Software-based Fault Tolerance – Mission (Im)possible?

Peter Ulbrich

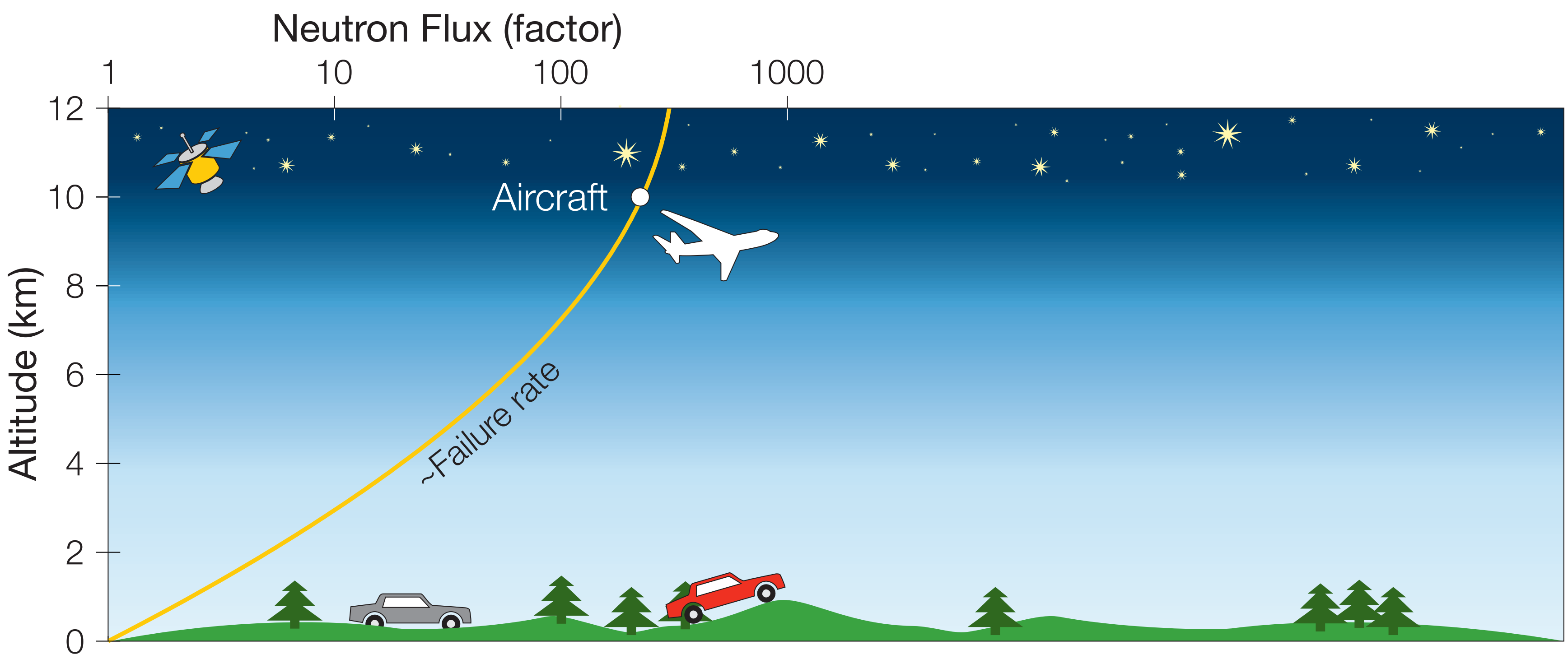
The 29th CREST Open Workshop on Software Redundancy
November 18, 2013

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

System Software Group

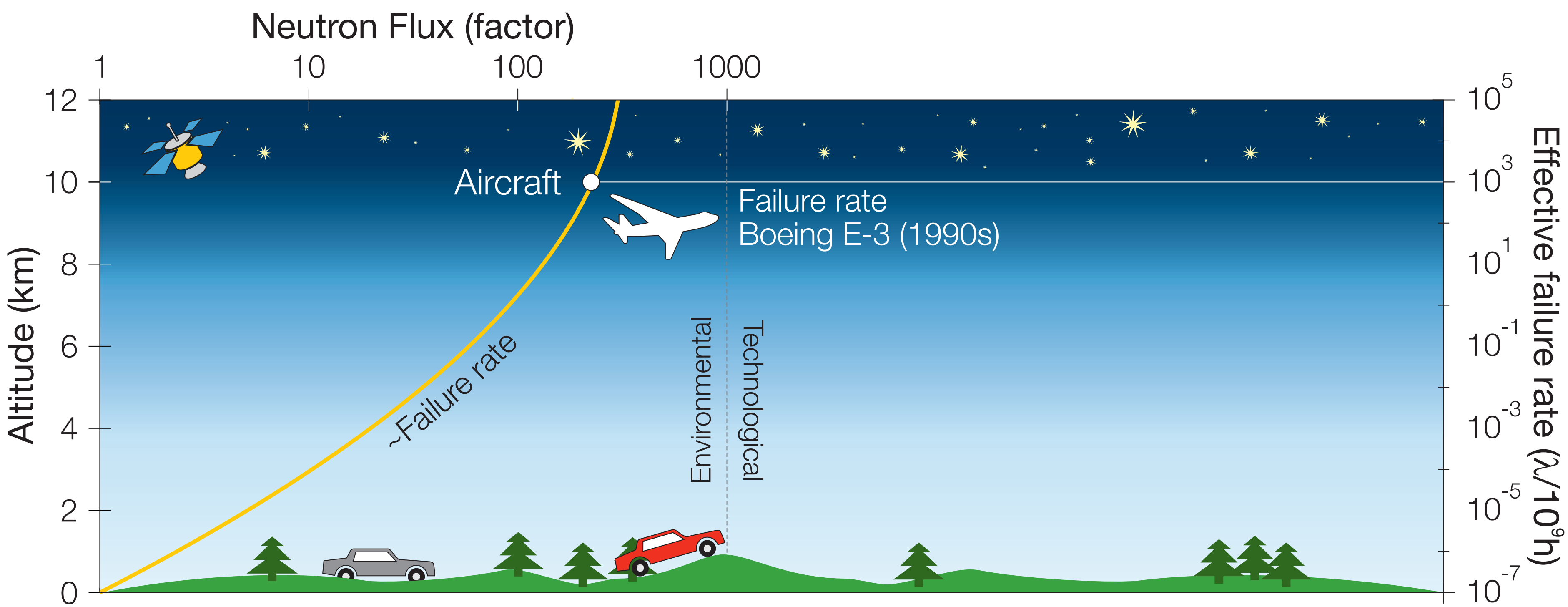
Embedded Systems Initiative

Soft Errors – A Growing Problem



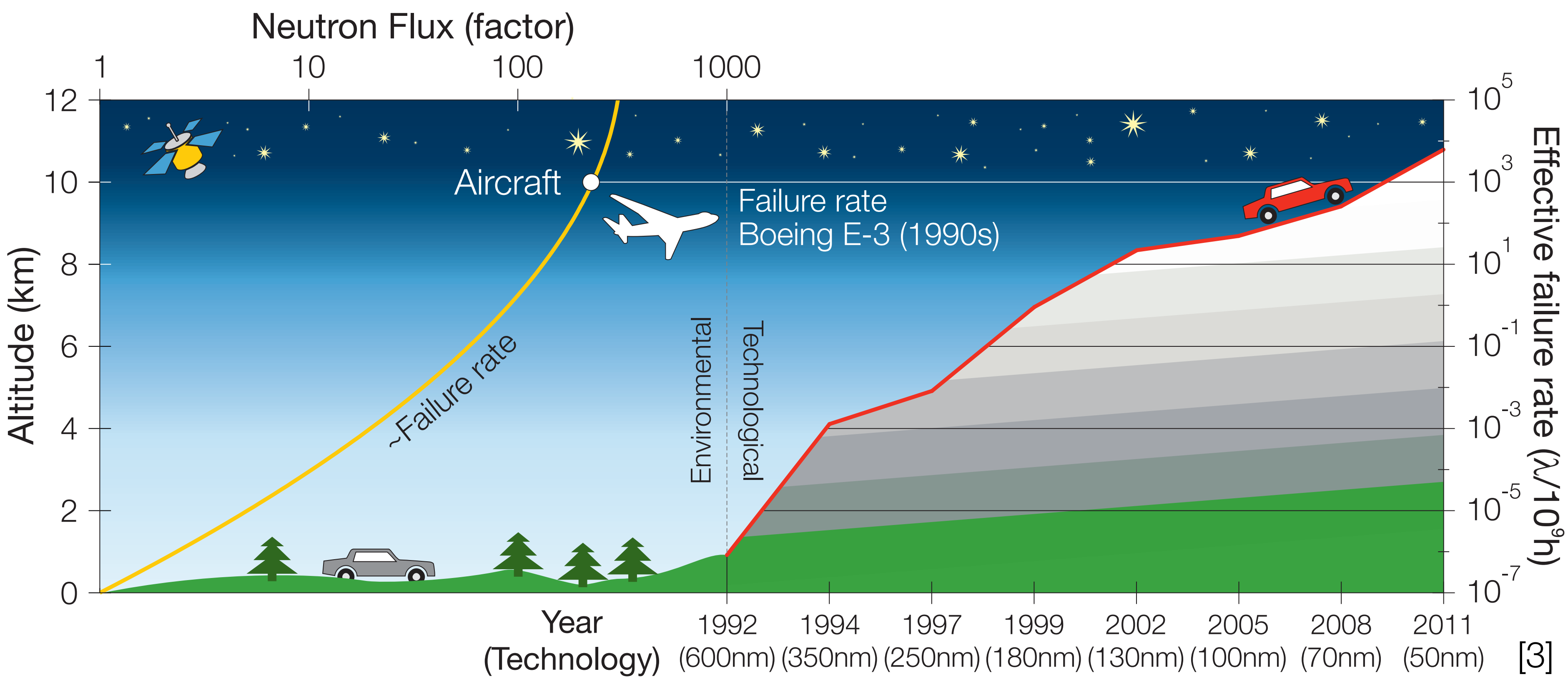
- **Soft-Errors (Transient hardware faults)**
  - Induced by e.g., radiation, glitches, insufficient signal integrity
  - Affecting microcontroller logic

Soft Errors – A Growing Problem



- **Soft-Errors (Transient hardware faults)**
  - Induced by e.g., radiation, glitches, insufficient signal integrity
  - Affecting microcontroller logic

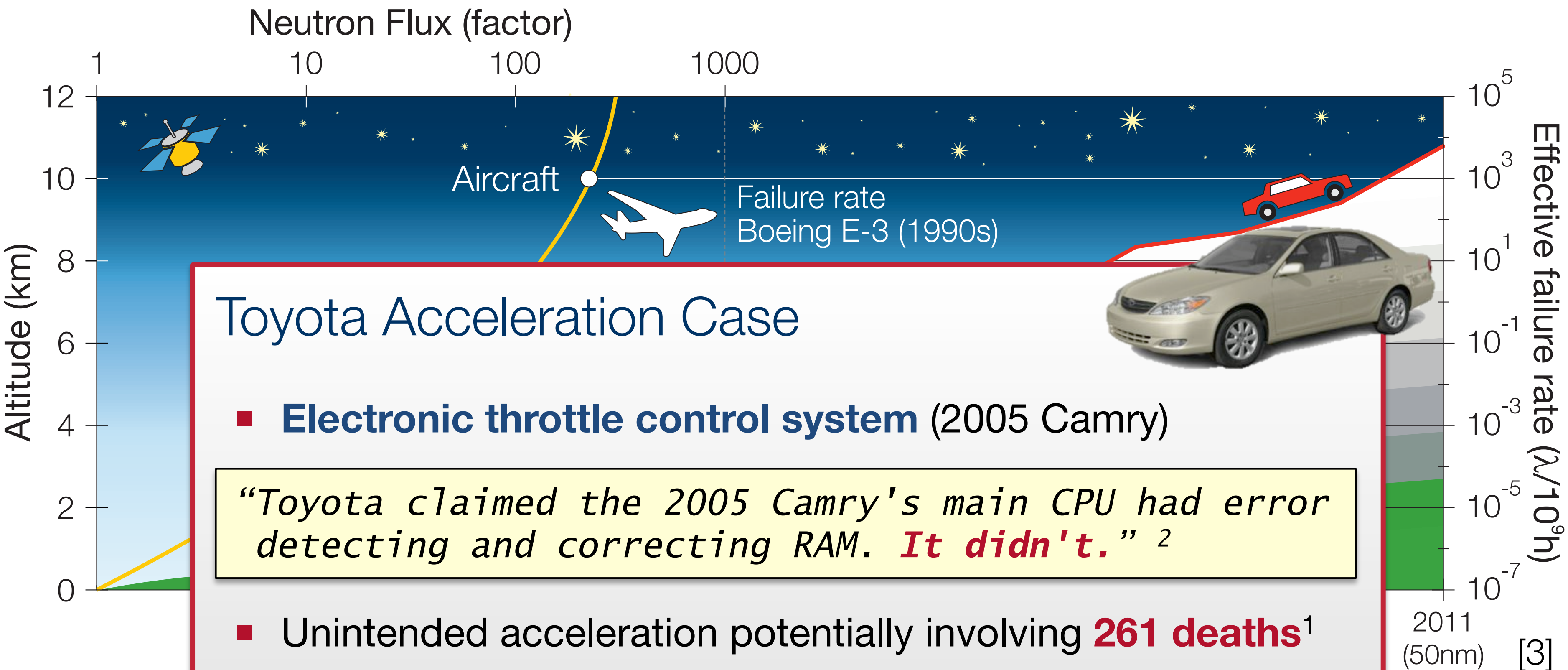
Soft Errors – A Growing Problem



[3]

- **Soft-Errors (Transient hardware faults)**
  - Induced by e.g., radiation, glitches, insufficient signal integrity
  - Affecting microcontroller logic
- **Future hardware designs: more performance and parallelism**
  → On the price of being less and less reliable

Soft Errors – A Growing Problem



Toyota Acceleration Case

- **Electronic throttle control system** (2005 Camry)

> *"Toyota claimed the 2005 Camry's main CPU had error detecting and correcting RAM. **It didn't.**"* [2]

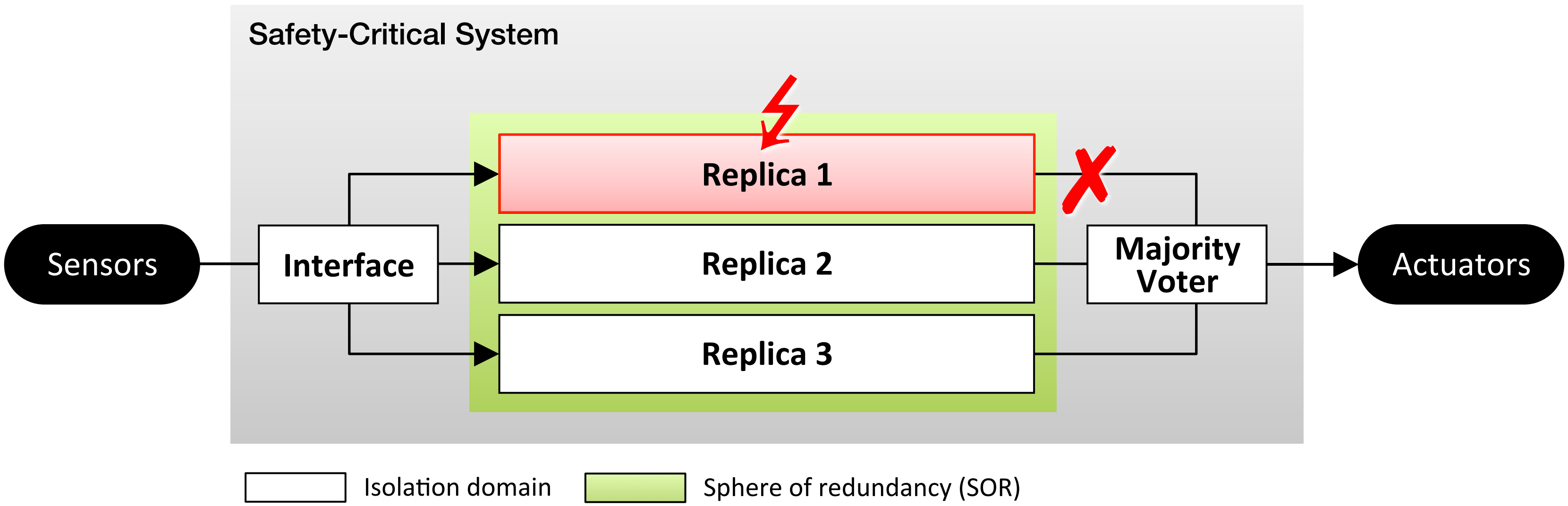
- Unintended acceleration potentially involving **261 deaths**[1]
- Experts identified **soft errors** as possible cause[1]

[1] US News, Mar 17, 2010 [2] Investigation Report, EDN Network, Oct 28, 2013

- So
  - Affecting microcontroller logic

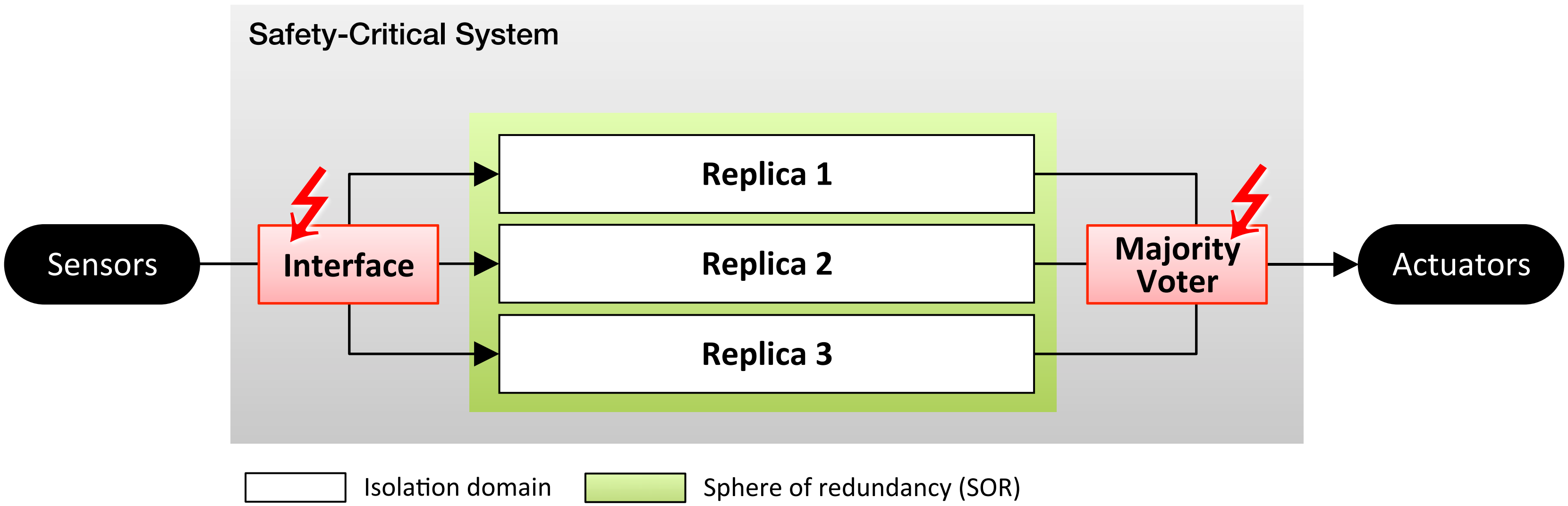
- **Future hardware designs: more performance and parallelism**
  → **On the price of being less and less reliable**

Software-Based Fault Tolerance



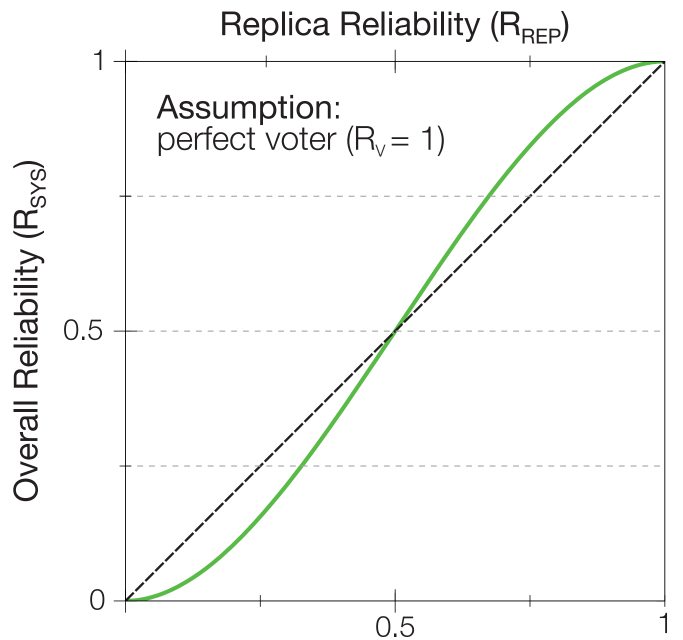
- **Software-based redundancy**
  - Triple Modular Redundancy (e.g., recommended by ISO 26262)
  - ✓ Selective and adaptive
  - ✓ Resource efficient

Software-Based Fault Tolerance



- **Software-based redundancy**
  - Triple Modular Redundancy (e.g., recommended by ISO 26262)
  - ✓ Selective and adaptive
  - ✓ Resource efficient
- **Single points of failure**
  - Interface and Majority Voter
  - Allowing for Silent Data Corruptions (SDC)
  - → Replication is impossible!

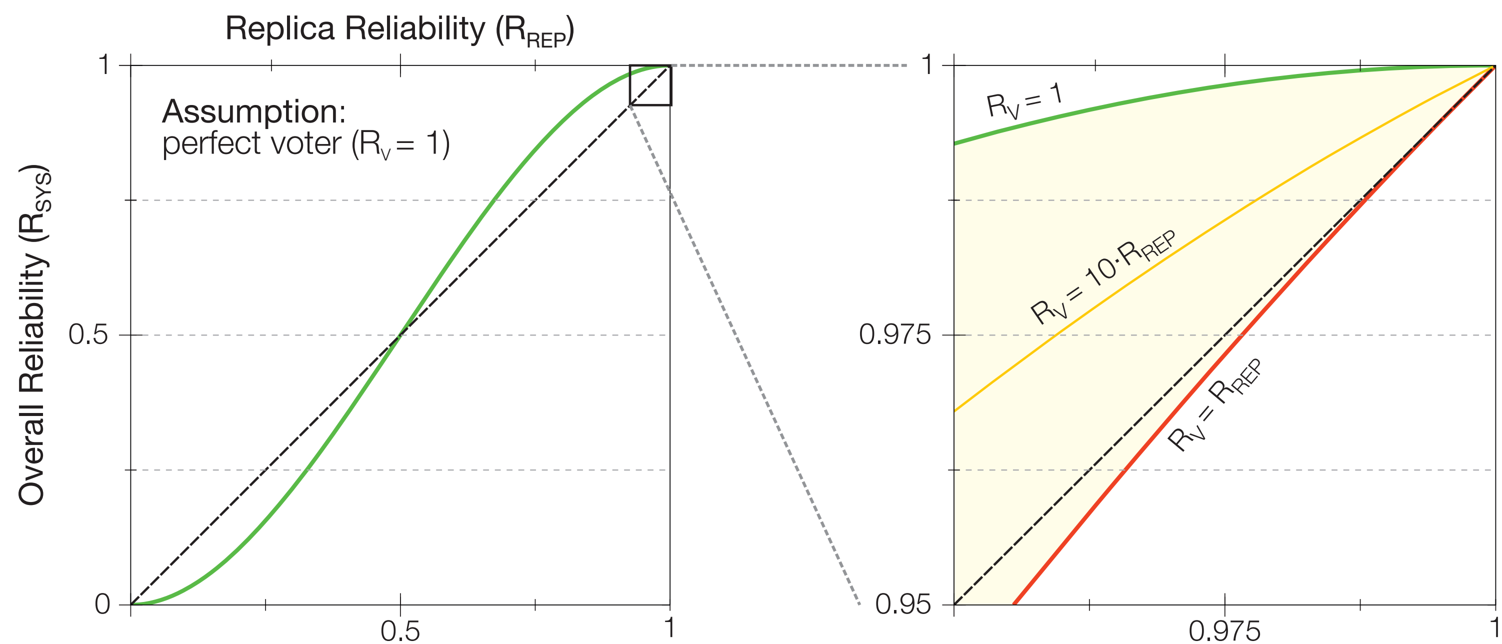
Threats to Applicability – Mission failed?



- **Triple modular redundancy reliability**

$$R_{TMR} = R_{Voter} \cdot R_{2-of-3}$$

Threats to Applicability – Mission failed?

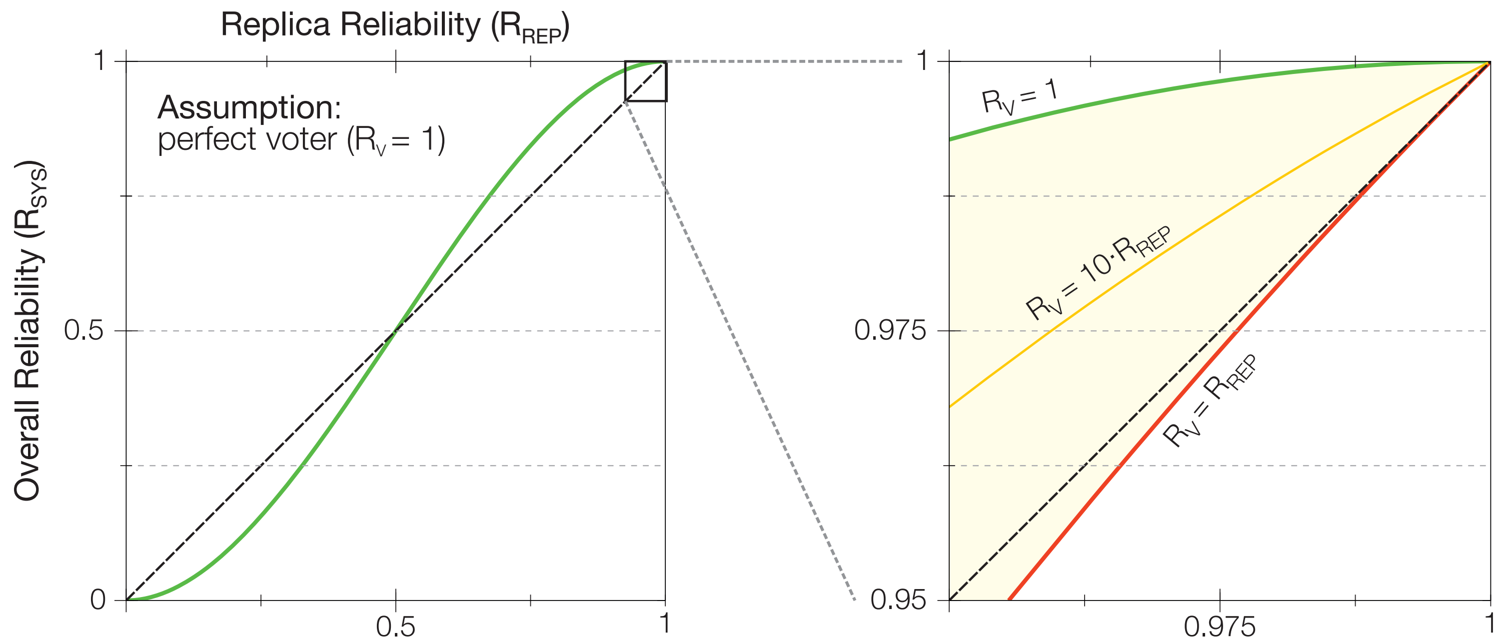


- **Triple modular redundancy reliability**

$$R_{TMR} = R_{Voter} \cdot R_{2-of-3}$$

- **Voting on unreliable hardware?**
  - Very small → residual error probability?
  - Risk analysis → inherently complex (no random error distribution! [4])

Threats to Applicability – Mission failed?



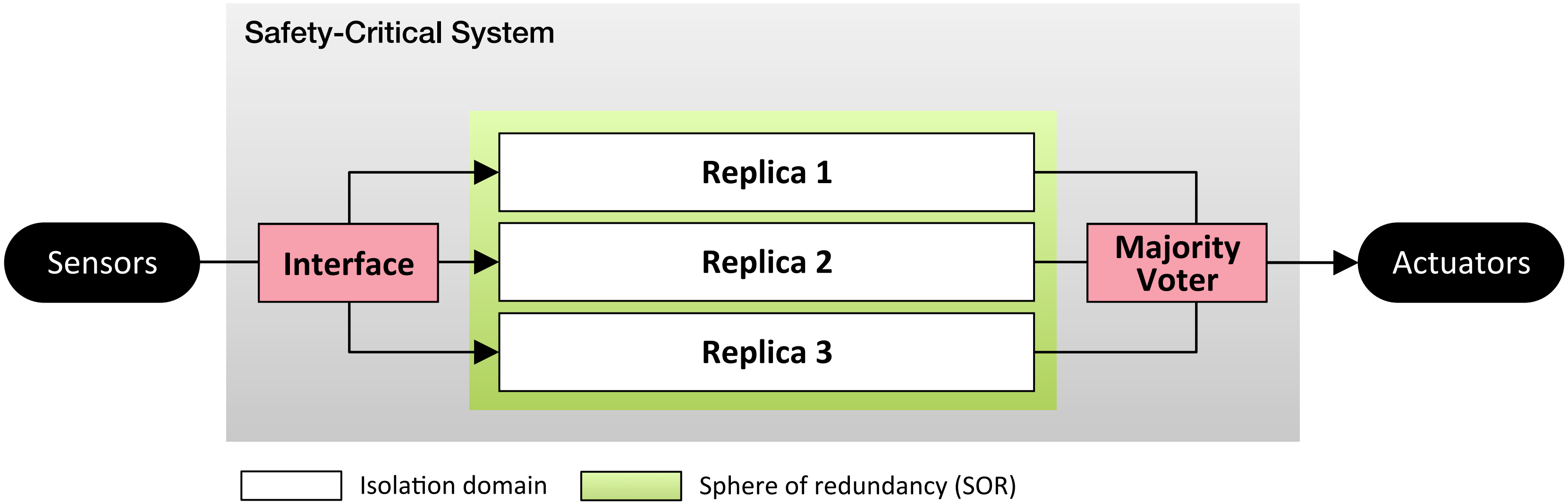
- **Triple modular redundancy reliability**

$$R_{TMR} = R_{Voter} \cdot R_{2-of-3}$$

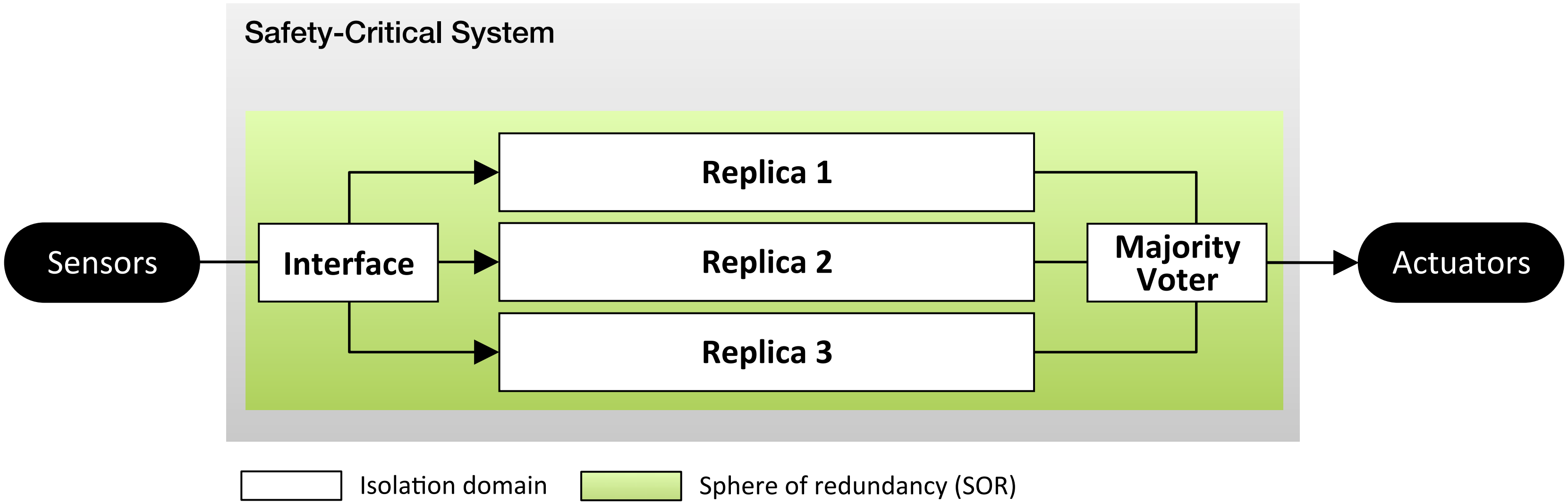
- **Voting on unreliable hardware?**
  - Very small → residual error probability?
  - Risk analysis → inherently complex (no random error distribution! [4])

→ Dealbreaker for software-based TMR

Research Aims

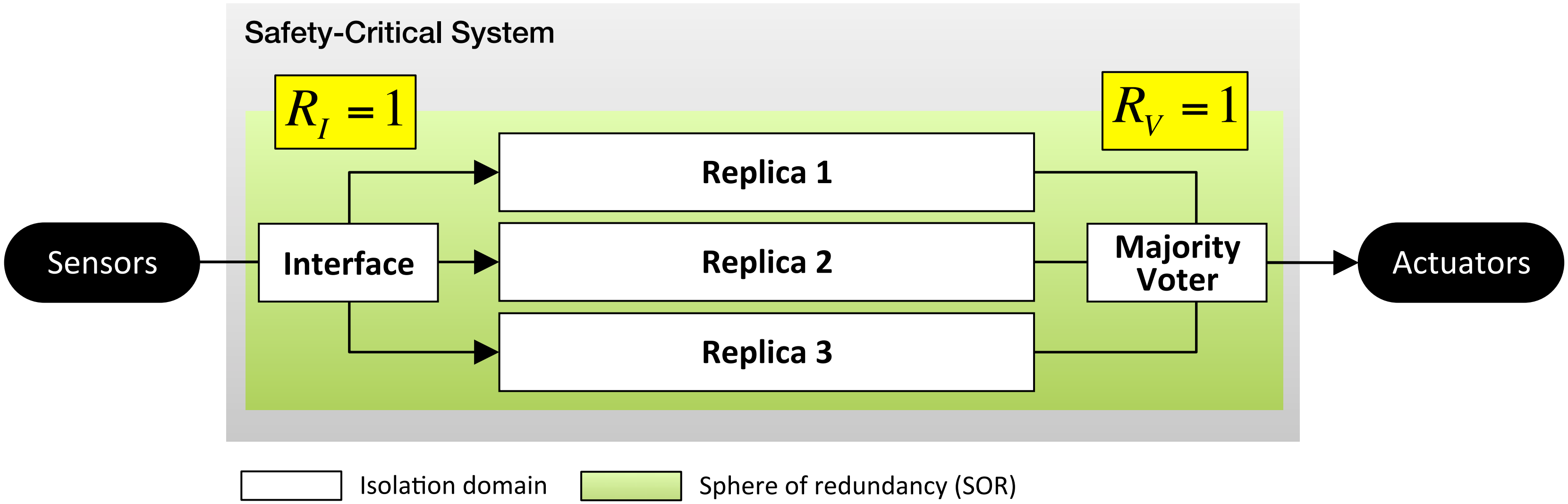


Research Aims



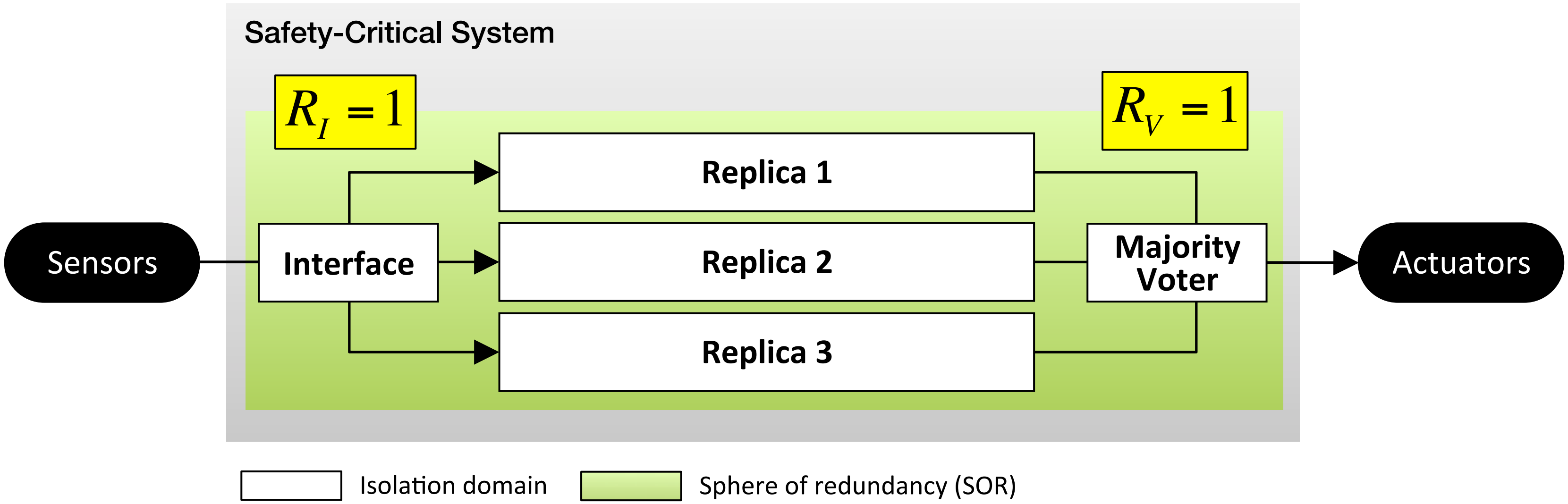
- Eliminate single points of failure
  -
  -

Research Aims



- Eliminate single points of failure
- Constrain residual error probability

Research Aims

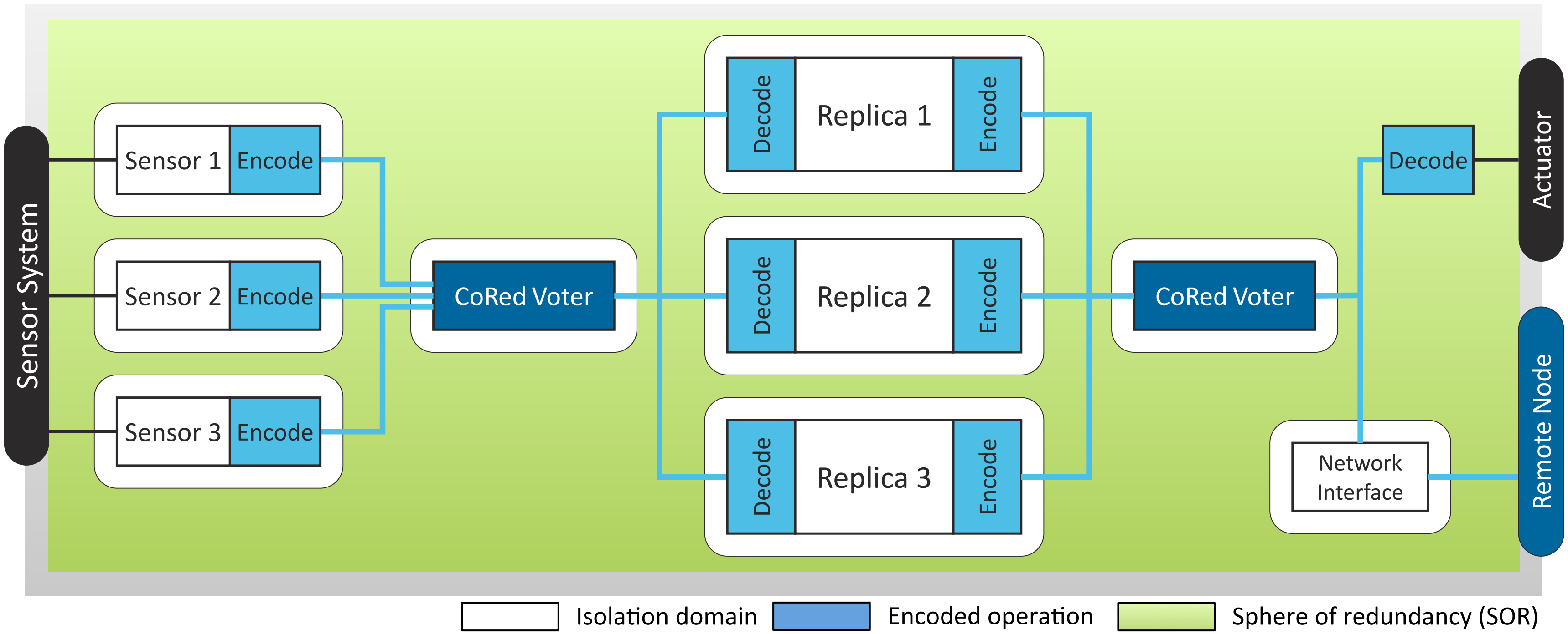


- Eliminate single points of failure
- Constrain residual error probability
- Dependability as a resource efficient option

Agenda

- Introduction
- The Combined Redundancy approach (*CoRed*)
  - Holistic protection – eliminating single points of failure
  - Arithmetic coding
  - Dependable voting
- Constraining residual error probability
  - From coding theory to application – lessons learned
  - Finding appropriate parameters
  - Circumvent implementation pitfalls
- Evaluation
  - Use case
  - Experimental setup
  - Fault-injection results
- Conclusion

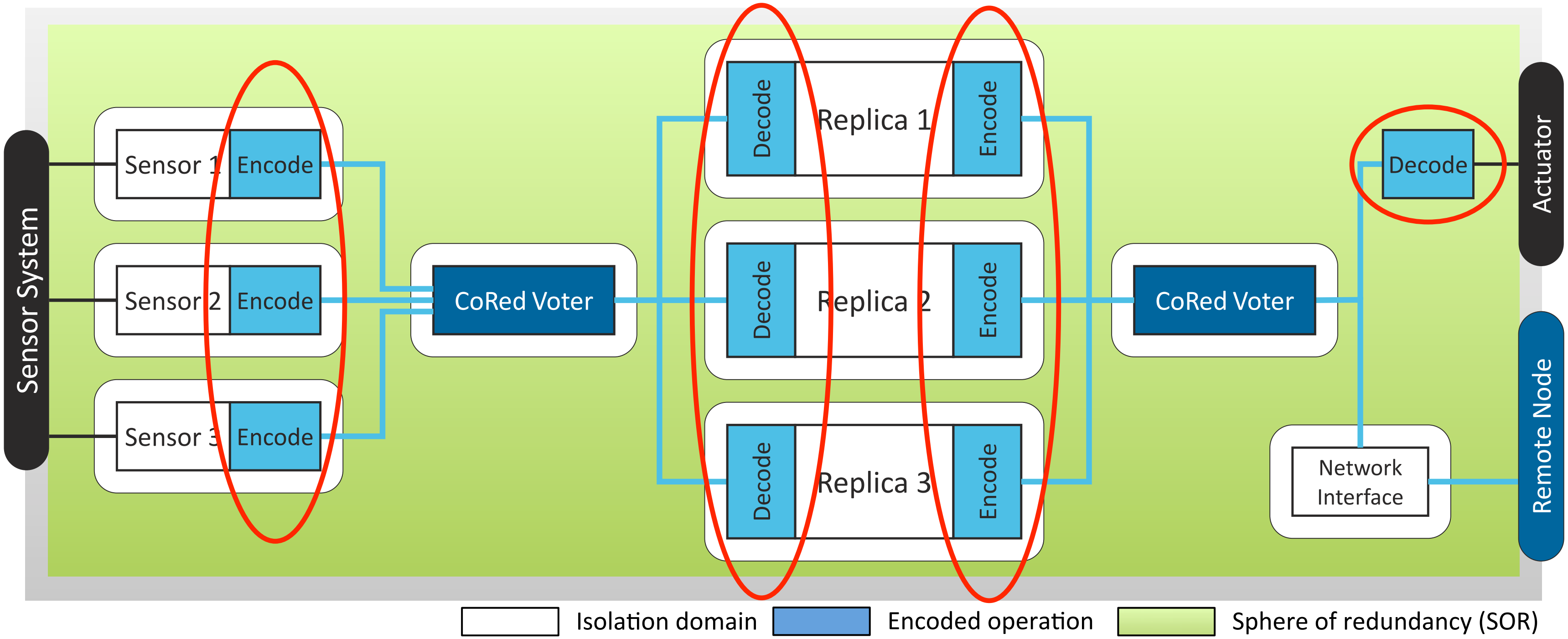
CoRed Overview – Holistic Protection Approach



- **The Combined Redundancy Approach (*CoRed*)**

TMR + {

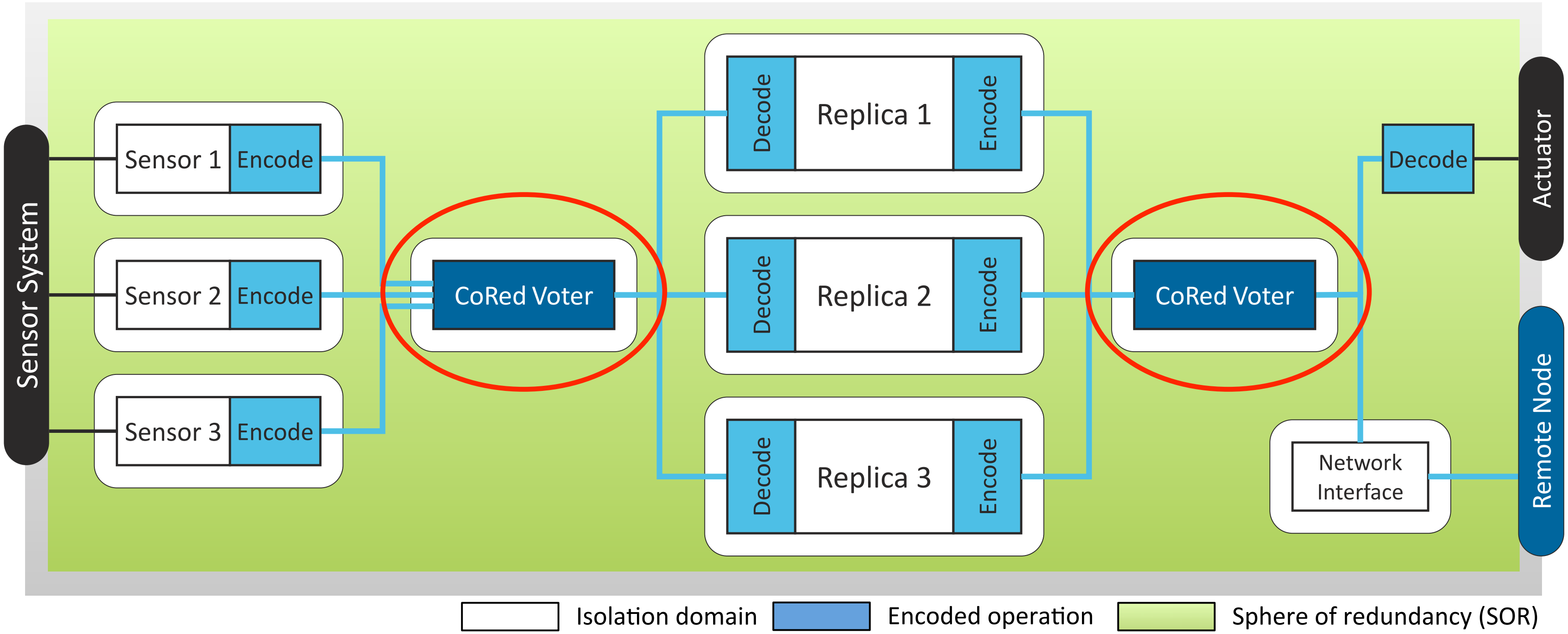
CoRed Overview – Holistic Protection Approach



- **The Combined Redundancy Approach (*CoRed*)**

TMR + { Data-flow encoding

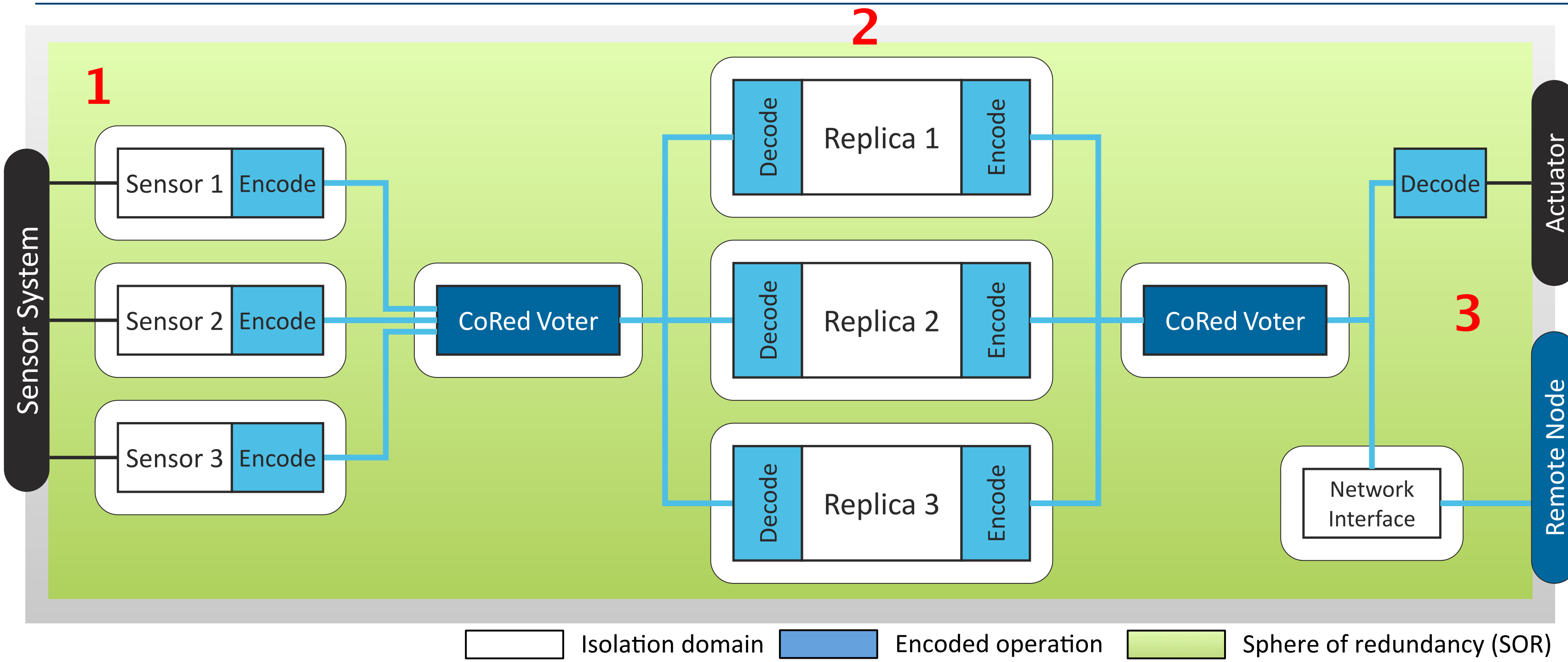
CoRed Overview – Holistic Protection Approach



- **The Combined Redundancy Approach (*CoRed*)**

TMR + { Data-flow encoding, Dependable voters }

CoRed Overview – Holistic Protection Approach



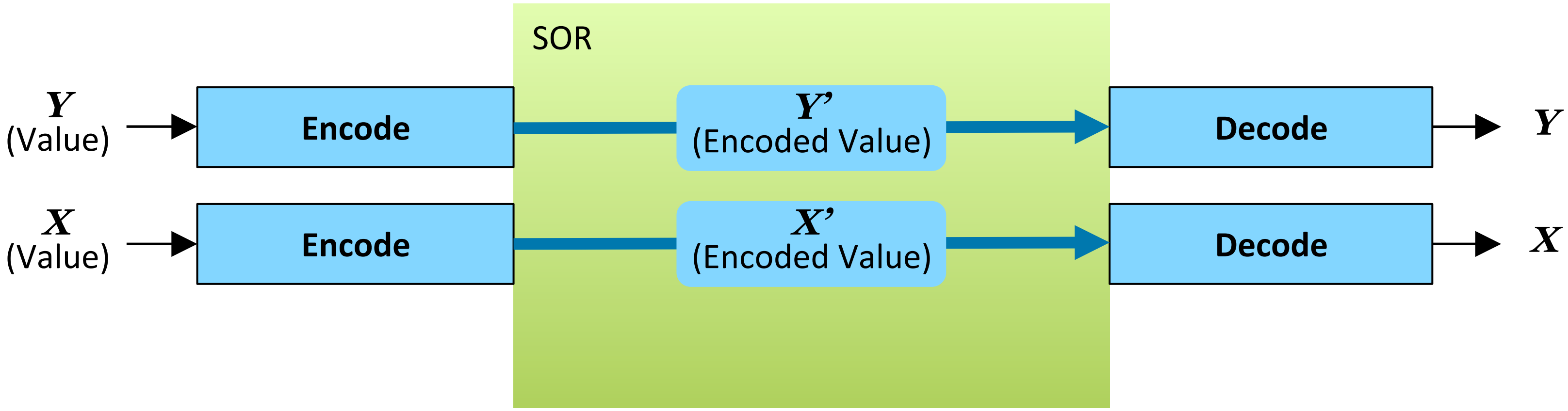
- **The Combined Redundancy Approach (*CoRed*)**

  TMR + { Data-flow encoding / Dependable voters }

- **Holistic protection approach for control applications**
  - Input to output protection

    1 Reading inputs → 2 Processing → 3 Distributing outputs

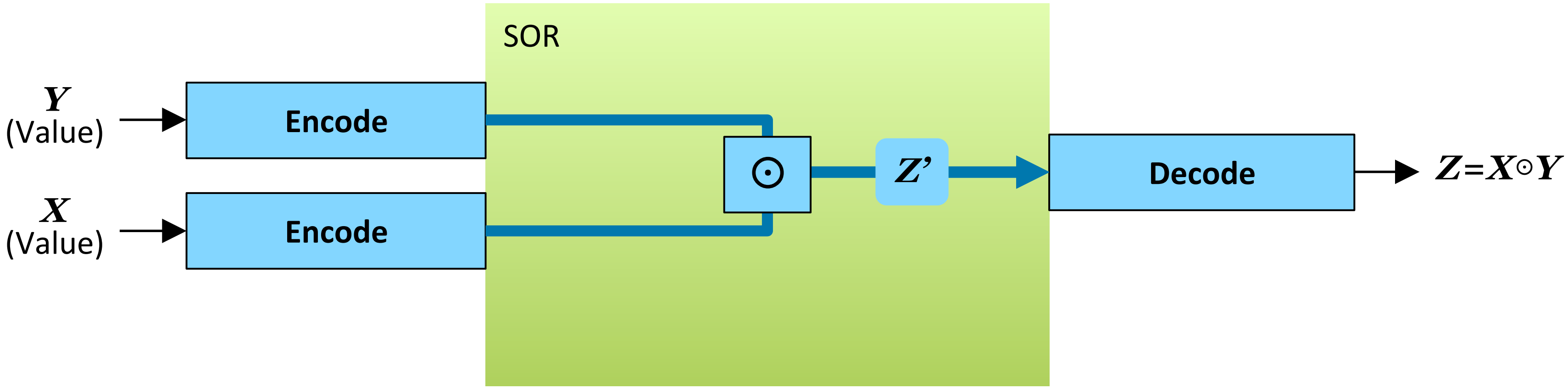
Eliminating Input and Output Vulnerabilities



- **Arithmetic Codes → ANBD Code**
  - Based on VCP [5]
  - **Data integrity:** Key
  - **Address integrity:** Per variable signature
  - **Outdated data:** Timestamp

$$v' = A \cdot v + B + D$$

Eliminating Input and Output Vulnerabilities

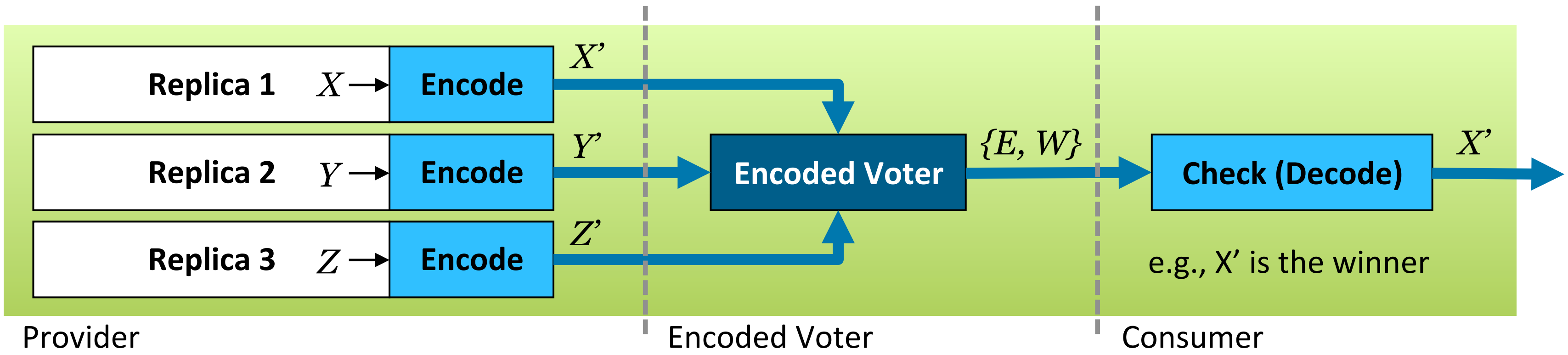


- **Arithmetic Codes → ANBD Code**
  - Based on VCP [5]
  - **Data integrity:** Key
  - **Address integrity:** Per variable signature
  - **Outdated data:** Timestamp

$$v' = A \cdot v + B + D$$

- **Set of arithmetic operators (+, -, *, =, …)**
  - Checksum vs. Arithmetic code (AN code)
  - AN Code → Encoded data operations
  - Enabler for dependable voter

CoRed Dependable Voter – Basics



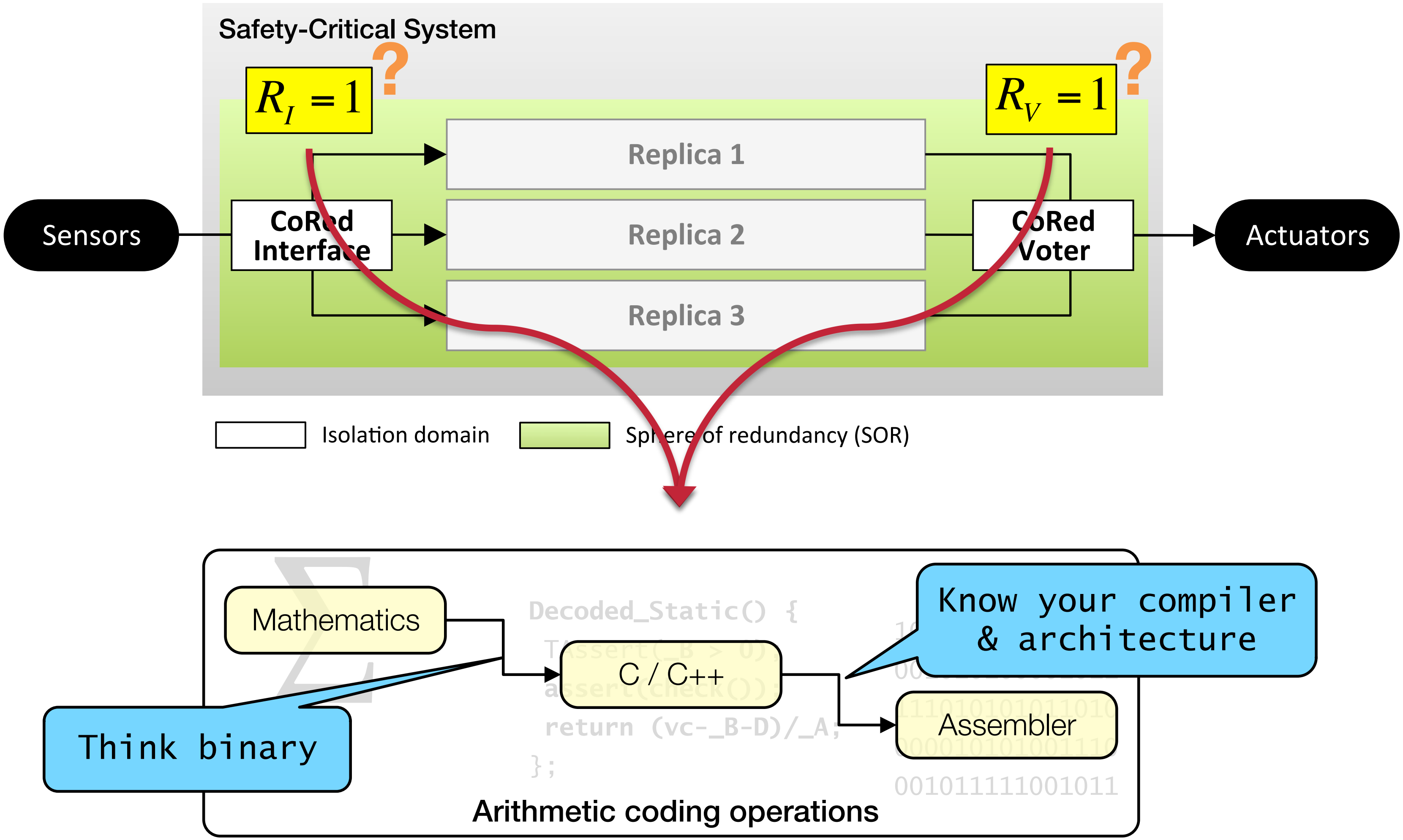
- CoRed **Dependable Voter**
  - Input: variants (*X'*, *Y'*, *Z'*)
  - Output: Equality set (*E*) and encoded winner (*W*)
  - No decoding necessary

- **Control-flow signatures**
  - Static signature (expected value): Compile-time
    → Used as return value E
  - Dynamic signature (actual value): Runtime, computed from variants
    → Applied to winner W
  - Validation: Subsequent check (decode)

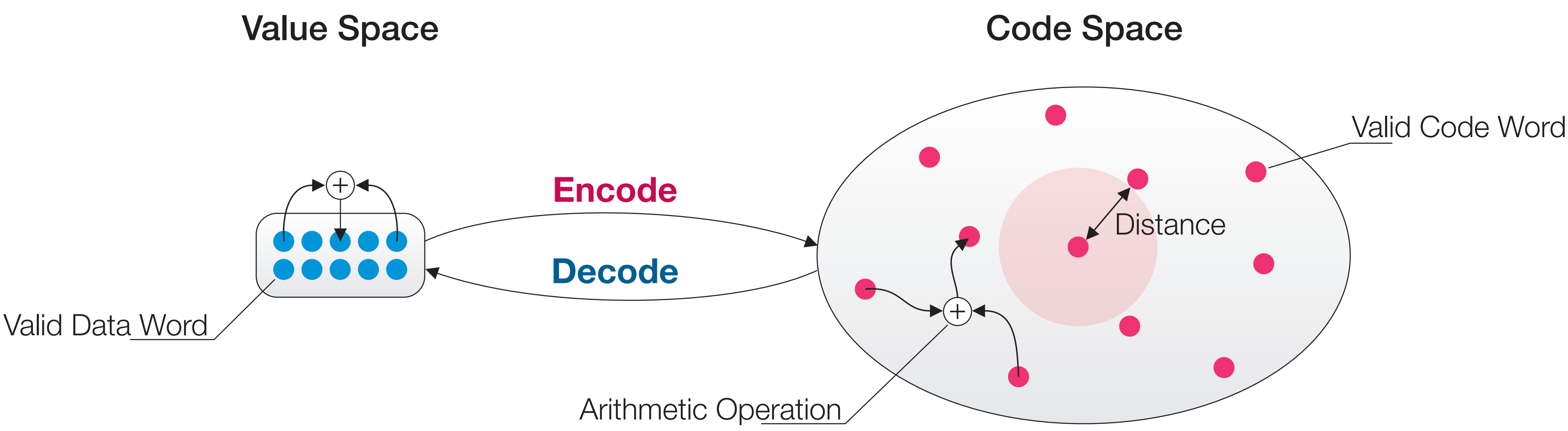
Agenda

- **Introduction**
- **The Combined Redundancy approach (*CoRed*)**
  - Holistic protection – eliminating single points of failure
  - Arithmetic coding
  - Dependable voting
- **Constraining residual error probability**
  - From coding theory to application – lessons learned
  - Finding appropriate parameters
  - Circumvent implementation pitfalls
- **Evaluation**
  - Use case
  - Experimental setup
  - Fault-injection results
- **Conclusion**

From Coding Theory to Application



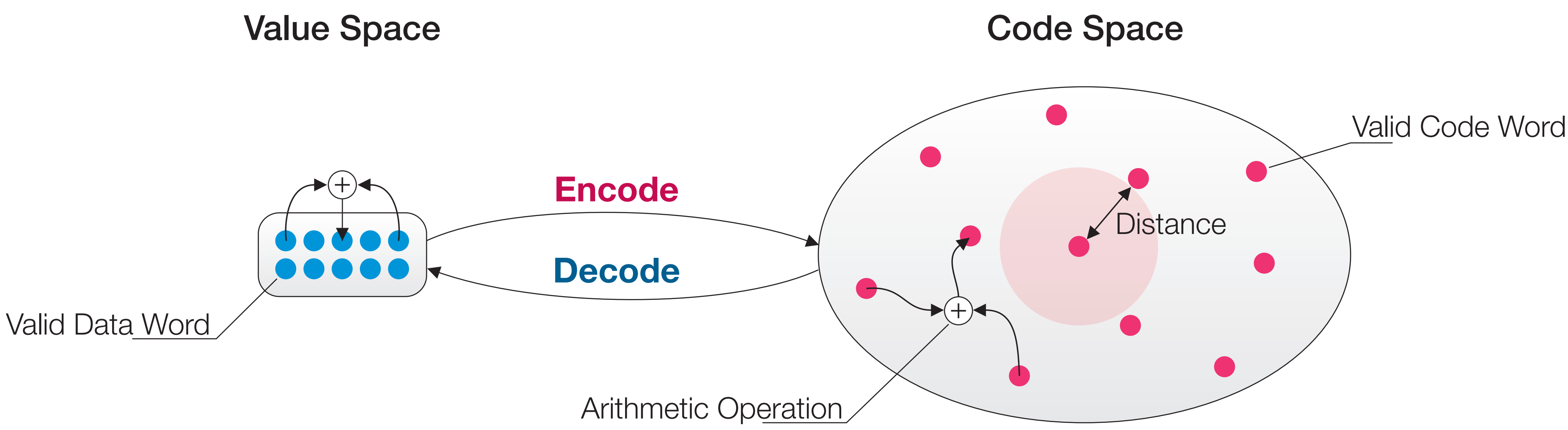
Constraining residual error probability



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

$$v' = A \cdot v + B + D$$

Constraining residual error probability



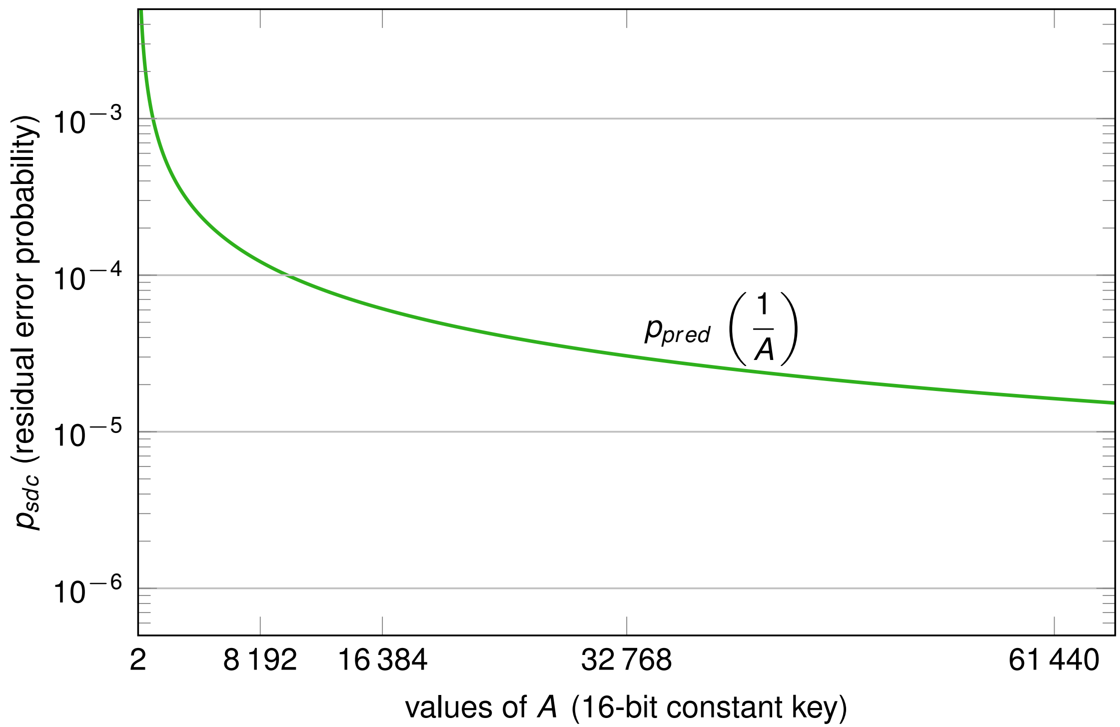
- **Coding theory**
  - Data word + redundant information = code word
  - Fault detection → distance between code words

$$v' = A \cdot v + B + D$$

- **Residual error probability**
  - Chance for code-to-code word mutation
  - Fundamental property for fault tolerance mathematics

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

Constraining residual error probability



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

$$v' = A \cdot v + B + D$$

- **Residual error probability**
  - Chance for code-to-code word mutation
  - Fundamental property for fault tolerance mathematics

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

Choosing Keys and Signatures

- **Mathematics: prime numbers**
  - Intuitively plausible
  - Literature: little help to find suitable As
- **Practitioner's approach: min. Hamming distance**
  - Distance (d) between code words (# unequal bits)
  - d-1 bit error detection capabilities

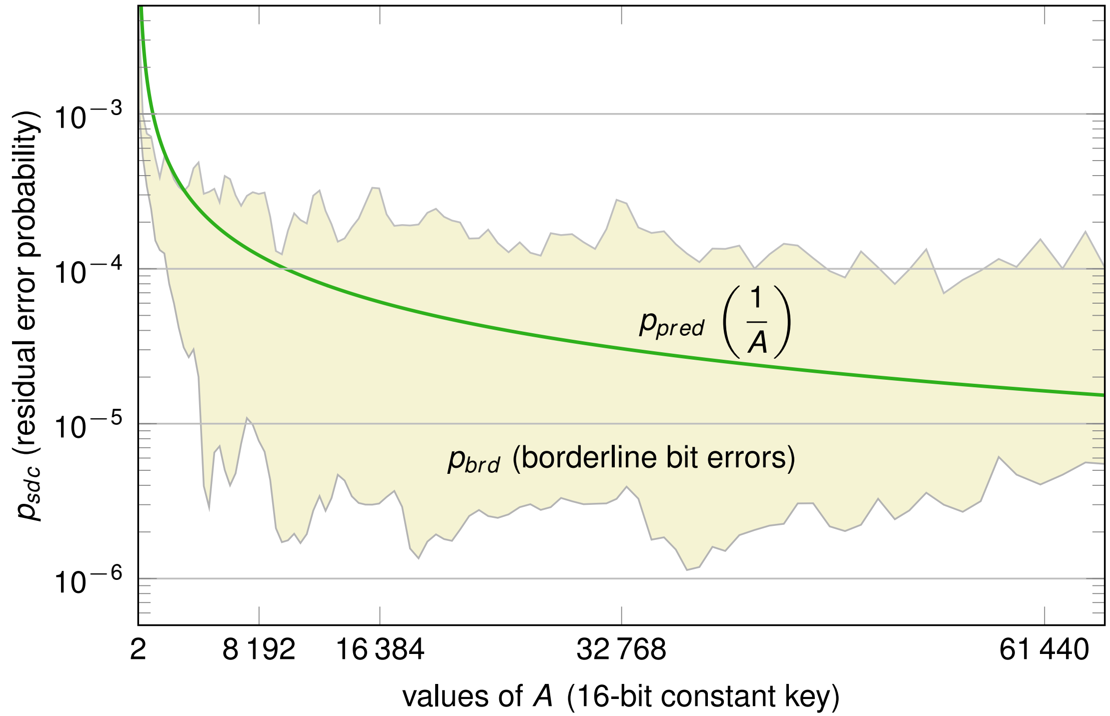
1	0	1	0
1	**1**	**0**	0

- **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!**

Consistence with Coding Theory – Mission Failed?

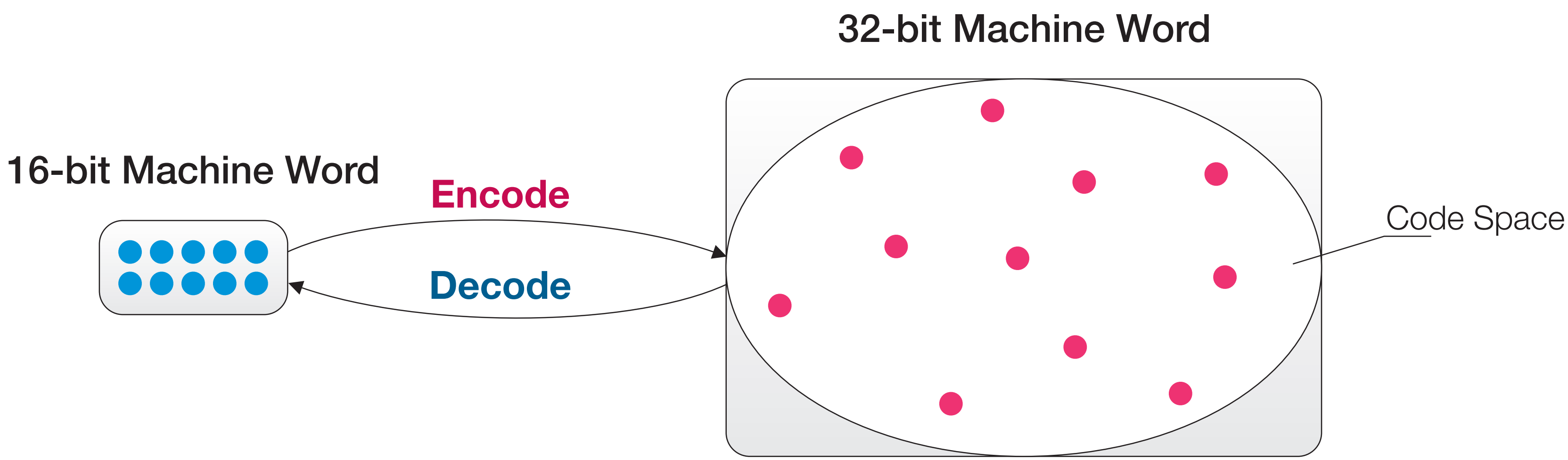


- **Fault-simulation → entire fault-space**
  - Each and every A, v and fault pattern
  - 6.5×10^{16} experiments for 16 bit As and 1-8 bit soft errors

→ Excess of predicted residual error probability

→ Violation of predicted fault-detection capabilities

Think Binary



- **Binary representation of code words**
  - Coding theory is unaware of machine word sizes
  - → Dangerous over- and underflow conditions
  - Extended AN code (EAN) implementation

→ **Compliance with coding theory!**

- **Improved code reliability ($A = 251$)**
  - Predicted 3×10^{-3}
  - Common implementation [4] $\approx 1.3 \times 10^{-2}$
  - EAN implementation $\approx 1.5 \times 10^{-5}$

→ **Improvement by orders of magnitude!**

Know your Compiler and Architecture

- **On target fault-injection → entire fault space**
  - Each and every register, flag, instruction and execution path
  - FAIL* fault injection framework [6]

→ Violation of predicted fault-detection capabilities

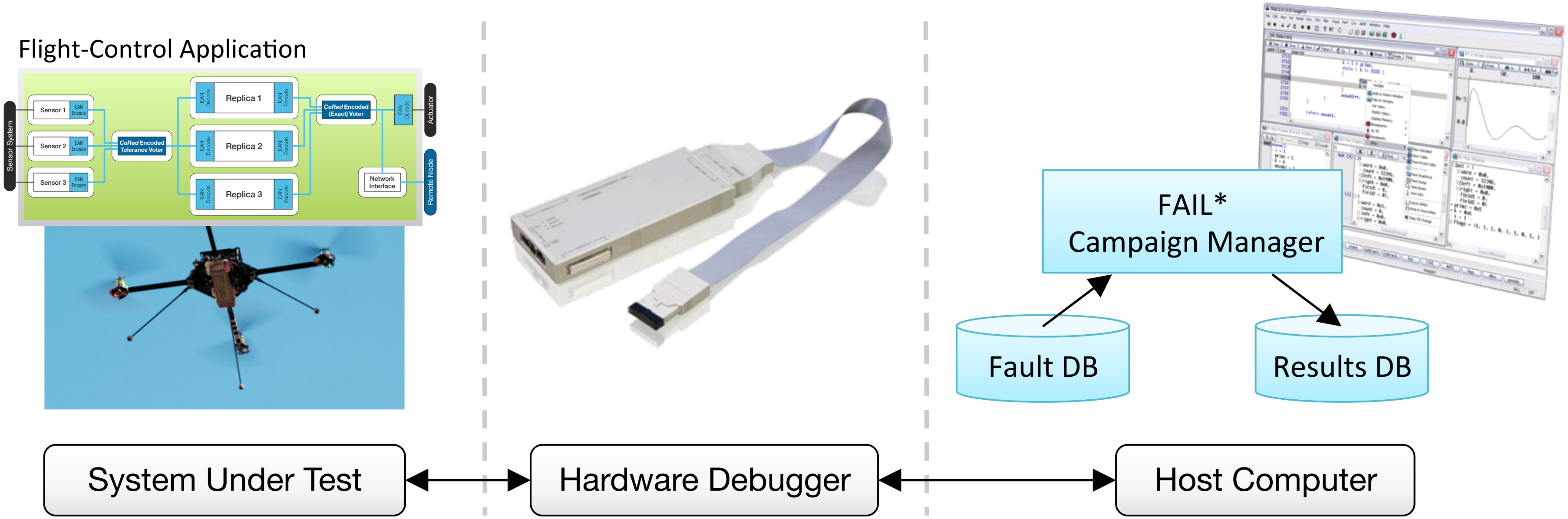
- **Architecture specifics**
  - Absence of compound test-and-branch (e.g., IA32 architecture)
  - Control-flow information is stored in single bit
  - → Redundancy is lost
  - → Additional range checks
- **Undefined Execution Environment**
  - Zombie values → leaking from caller to voter function
  - Compiler laziness leaves encoded values in registers
  - → Isolation assumptions violated
  - → Cleaning local storage restores isolation

→ Tight feedback loop with fault-injection experiments

Agenda

- **Introduction**
- **The Combined Redundancy approach (*CoRed*)**
  - Holistic protection – eliminating single points of failure
  - Arithmetic coding
  - Dependable voting
- **Constraining residual error probability**
  - From coding theory to application – lessons learned
  - Finding appropriate parameters
  - Circumvent implementation pitfalls
- **Evaluation**
  - Use case
  - Experimental setup
  - Fault-injection results
- **Conclusion**

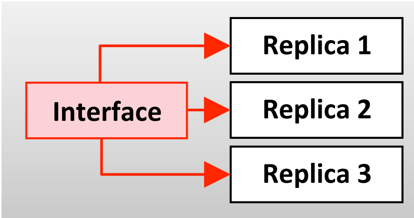
Evaluation – Experimental Setup

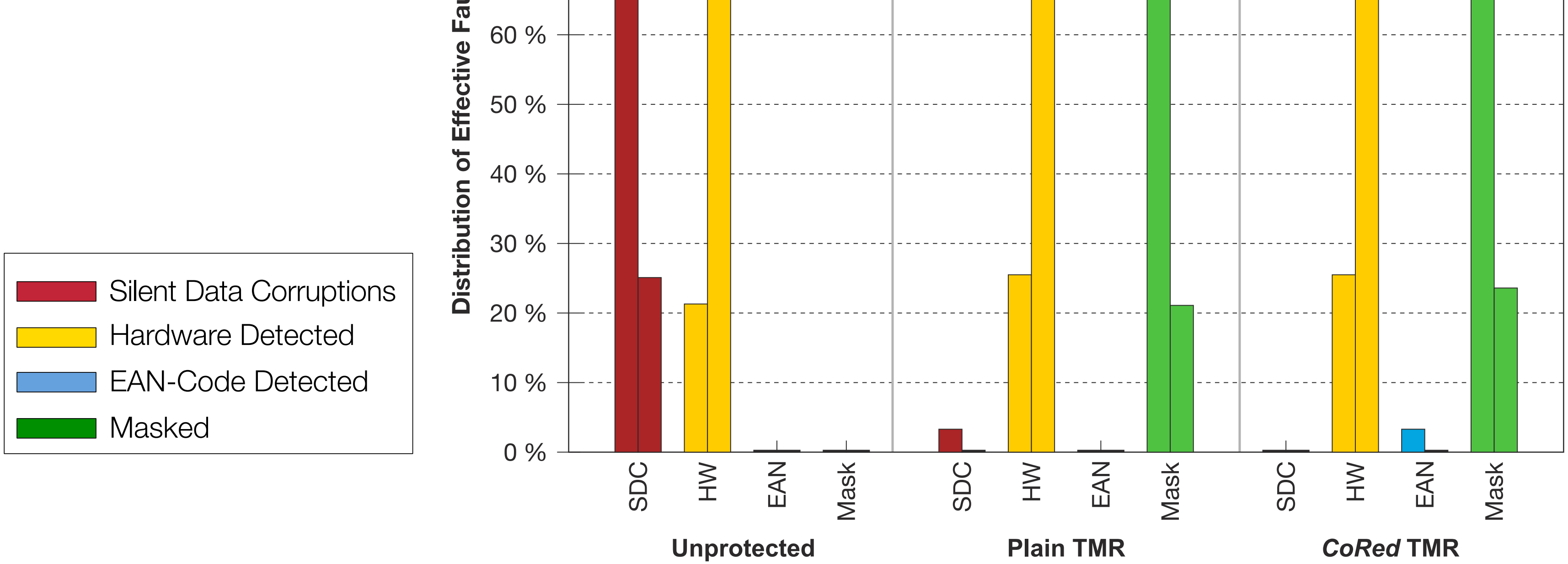


Categories: Fail Silent, Masked,
Hardware Detected, EAN-Code, Control-Flow,
Silent Data Corruption

Outcome:	401,592	experiments
Effective:	67,617	errors

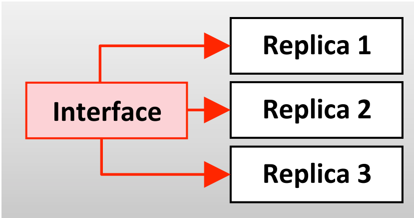
Evaluation – Experimental Results (1)

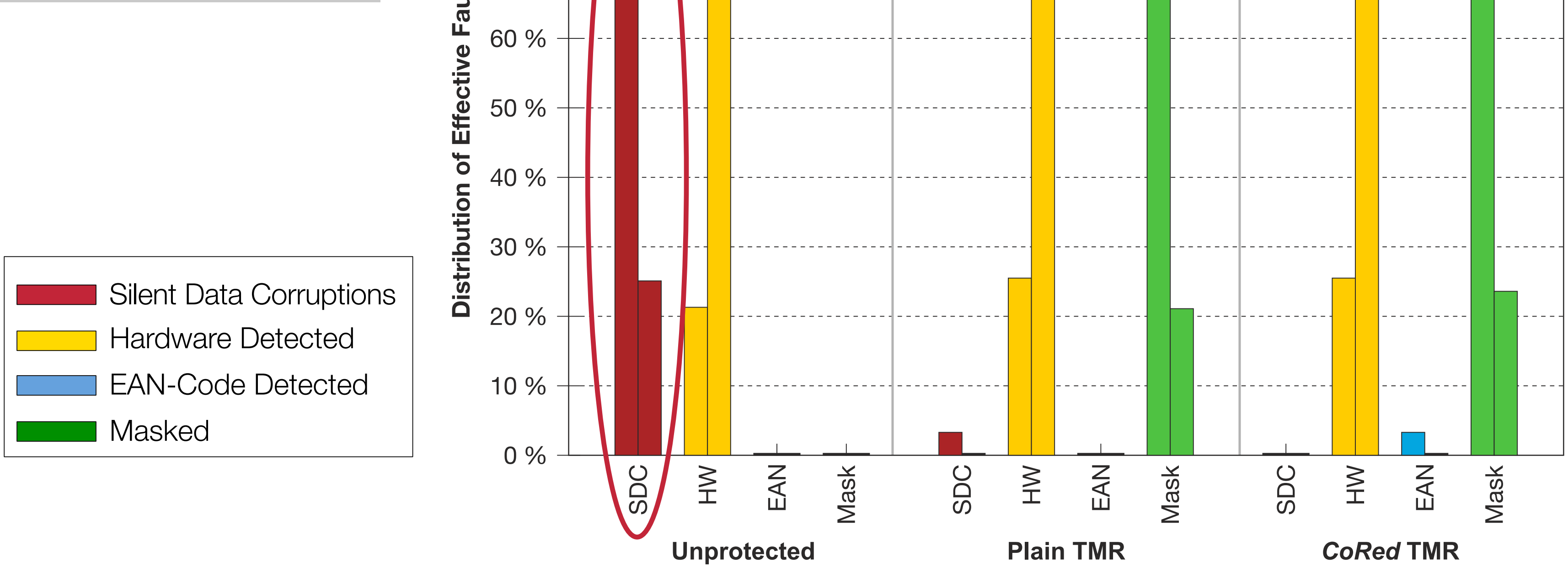




- **Redundant execution campaign (Interface)**
  - Total: ~45,000 Errors

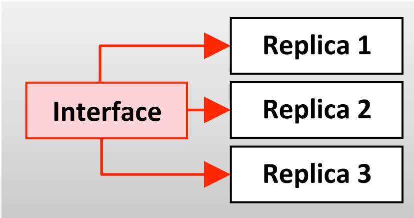
Evaluation – Experimental Results (1)

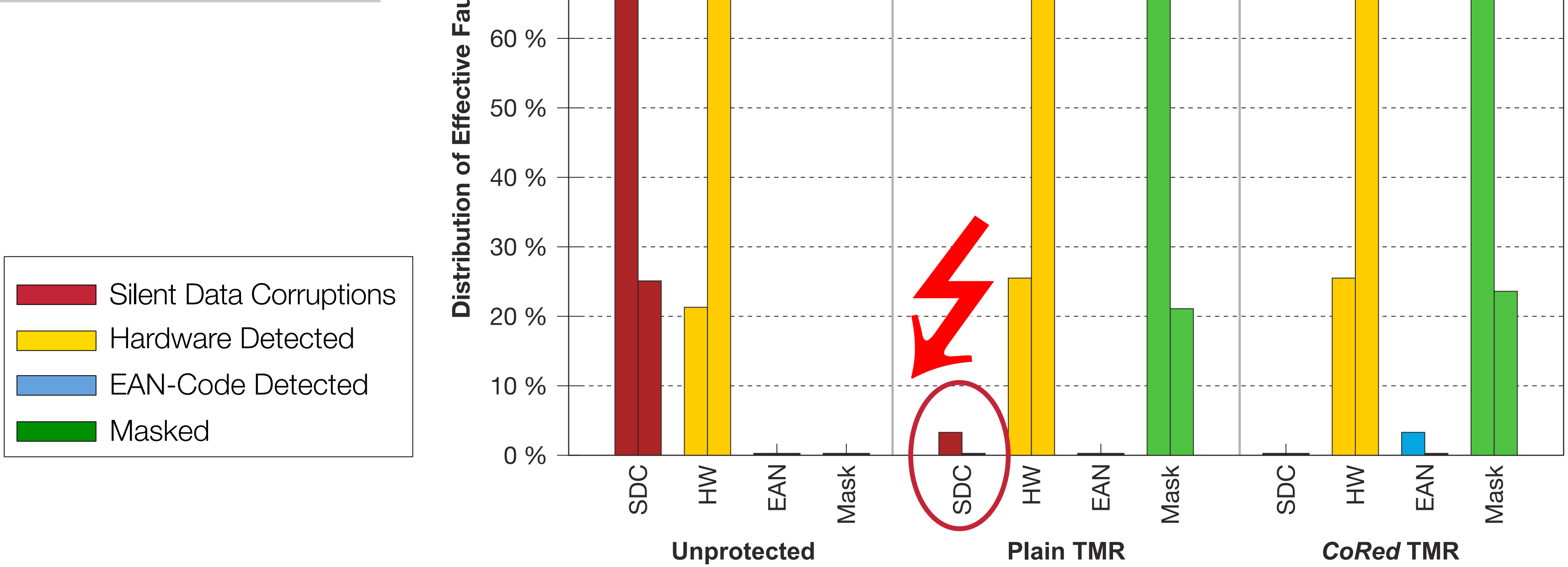




- **Redundant execution campaign (Interface)**
  - Total: ~45,000 Errors
    Unprotected: Suffers from 3,622 corruptions!

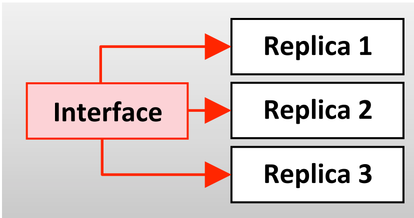
Evaluation – Experimental Results (1)

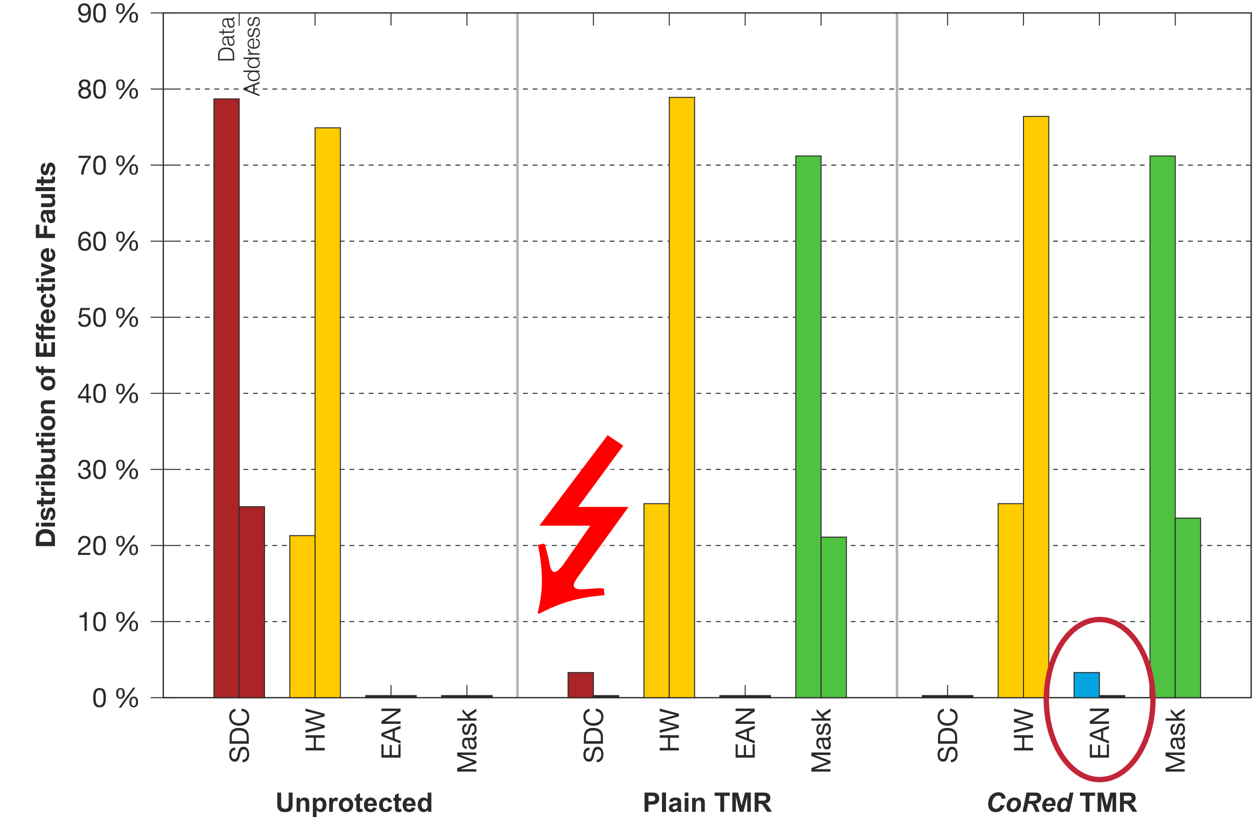




- **Redundant execution campaign (Interface)**
  - Total: ~45,000 Errors
    Unprotected: Suffers from 3,622 corruptions!
  - TMR: Suffers from 71 corruptions!

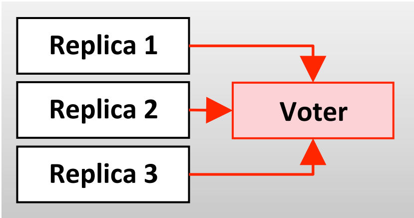
Evaluation – Experimental Results (1)

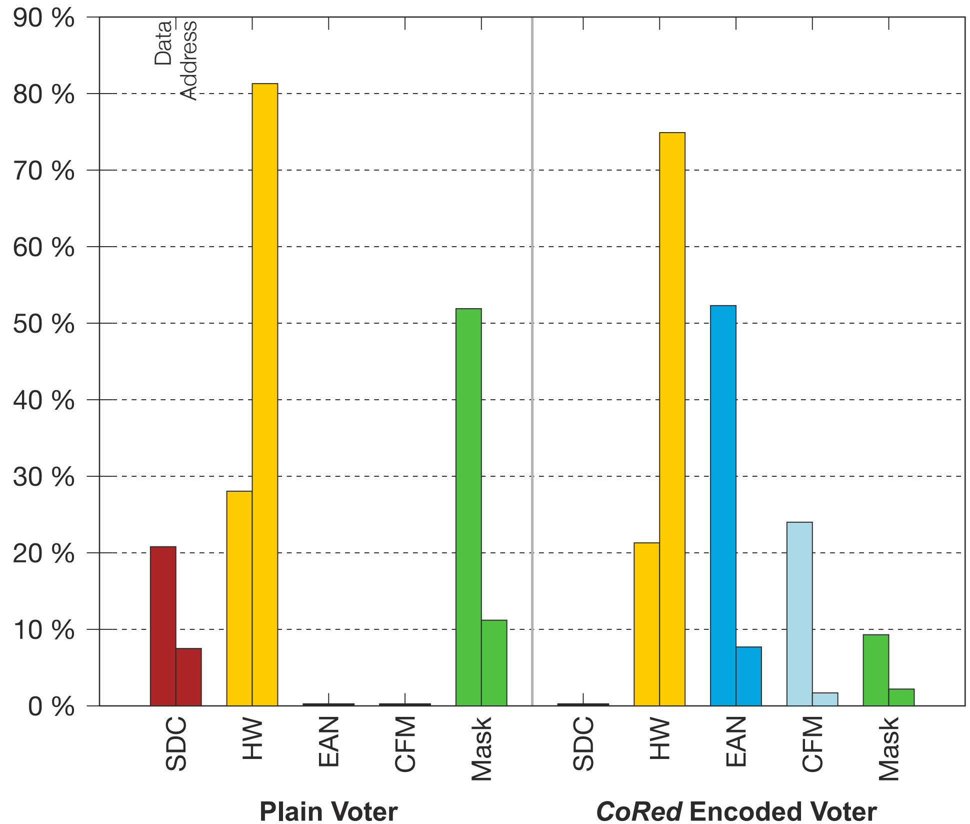




- **Redundant execution campaign (Interface)**
  - Total: ~45,000 Errors
    Unprotected: Suffers from 3,622 corruptions!
  - TMR: Suffers from 71 corruptions!
  - CoRed: Remaining corruptions are covered → 0 corruptions

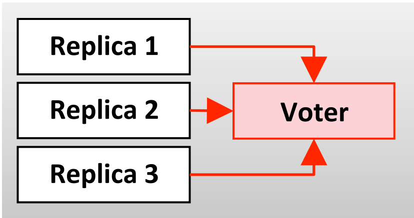
Evaluation – Experimental Results (2)

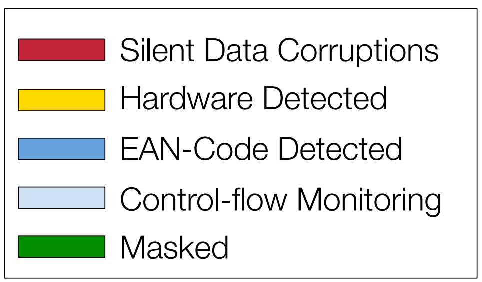


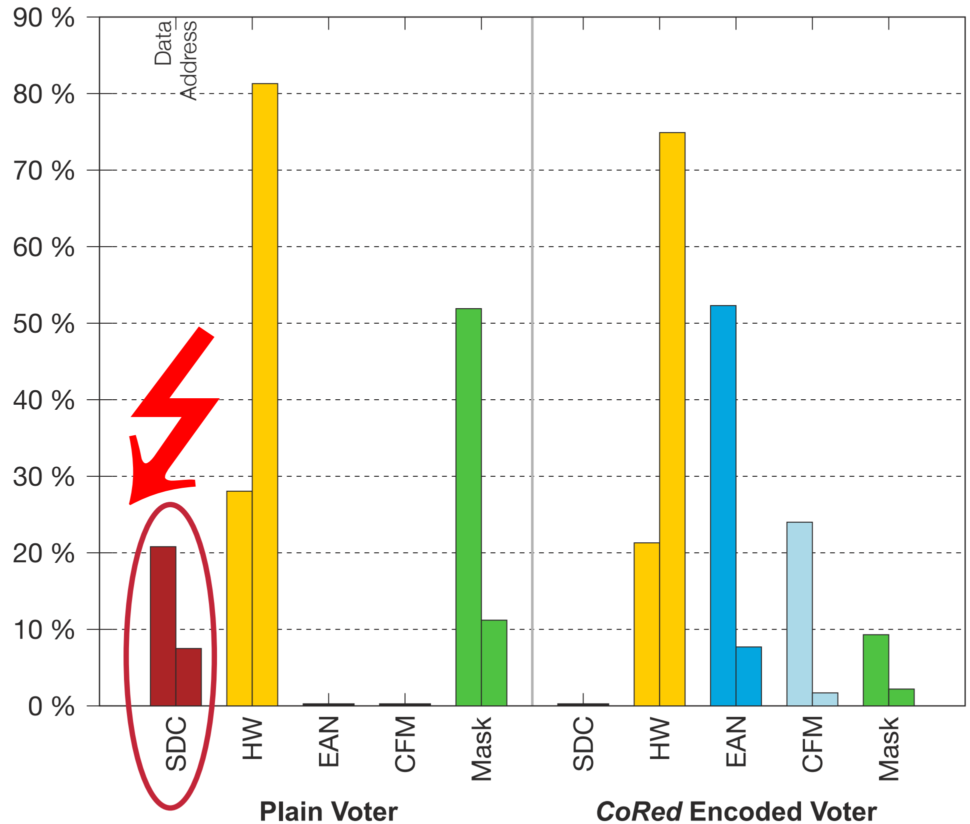


- **Voter campaign**

Evaluation – Experimental Results (2)



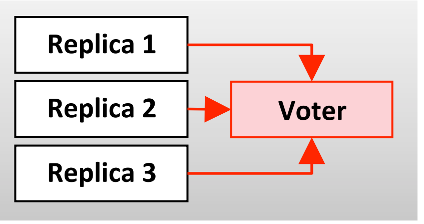


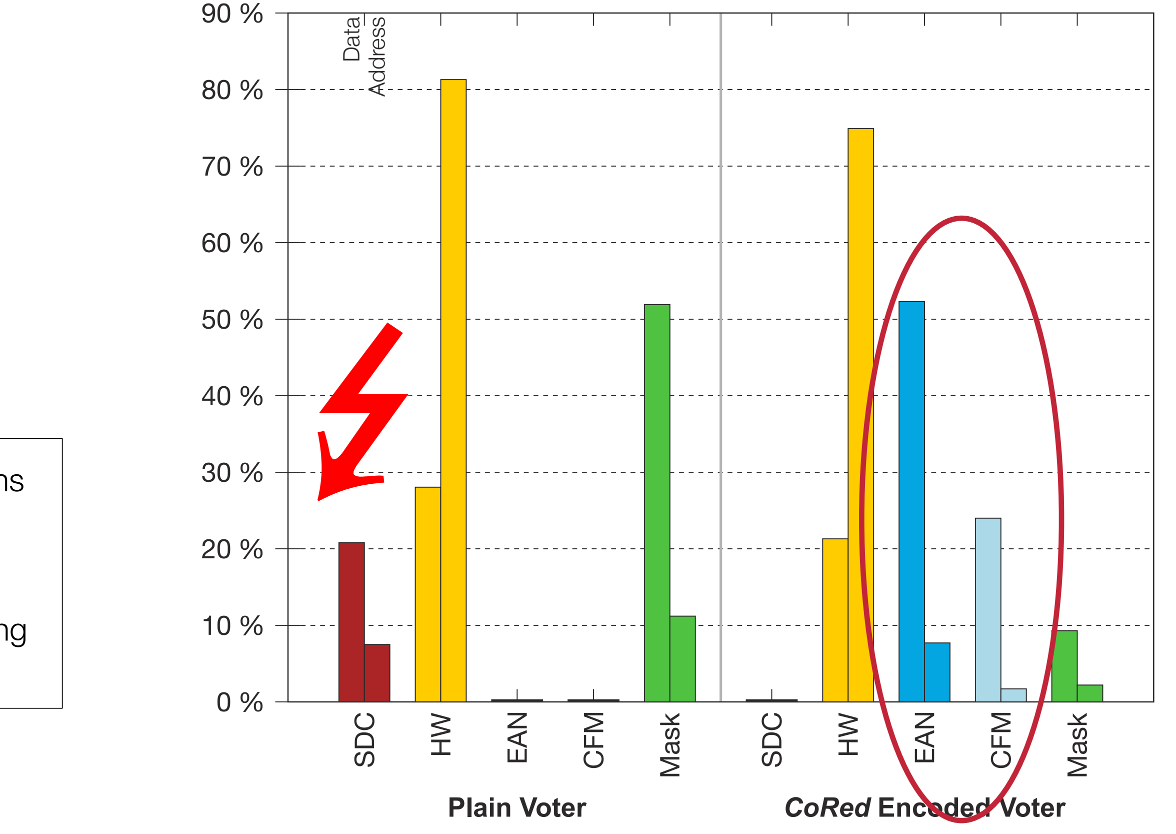


- **Voter campaign**
  - Plain voter:

    Total ~11,000 2,465 masked 7,245 retry 1,223 corruptions

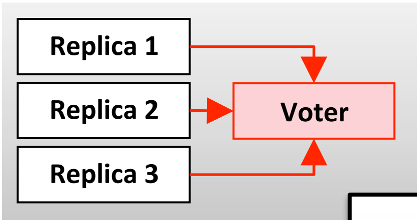
Evaluation – Experimental Results (2)

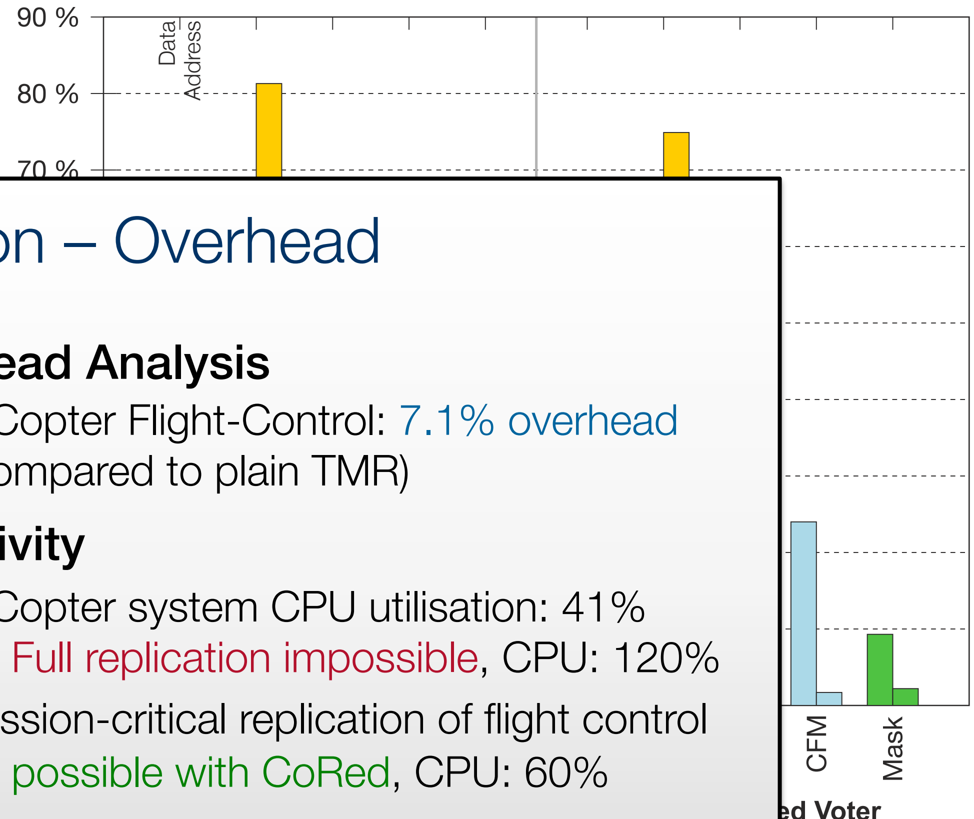




- **Voter campaign**
  - Plain voter:
    Total ~11,000 2,465 masked 7,245 retry 1,223 corruptions
  - CoRed Dependable Voter:
    Total ~26,000 1,228 masked 24,682 retry 0 corruptions

Evaluation – Experimental Results (2)





Evaluation – Overhead

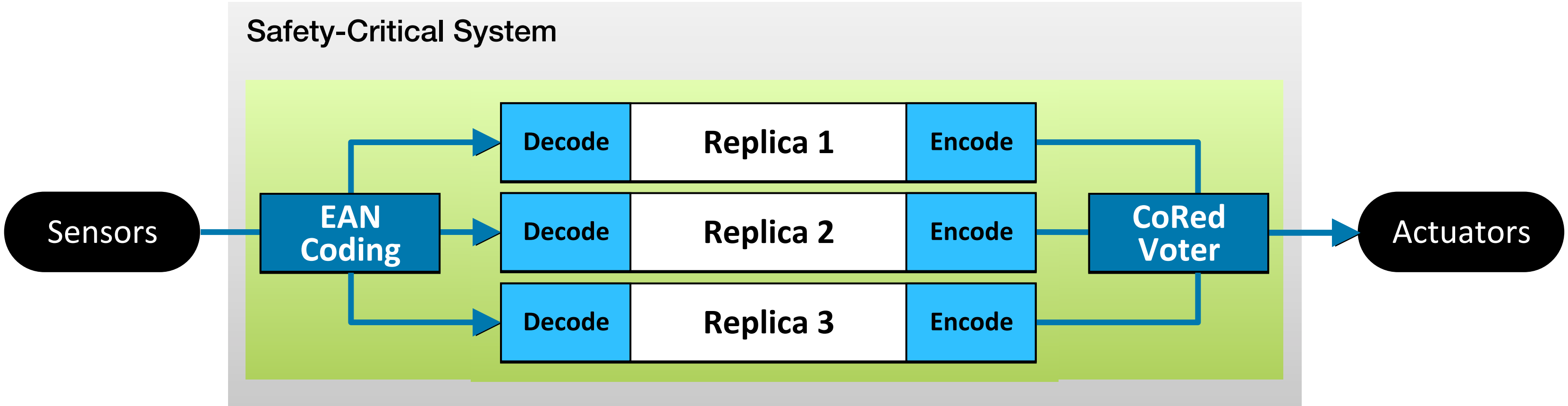
- **Overhead Analysis**
  - I4Copter Flight-Control: 7.1% overhead (compared to plain TMR)
- **Selectivity**
  - I4Copter system CPU utilisation: 41%
    → Full replication impossible, CPU: 120%
  - Mission-critical replication of flight control
    → possible with CoRed, CPU: 60%

- **Voter campai**
  - Plain voter:
    Total ~11,000 2,465 masked 7,245 retry 1,223 corruptions
  - CoRed Voter:
    Total ~26,000 1,228 masked 24,682 retry 0 corruptions

Conclusion

☑ **Eliminate single points of failure [1]**

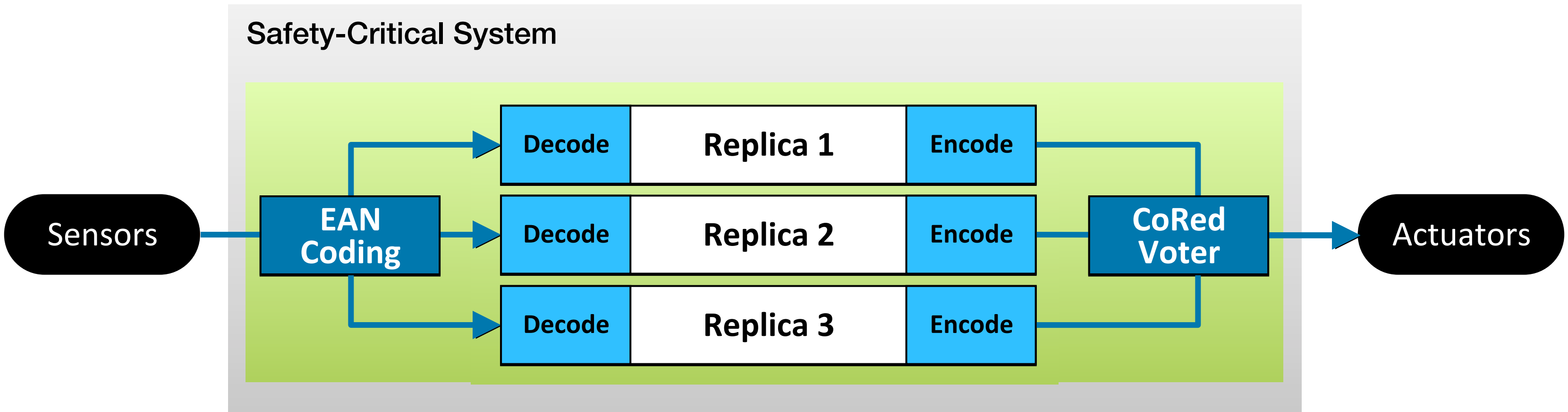
Conclusion



☑ **Eliminate single points of failure [1]**

- TMR + Encoding: Combined Redundancy approach
- Key feature: CoRed Dependable Voter

Conclusion



☑ **Eliminate single points of failure [1]**

- TMR + Encoding: Combined Redundancy approach
- Key feature: CoRed Dependable Voter

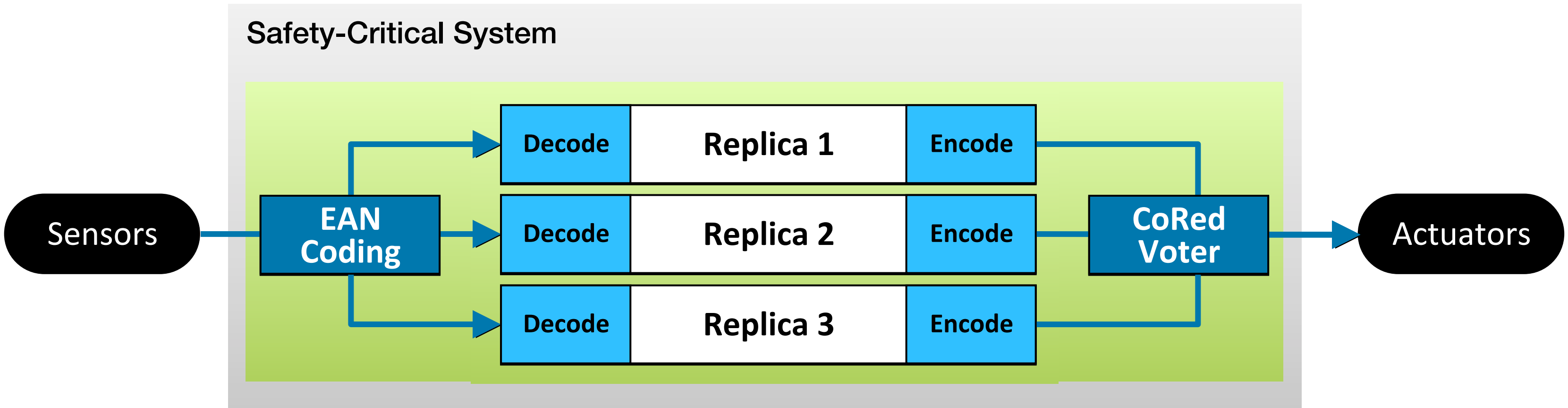
☑ **Constrain residual error probability [2]**

- Parameterisation guidelines: choosing the right A
- Binary aware implementation: complying with coding theory
- Factor 1000 improvement

☑ **Dependability as a resource efficient option**

- Only 7.1% overhead (flight control example)

Conclusion



☑ **Eliminate single points of failure [1]**
- TMR + Encoding: Combined Redundancy approach
- Key feature: CoRed Dependable Voter

☑ **Constrain residual error probability [2]**
- Parameterisation guidelines: choosing the right A
- Binary aware implementation: complying with coding theory
- Factor 1000 improvement

☑ **Dependability as a resource efficient option**
- Only 7.1% overhead (flight control example)

→ Bullet-proof software-based fault tolerance is possible

Thank you!

?

(1) Ulbrich, Peter; Hoffmann, Martin; Kapitza, Rüdiger; Lohmann, Daniel; Schmid, Reiner; Schröder-Preikschat, Wolfgang: “*Eliminating Single Points of Failure in Software-Based Redundancy*”, Proceedings of the 9th European Dependable Computing Conference (EDCC '12), 2012.

(2) Hoffmann, Martin; Ulbrich, Peter; Dietrich, Christian; Schirmeier, Horst; Lohmann, Daniel; Schröder-Preikschat, Wolfgang: *“A Practitioner's Guide to Software-based Soft-Error Mitigation Using AN-Codes“*, Proceedings of the 15th IEEE International Symposium on High Assurance Systems Engineering (HASE '14), 2014.

References

(3) P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi, “Modelling the effect of technology trends on the soft error rate of combinational logic,” in DSN ’02: Proceedings of the 2002 International Conference on Dependable Systems and Networks

(4) Edmund B. Nightingale, John R Douceur, and Vince Orgovan, Cycles, Cells and Platters: An Empirical Analysis of Hardware Failures on a Million Consumer PCs, in Proceedings of EuroSys 2011

(5) Forin, “Vital coded microprocessor principles and application for various transit systems”, 1989

(6) Schirmeier, Horst ; Hoffmann, Martin ; Kapitza, Rüdiger ; Lohmann, Daniel ; Spinczyk, Olaf : “FAIL: Towards a Versatile Fault-Injection Experiment Framework”, 25th International Conference on Architecture of Computing Systems, 2012