When The CRC and TCP Checksum Disagree

Jonathan Stone
Stanford Distributed Systems Group
jonathan@dsg.stanford.edu

Craig Partridge
BBN Technologies
craig@bbn.com

ABSTRACT

Traces of Internet packets from the past two years show that between 1 packet in 1,100 and 1 packet in 32,000 fails the TCP checksum, even on links where link-level CRCs should catch all but 1 in 4 billion errors. For certain situations, the rate of checksum failures can be even higher: in one hour-long test we observed a checksum failure of 1 packet in 400. We investigate why so many errors are observed, when link-level CRCs should catch nearly all of them.

We have collected nearly 500,000 packets which failed the TCP or UDP or IP checksum. This dataset shows the Internet has a wide variety of error sources which can not be detected by link-level checks. We describe analysis tools that have identified nearly 100 different error patterns. Categorizing packet errors, we can infer likely causes which explain roughly half the observed errors. The causes span the entire spectrum of a network stack, from memory errors to bugs in TCP.

After an analysis we conclude that the checksum will fail to detect errors for roughly 1 in 16 million to 10 billion packets. From our analysis of the cause of errors, we propose simple changes to several protocols which will decrease the rate of undetected error. Even so, the highly non-random distribution of errors strongly suggests some applications should employ application-level checksums or equivalents.

1. INTRODUCTION

In a private talk in January 1998, Vern Paxson[9] stated that in recent Internet packet traces he observed that about 1 datagram in 7,500 passed its link-level CRC but failed the TCP or UDP checksum. Assuming the checksum was correct this result meant that the data was damaged in transit. Furthermore, the damage took place not on the transmission links (where it would be caught by the CRC) but rather must have occurred in one of the intermediate systems (routers and bridges) or the sending or receiving hosts.

We found this phenomenon of interest for two reasons. First, the error rate is disturbingly high. A naive calculation suggests that with a typical TCP segment size of a few hundred bytes, a file transfer of a million bytes (e.g., the size of a modest software down-load) might well have an undetected error. (We hasten to emphasize this calculation *is* naive. As we discuss later in the paper, a more realistic calculation requires an understanding of the types of errors.) Understanding why these errors occur could have a major impact on the reliability of Internet data transfers.

Second, there has been a long-running debate in the networking community about just how valuable the TCP (and UDP) checksum is. While practitioners have long argued on anecdotal evidence and personal experience that the checksum plays a vital role in preserving data integrity, few formal studies have been done. Studying these errors seemed a good chance to improve our understanding of the role of the checksum.

In this paper we report the results of two years of analysis, using traffic traces taken at a variety of points in the Internet. While we do not have a complete set of explanations (about half the errors continue to resist classification or identification) we can explain many of the errors and discuss their impact.

2. PRIOR WORK

In the Internet protocol suite, CRCs and the Internet checksum play a complementary role. CRCs are used to detect link-level transmission errors. The Internet checksum, used by most of the Internet protocols, is designed to detect higher-level errors.

CRCs are based on polynomial arithmetic, base 2[1]. The most commonly used CRC, CRC-32 [5] is a 32-bit polynomial that will detect all errors that span less than 32 contiguous bits and all 2-bit errors less than 2048 bits apart. For most other types of errors (including at least some systems where the distribution of values is non-uniform[14]), the chance of not detecting an error is 1 in 2^{32} or 1 in 4 billion. Other 32-bit CRCs are also used, with error-detecting characteristics which are broadly similar. Differences between CRCs are beyond the scope of this paper: relative to 16-bit checksums, they are all effectively equivalent.

Checksums are simpler error checks, designed to balance the cost of computation (typically in software) against the chance of successfully detecting an error. Many checksums exist; we discuss only one, the Internet checksum.

The Internet checksum is a 16-bit ones-complement sum of the data[12][13][2]. This sum will catch any burst error of 15 bits or less[11], and all 16-bit burst errors except for those which replace one one's complement zero with another (i.e., 16 adjacent 1 bits replaced by 16 zero bits, or vice-versa). Over uniformly distributed data, it detects other types of errors at a rate proportional to 1 in 2^{16}. Over non-uniformly distributed data its performance may be sharply worse. One study found a detection rate of only 1 in 2^{10}[14].

Very few studies have been done on how CRCs and checksums actually perform in the real world of the Internet. Wang and Crowcroft showed that CRC-32 was effective at catching certain types of data reordering[15]. In a simulation of network file transfers using IP over ATM with real file system data, Stone et al. [14] showed that CRC-32 caught a wide range of packet splices in which ATM cells from different packets were combined, but that the Internet checksum detected cell erasures at a rate of only 1 in 2^{10}. A commercial experiment of sorts was done in the 1980s when SUN Microsystems and other vendors to ship Network File System (NFS) implementations with the UDP checksum disabled. The vendors argued that the protection of the Internet checksum was superfluous given that link level data was protected by CRC-32 and the checksum computation clearly hurt performance. Experience, however, soon led customers to demand that the UDP checksum be enabled.[1]

3. METHODOLOGY

One of the major challenges in this study was simply acquiring enough data to analyze. Unlike many other studies which use only packet headers, we needed to examine full packets to properly analyze checksum errors. Due to privacy concerns, it is extremely difficult to get permission to put a packet capture device on a network. And those sites that capture traffic for their own internal analysis are typically unable or unwilling (again for privacy reasons) to share their raw traces with others. Much of our time has been spent negotiating access to data.

In the end, our data tends to come in two forms. First, on a limited number of networks we were allowed to run our own packet capture software. Because our capture software captures only a small fraction of all packets (and mostly packets in error), we were permitted to run it on a few networks and acquire a considerable amount of data.

Second, a few sites were gracious enough to run our software on their stored full-packet traces, and provide us with the results. These sites gave us access to data, especially for long haul networks, that we might otherwise have been unable to get.

[1]Indeed, one of the authors worked for a computer vendor who found its own NFS data was periodically trashed by power fluctuations on the bus. The experience helped convince the vendor to ship its NFS product with UDP checksums enabled. The other author used systems from another vendor, where a bus adaptor between a 16-bit Ethernet and a 32-bit I/O bus on one machine occasionally rearranged bytes within a 32-bit word.

3.1 The Capture Program

The capture program uses the Berkeley Packet Filter (BPF)[7], via the `libpcap` library, to watch all packets passing on a wire. For each complete packet, the software calculates the checksum and if the checksum fails, the entire packet is saved. We do not attempt to reassemble and check fragmented IP datagrams. Since IP fragments are quite rare, we do not expect this omission to have had a major impact on the results.

If the IP datagram contains a TCP segment, the capture program records what bytes were in the segment (on the assumption that the sequence numbers are correct!) and then looks for valid retransmissions of those bytes. Capturing the retransmissions makes it possible to compare the bad data with the good data and see what type of damage was done to the packet.[2] At present, the software captures only TCP retransmissions, in part due to the challenge of getting our capture program to run at line speed. There are UDP protocols, such as NFS, that reliably retransmit and could be obviously be analyzed in the same fashion. Further, for TCP ACKs with errors but no payload, we do not capture retransmissions. Any subsequent ACKs are likely to update the ACK field and window, leaving us with little to compare. Also, an error in a 20-byte TCP header may have corrupted the TCP port numbers, leaving no way to find the right connection.

3.2 Capture Sites

The data presented in this paper was captured at four sites, described below.

CAMPUS: Next to the router of the computer science building at Stanford University. This data set was taken over one week during a term in 1999.

DORM: On a broadcast-Ethernet (10base2) LAN in a Stanford University residential dormitory. This data set was taken over the course of three months during a term in 1999.

DOE-LAB: At the firewall of Lawrence Berkeley National Laboratory. This data was provided by Vern Paxson and represents four distinct data sets taken at different times on several days in late 1998. This dataset is a trace of inbound traffic only.

WEB-CRAWL: A trace taken at 10Mbit Ethernet hub connected to a web spider, which walked approximately 15 million URLs over the course of two days of tracing in May 2000.

3.3 The Analysis Software

The analysis software is a suite of programs that takes traces from the capture program, and attempts to make sense of the errors. For TCP segments with data, the programs compare the correct and errored packets to determine the source of error. For non-TCP errors, or TCP errors where we did not catch valid copy of the datagram, there is little we can do except count the error and, in some cases, analyze the

[2]We would like to thank Matt Mathis for suggesting capturing retransmissions.

path the errored packet took in the hopes of getting some insight into how it was damaged.

Finding Twin Packets

The core of our analysis is to compare a bad packet, with any retransmissions on the same connection which have the same or overlapping sequence-number range. From these retransmissions, we construct an image of what the bad packet *should* have been. For clarity, we call these packets *twins*: the recorded erroneous packet is the *bad* twin, and the reconstructed packet is the *good* twin.

The simplest case of constructing the good twin is also the most common case: both the good and the bad twin have identical TCP sequence numbers and length. Then the good twin is exactly the retransmission. We can compare not only the TCP payload of the good and bad 'twin' packets, but also the TCP pseudo-header (taking care to note that some pseudo-header fields such as the ACKed sequence number and the PSH/FIN bits can legitimately be changed on a retransmission).

Packets whose sequence-number and length strictly include a bad packet can be compared similarly, after deleting bytes outside the sequence-number space of the bad packet, and adjusting the pseudo-header datagram-length and the checksum field accordingly.

The hardest case, which is also rare, is when no single packet covers the payload of the bad packet, and we have to take good bytes from two or more distinct retransmissions.

Figure 2 shows TCPdump output of one set of twins (with addresses deleted for anonymity).

```
08:27:02.907787 X.X.X.X.22 > Y.Y.Y.Y.38201: P
3286558421:3286558441(20) ack 1212716141 win 25144
    4500 003c d7e4 4000 f506 9029 XXXX XXXX
    YYYY YYYY 0016 9539 c3e4 e6d5 4848 946d
    5018 6238 9e26 0000 0000 000a 7476 b63b
    203f a89e 751f fa39 5e13 f425
```

Figure 1: A Bad Twin ...

```
08:27:02.907787 X.X.X.X.22 > Y.Y.Y.Y.38201:
[tcp sum ok]
3286558421:3286558441(20) ack 3221241833 win 8760
    4500 003c d7e7 4000 f506 9026 XXXX XXXX
    YYYY  YYYY 0016 9539 c3e4 e6d5 0848 d455
    5010 2238 9e06 0000 0000 000a 7476 b63b
    203f a89e 751f fa39 5e13 f425
```

Figure 2: and Matching Good Twin.

The TCP payload of both packets is identical, but five 16-bit words of the TCP headers have differences. For example, the bad twin has 0x4848 in the high-order half of the ACK field instead of 0x0848, the low half is 0x946d not 0x5010 (implying an reception rate impossible on Fast Ethernet), and the TCP PSH bit changed. After acquiring the data sets, our analysis task was to find explanations for several tens of thousands of 'twins' such as these.

Analyzing the Twins

Our analysis tools run in two separate passes. First we preprocess each output libpcap file from the capture program. This pass writes each bad packet, plus any captured retransmissions, into a separate libpcap file. At this stage we separate out the UDP traffic, IP header errors, and any singletons – bad packets that have no retransmission, and so no good twin – which we cannot analyze further. Our analysis tools catch essentially all packets (e.g., 32 drops out of 200 million packets). Singletons are thus likely due to errors that corrupted the 96-bit TCP connection identifier or which corrupted the sequence number.

Then, over each set of twins, we run a second pass which examines the bit-by-bit differences between the twins (or triplets). This phase tries to construct an explanation for each error. That is, we try to create a sequence of bit substitutions or byte substitutions, insertions, or deletions, which will transform the good packet into its corrupted bad twin.

We start by identifying the starting and ending byte offsets at which the twins differ. As part of this process we can trivially identify errors which cause differences in a single bit, byte or word.

For the remaining packets, we look for longer insertions or deletions. Starting from the first byte which differs, we look for a window of up to 64 bytes for a deletion or insertion that will re-synchronize the bad packet with its good twin. To reduce computational cost, we use a heuristic of only accepting insertions which re-synchronize the two packets for more bytes than the purported deleted or inserted length. For insertion errors, we also check to see if the inserted data occurred in the packet in a window of 64 bytes preceding the insertion. If it does, the error is classified as a *sliding* error, where the inserted data is a duplicate of earlier packet content. (One potential cause of sliding errors is an bug in DMA hardware, which causes a DMA engine's read pointer to suddenly jump backward.)

The next phase in analysis is to look for multiple occurrences of errors of the same type. The first case is repeated single-byte deletions. Serial UARTs which require an interrupt for every byte, like the NS16450, are still commonplace, and we expected to see single-byte deletions due to overruns on SLIP links. We also expect such overruns to cluster. We also check for multiple longer (64-byte) insertions, deletions, or sliding errors. Our rationale is that where DMA errors strike once, they may well strike again.

Our analysis tools also use several 'pretty-printers' which summarize differences in TCP payloads. For the privacy reasons in section 3, the pretty-printers do not show actual packet contents. Instead, they show byte-by-byte Hamming distances, or histograms of Hamming distances, or byte-by-byte xor's of the data. We can also print these out visually, one byte every 64 characters, to check for periodic differences. Figure 3 shows pretty-printer output on a pair of 'twins' chosen at random. The first line is a TCPdump-like header. Matching bytes are shown as spaces; bad bytes are shown by their Hamming distance, or `0` or `F` for replacement by all-zero or all-one bytes. Ten bursts errors, each a multiple of 32 bytes long, each starting some 24 bytes from a

```
171.64.77.1.XXX(www) > 199.108.89.13(28716) len 1460
+----------------------------------------------------------------+
|                                                                |
|                                                                |
|                                3455235544734454423434545o762426     |
|                                                                |
|                                4255443145374355174674344142553655444|
|35642355348416643543o654546                                 25434|
|75452557534343 4oo4336556345                                     |
|                                                           64334|
|3447426446453425435562333555454433255445646343444454325353456442|
|44333655135544o444485526543                                 43244|
|2363332335424423334472623 4o656242466444576125463336434435335464 4|
|35343535644545652254323434 5                                54355|
|2o355455434252332475354534446353652252544434 43334142443645     |
|                                635434346436644625341645454442333 2145|
|5456762175736455433544444343434414352254356521465265436434 5     |
|                                23346143541442564446544 4fo644553o4552|
|43454365fo37634463355464323634561522542342447325254434345644456 4|
|2544534446255233423346232 36                                     |
|                                                                |
|                                                                |
|                                5434445233155554635444574122565537514|
|455634354667324444365334444                                 64454|
|63453265565345245325323514465351116743464435254755 34|-----------+
+-------------------------------+-------------------+
0                                                              63
```

Figure 3: Pretty-printer output: per-byte Hamming distances

64-byte boundary, are easy to see. Figure 4 shows a second example, where for 128 bytes, every 4th byte is bad, except for the 52nd and and 104th. Tedious manual examination of thousands of pretty-printer packet dumps uncovered a number of error patterns, many of which we had not expected. As we have discovered new patterns, we have added code to the classifier to recognize them.

```
207.24.0.129(www) > 171.64.71.YY(12669) len 1460
+----------------------------------------------------------------+
|                                                                |
|                                                                |
|                                                                |
|                                                                |
|                                                                |
|                                                                |
|                                                                |
|                                                                |
|                                                                |
|                                                                |
|                                                                |
|                                                                |
|                                1   3   3   2   2   3   3   2   |
|2   3   3   2       2   1   2   3   3   2   2   1   2   3   3   |
|1   2   2   2       1   3   3                                   |
|                                                                |
|                                                                |
|                                                                |
|                                                                |
|                                                                |
|                                                                |
|                                                                |
|                                                    |-----------+
+-------------------------------+-------------------+
0                                                              63
```

Figure 4: pretty-printer output with every 4th byte bad

Finally, for packets which are otherwise unclassified, we bin packets into buckets, based on where the first bad byte occurs (toward the head, in the middle, or at the tail) and on what fraction of the packet after the first byte is bad (less than 5%, more than 5%, or somewhere in between).

In addition, for each error category, we sort the output of the analysis pass by source IP address, then look for clustering of errors at specific IP addresses. When we know the topology of local sources, we can also look for clustering of error classes on specific subnets within the local or campus LAN.

4. ANALYSIS

We start our analysis by presenting some basic information about each site and the data collections taken. We then look at various questions about error rates and the form errors take.

The main thrust of our analysis is an attempt to determine what caused each type of error. We have successfully identified the source of about half the errors. The discussion of these errors takes up the majority of this section. Last, we discuss the likelihood of undetected errors in data traffic.

Trace Name	Total Pkts	Errors	Protocol		
			IP	UDP	TCP
CAMPUS	1079M	33851	0	8878	24973
DOE-LAB	600M	37295	0	173	37122
DORM	94M	11578	1278	613	9687
WEB-CRAWL	436M	396832	0	0	396832
Total		479556	1278	9664	468614

Table 1: Trace Sites and Basic Statistics.

4.1 Basic Statistics for The Data Collections

Table 1 gives a summary of the overall error rate at each of the three sites listed in section.3.2. The figures for the DoE trace include all four trace files and the total packet count for the DoE trace is extrapolated from the packet counts for two of the four traces.

Three important (and striking) observations are visible in this table.

The first observation is wide variation in error rates. The WEB-CRAWL data was heavily affected by the ACK-of-FIN bug (see section 4.3.3), and suffered the highest error rate: 1 in 1,100 packets with an hourly peak (not shown) of 1 in 400 packets. The DOE-LAB mean rate (the value varies among the DoE traces) is 1 in about 15,000 packets The lowest rate was in the CAMPUS trace: about 1 in about 31,900 packets.

The second observation is the large number of IP header errors in the DORM trace. The DORM trace is the only one collected adjacent to a variety of end-hosts. All our other traces were captured at points one or more hops removed from most hosts, so the first-hop routers would have already dropped packets with malformed IP headers.

The third observation is the high UDP error count in the CAMPUS trace, which is addressed in section 4.3.3.

It is important to keep in mind that we were only able to compare good and bad twins for TCP data segments. The information on TCP segments seen and TCP data segments see is summarized in Table 2.

4.2 General Observations

As the next section illustrates, we found that just about any factor that could cause a bad checksum periodically does. We found buggy software, and defective hardware, and problems in both end-systems and routers. Nonetheless, there are also some general observations that we can make about how bad checksums occur.

Trace Name	TCP Errors		
	Total	No Data	Data
CAMPUS	24793	11482	13491
DoE-LAB	37122	24286	12836
WEB-CRAWL	396832	391950	4882
DORM	9687	2876	6811

Table 2: Preliminary breakdown of TCP statistics

First, there's an interesting observation about TCP dynamics and the data in Table 2. Typically, a TCP sends one ACK for every two data segments. Furthermore, the typical non-data segment is 40 bytes long while the average data segment is between 128 and 256 bytes long. So if error rates are related to the amount of data, we would expect to see non-data segments to represent at most about 15% of the errored segments. Instead we see non-data segments representing between 30% and 65% of errors (not including the WEB-CRAWL data, which is skewed by the ACK-of-FIN bug discussed in section 4.3.3). The suggestion is that there is a large set of errors that are not related to length but rather per packet or perhaps, per header.

Second, the errors cluster. Certain hosts (or their associated path through the network) in each trace represent a large fraction of the errors that we saw.

Finally, for the WEB-CRAWL data set, we were able to get HTML headers from most hosts. These headers indicate the type of server software the hosts were running and allows us to guess with some accuracy at the TCP implementation on each server. An analysis of this data suggests that, after the ACK-of-FIN bug errors are eliminated, approximately 32% of the errors came from systems running a version of UNIX and 68% came from systems running Microsoft Windows or NT. Those numbers are similar to the fractions of web servers generally reported to run UNIX and Microsoft operating systems. This similarity suggests that, excluding the dominant ACK-of-FIN software bug, a large fraction of the observed checksum errors are occurring independently of the choice of end-system software (e.g., are caused either by end-host hardware or by equipment in the network path).

4.3 Sources of Errors

Our tools currently recognize 40 distinct error classes in TCP segments. The error classes found in the traces are described in Table 3. The size of the table is a strong reminder that there is no one source of errors.

With 468,434 TCP individual errors seen (about 78,500 errors excluding the ACK-of-FIN bug) and 40 basic categories of errors, there is not space to discuss all the analysis results. Instead this section focuses on examples of different types of errors, especially where which we believe the underlying causes or remedies are representative of a broader range of categories.

Errors fall into four broad groups: errors in end-host hardware, errors in end-host software, errors in router memory; and errors at the link level or in network-interface hardware. A final category is errors we cannot fully explain.

4.3.1 Confirming the Causes of Errors

The reader will note that we have been unable to verify the source of many of the errors described. This lack is not for lack of trying. We made several attempts to gain access to hardware and software we suspected was buggy.

For the CAMPUS trace, we were permitted to contact administrators of hosts we identified as error sources. Unfortunately, only three machines inside the CS department at Stanford had more than one error apiece. We also contacted the owners of the Macintosh in section 4.3.5.

In the DORM trace, the organization providing the trace data explicitly forbade us to contact the owners of hosts that we located through our traces. The concern was that owners might view our study as an invasion of their privacy.

In most other cases, the administrators of the affected hosts and routers did not respond to repeated emails and telephone calls.

4.3.2 End-host hardware errors

The DORM trace, taken on a stub broadcast 10Mbit Ethernet segment, was the best-placed to observe end-host errors: it is the only trace where our collection point was on the same network segment as the end-hosts it monitored.

The first surprise in the DORM trace was the number of errors in the IP header. Of the 24 hosts on the DORM broadcast segment, two hosts sent packets with bad IP header checksums. One particular host sent three packets where the Ethernet CRC was correct, but the low two-order bytes of the source IP address were replaced with two bytes from another local IP address. (This error could be caused by a driver bug, or a hardware memory error). A second host sent several packets with single-byte deletions.

Overall, five different systems, using network cards from three manufacturers, demonstrated errors that could be hardware errors. The two hosts with bad header checksums and a third host that shifted addresses by 16-bit positions all had OUIs indicating they used Realtek Ethernet interfaces. We caught one error (a 16-bit replacement with 0x0) from a host with a D-Link Ethernet interface, and several deletion errors from a host with an Addtron Ethernet interface.

4.3.3 End-host software errors

While the evidence suggests that most errors (other than ACK-of-FIN) are hardware related, there were plenty of software errors in the traces. This section discusses some of the most interesting ones. (One further class of end-host errors in the CAMPUS trace is discussed below in section 4.3.5.)

ACK-of-FIN

The largest single error found (in terms of bad checksums) is the ACK-of-FIN bug detected in the WEB-CRAWL traces. Over a quarter million possible ACK-of-FIN errors were detected in the WEB-CRAWL trace.

The ACK-of-FIN bug was a subtle (and now fixed) bug in Windows NT. If the NT software was in the TIME-WAIT state and received a retransmission of a TCP FIN, the NT

Category	Description	CAMPUS	DoE-LAB	DORM	CRAWL
Added PSH+ TCPlen bad	Bad twin has PSH set, the good twin does not. TCPlen is also implausible	5	0	0	0
Dropped PSH+ TCPlen bad		4	0	0	0
TCP hdr offset	The TCP header length of the bad twin is invalid (less than 5), or was heuristically found to be incorrect.	10	0	0	0
Other flags+TCPlen bad		27	0	0	0
all bytes bad	No payload byte in the bad twin matches the good twin.	89	478	6	41
entire tail bad	From the first mismatch to the end of the bad twin, no bytes match.	194	204	161	202
most bytes bad	80% or more of the bytes in the bad twin's payload do not match.	1544	3502	95	291
most bad = 0x00	No pattern, other than 80% or more of the bad twin's incorrect payload bytes are 0x00.	599	90	13	78
most bad = 0x00 or 0xFF	No pattern, other than 80% or more of the bad twin's incorrect payload bytes are 0x00 or 0xFF	0	44	0	0
payload ok	Both payload and header (possibly modified to correct TCP header length) match.	310	480	600	141
one region bad	The non-matching bytes in the bad twin are in one contiguous region.	177	211	128	163
pure insertion	A simple insertion.	47	48	248	11
pure deletion	The bad twin has one deletion.	25	28	2	28
insertion tail	The bad twin appears to have an insertion where the good twin ends.	0	3	2	1
deletion tail	The bad twin has had bytes deleted all the way to its end.	3	2	3	0
transposition	a deletion where the deleted bytes were re-inserted later in the packet.	192	185	27	126
every 4th byte bad	A burst error where only every 4th byte is altered. (see Fig. 4).	62	66	11	44
4th-byte nybble bad	as for 4th byte bad, but only the low-order nybble is corrupt.	0	4	0	0
most tail bad	As for tail bad, but allowing up to 20% of the bytes in the burst to match.	627	735	4311	474
singleton	An isolated bad segment with no good twin to compare.	4341	3235	394	1121
sliding error	see sec. 3.3	3	22	5	15
xsum field same	The bad payload matches the good twin. The TCP flags and connection ID match. the checksum field match. The ACK or window field of the bad twin is likely corrupt.	547	202	1	85
Unclassified – catch-all.		1140	1357	690	606
	Always-caught errors				
one byte bad	Good and bad twins differ only in one payload byte.	47	21	4	47
last byte bad	Good and bad twins differ only in the very last byte of an odd-length segment.	91	1132	3	69
one word bad	Payload differences confine to a single aligned 16-bit word.	25	64	18	16
single-bit error	Good and bad twins differ only in one byte of the payload.	2011	844	236	619
tandem single bit	two single-bit errors in adjacent bytes.	451	3	1	0
Added PSH	The bad twin has the PUSH bit set, the good twin does not.	100	8	217	7
Dropped PSH	Bad twin has PSH clear.	74	7	6	120
Reserved bits changed	TCP's reserved header bits changed.	10	0	0	0
Other flags error	TCP flag bits other than SYN, ACK, FIN, PSH, RST, were changed.	138	13	0	107
IPLEN smaller	Payload of the bad twin match the good twin, but the IP datagram length of the bad twin is smaller.	794	474	419	608
twice counted	misaligned-seqno bad packets, compared to two good packets	-196	-626	-790	-138
Total		13491	12836	6811	4882
ACK-only	ACK-only, no data to compare (see Table 2).	11482	24286	2876	391950
Grand Total		24793	37122	9687	396832

Table 3: Description of basic TCP error categories.

software would (due to an incorrectly initialized buffer) send an ACK with a bad checksum. While no data was damaged due to this bug (all data is sent and acknowledged before any FINs are sent), the bug did cause a slight extra TIME-WAIT load on the server, as the retransmitted FINs would extend the time the server spent in TIME-WAIT.

It turns out that the web spider doing the crawl, for reasons not yet determined, frequently lost the initial ACK sent in response to its own FIN[3]. As a result, the spider was routinely triggering the ACK-of-FIN bug, which explains the high number of errors in this trace. However, the ACK-of-FIN bug has been seen in other traces, at much lower frequencies.

Bad LF in CR/LF

The very first error pattern adduced from the data was a group of errors where the client end of an SMTP, NNTP or HTTP transaction is sending an odd-length line to the server. The packet should end in an ASCII CR/LF pair, and the good twin does, but the bad twin substitutes some other value for the last odd byte. The combination of occurring only with odd-length packets, yet with correct retransmissions, convinces us that this is a software error. The DoE-Lab trace contains 1123 instances of this error, from 25 distinct hosts. The Campus trace contains 82 more instances. The Web-Crawl contains 24 instances, of which 21 are from from a single host, which has been idenfied as running Solaris 2.5. The software on all of the other hosts has not been identified.

Bad Hosts

Another surprise in our traces was the large fraction of errors which are due to persistently-misbehaving hosts. One example is the Campus trace, where 6340 packets out of 33,000 or roughly one-fifth of the errors in the entire trace, occurred in two consecutive UDP streams between a single pair of hosts within one two-hour period. One of the two hosts has a WWW server with a banner suggesting the misbehaving flow was a RealAudio session. UDP errors like these are not a unique case. In one experiment (not otherwise presented here) we found a Quake server in Sweden from which 1 in 5 UDP packets sent had invalid checksums. The Dorm trace includes a TCP stream with similar misbehavior. In a one-hour window of that trace, an off-campus news server sent 2630 bad packets to a single client. We were unable to contact the owners of these systems.

4.3.4 Router memory errors

Table 3 shows that single-bit errors are relatively common. For each single-bit error, our analysis tools report the offset of the bit and whether the error set or cleared the bit. If we examine occurrences of bit offsets modulo 8 (to track bit offsets within bytes) and sort the occurrences by source IP addresses, we can find a set of hosts which may share a common bad bit. If we know the topology and the paths to each of the sources, and assuming a single bad bit, we can infer the probable location.

bit pos	7	6	5	4	3	2	1	0
On	4	0	0	6	45	142	8	70
Off	0	12	0	0	0	162	336	31

Table 4: Frequency of single-bit errors in DOE-Trace, by bit position.

For the DoE-LAB trace, we managed to obtain `traceroute` paths between the trace-collection point, to each recorded source of single-bit errors. (Note that the traceroute data is slightly later than the trace, and gives the reverse route to the source of the inbound erroneous packets.) Table 4 shows the frequency of each of the bit offsets. The most frequent error is bit 1 forced off. The only packets with that error came from 5 distinct hosts; all but a single occurrence (335 out of 336) came from just 4 hosts with addresses on 207.115.200.0/24. The traceroute data shows route flapping to these hosts. As illustrated in figure 5, the routes to each host diverge at the fifth hop router and converge again at twelfth hop router.

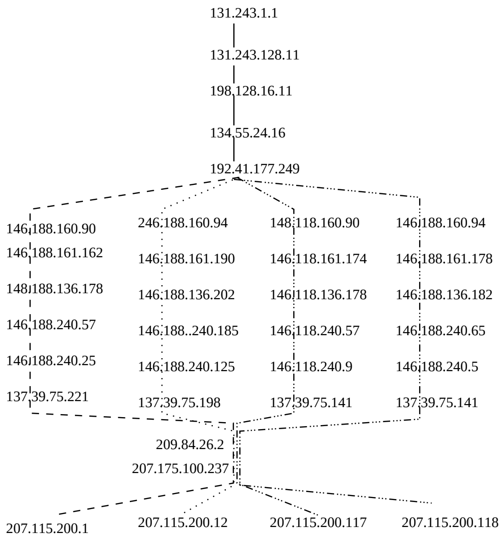

Figure 5: Paths for a Bit Error Pattern.

The fifth hop router in figure 5 regularly showed up in traceroutes for packets with errors, but the single-bit errors only occurred in the packets from net 207. The logical inference, therefore, is that the bit errors are occurring past the fifth hop, and (assuming a single source of errors), after the paths converge again at the twelfth hop. As further confirmation, we also observe that of these 336 errors, all 335 from the four 207.115.120.0/24 hosts are at a byte offset of 0 (mod 4), whereas the sole remaining bit-1-off error occurred at a different byte offset.

The next-most-frequent error, bit 2 off, shows a similar pattern: three hosts, 210.157.0.13 210.157.0.16, 210.157.0.48, are the only sources showing that error. The error seems to be a router between the sixth and last hop in the path.

[3]In several cases, the adjacent monitoring machine had logged the first (valid) ACK sent by an NT server in response to the spider's FIN. We have no explanation for why the spider does not process the first ACK. The Ethernet cards on both the spider and the monitor machine were correctly configured to half-duplex. Other hardware misconfiguration has not been ruled out.

Memory errors in routers may also account for the categories 'every fourth byte bad' (see Figure 4 and a similar pattern where the low-order four bits of each fourth byte are bad. While it has been suggested that these fourth-byte-bad errors are clocking-domain problems, (e.g., between the 4b/5b encoder and the remainder of a FDDI or Fiber-Channel network interface), those problems can likely be excluded because the link-level CRC should detect them. One anonymous reviewer suggested these may be due to DRAM-readout errors. Whatever the precise cause, the every-fourth-byte errors are clearly a pattern that is typical for hardware errors and not software errors.

4.3.5 Link Errors

In general, link errors should be caught by the CRC. However, there are cases where the link level protocols can interact to cause higher level checksum errors. The most notable situation is header compression and we looked vigorously for errors of this sort.

Van Jacobson Header-Compression

We isolated several dozen cases where a TCP connection showed several otherwise unexplained errored datagrams which were consecutive in TCP sequence-number space. In these cases, the datagram length is incorrect and both the checksum-field contents and the TCP payload of a bad packet match the *next* good packet.

One possible cause is decompression of a link using Van Jacobson[4] header-compression (VJ-HC), where the decompresser dropped a packet (perhaps due to a link-level CRC error). When the receiver decompresses the next correctly-received packet of that flow after the drop, the decompresser will reconstitute the packet header from deltas which do not include the sequence-number delta of the dropped packet. (see section 4.1 in [4]). The net effect is that sequence numbers are cut off each packet and pasted on to the succeeding packet. This continues until an end-to-end TCP retransmissions kick in, whereupon the backward jump in sequence-number space causes the VJ-HC sender to send an uncompressed header, which finally re-synchronizes the receiver.

There are two noteworthy points here. Section 4 of RFC 1141[4] strongly recommends that VJ-HC receivers should be used with framing level CRCs to detect errors, and that after a link-level error, decompressors should discard all frames until they see an uncompressed packet. However, our sampling of hosts known to use PPP and VJ-HC, indicates that the deployed base is not discarding packets.

Second, packet drops on links with non-compliant VJ header compression will cause the next few packets – three to five, in our examples – to be decompressed with a sequence number that originally came from the preceding segment. The TCP checksum of the decompressed packet is therefore incorrect, and the receiver will silently discard it.

False header-compression errors

The two-pass nature of our analysis tools made it difficult to find header compression errors: the tools split captured packets into separate files, one for each bad packet. We modified our tools to look for patterns of successive packets with checksums shifted from one packet to the next as an error in header-compression would suggest. We promptly found several possible errors. However, to our surprise, not all were the result of header compression.

The first example we found was a nearby, but off-campus host from the CAMPUS trace. We successfully contacted its owner. The machine in question (a PowerMac 8100) was connected via a dedicated T-1 and had never used header compression. A second, similar Mac in the CAMPUS trace, used as a webserver, also showed a high rate of errors. Our best hypothesis is that we are seeing the effect of a bug in the single-address-space, STREAMS implementation of TCP in Mac OS, which shares buffers between device drivers, TCP, and the user application.[4]

Real header-compression errors

After this surprise, we we approached the problem of finding header compression errors from the other end. We verified our tools against synthetic generated VJ-HC traffic. Then we selected several hosts from one of our datasets which we knew to be connected via CSLIP or PPP links and which were not Macintoshes. When we reran the modified analysis tools over traffic from those hosts, they diagnosed (correctly, we assume) VJ-HC errors. But since our tools cannot distinguish true VJ-HC checksum errors from the MacOS bug mentioned above, we cannot estimate the affect that packet drops on VJ-HC links has on actual checksum failures.

4.3.6 Other Remarks

Space prohibits describing every class. The examples discussed above give the flavour of the full classification. But a few additional points are worth mentioning.

The first is errors in the TCP header. Our methodology relies on TCP sequence numbers to match up erroneous packets with their retransmission. Two factors suggest that reliance on TCP headers numbers is a serious limitation. First, nearly 5% of the observed errors are singletons: bad packets where we saw no matching retransmission. Our capture tools run at nearly wire rate, so a likely cause for singletons is an erroneous TCP sequence number or TCP port.

Second, the analysis tools do some limited check for corrupted sequence numbers or TCP headers. For all 'twins' where good and bad twins have overlapping but non-identical sequence numbers, our tools compare the TCP payload of the good and bad twins, but instead of using the TCP header length and sequence number from the bad twin to compute payload offsets, we use the header field form the good twin for both packets, good and bad. If the modified bad TCP payload is identical to the good payload, our tools assume the header was corrupted, not the payload.

[4]Contacts at Apple have told us that older versions of Mac OS had a subtle, hard-to-find race condition in the driver for the built-in Ethernet on the PowerMac 8100, which is known to have caused data corruption in reception. Perhaps a similar error existed on the transmission side?

4.4 How Many Errors Does TCP Reliably Detect?

One very simple (but informative) question is: How frequent are errors that might get past the TCP checksum? Once we answer how many errors get past the TCP checksum, we can then estimate how frequently bad packets are getting past both the CRC and checksum, and estimate the rate of bit rot due to Internet data transfer.

Looking at this problem analytically, the problem can be stated as follows: Every packet caught by our packet-capture tool was subjected to some set of errors. If the same sequence of errors occurred to a different packet, or if the original packet had different contents, would the TCP checksum still have caught the error?

The TCP checksum will always detect a single error that is up to 15 contiguous bits long, and all 16-bit burst errors except two: substitutions of 0x0000 for 0xFFFF and vice-versa[11]. Longer errors and most multiple errors are caught statistically, where the particular likelihood of detection varies depending the particular characteristics of the data being sent and the types of errors being experienced. So the chance of an undetected error is:

$$P_{ue} = 1 - P_{ef} - P_{ead} - P_{edp}$$

Where ue is undetected errors, ef is error free packets, ead is errors always detected the TCP checksum, and edp is errors detected probabilistically by the checksum. The captured checksum data does not give us any of these probabilities directly, but we have a large enough sample to get fairly good approximations.

First, we observe that the packets our capture tools deemed to be good represent both undetected errors and error free packets, or $P_{ue} + P_{ef}$. However we know that P_{ue} is at best 1/65535 of P_{edp} and may be as poor as 1/1024 of P_{edp}[14]. Given a value for P_{edp}, we can compute the likely value of P_{ue}.

Finding P_{edp}, however, is not easy. The problem is that to fully determine what errors are always caught (and thus can be excluded), we need a very thorough understanding of each error. Here's a contrived but illustrative example.

Consider a hardware error which occasionally overwrites the contents of a 32-bit word to 0x00000000. If this error strikes a word that contains random data, our tools will identify it as a zero-replacement burst error that is caught only probabilistically.[5] But suppose the error instead strikes a word which previously had only one nonzero bit – say 0x000010000. When we compare the resulting bad twin to its good twin, we would infer a single-bit error, and incorrectly claim the error is always caught. One less-contrived example is repeated single-byte errors. If the good bytes are neither 0x000 nor 0xFF, these will always be caught. But the same net result could also be caused by a burst error where, by coincidence, both the original and replacement data were all zeros. Ambiguities like this are an unavoidable part of inductive reasoning.

Given the inherent uncertainty of inferring causes given only symptoms, error, we decided to simply look for burst errors of 16 bits or less. The effect of this decision is probably to somewhat overestimate the error patterns the TCP checksum will catch.

Trace Name	P_{edp}	P_{ue} Range	
		Low	High
DORM	0.0000628404	0.0000000010	0.0000000614
CAMPUS	0.0000090361	0.0000000001	0.0000000088
DoE-LAB	0.0000171166	0.0000000003	0.0000000167
CRAWL	0.0000075436	0.0000000001	0.0000000074

Table 5: Estimated Rates of Undetected Errors

Table 5 lists P_{edp} and the range of values for P_{ue} for the four traces. The value for P_{edp} is only for data bearing segments whose errors are caught probabilistically. Defective ACKs are assumed to always be caught.[6]

The range of values for P_{ue} suggests that between one (data) packet in every 16 million packets and one packet in every 10 billion packets will have an undetected checksum error. The wide range reflects the diversity of traffic patterns, and also the impact that a few bad hosts or routers can have on the error rates see at a particular. The smaller number is, of course, the more worrisome. It suggests that some hosts could regularly be accepting bad TCP data.

5. HOW TO REDUCE THE ERROR RATE?

Regardless of whether the errors are in hardware or software, there are only three sources of error: the sending host, the routers along the path, and the links between them. In general, the CRC will detect the errors on the links and network interfaces will log them, thus making the errors visible. So our problem is with the hosts and routers.

5.1 Don't Trust Hardware

Historically, most software engineers have had faith that the hardware will work correctly and, in some cases, save them from software errors. Probably the strongest message of this study is that the networking hardware is often trashing the packets which are entrusted to it. A tremendous number of the errors in Table 3 are clearly hardware-related errors (e.g., the every-4th-byte-bad and the bad tails).

The DORM trace suggests that many these errors occur before the packet leaves the sending host. In the DORM trace, the strikingly large number of IP header errors must have occurred before or during computation of the link-level CRC at the source host's outgoing interface.

[5]Since there are 2^16 unordered pairs $(x, 1-x)$ which are congruent to 1, and each pair can appear in either order in the original, this specific case will be caught at a rate of 1 in 2^{15}, assuming uniformly-distributed input.

[6]If a damaged ACK somehow becomes the first to ack some data, then either (a) the data was indeed received, in which case the ack has done no harm; or (b) the data was lost, in which case the ack will cause the connection endpoints to become inconsistent and the connection will eventually fail.

In such a situation, the safest thing to do is checksum the data as early as possible in the transmission path: before the data can suffer DMA errors or data path errors in the network interface. In the past, one of the authors has periodically recommended improving transmission performance by doing the checksum as part of the DMA process or in the transmission path in the network interface[3]. Based on this study, we can now say that advice is wrong because it leaves data too exposed to hardware errors.

5.2 Reporting Host Errors

For hosts, one of the problems is that the sending host gets no feedback that it is sending bad packets. Admittedly the host has to retransmit packets somewhat more frequently, but typically not enough to be obvious. What we need is some way for hosts to be informed that they are sending packets with bad checksums. This feedback can then be used by host administrators to cause their hosts to be repaired.

To achieve this feedback, we propose adding a new parameter code, 'transport checksum problem', to the ICMP 'parameter message. Rather than silently dropping packets with invalid TCP or UDP checksums, hosts would send back a 'transport parameter checksum' for each bad packet, along with the header of the offending packet. Hosts which emit thousands of errors over a short period would receive thousands of ICMP messages about their bad checksums. Misbehaving implementations would then be noticed in short order. The impact on hosts with correct network stacks will be negligible: approximately 1 checksum ICMP generated per several thousand packets.

5.3 Reporting Router Errors

Determining which routers have errors, and informing them of it, is harder than informing hosts. The obvious answer, namely using a mechanism such as Router Alert[6, 8] to inform routers along the path that there was an error, doesn't work well for two reasons. First, even on a router that is not generating errors, the rate of reporting is likely to be high: a few thousand notifications per second on a multi-gigabit router in the backbone. Second, the asymmetry of routing paths means that notification must be done by the sender, not the receiver, of the bad packet.

We believe the correct approach is to use monitoring tools at the edge of the network such as those developed by Paxson. The nice feature of these tools is that they are designed to track paths in detail and can therefore can rapidly pinpoint errors. Furthermore, as Paxson[10] has shown, an edge monitoring tool deployed at a fairly small number of sites can effectively test a wide range of paths. One issue of concern, however, is the volume of traffic (tens of thousands of packets) that needs to be sent over a path to effectively test it for errors.

5.4 Protect Truly Valuable Data

In the final analysis, errors are occurring frequently enough that if the consequences of data corruption are large, for instance, for financial data, the application should add a stronger application-level checksum.

Note that many encryption solutions such as IPsec do not provide additional protection. The encryption is applied too late in the transmission process, often after the data has passed through a DMA engine. Rather the application must add the checksum before handing its data to TCP (ala SSL).

6. CONCLUSIONS

It is a well-known irony that the very robustness of fault-tolerant systems can conceal a large number of correctable errors.

In the Internet, that means we are sending large volumes of incorrect data without anyone noticing. Our trace data shows that the TCP and UDP checksums are catching a significant number of persistent errors. In practice, the checksum is being asked to detect an error every few thousand packets. After eliminating those errors that the checksum always catches, the data suggests that, on average, between one packet in 10 billion and one packet in a few millions will have an error that goes undetected. The exact range depends on the type of data transferred and the path being traversed.

While these odds seem large, they do not encourage complacency. In every trace, one or two 'bad apple' hosts or paths are responsible for a huge proportion of the errors. For applications which stumble across one of the 'bad-apple' hosts, the expected time until a corrupted data is accepted could be as low as a few minutes. When compared to undetected error rates for local I/O (e.g., disk drives), these rates are disturbing.

Our conclusion is that vital applications should strongly consider augmenting the TCP checksum with an application sum.

7. ACKNOWLEDGMENTS

Thanks to Vern Paxson for reporting the observed error rate, for access to traces and for comments on our work as it progressed.

Thanks to Stanford Networking Services for kind permission and assistance in running the experiments at Stanford. Thanks to Stuart Cheshire of Apple Computer and Tony Hain of Microsoft for their kind assistance in tracking possible software bugs. Thanks also to Kim (KC) Claffy, Steve Deering, and David Cheriton for their comments on earlier presentations of this work.

8. REFERENCES

[1] Blahut, R. *Theory and Practice of Error Control Codes.* Addison-Wesley, 1994.

[2] Braden, R., Borman, D., and Partridge, C. Computing the Internet Checksum. Intenet Request For Comments RFC 1071, ISI, September 1988. (Updated by RFCs 1141 and 1624).

[3] C. Partridge. *Gigabit Networking.* Addison-Wesley, 1993.

[4] Jacobson, V. Compressing TCP/IP headers for low-speed serial links. Internet RFC 1144, Information Sciencies Institute, Feb 1990.

[5] JOSEPH L. HAMMOND, J., AND ET. AL. Development of a Transmission Error Model and an Error Control Model. Tech. rep., Georgia Institute of Technology, May 1975. Prepared for Rome Air Development Center.

[6] KATZ, D. IP Router Alert Option. Internet RFC 2113, Information Sciences Institute, February 1997.

[7] MCCANNE, S., AND JACOBSON, V. The BSD Packet filter: a new architecture for user level packet capture. In *Proc. USENIX '93 Winter Conference* (January 1993), pp. 259–269.

[8] PARTRIDGE, C., AND JACKSON, A. IPv6 Router Alert Option. Internet RFC 2711, Information Sciences Institute, October 1999.

[9] PAXSON, V. Social Forces and Traffic Behavior. End-to-End Research Group Meeting, Berkely, CA.

[10] PAXSON, V. End-to-end internet packet dynamics. *IEEE Transactions on Networking 7*, 3 (June 1999), 277–292.

[11] PLUMMER, W. W. TCP Checksum Function Design. Internet Engineering Note 45, BBN, 1978. Reprinted in [2].

[12] POSTEL, J. Transmission Control Protocol. Internet Request for Comments RFC 793, ISI, September 1981. 3.

[13] RIJSINGHANI, A. Computation of the internet checksum via incremental update. Internet Request For Comments RFC 1624, Information Sciencies Institute, May 1994.

[14] STONE, J., GREENWALD, M., HUGHES, J., AND PARTRIDGE, C. Performance of checksums and CRCs over real data. *IEEE Trans. on Networks* (October 1998).

[15] WANG, Z., AND CROWCROFT, J. SEAL Detects Cell Misordering. *IEEE Network Magazine 6(4)* (July 1992), 8–19.