These tutorials are a simplified introduction, and are not sufficient on their own to achieve system safety. You are responsible for the safety of your system.

~“I have a bad feeling about this.”~
— Star Wars, Episode k \( k = 1 \ldots 9 \)
Anti-Patterns for Data Integrity:
- No checks on memory data
  - Program image and configuration
  - RAM and other data integrity
- No end-to-end message checks
  - Using checksum instead of CRC

Memory & data integrity
- Detecting data corruption:
  - Mirroring, Parity & SECMED codes, Checksum, CRC
  - If data word consistent with error code, then no detectable error
  - Random hash as a starting point: random k-bit error code by chance misses $1/2^k$ errors
- Malicious faults require cryptographically strong integrity check
  - All error codes discussed here are easy to attack
Sources of Data Faults

Hardware faults
- Network message bit flips
- Bad EEPROM/Flash writes
- “Bit rot” (storage degrades over time)

Single event upsets: Soft Errors
- Affect both memory & CPU logic
- Error detecting codes usually don’t help with CPU logic faults!

Software corruption
- Bad pointers, buffer overflow, etc.
Overview of Data Integrity Mechanisms

- **Key term: Hamming Distance (HD)**
  - Smallest # of bit flips possibly undetected
  - Flips across data value and error code
  - Higher HD is better (more errors detected)

- **Parity**: detects single bit errors (HD=2)
  - Store one bit that holds XOR of all bits

- **Mirroring** (HD=2, but cheap computation)
  - Store data twice: plain and inverted bits
    - E.g.: 0x55 → {0x55, 0xAA} two-byte pair

- **SEC**: (Hamming Code) correct single bit errors

- **SECDED**:
  - Single Error Correction, Double Error Detection
  - Use a Hamming Code + parity bit to give HD=4
  - Size approximately $1 + \log_2$ (number of data bits)

<table>
<thead>
<tr>
<th>HD</th>
<th>Flips Detected</th>
<th>Flips Undetected</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>None</td>
<td>1+</td>
<td>No Error Code</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2+</td>
<td>Parity, Checksum, Mirroring, Any CRC</td>
</tr>
<tr>
<td>3</td>
<td>1-2</td>
<td>3+</td>
<td>Hamming (SEC), Some CRCs, Short Fletcher</td>
</tr>
<tr>
<td>4</td>
<td>1-3</td>
<td>4+</td>
<td>Some CRCs, SECDED</td>
</tr>
<tr>
<td>5+</td>
<td>HD-1</td>
<td>HD+</td>
<td>Good CRC</td>
</tr>
</tbody>
</table>
“Add” up all the data bits
- XOR all data words (HD=2)
  - Detects 1-bit errors
- 2’s complement addition (HD=2)
  - Detects 1-bit and most 2-bit errors
- 1’s complement addition (HD=2)
  - Wraps carry bit, so slightly better

Complex checksums:
- Fletcher checksum (HD=2, HD=3)
  - Keeps two running 1’s comp. sums
  - HD=3 at short lengths, HD=2 at long lengths
- Adler checksum (HD=2, HD=3)
  - Uses prime moduli counters
  - Usually Fletcher is better (and faster)

Error rate BER = $10^{-6}$
Cyclic Redundancy Check (CRC)

The mechanism:
- Shift and XOR of selected feedback bits
- Accumulated residue in shift register is the CRC “checksum” value

The math:
- The data and the feedback bit pattern are both binary coefficient polynomials
- Error code is remainder from polynomial division of data by feedback over GF(2)

Feedback polynomial selection matters
- Some popular polynomials are poor choices, including international standards (!)
- Some rules of thumb are misguided (e.g., (x+1) divisibility for high HD)
- Best polynomials are found via brute force search of exact evaluations

> See: https://users.ece.cmu.edu/~koopman/crc/
Best Practices For Data Integrity

- Ensure sufficient data integrity
  - CRC on network packets
  - Periodic CRC on flash/EEPROM data
  - Appropriate memory integrity check on RAM

- Pitfalls:
  - Assuming mirroring is enough
    - What about data on stack?
    - What about data inside operating system?
  - Assuming memory data integrity is all you need
    - What about corrupted calculations?
  - Using a checksum when you should use a CRC
  - Many subtle pitfalls for the unwary. See FAA report: https://goo.gl/uKFmHr
nano? REAL PROGRAMMERS USE emacs

HEY. REAL PROGRAMMERS USE vim.

WELL, REAL PROGRAMMERS USE ed.

NO, REAL PROGRAMMERS USE cat.

REAL PROGRAMMERS USE A MAGNETIZED NEEDLE AND A STEADY HAND.

EXCUSE ME, BUT REAL PROGRAMMERS USE BUTTERFLIES.

THEY OPEN THEIR HANDS AND LET THE DELICATE WINGS FLAP ONCE.

THE DISTURBANCE RIPPLES OUTWARD, CHANGING THE FLOW OF THE EDDY CURRENTS IN THE UPPER ATMOSPHERE.

WHICH ACT AS LENSES THAT DEFLECT INCOMING COSMIC RAYS, FOCUSING THEM TO STRIKE THE DRIVE PLATTER AND FLIP THE DESIRED BIT.

NICE. 'COUSE, THERE'S AN EMACS COMMAND TO DO THAT.

OH YEAH! GOOD OL' C-x M-c M-butterfly...

DAMMIT, EMACS.