Performance of Checksums and CRCs over Real Data

*Craig Partridge (*Bolt Beranek and Newman, Inc*)*†
*Jim Hughes (*Network Systems Corporation*)*
and
*Jonathan Stone (*Stanford University*)*

Abstract

Checksum and 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 CRCs 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. In one dramatic case, 0.01% of the check values appeared nearly 19% of the time. These results have implications for CRCs and checksums when applied to real data. They also cause a spectacular failure rate for the both TCP and Fletcher's checksums when trying to detect certain types of packet splices.

1. 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 random. (See, for instance, the work on Fletcher's checksum [1] and the AAL5 CRC [2,3]). If one assumes random data one can show a number of nice error detection properties for various checksums and CRCs. But in the real world, communications data is rarely random. Much of the data is character data, which has distinct skewing towards certain values (for instance, the character 'e' in English). Binary data has similarly non-random distribution of values, such as a propensity to contain zeros.

This paper reports on experiments with running various checksums and CRCs over real data from UNIX filesystems. We show that the non-random distribution of values in real data causes extremely irregular distributions of checksum and CRC values. In some tests, less than 0.01% of the possible checksum values occurred over 19% of the time. We particularly examine the effects of this phenomenon when applied to the TCP checksum [4,5].

2. CRCs vs. Checksums

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

CRCs are based on polynomial arithmetic, base 2. CRC-32 [6] 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, the TCP checksum [4,5] (also used for IP and UDP) and Fletcher's checksum [1], 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 16 bits or less, and over uniformly distributed values of data 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 1970s 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 is a running sum of the data in 8-bit chunks. The other sum is a running sum of the intermediate values of the first sum. The two 8-bit sums are combined to generate a 16-bit checksum. Fletcher also defined a 32-bit version, where 16-bit sums are kept. The algorithm was defined for ones and twos complement arithmetic. The version used for the TP4 checksum and in this paper uses 8-bit chunks and twos complement arithmetic. Performed in twos complement, the 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. The major known failing of the checksum is that it is unaffected by zeros being added or deleted from either the front or the back of a packet [8].

Historically, the TCP checksum and Fletcher's checksum have been viewed as offering a sharp choice between performance

† Craig's work was supported, in part, by the U.S. Department of Defense.

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). As a result, measurements have typically shown the TCP checksum to be two to four times faster [9]. However, that difference may be declining on newer processors (where the memory access time dominates any computational cost).

3. 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 has not been published and helps motivate the rest of the paper and so is explained briefly here.

3.1. 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. Figure 3.1 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.

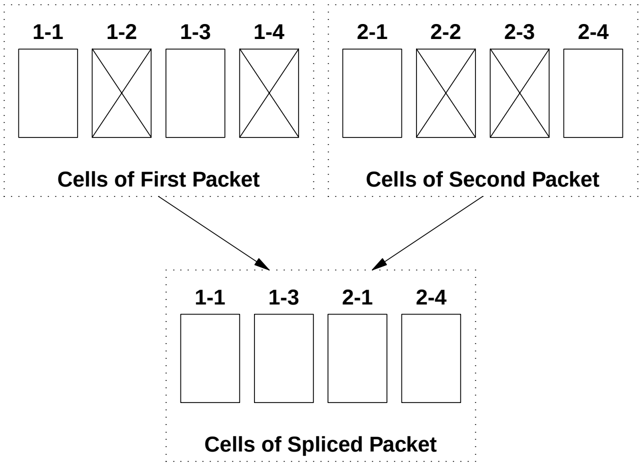

Figure 3.1: Example AAL5 Splice

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 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 checking 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 Bill Marshall and Chuck 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.

3.2. 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 TCP header fields were filled in as if the file transfer was being done over the loopback interface (127.0.0.1) and the TCP sequence number was incremented for each packet. All IP header fields included in the TCP checksum were also filled in. Other IP header fields were set to zero. 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. If either failed, the program then checked to see if the data in the splice matched the data in the first or second segment.

Table 3.1: CRC and TCP Checksum Results
(256 Byte packets on systems at NSC)

code	splices	%
X	37	0.00000001114
HX	37	0.00000001114
C	3177810066	0.95747494293
HC	23000361	0.00693001434
XC	158753861649	47.83257698434
HXC	165883503938	49.98073993369
FC	4056678126	1.22227810238
total	331894854214	99.99999999996

The test program was run over file systems at Network Systems Corporation (NSC), the Swedish Institute of Computer Science (SICS), and Stanford University, as well as over a modest quantity of random data (generated using the `random` routine in the UNIX library). The TCP segment sizes examined were 256 bytes long, except for runt packets at the end of files. The code column indicates the type of splice. A C indicates the splices were detected by the CRC, an X indicates the splices were detected by the checksum, an H indicates the splices were detected by the TCP header checks, and an F indicates that the splice had data identical to the first packet. A P indicates the splices passed both the CRC and the checksum, or in section 5, where the CRC was not computed, indicates splices that passed the checksum. The NSC data is consolidated. The SICS and Stanford data are by file system. The total number of splices is greater than 2^{32}, and the percentage columns do not always add to 100%, due to floating-point error.

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 0.0000000232% 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 0.001526% 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 data must equal zero.

Table 3.2: CRC and TCP Checksum Results
(256 Byte packets on systems at SICS)

system	code	splices	%
anat.sics.se	X	2	0.0000000425
/home1	HX	2	0.0000000425
54,381 files	C	27215729	0.5779950000
5,182,439 pkts	HC	100667	0.0021379200
(5-9-95)	XC	2292853743	48.6946000000
2% executables	HXC	2354575619	50.0054000000
	FC	33896991	0.7198890000
	total	4708642753	100.0000220050
anhur.sics.se	C	3915848	0.3075490000
/gnu	HC	20544	0.0016135200
14,210 files	XC	627917291	49.3164000000
1,400,553 pkts	HXC	636661787	50.0032000000
(5-9-95)	FC	4726863	0.3712460000
	total	1273242333	100.0000085200
anhur.sics.se	C	17206539	1.2252400000
/home2	HC	15190	0.0010816500
11,976 files	XC	664676011	47.3301000000
1,538,940 pkts	HXC	702017150	49.9890000000
(5-9-95)	FC	20427081	1.4545700000
13% executables	total	1404341971	99.9999916500
aten.sics.se	X	1	0.0000000239
/home1	C	12638895	0.3019130000
51,745 files	HC	101221	0.0024179300
4,603,993 pkts	XC	2065716781	49.3450000000
(5-10-95)	HXC	2092873982	49.9937000000
	FC	14942433	0.3569390000
	total	4186273313	99.9999699539
aten.sics.se	X	1	0.0000000223
/src3	C	20643760	0.4600540000
43,966 files	HC	441607	0.0098413800
4,926,979 pkts	XC	2199597472	49.0189000000
(5-9-95)	HXC	2243111896	49.9886000000
	FC	23451293	0.5226210000
	total	4487246029	100.0000164023
fafner.sics.se	X	1	0.0000000226
/opt	HX	3	0.0000000679
116,493 files	C	127091923	2.8771900000
4,952,333 pkts	HC	64427	0.0014585400
(5-9-85)	XC	1927868532	43.6444000000
0.2% executables	HXC	2211698912	50.0699000000
	FC	150496407	3.4070400000
	total	4417220205	99.9999886305
goodyear.sics.se	X	1	0.0000000274
/home1	HX	1	0.0000000274
38,918 files	C	6354096	0.1738850000
4,016,824 pkts	HC	44398	0.0012149900
(5-11-95)	XC	1813450458	49.6265000000
	HXC	1827095605	49.9999000000
	FC	7251253	0.1984360000
	total	3654195812	99.9999360448
max.sics.se	C	8308523	1.9317600000
/usr	HC	25604	0.0059530200
10,874 files	XC	196565048	45.7021000000
480505 pkts	HXC	215269141	50.0508000000
(5-9-95)	FC	9932712	2.3093900000
SPARC executables	total	430101028	100.0000030200
soma.sics.se	HX	2	0.0000001111
/home1	C	6440806	0.3578350000
16,744 files	HC	78526	0.0043627000
1,974,681 pkts	XC	888128423	49.3422000000
(5-10-95)	HXC	899738559	49.9872000000
	FC	5552201	0.3084660000
	total	1799938517	100.0000638111

The tables show that for real data, the CRC failure rate is almost perfectly consistent with the expected failure rate for random data. (The difference between our results and those of Marshall and Kalmanek is the FC entries. Our tests incremented the TCP sequence number for each segment while their's did not. The difference means that splices that match the first packet will fail the CRC in our tests, but would have passed the CRC in their tests). For TCP, however, the story is different. Between 0.2% and 3% of the splices (those coded C and HC) passed the checksum.

Table 3.3: CRC and TCP Checksum Results
(256 Byte packets on systems at Stanford)

system	code	splices	%
pescadero.stanford.edu	C	24293449	1.4294800000
/u2	HC	16170	0.0009514810
36,215 files	XC	793495112	46.6911000000
1,896,212 pkts	HXC	850863326	50.0668000000
(5-11-95)	FC	30788067	1.8116400000
	total	1699456124	99.9999714810
bump.stanford.edu	HX	1	0.0000000332
/usr	C	8652367	0.2873880000
24,262 files	HC	95320	0.0031660500
3,298,360 pkts	XC	1486821267	49.3847000000
(5-11-95)	HXC	1504581412	49.9747000000
RS6000 executables	FC	10538800	0.3500460000
	total	3010689167	100.0000000832

4. 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 (especially zeros) and how data from one packet can be mixed with data from another packet.

4.1. Failure Scenarios

The TCP checksum has two interesting properties. First, it is not dependent on the order of its data. The same data (considered as a series of 16-bit integers) can be permuted in any order and still have the same sum. Second, 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.

4.2. Distributions of the TCP Checksum

A key issue as we consider different error scenarios is whether the TCP checksum algorithm gives an even distribution of checksum values over random data. Given random pieces of data, a good checksum or CRC should uniformly scatter the checksums 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. So, before considering more complex error scenarios, the first challenge is to determine whether the TCP checksum inherently has hotspots.

The simplest way to test whether a checksum has hot spots is to apply the checksum algorithm to a large number of random pieces of data, compute how many times each checksum occurred, and then plot, for a given frequency, x, how many checksums y occurred that many times. If the checksums are uniformly distributed, the graph should be a normal distribution.

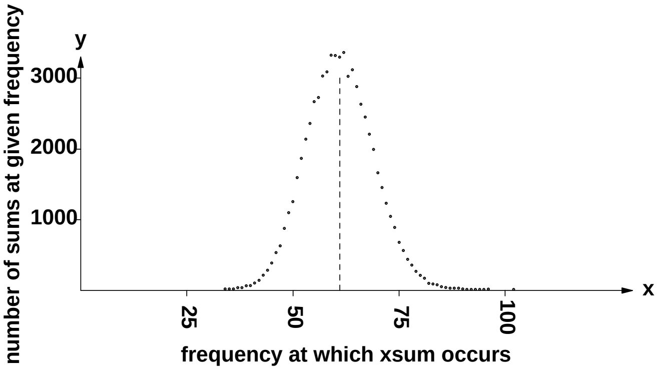

Figure 4.1: TCP Checksum Frequencies
Random Data (48-byte)
[Run of 5/8/95: 4 mil. sums]

Tests with random data (generated using the UNIX `random` function) strongly suggest the TCP checksum uniformly distributes its checksums when given random data. The plot for 1 million checksums of 48-byte chunks of random data is shown in Figure 4.1. Given that the TCP checksum uniformly distributes values when given random data, we must look for more complex relations between the TCP checksum and its data to explain the high rate of failures on splices.

4.3. The Distribution of Checksum Values

If the distribution of cell checksum values is completely uniform, we can roughly approximate the chance of a cell in the first packet having the same checksum as a cell in the second packet by using the binomial distribution. Given a packet, length n, the chance that a given checksum value occurs at least once in the packet is:

$$\sum_{i=1}^{n} \binom{n}{i} p^i (1-p)^{n-i} \qquad p = .000015259$$

Except for the case where $i = 1$ the terms in this equation rapidly go to zero, so one can approximate the chance of two cells in the respective packets having the same sum multiplying 0.000015259 by n. A more accurate set of values for $n = 5$ is shown in Table 4.1.[1] The table shows the chance that a particular sum appeared a certain number of times in the second packet, given that the sum appeared one or more times in the first packet.

If two cells have the same checksum, how many valid splices can they create? TCP would give the same checksum for all $n!$ permutations of the $n-1$ cells of the first packet and the one cell of the second packet. So one might think there would be $n!$ valid splices. However, ATM requires that cells be transmitted in order, so only one permutation exists: the cells from the first packet followed by the cells from the second packet. Putting these results together, the chance of a valid splice occurring a random world is the chance of two cells having the same checksum multiplied by the fraction of splices in which the cell from the second packet can replace its twin in the first packet. For even small n, this value is extremely small. For $n = 5$ it is about 0.000003.

[1] Readers should note that for 256 byte packets, there are 7 cells per packet, not 5. The value $n = 5$ here was chosen simply because it is the largest number we could legibly fit on the page. The probabilities for $n = 5$ and $n = 7$ are very similar.

Table 4.1: Conditional Checksum Probabilities on Random Data

Given that a checksum value occurs x times in one set of 5 cells, how many times, y, does the checksum value occur in the next 5 cells?

x	y=0	y=1	y=2	y=3	y=4	y=5
1	0.999862677382	0.000076285687	0.000000002328	0.000000000000	0.000000000000	0.000000000000
2	0.000061028550	0.000000004656	0.000000000000	0.000000000000	0.000000000000	0.000000000000
3	0.000000001397	0.000000000000	0.000000000000	0.000000000000	0.000000000000	0.000000000000
4	0.000000000000	0.000000000000	0.000000000000	0.000000000000	0.000000000000	0.000000000000
5	0.000000000000	0.000000000000	0.000000000000	0.000000000000	0.000000000000	0.000000000000

Table 4.2: Conditional Checksum Probabilities on Real Data (fafner.sics.se:/opt)

x	y=0	y=1	y=2	y=3	y=4	y=5
1	0.8188250717	0.0126745169	0.0032303349	0.0011729930	0.0008442541	0.0005629384
2	0.0088466738	0.0065433916	0.0033164332	0.0011795923	0.0012690670	0.0001506336
3	0.0033005104	0.0023542735	0.0023399238	0.0015566367	0.0009947726	0.0002762128
4	0.0024117107	0.0033737553	0.0027796619	0.0023851215	0.0015081009	0.0400142085
5	0.0018292045	0.0027993831	0.0017058891	0.0015135492	0.0501021707	0.0201390142

Table 4.3: Conditional Checksum Probabilities on Real Data (anhur.sics.se:/home2)

x	y=0	y=1	y=2	y=3	y=4	y=5
1	0.9953196864	0.0015850563	0.0002118544	0.0000337683	0.0000048791	0.0000019259
2	0.0014137752	0.0004401436	0.0001526636	0.0000396746	0.0000091162	0.0000023111
3	0.0001951629	0.0001155570	0.0000597044	0.0000359511	0.0000154076	0.0000003852
4	0.0000530278	0.0000216990	0.0000195163	0.0000222126	0.0000109137	0.0000493043
5	0.0000247805	0.0000206719	0.0000174619	0.0000133532	0.0000719021	0.0000381338

Table 4.4: Conditional Checksum Probabilities on Real Data (bump.stanford.edu:/usr)

x	y=0	y=1	y=2	y=3	y=4	y=5
1	0.9709298340	0.0020089962	0.0002286810	0.0000954607	0.0001466410	0.0000577010
2	0.0023987245	0.0011726727	0.0006384265	0.0004155080	0.0004798814	0.0001462618
3	0.0009802349	0.0004860988	0.0006397913	0.0004921646	0.0003992061	0.0001901631
4	0.0006842992	0.0004100487	0.0008237370	0.0006968099	0.0007820345	0.0045046220
5	0.0004462920	0.0004760902	0.0005193091	0.0005259057	0.0058064964	0.0024179076

Table 4.5: Conditional Checksum Probabilities on Real Data (pescadero.stanford.edu:/u1)

x	y=0	y=1	y=2	y=3	y=4	y=5
1	0.9458134207	0.0030865192	0.0003125627	0.0000583623	0.0000535736	0.0000580630
2	0.0028412977	0.0005326435	0.0002656733	0.0000901873	0.0001006625	0.0000681393
3	0.0008459046	0.0001763839	0.0001374757	0.0000848997	0.0000633506	0.0000468894
4	0.0005934002	0.0001540366	0.0001508442	0.0000791134	0.0001131331	0.0152492286
5	0.0004217052	0.0004531311	0.0004376676	0.0003826973	0.0192650557	0.0080639779

When we look at real data, however, the results are very different. Tables 4.2 through 4.5 show the same information as Table 4.1 but for real packets of length 5 from file systems at SICS. The probabilities are wildly different from those found for random data. On the opt file system, 18% percent of packet pairs have at least one sum in both packets. On the other file system (home2), only 0.5% of the packet pairs share at least one sum, but the distribution is tail-heavy − a lot of packet pairs share multiple sums in common. However, all the real filesystem distributions are in sharp contrast to the expected values for random data, where only 0.00138% of packet pairs share sums, and all but the very upper-left corner of table 4.1 contains zeroes.

This data shows that the TCP checksum on real data has hotspots. Particular sums appear far more frequently than random chance would suggest. One can also see this behavior in Figures 4.2 and 4.3. The figures are a plot of the frequency with which checksums occurred (using the same approach as in Figure 4.1). The checksums came from computing the checksum on successive 48-byte chunks of two file systems from SICS. This plot should be a normal curve centered around the vertical dashed line. Instead the curve is extremely tail heavy for one system and somewhat tail heavy for the other. If one examines the data closely, one discovers two things. First, that the checksum value of zero occurs between 7% and 15% of the time. Second, that 65 most frequently occuring checksum values other than zero (0.1% of the checksum space) account for between 1% and 4.5% of the checksum values seen.

It is important to note that the hotspots are caused by the data, not by some inherent characteristic of the TCP checksum. For instance, while the TCP checksum for a cell is zero much of

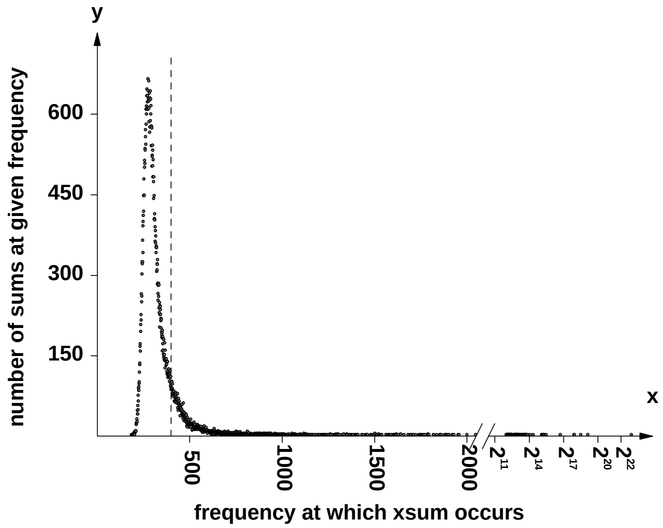

Figure 4.2: TCP Checksum Frequencies
Real Data (fafner.sics.se:/opt 48-byte chunks)
[Run of 5/10/95: 26,176,974 sums]

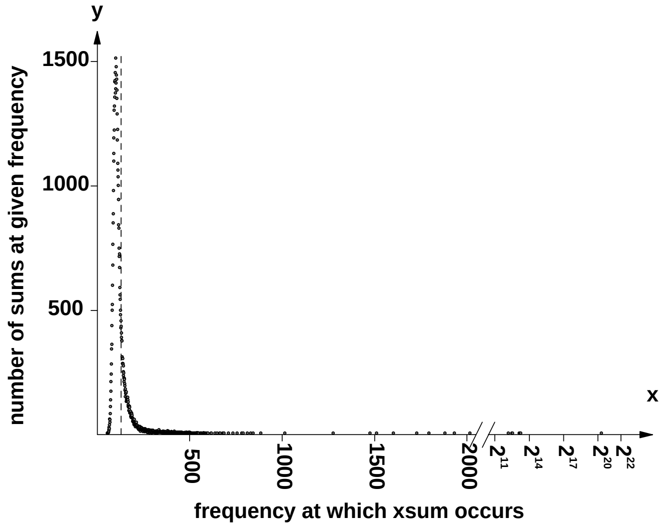

Figure 4.3: TCP Checksum Frequencies
Real Data (anhur.sics.se:/home2 48-byte chunks)
[Run of 5/10/95: 8,355,409 sums]

the time, in almost all of those situations, the data in the cell is all zeros.

4.4. The Role of Zeros

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). But we were interested to find some unanticipated effects of zeroed data.

Consider the situation in which the two adjacent packets both have data which is entirely zero (between 2% and 11% of adjacent packets were all zeros on filesystems at SICS). Most of the possible splices will involve replacing one zeroed cell with another, which will result in packets which pass the checksum because they equal the first packet, but fail the CRC (because the CRC is in the trailing packet, and thus includes a different TCP sequence number). Almost all of the splices labelled FC above in the tables in section 3 are the result of mixing zero cells.

However, another form of splice can also occur. When the TCP checksum is computed, it will sum the TCP pseudo header and then a series of zeros. The resulting checksum will just be the checksum of the TCP pseudo header. Then under TCP rules, the inverse of the checksum value is placed in the TCP checksum field, so that a checksum of a valid segment will sum to zero.

Since the TCP/IP header fits in the first cell (unless there are IP options), the result is that all cells except the first cells of each packet have a checksum of zero, and the TCP portion of the first cells has a checksum of 0xFFFF (the other zero in ones complement). In our tests, the portion of the IP header not included in the TCP pseudo header was set to zero, with the result that the checksum of the first cells stayed 0xFFFF. Since 0xFFFF and 0 are equal in ones complement, this meant that the header cell of the second packet had the same checksum as a cell of zeros and could replace a cell of zeros in a splice and pass the TCP checksum. Many of the splices labelled C come from replacing a zeroed cell in the first packet with the header cell from the second packet. Observe that both header cell replacement and mixing of zero cells can also occur even if only parts of both cells are zeros.

What if the IP header had been filled in? The IP header also contains a checksum, which makes the IP header sum to zero. So the first cell would have two overlapping headers, each of which, when their checksum is included, would sum to zero. The net result is that the checksum of the first cell, with the IP header filled in, would be the inverse checksum of TCP pseudo header, namely the IP source and destination addresses plus the TCP length and the TCP protocol id.[2] While we have not analyzed the typical combinations of the IP address space and TCP length as a checksum value, we suspect they are not a very good error check. For one thing, with the exception of the length, these fields do not typically change during the lifetime of a TCP connection, and the length is usually constant for file transfers.

4.5. Zeros Are Not the Whole Story

While the presence of groups of zeros in the data explain a large number of the splices that pass the TCP checksum, they do not come close to explaining all of them. There are several ways to illustrate this point. One way is to observe that the number of zero checksums is simply the highest value (on the *x* axis) in Figures 4.2 and 4.3. If that point is removed, the distributions are still tail heavy. Another is to look at Tables 4.6 and 4.7, which show the conditional probabilities that the same checksum appears in two adjacent packets, but does not include zero

[2] Imagine checksumming the 40-byte combined TCP/IP header. After summing the IP header, the sum will be zero. So the checksum will simply be the sum of the TCP header and we know the sum of the TCP header plus the pseudo header would have been zero.

checksums. The tables are derived from the same file systems as Tables 4.2 and 4.3, and while the probabilities are better, they are still not close to the expected values in Table 4.1. It is still very common for two adjacent packets to have cells with the same checksum. And even if one eliminates all packets with zeroed cells from consideration, the number of splices that pass the checksum is between 0.0058% and 0.1%, which is 4 to 75 times larger than random chance would suggest. So we still need to explore other error scenarios that exploit the skewed checksums.

Table 4.6: Conditional Checksum Probabilities on Real Data (fafner.sics.se:/opt)
Zero checksums (0x0 and 0xFFFF) not included.

x	y=0	y=1	y=2	y=3	y=4	y=5
1	0.9488523081	0.0140521197	0.0025279461	0.0003277124	0.0001275698	0.0000266086
2	0.0096444439	0.0062813985	0.0024742840	0.0009447371	0.0003739881	0.0000501469
3	0.0023721214	0.0012702247	0.0016497748	0.0012360519	0.0004553711	0.0001027856
4	0.0009740155	0.0006359355	0.0010509489	0.0008959251	0.0004334347	0.0006680616
5	0.0003891167	0.0002626149	0.0002574533	0.0002934505	0.0009277397	0.0004417109

Table 4.7: Conditional Checksum Probabilities on Real Data (anhur.sics.se:/home2)
Zero checksums (0x0 and 0xFFFF) not included.

x	y=0	y=1	y=2	y=3	y=4	y=5
1	0.9953212093	0.0015858241	0.0002117855	0.0000337573	0.0000048775	0.0000019253
2	0.0014120316	0.0004400004	0.0001526139	0.0000396616	0.0000091132	0.0000023104
3	0.0001950994	0.0001155194	0.0000596850	0.0000359394	0.0000154026	0.0000003851
4	0.0000530105	0.0000216920	0.0000195099	0.0000222054	0.0000109102	0.0000492883
5	0.0000247725	0.0000206651	0.0000174563	0.0000133489	0.0000718787	0.0000381214

4.6. Cell for Cell Replacement

The easiest error scenario is that a cell in the second packet has the same checksum as a cell in the first packet and replaces that cell in the splice. The resulting splice will pass the TCP checksum.

The checksum hotspots mean that there is a higher chance of a cell in the second packet successfully replacing the cell in the first packet. However, because ATM requires that cells be delivered in order, if two cells can replace each other, there is only one possible bad splice out of the hundreds or thousands of possible splices for a pair of packets. When one weights the chance of matching cells by the fraction of valid splices those cells can produce, one finds that cell for cell replacement can explain some (perhaps as much as half) of the checksum failures.

4.7. MultiCell Replacement

Multicell replacement is when two or more cells in the first packet are replaced by two or more cells in the second packet, and the replacement cells have the same aggregate checksum as the cells they replace. Our test data indicates that multicell replacement is the reason for the high number of splices that pass the checksum, when zeroed cells are included. When zeroed cells are excluded, multicell replacement appears to account for the remaining splices that pass the TCP checksum.

The importance of multicell replacement comes from two factors. First the skewed distribution of checksum for individual cells persists when one computes the sum of two or three cells' checksums. Figure 4.5 illustrates the point by showing the distribution for sums of two cell checksums on the same file system used for the earlier figures. The distribution of three cell sums is very similar.

Second, there are far more opportunities for multicell checksums to match than for one cell checksums to match. For a packet of length *n* cells, there are *n* one cell checksums in the first packet that might have a match among the *n* cell checksums of the second packet. But there are $\sum_{i=1}^{n-1} i$ sums of two checksums in the first packet that might have a match in the second packet.

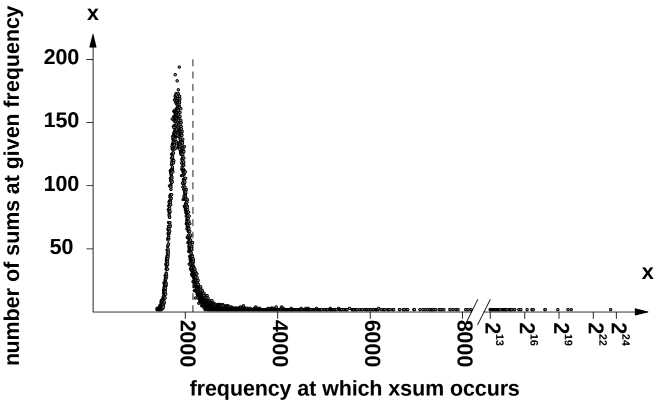

Figure 4.5: Frequencies for Sums of Two Checksums
Real Data (fafner.sics.se:/opt 48-byte chunks)
[Run of 1/30/95: 141,701,500 sums]

The results can be striking. For instance, in one test there were 15,380,108 cases where a single checksum in one five-cell packet matched a checksum in the following packet. However, there were 85,3505,65 cases where a two-cell sum in the first packet matched a two-cell sum in the second packet.

4.8. Location of Checksum Field in Packet

Even multi-cell replacement is not the whole story. At least for TCP, checksums located in a trailer will perform better at detecting splices than checksums located in a packet header. Consider the scenario where a burst of cell loss splices the front of one packet into the tail of the following packet. The resulting splice 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. Figure 4.1. shows this diagrammatically. In contrast, the CRC in such a splice will be the CRC value from the second packet. This CRC was computed over the entire second packet, including the second packet's TCP header fields. Since our

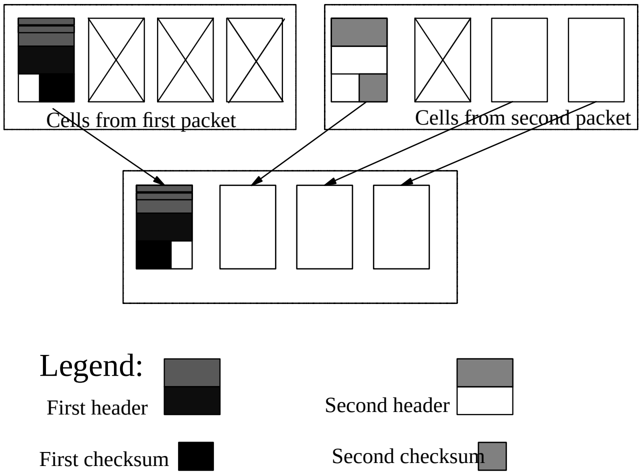

Figure 4.6: Fate-sharing with checksums in headers

Figure 4.6: Fate-sharing with checksums in headers

simulator never retransmits TCP segments, the sequence number fields of two adjacent TCP packets are guaranteed to be different. The AAL5 CRC is therefore has an excellent chance of detecting such splices, even when the TCP payload in the splice is identical to the first packet. The two key factors here are that cells are not reordered, and that the AAL-5 CRC is located at the end of the last cell of a packet. Those two facts guarantee that the AAL5 checksum and the TCP header can never share fate in a splice. If the TCP checksum was at the end of the TCP packet, instead of in the header, the TCP checksum would not fate-share either. Figure 4.7 shows the same splice as Fig. 4.6, but with the TCP checksum located in a packet trailer instead of the packet header. Here, the fate-sharing is avoided, the resulting splice has the TCP header from the first packet and the checksum from the second packet. Even if the new cells in the splice have the same checksum as the cells they replaced, the TCP checksum still has a good chance of detecting the splice.

We conducted an experiment to see if checksum fate-sharing was a real effect. We changed the simulator to model a protocol identical 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. In one experimental run over a single filesystem of real data, the undetected splice rate dropped to 1 in 2ˆ16. We conclude that protocol designers should consider placing checksums in pacekt trailers rather than headers, as has been standard practice in Internet protocols to date.

5. Alternatives: Compression and Fletcher's Checksum

So far, we have focussed on explaining the failure of the TCP checksum to catch a large number of splices. This section briefly considers some related topics, in particular, whether compressing the data improves checksum performance and whether other checksums, in particular, Fletcher's checksum performs better.

The TCP checksum's failure to detect many splices is apparently due to regular patterns (most notably zeros) in the data being summed. One obvious way to deal with repeated data patterns is to compress the data. As an experiment to verify that the problem was in repeated data patterns, 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 results are shown in Table 5.1. The interesting result is that the number of splices that passed the checksum is approximately 0.0018%, which is very close to the expected rate, on random data, of 0.0015%. So compression clearly helps.

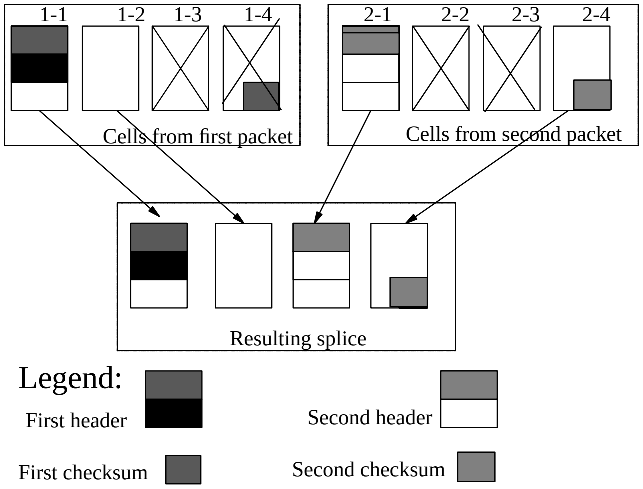

Figure 4.7: Fate splitting with checksums in trailers

Figure 4.7: Fate splitting with checksums in trailers

Table 5.1: CRC and TCP Checksum Results
(256 Byte packets on systems at SICS)

system	code	splices	%
fafner.sics.se	C	16295	0.0010513786
/opt compressed	HC	11763	0.0007589670
1,679,166 pkts	XC	775856442	50.0595000000
	HXC	773933354	49.9354000000
(5-9-85)	FC	51902	0.0033488000
	total	1549869756	100.0000591456

Another obvious question to ask is whether another checksum algorithm would perform better. The clear candidate checksum is Fletcher's checksum, which has provably stronger error properties. Table 5.2 shows the results for Fletcher's checksum for three filesystems. The results can be compared with those for the TCP checksum on those filesystems in Table 4.2. The one difference is that in the tests for Table 5.2 the AAL5 CRC was turned off, so the splices that the Fletcher's checksum failed to catch are under codes P and H (rather than C and HC in Table 4.2). The results were very surprising − Fletcher's checksum was worse than the TCP checksum at catching invalid splices! The result is surprising for two reasons. First, Fletcher's checksum is supposedly stronger. Second, when looking at plots of checksums over 48-byte chunks (Figure 5.1), the Fletcher's checksum looks to have a curve similar to that of TCP and because the value of Fletcher's checksum is position dependent (i.e., the contribution of the bytes of a cell to the checksum depends on where the cell lies in a packet) we would expect Fletcher's checksum to be better than the TCP checksum at detecting the cell reordering that occurs in splices. We have no explanation for this result except to speculate it is related to the checksum's lack of sensitivity to zeros.

Table 5.2: Fletcher's Checksum Results
(256 Byte packets on systems at SICS)

system	code	splices	%
anhur.sics.se	P	16927890	1.2053800000
/home2	H	38931878	2.7722200000
11,978 files	X	664962344	47.3500000000
1,538,959 pkts	HX	663107668	47.2179000000
(5-11-95)	F	20427026	1.4545500000
	total	1404356806	100.0000500000
aten.sics.se	P	604624	0.1873100000
/home1	H	18436295	0.5711500000
40,799 files	X	1600416300	49.5804000000
3,550,000 pkts	HX	1595307193	49.4221000000
(5-11-95)	F	7718380	0.2391130000
	total	3227924408	100.0000730000
fafner.sics.se	P	57935874	1.8679900000
/opt	H	186056738	5.9989100000
26,896 files	X	1420744704	45.8082000000
3,400,000 pkts[4]	HX	1364400588	43.9915000000
(5-10-95)	F	72372832	2.3334700000
	total	3101510736	100.00007000

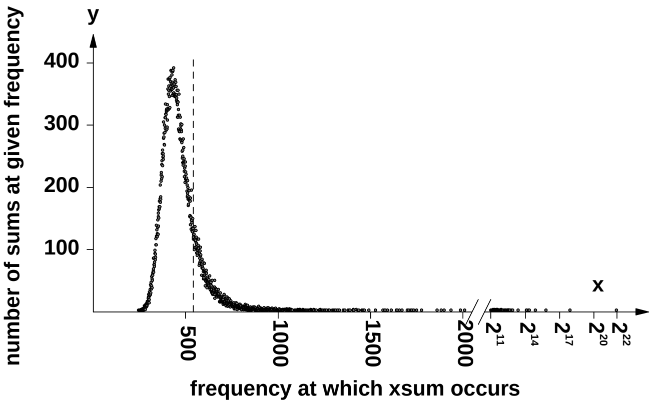

Figure 5.1: Frequencies for Fletcher's Checksum
Real Data (fafner.sics.se:/opt 48 bytes)
[Run of 1/30/95: 35,425,431 sums]

6. Observations

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

First, checksum distributions on modest amounts of real data are substantially different from the distributions one would anticipate for random data. Another (more controversial) way to express this point is that it does not seem to matter much which of the common CRCs or checksums one uses on small amounts of data, because the distribution of values in the data (and in particular, the frequency with which zeros occur) forces a skewed distribution.

Second, for the TCP checksum (and apparently Fletcher's checksum as well), the skewed distribution makes failure in the face of combined or reshuffled data more likely. In particular, if a router or host has a buffering problem that causes adjacent packets to be merged, the TCP checksum might fail 1% of the time rather than the 0.0015% of the time that purely random data distribution would suggest.

Third, 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.

Finally, there's a strong suggestion that the common practice of adjusting checksum fields to cause a packet or segment's checksum to sum to zero is a bad idea. If a very common type of packet is one that mixes a little non-zero data (such as headers) with a lot of zero data, why should we intentionally insert values into the checksum field such that the non-zero data sums to zero too?

7. Recommendations

While these scenarios may seem worrisome, 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. The chance of a splice is clearly less than the chance of an ATM cell loss and the chance of an ATM cell loss is often estimated at 1 in 10^8 or less. Furthermore, the ATM CRC will fail to detect a splice approximately at a rate of 1 in 2^{32}, so the chance of the TCP checksum being called upon to detect a splice is much less than 1 in $10^8 \times 2^{32}$ or less than one chance in 10^{17}.

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 even less effective error detection method than first thought, because real data is not random, and in particular, contains a lot of zeros.

As a preliminary recommendation, we also suggest that protocol designers consider avoiding the practice of adjusting the checksum fields so that the packet sums to zero and consider alternative approaches. We also suggest that protocol designers consider avoiding the practice of placing checksums in a protocol header, but instead append them as a trailer to the data being checksummed.

Acknowledgements

The authors would like to acknowledge the help of Chuck Kalmanek and Bill Marshall of Bell Labs, who discussed issues of study design. We also gratefully acknowledge the help of David Feldmeier of Bellcore and Lansing Sloan of Lawrence Livermore, who helped us with substantially faster CRC computation algorithms, and the Swedish Institute of Computer Science, which allowed us to use its filesystems and one of its fast multiprocessors for some of the test runs.

[3] This run completed only 2/3's of the file system due to a system failure.

References

1. J. Fletcher, ‘‘An Arithmetic Checksum for Serial Transmissions,’’ *IEEE Trans. on Communication*, Vol. 30, No. 1, January 1982, pp. 247-252.

2. Z. Wang and J. Crowcroft, ‘‘SEAL Detects Cell Misordering,’’ *IEEE Network Magazine*, Vol. 6, No. 4, July 1992, pp. 8-19.

3. D. Greene and B. Lyles, ‘‘Reliability of Adaptation Layers,’’ *Protocols for High-Speed Networks, III (Proc. IFIP 6.1/6.4 Workshop)*, ed. B. Pehrson, P. Gunningberg, and S. Pink, 1992.

4. J. Postel, ‘‘Transmission Control Protocol,’’ Internet Request for Comments No. 793, September 1981.

5. R. Braden, D. Borman, and C. Partridge, ‘‘Computing the Internet Checksum’’, Internet Request for Comments No. 1071, September 1988. (Updated by RFCs 1141 and 1624).

6. Joseph L. Hammond, Jr, et al., ‘‘Development of a Transmission Error Model and an Error Control Model,’’ Georgia Institute of Technology, prepared for Rome Air Development Center, May 1975.

7. William W. Plummer, ‘‘TCP Checksum Function Design,’’ Internet Engineering Note No. 45, June 1978. Reprinted in reference [5].

8. Anastase Nakassis, ‘‘Fletcher’s Error Detection Algorithm: How to implement it efficientlyy and how to avoid the most common pitfalls,’’ *ACM SIGCOMM Computer Communication Review*, Vol. 18, No. 5, October 1988, pp. 63-88.

9. Keith Sklower, ‘‘Improving the Efficiency of the OSI Checksum Calculation,’’ *ACM SIGCOMM Computer Communication Review*, Vol. 19, No. 5, October 1989, pp. 44-55.

Appendix: Notes on Tests

The major challenge in doing this work was getting the main test program to run fast enough. At the start of this effort, it took weeks to do a single data run, even when using 150 workstations in parallel at NSC, and the authors were searching for Cray time. However, after effort, we optimized the program so that a large data run now can be done in a day on a workstation and an evening on some faster machines.

The first version of the program was the obvious version. The program read blocks of data from the file system, and for each block, put a TCP/IP header on the front and broke the resulting packet into cell-sized chunks. Then, for each possible permutation of cells from two adjacent packets that could cause a valid splice, the program computed the TCP checksum and CRC for the data in the splice. The TCP checksum was computed by adding the precomputed checksums for each cell (making use of the TCP checksum’s additive properties), but the CRC was computed from scratch each time. Since computing the CRC in software is expensive, the program was quite slow. For 256 byte packets it could test about 16 million splices per CPU hour on a low end SparcStation, which meant over 270 CPU hours of computation were required to have a good chance finding a single CRC that failed to catch a splice.

The second version of the program was enhanced to compute the partial CRC for each cell and then use polynomial functions provided by Dave Feldmeier to combine the results. This version was substantially faster (about 20 million splices per CPU hour) but still had inefficiencies. The major performance bottleneck was that adding each cell’s CRC required a multiword multiplication (essentially multiplying the cell’s CRC by the cell’s offset from the end of the packet). This is inefficient because the same multiplication is done many times. For instance, consider that about half of all valid splices start with the first cell of the first packet and that a TCP/IP datagram with 256 bytes of data has 462 possible splices.

In the latest version (used to find the results for this paper) all partial results are computed once and stored in tables (including the value of the partial cell CRCs at each possible offset in a splice). The program simply sums the partial results and tests the results. This version tests about 110 million splices per CPU hour on a workstation. And on a Sun SC2000 we achieved 325 million splices per CPU hour per processor, which meant we could find a CRC failure every few hours.