Software and Digital Systems Program - Data Integrity Techniques

25 September 2014

Prof. Philip Koopman
Carnegie Mellon University
koopman@cmu.edu

Co-PIs: Kevin Driscoll, Brendan Hall
Honeywell Laboratories

Funded by FAA Contract DTFACT-11-C-00005

This work was supported by the Federal Aviation Administration, Aircraft Certification Service, and Assistant Administration for NextGen, William J. Hughes Technical Center, Aviation Research Division, Atlantic City International Airport, New Jersey 08405.
The findings and conclusions in this presentation are those of the author(s) and do not necessarily represent the views of the funding agency. This presentation does not constitute FAA policy.

Electrical & Computer ENGINEERING

Investigators

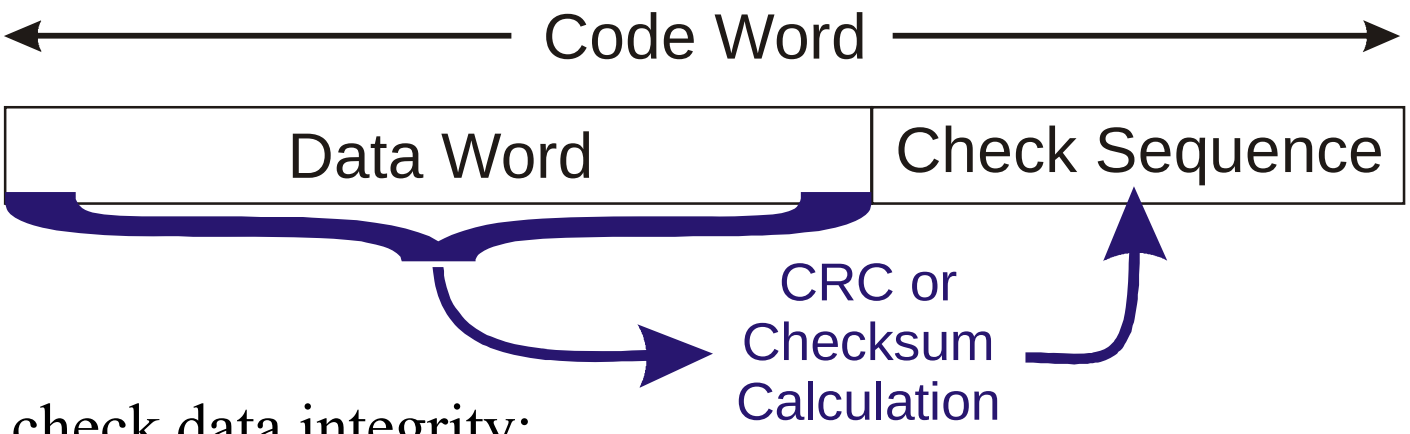
- **Philip Koopman** – CRC/Checksum Evaluation
  - Professor at Carnegie Mellon University (PA)
  - Embedded systems research, emphasizing dependability & safety
  - Industry experience with transportation applications
- **Kevin Driscoll** – Industry Practices & Criticality Mapping
  - Engineering Fellow, Honeywell Laboratories (MN)
  - Ultra-dependable systems research & security
  - Extensive data communications experience for aviation
- **Brendan Hall** – Industry Practices & Criticality Mapping
  - Engineering Fellow, Honeywell Laboratories (MN)
  - Fault tolerant system architectures & development process
  - Extensive experience with aviation computing systems

Agenda

- **Introduction**
  - Motivation – most folk wisdom about CRCs & Checksums is incorrect
  - Overview of "error detection concepts for poets"
- **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
- **Selecting a Good CRC**
  - Use data, not folklore
- **Mapping to functional criticality levels** (summary)
  - Hamming Distance matters the most
- **System level effects & cross-layer interactions**
  - CAN/ARINC-825 as an object lesson
- **Q&A**

Checksums and CRCs Protect Data Integrity

- Compute check sequence when data is transmitted or stored
  - **Dataword**: the data you want to protect (can be any size from bits to GBytes)
  - **Check Sequence**: the result of the CRC or checksum calculation
    - Can be thought of a pseudo-random number based on dataword value
  - **Codeword** = Dataword 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 still **might be data corruption** (undetected errors)…
      But if there is, you didn't detect it in this case.

Potential CRC/Checksum Usage Scenarios

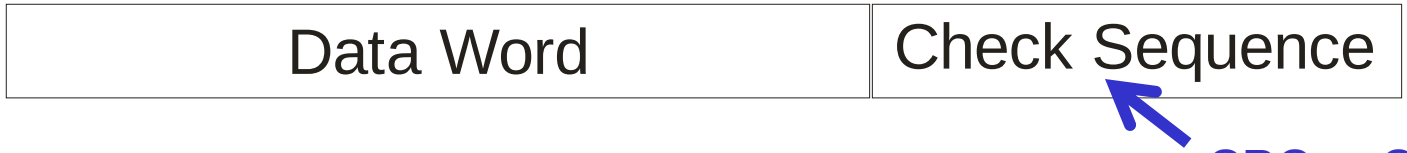
- Network packet integrity check
- Image integrity check for software update
- Boot-up integrity check of program image
  - e.g., flash memory data integrity check
- FPGA configuration integrity check
- Configuration integrity check
  - e.g., operating parameters in EEPROM
- RAM value integrity check
- A key consideration is residual undetected error rate
  - Not all checksums/CRCs are created equal

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!** (unless you've kept up with recent research)
  - There wasn't enough compute power to find optimal CRCs until recently… so early results are often *not* very good (and some are awful)
  - There is a lot of incorrect writing on this topic … that at best incorrectly 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)

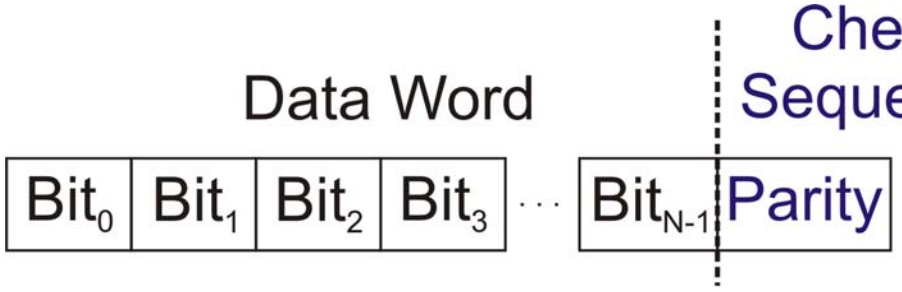
- The general idea of an error code is to mix all the bits in the data word to produce a condensed version (the check sequence)
  - Ideally, every bit in the data word affects many check sequence bits
  - Ideally, bit errors in the code word have high probability of being detected
  - Ideally, more probable 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



$$Parity = Bit_0 \oplus Bit_1 \oplus Bit_2 \oplus Bit_3 \oplus \ldots \oplus Bit_{N-1}$$

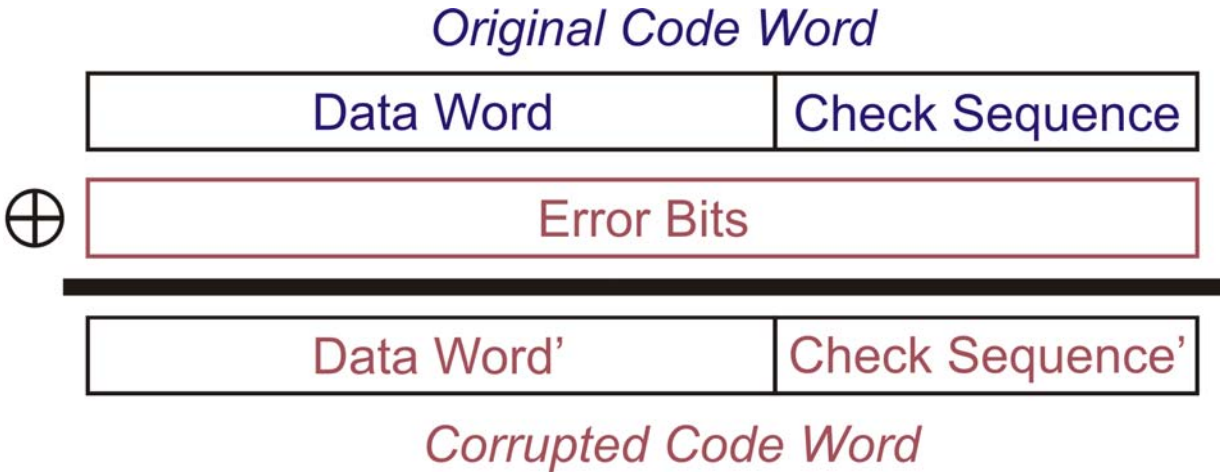
Note: ⊕ is eXclusive OR (XOR)
Parity = 0 for even number of "1" bits
Parity = 1 for odd number of "1" bits

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

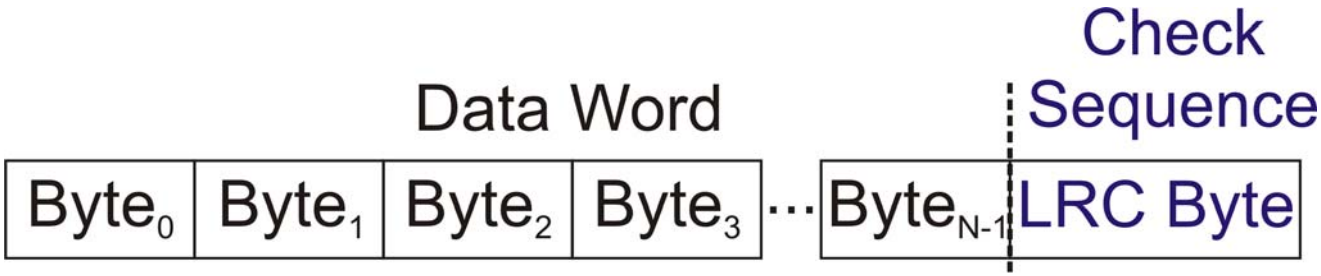
- Data corruption is "bit flips" ("bisymmetric inversions")
  - Each bit has some probability of being inverted
  - **"Weight"** of error word is number of bits flipped (number of "1" bits in error)



- 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:

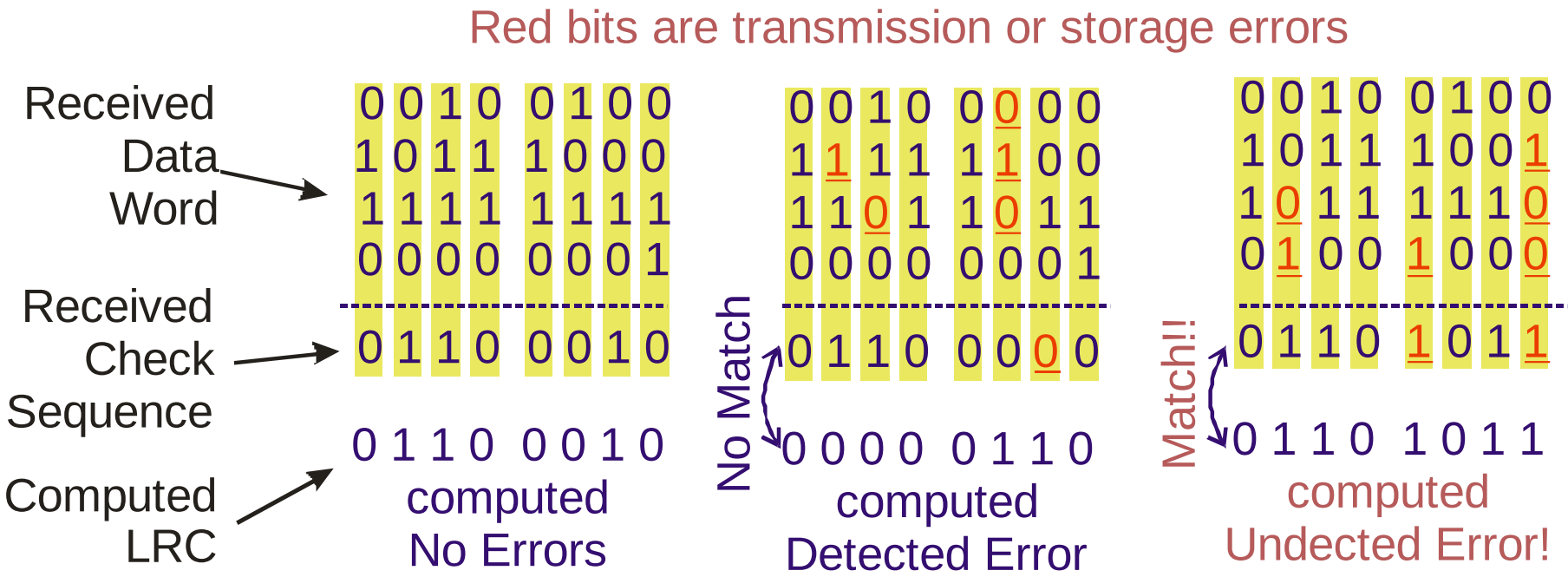
$$
\begin{array}{rcccccccc}
 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
\oplus & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
\oplus & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\oplus & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
\hline
 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0
\end{array}
$$

Result is parity of each vertical bit slice

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

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!



- 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

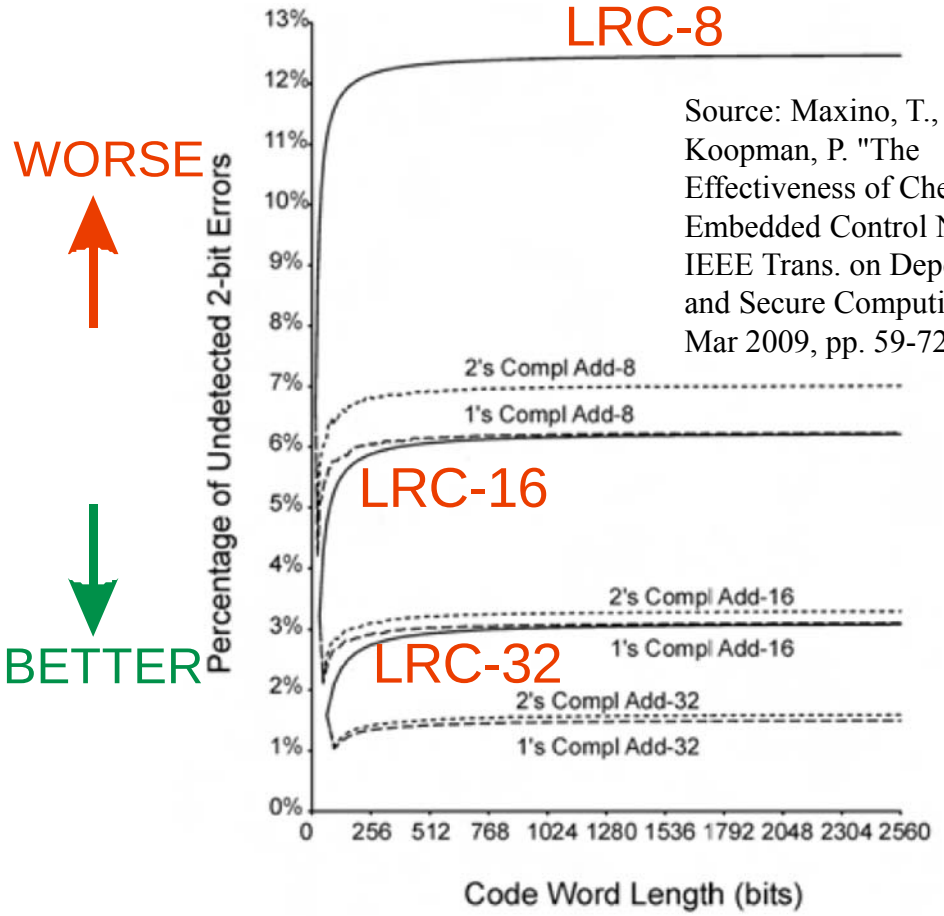
Error Code Effectiveness Measures

- 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.

CHECKSUM OPERATION & ERROR DETECTION PERFORMANCE

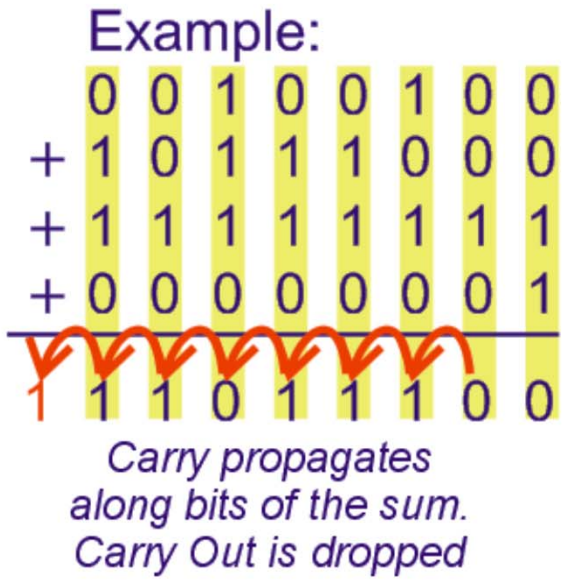
Checksums

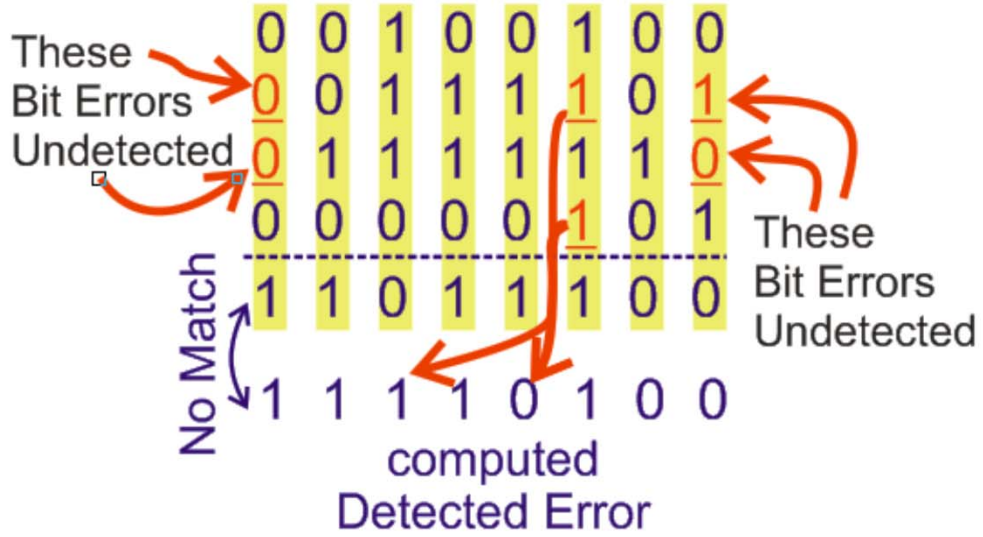
- A checksum "adds" together "chunks" of data
  - The "add" operation may not be normal integer addition
  - The chunk size is typically 8, 16, or 32 bits

- We'll discuss:
  - Integer addition "checksum"
  - One's complement "checksum"
  - Fletcher Checksum
  - Adler Checksum
  - ATN Checksum (AN/466)

Integer Addition Checksum

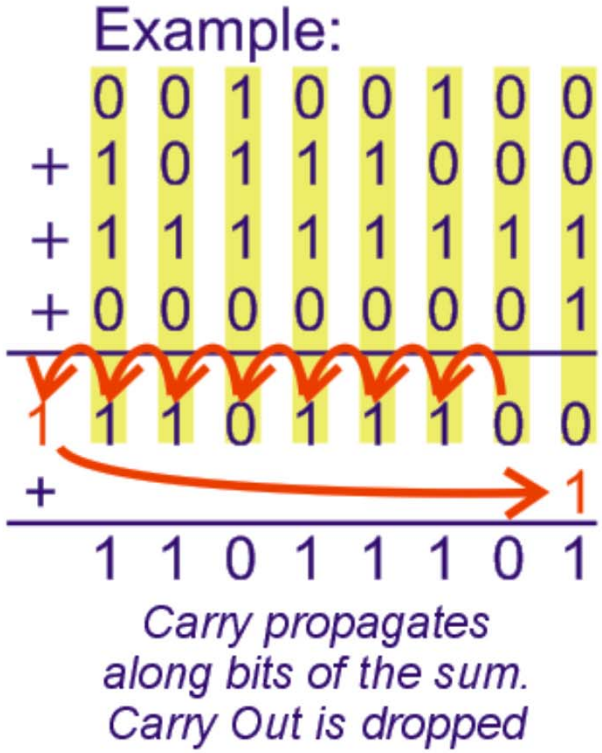
- Same as LRC, except use integer "+" instead of XOR
  - The carries from addition promote bit mixing between adjacent columns
    - Can detect errors that make two bits go 0 ➔ 1 or 1 ➔ 0 (except top-most bits)
    - Cannot detect compensating errors (one bit goes 0 ➔1 and another 1 ➔ 0)
  - Carry out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

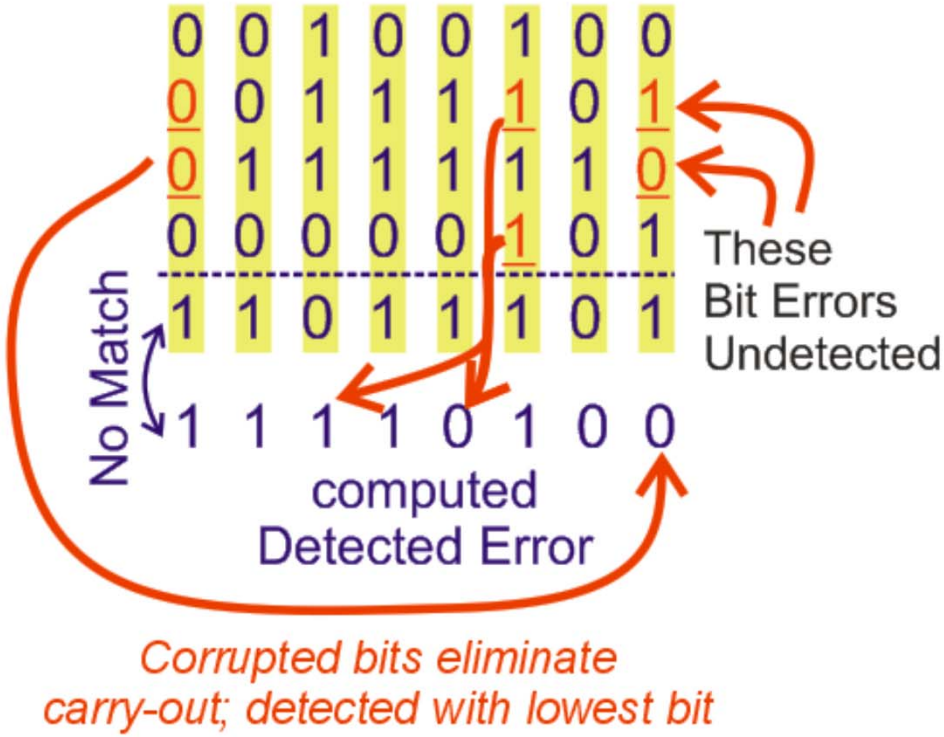




One's Complement Addition Checksum

- 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

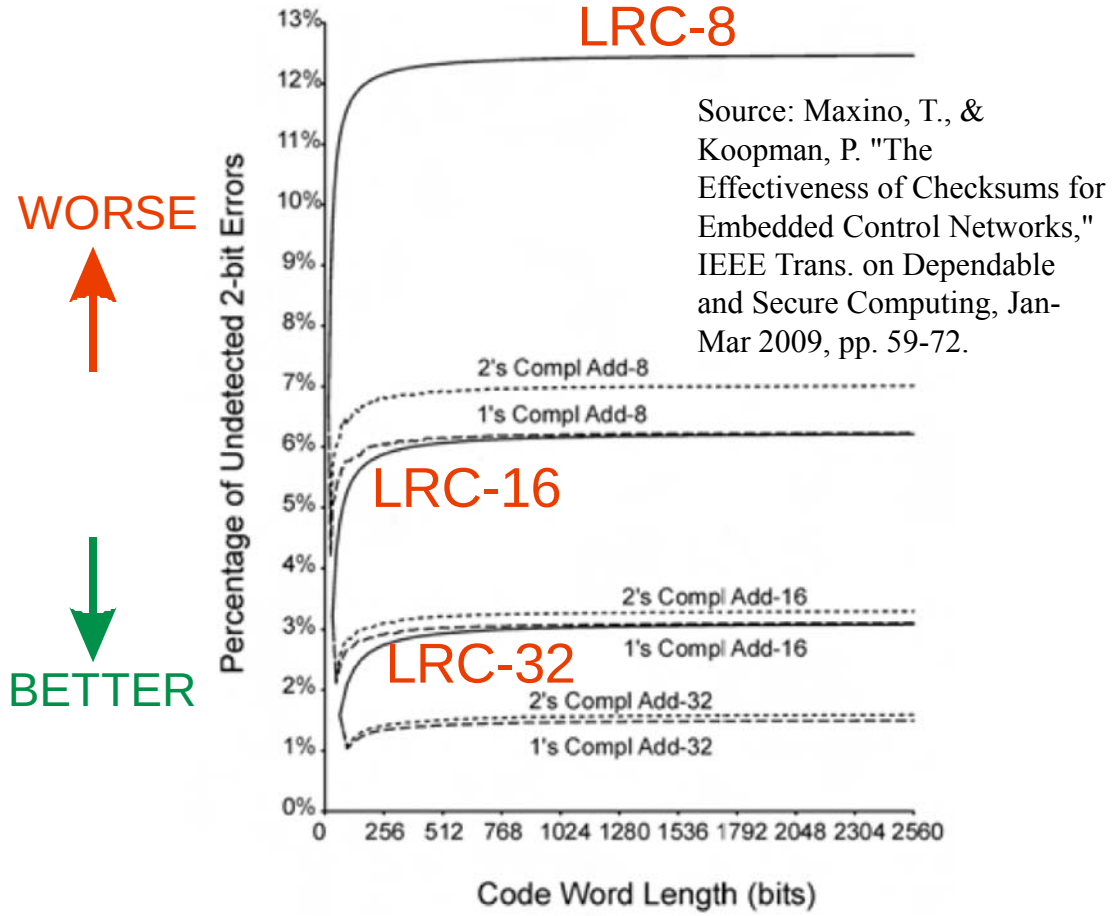




Checksums Are Better Than LRCs

- 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



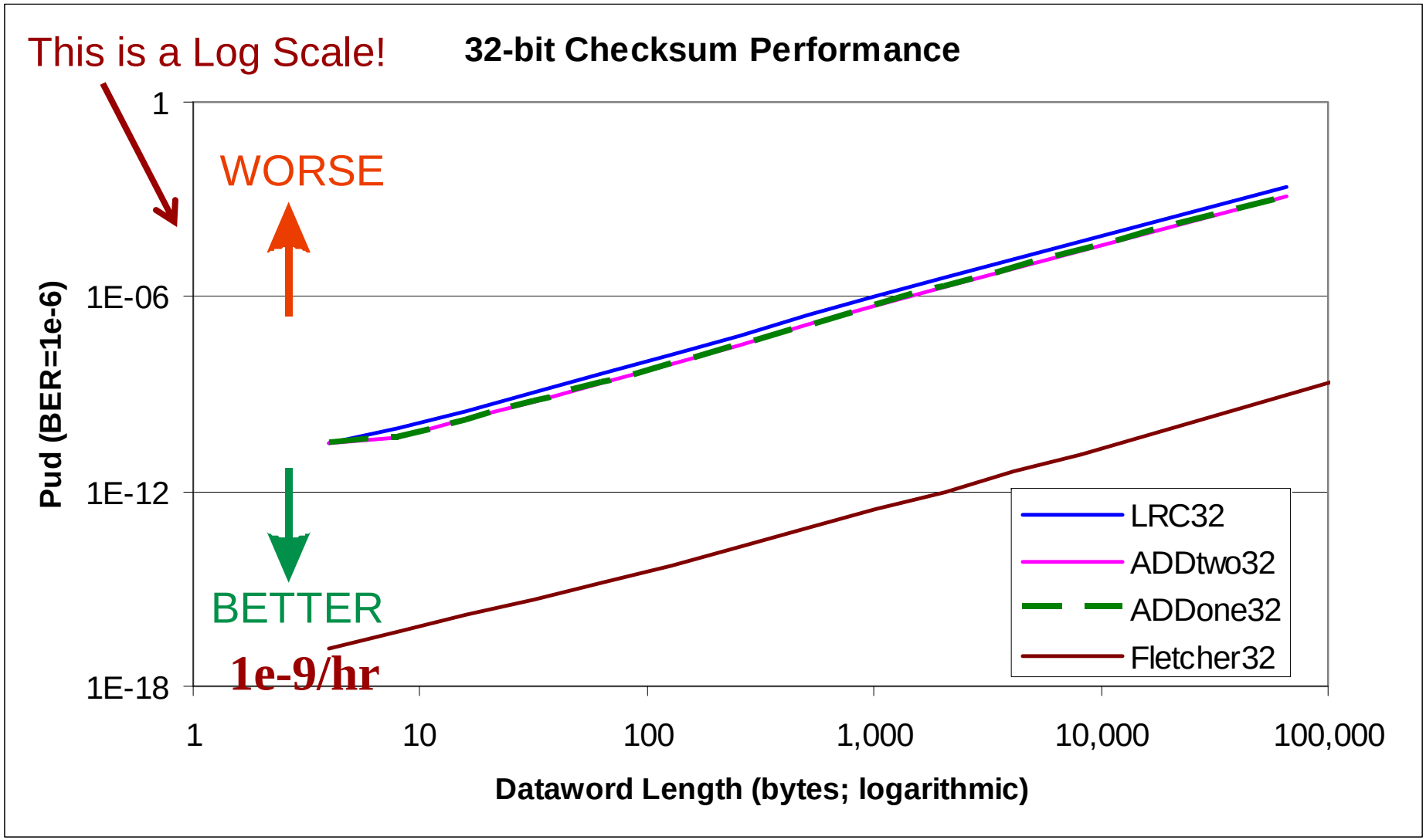
Source: Maxino, T., & Koopman, P. "The Effectiveness of Checksums for Embedded Control Networks," IEEE Trans. on Dependable and Secure Computing, Jan-Mar 2009, pp. 59-72.

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.

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: **$A = A + Byte_i$; B = B + A;**
    - One's complement addition!
  - **Result is A concatenated with B** (16-bit result)
- Significant improvement comes from the running sum B
  - $B = Byte_{N-1} + 2*Byte_{N-2} + 3*Byte_{N-3} + \ldots$
  - Makes checksum order-dependent (switched byte order detected)
  - **Gives HD=3** (all 2-bit errors) *until the B value rolls over*
    - For example, $256*Byte_{N-256}$ does not affect B

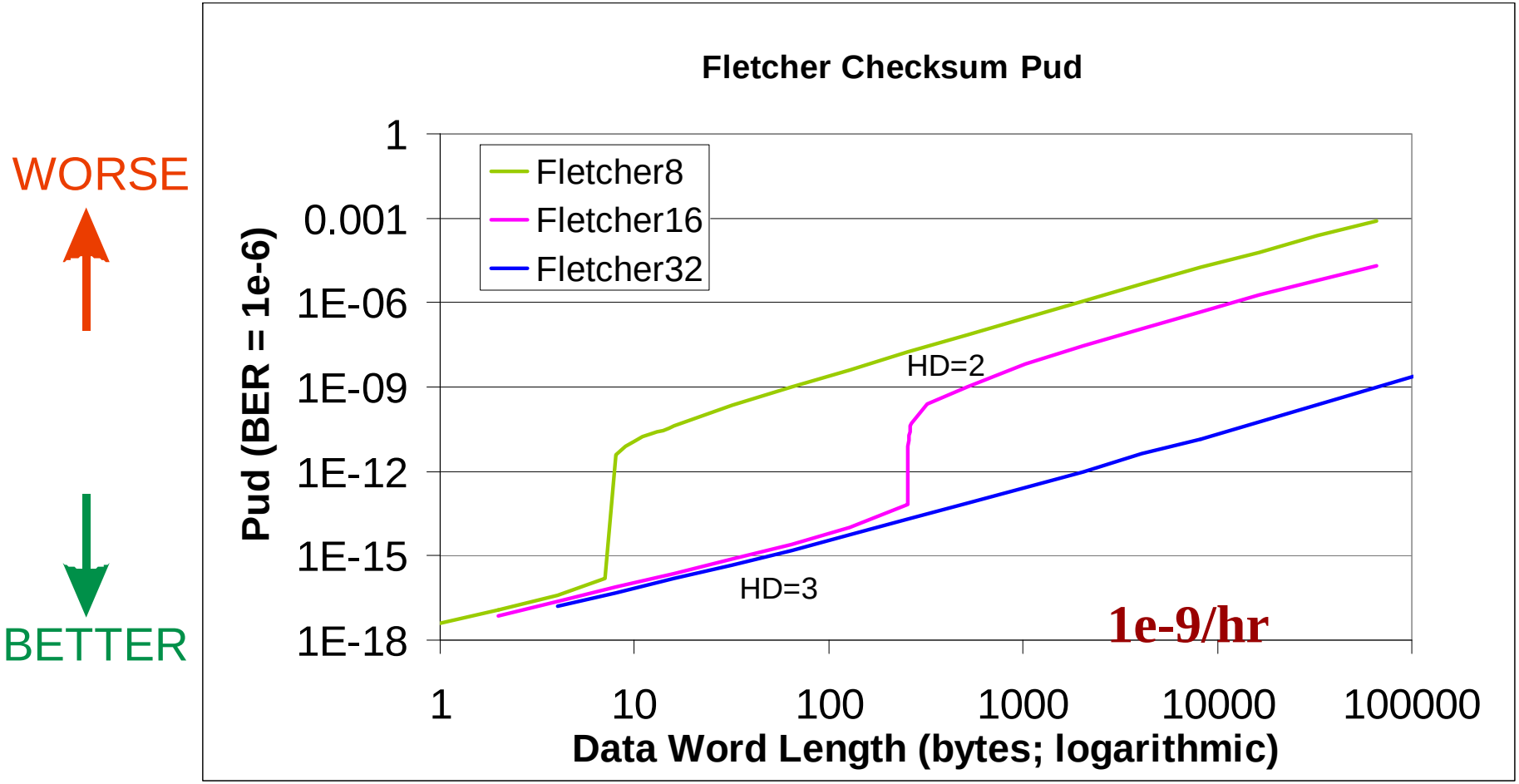
Fletcher Is A Better Checksum



Note: Pud is per message; perhaps up to 10^9 msgs/hr

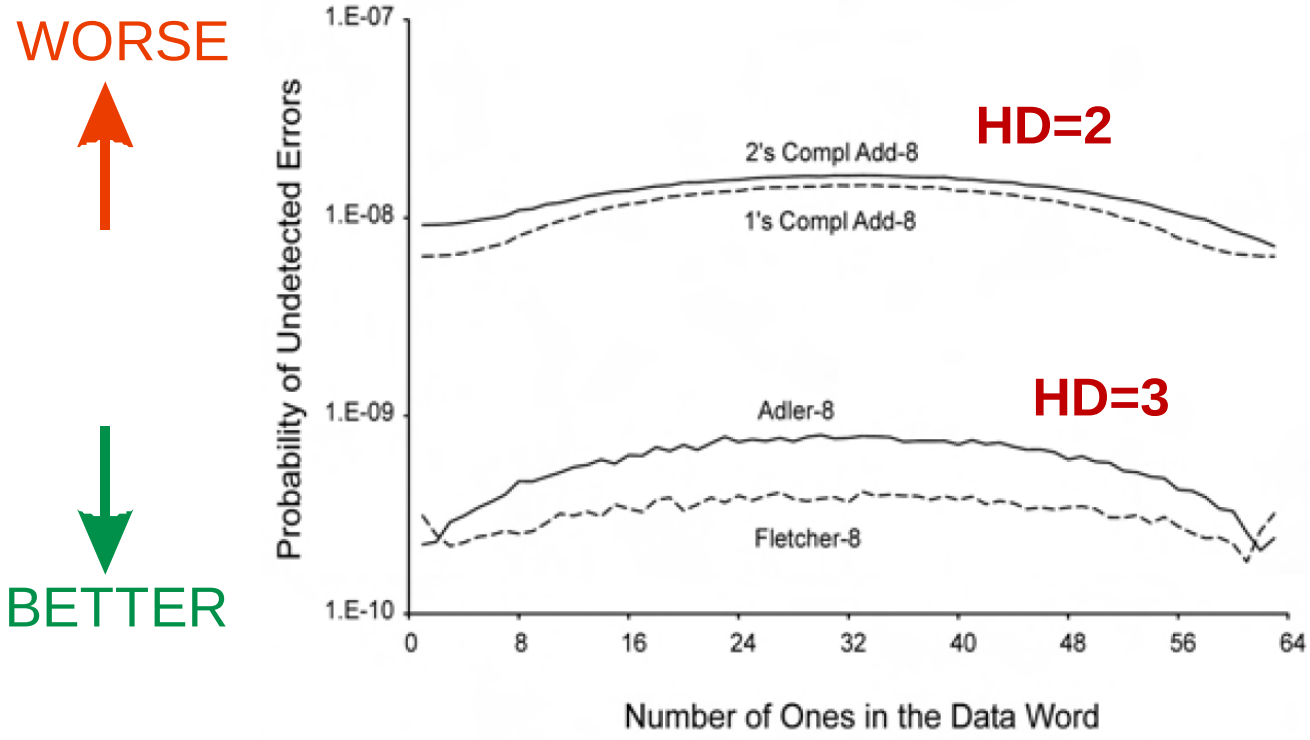
Fletcher Achieves HD=2 or HD=3

- HD Break point depends upon size
  - E.g., Fletcher 8 is two 4-bit running sums



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

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)

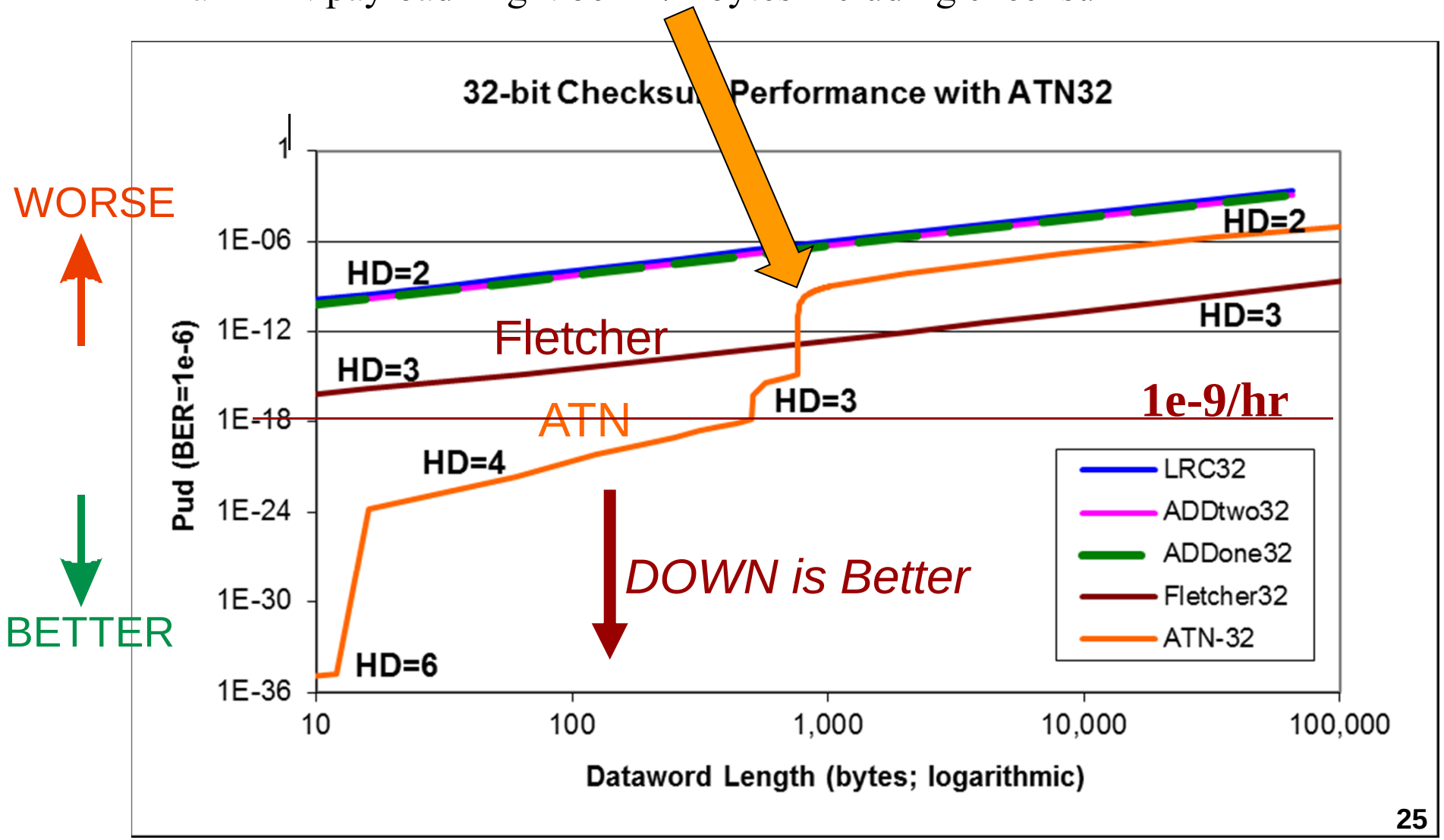
- In practice, it is *not worth it*
  - For most sizes and data lengths Adler is worse than Fletcher
  - In the best case it is only very slightly better
    - But computation is more expensive because of the modular sum

ATN-32 Checksum

- ATN-32 is a modified Fletcher Checksum
  - [AN/466] "Manual on Detailed Technical Specifications for the Aeronautical Telecommunication Network (ATN) using ISO/OSI Standards and Protocols, Part 1 – Air-Ground Applications." Doc 9880, International Civil Aviation Organization, Montreal, First Edition, 2010.
- Uses four running 8-bit sums:
  - **SUMA = SUMA $+_{one}$ next data chunk**
  - **SUMB = SUMB $+_{one}$ SUMA**
  - **SUMC = SUMC $+_{one}$ SUMB**
  - **SUMD = SUMD $+_{one}$ SUMC**
  - Final bytes are a defined mixture of above four sums
- Performance is better than Fletcher32 for short lengths
  - Appears to be optimized for short messages
  - But significantly worse than Fletcher32 for long lengths!
    - (See next slide)

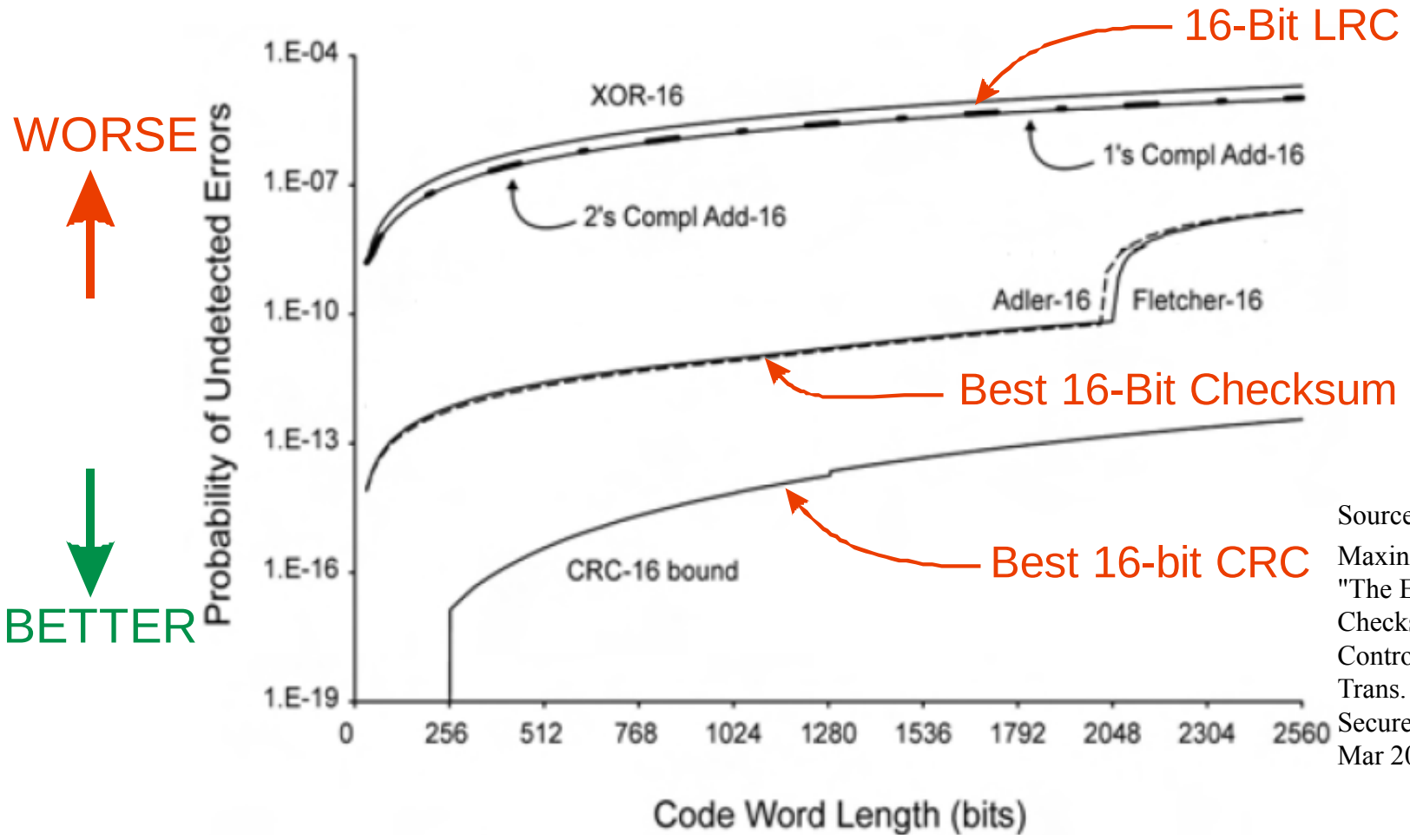
ATN-32 Is Only Good At Short Lengths

- HD=3 up to 504 bytes; HD=2 at and above 760 bytes
- Max ATN payload might be 1471 bytes including checksum



Can We Do Even Better? YES!

- There are attempts at better checksums
- Can often get HD=6 (detect all 1, 2, 3, 4, 5-bit errors) with a CRC



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.

For this graph, assume Bit Error Rate (BER) = 10^{-5} flip probability per bit

CRC OPERATION & ERROR DETECTION PERFORMANCE

Cyclic Redundancy Codes (CRCs)

- **CRCs Use Division Instead Of Addition**
- Intuitive description of why they are better than checksums:
  - 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 mod 251 = 166 ← 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 mod 100 = 91 ← 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

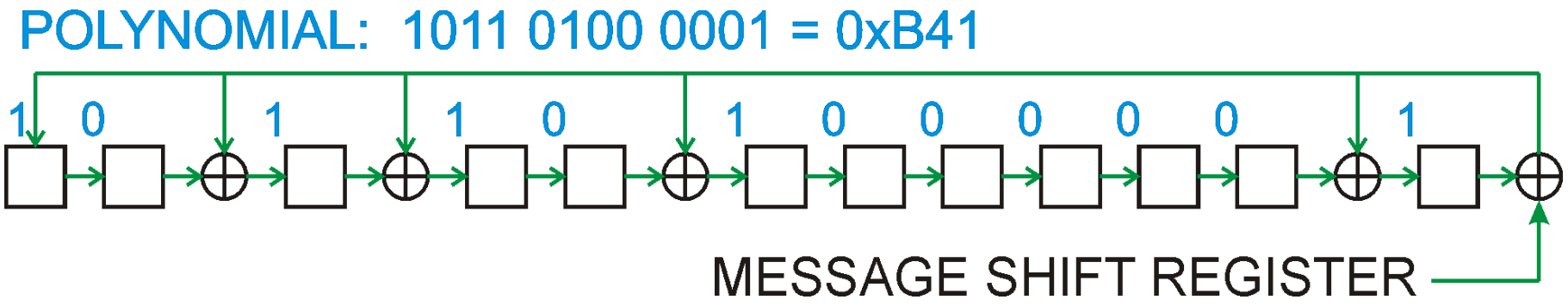
- Use polynomial division over Galois Field(2) (remember that from high school?) (this is a mathematician thing)
  - At a hand-waving level this is division using Boolean Algebra
    - "Add" and "Subtract" used by division algorithm both use XOR

```
11010011101100 000 <--- Data Word left shifted by 3 bits
1011               <--- 4-bit divisor is 1011  x³ + 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
-----------------   Remainder is the Check Sequence
00000000000000 100 <--- Remainder (3 bits)
```

[Wikipedia]

CRC Operation – Hardware View

- Cyclic Redundancy Code operation
  - Computes a (non-secure) message digest using shift and XOR
  - This is a hardware implementation of polynomial division
    - (Same math & mechanism as a Linear Feedback Shift Register pseudo-random number generator)



$0xB41 = x^{12}+x^{10}+x^9+x^7+x+1$ (the "+1" is implicit in the hex value)

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

(presentation uses implicit +1 notation for hex values)

- The "magic" is in picking a good polynomial
  - OK, great, so which polynomial do I pick?

Isn't Optimality A Long-Solved Problem? (No)

- Compute power wasn't available to evaluate error code performance when they were invented
  - So originators relied on mathematical analysis…
    … which gives bounds and heuristics, but not exact answers
  - Exact enumeration of error detection performance is recent
- Literature is poorly accessible to computer engineers
  - Much of it is written in dense math that is hard to apply
    - Many papers concentrate on interesting math rather than applicable results
    - Sources that are readily accessible may have incorrect folklore
  - Practitioners mostly use what someone else previously used
    - Assume it must be good (often this is incorrect)
    - Unaware of limitations or mistakes in previous work
  - Practical application knowledge poorly documented
    - There are a handful of protocol experts who know this stuff, mostly originating in the European rail domain

A Flawed, But Typical, 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, 2nd ed.*, 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 Flawed, But Typical, 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, 2nd ed.*, 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 Flawed, But Typical, 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, 2nd ed*, 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.

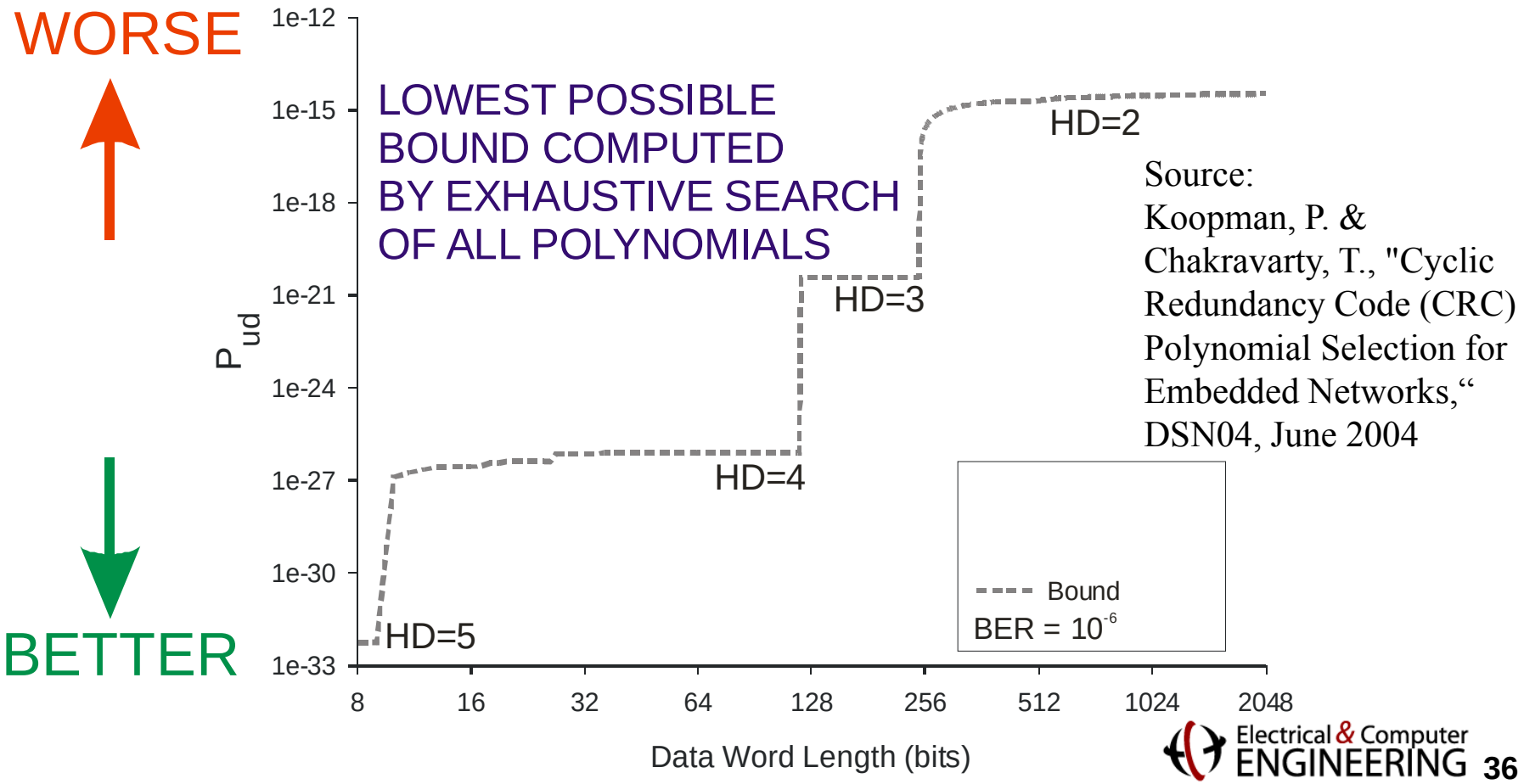
» *Numerical Recipes in C, 2nd ed.*, Press et al.

- 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 get good results by picking at random!**

NOTE: 3rd Edition fixed problems *based on our feedback*

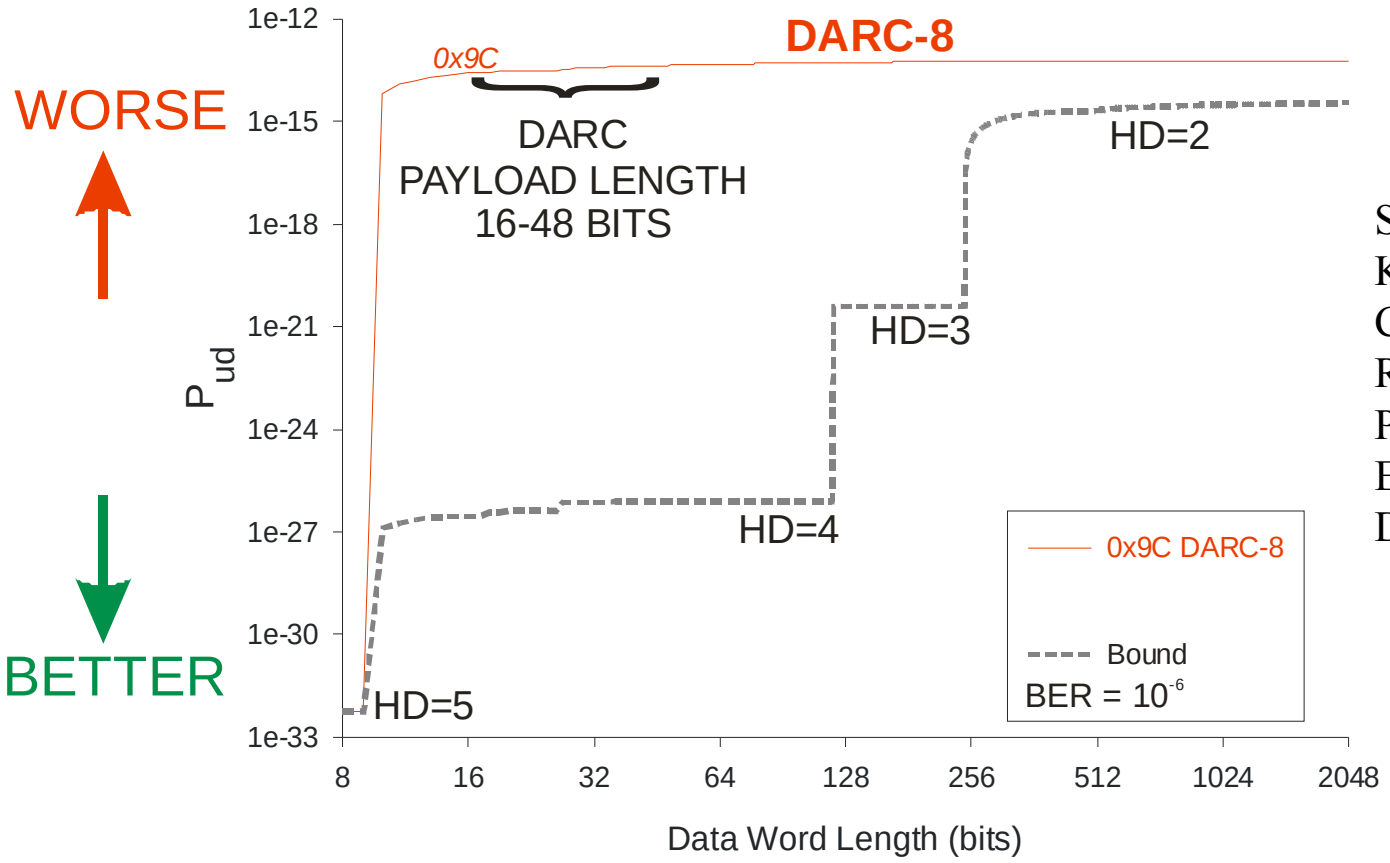
Example – 8 Bit Polynomial Choices

- 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?

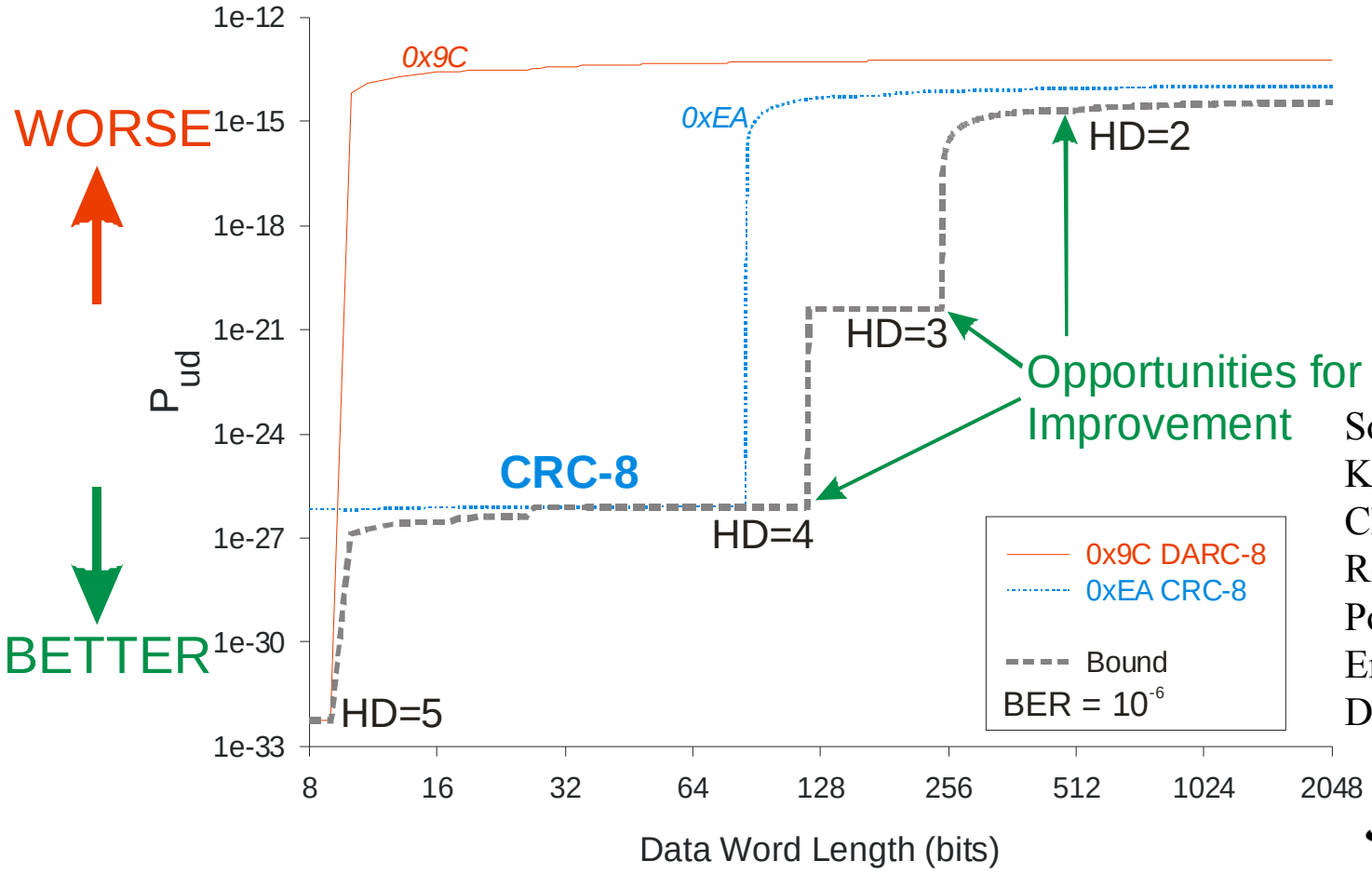
- DARC (Data Radio Channel), ETSI, October 2002
  - DARC-8 polynomial is optimal for 8-bit payloads
  - BUT, DARC uses 16-48 bit payloads, and misses some 2-bit errors
  - Could have detected all 2-bit and 3-bit errors with same size CRC!



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

CRC-8 Is Better

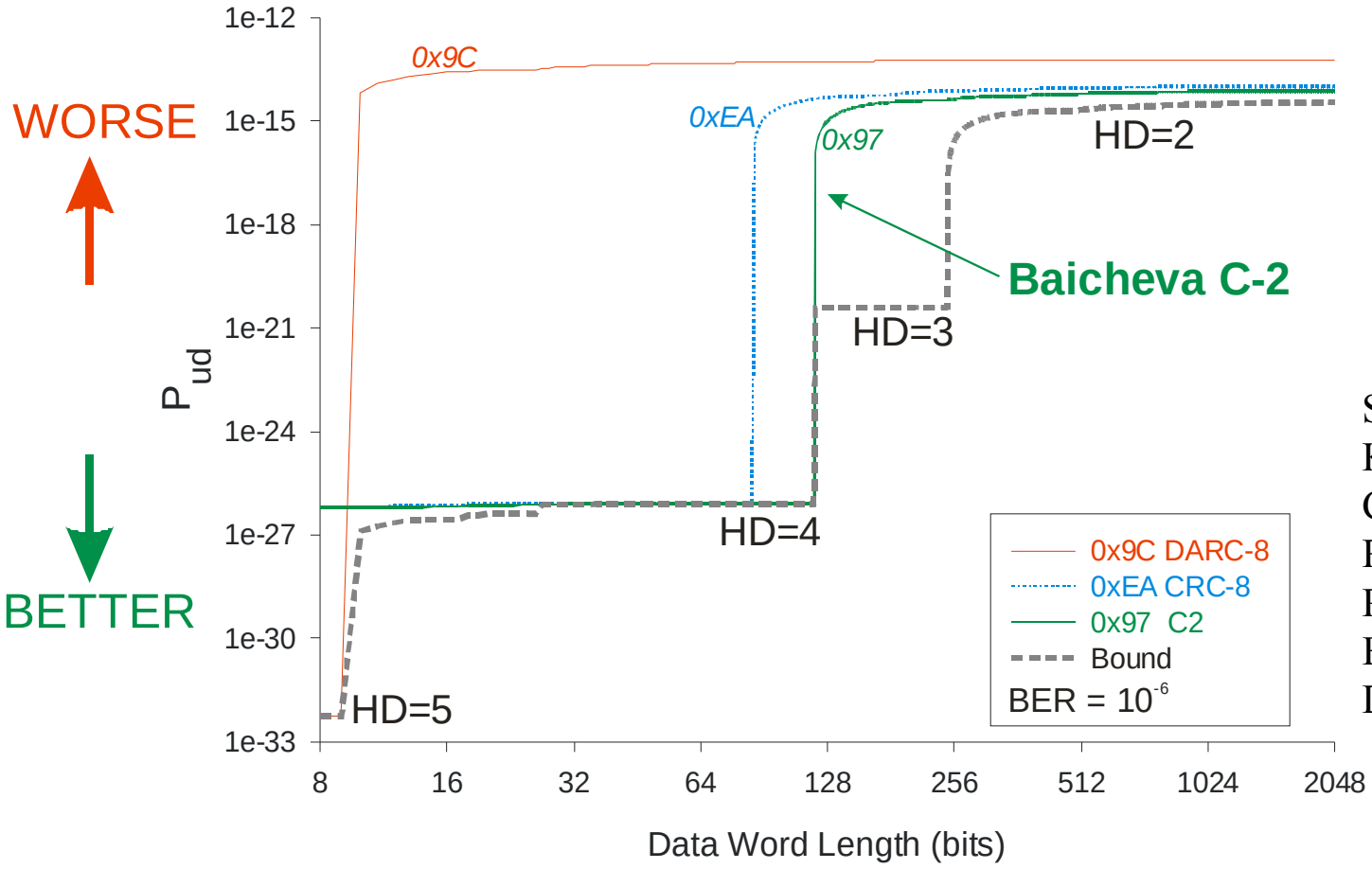
- CRC-8 (0xEA) is in very common use
  - Good for messages up to size 85
  - But, room for improvement at longer lengths. Can we do better?



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

Baicheva's Polynomial C2 Is Even Better

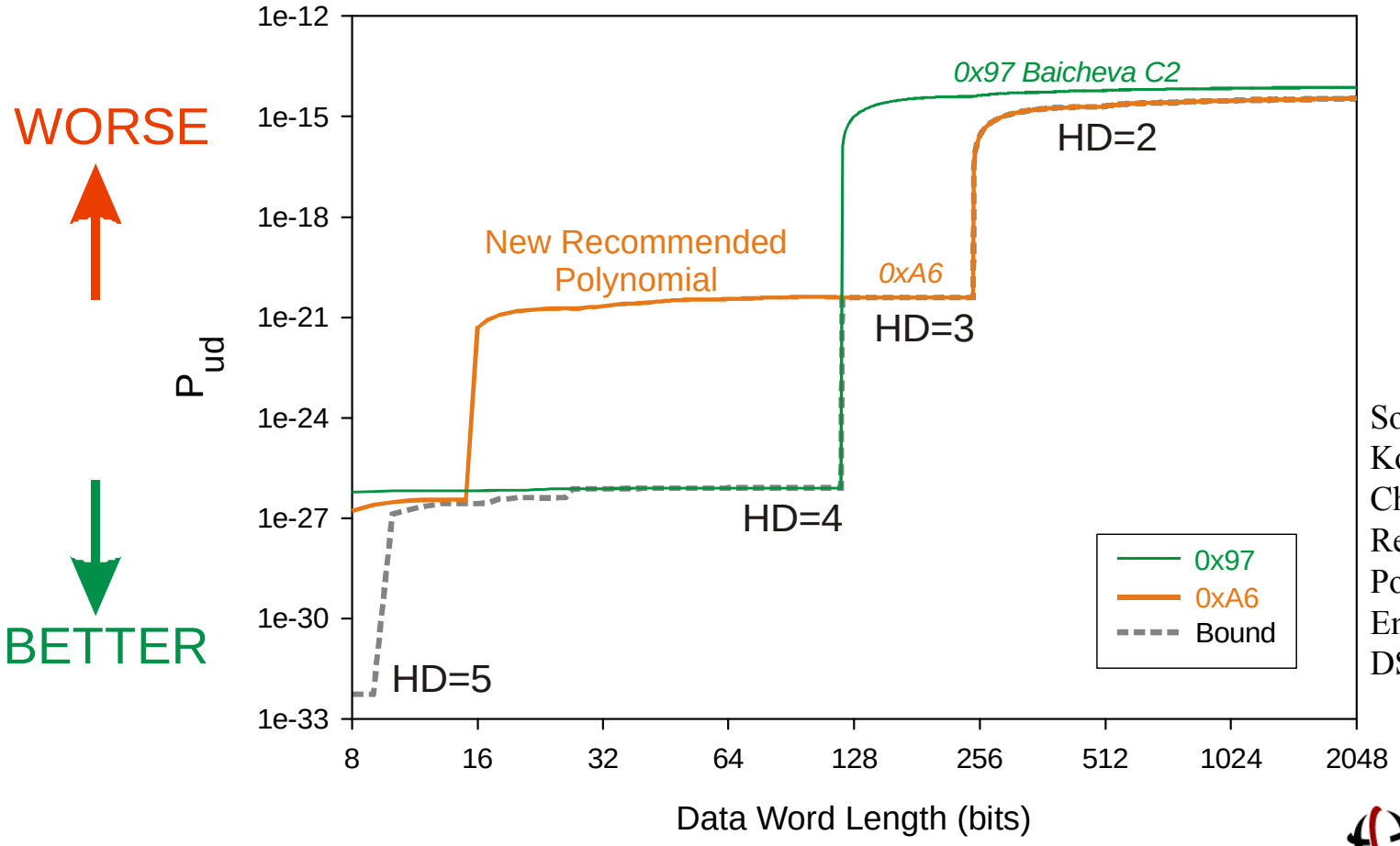
- [Baicheva98] proposed polynomial C2, 0x97
  - Recommended as good polynomial to length 119
  - Dominates 0xEA (better P_{ud} at every length)



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?

- Use one of:
  - OxE7 (for good HD=4 performance)
  - 0xA6 (for good HD=4 performance and better small size performance – shown below)



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

SELECTING A GOOD CRC

Optimal Polynomials For Small CRCs

- Updated Results as of August 2014:

Max length at HD / Polynomial	CRC Size (bits)													
	3	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
HD=2	**0x5**	**0x9**	**0x12**	**0x33**	**0x65**	**0xe7**	**0x119**	**0x327**	**0x5db**	**0x987**	**0x1abf**	**0x27cf**	**0x4f23**	**0x8d95**
HD=3	4 **0x5**	11 **0x9**	26 **0x12**	57 **0x33**	120 **0x65**	247 **0xe7**	502 **0x119**	1013 **0x327**	2036 **0x5db**	4083 **0x987**	8178 **0x1abf**	16369 **0x27cf**	32752 **0x4f23**	65519 **0x8d95**
HD=4			10 **0x15**	25 **0x23**	56 **0x5b**	119 **0x83**	246 **0x17d**	501 **0x247**	1012 **0x583**	2035 **0x8f3**	4082 **0x12e6**	8177 **0x2322**	16368 **0x4306**	32751 **0xd175**
HD=5					4 **0x72**	9 **0xeb**	13 **0x185**	21 **0x2b9**	26 **0x5d7**	53 **0xbae**	52 **0x1e97**	113 **0x212d**	136 **0x6a8d**	241 **0xac9a**
HD=6						4 **0x9b**	8 **0x13c**	12 **0x28e**	22 **0x532**	27 **0xb41**	52 **0x1e97**	57 **0x372b**	114 **0x573a**	135 **0x9eb2**
HD=7								5 **0x29b**	12 **0x571**	11 **0xa4f**	12 **0x12a5**	13 **0x28a9**	16 **0x5bd5**	19 **0x968b**
HD=8									4 **0x4f5**	11 **0xa4f**	11 **0x10b7**	11 **0x2371**	12 **0x630b**	15 **0x8fdb**

http://users.ece.cmu.edu/~koopman/crc/index.html

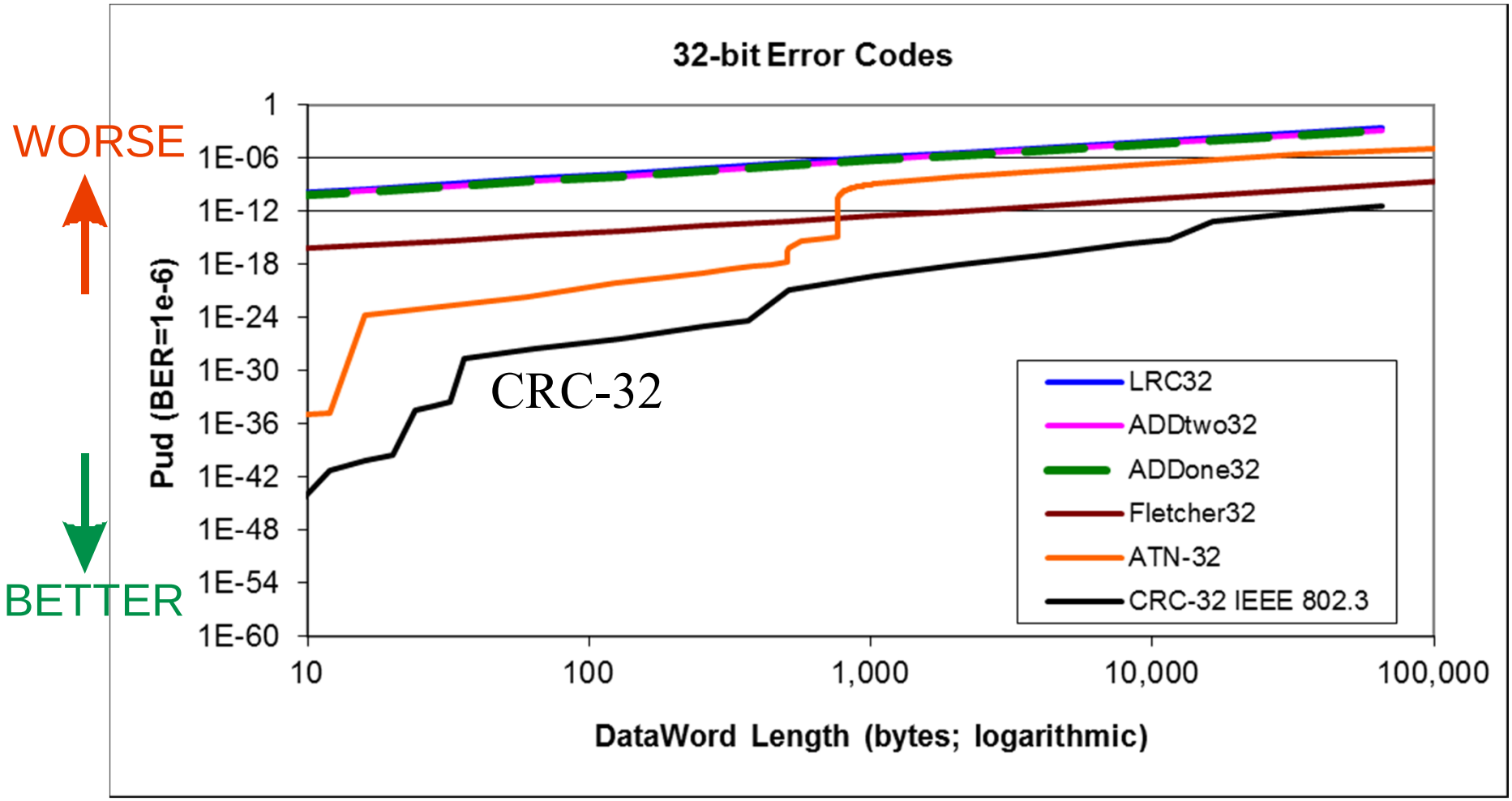
More On Picking A Good CRC

- 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

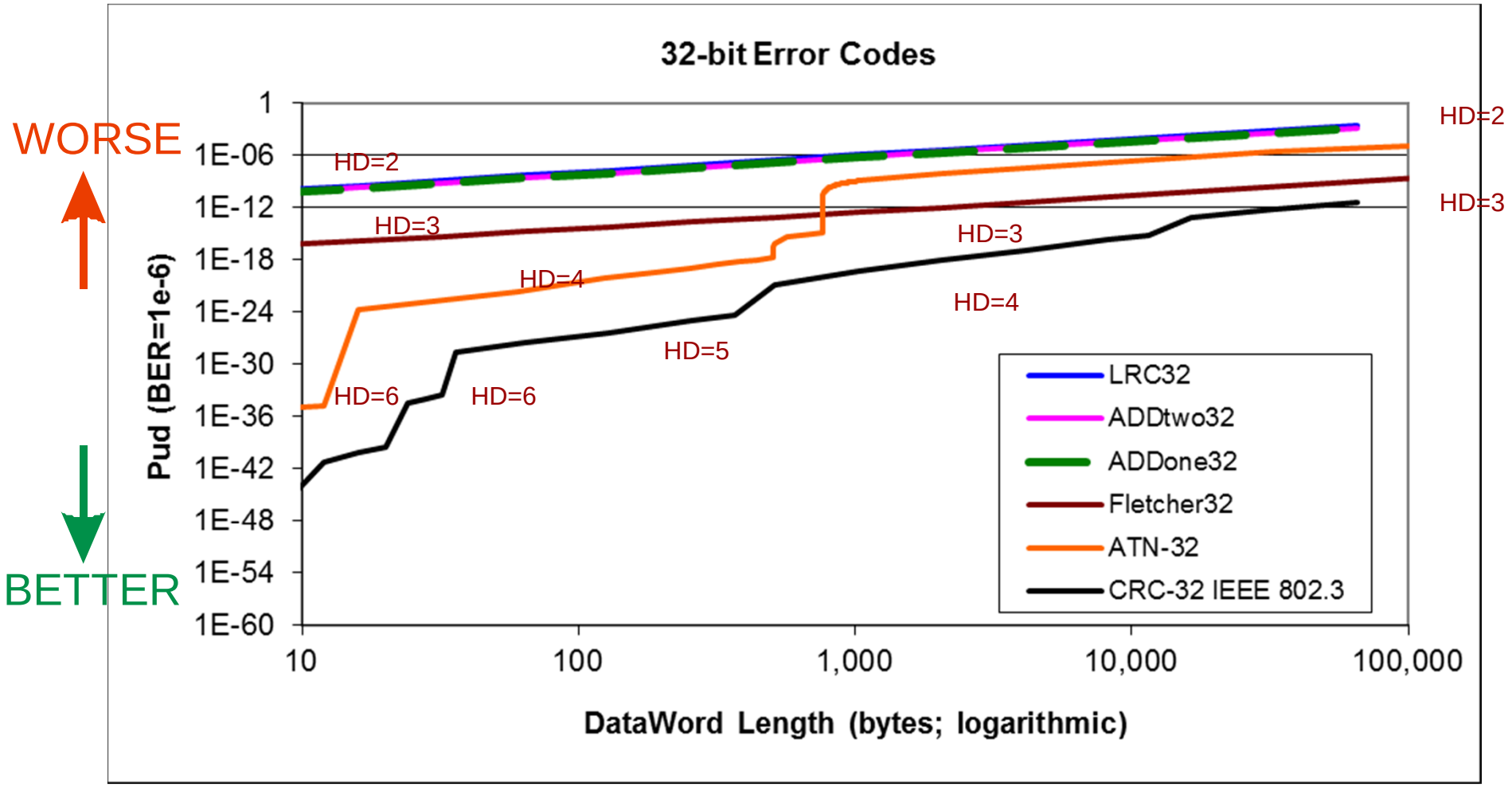
- Good values also known for 24-bit and 32-bit polynomials
  - IEEE 802.3 standard gives HD=6 up to 268-bit data words
  - But 0xBA0DC66B gives HD=6 up to 16,360-bit data words
    - Koopman, P., "32-bit cyclic redundancy codes for Internet applications," International Conference on Dependable Systems 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

CRC-32 Beats All Checksums



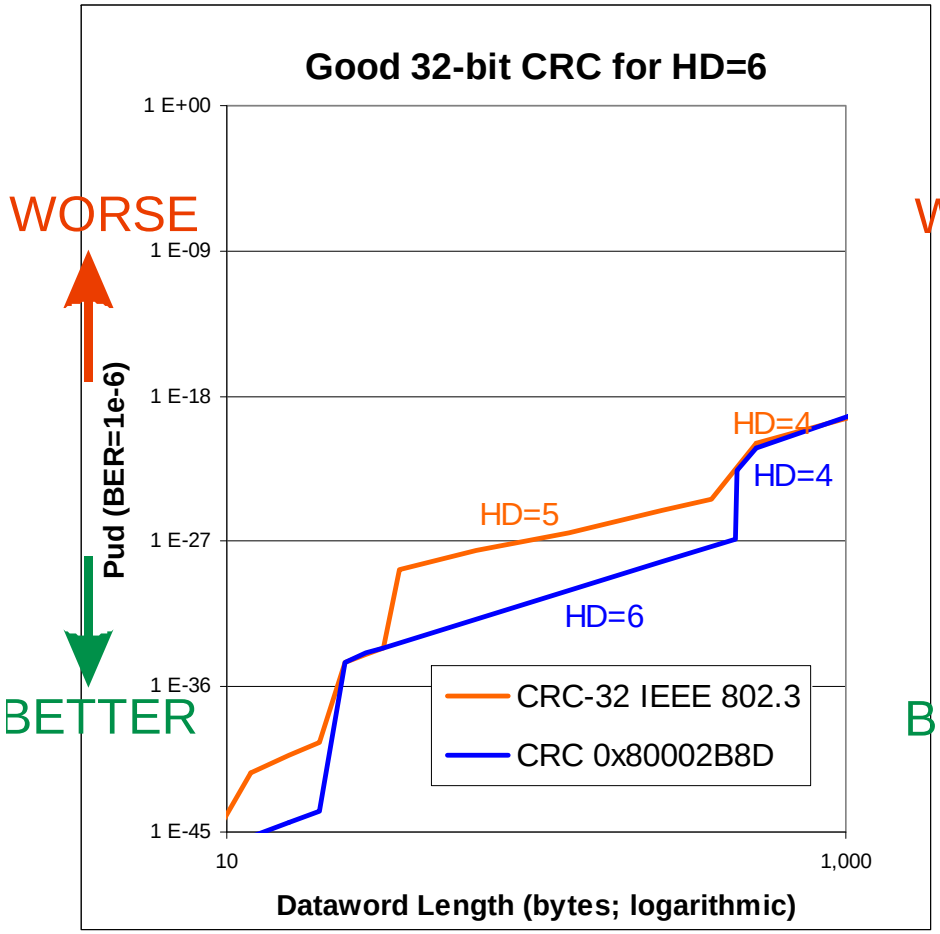
It's All About The HD

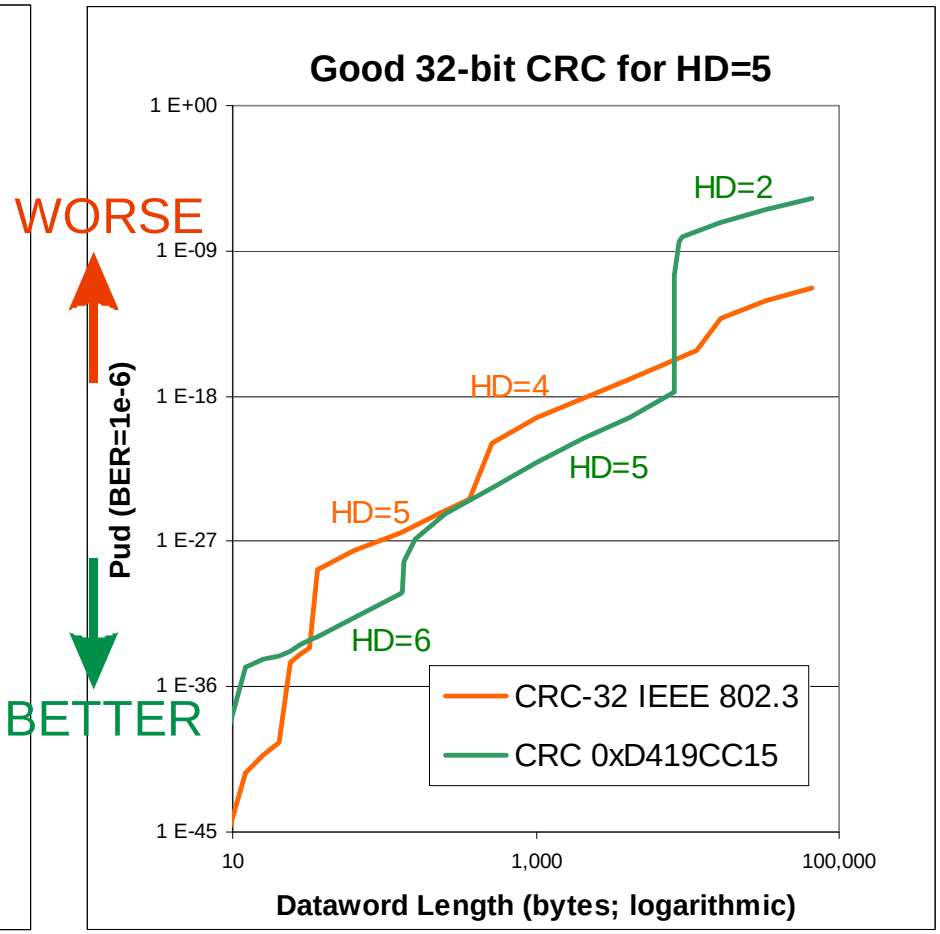
- Better bit mixing of Fletcher & CRC helps at any particular HD



But CRC-32 Is Almost Never Optimal

- No single CRC is "best" everywhere
  - CRC-32 is OK for general purpose use
    ... but gives up 1 or 2 bits of HD and commonly used dataword sizes





Good CRC Choices – 24 & 32 Bits

Polynomial	HD=2	HD=3	HD=4	HD=5	HD=6
0x8F90E3	Good	2,097,148	2,858	74	5
0x9945B1	Longer	--	1,048,575	--	102
0x98FF8C	Longer	--	--	509	28
0xBD80DE	Longer	--	509	--	253
0x80000057	Good	536,870,907	--	346	40
0x80002B8D	Longer	--	268,435,451	--	440
0xD419CC15	Longer	--	--	8188	132
0x90022004	Longer	--	8188	--	4092
0x8F6E37A0 (iSCSI)	Longer	--	268,435,451	--	655
0x82608EDB (CRC-32)	Longer	536,870,907	11,450	371	33

(Table shows maximum dataword length in bytes at that HD; *32-bit selections are preliminary*)

Error Codes Dataword Length (bytes) At HD

Code	HD=2	HD=3	HD=4	HD=5	HD=6
LRC	All	/ / / / / / / /	/ / / / / / / /	/ / / / / / / /	/ / / / / / / /
ADDtwo32	All	/ / / / / / / /	/ / / / / / / /	/ / / / / / / /	/ / / / / / / /
ADDone32	All	/ / / / / / / /	/ / / / / / / /	/ / / / / / / /	/ / / / / / / /
Fletcher32	Longer	8191	/ / / / / / / /	/ / / / / / / /	/ / / / / / / /
ATN32	Longer	760	504	--	12
CRC-32	Longer	~512 MB	11450	371	33

- *Higher at longer dataword lengths is better*
- Given same HD, better bit mixing gives lower undetected fraction
  - Fletcher better mixing than ADD/LRC checksums
  - ATN-32 better mixing than Fletcher32
  - CRC better mixing than Fletcher

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
  - Lookup table stays loaded in cache
  - 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 from memory!
  - We are working on a more definitive speed tradeoff study

MAPPING TO FUNCTIONAL CRITICALITY LEVELS

(SUMMARY VERSION)

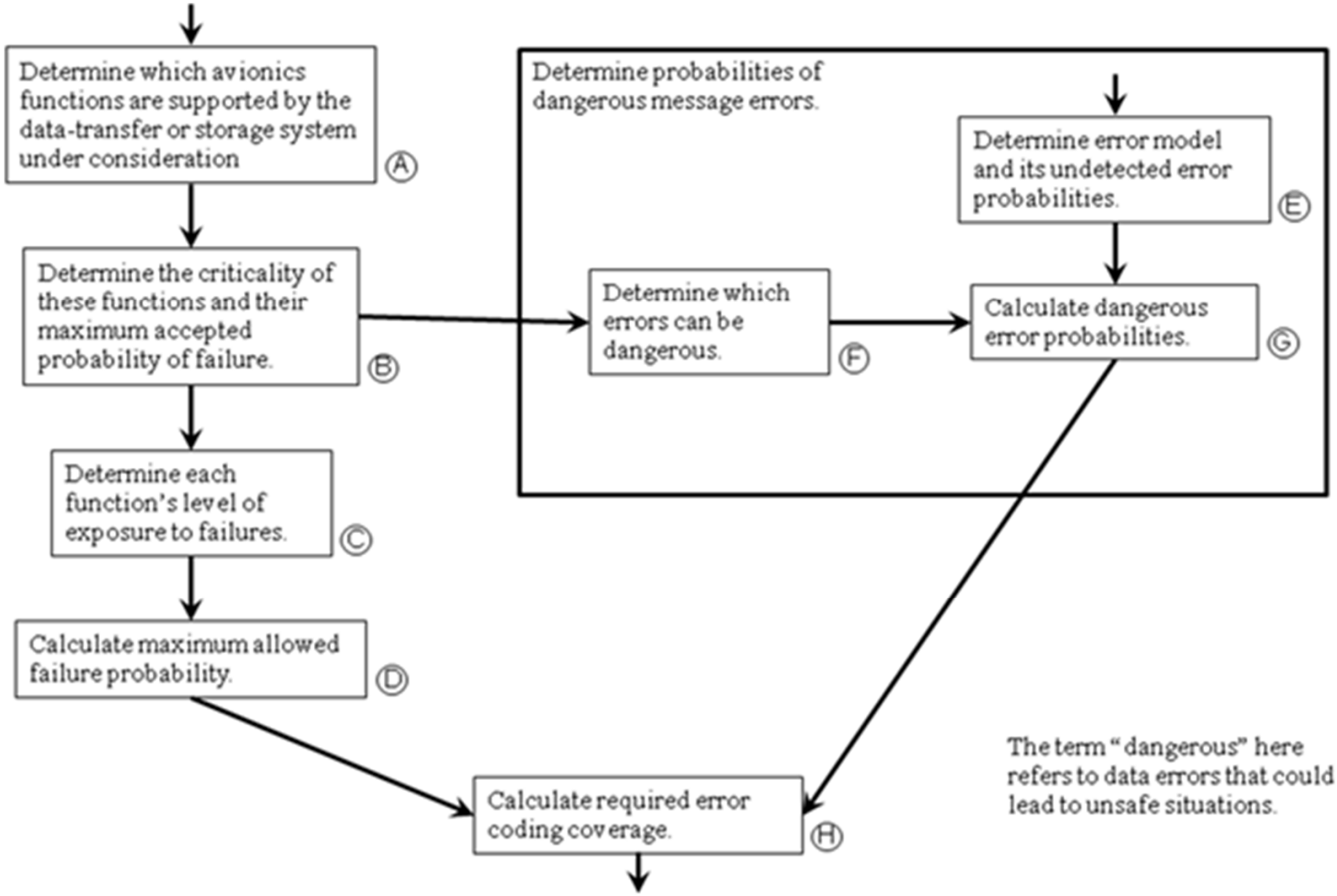
Full version recorded webinar:
http://tinyurl.com/DataIntegrityRecording

(and see backup slides for this presentation)

Functional Criticality Mapping Overview

- No one-size-fits-all definitive rule
  - If we formulated one, it would be excessively conservative
- Need to formulate an error coverage argument based on:
  - Criticality of the function(s) using the data
  - How errors could affect those functions
    - Possibly mitigated by architectural features (e.g. redundancy with voting)
  - Probability of error occurrence given size of data to be protected
  - Possibly other subordinate factors
- In terms of error coding, the dominant factors are:
  - Hamming distance of the code
  - Error exposure (BER, message size, and message rate)
  - Consequence of undetected errors

Requirements Determination Flowchart



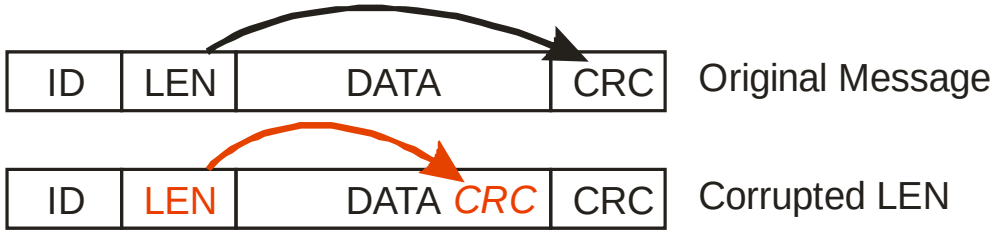
A Simplified Approach to Coverage

- Compute expected number of message errors per hour
  - For all 1-bit errors, 2-bit, 3-bit, etc.
    - Errors per hour is a function of BER, msgs/hr, and total message length
  - At some point the expected number is below your target value
    - Maybe you won't ever see 5 random bit errors in the same message during life of aircraft (1e-9/hr)
- Pick a code with a HD high enough
  - Assume "k" is the number of bit errors below your target
    - For example, 1-, 2-, 3-, 4- bit error probabilities are all above target rate
    - But 5-bit errors are less likely than 1e-9/hr … this means k=5 for you
  - Pick an error code that has HD=k
    - This gives perfect coverage above k, and conservatively assume zero coverage at k
    - If on the border, need detailed checksum coverage info
- Do you use a checksum or CRC?
  - For k=2 a checksum will suffice; sometimes for k=3
  - At k=4 or more, you usually need a CRC
  - CRC always provides better coverage

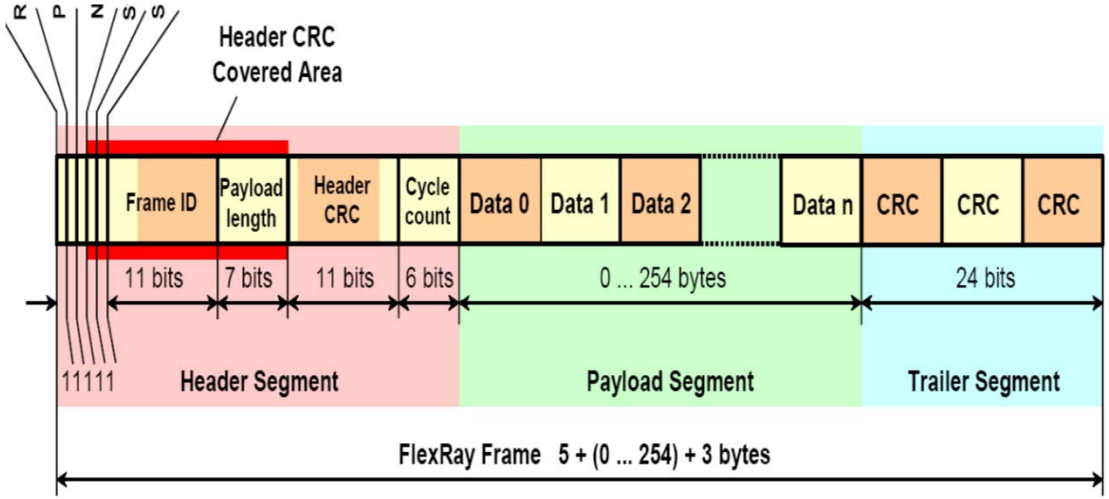
SYSTEM-LEVEL EFFECTS AND CROSS-LAYER INTERACTIONS

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



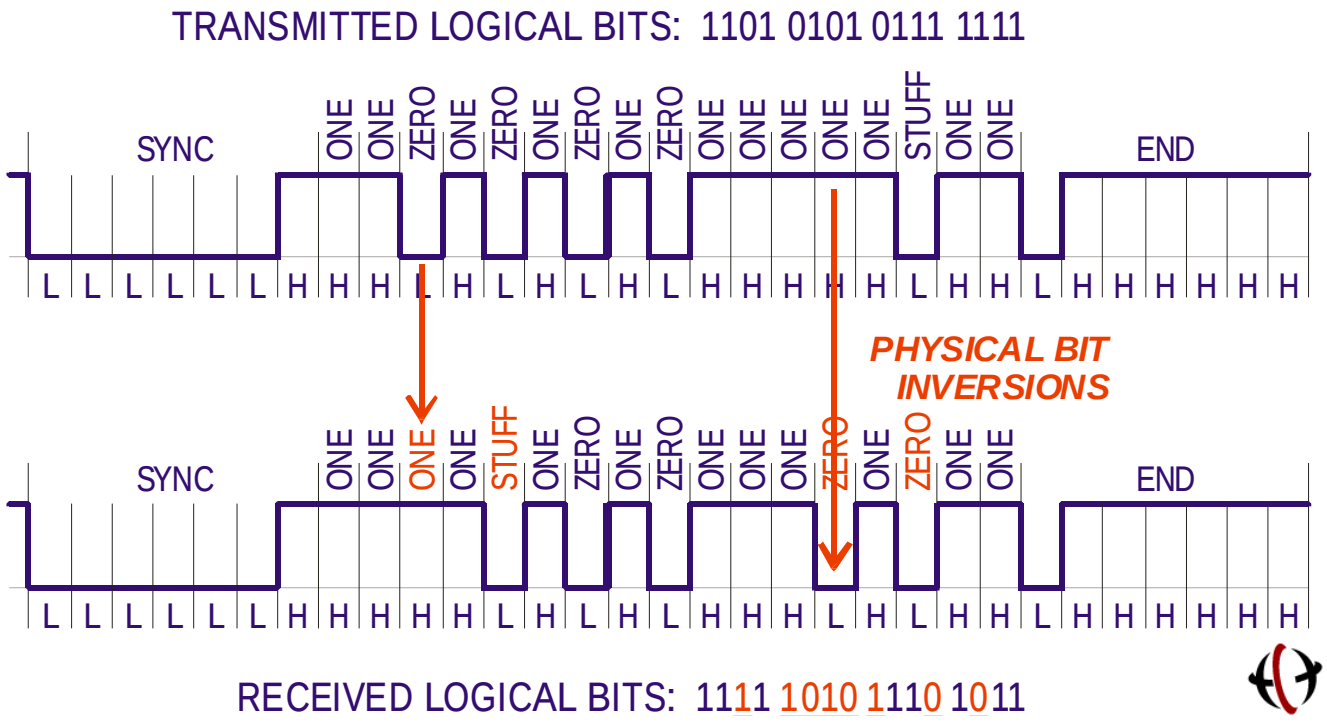
Source: FlexRay Standard, 2004

Figure 4-1: FlexRay frame format.

CAN Bit Stuffing Vulnerability (ARINC-825)

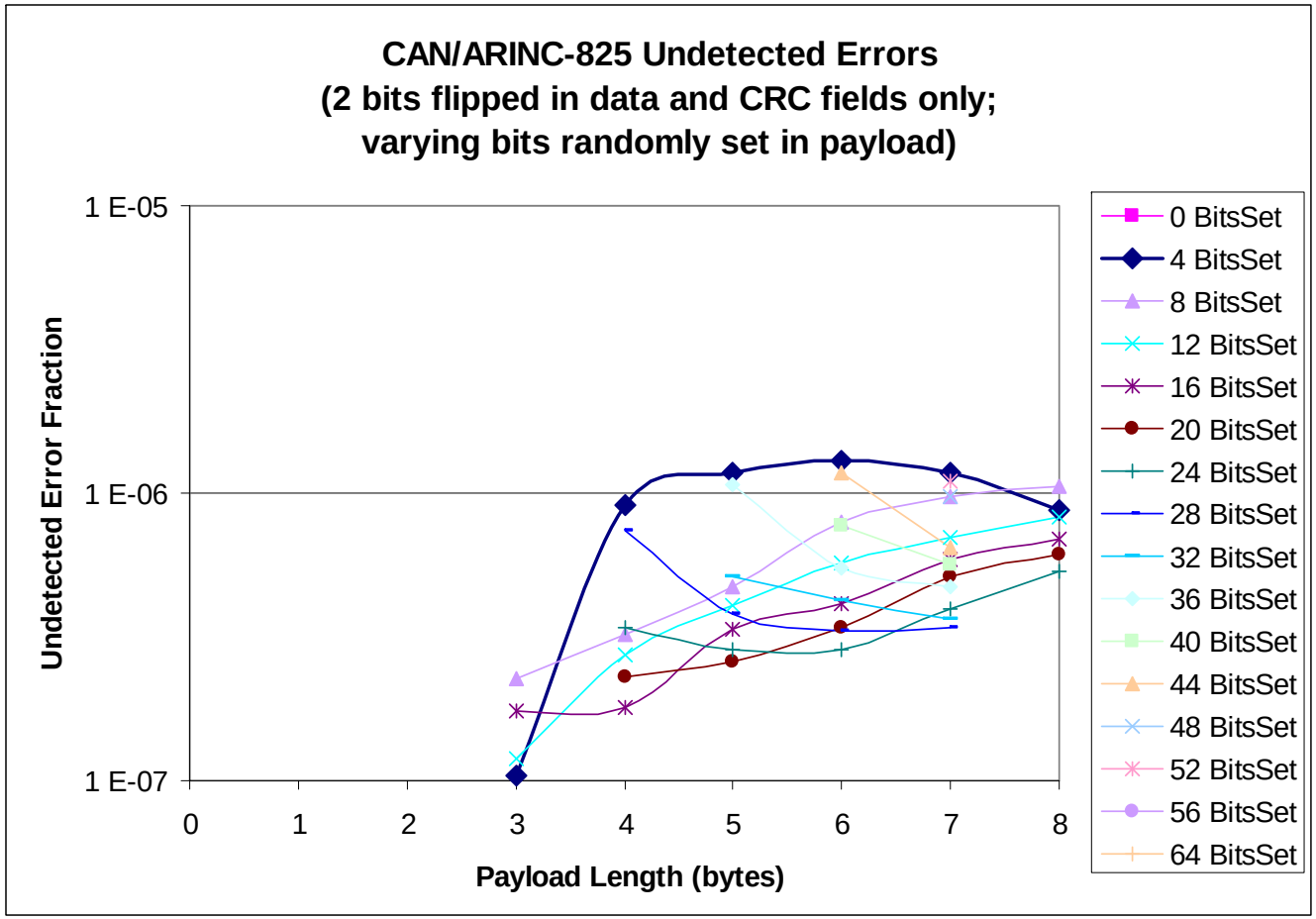
- CAN bit stuffing: add stuff bit after 5-in-a-row
- Two bit errors in just the wrong place cause a problem
  - Cascades to an essentially arbitrary number of bit errors
  - Thus, CAN fails to detect some 2-bit errors in data payload
  - FlexRay uses bytes with start/stop bits to avoid this problem

Cascaded bit stuff error example:



Effect of CAN Bit Stuffing Vulnerability

- Gives effective HD=2 instead of CRC HD=6 for ARINC-825
  - Worst case undetected error fraction at HD=2 is about 1.3E-6
    - (About 3.3% of errors that would be undetected by a good HD=2 code)



ARINC-825 MIC Excerpts (High Integrity)

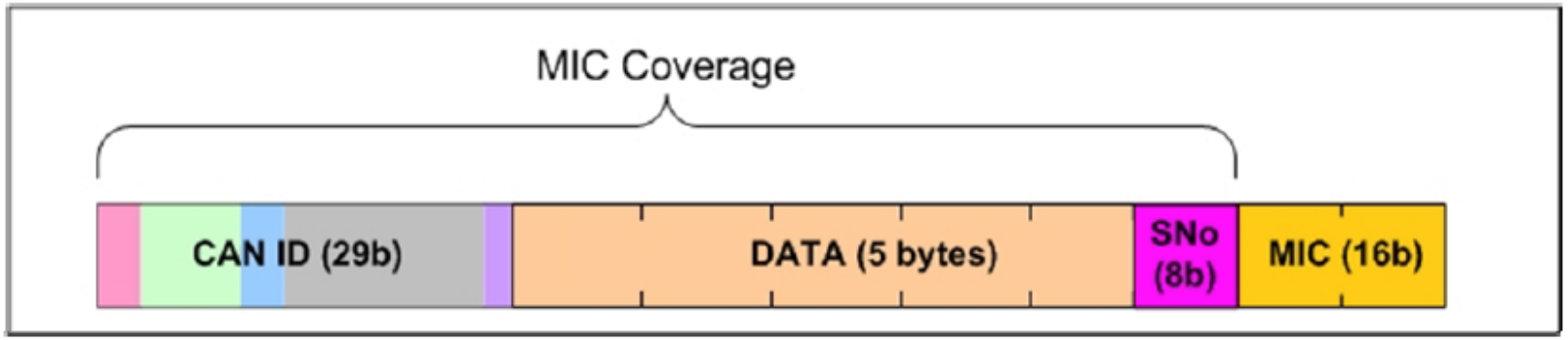


Figure 5-15 – High Integrity Message Protocol Format

5.7.4 Message Integrity Check (MIC)

High integrity messages shall use a 16-bit cyclic redundancy code for the Message Integrity Check (MIC), defined by the following polynomial,

$x^{16}+ x^{15}+ x^{14}+ x^{11}+ x^{6}+ x^{5}+ x^{3}+ x^{2}$ (also known as 0xC86C).

Missing +1 (?)

The MIC shall include the CAN message ID, the data payload, and the SNo.

The following shall be used to compute the CRC:

1. The initial CRC value shall be 0xFFF.
2. Each input byte shall be reflected about its center.
3. The final CRC shall be reflected about its center and XOR with 0xFFF.
4. The message ID is assumed to be 32 bits with the leading 3 bits set to zero.
5. An algorithm that properly computes the CRC shall produce 0xC405 when it computes the CRC on the following test string of characters, “0123456789”.

Why not 0xFFFF?

Couldn’t reproduce this outcome

ARINC-825 MIC Issues

- 16-bit High Integrity Message ID specified by standard
  - 16-bit CRC added to end of message
  - CRC specification has at least typographical errors, maybe more issues depending upon error detection goals
- Apparent errors in ARINC-825 standard
  - Gives credit for HD=5 for base CAN system (pg. 22)
    - Does not take into account the CAN HD=2 problem
  - MIC polynomial is either incorrect or only gives 14-bit error detection effectiveness instead of 16-bit effectiveness (pg. 60)
    - Polynomial given doesn't have a +1 … either typo or incorrect design
    - Seed values seem odd (0xFFF instead of 0xFFFF)
    - Can't reproduce the stated test string
- ARINC-825 working group has been cordial and working with us to update this section of the standard

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?

Multiple CRC Evaluation

- Common to assert that multiple CRCs provide extra benefit
  - For example, two 16-bit CRCs said to be as good as one 32-bit CRC
  - Arguments are, at best, based on factorization math
- For example, one system we've seen uses two 16-bit CRCs
  - One CRC has HD=4
    - Other has worse HD
    - Designed to avoid common factors … but …
  - Searching found a great many 4-bit errors that get past both
  - Result was still HD=4, but with lower undetected error fraction
- Recommendations:
  - Unless it can be proven otherwise via exhaustive analysis, HD not improved by adding a second CRC
    - (In our experience, mathematical arguments based on factorization are flawed)
  - Second CRC without common factors improves undetected error fraction, but usually gives up a lot of HD compared to a single bigger CRC

Parity + Checksum or CRC

- ARINC-629 uses parity per word + checksum
  - Parity for each 16-bit data chunk
  - (a) 16-bit CRC across 256 x 16-bit dataword size = 4096 bits
    OR
  - (b) 16-bit ADDtwo16 across that same 4096 bit dataword

- Analysis:
  - CRC gives HD=4
    - Parity doesn't preclude chance of two words, each with a two-bit error, evading CRC detection
  - ADDtwo16 requires 2 errors to be in different data chunks
    - But, each data chunk has parity, and needs 2 errors in that data chunk to fool parity
    - So, net result is HD=4
  - Conclusion: CRC and ADDtwo16 both give HD=4 for this example when parity is present.
    - CRC gives better undetected error fraction however

Memory-Specific CRC Properties

- Multi-bit errors can cause errors one row apart
  - BUT, which bit they hit depends upon interleaving strategy
  - So, need detailed physical info to give a specific result
- However, there is a CRC property that helps
  - HD for a burst error of size m is the same as a dataword size m
  - If all errors in dataword are within m bits of each other, HD is at least as good as it would be for entire dataword message size m
    - (Almost always better undetected error fraction, but often same HD)
- Example of use:
  - CRC 0x90022004 used on a 4 Gbyte memory image
    - HD=2 for entire memory image (~1/4G undetected fraction for errors)
    - HD=4 for all multi-bit errors within span of 65506 bits
    - HD=6 for all multi-bit errors within span of 32738 bits

High Level Results

- It's all about the Hamming Distance (HD)
  - For BER model, number of 100% detected error bits dominates error detection ratio for higher numbers of bit errors
    - Bit Error Ratio (BER) model assumes random independent bit errors
  - CRCs can have *much* better HD than checksums

- Mapping to criticality levels:
  - There is no one-size-fits all answer per functional criticality level
  - Decide how often system can tolerate an undetected error
    - Based on BER, compute worst number of bit errors in one message
  - Select error code based on HD to detect all expected errors
    - HD gives large improvements (perhaps 1e5 or better per HD improvement)
    - Residual undetected error rate at that HD is a tie-breaker

CRC/Checksum Seven Deadly Sins (Bad Ideas)

1. Picking a CRC based on a popularity contest instead of analysis
   - This includes using "standard" polynomials such as IEEE 802.3
2. Saying that a good checksum is as good as a bad CRC
   - Many "standard" polynomials have poor HD at long lengths
3. Evaluating with randomly corrupted data instead of BER fault model
   - Any useful error code looks good on random error patterns vs. BER random bit flips
4. Blindly using polynomial factorization to choose a CRC
   - It works for long dataword special cases at HD=3 or HD=4, but not beyond that
   - Divisibility by (x+1) doubles undetected fraction on even # bit errors
5. Failing to protect message length field
   - Results in pointing to data as FCS, giving HD=1
6. Failing to pick an accurate fault model and apply it
   - "We added a checksum, so 'all' errors will be caught" (untrue!)
   - Assuming a particular standard BER without checking the actual system
7. Ignoring interaction with bit encoding
   - E.g., bit stuffing compromises HD to give HD=2
   - E.g., good 8b10b encoding seems to be OK, but lax decoding may not be; results depend on specific CRC polynomial

Possible Future Work Topics

- Other evaluation criteria
  - Other fault models (e.g., bit slip)
  - Other error detection effectiveness measures (e.g., data byte ordering errors)
- Better understanding of cross-layer interactions
  - Better understanding of bit encoding effects (some good/some bad)
    - Understanding 8b10b requires multi-burst error analysis techniques
  - Taking credit for message/data framing effects
    - E.g., per-block CRC in addition to entire memory CRC
- Alternative and hybrid approaches
  - Is there a pair of 16-bit CRCs that gives higher HD as a pair?
  - Are cryptographically secure hash functions suitable for HD=1 applications?
- Analytically exact evaluation of other codes
  - 64-bit CRC effectiveness
- High level system effects
  - Interaction of compression and encryption (put CRC outside encrypted data)
  - Are safety layer CRCs really as effective as people think?

Review

- **Introduction**
  - Motivation – most folk wisdom about CRCs & Checksums is incorrect
  - Overview of "error detection concepts for poets"
- **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
- **Selecting a Good CRC**
  - Use data, not folklore
- **Mapping to functional criticality levels** (summary)
  - Hamming Distance matters the most
- **System level effects & cross-layer interactions**
  - CAN/ARINC-825 as an object lesson
- **Q&A**

Questions?

http://users.ece.cmu.edu/~koopman/crc/

http://checksumcrc.blogspot.com/

BACKUP SLIDES

Included in hand-outs to provide more detailed information. Many are discussed in a recorded webinar:

http://tinyurl.com/DataIntegrityRecording

MAPPING TO FUNCTIONAL CRITICALITY LEVELS

(FULL VERSION)

Functional Criticality Mapping Overview

- No one-size-fits-all definitive rule
  - If we formulated one, it would be excessively conservative
- Need to formulate an error coverage argument based on:
  - Criticality of the function(s) using the data
  - How errors could affect those functions
    - Possibly mitigated by architectural features (e.g. redundancy with voting)
  - Probability of error occurrence given size of data to be protected
  - Possibly other subordinate factors
- In terms of error coding, the dominant factors are:
  - Hamming distance of the code
  - Error exposure (BER, message size, and message rate)
  - Consequence of undetected errors

Criticality Mapping Outline

- Some background information
  - –Scope
    - Error detection usage in avionics
    - This program's investigation area
  - –Existing databus guidance (FAA Advisory Circular 20-156 paragraph 4)
- A flowchart of the process
- An explanation for each of the flowchart steps
  - –Fictitious example used to illustrate the steps

Scope – Main Avionics Applications

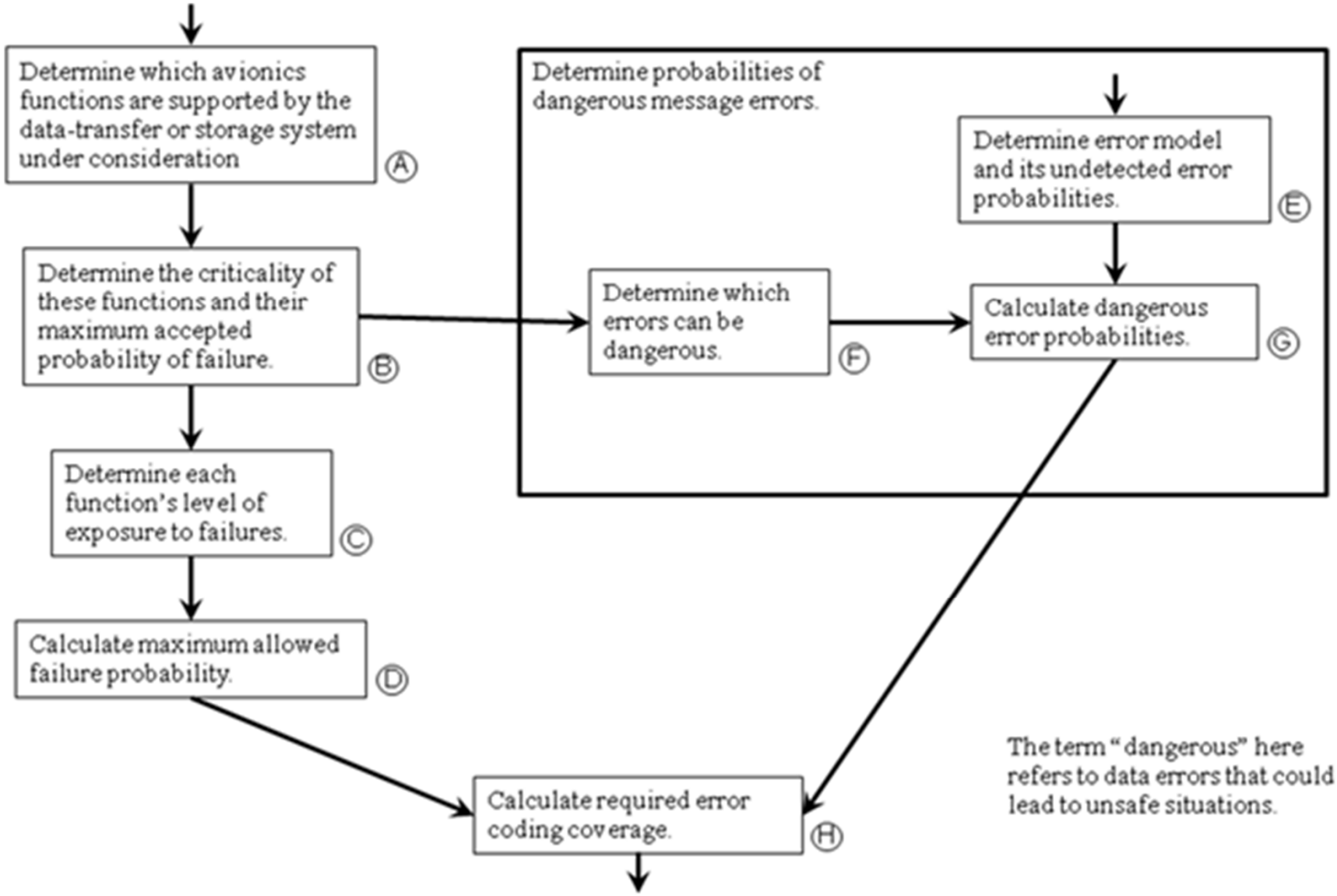
- Error coding for data-transfer systems
  - Dynamic, data is moving
  - Includes on-board data networks and off-board communication
  - Some error detection coding guidance exists (FAA Advisory Circular 20-156 ¶ 4)
- Error coding for memory
  - Static, data is at rest; can be volatile (RAM) or non-volatile (flash, PROM)
  - Memory typically is hierarchical (Flash ➔ RAM ➔ cache levels)
  - No existing specific guidance for error detection coding requirements?*
- Error coding for transferable media
  - More like memory, even though data is moving
  - Magnetic disks, optical discs, and flash devices (USB, SD, etc,)
  - No existing specific guidance for error detection coding requirements?*
- We only look at "random" error coverage
  - Exclude complex IC failures
  - Exclude symbol encoding, data compression, or encryption effects.

* While not guidance, standards such as ARINC 665, RTCA DO-200A, and RTCA DO-201A describe some error detection methods.

Data-Transfer Requirements Background

- Data-transfer systems
  - The text of FAA Advisory Circular 20-156 "Aviation Databus Assurance" that pertains to databus error tolerance is the beginning of its paragraph 4, to wit:
    4. DATA INTEGRITY REQUIREMENTS. Data that passes between LRUs, nodes, switches, modules, or other entities must remain accurate and meet the data integrity of the aircraft functions supported by the databus. To ensure data integrity through error detection and corrections resulting from databus architectures, evaluate databus systems for the following:
    a. The maximum error rate per byte expected for data transmission.
    b. Where applicable, databus-provided means to detect and recover from failures and errors to meet the required safety, availability and integrity requirements. Examples are cyclic redundancy checks, built-in detection, hardware mechanisms, and architecture provisions.
    c. […]
  - While specific to on-board databus systems, applicable to any data transfers
    - RF channels
    - "Sneaker net" transferable media
- We use "data-transfer system" to mean
  - One or more data networks within an aircraft and/or
  - Any other mechanisms that transfer data into an avionics system via messaging

Requirements Determination Flowchart



Example for Coverage Determination

- Fictitious example avionics data-transfer system
  - Picked purposely to not represent any existing or planned system
  - To paraphrase Hollywood: All bits appearing in this example are fictitious. Any resemblance to real bits, living or dead, is purely coincidental.

- Example: autopilot control panel that uses a redundant data bus network to talk to a set of autopilot computers
  - This example (artificially)* increases the criticality of this function by ignoring the fact that pilots are required to know and maintain assigned altitudes, independent of the autopilot

- The steps in the flowchart are broken out in the following sides, with an "Application to example system" bullet that describes how that step would be applied to this example

* some may argue this increase is actually valid, because some pilots rely on the autopilot when they shouldn't

Requirements Determination Flowchart Steps Ⓐ and Ⓑ

- Ⓐ Determine which avionics functions are supported by the data-transfer or storage system under consideration
- Ⓑ Determine the criticality of these functions and their maximum accepted probability of failure.

- Application to example system:
  An FHA/SSA assessment of the system finds that messages from the control panel to the computer send altitude-hold changes as increments or decrements to the existing altitude-hold value and that an undetected error in these increments or decrements could cause an aircraft to try hold an incorrect altitude. This could lead to a mid-air collision, a condition that cannot be allowed with a probably more than 1e-9 per flight-hour (catastrophic failure probability limit per FAA AC 25.1309).

Requirements Determination Step ©

- Determine each function's level of exposure to failures.

- Application to example system:
  Because this system uses increments or decrements to the existing altitude-hold value rather than (re-)sending an absolute value for the altitude hold, any undetected transient error in a data bus message transfer can cause a permanent error in the autopilots' altitude hold value. The latter cannot be allowed to happen with a probability more than 1e-9 per flight-hour. Thus, the system must ensure that undetected transient errors in the data bus system do not exceed this probability. The system has redundancy mechanisms to deal with missing or rejected messages, but no integrity mechanisms above the network. Therefore, the network exposes the system to integrity failures.

Requirements Determination Step Ⓓ

- Calculate maximum allowed failure probability.

- Application to example system:
  Because this network transfers data that must not fail with a probability more than 1e-9 per flight-hour and there is no additional integrity mechanisms in the example system, messages must not have undetectable errors with the probability greater than this.

  – Max acceptable Pud is 1e-9/flight-hour

Requirements Determination Step ©

- Determine error model and its undetected error probabilities.

- Application to example system:
  The autopilot control panel is connected via data buses directly to the autopilot computers, with no intervening switches or other complex logic devices. So, we can use the results of Bit Error Ratio (BER) testing (see BER testing discussion in the following sides) to determine error probabilities. If there had been any complex devices in the data path, there would be no way to determine individual bit error probabilities nor the probabilities for specific bit error patterns.
- For this example, we will say this testing showed a BER of 1e-8 (meets ARINC 664 requirement).
  - Measured BER = 1e-8

Finding a Bit Error Ratio (BER)

- Need to confirm BER by testing
  - Just because standard specifies a BER doesn't mean you meet it
  - Applies to COTS and custom components
- Aircraft aren't the same as cars or offices
  - Connectors and cables typically not per COTS standard
  - Aircraft environment is not the same as home or office environment
  - Standards only require meeting particular test scenarios
  - COTS components only need to work for main customers (e.g. cars)
    - These parts do not have to meet BER requirements for all scenarios
    - In particular, they typically are not designed to meet the standard's BER in an avionics environment with a confidence level commensurate with that required by avionics (e.g. AC 25.1309)
  - Can't rely on test results from COTS part manufacturers/vendors
- Thus, all data-transfer systems need to be BER tested
  - At least during system testing
  - Arguably during system life to detect system degradation

Bit Error Ratio (BER) Testing

- Test BER in avionics worst-case environment (e.g. per DO-160)
- A worst-case bit pattern must be transmitted
  - Network standards typically define these patterns
    - If more than one worst-case bit pattern is defined, all such patterns must be tested
  - Certain pathological bit patterns (such as Ethernet IEEE 100BASE-TX "killer packets") can be ignored if mechanisms prevent them from ever being transmitted
- Need worst-case eye pattern (most degraded signal) at receiver
  - Some test equipment can generate bit patterns with worst-case eyes
  - If this equipment is not available, build a physical analog of the worst-case cable
  - Note that error prediction calculations from eye measurements require a very accurate model of the actual receiver
    - Test equipment that calculate BER from eye measurements cannot be used
    - Need to test the receiver that will be used in the actual flight system
- BER = # bad bits output by receiver-under-test / total bits sent
  - A sufficient number of bits needs to be transmitted such that the confidence in the BER test results is commensurate with that needed by the avionics' certification plan

BER Degradation

- During aircraft life, failures might increase the BER
  - This higher BER might overwhelm error detection mechanisms
  - A fraction (usually about 1/2^k for a k-bit error code) of errors are undetected
    - Higher BER ➔ more likely to be undetected errors
- Testing BER directly on the fly is typically infeasible
  - But, elevated number of detected errors implies elevated BER
  - Suggestion: compute expected detected errors / hour.
    - If this is significantly exceeded, you know BER specification has been violated
    - In response, system must stop trusting data source or network due to high BER
- Important point: detected errors only a symptom, not the problem
  - It is undetected errors that are the problem
  - If a bus or message source is throwing you a lot of erroneous messages, it is very likely a few messages that appear OK actually have undetected errors.

Requirements Determination Step Ⓕ

- Determine which errors can be dangerous.

- Application to example system:
  The messages between the autopilot control panel and the autopilot computer are 64 bytes long*, 4 bits of which are the critical altitude change data. If any of these four bits are corrupted and not detected, the aircraft will to try hold an incorrect altitude.

* 64 bytes is the minimum Ethernet frame (message) size --- including Ethernet addressing, length/type, and CRC fields; all of which are covered by the CRC

Requirements Determination Step ©

- Calculate dangerous error probabilities.

- Application to example system:
  The probability that a message has an attitude data error is the probability of any of the 4 altitude bits being erroneous. With a 1e-8 BER, this probability is:
  $1 - (1 - 1e\text{-}8)^4 \approx 3.999999940000004e\text{-}8$
  If this system sends this message at 20 Hz, then 72,000 messages are sent per hour (20 msgs/sec * 60 sec/min * 60 min/hr). The probability that one or more of these messages has an altitude data error in an hour is:
  $1 - (1 - 3.999999940000004e\text{-}8)^{72,000} \approx 2.875856792805693 4555e\text{-}3$
  Because this is greater than 1e-9, we need to do error detection.

Note: The equation form "1 - (1 - *p*) ^ *n*" gives the probability that one or more of *n* events, each of independent probability *p*, occur. This can be approximated by *n* * *p* if *p* is small enough. However, this approximation can be used only if one is sure that the loss of accuracy is inconsequential. In this case, we can't be sure of that, a priori, since the example's failure probability limit is very small (1e-9 per flight-hour).

Requirements Determination Step ⓗ

- Calculate required error coding coverage.

- The coverage of an error coding design has to be such that:
  - the conditional probabilities of particular error bit patterns (which may depend on BER, number of critical bits, and message length) occurring
  - times the probability of the error coding design failing to detect these patterns
  - is less than the maximum allowable failure probability for a message.

- Application to example system is given on next slide

Requirements Determination Step Ⓗ (cont.)

- Application to example system:
  - Adding **C** to represent simple coverage (the probability that an error in the four altitude bits will be detected) to the dangerous error probabilities equation from step Ⓖ and making it less than 1e-9 gives:
    $1 - (1 - (3.9999999400000004e\text{-}8 * \mathbf{(1 - C)}))^{72,000} \leq 1e\text{-}9$
  - Solving for **C**, gives $\mathbf{C} \geq 0.99999965277777239583569847 6119$
  - Thus, the error detection must have > 99.99997% coverage on each message
  - This would be sufficient coverage for an error detecting code only if all the following were true:
    1) The hardware/software that does the error detection never fails (cannot be guaranteed)
    2) The error code needs to protect only these 4 bits rather than the whole message (rarely done in practice)
    3) The error code itself never suffers any errors (cannot be guaranteed)
  - The first of these would be covered in the next slide with the other two combined and covered in slides after that

Error Detection Mechanism Failure

- For this example scenario, assume that the error detection mechanisms' failure rate is 1e-7 per hour
  - The probability for the error detection mechanisms being functional for t hours is: e ^ -(1e-7 * t)
- With "scrubbing" (testing that the error detection mechanisms are still functioning correctly) before each (3-hour max) flight, the probability of an error detection mechanism failing in flight is less than 3e-7.
  - The combined probability of an error detection mechanism failing and an error occurring in the four critical bits in an hour is: 3e-7 * 2.875856792805693455 5e-3 $\approx$ 8.6e-10 $\leq$ 1e-9
  - Without "scrubbing", the probability of an error detection mechanism failing sometime during the life of aircraft would be too high
- There are two basic ways of doing scrubbing
  - "black box" (applying all possible error patterns and seeing that they all are detected)
  - "white box" (testing the components within the mechanism and their interconnects)
- Typically, "black box" testing is not feasible because of the huge number of possible error patterns
  - Most COTS devices don't have the ability to do "white box" testing
  - This means that the error detection mechanisms within COTS devices usually cannot be trusted and mechanisms that allow themselves to be "white box" tested need to be added to cover COTS devices

Other Errors in the Message

- What if, as is usual, our error detecting code covers whole 64-byte message?
  - HD=5 doesn't guarantee all 4 altitude bits are protected (i.e., not 100% coverage)
    - Especially since errors could also be in error code too
    - So this means we need to do probability analysis to see what is good enough
  - Other bits in message also might be corrupted, using up part of HD quota
  - These bad bits could be anywhere in the message, including in the error code itself
- A practical approach is to treat the whole message as critical
  - Thus: 72,000 messages/hr * 64 bytes/msg
  - $1 - (1 - 1e\text{-}8)^{(64*8)} = 5.119986918e\text{-}6$ errors/msg
  - $1 - (1 - 5.119986918e\text{-}6)^{(72000)} = 0.308326$ errors/hr, but want < 1e-9
- Example, "perfect" 16-bit random error detection HD=1 checksum; $1\text{-}C = 1/2^{16}$
  - $1 - (1 - (5.119986918e\text{-}6 * 2^{-16}))^{(72000)} = 5.62497e\text{-}6$ – not good enough!
- Example, "perfect" 32-bit random error detection HD=1 checksum; $1\text{-}C = 1/2^{32}$
  - $1 - (1 - (5.119986918e\text{-}6 * 2^{-32}))^{(72000)} = 8.79297e\text{-}11$
  - So, it is likely that a good 32-bit checksum will suffice
    - HD=1 is enough in this case; not 100% coverage; but good enough for 1e-9

– If we use HD>1 how does it help?

Using HD To Exploit 16-bit CRC

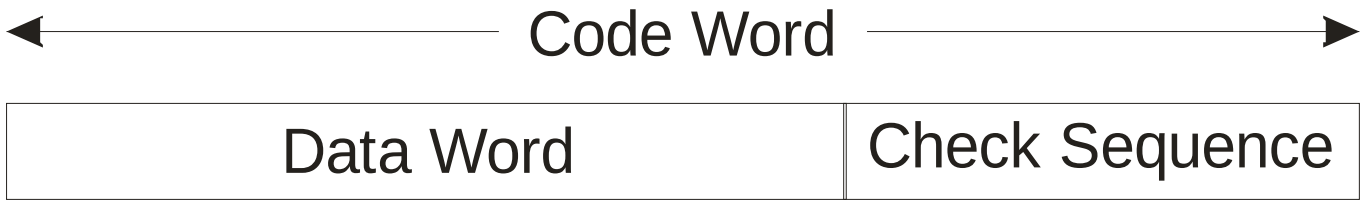
- HD=n means all 1, 2, … n-1 bit errors are detected
- For example, any HD=4 code (e.g., 16-bit CRC) will do the job
  - Select 0xBAAD, which gives HD=4 $(x^{16}+x^{14}+x^{13}+x^{12}+x^{10}+x^8+x^6+x^4+x^3+x+1)$
  - HD=4 coverage (512+16) bits codeword is 64510 undetected / combin(528,4)
- Sum all probabilities that there are k bit errors
  - 1 bit, 2-bit, 3-bit errors – always caught; not relevant
  - 4 bit errors: (all combos * four bits error * rest of bits non-error)
    - combin(528,4) * (1e-8)$^{(4\text{ bad bits})}$ * (1-1e-8)$^{(64*8+16-4\text{ good bits})}$ = 3.201666e-23 errors/message
    - Zero HD=4 coverage: 1 – (1 – 3.201666e-23)$^{(72000\text{ msgs/hr})}$ = 2.3052e-18 undetected errors/hr (any HD=4 code)
    - 0xBAAD coverage: 1 – (1 – 3.201666e-23*(64510 undetected/combin(528,4)))$^{(72000)}$ = 4.6447e-23 undetected errors/hr (good 16-bit CRC)
  - 5 bit errors: contribution too small to really matter
    - combin(528,5)*(1e-8)$^{(5)}$*(1-1e-8)$^{(64*8+16-5)}$ = adds: 3.355346e-29 /message
    - Zero coverage: 1 – (1 – 3.355346e-29)$^{(72000)}$ = adds: 2.415849e-24 / hr
  - No point worrying about very unlikely 6+ bit errors

A Simplified Approach to Coverage

- Compute expected number of message errors per hour
  - For all 1-bit errors, 2-bit, 3-bit, etc.
  - Errors per hour is a function of BER, msgs/hr, and total message length
  - At some point the expected number is below your target value
    - For example, below 1e-9 … in other words you just won't see that many bit errors
- Pick a code with a HD high enough
  - Assume "k" is the number of bit errors below your target
    - For example, 1-, 2-, 3-, 4- bit error probabilities are all above target rate
    - But 5-bit errors are less likely than 1e-9/hr … this means k=5 for you
  - Pick an error code that has HD=k
    - This gives perfect coverage above k, and conservatively assume zero coverage at k
    - If on the border, need detailed checksum coverage info
- Do you use a checksum or CRC?
  - For k=2 a checksum will suffice; sometimes for k=3
  - At k=4 or more, you usually need a CRC
  - CRC always provides better coverage

OTHER BACKUP SLIDE TOPICS

Error Detection Overview



- Error Detection: pick a "good" mathematical hash function
  - Check sequence is hash(data word), e.g., sent after data word in a message
  - Receiver re-computes the hash and checks for code word validity:
    - If hash(data word) == Check Sequence then assume data word is error free
- Common hash function approaches:
  - Checksum: various schemes to "add up" chunks of data word
  - CRC: use a linear feedback shift register arrangement to mix bits
- Want good "bit mixing" to detect combinations of bit errors
  - Undetected error fraction is ratio of bit errors that are undetectable
  - Usually want good error detection properties for *small numbers of bit errors*
  - Hamming Distance (HD): min # bit errors that might be undetected
    - E.g., HD=4 means 100% of 1-, 2-, 3-bit errors are detected

Terminology– Error Codes

- Error Coding
  - Generic term for parity, checksum, or CRC calculation
- Checksum
  - Error code based primarily on addition of data chunks
  - E.g., one's complement sum, Adler, Fletcher, ATN-32
- CRC: Cyclic Redundancy Code
  - Error code based upon polynomial division over GF(2)
- Polynomial: CRC feedback polynomial
  - Primitive polynomial generates maximal length LFSR sequence
    - (These slides use 'implicit +1' notation for hexadecimal values)
  - Polynomial may have prime factors (e.g., divisible by (x+1))
    - Factors of polynomial influence error detection effectiveness
- Code Size
  - Number of bits in the error code result
  - By construction, this is same as check sequence size

Terminology – Error Detection Effectiveness

- Hamming Distance:
  - Minimum number of bit errors that is undetectable
  - Hamming Weight is a number of undetectable errors for a given number of bit errors (e.g., 4161 undetectable 4-bit errors)
- BER: Bit Error Ratio
  - Probability per bit of bisymmetric inversion error (e.g., 10^{-6})
    - Usually assumes independent random inversions
  - P_{ud}: Probability of undetected error, assuming some BER
    - NOTE: even if very small, there are many messages across system life
- Burst Error
  - Arbitrary corruption of k consecutive bits
- Error Check
  - Determining whether a check sequence matches the data word
  - Compute code on data word and match result to check sequence

CRC Performance Data Example

http://users.ece.cmu.edu/~koopman/crc/

```
# 0x15 = x^5 +x^3 +x +1  (0x2b) <=> (0x1a; 0x35)
#Len     Poly     HW(2)    HW(3)    HW(4)    HW(5)    HW(6)    HW(7)
4        0x15     0        0        10       0        4        0        1
5        0x15     0        0        16       0        12       0        2
6        0x15     0        0        25       0        27       0        8
7        0x15     0        0        38       0        52       0        31
8        0x15     0        0        55       0        96       0        85
9        0x15     0        0        77       0        168      0        201
10       0x15     0        0        105      0        280      0        433
11       0x15     1        0        133      0        469
12       0x15     2        0        168      0        742
13       0x15     3        0        211      0        1127
14       0x15     4        0        263      0        1659
15       0x15     5        0        324      0        2388
16       0x15     6        0        397      0        3352
```

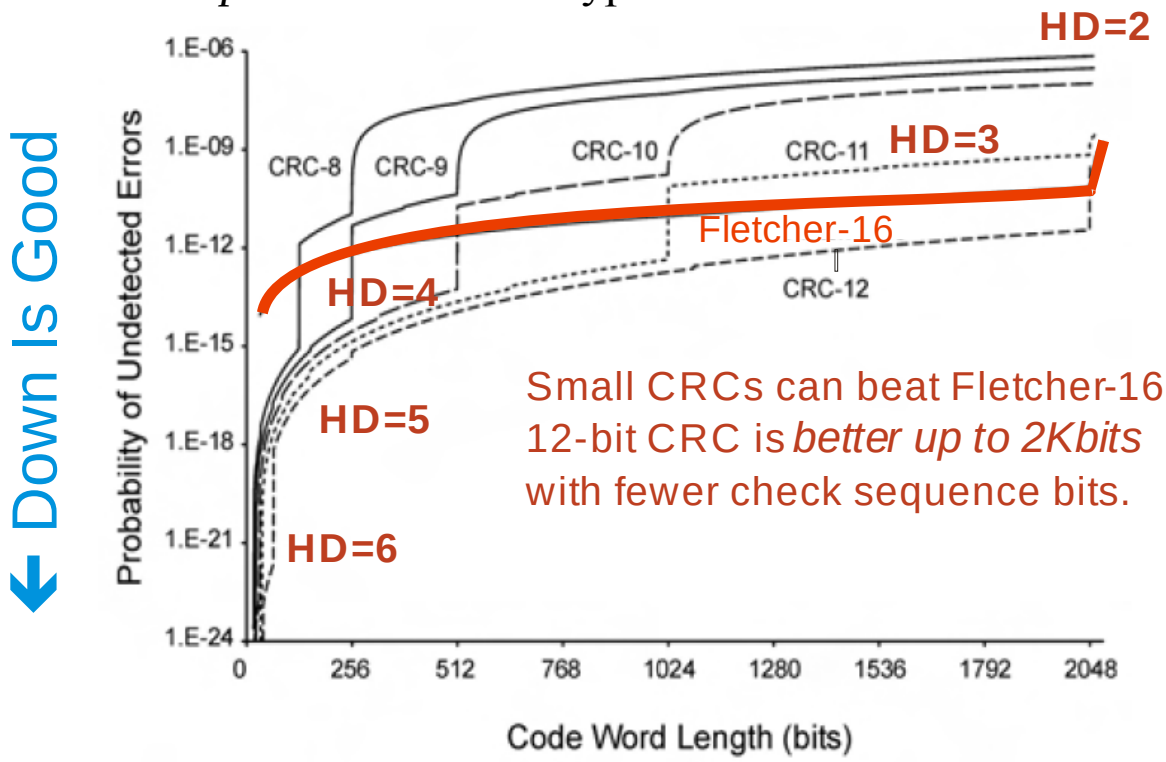
- **Interpretation for 5-bit CRC 0x15 (0x2b in all-bits notation)**
  - HD=4 from dataword length of 4 bits through 10 bits
  - HD=2 at and above dataword length of 10 bits (non-zero Hamming Weight in HW(2) column)
  - At dataword length=16 there are 6 undetected 2-bit errors, zero undetected 3 bit errors, 397 undetected 4-bit errors, and 3352 undetected 6-bit errors.
  - At dataword length=16 there are combination((16+5),4) = 5985 possible 4 bit errors in dataword and CRC, so probability of undetected error given that 4 independent bit errors have occurred in one message is 4/ combination((16+5),4) = 0.000668 = 0.0668%
  - Note that probability of a 2-bit error is much higher, and that Bit Error Rate also has to be taken into account
  - Performance for 0x1a is identical since it is the mirror polynomial

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

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.

Here There Be Dragons…

- 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.

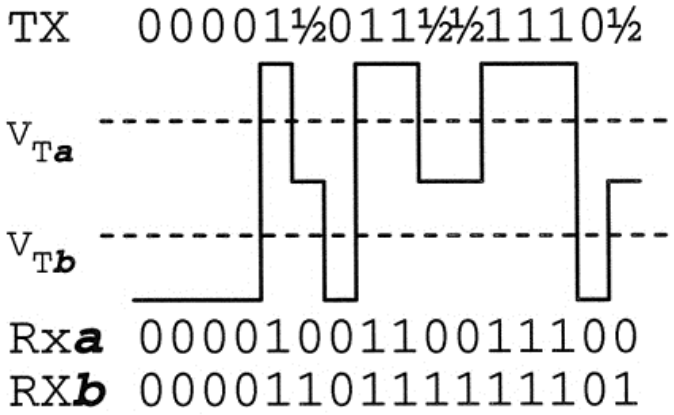
8b10b Encoding Effects for Gbit Ethernet

- Each 8 bits of data physically encoded as 10 bits
  - Maintains DC balance … but …
  - … not enough DC-balanced patterns in 10 bits … so …
  - … uses fairly complex "Running Disparity" imbalance tracking
    - [James05] shows that strict RD book-keeping and detection of encoding errors is required to avoid compromising error detection

- **Results:**
  - Concern that this could lead to cascaded bit errors
  - About a month of random simulations on 8 CPU cores did not find a problem
    - So it is a rare problem if one exists .. but hard to say how rare it is
  - Based on informal analysis looks like issue will be that one bit transmission errors will cause bursts of 5 bit data errors
    - Further analysis required to prove that no HD=2 or HD=3 vulnerability caused by this effect
    - Open research topic: multi-burst coverage of IEEE 802.3 polynomial

Byzantine CRC

- Byzantine failures for CRCs and Checksums



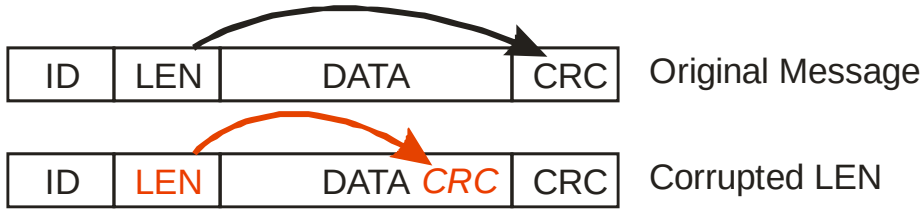
Example Schrodinger's CRC caused by non-saturated voltage values on a data bus. Two receivers (***a*** and ***b***) can see the same message as having two different values, and each view having a valid CRC

  - 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

Unprotected Length Field

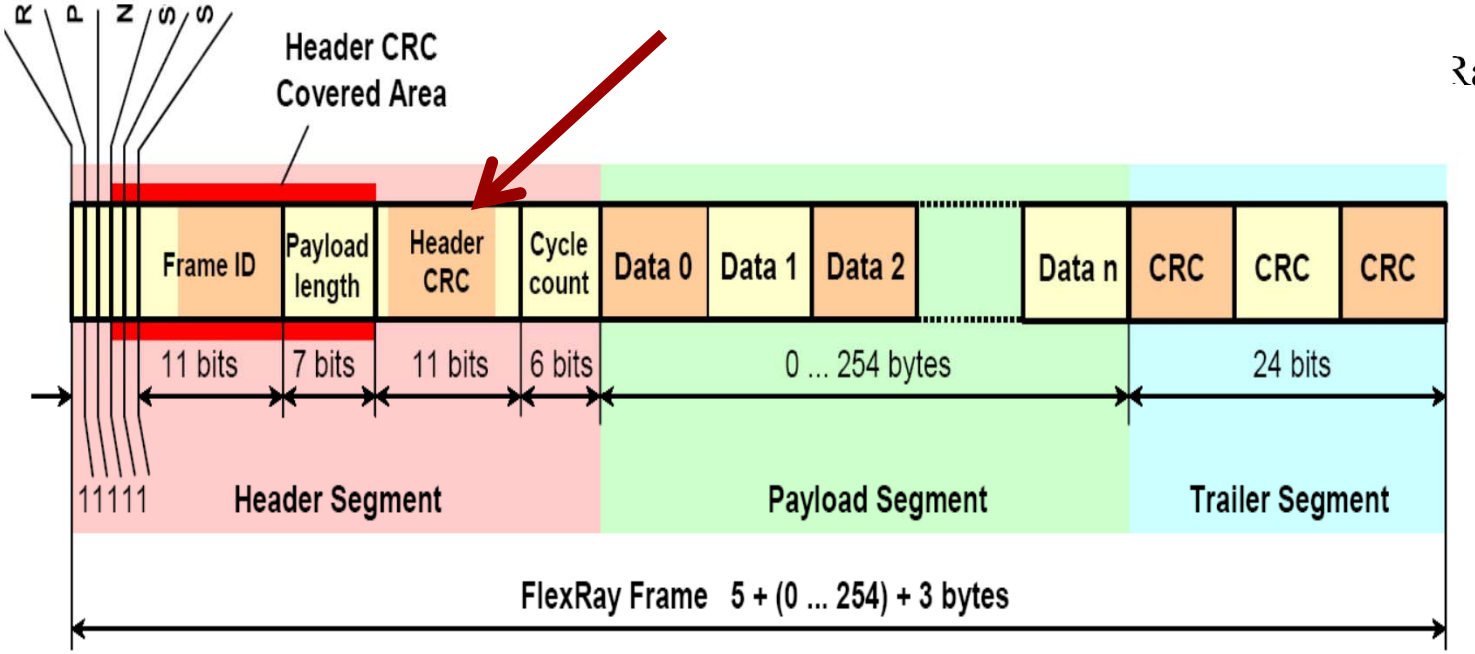
- Many protocols have an unprotected length field
  - Error in length field can point to wrong CRC location
  - Wrong CRC location means 1 bit error (in length field) can escape CRC detection (HD=1)



- Common mitigation methods
  - Parity on length field (ARINC-629)
  - Assert that incorrect length message always caught
    - Many, including CAN/ARINC-825
  - Independent CRC on length/header field (FlexRay)

CAN vs. FlexRay Length Field Corruptions

- FlexRay solves this with a header CRC



Ray Standard, 2004

Figure 4-1: FlexRay frame format.

- Recommendation:
  - It is *essential* that length field corruptions be caught, or it invalidates the HD-based safety arguments
  - That means there must be validation of length field corruption being caught

Acronyms

- AC Advisory Circular
- ARINC Aeronautical Radio, Incorporated
- ATN Aeronautical Telecommunication Network
- BER Bit Error Ratio
- CAN Controller Area Network
- CCITT Comité Consultatif International Téléphonique et Télégraphique
- COTS Commercial, Off-The-Shelf
- CRC Cyclic Redundancy Code
- FAA Federal Aviation Administration
- FCS Frame Check Sequence
- HD Hamming Distance
- HW Hamming Weight
- IEC International Electrotechnical Commission
- IEEE Institute of Electrical and Electronics Engineers
- KB KiloBytes (1024 bytes)
- LFSR Linear Feedback Shift Register
- LRC Longitudinal Redundancy Check
- MB MegaBytes (1024 KB)
- Mbit MegaBits (1024K bits for memory, 1 million bits for networks)
- Pud Probability of Undetected Error
- RAM Random Access Memory
- RTCA RTCA, Inc. (formerly Radio Technical Commission for Aeronautics)
- TCP Transmission Control Protocol
- XOR Exclusive Or