A Practitioner‘s Guide to
Software-based Soft-Error Mitigation Using AN-Codes

Peter Ulbrich

15th IEEE International Symposium on High Assurance Systems Engineering
January 09, 2013
Soft Errors – A Growing Problem

- **Soft-Errors (Transient hardware faults)**
  - Caused by (cosmic) radiation
Soft Errors – A Growing Problem

- **Soft-Errors (Transient hardware faults)**
  - Caused by (cosmic) radiation
  - **Performance (technology) vs. reliability**
- **Soft-Errors (Transient hardware faults)**
  - Caused by (cosmic) radiation
  - Performance (technology) vs. reliability

- **Software-based fault-tolerance**
  - Selective and resource-efficient (costs!)
  - Vital component: *Arithmetic error coding* (AN codes)
The Combined Redundancy Approach (CoRed) [1]

Ulbrich, Peter; Hoffmann, Martin; Kapitza, Rüdiger; Lohmann, Daniel; Schmid, Reiner; Schröder-Preikschat, Wolfgang: "Eliminating Single Points of Failure in Software-Based Redundancy", EDCC 2012.
Combined Redundancy Approach (CoRed) [1]

The Combined Redundancy Approach (CoRed) [1]

Triple Modular Redundancy

(1) Ulbrich, Peter; Hoffmann, Martin; Kapitza, Rüdiger; Lohmann, Daniel; Schmid, Reiner; Schröder-Preikschat, Wolfgang: *Eliminating Single Points of Failure in Software-Based Redundancy*, EDCC 2012.
The Combined Redundancy Approach (CoRed) [1]

(1) Ulbrich, Peter; Hoffmann, Martin; Kapitza, Rüdiger; Lohmann, Daniel; Schmid, Reiner; Schröder-Preikschat, Wolfgang: "Eliminating Single Points of Failure in Software-Based Redundancy", EDCC 2012.
The Combined Redundancy Approach (CoRed) [1]

(1) Ulbrich, Peter; Hoffmann, Martin; Kapitza, Rüdiger; Lohmann, Daniel; Schmid, Reiner; Schröder-Preikschat, Wolfgang: "Eliminating Single Points of Failure in Software-Based Redundancy", EDCC 2012.
The Combined Redundancy Approach (CoRed) [1]

→ Key element: CoRed Dependable Voter

(1) Ulbrich, Peter; Hoffmann, Martin; Kapitza, Rüdiger; Lohmann, Daniel; Schmid, Reiner; Schröder-Preikschat, Wolfgang: "Eliminating Single Points of Failure in Software-Based Redundancy", EDCC 2012.
Problem Statement

Goals:
- Full 1-bit fault coverage
- Get what you’re paid for

Implementation:
- UAV Flight-Control
- DanceOS – Safety RTOS
- KESO Embedded JVM

Peter Ulbrich – ulbrich@cs.fau.de
Problem Statement

Goals:
- Full 1-bit fault coverage
- Get what you’re paid for

Implementation:
- UAV Flight-Control
- DanceOS – Safety RTOS
- KESO Embedded JVM

Problems:
- Experiments showed discrepancies (in line with [3])
- Implications on error probability?

Peter Ulbrich – ulbrich@cs.fau.de
Problem Statement

Goals:
- Full 1-bit fault coverage
- Get what you’re paid for

Implementation:
- UAV Flight-Control
- DanceOS – Safety RTOS
- KESO Embedded JVM

Problems:
- Experiments showed discrepancies (in line with [3])
- Implications on error probability?

→ Practitioners cannot blindly rely on coding theory!
Agenda

- Introduction

- Background
  - The CoRed Dependable Voter
  - Arithmetic Error Coding

- Think Binary
  - Choosing Appropriate Keys
  - Pitfall 1: Mapping Code to Binary

- Know Your Compiler & Architecture
  - Pitfall 2: Inter-Instruction State
  - Pitfall 3: Undefined Execution Environment
  - Multi-Bit Faults – A Glimpse

- Conclusions & Lessons Learned
The *CoRed* Dependable Voter – Basics

- **Complex encoded comparison operation**
- **Data-flow integrity**
  - Input: Variants \((X_C, Y_C, Z_C)\)
  - Output: Constant signature \((B_E)\) and encoded winner \((W_C)\)
  - Validation: Subsequent check (decode)
- **Control-flow integrity**
  - **Static signature** (expected value): Compile-time
    \(\rightarrow\) Used as return value \(E\)
  - **Dynamic signature** (actual value): Runtime
    \(\rightarrow\) Applied to winner \(W_C\)
Arithmetic Error Coding – Basics

- **General coding theory**
  - Data word + redundant information = code word
  - Fault detection $\rightarrow$ distance between code words
Arithmetic Error Coding – Basics

- **General coding theory**
  - Data word + redundant information = code word
  - Fault detection → distance between code words

- **Arithmetic error codes**
  - Can cope with computational flaws
  - Arithmetic operators (+, -, ×, =, ...)

\[ \nu_c = A \cdot \nu \]

Encoded value       Constant (Key)       Value
Arithmetic Error Coding – Basics

- **General coding theory**
  - Data word + redundant information = code word
  - Fault detection → distance between code words

- **Arithmetic error codes**
  - Can cope with computational flaws
  - Arithmetic operators (+, -, ×, =, …)

\[ \nu_c = A \cdot \nu + B_v + D \]

- Encoded value
- Constant (Key)
- Value
- Signature
- Timestamp

Peter Ulbrich – ulbrich@cs.fau.de
What to Expect? – Residual Error Probability

- Silent Data Corruption (SDC)
  - Undetectable code-to-code word mutation

- Residual error probability
  - Chance for a SDC
  - Fundamental property for safety assessment

\[
p_{sd}(\frac{1}{A})
\]

\[
p_{sd} = \frac{\text{valid code words}}{\text{possible code words}} \approx \frac{1}{A}
\]

→ The bigger key \( A \), the better?

Peter Ulbrich – ulbrich@cs.fau.de
Agenda

- Introduction
- Background
  - Arithmetic Error Coding
  - The CoRed Dependable Voter
- Think Binary
  - Choosing Appropriate Keys
  - Pitfall 1: Mapping Code to Binary
- Know Your Compiler & Architecture
  - Pitfall 2: Inter-Instruction State
  - Pitfall 3: Undefined Execution Environment
  - Multi-Bit Faults – A Glimpse
- Conclusions & Lessons Learned
Think Binary – Choosing Appropriate Keys?

- **Theory:** *prime numbers* [4]
  - Intuitively plausible
  - Non-primes suitable as well? [3]
Think Binary – Choosing Appropriate Keys?

- **Theory:** prime numbers [4]
  - Intuitively plausible
  - Non-primes suitable as well? [3]

- **Practitioner’s approach:** min. Hamming distance
  - Distance \( d \) between code words (# unequal bits)
  - \( d-1 \) bit error detection capabilities

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

\( d = 2 \)
Think Binary – Choosing Appropriate Keys?

- **Theory:** prime numbers [4]
  - Intuitively plausible
  - Non-primes suitable as well? [3]

- **Practitioner’s approach:** min. Hamming distance
  - Distance \( d \) between code words (# unequal bits)
  - \( d-1 \) bit error detection capabilities

- **Brute force**
  - \( 1.4 \times 10^{14} \) experiments for all 16 bit \( As \)
Think Binary – Choosing Appropriate Keys?

- **Theory:** prime numbers [4]
  - Intuitively plausible
  - Non-primes suitable as well? [3]

- **Practitioner’s approach:** min. Hamming distance
  - Distance \( (d) \) between code words (# unequal bits)
  - \( d-1 \) bit error detection capabilities

- **Brute force**
  - \( 1.4 \times 10^{14} \) experiments for all 16 bit \( A \)s
  - \( A = 58,368 \quad d_{\text{min}} = 2 \quad \# \text{errors detectable} = 1 \)
Think Binary – Choosing Appropriate Keys?

- Theory: prime numbers [4]
  - Intuitively plausible
  - Non-primes suitable as well? [3]

- Practitioner’s approach: min. Hamming distance
  - Distance \( d \) between code words (# unequal bits)
  - \( d-1 \) bit error detection capabilities

- Brute force
  - \( 1.4 \times 10^{14} \) experiments for all 16 bit \( A \)s
    
    \[
    \begin{array}{c|cccc}
    A  & 1 & 0 & 1 & 0 \\
    58,368 & d_{\text{min}} = 2 & \text{#errors detectable} = 1 \\
    58,831 & 3 & 2 \\
    \end{array}
    \]
Think Binary – Choosing Appropriate Keys?

- **Theory: prime numbers** [4]
  - Intuitively plausible
  - Non-primes suitable as well? [3]

- **Practitioner’s approach: min. Hamming distance**
  - Distance ($d$) between code words (# unequal bits)
  - $d$-1 bit error detection capabilities

- **Brute force**
  - $1.4 \times 10^{14}$ experiments for all 16 bit $A$s

<table>
<thead>
<tr>
<th>$A$</th>
<th>$d_{\text{min}}$</th>
<th>#errors detectable</th>
</tr>
</thead>
<tbody>
<tr>
<td>58,368</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>58,831</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>58,659</td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>

|x| y| d = 2|
Think Binary – Choosing Appropriate Keys?

- **Theory:** prime numbers [4]
  - Intuitively plausible
  - Non-primes suitable as well? [3]

- **Practitioner’s approach:** min. Hamming distance
  - Distance (d) between code words (# unequal bits)
  - d-1 bit error detection capabilities

- **Brute force**
  - 1.4×10¹⁴ experiments for all 16 bit A's
    
    | A   | d_min | #errors detectable |
    |-----|-------|--------------------|
    | 58,368 | 2     | 1                  |
    | 58,831 | 3     | 2                  |
    | 58,659 | 6     | 5                  |

→The bigger the better is misleading!
Double Check – Implementation in the Spotlight

- **Fault-simulation → entire fault-space**
  - Each and every $A$, $v$ and fault pattern
  - $6.5 \times 10^{16}$ experiments for 16 bit $A$s and 1-8 bit soft errors
Fault-simulation → entire fault-space
- Each and every $A$, $v$ and fault pattern
- $6.5 \times 10^{16}$ experiments for 16 bit $A$s and 1-8 bit soft errors

→ Excess of predicted residual error probability
Fault-simulation → entire fault-space

- Each and every $A$, $\nu$ and fault pattern
- $6.5 \times 10^{16}$ experiments for 16 bit $A$s and 1-8 bit soft errors

→ Excess of predicted residual error probability
→ Mismatch with Hamming distance experiments
Pitfall 1: Mapping Code to Binary

- **Pitfall 1:** Binary representation of code words
  - Coding theory is unaware of machine word sizes
  - Dangerous over- and underflow conditions
Pitfall 1: Mapping Code to Binary

- **Pitfall 1: Binary representation of code words**
  - Coding theory is unaware of machine word sizes
  - Dangerous over- and underflow conditions

- **EAN Patch**: decode($v_C$, A, B, D)
  - Additional range checks → Prevent code space violation
Agenda

- Introduction
- Background
  - Arithmetic Error Coding
  - The CoRed Dependable Voter
- Think Binary
  - Choosing Appropriate Keys
  - Pitfall 1: Mapping Code to Binary
- Know Your Compiler & Architecture
  - Pitfall 2: Inter-Instruction State
  - Pitfall 3: Undefined Execution Environment
  - Multi-Bit Faults – A Glimpse
- Conclusions & Lessons Learned
Analysing the Assembly

- **Fault-Injection with FAIL* [5]**
  - Based on Bochs simulator
  - *Each and every* register, flag, instruction and execution path
  - Fault-space pruning → Feasibility
Analysing the Assembly

- **Fault-Injection with FAIL* [5]**
  - Based on Bochs simulator
  - *Each and every* register, flag, instruction and execution path
  - Fault-space pruning $\rightarrow$ Feasibility

- **Experimental setup**
  - Implementation: C++
  - Compiler: GCC 4.7.2-5 (IA32), -O2
  - Footprint:

<table>
<thead>
<tr>
<th></th>
<th>CoRed Voter</th>
<th>Simple Voter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions</td>
<td>92</td>
<td>38</td>
</tr>
<tr>
<td>Memory (Bytes)</td>
<td>301</td>
<td>112</td>
</tr>
</tbody>
</table>

- RTOS: Spatial and temporal isolation
Analysing the Assembly

- **Fault-Injection with FAIL* [5]**
  - Based on Bochs simulator
  - *Each and every* register, flag, instruction and execution path
  - Fault-space pruning → Feasibility

- **Experimental setup**
  - Implementation: C++
  - Compiler: GCC 4.7.2-5 (IA32), -O2
  - Footprint:

<table>
<thead>
<tr>
<th></th>
<th>CoRed Voter</th>
<th>Simple Voter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions</td>
<td>92</td>
<td>38</td>
</tr>
<tr>
<td>Memory (Bytes)</td>
<td>301</td>
<td>112</td>
</tr>
</tbody>
</table>

- RTOS: Spatial and temporal isolation

→ *Violation of predicted fault-detection capabilities*
Know your Compiler and Architecture

- Pitfall 2: Architecture specifics
  - Example: Absence of compound test-and-branch
  - Control-flow information is stored in single bit
    → Redundancy is lost

```c
/* if (a == b) */
cmp eax, ebx
je Lequal
```
Pitfall 2: Architecture specifics
- Example: Absence of compound test-and-branch
- Control-flow information is stored in single bit
  → Redundancy is lost

EAN Patch: $\text{apply}(v_c, \text{sig}_{\text{DYN}})$
- Malicious control-flow → Signature overflow → Additional check

```c
/* if (a == b) */
cmp eax, ebx
je Lequal
```
Know your Compiler and Architecture

- **Pitfall 2: Architecture specifics**
  - Example: Absence of compound test-and-branch
  - Control-flow information is stored in single bit
    → Redundancy is lost
  - **EAN Patch:** apply($v_C$, $\text{sig}_{\text{DYN}}$)
    - Malicious control-flow → Signature overflow → Additional check

- **Pitfall 3: Undefined Execution Environment**
  - **Compiler laziness** leaves encoded values in registers
  - **Zombie values** → leaking from caller to voter function
    → Isolation assumptions violated

```c
/* if (a == b) */
cmp eax, ebx
je Lequal
```
Know your Compiler and Architecture

- **Pitfall 2: Architecture specifics**
  - Example: Absence of compound test-and-branch
  - Control-flow *information is stored in single bit* → Redundancy is lost

- **EAN Patch**: apply\((v_c, \text{sig}_\text{DYN})\)
  - Malicious control-flow → Signature overflow → Additional check

- **EAN Patch**: vote\((x_c, y_c, z_c)\)
  - Cleaning the local storage restores isolation

- **Pitfall 3: Undefined Execution Environment**
  - Compiler laziness leaves encoded values in registers
  - Zombie values → leaking from caller to voter function → Isolation assumptions violated

```c
/* if (a == b) */
cmp eax, ebx
je Lequal
```
## 3 Fault-Injection Campaigns:

- Instructions and
- General purpose registers and CPU flags
- Program counter

<table>
<thead>
<tr>
<th></th>
<th>Instructions</th>
<th>Registers and Flags</th>
<th>Program Counter</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Simple</td>
<td>CoRed</td>
<td>Simple</td>
</tr>
<tr>
<td>OK</td>
<td>784</td>
<td>2772</td>
<td>1040</td>
</tr>
<tr>
<td>Detected (Code)</td>
<td>-</td>
<td>995</td>
<td>-</td>
</tr>
<tr>
<td>Detected (Trap)</td>
<td>93</td>
<td>246</td>
<td>8</td>
</tr>
<tr>
<td>Detected (Isolation)</td>
<td>825</td>
<td>1834</td>
<td>1825</td>
</tr>
<tr>
<td>Detected (Timeout)</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Undetected (SDC)</td>
<td>450</td>
<td>0</td>
<td>807</td>
</tr>
</tbody>
</table>
### Fault-Injection Campaigns – Final Results

<table>
<thead>
<tr>
<th></th>
<th>Instructions</th>
<th>Registers and Flags</th>
<th>Program Counter</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Simple</td>
<td>CoRed</td>
<td>Simple</td>
</tr>
<tr>
<td>OK</td>
<td>784</td>
<td>2772</td>
<td>1040</td>
</tr>
<tr>
<td>Detected (Code)</td>
<td>-</td>
<td>995</td>
<td>-</td>
</tr>
<tr>
<td>Detected (Trap)</td>
<td>93</td>
<td>246</td>
<td>8</td>
</tr>
<tr>
<td>Detected (Isolation)</td>
<td>825</td>
<td>1834</td>
<td>1825</td>
</tr>
<tr>
<td>Detected (Timeout)</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Undetected (SDC)</td>
<td>450</td>
<td>0</td>
<td>807</td>
</tr>
</tbody>
</table>

#### 3 Fault-Injection Campaigns:
- Instructions and
- General purpose registers and CPU flags
- Program counter

→ **CoRed dependable voter performs as EXPECTED!**
Multi-Bit Faults – The Good, the Bad and the ...

<table>
<thead>
<tr>
<th></th>
<th>Good A = 58,659</th>
<th>Bad A = 58,368</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>38,639</td>
<td>38,639</td>
</tr>
<tr>
<td>Detected (Code)</td>
<td>21,596</td>
<td>21,519</td>
</tr>
<tr>
<td>Detected (Trap)</td>
<td>47</td>
<td>47</td>
</tr>
<tr>
<td>Detected (Isolation)</td>
<td>60,438</td>
<td>60,438</td>
</tr>
<tr>
<td>Detected (Timeout)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Undetected</td>
<td>0</td>
<td>77</td>
</tr>
</tbody>
</table>

- **2-bit Fault-injection experiments**
  - Full fault space coverage
  - Triple check fault-detection capabilities

- **Distances:** $d_{\text{good}} = 6$, $d_{\text{bad}} = 2$
## Multi-Bit Faults – Tighten the Rules

<table>
<thead>
<tr>
<th></th>
<th>3-bit faults</th>
<th>4-bit faults</th>
<th>5-bit faults</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>33.742%</td>
<td>33.605%</td>
<td>33.544%</td>
</tr>
<tr>
<td>Detected (Code)</td>
<td>18.209%</td>
<td>18.356%</td>
<td>18.431%</td>
</tr>
<tr>
<td>Detected (Trap)</td>
<td>0.001%</td>
<td>&lt;0.001%</td>
<td>0%</td>
</tr>
<tr>
<td>Detected (Isolation)</td>
<td>47.993%</td>
<td>48.030%</td>
<td>48.023%</td>
</tr>
<tr>
<td>Detected (Timeout)</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Undetected</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Fault Space</td>
<td>$3.59 \times 10^6$</td>
<td>$1.03 \times 10^8$</td>
<td>$2.90 \times 10^9$</td>
</tr>
<tr>
<td>Coverage</td>
<td>16.13%</td>
<td>0.59%</td>
<td>0.04%</td>
</tr>
</tbody>
</table>
Agenda

■ Introduction

■ Background
  ■ Arithmetic Error Coding
  ■ The CoRed Dependable Voter

■ Think Binary
  ■ Choosing Appropriate Keys
  ■ Pitfall 1: Mapping Code to Binary

■ Know Your Compiler & Architecture
  ■ Pitfall 2: Inter-Instruction State
  ■ Pitfall 3: Undefined Execution Environment
  ■ Multi-Bit Faults – A Glimpse

■ Conclusions & Lessons Learned

Peter Ulbrich – ulbrich@cs.fau.de
Conclusions & Lessons Learned

Soft Errors – A Growing Problem

- Soft-Errors (Transient hardware faults)
  - Caused by (cosmic) radiation
  - Performance (technology) vs. reliability
- Software-based fault-tolerance
  - Selective and resource-efficient (costs!)
  - Vital component: Arithmetic error coding (AN codes)

→ Software-based fault-tolerance is hard to implement
→ Missing tool support
Conclusions & Lessons Learned

Soft-Errors (Transient hardware faults)

- Caused by (cosmic) radiation
- Permanence (technology) vs. reliability

Software-based fault-tolerance

- Selective and resource-efficient (costs!)
- Vital component: Arithmetic (AN codes)

Soft Errors – A Growing Problem

- Software-based fault-tolerance is hard to implement
- Missing tool support

Pitfall 1: Mapping Code to Binary

- Pitfall 1: Binary representation of code words
  - Coding theory is unaware of machine word sizes
  - Dangerous over- and underflow conditions

- **EAN Patch:** \( \text{decode}(v_c, A, B, D) \)
  - Additional range checks → Prevent code space violation

  \[
  \text{EAN Decode} \rightarrow W
  \]
Conclusions & Lessons Learned

Know your Compiler and Architecture

- **Pitfall 2: Architecture specifics**
  - Example: Absence of compound test-and-branch
  - Control-flow information is stored in single bit
  - Redundancy is lost

- **EAN Patch: apply\(v_C, \sigma_{\text{DYN}}\)**
  - Malicious control-flow $\rightarrow$ Signature overflow $\rightarrow$ Additional check

- **Pitfall 3: Undeﬁned Execution Environment**
  - Compiler laziness leaves encoded values in registers
  - Zombie values $\rightarrow$ leaking from caller to voter function
  - Isolation assumptions violated

→ Little obvious source of vulnerabilities
→ Tight feedback loop with FI required
→ Isolation and OS-support mandatory
Conclusions & Lessons Learned

Know your Context

Multi-Bit Faults – Tighten the Rules

<table>
<thead>
<tr>
<th></th>
<th>3-bit faults</th>
<th>4-bit faults</th>
<th>5-bit faults</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>33.742%</td>
<td>33.605%</td>
<td>33.544%</td>
</tr>
<tr>
<td>Detected (Code)</td>
<td>18.209%</td>
<td>18.356%</td>
<td>18.431%</td>
</tr>
<tr>
<td>Detected (Trap)</td>
<td>0.001%</td>
<td>&lt;0.001%</td>
<td>0%</td>
</tr>
<tr>
<td>Detected (Isol)</td>
<td></td>
<td></td>
<td>48.023%</td>
</tr>
<tr>
<td>Detected (Time)</td>
<td></td>
<td></td>
<td>0%</td>
</tr>
<tr>
<td>Undetected</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fault Space</td>
<td>3.59 × 10^6</td>
<td>1.03 × 10^8</td>
<td>2.90 × 10^9</td>
</tr>
<tr>
<td>Coverage</td>
<td>16.13%</td>
<td>0.59%</td>
<td>0.04%</td>
</tr>
</tbody>
</table>

→ Tooling speed is crucial!
Conclusions & Lessons Learned

Combined Redundancy Approach

→ Key element: CoRed Dependable Voter

(1) Ulbrich, Peter; Hoffmann, Martin; Kapitza, Rüdiger; Lohmann, Daniel; Schmid, Reiner; Schröder-Preikschat, Wolfgang: "Eliminating Single Points of Failure in Software-Based Redundancy". EDCC 2012.
Thank you!

Implementation and further experimental results:
http://www4.cs.fau.de/Research/CoRed