

# *Tutorial: Checksum and CRC Data Integrity Techniques for Aviation*

*May 9, 2012*

Philip Koopman  
Carnegie Mellon University  
koopman@cmu.edu

Co-PIs:  
Kevin Driscoll  
Brendan Hall  
Honeywell Laboratories

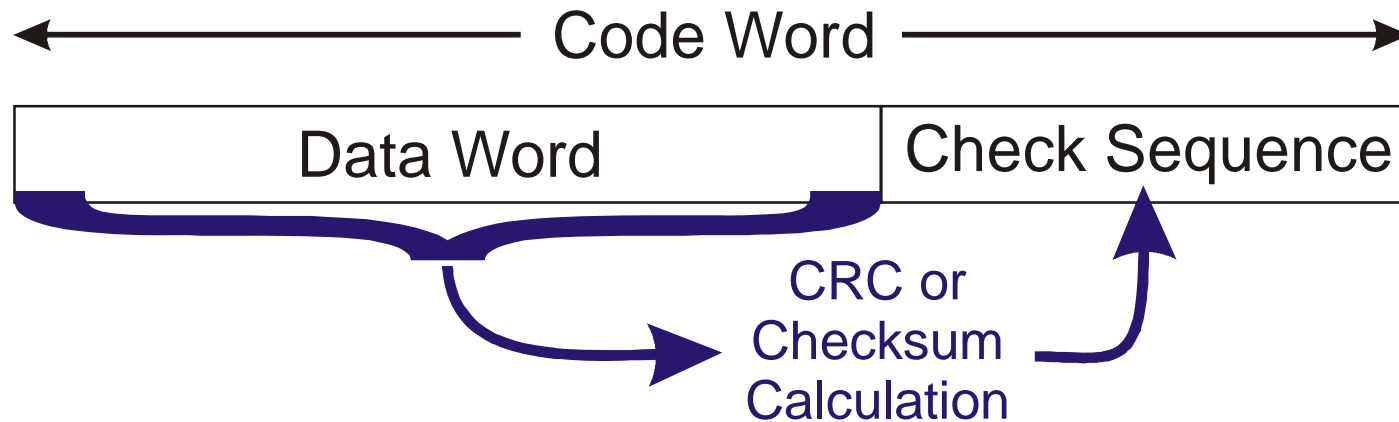
The views and opinions expressed in this presentation are those of the author, and are not necessarily those of the Federal Aviation Administration, who sponsored this work under contract DTFAC-11-C-00005. This presentation does not contain any technical conclusions funded by this work.

# Agenda

- Introduction
  - Motivation – why isn't this a solved problem?
  - Parity computations as an example
  - Error code construction and evaluation (without scary math)
  - Example using parity codes
- Checksums
  - What's a checksum?
  - Commonly used checksums and their performance
- Cyclic Redundancy Codes (CRCs)
  - What's a CRC?
  - Commonly used CRC approaches and their performance
- Don't blindly trust what you hear on this topic
  - A good CRC is almost always **much** better than a good Checksum
  - Many published (and popular) approaches are suboptimal or just plain wrong
  - There are some topics to be careful of because we don't know the answers
- Q&A

# Checksums and CRCs Protect Data Integrity

- Compute check sequence when data is transmitted or stored
  - **Data Word**: the data you want to protect (can be any size; often Mbytes)
  - **Check Sequence**: the result of the CRC or checksum calculation
  - **Code Word** = Data Word with Check Sequence Appended



- To check data integrity:
  - Retrieve or receive Code Word
  - Compute CRC or checksum on the received Data Word
  - If computed value equals Check Sequence then no data corruption found
    - (There might be data corruption! But if there is, you didn't detect it.)

# Potential CRC/Checksum Usage Scenarios

- Network packet integrity check
- Image integrity check for software update
- Boot-up integrity check of program image
  - e.g., flash memory data integrity check
- FPGA configuration integrity check
- Configuration integrity check
  - e.g., operating parameters in EEPROM
- RAM value integrity check

# Why Is This A Big Deal?

- Checksums are pretty much as good as CRCs, right?
  - In a word – **NO!**
  - Typical studies of checksums compare them to horrible CRCs
  - Would you prefer to detect all 1 & 2-bit errors (checksum) or all possible 1, 2, 3, 4, 5-bit errors (CRC) for about the same cost?
- CRCs have been around since 1957 – aren't they a done deal?
  - In a word – **NO!**
  - There wasn't enough compute power to find optimal CRCs until recently... so early results are often *not* very good
  - There is a lot of incorrect writing on this topic ... that at best assumes the early results were good
  - Many widespread uses of CRCs are mediocre, poor, or broken
- Our goal today is to show you where the state of the art really is
  - And to tune up your sanity check detector on this topic
  - Often you can get *many orders of magnitude better error detection* simply by using a good CRC at about the same cost

# Error Coding For Poets (who know a little discrete math)

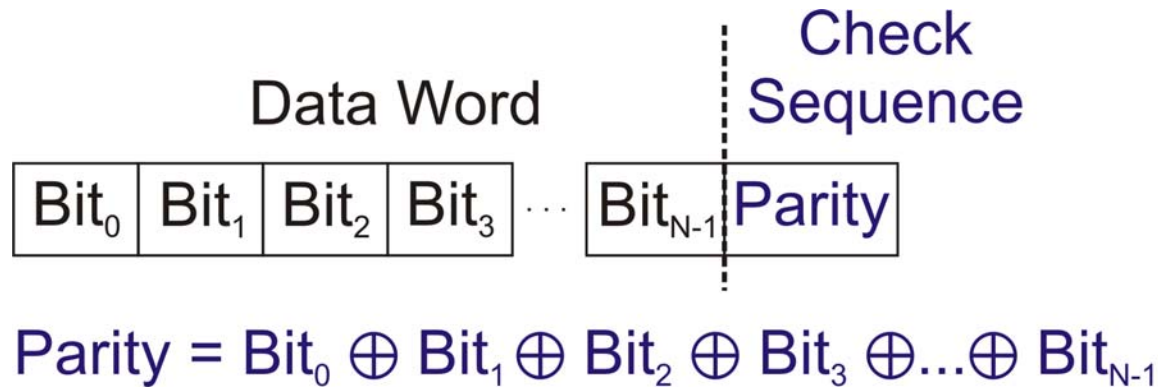
- The general idea of an error code is to mix all the bits in the data word to produce a condensed version (the check sequence)
  - Ideally, every bit in the data word affects many check sequence bits
  - Ideally, bit errors in the code word have high probability of being detected
  - Ideally, more probable errors with only a few bits inverted detected 100% of the time
  - At a hand-wave, similar to desired properties of a pseudo-random number generator
    - The Data Word is the seed value, and the Check Sequence is the pseudo-random number



- The ability to do this will depend upon:
  - **The size of the data word**
    - Larger data words are bigger targets for bit errors, and are harder to protect
  - **The size of the check sequence**
    - More check sequence bits makes it harder to get unlucky with multiple bit errors
  - **The mathematical properties of the “mixing” function**
    - Thorough mixing of data bits lets the check sequence detect simple error patterns
  - **The type of errors you expect to get** (patterns, error probability)

# Example: Parity As An Error Detection Code

- Example error detection function: **Parity**
  - XOR all the bits of the data word together to form 1 bit of parity



Note:  $\oplus$  is eXclusive OR (XOR)

Parity = 0 for even number of "1" bits

Parity = 1 for odd number of "1" bits

## XOR Facts:

$$0 \oplus 0 = 0$$

$$0 \oplus 1 = 1$$

$$1 \oplus 0 = 1$$

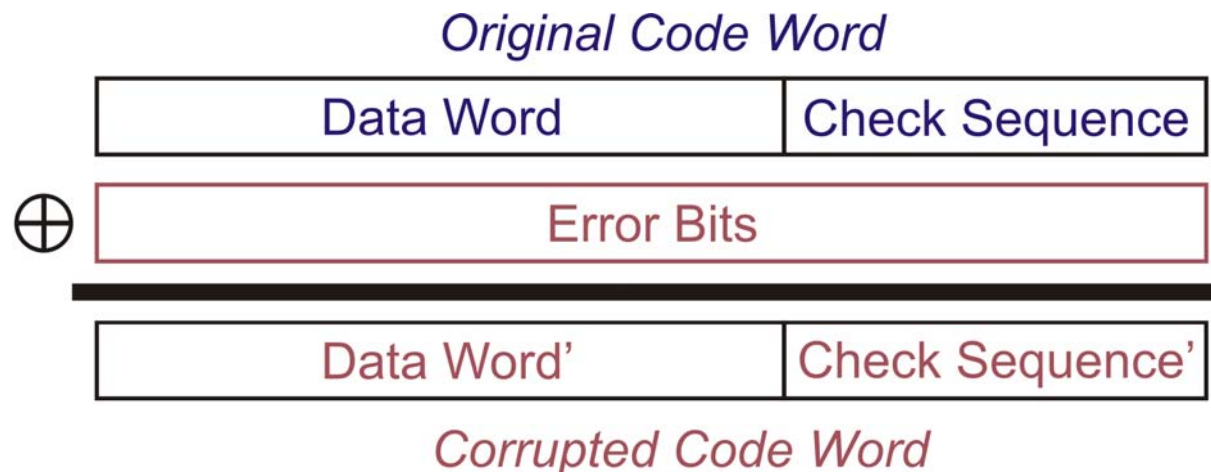
$$1 \oplus 1 = 0$$

(Think of it as Boolean addition and subtraction.)

- How good is this at error detection?
  - Only costs one bit of extra data; all bits included in mixing
  - Detects all odd number of bit errors (1, 3, 5, 7, ... bits in error)
  - Detects NO errors that flip an even number of bits (2, 4, 6, ... bits in error)
  - Performance: detects up to 1 bit errors; misses all 2-bit errors
  - Not so great – can we do better?

# Basic Model For Data Corruption

- Data corruption is “bit flips” (“bisymmetric inversions”)
  - Each bit has some probability of being inverted
  - “**Weight**” of error word is number of bits flipped (number of “1” bits in error)

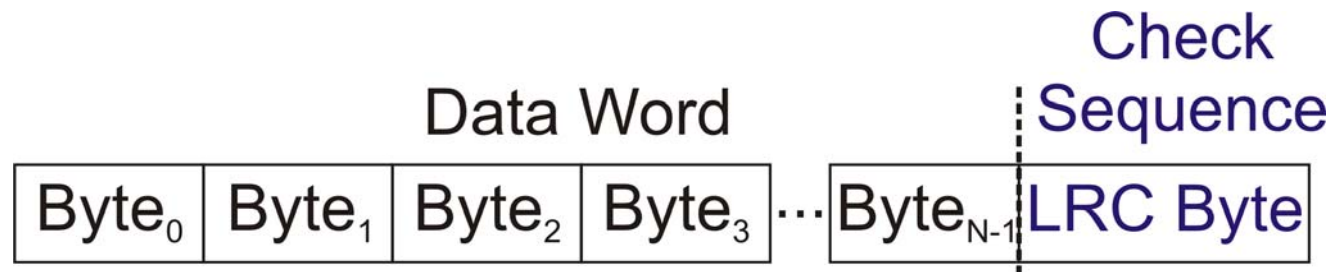


- Error detection works if the corrupted Code Word is invalid
  - In other words, if corrupted Check Sequence doesn't match the Check Sequence that would be computed based on the Data Word
  - If corrupted Check Sequence just happens to match the Check Sequence computed for corrupted data, you have an **undetected error**
  - All things being equal (which they are **not!!!**) probability of undetected error is 1 chance in  $2^k$  for a k-bit check sequence



# Example: Longitudinal Redundancy Check (LRC)

- LRC is a byte-by-byte parity computation
  - XOR all the bytes of the data word together, creating a one-byte result
  - (This is sometimes called an “XOR checksum” but it isn’t really integer addition, so it’s not quite a “sum”)



Example:

	0	0	1	0	0	1	0	0
⊕	1	0	1	1	1	0	0	0
⊕	1	1	1	1	1	1	1	1
⊕	0	0	0	0	0	0	0	1
<hr/>								
	0	1	1	0	0	0	1	0

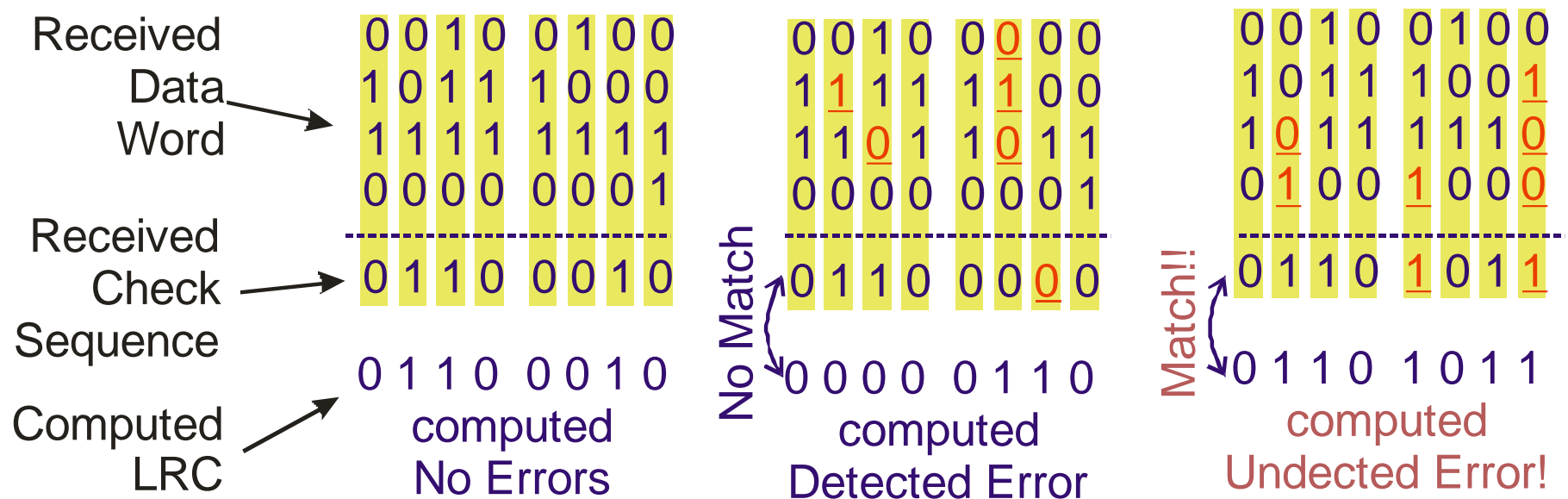
*Result is parity of each vertical bit slice*

$$\begin{array}{l}
 \text{Byte}_0 \\
 \oplus \text{Byte}_1 \\
 \oplus \text{Byte}_2 \\
 \oplus \text{Byte}_3 \\
 \oplus \dots \\
 \oplus \text{Byte}_{N-1} \\
 \hline
 \text{LRC BYTE}
 \end{array}$$

# How Good Is An LRC?

- Parity is computed for each bit position (vertical stripes)
  - Note that the **received copy of check sequence** can be corrupted too!

Red bits are transmission or storage errors



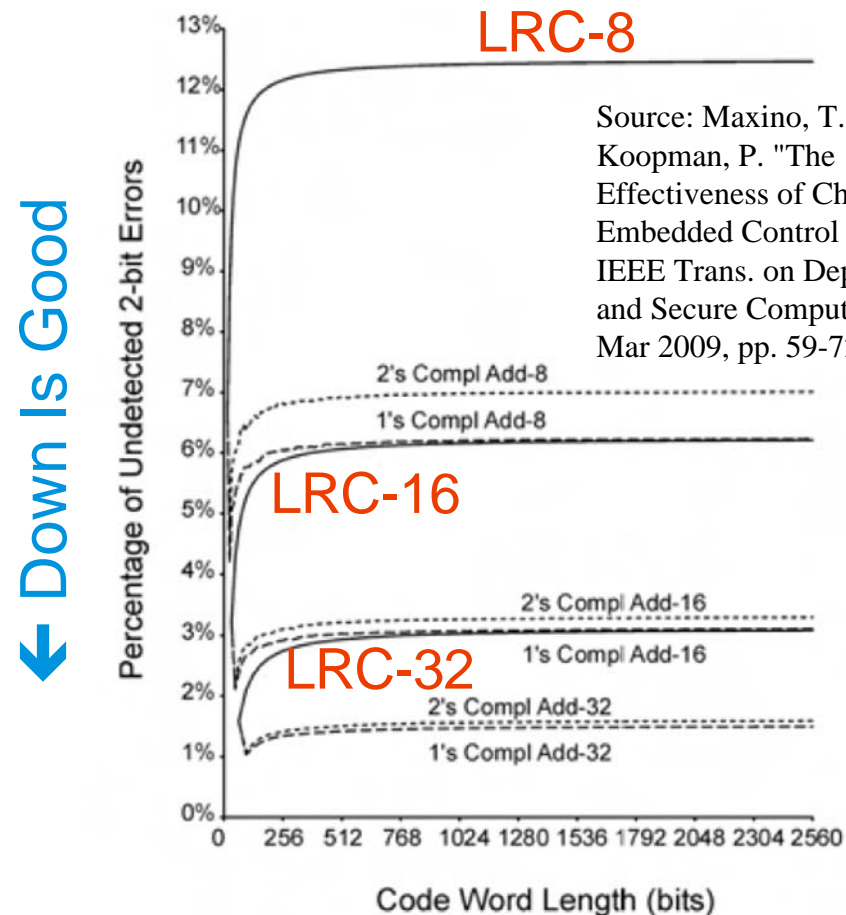
- Detects all odd numbers of bit errors in a vertical slice
  - Fails to detect even number of bit errors in a vertical slice
  - Detects all 1-bit errors; Detects all errors within a single byte
  - Detects many 2-bit errors, **but not all 2-bit errors**
    - Any 2-bit error in same vertical slice is undetected

# Error Code Effectiveness Measures

- Metrics that matter depend upon application, but usual suspects are:
  - **Maximum weight** of error word that is 100% detected
    - **Hamming Distance (HD)** is lowest weight of any undetectable error
    - For example, HD=4 means all 1, 2, 3 bit errors detected
  - **Fraction of errors** undetected for a given number of bit flips
    - Hamming Weight (HW): how many of all possible m-bit flips are undetected?
      - E.g. HW(5)=157,481 undetected out of all possible 5-bit flip Code Word combinations
  - **Fraction of errors** undetected at a given random probability of bit flips
    - Assumes a **Bit Error Ratio (BER)**, for example 1 bit out of 100,000 flipped
    - Small numbers of bit flips are most probable for typical BER values
  - **Special patterns** 100% detected, such as adjacent bits
    - Burst error detection – e.g., all possible bit errors within an 8 bit span
  - Performance usually depends upon data word size and code word size
- Example for LRC8 (8 bit chunk size LRC)
  - HD=2 (all 1 bit errors detected, not all 2 bit errors)
  - Detects all 8 bit bursts (only 1 bit per vertical slice)
  - Other effectiveness metrics coming up...

# LRC-8 Fraction of Undetected Errors

- Shows Probability of Undetected 2-bit Errors for:
  - LRC
  - Addition checksum
  - 1's complement addition checksum
- 8-bit addition checksum is almost as good as 16-bit-LRC!
  - So we can do better for sure

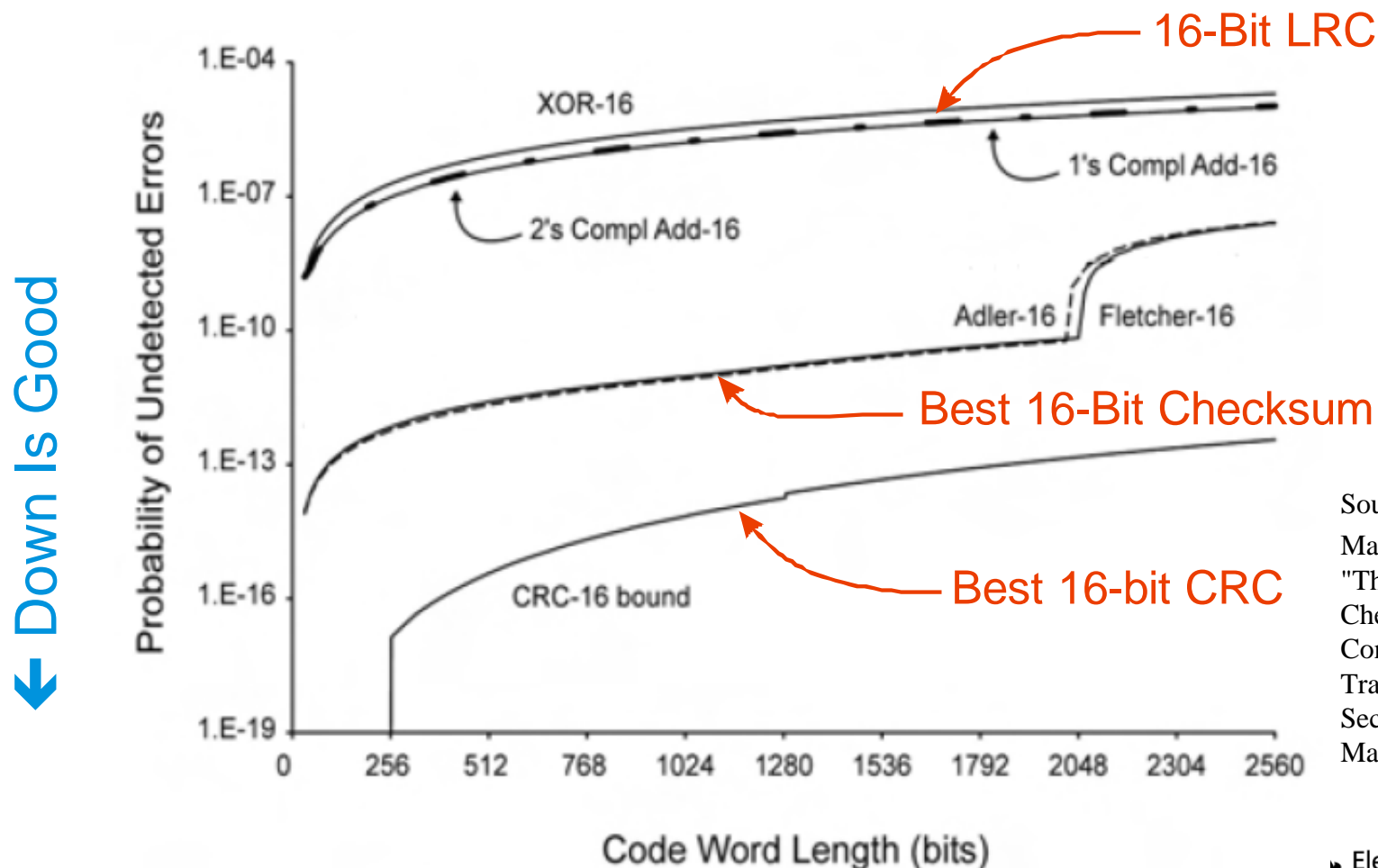


Source: Maxino, T., & Koopman, P. "The Effectiveness of Checksums for Embedded Control Networks," IEEE Trans. on Dependable and Secure Computing, Jan-Mar 2009, pp. 59-72.

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.

# Can We Do Even Better? YES!

- Can often get HD=6 (detect all 1, 2, 3, 4, 5-bit errors) with a CRC
- For this graph, assume Bit Error Rate (BER) =  $10^{-5}$  flip probability per bit



Source:

Maxino, T., & Koopman, P.  
 "The Effectiveness of  
 Checksums for Embedded  
 Control Networks," IEEE  
 Trans. on Dependable and  
 Secure Computing, Jan-  
 Mar 2009, pp. 59-72.

# Checkpoint – What's Coming Next

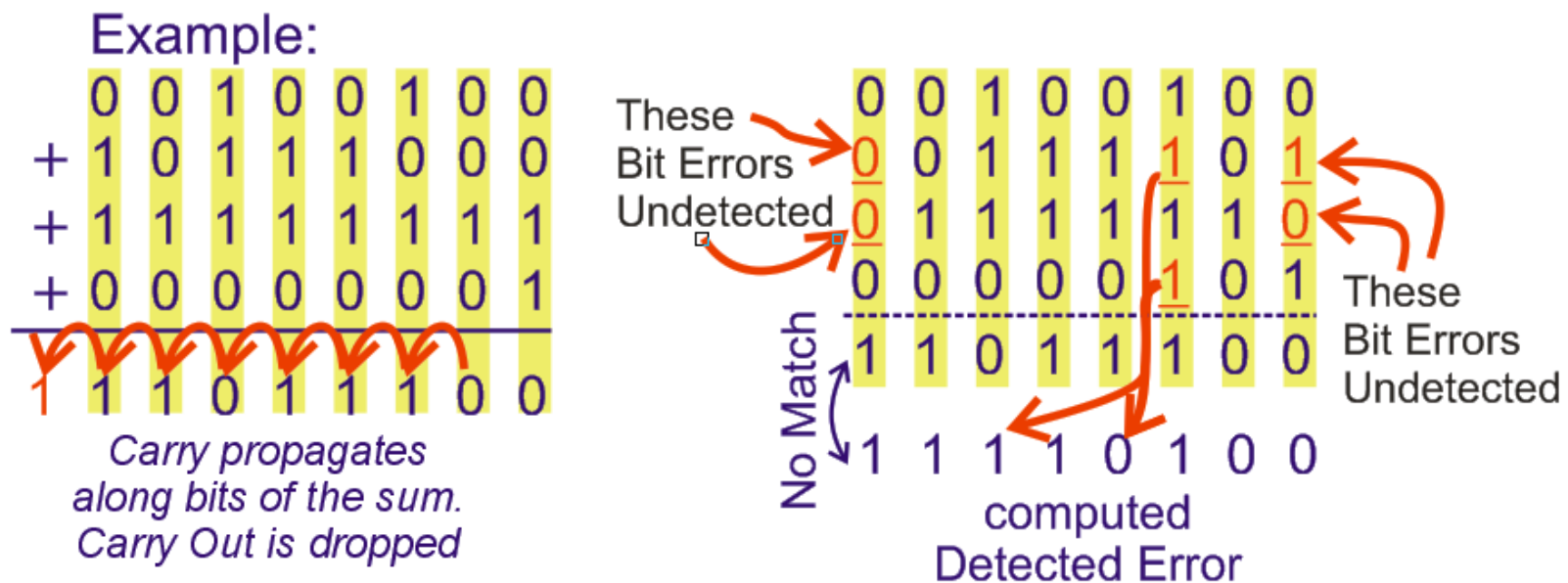
- You now have basic vocabulary and background
- Let's talk about better ways to detect errors
  - Checksums
  - Cyclic Redundancy Codes (CRCs)
  - Evaluation strategies
  - Pitfalls

# Checksums

- A checksum “adds” together “chunks” of data
  - The “add” operation may not be normal integer addition
  - The chunk size is typically 8, 16, or 32 bits
- We’ll discuss:
  - Integer addition “checksum”
  - One’s complement “checksum”
  - Fletcher Checksum
  - Adler Checksum
  - ATN Checksum (AN/466)

# Integer Addition Checksum

- Same as LRC, except use integer “+” instead of XOR
  - The carries from addition promote bit mixing between adjacent columns
    - Can detect errors that make two bits go  $0 \rightarrow 1$  or  $1 \rightarrow 0$  (except top-most bits)
    - Cannot detect compensating errors (one bit goes  $0 \rightarrow 1$  and another  $1 \rightarrow 0$ )
  - Carry out of the top bit of the sum is discarded
    - No pairs of bit errors are detected in top bit position

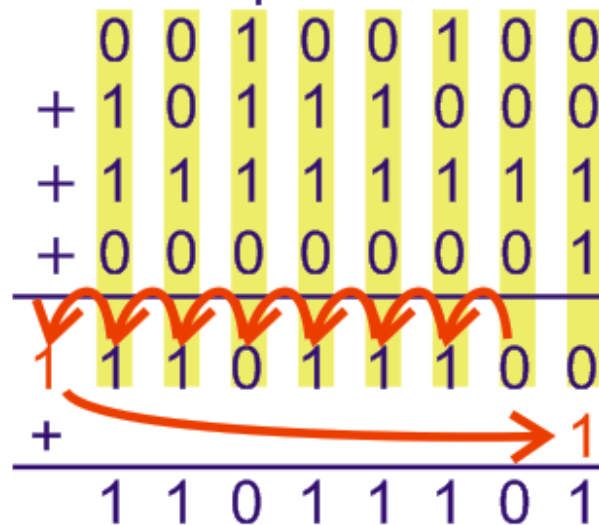




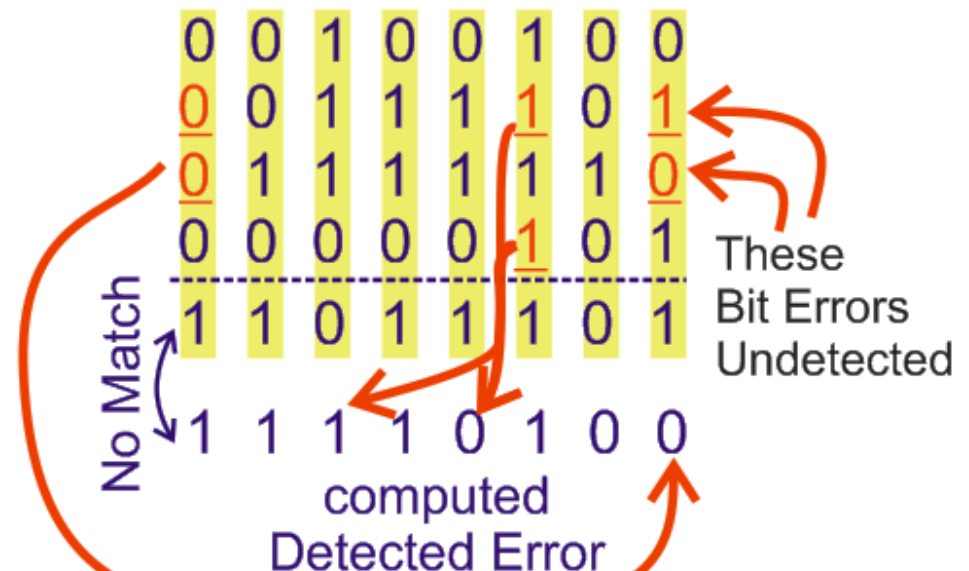
# One's Complement Addition Checksum

- Same as integer checksum, but add Carry-Out bits back
  - Plugs error detection hole of two top bits flipping with the same polarity
  - But, doesn't solve problem of compensating errors
  - Hamming Distance 2 (HD=2); some two-bit errors are undetected

### Example:



Carry propagates  
along bits of the sum.  
Carry Out is dropped



Corrupted bits eliminate carry-out; detected with lowest bit

# Fletcher Checksum

- Use two running one's complement checksums
  - For fair comparison, each running sum is half width
  - E.g., 16-bit Fletcher Checksum is two 8-bit running sums
  - Initialize:  $A = 0; B = 0;$
  - For each byte in data word:  $A = A + \text{Byte}_i; B = B + A;$ 
    - One's complement addition!
  - Result is A concatenated with B (16-bit result)
- Significant improvement comes from the running sum B
  - $B = \text{Byte}_{N-1} + 2 * \text{Byte}_{N-2} + 3 * \text{Byte}_{N-3} + \dots$
  - Makes checksum order-dependent (switched byte order detected)
  - **Gives HD=3** until the B value rolls over
    - For example,  $256 * \text{Byte}_{N-256}$  does not affect B

# Adler Checksum

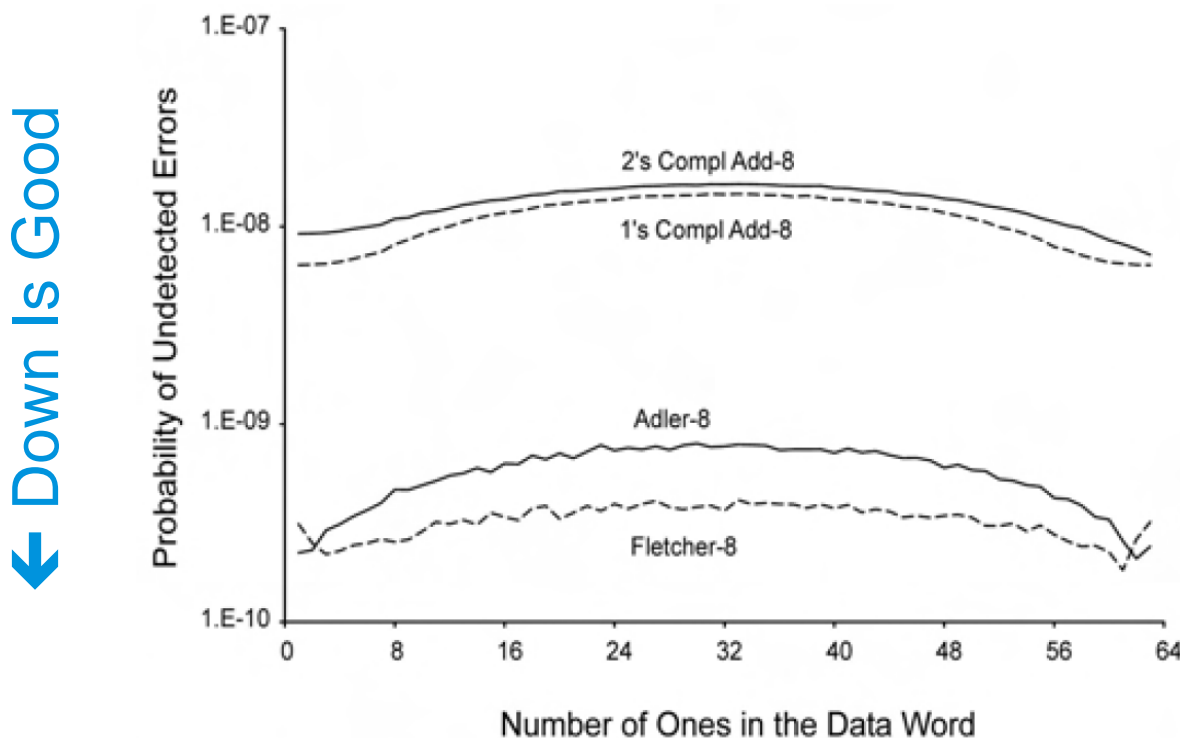
- Intended to be an improvement on Fletcher Checksum
  - One's complement addition is the same as modulo 255 addition
  - Adler checksum uses a prime integer as a modulus
    - 251 instead of 255 for Adler 16 (two 8-bit sums)
    - 65521 instead of 65535 for Adler 32 (two 16-bit sums)
- In practice, it is not worth it
  - For most sizes and data lengths Adler is worse than Fletcher
  - In the best case it is only very slightly better
    - But computation is more expensive because of the modular sum

# ATN-32 Checksum [AN/466]

- Aviation-specific riff on Fletcher Checksum
  - Four running 1-byte sums (one's complement addition)
  - Potentially gives good mixing for 8-bit data chunks
- Algorithm:
  - Initialize C0, C1, C2 and C3 to zero
  - For each Data Word byte:  
 $C0 += \text{Byte}_i; \quad C1 += C0; \quad C2 += C1; \quad C3 += C2;$   
(one's complement addition, as with Fletcher checksum)
  - 32-bit check sequence is a particular formula of C0..C3
- No apparent published analysis of error detection results
  - Standard says it provides good protection, but no quantitative assessment
  - We'll take a look at this and other relevant error codes in our study

# Checksum Performance Is Data Dependent

- The data values affect checksum performance
  - Worst-case performance is equal number of zeros and ones
  - Below is 64-bit data word and BER of  $10^{-5}$



Source:

Maxino, T., & Koopman, P. "The Effectiveness of Checksums for Embedded Control Networks," IEEE Trans. on Dependable and Secure Computing, Jan-Mar 2009, pp. 59-72.

- This means need to take into account data values when assessing performance

# Cyclic Redundancy Codes (CRCs)

- **CRCs Use Division Instead Of Addition**
- Intuitive description:
  - Addition does OK but not great mixing of Data Word bits
  - What about using the remainder after division instead?
- Integer analogy: remainder after integer division
  - $2,515,691,591 \bmod 251 = 166$   $\leftarrow$  8-bit check sequence
    - Any simple change to the input number (Data Word) changes remainder
  - But, need to pick a clever divisor
    - E.g.,  $2,515,691,591 \bmod 100 = 91$   $\leftarrow$  unaffected by most digits
    - Probably want something like prime number 251, but may be more complex than that to avoid “wasting” result values of 252, 253, 254, 255
  - ISBNs use this technique for the last digit, with divisor of 11
    - An “X” at the end of an ISBN means the remainder was 10 instead of 0..9
  - Also, want something that is efficient to do in SW & HW
    - Original CRCs were all in hardware to maximize speed and minimize hardware cost

# Mathematical Basis of CRCs

- Use polynomial division (remember that from high school?)  
over Galois Field(2) (this is a mathematician thing)
  - At a hand-waving level this is division using Boolean Algebra
    - “Add” and “Subtract” used by division algorithm both use XOR

```

11010011101100 000 <--- Data Word left shifted by 3 bits
1011               <--- 4-bit divisor is 1011  $x^3 + x + 1$ 
01100011101100 000 <--- result of first conditional subtraction
 1011             <--- divisor
00111011101100 000 <--- result of second conditional subtraction
 1011             <--- continue shift-and-subtract ...
00010111101100 000
 1011
00000001101100 000
 1011
00000000110100 000
 1011
00000000011000 000
 1011
00000000001110 000
 1011
00000000000101 000
 101 1
-----
00000000000000 100 <--- Remainder (3 bits)

```

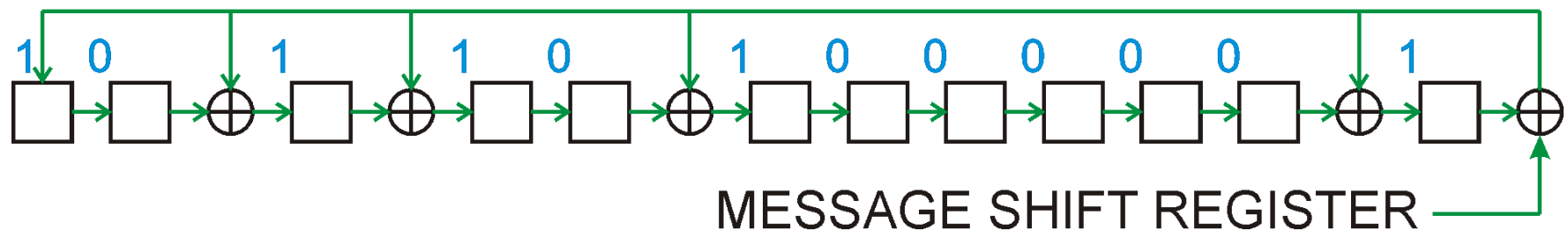
[Wikipedia]

Remainder is the Check Sequence

# Hardware View of CRC

- CRC also has a clever hardware implementation:
  - The feedback “polynomial” is the divisor; shift register holds remainder

POLYNOMIAL: 1011 0100 0001 = 0xB41



Example Feedback Polynomial:

$$0xB41 = x^{12} + x^{10} + x^9 + x^7 + x + 1 \quad (\text{the “+1” is implicit in the hex value})$$

$$= (x+1)(x^3 + x^2 + 1)(x^8 + x^4 + x^3 + x^2 + 1)$$

- The tricky part is in picking the right Feedback Polynomial (divisor)
  - The best ones are not necessarily “prime” (irreducible) nor “primitive”
  - A lot of what is published on this topic has problems



# A Typical Legacy CRC Selection Method

An  $M$ -bit long CRC is based on a primitive polynomial of degree  $M$ , called the generator polynomial. Alternatively, the generator is chosen to be a primitive polynomial times  $(1 + x)$  (this finds all parity errors). For 16-bit CRC's, the CCITT (Comité Consultatif International Télégraphique et Téléphonique) has anointed the "CCITT polynomial," which is  $x^{16} + x^{12} + x^5 + 1$ . This polynomial is used by all of the protocols listed in the table. Another common choice is the "CRC-16" polynomial  $x^{16} + x^{15} + x^2 + 1$ , which is used for EBCDIC messages in IBM's BISYNCH [1]. A common 12-bit choice, "CRC-12," is  $x^{12} + x^{11} + x^3 + x + 1$ . A common 32-bit choice, "AUTODIN-II," is  $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$ . For a table of some other primitive polynomials, see §7.4.

» *Numerical Recipes in C*, Press et al. 1992

- But, there are some problems:
  - Many good polynomials are not primitive nor divisible by  $(x+1)$
  - Divisibility by  $(x+1)$  doubles undetected error rate for even # of bit errors

# A Typical Polynomial Selection Method

An  $M$ -bit long CRC is based on a primitive polynomial of degree  $M$ , called the generator polynomial. Alternatively, the generator is chosen to be a primitive polynomial times  $(1 + x)$  (this finds all parity errors). For 16-bit CRC's, the CCITT (Comité Consultatif International Télégraphique et Téléphonique) has anointed the “CCITT polynomial,” which is  $x^{16} + x^{12} + x^5 + 1$ . This polynomial is used by all of the protocols listed in the table. Another common choice is the “CRC-16” polynomial  $x^{16} + x^{15} + x^2 + 1$ , which is used for EBCDIC messages in IBM's BISYNCH [1]. A common 12-bit choice, “CRC-12,” is  $x^{12} + x^{11} + x^3 + x + 1$ . A common 32-bit choice, “AUTODIN-II,” is  $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$ . For a table of some other primitive polynomials, see §7.4.

» *Numerical Recipes in C*, Press et al.

- But, there are some problems:
  - Many good polynomials are not primitive nor divisible by  $(x+1)$
  - Divisibility by  $(x+1)$  doubles undetected error rate for even # of bit errors
  - How do you know which competing polynomial to pick?

# A Typical Polynomial Selection Method

An  $M$ -bit long CRC is based on a primitive polynomial of degree  $M$ , called the generator polynomial. Alternatively, the generator is chosen to be a primitive polynomial times  $(1 + x)$  (this finds all parity errors). For 16-bit CRC's, the CCITT (Comité Consultatif International Télégraphique et Téléphonique) has anointed the “CCITT polynomial,” which is  $x^{16} + x^{12} + x^5 + 1$ . This polynomial is used by all of the protocols listed in the table. Another common choice is the “CRC-16” polynomial  $x^{16} + x^{15} + x^2 + 1$ , which is used for EBCDIC messages in IBM's BISYNCH [1]. A common 12-bit choice, “CRC-12,” is  $x^{12} + x^{11} + x^3 + x + 1$ . A common 32-bit choice, “AUTODIN-II,” is  $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$ . For a table of some other primitive polynomials, see §7.4.

» *Numerical Recipes in C*, Press et al.

- But, there are some problems:
  - Many good polynomials are not primitive nor divisible by  $(x+1)$
  - Divisibility by  $(x+1)$  doubles undetected error rate for even # of bit errors
  - How do you know which competing polynomial to pick?
  - This CRC-12 polynomial is incorrect (there is a missing  $+x^2$ )

# A Typical Polynomial Selection Method

An  $M$ -bit long CRC is based on a primitive polynomial of degree  $M$ , called the generator polynomial. Alternatively, the generator is chosen to be a primitive polynomial times  $(1 + x)$  (this finds all parity errors). For 16-bit CRC's, the CCITT (Comité Consultatif International Télégraphique et Téléphonique) has anointed the "CCITT polynomial," which is  $x^{16} + x^{12} + x^5 + 1$ . This polynomial is used by all of the protocols listed in the table. Another common choice is the "CRC-16" polynomial  $x^{16} + x^{15} + x^2 + 1$ , which is used for EBCDIC messages in IBM's BISYNCH [1]. A common 12-bit choice, "CRC-12," is  $x^{12} + x^{11} + x^3 + x + 1$ . A common 32-bit choice, "AUTODIN-II," is  $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$ . For a table of some other primitive polynomials, see §7.4.

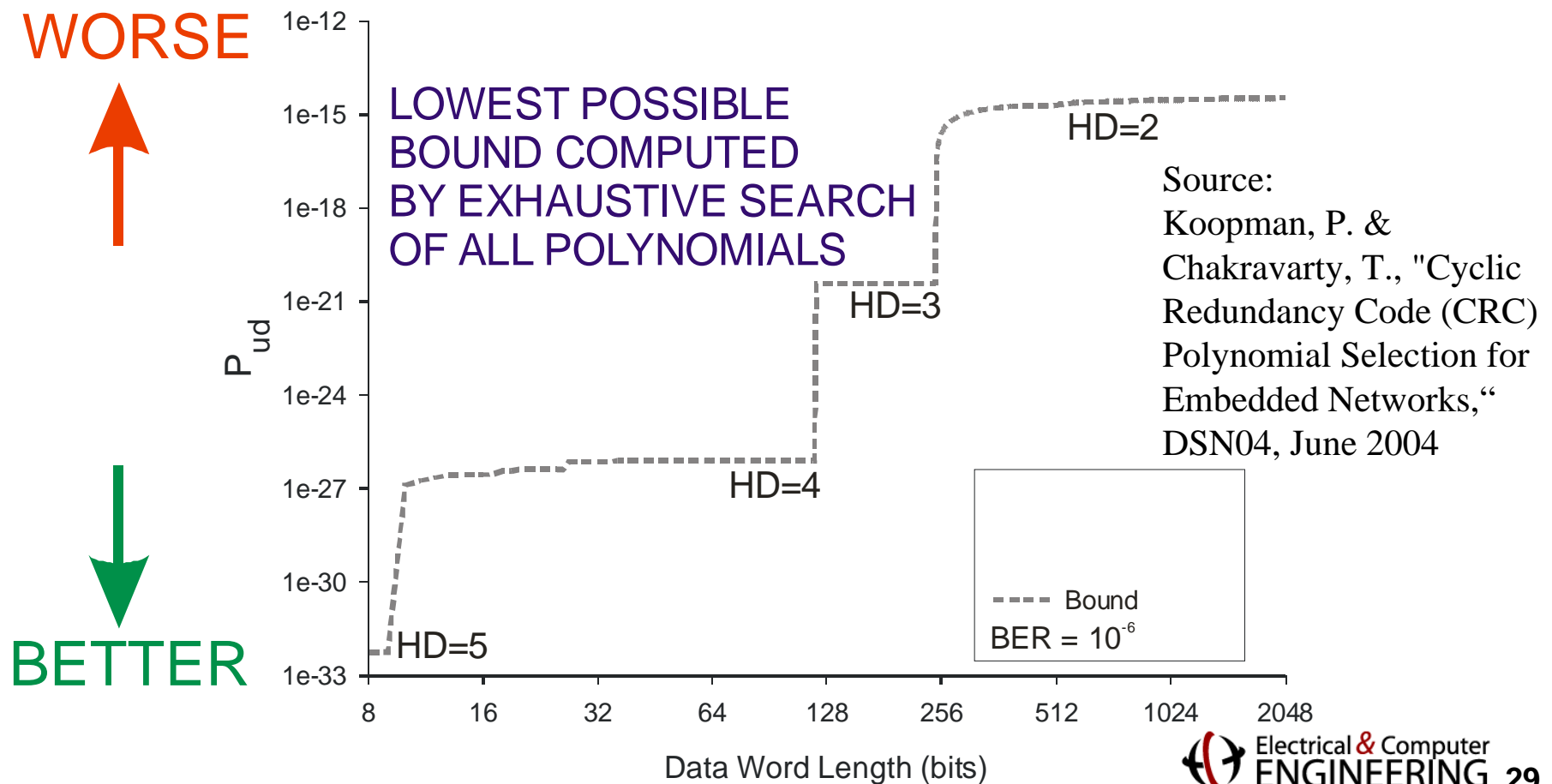
» *Numerical Recipes in C*, Press et al.

- But, there are some problems:
  - Many good polynomials are not primitive nor divisible by  $(x+1)$
  - Divisibility by  $(x+1)$  doubles undetected error rate for even # of bit errors
  - How do you know which competing polynomial to pick?
  - This CRC-12 polynomial is incorrect (there is a missing  $+x^2$ )
  - You can't pick at random from a list!

**(BTW, 3<sup>rd</sup> edition has updated this material and gets it right)**

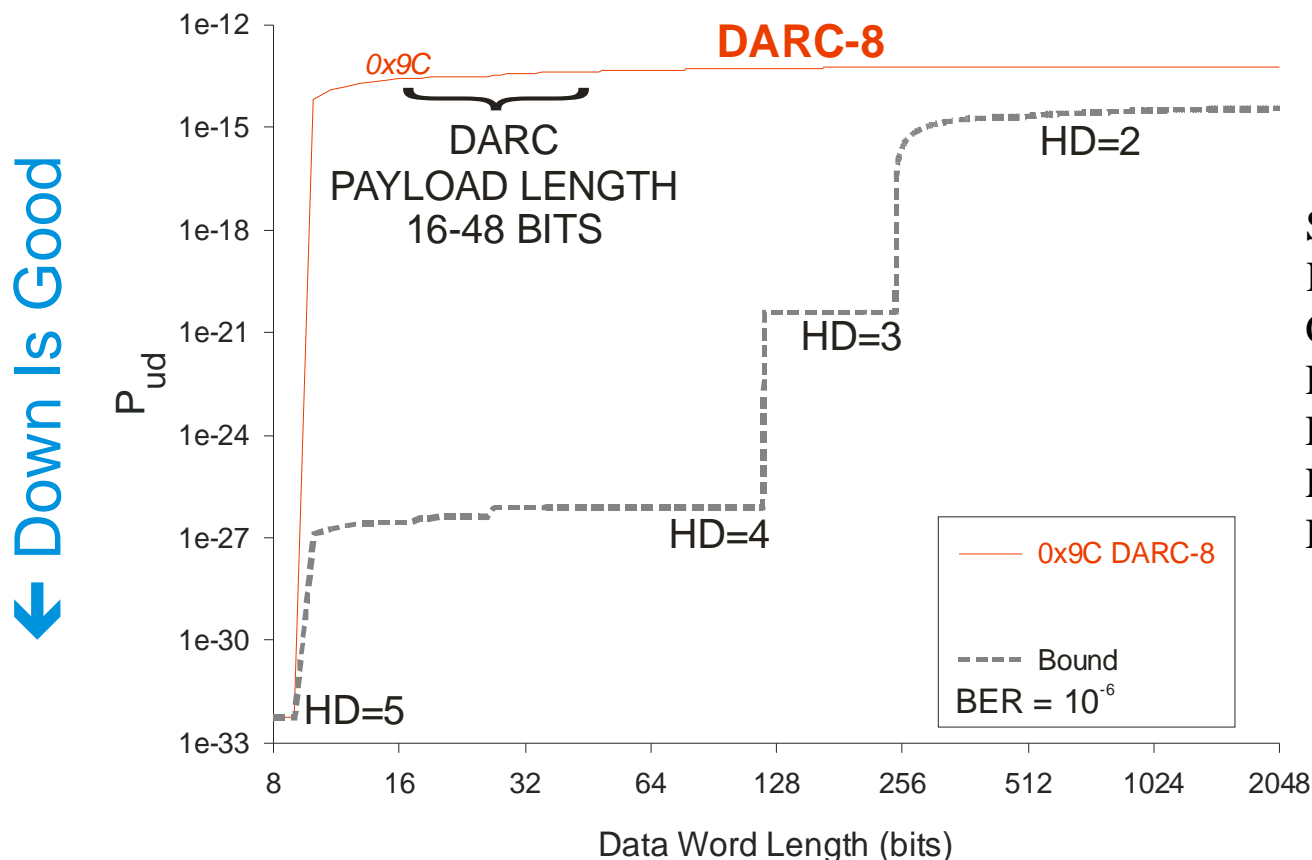
## Example – 8 Bit Polynomial Choices

- $P_{ud}$  (undetected error rate) is one way to evaluate CRC effectiveness
  - Uses Hamming weights of polynomials
  - Uses assumed random independent Bit Error Rate (BER)



# What Happens When You Get It Wrong?

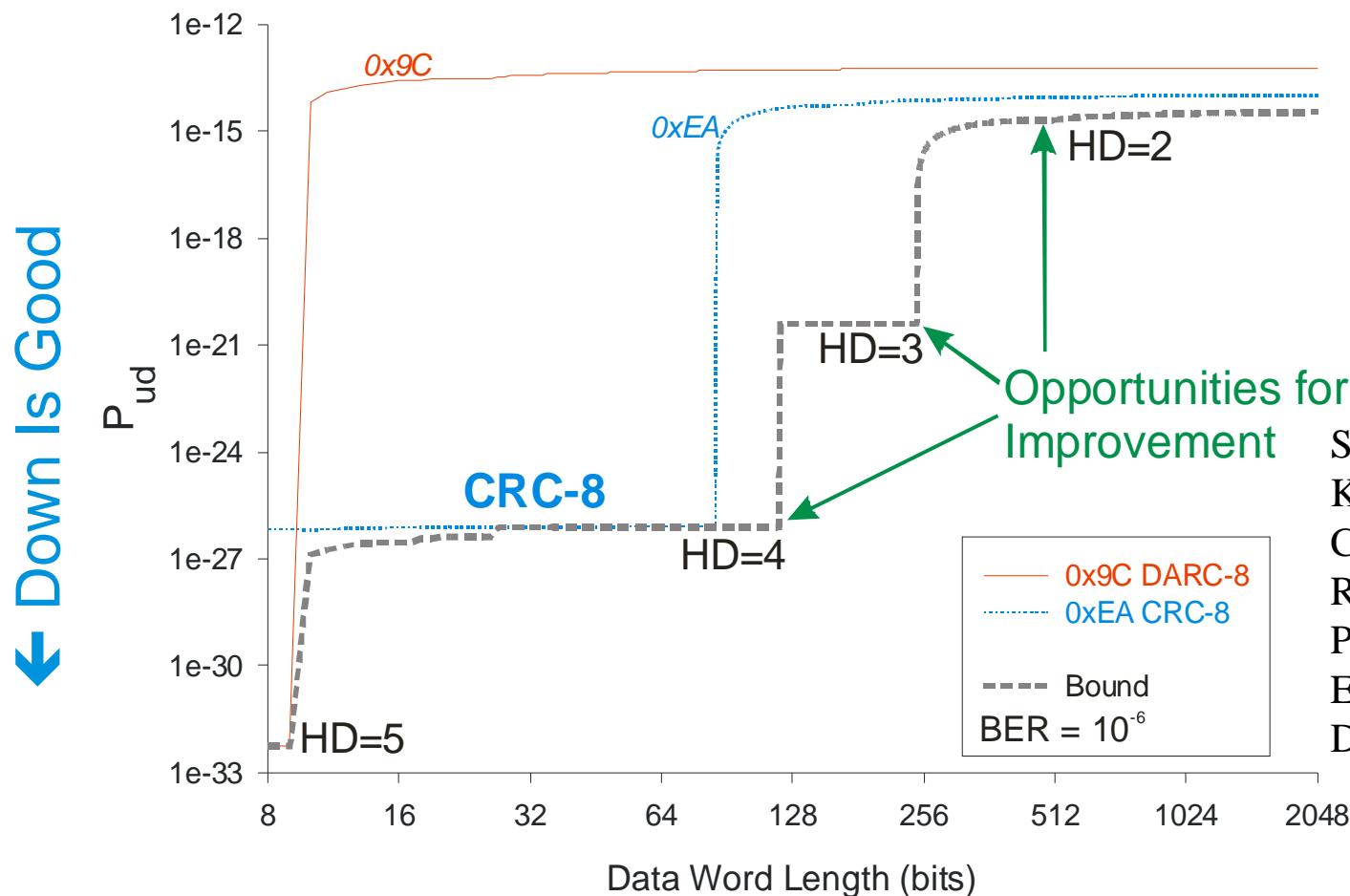
- DARC (Data Radio Channel), ETSI, October 2002
  - DARC-8 polynomial is optimal for 8-bit payloads
  - BUT, DARC uses 16-48 bit payloads, and misses some 2-bit errors
  - Could have detected all 2-bit and 3-bit errors with same size CRC!



Source:  
Koopman, P. &  
Chakravarty, T., "Cyclic  
Redundancy Code (CRC)  
Polynomial Selection for  
Embedded Networks,"  
DSN04, June 2004

# CRC-8 Is Better

- CRC-8 (0xEA) is in very common use
  - Good for messages up to size 85
  - But, room for improvement at longer lengths. Can we do better?

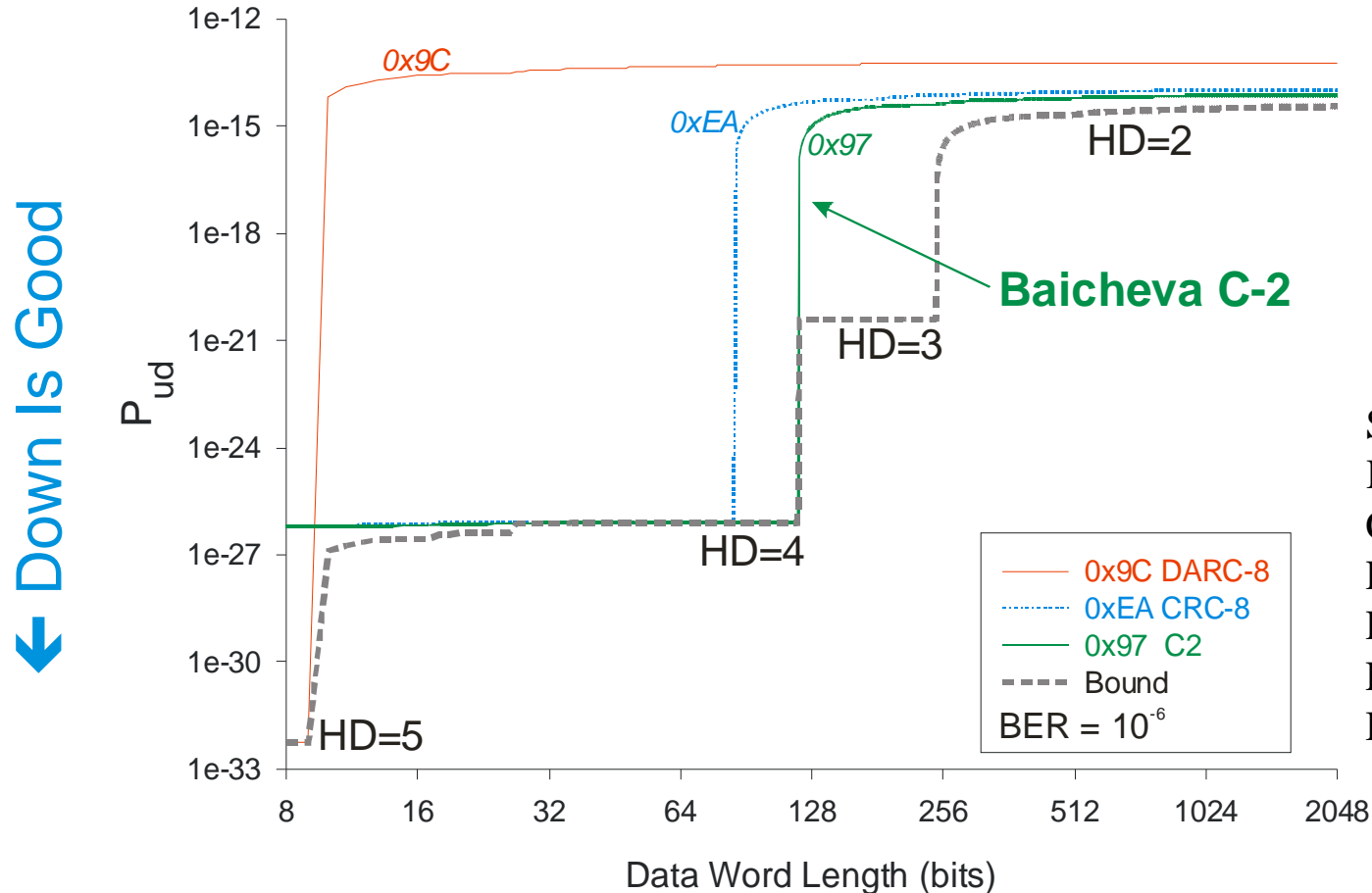


Source:  
Koopman, P. &  
Chakravarty, T., "Cyclic  
Redundancy Code (CRC)  
Polynomial Selection for  
Embedded Networks,"  
DSN04, June 2004



# Baicheva's Polynomial C2

- [Baicheva98] proposed polynomial C2, 0x97
  - Recommended as good polynomial to length 119
  - Dominates 0xEA (better  $P_{ud}$  at every length)

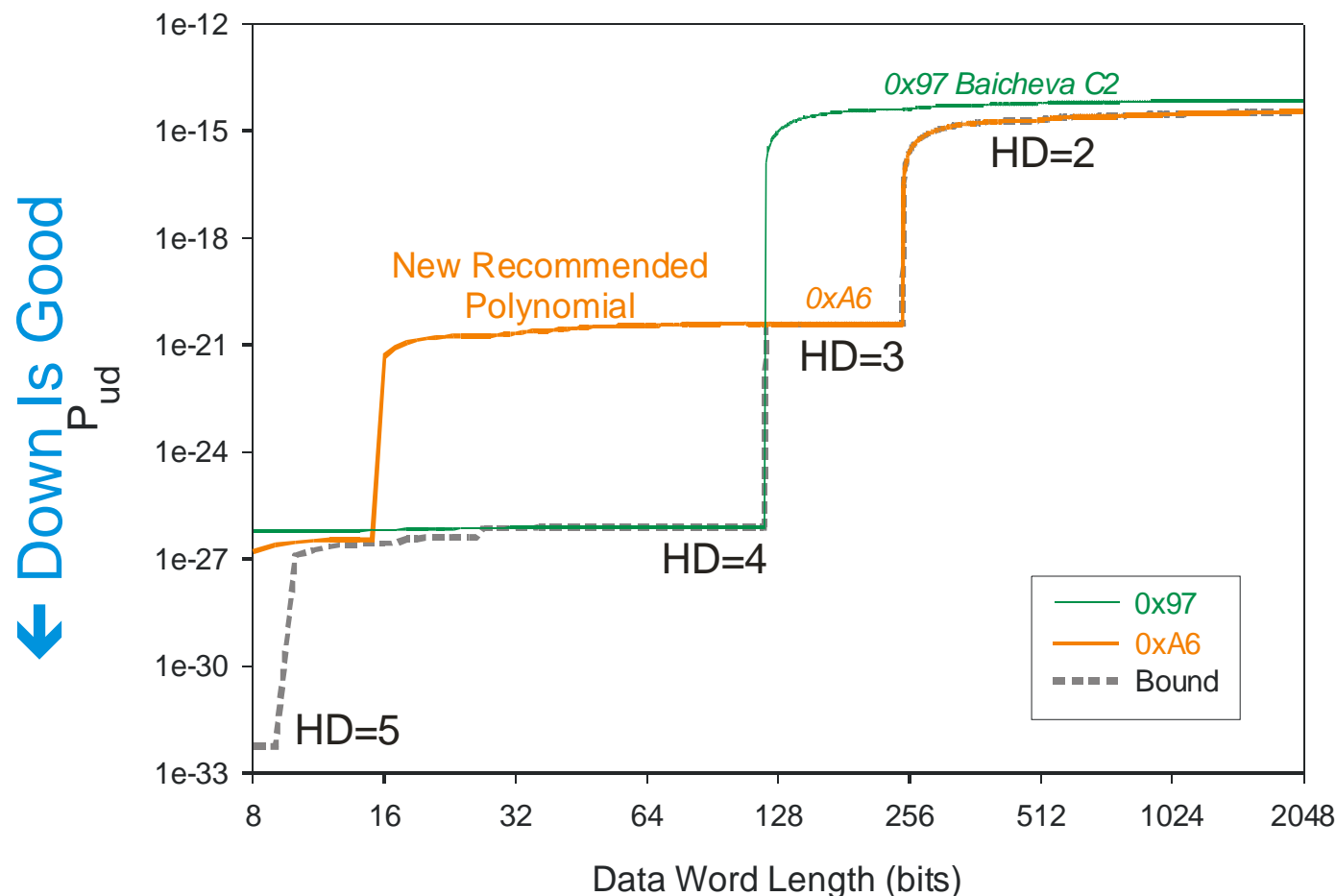


Source:  
Koopman, P. &  
Chakravarty, T., "Cyclic  
Redundancy Code (CRC)  
Polynomial Selection for  
Embedded Networks,"  
DSN04, June 2004



# But What If You Want the HD=3 Region?

- No previously published polynomials proposed for HD=3 region
  - We found that 0xA6 has good performance
  - Better than C2 and near optimal at all lengths of 120 and above



Source:  
Koopman, P. &  
Chakravarty, T., "Cyclic  
Redundancy Code (CRC)  
Polynomial Selection for  
Embedded Networks,"  
DSN04, June 2004

# Optimal Polynomials For Small CRCs

- P. Koopman, T. Chakravathy, “Cyclic Redundancy Code (CRC) Polynomial Selection for Embedded Networks”, The International Conference on Dependable Systems and Networks, DSN-2004.

**Table 3. “Best” polynomials for HD at given CRC size and data word length.**  
Underlined polynomials have been previously published as “good” polynomials.

Max length at HD Polynomial	CRC Size (bits)													
	3	4	5	6	7	8	9	10	11	12	13	14	15	16
HD=2	2048+ <u>0x5</u>	2048+ <u>0x9</u>	2048+ <u>0x12</u>	2048+ <u>0x21</u>	2048+ <u>0x48</u>	2048+ <u>0xA6</u>	2048+ <u>0x167</u>	2048+ <u>0x327</u>	2048+ <u>0x64D</u>	–	–	–	–	–
HD=3		11 <u>0x9</u>	26 <u>0x12</u>	57 <u>0x21</u>	120 <u>0x48</u>	247 <u>0xA6</u>	502 <u>0x167</u>	1013 <u>0x327</u>	2036 <u>0x64D</u>	2048 <u>0xB75</u>	–	–	–	–
HD=4			10 <u>0x15</u>	25 <u>0x2C</u>	56 <u>0x5B</u>	119 <u>0x97</u>	246 <u>0x14B</u>	501 <u>0x319</u>	1012 <u>0x583</u>	2035 <u>0xC07</u>	2048 <u>0x102A</u>	2048 <u>0x21E8</u>	2048 <u>0x4976</u>	2048 <u>0xBAAD</u>
HD=5						9 <u>0x9C</u>	13 <u>0x185</u>	21 <u>0x2B9</u>	25 <u>0x5D7</u>	53 <u>0x8F8</u>	none	113 <u>0x212D</u>	136 <u>0x6A8D</u>	241 <u>0xAC9A</u>
HD=6							8 <u>0x13C</u>	12 <u>0x28E</u>	22 <u>0x532</u>	27 <u>0xB41</u>	52 <u>0x1909</u>	57 <u>0x372B</u>	114 <u>0x573A</u>	135 <u>0xC86C</u>
HD=7									12 <u>0x571</u>	none	12 <u>0x12A5</u>	13 <u>0x28A9</u>	16 <u>0x5BD5</u>	19 <u>0x968B</u>
HD=8										11 <u>0xA4F</u>	11 <u>0x10B7</u>	11 <u>0x2371</u>	12 <u>0x630B</u>	15 <u>0x8FDB</u>

# More On Picking A Good CRC

- Important to select CRC polynomial based on:

- Data Word length
- Desired HD
- Desired CRC size

Safety-critical applications commonly select **HD=6** at max message length

- Good values also known for 24-bit and 32-bit polynomials

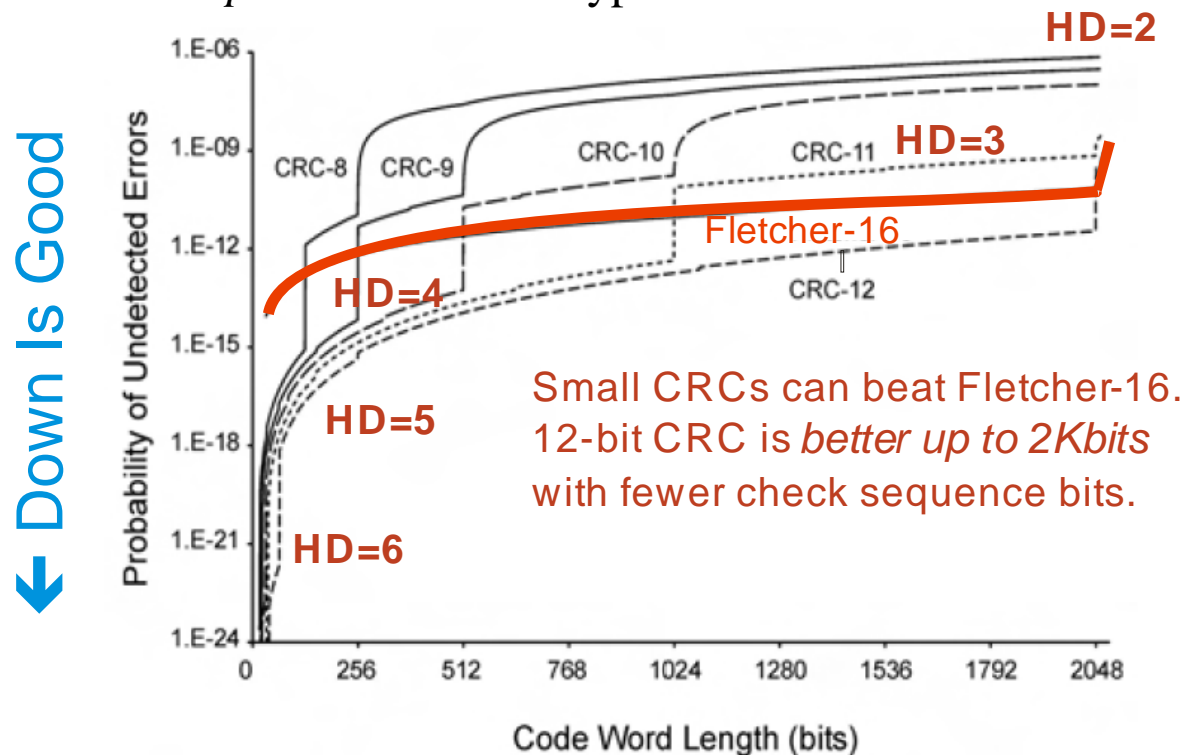
- IEEE 802.3 standard gives HD=6 up to 268-bit data words
- But 0xBA0DC66B gives HD=6 up to 16,360-bit data words
  - Koopman, P., "32-bit cyclic redundancy codes for Internet applications," International Conference on Dependable Systems and Networks (DSN), Washington DC, July 2002
- We're working on assembling these in a convenient format

- Be careful of published polynomials

- Get them from refereed publications, not the web
- Even then, double-check everything!
  - (We found a typo within the only published HD=6 polynomial value in an IEEE journal)
- A one-bit difference can change great → horrible
- Mapping polynomial terms to feedback bits can be tricky

# Are Checksums Or CRCs Better?

- Checksums can be slightly faster in software (this is usually overstated)
  - But tend to give far worse error performance
    - Most checksum folklore is based on comparing to a *bad* CRC or with *non-representative* fault types



Source:

Maxino, T., & Koopman, P.  
 "The Effectiveness of  
 Checksums for Embedded  
 Control Networks," IEEE  
 Trans. on Dependable and  
 Secure Computing, Jan-Mar  
 2009, pp. 59-72.

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.

# Aren't Software CRCs Really Slow?

- Speedup techniques have been known for years
  - Important to compare best implementations, not slow ones
  - Some CPUs now have hardware support for CRC computation
- 256-word lookup table provides about 4x CRC speedup
  - Careful polynomial selection gives 256-byte table and ~8x speedup
  - Intermediate space/speedup approaches can also be used
  - Ray, J., & Koopman, P. "Efficient High Hamming Distance CRCs for Embedded Applications," DSN06, June 2006.
- In a system with cache memory, CRCs are probably not a lot more expensive than a checksum
  - Biggest part of execution time will be getting data bytes into cache!
  - We are working on a more definitive speed tradeoff study

# Additional Checksum & CRC Tricks

- Use a “seed” value
  - Initialize Checksum or CRC register to other than zero
  - Prevents all-zero data word from resulting in all-zero check sequence
  - Can be used (with great care) to mitigate network masquerading
    - Transmitters with different seed values won’t “see” each others’ messages
- Be careful with bit ordering
  - CRCs provide burst error detection up to CRC size
  - Unless you get the order of bits wrong (as in Firewire)
  - Unless you put CRC at front instead of back of message
- CRC error performance is independent of data values
  - It is only the patterns of error bits that matter

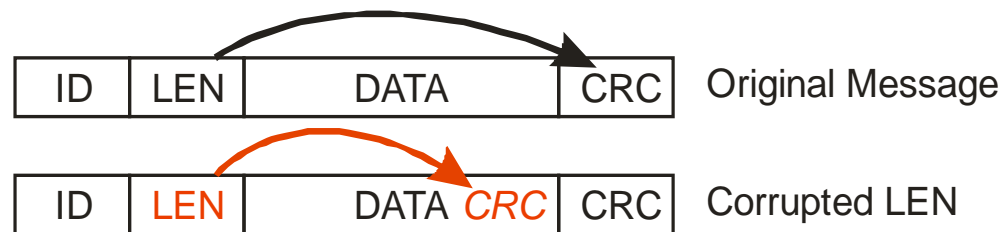
# Here There Be Dragons...

Other places to be wary (out of scope for our current research)

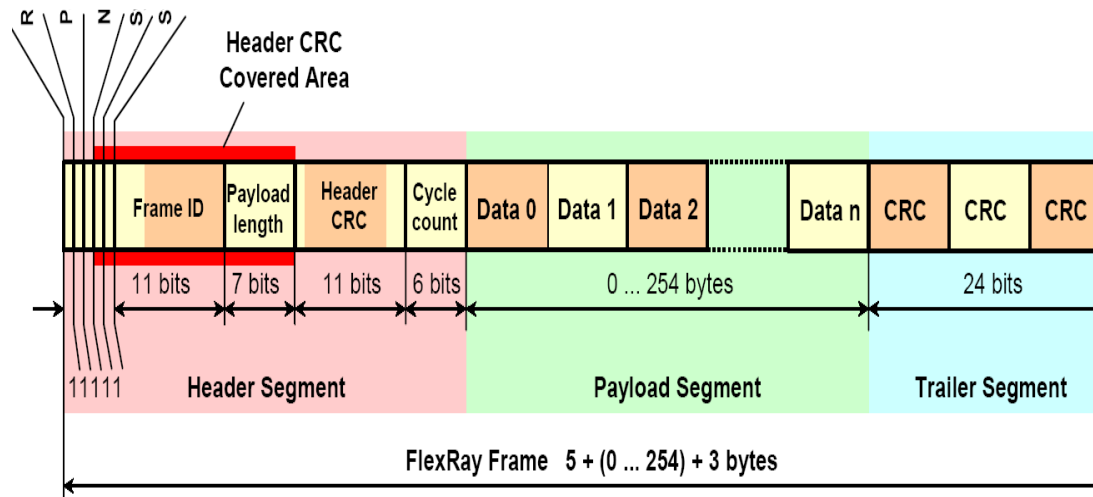
- Bit encoding interacts with CRCs
  - A one- or two-bit error can cascade into multiple bits as seen by the CRC
    - For example, bit stuffing errors can cascade to multi-bit errors
    - For example 8b10b encoding can cascade to multi-bit errors
  - Sometimes bit encoding can help (e.g., Manchester RZ encoding) by making it likely corruption will violate bit encoding rules
- Watch out for errors in intermediate stages
  - A study of Ethernet packets found errors happened in routers!
  - 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.

# CAN vs. FlexRay Length Field Corruptions

- CAN does not protect length field
  - Corrupted length field will point to wrong location for CRC!
  - **One bit error** in length field circumvents HD=6 CRC



- FlexRay solves this with a header CRC to protect Length



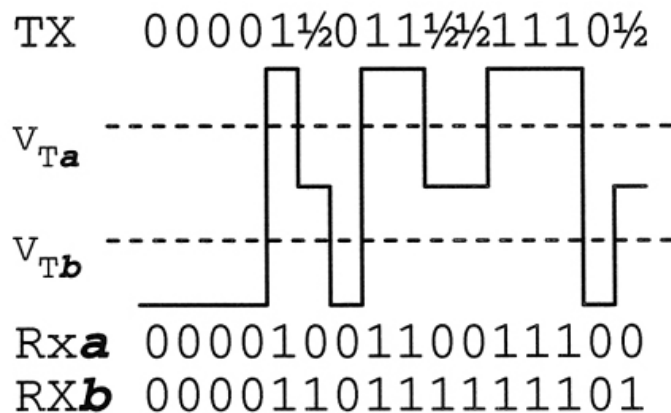
Source: FlexRay Standard, 2004

Figure 4-1: FlexRay frame format.



# Byzantine CRC

- Byzantine failures for CRCs and Checksums



Example Schrodinger's CRC caused by non-saturated voltage values on a data bus. Two receivers (*a* and *b*) can see the same message as having two different values, and each view having a valid CRC

- Paulitsch, Morris, Hall, Driscoll, Koopman & Latronico, "Coverage and Use of Cyclic Redundancy Codes in Ultra-Dependable Systems," DSN05, June 2005.
- Memory errors may be complex and value-dependent
  - A cosmic ray strike may take out multiple bits in a pattern

# Composite Checksum/CRC Schemes

- Idea: use a second error code to enhance error detection
  - Rail systems add a 32-bit “safety CRC”
  - Checksum + CRC can be a win ([Tran 1999] on CAN)
  - ATN-32 is a checksum used in context of network packet CRC
- Youssef et al. have a multi-CRC aviation proposal
  - Combines ideas such as “OK to miss an error if infrequent”
  - Uses composite CRCs based on factorization
  - Evaluated with random experiments
- Issue to consider:
  - What HD do you really get with a composite scheme?
    - E.g., which error patterns slip past both CRCs?
  - Are diverse checksum+CRC approaches better than dual CRC approaches?

# Review

- Introduction
  - Motivation – why isn't this a solved problem?
  - Parity computations as an example
  - Error code construction and evaluation (without scary math)
  - Example using parity codes
- Checksums
  - What's a checksum?
  - Commonly used checksums and their performance
- Cyclic Redundancy Codes (CRCs)
  - What's a CRC?
  - Commonly used CRC approaches and their performance
- Don't blindly trust what you hear on this topic
  - A good CRC is almost always **much** better than a good Checksum
  - Many published (and popular) approaches are suboptimal or just plain wrong
  - There are some topics to be careful of because we don't know the answers
- Q&A

# Investigators

- Philip Koopman                      Koopman@cmu.edu
  - Assoc. Prof of ECE, Carnegie Mellon University (PA)
  - Embedded systems research, emphasizing dependability & safety
  - Industry experience with transportation applications
- Kevin Driscoll                      Kevin.Driscoll@honeywell.com
  - Engineering Fellow, Honeywell Laboratories (MN)
  - Ultra-dependable systems research & security
  - Extensive data communications experience for aviation
- Brendan Hall                      Brendan.Hall@honeywell.com
  - Engineering Fellow, Honeywell Laboratories (MN)
  - Fault tolerant system architectures & development process
  - Extensive experience with aviation computing systems