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

Martin Hoffmann*, Peter Ulbrich*, Christian Dietrich*, Horst Schirmeier†
Daniel Lohmann*, Wolfgang Schröder–Preikschat*
*Chair of Distributed Systems and Operating Systems, Friedrich–Alexander University Erlangen–Nuremberg, Germany
{hoffmann,ulbrich,dietrich,lohmann,wosch}@cs.fau.de
†Department of Computer Science 12, Technische Universität Dortmund, Germany
horst.schirmeier@tu-dortmund.de

***Abstract*—Arithmetic error coding schemes (AN codes[1]) are a well known and effective technique for soft error mitigation. Although coding theory being a rich area of mathematics, their implementation seems to be fairly easy. However, compliance with the theory can be lost easily while moving towards an actual implementation – finally jeopardizing the aspired fault-tolerance characteristics. In this paper, we present our experiences and lessons learned from implementing AN codes in the *CoRed* dependable voter. We focus on the challenges and pitfalls in the transition from maths to machine code for a binary computer from a systems perspective. Our results show, that practical misconceptions (such as the use of prime numbers) and architecture-dependent implementation glitches occur at every stage of this transition. We identify typical pitfalls and describe practical measures to find and resolve them. Our measures eliminate all remaining SDCs in the *CoRed* voter, which is validated by an extensive fault-injection campaign that covers 100 percent of the fault space for 1-bit and 2-bit errors.**

I. Introduction

As technology scales, hardware designs for embedded systems offer more performance and parallelism for the price of being less reliable. Therefore, soft-error mitigation is one of the major challenges for safety-critical applications and systems. Besides adding costly hardware redundancy, virtually sacrificing the technology gain, software-based fault-tolerance offers a selective and resource-efficient alternative.

As systems engineers, we aim for a selective manipulation of *dependability* as a non-functional property at the operating-system level and independent of any specific hardware support. Here, arithmetic error coding, or *AN codes*[1] for short, can be one vital component to tackle transient hardware faults in software.

A. The CoRed Approach

In our previous work, we presented *Combined Redundancy* (*CoRed*) [1], a highly effective, software-based fault-tolerance approach for mixed-criticality control applications. *CoRed* engages right in-between application and operating system and combines proven triple modular redundancy (TMR) with AN codes to minimize soft error effects and improve the actual reliability. *CoRed*'s modular architecture facilitates the composition of various fault-tolerant components (see Figure 1). At the same time, it eliminates dangerous single points of failure

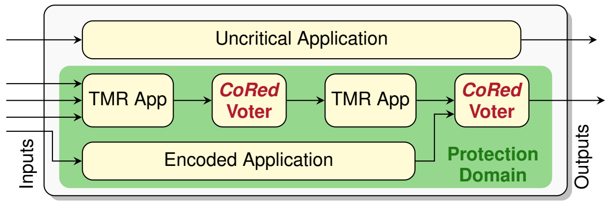


Fig. 1. *CoRed* facilitates mixed-criticality systems by providing a safe interconnect between different fault-tolerance schemes. Its key component is the *CoRed* voter, which is hardened against soft errors.

(SPOFs) caused by gaps between fault-tolerant components. For that, we introduced the *CoRed dependable voter* (*CoRed* voter): a high-reliability voting schema, based on our *Extended AN* (EAN) error coding implementation (see Section II).

We successfully employed *CoRed* in the mission-critical flight control of the *I4Copter* quad rotor UAV [2], which has been specifically designed to resemble real-world control applications. Here, *CoRed* maintains a continuous protection domain from the sensors' inputs up to the actuator output elements. In this type of application, the *CoRed* voter is of particular value due to the complex component interconnection. Therefore, we see it as the *key element* to ensure the overall reliability of the safety-critical control applications.

B. Problem Statement

Any divergence from an assumed residual error probability complicates the safety-assessment and impedes the general applicability of our approach. Therefore, our goal with the *CoRed* voter was to provide full fault coverage for single-bit soft errors and to reach the residual error probability predicted by the arithmetic error coding theory in general. However, during its implementation we experienced certain inconsistencies with the assumed fault-detection capabilities.

Experiments with our error-coding framework revealed unexpected *silent data corruptions* (SDCs) and discrepancies from the theory's predicted error probabilities. These deviations are in line with the observations made by other researchers [3] and we were able to reproduce their results as well. However, we were unable to find any profound explanation for this phenomenon in our earlier work.

Applying expertise from the low-level systems domain, we were able to trace back the sources of discrepancy to *every*

This work was partly supported by the Bavarian Ministry of State for Economics, Traffic and Technology under the (EU EFRE funds) grant no. 0704/883 25 and the German Research Foundation (DFG) priority program SPP 1500 under grant no. LO 1719/1-1 and SP 968/5-1.

[1]Named by the integers A (constant key) and N (value).

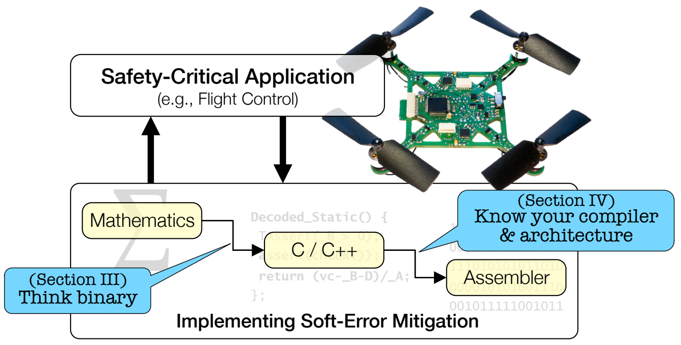


Fig. 2. From theory to application: Implementing arithmetic error coding is attended by various pitfalls and challenges.

stage of the transition from the coding theory to the machine-dependent implementation (Figure 2). The identified issues include misconceptions regarding binary number representation and ranges, silent assumptions about the compiler, and specific characteristics of the hardware platform. Although the *semantics* of the algorithm are preserved in the transition over all stages, the concrete *execution* can significantly diverge.

In consequence, a practitioner cannot rely blindly on coding theory, as it is extremely difficult to predict the implications on the residual error probability when the algorithm is realized on a concrete hardware platform.

C. Our Contribution

In this paper we present a practitioner's guide on how to deal with these challenges by illuminating typical problem areas and presenting feasible solutions. A detailed re-evaluation of the *CoRed* voter implementation allows for a deeper insight into the transition steps from coding theory to its machine-dependent implementation. We show the difficulties of a binary-aware parametrization of arithmetic error coding by a set of constructive experiments. Moreover, we provide an EAN code as well as a voter implementation, which overcome the transformation issues and preserve the intended execution behaviour. At large, the error coding as well as the *CoRed* voter behave as predicted, eventually relieving the practitioner from dealing with non-functional dependability aspects. In our opinion, the gained experience can be generalized to common guidelines for the parametrization, implementation, and evaluation of arithmetic error coding in practice.

The key contributions of this paper are:

- A binary-aware analysis of AN code parametrization and fault-detection capabilities.
- Disclosure of typical pitfalls on the transition from coding theory to machine-level instructions.
- An improved EAN code and *CoRed* voter implementation, which comply with the coding theory.
- Experimental verification and fault injection of the *CoRed* voter, covering the *entire* 1 & 2-bit fault space.
- A first glance on the implications of multi-bit faults.

II. Background & Related Work

Coding theory is a rich area of mathematics. However, from a practitioner's point of view, the only thing that matters is effectiveness – and potentially overhead. Before immersing in the challenges and pitfalls we faced within *CoRed*'s EAN code and voter implementation, this section introduces the necessary coding basics and the assumed fault model in a nutshell, and details the corresponding related work.

A. Fault Model

From a software perspective, soft errors manifest as malicious defects within machine-level instructions, being the software's most basic structural elements. Independent from the underlying hardware, three fault classes can be distinguished: *operand*, *operator*, and *arithmetic errors* [4]. To put it simple, soft errors may lead to corrupted data, unexpected operations or simply wrong results. Complex fault patterns and higher levels of abstraction can be mapped to these elementary fault classes [5]. Accordingly, the effectiveness of an error coding scheme is conceptually tied to the fault coverage of this model. It therefore serves as a comprehensive classification base throughout the rest of this paper.

B. Arithmetic Error Coding

Common error codes, such as parity or cyclic redundancy checks, are widely used in communication and memory applications [6]. They are, however, restricted to data flows and not designed to cover computational flaws such as operator or arithmetic errors. In contrast, arithmetic error coding can cope with all error classes and therefore provides protection for the actual program execution as well. The major difference between data and arithmetic error coding is a set of code-preserving arithmetic operations, which allow for computation with the encoded values.

There exist different flavours of arithmetic error coding schemes which differ mostly in their fault-model coverage: Residue codes [7] and AN codes are the simplest form and use only the *plain value* (v) and the *constant* (A) for encoding. AN codes are commonly used for software-based soft-error mitigation [8], [9], although they lack coverage for certain operator and address errors. To improve on this, ANB codes introduce an additional per-variable *signature* (B), which allows for detecting interchanged operands and operators [10]. Finally, Forin [4] proposed *timestamps* (D) to cover outdated operands as the last gap in the codes fault coverage, forming the ANBD coding scheme. The encoding is defined by the standard AN and the BD fraction, as:

$$\underset{\text{Encoded value}}{v_c} = \overbrace{\underset{\text{Constant}}{A} \cdot \underset{\text{Value}}{v}}^{AN} + \overbrace{\underset{\text{Signature}}{B_v} + \underset{\text{Timestamp}}{D}}^{BD}$$

Initially designed for hardware-level protection [4], ANBD codes are also employed in software-based approaches. Here, they commonly safeguard entire applications and systems [10], [11], [12], [3]. We use ANBD codes as the basis for our *CoRed* design because of their complete fault coverage.

C. The CoRed EAN Code Implementation

To avoid confusion with other implementations, we labelled our ANBD coding framework *Extended AN* (EAN). While it fully adopts the ANBD coding theory and concepts, its implementation is specifically tailored to our needs and incorporates the improvements we will discuss later in this paper.

Within the scope of this paper, the most significant difference is the additional equality operator provided by EAN. For the specific purpose of finding a majority, we simplified the standard comparison according to Equation 1. The equality of two encoded values can be easily determined by observing their difference:

$$\begin{aligned}\overbrace{(Ax + B_x + D)}^{x_c} - \overbrace{(Ay + B_y + D)}^{y_c} &= Ax - Ay + B_x - B_y \\ &= B_x - B_y \Leftrightarrow x = y\end{aligned} \tag{1}$$

If equal, the difference of the encoded values (x_c, y_c) must match with the constant difference of their signatures (B_x, B_y).

D. The CoRed Voter

Generally speaking, the *CoRed* voter can be seen as an encoded application itself. Its implementation is comprehensible and self-contained and therefore, in our opinion, especially suitable as a showcase. For simplicity's sake we *attribute timestamps to signatures* within the voting algorithm and throughout the rest of this paper.

Figure 3 illustrates the voter's fundamental algorithm. The basic idea is to find a quorum on *encoded* values without losing the code's protection. Therefore, the voter accepts the three encoded variants (x_c, y_c, z_c) and provides a winner (win) along with the voting result (equality set E) in terms of a *constant signature* B_E (lines 15, 18, 22, and 25), defined by:

$$\begin{aligned}\overbrace{\{x_c, y_c, z_c\}}^{\text{equality set: } E} &\Leftrightarrow \overbrace{(B_x - B_y) + (B_x - B_z)}^{\text{constant signature: } B_E} \\ \{x_c, y_c\} &\Leftrightarrow (B_x - B_y) \\ \{x_c, z_c\} &\Leftrightarrow (B_x - B_z) \\ \{y_c, z_c\} &\Leftrightarrow (B_y - B_z) \\ \{\} &\Leftrightarrow noDecision\end{aligned} \tag{2}$$

With the data integrity ensured, the voting algorithm itself can still suffer from soft errors, for example causing false branch decisions. We solved this problem by introducing program flow checks in terms of EAN scope signatures. The voter computes a *dynamic signature*, based on the branch decisions that lead to a certain verdict in the correct case, and applies this signature to the selected variant (lines 14, 17, 21, and 24) before returning it as the winner. The voting procedure was error-free, if static and dynamic signature match. To put it simple, the control-flow is recomputed at run-time and stamped to the winner. Outside the voter's protection domain, any succeeding software component can subsequently validate the correctness of the voting decision and the value itself, by inversely applying the constant program flow signature (B_E).

E. Residual Error Probability

After setting the coding schema, its parametrization is the second step. From a bird's eye view, any kind of code is simply based on the transformation of data (n bit) into code words ($n + k$ bits) by adding k bits of redundant information. The fault detection is subsequently based on the distance between valid code words. The chance for a SDC to mutate a valid code word into another valid word is thereby defined as the *residual error probability* [13]:

$$p_{sdc} = \frac{\text{valid code words} - 1}{\text{possible code words} - 1} = \frac{2^{\,n} - 1}{2^{\,n+k} - 1} \approx \frac{1}{2^{\,k}} \tag{3}$$

```
Require: Static Signatures: $B_x$, $B_y$, $B_z$
 1: function DECODE($v_c$, $A$, $B$)
 2:     if $v_c \bmod A \neq B$ then SIGNAL_DUE()
 3:     return $(v_c - B)$ div $A$                                ①
 4: end function
 5:
 6: function APPLY($v_c$, $sig_{dyn}$)
 7:     return $v_c + sig_{dyn}$                                   ②
 8: end function
 9:
10: function VOTE($x_c$, $y_c$, $z_c$)                             ③
11:     if $(x_c - y_c) = (B_x - B_y)$ then
12:         if $(x_c - z_c) = (B_x - B_z)$ then
13:             $win \leftarrow$ APPLY$(x_c, (x_c - y_c) + (x_c - z_c))$
14:             return $B_E \leftarrow (B_x - B_y) + (B_x - B_z)$
15:         else
16:             $win \leftarrow$ APPLY$(x_c, (x_c - y_c))$
17:             return $B_E \leftarrow (B_x - B_y)$
18:         end if
19:     else if $(y_c - z_c) = (B_y - B_z)$ then
20:         $win \leftarrow$ APPLY$(y_c, (y_c - z_c))$
21:         return $B_E \leftarrow (B_y - B_z)$
22:     else if $(x_c - z_c) = (B_x - B_z)$ then
23:         $win \leftarrow$ APPLY$(x_c, (x_c - z_c))$
24:         return $B_E \leftarrow (B_x - B_z)$
25:     else
26:         $win \leftarrow noDecision$
27:         SIGNAL_DUE()
28:     end if
29: end function
```

Fig. 3. The *CoRed* voter algorithm. It takes the encoded variants (v_x, v_y, v_z). The majority is found by encoded operations, leading to a unique *static signature* B_E. This signature is dynamically recomputed and stamped (APPLY) to the winner (win) before returning both as a verdict.

This estimation assumes that every kind of error and data corruption is equally probable – regardless of the number of bits flipped. In short, the more bits we spend for information redundancy, the lower p_{sdc}. In the case of arithmetic error coding, the number of possible code words is $A \cdot 2^n$, resulting in $p_{sdc} \approx {}^1/_A$. Therefore, the larger A, the lower the residual error probability. Figure 4 depicts the correlation between A and p_{sdc} (p_{pred} in the figure) for 16-bit numbers.

III. Think Binary

Due to our specific implementation with the dynamic signatures, the *CoRed* voter behaves like any other ANBD operation when seen from outside. Accordingly, we expected the residual error probability to be consistent with the coding theory. With the implementation of *CoRed* we ultimately aimed for a full single-bit soft-error fault coverage according to the software-level fault model from Section II-A. In this section we detail the challenges and pitfalls that arise from the transitions from plain math to a binary computing system.

Choosing Keys and Signatures

Our first step was to find suitable parameters for the EAN's keys and signatures. As mentioned before, it seems to be wise to choose a large number for A. Moreover, Forin [4] recommends prime numbers, mainly to reduce the number of possible factors between different coding streams and operations. Although the use of prime numbers seems intuitively plausible from a mathematical point of view, their particular advantage is left unsubstantiated. In fact, Schiffel [3] found some non-prime numbers to be equally suitable or even superior for parametrization – without giving good reasons. Thus, all of these considerations are of little help to find an A that reliably detects a certain number of bit flips.

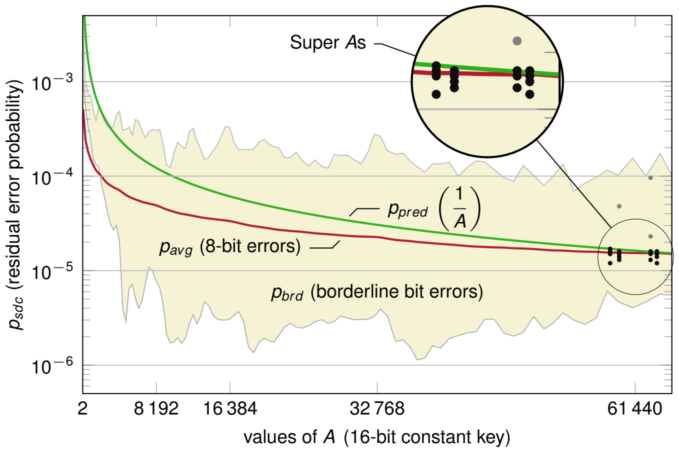


Fig. 4. p_{sdc} versus size of A, compared: p_{pred} plots the coding theory prediction. p_{avg} graphs the measured value for 8-bit errors, exemplary for average performance. p_{brd} illustrates variance of borderline bit errors that overstress the code by exactly 1-bit (2, 3 or 4-bit errors, depending on A).

From a binary point of view, in fact, the only purpose of A is to generate *robust* bit patterns. Consequently, it might seem obvious to use the *minimum Hamming distance* (d_H) between all possible code words as the measure of choice. We therefore computed the distances for all 32-bit codes. That is, for each and every 16-bit value v and key A. We found d_H to range from one to six depending on A, which means that zero to five-bit errors should be detectable by the respective codes. Interestingly, although we could observe the expected gain in distance with A growing, the values of d_H varied *significantly* between adjacent values. For example, the distance for non-prime 58 368 is two, that of prime 58 831 is three, and finally the minimum Hamming distance of 58 659 is *six* – being a non-prime value. The main reason is that ANBD codes are non-systematic codes [14], meaning that the assumed n data and k check bits are stored inseparable and processed together. Hence, the code's minimum distance is not necessarily related to the k bits used for representing A, but may vary according to the binary representation of $A \cdot v$. As a result, the *bigger the better is misleading* in this case.

To our surprise we found the results to be deviating from the literature. Schiffel [3] for example performed fault injection experiments for a small number of As, which indicate a residual error probability even for faults with a number of bits flips that is below d_H. These deviations in turn would in principle render the Hamming distance useless for selecting an appropriate A to reliably detect a certain number of bit flips. We decided to perform fault-simulation experiments to double check whether the fault-detection capabilities of the codes correspond to the computed minimal Hamming distances. These experiments covered each and every combination of possible vs, As and bit error patterns. Again we were surprised to actually observe the exact same deviations in p_{sdc} and silent data corruptions, which should have been detected according to the code's d_H.

We found the problem in the mapping of encoded values to their binary representation, for example 32-bit machine words. Of these, a practitioner usually assigns n bits to data and k additional bits to accommodate A, with $A \leq 2^k$ and $n+k \leq 32$. Choosing $A = 2^k$ is obviously unreasonable as this would lead to a simple bit shift. Selecting $A < 2^k$, however, results in an incomplete utilization of the machine word, which leaves a residue in terms of unused values. To put it simple, AN codes tend to result in odd value ranges that cannot be represented by exactly a power of two bits. The residue is again non-systematic and can neither be attributed to certain bit positions nor a specific number of bits. However, the coding theory is unaware of this mapping issue and assumes a self-contained code space and value range. Consequently, soft errors striking these unused bits can still lead to SDCs as the mutation may result in a valid but unused code word with $v > 2^n$.

Pitfall 1: Mapping Code to Binary

Our first pitfall is therefore the *mapping of code to binary space*: Due to the different and sometimes odd word sizes of plain and encoded data, dangerous over- and underflow conditions, coming to light only in the presence of soft errors, are not always obvious to the developer. This particular problem led to the observed discrepancies between d_H, predicted p_{sdc} and fault-simulation experiments. By adding a simple range check we were able to fix this issue, resulting in the following patch for the *CoRed* voter:

1

```
1: function DECODE($v_c$, $A$, $B$)
2:     if $v_c > v_{c,max}$ or $v_c \bmod A \neq B$ then
3:         SIGNAL_DUE()
4:     end if
5:     return $(v_c - B)$ div $A$
6: end function
```

With this improved implementation of DECODE, the fault-simulation results of EAN match with the fault-detection capabilities as expected by the minimal Hamming distance. On top of that, this simple check can prevent a huge loss of reliability to the code in general. For example, the predicted overall p_{sdc} is approximately 0.003 for $A = 251$. For 64-bit code words, Schiffel measured the double-bit error's $p_{sdc} \approx 0.013$, as we did in our first attempt. However, with the patch applied the actual p_{sdc} amounts to less than 0.000015 – a *factor of 1000*.

Consequently, d_H is a *valid decision criterion* for selecting A. The good news is that any A exhibits a sufficient distance for single-bit errors, except the aforementioned powers of two. As expected, the top performers reside in the upper end of the value range, irrespective of the ubiquitous variations. We termed the best of them, with a distance of six, *Super A*s[1] – non of them being prime. As expected, bit errors $< d_H$ are reliably detected by the EAN code. However, with bit errors $\geq d_H$ SDCs are still possible, and again we found A to have a significant influence on the actual p_{sdc}. We therefore evaluated the multi-bit error performance (up to 8 bits) for all 16-bit As as well. Although the resulting codes generally behaved as predicted, we identified the borderline bit errors to be dangerous. These overstress the code by one bit (exactly d_H errors) and induce huge variations in the resulting p_{sdc}. Figure 4 shows the predicted (p_{pred}) and the measured residual error probabilities versus the size of A. To keep the diagram readable, we simplified the plot by combining borderline bit errors in p_{brd} and hand-picking p_{avg} as a representative for the benign average case performance. As the actual borderline depends on the respective A's distance, p_{brd} fuses two to four-bit errors and graphs their range. In contrast, the average case, non-borderline error probability (p_{avg}) is near below the prediction and virtually free of scatter.

[1]Super As: 58 659, 59 665, 63 157, 63 859, and 63 877.

Alarmed by the partial exceeding of p_{pred} by borderline bit errors, we extended our experiments to cover *all* possible bit errors – due to the exponential growth in computation time, we limited these experiments to 16-bit codewords. Figure 5 illustrates the residual error probability distribution relative to the number of bits flipped. The three As shown here exemplify typical distribution patterns. For $A = 73$ the leading borderline (double-bit errors) p_{sdc} is orders of magnitude off the other values, whereas this is the case at the trailing edge (15-bit errors) for $A = 199$. Likewise both sides can be affected, as for $A = 239$. In all these malicious examples, p_{pred} is violated by the borderline outliers. Fortunately, we found 33 percent of all 8-bit As to behave benignly, staying below p_{pred} under all circumstances. Interestingly, this share varies significantly between non-prime (24.5 %) and prime (65.5 %) numbers. For the super As, we conducted the same experiments for up to 32-bit errors, as shown in Figure 4. They proved to be still very suitable as three of them behaved benignly and the other two being of trailing edge pattern (27 and 28-bit errors). The bottom line is, that one should consider the p_{sdc} distribution in addition to the Hamming distance for choosing A.

As the signatures B are appended additively, they do not directly interfere with the choice of A. To fulfil the intended purpose of detecting operator and addressing errors, they have to remain in force under arithmetic operations and bit errors. Besides the mandatory prerequisites, such as $B < A$, we applied the same evaluation process to the set of signatures used in the *CoRed* voter and selected them to match the code's capabilities. For a detailed description on the selection of the signatures and timestamps in general, we refer to Schiffel [3].

IV. Know Your Compiler & Architecture

In the previous section we showed that our improved EAN implementation can comply with coding theory by choosing appropriate parameters. In the following, we uncover further pitfalls that arise from the platform specifics and the transition from programming language to machine code.

A. Fault-Injection Experimental Setup with FAIL*

For a detailed reliability analysis of the assembly emitted by the compiler, we chose a practical approach: extensive fault injection (FI) campaigns. A FI *campaign* is a systematic series of experiments, which is defined by a fault *pattern*, a *location* (memory address or register) and a point in *time*. We focus on the generally accepted single-error single-bit assumption, as these amount to over 95 % of the overall soft-error rate [15], [16], [17]. Injecting all possible single-bit flips in all registers for every instruction (point in time) within the *CoRed* voter would result in a huge amount of experiments. Therefore, we relied on the generic FAIL* [18] FI framework. It offers sophisticated fault space pruning to economise the experiments to *effective* faults, without losing full fault-space coverage.

We used FAIL*'s IA32 backend, which is based on the Bochs emulator [19] and is the most mature one. The common workflow is to perform a correct and complete *golden run* of the program under test and to record each and every instruction and memory access. The instructions are disassembled to extract the register usage. Subsequently, the experiments are derived stepwise from the golden run: each instruction determines an injection time, whereas the registers define the locations. Each

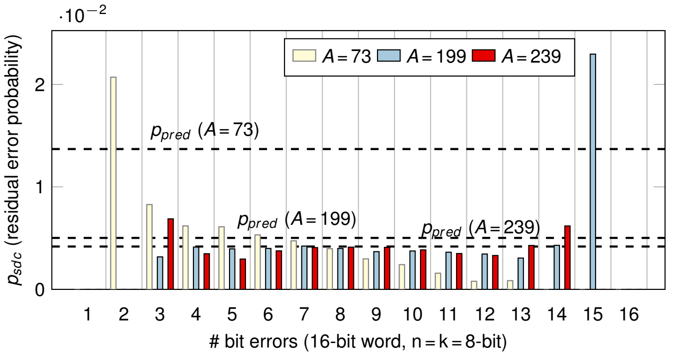


Fig. 5. A detailed view on the p_{sdc} distribution for multi-bit errors (16-bit code words). The selected As (all malicious in this example) illustrate typical borderline patterns: outliers located on the leading, trailing or on both sides.

individual experiment is stored in a database. FAIL* is then used to parallelise and execute the resulting campaigns and to consolidate the results within the database.

B. Voter Implementation, Environment, and Test Input

We compiled the *CoRed* voter as well as an unprotected (simple) voter for comparison to the IA32 architecture, using Debian Linux with GCC 4.7.2-5, optimization level `-O2`. Both voters are implemented in C++, with the EAN primitives being available as a C++ library. After compilation, the simple and the *CoRed* voter consist of 38 and 92 machine instructions respectively and occupy 112 (301) bytes of memory. Both voters call no other functions and operate only on their arguments.

We ensured that all memory loads and stores are done via registers[2], as we inject faults only in registers, instructions, and the program counter. Therefore, we manually restricted the IA32 instruction set to a subset that only works directly on registers. Hence, all values used for computation are visible in registers and the assumptions we used for pruning the FI experiments are valid. Additionally, we employed fine-grained spatial and temporal isolation provided by the operating system: jumps out of scope and illegal memory accesses are detected by an memory management unit (MMU). Likewise, deadlines are enforced by a watchdog. Isolation violations as well as further hardware traps, such as *invalid instruction*, are considered as detected errors, since they can be handled actively.

In order to achieve full branch coverage, we executed both voters with all possible equality sets (E) reflecting all combinations of agreeing and differing input values: $x{=}y{=}z$, $x{\neq}y{=}z$, $x{=}y{\neq}z$, $x{\neq}z{=}y$, and $x{\neq}y{\neq}z$. As we already verified the fault-detection capabilities of EAN codes for all possible input data (v) in Section III, we used a fixed set of inputs as part of the following instruction-level FI.

C. Acid Test: FI in Instructions and General Purpose Registers

As a first acid test, we injected errors in terms of single-bit flips into the voters' static instructions. Likewise, we ran experiments for all general purpose registers (GPRs). Table I shows the experimental results. Note that the total number of experiments differs due to the different sizes in code and data. The bottom line is, that the unprotected voter suffered significantly from dangerous SDCs, amounting to over 21 percent of all effective errors. In contrast, the *CoRed* voter

[2]This is by definition the case for RISC systems. For a CISC architecture, like IA32, this has to be ensured explicitly.

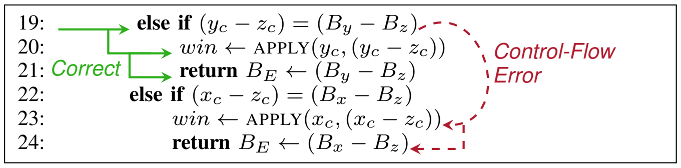
	Instructions		Registers and Flags		Program Counter		2-bit faults in General Purpose Registers	
	Simple	EAN	Simple	EAN	Simple	EAN	Super $A = 58\,659$	Bad $A = 58\,368$
Ok (No influence)	784	2772	1040	3204	127	267	38 639	38 639
Detected (EAN)	–	995	–	1435	–	420	21 596	21 519
Detected (HW-Trap)	93	246	8	41	21	241	47	47
Detected (Jump outside .text)	149	208	173	559	2614	5614	471	471
Detected (Inv. Mem. Acc.)	676	1626	1652	3177	190	626	59 967	59 967
Detected (Timeout)	0	1	0	0	0	0	0	0
Undetected (SDC)	450	0	807	0	152	0	0	77
$\sum$	2152	5848	3680	8416	3104	7168	120 720	120 720

TABLE I. Results of the voter fault-injection campaigns. The entire 1-bit fault space is covered by injecting instructions, registers, and program counter. Double-bit errors show the effect of an adversely chosen A.

performed as expected with more than 17 percent of the errors caught by the EAN code. Although these experiments already cover the vast majority of possible fault locations, further experiments are necessary to achieve full fault-model coverage.

D. CPU status: The flags register

The flags register, holding the CPU status and the arithmetic flags is just one more register but yielded more surprising results: The *CoRed* voter accesses the flags only for comparisons and conditional jumps, and in 12 experiments it failed silently. The reason is that as most pipelined instruction set architectures (ISAs), IA32 lacks a compound test-and-branch instruction. Instead, this conditional control-flow decision is separated (e.g., **`cmp eax, edx`** and **`je`** `Lequal`). A test instruction that sets arithmetic flags, followed by an conditional jump using these flags. In this short example two registers (`eax`, `ebx`) are compared, and the *zero flag* is set on equality. The jump-equal instruction redirects the control flow to the label `Lequal` if the zero flag is set. However, between those two instructions the information about the control-flow change *is stored only in a single bit* in the flags register. This is a striking example for the impact of the last transition step, leading to an unexpected single point of failure and SDCs. The following listing exemplifies the effects on the voting algorithm:


```
19:         else if $(y_c - z_c) = (B_y - B_z)$
20:             $win \leftarrow$ APPLY$(y_c, (y_c - z_c))$
21:             return $B_E \leftarrow (B_y - B_z)$
22:         else if $(x_c - z_c) = (B_x - B_z)$
23:             $win \leftarrow$ APPLY$(x_c, (x_c - z_c))$
24:             return $B_E \leftarrow (B_x - B_z)$
```

In this case, two unequal but correctly encoded values are compared. Hence, the voter should take the `else`-branch. However, when the zero flag is altered right in between comparison and branch, an incorrect winner is selected.

Pitfall 2: Inter-Instruction State

When $x \neq z$ but a dynamic signature $(x_c - z_c)$ is calculated because of *corrupted inter-instruction state*, the dynamic signature is off by a multiple of A and subsequently $> A$. In consequence, a somewhat valid but too large signature is generated. After identifying the cause of the problem, the remedy is straightforward: The APPLY function is replaced by a version with an additional range check that verifies if the dynamic signature is absolutely smaller than A.

2

```
function APPLY($v_c$, $sig_{dyn}$)
    if $\|sig_{dyn}\| \geq A$ then SIGNAL_DUE()
    end if
    return $v_c + sig_{dyn}$
end function
```

E. Corrupting the Program Counter

Extending the FI to the *program counter* (PC), we observed another unexpected and extremely rare phenomenon, observable in only three corner-case experiments. Injecting single-bit flips in the PC resulted in random jumps by an offset of power of two, leading to an incorrect control flow. Most of those jumps are detected by the MMU-based isolation. However, intra-function jumps still remain undetected and may lead to SDCs.

We subsequently injected only effective faults in the PC that do not trigger MMU exceptions. By investigating the resulting SDCs, we found *zombie* values in registers as the surprising cause for the errors. The parameters, three correctly encoded variants, are computed and stored on the program stack by the caller. However, these values also remain in registers by coincidence. The corner-cases manifested themselves as control flow deviations leading from the very beginning of VOTE to one of the exit sequences. As the registers still hold the valid but incorrect zombie values at this point, a decodable but malicious result is calculated.

Pitfall 3: Undefined Execution Environment

The problem arises from our *assumption of a clean execution environment* for our voter, where *components are properly isolated* from each other. The lack of CPU state isolation between voter caller and the voter function (the compiler lazily leaves non-live values in GPRs) leaks correctly encoded values to the voter's exit sequence. We eradicated all SDCs by clearing the local storage, in our case only the registers, before starting the voting sequence:

3

```
function VOTE($x_c$, $y_c$, $z_c$)
    ZERO_LOCAL_STORAGE()
    $win = 0$
    if $(x_c - y_c) = (B_x - B_y)$ then
…
```

The *Program Counter* column in Table I contains the results for the FI experiments. As already mentioned, most errors are detected by the MMU. However, the simple voter again suffered from almost five percent of SDCs, whereas the *CoRed* voter remains SDC free and detects 420 faults in the decoding phase.

V. Tighten the Rules – Multi-bit Faults

So far, we focused on single-bit FI experiments (cf. Section II-A, fault model). However, multi-bit flips may be an issue and should be detectable by EAN codes that exhibit an appropriate d_H anyway. With FAIL* at hand, we were able to gain an insight into the domain of multi-bit faults. In the following, we briefly present and discuss our first results from

Multibit Faults in GPR ($A = 58\,659$)	3 bits	4 bits	5 bits
Ok (No influence)	33.742 %	33.605 %	33.544 %
Detected (EAN)	18.209 %	18.356 %	18.431 %
Detected (HW-Trap)	0.001 %	<0.001 %	0 %
Detected (Outside .text)	0.054 %	0.009 %	0.001 %
Detected (Inv. Mem. Acc.)	47.993 %	48.030 %	48.023 %
Detected (Timeout)	0 %	0 %	0 %
Undetected (SDC)	0	0	0
$\sum$	579 838	627 886	1.3×10^6
Size of fault space	3.59×10^6	1.03×10^8	2.90×10^9
Covered fault space	16.13 %	0.59 %	0.04 %

TABLE II. MULTI-BIT ERROR FAULT-INJECTION RESULTS.

campaigns injecting 2 to 5-bit faults. While the semantics of a single-bit flip is relatively easy to describe (there is only one injection time, one injection location, and a single injection pattern) this is not as straightforward for multi-bit faults. For a first examination we therefore restricted our experiments by only varying the injection pattern and flipping a number of bits.

With double-bit faults we were able to examine the influence of a poorly chosen A, backing the expected behaviour on assembly level. We evaluated *all* 120 720 combinations of double-bit flips for the *CoRed* voter (14 16-bit register operations, 240 32-bit operations). As predicted (cf. Section III), using an appropriate A, the voter was able to detect all double-bit faults (*Super* A – Table I). In contrast, using a poorly chosen A with a Hamming distance of two resulted in 77 SDCs (*Bad* A – Table I). Hence, the choice of A had the expected impact on the residual error probability.

For 3–5 bit faults (Table II), we only performed random (Monte-Carlo) experiments to cope with the exploding fault space. In our first campaigns, we managed to cover 16 percent of the 3-bit fault space, with significantly lower percentages for 4 and 5-bit faults. As expected, we could not see any SDCs, although these statistical results do not prove the absence in turn. In future work, we plan to extend and improve the fault model to even cover multi-bit multi-errors, that is multiple patterns, locations and points in time.

VI. DISCUSSION

Even with proper coding theory at hand, bullet-proof software-based fault tolerance is hard to implement – as our detailed investigation of *CoRed*'s EAN and voter implementation revealed. Although arithmetic coding schemes are mathematically sound and well understood, the transition to real hardware is highly technical. It may weaken the abstract model to a point where the fault-mitigation capability suffers and does not meet the predicted residual error probability. So, from a practitioner's perspective, it is vital to grasp the transition and take the implementation platform into account.

A. Lessons Learned

Section III detailed the impact of binary code representation and encoding parameters on the residual error probability. The first pitfall and its solution – adding a range check – might appear trivial and caused by sloppy software development in the first place. However, the consideration of soft errors in a traditional development process and quality assurance is not necessarily given. Developers cannot find much support by programming languages, compilers or other static analysis tools, as these are usually are not aware of or even consider unreliable execution. Furthermore, at least for non-systematic codes, the mapping between theory and binary representation is non-trivial. We were especially surprised by the variations in d_H as well as the specific multi-bit p_{sdc} distribution. Consequently, rules of thumb, as the generally recommended prime numbers, are not necessarily applicable on the binary level. Overall, in-depth system knowledge and thorough evaluation of the encoding parameters are essential but worth the effort – as we showed with the decrease in p_{sdc} for $A = 251$ by a factor of 1000.

The second pitfall takes the same line as its predecessor but illustrates another problem: control-flow errors. Here, the processor architecture and its inter-instruction states are a dangerous and little obvious source of vulnerabilities. Pitfall 3 falls into the same category but is caused by either the laziness of the compiler or the incompleteness of the aspired spatial isolation. That SDCs may arise from the way the compiler assigns registers, is probably one of the most obscure error pattern. The solution is to provide a clean execution environment and perfect isolation. We therefore advocate for the system software as the right layer for implementing fault-tolerance techniques.

Furthermore, Section IV showed the general importance of sophisticated fault-injection tools. Pitfalls 2 and 3 were discovered only by systematic fault-injection experiments conducted with the FAIL* framework. Pitfall 2 manifested itself in less than 20 out of over 8000 experiments. Therefore, Monte-Carlo experiments lack a trustworthy statement on the absence of SDCs. FAIL* enabled us to systematically cover the *entire* fault space. Furthermore, as it is based on emulation, we were able to incorporate flag register and program-counter injections that were impossible with the hardware–debugger-based framework we used in the original *CoRed* evaluation. Here, tooling speed is crucial for short measure-improve cycles and for the iterative evaluation of software-based fault-tolerance. For example, the GPR experiments improved from two weeks with the debugger to 15 minutes using FAIL* on an off-the-shelf desktop computer.

B. Threats to Validity

Threats to validity arise mainly from the fault-model assumptions and the characteristics of the IA32 architecture used for our fault-injection experiments.

The single-fault assumption is certainly the most important one we made for the fault model. This means, for a certain period (experiment), faults occur only once and are limited to a single data word. However, this is only valid for RISC architectures where instructions and operands are separated (fixed length instructions). Otherwise, an operator error could lead to an totally different alignment of the operands. This would in turn manifest in an unknown number of bit flips and potentially cause multi-word faults. From this point of view, the IA32 seems to be a very bad choice as it neither features fixed length instructions nor separated instruction and data memory. As the current version of FAIL* only provides a full feature set for the IA32 architecture, we had to circumvent this issue by restricting the compiler to a RISC-equivalent instruction subset. We therefore consider the general fault model still valid.

Another simplification was to omit most of the input parameter space (v) in the fault-injection campaigns. In our opinion, it is sufficient to cover the entire input parameter space for v by means of simulation experiments (see Section III) as validation for data errors. As the data and control flow errors can be seen independently, according to the aforementioned single-fault assumption, we can focus on control-flow errors within the fault-injection experiments.

We are aware of the fact that our experimental approach represents a platform-specific evaluation rather than a general validation. However, we believe it to be absolutely accurate with respect to the assumed fault model. Certainly, our results depend on the specific architecture and compiler employed and have to be reasserted when switching the platform. Although we cannot exclude further pitfalls caused by architectural specifics, the guidelines presented in this paper can significantly help to simplify this task. We identified a tight feedback loop with fault-injection experiments as a suitable tool for accompanying this process. The decoupling of a small system part – in our case the *CoRed* voter – by spatial and temporal isolation allowed us to put the implementation to the acid. For a fault-tolerant system it is crucial to utilize isolation techniques provided by the system software to limit unpredictable side-effects.

VII. Conclusion and Outlook

Arithmetic error coding schemes (AN codes) are a mathematically sound and well understood technique to effectively mitigate soft errors. However, even with proper coding theory at hand, software-based fault tolerance is hard to implement – and even more difficult to verify. In reality, the resulting residual error probability can deviate from theory by orders of magnitude. On the example of the *CoRed* voter, we investigated the roots of these deviations from a systems perspective – and found them in implementation glitches introduced in *every* stage of the transition from coding theory to the machine code for a specific architecture. By exhaustive simulation and fault-injection experiments (100 % fault space coverage for single and double-bit faults), we identified typical pitfalls that, if addressed, result in the reliable detection of *all* errors and an implementation that matches the predictions from theory. Accordingly, developers can systematically improve reliability up to the elimination of all software-visible errors. With *CoRed* pushed to the coding theory's prediction, we finally achieved our initial goal to provide reliable, hardware-independent and selective protection of safety-critical applications, which can be even applied to existing systems.

From the experiences made, we see both supportive fault-aware tooling *as well as* fault-tolerant system software as essential weapons in the battle against soft errors. We envision *CoRed* [1] and FAIL* [18] to be part of an integrated approach for the development and the verification of safety-critical systems. One step in this direction is the advancement of *CoRed*'s concepts to the *dOSEK* [20] dependable operating system. Extending FAIL*'s functionality (e.g., multi-bit fault injection) and devising methods to deal with the resulting fault-space explosion is another one.

The bottom line of this paper is: When implementing error coding schemes, it is vital to *also* take a systems perspective. Know your system and validate each step in the transition from abstract concepts to binary code. *Bullet-proof software-based fault tolerance is possible.*

References

[1] P. Ulbrich, M. Hoffmann, R. Kapitza, D. Lohmann, W. Schröder-Preikschat, and R. Schmid, "Eliminating single points of failure in software-based redundancy," in *9th Eur. Dep. Computing Conf. (EDCC '12)*. Washington, DC, USA: IEEE, May 2012, pp. 49–60.

[2] P. Ulbrich, R. Kapitza, C. Harkort, R. Schmid, and W. Schröder-Preikschat, "I4Copter: An adaptable and modular quadrotor platform," in *26th ACM Symp. on Applied Computing (SAC '11)*. New York, NY, USA: ACM, 2011, pp. 380–396.

[3] U. Schiffel, "Hardware error detection using AN-codes," Ph.D. dissertation, Technische Universität Dresden, Fakultät Informatik, 2011.

[4] P. Forin, "Vital coded microprocessor principles and application for various transit systems." in *Symp. on Control, Computers, Communication in Transportation (CCCT '89)*, Sep. 1989, pp. 79–84.

[5] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante, *Software-Implemented Hardware Fault Tolerance*. Heidelberg, Germany: Springer, 2006.

[6] W. W. Peterson and E. J. Weldon, *Error-correcting codes*, 2nd ed. Cambridge, MA, USA: MIT Press, 1972.

[7] A. Avižienis, G. Gilley, F. P. Mathur, D. Rennels, J. Rohr, and D. Rubin, "The star (self-testing and repairing) computer: An investigation of the theory and practice of fault-tolerant computer design," *IEEE Transactions on Computers*, vol. 20, no. 11, pp. 1312–1321, 1971.

[8] N. Oh, S. Mitra, and E. McCluskey, "Ed4i: Error detection by diverse data and duplicated instructions," *IEEE Transactions on Computers*, vol. 51, no. 2, pp. 180–199, 2002.

[9] J. Chang, G. Reis, and D. August, "Automatic instruction-level software-only recovery," in *36th Int. Conf. on Dep. Systems & Networks (DSN '06)*. Washington, DC, USA: IEEE, 2006, pp. 83–92.

[10] U. Wappler and C. Fetzer, "Software encoded processing: Building dependable systems with commodity hardware," in *26th Int. Conf. on Comp. Safety, Reliability, and Security (SAFECOMP '07)*, F. Saglietti and N. Oster, Eds. Heidelberg, Germany: Springer, 2007, pp. 356–369.

[11] C. Fetzer, U. Schiffel, and M. Süßkraut, "AN-encoding compiler: Building safety-critical systems with commodity hardware," in *28th Int. Conf. on Comp. Safety, Reliability, and Security (SAFECOMP '09)*, B. Buth, G. Rabe, and T. Seyfarht, Eds. Heidelberg, Germany: Springer, 2009.

[12] U. Schiffel, A. Schmitt, M. Süßkraut, and C. Fetzer, "ANB- and ANBDmem-encoding: detecting hardware errors in software," in *29th Int. Conf. on Comp. Safety, Reliability, and Security (SAFECOMP '10)*, E. Schoitsch, Ed. Heidelberg, Germany: Springer, 2010, pp. 169–182.

[13] R. A. Frohwerk, "Signature analysis: A new digital field service method," *Hewlett-Packard Journal*, vol. 28, no. 9, pp. 2–8, 1977.

[14] T. R. N. Rao, *Error Coding for Arithmetic Processors*, 1st ed. Orlando, FL, USA: Academic Press, 1974.

[15] X. Li, K. Shen, M. C. Huang, and L. Chu, "A memory soft error measurement on production systems," in *2007 USENIX ATC*. Berkeley, CA, USA: USENIX, 2007, pp. 1–14.

[16] J. Maiz, S. Hareland, K. Zhang, and P. Armstrong, "Characterization of multi-bit soft error events in advanced SRAMs," in *Intern. Electron Devices Meeting (IEDM '03)*. New York, NY, USA: IEEE Press, 2003.

[17] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, "Ferrari: A flexible software-based fault and error injection system," *IEEE Transactions on Computers*, vol. 44, pp. 248–260, 1995.

[18] H. Schirmeier, M. Hoffmann, R. Kapitza, D. Lohmann, and O. Spinczyk, "FAIL*: Towards a versatile fault-injection experiment framework," in *25th Intern. Conf. on Architecture of Comp. Systems*, ser. Lecture Notes in Informatics, vol. 200. Gesellschaft für Informatik, Mar. 2012.

[19] K. P. Lawton, "Bochs: A portable PC emulator for Unix/X," *Linux Journal*, vol. 1996, no. 29es, p. 7, 1996.

[20] M. Hoffmann, C. Dietrich, and D. Lohmann, "dOSEK: A dependable RTOS for automotive applications," in *19th Int. Symp. on Dependable Computing (PRDC '19)*. Washington, DC, USA: IEEE, Dec. 2013, fast abstract, to appear.

Implementation and further experimental results:

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