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

Peter Ulbrich

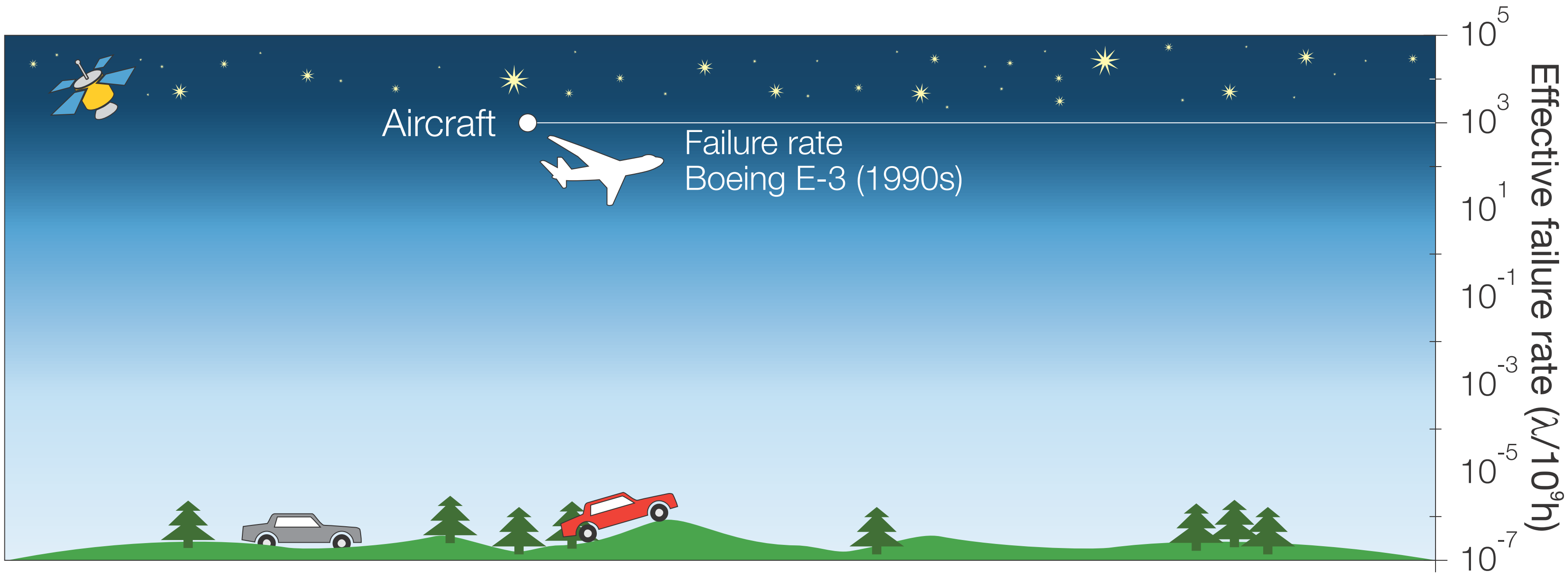
15th IEEE International Symposium on High Assurance Systems Engineering
January 09, 2013

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

System Software Group

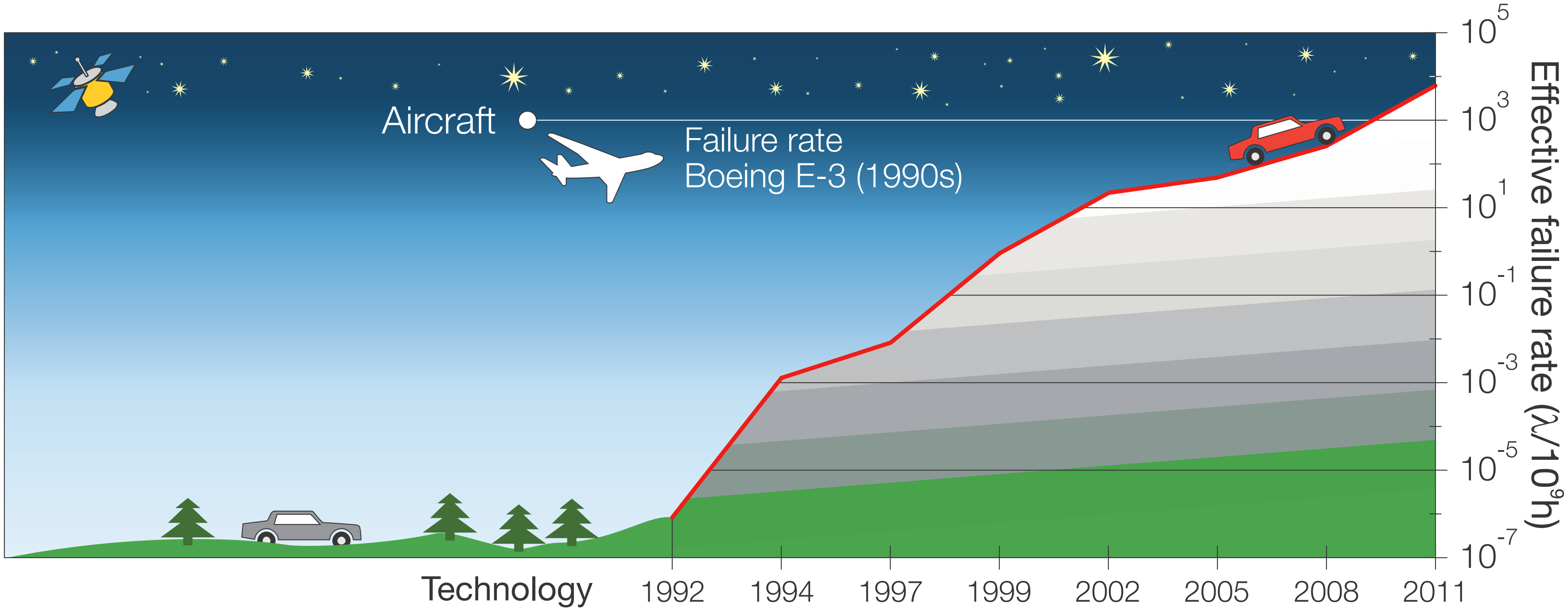
Embedded Systems Initiative

Soft Errors – A Growing Problem



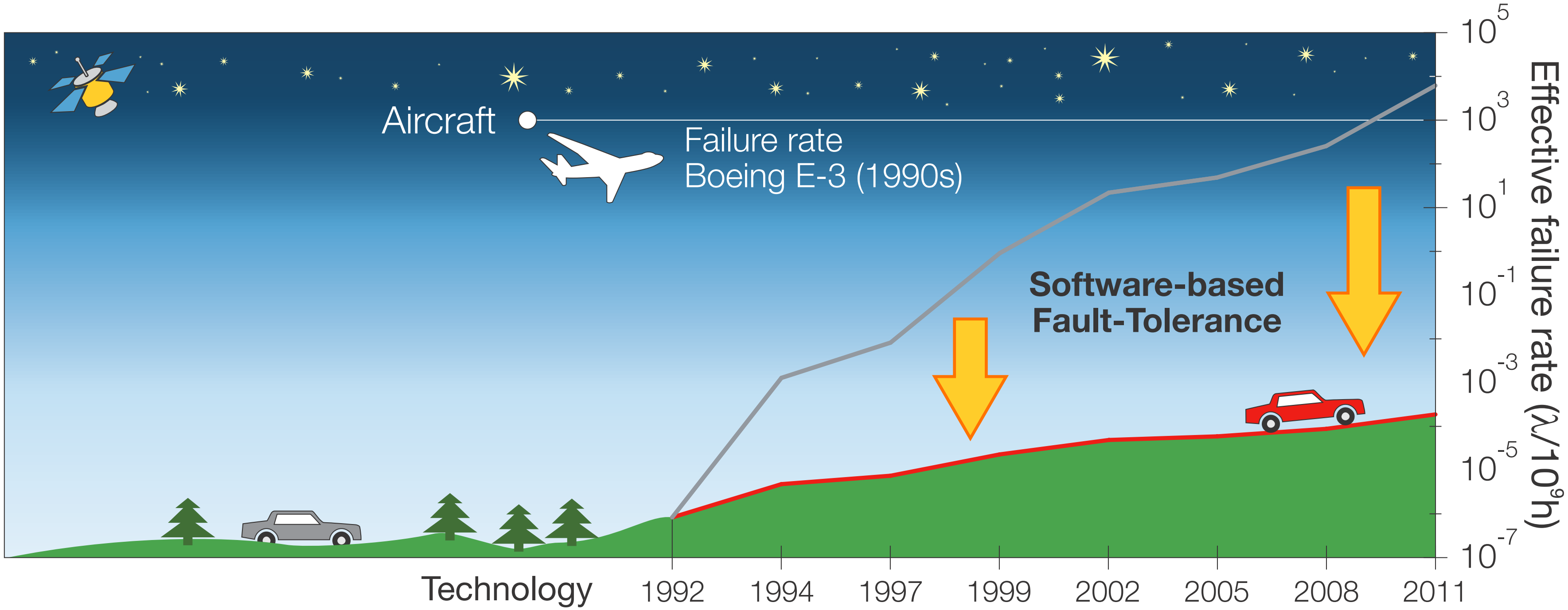
- **Soft-Errors (Transient hardware faults)**
  - Caused by (cosmic) radiation

Soft Errors – A Growing Problem



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

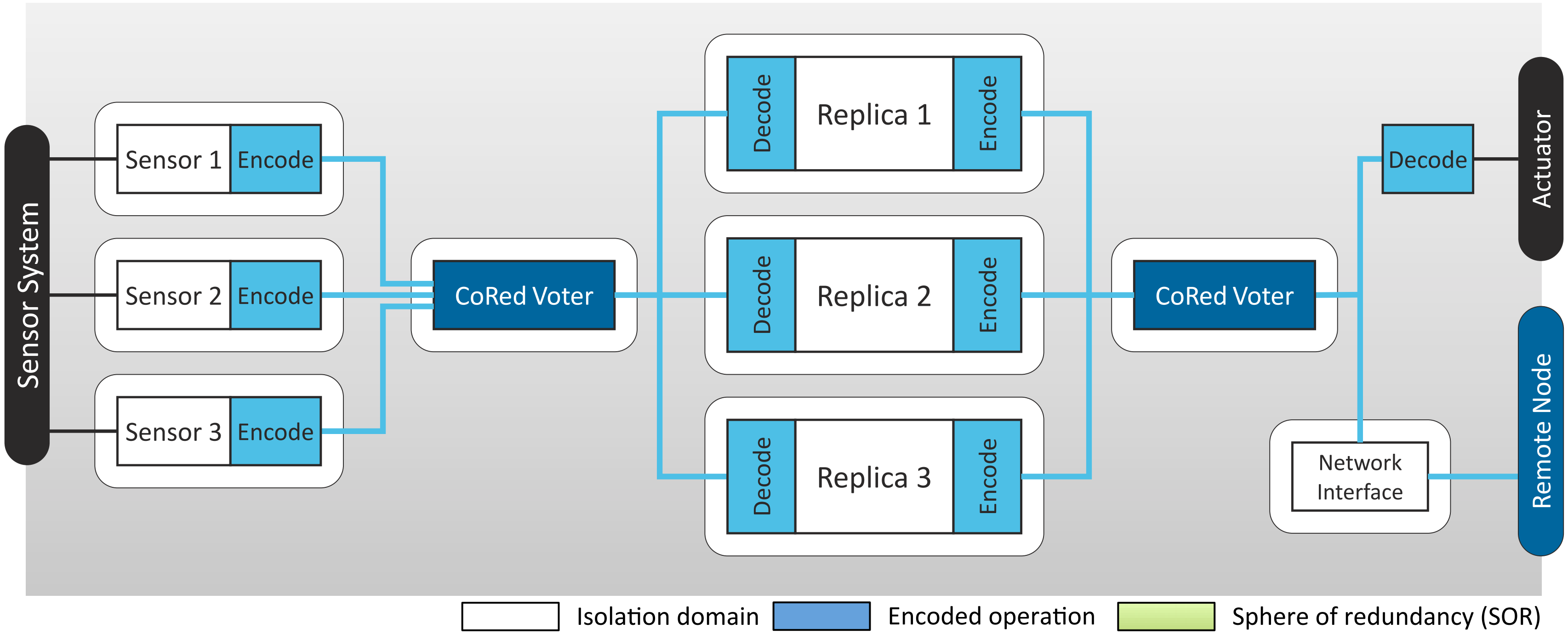
Soft Errors – A Growing Problem



- **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)

Combined Redundancy Approach

CoRed

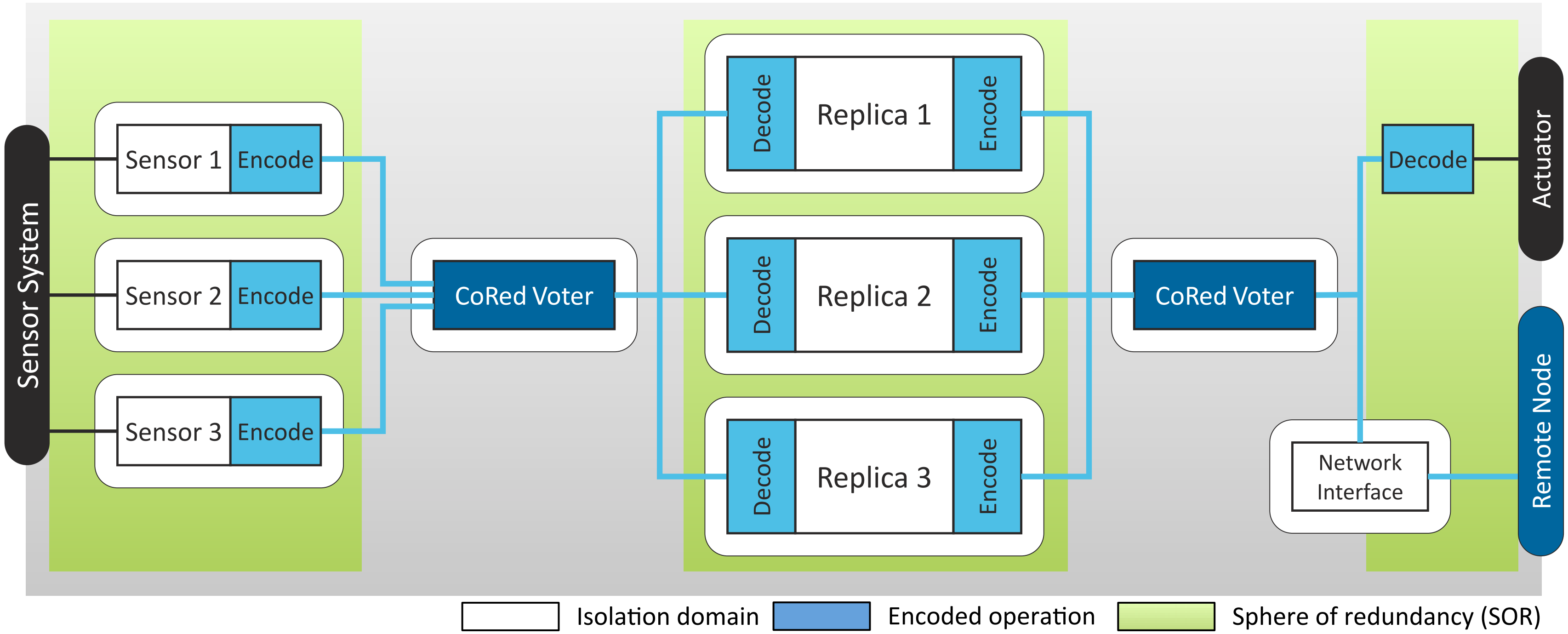


The Combined Redundancy Approach (***CoRed***) [1]

(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.

Combined Redundancy Approach

CoRed



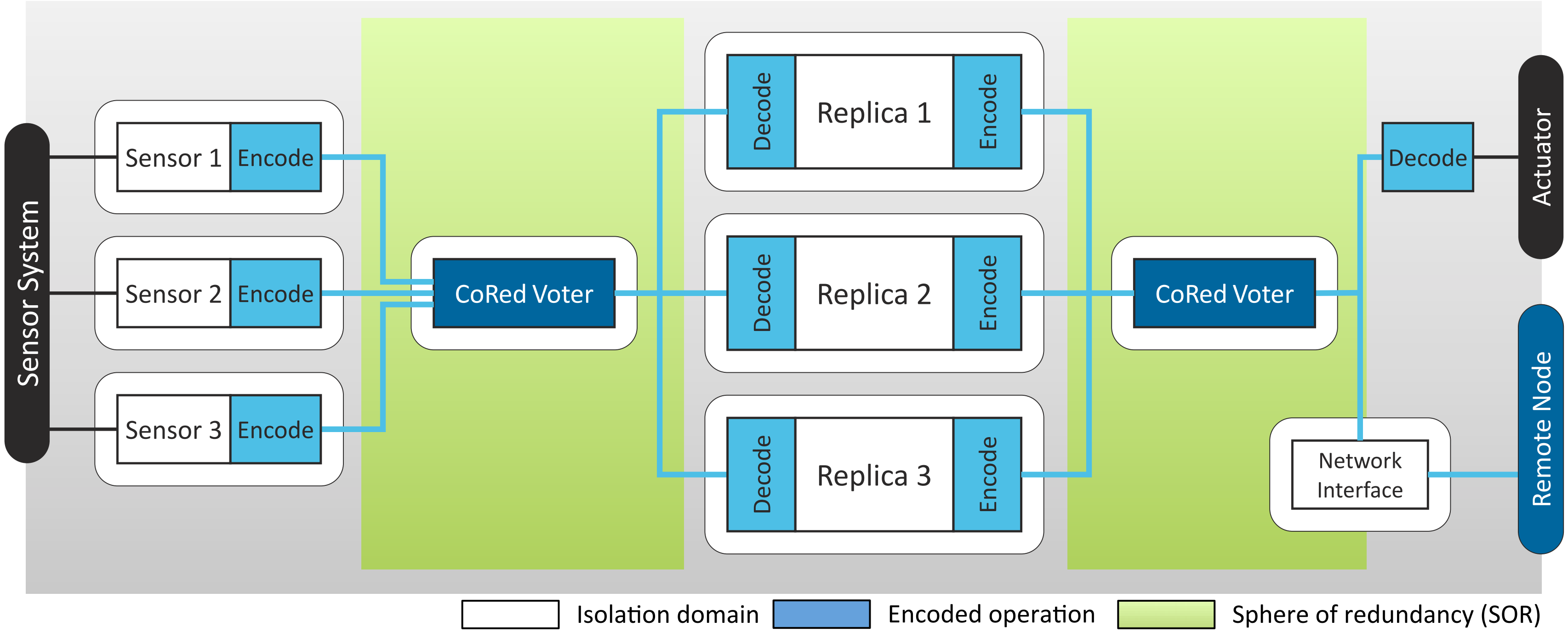
The Combined Redundancy Approach (***CoRed***) [1]

Triple Modular Redundancy

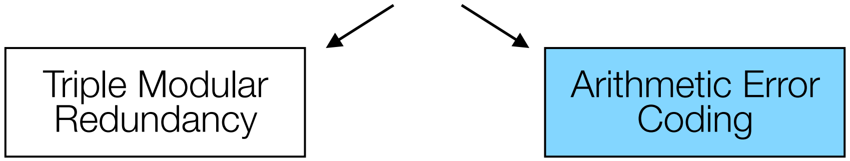
(1) Ulbrich, Peter; Hoffmann, Martin; Kapitza, Rüdiger; Lohmann, Daniel; Schmid, Reiner; Schröder-Preikschat, Wolfgang: "*Eliminating Single Points of Failure in Software-Based Redundancy*", EDCC 2012.

Combined Redundancy Approach

CoRed



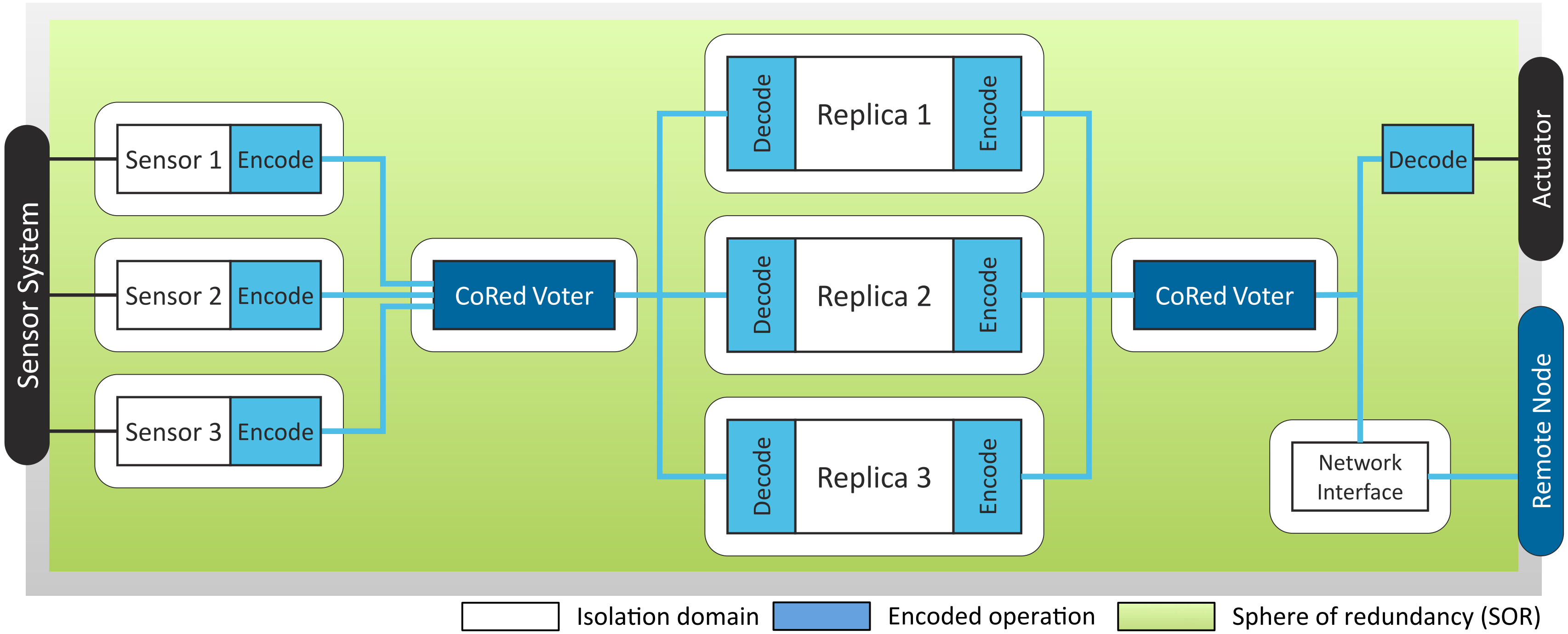
The Combined Redundancy Approach (***CoRed***) [1]



(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.

Combined Redundancy Approach

CoRed



The Combined Redundancy Approach (***CoRed***) [1]

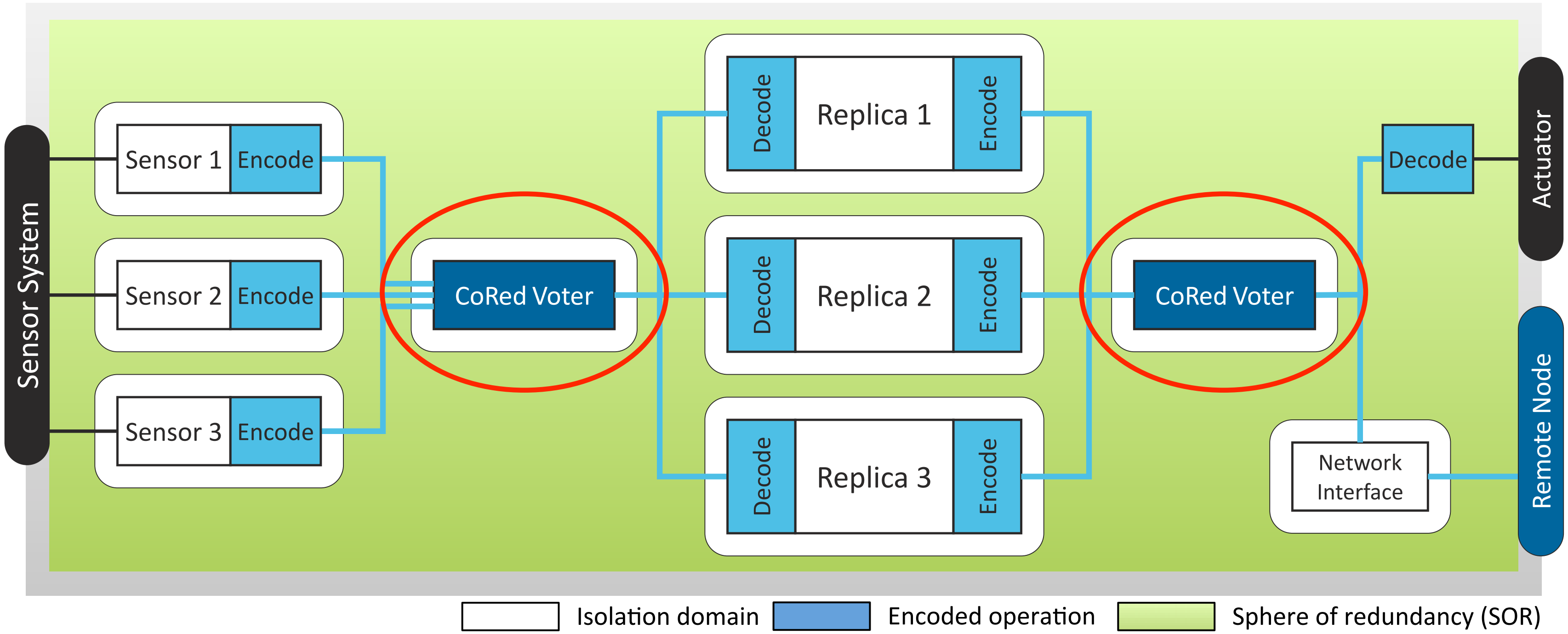
Triple Modular Redundancy

Arithmetic Error Coding

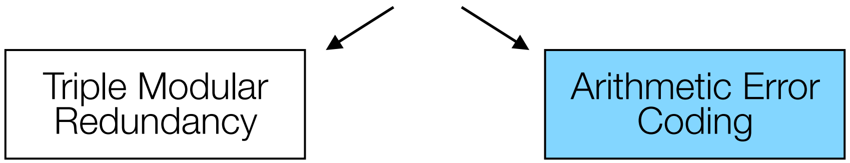
(1) Ulbrich, Peter; Hoffmann, Martin; Kapitza, Rüdiger; Lohmann, Daniel; Schmid, Reiner; Schröder-Preikschat, Wolfgang: "*Eliminating Single Points of Failure in Software-Based Redundancy*", EDCC 2012.

Combined Redundancy Approach

CoRed



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

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?

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:

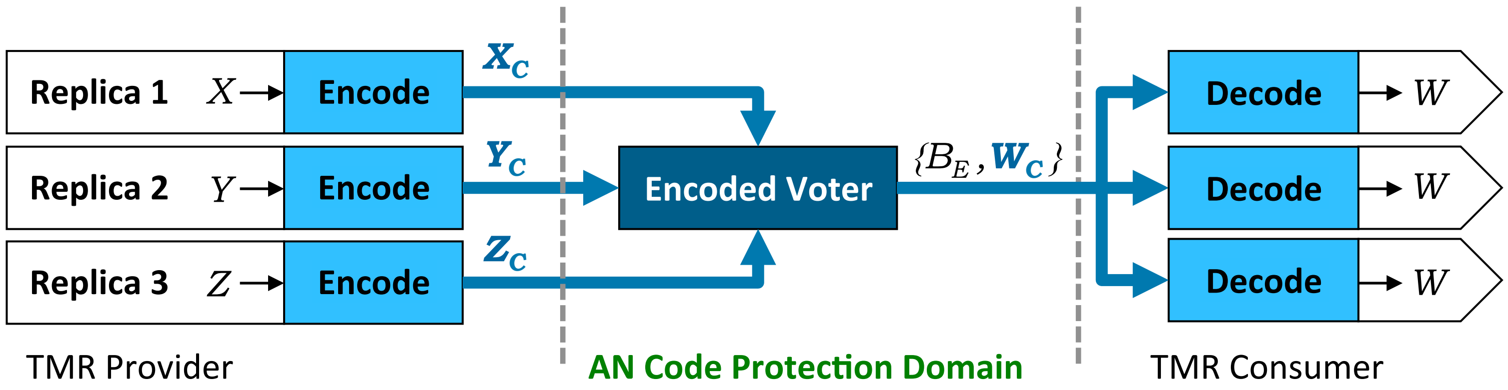
- Experiments showed discrepancies (in line with [3])
- Implications on error probability?

→Practitioners cannot blindly rely on coding theory!

Agenda

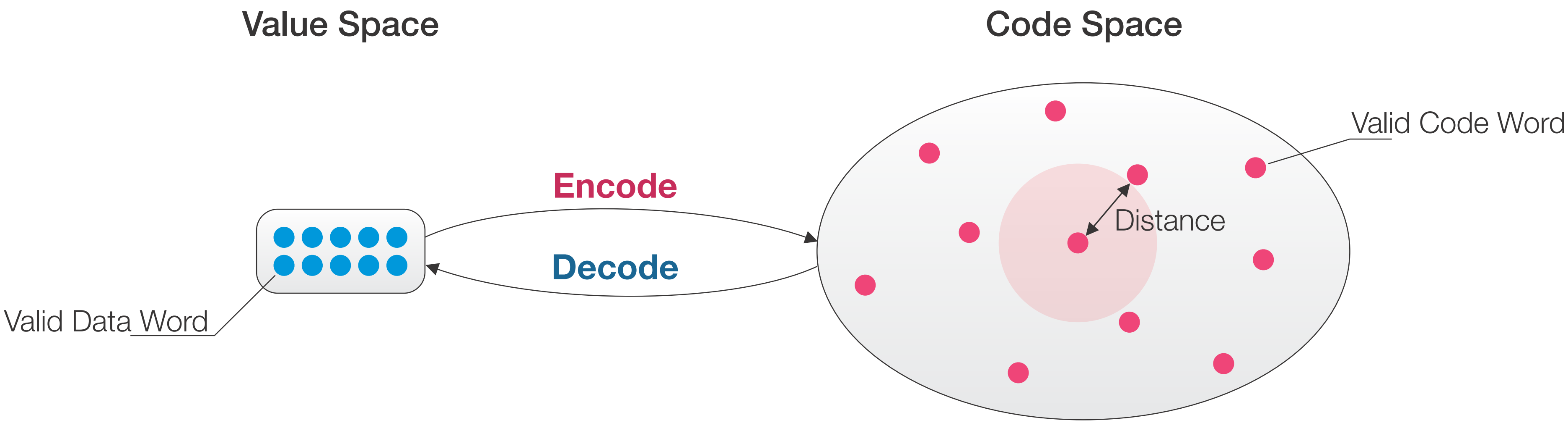
- Introduction
- **Background**
  - The *CoRed* Dependable Voter
  - Arithmetic Error Coding
- **Think Binary**
  - Choosing Appropriate Keys
  - Pitfall 1: Mapping Code to Binary
- **Know Your Compiler & Architecture**
  - Pitfall 2: Inter-Instruction State
  - Pitfall 3: Undefined Execution Environment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

The *CoRed* Dependable Voter – Basics



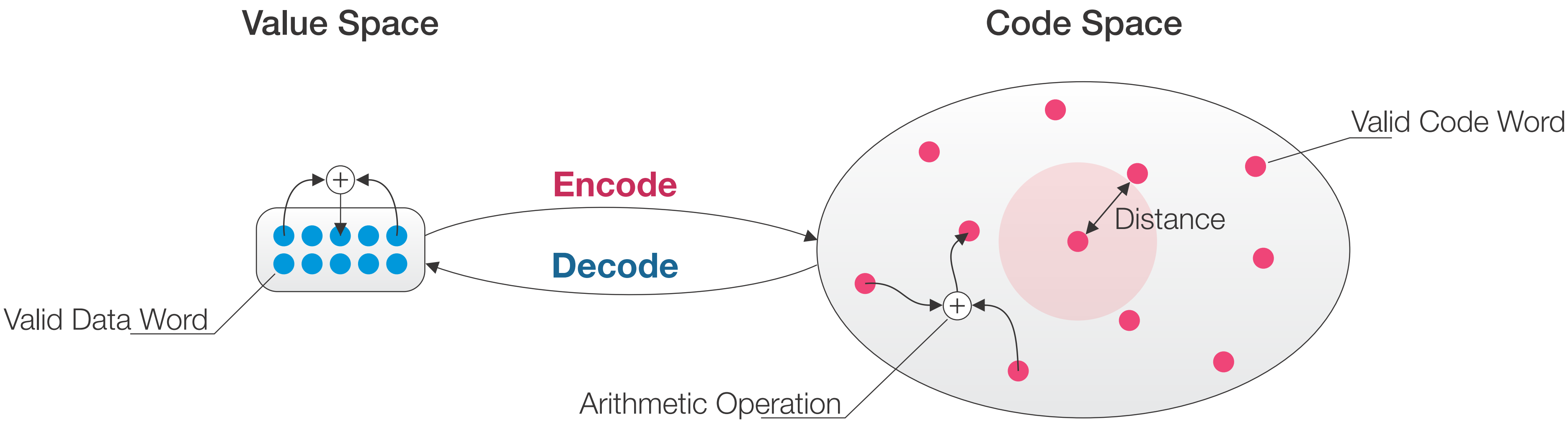
- **Complex encoded comparison operation**
- **Data-flow integrity**
  - Input: Variants (X_C, Y_C, Z_C)
  - Output: Constant signature (B_E) and encoded winner (W_C)
  - Validation: Subsequent check (decode)
- **Control-flow integrity**
  - Static signature (expected value): Compile-time
    → Used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

Arithmetic Error Coding – Basics



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

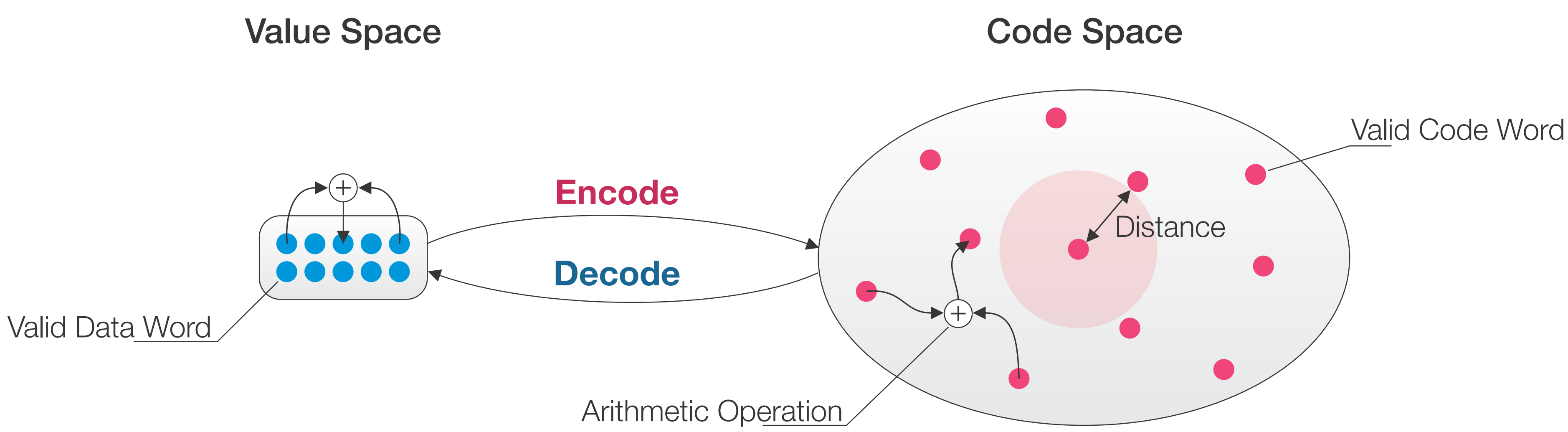
Arithmetic Error Coding – Basics



- **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 (+, -, ×, =, …)

$$\underbrace{v_c}_{\text{Encoded value}} = \boxed{\underbrace{A}_{\text{Constant (Key)}} \bullet \underbrace{v}_{\text{Value}}}$$

Arithmetic Error Coding – Basics

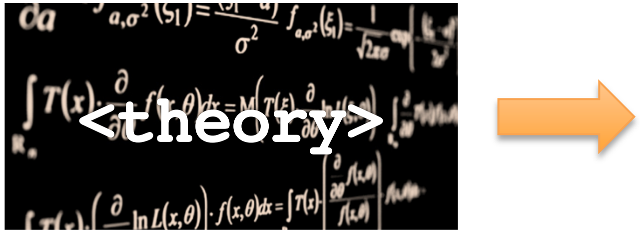


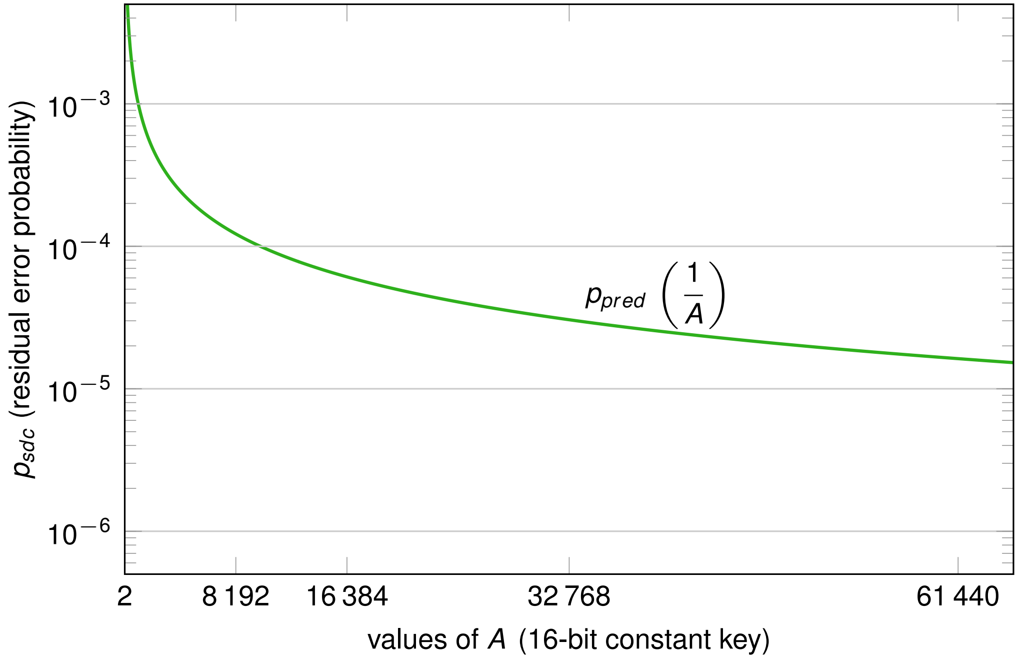
- **General coding theory**
  - Data word + redundant information = code word
  - Fault detection → distance between code words
- **Arithmetic error codes**
  - Can cope with computational flaws
  - Arithmetic operators (+, -, ×, =, …)

$$v_c = \boxed{A \bullet v} + \boxed{B_v + D}$$

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

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

$$p_{sdc} = \frac{\text{valid code words}}{\text{possible code words}} \approx \frac{1}{A}$$

→ The bigger key *A*, the better?

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

x	1	0	1	0
y	1	**1**	**0**	0

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

x	1	0	1	0
y	1	**1**	**0**	0

d = 2

- **Brute force**
  - 1.4×10^{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

x	1	0	1	0
y	1	**1**	**0**	0

d = 2

- **Brute force**
  - 1.4×10^{14} experiments for all 16 bit As

    A = 58,368 d_{min} = 2 #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

x	1	0	1	0
y	1	**1**	**0**	0

d = 2

- **Brute force**
  - 1.4×10^{14} experiments for all 16 bit As

A = 58,368	d_{min} = 2	#errors detectable = 1
58,831	3	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

x	1	0	1	0
y	1	**1**	**0**	0

d = 2

- **Brute force**
  - 1.4×10^{14} experiments for all 16 bit As

A = 58,368	d_{min} = 2	#errors detectable = 1
58,831	3	2
58,659	**6**	**5**

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

x	1	0	1	0
y	1	**1**	**0**	0

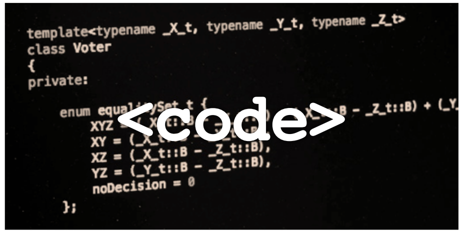
d = 2

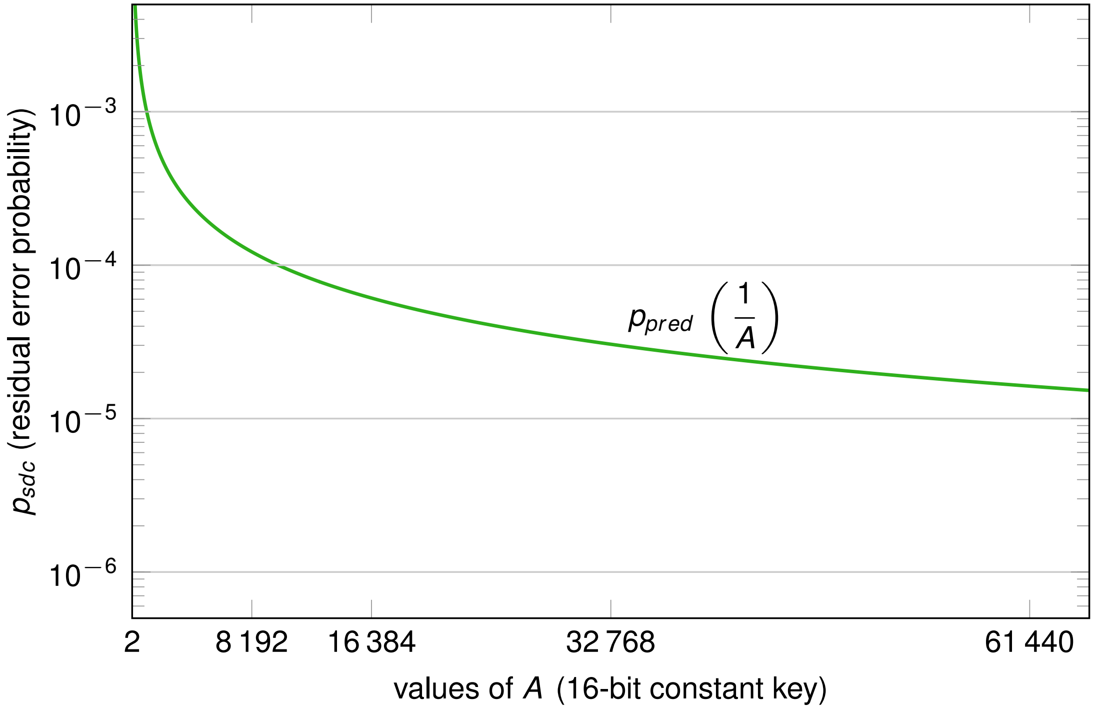
- **Brute force**
  - 1.4×10^{14} experiments for all 16 bit As

A =	d_{min} =	#errors detectable =
58,368	2	1
58,831	3	2
58,659	**6**	**5**

→The bigger the better is misleading!

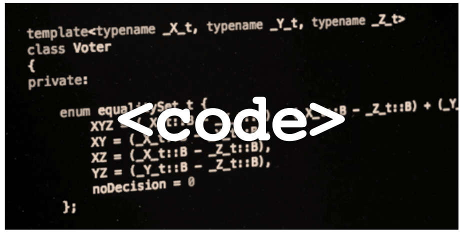
Double Check – Implementation in the Spotlight

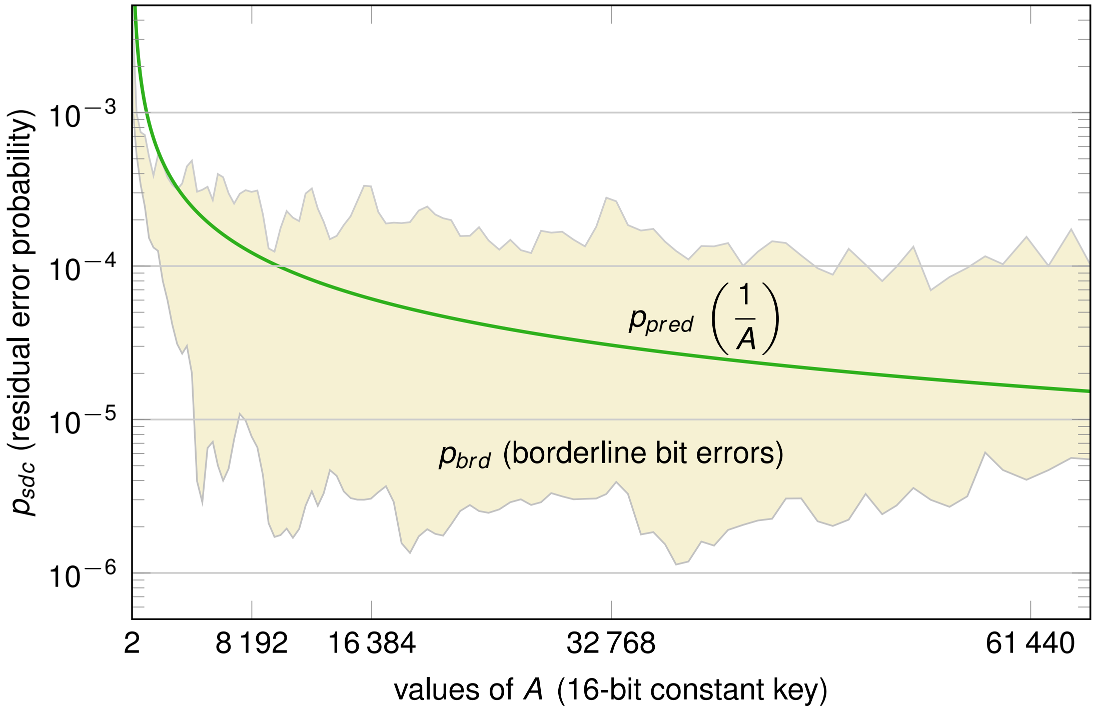




- **Fault-simulation → entire fault-space**
  - Each and every A, v and fault pattern
  - 6.5×10^{16} experiments for 16 bit As and 1-8 bit soft errors

Double Check – Implementation in the Spotlight

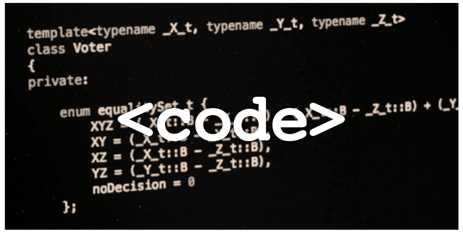


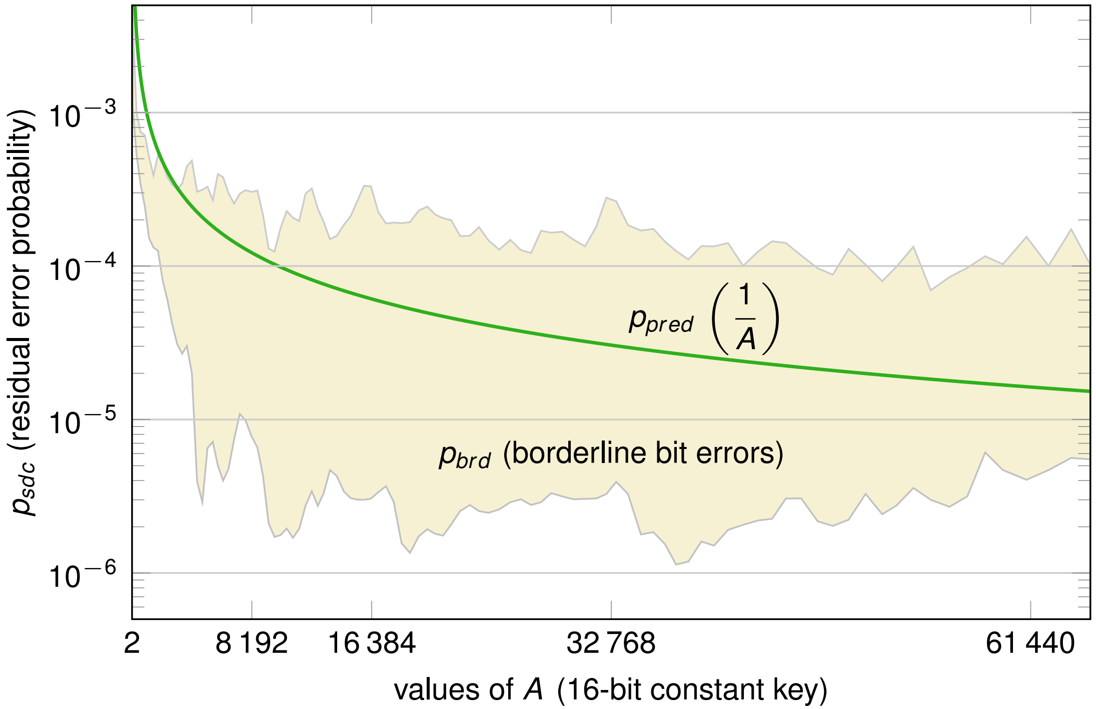


- **Fault-simulation → entire fault-space**
  - Each and every A, v and fault pattern
  - 6.5×10^{16} experiments for 16 bit As and 1-8 bit soft errors

→ Excess of predicted residual error probability

Double Check – Implementation in the Spotlight



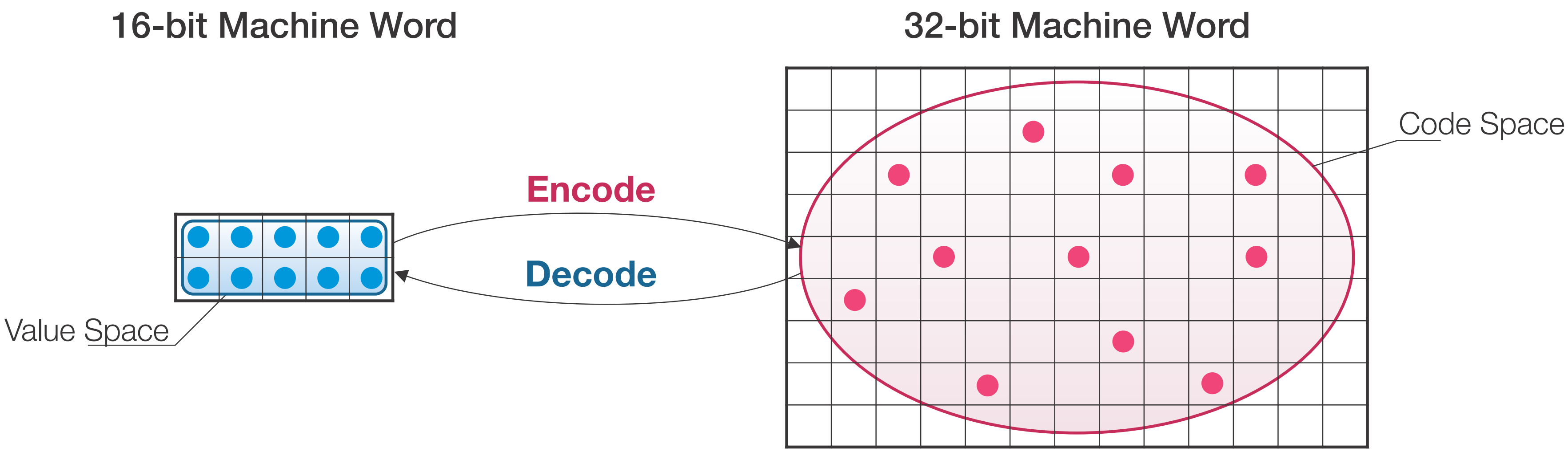


- **Fault-simulation → entire fault-space**
  - Each and every A, v and fault pattern
  - $6.5{\times}10^{16}$ experiments for 16 bit As and 1-8 bit soft errors

→ Excess of predicted residual error probability

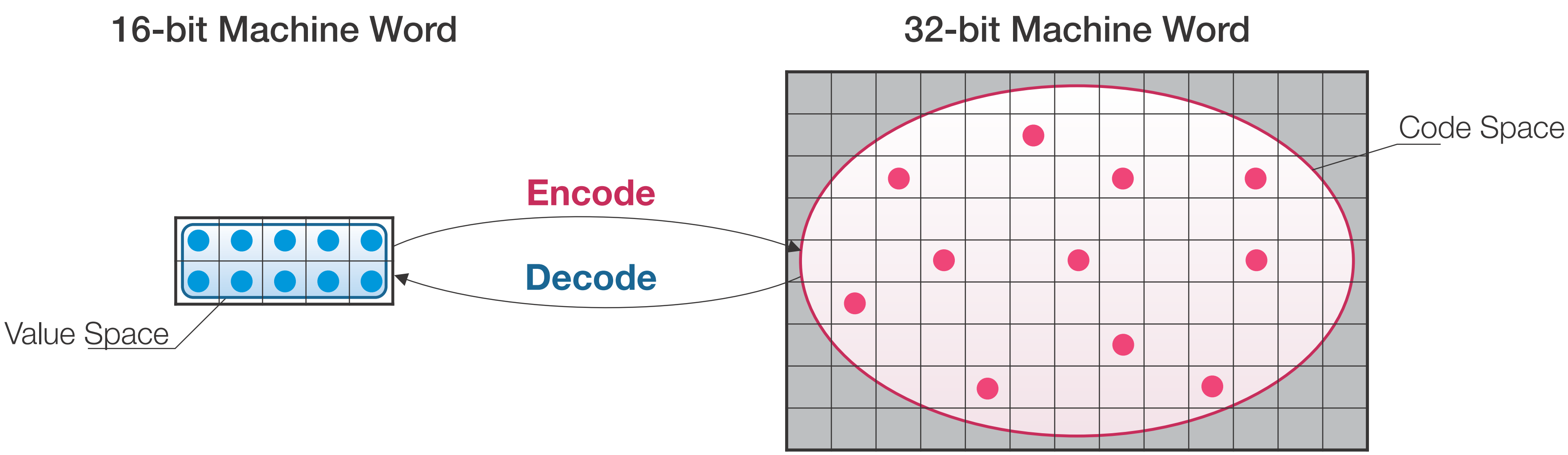
→ Mismatch with Hamming distance experiments

Pitfall 1: Mapping Code to Binary



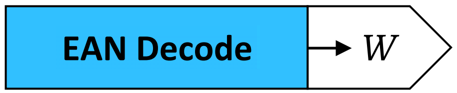
- **Pitfall 1: Binary representation of code words**
  - Coding theory is unaware of machine word sizes
  - → Dangerous over- and underflow conditions

Pitfall 1: Mapping Code to Binary



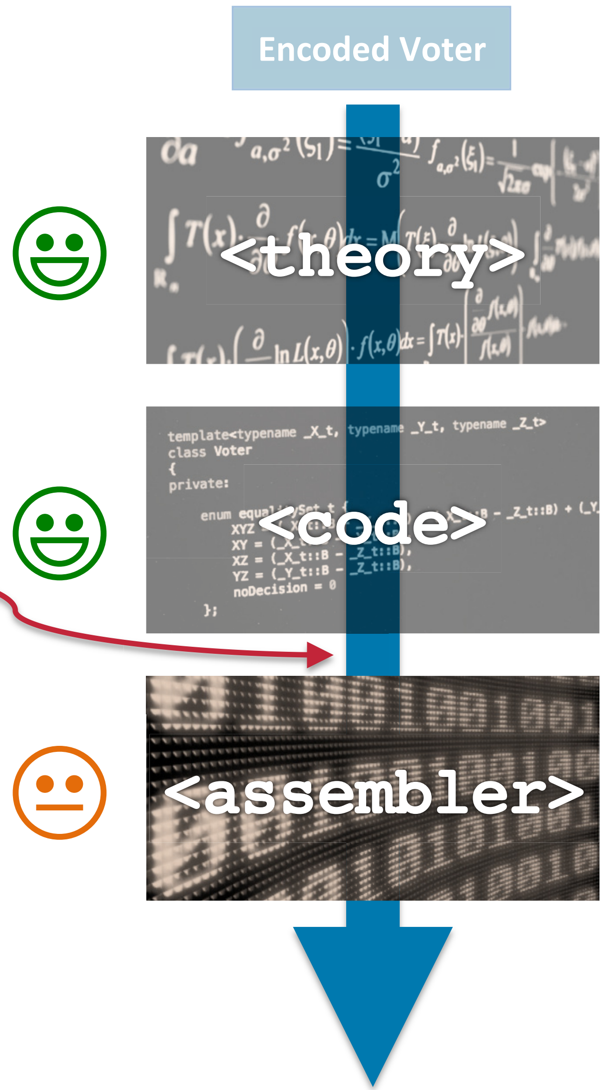
- Pitfall 1: **Binary representation of code words**
  - Coding theory is unaware of machine word sizes
  - → Dangerous over- and underflow conditions

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



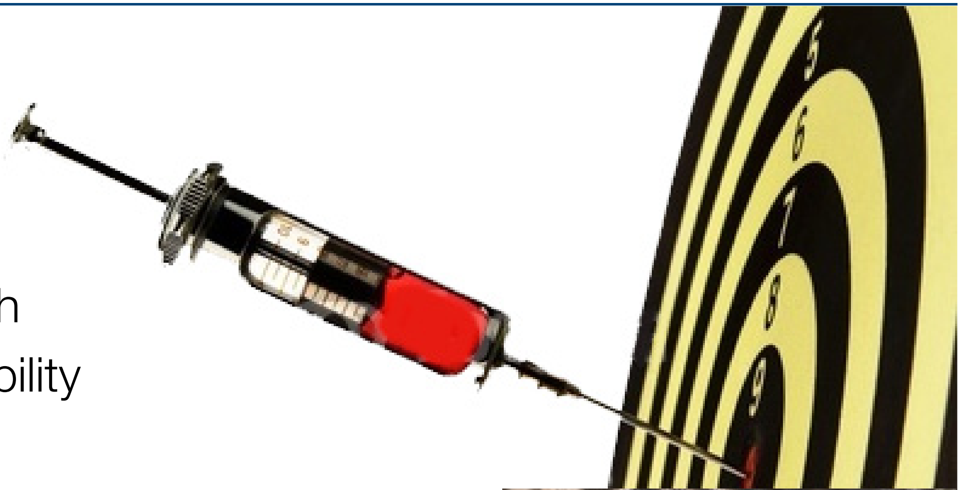
Agenda

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

	CoRed Voter	Simple Voter
Instructions	92	38
Memory (Bytes)	301	112

  - 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:

	CoRed Voter	Simple Voter
Instructions	92	38
Memory (Bytes)	301	112

  - 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

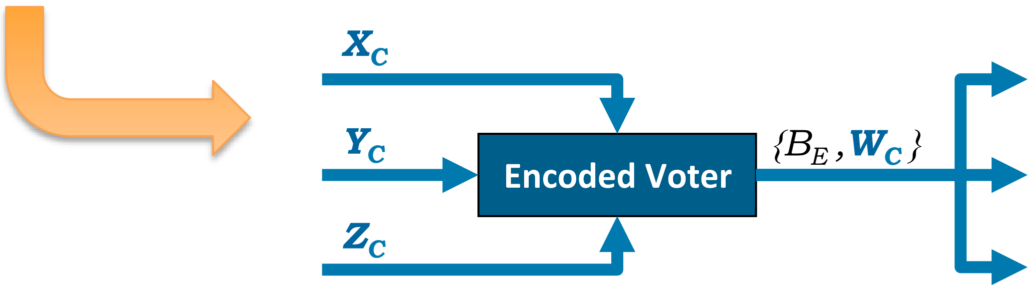
```
/* if (a == b) */
cmp eax, ebx
je Lequal
```

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

```
/* if (a == b) */
cmp eax, ebx
je Lequal
```

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

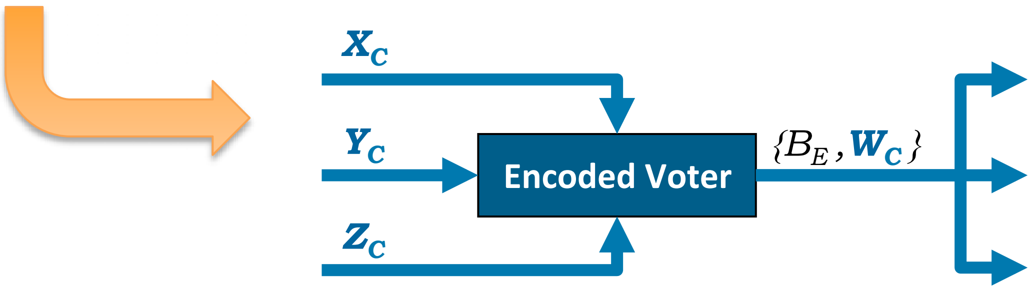


Know your Compiler and Architecture

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

```
/* if (a == b) */
cmp eax, ebx
je Lequal
```

- **EAN Patch:** `apply(`v_C`,`sig_{DYN}`)`
  - Malicious control-flow → Signature overflow → Additional check



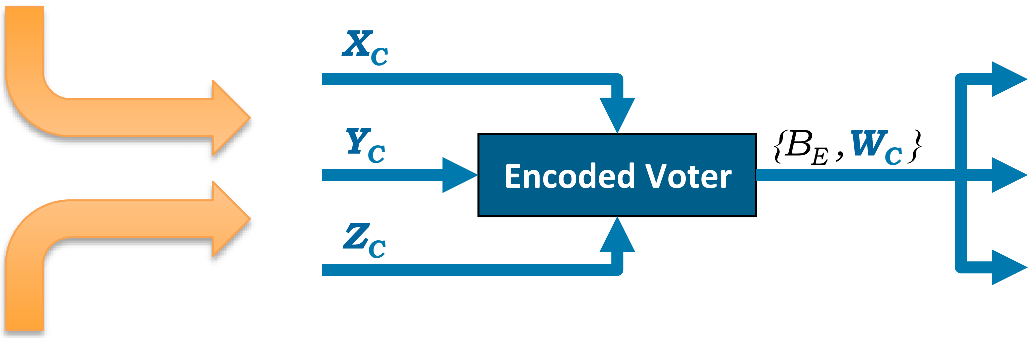
- Pitfall 3: **Undefined Execution Environment**
  - Compiler laziness leaves encoded values in registers
  - Zombie values → leaking from caller to voter function
  - → Isolation assumptions violated

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

```
/* if (a == b) */
cmp eax, ebx
je Lequal
```

- **EAN Patch:** `apply(`v_C`,`sig_{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

Fault-Injection Campaigns – Final Results

	Instructions		Registers and Flags		Program Counter	
	Simple	CoRed	Simple	CoRed	Simple	CoRed
OK	784	2772	1040	3204	127	267
Detected (Code)	-	995	-	1435	-	420
Detected (Trap)	93	246	8	41	21	241
Detected (Isolation)	825	1834	1825	3736	2804	6240
Detected (Timeout)	0	1	0	0	0	0
Undetected (SDC)	**450**	0	**807**	0	**152**	0

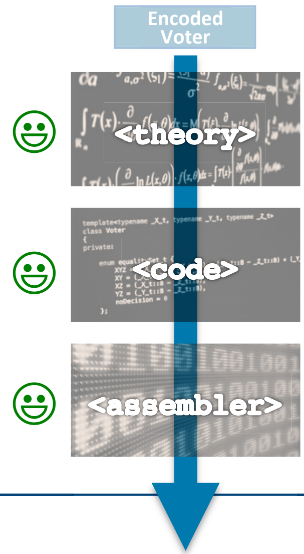
- **3 Fault-Injection Campaigns:**
  - Instructions and
  - General purpose registers and CPU flags
  - Program counter

Fault-Injection Campaigns – Final Results

	Instructions		Registers and Flags		Program Counter	
	Simple	CoRed	Simple	CoRed	Simple	CoRed
OK	784	2772	1040	3204	127	267
Detected (Code)	-	995	-	1435	-	420
Detected (Trap)	93	246	8	41	21	241
Detected (Isolation)	825	1834	1825	3736	2804	6240
Detected (Timeout)	0	1	0	0	0	0
Undetected (SDC)	**450**	0	**807**	0	**152**	0

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

→ **CoRed dependable voter performs as EXPECTED!**



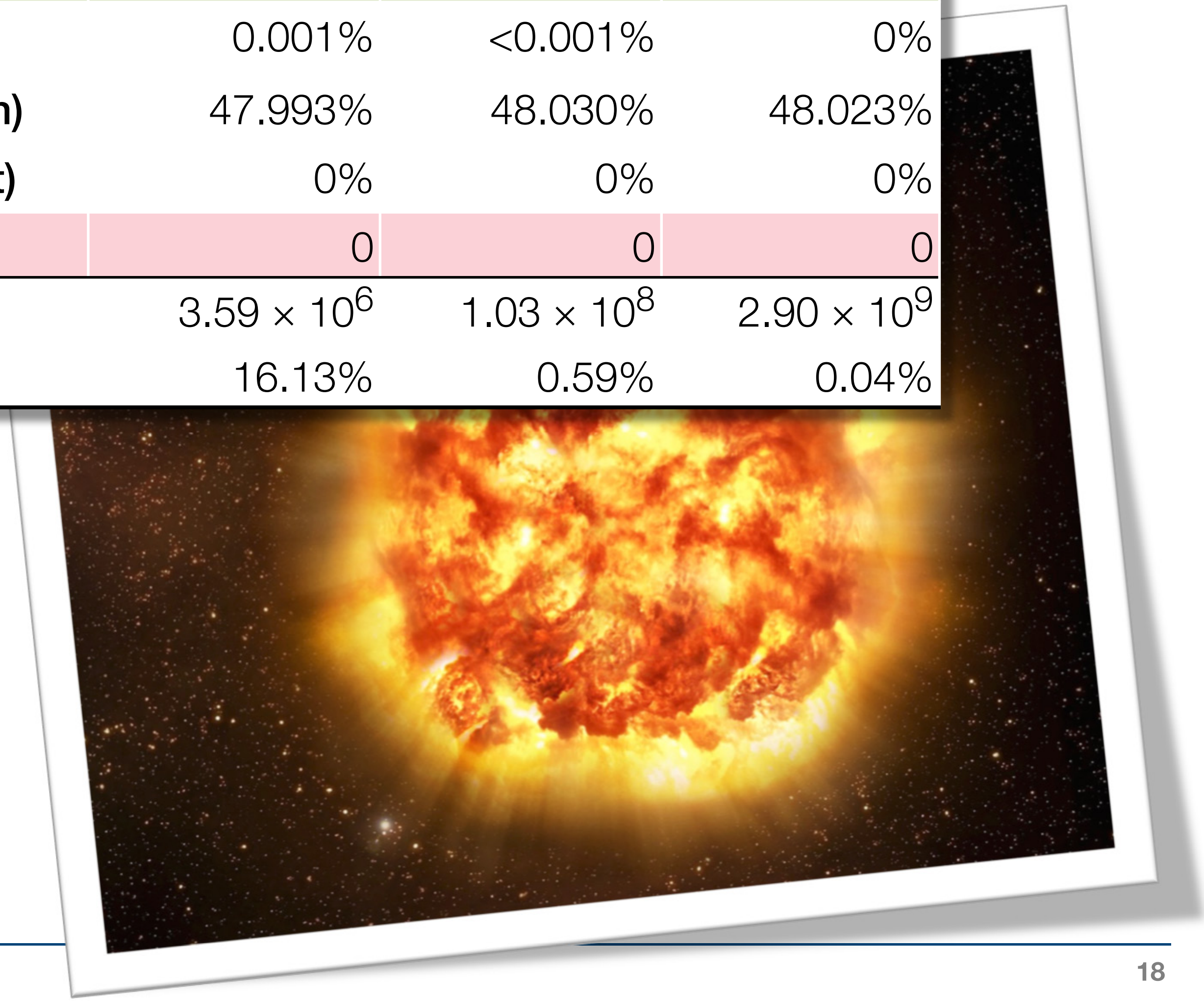
Multi-Bit Faults – The Good, the Bad and the …

	Good A = 58,659	Bad A = 58,368
OK	38639	38639
Detected (Code)	21596	21519
Detected (Trap)	47	47
Detected (Isolation)	60438	60438
Detected (Timeout)	0	0
Undetected	0	**77**

- **2-bit Fault-injection experiments**
  - Full fault space coverage
  - Triple check fault-detection capabilities
- **Distances: d_{good} = 6, d_{bad} = 2**

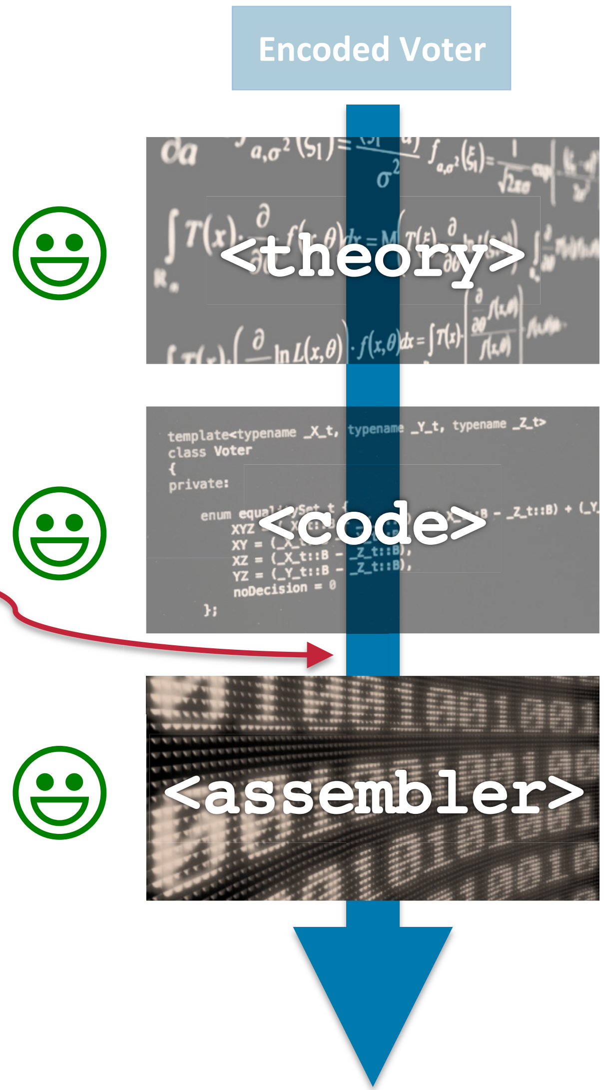
Multi-Bit Faults – Tighten the Rules

	3-bit faults	4-bit faults	5-bit faults
OK	33.742%	33.605%	33.544%
Detected (Code)	18.209%	18.356%	18.431%
Detected (Trap)	0.001%	<0.001%	0%
Detected (Isolation)	47.993%	48.030%	48.023%
Detected (Timeout)	0%	0%	0%
Undetected	0	0	0
Fault Space	3.59×10^6	1.03×10^8	2.90×10^9
Coverage	16.13%	0.59%	0.04%

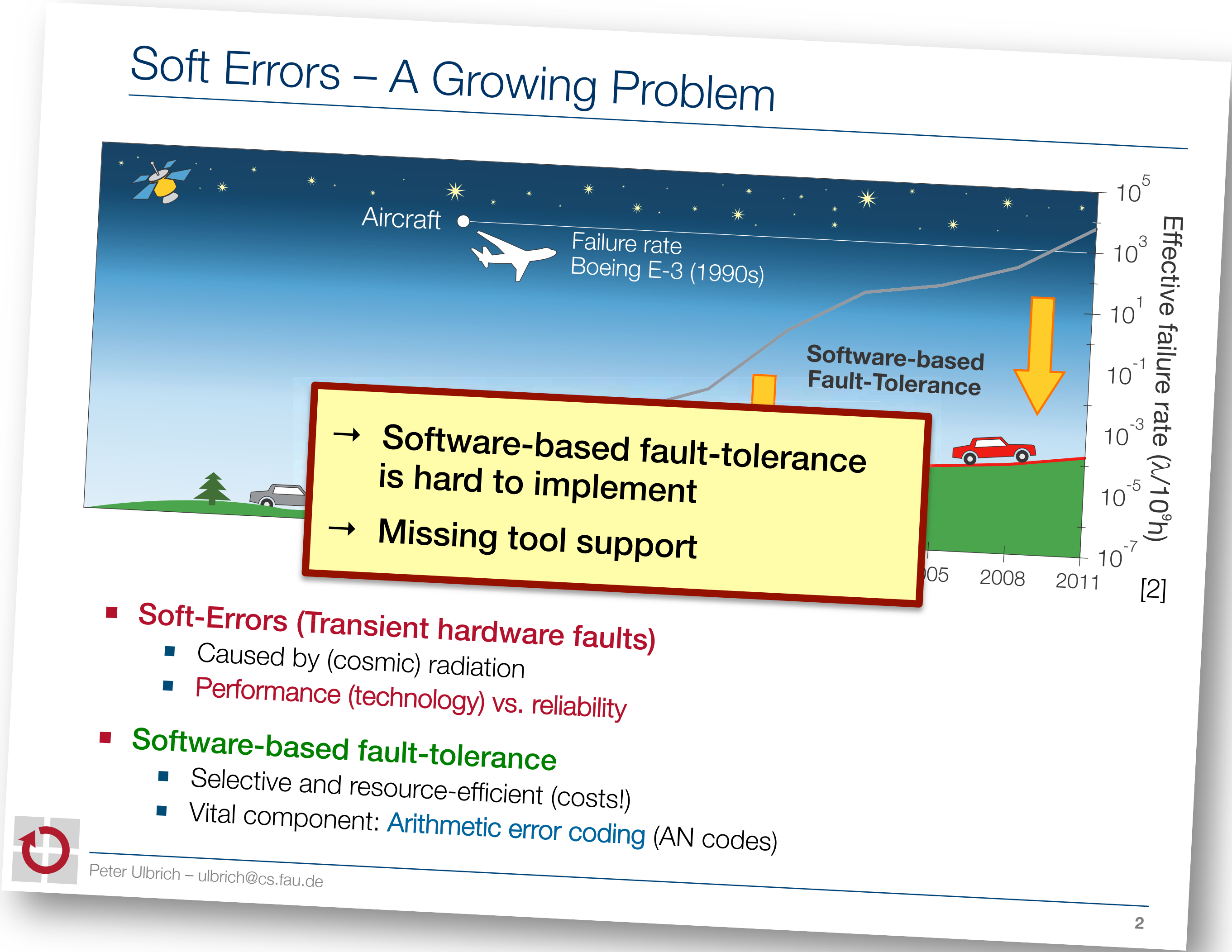


Agenda

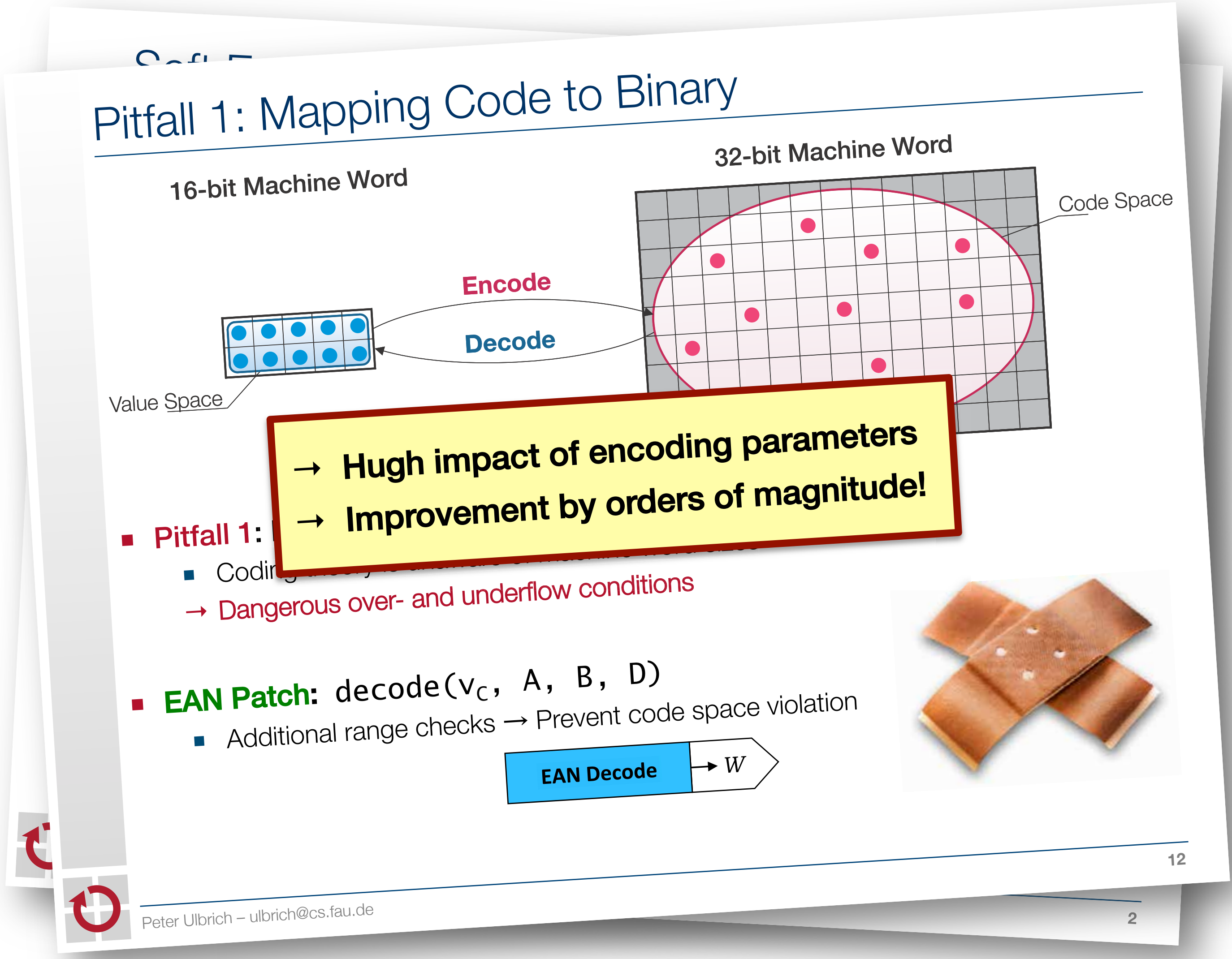
- Introduction
- Background
  - Arithmetic Error Coding
  - The *CoRed* Dependable Voter
- **Think Binary**
  - Choosing Appropriate Keys
  - Pitfall 1: Mapping Code to Binary
- **Know Your Compiler & Architecture**
  - Pitfall 2: Inter-Instruction State
  - Pitfall 3: Undefined Execution Environment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



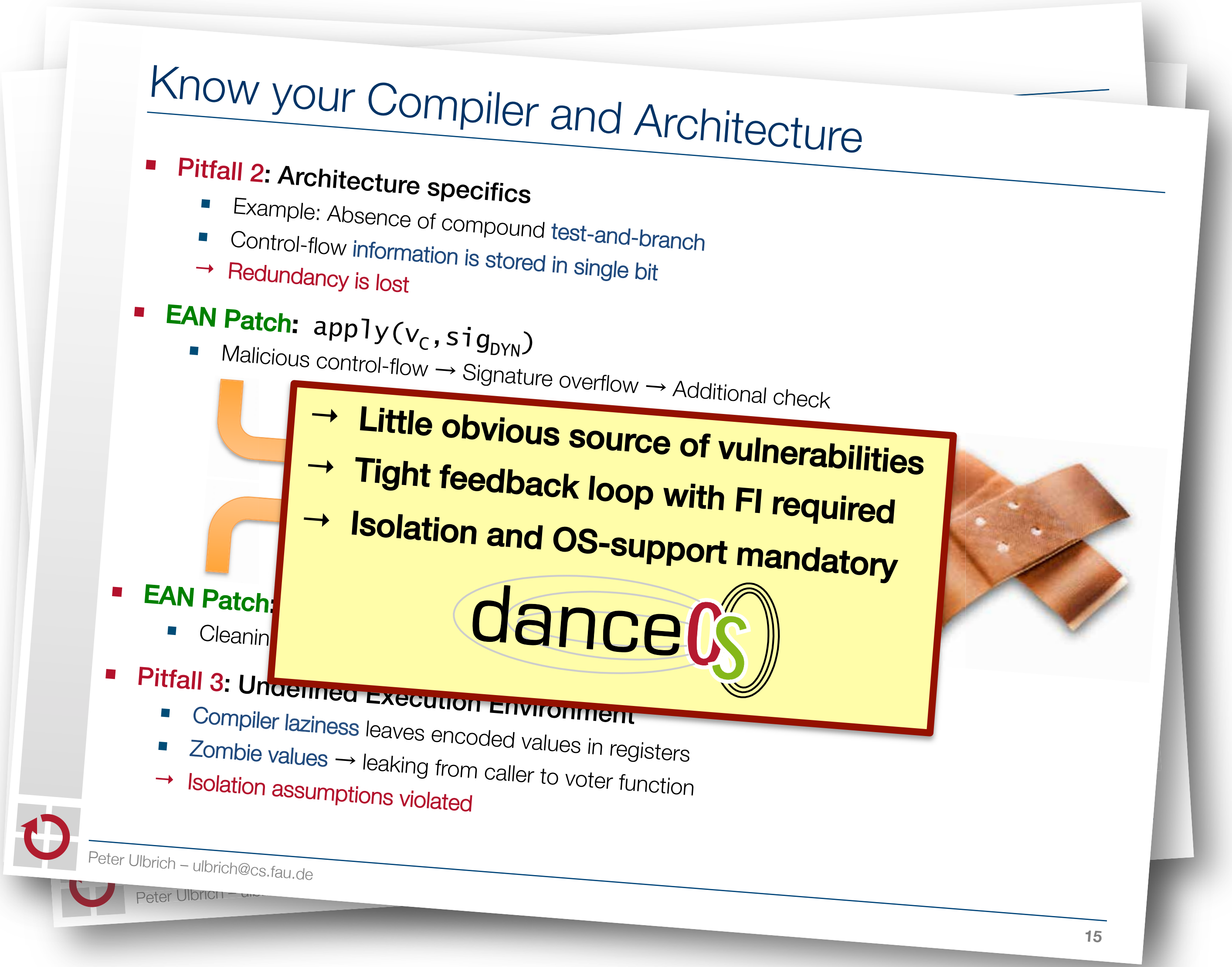
Conclusions & Lessons Learned



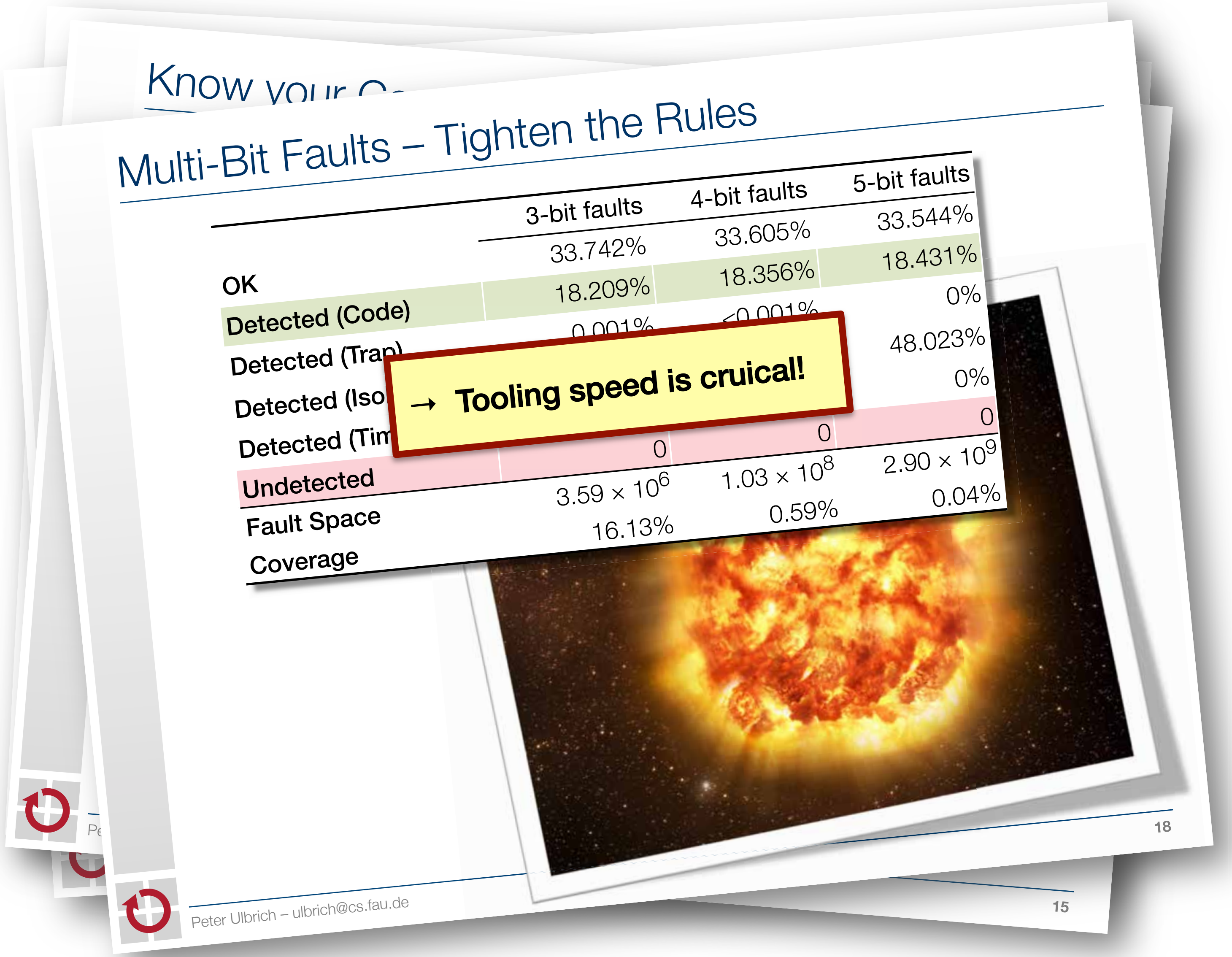
Conclusions & Lessons Learned



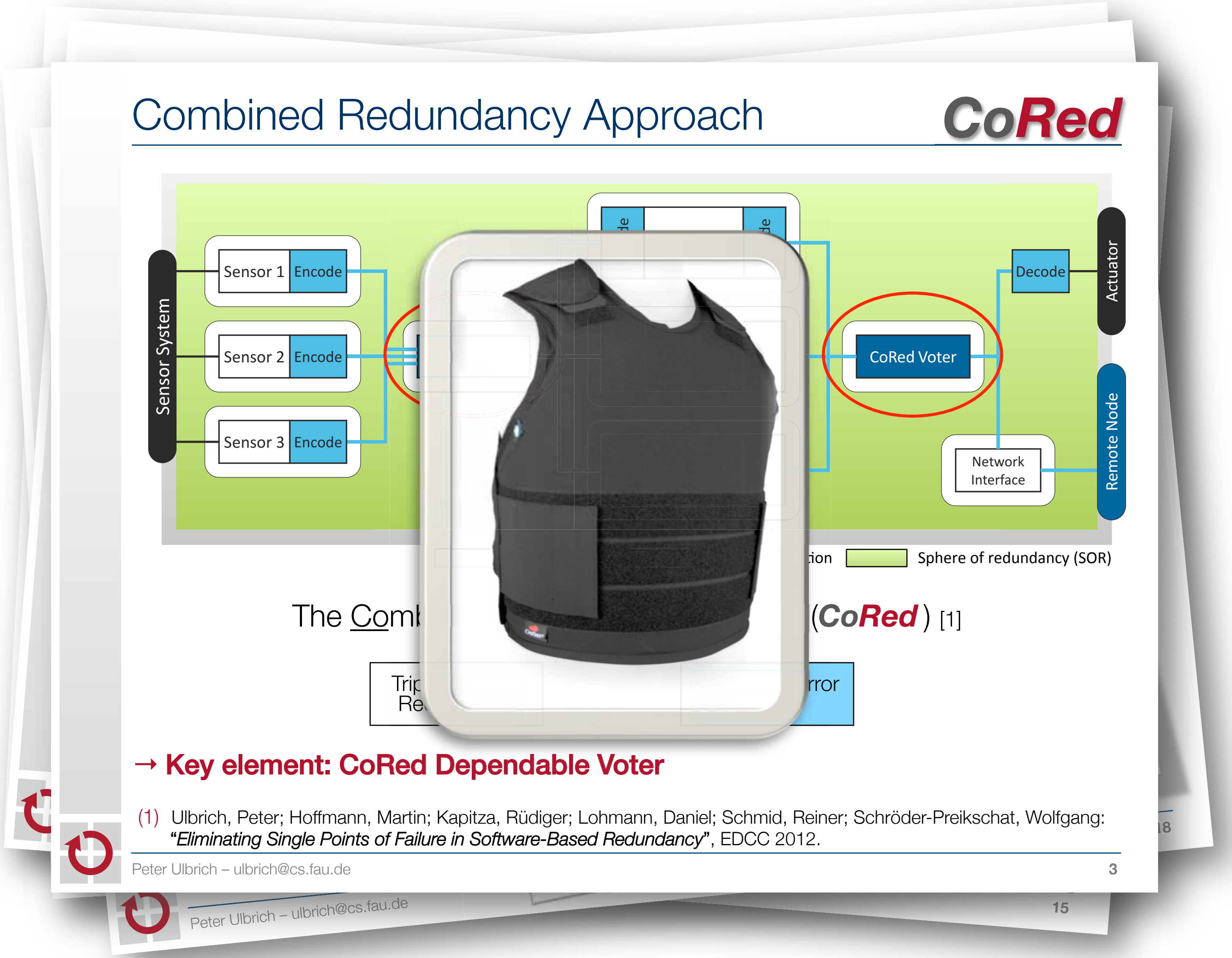
Conclusions & Lessons Learned



Conclusions & Lessons Learned



Conclusions & Lessons Learned



Implementation and further experimental results:

`http://www4.cs.fau.de/Research/CoRed`

Thank you!

?