Performance of Checksums and CRC's over Real Data

Jonathan Stone, Michael Greenwald, *Member, IEEE,* Craig Partridge, *Senior Member, IEEE,* and James Hughes

***Abstract*—Checksum and cyclic redundancy check (CRC) algorithms have historically been studied under the assumption that the data fed to the algorithms was uniformly distributed. This paper examines the behavior of checksums and CRC's over real data from various UNIX file systems. We show that, when given real data in small to modest pieces (e.g., 48 bytes), all the checksum algorithms have skewed distributions. These results have implications for CRC's and checksums when applied to real data. They also can cause a spectacular failure rate for both the TCP and ones-complement Fletcher checksums when trying to detect certain types of packet splices. When measured over several large file systems, the 16 bit TCP checksum performed about as well as a 10-bit CRC.**

We show that for fragmentation-and-reassembly error models, the checksum contribution of each fragment are, in effect, colored by the fragment's offset in the splice. This coloring explains the performance of Fletcher's sum on nonuniform data, and shows that placing checksum fields in a packet trailer is theoretically no worse than a header checksum field. In practice, TCP trailer sums outperform even Fletcher header sums.

***Index Terms*—Codes, internetworking.**

I. Introduction

THE behavior of checksum and cyclic redundancy check (CRC) algorithms have historically been studied under the assumption that the data fed to the algorithms was uniformly distributed (see, for instance, the work on Fletcher's checksum [2] and the AAL5 CRC [4] and [12]). If one assumes random data drawn from a uniform distribution one can show a number of nice error detection properties for various checksums and CRC's. But in the real world, communications data is rarely random. Much of the data is character data, which has distinct skewing toward certain values (for instance, the character "e" in English). Binary data has similarly nonrandom distribution of values, such as a propensity to contain zeros.

This paper reports on experiments with running various checksums and CRC's over real data from UNIX file systems. We show that the highly nonuniform distribution of values and the strong local correlation in real data causes extremely irregular distributions of checksum and CRC values. In some tests, less than 0.1% of the possible checksum values occurred over 5% of the time. We particularly examine the effects of this phenomenon when applied to the Internet checksum used for IP, TCP, and UDP [1], [9] and compare it to two variations of Fletcher's checksum. We also report on an experiment with placing the standard TCP checksum in a packet trailer. A trailer checksum noticeably increases the checksum's effectiveness, and we prove why this is so.

Manuscript received December 9, 1996; revised May 18, 1998; approved by IEEE/ACM Transactions on Networking Editor J. Crowcroft. This work was supported in part by the Defense Advanced Research Projects Agency under Army Contract DABT63-91-K-0001 and in part by the U.S. Department of Defense.

J. Stone and M. Greenwald are with Stanford University, Stanford, CA 94305 USA (e-mail: jonathan@dsg.stanford.edu; michaelg@dsg.stanford.edu).

C. Partridge is with BBN Technologies (part of GTE Corporation), Cambridge, MA 02318 USA (email: craig@bbn.com).

J. Hughes is with the Storage Technology Corporation, Brooklyn Park, MN 55428 USA (hughes@stortek.com).

Publisher Item Identifier S 1063-6692(98)07368-3.

II. CRC's Versus Checksums

Before examining the behavior of different algorithms, it is worth briefly discussing the CRC and checksum algorithms we used.

CRC's are based on polynomial arithmetic, base 2. CRC-32 [5] is a 32-bit polynomial with several useful error detection properties. It will detect all errors that span less than 32 contiguous bits within a packet and all 2-bit errors less than 2048 bits apart. It will also detect all cases where there are an odd number of errors. For other types of errors, if they occur in data which has uniformly distributed values, the chance of not detecting an error is 1 in 2^{32}.

The concept of a checksum is less well defined. For the purposes of data communication, the goal of a checksum algorithm is to balance the effectiveness at detecting errors against the cost of computing the check values. Furthermore, it is expected that a checksum will work in conjunction with other, stronger, data checks such as a CRC. For example, MAC layers are expected to use a CRC to check that data was not corrupted during transmission on the local media, and checksums are used by higher layers to ensure that data was not corrupted in intermediate routers or by the sending or receiving host.

The fact that checksums are typically the secondary level of protection has often led to suggestions that checksums are superfluous. Hard won experience, however, has shown that checksums are necessary. Software errors (such as buffer mismanagement) and even hardware errors (such as network adapters with poor DMA hardware that sometimes fail to fully DMA data) are surprisingly common and checksums have been very useful in protecting against such errors.

The two most popular checksums are the Internet checksum used for IP, TCP, and UDP [1], [9], and Fletcher's checksum [2]. They represent different balances between performance cost and error detection.

The TCP checksum is a 16-bit ones-complement sum of the data. This sum will catch any burst error of 15 bits or less [8], and all 16-bit burst errors except for those which replace one 1's complement zero with another (i.e., 16 adjacent 1 bits replaced by 16 zero bits, or vice-versa). Over uniformly distributed data, it is expected to detect other types of errors at a rate proportional to 1 in 2^{16}. The checksum also has a major limitation: the sum of a set of 16-bit values is the same, regardless of the order in which the values appear. The checksum was chosen by the Internet community in the late 1970's after experimentation on the ARPANET suggested the checksum was good enough and could be implemented efficiently.

Fletcher's checksum is designed to be a more robust error detecting code. The checksum keeps two sums. One sum A is a running sum of the data in 8-bit chunks. The other sum B is a running sum of each byte multiplied by its position from the end of the packet. This multiplication incorporates positional information into the checksum to protect against movement or transposition of data within the packet. The two 8-bit sums are concatenated to generate a 16-bit checksum. Fletcher also defined a 32-bit version, where 16-bit sums are kept. The algorithm was defined for both ones and twos-complement arithmetic. The version used for the TP4 checksum and in this paper uses 8-bit chunks. When performed in twos-complement, this 16-bit checksum detects all single bit errors, a single error of less than 16 bits in length, and all double bit errors separated by 16 bits or less. Though TP4 uses only the twos-complement version, we investigated both ones- and twos-complement Fletcher sums.

Historically, the TCP checksum and Fletcher's checksum have been viewed as offering a sharp tradeoff between performance and error detection capabilities. The TCP checksum requires one or two additions per machine word of data (assuming the machine word is a multiple of 16 bits long), while Fletcher's sum requires two additions per byte (even if the computation is done in word-sized chunks) [11]. As a result, measurements have typically shown the TCP checksum to be two to four times faster [6], [11]. However, that difference may be declining on newer processors, where the memory access time dominates any computational cost.

III. Work with AAL5

This study began as a study of the error scenarios for packet splices in asynchronous transfer mode (ATM) adaptation layer 5 (AAL5). The AAL5 work helps motivate the rest of the paper and so is explained briefly here.

A. What is a Packet Splice?

AAL5 sends packets as a series of ATM cells, with the last cell specially marked using a bit in the ATM header. A *packet splice* occurs when the right number of cells are dropped such that pieces of two adjacent packets are combined so that they appear to represent one AAL5 packet. Fig. 1 illustrates a splice: two four-cell packets suffer a loss of four cells, such that the first and third cell of the first packet and the first and last cells of the second packet are spliced together to look like a single four-cell packet. It should be noted that ATM does not allow cells to be re-ordered, thus the number of possible splices is limited to those that merely drop, and do not reorder, cells.

Several conditions must be met for a splice to be valid. First, AAL5 stores the length of the packet in the last cell, so the size of the splice must be consistent with the AAL5 length in the last cell. Second, because AAL5 specially marks the last

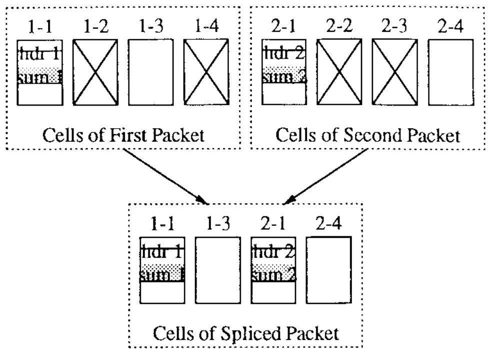

Fig. 1. Example AAL5 splice.

cell of every packet, the last cell of the first packet cannot be part of the splice. Third, the first 40 bytes of the first cell must be a valid TCP/IP header (i.e., have a length consistent with the packet length and certain bits must be set). Unless all three of these requirements are met, the splice will be easily detected without confirming the CRC or checksum.

If the three requirements are met, then the splice has to be detected by either the AAL5 CRC (CRC-32) or the higher layer protocol's checksum (such as the TCP or Fletcher's checksum).

In 1993, an informal study by B. Marshall and C. Kalmanek at AT&T Bell Labs simulated file transfers from a UNIX filesystem (using real data from the filesystem) and examined the performance of the AAL5 CRC. They found a surprising number of cases where the packet splice passed the AAL5 CRC, leading them to wonder if the AAL5 CRC was strong enough. With Marshall's and Kalmanek's assistance, the authors set out to do a more complete set of tests. Those results were reported in an earlier version of this paper, presented at SIGCOMM'95 [7]. Some open questions and surprising results led us to perform a new and more comprehensive series of tests to resolve these issues.

B. Testing Splices

Our test program simulated a file transfer with the file transfer protocol (FTP) of all files on a file system (or selected directories of a file system) via TCP/IP using AAL5 over ATM. All IP and TCP header fields were filled in as if the file transfer were being done over the loopback interface (127.0.0.1). For each packet, the TCP sequence number was incremented by the data length, and the IP ID was incremented by one. The program then examined all possible splices of two adjacent TCP segments and checked to see if either the TCP checksum or AAL5 CRC failed to detect the splice. The program did not concern itself with splices whose data exactly matched a valid packet, nor with those splices that were detected by IP, TCP, or AAL5 header/trailer checks.

The test program was run over file systems at Storage Technologies, the Swedish Institute of Computer Science (SICS), and Stanford University. The TCP segment sizes examined were 256 bytes long, except for runt packets at the end of files. The first row in Tables I–III counts the total number of splices inspected. The next row counts how many invalid splices were detected by simple header checks, and

TABLE I
CRC AND TCP CHECKSUM RESULTS
(256 BYTE PACKETS ON SYSTEMS AT STORTEK)

system	code	% remaining	splices
st05	Total		7186841747
46411 files	Caught by Header		3593444113
4856193 pkts	Identical data		17498067
(98-05-04)	Remaining splices		3575899567
	Missed by CRC	0.0000000000	0
	Missed by TCP	0.0459554853	1643322
st11	Total		6306945748
45627 files	Caught by Header		3152782063
6896637 pkts	Identical data		22324135
(98-05-04)	Remaining splices		3131839550
	Missed by CRC	0.0000000319	1
	Missed by TCP	0.0610412816	1911715
st23	Total		4920441461
29444 files	Caught by Header		2459789331
4372688 pkts	Identical data		50703652
(98-05-04)	Remaining splices		2409948478
	Missed by CRC	0.0000000830	2
	Missed by TCP	0.0568444518	1369922
st25	Total		8748322301
38187 files	Caught by Header		4372322214
9531889 pkts	Identical data		65900443
(98-05-04)	Remaining splices		4310099644
	Missed by CRC	0.0000000464	2
	Missed by TCP	0.1103037608	4754202
st27	Total		5012189213
22319 files	Caught by Header		2505005350
5461908 pkts	Identical data		16574413
(98-05-04)	Remaining splices		2490609450
	Missed by CRC	0.0000000402	1
	Missed by TCP	0.0439271199	1094053
st29	Total		5756622285
57299 files	Caught by Header		2878637775
6314509 pkts	Identical data		19999951
(98-05-04)	Remaining splices		2857984559
	Missed by CRC	0.0000000350	1
	Missed by TCP	0.0552609704	1579350
st49	Total		5696462431
17663 files	Caught by Header		2846361632
6196298 pkts	Identical data		16371605
(98-05-04)	Remaining splices		2833729194
	Missed by CRC	0.0000000000	0
	Missed by TCP	0.0766246826	2171336
st51	Total		4584391161
16864 files	Caught by Header		2290882985
4990431 pkts	Identical data		14136325
(98-05-04)	Remaining splices		2279371851
	Missed by CRC	0.0000000000	0
	Missed by TCP	0.0693654262	1581096
st52	Total		8309068498
58132 files	Caught by Header		4153260212
9082777 pkts	Identical data		40561081
(98-05-04)	Remaining splices		4115247205
	Missed by CRC	0.0000000000	0
	Missed by TCP	0.1726656418	7105618

TABLE II
CRC AND TCP CHECKSUM RESULTS (256 BYTE PACKETS ON SYSTEMS AT SICS)

system	code	% remaining	splices
sics.se	Total		3183838883
`/src1`	Caught by Header		1594737950
48,817 files	Identical data		11000914
3,520,967 pkts	Remaining splices		1578100019
(11-24-97)	CRC	0.0000000000	0
	TCP	0.0411719151	649734
sics.se	Total		2902904306
`/src2`	Caught by Header		1450715240
11,492 files	Identical data		12039586
3,162,423 pkts	Remaining splices		1440149480
(11-24-97)	CRC	0.0000000000	0
	TCP	0.0344980161	496823
sics.se	Total		12074080447
`/src3`	Caught by Header		6031140841
7,845 files	Identical data		12062020
13,097,058 pkts	Remaining splices		6030877586
(12-17-97)	CRC	0.0000000000	0
	TCP	0.0088341538	532777
sics.se	Total		5025946678
`/src4`	Caught by Header		2512845921
33,912 files	Identical data		22171407
5,496,043 pkts	Remaining splices		2490929350
(12-17-97)	CRC	0.0000000000	0
	TCP	0.0198888017	495416
sics.se	Total		21107489268
`/issl`	Caught by Header		10557354562
204,601 files	Identical data		126239615
23,178,376 pkts	Remaining splices		10423895091
(12-17-97)	CRC	0.0000000192	2
	TCP	0.2238580377	23334727
sics.se	Total		6560349785
`/opt`	Caught by Header		3286741967
141,453 files	Identical data		152672075
7,312,235 pkts	Remaining splices		3120935743
(11-24-97)	CRC	0.0000000320	1
0.2% executables	TCP	0.1703438788	5316323
sics.se	Total		8630623470
`/solaris`	Caught by Header		4318348898
98,211 files	Identical data		92736322
9,502,013 pkts	Remaining splices		4219538250
(12-17-97)	CRC	0.0000000474	2
	TCP	0.1068534691	4508723
sics.se	Total		33661656216
`/cna`	Caught by Header		16832727499
248,611 files	Identical data		196026754
36,859,417 pkts	Remaining splices		16632901963
(12-17-97)	CRC	0.0000000180	3
	TCP	0.1866982627	31053339

so did not need to check the checksum. The row labeled "identical data" records how many splices resulted in packets that were identical to one of the original packets, and hence would not result in corrupted data (the checksum, of course, was identical). The "Remaining" packets were all incorrect and depended on the checksum and the CRC to detect the corruption. All percentages listed are computed as percent of "remaining splices." The rows following "remaining" list the splices missed by the CRC test and the TCP checksum test. There were no splices missed by both CRC and the TCP checksum. The data from each site are broken down by file system. The total number of splices is greater than 2^{32}.

We would expect that the CRC of a splice would match the CRC of the original AAL5 packet at a rate of 1 in 2^{32} (or 0.000 000 023 2% of the time). Similarly, we would expect that the TCP checksum would fail to catch bad splices at a rate of 1 in 2^{16} (or 0.001 526% of the time). Observe that for the CRC, the CRC must match the CRC of the second AAL5 packet, while for TCP, the checksum over the entire splice must equal zero.

The tables show that for real data, the CRC failure rate is almost perfectly consistent with the expected failure rate for random data, and is therefore not the subject of much further investigation in this paper.[1] For TCP, however, the story is

[1]The difference between our results and those of Marshall and Kalmanek are the "identical data" entries. Given that the payloads were identical, it is not a failure if the CRC does not detect these splices as no data-corruption occurs. Their tests did not distinguish the cases of splices with identical data from splices with different data but congruent checksums.

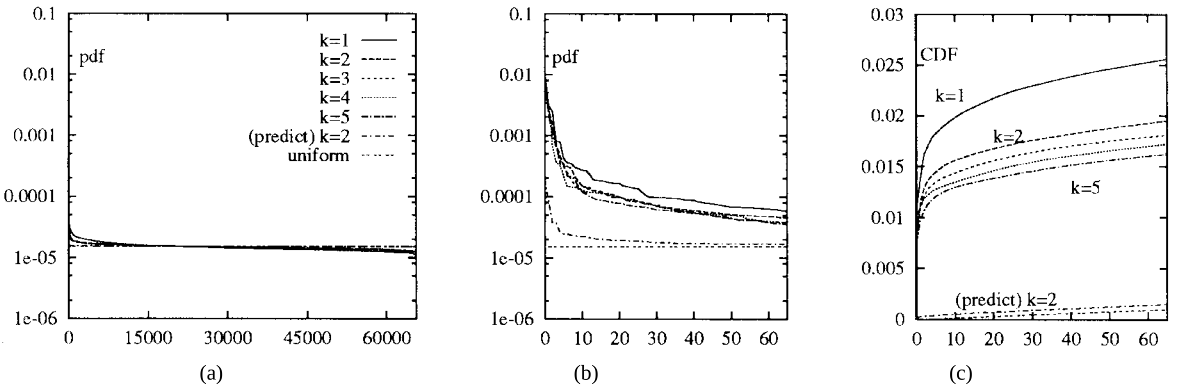

Fig. 2. Distribution of TCP checksum over blocks of k cells in *smeg.dsg.stanford.edu:*`/u1` (a) full probability dist. function; (b) pdf: 65 most common values; and (c) CDF: 65 most common values.

TABLE III
CRC AND TCP CHECKSUM RESULTS
(256 BYTE PACKETS ON SYSTEMS AT STANFORD)

system	code	% remaining	splices
smeg.stanford.edu	Total		8863295657
`/u1`	Caught by Header		4442709123
198,352 files	Identical data		25715994
9,901,213 pkts	Remaining splices		4394870540
(8-20-97)	CRC	0.0000000228	1
	TCP	0.0707199443	3108050
pompano.stanford.edu	Total		1197495954
`/usr/local`	Caught by Header		599005787
11,468 files	Identical data		6024593
1,314,390 pkts	Remaining splices		592465574
(11-26-97)	CRC	0.0000000000	0
	TCP	0.0269563342	159707

different. Between 0.008% and 0.22% of the bad splices passed by the header checks passed the checksum. This is between a factor of 10 and 100 worse than expected, and requires some explanation.

IV. EXPLAINING THE TCP CHECKSUM FAILURES

Why does the TCP checksum fail to detect so many splices? The reasons have to do with the distribution of data values and how data from one packet can be mixed with data from another packet.

A. Failure Scenarios

We can compute the TCP checksum in pieces and then add the pieces to get the complete packet sum. So, we can think of the TCP checksum of a packet broken into ATM cells as being the sum of the individual checksums of each 48-byte cell.

The usual requirement for a splice to pass the TCP checksum is that the checksum of the splice add up to the checksum of the entire first packet contributing to the splice. Because the splice contains cells of the first and second packets, this requirement can also be expressed as a requirement that the checksum of the cells from the first packet not included in the splice must equal the checksum from the cells of the second packet that are included in the splice. If just one cell from the second packet is included in the splice, this requirement reduces to the requirement that the checksum of the cell from the second packet have the same sum as the cell it replaces. In multicell replacements, the sum of the mixes of cells must be equal.

B. Distributions of the TCP Checksum

Given random data, a good checksum or CRC should uniformly scatter the checksum values over the entire checksum space. Obviously a checksum algorithm that does not uniformly distribute checksum values (i.e., has hotspots) will be more likely to have multiple cells with the same checksum. Theorem 6 in Appendix A proves that, over uniformly distributed data, the TCP checksum algorithm gives a uniform distribution of checksum values.[2] Thus, any hotspots in the distribution of checksum values are due to nonuniformity of the data, and are not inherent in the TCP checksum algorithm.

C. The Distribution of Checksum Values over Single Cells

If the distribution of 16-bit words is completely uniform, the chance of an arbitrary sequence of data in the first packet having the same checksum as an equal-sized arbitrary sequence of data in the second packet is $1/M$, where $M = 2^{16}$. However, the distribution of values over real data is not uniform.

Fig. 2 shows three plots summarizing the distribution of checksum values on the filesystem `/u1` on smeg.dsg.stanford.edu. The x-axis represents different checksum values, sorted by frequency to better show the distribution. In the PDF graphs, the y-axis is the probability that the given checksum value occurred. Fig. 2(a) shows the entire PDF, and (b) shows a blowup of the most frequent 65 values (0.1%) The CDF [Fig. 2(c)] shows the same 65 values, but here the y-value for a given x represents the cumulative probability that any of the most common x values occurred. If the distribution were uniform then the PDF should simply be a horizontal line at $1/M$, and the CDF a straight line with slope $1/M$.

This data shows that the TCP checksum on real data has hotspots. In the file system shown in the figure (smeg:`/u1`), the top 0.1% of the checksum values occurred 2.5% of the time. If one examines this distributional data over many filesystems, one discovers two things. First, that the single most common

[2]The actual results are stronger: if just one word in the packet is uniformly distributed over all 2^{16} possible values, then the checksum of the entire packet is uniformly distributed over all possible values.

checksum value (usually zero) occurs between 0.01% and 1% of the time. Second, that for 48 byte cells the 65 next most frequently occurring checksum values (0.1% of the checksum space) account for between 1% and 5% of the checksum values seen.

D. Checksum Distribution over Larger Blocks of Data

Although for uniformly distributed data values the probability distribution of the checksum is uniform independent of the length of the block of data, this is not true for nonuniform data. In that case, the expected probability distribution of the checksum may be computed by

$$P_k[i] = \sum_{j=0}^{M} (P_{k-1}[j] P_1[i-j])$$

where $P_k[i]$ is the probability that the checksum over a block of length k is equal to i, and where $i - j$ is taken mod M. The dotted line in Fig. 2 labeled "Predict $k = 2$" shows the expected distribution of checksums over blocks 2 cells long, given the checksum distribution over one cell given by $k = 1$.

So, if the nonuniformity is uniform—that is, that every cell of data is drawn from the same probability distribution, and that the sum is the sum of *independent* samples—then we would expect the distribution of the sums to conform closely to the dotted line in our graphs. The predicted value for $k = 2$ is already close to uniform for all but the 20 most common values, even though $k = 1$ is decidedly nonuniform. Corollary 3 and Theorem 4 in the Appendix show that, regardless of the original distribution, the distribution should get more uniform as k increases.

However, our measurements show that the nonuniformity extends to larger chunks than single words or cells, and that the checksum of one cell *is* correlated with the checksums of the neighboring cells. The lines labeled $k = 2$, $k = 3$, etc. show the measured distribution of checksums over samples of blocks of length k cells over real data in our file system. The data does get more uniform (seen most clearly in the CDF), but nowhere as quickly as it should if the cells were roughly independent. We believe the samples should be somewhat representative even of noncontiguous blocks. Once again, the checksum values are sorted in decreasing order of probability, to give a clearer picture of the distribution. Note that even over the larger block sizes, although the probability of a match decreases slightly, the distribution is still significantly more nonuniform than expected.

E. Filesystem-Level Nonuniformity is Not the Answer

Given the nonuniform distribution, what, then, is the expected failure rate of the IP/TCP checksum in detecting splices for a given distribution P of checksum values? As discussed above, it is simply the probability that the checksum over the cells missing from the first packet is equal to the checksum over the cells present from the second packet. For a given probability distribution P, and assuming that every cell is drawn from the same distribution, this probability is

$$P(\text{failure}) = \sum_{i=0}^{M} P[i]^2.$$

TABLE IV
PROBABILITY (AS %) OF CHECKSUM MATCH FOR SUBSTITUTIONS OF LENGTH k CELLS

Length	Uniform	Predicted	Measured
		Global	Global
1	0.001526	0.02126770	0.02126770
2	0.001526	0.00153019	0.01494399
3	0.001526	0.00152590	0.01348366
4	0.001526	0.00152590	0.01416288
5	0.001526	0.00152590	0.01108446

Table IV computes this probability using the measurements of the Stanford file system from Fig. 2. It lists the probability that the checksums of two blocks, each k cells long, drawn from anywhere in the same filesystem, will be equal. For each block of length k cells, the "uniform" column shows the expected probability given uniform distribution. The "predicted" column shows the probability predicted assuming each cell is drawn from the identical, nonuniform distribution. (The particular distribution is the one actually measured for single cells over the smeg:/u1 file system.) The last column lists the probability actually measured for each block size over the entire file system.

Table IV shows us that the actual measurements do not match the predictions. There are two issues our initial model ignored.

First, we have measured the probability distribution over the entire file system for chunks of k cells, but we know that distribution of data values is heavily dependent on file type (binary versus character, executable versus GIF, even Shakespeare versus Joyce). Splices come from adjacent packets, which usually come from the same file. Thus, real failure rates could be higher than the averaged global distribution would suggest.

For example, consider an extreme (and extremely hypothetical) case in which a file system consists of half binary and half textual data. Imagine that 90% of the cell-sized chunks of binary data had a checksum of 0 × 0000, and that 90% of the cell-sized chunks of textual data had a checksum of 0 × 1F00. Considered globally, we would find 0 × 0000 45% of the time and 0 × 1F00 45% of the time, so $\sum p^2$ would be approximately 32% and we would predict about 32% of the packet splices would incorrectly pass the checksum. However, in reality, for any given file the local distribution would find the most common checksum 90% of the time, and thus the failure rate would be about 81%. Therefore, the global distribution of checksums (measured across an entire filesystem) is not sufficient to accurately predict checksum failure rate: a more localized distribution of checksums is needed.

Second, if two cells have congruent checksums because the data was identical, then replacement of one cell by the other is not a checksum failure—the packet is unaltered and no corruption will occur. To accurately predict meaningful checksum failures, then, we need to subtract both congruent and equal cells from the probability of a match. In a system with uniformly distributed data the odds of finding two 48 byte cells with identical data is 1 in 2^{384}, which is so unlikely as to be utterly neglible. However, in practice it occurs far more frequently. Our actual measurements show that the

TABLE V
PROBABILITY (AS %) OF CHECKSUM MATCH FOR SUBSTITUTIONS OF LENGTH k CELLS BASED ON LOCAL DATA

Length (k)	Measured Global	Local Congruence	Exclude Identical
1	0.02126770	1.58305972	0.20704272
2	0.01494399	1.30267681	0.17226800
3	0.01348366	1.21236431	0.16614066
4	0.01416288	1.15970577	0.16316988

most common reason for checksum congruence is identical data—identical splices occur 20 to 40 times more frequently than congruent-but-unequal splices. This result is another example of nonuniform distribution of the data, but, in this case, a benign one: the undetected error has not corrupted user data.[3]

F. Localized Nonuniformity of Data

Table V shows how the probability changes when we restrict the comparisons to only look at local data. The first column (identical to the last column in Table IV) displays the probability of taking two blocks of data, each k cells long, from anywhere in the entire file-system, and finding that their IP checksums were congruent to each other. The column labeled "locally congruent" shows the same probability if we limit the search to be within 2 packet lengths (512 bytes). (In order to increase the sample size for the local comparisons, we did not restrict ourselves to contiguous blocks). The final column shows how the probability decreases when we exclude checksum matches for a pair of blocks that contained identical data, as such a substitution would not result in any data corruption. It is still significantly higher than the global rate. (Recall, that if the data were uniformly distributed then every entry in this table should be 0.001 526%). If the checksum failures are purely a result of nonuniform distribution, then these sample probabilities should track the measured TCP checksum failure rates.

Table VI compares this distribution data for several file systems with the actual rate of checksum failures for comparable-length substitutions. Note that there is a minor difference between the way data was collected to compute the predicted values and the measured values. The predicted values are computed for full 48-byte cells. However, the measured values include a large number of cells with only 8 bytes of data. The reason is that the measured data was collected as part of computing possible packet failures for 256-byte packets, and with the 40-byte TCP/IP header, a 256-byte packet has only 8 bytes of data in the first and last cells.

However, even allowing for the deficiencies in measuring actual failures, what Table VI suggests is that local congruence still does not fully explain the checksum failure rate. To fully explain the failure rate, we need a new way to think about checksum data patterns, which is presented below in Section V-D.

V. REDUCING CHECKSUM FAILURES

In this section, we look at various ways to reduce the checksum failure rates.

[3]But the lost packet must still be recovered; see Section V-D.

TABLE VI
CHECKSUM FAILURES ON REAL DATA PROBABILITY (AS %) OF CHECKSUM CONGRUENCE FOR BLOCKS OF LENGTH k CELLS

smeg.dsg.stanford.edu:/u1				
	k=1	k=2	k=3	k=4
Predicted	0.0212677	0.0015302	0.0015259	0.0015259
Measured Global	0.0212677	0.0149440	0.0134837	0.0141629
Local Congruence	1.5830597	1.3026768	1.2123643	1.1597058
Exclude Identical	0.2070427	0.1722680	0.1661407	0.1631699
Actual	0.1026797	0.1581733	0.0907984	0.0568881
sics.se:/opt				
Predicted	1.1422436	0.0150023	0.0016907	0.0015280
Measured Global	1.1422436	0.9493377	0.8852883	0.8291802
Local Congruence	10.7766645	9.6723695	9.3490614	9.0170788
Exclude Identical	0.3872216	0.4732675	0.6897936	0.6086173
Actual	0.1085216	0.5551069	0.2130342	0.1183174
sics.se:/src1				
Predicted	0.0320218	0.0015407	0.0015259	0.0015259
Measured Global	0.0320218	0.0182235	0.0163506	0.0169735
Local Congruence	1.7774385	1.5562402	1.4326226	1.4595770
Exclude Identical	0.2537653	0.1938629	0.1418823	0.2196530
Actual	0.1037989	0.1143002	0.0499086	0.0283392
sics.se:/src2				
Predicted	0.0204720	0.0015312	0.0015259	0.0015259
Measured Global	0.0204720	0.0154869	0.0146764	0.0140808
Local Congruence	2.1467761	1.9016190	1.8495718	1.7812772
Exclude Identical	0.1094778	0.0995097	0.1345251	0.1108649
Actual	0.1747045	0.1339748	0.0429196	0.0210642

TABLE VII
CRC AND TCP CHECKSUM RESULTS, COMPRESSED DATA (256 BYTE PACKETS ON SYSTEMS AT SICS)

system			**splices**
fafner.sics.se	Total		1549869756
compressed	Caught by Header		773945117
`/opt`	Identical data		51902
1,679,166 pkts	Remaining splices		775872737
(5-9-95)	% of remaining splices missed by		
	CRC	0.0000000000	0
	TCP	0.0021002156	16295

A. Regaining a Uniform Distribution: Compression

We claim that the TCP checksum's failure to detect many splices is due to the nonuniform distribution of the data being summed. One obvious way to deal with nonuniform data patterns is to compress the data. As an experiment to verify that our diagnosis was correct, we compressed all the files in the file system at SICS that gave the TCP checksum the most trouble (*/opt* on `fafner.sics.se`) and ran our tests on the compressed files. (The compression was Lempel–Ziv, and was performed using the UNIX compress command.) The results are shown in Table VII, using the same format as Table I. The interesting result is that the number of splices that passed the checksum is approximately 0.0021%, which is close to the expected rate on uniform data of 0.0015%. This result is a hundred times improvement over the 0.17% rate before compression. So compression clearly helps.

B. Alternative Checksums: Fletcher

It is not always possible or desirable to compress the data. Another obvious question to ask is whether, without data compression, another checksum algorithm would perform better than TCP's. An obvious candidate checksum is Fletcher's

TABLE VIII
FLETCHER'S CHECKSUM RESULTS (256 BYTE PACKETS ON SYSTEMS)

System	Missed by	%	splices
sics.se	TCP	0.1703438788	5316323
/opt	F255	0.0044358811	138441
	F256	0.0091286724	284900
smeg.stanford.edu	TCP	0.0707199443	3108050
/u1	F255	0.0862324604	3789805
	F256	0.0046759739	205503
pompano.stanford.edu	TCP	0.0269563342	159707
/usr/local	F255	0.0022121117	13106
	F256	0.0029058228	17216
sics.se	TCP	0.0411719151	649734
/src1	F255	0.0067998225	107308
	F256	0.0054134085	85429
sics.se	TCP	0.0344980161	496823
/src2	F255	0.0023053857	33201
	F256	0.0039193848	56445

checksum [13].[4] With our error model, where cells are dropped but no random data is inserted, we might expect the positional B term to improve error detection.

As with TCP, we can compute and analyze Fletcher's checksum over individual cells rather than entire packets. Recall that the B term of the Fletcher checksum is computed by multiplying each byte by its offset from the end of the packet. We can also compute a local Fletcher checksum over one cell i as A_i, and B_i. To compute the contribution of an individual cell to the total Fletcher sum for the packet, we add A_i to A_{packet} and add R_i to B_{packet}, where $R_i = (B_i + A_i L)$ and L is the offset of the end of the cell from the end of the packet. It should be noted that since all the shifts of data are by a multiple of the cell size (48 bytes), the contribution of the B term for each cell to detect motion is limited to 1 from, at most, $M/GCD(M, 48)$ values (85 and 16 for 1 and 2's complement, respectively). Both 85 and 16 are considerably smaller than M (255 or 256, respectively).

Table VIII shows the actual results for both 1's complement ($\bmod$ 255) and 2's complement ($\bmod$ 256) Fletcher's checksum over several filesystems. The results of the TCP checksum on those filesystems is included for comparison.

We see that Fletcher's, in general, out-performs the TCP checksum, and in some cases comes within a factor of 2 to a 1 in 2^{16} miss rate. This performance is curious given our results so far. First, Corollary 8 in the Appendix shows that, for uniformly distributed data and replacements larger than single words, Fletcher should not be any stronger than IP/TCP. Second, two empirical measures show that both TCP and Fletcher have a similar nonuniform distribution over individual cells. When looking at plots of checksums over 48-byte cells (Fig. 3), the Fletcher's checksum looks to have a nonuniform curve similar to that of TCP. And when we look at the probability of the checksum that two randomly chosen cells[5] in the file system match each other, we find a probability of 0.016% for Fletcher 255, 0.013% for Fletcher 256, and 0.011% for IP/TCP.

[4]Unlike [13], our Fletcher's results perform a sum-to-zero inversion on the transmitted checksum. See Section VI-C.

[5]This calculation includes all cells, including the short cell at the end of each packet, so the number does not match the "Measured Global" for $k = 1$, given earlier.

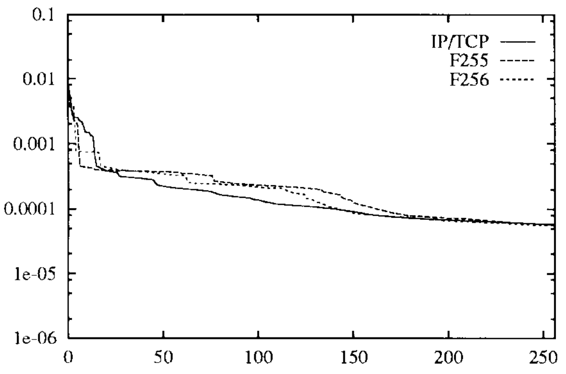

Fig. 3. PDF of TCP checksum, F255, and F256 over 48 byte cells in *smeg.dsg.stanford.edu*:/u1. Most common 256 values.

Why then does Fletcher perform better than the TCP checksum? The most obvious effect is that the positional dependence of Fletcher's checksum effectively increases the number of cells changed in a splice. The vast majority of splices which pass IP and TCP header checks include the header cell from the first packet, and therefore the checksum field from the first packet. Each cell from the first packet not included in the splice moves *all* the subsequent cells from the first packet closer to the start of the splice—thus increasing the L_i's component of their contribution to the B field of the splice's checksum, when compared to their L_i contribution to the first packet's checksum. And even if the inserted cells from the second packet are identical to the dropped cells, their L_i's for the dropped cells is *different* than the L_i's of the inserted cells, as they appear later in the splice than in the first packet. The positionality of Fletcher's checksum means that the effective size of the splice is not just the total number of cells replaced, but includes any intervening, "reshuffled" cells from the first packet which lie between the first drop and the last replacement. (Note that this result has no effect on splices that join a prefix of the first packet to a suffix of the second.)

The cause of the performance difference is subtle. Recall that the condition for checksum failure is that the sum of the 8-bit A_i's be congruent and that the sum of the $(B_i + A_i L_i)$'s also be congruent. The condition on the A's is identical to the condition for IP checksums. Since the data cells are drawn from the same highly localized nonuniform distribution, their 8-bit A_i terms have a fairly good chance of being congruent—at least 256 times more than the standard 16-bit TCP sum. But for the B term, each of the R_i terms for individual cells are multiplied by L_i. This multiplication permutes the entire distribution. Thus, a given highly probable r drawn from one R_i is unlikely to be drawn from R_j, the distribution of R terms for a nearby cell drawn from the same distribution. In effect, the contribution of each cell to the B term of its packet is *colored* by its offset from the end of the packet (think of coloring the cells by their L_i number). This coloring, and the nonuniformity, combine to make undetected splices less likely. It is well known (we provide a proof in Lemma 9) that the probability of drawing two identical values from a nonuniform distribution is always higher than the probability of drawing two values that differ by any fixed amount. (This issue is discussed further in the Appendix).

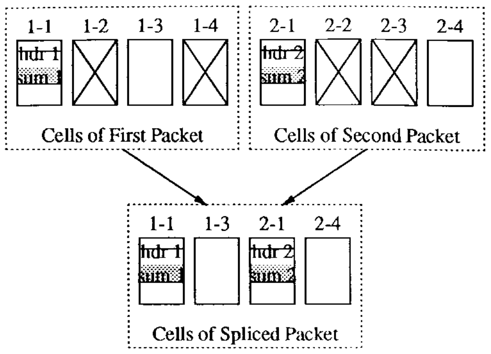

Fig. 4. Header checksum fate.

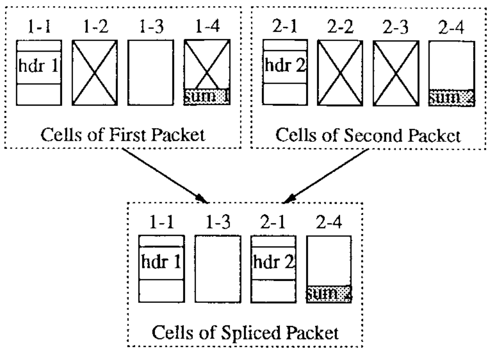

Fig. 5. Trailer checksum fate.

Since the data is nonuniform, some terms are more likely than others. The coloring effect of the B term means that the overall B sums of a splice are less likely to be congruent to the original checksum than if the data was uniformly distributed.

The end result is that the standard TCP checksum fails if two observations drawn from the same distribution are equal, while Fletcher fails if two observations drawn from the same distribution differ by a particular amount (where the exact amount varies from splice to splice). Thus, nonuniformity of the data actually strengthens the B field of the checksum.

Ones complement Fletcher, however, has a weakness that sometimes offsets its probabilistic advantage: since bytes containing either 255 or 0 are considered identical by the checksum, certain common pathological cases cause total failure of Fletcher-255. This problem is discussed in more detail in Section V-E.

C. Trailer Checksums: Making Nonuniformity Work for Us

Fletcher-256 succeeds in detecting more splices than TCP by taking advantage of the nonuniformity of the data distributions, but it still has drawbacks. It is more expensive to compute, and the nonuniformity can only strengthen 8 bits of the checksum. It turns out that we can use a similar trick to exploit nonuniformity for the standard Internet checksum, with no computational cost. Further, we can strengthen the *entire* 16-bit sum, giving us (for some distributions) 16-bit checksums that are even stronger than 1 in 2^{16}.

The key observation is that with header checksums, the packet header and the packet checksum are located in the same cell of a packet. Thus, either both the header and the checksum covering it are present in a given splice, or neither are. The IP header check and syntactic TCP header checks ensure that almost all splices which are actually checksummed include the header from the first packet. The resulting splices will have the first packet's TCP header, including TCP sequence number, ACK field, and checksum. As long as the replacement cells in the splice have the same overall checksum as the original packets, the TCP checksum will not detect the splice. Fig. 4 shows one such splice diagrammatically. If the TCP checksum was at the end of the TCP packet, instead of in the header, the TCP checksum value would not share fate with the TCP pseudo-header which it covers. Fig. 5 shows the same splice as Fig. 4, but with the TCP checksum located in a packet trailer instead of the packet header. Here, the resulting splice has the TCP header from the first packet and the checksum from the second packet. (It also has the header from the second packet, but only half the splices will do so.)

The data cells are, as with Fletcher's sum, all drawn from the same localized nonuniform distribution and are more likely than 1 in 2^{16} to have congruent sums. But compare the two headers. The only field that changes between adjacent TCP packets in a given flow is the TCP sequence number. The difference between the checksums of the header cells of adjacent packets in a single flow is therefore strongly clustered around the size of the payload. In other words there are actually three different distributions of cells in a packet pair: the payload data, the first header, and the second header. If we separate the checksum value away from the header that it sums and put it in a trailer, we can ensure that there are always three different colors in any given splice—even for splices that only make color-preserving substitutions (e.g., data cell for data cell). Again by Lemma 9, this higher degree of coloring leads to a higher probability of detecting a splice than the standard header checksum, as we show below by case analysis.[6]

What is the probability of a trailer checksum failing? It is simply that the checksum of the cells inserted from the first packet equal the checksum of the cells dropped from the second packet. (Note that we take the second packet—the source of the trailer sum—as the original, and counting cells from the first packet as insertions.)

The inserted cells from the first packet almost always include a header cell. The inserted cells from the first packet thus have a sum drawn from a distribution that consists of one header cell and k data cells. If the second header is dropped, then we again have k data cells and 1 header cell. However, in half of the splices the dropped cells are all data cells, in which case their sum consists of $k+1$ data cells. The resulting probability will be lower than the probability of an exact match between two checksums drawn from the same distribution (as shown in Lemma 9). The failure probability appears to be reduced by at most a factor of two.

In the remaining half of the splices, the second header is one of the cells dropped. Here the distribution of checksums of the header cell of the second packet does not match the

[6]Our study of trailer checksums was originally motivated by noticing that the AAL5 trailer checksums avoided this fate-sharing, and conjecturing that exploiting the predictable header differences with TCP trailer checksums would improve performance. Trailers performed surprisingly well, leading us to the preceding reanalysis of the Fletcher checksum results.

TABLE IX
TRAILER CHECKSUM RESULTS (256 BYTE PACKETS ON SYSTEMS)

Filesystem	TCP Misses	Trailer Misses
Uniform	0.001526	0.001526
sics.se:/opt	0.170344	0.004105
smeg.stanford.edu:/u1	0.070720	0.001735
pompano.stanford:/usr/local	0.026956	0.001604
sics.se:/src1	0.041172	0.002351
sics.se:/src2	0.034498	0.002100

distribution of the first header cell. There are two causes for this result. First, we treat the first header as a header but the second as data, which means we checksum the IP header of the second cell, but not of the first. Second, the header is mostly constant between packets except for an increase in the IP ID field and the TCP sequence number. (Note that in this scenario there is no checksum in the header: the field is left zero; though a practical trailer implementation might perhaps choose to swap the checksum value and the last two bytes of the packet.)

How much lower will the probability of failure be? We conducted an experiment to measure the effectiveness of trailer checksums. We changed the simulator to model a protocol identical to TCP, except that the TCP header checksum is left zero, and the checksum value is appended to the end of the TCP data. The results are shown in Table IX. The failure rate of trailer checksums were significantly better than those of TCP and Fletcher. We note that the failure rate was actually below 2^{-16} for significant fractions of some file systems. In most cases we noted a failure rate 20–50 times lower than for header checksums.

We can further test the distribution-coloring analysis by making predictions about the standard header TCP checksum. The number of splices which do not include the header of the first packet are negligible, so there are only two cases: the first header cell followed by all-data cells, and the header from the first packet followed by a mix of data cells and the header from the second packet. In the latter case, the splice has replaced a data cell with the header cell from the second packet, and thus should be much less likely to match than the first case. When we went back and examined the data, this prediction was correct. Although roughly half of the splices surviving the header check have the second header included, only 1 in 2^{16} of those passed the TCP checksum. The TCP header checksum was 100–200 times more effective against splices that contained the second header. This result both supports our explanation of the good performance of trailer TCP checksums, and further confirms the utility of our distribution-coloring analysis of checksums.

D. Adding Cell-Coloring to our Model

We are now ready to return to the discrepancies between our "exclude identical" probabilities in Table VI of Section IV-F and the actual measured failure rate. Recall that our sample probabilities predicted total failure rates very accurately for small k (the number of cells in a block), but by the time k increased to 4, the model over-predicted the measured failures by a factor of 3 or 4.

The piece that was missing from our model was the cell-coloring. The sample probabilities in our model were computed using only pure data cells, and thus missed the header effect. In our actual splice simulation, some substitutions of k cells replace a data cell with a header cell. The failure rate for the substitutions with headers should be 1 in 2^{16}, which is ignorable.

What is the probability that a substitution of length k replaced a data cell with a header cell? This is easy to compute. All k cells dropped from the first packet will be data cells. There are $\binom{6}{k-1}$ possible choices of k cell insertions from the second packet (recall that we must insert the trailing cell of the second packet in the splice). Of these, only $\binom{5}{k-1}$ do not contain the header cell of the second packet. Therefore, to predict the actual failure rate of a k-cell substitution from our "exclude identical" samples, we must reduce the sample probability by a factor of $\binom{5}{k-1}/\binom{6}{k-1}$ which equals $(7-k)/6$. Our sample probabilities now closely match the actual measured failure probabilities, and we are reasonably confident that we have explained the behavior we have observed. Further, the improved performance due to trailer checksums in our packet-splice model seems to be real.

In the past, protocol designers have proposed trailer checksums for various engineering reasons. As far as we know, the argument about improved checksum behavior was not advanced. We conclude that protocol designers should reconsider placing checksums in packet trailers rather than headers, as has been standard practice in Internet protocols to date.

Trailer checksums suffer one apparent drawback. They may unnecessarily reject splices that are identical to an original packet. Consider the scenario where a burst of cell loss splices the front of one packet onto the tail of the following packet, as in Fig. 4. If the payload of the splice is identical to the payload of the original packet, then the header checksum should match (since the header of the splice is the header of the first packet), and the packet is accepted. But with trailer checksums, (as in Fig. 5, when the payload is identical to the first packet the checksum cannot match: it was computed with the sequence number of the second packet, not the first. So if the contents are identical the checksums will match only if the difference between the inserted and dropped cells is congruent to the difference in sequence number (the payload) between the two packets. By Lemma 9, this is very unlikely. Thus the splice will be rejected even when the contents is correct. The corresponding case (header of the second packet, payload of the first) never comes up, since our error model requires cells to remain ordered. In summary, trailer checksums have a very good chance of detecting a splice even if the resulting packet is a "good" packet.

Table X demonstrates this effect on the filesystem /u1 at *smeg.dsg.stanford.edu.* The number of identical splices rejected by trailer checksums is larger than the number of bad splices they detect that the TCP checksum missed.

The two numbers, however, are not comparable. TCP missed checksums represent undetected data corruption. Spurious rejection by the trailer checksum represents (at worst) a possible performance penalty, it does not cause any data corruption.

TABLE X
HEADER VERSUS TRAILER CHECKSUM FAILURE RATES

False Positive/Negative	header	trailer
Fails checksum, data identical	0	25,348,910
Passes checksum, data changed	3,108,050	76,270
Fails checksum, data identical	0.0%	0.57%
Passes checksum, data changed	0.07%	0.002%

Comparing missed splices, the trailer checksum misses less than 3% as often as the standard sum, but at the cost of reporting checksum failure on splices that accidentally resulted in a valid packet.

However, a splice in a real network always means that at least one packet has been lost, even if the splice is identical to one of the original packets. So a TCP retransmission will be necessary regardless. Thus the incremental performance impact of triggering retransmission one packet earlier when an identical splice is discarded is not clear.

E. Locality of Failure: Pathological Data Patterns

Nonuniformity in the distribution of checksums comes from two causes: nonuniformity of the underlying data, and weakness of a given checksum algorithm for certain patterns of data. That files on a computer system are highly structured is no surprise. We did not expect, however, to discover so many examples of files that were particularly vulnerable to splice-errors.

Though the Fletcher checksums consistently show a lower rate of failure than the standard Internet checksum, they also show a very high degree of locality. Sampling the checksum statistics incrementally during each whole-filesystem run showed sharp spikes in the rate of undetected splices, at the level of individual directories or even files. Manual examination of these files shows that, for each of the checksum algorithms, real data contains pathological data patterns which cause extremely localized rates of high failure.

The most dramatic case is the mod-255 Fletcher sum. This sum has two zeros, 0 and 255. Both these values contribute zero to the cell checksum. Thus, Fletcher mod-255 is susceptible to splice failure on long runs of mixed 0 and 255 bytes. The most dramatic example of this effect is one directory from the Stanford filesystem containing several 8-bit `.pbm` graphs of Internet-backbone RTT measurements.

These graphs were plotted as black-and-white, and thus each byte is either 0 or 255. On these data, combinatorially, 1 in 2 of all permutations are caught by header checks and 1 in 2 of the remainder include a header cell. None of the remaining 25% of all possible permutations are caught by the mod-255 fletcher. This one directory of files caused so many Fletcher mod-255 failures that on this filesystem, mod-255 Fletcher performs worse than the IP/TCP checksum.

Similarly, spectacular mod-255 failures occurred in the Stanford filesystem with a file from a popular PC word processor. This file contained runs of approximately 200 all-zero bytes, followed by a similar number of all-one bytes, between each section of a document.

Fletcher's mod-256 sum behaves slightly differently. It has only one zero, and is not subject to the same dramatic failure as the mod-255 sum. Pathological data patterns for mod-256 do occur, but less frequently. One case we have isolated is hex-encoded PostScript bitmaps which contain identical segments of horizontal lines (e.g., bitmaps containing solid blocks of color, or bitmaps containing parallel lines. Font definitions appear to be a particularly common case). Many common bitmaps appear to have a width, W, that is a power of two. Thus, each ASCII-encoded binary line commonly consists of many "FF's," and a small number of other two byte values (e.g., "F7") that repeat precisely $W+1$ apart (The extra byte is due to an ASCII newline.) Though not immediately obvious on inspection, these just happen to combine in such a way that the contribution of 48-byte cells allows splices. We observed a similar effect in BinHex-encoded Macintosh documents stored on our Unix filesystem: very similar lines of 64 bytes followed by an ASCII newline.

Though the overall rate of TCP sum failures is higher than the other sums, and appears to be noisier, we have also isolated a few pathological cases for the standard Internet checksum. One example is Unix gmon.out profiling data. These files often consist mostly of zero entries, with a scattering of a small number of nonzero entries. The nonzero values are often identical. Packetizing this data results in a very small number of checksums. A very large number of splices pass the checksum, resulting in what appears to be scrambled files. A second example is the PostScript bitmap data file mentioned above, which showed pathological behavior for the Internet checksum as well.

Our central point is that the existence of pathological patterns for a given sum is not just theoretical; these patterns occur surprisingly frequently in real filesystem data.

VI. CONJECTURES

In the course of our research, we investigated several plausible conjectures that might have explained the TCP checksum failures. We briefly describe several of these blind-alleys.

A. The Role of Zero Data

The frequency of the zero checksum led us to study the effects of zeroed data on the checksum. It is no surprise that there are a lot of zeros in filesystem data (the UNIX filesystem has long been optimized such that completely zero blocks did not need to be saved on disk). However, knowing that arbitrarily long zero blocks do not change the IP checksum (zero is the additive identity), we wondered whether this property significantly affected the failure rate independent of the simple fact of their high frequency. In other words: Is there something special about zero? If we replaced all the zeros in the filesystem with different values, would the failure rate change?

An approximate first answer is that no, zero is not special because it is the additive identity. If we add one to every word in the file system then the sum of every cell would increase by 24 (48 bytes divided by 2). Similarly, it is easy to demonstrate that the distribution of the sum of any number of cells will contain the same set of values and frequencies, although their mapping will be permuted. So the rate of checksum failure would be unchanged.[7]

It is, however, true that if any single value shows up a disproportionate amount of the time then the failure rate will increase. However, the reason that zero in particular is so common is that several totally independent formats all happen to choose zero as a common element. Further, it is likely that this will continue to be the case. Fortunately, although zero checksums do show up very frequently, it is often the result of cells consisting entirely of zeros. A substitution of one all-zero cell for another causes no harm. The problem, therefore, is the frequency of nonzero cells whose checksum is zero, in proximity to all-zero cells or to each other.

B. Zero Congruent IP/TCP Header Cells

The TCP checksum is computed over a pseudo-header that covers all but eight bytes from the IP header. The TCP checksum is then inverted before it is stored in the header. Inverting the checksum causes the computed checksum of an error-free TCP datagram, (including the TCP header and checksum), to be zero.

A full IP header also contains an inverted ones-complement checksum, which means that the sum of the IP header is also zero. Since all but eight bytes from the IP header are also covered by the TCP checksum, the checksum over the combined TCP/IP header is not zero, but rather the checksum of the overlap: containing the IP source and destination addresses, the length, and the TCP protocol ID.

Our earlier results [7] were based on simulations that left the eight nonoverlapping bytes, including the IP header checksum, set to zero. The result is to cause the combined TCP/IP header to checksum to zero. Consider, therefore, two packets consisting of data that is all zero. The TCP/IP header will have a checksum of zero. The data is zero, so the checksum of the first cell will only be the sum of the header. When the checksum is inverted and stored into the header, we are left with a nonzero cell with a checksum of zero. In our earlier work, these cells were a major source of nonzero cells with a checksum of zero. What is worse, these cell show up precisely in the case when all the cells around it are zero cells (or at least zero-congruent). Thus replacement was common and a major source of splice failures. Filling in the IP header reduced the error rate by three orders of magnitude.

We had conjectured that filling in the IP header would not have much of an effect, because the length, IP addresses, and protocol type do not change between packets during the file transfer, and so the checksums of the header cell remain constant. However, even a constant, nonzero, value is sufficient to distinguish between header cells and zero filled data cells. This simulator deficiency also led us to give undue emphasis to the role of zero-congruent data (as mentioned above).

[7] Zero *is* special, as we showed in the section on pathological cases, but not because it is the additive identity and does not affect the checksum. Zero's specialness comes from the fact that it is represented by both 0 × 0000 and 0 × FFFF. In reality, adding 1 to every word in the file system *would* change the distribution of checksums, and might reduce the probability of the most probable value. Cells containing 0 × FFFF's would be shifted by less than 24. Whether this would increase or decrease the most probable value depends on the distribution of values in each filesystem.

C. Inverted Checksums

Under the TCP and IP specification, the inverse of the checksum is placed in the packet header. This implies that the checksum of a valid segment will be zero. In [7] we cautioned implementors against this approach, since for mostly—zero packets the header cell, too, would be zero. This still is reasonable advice for packet formats as it reduces the frequency of zero congruent cells. However, it is not relevant to TCP and IP because of the overlap of the headers we noted above. To test this conjecture, we ran our tests with a modified version of the TCP checksum that did not invert the checksum before storing it into the packet. The results with the noninverted checksum were almost identical to the results with an inverted checksum.

D. Corrections to SIGCOMM'95 Version

As noted in Section VI-B the data in our earlier paper [7] is not accurate. Completely filling in the IP header reduces the overall rate of errors by a factor of from 200 to 1000. In addition, the Fletcher checksum code was mis-implemented as a mixture of mod-255 and mod-256 arithmetic, which led to the Fletcher splice failure rate being higher than the standard TCP checksum. We retract that result; it was an artifact of the buggy Fletcher implementation. That bug was also the motivation for our current investigation of both mod-255 and mod-256 Fletcher sums. The artificially high Fletcher failure rates also inspired the original work on trailer checksums.

The previous results also suffered from a number of other minor bugs, whose effect was insignificant compared to the two problems above. They are detailed in the Appendix.

VII. Observations and Recommendations

The results of the previous sections lead to a number of interesting observations.

First, a nonuniform distribution of data makes failure of the TCP checksum far more likely than one would naively expect. The undetected splice rate in our data for the 16-bit TCP checksum over real data is comparable to uniform data with a 10-bit checksum.

Second, checksum distributions on modest amounts of real data are substantially different from the distributions one would anticipate for uniformly distributed data. This skewed distribution *does* result in significantly higher failure rates of the TCP checksum. In particular, if a router or host has a buffering problem that causes adjacent packets to be merged, the TCP checksum might fail 0.1% of the time rather than the 0.0015% of the time that purely random data distribution would suggest.

While these scenarios may seem worrisome, there are three pieces of good news.

First, it is important to keep in mind that these error scenarios are all quite rare. This work was initially motivated by studying extremely uncommon AAL5 error scenarios—an error model derived from ATM cell drop splicing two packets into one. In practice, such cell loss can occur due to either congestion or corruption. However, dropping ATM cells independently of each other is now known to cause goodput problems [10]. ATM switch vendors are addressing this problem by employing Early Packet Discard, which discards all cells in a packet and eliminates the chance of a splice. Cell loss due to corruption is often estimated at 1 in 10^8 or less. The ATM CRC will fail to detect a splice approximately at a rate of 1 in 2^{32}. Therefore, the chance of the TCP checksum being called upon to detect a splice is much less than 1 in $10^{-8} \cdot 2^{-32}$ or less than one chance in 10^{17}.

Second, the packet splice model is, in some sense, a worst-case error model because the substitutions tend to be similar to the data that they replace. This is possibly also true of buffer-management errors, or errors in fragment reassembly. However, in the alternative error models where data is replaced by garbage, while the nonuniformity of the data may still reduce the effectiveness of checksums, it will only reduce it to the extent that the distribution of the replacement data matches the distribution of the original data. Here, the frequency of long runs of 0's or 1's in the payload may make us slightly more vulnerable to hardware errors that produce similar runs of data. However, hardware failure that produces random bits are unlikely to produce runs of data that look a lot like English prose.

Third, and finally, remedies exist to improve the ability of checksums to work on nonuniform data.

- Compressing data clearly improves the performance of checksums. Since compression also typically reduces file transfer times and saves disk space, there's a strong motivation for FTP archives to compress their files.
- In the future, in the absence of compression, protocol designers should consider avoiding the practice of placing checksums in a protocol header, but instead append them as a trailer to the data being checksummed.
- In general, the checksums are rarely placed in a situation where it is the primary method of failure detection. (We are aware of one exception to this rule. The TCP checksum is the primary method of error detection over SLIP and compressed SLIP links. That's probably not wise.)

What this work simply shows is that checksums are an even less effective error detection method than first thought, because real data often has interesting distributions, and those distributions increase the likelihood of checksum failure.

Appendix

This paper contains assertions which depend upon statements that are easily proven, yet not immediately obvious. For those interested in the formal justification of some of the statements, we present more detail in this appendix.

A. Distributions of Checksums

We use the notation $P+R$ to denote the distribution which arises by applying *any* commutative, total function $+$ with a unique inverse on a pair of values drawn from distributions P and R respectively. (In all of our cases, we are interested in the usual arithmetic addition operator.) Call PMax(P) the probability of the most likely value in the discrete distribution, P. (We define PMin(P) similarly.) And define $P[i]$ is the probability of selecting i from P.

Lemma 1: $\mathrm{PMax}(P+R) \le \min \mathrm{PMax}(P), \mathrm{PMax}(R))$.

Proof: For any given x, the probability that the value drawn from $P+R=x$ is given by $S[x]=\sum_i P[i]R[x-i]$. Assume x is the most probable element of S. Without loss of generality, assume that PMax(P) $\le$ PMax(R). $S[x] = \sum_i P[i]R[x-i] \le \mathrm{Pmax}(P) \sum_i R[x-i] \le \mathrm{Pmax}(P)$ (since $\sum_j R[j]=1$). Equality would only hold if P were uniformly distributed and if $R[j] \ne 0 \Rightarrow P[x-j] \ne 0$. □

Lemma 2: If $\forall i$, $(P[i]=0) \Rightarrow (R[x-i]=0)$, then $\mathrm{PMin}(P+R) \ge \max(\mathrm{PMin}(P), \mathrm{PMin}(R))$.

Proof: Consider the previous proof. Given the nonzero condition on $R[j]$, we are guaranteed that every value in R appears, and so $\sum_j R[j]=1$, thus $S[x]=\sum_i P[i]R[x-i] \ge$ Pmin(P) $\sum_i R[x-i] \ge$ Pmin(P). □

This is unremarkable for unbounded discrete distributions. For the maximum, as the number of possible values grows, the probability of any single value must decrease. The conditions on the min require that $|P| \ge |R|$, and that $|S|=|P|$, so it is also unsurprising that the minimum doesn't decrease. However, for bounded distributions, e.g., distributions over the integers $\mathrm{mod}\ M$, this leads to the following more interesting results.

Corollary 3: Consider a probability distribution P over the integers $\mathrm{mod}\ M$. The distribution of the sum, $\mathrm{mod}\ M$, of j integers drawn from P gets "more uniform" as j increases, in the sense that the minimum probability of any number gets larger and the max probability gets smaller. □

Computation: If we have a random variable which can take on M values, with a known distribution of values, then the probability ($P_j[s=k]$) of the sum s of j values drawn from this distribution is equal to k is

$$\sum_{i=0}^{M} (P_{j-1}[s=(k-i) \bmod M] \times P_1[s=i]). \tag{1}$$

□

Corollary 3 shows that each time we add another number to the sum mod M and look at the probability distribution, we increase PMin(P) and decrease PMax(P). We can prove another useful result: for large enough j, PMin(P) and PMax(P) both approach $1/M$ and the distribution approaches uniform.

If P has some zero probability values, then some values in the sum of P might also have zero probability, unless the gcd of M and the entries occurring with nonzero probability is 1. The following theorem applies even if a sum of a distribution only has M' values with nonzero probability in the following sense: all nonzero values will tend to be equal to $1/M'$.

Theorem 4 (Central Limit Theorem): The sum, $\mathrm{mod}\ M$, of a large number of independent observations from any distribution P tends to have a uniform distribution.

Proof: We will show that for any given $\epsilon > 0$, there is some j such that PMax(P_j) $\le 1/M+\epsilon$. Since PMax(P_j) is nonincreasing as j grows, we know this also holds for all $k > j$. Use the notation $\max_j$ to mean PMax(P_j), and $\min_j$ to mean PMin(P_j), when the meaning is clear.

Assume there is a distribution, P, where $\max_j > 1/M+\epsilon$ for all values of j. We can compute a strict upper bound for $\max_{j+1}$ based on $\max_j$. The largest possible value of

$\max_{j+1}$ will arise when the most probable terms from P_0 match the most probable terms from P_j (cf. exercise in concrete mathematics [3], at the bottom of page 38). Assume the probability for the $M-1$ most common values in P_j are all $\max_j$, and there is 1 value whose probability is $\leq 1/M$. For P, there is at least one value with probability $\max_0$, one with probability $\min_0$, and $M-2$ values whose probability sums to $1-\max_0-\min_0$.

$$\begin{aligned}\max_{j+1} &\leq \max_0 \max_j + (1-\max_0-\min_0)\max_j + \min_0 \frac{1}{M}\\ \max_{j+1} &\leq \max_j - \min_0 \times \left(\max_j - \frac{1}{M}\right)\\ \max_{j+1} &\leq \max_j - \min_0 \times \epsilon\end{aligned}$$

but after adding $j = \max_0/(\min_0\epsilon)$ times, $\max_j$ would be less than 0, given our assumption that $\max_j$ is always greater than $1/M+\epsilon$. So, our assumption must be false.

Thus, for any distribution P and for any ϵ, there is some number j of additions, such that $\mathrm{PMax}(P_j) < 1/M + \epsilon$, so the distribution of P_j tends to the uniform distribution as j gets larger. □

B. Distributions of Some Checksums over Uniformly Distributed Data

Most existing evaluations of competing checksum algorithms have assumed that single bit errors were common. It is now frequently true that there are in the data-link layer to protect the integrity of cells on the wire, and ECC to correct memory errors while packets sit in buffers on routers. Thus, the errors that the TCP checksum must protect against are no longer single or double bit errors (which will be detected or corrected by other means), but rather substitution of longer runs of "good" data by (possibly different length) runs of "other" data. How do the IP checksum and Fletcher compare under this substitution model?

This section discusses what their expected behavior would be under substitution errors if the data were, in fact, uniformly distributed.[8]

If we assume all packets are equally likely, then if we look at any unit smaller than the size of the substitution, we can assume that an error consists of replacements drawn uniformly from all strings.

Lemma 5: The sum $S \bmod M$ of N numbers, will be uniformly distributed among all M values assuming there is at least one term, U, in the sum which takes on values uniformly distributed $\bmod M$

Proof: Assume $S-U \bmod M$ has an arbitrarily skewed distribution. $\mathrm{PMax}(U) = 1/M$, and $\mathrm{PMin}(U) = 1/M$. By Lemmas 1 and 2, $1/M \geq \mathrm{PMax}(S) \geq \mathrm{PMin}(S) \geq 1/M$. Thus, the probability that $S = x$ for any given x will be precisely $1/M$, so the probabilities are all equal and the distribution is uniform. □

Theorem 6: Given uniformly distributed data and the substitution model above, the IP checksum of the modified packet is uniformly distributed over all possible values.

Proof: We assume that errors are replacements drawn from the uniform distribution. Then (assuming replacements larger than a single 16-bit word) every word within the replaced chunk will be uniformly distributed $\bmod M$. Therefore, by Lemma 5, the IP checksum will be uniformly distributed under the assumed substitutions, since it is the sum of uniformly distributed words. That is, the checksum will only fail to detect errors (by the replacement string contributing an identical sum to the checksum as the original string) with a probability of 1 out of $2^{16}-1$. □

Theorem 7: Given uniformly distributed data and the substitution model above, the Fletcher checksum of the modified packet is uniformly distributed over all possible values.

Proof: The same reasoning can be applied to the Fletcher checksum over a chunk of data of size N. The Fletcher checksum consists of two sums. The first is the sum, mod M, of all the bytes in the chunk. The second is the sum mod M of each byte weighted by its offset, L, from the end of the chunk. Call these two sums, respectively, $K1$ and $K2$. The contribution of this chunk (assuming it is L_C from the end of the packet) to the Fletcher checksum of the entire packet is straightforward. $K1$ is added, mod M, to the mod M sum of the rest of the packet. $L_C \times K1 + K2$ is added, mod M, to the weighted sum of the rest of the packet. If $K1$ for each chunk is uniformly distributed, then so will $\sum K1$. If each $K2$ is uniformly distributed, then so will $\sum_C (L_C \times K1 + K2)$, since by Lemma 5, we only need one uniformly distributed term (and $K2$ is, although $L_C \times K1$ might not be).

That $K1$ is uniformly distributed follows directly from the lemma. $K2$ is only slightly more complicated. As long as the chunks are large enough so that the there is a byte B with offset L_0 from the end of the chunk, such that L_0 is relatively prime to M (i.e., $\gcd(L_0, M) == 1$), then B's contribution to $K2$ is uniformly distributed among all M values, and therefore, $K2$ itself is also uniformly distributed. Since $L_O = 1$ is relatively prime to M, as long as the chunk is at least $2\log_2 M$ bits long, we can apply Lemma 5.

We must also show that $K2$ is independent of $K1$, else $\{K1, K2\}$ will not be uniformly distributed. Suppose the last 2 bytes of the chunk are B_1 and B_0. Under the assumption of uniform distribution of the data, B_1 *and* B_0 are both independent and uniformly distributed. B_0 does *not* affect $K2$ since it is multiplied by 0. As we show the uniform distribution of $K2$ by varying B_1 (as we did in the lemma above), for each B_1 we can choose *any* value for B_0 to allow $K1$ to take on all values equally, without affecting $K2$. So, for each value **{K** that $K2$ might take on, $K1$ is independent and uniformly distributed.□

One last complication arises with the Fletcher checksum. Like IP, Fletcher defines the values inserted into the checksum field to be the *negation* of the checksum of the rest of the packet, so that the packet sums to 0. With Fletcher this requires the two bytes of the checksum to be the solution to a system of

[8] It is worth noting that one point of the preceding paper is that data values are *not* distributed uniformly and *are* correlated with nearby values, and that, therefore, errors, under the substitution model, are also not distributed uniformly and checksums do not perform as well as expected. This work on uniformly distributed data is still interesting on three counts. First, statements in the main body of the paper depend on results presented here. Second, it provides us with a benchmark against which to measure the actual measured error rate (i.e., what is due to the substitution model and what is do to nonuniform data). Third, encryption and compression are both becoming more common and both tend to produce uniformly distributed data.

simultaneous equations. We must show that these two *specific* bytes are independent, since we can no longer magically choose offsets 0 and 1.

Assume the Fletcher checksum $\{F1, F2\}$, is stored in adjacent bytes with offsets L_1 and $L_2 = L_1 - 1$ from the end of the packet. $0 = F1 + F2 + K1 \bmod M$, and $0 = K2 + F1 \times L_1 + F2 \times (L_1 - 1) \bmod M$:

$$\begin{aligned} F1 &= M - K1 - F2 \\ 0 &= K2 + (M - K1 - F2)L_1 + F2(L_1 - 1) \\ 0 &= K2 - L_1K1 - F2L_1 + F2L_1 - F2 \\ 0 &= K2 - L_1K1 - F2 \\ F2 &= K2 - L_1K1 \\ F1 &= M - K1 - K2 + L_1K1 \\ &= M + K1(L_1 - 1) - K2. \end{aligned}$$

Since $K2$ is uniformly distributed $\bmod M$, so are both $F2$ and $F1$. Since $F1 = F2 - K1 \bmod M$, then $F1$ is still uniformly distributed even if we hold $F2$ fixed (since we can vary $K2$ internal to $F2$). Therefore, $F1$ is independent of $F2$. □

Note that L_1K1 will not, in general, be uniformly distributed mod M, since we can't assume that $\gcd(L_1, M) = 1$ (in fact, in our example, L_1 was always equal to 260. $\gcd(260, 255) = 5$ and $\gcd(260, 256) = 4$).

As a curiosity, further note that if $L_2 - L_1$ were not relatively prime to M, then $F1$ and $F2$ would not have been independent or uniformly distributed. (In fact, the equations would not have always had solutions).

Corollary 8: Given uniformly distributed data, and the substitution model described above, IP and Fletcher checksums are equivalently powerful. □

C. Header Checksums Versus Trailer Checksums

The body of the paper claims that under our splice error model, trailer checksums are stronger than header checksums for nonuniformly distributed data and, no worse for uniformly distributed data. Here we prove that claim.

Lemma 9: Consider drawing 2 samples, X_0 and X_1, from *any* discrete distribution. The probability that $X_0 = X_1$ is greater than or equal to the probability that $X_0 = (X_1 + d) \bmod M$ for any given d.

Proof: To see this, note that the probability of the former (identical match) is simply $\sum_{i=0}^{M} P[i]^2$. The probability of the latter (d greater than the first) is $\sum_{i=0}^{M} P[i]P[i + d]$, where $i + d$ is taken $\bmod M$. Double both sums and rearrange terms. Since $(P[i]^2 + P[i + d]^2) \geq 2P[i]P[i + d]$, the former sum is greater than the latter sum. □

Consider our error model: we substitute j cells from the first packet with j other cells from the second packet. We keep the header cell of the first packet and we keep the trailer cell of the second packet. For a header checksum to fail, the sum s_1 and s_2 of each collection of j cell partial checksums must be equal. For a trailer checksum to fail, the sum s_1 of the j cells missing from the first packet must be d less than the s_2, assuming that the checksum of the header cell of packet 1 is d less than the checksum of the header cell of packet 2. We distinguish d, the difference between the header cells, since the header cells are drawn from a very different distribution than the data cells, and further, the distribution of the difference of two consecutive header cells is strongly clustered around $d = 256$. Thus, we have Theorem 10.

Theorem 10: Under our error model of splicing, a trailer checksum will always be at least as powerful as a header checksum.

Proof: For any given splice we have substituted j cells. Equation (1) gives us the probability distribution of the sum of j cells. The probability that the header checksum fails is the probability that two samples drawn from P_j are equal. As discussed above, for trailer checksums there is a fixed d, usually 256 in our simulation, computable by looking at the 2 header cells. The probability that the trailer fails is the probability that two samples from P_j differ by d. Lemma 9 shows that the former is more likely than the latter, thus header checksums are weaker than trailer checksums. □

Note, that in fact, this depends only on the property that the probability of the checksums over the header cells of two adjacent packets be congruent is lower than the probability that two data cells from the same packet be congruent. For computing the actual probability of trailer checksum failure it is useful to be able to model d as a constant 256, but this is not required for the proof.

RETRACTIONS FROM THE SIGCOMM'95 PAPER

An earlier version of this paper appeared in SIGCOMM'95 [7]. The central point of that paper still holds: nonuniform distribution of data results in the IP checksum being weaker than expected. Several conjectures expressed in [7] have been resolved and were addressed in the main body of this paper.

However, several minor points and computational details were not correct and we retract them.

First, we expressed surprise (as well we should have) that the Fletcher checksum performed *worse* than the IP checksum. Performance tuning of the Fletcher checksum code used in that paper resulted in an incorrect implementation. The Fletcher code also used a mixture of $\bmod 256$ and $\bmod 255$ arithmetic and was not computing an accurate Fletcher checksum for either $\bmod 255$ or $\bmod 256$ Fletcher.

The numbers reported for the Fletcher checksum in that paper were, therefore, not accurate. The corrected numbers reported in this version of the paper show the expected result—Fletcher's detects more splices than TCP. However, the bugs in [7] and its anomalously poor results motivated us to investigate both $\bmod 255$ and $\bmod 256$ Fletcher, uncovering the pathological cases for $\bmod 255$ Fletcher reported here.

The SIGCOMM'95 paper reports numbers where the IP header fields not covered by the TCP checksum were left as zero.

Though covered in the body of this paper, it is important to emphasize it again here: filling in the header significantly reduced the number of matches for zero-congruent cells, and therefore reduced the total number of misses (by three orders of magnitude in some cases). By zero-filling in the IP header in [7] we over-stated the significance of splices including zero-congruent cells and focused too closely on misses involving zero-filled or zero-congruent cells.

Several additional, but relatively minor bugs in the simulator compromised the accuracy of the numbers of all checksum algorithms in [7] (but only to a small factor).

First, we used the AAL5 length from the second packet, rather than the apparent IP length from the first cell, for checksum computation. This miscomputed checksums by including data from the last cell beyond the end of the IP payload in the checksum.

Second, this same error arose when testing whether packets were "identical" in payload. This resulted in counting certain splices as checksum failures, when in fact they were simply identical to the original packet, or where the first packet was a prefix of the splice.

Third, we miscomputed the checksum for short packets—that is, packets where the apparent IP header length made the entire TCP packet fit into the first cell and the AAL5 trailer in the second cell. It is well known that a TCP packet with any user data fills at least two ATM cells. But for packets with 1 to 8 bytes of TCP payload, the entire IP/TCP datagram fits in only one cell and the second cell contains only an AAL5 trailer. Knowing that TCP data packets always take two cells, the simulation in [7] erroneously added a partial checksum for the second cell.

These erroneous calculations did not change the larger picture of TCP checksum performance, but did require us to recompute all data for this version of the paper.

Finally, our code and raw data are available via email request to the authors.

Acknowledgment

The authors would like to acknowledge the help of C. Kalmanek and B. Marshall of Bell Labs, who discussed issues of study design. They also gratefully acknowledge the help of D. Feldmeier of Bellcore and L. Sloan of Lawrence Livermore, who helped with substantially faster CRC computation algorithms, to the Swedish Institute of Computer Science, which allowed them to use its filesystems and one of its fast multiprocessors for some of the test runs, and to the engineers of StorTek for enduring long running background programs on their machines on several occasions. They would also like to thank J. Crowcroft, TON editor, and his reviewers, who pushed them to look even more deeply at some checksum behavior. The result was a multiyear delay in publication as they collected more data but also a much deeper understanding of checksums, which they hope this paper conveys.

References

[1] R. Braden, D. Borman, and C. Partridge, "Computing the Internet checksum," Internet Request For Comments RFC 1071, Sept. 1988 (Updated by RFC's 1141 and 1624.)

[2] J. Fletcher, "An arithmetic checksum for serial transmissions," *IEEE Trans. Commun.*, vol. COM-30, pp. 247–252, Jan. 1983.

[3] R. Graham, D. Knuth, and O. Patashnik, *Concrete Mathematics: A Foundation for Computer Science.* New York: Addison Wesley, 1989.

[4] D. Greene and B. Lyles, "Reliability of adaptation layers," in *Protocols for High-Speed Networks III, Proc. IFIP 6.1/6.4 Workshop,* B. Pehrson, P. Gunningberg, and S. Pink, Eds., 1992.

[5] J. L. Hammond, Jr. *et al.*, "Development of a transmission error model and an error control model," Tech. Rep., Georgia Inst. of Technol., May 1975, prepared for Rome Air Development Center.

[6] A. Nakassis, "Fletcher's error detection algorithm: How to implement it efficiently and how to avoid the most common pitfalls," *Comput. Commun. Rev.*, vol. 18, no. 5, pp. 63–88, Oct. 1988.

[7] C. Partridge, J. Hughes, and J. Stone, "Performance of checksums and CRC's over real data," in *Proc. SIGCOMM'95,* vol. 25, no. 4, of *Computer Communication Review.* Boston, MA, 1996, pp. 68–76.

[8] W. W. Plummer, "TCP checksum function design," Internet Engineering Note 45, BBN, 1978 (reprinted in [1]).

[9] J. Postel, "Transmission control protocol," Internet Request for Comments RFC 793, vol. 3, Sept. 1981.

[10] A. Romanow and S. Floyd, "Dynamics of TCP traffic over ATM networks," *IEEE J. Select. Areas Commun.*, vol. 13, pp. 633–641, May 1995.

[11] K. Sklower, "Improving the efficiency of the OSI checksum calculation," *Comput. Commun. Rev.*, vol. 19, no. 5, pp. 44–55, Oct. 1989.

[12] Z. Wang and J. Crowcroft, "SEAL detects cell misordering," *IEEE Network Mag.*, vol. 6, pp. 8–19, July 1992.

[13] J. Zweig and C. Partridge, "TCP alternate checksum options," Internet Request For Comments RFC 1143, Feb. 1990.

Jonathan Stone received the M.Sc. degree with distinction from Victoria University of Wellington, New Zealand, and is currently working toward the Ph.D. degree at Stanford University, Stanford, CA.

Michael Greenwald (M'98) received the S.B. degree from the Massachusetts Institute of Technology, Cambridge, MA, and is currently working toward the Ph.D. degree at Stanford University, Stanford, CA.

He will be an Assistant Professor at the University of Pennsylvania, Philadelphia.

Craig Partridge (M'88–SM'91) received the A.B., S.M, and Ph.D. degrees from Harvard University, Cambridge, MA.

He is a Chief Scientist with BBN Technologies, Cambridge, MA, a part of GTE Corporation. He is also a Consulting Assistant Professor at Stanford University, Stanford, CA.

James Hughes (M'90) is a Fellow at Storage Technology Corporation specializing in High Performance Channels and Networks and Information Security Issues.