. Implementation Options

Realizing a prototype of RoboNorm that can process packets at line speed on Gbps (or faster) links requires an implementation that uses memory frugally, and performs only a small amount of per-packet processing (in terms of computation and memory accesses). We now argue that the design of RoboNorm lends itself to such an implementation.

RoboNorm deployed on our Gbps links requires around 2.5 MB of memory (§4), an amount that can readily fit on-chip. The common case packet-processing in RoboNorm involves (a) looking up a hash table and (b) manipulating the hash lists, either by adding new segment hashes at the end of the list (when new data arrives) or clearing hashes from the beginning of the list (on an ACK). Both these operations can be performed efficiently in hardware: much work has been done on how to perform hardware hash table lookups efficiently [12], and we can make the common-case hash list operations inexpensive by maintaining pointers to the start and end of each hash list. Thus, per-packet processing in RoboNorm would involve only a few accesses to on-chip memory in the common case. Retransmitted segments, however, may require traversing the segment hash list to compare hashes, or more complex operations involving partially overlapping segments. But because retransmissions form around 0.5% of all packets (row 4 of Table 1), we can handle such operations on a slow path or in software without introducing perceptible delays in packet processing.

6. Attacks on RoboNorm

This section discusses attacks an adversary can launch to undermine RoboNorm's correct operation, either by exhausting its memory or by breaking the hash function used to generate segment hashes, and the defenses we propose.

Memory Exhaustion. We observe that even a carefully provisioned normalizer cannot handle workloads that consume unreasonably large amounts of memory. For example, in the worst case, workloads could consist of TCP segments with 1-byte payloads and very large clearing times, or a large number of connections with very little data outstanding per connection. Provisioning the memory of RoboNorm for such workloads is clearly impractical. This means that RoboNorm has to deal with the possibility of the system running out of space. In this section, we describe mechanisms that enable RoboNorm to gracefully handle memory exhaustion arising from either benign reasons (e.g., sudden spike in traffic volume due to a flash crowd), or state-holding attacks on the normalizer by a malicious adversary. We examine each component of RoboNorm in turn (§6.1, §6.2, §6.3).

Breaking the hash function. If an attacker can successfully create collisions under the hash function used by RoboNorm, he can evade detection by RoboNorm by generating inconsistent TCP segments with identical hashes. §6.4 describes appropriate choice of hash functions that makes

the success probability of such attacks negligible.

We argue that attacks that exhaust the computational capacity of RoboNorm are not a threat to the system. An attacker can try to exhaust the computational capacity of RoboNorm by sending packets that cause the normalizer to do a lot of work (e.g., partially overlapping segments). But because RoboNorm delegates the processing of such packets to a slow path (§5), such attacks will largely slow down only the attacker's traffic. Moreover, because very few connections actually have packets on the slow path, the amount of collateral damage the attacker can inflict on benign connections is limited.

The defenses proposed in this section necessarily complicate the design of RoboNorm, but are required for robust operation. Indeed, when considering these added complications, we should keep in mind that normalizer designs that buffer complete payloads suffer from greater vulnerability to memory exhaustion attacks than does RoboNorm.

6.1. Hash store

Eviction policy. When the hash store is full, RoboNorm must evict some old segment hash(es) to make room for new ones. Evicting a hash must amount to resetting the TCP connection, because RoboNorm will no longer be able to check the consistency of a retransmission of that segment. We use a simple cost-benefit analysis to pick TCP connections to evict hashes from. The benefit accrued from picking a particular connection for eviction is equal to the amount of memory the connection consumes—not simply the instantaneous amount of memory it is currently using, but the time-averaged amount it will use for as long as it is active. We must balance the benefit against the cost of evicting the connection. The metric we use for assessing the eviction cost is the loss in network utilization because of the connection's termination. Our eviction policy is then to evict the connection with the highest benefit-to-cost ratio of eviction, i.e., the connection with the highest ratio of the fraction of memory used to the fraction of link bandwidth consumed.

If connection i has data arriving at rate λ_i segments per second and has an average segment size s_i, then the fraction of the link capacity C it uses is $\lambda_i s_i / C$. Regarding its relative memory consumption, by Little's Law the average number of segment hashes connection i consumes is $\lambda_i \delta_i$, where δ_i is the observed average time for the connection's segments to clear. Let H be the total capacity of the hash store, in units of segment hashes. Then we can compute the benefit-cost ratio of eviction of connection i as:

$$\frac{\lambda_i \delta_i}{H} \Big/ \frac{\lambda_i s_i}{C} = \frac{\delta_i C}{s_i H}$$

Since H and C are constants, to find the connection with the highest benefit-cost ratio we look for j that maximizes δ_j / s_j. Thus, our eviction policy boils down to picking connections that either keep segment hashes in the system for too long (large δ_j), or use too many hashes by sending very small segments (small s_j).

Implementing this scheme requires a bit of bookkeeping to determine δ_i of a connection. We can approximate this by averaging the following sample value for each of the connection's segments seen so far: for a cleared segment, the sample is equal to the time it took for the segment to be cleared. For a segment hash still in the system, the sample is equal to its age (the length of time since its creation). Tracking segment age requires associating timestamps with segment hashes.[7] Note that s_i is easy to estimate by averaging the lengths of all the segment hashes seen thus far. Finally, we also need a way to quickly find which connections have the highest δ_i / s_i ratio. We observe that doing so is directly analogous to the Deficit Round Robin computations that modern high-speed routers already implement.

Coalescing hashes. To avoid penalizing benign connections with a small average segment size, we introduce the notion of *coalescing* segment hashes. Coalescing is the process of replacing two (or more) contiguous segment hashes of a connection with one segment hash covering the combined sequence number space, thus reducing the number of segment hashes that need to be stored. Not all hash functions are amenable to segment hash coalescing; we discuss suitable hash functions in §6.4.

When a connection has multiple segment hashes with a small number of bytes in each, coalescing the hashes saves the connection from eviction by increasing its average segment size s_i, while increasing δ_i by (only) the inter-arrival time between the first and last coalesced segments. The only downside of coalescing hashes is that an exactly overlapping retransmission of a coalesced segment will now have to be handled like a partially overlapping retransmission, resulting in increased delays for the connection, and an increase in the amount of memory consumed by the connection in the retransmission buffer. Thus, the combined size of the segments used to form a coalesced segment hash must be limited, say to the maximum TCP segment size. So, if the connection with the largest value of δ_i / s_i either does not have enough segment hashes to coalesce, or if the segment hashes have a large enough size already, then we will still have to evict it to make room in the hash store.

Effect of adversary. The eviction policy of RoboNorm also significantly increases the work-factor that adversaries must apply to impair benign connections. In the absence of such an eviction policy, adversaries can consume segment

[7] In fact, we can probably use a single per-connection timer instead, similar to many implementations of TCP's RTT estimation. However, we have not yet developed the specifi cs of such an approach.

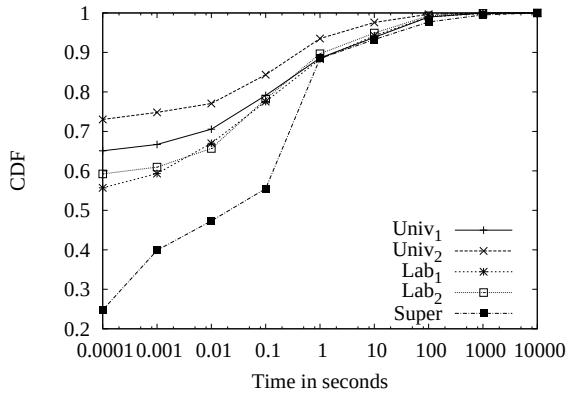

Figure 6. CDF of the maximum duration of inactivity between periods of activity for connections with no outstanding data.

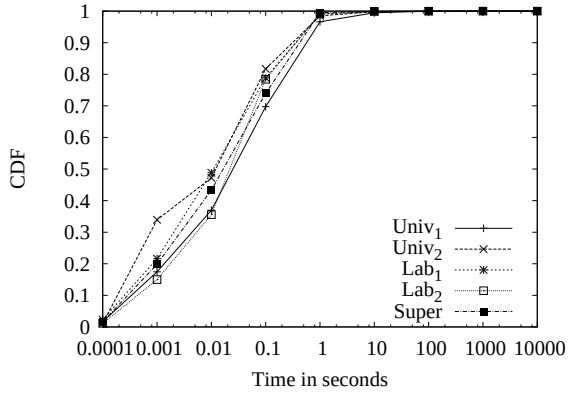

Figure 7. CDF of the maximum duration of inactivity between periods of activity for connections with outstanding data.

hash memory "on the cheap" in one of two ways: (a) by forcing their hashes to stay uncleared for a long time (e.g., by sending data above a sequence hole, which hence will not be acked), or (b) by sending data in small segments, thus consuming a large number of segment hashes. However, the former case increases the δ_i of their connections, and the latter reduces their s_i. Either makes their own connections prone to eviction or coalescing.

6.2. Connection table

The connection table can easily become filled with adversarial connections that have not been successfully established, benign connections that stay idle for long periods of time, or connections that failed to terminate cleanly. To conserve space in the connection table, we augment the table with two Bloom filters as described below.

Keeping track of connection establishment. Initializing state on the first data packet and not on SYN packets makes RoboNorm resilient to SYN floods without requiring any additional mechanism (per §3.1). But not tracking connection establishment by way of the TCP 3-way handshake also makes the system prone to attacks where an adversary creates state in the connection table by sending data packets on non-existent connections that go unanswered, while preventing the receiver from sending RSTs by ensuring the packets do not really reach the receiver (e.g., by setting a low enough TTL or sending packets to unreachable hosts). RoboNorm overcomes this problem by keeping track of connection establishment with very little state in the following manner: upon seeing the SYN ACK segment for a connection, RoboNorm hashes the connection tuple and expected sequence number of the first segment into a *SYN ACK Bloom Filter* (SABF). We then check this Bloom filter for the presence of the connection tuple when the first data packet arrives, as explained below.

Compressing state for connections with no outstanding data. Some connections (e.g., interactive SSH) tend to stay idle for long periods of time without having any data in flight. Figure 6 shows the CDF of the duration of inactivity (that was eventually followed by some activity) for connections that had no outstanding data. We find from the figure that a small number (1–2%) of connections remain idle for hundreds of seconds before sending data again. Because RoboNorm does not need to store any per-connection state for such connections other than the fact that they exist, we can hash the corresponding connection tuples into an *Inactive Connection Bloom Filter* (ICBF) during the idle periods. From our traces, we find that the peak number of connections in the connection table is 35–50% lower if we move connections to the ICBF after 5 minutes of inactivity.

Connection initialization and termination. Given the mechanisms above, we slightly modify the algorithms in §3.1 to initialize and terminate connection state as follows. When a data packet with no entry in the connection table arrives, RoboNorm installs a corresponding entry in the connection table only if it finds the tuple in either the SABF or ICBF. RoboNorm must periodically time out entries in the Bloom filters to prevent them from filling up. We can time out connections in the ICBF at a coarse granularity (e.g., after a few hours) in order to clean up connections that did not terminate properly. Connections in the SABF can be timed out more frequently (e.g., in 10 seconds). There are several ways to efficiently time out connections by keeping track of which Bloom filter bits were accessed in the previous timeout period [12].

For a false positive rate under 0.01%, storing 100,000 connection tuples requires a Bloom filter of about 400,000 cells. Since we require 2 bits per cell of the Bloom filter (the second tracks any access in the timeout period), the size of each Bloom filter is only 100 KB.

Eviction policy. When RoboNorm runs out of space in the connection table, it locates an *inactive connection*, i.e., one for which it has not seen any acknowledgments over the past τ seconds. If the connection has no data pending, we reclaim the connection's slot and place it in the ICBF. If the connection has data outstanding, we must terminate it. From Figure 7, we see that the fraction of connections with outstanding data that stay idle for more than a few seconds is negligible. Thus a value of around 10 sec for τ will result in very few connections being terminated; moreover, terminating such connections is not a large loss because the connection was having great difficulty making progress. If all connections are currently active and have pending data, then absent any other mechanism, either the new connection must be dropped or an existing one reset to make space for the new connection.

Effect of adversary. RoboNorm's policy of evicting inactive connections when running out of space resists adversarial attempts to exhaust the connection table. For example, if the adversary creates connections that do not clear data, or send no data at all, they will be flagged as inactive and eventually evicted. Thus, the adversary can exhaust the connection table only by opening a large number of connections and actively sending data on all of them. If the adversary accomplishes the above by controlling many zombies, this problem is identical to protecting a web server from a network of bots seeking to exhaust its resources, and we can employ one of the numerous defenses used in such situations (e.g., per-IP quotas, white-listing/blacklisting groups of IP addresses, profiles over the IP space). However, it is known that such defenses must be applied with care in order not to penalize legitimate connections.

6.3. Retransmission Buffer

We now discuss how to respond to exhaustion of the memory allocated to retransmission buffers. Recall that we only employ such buffers in the face of connections with partially overlapping retransmissions, which, as indicated in Table 2 (row 2), are rare—less than 0.5% of all connections at any time—with the result that the adversary does not have much leverage to cause collateral damage to benign connections.

That said, a reasonable approach to take is that when the retransmission buffer space is under stress, we limit the amount of memory consumed by a connection in the retransmission buffer. This limit must be at least as large as the maximum TCP segment size and the maximum size of a coalesced segment hash (and this imposes a limit on how much coalescing we can perform on the connection's segment hashes, see §6.1). If a connection exceeds this limit, we drop the excess partially overlapping segments without buffering them; doing so will only increase the perceived loss rate of the connection. Otherwise we can again use δ_i/s_i (as in §6.1) to select another connection's segments to evict. Note that evicting the buffered segments of a connection does not require terminating the connection unless we have promoted an ACK on its behalf (about 5 times more rare, see row 4 of Table 2); it only slows down the connection by requiring additional retransmissions.

6.4. Hash Function

In order to be able to coalesce hashes and thwart memory exhaustion attacks on the hash store (as per the discussion in §6.1), the hash function used to construct segment hashes must have the property that the hash of the concatenation of two byte strings is derivable from the hashes of the two individual byte strings. In the rest of this section, we provide an example of one hash function with this property, and discuss its security properties.

Consider the following universal hash function [13]: the n-bit hash of a bitstring X is computed as $H_n(X) = (a_n \cdot X + b_n) \pmod{p_n}$, where X is the numeric value of the bit-string, p_n is an n-bit prime number that is kept secret and a_n and b_n are random numbers chosen from $\{1, \ldots, p-1\}$ and $\{0, \ldots, p-1\}$ respectively. Notice that to obtain the hash $H_n(X.Y)$ of the concatenation of two k-bit strings X and Y, one simply calculates $[H_n(Y) + 2^k \cdot (H_n(X) - b_n)] \pmod{p_n}$. The RoboNorm design uses 8-byte hashes (i.e., $n = 64$); we now defend this choice in the context of possible attacks on the hash function.

An attacker can compromise the correctness guarantees of RoboNorm by producing collisions under RoboNorm's hash function, and subsequently generating inconsistent retransmissions without being detected. The attacker can test whether he has successfully generated a collision by generating an inconsistent retransmission using two bit-strings that the attacker believes hash to the same value, and checking whether RoboNorm can detect the inconsistency. If the connection does not get killed in spite of the inconsistent retransmission, the attacker knows that he has created a collision under RoboNorm's hash function.

We argue that the attacker cannot break the hash function by guessing p_n, a_n, and b_n. Recall that the number of prime numbers less than any number m is $O(\frac{m}{\log m})$. Thus the number of n-bit prime numbers is approximately $O(\frac{2^n}{n}) - O(\frac{2^{n-1}}{n-1}) \approx O(\frac{2^{n-1}}{n-1})$. To break the hash func-

tion by guessing p_n, a_n, and b_n, the attacker must search through each of the possible $O(\frac{2^{n-1}}{n-1})$ primes, and for each prime the approximately 2^n possible values of the random numbers a_n and b_n. We can see that guessing the hash function in this manner is computationally infeasible for $n = 64$ bit hashes that RoboNorm uses.

If the attacker does not know p_n, a_n, or b_n, then the only way he can hope to generate colliding strings is by randomly guessing pairs of strings. By the property of universal hash functions, the probability that the attacker guesses a retransmission that hashes to the same value as any given n-bit original segment hash is $\frac{1}{2^n}$. This probability is vanishingly small for $n = 64$ bit hashes, even if the attacker splits his search amongst a large number of (say, a billion) parallel connections.

To summarize, coping with attacks on RoboNorm boils down to developing strategies to handle memory exhaustion gracefully and choosing appropriate secure hash functions. We employ two principal ideas here: first, a simple benefit-to-cost eviction scheme that we applied to both the hash store and retransmission buffer; and second, additional Bloom filters to augment the connection table.

7. Related Work

Section 1 briefly described the history of evasion attacks and the normalization problem. Recent work [6] addresses one type of evasions, namely an attacker attempting to prevent a specific signature match against text they transmit. The authors developed a scheme based on introducing a modest change in end-system TCP behavior in order to allow a monitor to detect attempts to ambiguously transmit byte sequences that match a given set of signatures. Their scheme is appealing in that by exploiting the introduced end-system change, they avoid needing to reassemble TCP byte streams. However, their scheme is also significantly limited in that it only applies to evasions that correspond to directly manipulating a known byte-sequence signature. As such, the scheme does not handle cases where the ambiguity does not constitute an actual attack in itself, but only confuses the monitor's protocol parsing and obscures the occurrence of an attack later in the stream.

Sugawara et al. [14] describe an FPGA-based solution to efficient TCP stream-level signature detection. Their system detects inconsistent retransmissions by storing hashes of transmitted packets. To handle retransmissions that do not overlap with original segment boundaries, the authors simply propose holding onto the partial overlaps till other packets that "fill the gap" arrive. However, our trace evaluation shows that such an approach will result in a significant number of connections stalling on pending consistency checks (see Figure 4(ii)); RoboNorm addresses this problem with the ACK promotion mechanism (§3.3).

Normalization as a general feature has been incorporated into secure operating systems [15] and commercial products [16]. Some of these latter include explicit options to check for inconsistent retransmissions [17], but do not provide technical details as to how such detection works. From informal discussions with other vendors, it appears that a common approach is to use payload hashes, but without addressing the crucial problem of misaligned retransmissions for which the hashes cannot be matched.

Shankar and Paxson explored a different approach to defending against evasion attacks which they term "Active Mapping" [18]. Here, the idea is for the network monitor to proactively determine how specific end systems and network paths will resolve potential ambiguities. While this approach is a valid point in the overall design space, we argue that eliminating ambiguities, rather than attempting to correctly guess their outcome, provides a more robust foundation for security monitoring technology.

Work by Levchenko et al. demonstrates in formal terms that many security detection tasks (e.g., detecting SYN flooding, port scans, connection hijacking and evasion attacks) fundamentally require maintaining per-connection state [19]. This finding highlights the importance of reducing the amount of per-connection state.

In work that is complementary to ours, Dharmapurikar et al. explore how to robustly reassemble TCP byte streams when faced with adversaries who attempt to overwhelm the accompanying state management [20]. Reassembly involves maintaining out-of-order data only until sequence "holes" are filled, while normalization requires maintaining data until it is acknowledged and hence requires a different solution.

8. Conclusion

Defending networks against today's attackers is especially challenging for modern intrusion detection/prevention systems for two reasons: the sheer amount of state they must maintain, and the possibility of resource-exhaustion attacks on the defense system itself. Our work shows how to cope with these challenges in the context of a TCP stream normalizer called RoboNorm, whose job is to detect all instances of inconsistent TCP retransmissions.

The two currently used methods to detect inconsistent retransmissions—maintaining complete contents of unacknowledged data, or maintaining only the corresponding hashes—suffer from a set of flaws each. Systems that maintain complete contents consume an amount of memory problematic for high-speed operation. Systems that maintain hashes cannot verify the consistency of the 20–30% of retransmissions that fail to preserve original segment boundaries; as a result attackers can easily encode their evasions in these unverified segments. RoboNorm stores hashes

of outstanding data and, with a careful design and occasional alteration of end-to-end semantics, verifies the consistency of *all* retransmissions. The resulting design is necessarily somewhat complex, but still has a compact state machine and is implementable at high speeds.

In considering resource exhaustion attacks, the observation that provisioning for a worst-case traffic pattern is simply impractical led us to develop a simple benefit-to-cost framework to evict connections when space is at a premium. Another challenge is deciding when to initialize state for new connections and when to reclaim state for active connections with no outstanding data (of which there are many); for both problems, we outline how we can use Bloom filters effectively.

Evaluating RoboNorm on a set of traces collected from different networks shows that it consumes 1–2 orders of magnitude less memory than the approach of storing all outstanding bytes, while guaranteeing that all inconsistent retransmissions will be detected. Thus, our most important conclusion is that high-speed TCP stream normalization does not have to choose between correctness and implementability—it can achieve both goals, while resisting a range of resource exhaustion attacks.

Acknowledgments

We thank Stefan Savage, Robin Sommer, Nick Weaver, and the anonymous reviewers for many useful comments and suggestions. This work was supported by the National Science Foundation grants ITR/ANI-0205519, NSF-0433702, CNS-0627320, CNS-0716273, NSF-0716636, and in part by a Cisco graduate fellowship, for which we are grateful. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors or originators and do not necessarily reflect the views of the National Science Foundation or Cisco Systems.

References

[1] M. Roesch, "Snort: Lightweight intrusion detection for networks," in *Proc. USENIX LISA*, November 1999.

[2] V. Paxson, "Bro: a System for Detecting Network Intruders in Real-Time," *Computer Networks*, vol. 31, no. 23-24, pp. 2435–2463, 1999.

[3] T. H. Ptacek and T. N. Newsham, "Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection," Secure Networks, Inc., Tech. Rep., Jan. 1998, http://insecure.org/stf/secnet_ids/secnet_ids.html.

[4] G. R. Malan, D. Watson, F. Jahanian, and P. Howell, "Transport and Application Protocol Scrubbing," in *Proc. IEEE INFOCOM*, Apr. 2000.

[5] M. Handley, V. Paxson, and C. Kreibich, "Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics," in *Proc. USENIX Security Symposium*, Aug. 2001.

[6] G. Varghese, J. A. Fingerhut, and F. Bonomi, "Detecting Evasion Attacks at High Speeds Without Reassembly," in *Proc. ACM SIGCOMM*, Sept. 2006.

[7] Dug Song, "fragroute," http://monkey.org/~dugsong/fragroute/.

[8] Sean Kerner, "Open Source Metasploit Improves Evasion," Aug. 2006, http://www.internetnews.com/dev-news/article.php/3624501.

[9] Federico Biancuzzi, "Metasploit 3.0 day," May 2007, http://www.securityfocus.com/columnists/439.

[10] J. Postel, *RFC 793: Transmission Control Protocol*, Sept. 1981.

[11] H. Dreger, A. Feldmann, V. Paxson, and R. Sommer, "Operational Experiences with High-Volume Network Intrusion Detection," in *Proceedings of CCS*, 2004.

[12] F. Bonomi, M. Mitzenmacher, R. Panigrah, S. Singh, and G. Varghese, "Beyond Bloom Filters: From Approximate Membership Checks to Approximate State Machines," in *Proc. ACM SIGCOMM*, Sept. 2006.

[13] J. L. Carter and M. N. Wegman, "Universal classes of hash functions (extended abstract)," in *Proc. of ACM Symposium on Theory of computing*, 1977, pp. 106–112.

[14] Y. Sugawara, M. Inaba, and K. Hiraki, "High-speed and Memory Efficient TCP Stream Scanning Using FPGA," in *Proc. International Conference on Field Programmable Logic and Applications*, Aug. 2005.

[15] OpenBSD, "PF: Scrub (Packet Normalization)," 2006, http://www.openbsd.org/faq/pf/scrub.html.

[16] Cisco Systems, Inc., "Configuring TCP/IP Normalization and IP Reassembly Parameters," 2006, www.cisco.com/univercd/cc/td/doc/product/lan/cat6000/mod_icn/ace/ace_301/securgd/tcpipnrm.pdf.

[17] ——, "Configuring TCP Normalization," 2006, http://www.cisco.com/en/US/products/ps6120/products_configuration_guide_chapter09186a008054ecb8.html#wp1051891.

[18] U. Shankar and V. Paxson, "Active Mapping: Resisting NIDS Evasion Without Altering Traffic," in *Proc. IEEE Symposium on Security and Privacy*, May 2003.

[19] K. Levchenko, R. Paturi, and G. Varghese, "On the Difficulty of Scalably Detecting Network Attacks," in *Proc. ACM CCS*, Oct. 2004.

[20] S. Dharmapurikar and V. Paxson, "Robust TCP Stream Reassembly in the Presence of Adversaries," in *Proc. USENIX Security Symposium*, Aug. 2005.

#	Trace Characteristics	$\textbf{Univ}_1$	$\textbf{Univ}_2$	$\textbf{Lab}_1$	$\textbf{Lab}_2$	Super
1	Date recorded	31Aug04	07Apr05	20Sep05	16Jan04	26Aug04
2	Trace duration (sec)	300*	7,221	6,167	4,345	3,606
3	Contents	All hdrs	All pkts	All TCP	All hdrs	TCP hdrs
4	Reported capture losses	$3.3 \cdot 10^{-6}$	0	$3.0 \cdot 10^{-3}$	$4.8 \cdot 10^{-5}$	1.56%

Table 5. Summary of the collection method for the traces used in the paper.

A Trace Collection

Table 5 summarizes the collection methodology for the traces used in the paper. The volume of traffic at most of the sites is sufficiently high that it is difficult to capture packet traces without loss. The exception to this is the Univ_2 trace, which was recorded using specialized hardware that is able to keep up with the high volume. For the other sites, while we incurred non-zero capture losses, the reported rates were low, as shown in row 4 of the table, other than for Super, which incurred 1.56% reported losses. For the most part, losses introduce imprecision in our quantification of frequencies of various packet transmission patterns, but should not cause significant bias, since it is reasonable to assume that packet capture loss does not particularly correlate with packet transmission patterns.

We also note that Univ_1 is unusual compared to the other traces in that it represents a composite made out of 19 independently captured traces, each of which utilized per-connection sampling to record approximately 1/19th of the total TCP (and UDP) traffic (the only available way to capture traffic in that environment without massive losses). Each of these sub-traces spanned 300 seconds, and were recorded one after another. To derive results from the composite trace, we either add up per-subtrace figures (when computing aggregates), or take maxima across them (when assessing per-connection worst-case behavior).

B Sizing a Segment Hash

First, a segment hash must contain a 8-byte collision-resistant hash of the contents of the corresponding TCP segment. Next, we need to associate a range of sequence numbers with each hash. We can do so by explicitly storing the ending sequence number as a 2-byte offset from the starting sequence number, and implicitly determining the starting sequence number by assuming it comes 1 octet after the ending sequence number of the previous segment hash (or from the cumulative acknowledgment sequence number in the connection record, for the first segment hash). With such a scheme, we also need to introduce dummy segment hashes for any "holes" in the sequence space for which we have not received any data. The overhead of such dummy segment hashes is expected to be low because connections rarely have more than one such hole at a time [20]. Third, we can track the age of a segment (required by the eviction policies, see §6) with millisecond precision using a two-byte timestamp. Finally, we need a pointer to the next segment hash. We assume such pointers require 3 bytes, as our data structures fit comfortably in the 16 MB range that we can address using 24-bit pointers. Adding these numbers up, we find that each segment hash consumes 15 bytes.

C Sizing the Connection Table

For each direction of a connection, we need to store a 3-byte pointer to the start of the segment hash list of the connection, and a 3-byte pointer to the retransmission buffer space for the connection (nil if the connection does not need one). We also need 4 bytes to store the cumulative acknowledgment sequence number for that direction, as this provides the basis for the relative sequence number used in the first hash of the segment hash list. To append new hashes to the hash list quickly without traversing the entire list, we also maintain a 3-byte pointer to the end of the hash list and a 4 byte sequence number of the last byte seen so far. Thus, each direction consumes 17 bytes.

If the normalizer is working on both directions of data, we need 34 bytes as described above, plus the 12-byte connection tuple (source and destination IP addresses and port numbers, 4+4+2+2 bytes), plus some space to hold bookkeeping information (average clearing time of hashes and average segment size per connection; see §6), which we assume that with careful choice of units and encoding requires 2 more bytes, a total of 48 bytes. Note that we store state for both directions of the connection in the same connection record, even though our algorithms treat each direction separately.