The Effectiveness of Checksums for Embedded Networks

Theresa C. Maxino

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Electrical and Computer Engineering

Department of Electrical and Computer Engineering

Carnegie Mellon University

Pittsburgh, Pennsylvania, USA

May 2006

Advisor: Prof. Philip J. Koopman

Second Reader: Prof. Priya Narasimhan

Abstract

Embedded control networks commonly use checksums to detect data transmission errors. However, design decisions about which checksum to use are difficult because of a lack of information about the relative effectiveness of available options. We study the error detection effectiveness of the following commonly used checksum computations for embedded networks: exclusive or (XOR), two's complement addition, one's complement addition, Fletcher checksum, Adler checksum, and cyclic redundancy codes (CRC). A study of error detection capabilities for random independent bit errors and burst errors reveals that XOR, two's complement addition, and Adler checksums are suboptimal for typical application use. Instead, one's complement addition should be used for applications willing to sacrifice error detection effectiveness to reduce compute cost, Fletcher checksum for applications looking for a balance of error detection and compute cost, and CRCs for applications willing to pay a higher compute cost for further improved error detection.

Index Terms

Real-time communication, networking, embedded systems, checksums, error detection codes.

Acknowledgments

I would like to thank my academic advisor, Prof. Philip J. Koopman, for the invaluable insights, help, and guidance he has constantly provided me. Phil, thank you for showing me the ropes. Thanks are also due to the members of my research group, Elizabeth Latronico, Jennifer Morris, Kathryn Bergmann, and Justin Ray, for their insightful comments and critiques. Justin, thanks for the geeky discussions on CRCs and checksums in general. I would also like to thank Prof. Priya Narasimhan, Mark Adler, and the anonymous reviewers for their helpful comments and suggestions.

For being there when I needed them, I thank all my friends both here in Pittsburgh and scattered all over the world. Special thanks to Smriti Gupta, Chutika Udomsinn, Tudor Dumitras, Benedicto Cojuangco, and Francoise Adriano. You have all helped make the journey fun and worthwhile.

Very special thanks go to my mom and dad, Vicky and Gerry, my sisters, Clare and Cathie, Robert, my aunts, Cora and Teri, and the rest of my extended family for their constant love and support in all that I do. Your belief in me and my dreams has kept me going when I wanted to give up, and has made me want to be the best I can be.

Most of all, I would like to thank the Almighty for His eternal love, blessings, and guidance. Thank you, Lord, for giving me the grace and strength to accomplish all that I have to do.

This research was supported in part by a grant from Bombardier Transportation, the General Motors Collaborative Research Laboratory at Carnegie Mellon University, and the Pennsylvania Infrastructure Technology Alliance (PITA).

Portions of this thesis have been published as a Student Forum paper in the International Conference on Dependable Systems and Networks (DSN-2006).

CONTENTS

I Introduction 1

II Background and Related Work 2

III Effectiveness Evaluation 4

III-A Methodology 4

III-B XOR Checksum 6

III-C Two's Complement Addition Checksum 7

III-D One's Complement Addition Checksum 10

III-E One's Complement Fletcher Checksum 11

III-F Adler Checksum 13

III-G Cyclic Redundancy Codes (CRCs) 14

IV Revisiting Checksum Selection Choices 14

IV-A Effect of Data Value & Error Value Distributions on Effectiveness 15

IV-B Fletcher Checksum Compared to CRCs 18

IV-C One's Complement Fletcher checksum compared to Adler checksum 20

V Error Detection and Compute Cost Tradeoffs 21

VI Conclusions 23

References 24

Appendix I: Two's Complement Addition Checksum Formula Derivations 27

I-A All Zero Data 27

I-B All One Data 27

Appendix II: One's Complement Addition Checksum Formula Derivation 28

I. Introduction

A common way to ensure network message data integrity is to append a checksum. While it is well known that Cyclic Redundancy Codes (CRCs) are effective at error detection, a great many embedded networks employ less effective checksum approaches to reduce computational costs in highly constrained systems. (Even high-volume embedded applications cannot typically afford to have custom hardware built for CRC support.) Sometimes such cost/performance tradeoffs are justified. However, sometimes designers relinquish error detection effectiveness without gaining commensurate benefits in computational speed increase or memory footprint reduction.

Controller Area Network (CAN) [1], FlexRay [2], and TTP/C [3], among others, use a CRC to ensure data integrity. However, use of less capable checksum calculations abounds. One widely used alternative is the exclusive or (XOR) checksum, used by HART [4], Magellan [5], and many special-purpose embedded networks (e.g., [6], [7], [8]). Another widely used alternative is the two's complement addition checksum, used by XMODEM [9], Modbus ASCII [10], and some proprietary communication protocols (e.g., [11], [12]).

Beyond these options, ostensibly better alternate checksum approaches are commonly used in the non-embedded networking community. The non-embedded checksum examples we consider as alternatives are the one's complement addition checksum in the Transmission Control Protocol (TCP) [13], the Fletcher checksum used as a TCP alternate checksum [14], and the Adler checksum [15]. These alternate checksums would appear to offer potential improvements for embedded applications, but are not widely used in embedded applications yet.

Despite the fact that checksum techniques have been in widespread use for decades, there is surprisingly little information available about their relative performance and effectiveness. While using improved alternate checksums could achieve better error detection performance in new protocols, the amount of improvement possible is unclear from the existing literature. Because new embedded network protocols are continuously being created for various applications, it is worth knowing which checksum approaches work best.

This thesis examines the most commonly used checksum approaches in embedded networks, and evaluates their comparative error detection effectiveness as well as cost/performance tradeoff points. Checksums examined include: XOR, two's complement addition, one's complement addition, Fletcher checksum, Adler checksum, and CRC. (We use the term "checksum" loosely in describing the CRC, but this usage is consistent with common use of the generic term "checksum" to mean a function that

computes a Frame Check Sequence (FCS) value, regardless of the mathematics actually employed.) We describe checksum performance for random independent bit errors and burst errors. In addition, we describe the types of data and error patterns that are most problematic for each type of checksum based on examinations of both random and patterned data payloads.

Our results indicate that some common practices can be improved, and some published results are misleading. We conclude that one's complement addition checksums should be used for very lightweight checksums, that Fletcher checksums should be used instead of Adler checksums for most intermediate-complexity checksums, and that CRCs offer performance that provides much superior error detection to a Fletcher checksum for many embedded networks that can afford the computational cost.

Contributions. Concretely, the contributions of this thesis are (1) a quantitative comparison of the error detection effectiveness of the most common checksum algorithms, namely, XOR, two's complement addition, one's complement addition, Fletcher checksum, Adler checksum, and CRC, (2) formulas for computing the percentage of undetected two-bit errors for XOR, two's complement addition, and one's complement addition checksums, (3) causes of undetected errors for the various checksums, and (4) corrections of some common misconceptions regarding error detection effectiveness, Fletcher checksum, Adler checksum, and CRCs.

II. Background and Related Work

A checksum is an error detection mechanism that is created by "summing up" all the bytes or words in a data word to create a checksum value, often called a Frame Check Sequence (FCS) in networking applications. The checksum is appended to the data word (the message payload) and transmitted with it. Network receivers recompute the checksum of the received data word and compare it to the received checksum value. If the computed and received checksum match, then it is unlikely that the message suffered a transmission error. Of course it is possible that some pattern of altered bits in the transmitted message just happens to result in an erroneous data word matching the transmitted (and also potentially erroneous) checksum value. There is a tradeoff between the computing power used on the checksum calculation, size of the FCS field, and probability of such undetected errors.

Commonly used checksums generally fall into three general areas of cost/performance tradeoff. The simplest, and least effective, checksums involve a simple "sum" function across all bytes or words in a message. The three most commonly used simple "sum" functions are XOR, two's complement addition, and one's complement addition. These checksums provide fairly weak error detection coverage, but have very low computational cost. While error detection analysis of these checksums is not terribly difficult, we

have not found a complete descriptive treatment of their error detection effectiveness in the literature. [16] provides an analytic comparison of their error detection effectiveness, but does not provide quantitative data.

The most expensive commonly used checksum is a CRC. Strictly speaking a CRC is not a sum, but rather an error detection code computed using polynomial division. CRC computation can be a significant CPU load, especially for very small processors typical of many embedded systems.

Because CRC computation is so expensive, two intermediate-cost checksums have been proposed for use. The Fletcher checksum [17] and the later Adler checksum [15] are both designed to give error detection properties almost as good as CRCs with significantly reduced computational cost. In the late 1980s, Nakassis [18] and Sklower [19] published efficiency improvements for Fletcher checksum implementations. While Fletcher and Adler checksum error detection properties are almost as good as a relatively weak CRC, they are far worse than good CRCs for some important situations.

Although the checksums we consider have been known for years or decades, relatively little is published about checksum error detection performance. Fletcher published error detection information in his original paper [17]. [16] and [20], taken together, present an analytic comparison of all the checksums we examine. Early CRC effectiveness studies were lists of optimal CRC polynomials for specific lengths (e.g., [21][22][23]).

Stone, et al. [24][25][26] measured network checksum effectiveness of one's complement addition checksum, Fletcher checksum, and CRC. In their study, they found that one's complement addition checksum and Fletcher checksum had much higher probabilities of undetected errors than they expected, and the reason they gave was non-uniformity of network data.

Koopman [27][28] investigated the CRC polynomials currently in use and proposed alternatives which provided better performance. Those papers proposed a polynomial selection process for embedded networks and determined the optimum bounds for CRC-3 to CRC-16 for data words up to 2048 bits. We use these results as the source of our CRC data.

McAuley [29] has proposed the Weighted Sum Codes (WSC) algorithm as an alternative to the Fletcher checksum and CRC. Feldmeier [30] conducted a comparison of WSC against one's complement addition checksum, XOR checksum, block parity, Fletcher checksum, and CRC. He asserted that WSC is as good as CRC in error detection and as fast as Fletcher checksum in compute speed. Because WSCs are not commonly used, we leave an examination of that checksum to future work.

In the balance of this thesis we describe several algorithms for computing checksums in order of increasing computational cost, from the XOR checksum to CRCs. We have evaluated error detection

effectiveness via a combination of analysis and simulation results. Moreover, we give insight into the strengths and weaknesses of each checksum approach, with an emphasis on particular vulnerabilities to undetected errors based on data values and bit error patterns. We describe inaccuracies in published claims or commonly held beliefs about the relative effectiveness of checksums, and examine the cost effectiveness of various alternatives. We also confirm some cases in which commonly used checksum approaches are good choices.

III. Effectiveness Evaluation

A. Methodology

To carry out the experiments in this thesis, we implemented the various checksum algorithms in C++. All experiments were carried out on Pentium machines running the Windows operating system. Evaluation of the performance of each checksum algorithm was performed via simulated fault injection. Each experiment consisted of generating a message payload (data word) with a specific data value, then computing a checksum across that data word. The resultant code word (data word plus checksum) was subjected to a specific number of bit inversion faults, simulating bit inversion errors during transmission. The checksum of the faulty data word was then computed and compared against the (potentially also faulty) FCS value of the faulty codeword. If the FCS value of the faulty code word matched the checksum computed across the faulty data word, that particular set of bit inversions was undetected by the checksum algorithm used. Identical data word values were used for all checksums, except for CRCs which are known to be data-independent [27] (data word values do not affect error detection performance). The data used in each experiment varied, including random data as well as all zeros, all ones, and repeated data patterns.

Burst error experiments were conducted in a similar manner to the bit error experiments except that instead of subjecting the resultant code word to bit inversions, the code word was subjected to a specific number of contiguous-bit burst errors. Non-contiguous-bit burst error experiments were not conducted because these can be thought of as special cases of the bit error experiments.

To simplify comparisons, only data word lengths that were multiples of the checksum size were used in this study, as is common in real applications. We refer to chunks of the data word the size of the checksum as blocks. For example, for a two's complement addition checksum, a 48-bit data word used with a 16-bit checksum would result in a computation that divides the 48-bit data word into three 16-bit blocks that are added to compute the 16-bit checksum, forming a $48 + 16 = 64$ bit code word.

The Hamming Distance (HD) of a checksum is the smallest number of bit errors for which there is at least one undetected case. For example, a CRC with HD=4 would detect all possible 1-, 2-, and

3-bit errors, but would fail to detect at least one 4-bit error out of all possible 4-bit errors. With the assumption of random independent bit errors in a binary symmetric channel, the main contributing factor to checksum effectiveness for most embedded applications is the fraction of undetected errors at the Hamming Distance, because the probability of more errors occurring is significantly less likely (e.g., 3-bit errors are approximately a million times more common than 4-bit errors assuming a random independent Bit Error Rate (BER) of 10^{-6}). Thus, analysis of undetected errors at the HD is performed to give insight into experimental results.

For bit-error experiments, the faults injected in each experiment were either all possible 1-, 2-, or 3-bit errors in the code word for each data word value examined. Where necessary, experiments with 4-bit errors in the code word were also conducted. The total number of undetected errors for each particular experiment was then noted. At least ten trials were made for each type of experiment and the mean for all the trials was obtained. For example, ten experiments were performed where all possible 2-bit errors were injected into a (504+8)-bit code word with random data and an 8-bit checksum size. (For experiments where the ratio of the standard deviation to the mean was greater than five percent, we performed 100 trials. We determined the number 100 by determining the point at which the standard deviation to mean ratio reached its asymptotic value.)

Some of the graphs in this thesis show the percentage of undetected errors (Pct_{ud}) which is the ratio of undetected errors with respect to the total number of all possible errors for that particular bit-error degree. Other graphs show the probability of undetected errors (P_{ud}) for the HD at each code word length. P_{ud} is different because it uses the Hamming weight (HW) which is the number of undetected errors at the HD of the checksum at that particular code word length, and multiplies this by a given BER. P_{ud} is often more useful than just Pct_{ud} because it takes into account the BER and permits comparison of codes with different HDs. P_{ud} can be computed as follows

$$P_{ud} = HW(BER^{x})(1 - BER)^{n-x}$$

where n is the code word length in bits, and x is the number of random independent bit errors.

For burst error experiments, contiguous-bit burst errors starting from two bits up to at least $(k+1)$ bits (k=checksum size) were injected in the code word for each data word value examined. For a particular burst-error degree, all possible contiguous-bit burst error values were injected. Using the same example as above, all possible contiguous 9-bit burst errors were injected into a (504+8)-bit code word with random data and an 8-bit checksum size. Most experiments were stopped as soon as there was one undetected burst error for that burst-error degree. The exceptions to this were the experiments for Fletcher and Adler

checksums. We performed experiments up to $k+1$ contiguous bit bursts even though there were already undetected burst errors at $k/2$ contiguous bits (k=checksum size).

In some instances, we found it useful to perform comparisons of the probability of undetected errors to the commonly held notion of checksum effectiveness being approximately equal to $1/2^k$, where k is the checksum size in bits. The intuitive argument for this is that because there are 2^k possible checksum values for a k-bit checksum, given more or less random data and random corruption, there is a $1/2^k$ chance of the FCS just happening to match the corrupted data values by chance (e.g., a one-in-256 probability of undetected error for an 8-bit checksum).

B. XOR Checksum

Exclusive Or (XOR) checksums are computed by XOR'ing blocks of the data word together. The order in which blocks are processed does not affect the checksum value. One can think of an XOR checksum as a parity computation performed in parallel across each bit position of data blocks (bit i of the checksum is the parity of all block bits i, e.g., bit 3 of the checksum is the parity of bit 3 of all blocks).

The XOR checksum is data-independent (error detection performance is not affected by data word values). Because it is a parity computation, the XOR checksum has a Hamming Distance (HD) of 2, detecting all one-bit errors, but not some two-bit errors. In particular, it fails to detect any even number of bit errors that occur in the same bit position of the checksum computational block. It detects any bit error pattern that results in an odd number of errors in at least one bit position, which includes all situations in which the total number of bit errors is odd. It also detects all burst errors up to k bits in length (k equal to the checksum size), because two bits must align in the same position within a block to become undetected. Burst errors greater than k bits in length are detectable if they result in an odd number of actual bits being inverted or if the inverted bits do not align in the same bit position in the affected blocks.

For block size of k and checksum size k, in every pair of data blocks there are exactly k possible undetected two-bit errors (one undetected two-bit error for each bit of the block, in which errors happen to occur to the same bit position in the two blocks). For an n-bit code word, we multiply by the number of combinations of k-size blocks in the code word taken two at a time. Thus, the number of undetected two-bit errors is

$$k * \begin{pmatrix} n/k \\ 2 \end{pmatrix} = \frac{k(n/k)((n-k)/k)}{2} = \frac{n(n-k)}{2k}$$

The total number of possible two-bit errors for an n-bit code word is

$$\binom{n}{2} = \frac{n(n-1)}{2}$$

Dividing the number of undetected two-bit errors by the total number of two-bit errors gives us the fraction of undetected two-bit errors as

$$\frac{n-k}{k(n-1)}$$

where n is the code word length in bits and k is the checksum size in bits.

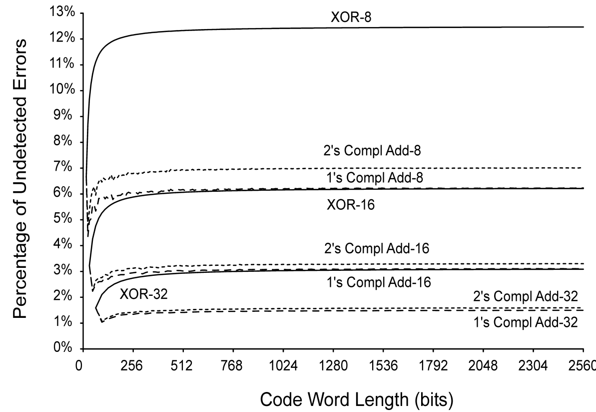

Fig. 1. Percentage of undetected errors over total number of two-bit errors for 8-, 16-, and 32-bit XOR, two's complement addition, and one's complement addition checksums. Two's complement addition and one's complement addition data values are the mean of 10 trials using random data.

From the above equation, we can see that as n approaches infinity, the percentage of undetected two-bit errors becomes approximately $1/k$ (k being checksum size), which is rather poor performance for a checksum (i.e., 12.5% undetected errors for an 8-bit checksum, and 3.125% for a 32-bit checksum). Simulation results confirm this analysis. (See Figure 1.) The XOR checksum has the highest probability of undetected errors for all checksum algorithms in this study, and is not as effective as addition-based checksums for general purpose error detection uses.

C. Two's Complement Addition Checksum

The two's complement addition checksum ("add checksum" for short) is obtained by performing an integer two's complement addition of all blocks in the data word. Carry-outs of the accumulated sum are discarded, as in ordinary single-precision integer addition. The order in which blocks are processed does

not affect the checksum value. The add checksum is data-dependent, with the probability of undetected errors varying with the data word value. The add checksum detects all one-bit errors in the code word, and has a Hamming Distance of 2 for all code word lengths.

An add checksum can be thought of as an improvement of XOR checksums in that bit "mixing" between bit positions of the data blocks is accomplished via bit-by-bit carry-outs of the binary addition. The effectiveness of mixing depends on the data being added, which determines the pattern of carry bits across various bit positions.

A significant cause of undetected errors is when a pair of bit errors in different data blocks line up at the same bit position within the blocks, and the data in those bit positions contains a 1 in one block and a 0 in the other block. The resultant erroneous data blocks have a 0 in the first block and a 1 in the other block, resulting in the same sum.

A second important source of undetected errors is when the most significant bit (MSB) positions of any two data blocks are inverted, regardless of value. This type of error is undetected because the sum remains the same and carry-out information from that position is lost during the computation, making it impossible to detect a pair of ones changed to zeros or a pair of zeros changed to ones in the MSB.

A third source of undetected errors is a non-carry-generating bit being inverted in the data word and the bit in the corresponding bit position in the checksum also being inverted.

Because data-dependent error detection vulnerabilities involve a concurrent inversion of 1 and 0 bits in the same bit position, the add checksum performs worst when each bit position has an equal number of zeros and ones. For this reason, random data gives very nearly the worst case for undetected errors because it tends to have the same number of zeros and ones in each bit position. Given that random data is often used to evaluate checksums, but real data sent in network messages often has a strong bias toward zeros due to unused data fields, random data evaluation of add checksums can be considered pessimistic for many cases. The add checksum performs best when data is all ones or all zeros, because inverting a pair of identical bits causes a carry bit effect that is readily detected.

Even for worst case data, as can be seen from Figure 1, add checksum is almost twice as effective as the XOR checksum for long data words. This is because the primary cause of undetected errors is inverted bits that are both differing and in the same bit position, whereas XOR undetected errors also occur for bit values that don't necessarily differ.

For worst case data, add checksum has an undetected two-bit error percentage approximately equal to $(k+1)/2k^2$, where k is the checksum size. This equation can be arrived at by adding together the undetected error percentages for each bit position. The MSB bit has an undetected error percentage equal

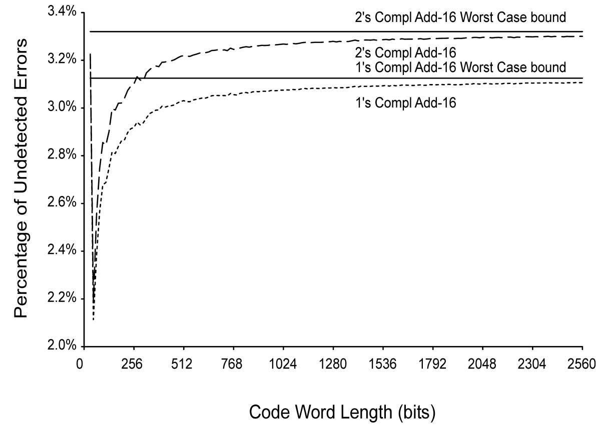

Fig. 2. Percentage of undetected two-bit errors for 16-bit two's complement addition and one's complement addition checksums. Data points for both checksums are the mean of 100 trials using random data. Worst case bound is from the given formula.

to that of XOR, $1/k$. All the other bits have an undetected error percentage half of XOR $(1/2k)$ because only 0-1 and 1-0 error combinations will be undetected. Multipying the two ratios by the number of bits in each block and then adding them together gives us

$$\frac{1}{k}\left(\frac{1}{k}\right)+\frac{1}{2k}\left(\frac{k-1}{k}\right)=\frac{1}{k^2}+\frac{k-1}{2k^2}=\frac{k+1}{2k^2}$$

However, this equation is just an approximation because it does not take into account the third source of undetected errors mentioned previously nor the fact that some of the 0-1 and 1-0 error combinations will be detectable due to carry-bit generation. It is a useful approximation, however, and can be thought of as a bound, as can be seen in Figure 2.

For long data words with all zero or all one data, add checksum asymptotically fails to detect approximately $1/k^2$ of two-bit errors, where k is the checksum size in bits. (See Appendix I for formula derivations.)

Figure 3 shows simulation results for exactly identical numbers of zero and one data (alternating 0xFF and 0x00 values), all zeros, and all ones. Randomly generated data word values were very close to the worst case as expected, and are omitted from the figure.

The add checksum detects all burst errors up to k bits in length where k is the checksum size. Burst errors greater than k bits may or may not be detected depending on the number of bits inverted and their bit positions. The same reasons for undetected bit errors apply to burst errors. Thus, if a burst error greater than k bits occurs but the inverted bits do not have the same bit positions or otherwise do not fall into any of the three categories of undetected errors mentioned earlier, then it is unlikely that the burst

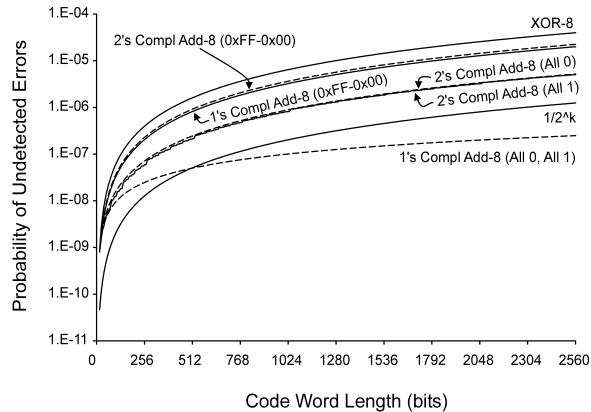

Fig. 3. Probability of undetected errors for 8-bit checksums using different data and a BER of 10^{-5}

error will go undetected.

D. One's Complement Addition Checksum

The one's complement addition checksum is obtained by performing an integer one's complement addition of all blocks of the data word. One's complement addition can be performed on two's complement hardware by "wrapping around" the carry-out of the addition operation back into the checksum. In particular, if adding a block to the running checksum total results in a carry-out, then the running checksum is incremented. Speed optimizations are known for hardware that does not support carry bits (e.g., [13]). The order in which blocks are processed does not affect the checksum value.

The main performance difference between one's complement and two's complement addition checksums is in the error detection capability of bit inversions affecting the MSB of blocks. Because the carry-out information of the MSB is preserved via being wrapped around and added back into the least significant bit, bit inversions that affect a pair of ones or a pair of zeros in the MSB are detected by one's complement addition checksums, but are undetected by two's complement addition checksums. ([16] gives a similar explanation.) One's complement addition checksums detect all burst errors up to $k-1$ bits in length, where k is the checksum size in bits. Some k-bit burst errors are undetectable because of the wrap-around of carry-outs back into the low bits of the checksum. Burst errors greater than k bits in length will be undetectable if they fall into any of the categories of undetectable errors previously described. Other than the error detection performance for MSB bits, the behavior of one's and two's complement addition checksums is identical, with one's complement addition checksum having a slightly lower probability of undetected errors for random independent bit errors (Figure 3).

At asymptotic lengths, the probability of undetected errors for all 0 and all 1 data approaches $2/n$ where n is the code word length in bits. (See Appendix II for formula derivation.)

For worst case data at asymptotic lengths, approximately $1/2k$ of all possible two-bit errors are detected. This is half of the ratio of undetected errors for XOR checksum. The intuitive logic behind this is that for each bit position only 0-1 and 1-0 error combinations will be undetected, unlike in XOR checksum where 0-0 and 1-1 error combinations are also undetectable. Looking at Figure 1, it can be seen that one's complement addition checksum is as good as XOR checksum at half the checksum size for random independent bit errors on random data.

E. One's Complement Fletcher Checksum

The Fletcher checksum [17][14] is only defined for 16-bit and 32-bit checksums, but in principle could be computed for any block size with an even number of bits. We use the one's complement addition version, which provides better error detection than the two's complement addition version [18]. We confirmed this experimentally. (Throughout this thesis "Fletcher checksum" means "one's complement addition Fletcher checksum.")

A Fletcher checksum is computed with a block size j that is half the checksum size k (for example, a 32-bit Fletcher checksum is computed with a block size of 16 bits across the data word, yielding a 32-bit checksum value). The algorithm used to compute the checksum iterating across a set of blocks D_0 to D_n is:

Initial values: $sumA = sumB = 0;$

For increasing i : { $sumA = sumA + D_i;$

$sumB = sumB + sumA;$ }

$sumA$ and $sumB$ are both computed using the same block size j. The resulting checksum is $sumB$ concatenated with $sumA$ to form a checksum that is twice the block size. The accumulation of $sumB$ makes the checksum sensitive to the order in which blocks are processed.

Fletcher checksum error detection properties are data-dependent. As with addition-based checksums, the highest probability of undetected error occurs when data in each bit position of the blocks is equally divided between zeros and ones. Random data word values also give approximately worst case error detection performance due to a relatively equal distribution of zeros and ones in each bit position. When data is all zero, the only undetected error is one in which all bits in a single block are changed from zeros to ones. (Recall that 0xFF also represents zero in 8-bit one's complement notation.)

The Fletcher checksum can detect all burst errors that are less than j (or $k/2$) bits long, where j is the block size which is half the checksum size k. As expected, it is vulnerable to burst errors that invert bits

in a block from all zero to all one or vice versa (the Adler checksum has the same vulnerability). Our experiments have verified that excluding this special type of burst errors, Fletcher checksum can detect all burst errors less than k bits, where k is the checksum size. [17] gives a more detailed explanation of burst error detection properties.

The Fletcher checksum has HD=3 up to a certain, modulo-dependent, code word length and HD=2 for all remaining code word lengths. We have confirmed experimentally that two-bit errors are detected for data word lengths less than $(2^{k/2} - 1) * (k/2)$ bits where k is the checksum size and $(2^{k/2} - 1)$ is equal to the Fletcher checksum modulus. [17] states further that all two-bit errors are detected provided that they are separated by fewer than $(2^{k/2} - 1) * (k/2)$ bits, k being the checksum size.

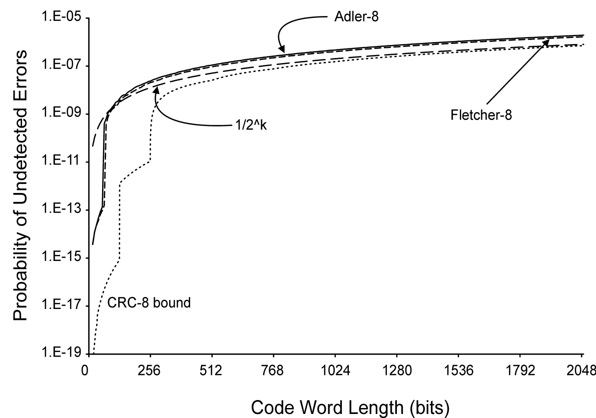

Fig. 4. Probability of undetected errors for 8-bit Fletcher and Adler checksums using random data and BER of 10^{-5}. Data values for both Fletcher and Adler checksums are the mean of 10 trials. CRC-8 bound values are optimal values for all 8-bit CRC polynomials.

We have also confirmed experimentally that an 8-bit Fletcher checksum has HD=2 for code word lengths of 68 bits and above, with HD=3 below that length. A 16-bit Fletcher checksum has HD=2 starting at 2056-bit codeword lengths. According to the equation, a 32-bit Fletcher checksum is expected to have HD=2 starting at a code word length of 1,048,592 bits.

Figure 4 shows the probability of undetected errors of the Fletcher checksum. In general, Fletcher checksums are significantly worse than CRCs, even when both achieve the same HD. In particular, Fletcher checksums have a significantly higher probability of undetected error than $1/2^k$ for long lengths, while CRCs typically have a slightly lower probability of undetected error than $1/2^k$, k being checksum size. The significance of $1/2^k$ here is that it is a commonly held notion that all checksums have the same error detection effectiveness equal to $1/2^k$ where k is the checksum size. Further discussion of Fletcher

checksum performance is in Section IV-B. The CRC bound shown in the figure is the lowest probability of undetected errors of any 8-bit CRC. Of all the checksum algorithms in this study, Fletcher checksum has the next best overall error detection capability after CRC except for the special case of 16-bit Adler checksum at short lengths.

F. Adler Checksum

The Adler checksum [15] is only defined for 32-bit checksums, but in principle could be computed for any block size with an even number of bits. The Adler checksum is similar to the Fletcher checksum and can be thought of in the following way. By using one's complement addition, the Fletcher checksum is performing integer addition modulo 255 for 8-bit blocks, and modulo 65535 for 16-bit blocks. The Adler checksum instead uses a prime modulus in an attempt to get better mixing of the checksum bits. The algorithm is identical to the Fletcher algorithm, except sumA is initialized to 1 and each addition is done modulo 65521 (for 32-bit Adler checksum) instead of modulo 65535. As with a Fletcher checksum, the result is sensitive to the order in which blocks are processed.

Although the Adler checksum is not officially defined for other data word lengths, we used the largest prime integers less than $2^4 = 16$ and less than $2^8 = 256$ to implement 8- and 16-bit Adler checksums for comparison purposes. Because the algorithm is similar to that for Fletcher checksums, Adler checksums have similar performance properties. (See Figure 4.) We have confirmed experimentally that two-bit errors are detected for data word lengths less than $M * (k/2)$ bits where k is the checksum size and M is equal to the Adler checksum modulus. Our experiments show that Adler-8 has HD=3 below 60 bits (using modulo 13 sums), and that Adler-16 has HD=3 below 2024 bits (using modulo 251 sums). From the equation, Adler-32 is expected to have HD=3 below 1,048,368 bits. For code word lengths greater than those mentioned above, Adler checksum has HD=2. As with Fletcher checksums, the worst case for undetected error probability is with an equal number of zeros and ones in each data block bit position, meaning that random data has nearly worst-case undetected error performance.

Adler-8 and Adler-16 can detect all burst errors that are less than j (or $k/2$) bits long where j is the block size which is equal to half the checksum size k. Adler-32 detects all burst errors up to 7-bits long. ([15] defines Adler-32 blocks to be 1 byte or 8 bits wide with 16-bit running sums so $j = 8$ for Adler-32.) Excluding burst errors that change data in the data blocks from all zero to all one or vice-versa, all burst errors less than $k-1$ are detected. This is one bit less than Fletcher checksum, which is unexpected. ([20] states that Adler checksum has a higher probability of undetected burst errors than Fletcher checksum but does not explicitly state that burst error detection is one bit less.) The reason for this is that Adler

checksums use a prime modulo which is less than $2^k - 1$ whereas Fletcher checksums use a modulo equal to $2^k - 1$, k being checksum size. A comparison of Fletcher and Adler checksum performance is given in Section IV-C.

G. Cyclic Redundancy Codes (CRCs)

The simplest version of a CRC computation uses a shift-and-conditional-XOR approach to compute a checksum [31]. Faster methods of computation are available (e.g., [31][32][33] based on complete or partial lookup tables), but are still slower than the other checksum techniques discussed. The selection of a good generator polynomial is crucial to obtaining good error detection properties, and is discussed in [28].

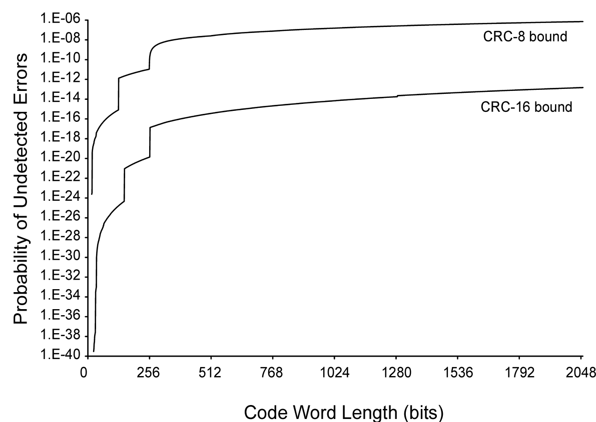

Fig. 5. Bounds for probability of undetected errors for all 8-bit CRCs and all 16-bit CRCs (BER = 10^{-5})

All CRCs are data-independent and have a Hamming Distance of at least 2 for all code word lengths. Most polynomials have HD=3 or higher for some lengths less than 2^k where k is the checksum size. Also, some polynomials detect all odd-bit errors, at the expense of worse error detection ability for even-bit errors. All burst errors are detected up to k bits in length where k is the checksum size in bits. Figure 5 shows the bounds for the lowest probability of undetected errors for 8-bit CRCs and 16-bit CRCs.

IV. Revisiting Checksum Selection Choices

In this section we examine some published and folk wisdom misconceptions about checksum performance in light of our experiments. Of course there will be individuals who do not hold, and publications which do not contain these misconceptions, but we have observed these issues to arise often enough that they warrant specific attention.

A. *Effect of Data Value & Error Value Distributions on Effectiveness*

When data is uniformly distributed, it is common for an assumption to be made that all checksum algorithms have the same probability of undetected errors (P_{ud}) of $1/2^k$, where k is the checksum size in bits. The intuitive argument is that because there are 2^k possible checksum values for a k-bit checksum, given more or less random data and random corruption, there is a $1/2^k$ chance of the FCS just happening to match the corrupted data values by chance (e.g., a one-in-256 probability of undetected error for an 8-bit checksum). While this is true for completely random data and corruption, most data isn't really random, and neither do many types of corruption result in total random data scrambling. More often, checksum effectiveness is controlled by the limiting case of patterned data, and corruption that is patterned or only affects a few bits.

As an example, Stone et al. [25] studied the behavior of one's complement addition checksum (which they examined in the context of its use as the TCP checksum), Fletcher checksum, and CRC across real network data. They observed that one's complement addition checksum and Fletcher checksum had a P_{ud} which was far worse than the value of $1/2^k$ that they expected. They theorized that the reason for this was because of the non-uniform distribution of the data they were using, and in their Corollary 8 claim that if data had been uniformly distributed, the IP and Fletcher checksums would have been equivalently powerful. [20] furthers this point of view in its analysis. While this may be true, the explanation is not necessarily useful in predicting the effectiveness of checksums operating on non-random data. Moreover, we think that it is also important to consider the effectiveness of checksums at detecting small numbers of corrupted bits even on data with randomly distributed values such as encrypted data or compressed data (e.g., streaming video).

We have found it productive to look at the effectiveness of a checksum in terms of how much the checksum value varies based on relatively small changes to the data value used as input. One way of looking at this is evaluating the effectiveness of a checksum computation in terms of its effectiveness as a pseudo-random number generator. The better the generator, the more likely that multiple bits of the output will be affected by even a single bit change in input value. An XOR checksum, for example, changes only one bit of computed FCS value for a one bit change in data value. One's complement and twos' complement addition change only one bit of FCS value for a single bit of data value changed in the worst case (when there are no carries changed) and many bits of data value in the best case. Fletcher and Adler checksums typically change several bits in the FCS for a single bit data value change, with the changes more pronounced in the high half of the FCS. A single bit change in the data value for a

CRC in typical cases has the potential to affect all the bits of the FCS.

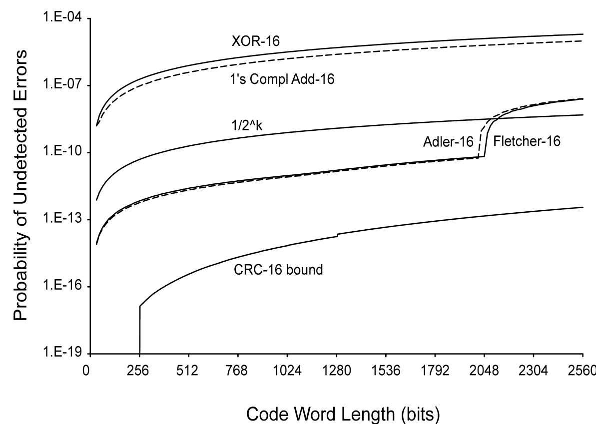

Fig. 6. Performance of 16-bit checksums with random data and random independent bit errors (BER of 10^{-5}). Data values for one's complement addition, Fletcher, and Adler checksums are the mean of 10 trials.

The results of effectiveness can be seen when examining performance for small numbers of bit errors. Figure 6 shows different checksum algorithms applied to the same uniformly distributed random data. The graph clearly shows that P_{ud} is dependent on the algorithm used. XOR and one's complement addition checksums perform the worst, while CRC performs the best. (Two's complement addition checksum is not included here for clarity of presentation.) This is a result of the different effectiveness of checksums at generating different outputs for small changes in input. Fletcher, Adler, and CRC algorithms attain better than $1/2^k$ for short messages due to their mathematical properties, but only a good CRC does better than $1/2^k$ for all data lengths.

The worst case for a weak checksum algorithm is a small number of bit errors that don't mix the results very much. As the number of bit errors in a single code word increases, then all checksums converge to the $1/2^k$ limit value, making the choice of checksum algorithm moot. So, to the degree that corruption of data does not result in totally random erroneous data, the selection of checksum algorithms is important.

P_{ud} is further influenced by the data word content when data-dependent checksums such as add, one's complement addition, Fletcher, and Adler are used. Data-dependent here means that P_{ud} for these checksums varies depending on the data word content, unlike in XOR checksum and CRC where P_{ud} remains the same regardless of data word content. The percentage of undetected errors is least when the data is all zero or all one. The percentage increases when the number of zeros and ones in each bit position in the data is more equal. In view of this, the highest percentage of undetected errors usually occurs for random data having an even number of ones and zeros in every bit position.

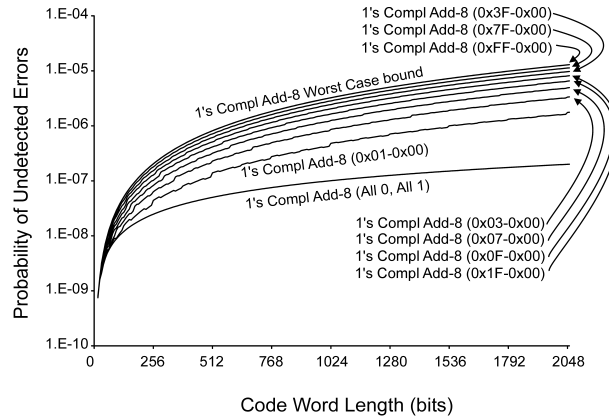

Fig. 7. Probability of undetected errors for 8-bit one's complement add checksum (BER of 10^{-5}) using different data patterns. Data patterns were of the form XX-00 where XX was varied from 00 to 0xFF.

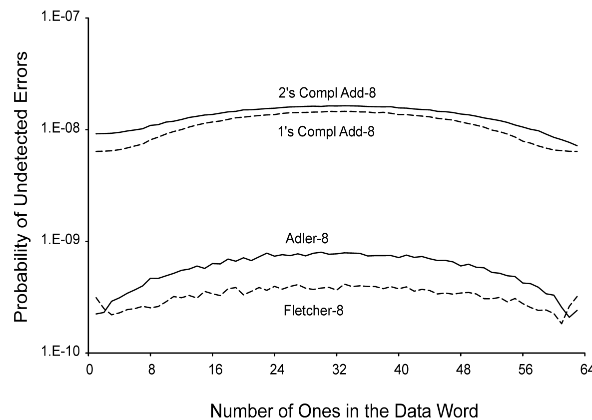

Fig. 8. Probability of undetected errors in random data for a 64-bit data word using 8-bit checksums and a BER of 10^{-5}. Data values are the mean of 100 trials.

Figure 7 shows this effect for one's complement add checksum. In this experiment, the different values were alternated with 0x00 bytes. In the figure, the effect of increasing the number of bit positions where there is an equal number of ones and zeros can be clearly seen. It can also be noted that the worst case bound coincides with the line for the 0xFF-0x00 data pattern. Figure 8 shows the resulting P_{ud} when the number of randomly-placed ones in the data word is increased from 1 to 63 in a 64-bit data word message. The probability of undetected errors increases from when data is 100% zeros or ones to when data is 50% zeros and 50% ones in every bit position in the data word.

B. Fletcher Checksum Compared to CRCs

Some previous work (e.g., [17][15]) has claimed that Fletcher checksum and Adler checksum are comparable to CRC in error detection capabilities. However, in all cases CRC is substantially better at the same HD, and in many important cases achieves better HD. [20] computed the undetected error probabilities for some CRC-32 polynomials and Fletcher-32 and Adler-32 checksums for a length of 8KB or 65536 bits. Their results show that CRC outperforms both Fletcher-32 and Adler-32.

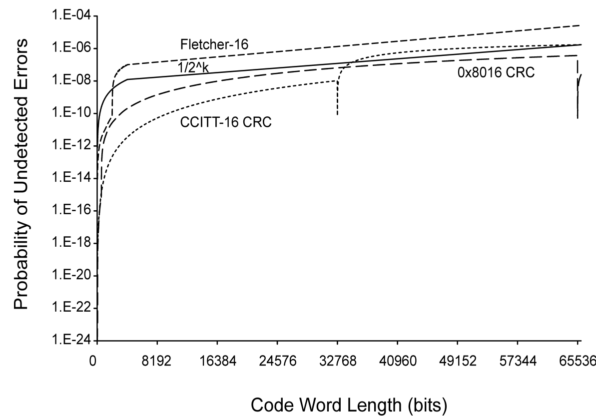

Fig. 9. Probability of undetected errors for 16-bit Fletcher checksum and two 16-bit CRCs at a BER of 10^{-5}. Data values for Fletcher checksum are the mean of 10 trials using random data.

[17] states that for asymptotically long code words, the one's complement addition Fletcher checksum detects $1/(2^{k/2}-1)^2$ of all possible errors, which is only slightly less than $1/2^k$ (k = checksum size). "All possible errors" here seems to mean all bit errors regardless of number, and all burst errors regardless of length - in effect, this is similar to the random data argument (P_{ud} is always equal to $1/2^k$) we mentioned earlier under which any checksum performs about that well.

Most embedded systems have code word lengths that are much shorter than 2^k bits, k being checksum size. Even the Ethernet Maximum Transmission Unit (MTU), for example, is only 1500 bytes (or 12,000 bits) which is much shorter than 65535 bits. At these lengths CRCs often have distinct advantages. By using Fletcher and Adler checksums at these lengths, error detection capability is considerably suboptimal - at least one bit of HD in error detection capability is effectively being given up.

Figure 9 shows a comparison of Fletcher checksum to two CRC polynomials, the common CCITT-16 and the 0x8016 polynomial which performs better than CCITT-16 starting at 32Kbits where it has HD=3 compared to CCITT-16's HD=2. For short code word lengths, the Fletcher checksum is at least 1 bit of HD worse than CRCs. The dips in the figure mark the points where CCITT-16 changes from HD=4 to

HD=2 at 32Kbits, and where 0x8016 changes from HD=3 to HD=2 at 64Kbits.

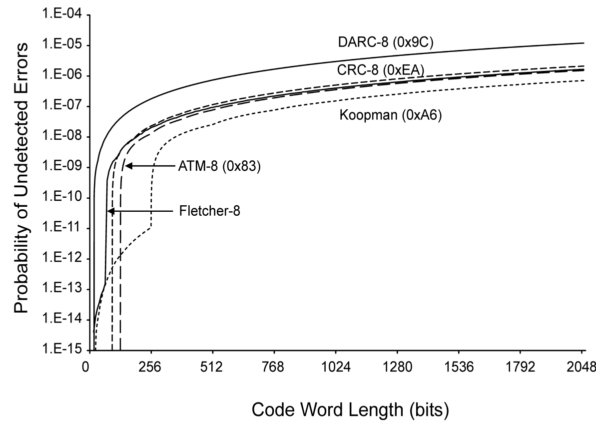

Fig. 10. Probability of undetected errors for some common CRCs and Fletcher-8 at a BER of 10^{-5}. Data values for Fletcher-8 are the mean of 10 trials using random data.

We would like to reiterate what was said in [28], that the selection of the best CRC polynomial for the desired checksum size is of utmost importance. Figure 10 shows that incorrect selection of a CRC polynomial can result in worse error detection perfromance than the Fletcher checksum. Applications that use DARC-8 would be better off using Fletcher-8, while, applications that use CRC-8 and ATM-8 but don't use code words shorter than 128 bits would get comparable performance to Fletcher-8. However, polynomial 0xA6 would be a better choice for applications that want to use an 8-bit CRC above a 128-bit data word length.

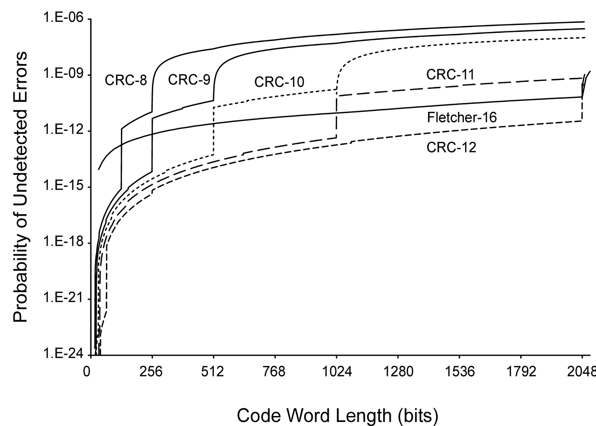

Fig. 11. Probability of undetected errors for Fletcher-16 and CRC bounds for different CRC widths at a BER of 10^{-5}. Data values for Fletcher-16 are the mean of 10 trials using random data.

Comparison of Fletcher checksum effectiveness to CRCs was also performed on CRC checksum sizes less than the Fletcher checksum size (see Figure 11). Optimal CRC bounds from 8-bit CRC to 12-bit CRC were plotted against the 16-bit Fletcher checksum. The resulting graph shows that it is possible for an optimal CRC polynomial with a smaller checksum size to outperform the Fletcher checksum. CRC-8, CRC-9, and CRC-10 all perform better than Fletcher-16 for code word lengths less than 128, 256, and 512 bits, respectively. CRC-11 performs better than Fletcher-16 for code word lengths less than 1024 bits, performs worse for greater than 1024 but less than 2048 bits, and then performs comparably for lengths greater than 2048 bits. CRC-12 consistently outperforms Fletcher-16 for all code word lengths. Optimal CRCs with more than 12 bits will perform better than Fletcher-16, and thus, are omitted from the graph.

C. One's Complement Fletcher checksum compared to Adler checksum

The Adler checksum has been put forward as an improvement of Fletcher checksum [15], and it is commonly believed that the Adler checksum is unconditionally superior to the Fletcher checksum (e.g., [34][35]). (In private communication, Mark Adler stated that what [15] meant was that Adler-32 is an improvement over Fletcher-16. At that time, he was not aware of Fletcher-32, but this point is not widely known and is not apparent in [15].)

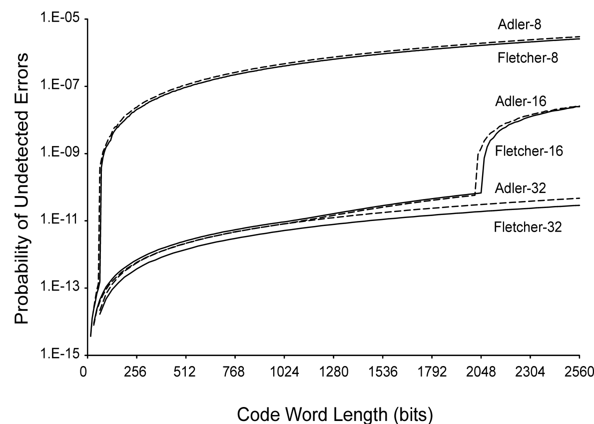

Fig. 12. Comparison of 8-, 16-, and 32-bit Fletcher and Adler checksums using random data at a BER of 10^{-5}. Data point values are the mean of 10 trials.

The better mixing of bits that Adler checksum provides due to its prime modulus has been claimed to provide better error detection capabilities than Fletcher checksum. We have found that this is often not

the case (see Figure 12). [20] also shows that Fletcher-32 is better than Adler-32 for 65536 bit lengths, but does not comment on shorter lengths.

The Adler checksum outperforms the Fletcher checksum only for 16-bit checksums, and only in that checksum's HD=3 performance region (see Figure 12). The issue is that while the prime modulus in the Adler checksum results in better mixing, there are fewer "bins" (i.e., valid FCS values) available for code words. In most cases, this reduction in bins outweighs the gains made by better mixing. Thus, the Fletcher checksum is superior to the Adler checksum in all cases except for Adler-16 used on short data word lengths. And even then, the improvement in error detection effectiveness might not be worth the increase in complexity and computational cost of performing modular addition.

V. Error Detection and Compute Cost Tradeoffs

Selection of the best checksum for a given application is usually not based on error detection properties alone. Other factors such as computational cost frequently come into play as well.

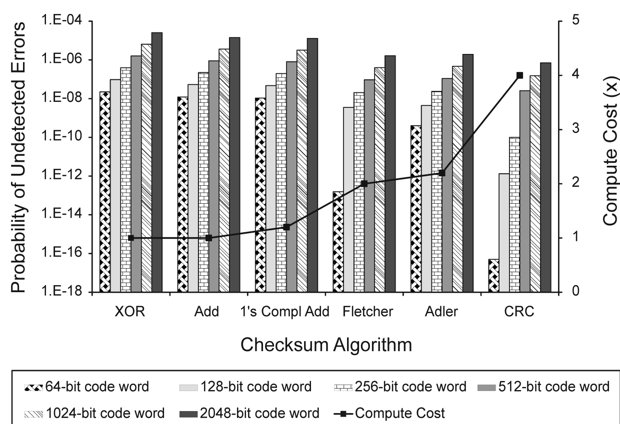

Fig. 13. Error Detection Effectiveness vs. Compute Cost Tradeoffs for 8-bit Checksums (BER of 10^{-5}) The compute cost unit x is equal to the compute cost of the XOR checksum.

Feldmeier's study on fast software implementations for checksum algorithms [30] showed that one's complement addition checksum is approximately twice as fast as Fletcher checksum and Fletcher checksum is at least twice as fast as CRC. To get a better idea of the tradeoffs per checksum algorithm, the results of our study were plotted against the results of Feldmeier's paper. (See Figure 13)

The XOR checksum has the smallest compute cost of all checksum algorithms. However, it also has the worst error detection properties among all the checksum algorithms. Its error detection properties do not significantly change with code word length.

Two's complement addition checksum ("add checksum" for short), if executed in hardware, has a slightly higher compute cost than XOR checksum due to propagation delay of carry bits. If executed in software, the add checksum has the same compute cost as XOR checksum. At little or no extra cost, the add checksum is nearly twice as good as the XOR checksum in error detection properties. Code word length does not significantly affect its error detection ability.

For some hardware implementations, one's complement addition checksum has no additional cost over add checksum. In software implementations, it has a very slightly higher compute cost than add checksum because of the MSB carry bit incorporation. It makes up for this slightly higher cost by being slightly better at error detection than add checksum. Its error detection properties are not affected by code word length. The one's complement addition checksum should usually be used instead of XOR checksum and add checksum, unless there are very compelling reasons for not doing so.

The Fletcher checksum has double the compute cost of one's complement addition checksum due to its having two running sums instead of one, but is at least an order of magnitude better at error detection at long code word lengths. For short code word lengths, it is a number of orders of magnitude better than the one's complement addition checksum due to its HD=3 error detection. As can be seen from Figure 13, error detection properties of the Fletcher checksum sharply deteriorate after a certain code word length.

The Adler checksum has a slightly higher compute cost than the Fletcher checksum due to its use of a prime for its modulo. It has, at most, a comparable error detection property to Fletcher checksum. Like Fletcher checksum, its error detection ability also drops off after a certain code word length. (It drops off at 60 bits for Adler-8, which is why Adler checksum appears to be much worse than Fletcher checksum in Figure 13.) When given a choice between using the Fletcher checksum and Adler checksum for short codeword lengths, the Fletcher checksum should almost always be used. Not only does it have a lower compute cost, but also, it has better overall error detection properties.

The CRC has the highest compute cost of all checksum algorithms. It is generally double the compute cost of Fletcher checksum. However, it also has the best error detection properties of all the checksum algorithms. For the same checksum size, an optimal CRC polynomial is orders of magnitude better than Fletcher checksum for code word lengths less than 2^k where k is the checksum size. For code word lengths longer than this, an optimal CRC polynomial is approximately an order of magnitude better than Fletcher checksum. Among all the checksum algorithms studied, the CRC has the greatest variance in error detection ability with respect to code word length.

To serve as verification of Feldmeier's results, the compute costs of the actual code used to obtain

the data for this study was determined. As expected, XOR and add checksums have exactly the same cost when implemented in software. One's complement addition checksum had a compute cost of 1.8 times that of add checksum. This could be explained by the fact that the C implementation did not use the "add with carry" instruction. If this instruction were used, the compute cost would have been the same as that of add checksum. Fletcher checksum had a compute cost of 4 times that of add checksum. The reason for this is that the authors were using a non-optimized version of the Fletcher checksum. As shown in [30], the Fletcher checksum can be optimized to twice the compute cost of add checksum. Adler checksum had a compute cost of 1.25 times that of the Fletcher checksum, which is as expected.

Figure 13 shows that the shorter the code word length, the greater the benefit of using a CRC compared to other checksum algorithms. For code word lengths greater than 2^k with k equal to checksum size, the benefit of using a CRC drops sharply because it only provides HD=2 error detection performance. (Results are similar for larger checksums, with the lengths at which CRCs provide better than HD=2 substantially longer.) Thus, while it may be difficult to justify the increased computational cost of using a CRC for the large data words found in typical enterprise and desktop computing environments, the story is quite different for short messages (often less than 100 bits) typically found in embedded networks. For embedded network applications, using a CRC can bring orders of magnitude better error detection performance for a factor of about 4 performance penalty. ([33] presents some CRC polynomials that have better error detection performance at comparable cost to common CRC polynomials.) While Fletcher and Adler checksums can provide HD=3 at short message lengths, they are outperformed by a good CRC at all message lengths by at least one bit of HD.

Of course these performance numbers are just approximate ratios and many factors determine the best choice for a particular application. However, the general notion in widespread circulation that Fletcher and Adler checksums are more or less as good as a CRC at dramatically less computation cost are not really accurate for embedded applications. Checksums other than CRC give up orders of magnitude in error detection effectiveness in return for a factor of two to four speedup. Moreover, applications that use an XOR checksum could have significantly better error detection for essentially the same computational cost simply by using a two's complement addition or preferably a one's complement addition checksum.

VI. Conclusions

The error detection properties of checksums vary greatly. The probability of undetected errors for a k-bit checksum is not always $1/2^k$ in realistic applications as is sometimes thought. Rather, it is dependent on factors such as the type of algorithm used, the length of the code word, and the type of data contained

in the message. The typical determining factor of error detection performance is the algorithm used, with distinct differences evident for short messages typical of embedded networks.

Even for moderately long messages, error detection performance for random independent bit errors on arbitrary data should be considered as a potentially better (and simpler) model than $1/2^k$ where k is the checksum size in bits. Behavior on small numbers of bit errors seems to be a limiting factor of overall error detection performance.

Based on our studies of undetected error probabilities, for applications where it is known that burst errors are the dominant source of errors, XOR, two's complement addition, and CRC checksums provide better error detection performance than one's complement addition, Fletcher, and Adler checksums.

For all other applications, a "good" CRC polynomial, whenever possible, should be used for error detection purposes. It provides at least one additional bit of error detection capability (more bits of Hamming Distance) compared to other checksums, and does so at only a factor of two to four times higher computational cost. In applications where computational cost is a severe constraint, Fletcher checksum is typically a good choice. The Fletcher checksum has lower computational cost than the Adler checksum and, contrary to popular belief, is also more effective in most situations. In the most severely constrained applications, one's complement addition checksums should be used if possible, with two's complement addition a less effective alternative. There is generally no reason to continue the common practice of using an XOR checksum in new designs, because it has the same software computational cost as an addition-based checksum but is only about half as effective at detecting errors.

References

[1] R. Bosch GmbH, *CAN Specification Version 2.0*, Sept. 1991.

[2] FlexRay Consortium, *FlexRay Communications System Protocol Specification Version 2.1*, May 2005.

[3] TTTech Computertechnik AG, *Time Triggered Protocol TTP/C High-Level Specification Document, Protocol Version 1.1*, Nov. 2003, specification Edition 1.4.3.

[4] The HART Book. The hart message structure. What is HART?, http://www.thehartbook.com/technical.htm. accessed Dec. 2005.

[5] Thales Navigation, *Data Transmission Protocol Specification for Magellan Products Version 2.7*, Feb. 2002.

[6] MGE UPS Systems, *Simplified SHUT and HID specification for UPS*, Mar. 2002.

[7] Vendapin LLC, *API Protocol Specification for CTD-202 USB Version & CTD-203 RS-232 Version Card Dispenser*, Aug. 2005.

[8] Q. Lian, Z. Zhang, S. Wu, and B. Y. Zhao, "Z-Ring: Fast prefix routing via a low maintenance membership protocol," in *13th IEEE International Conference on Network Protocols 2005*, Nov. 6–9, 2005, pp. 132–146.

[9] W. Christensen. (1982, Jan.) Modem protocol documentation. http://www.textfiles.com/apple/xmodem. Revised version (1985). accessed Dec. 2005. [Online]. Available: http://www.textfiles.com/apple/xmodem

[10] Modicon, Inc., *Modbus Protocol Reference Guide*, June 1996, pI-MBUS-300 Rev. J.

[11] McShane Inc. Calculating the checksum. Communications Protocol, http://mcshaneinc.com/html/Library_CommProtocol.html. accessed Dec. 2005.

[12] Opto 22, *Optomux Protocol Guide*, Aug. 2005.

[13] R. Braden, D. Borman, and C. Partridge, "Computing the internet checksum," Network Working Group Request for Comments (RFC) 1071, Sept. 1988.

[14] J. Zweig and C. Partridge, "TCP alternate checksum options," Network Working Group Request for Comments (RFC) 1146, Mar. 1990.

[15] P. Deutsch and J.-L. Gailly, "ZLIB compressed data format specification version 3.3," Network Working Group Request for Comments (RFC) 1950, May 1996.

[16] W. W. Plummer, "TCP checksum function design," *ACM SIGCOMM Computer Communication Review*, vol. 19, no. 2, pp. 95–101, Apr. 1989.

[17] J. G. Fletcher, "An arithmetic checksum for serial transmissions," *IEEE Transactions on Communications*, vol. COM-30, no. 1, pp. 247–252, Jan. 1982.

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

[19] K. Sklower, "Improving the efficiency of the OSI checksum calculation," *Computer Communication Review*, vol. 19, no. 5, pp. 44–55, Oct. 1989.

[20] D. Sheinwald, J. Satran, P. Thaler, and V. Cavanna, "Internet protocol small computer system interface (iSCSI) cyclic redundancy check (CRC) / checksum considerations," Network Working Group Request for Comments (RFC) 3385, Sept. 2002.

[21] T. Baicheva, S. Dodunekov, and P. Kazakov, "On the cyclic redundancy-check codes with 8-bit redundancy," *Computer Communications*, vol. 21, pp. 1030–1033, 1998.

[22] ——, "Undetected error probability performance of cyclic redundancy-check codes of 16-bit redundancy," *IEEE Proceedings Communications*, vol. 147, no. 5, pp. 253–256, Oct. 2000.

[23] P. Kazakov, "Fast calculation of the number of minimum-weight words of crc codes," *IEEE Transactions on Information Theory*, vol. 47, no. 3, pp. 1190–1195, Mar. 2001.

[24] C. Partridge, J. Hughes, and J. Stone, "Performance of checksums and CRCs over real data," *ACM SIGCOMM Computer Communication Review. Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, vol. 25, no. 4, pp. 68–76, Oct. 1995.

[25] J. Stone, M. Greenwald, C. Partridge, and J. Hughes, "Performance of checksums and CRC's over real data," *IEEE/ACM Transactions on Networking*, vol. 6, no. 5, pp. 529–543, Oct. 1998.

[26] J. Stone and C. Partridge, "When the CRC and TCP checksum disagree," *ACM SIGCOMM Computer Communication Review. Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, vol. 30, no. 4, pp. 309–319, Oct. 2000.

[27] P. Koopman, "32-bit cyclic redundancy codes for internet applications," in *International Conference on Dependable Systems and Networks 2002*, June 23–26, 2002, pp. 459–468.

[28] P. Koopman and T. Chakravarty, "Cyclic redundancy code (CRC) polynomial selection for embedded networks," in *International Conference on Dependable Systems and Networks 2004*, June 28 – July 1, 2004, pp. 145–154.

[29] A. J. McAuley, “Weighted sum codes for error detection and their comparison with existing codes,” *IEEE/ACM Trans. on Networking*, vol. 2, no. 1, pp. 16–22, Feb. 1994.

[30] D. C. Feldmeier, “Fast software implementation of error detection codes,” *IEEE/ACM Transactions on Networking*, vol. 3, no. 6, pp. 640–651, Dec. 1995.

[31] T. Ramabadran and S. Gaitonde, “A tutorial on CRC computations,” *IEEE Micro*, vol. 8, no. 4, pp. 62–75, Aug. 1988.

[32] D. V. Sarwate, “Computation of cyclic redundancy checks via table look-up,” *Communications of the ACM*, vol. 31, no. 8, pp. 1008–1013, Aug. 1988.

[33] J. Ray and P. Koopman, “Efficient high hamming distance crcs for embedded applications,” in *International Conference on Dependable Systems and Networks 2006*, June 2006.

[34] Wikipedia. Adler-32. http://en.wikipedia.org/wiki/Adler-32. accessed Dec. 2005.

[35] K. H. Fritsche, “Tinytorrent: Combining bittorrent and sensornets,” The University of Dublin, Trinity College, Dublin, Ireland, Tech. Rep. TCD-CS-2005-74, Dec.

Appendix I

Two's Complement Addition Checksum Formula Derivations

The formulas for the percentage of undetected errors for all zero and all one data are derived as follows.

A. All Zero Data

For an n-bit code word with all zero data, the number of undetected two-bit errors is equal to the sum of the total number of bits in the code word n minus the checksum size k, and the combination of all the most significant bits in the data word taken two at a time,

$$(n-k) + \binom{((n-k)/k)}{2} = (n-k) + \frac{((n-k)/k)((n-2k)/k)}{2}$$

$$= \frac{2(n-k) + ((n-k)/k)((n-2k)/k)}{2} = \frac{2k^2(n-k) + (n-k)(n-2k)}{2k^2}$$

The total number of possible two-bit errors for an n-bit code word is

$$\binom{n}{2} = \frac{n(n-1)}{2}$$

Dividing the number of undetected two-bit errors by the total number of two-bit errors gives us the percentage of undetected two-bit errors as

$$\frac{\frac{2k^2(n-k)+(n-k)(n-2k)}{2k^2}}{\frac{n(n-1)}{2}} = \frac{2k^2(n-k) + (n-k)(n-2k)}{nk^2(n-1)}$$

where n is the code word length in bits and k is the checksum size in bits.

B. All One Data

For an n-bit code word with all one data, the equation used depends on whether the most significant bit (MSB) of the checksum is 1 or 0. The MSB changes every $(2^k/2) * k$ bits of data word length. For example, in an 8-bit checksum, the MSB of the checksum changes after every 1024 data word bits.

Looking at the 1st, 3rd, 5th, and so on set of data words, it can be seen that the MSB of the checksum is 1. For this case, an n-bit code word will have an undetected two-bit error equal to the checksum size k minus the number of ones i in the binary form of $((n/k) - 2)$ multiplied by data word length $((n-k)/k)$, plus the combination of all the most significant bits in the data word taken two at a time:

$$(k-i)((n-k)/k) + \binom{((n-k)/k)}{2} = (k-i)\left(\frac{n-k}{k}\right) + \frac{(n-k)(n-2k)}{2k^2}$$

Considering the 2nd, 4th, 6th, and so on set of data words, the undetected two-bit error for an n-bit code word is equal to one plus the checksum size k minus the number of ones i in the binary form of $((n/k)-2)$ multiplied by data word length $((n-k)/k)$, plus the combination of all the most significant bits in the data word taken two at a time:

$$(k-i+1)((n-k)/k) + \binom{((n-k)/k)}{2} = (k-i+1)\left(\frac{n-k}{k}\right) + \frac{(n-k)(n-2k)}{2k^2}$$

The reason for the addition of 1 to the 2nd equation is that having a value of 0 in the MSB causes the bits in the MSB column to generate additional undetected errors.

The two equations above when divided by the number of all possible two-bit errors (n bit combinations taken two at a time) will yield the following equations.

For the 1st, 3rd, 5th, and so on set of $(2^k/2)*k$ data words,

$$\frac{2(k-i)((n/k)-1) + ((n/k)-1)((n/k)-2)}{n(n-1)}$$

where n is code word length in bits, k is checksum size in bits, and i is the number of zeros in the checksum or the number of ones in the binary form of $((n/k)-2)$ within checksum width.

For the 2nd, 4th, 6th, and so on set of $(2^k/2)*k$ data words,

$$\frac{2(k-i+1)((n/k)-1) + ((n/k)-1)((n/k)-2)}{n(n-1)}$$

where n is code word length in bits, k is checksum size in bits, and i is the number of zeros in the checksum or the number of ones in the binary form of $((n/k)-2)$ within checksum width.

APPENDIX II

ONE'S COMPLEMENT ADDITION CHECKSUM FORMULA DERIVATION

For all zero and all one data, only one equation is needed because there are no undetected two-bit errors due to the MSB. The equation is equal to the one from two's complement add checksum for all zero data minus the undetected bit errors caused by the MSB. Thus, percentage of undetected errors is equal to:

$$\frac{(n-k)}{\binom{n}{2}} = \frac{(n-k)}{\frac{n(n-1)}{2}} = \frac{2(n-k)}{n(n-1)}$$

where n is the code word length in bits and k is the checksum size in bits.