Robust TCP Stream Reassembly In the Presence of Adversaries

Sarang Dharmapurikar
Washington University in Saint Louis
sarang@arl.wustl.edu

Vern Paxson
International Computer Science Institute, Berkeley
vern@icir.org

Abstract

There is a growing interest in designing high-speed network devices to perform packet processing at semantic levels above the network layer. Some examples are layer-7 switches, content inspection and transformation systems, and network intrusion detection/prevention systems. Such systems must maintain per-flow state in order to correctly perform their higher-level processing. A basic operation inherent to per-flow state management for a transport protocol such as TCP is the task of reassembling any out-of-sequence packets delivered by an underlying unreliable network protocol such as IP. This seemingly prosaic task of reassembling the byte stream becomes an order of magnitude more difficult to soundly execute when conducted in the presence of an *adversary* whose goal is to either subvert the higher-level analysis or impede the operation of legitimate traffic sharing the same network path.

We present a design of a hardware-based high-speed TCP reassembly mechanism that is robust against attacks. It is intended to serve as a module used to construct a variety of network analysis systems, especially intrusion prevention systems. Using trace-driven analysis of out-of-sequence packets, we first characterize the dynamics of benign TCP traffic and show how we can leverage the results to design a reassembly mechanism that is efficient when dealing with non-attack traffic. We then refine the mechanism to keep the system effective in the presence of adversaries. We show that although the damage caused by an adversary cannot be completely eliminated, it is possible to mitigate the damage to a great extent by careful design and resource allocation. Finally, we quantify the trade-off between resource availability and damage from an adversary in terms of *Zombie equations* that specify, for a given configuration of our system, the number of compromised machines an attacker must have under their control in order to exceed a specified notion of "acceptable collateral damage."

1 Introduction

The continual growth of network traffic rates and the increasing sophistication of types of network traffic processing have driven a need for supporting traffic analysis using specialized hardware. In some cases the analysis is in a purely passive form (intrusion detection, accounting, performance monitoring) and for others in an active, in-line form (intrusion prevention, firewalling, content transfor-

mation, network address translation). Either way, a key consideration is that increasingly the processing must operate at a semantic level higher than the network layer; in particular, we often can no longer use stateless processing but must instead maintain per-flow state in order to correctly perform higher-level analyses.

Such stateful analysis brings with it the core problem of *state management*: what hardware resources to allocate for holding state, how to efficiently access it, and how to reclaim state when the resources become exhausted. Designing a hardware device for effective state management can require considerable care. This is particularly the case for in-line devices, where decisions regarding state management can adversely affect network operation, such as prematurely terminating established TCP connections because the device no longer has the necessary state to correctly transform the flow.

Critically, the entire problem of state management becomes an order of magnitude more difficult when conducted in the presence of an *adversary* whose goal is to either subvert the hardware-assisted analysis or impede the operation of legitimate traffic along the same network path.

Two main avenues for subverting the analysis ("evasion") are to exploit the ambiguities inherent in observing network traffic mid-stream [18, 12] or to cause the hardware to discard the state it requires for sound analysis. If the hardware terminates flows for which it has lost the necessary state, then the adversary can pursue the second goal of impeding legitimate traffic—i.e., denial-of-service, where rather than targeting the raw capacity of the network path or end server, instead the attacker targets the newly-introduced bottleneck of the hardware device's limited state capacity.

Issues of state-holding, disambiguation, and robust operation in the presence of flooding arise in different ways depending on the semantic level at which the hardware conducts its analysis. In this paper we consider one of the basic building blocks of higher-level analysis, namely the conceptually simple task of reassembling the layer-4 byte streams of TCP connections. As we will show, this seemingly prosaic bookkeeping task—just track the connection's sequence numbers, buffer out-of-sequence data,

lay down new packets in the holes they fill, and deliver to the next stage of processing any bytes that are now in-order—becomes subtle and challenging when faced with *(i)* limited hardware resources and, more importantly, *(ii)* an adversary who wishes to either undermine the soundness of the reassembly or impair the operation of other connections managed by the hardware.

While fast hardware for robust stream reassembly has a number of applications, we focus our discussion on enabling high-speed *intrusion prevention*. The basic model we assume is a high-speed, in-line network element deployed at a site's gateway (so it sees both directions of the flows it monitors). This module serves as the first stage of network traffic analysis, with its output (reassembled byte streams) feeding the next stage that examines those byte streams for malicious activity. This next stage might also execute in specialized hardware (perhaps integrated with the stream reassembly hardware), or could be a functionally distinct unit.

A key consideration is that because the reassembly module is in-line, it can *(i)* normalize the traffic [12] prior to subsequent analysis, and *(ii)* enable intrusion *prevention* by only forwarding traffic if the next stage signals that it is okay to do so. (Thus, by instead signaling the hardware to discard rather than forward a given packet, the next stage can prevent attacks by blocking their delivery to the end receiver.) As we will discuss, another key property with operating in-line is that the hardware has the potential means of *defending* itself if it finds its resources exhausted (e.g., due to the presence of state-holding attacks). It can either reclaim state that likely belongs to attacker flows, or else at least exhibit graceful degradation, sacrificing performance first rather than connectivity.

A basic notion we will use throughout our discussion is that of a sequence gap, or *hole*, which occurs in the TCP stream with the arrival of a packet with a sequence number greater than the expected sequence number. Such a hole can result from packet loss or reordering. The hardware must buffer out-of-order packets until a subsequent packet fills the gap between the expected and received sequence numbers. After this gap is filled, the hardware can then supply the byte-stream analyzer with the packets in the correct order, which is crucial for higher-level semantic analysis of the traffic stream.

At this point, the hardware can release the buffer allocated to the out-of-order packet. However, this process raises some natural questions: if the hardware has to buffer all of the out-of-order packets for all the connections, how much buffer does it need for a "typical" TCP traffic? How long do holes persist, and how many connections exhibit them? Should the hardware immediately forward out-of-order packets along to the receiver, or only after they have been inspected in the correct order?

To explore these questions, we present a detailed trace-driven analysis to characterize the packet re-sequencing phenomena seen in regular TCP/IP traffic. This analysis informs us with regard to provisioning an appropriate amount of resources for packet re-sequencing. We find that for monitoring the Internet access link of sites with several thousand hosts, less than a megabyte of buffer suffices.

Moreover, the analysis helps us differentiate between benign TCP traffic and malicious traffic, which we then leverage to realize a reassembly design that is robust in the presence of adversaries. After taking care of traffic normalization as discussed above, the main remaining threat is that an adversary can attempt to overflow the hardware's re-sequencing buffer by creating numerous sequence holes. If an adversary creates such holes in a distributed fashion, spreading them across numerous connections, then it becomes difficult to isolate the benign traffic from the adversarial.

Tackling the threat of adversaries gives rise to another set of interesting issues: what should be done when the buffer overflows? Terminate the connections with holes, or just drop the buffered packets? How can we minimize the collateral damage? In light of these issues, we devise a buffer management policy and evaluate its impact on system performance and security.

We frame our analysis in terms of *Zombie equations*: that is, given a set of operational parameters (available buffer, traffic volume, acceptable collateral damage), how many total hosts ("zombies") must an adversary control in order to inflict an unacceptably high degree of collateral damage?

The rest of the paper is organized as follows. Section 2 discusses the related work. In Section 3 we present the results of our trace-driven analysis of out-of-sequence packets. Section 4 describes the design of a basic reassembly mechanism which handles the most commonly occurring re-ordering case. In Section 5, we explore the vulnerabilities of this mechanism from an adversarial point of view, refine it to handle attacks gracefully, and develop the Zombie equations quantifying the robustness of the system. Section 6 concludes the paper.

2   Related Work

Previous work relating to TCP stream reassembly primarily addresses *(i)* measuring, characterizing and modeling packet loss and reordering, and *(ii)* modifying the TCP protocol to perform more robustly in the presence of sequence holes.

Paxson characterized the prevalence of packet loss and reordering observed in 100 KB TCP transfers between a number of Internet hosts [16], recording the traffic at both sender and receiver in order to disambiguate behavior. He

found that many connections are loss-free, and for those that are not, packet loss often comes in bursts of consecutive losses. We note that such bursts do not necessarily create large sequence holes—if all packets in a flight are lost, or all packets other than those at the beginning, then no hole is created. Similarly, consecutive retransmissions of the same packet (which would count as a loss burst for his definition) do not create larger holes, and again might not create any hole if the packet is the only one unacknowledged. For packet reordering, he found that the observed rate of reordering varies greatly from site to site, with about 2% of all packets in his 1994 dataset being reordered, and 0.6% in his 1995 dataset. However, it is difficult to gauge how we might apply these results to today's traffic, since much has changed in terms of degree of congestion and multipathing in the interim.

Bennett and colleagues described a methodology for measuring packet reordering using ICMP messages and reported their results as seen at a MAE-East router [5]. They found that almost 90% of the TCP packets were reordered in the network. They provide an insightful discussion over the causes of packet reordering and isolate the packet-level parallelism offered by packet switches in the data path as the main culprit. Our observations differ significantly from theirs; we find that packet reordering in TCP traffic affects 2–3% of the overall traffic. We attribute this difference to the fact that the results in [5] reflect an older generation of router architecture. In addition, it should be mentioned that some router vendors have modified or are modifying router architectures to provide connection-level parallelism instead of packet level-parallelism in order to eliminate reordering [1].

Jaiswal and colleagues presented measurements of out-of-sequence packets on a backbone router [13]. Through their passive measurements on recent OC-12 and OC-48 traces, they found that packet reordering is seen for 3–5% of overall TCP traffic. This more closely aligns with our findings. Gharai and colleagues presented a similar measurement study of out-of-order packets using end-to-end UDP measurements [11]. They too conclude that reordering due to network parallelism is more prevalent than the packet loss.

Bellardo and Savage devised a clever scheme for measuring TCP packet reordering from a single endpoint and discriminating between reordering on the forward path with that on the reverse path [4]. (For many TCP connections, reordering along one of the paths is irrelevant with respect to the formation of sequencing holes, because data transfers tend to be unidirectional, and reordered acknowledgments do not affect the creation of holes.) They quantify the degree that reordering rates increase as the spacing between packets decreases. The overall reordering rates they report appear consistent with our own observations.

Laor et. al. investigated the effect of packet reordering on application throughput [14]. In a laboratory with a Cisco backbone router connecting multiple end-hosts running different OSes, they measured HTTP throughput as a function of injected packet reordering. Their experiments were however confined to cases where the reordering elicited enough duplicate-ACKs to trigger TCP's "fast retransmission" in the sender. This leads to a significant degradation in throughput. However, we find that this degree of reordering does not represent the TCP traffic behavior seen in actual traffic, where very few reordered packets cause the sender to retransmit spuriously.

Various algorithms have been proposed to make TCP robust to packet reordering [6, 7, 21]. In [6], Blanton and Allman propose to modify the duplicate-ACK threshold dynamically to minimize the effect of duplicate retransmissions on TCP throughput degradation. Our work differs from theirs in that we use trace-driven analysis to guide us in choosing various parameters to realize a robust reassembly system, as well as our interest in the complications due to sequence holes being possibly created by an adversary.

In a project more closely related to our work, Schuehler et al. discuss the design of a TCP processor that maintains per-flow TCP state to facilitate application-level analysis [20]. However, the design does not perform packet reordering—instead, out-of-order packets are dropped. There are also some commercial network processors available today that perform TCP processing and packet reassembly. Most of these processors, however, are used as TCP offload engines in end hosts to accelerate TCP processing. To our knowledge, there are few TCP processors which process and manipulate TCP flows in-line, with Intel's TCP processor being one example [10]. TCP packets are reordered using a CAM that stores the sequence numbers of out-of-order packets. When a new data packet arrives, the device compares its sequence number against the CAM entries to see if it plugs any of the sequence holes. Unfortunately, details regarding the handling of edge cases do not appear to be available; nor is it clear how such processors handle adversarial attacks that aim to overflow the CAM entries.

Finally, Paxson discusses the problem of state management for TCP stream reassembly in the presence of an adversary, in the context of the Bro intrusion detection system [17]. The problem is framed in terms of when to release memory used to hold reassembled byte streams, with the conclusion being to do so upon observing an acknowledgment for the data, rather than when the data first becomes in-sequence, in order to detect inconsistent TCP retransmissions. The paper does not discuss the problem of managing state for out-of-sequence data; Bro simply buffers such data until exhausting available memory, at which point the system fails.

	$Univ_{sub}$	$Univ_{19}$	Lab_{lo}	Lab_2	Super	T3	Munich
Trace duration (seconds)	303	5,697 / 300*	3,602	3,604	3,606	10,800	6,167
Total packets	1.25M	6.2M	1.5M	14.1M	3.5M	36M	220M
Total connections	53K	237K	50K	215K	21K	1.04M	5.62M
Connections with holes	1,146	17,476	4,469	41,611	598	174,687	714,953
Total holes	2,048	29,003	8,848	79,321	4,088	575K	1.88M
Max buffer required (bytes)	128 KB	91 KB	68 KB	253K	269 KB	202 KB	560KB
Avg buffer required (bytes)	5,943	2,227	3,111	13,392	122	28,707	178KB
Max simultaneous holes	15	13	9	39	6	94	114
Max simultaneous holes in single connection	9	16	6	16	6	85	61
Fraction of holes with < 3 packets in buffer	90%	87%	90%	87%	97%	85%	87%
Fraction of connections with single concurrent hole	96%	98%	96%	97%	97%	95%	97%
Fraction of holes that overlap hole on another connection of same *external* host (§ 5.1)	0.5%	0.02%	0.06%	0.06%	0%	0.46%	0.02%

Table 1: **Properties of the datasets used in the study**

3 Trace Analysis

In this section we present an analysis of a set of TCP packet header traces we obtained from the access links at five sites: two large university environments, with about 45,000 hosts (the $Univ_{sub}$ and $Univ_{19}$ traces) and 50,000 hosts (*Munich*), respectively; a research laboratory with about 6,000 hosts (*Lab_{lo}*, and *Lab_2*); a supercomputer center with 3,000 hosts (*Super*); and an enterprise of 10,000 hosts connected to the Internet via a heavily loaded T3 link (*T3*). The *Super* site is unusual in that many of its hosts are not typical end-user machines but rather belong to CPU "farms," some of which make very high-speed and high-volume connections; and the *T3* site funnels much of its traffic through a small number of Web and SMTP proxies.

While we cannot claim that these traces are broadly representative, they do span a spectrum from many hosts making small connections (the primary flavor of $Univ_{sub}$, $Univ_{19}$, and *Munich*) to a few hosts making large, fast connections (*Super*). We obtained traces of the inbound and outbound traffic at each site's Internet access link; for all but *T3*, this was a Gbps Ethernet. The volume of traffic at most of the sites is sufficiently high that it is difficult to capture packet header traces without loss. The exception to this is the *Munich* dataset, which was recorded using specialized hardware able to keep up with the high volume. For the other sites, most of the traces we obtained were filtered, as follows.

We captured $Univ_{sub}$ off-hours (10:25 PM, for 5 minutes) with a filter that restricted the traffic to one of the university's three largest subnets. tcpdump reported very little loss (88 packets out of 22.5M before filtering). Thus, this trace is best interpreted as reflecting the performance

we would see at a good-sized university rather than a quite large university.

We captured $Univ_{19}$ in a somewhat unusual fashion. We wanted a trace that reflected the aggregate university traffic, but this volume far exceeded what tcpdump could capture on the monitoring host. While sampling is a natural fallback, a critical requirement is that we must be able to express the sampling in a form realizable by the kernel's BPF filtering engine—we cannot bring the packets up to user space for sampling without incurring a high degree of measurement loss. We instead used a BPF expression that adds the addresses and ports (both source and destination) together and then computes their residue modulo a given prime P (which can be expressed in BPF using integer division and multiplication). We take all of the packets with a given residue, which gives us a 1-in-P per-connection sample.

$Univ_{19}$ was captured using $P = 19$ and cycling through all of the possible residues $0 \ldots 18$, capturing 5 minutes per residue. The traces were made back-to-back during a workday afternoon. Out of 631M total packets seen by the packet filter, tcpdump reported 2,104 drops. We argue that this is an acceptable level, and note that the presence of measurement drops in the trace will tend to introduce a minor bias towards a more pessimistic view of the prevalence of holes.

We then form $Univ_{19}$ by analyzing the 19 traces and combining the results, either adding up the per-subtrace figures, or taking the maximum across them. For example, when computing the maximum buffer size induced by holes, we take the maximum of the maximums computed for each of the 19 traces. Doing so gives a likely approximation to what would have been seen in a full 5-

minute trace, since we find that "local" maxima are generally short-lived and thus would not likely overlap in time.

On the other hand—and this is a major point to bear in mind—the short duration of the $Univ_{sub}$ and $Univ_{19}$ traces introduces a significant bias towards *underestimating* the prevalence of holes. This comes both from the short lifetimes of the traces (less opportunity to observe diverse behavior) and, perhaps more significantly, from the *truncation effect*: we do not analyze connections that were already in progress when a trace began, or that have not finished upon termination of the trace, because we do not accurately know their state in terms of which packets constitute holes (upon trace startup) or how long it takes holes to resolve (upon trace termination). This will tend to bias our analysis towards under-representing long-running connections, and these may in turn be responsible for a disproportionate number of holes. However, the overall consistency of the results for the university traces with those for the longer-lived traces suggests that the general findings we base on the traces—that buffer required for holes are modest, connections tend to have few holes that take little per-hole memory, and that holes resolve quickly—remain plausible. In addition, the similar *Munich* environment does not suffer from these biases. Its results mostly agree qualitatively with those from $Univ_{19}$, except it shows a much higher level of average concurrent holes, which appears to be due to a higher prevalence of fine-grained packet reordering.

The first of the research lab traces, Lab_{lo}, was extracted from ongoing tracing that the lab runs. This tracing uses an elaborate filter to reduce the total traffic to about 5% of its full volume, and this subset is generally recorded without any measurement drops. The filtering includes eliminating traffic corresponding to some popular ports; in particular, HTTP, which includes the majority of the site's traffic. Thus, this trace is more simply a touchstone reflecting a lower-volume environment.

Lab_2 includes all packet headers. It was recorded during workday afternoon hours. The packet filter inspected 46M packets, reporting about 1-in-566 dropped. *Super* is a full header trace captured during workday afternoon hours, with the filter inspecting 13.5M packets and reporting no drops.

$T3$ is a three-hour full header trace captured during workday afternoon hours, with the filter capturing 101M packets and reporting no drops. The mean inbound data rate over the three hours was 30.3 Mbps (with the link having a raw capacity of 44.7 Mbps); outbound was 11.0 Mbps. Note that the actual monitoring was at a Gbps Ethernet link just inside of the T3 bottleneck, so losses induced by the congested T3 on packets arriving from the exterior Internet would show up as holes in the trace, but losses induced on traffic outbound from the site would not. However, the figures above show that the congestion was
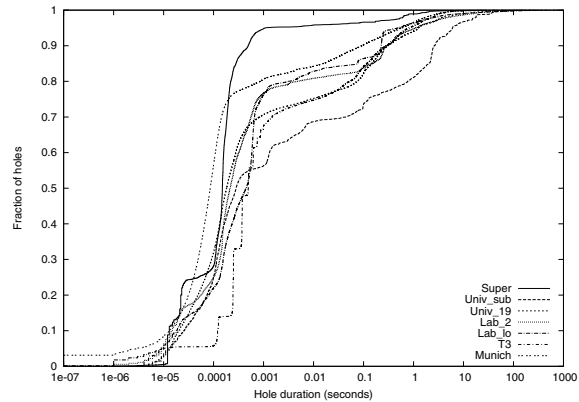

Figure 2: **Cumulative distribution function of the duration of holes. Most of the holes have a lifetime of less than 0.01 seconds.**

primarily for the inbound traffic. We note that monitoring in this fashion, just inside of a bottleneck access link, is a natural deployment location for intrusion prevention systems and the like.

Table 1 summarizes the datasets and some of the characteristics of the sequence holes present in their TCP connections. We see that holes are very common: in $Univ_{sub}$ and *Super*, about 3% of connections include holes, while in Lab_{lo} and *T3*, the number jumps to 10–20%. Overall, 0.1%–0.5% of all packets lead to holes.

Figure 1 shows how the reassembly buffer occupancy changes during the traces. Of the four traces, *Super* is peculiar: the buffer occupancy is mostly very low, but surges to a high value for a very short period. This likely reflects the fact that *Super* contains fewer connections, many of which do bulk data transfers.

It is important to note the de-synchronized nature of the sequence hole creation phenomenon. A key point is that the buffer occupancy remains below 600 KB across all of the traces, which indicates that stream reassembly over a set of several thousand connections requires only a modest amount of memory, and thus may be feasible at very high speeds.

It is also noteworthy how some holes last for a long duration and keep the buffer level elevated. These are visible as plateaus in several of the figures—for example, between $T = 100$ and $T = 150$ in the $Univ_{sub}$ plot—and are due to some long lived holes whose durations overlap, whereas for the *Munich* trace we observed that the average buffer occupancy is significantly higher than the rest of the traces. This too is a result of concurrent but short-lived holes, although the average number of concurrent holes for this trace is larger (around 60) compared to the other traces (< 5 concurrent holes on an average).

The frequent sudden transitions in the buffer level show that most of the holes are quite transient. Indeed, Figure 2
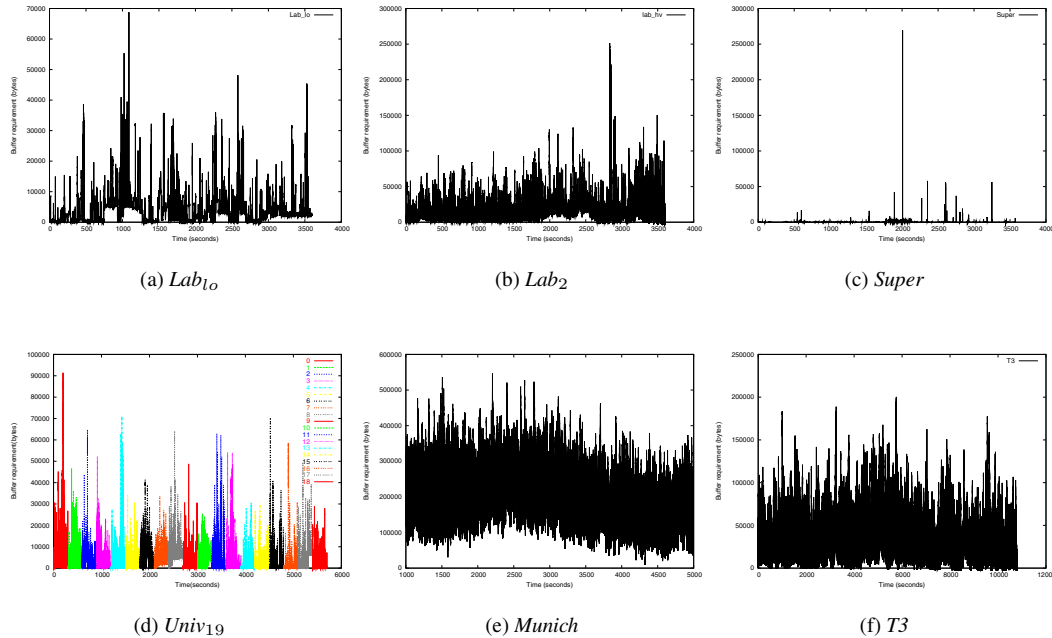
(a) Lab_{lo} (b) Lab_2 (c) *Super*

(d) $Univ_{19}$ (e) *Munich*   (f) *T3*

Figure 1: **Reassembly buffer occupancy due to unfilled holes. $Univ_{sub}$, which we omitted, is similar to the elements of $Univ_{19}$.**
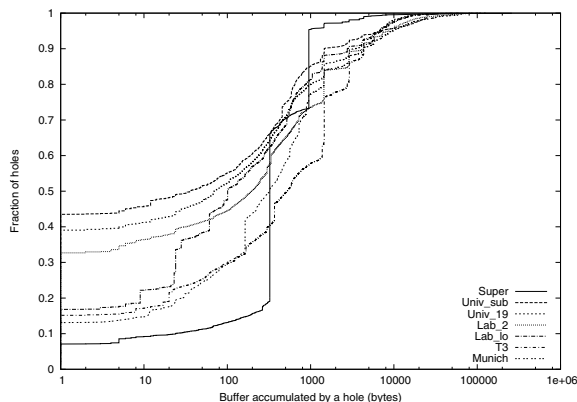

Figure 3: **Cumulative distribution of the buffer accumulated by a hole.**

shows the cumulative distribution of the duration of holes. Most holes have a very short lifetime, strongly suggestive that they are created due to packet reordering and not packet loss, as in the latter case the hole will persist for at least an RTT, significantly longer than a millisecond for non-local connections. The average hole duration is less than a millisecond. In addition, the short-lived holes have a strong bias towards the out-of-order packet (sent later, arriving earlier) being smaller than its later-arriving predecessor, which is suggestive of reordering due to multi-pathing.

Finally, in Figure 3 we plot the cumulative distribution of the size of the buffer associated with a single hole. The graph shows that nearly all holes require less than 10 KB of buffer. This plot thus argues that we can choose an appropriate limit on the buffer-per-hole so that we can identify an adversary trying to claim an excessively large portion.

4 System architecture

Since our reassembly module is an *in-line* element, one of its key properties is the capability to transform the packet stream if needed, including dropping packets or killing connections (by sending TCP RST packets to both sides and discarding the corresponding state). This ability allows the system to make intelligent choices for more robust performance. TCP streams semantically allow nearly arbitrary permutations of sequence hole creation (illustrated in Figure 4 below). In particular, all of the following possible scenarios might in principle occur in a TCP stream: very long-lived holes; holes that accumulate large amounts of buffer; large numbers of simultaneous holes in a connection; presence of simultaneous holes in both directions of a single connection; and/or a high rate of sequence hole creation. *However*, as our trace analysis shows, most of these cases are highly rare in typical TCP traffic.

On the one hand, we have an objective to preserve

end-to-end TCP semantics as much as possible. On the other hand, we have limited hardware resources in terms of memory and computation. Hence, we adopt the well-known principle of "optimize for the common case, design for the worst case," i.e., the system should be efficient in handling commonly-seen cases of reordering, and should not catastrophically fail when faced with a worst-case scenario, but exhibit graceful degradation. Since the traces highlight that the highly dominant case is that of a single, short-lived hole in just one direction within a connection, we design the system to handle this case efficiently. We then also leverage its capability of dropping packets in order to restrict the occurrence of uncommon cases, saving us from the complexity of having to accommodate these.

With this approach, most of the TCP traffic passes unaltered, while a very small portion experiences a higher packet loss rate than it otherwise would. Note that this latter traffic is likely already suffering from impaired performance due to TCP's congestion-control response in the presence of packet loss, since multiple concurrent holes are generally due to loss rather than reordering. We further note that dropping packets is much more benign than terminating connections that exhibit uncommon behavior, since the connection will still proceed by retransmitting the dropped packet.

The reader might wonder: Why not drop packets when the first hole is created? Why design a system that bothers buffering data at all? The simple answer: the occurrence of a single connection hole is very common, much more so than multiple holes of any form, and we would like to avoid the performance degradation of a packet drop in this case.

4.1 Maintaining Connection Records

Our system needs to maintain TCP connection records for thousands of simultaneous connections, and must access these at high speeds. For such a high-speed and high-density storage, commodity synchronous DRAM (SDRAM) chip is the only appropriate choice. Today, Dual Data Rate SDRAM modules operating at 166 MHz and with a capacity of 512 MB are available commercially [15]. With a 64-bit wide data bus, such an SDRAM module offers a raw data throughput of $64 \times 2 \times 166 \times 10^6 \approx 21$ Gbps. However, due to high access latency, the actual throughput realized in practice is generally much less. Nevertheless, we can design memory controllers to exploit bank-level parallelism in order to hide the access latency and achieve good performance [19].

When dimensioning connection records, we want to try to fit them into multiples of four SDRAM words, since modern SDRAMs are well suited for burst access with such multiples. With this practical consideration, we de-

sign the following connection record. First, in the absence of any sequence hole in the stream, the minimum information we need in the connection record is:

- CA, SA: client / server address (4 bytes + 4 bytes)
- CP, SP: client / server port (2 bytes + 2 bytes)
- Cseq: client's expected sequence number (4 bytes)
- Sseq: server's expected sequence number (4 bytes)
- Next: pointer to the next connection record for resolving hash collisions (23 bits)
- Est: whether the connection has been established, i.e., we've seen both the initial SYN and a SYN-ACK (1 bit). This bit also helps us in identifying SYN floods.

Here, we allocate 23 bits to store the pointer to the next connection record, assuming that the total number of records does not exceed 8M. When a single sequence hole is present in a connection, we need to maintain the following extra information:

- CSH: Client hole or server hole (1 bit)
- HS: hole size (2 bytes)
- BS: buffer size (2 bytes)
- Bh, Bt: pointer to buffer head / tail (2 bytes + 2 bytes)
- PC: IP Packet count in the buffer (7 bits)

The flag CSH indicates whether the hole corresponds to the client-to-server stream or the server-to-client stream. Hole size tells us how many bytes are missing, starting from the expected sequence number of the client or the server. Buffer size tells how many bytes we have buffered up, starting from the end of the hole. Here we assume that both the hole size and the buffer size do not exceed 64 KB. We drop packets that would cause these thresholds to be exceeded, a tolerable performance degradation as such packets are extremely rare. Finally, Bh and Bt are the pointers to the head and tail of the associated buffer. We access the buffer at a coarse granularity of a "page" instead of byte. Hence, the pointers Bh and Bt point to pages. With two bytes allocated to Bh and Bt, the number of pages in the buffer must not exceed 64K. We can compactly arrange the fields mentioned above in four 8-byte SDRAM words.

We keep all connection records in a hash table for efficient access. Upon receiving a TCP packet, we compute a hash value over its 4-tuple (source address, source port, destination address, destination port). Note that the hash value needs to be independent of the permutation of source and destination pairs. Using this hash value as the address in the hash table, we locate the corresponding connection. We resolve hash collisions by chaining the colliding records in a linked list. A question arises here regarding possibly having to traverse large hash chains. Recall that by using a 512 MB SDRAM, we have space

to maintain 16M connection records (32 bytes each). However, the number of concurrent connections is *much* smaller (indeed, Table 1 shows that only *T3* exceeded 1M connections *total*, over its entire 3-hour span). Thus, the connection record hash table will be very sparsely populated, greatly reducing the probability of hash collisions.[1] Even with an assumption of 1M concurrent connections, theoretically the memory accesses required for a successful search will be $T = 1+(1M-1)/(2\times16M) \approx 1.03$ [8].

The following pseudo-code summarizes the algorithm for accessing connection records:

```
 1.  P = ReceivePacket()
 2.  h = (P.SA, P.SP, P.DA, P.DP)
 3.  {CPtr, C} = LocateConn(h)
 4.  if (C is valid)
 5.     UpdateConn(CPtr, C, P)
 6.  else if (P.Syn and ! P.Ack) then
 7.     C=CreateConn(h, P.Cseq)
 8.     InsertConn(C, CPtr)
 9.     Forward(P)
10.  else
11.     DropPacket(P)
```

LocateConn() locates the connection entry in the hash table using the header values and returns the {record_pointer, record} pair (C indicates the actual connection record and CPtr indicates a pointer to this record). If a record was not found and if the packet is a SYN packet then we create a record for the connection, otherwise we drop the packet. If we find the record then we update the record fields after processing the packet. We describe UpdateConn() in the next section.

4.2 Reordering Packets

TCP's general tolerance for out-of-order datagram delivery allows for numerous types of sequence hole creation and plugging. Figure 4 illustrates the possibilities. In this figure, a line shows a packet, and the absence of a line indicates a missing packet or a hole. As shown, a stream can have a single hole or multiple simultaneous holes (we consider only one direction). An arriving packet can plug one of the holes completely or partially. When it closes it partially, it can do so from the beginning or from the end or in the middle. In all such cases, the existing hole is narrowed, and in the last case a new hole is also created. Moreover, a packet can also close multiple simultaneous holes and overlap with existing packet sequence numbers.

In order to interpret the packet content consistently, whenever packet data overlaps with already-received data, we must first normalize the packet, as discussed in the Introduction. In the case of packet overlap, a simple normalization approach is to discard the overlapping data from the new packet. Thus, cases (A) to (F), cases (J) to (K), and cases (O) to (R) all require normalization. In cases (F), (K) and (P) (which, actually, were never seen in our trace analysis), the arriving packet provides multiple valid
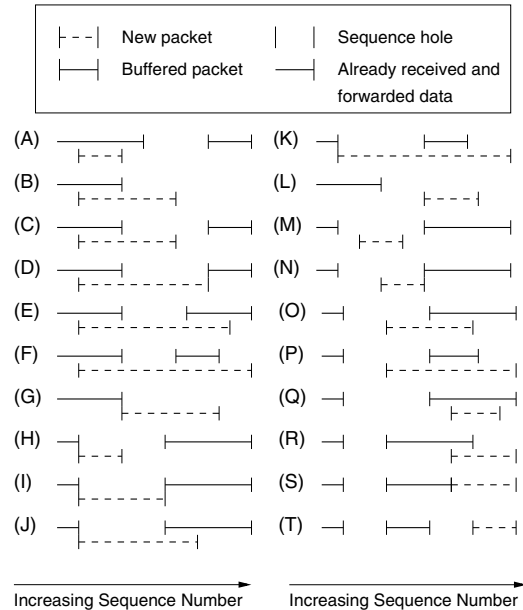

Figure 4: **Various possibilities of sequence hole creation and plugging.**

segments after normalization. In these cases, we retain only the first valid segment that plugs the hole (partially or completely) and discard the second. It is easy to see that once a packet is normalized, the only cases left to deal with are (G,H,I,L,M,N,S,T).

A key question at this point is whether to forward out-of-sequence packets as they arrive, or hold on to them until they become in-sequence. Buffering packets without forwarding them to the end hosts can affect TCP dynamics significantly. For instance, if a packet is lost and a hole is created, then buffering all the packets following the missing packet and not forwarding them will prevent the receiver from sending duplicate-ACKs to the sender, foiling TCP fast retransmission and degrading performance significantly. Hence, for our initial design we choose to always forward packets, whether in-order or out-of-order (delivering in-order packets immediately to the byte-stream analyzer, and retaining copies of out-of-order packets for later delivery). We revisit this design choice below.

When a new packet plugs a sequence hole from the beginning then the packet can be immediately inspected and forwarded. If it closes a hole completely then we can now pass along all the buffered packets associated with the hole for byte-stream analysis, and reclaim the associated memory. (Note that we do not reinject these packets into the network since they were already forwarded to the end host.)

We should note that it is possible for a hole to be created due to a missing packet, however the correspond-

ing packet reaches the destination through another route. In this case, although the missing packet will never arrive at the reassembly system, the acknowledgment for that packet (or for packets following the missing packet) can be seen going in the opposite direction. Hence, if such an acknowledgment is seen, we immediately close the hole. (See [17] for a related discussion on retaining copies of traffic after analyzing them until observing the corresponding acknowledgments.) If the Ack number acknowledges just a portion of the missing data then we narrow the hole rather than close it. In any case, the released packets will remain uninspected since the data stream is not complete enough to soundly analyze it.

We can summarize the discussion above in the following pseudocode. For clarity, we write some of the conditional statements to refer to the corresponding cases in the Figure 4.

```
1. UpdateConn(CPtr, C, P)
2.    if (hole in other direction) then
3.       if (P.Ack > C.Seq) then
4.          if (hole closed completely) then
5.             FreeBuffer(C.Bh-C.Bt)
6.          WriteConn(C, CPtr)
7.    if (Case (A-F, J-K, O-R)) then
8.       Normalize(P)
9.    if (Case (G, H, I, L, N, S) then
10.      Forward(P)
11.      if (Case (G, H, I)) then
12.         WriteConn(C, CPtr)
13.         Analyze(P)
14.         if (Case (I)) then
15.            Analyze(C.Bh-C.Bt)
16.      else if (Case (L, N, S)) then
17.         Buffer(P, C, CPtr)
18.   else if (Case (M, T)) then
19.      DropPacket(P)
```

4.3   Buffering out-of-order packets

Storing variable-length IP packets efficiently requires a memory management algorithm such as paging or a buddy system. Due to its simplicity and low overhead, in our design we use paging. We segment available memory into fixed-length pages (chunks) allocated to incoming packets as required. If a packet is bigger than a page then we store it across multiple pages, with all pages linked in a list. We use a pointer to the first page as the packet pointer. If a packet is smaller than the page size then we use the remaining space in its page to store the next packet, completely or partially.

To track a packet chunk, we need the following (see Figure 5):

- Conn: pointer to the connection associated with the buffer (3 bytes)
- Next: pointer to the next page (3 bytes)
- FrontOrBack (FB): whether the page is filled starting from its beginning or its end (1 bit)
- Offset (Of): pointer to boundary between valid data and unused portion of the page (11 bits)
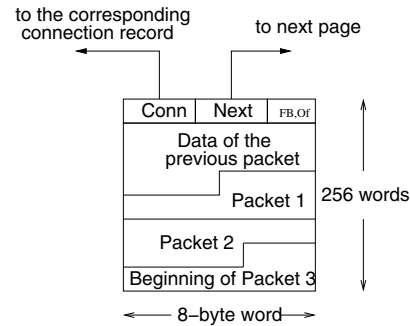

Figure 5: **Page record. Note that packets can be split across two pages. The page shown here holds data for a packet that starts in a previous chunk, and another that ends in a later chunk. A packet starts immediately from the next byte where the previous packet ends. (For convenience, in our design buffered packets include their full TCP/IP headers, although we could compress these.)**

Conn is needed in the case of a premature eviction of a page requiring an update of the corresponding connection record. *Next* points to the next page used in the same hole buffer. *FrontOrBack* allows us to fill pages either from their beginning (just after the header) or their end. We fill in front-to-back order when appending data to extend a hole to later sequence numbers. We fill in back-to-front order when *prepending* data to extend to earlier sequence numbers. If *FrontToBack* is set to FRONT, then *Offset* points to where valid data in the page ends (so we know where to append). If it is set to BACK, then *Offset* points to where valid data begins (so we know where to prepend).

When we free memory pages, we append them to a free-list with head (FreeH) and tail (FreeT) pointers. The pseudocode for our initial design (which we will modify in the next section) is:

```
1. BufferPacket_v1(P, C, CPtr)
2.    WritePage(P, C.Bt)
3.    if (insufficient space in C.Bt) then
4.       if (free page not available) then
5.          EvictPages()
6.       x = AllocatePage(FreeH)
7.       WritePage(P, x)
8.       AppendPage(x, C.Bt)
9.    WriteConn(C, CPtr)
```

In the pseudocode above, we first start writing the packet in the free space of the tail page (line 2), updating the page's *Offset* field in the process. (If *FrontOrBack* is set to BACK for this page, then we know immediately that we cannot append further to it.) If this fills up the page and a portion of the packet remains to be written (line 3), we need to allocate a free page for the remaining portion (a single page suffices since pages are larger than maximum-sized packets). To do that, we first check if we have any free pages left in the memory. If not, then we must evict some occupied pages. (See below for discussion of the
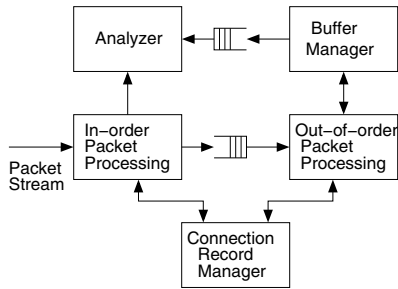
Figure 6: **Block diagram of the system.**

eviction policy.) After possibly freeing up pages and appending them to the free-list, we write the packet into the first free page, FreeH (lines 6–7). When done, we append the page to the list of pages associated with the connection (line 8). Finally, we update the connection record and write it back at the corresponding connection pointer (line 9).

Note that in some cases we need to *prepend* a page to the existing hole buffer instead of appending it (e.g., Case (N) in Fig. 4). In this case, we allocate a new page, set *FrontOrBack* to BACK, copy the packet to the end of the page, and set *Offset* to point to the beginning of the packet rather than the end. The pseudocode for this case is similar to the above, and omitted for brevity.

4.4 Block Diagram

We can map the various primitives of connection and buffer management from the previous sections to the block diagram shown in Figure 6. Module *In-order Packet Processing* (IOPP) handles the processing of in-order packets (cases (G,H,I) in Figure 4). It contains the primitives CreateConn(), InsertConn(), LocateConn(), ReadConn(), WriteConn() and Normalize(). It passes all in-order packets to the *Analyzer* module, which is the interface to the byte-stream analyzer (which can be a separate software entity or an integrated hardware entity). When a hole is closed completely, the buffer pointers (C.Bh and C.Bt) are passed to the Analyzer, which reads the packets from *Buffer Manager*. For out-of-order packets (cases (L,N,S)), the packet is passed to the *Out-of-order packet processing* (OOPP) module. This module maintains the primitives WritePage(), EvictPage(), AppendPage() and WriteConn(). When the packet needs buffering, the corresponding connection record can be updated only after allocating the buffer. Hence, these delayed connection updates are handled by WriteConn() of OOPP. The *Connection Record Manager* arbitrates requests from both IOPP and OOPP.

To accommodate delays in buffering a packet, we use a small queue between IOPP and OOPP, as shown in Fig-

ure 6. Note that the occupancy of this queue depends on how fast out-of-order packets can arrive, and at what speed we can process them (in particular, buffer them). Since we have two independent memory modules for connection record management and packet buffering, we can perform these processes in a pipelined fashion. The speedup gained due to this parallelism ensures that the OOPP is fast enough to keep the occupancy of the small queue quite low. In particular, recall that the raw bandwidth of a DDR SDRAM used for buffering packets can be as high as 21 Gbps. Even after accounting for access latency and other inefficiencies, if we assume a throughput of a couple of Gbps for OOPP, then an adversary must send out-of-order packets at multi-Gbps rates to overflow the queue. This will cause more collateral damage simply by completely clogging the link than by lost out-of-order packets.

5 Dealing with an Adversary

We now turn to analyzing possible vulnerabilities in our stream reassembly mechanism in terms of ways by which an adversary can launch an attack on it and how we can avoid or at least minimize the resulting damage.

In a memory subsystem, the obvious resources that an attacker can exploit are the memory space and the memory bandwidth. For our design, we have two independent memory modules, one for maintaining connection state records and the other for buffering out-of-order packets. Furthermore, attacking any of the design's components, or a combination, can affect the performance of other components (e.g., bytes-stream analyzer).

For our system, the buffer memory space is particularly vulnerable, since it presents a ready target for overflow by an attacker, which can then potentially lead to abrupt termination of connections. We refer to such disruption as *collateral damage*.

We note that an attacker can also target the connection record memory space. However, given 512 MB of memory one can potentially maintain 16M 32-byte connection records. If we give priority to evicting non-established connections first, then an attacker will have great difficulty in overflowing this memory to useful effect. We can efficiently perform such eviction by choosing a connection record to reuse at random, and if it is marked as established then scanning linearly ahead until finding one that is not. Under flooding conditions the table will be heavily populated with non-established connections, so we will not spend much time finding one (assuming we have been careful with randomizing our hash function [9]).

Attacks on memory bandwidth can slow down the device's operation, but will not necessarily lead to connection evictions. To stress the memory bandwidth, an at-

tacker can attempt to identify a worst-case memory access pattern and cause the system to repeatedly execute it. The degree to which this attack is effective will depend on the details of the specific hardware design. However, from an engineering perspective we can compute the resultant throughput in this case and deem it the overall, guaranteed-supported throughput.

Finally, as we show in the next section, if we do not implement appropriate mechanisms, then an attacker can easily overflow the hole buffer and cause collateral damage. Hence, we focus on this particular attack and refine our system's design to minimize the resulting collateral damage.

5.1 Attacks on available buffer memory

While our design limits each connection to a single hole, it does not limit the amount of buffer a single connection can consume for its one hole. A single connection created by an adversary can consume the entire packet buffer space by accumulating an arbitrary number of packets beyond its one hole. However, we can contain such connections by limiting per-connection buffer usage to a predetermined threshold, where we determine the threshold based on trace analysis. As shown in Figure 3, 100 KB of buffer suffices for virtually all connections.

Unfortunately, an adversary can then overflow the buffer by creating multiple connections with holes while keeping the buffer of each within the threshold. A simple way to do this is by creating connections from the same host (or to the same host) but using different source or destination ports. However, in our trace analysis we observed very few instances of a single host having multiple holes concurrently on two different connections. Here, it is important to observe that the adversary is essentially an *external client* trying to bypass our system into a protected network. While we see several instances of a single client (e.g. web or email proxy) *inside* the protected network creating concurrent connections with holes, these can be safely discounted since these hosts do not exhibit any adverse behavior unless they are compromised. From our traces, it is very rare for a (legitimate) external client to create concurrent connections with holes (the last row of Table 1). This observation holds true even for *T3*'s congested link.

We exploit this observation to devise a simple policy of allowing just one connection with a hole per external client. With this policy, we force the attacker to create their different connections using different hosts.

To realize this policy, we need an additional table to track which external clients already have an unplugged hole. When we decide to buffer a packet, we first check to see if the source is an internal host by comparing it against a white-list of known internal hosts (or prefixes).

If it is an external client and already has an entry in the table then we simply drop the packet and disallow it to create another connection with hole.

Our modified design forces an adversary to use multiple attacking hosts (assuming they cannot spoof both sides of a TCP connection establishment handshake). Let us now analyze this more difficult case, for which it is problematic to isolate the adversary since their connections will adhere to the predefined limits and appear benign, in which case (given enough zombie clients) the attacker can exhaust the hole buffer. Then, when we require a new buffer for a hole, we must evict an existing buffer (dropping the new packet would result in collateral damage if it belongs to a legitimate connection).

If we use a deterministic policy to evict the buffer, the adversary may be able to take this into account in order to protect its buffers from getting evicted at the time of overflow (the inverse of an adversary willfully causing hash collisions, as discussed in [9]). This leads us to instead consider a randomized eviction policy. In this policy, we chose a buffer page at random to evict. We can intuitively see that if most of the pages are occupied by the adversary, then the chances are high that we evict one of the adversary's pages. This diminishes the effectiveness of the attacker's efforts to exhaust the buffer, as analyzed below.

5.1.1 Eviction and Connection Termination

Eviction raises an important issue: what becomes of the analysis of the connection whose out-of-sequence packets we have evicted? If the evicted packet has already reached the receiver and we have evicted it prior to inspection by the byte-stream analyzer (which will generally be the case), then we have a potential evasion threat: an adversary can send an attack using out-of-order packets, cause the device to flush these without first analyzing them, and then later fill the sequence hole so that the receiver will itself process them.

In order to counter this evasion, it appears we need to terminate such connections to ensure air-tight security. With such a policy, benign connections pay a heavy price for experiencing even a single out-of-order packet when under attack. We observe, however, that if upon buffering an out-of-sequence packet we do *not* also forward the packet at that time to the receiver, then we do not need to terminate the connection upon page eviction. By simply discarding the buffered packet(s) in this case, we force the sender to retransmit them, which will give us another opportunity to reassemble the byte stream and provide it to the intrusion-prevention analyzer. Thus, we degrade the collateral damage from the severe case of abnormal connection termination to the milder case of reduced performance.

Before making this change, however, we need to re-

visit the original rationale behind always forwarding out-of-sequence packets. We made that decision to aid TCP retransmission: by letting out-of-order packets reach the end host, the ensuing duplicate-ACKs will trigger TCP's "fast retransmission." However, a key detail is that triggering fast retransmission requires at least 3 duplicate-ACK packets [3]. Thus, if the receiver receives fewer than three out-of-order packets, it will not trigger fast retransmission in any case. We can then exploit this observation by always buffering—without forwarding—the first two out-of-order packets on given TCP stream. When the third out-of-order packet arrives, we release all three of them, causing the receiver to send three duplicate-ACKs, thereby aiding the fast retransmission.[2] In the pseudocode of UpdateConn (Section 4.2), lines 16 and 17 change as follows:

```
1.   else if (Case (L, N, S)) then
2.       BufferPacket(P, C, Cptr)
3.       if (Case (N, S) and C.PC >= 2) then
4.           if (C.PC == 2) then
                 # Forward the previously
                 # unforwarded packets.
5.               Forward(C.Bh - C.Bt)
6.           Forward(P)
```

With this modification, we ensure that if a connection has fewer than three out-of-order packets in the buffer, then their eviction does not require us to terminate the connection. Our trace analysis indicates that such connections are far-and-away the most common (the 10th row of Table 1). Hence, this policy protects most connections even using random page eviction. Thus, our final procedure is:

```
1.   EvictPages()
2.       x = random(1, G)
3.       p = ReadPage(x)
4.       C = ReadConnection(p.Conn)
5.       Deallocate(C.bh, C.bt)
6.       if (C.PC > 2) then
                 # Must kill since already
                 # forwarded packets.
7.           KillConnection(C)
8.       else
                 # Update to reflect Deallocate.
9.           WriteConn(C)
```

5.1.2 Analysis of randomized eviction

How many attempts must an adversary make in order to evict a benign page? Consider the following parameters. Let M be the total amount of memory available and g the page size. Hence, the total number of pages available is $P = M/g$, assuming that M is a multiple of g. Let M_l denote the amount of memory occupied by legitimate buffers at a given time. Hence, the number of pages of legitimate buffers, P_l, is simply $P_l \approx M_l/g$ (the equality is not exact due to page granularity).

Let T denote the threshold of per-connection buffer in terms of pages, i.e., a connection can consume no more than T pages for buffering out-of-sequence data. Let C denote the number of connections an adversary uses for their attack. Several cases are possible:

Case 1: $C \leq \frac{P-P_l}{T}$. In this case, the adversary does not have enough connections at their disposal. We have sufficient pages available to satisfy the maximum requirements of all of the adversary's connections. Thus, the adversary fills the buffer but still there is enough space to keep all benign buffer pages, and no eviction is needed.

Case 2: $C > \frac{P-P_l}{T}$. In this case, the adversary has more connections at their disposal than we can support, and thus they can drive the system into eviction. On average, the adversary needs to evict P/P_l pages in order to evict a page occupied by legitimate buffers. Let b evictions/second be the *aggregate* rate at which the adversary's connections evict pages.

If we denote the rate of eviction of legitimate pages by e then we have the following expression:

$$e = \frac{P_l}{P}b \qquad (1)$$

We now express the eviction rate, b, as a function of other parameters. Recall that if the number of packets accumulated by a hole is fewer than three, then upon eviction of a page containing these packets, we do not need to terminate the connection. If we have buffered three or more packets following a hole, then evicting any of them will cause the eviction of *all* of them due to our policy of terminating the corresponding connection in this case. Thus the adversary, in order to protect their own connections and avoid having all of their pages reclaimed due to the allocation of a single new page, would like to fit in the first of these two cases, by having fewer than three packets in the buffer. However, the adversary would also like to occupy as much buffer as possible. Assuming pages are big enough to hold a single full-sized packet, then to remain in the first case the best they can do is to send two packets that occupy two pages, leading to the following sub-case:

Case 2a: $C \geq \frac{P-P_l}{2}$. In this case, if adversary evicts one of their own pages, then this will not cause termination of the connection; only a single page is affected, and replaced by another of the adversary's pages: such evictions do not cost the adversary in terms of resources they have locked up.

We now consider r, the rate at which a single adversary connection can send data. The adversary's corresponding aggregate page creation rate is $(r/g)C$, leading to:

$$b \leq \frac{rC}{g}$$

which becomes an equality when the buffer is full. (Recall that b is the page eviction rate.) Thus, when the buffer is full, Eqn. 1 then gives us:

$$e = \frac{P_l rC}{Pg}$$

Which can be expressed as our first *Zombie Equation*:

$$C = \frac{P}{P_l} \frac{eg}{r} \qquad (2)$$

As this expression shows, the adversary needs a large number of connections if the proportion of "spare" pages is large.

On the other hand, if the adversary uses a lesser number of connections, then we have the following case:

Case 2b: $C < \frac{P-P_l}{2}$. In this case, to cause evictions the adversary's connections must all consume three or more pages. However, as discussed above, if one of the connection's pages is evicted then the connection will be terminated and *all* of its pages evicted. Thus, eviction immediately frees up a set of pages, and adding more pages is not going to cause eviction until the buffer is full again. Providing that in steady-state most evictions are of the adversary's pages (which we can achieve using $P \gg P_l$), in this case they are fighting a losing battle: each new page they claim costs them a multiple of existing pages. Thus, it is to the adversary's detriment to be greedy and create a lot of pages.

We can make this argument more precise, as follows. Suppose every time the adversary evicts one of their own existing connections, it releases P_c pages (the number of pages that each of their connections occupies). For the next $P_c - 1$ page additions, no page needs to be evicted, since the buffer is not yet full. After the buffer is again full, the same process of eviction and repletion repeats. Thus, the rate of attempts to evict a buffer is simply once every P_c page additions.

The time required for P_c additions is $P_c g / rC$ (total size in bytes of the amount of buffer that must be consumed, divided by the aggregate rate at which the adversary can send data). Furthermore, since the number of pages the adversary must consume is $P - P_l$, if each connection consumes P_c pages, then we have $P_c = (P - P_l)/C$. Putting these two together, we have:

$$b = \frac{1}{P_c g / rC} = \frac{rC}{P_c g} = \frac{rC^2}{(P - P_l)g} \qquad (3)$$

Holding the other parameters fixed, this equation says that the rate of eviction varies quadratically with the number of connections available to the adversary. Intuitively, the quadratic factor comes from the fact that by increasing the number of connections, the adversary not only can increase their rate of page addition but also reduce their own page eviction rate, since now each individual connection needs to contribute fewer pages.

Substituting Eqn. 3 for b in Eqn. 1 and assuming $P \gg P_l$, we get:

$$e = \frac{rC^2 P_l}{(P - P_l)gP} \approx \frac{rC^2 P_l}{gP^2} = \frac{rC^2 M_l}{M^2} \qquad (4)$$

This gives us our second *Zombie Equation*:

$$C = M \sqrt{\frac{e}{rM_l}} \qquad (5)$$

Due to our policy of one-hole-per-host, the required connections in Eqns. 2 and 5 must originate from different hosts. Hence, the value of C essentially tells us how many total hosts ("Zombies") an adversary must command in order to launch this attack.

Finally, it is important to note that e is just the rate of eviction of benign buffer pages. As can been seen from Table 1, 85% or more of the time an evicted page hosts an un-forwarded packet (< 3 packets in the buffer), and hence its eviction causes only minor collateral damage (degraded TCP performance due to retransmission). If we denote the benign connection *termination* rate by E, then:

$$E \le (1 - 0.85)e = 0.15e \qquad (6)$$

expresses the rate of serious collateral damage.

We now evaluate these equations with parameters reflecting current technologies. We assume the availability of 128 MB and 512 MB DDR-SDRAM modules for buffering packets. Figure 3 shows that the maximum amount of buffer accumulated by a hole was observed to be around 100 KB. However, it can also be seen that for almost 95% of the cases it was below 25 KB. Therefore, it is reasonable to limit the per connection buffer to a 25 KB threshold, which translates into approximately $T < 13$ pages with a page size of 2 KB. ¿From Table 1, we see that there is a notable difference between the average buffer occupancy of the *Munich* trace compared to other traces. While for other traces, the average buffer requirement of legitimate connections is $M_l \le 30KB$, the same jumps to about 180 KB for the *Munich*. We will consider both these values of M_l. Finally, to get an idea of the degree of damage in a real life scenario, we pick three different zombie data-rates: 56 Kbps for dial-up zombies, 384 Kbps for the zombies with DSL, and 10 Mbps for high-speed zombies. With all these parameters fixed, the rate of collateral damage, E, can be plotted as a function of the number of zombies and their data rate r, as shown in Figure 7.

Each curve has three distinct regions: the first region when the eviction rate of benign connections is zero, corresponding to Case 1 analyzed above; the second region where the eviction rate increases quadratically with the number of zombies, reflecting Case 2b (it's in the adversary's interest to create large holes); and the third region where the eviction rate increases linearly, reflecting Case 2a (the adversary is better off creating small holes).

Note that the Y-axis is log-scaled. The abrupt change in eviction rate from region 2 to region 3 arises due to the assumption that *all* the connections of an adversary occupy
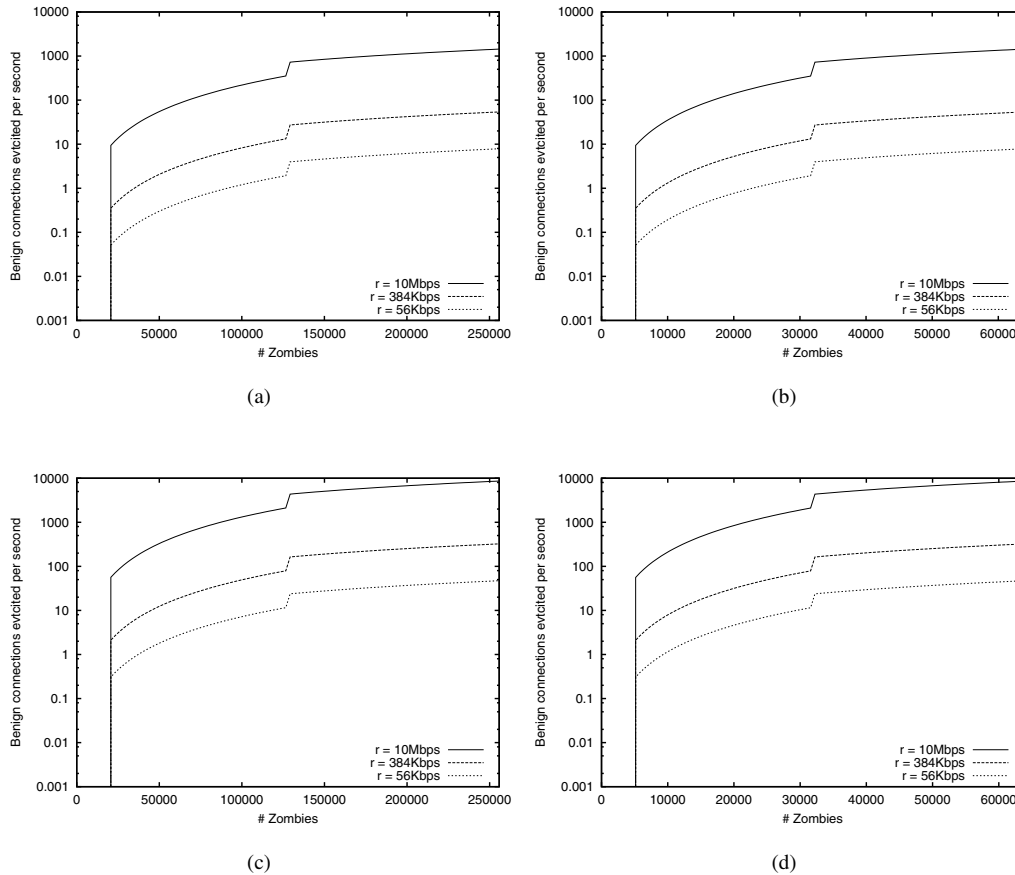
Figure 7: **Benign connection eviction rate as a function of different memory sizes and zombie data rates: dialup zombies (56 Kbps), DSL zombies (384 Kbps) and high-speed zombies (10 Mbps). (a) Total available memory, M=512 MB and average legitimate buffer occupancy M_l=30 KB (b) M=128 MB, M_l=30 KB (c) M=512 KB, M_l=180 KB (d) M=128 MB, M_l=180 KB. For all cases, we assume a 2 KB page size and a per-connection buffer threshold of 25 KB ($T < 13$ pages).**

the same number of pages in the buffer. This assumption results in *each* connection in region 2 having more than two pages and thus all pages are evicted upon eviction of any one of these pages; while *each* connection in region 3 has at most two pages, which are thus not evicted in ensemble. In practice, this change will not be so abrupt since each region will have a mix of connections with different page occupancy, the analysis of which requires more sophisticated mathematical tools beyond the scope of this paper.

As the figure shows, in the most favorable case (512 MB of buffer, and average buffer requirement of 30KB) for the adversary to cause collateral damage of more than 100 benign connection evictions per second, they need to control more than 100,000 machines, with each of them sending data at a rate of 10 Mbps. Moreover, for the same configuration, if these zombies number less than 20,000, then no damage occurs, since the buffer is large enough to keep the out-of-sequence packets of each

legitimate connection (provided none exceeds the threshold of 25 KB).

To summarize, our buffer management policy consists of three rules:

- Rule 1: Limit the reordering buffer consumed by each connection to a predefined threshold (which is carefully chosen through a trace-driven analysis).

- Rule 2: Upon overflow, randomly evict a page to assign to new packet.

- Rule 3: Do not evict a connection if it has less than three packets in the buffer.

We pause here and reflect: what would have been the effect of a naive buffer management policy consisting of only Rule 2? What if we just randomly evict a page on overflow ? First, we note that in the absence of Rule 3, the option of *not* evicting a connection upon its page eviction is ruled out, since otherwise the system is evadable.
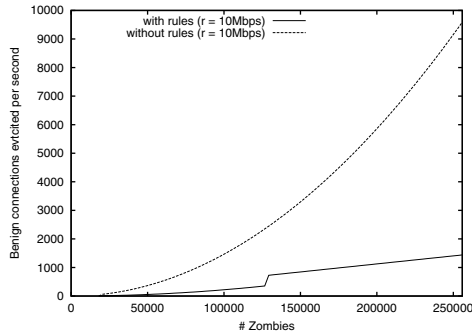
Figure 8: **Comparison of eviction rates for the devised buffer management policy (Rules 1, 2 & 3) and a naive policy of just random eviction (only Rule 2). The total available memory was assumed to be M=512 MB and the average buffer occupancy of benign connections was assumed to be M_l=30 KB. The zombie rate is 10 Mbps.**

Now, given that a buffer eviction is equivalent to connection eviction, we lose out on the improvement given by Eqn 6 (thus, now $E = e$). Secondly, in the absence of Rule 1 and Rule 3, Case 1 and Case 2a do not come into picture; the system behaves in the same way as it would in Case 2b. Hence the benign connection eviction rate is the same as given by Eqn 4.

We contrast the benign connection eviction rate in the two cases for a given configuration in Figure 8. Clearly, with the application of Rule 1 and 3, we reduce the damage to legitimate connections a great deal.

5.2 Designing for the general case

Our approach so far has been to design a system that handles the most common case of packet reordering, i.e., the occurrence of a single hole, at the cost of reduced TCP performance (even when not under attack) for the rare case of multiple concurrent holes. In this section we explore the design space for handling the case of multiple concurrent holes in a single connection.

It is important to note that any mechanism we design will ultimately have resource limitations. Hence, no matter how we configure our system, it is possible to come up with a pathological case (e.g., a very large number of concurrent holes in a connection—a semantically valid case) that will force us to break end-to-end semantics. In this light, our previous trace analysis provides us with a "reasonable" set of cases that merit engineering consideration. In particular, the observation that more than 95% of the connections have just a single concurrent sequence hole indeed presents a compelling case for designing for this case. Unfortunately, when we analyze multi-hole connections in terms of the number of holes in each, no modality appears that is nearly as sharp as the distinction between single-hole and multi-hole connections. Thus, picking a

"reasonable" number of per-connection holes is difficult. (For instance, the trace *T3* shows that a connection can exhibit 85 concurrent holes—though this turns out to reflect pathological behavior, in which a Web server returned an item in a single, large burst of several hundred six-byte chunks, many of which were lost.)

Allowing multiple concurrent holes requires maintaining per-hole state. The key difficulty here is that we cannot afford to use a large, arbitrarily extensible data structure such as a linked list. Once the data structure's size exceeds what we can store in on-chip RAM, an adversary can cause us to consume excessive CPU cycles iteratively traversing it off-chip. On the other hand, if we expand the size of each connection record to accommodate N holes rather than 1, which will allow us to make a small number of off-chip accesses to find the hole, this costs significant additional memory.

Since most connections have zero or one hole, we can realize significant memory savings by differentiating between the two types of connections: connections with at most one hole (per the data structure in our current design) and connections with up to N holes. We could partition memory to support these two different types. Connections initially exist only in the at-most-one-hole partition. As necessary, we would create an additional record in the multiple-holes partition. We would likely keep the original record, too, so we can easily locate the record for newly-received packets by following a pointer in it to the additional record.

A key issue here is sizing the second partition. If it is too large, then it defeats the purpose of saving memory by partitioning. On the other hand, if it is small then it becomes a potential target for attack: an adversary can create a number of connections with multiple holes and flood the second partition. Given this last consideration, we argue that extending the design for handling multiple sequence holes within single connections yields diminishing returns, given the resources it requires and the additional complexity it introduces. This would change, however, if the available memory is much larger than what is needed for the simpler, common-case design.

6 Conclusions

TCP packet reassembly is a fundamental building block for analyzing network traffic at higher semantic levels. However, engineering packet reassembly hardware becomes highly challenging when it must resist attempts by adversaries to subvert it. We have presented a hardware-based reassembly system designed for both efficiency and robust performance in the face of such attacks.

First, through trace-driven analysis we characterized the behavior of out-of-sequence packets seen in benign

TCP traffic. By leveraging the results of this analysis, we designed a system that addresses the most commonly observed packet-reordering case in which connections have at most a single sequence hole in only one direction of the stream.

We then focused on the critical problem of buffer exhaustion. An adversary can create sequence holes to cause the system to continuously buffer out-of-order packets until the buffer memory overflows. We showed that through careful design we can force the adversary to acquire a large number hosts to launch this attack. We then developed a buffer management policy of randomized eviction in the case of overflow and analyzed its efficacy, deriving *Zombie equations* that quantify how many hosts the adversary must control in order to inflict a given level of collateral damage (in terms of forcing the abnormal termination of benign connections) for a given parameterization of the system and bandwidth available to the attacker's hosts.

We also discussed a possible design space for a system that directly handles arbitrary instances of packet resequencing, arguing that due to its complexity, such a system yields diminishing returns for the amount of memory and computational resources we must invest in it.

We draw two broad conclusions from our work. First, it is feasible to design hardware for boosting a broad class of high-level network analysis even in the presence of adversaries attempting to thwart the analysis. Second, to soundly realize such a design it is critical to perform an extensive adversarial analysis. For our design, assessing traces of benign traffic alone would have led us to an appealing, SRAM-based design that leverages the property that in such traffic, holes are small and fleeting. In the presence of an adversary, however, our analysis reveals that we must switch to a DRAM-based design in order to achieve robust high-performance operation.

7   Acknowledgments

Notes

[1] Another consideration here concerns SYN flooding attacks filling up the table with bogus connection entries. We can considerably offset this effect by only instantiating connection entries based on packets seen from the local site.

[2] If alternate schemes for responding to duplicate-ACKs such as Limited Transmit [2] come into use, then this approach requires reconsideration.

References

[1] Internet core router test / packet ordering. *Light Reading, http://www.lightreading.com/document.asp?doc_id=4009& page_number=8*, March 2001.

[2] Mark Allman, Hari Balakrishnan, and Sally Floyd. Enhancing TCP's Loss Recovery Using Limited Transmit. RFC 3042, IETF, January 2001.

[3] Mark Allman, Vern Paxson, and W. Richard Stevens. TCP Congestion Control. RFC 2581, IETF, April 1999.

[4] John Bellardo and Stefan Savage. Measuring packet reordering. In *Proceedings of the second ACM SIGCOMM Workshop on Internet measurement*, pages 97–105. ACM Press, 2002.

[5] Jon C. R. Bennett, Craig Partridge, and Nicholas Shectman. Packet reordering is not pathological network behavior. *IEEE/ACM Trans. Netw.*, 7(6):789–798, 1999.

[6] Ethan Blanton and Mark Allman. On making TCP more robust to packet reordering. *SIGCOMM Comput. Commun. Rev.*, 32(1):20–30, 2002.

[7] Stephan Bohacek, Joo P. Hespanha, Junsoo Lee, Chansook Lim, and Katia Obraczka. TCP-PR: TCP for persistent packet reordering. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*. IEEE Computer Society, 2003.

[8] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. Prentice Hall, 1998.

[9] Scott Crosby and Dan Wallach. Denial of service via algorithmic complexity attacks. In *Proceedings of the 12th USENIX Security Symposium*. USENIX, August 2003.

[10] Jianping Xu et al. A 10Gbps ethernet TCP/IP processor. In *Hot Chips*, August 2003.

[11] Landan Gharai, Colin Perkins, and Tom Lehman. Packet reordering, high speed networks and transport protocol performance. In *Proceedings of IEEE ICCCN 2004*, 2004.

[12] Mark Handley, Christian Kreibich, and Vern Paxson. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *Proceedings of USENIX Security Symposium*, 2001.

[13] Sharad Jaiswal, Gianluca Iannaccone, Christophe Diot, Jim Kurose, and Don Towsley. Measurement and classification of out-of-sequence packets in a Tier-1 IP backbone. In *Proceedings of IEEE Infocom 2003*, 2003.

[14] Michael Laor and Lior Gendel. The effect of packet reordering in a backbone link on application throughput. *IEEE Network*, September 2002.

[15] Micron Inc. Double data rate (DDR) SDRAM MT8VDDT6464HD 512MB data sheet, 2004.

[16] Vern Paxson. End-to-end Internet packet dynamics. In *Proceedings of ACM SIGCOMM*, pages 139–154, Cannes, France, 1997.

[17] Vern Paxson. Bro: A system for detecting network intruders in real time. *Computer Networks*, December 1999.

[18] Thomas Ptacek and Thomas Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical Report, Secure Networks, 1998.

[19] Sarang Dharmapurikar and John Lockwood. Synthesizable design of a multi-module memory controller. Technical Report WUCS-01-26, October 2001.

[20] David Schuehler and John Lockwood. TCP-splitter: A TCP/IP flow monitor in reconfigurable hardware. In *Hot Interconnects-10*, August 2002.

[21] Ming Zhang, Brad Karp, Sally Floyd, and Larry Peterson. RR-TCP: A reordering-robust TCP with DSACK. In *Proceedings of the 11th IEEE International Conference on Network Protocols*. IEEE Computer Society, 2003.