The Effectiveness of Checksums for Embedded Control Networks

Theresa C. Maxino, *Member*, *IEEE*, and Philip J. Koopman, *Senior Member*, *IEEE*

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: exclusive or (XOR), two's complement addition, one's complement addition, Fletcher checksum, Adler checksum, and cyclic redundancy codes (CRCs). A study of error detection capabilities for random independent bit errors and burst errors reveals that the XOR, two's complement addition, and Adler checksums are suboptimal for typical network use. Instead, one's complement addition should be used for networks willing to sacrifice error detection effectiveness to reduce computational cost, the Fletcher checksum should be used for networks looking for a balance between error detection and computational cost, and CRCs should be used for networks willing to pay a higher computational cost for significantly improved error detection.

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

1 Introduction

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

The area of interest for this study is embedded control networks. These networks are generally optimized for relatively short periodic time-critical messages on single-hop networks for use in embedded control applications (for example, in automobiles, rail applications, elevators, and in-cabinet industrial control). Because many applications have unique requirements, it is common to create custom protocols on a per-application or per-vendor basis. Ethernet and general-purpose enterprise network components such as switches are generally not used because of their cost and poor suitability for compact messages with real-time deadlines.

Representative embedded control network protocols are Controller Area Network (CAN) [1], FlexRay [2], and TTP/C [3], which use a CRC to ensure data integrity. However, the 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 (for example, [6], [7], and [8]). Another widely used alternative is the two's complement addition checksum, historically used by XMODEM [9] and currently used by Modbus ASCII [10] and some proprietary communication protocols (for example, [11] and [12]).

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

Despite the fact that checksum techniques have been in widespread use for decades in embedded networks, there is surprisingly little information available about their relative performance and effectiveness. Although 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 networks, it is worth knowing which checksum approaches work best.

This paper examines the most commonly used checksum approaches in embedded networks and evaluates their comparative error detection effectiveness, as well as cost/performance trade-off 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 the common use of the generic term "checksum" to mean a function that

- *T.C. Maxino is with Lexmark Research and Development Corporation, InnOvé Plaza, Samar Loop Corner Panay Road, Cebu Business Park, Cebu City, 6000 Philippines. E-mail: tmaxino@alumni.cmu.edu.*
- *P.J. Koopman is with Carnegie Mellon University, Hamerschlag Hall A308, 5000 Forbes Ave., Pittsburgh, PA 15213. E-mail: koopman@cmu.edu.*

Manuscript received 2 Sept. 2006; revised 11 June 2007; accepted 17 July 2007; published online 25 July 2007.
For information on obtaining reprints of this article, please send e-mail to: tdsc@computer.org, and reference IEEECS Log Number TDSC-0124-0906.
Digital Object Identifier no. 10.1109/TDSC.2007.70216.

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 a binary symmetric channel. 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.

2 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 an FCS in networking applications. The checksum is appended or prepended to the data word (the message payload) and transmitted with it, making this a systematic code in which the data being sent is included in the code word unchanged. 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 trade-off among the computing power used on the checksum calculation, the size of the FCS field, and the probability of such undetected errors.

Commonly used checksums generally fall into three general areas of cost/performance trade-off. 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. References [16], [17], [18], [19], and [20] have analyzed the error detection effectiveness of two's complement addition and one's complement addition checksums. Reference [21] provides an analytic comparison of the error detection effectiveness of XOR, two's complement addition, one's complement addition, and CRC 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. Early CRC effectiveness studies were lists of optimal CRC polynomials for specific lengths (for example, [22], [23], and [24]). Funk [25] and Koopman [26], [27] investigated the CRC polynomials currently in use and proposed alternatives that provided better performance. Those papers proposed a polynomial selection process for embedded networks and determined the optimum bounds for 3- to 16-bit CRCs for data words up to 2,048 bits. We use these results as the source of our CRC data.

Because CRC computation is so expensive, two intermediate-cost checksums have been proposed for use in nonembedded networks. The Fletcher checksum [28] and the later Adler checksum [15] are both designed with a goal of giving error detection properties competitive with CRCs with significantly reduced computational cost. In the late 1980s, Nakassis [29] and Sklower [30] published efficiency improvements for Fletcher checksum implementations that also are useful to speed up one's complement addition checksums. Although 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. Fletcher published error detection information in his original paper [28], whereas [21] and [31] present further analysis.

Stone et al. [32], [33], [34] measured the network checksum effectiveness of the one's complement addition checksum, Fletcher checksum, and CRC. In their study, they found that the one's complement addition checksum and Fletcher checksum had much higher probabilities of undetected errors than they expected, and one reason they gave was the nonuniformity of network data. They also found that there were sources of faults that seemed to be from network adapters, switches, and sources other than the types of faults we model in this paper. However, there is no experience to suggest the degree to which such faults are present in embedded control networks, which have much simpler network adapters and usually do not have standard-component switches. Thus, these fault models are not considered in our work.

McAuley [35] has proposed the Weighted Sum Code (WSC) algorithm as an alternative to the Fletcher checksum and CRC. Feldmeier [36] conducted a comparison of WSC against the one's complement addition checksum, XOR checksum, block parity, Fletcher checksum, and CRC and concluded that it was an attractive alternative for some situations. However, WSC does not provide guaranteed detection of as many error bits in one message as many commonly used CRCs in situations of interest to embedded network designers. Also, the algorithm is more complex than a CRC, and the speed benefits are primarily attractive for 64-bit checksum values [36], which are generally too large to be of interest in embedded networks. Therefore, WSCs are considered out of the scope of the current study.

In the balance of this paper, 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.

3 Effectiveness Evaluation

The effectiveness of different error detection codes depends on the operating environment, the data being sent on the embedded network, and the types of bit errors caused by noise and other sources of faults. In general, there is no comprehensive model of faults available for embedded networks. Moreover, applying fault information from enterprise networks is likely to be misleading because of widely different operating environments and requirements (for example, it is common for embedded networks to use unshielded cabling as a cost reduction measure). Therefore, the approach taken in this study is to use fault models that are commonly used in practice and note instances in which there are specific vulnerabilities in a checksum approach. The metrics used are the probability of undetected random independent bit errors in a binary symmetric channel, the probability of undetected burst errors, and the maximum number of bit errors guaranteed to be detected in a single message.

The evaluation of the effectiveness of checksums typically requires a combination of analytic and experimental approaches. Finite-field operations (for example, XOR) can often be evaluated analytically. However, the interbit carries inherent in integer addition operations make analytic approaches to understanding error detection effectiveness very complex. The approach we take is using analysis to approximate the results to the degree practical, with simulations used to validate the analytic results and fill in otherwise intractable areas.

To carry out the experiments in this paper, we implemented the various checksum algorithms in C++. The 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 and 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 code word. 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 [26] (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.

Random independent bit error experiments were conducted by preselecting a set number of bit errors to introduce and then injecting these errors into the code word. (Different numbers of bit errors were chosen for different experiments, and graphs take into account the decreasing probability of higher numbers of bit errors in a particular message of a given size.) The faults injected in each experiment were 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 10 trials were made for each type of experiment, and the mean for all the trials was obtained. For example, 10 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 5 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.)

Burst errors are random bit errors that are confined to a span of no more than b bits for a b-bit burst error within a given message. Any number of bits may be corrupted within the burst. 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 specific burst error lengths with all possible bit error patterns within the burst boundaries and all possible locations of the burst position.

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 networks is the fraction of undetected errors at the HD, because the probability of more errors occurring is low (for example, 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, the analysis of undetected errors at the HD is performed to give insight into experimental results.

Some of the graphs in this paper 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}). P_{ud} accounts for the differing probability of each number of bit errors, with increasing numbers of random independent bit errors being substantially less probable. P_{ud} is often more useful than just Pct_{ud}, because it takes into account the BER and permits the 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. This equation first determines the probability of a particular code word error having precisely x errors on exactly one possible combination of bits (which means that x bits are erroneous and $n - x$ bits are nonerroneous). It then multiplies by Hamming Weight (HW), which is the number of undetectable errors with that number of bits, giving the probability of an undetectable error. For these purposes, the HW of interest is the HW at the HD. For example, if there were zero undetectable errors for 2-bit and 3-bit errors but 173 undetectable 4-bit errors, then the HD would be four (giving $x = 4$ in this equation) and HW would be 173. HW values for successively higher numbers of bit errors must be summed together to find the exact P_{ud} value. However, the HW at the HD dominates P_{ud} for codes examined in this paper, making this approximation sufficient for graphing purposes.

For burst error experiments, burst error lengths starting from 2 bits up to at least $(k + 1)$ bits $(k = \text{checksum size})$ were injected in the code word for each data word value examined. For a particular burst error degree, all possible burst error values were injected. For example, all possible 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 because the usual burst error metric of interest is the maximum length at which burst errors are guaranteed to be detected. The exceptions to this were the experiments for Fletcher and Adler checksums. We performed experiments up to $k + 1$ bursts for consistency with other experiments, even though there were already undetected burst errors at $k/2$ bits $(k = \text{checksum size})$.

In some instances, we found it useful to perform comparisons of the probability of undetected errors to test 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 belief is that because there are 2^k possible checksum values for a k-bit checksum, for 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 (for example, a one-in-256 probability of undetected error for an 8-bit checksum).

To simplify comparisons, only data word lengths that were multiples of the checksum size were used in this study, as is common in real networks. 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.

3.1 Exclusive Or Checksum

XOR checksums are computed by XORing 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, for example, 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 an HD of two, detecting all 1-bit errors but not some 2-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 is equal to the checksum size), because 2 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 pairs of inverted bits do not align in the same bit position in the affected blocks.

For a block size of k and checksum size k, in every pair of data blocks, there are exactly k possible undetected 2-bit errors (one undetected 2-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 2-bit errors is

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

The total number of possible 2-bit errors for an n-bit code word is

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

Dividing the number of undetected 2-bit errors by the total number of 2-bit errors gives us the fraction of undetected 2-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.

From the above equation, we can see that as n approaches infinity, the percentage of undetected 2-bit errors becomes approximately $1/k$ (k being checksum size), which is rather poor performance for a checksum (for example, 12.5 percent undetected errors for an 8-bit checksum and 3.125 percent for a 32-bit checksum). Simulation results confirm this analysis. (See Fig. 1. Note that subsequent figures have a different vertical axis to help distinguish closely spaced curves. Fig. 6 provides a comparison across checksum techniques on a single graph.) 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.

3.2 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 1-bit errors in the code word and has an HD of two 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 carries of the binary addition. The effectiveness of mixing depends on the

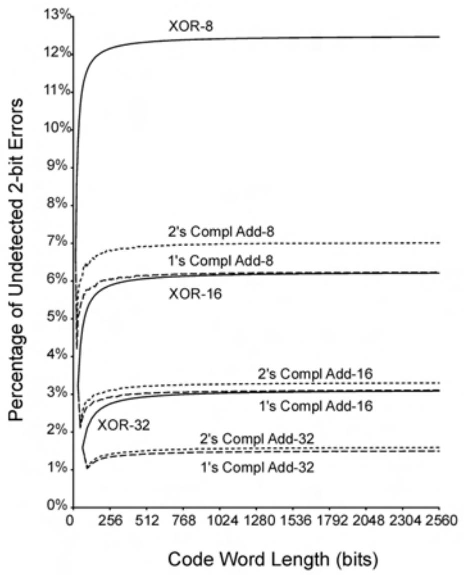

Fig. 1. Percentage of undetected 2-bit errors over the total number of 2-bit errors for 8-, 16-, and 32-bit XOR, two's complement addition, and one's complement addition checksums. Two's complement addition and one's complement addition data values are the mean of 100 trials using random data.

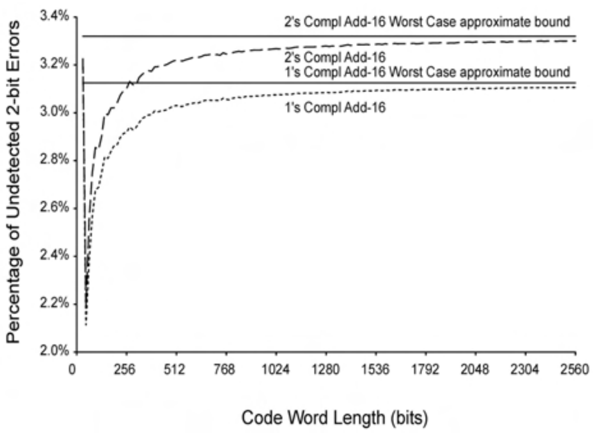

Fig. 2. Percentage of undetected 2-bit errors for 16-bit two's complement addition and one's complement addition checksums. The data points for both checksums are the mean of 100 trials using random data. The worst case bound is from the given formula.

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 lines up at the same bit position within the blocks, and the data in those bit positions contains a one in one block and a zero in the other block. The resultant erroneous data blocks have a zero in the first block and a one 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 the 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 position.

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 one and zero 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 (for example, [33] mentions this, and it is also common in embedded networks), the random data evaluation of add checksums can be considered pessimistic for many cases. The add checksum performs best when the 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 in Fig. 1, the 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 do not necessarily differ.

For worst case data, the add checksum has an undetected 2-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 has an undetected error percentage equal to that of XOR, $1/k$. All the other bits have an undetected error percentage that is half that of XOR $(1/2k)$ because only 0-1 and 1-0 error combinations will be undetected. Multiplying 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 an approximate bound, as can be seen in Fig. 2.

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

Fig. 3 shows simulation results for exactly identical numbers of zero and one data (alternating 0xFF and 0x00 values), all zeros, and all ones. The 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

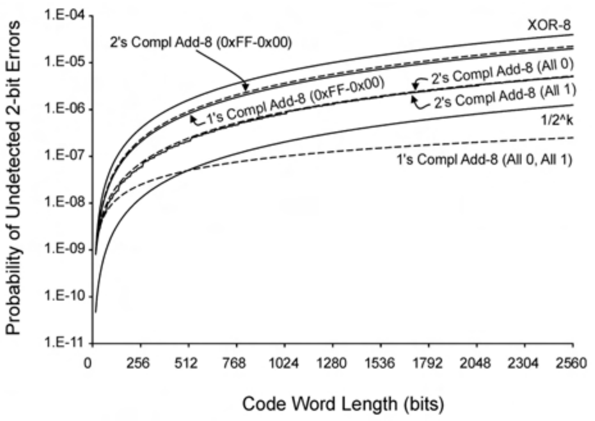

Fig. 3. Probability of undetected 2-bit errors for 8-bit checksums using different data and a BER of 10^{-5}.

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 error will go undetected.

3.3 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 (for example, [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. (Reference [21] 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 wraparound 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 bits in the MSB position, the behavior of one's and two's complement addition checksums is identical, with the one's complement addition checksum having a slightly lower probability of undetected errors for random independent bit errors (Fig. 3).

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

For worst case data at asymptotic lengths, approximately $1/2k$ of all possible 2-bit errors, where k is the checksum size, are detected. This is half of the ratio of undetected errors for the 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 the XOR checksum where 0-0 and 1-1 error combinations are also undetectable. Looking at Fig. 1, it can be seen that the one's complement addition checksum is almost as good as the XOR checksum at half the checksum size for random independent bit errors on random data.

3.4 One's Complement Fletcher Checksum

The Fletcher checksum [28], [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 [29]. We confirmed this experimentally. (Throughout this paper, "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 from D_0 to D_n is

$$\begin{aligned}
&\text{Initial values :} && sumA = sumB = 0;\\
&\text{For increasing } i: && \{\, sumA = sumA + D_i;\\
& && sumB = sumB + sumA;\,\}.
\end{aligned}$$

$sumA$ 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 the 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 the data is all zeros, 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 that is half the checksum size k. As expected, it is vulnerable to burst errors that invert bits in a block from all zeros to all ones or vice versa. (The Adler checksum has the same vulnerability. Whether such an error is likely depends on the bit encoding technique, with, for example, Manchester encoding being vulnerable to this sort of error if a half-bit "slip" occurs that causes a 180-degree phase shift in received data waveform edges.) Our experiments have verified that excluding this special type of burst error, the

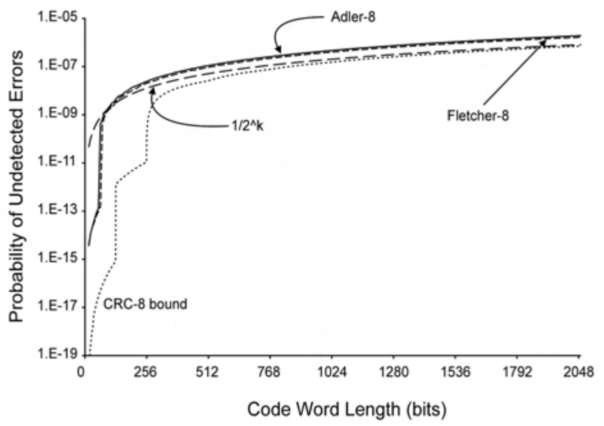

Fig. 4. Probability of undetected errors for 8-bit Fletcher and Adler checksums using random data and a BER of 10^{-5}. Data values for both Fletcher and Adler checksums are the mean of 10 trials. CRC-8 bound values are optimal values for all 8-bit CRC polynomials.

Fletcher checksum can detect all burst errors less than k bits, where k is the checksum size. Reference [28] gives a more detailed explanation of burst error detection properties.

The Fletcher checksum has $\mathrm{HD} = 3$ up to a certain modulo-dependent code word length and $\mathrm{HD} = 2$ for all remaining code word lengths. We have confirmed experimentally that 2-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. Reference [28] states further that all 2-bit errors are detected provided that they are separated by fewer than $(2^{k/2} - 1) * (k/2)$ bits, with k being the checksum size.

We have also confirmed experimentally that an 8-bit Fletcher checksum has $\mathrm{HD} = 2$ for code word lengths of 68 bits and above, with $\mathrm{HD} = 3$ below that length. A 16-bit Fletcher checksum has $\mathrm{HD} = 2$ starting at 2,056-bit code word lengths. According to the equation, a 32-bit Fletcher checksum is expected to have $\mathrm{HD} = 2$ starting at a code word length of 1,048,592 bits.

Fig. 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, whereas CRCs typically have a slightly lower probability of undetected error than $1/2^k$, with k being the 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 given in Section 4.2. 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, the Fletcher checksum has the next-best overall error detection capability after CRC, except for the special case of the 16-bit Adler checksum at short lengths.

3.5 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 65,535 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 one, and each addition is done modulo 65,521 (for the 32-bit Adler checksum) instead of modulo 65,535. 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 Fig. 4.) We have confirmed experimentally that 2-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 $\mathrm{HD} = 3$ below 60 bits (using modulo 13 sums) and that Adler-16 has $\mathrm{HD} = 3$ below 2,024 bits (using modulo 251 sums). From the equation, Adler-32 is expected to have $\mathrm{HD} = 3$ below 1,048,368 bits. For code word lengths greater than those mentioned above, the Adler checksum has $\mathrm{HD} = 2$. As with Fletcher checksums, the worst case for the 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 that is equal to half the checksum size k. Adler-32 detects all burst errors up to 7 bits long. (Reference [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 zeros to all ones or vice versa, all burst errors less than $k - 1$ are detected. This is 1-bit less than the Fletcher checksum, which was unexpected since they use an almost identical mathematical basis. (Reference [31] states that the Adler checksum has a higher probability of undetected burst errors than the Fletcher checksum but does not explicitly state that the burst error detection coverage is 1 bit shorter in length.) The reason for this is that Adler checksums use a prime modulo that is less than $2^k - 1$, whereas Fletcher checksums use a modulo equal to $2^k - 1$, with k being the checksum size. A comparison of Fletcher and Adler checksum performance is given in Section 4.3.

3.6 Cyclic Redundancy Codes

The simplest version of a CRC computation uses a shift-and-conditional-XOR approach to compute a checksum [37]. Faster methods of computation are available (for example, [37], [38], and [39] 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 [27].

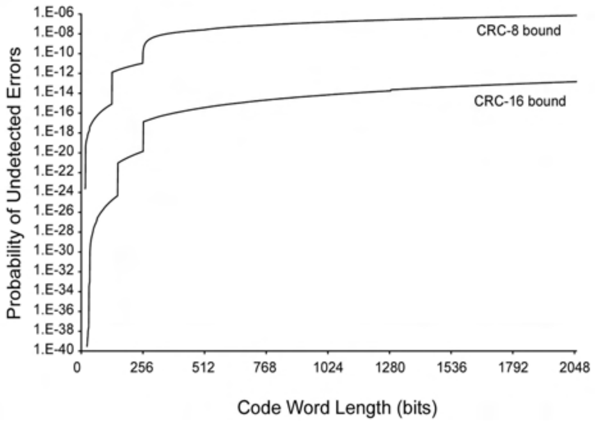

Fig. 5. Bounds for the probability of undetected errors for all 8-bit CRCs and all 16-bit CRCs $(\mathrm{BER} = 10^{-5})$.

All CRCs are data independent and have an HD of at least two for all code word lengths. Most polynomials have $\mathrm{HD} = 3$ 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. Fig. 5 shows the bounds for the lowest probability of undetected errors for 8-bit CRCs and 16-bit CRCs.

4 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 are individuals who do not hold and publications that do not contain these misconceptions, but we have observed these issues to arise often enough that they warrant specific attention.

4.1 Effect of Data Value and Error Value Distributions on Checksum Effectiveness

When data are 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 (for example, a one-in-256 probability of undetected error for an 8-bit checksum). Although this is true for completely random data and corruption, most data is not 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. [33] 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 the one's complement addition checksum and the Fletcher checksum had a P_{ud} that was far worse

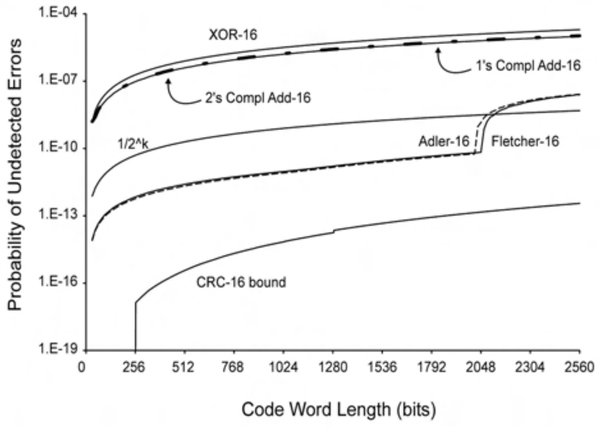

Fig. 6. Performance of 16-bit checksums with random data and random independent bit errors (BER of 10^{-5}). The data values for two's complement addition, one's complement addition, Fletcher, and Adler checksums are the mean of 10 trials.

than the value of $1/2^k$ that they expected. They theorized that one of the reasons for this was because of the nonuniform distribution of the data they were using, and in their Corollary 8, they claim that if data had been uniformly distributed, the IP (one's complement addition) and Fletcher checksums would have been equivalently powerful. Reference [31] furthers this point of view in its analysis. Although this may be true, the explanation is not necessarily useful in predicting the effectiveness of checksums operating on nonrandom data.

We think that it is also important to consider the effectiveness of checksums at detecting small numbers of corrupted bits. The key is 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 pseudorandom 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. (This criterion is a form of the avalanche property that is characteristic of good cryptographically secure hash functions, for example as discussed in [40]. "Better" hash functions produce a "more random" output.) An XOR checksum, for example, changes only 1-bit of computed FCS value for a 1-bit change in data value. One's complement and two's complement addition change only 1 bit of the 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.

The results of effectiveness can be seen when examining performance for small numbers of bit errors. Fig. 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. The XOR, two's complement, and one's complement addition

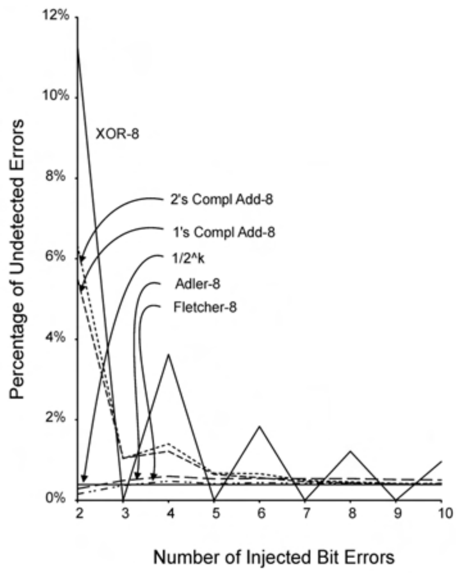

Fig. 7. Percentage of undetected errors in random data for increasing numbers of injected bit errors in a 64-bit data word using 8-bit checksums. The data values are the mean of 10 trials.

checksums perform the worst, whereas CRC performs the best. (Two's complement addition checksum results are very similar to those of the one's complement addition checksum with this vertical axis scale.) This is a result of the different effectiveness of checksums at generating different outputs for small changes in input. The 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 in this figure.

The worst case for a weak checksum algorithm is a small number of bit errors that do not mix the results very much. As the number of bit errors in a single code word increases, all checksums converge to the $1/2^k$ limit value, making the choice of checksum algorithm moot. (See Fig. 7.) Therefore, up to the degree that the 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 two's complement addition, one's complement addition, Fletcher, and Adler are used. Data dependent here means that the P_{ud} for these checksums varies depending on the data word content, unlike in the XOR checksum and CRC where P_{ud} remains the same regardless of the data word content. The percentage of undetected errors is least when the data is all zeros or all ones. 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. 8 shows this effect for the one's complement addition 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 are 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. Fig. 9 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 percent zeros or ones to when data is 50 percent zeros and 50 percent ones in every bit position in the data word.

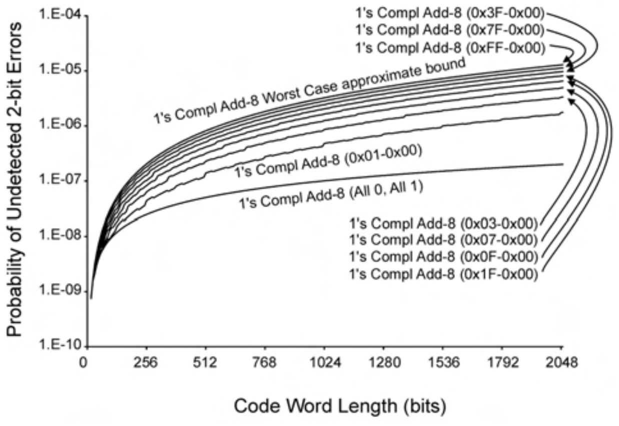

Fig. 8. Probability of undetected 2-bit errors for 8-bit one's complement addition checksum (BER of 10^{-5}) using different data patterns. The data patterns were pairs of bytes of the form XX-00, where XX was the first byte of a repeated pair, varied from 00 to 0xFF, and the second byte was always zero.

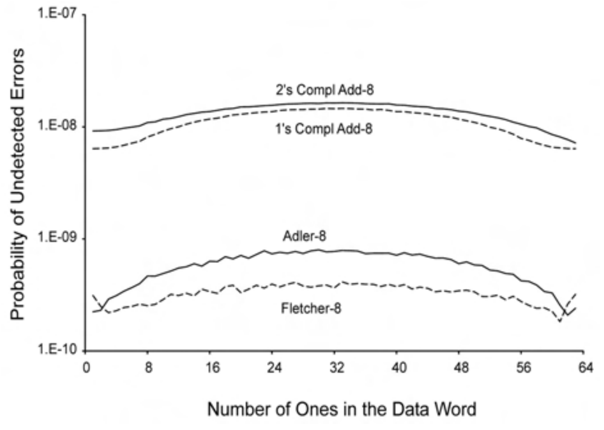

Fig. 9. Probability of undetected errors in random data for a 64-bit data word using 8-bit checksums and a BER of 10^{-5}. The data values are the mean of 100 trials.

4.2 Fletcher Checksum Compared to Cyclic Redundancy Codes

Some previous work (for example, [28] and [15]) lead many practitioners to the conclusion that the Fletcher checksum and Adler checksum are comparable to CRC in error detection capabilities in general or at least for short data word lengths. (This is not quite what those papers claim, but it is nonetheless common "folk wisdom" these many years later and is in fact true to a degree for some "bad" CRCs in widespread use.) However, in all cases, a *good* CRC is substantially better at the same HD and, in many important cases, achieves better HD than either Fletcher or Adler checksums. Sheinwald et al. [31] computed the undetected error probabilities for some CRC-32 polynomials and Fletcher-32 and Adler-32 checksums for a length of 8 Kbytes or 65,536 bits. Their results show that CRC outperforms both Fletcher-32 and Adler-32.

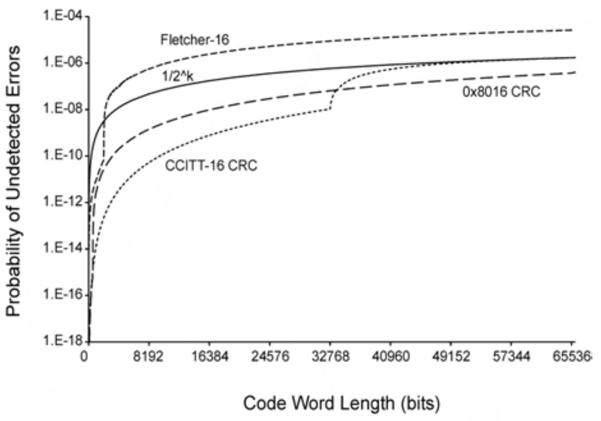

Fig. 10. Probability of undetected errors for a 16-bit Fletcher checksum and two 16-bit CRCs at a BER of 10^{-5}. The data values for the Fletcher checksum are the mean of 10 trials using random data.

Reference [28] 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 $1/2^k$ ($k = \mathrm{checksum\ size}$). "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, with k being the checksum size. Even the comparatively large Ethernet Maximum Transmission Unit (MTU), for example, is only 1,500 bytes (or 12,000 bits), which is much shorter than 65,535 bits. At these lengths, CRCs often have distinct advantages. By using Fletcher and Adler checksums at these lengths, the error detection capability is considerably suboptimal—at least 1-bit of HD in error detection capability is effectively being given up, and often, it is more.

Fig. 10 shows a comparison of the Fletcher checksum to two CRC polynomials, the common CCITT-16 and the 0x8016 polynomial, which performs better than CCITT-16 starting at 32 Kbits, where it has $\mathrm{HD} = 3$ compared to CCITT-16's $\mathrm{HD} = 2$. For short code word lengths, the Fletcher checksum is at least 1 bit of HD worse than CRCs.

We would like to reiterate what was said in [27], which is that the selection of the best CRC polynomial for the desired checksum size is of utmost importance. Fig. 11 shows that the incorrect selection of a CRC polynomial can result in worse error detection performance than the Fletcher checksum. Networks that use DARC-8 would be better off using Fletcher-8, whereas networks that use CRC-8 and ATM-8 but do not use code words shorter than 128 bits would get comparable performance to Fletcher-8. However, polynomial 0xA6 would be a better choice for networks that want to use an 8-bit CRC above a 128-bit data word length.

A comparison of the Fletcher checksum effectiveness to that of CRCs was also performed on CRC checksum sizes less than the Fletcher checksum size (see Fig. 12). Optimal CRC bounds from an 8-bit CRC to a 12-bit CRC were plotted against the 16-bit Fletcher checksum. The resulting graph shows that it is possible for an optimal

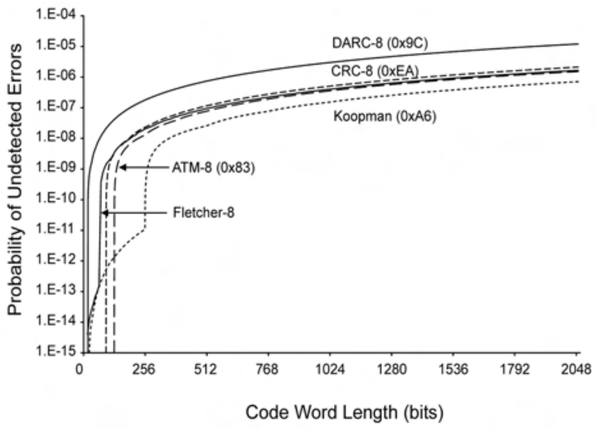

Fig. 11. Probability of undetected errors for some common CRCs and Fletcher-8 at a BER of 10^{-5}. The data values for Fletcher-8 are the mean of 10 trials using random data.

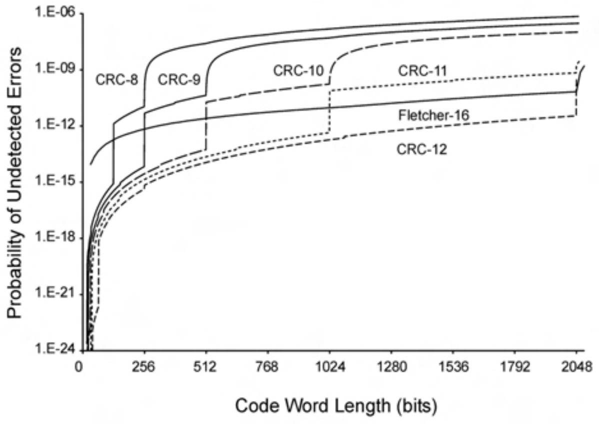

Fig. 12. Probability of undetected errors for Fletcher-16 and CRC bounds for different CRC widths at a BER of 10^{-5}. Data values for Fletcher-16 are the mean of 10 trials using random data.

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 1,024 bits, performs worse for lengths greater than 1,024 but less than 2,048 bits, and performs comparably for lengths greater than 2,048 bits. CRC-12 consistently outperforms Fletcher-16 for all code word lengths. Optimal CRCs with more than 12 bits will perform even better and thus are omitted from the graph.

4.3 One's Complement Fletcher Checksum Compared to the Adler Checksum

The Adler checksum has been put forward as an improvement of the Fletcher checksum [15], and it is commonly believed that the Adler checksum is unconditionally superior to the Fletcher checksum (for example, [41] and [42]). (In a private communication, M. Adler stated that what [15] meant was that Adler-32 is an improvement over Fletcher-16, which is true. At that time, he was not aware of Fletcher-32, but this point is not widely known and is not apparent in [15].)

The better mixing of bits that the Adler checksum provides due to its prime modulus has been claimed to

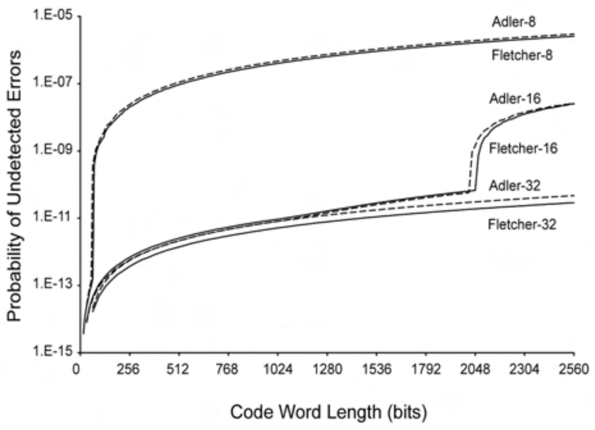

Fig. 13. Comparison of 8-, 16-, and 32-bit Fletcher and Adler checksums using random data at a BER of 10^{-5}. The data point values are the mean of 10 trials.

provide better error detection capabilities than the Fletcher checksum. We have found that this is often not the case (see Fig. 13). Reference [31] also shows that Fletcher-32 is better than Adler-32 for 65,536-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 Fig. 13). The issue is that although the prime modulus in the Adler checksum results in better mixing, there are fewer "bins" (that is, 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. Moreover, even then, the improvement in error detection effectiveness might not be worth the increase in complexity and computational cost of performing modular addition.

5 Error Detection and Computational Cost Trade-Offs

The selection of the best checksum for a given network is usually not based on error detection properties alone. Other factors such as computational cost frequently come into play as well. Feldmeier's study on fast software implementations for checksum algorithms [36] showed that the one's complement addition checksum is approximately twice as fast as the Fletcher checksum, and the Fletcher checksum is at least twice as fast as CRC. Our experience confirms the general validity of those results. We summarize the error detection versus cost trade-offs below based on Feldmeier's performance findings, which we have found to be generally representative of embedded network trade-offs.

5.1 Cost Performance Trade-Offs

The XOR checksum has the smallest computational 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 the code word length. The two's complement addition checksum ("add checksum" for short) has the same computational cost as the XOR checksum in software implementations. The code word length does not significantly affect its error detection ability.

In software implementations, the one's complement addition checksum has a computational cost similar to or very slightly higher than that of the add checksum because of the MSB carry-bit incorporation. (Optimizations similar to those used for Fletcher and Adler checksums, such as [29] and [30], are applicable to any checksum operation involving one's complement addition and make them almost as fast as a two's complement addition checksum.) It makes up for this slightly higher cost by being slightly better at error detection than the add checksum. Its error detection properties are not significantly affected by the code word length. The one's complement addition checksum should usually be used instead of the XOR checksum and the add checksum, unless there are compelling reasons for not doing so.

The Fletcher checksum has approximately twice the computational cost of the 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. The error detection properties of the Fletcher checksum sharply deteriorate after the $HD = 3$ length limit is reached.

The Adler checksum has a slightly higher computational cost than the Fletcher checksum due to its use of a prime modulus. It has, at most, a comparable error detection property to the Fletcher checksum. Like the Fletcher checksum, its error detection ability also drops off after a certain code word length. When given a choice between using the Fletcher checksum and the Adler checksum for short code word lengths, the Fletcher checksum is usually better. It has not only a lower computational cost but also better overall error detection properties.

The CRC has the highest computational cost of all checksum algorithms. It is generally double the computational cost of the 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 the 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 the Fletcher checksum. Among all the checksum algorithms studied, the CRC has the greatest variation in error detection ability with respect to the code word length. There are a number of speedup techniques available for CRC computations, especially for small embedded processors. Ray and Koopman [39] discuss CRC performance options, including the possibility of table-lookup optimizations.

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 the checksum size, the benefit of using a

CRC drops sharply because it only provides $\text{HD} = 2$ error detection performance. (Results are similar for larger checksums, with the lengths at which CRCs provide better than $\text{HD} = 2$ being substantially longer.) Thus, although 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 networks, using a CRC can bring orders of magnitude better error detection performance for a factor of about four performance penalty. (Ray and Koopman [39] present some CRC polynomials that have good error detection performance with faster computation speeds than other CRCs.) Although Fletcher and Adler checksums can provide $\text{HD} = 3$ at short message lengths, they are outperformed by a good CRC at all message lengths.

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 is not really accurate for embedded networks. Checksums other than CRC give up orders of magnitude in error detection effectiveness in return for a factor of two to four speedup. Moreover, networks 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.

5.2 General Checksum Guidelines

Below are general guidelines for checksum use based on the results discussed in this paper. Of course, each individual embedded network application will have its own constraints and trade-offs, but these guidelines should serve as a starting point for making informed design choices:

- Never use an XOR checksum when it is possible to use a two's complement addition checksum (or something even better).
- Use a one's complement addition checksum in preference to a two's complement addition checksum for random independent bit errors.
- Use a two's complement addition checksum in preference to a one's complement addition checksum for burst errors. If both burst and random independent bit errors matter, one's complement is probably the better choice.
- If computational resources are available, use a Fletcher checksum in preference to one's complement or two's complement addition checksums to protect against random independent bit errors. Do not use a Fletcher checksum if burst errors are the dominant fault expected and data consists predominantly of continuous strings of all zeros or all ones.
- If computational resources are available, use a CRC instead of any of the other checksums mentioned. It is generally better for both random independent bit errors and burst errors.
- Take into account the length of the data word when evaluating checksum performance. Performance can vary dramatically with the size of the data word, especially for CRCs.

6 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 networks 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, the error detection performance for random independent bit errors on arbitrary data should be considered as a potentially better (and simpler) model than a fixed fraction of undetected errors of $1/2^k$, where k is the checksum size in bits. The behavior on small numbers of bit errors can easily be a limiting factor of the overall error detection performance.

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

For all networks, 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 HD) compared to other checksums and does so at only a factor of two to four times higher computational cost. In networks where computational cost is a severe constraint, the Fletcher checksum is typically a good choice. The Fletcher checksum has a lower computational cost than the Adler checksum and, contrary to popular belief, is also more effective in most situations. In the most severely constrained networks, one's complement addition checksums should be used if possible, with two's complement addition being 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.

Appendix A

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.1 All-Zero Data

For an n-bit code word with all-zero data, the number of undetected 2-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 MSBs in the data word taken two at a time:

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

The total number of possible 2-bit errors for an n-bit code word is

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

Dividing the number of undetected 2-bit errors by the total number of 2-bit errors gives us the percentage of undetected 2-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.

A.2 All-One Data

For an n-bit code word with all-one data, the equation used depends on whether the MSB of the checksum is one or zero. 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 1,024 data word bits.

Looking at the first, third, fifth, and so on, set of data words, it can be seen that the MSB of the checksum is one. For this case, an n-bit code word will have an undetected 2-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 MSBs in the data word taken two at a time:

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

Considering the second, fourth, sixth, and so on, set of data words, the undetected 2-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 MSBs in the data word taken two at a time:

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

The reason for the addition of one to the second equation is that having a value of zero 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 2-bit errors (n bit combinations taken two at a time) will yield the following equations.

For the first, third, fifth, and so on, set of $(2^k/2)*k$ data words

$$\frac{2(k-i)\left(\frac{n}{k}-1\right)+\left(\frac{n}{k}-1\right)\left(\frac{n}{k}-2\right)}{n(n-1)},$$

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 the checksum width.

For the second, fourth, sixth, and so on, set of $(2^k/2)*k$ data words

$$\frac{2(k-i+1)\left(\frac{n}{k}-1\right)+\left(\frac{n}{k}-1\right)\left(\frac{n}{k}-2\right)}{n(n-1)},$$

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 the checksum width.

APPENDIX B
ONE'S COMPLEMENT ADDITION CHECKSUM FORMULA DERIVATION

For all-zero and all-one data, only one equation is needed because there are no undetected 2-bit errors due to the MSB. The equation is equal to the one from the two's complement addition checksum for all-zero data minus the undetected bit errors caused by the MSB. Thus, the 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.

ACKNOWLEDGMENTS

The authors would like to thank Professor Priya Narasimhan, Mark Adler, and the anonymous reviewers for their helpful comments and suggestions. 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).

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,* specification ed. 1.4.3, Nov. 2003.
[4] The HART Book, *The HART Message Structure. What Is HART?* http://www.thehartbook.com/technical.htm, 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," *Proc. 13th IEEE Int'l Conf. Network Protocols (ICNP '05),* pp. 132-146, Nov. 2005.

[9] W. Christensen, *Modem Protocol Documentation,* Revised version (1985), http://www.textfiles.com/apple/xmodem, Dec. 2005.

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

[11] McShane Inc., "Calculating the Checksum," *Comm. Protocol,* http://mcshaneinc.com/html/Library_CommProtocol.html, Dec. 2005.

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

[13] R. Braden, D. Borman, and C. Partridge, *Computing the Internet Checksum,* IETF RFC 1071, Sept. 1988.

[14] J. Zweig and C. Partridge, *TCP Alternate Checksum Options,* IETF RFC 1146, Mar. 1990.

[15] P. Deutsch and J.-L. Gailly, *ZLIB Compressed Data Format Specification Version 3.3,* IETF RFC 1950, May 1996.

[16] N.R. Saxena and E.J. McCluskey, "Analysis of Checksums, Extended-Precision Checksums, and Cyclic Redundancy Checks," *IEEE Trans. Computers,* vol. 39, no. 7, pp. 969-975, July 1990.

[17] A.M. Usas, "Checksum versus Residue Codes for Multiple Error Detection," *Proc. Eighth Ann. Int'l Symp. Fault-Tolerant Computing (FTCS '78),* p. 224, 1978.

[18] S.C. Tzou Chen and G.S. Fang, "A Closed-Form Expression for the Probability of Checksum Violation," *IEEE Trans. Systems, Man, and Cybernetics,* vol. 10, no. 7, pp. 407-410, July 1980.

[19] C. Jiao and L. Schwiebert, "Error Masking Probability of 1's Complement Checksums," *Proc. 10th Int'l Conf. Computer Comm. and Networks (ICCCN '01),* pp. 505-510, Oct. 2001.

[20] Y. Desaki, K. Iwasaki, Y. Miura, and D. Yokota, "Double and Triple Error Detecting Capability of Internet Checksum and Estimation of Probability of Undetectable Error," *Proc. Pacific Rim Int'l Symp. Fault-Tolerant Systems (PRFTS '97),* pp. 47-52, Dec. 1997.

[21] W.W. Plummer, "TCP Checksum Function Design," *Computer Comm. Rev.,* vol. 19, no. 2, pp. 95-101, Apr. 1989.

[22] T. Baicheva, S. Dodunekov, and P. Kazakov, "On the Cyclic Redundancy-Check Codes with 8-Bit Redundancy," *Computer Comm.,* vol. 21, pp. 1030-1033, 1998.

[23] T. Baicheva, S. Dodunekov, and P. Kazakov, "Undetected Error Probability Performance of Cyclic Redundancy-Check Codes of 16-Bit Redundancy," *IEEE Proc. Comm.,* vol. 147, no. 5, pp. 253-256, Oct. 2000.

[24] P. Kazakov, "Fast Calculation of the Number of Minimum-Weight Words of CRC Codes," *IEEE Trans. Information Theory,* vol. 47, no. 3, pp. 1190-1195, Mar. 2001.

[25] G. Funk, "Determination of Best Shortened Linear Codes," *IEEE Trans. Comm.,* vol. 44, no. 1, pp. 1-6, Jan. 1996.

[26] P. Koopman, "32-Bit Cyclic Redundancy Codes for Internet Applications," *Proc. Int'l Conf. Dependable Systems and Networks (DSN '02),* pp. 459-468, June 2002.

[27] P. Koopman and T. Chakravarty, "Cyclic Redundancy Code (CRC) Polynomial Selection for Embedded Networks," *Proc. Int'l Conf. Dependable Systems and Networks (DSN '04),* pp. 145-154, June 2004.

[28] J.G. Fletcher, "An Arithmetic Checksum for Serial Transmissions," *IEEE Trans. Comm.,* vol. 30, no. 1, pp. 247-252, Jan. 1982.

[29] A. Nakassis, "Fletcher's Error Detection Algorithm: How to Implement It Efficiently and How to Avoid the Most Common Pitfalls," *Computer Comm. Rev.,* vol. 18, no. 5, pp. 63-88, Oct. 1988.

[30] K. Sklower, "Improving the Efficiency of the OSI Checksum Calculation," *Computer Comm. Rev.,* vol. 19, no. 5, pp. 44-55, Oct. 1989.

[31] D. Sheinwald, J. Satran, P. Thaler, and V. Cavanna, *Internet Protocol Small Computer System Interface (iSCSI) Cyclic Redundancy Check (CRC)/Checksum Considerations,* IETF RFC 3385, Sept. 2002.

[32] C. Partridge, J. Hughes, and J. Stone, "Performance of Checksums and CRCs over Real Data," *Computer Comm. Rev., Proc. ACM SIGCOMM '95,* vol. 25, no. 4, pp. 68-76, Oct. 1995.

[33] J. Stone, M. Greenwald, C. Partridge, and J. Hughes, "Performance of Checksums and CRC's over Real Data," *IEEE/ACM Trans. Networking,* vol. 6, no. 5, pp. 529-543, Oct. 1998.

[34] J. Stone and C. Partridge, "When the CRC and TCP Checksum Disagree," *Computer Comm. Rev., Proc. ACM SIGCOMM '00,* vol. 30, no. 4, pp. 309-319, Oct. 2000.

[35] A.J. McAuley, "Weighted Sum Codes for Error Detection and Their Comparison with Existing Codes," *IEEE/ACM Trans. Networking,* vol. 2, no. 1, pp. 16-22, Feb. 1994.

[36] D.C. Feldmeier, "Fast Software Implementation of Error Detection Codes," *IEEE/ACM Trans. Networking,* vol. 3, no. 6, pp. 640-651, Dec. 1995.

[37] T. Ramabadran and S. Gaitonde, "A Tutorial on CRC Computations," *IEEE Micro,* vol. 8, no. 4, pp. 62-75, Aug. 1988.

[38] D.V. Sarwate, "Computation of Cyclic Redundancy Checks via Table Look-Up," *Comm. ACM,* vol. 31, no. 8, pp. 1008-1013, Aug. 1988.

[39] J. Ray and P. Koopman, "Efficient High Hamming Distance CRCs for Embedded Applications," *Proc. Int'l Conf. Dependable Systems and Networks (DSN '06),* June 2006.

[40] A.J. Menezes, P.C. van Oorschot, and S.A. Vanstone, *Handbook of Applied Cryptography.* CRC Press, 1997.

[41] Wikipedia, *Adler-32,* http://en.wikipedia.org/wiki/Adler-32, Dec. 2005.

[42] K.H. Fritsche, "TinyTorrent: Combining BitTorrent and Sensor-Nets," Technical Report TCD-CS-2005-74, Univ. of Dublin, Trinity College, Dec. 2005.

Theresa C. Maxino received the BS degree in computer engineering from the University of San Carlos, Cebu City, Philippines, in 1995 and the MS degree in electrical and computer engineering from Carnegie Mellon University, Pittsburgh, in 2006. She was with Advanced World Systems, Inc., for a number of years, working on the research and development of laser printers, specifically on the printer operating system and the printer controller. She is currently with Lexmark Research and Development Corporation and she was previously with Carnegie Mellon University. Her research interests lie in embedded systems, dependability, and security. She is a member of the IEEE.

Philip J. Koopman received the BS and MEng degrees in computer engineering from Rensselaer Polytechnic Institute in 1982 and the PhD degree in computer engineering from Carnegie Mellon University, Pittsburgh, in 1989. He was a US Navy submarine officer. During several years in industry, he was a CPU designer for Harris Semiconductor and an embedded systems researcher for United Technologies. Since 1996, he has been with Carnegie Mellon University, where he is currently an associate professor. His research interests include dependability, safety critical systems, distributed real-time embedded systems, and embedded systems education. He is a member of the ACM and a senior member of the IEEE.