Noname manuscript No.
(will be inserted by the editor)

Experiences with Software-based Soft-Error Mitigation Using AN-Codes

Martin Hoffmann · Peter Ulbrich · Christian Dietrich · Horst Schirmeier · Daniel Lohmann · Wolfgang Schröder–Preikschat

the date of receipt and acceptance should be inserted later

Abstract Arithmetic error coding schemes are a well known and effective technique for soft error mitigation. Although the underlying *coding theory* is generally a complex area of mathematics, its practical implementation is comparatively simple in general. However, compliance with the theory can be lost easily while moving towards an actual implementation, which finally jeopardizes the aspired fault-tolerance characteristics and effectiveness. In this paper, we present our experiences and lessons learned from implementing arithmetic error coding schemes (AN codes) in the context of our *Combined Redundancy* fault-tolerance approach. 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. This allowed us to eliminate all remaining silent data corruptions in the *Combined Redundancy* framework, which we validated by an extensive fault-injection campaign covering the entire fault space of 1-bit and 2-bit errors.

Keywords Fault Injection · Arithmetic Code · Dependability

1 Introduction

Recent developments in hardware design for embedded systems offer more performance and parallelism. This comes with shrinking structure sizes and operating voltages and thus for the price of being less reliable. Consequently, soft-error mitigation is one of the major challenges for safety-critical applications and systems [4]. Besides adding

M. Hoffmann · P. Ulbrich · C. Dietrich · D. Lohmann · W. Schröder–Preikschat
Chair of Distributed Systems and Operating Systems
Friedrich–Alexander University Erlangen–Nuremberg, 91058 Erlangen, Germany
E-mail: {hoffmann,ulbrich,dietrich,lohmann,wosch}@cs.fau.de

Horst Schirmeier
Department of Computer Science 12
Technische Universität Dortmund, 44221 Dortmund, Germany
E-mail: horst.schirmeier@tu-dortmund.de

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. Arithmetic error coding offers a high degree of effectiveness and is relatively easy to implement. At the same time, its robustness in terms of residual error probability can be easily determined according to the underlying coding theory. Consequently, we used AN codes as the foundation of our *Combined Redundancy* (*CoRed*) approach [32] (cf. Section 2.3). However, practical experiments uncovered *substantial* discrepancies between theoretical and actual effectiveness.

1.1 Problem Statement

Any divergence from the assumed residual error probability complicates the safety assessment and unnecessarily impedes the general applicability of our approach. Therefore, our goal with *CoRed* 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 inconsistencies with the code's assumed behaviour and 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 [27] 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* stage of the transition from the coding theory to the machine-dependent implementation (Figure 1). The identified issues include misconceptions regarding binary number representation and ranges, silent assumptions about the runtime environment, and specific characteristics of the hardware platform. Although the *semantics* of the coding 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.

1.2 About this paper

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 reevaluation of *CoRed*'s encoded majority voter, as its most vital component, 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 *Extended*

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

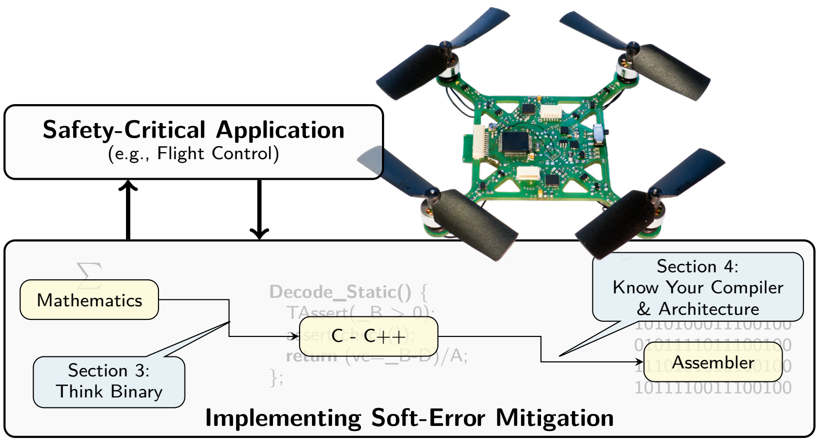

Fig. 1 From theory to application: Implementing arithmetic error coding is attended by various pitfalls and challenges.

AN code (EAN) as well as a voter implementation, which overcome the transformation issues and preserve the intended execution behavior. At large, the error coding an thereby *CoRed* itself 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.

This article is an extension of our paper *"A practitioner's guide to software-based soft-error mitigation using AN-codes"* [15], presented at the 15th IEEE International Symposium on High Assurance Systems Engineering (HASE 2014). The additional content comprises considerations of the arithmetic distance versus the Hamming distance as a measure of code quality (Section 3.1), and more details on selection strategies for the signature B. Sections 4 and 5 are extended by concrete examples of the encountered pitfalls and additional experimental results comparing the theoretical residual error probability with the experimental fault-injection results. Section 6 additionally discusses continuous fault-injection as a design concept for safety-critical systems. Finally, Section 7 revisits the restrictions and the generalizability of our approach in more detail.

1.3 Contributions

The key contributions of this paper are a binary-aware analysis of the AN code parametrization and its fault-detection capabilities. Based on these considerations, we propose means for the selection of highly robust code parameters. Here, we not only concentrate on theoretical limits, but also consider the behavior of the code on hardware level.

We further disclose typical pitfalls on the transition from coding theory to machine-level instructions. Learned from these lessons, we provide an improved *Extended AN* (EAN) code and *CoRed* voter implementation, which fully complies with the coding theory. An experimental verification and fault injection of the *CoRed* voter, covering the *entire* 1 & 2-bit fault space, shows the robustness of the presented approach.

2 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.

2.1 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* [10]. 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 [12]. 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. The basic fault hypothesis of our evaluation is the generally accepted single-error single-bit assumption, as these amount to over 95 % of the overall soft-error rate [18, 19, 16]. We further extended this fault model to single-error *multi-bit* faults.

2.2 Arithmetic Error Coding

Common error codes, such as parity or cyclic redundancy checks, are widely used in communication and memory applications [24]. 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 flavors of arithmetic error coding schemes which differ mostly in their fault-model coverage: Residue codes [2] 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 [23, 6], 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 [34]. Finally, Forin [10] proposed *timestamps* (D) to cover outdated operands as the last gap in the code's fault coverage, forming the ANBD coding scheme. The encoding is defined by the standard AN and the BD fraction, as:

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

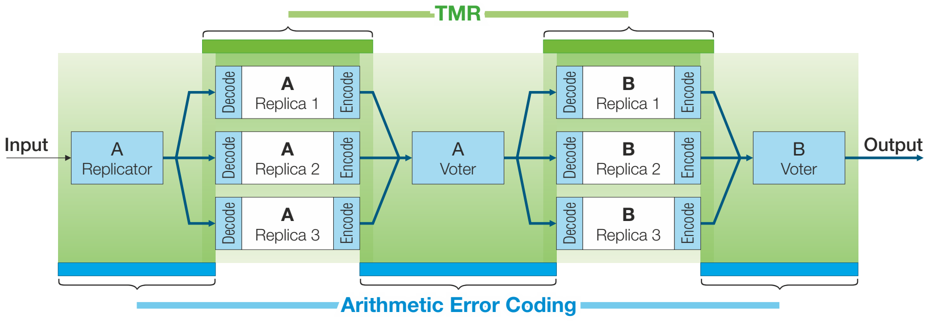

Fig. 2 As its name implies, *CoRed* combines TMR with arithmetic error coding. Thereby, input replication and voting are excluded from the reliable computing base and explicitly hardened against soft error effects. By their perfect interplay, both techniques form a consistent sphere of redundancy and provide effective fault-detection throughout the processing chain.

Initially designed for hardware-level protection [10], ANBD codes are also employed in software-based approaches. Here, they commonly safeguard entire applications and systems. *Software Encoded Processing* (SEP) [34] is an interpreter-based solution, transforming safety-critical applications at runtime. The resulting coverage is limited by the necessity of a dependable checker unit and due to inherent complexity of the runtime environment. Moreover, the runtime overhead is increased by several orders of magnitude (up to a factor of 10^5). Schiffel et al. evolved their concept to CEP [28], a compiler-based solution, which effectively reduces the induced overhead to a factor of 10^3 in the worst case. Nevertheless, the approach still requires a checker instance and cannot cover the entire execution path leading to inevitable SDCs [27]. In both cases, the remaining SDCs can be accounted to the approaches' complexity and implementation, which aims for generic application protection, rather to a conceptual flaw of the employed ANBD codes.

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

In contrast to the aforementioned encoding approaches, we were able to fully utilise the coverage and effectiveness provided by arithmetic error coding by reducing complexity and pursuing a tailored, application-specific concept. With *CoRed* [32] we presented a highly effective, software-based fault-tolerance approach for mixed-criticality control applications. *CoRed* engages in between application and operating system and combines proven *triple modular redundancy* (TMR) with arithmetic error coding.

The idea behind this combination is to utilise the effectiveness and efficiency of TMR for protecting the application. This, however exposes dangerous *single points of failure* (SPOFs), caused by gaps between the processing replicas and by the necessary replication infrastructure itself. These parts are usually attributed to the *reliable computing base* (RCB) [9], which is considered to be error-free. However, this does not pass the reality check for software-based approaches, as the residual error rates of respective techniques [1, 26, 6, 30] shows. To overcome this weakness and to eliminate the remaining SPOFs, *CoRed* incorporates arithmetic error coding to safeguard the replication infrastructure in the sense of the aspired RCB as shown in Figure 2. For that, we especially introduced the *CoRed dependable voter* (*CoRed* voter): a high-reliability voting schema based on our EAN error coding implementation (see Section 2.3.1).

By limiting the encoding to the generic replication infrastructure, we were able to put the implementation to the acid. We successfully employed *CoRed* in the mission-critical flight control of the *I4Copter* quad rotor UAV [33], 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 safety-critical control applications.

2.3.1 The CoRed EAN Code Implementation

To avoid confusion with other implementations, we labeled 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:

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

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). The timestamp D is identical in all encoded values at any point in time, but may change globally, for example between different voting steps.

2.3.2 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 omit the timestamp D throughout the rest of this paper, and pretend it to be part of the variable-specific signature B_v.

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 at any point in the process. 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 14, 17, 21, and 24), defined by:

$$\texttt{CoRed\_vote}(x_c, y_c, z_c) = \underbrace{\left\{\begin{array}{lcll} x_c + B_x + & \overbrace{(B_x - B_y) + (B_x - B_z)}^{\text{Dynamically Calculated Signatures}} & \textit{if} & \overbrace{\{x_c, y_c, z_c\}}^{\text{Equality Set}} \\ x_c + B_x + & (B_x - B_y) & \textit{if} & \{x_c, y_c\} \\ x_c + B_x + & (B_x - B_z) & \textit{if} & \{x_c, z_c\} \\ y_c + B_y + & (B_y - B_z) & \textit{if} & \{y_c, z_c\} \\ & \texttt{signal\_due()} & \textit{if} & \{\} \end{array}\right.}_{\text{Return Value: } win_c}$$

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 13, 16, 20, and 23) 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).

It is of special interest how the signatures are accessed by the *CoRed* voter. A general implementation would operate on encoded variables, where the static signature is carried along in memory. But the strength of our implementation arises from static knowledge about the signatures at compile time. We use the compiler to instantiate the algorithm with the three constant static signatures. This allows the compiler to do constant folding and to put operands into ROM, which is considered more robust against faults. Especially, all possible values of B_E are compile-time constants. The compiler is also instructed to avoid actual calls to functions by forced inlining. In doing so, we avoid indirect jumps on memory addresses when the control-flow returns from a function.

2.4 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 bits) 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* [11]:

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

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 6 depicts the correlation between A and p_{sdc} (p_{pred} in the figure) for 16-bit numbers.

```
Require: Static Signatures: $B_x$, $B_y$, $B_z$
function DECODE($v_c$, $A$, $B$)
    if $v_c \bmod A \neq B$ then SIGNAL_DUE()
    return $(v_c - B)$ div $A$                                  ①
end function

function APPLY($v_c$, $sig_{dyn}$)
    return $v_c + sig_{dyn}$                                    ②
end function

function VOTE($x_c$, $y_c$, $z_c$)                              ③
    if $(x_c - y_c) = (B_x - B_y)$ then
        if $(x_c - z_c) = (B_x - B_z)$ then
            $win \leftarrow$ APPLY$(x_c, (x_c - y_c) + (x_c - z_c))$
            return $B_E \leftarrow (B_x - B_y) + (B_x - B_z)$
        else
            $win \leftarrow$ APPLY$(x_c, (x_c - y_c))$
            return $B_E \leftarrow (B_x - B_y)$
        end if
    else if $(y_c - z_c) = (B_y - B_z)$ then
        $win \leftarrow$ APPLY$(y_c, (y_c - z_c))$
        return $B_E \leftarrow (B_y - B_z)$
    else if $(x_c - z_c) = (B_x - B_z)$ then
        $win \leftarrow$ APPLY$(x_c, (x_c - z_c))$
        return $B_E \leftarrow (B_x - B_z)$
    else
        $win \leftarrow noDecision$
        SIGNAL_DUE()
    end if
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.

3 Think Binary

Due to our specific implementation with 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 2.1. In this section we detail the challenges and pitfalls that arise from the transitions from plain math to a binary computing system.

3.1 Arithmetic and Hamming Distance

The robustness of any coding scheme against errors can be quantified by the minimal number of errors that have to occur to alter one valid code word into another valid word. In classic coding theory the *Hamming distance* [13] is commonly used to quantify this robustness. For two code words the Hamming distance is number of bits that differ in the binary representation. The *minimal Hamming distance* for a whole code is the minimum distance between all pairs of distinct code words. For this robustness metric, a *single bit-flip in the binary representation* of the code word is considered an error, and if the number of bit-flips stays below the minimal Hamming distance, detection always succeeds.

In the literature [25,22,31,5] the *arithmetic* distance [24] is often used to reason about the robustness of AN-codes. This metric was designed to reflect the errors that occur in an arithmetic unit, more precisely a binary adder. The binary addition is modeled as a sequence of 1-bit adders that transform two argument bits and one input-carry bit into an output bit and an output-carry, which is propagated into the next 1-bit adder. For the arithmetic distance metric, an error is a *wrongly propagated carry bit*, resulting in deviations of $\pm 2^j$. The arithmetic distance of two code words is the minimal number of additions or subtractions of powers-of-two that is needed.

The following equations exemplify the Hamming distance (hd) and the arithmetic distance (ad) for two code words. While three bit-flips are needed to transform 8 into 6, only one wrongly propagated carry is required.

$$hd(8_{10}, 6_{10}) = hd(1000_2, 0110_2) = 3 \tag{3}$$
$$ad(8_{10}, 6_{10}) = ad(6_{10} + 2^1, 6_{10}) = 1 \tag{4}$$

It can easily be shown that the number of required carry-errors is always *less or equal* to the number of required bit-flips [20,21]. Hence, the arithmetic distance is a pessimistic estimation for the quality of a code, since it includes an error model for an arithmetic unit.

From a practitioner's point of view there are two problems that make the arithmetic distance less desirable for quantifying the robustness of an AN-coded system. The arithmetic distance models the inner working of an simple *ripple-carry adder*. But in modern processors many different adding circuits and optimizations are used (e.g., carry-select, carry-lookahead, etc.) to cut down pipeline delays.

The second problem with the arithmetic distance is the implicit error propagation chain that is assumed by modeling an arithmetic unit (see Figure 4). The metric gives the number of carry-propagation errors that have to occur in the binary adder (1) to get a SDC when the value is used. Contrary, the Hamming distance gives the number of bits that can be corrupted at the point the value is used (2) without getting a SDC. In a real system, where arithmetic operations are mixed with other operations (control-flow, logical operations, rotations) and long delays, this error propagation chain is unknown and can, in the common case, not be reduced to a single arithmetic operation.

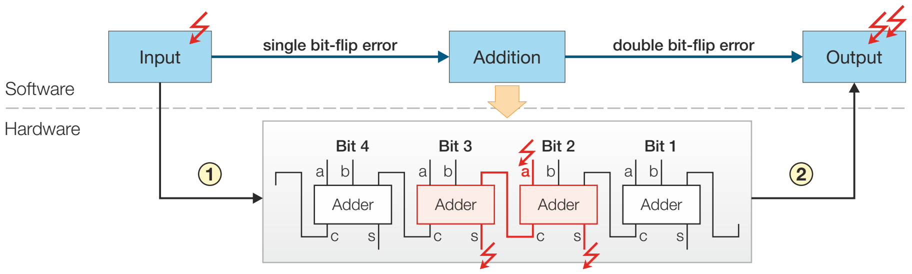

Fig. 4 Error propagation chain: The arithmetic distance states the robustness at (1), while the Hamming distance states the code robustness at (2).

Therefore, when choosing a good AN-code, we try to maximize the Hamming distance in order to tolerate as many bit-flips as possible, no matter where they origin from.

3.2 Choosing Keys and Signatures

In the process of implementing the *CoRed* voter, our first step was to find suitable parameters for the EAN's keys (A) and signatures (B). As mentioned before, it seems to be wise to choose a large number for A. Moreover, Forin [10] 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 [27] 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.

Hamming Distance as Selection Criterion

As argued in Section 3.1 the only purpose of A, from a binary point of view, is to generate *robust* bit patterns. Consequently, we choose the *minimum Hamming distance* (d_H) between all possible code words as the measure of choice. To find suitable As, we computed the distances for all 32-bit codes, i.e. for all possible 16-bit values v and keys 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 $A = 58\,368$ is two, that of prime $58\,831$ is three, and finally the minimum Hamming distance of $58\,659$ is *six*, even being a non-prime value. The main reason is that ANBD codes are non-systematic codes [25], 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.

Unexpected Deviations

To our surprise we found the results to be deviating from the literature. Schiffel [27], 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 would 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 $n + k \leq 32$.

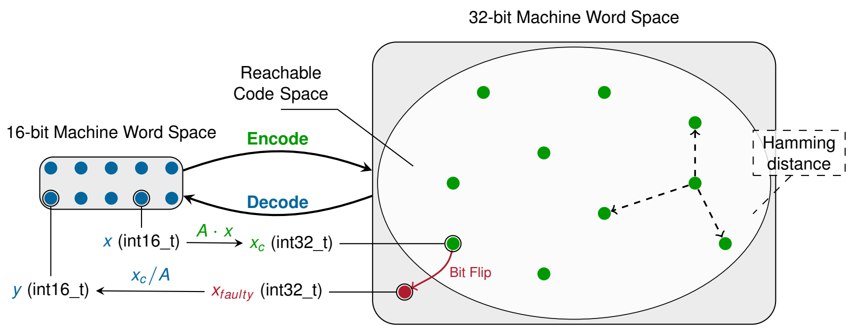

Fig. 5 The AN encoding of a plain 16-bit value leads to a 32-bit code word. Account must be taken on the fact that the resulting possible code cannot utilize the entire 32-bit space. The figure also shows a bit flip that shifts an encoded value out of the reachable code space. As further described in *Pitfall 1* this may still lead to a decodable but wrong plain value.

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, as exemplified in Figure 5. 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 $v > 2^n$. Our first pitfall is therefore the mapping of code to binary space:

Pitfall 1: Mapping Code to Binary

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: 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
```
1

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 almost 1000*.

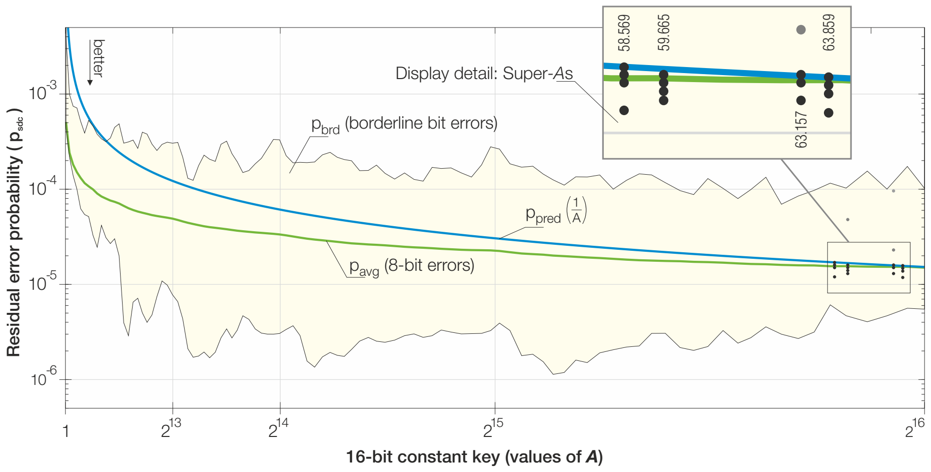

Fig. 6 p_{sdc} versus size of A, compared: p_{pred} plots the coding theory prediction. p_{avg} graphs the measured value for 8-bit errors, exemplary for average performance. p_{brd} illustrates variance of borderline bit errors that overstress the code by exactly 1-bit (2, 3 or 4-bit errors, depending on A).

Determining Highly Robust As

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], none 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 6 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.

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 code words. Figure 7 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

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

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 6. 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: One should consider the p_{sdc} distribution in addition to the Hamming distance for choosing A.

Selection Strategies for the Signature B

As the signatures B are appended additively, they do not directly interfere with the choice of A. A bit-flip results in a residue R, when the the value is decoded. The residue can be interpreted as a wrong signature $B'_v = B_v + R$ for a correctly encoded value. Since signatures were introduced to detect operator and address errors, it is crucial to choose signature values carefully, such that two signatures cannot be easily confused. Therefore, to keep the distinguishing property of signatures intact, we choose the set of used signatures according to the Hamming distance. All used signatures in the system should have the maximal distance, while not violating the mandatory prerequisites, such as $0 < B < A$.

From a practitioner's point of view it is not always possible to keep the number of used signatures in the whole system low enough to get an acceptable minimal Hamming distance among the Bs. In such cases different components may use the same signatures for different variables, under the assumption that no encoded value may flow from one component to the other one. This can be ensured with the help of proper temporal and spatial isolation by the OS, employing memory protection mechanisms and execution time monitoring.

Regarding the signature choice of the *CoRed* voter component, we performed the following steps: First, we have to identify all terms that are used as signatures: three input signatures (B_x, B_y, B_z) and four dependent signatures used for the equality sets. Then, we choose the input signatures such that all signatures, including the depended ones, have a pairwise maximal distance and are within the range $0 < B < A$. For the Super As, the choice $B_x = 29\,868$, $B_y = 23\,835$, $B_z = 12\,658$ results in a minimal Hamming distance of six for all signatures[2].

4 Know Your Compiler & Architecture

In the previous section we have shown 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.

4.1 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

[2] Based on a brute-force search. Code available at `http://www4.cs.fau.de/Research/CoRed`

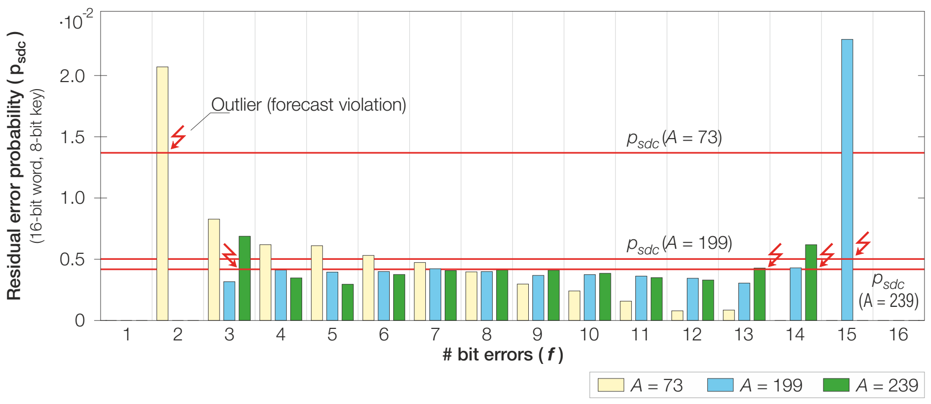

Fig. 7 A detailed view on the p_{sdc} distribution for multi-bit errors (16-bit code words). The selected As (all malicious in this example) illustrate typical borderline patterns: outliers located on the leading, trailing or on both sides.

systematic series of experiments, which is defined by a fault *pattern*, a *location* (memory address or register) and a point in *time*. As mentioned earlier, we focus on the generally accepted single-error single-bit assumption. Nevertheless, injecting all possible single-bit flips in all registers for every instruction (point in time) within the *CoRed* voter would still result in a huge amount of experiments. Therefore, we relied on the generic FAIL* [29] FI framework. It offers sophisticated fault space pruning to economize the experiments to *effective* faults, without losing full fault-space coverage.

We used FAIL*'s IA32 backend, which is based on the Bochs emulator [17] 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 individual experiment is stored in a database. FAIL* is then used to parallelize and execute the resulting campaigns, and to consolidate the results within the database.

4.2 Voter Implementation, Environment, and Test Input

We compiled the *CoRed* voter as well as an unprotected (simple) voter 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[3], 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

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

		Registers, Flags		Instructions		Program Counter	
		Simple	CoRed	Simple	CoRed	Simple	CoRed
Benign defects		1040	3204	784	2772	127	267
Detected	CoRed (Code)	–	1435	–	995	–	420
	HW-Trap	8	41	93	246	21	241
	Outside `.text`	173	559	149	208	2614	5614
	Invalid Memory	1652	3177	676	1626	190	626
	Timeout	0	0	0	1	0	0
Undetected (SDC)		**807** ↯	0	**450** ↯	0	**152** ↯	0
	$\sum$	3680	8416	2152	5848	3104	7168

Table 1 Results of the voter fault-injection campaigns. The entire 1-bit fault space is covered by injecting instructions, registers, and program counter.

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 a *memory protection unit* (MPU). 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 3.2, we used a fixed set of inputs as part of the following instruction-level FI.

4.3 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 1 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 performed as expected with more than 17 percent of the errors caught by the EAN code.

Injected faults in the *CoRed* variant are either contained by the hardware protection (trap, MPU), or by a result value that contains a detectable error. This also includes detected erroneous jumps outside the program code, arising from corrupted base or stack pointer registers (`EBP`, `ESP`) that deviate the control flow when the function returns. We conclude that the IA32 machine-code version of the *CoRed* voter conforms to the implementation regarding to GPR faults. However, beyond operand and operator errors, there are more possible fault locations, necessitating further fault experiments.

4.4 CPU status: The flags register

The flags register, holding the CPU status and the arithmetic flags, 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 IA32 lacks a compound test-and-branch instruction, as most pipelined *instruction set architectures* (ISAs). Instead, this conditional control-flow decision is separated into a test instruction that sets arithmetic flags, followed by an conditional jump using these flags (e.g., `cmp eax, edx` and `je Lequal`). 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:

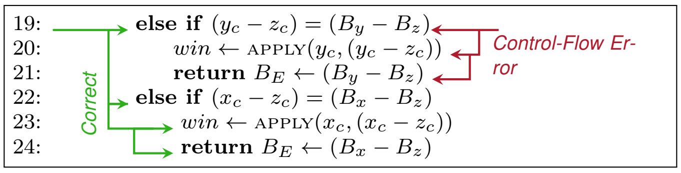
```
else if $(y_c - z_c) = (B_y - B_z)$
    $win \leftarrow$ APPLY$(y_c, (y_c - z_c))$
    return $B_E \leftarrow (B_y - B_z)$
else if $(x_c - z_c) = (B_x - B_z)$
    $win \leftarrow$ APPLY$(x_c, (x_c - z_c))$
    return $B_E \leftarrow (B_x - B_z)$
```

Fig. 8 Example of a control-flow error affecting the *CoRed* voter execution.

In this case, two unequal but *correctly encoded* values are compared. Hence, the voter should take the `else if`-branch. However, when the zero flag is altered right in between comparison and branch, an incorrect winner is selected. This results in the application of the dynamic signature $(y_c - z_c)$, but since the values are unequal the valid range for signatures $(0 < B < A)$ is left. And as both values are correctly encoded, the range is left by a multiple of A.

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: Since the comparison flag cannot be encoded, we cannot prevent the erroneous control flow. But the APPLY function can be replaced by a version with an additional range check that verifies if the dynamic signature is still in the range of valid signatures:

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

To illustrate this pitfall, consider the following, concrete example. Assuming the system was built with[4] $A = 199$, $B_x = 81$, $B_y = 35$, and $B_z = 13$, we invoke the *CoRed* voter with $(x_c, y_c, z_c) = (479, 632, 411)$, the encoded values corresponding to $(x, y, z) = (2, 3, 2)$. Since $y \neq z$ $[(y_c - z_c) \neq (B_y - B_z)]$, the branch in line 19 (Figure 8) should not be taken. But, due to the faulty flags register, it *is* taken, resulting in a control-flow error: In line 20 a winner is calculated by applying $(y_c - z_c) = 221$ $[= 199 + (B_y - B_z)]$ as a signature to y_c. Without the signature overflow check in APPLY, the winner is $y_c + 221 = 853$, which is a correctly encoded result, but decodes to 4 $[4 \cdot A + (B_y + (B_y - B_z)) = 853]$ – a SDC. With the signature range check in place, APPLY detects that 221 is larger than $B_{max} = 81$ and signals an error.

4.5 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 results in random jumps by an offset of powers of two, leading to an incorrect control flow. Most of those jumps are detected by the MPU-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 MPU exceptions. By investigating the resulting SDCs, we found *zombie* values in registers as the surprising cause for the errors. The parameters, three correctly encoded values, 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:

```
1: function VOTE($x_c$, $y_c$, $z_c$)                3
2:     ZERO_LOCAL_STORAGE()
3:     $win = 0$
4:     if $(x_c - y_c) = (B_x - B_y)$ then
5: ...
```

The *Program Counter* column in Table 1 contains the results for the FI experiments. As already mentioned, most errors are detected by the MPU. 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.

[4] The signatures were chosen by the methods discussed in Section 3.2 and have a pairwise minimal Hamming distance of six.

		Double-bit Errors in General Purpose Registers	
		Super $A = 58\,659$	Bad $A = 58\,368$
Benign Defects		38 639	38 639
Detected	CoRed (Code)	21 596	21 519
	HW-Trap	47	47
	Outside `.text`	471	471
	Invalid Memory	59 967	59 967
	Timeout	0	0
Undetected (SDC)		0	**77** ϟ
	$\sum$	120 720	120 720

Table 2 Double-bit errors show the effect of an adversely chosen A. Two bits were flipped in one GPR at one point in time. The bad A has a d_H of 2, while the super A has a d_H of 6.

5 Tighten the Rules – Multi-bit Faults

So far, we focused on single-bit FI experiments (cf. Section 2.1, fault model). However, multi-bit flips may become an issue in future hardware designs [8] 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 results from 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. As *single-event multiple-bit upsets* (MBUs) – i.e., multiple bits flipping at a *single* point in time – are expected to be much more probable than multiple single-bit flips spread over time [8], we 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 behavior on the machine-code 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 3.2), using an appropriate A, the voter was able to detect all double-bit faults (*Super A* – Table 2). In contrast, using a poorly chosen A with a Hamming distance of two resulted in 77 SDCs (*Bad A* – Table 2). Hence, the choice of A had the expected impact on the residual error probability.

We further investigated the connection between the *residual error probability* of an EAN code, as discussed in Section 3.2, and the concrete *CoRed* voter implementation (Table 3). We therefore carried out additional complete fault injections of the 2-bit faults with varying As. By choosing As with a minimal Hamming distance of two,

$A =$	22	44	1202	2404	12288	34346	58368
d_H	2	2	2	2	2	2	2
p_{sdc}(2-bit)	0.02749	0.02749	0.00029	0.00029	0.14247	0.00009	0.01633
FI(SDC, 2-bit)	138	143	4	4	591	4	77

Table 3 Detailed comparison of SDCs with varying A and $N = 65535$ for 2-bit faults. The examined EAN codes have a minimal Hamming distance (d_H) of 2. The simulated residual error probability (p_{sdc}) correlates to the measured SDC increase in the fault injection experiments ($\sigma = 0.9986$).

Multi-bit Register Faults ($A = 58\,659$)		**3 Bit**	**4 Bit**	**5 Bit**
Benign Defects		33.742 %	33.605 %	33.544 %
Detected	*CoRed* (Code)	18.209 %	18.356 %	18.431 %
	HW-Trap	0.001 %	<0.001 %	0 %
	Outside `.text`	0.054 %	0.009 %	0.001 %
	Invalid Memory	47.993 %	48.030 %	48.023 %
	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
Fault Space Coverage		16.13 %	0.59 %	0.04 %

Table 4 Multi-bit error fault-injection results. Multiple faults were injected into a GPR at one point in time. The immensely large fault space was probed by Monte-Carlo. We still cannot observe SDCs.

double-bit flips actually result in SDCs, as expected. The residual error probability (p_{sdc}) is the robustness of a single code word, while the fault injection results – FI(SDC, 2-bit) in Table 3 – incorporate the influence of the whole *CoRed* voter algorithm and its concrete instantiation on the hardware platform. Also the fault injections were carried out with a small set of input values, while the residual error probability was calculated for all possible code words. Nevertheless, as shown in Table 3, we measure a close correlation of $\sigma = 0.9986$. We conclude that the increase of SDCs stems only from the badly chosen A and the more vulnerable intermediate values used in the voter operation.

For 3–5 bit faults (Table 4), we only performed random (Monte-Carlo) experiments to cope with the exploding fault space. 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 their absence. 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.

6 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.

6.1 Lessons Learned

Section 3 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 to be caused by sloppy software development in the first place. However, the consideration of soft errors in a traditional development and quality assurance process is not necessarily given. Developers cannot find much support by utilizing programming languages, compilers or other static analysis tools, as these are usually 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. The third pitfall falls into the same category but is caused by either the compiler's unawareness of soft errors, 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 patterns. 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 4 showed the general importance of sophisticated fault-injection tools. Both the second and the third pitfall were discovered only by systematic fault-injection experiments conducted with the FAIL* framework. The second pitfall 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.

6.2 Continuous Fault Injection for Dependable Software Development

Failures of software components in the presence of transient faults are often unforeseeable and not intuitive by just looking at the code. Here, we identified a tight feedback loop with fault-injection experiments as a suitable tool for accompanying the process of dependable software development. Fault injections should be done on a regular basis like unit tests. In fact, unit tests can be used as benchmarks for the fault injection, since they are supposed to cover the entire code and important corner cases.

Clearly, such repetitive fault injection campaigns consume time and computing power, especially when the fault model is extended to multi-bit faults. Therefore it is desirable to keep the components that are part of the RCB small and testable, as it is already good practice in software engineering. In our case, the decoupling of a small system part – the *CoRed* voter – by spatial and temporal isolation allowed us to put the implementation to the acid.

Furthermore, the fault-injection loop should also be carried out on the same hardware architecture that is used afterwards for the component. We have seen in the presented pitfalls that the occurrence of SDCs is tightly coupled with the concrete architecture. Architectural specifics and how the compiler exploits the processor features is highly relevant for the reliability of a component.

It is worth to be noted that all architecture-independent concepts and experimental findings presented in Section 3 can be entirely reused without further effort. The remaining architecture-dependent tests rather concentrate on fault propagation and unexpected (or even unforeseeable) side effects. These evaluations highly benefit from our automated toolchain around the FAIL* framework.

7 Threats to Validity

We are aware of the fact that threats to validity arise from various sources. In this section, we face our experimental results with the most common issues from appropriateness to generalisability.

7.1 Construction Validity

We based the appropriateness of our experimental evaluation on a multi-stage approach: first, to simulate mutations of codewords for each and every set of coding parameters. Second, to inject fault patterns into the actual implementation on the ISA-level of the hardware platform.

Subsequently, we directly computed the general residual error probability from the fault-simulation, as all possible bit patterns are covered by these experiments. In contrast, neither probabilistic results can be easily derived from fault-injection experiments, nor can these results be generalised. Hence, we used them to demonstrate the absence of SDCs in a concrete *system under test* (SUT) – an inference that cannot be backed by the simulation.

7.2 Overall Consistency and Reliability

Both, the fault-simulation as well as the fault-injection experiments proved to be fully deterministic and reproducible. Considering the aforementioned pitfalls, the resulting measurements are consistent in terms of residual error probability and SDCs. Moreover, mathematical prediction as well as fault-simulation and fault-injection experiments seem to be in line and support each other, as the multi-bit fault-injection in Section 5 indicates.

7.3 Conclusion Validity

Soft-error mitigation and looking for the needle in the haystack share the same problems: a tiny needle (defect) and too much hay (fault-space). Drawing the right conclusions therefore requires a very large number of experiments as well as reasonable assumptions to limit the search space.

7.3.1 Restricted Fault Model

The single-fault assumption is certainly the most important one we made for the fault model. This means, for a certain period (e.g., voter execution), faults occur only once and are limited to a single data word. Generally, the single-fault assumption is largely covered by the rare occurrence of soft errors and the available data [16,18,19]. However, this is only valid for RISC architectures where instructions and operands are separated (fixed length instructions). Otherwise, an operator error could lead to a 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 architecture 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.

7.3.2 Fault-Space Coverage

A restricted fault-space coverage poses a significant threat to conclusion validity. Often, it is not possible to cover the entire fault space due to its dramatic growth. Researchers tend to circumvent this issue by means of statistical approximations, as for example using Monte-Carlo experiments. With the help of sheer computing power and FAIL*'s sophisticated fault-space pruning techniques, we were able to cover the entire fault space for 1 and 2-bit faults. Regarding the fault simulation we were even able to explore up to 16-bit faults. In both cases, we actually discovered rare corner cases, some of which completely distorted our previously made assumptions. The code-space violation in Figure 5 as well as the silent assumptions made by the run-time environment (as described in Pitfall 3) are striking examples.

7.3.3 Limited Input Parameter Space

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 3) 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.

The comparison of multi-bit fault injections with the simulated residual error probability for 2-bit faults (see Table 3) is an additional indicator that this simplification is sufficient. While p_{sdc} was calculated with all possible encoded values, the fault injection was done only with fixed input parameters[5]. Nevertheless, both values are highly correlated.

7.4 Internal Validity

Clearly, the accuracy of such simulator-based fault injections highly depend on the underlying system models and are often limited to the ISA level. Cho et al. [7] criticise

[5] Five input parameter sets for the four equality sets, and one for `signal_due()`.

this inaccuracy in favour of injections into sub-ISA–level hardware models leading to more subtle fault effects. Unfortunately, it is not a viable option to resort to such gate-level injections for dependability evaluations of larger software components. So, for a more realistic fault model, it might be interesting to investigate the general effects of low-level faults (e.g., affecting the memory hierarchy, timing details, pipelining, or out-of-order execution), and especially how they appear at the ISA level. Subsequently, sufficiently simple – yet detailed enough – high-level fault models should be synthesizable. Nevertheless, Cho et al. [7] also state that *"[. . .] accuracy is not necessarily a requirement. For example, an inaccurate error injection technique can be very useful as long as it is effective in driving the correct design decisions for building robust systems."* Even if a single-bit flip at the gate-level results in an unknown number of bit flips at ISA level, we think selecting the most effective and benign keys is a reasonable way to resolve this issue. Although we cannot exclude further pitfalls caused by architectural specifics, the guidelines presented in this paper can significantly help to simplify this task.

7.5 External Validity

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. For RISC-based SPARC or ARM architectures, this can be achieved with the help of recent FAIL* extensions interfacing the gem5 simulator [3].

Our findings from Section 3 are independent from the underlying architectural specifics and can be applied for future applications without any restrictions. The selection strategies for the signatures B and the proposed 16-bit *Super A*s, providing a high minimal Hamming distance, are not bound to any special hardware architecture. Nevertheless, finding similar *Super A*s for 32-bit or even 64-bit words might not be viable with the applied brute-force approach, which already took advantage of a high performance computing cluster.

The generalization of our approach with respect to the systematic fault-injection (Section 4) is clearly limited by the complexity of the application to be hardened. The recommended full fault-coverage hardly scales with oversized self-contained software components. On the other hand, especially in the safety-critical application domain, a maintainable, well-structured and therefore (unit-) testable design is mandatory, and also mitigating the scalability problem.

8 Conclusion

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* [32] and FAIL* [29] 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* [14] 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 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.

Acknowledgements

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 1719/1-2 and SP 968/5-2.

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

References

1. Aidemark, J., Vinter, J., Folkesson, P., Karlsson, J.: Experimental evaluation of time-redundant execution for a brake-by-wire application. In: 32nd Int. Conf. on Dep. Systems & Networks (DSN '02), pp. 210–215 (2002). DOI 10.1109/DSN.2002.1028902
2. Avižienis, A., Gilley, G., Mathur, F.P., Rennels, D., Rohr, J., Rubin, D.: The star (self-testing and repairing) computer: An investigation of the theory and practice of fault-tolerant computer design. IEEE Transactions on Computers **20**(11), 1312–1321 (1971). DOI 10.1109/T-C.1971.223133
3. Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., Saidi, A., Basu, A., Hestness, J., Hower, D.R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M.D., Wood, D.A.: The gem5 simulator. SIGARCH Comput. Archit. News **39**(2), 1–7 (2011). DOI 10.1145/2024716.2024718
4. Borkar, S.Y.: Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. IEEE Micro **25**(6), 10–16 (2005)
5. Braun, J., Geyer, D., Mottok, J.: Alternative measure for safety-related software. ATZelektronik worldwide **7**(4), 40–43 (2012). DOI 10.1365/s38314-012-0106-1
6. Chang, J., Reis, G., August, D.: Automatic instruction-level software-only recovery. In: 36th Int. Conf. on Dep. Systems & Networks (DSN '06), pp. 83–92. IEEE, Washington, DC, USA (2006). DOI 10.1109/DSN.2006.15

7. Cho, H., Mirkhani, S., Cher, C.Y., Abraham, J., Mitra, S.: Quantitative evaluation of soft error injection techniques for robust system design. In: Proceedings of the 50th annual Design Automation Conference, pp. 1–10 (2013)
8. Dodd, P.E., Massengill, L.W.: Basic mechanisms and modeling of single-event upset in digital microelectronics. IEEE Transactions on Nuclear Science **50**(3), 583–602 (2003). DOI 10.1109/TNS.2003.813129
9. Engel, M., Döbel, B.: The reliable computing base: A paradigm for software-based reliability. In: 1st Int. W'shop on Softw.-Based Methods for Robust Emb. Sys. (SOBRES '12), LNCS. Gesellschaft für Informatik (2012)
10. Forin, P.: Vital coded microprocessor principles and application for various transit systems. In: Symp. on Control, Computers, Communication in Transportation (CCCT '89), pp. 79–84 (1989)
11. Frohwerk, R.A.: Signature analysis: A new digital field service method. Hewlett-Packard Journal **28**(9), 2–8 (1977)
12. Goloubeva, O., Rebaudengo, M., Reorda, M.S., Violante, M.: Software-Implemented Hardware Fault Tolerance, 1 edn. Springer, New York, NY, USA (2006)
13. Hamming, R.W.: Error detecting and error correcting codes. Bell System technical journal **29**(2), 147–160 (1950)
14. Hoffmann, M., Dietrich, C., Lohmann, D.: dOSEK: A dependable RTOS for automotive applications. In: 19th Int. Symp. on Dependable Computing (PRDC '13). IEEE, Washington, DC, USA (2013). DOI 10.1109/PRDC.2013.22. URL `http://www.danceos.org/publications/PRDC-FAST-2013-Hoffmann.pdf`. Fast abstract
15. Hoffmann, M., Ulbrich, P., Dietrich, C., Schirmeier, H., Lohmann, D., Schröder-Preikschat, W.: A practitioner's guide to software-based soft-error mitigation using AN-codes. In: 15th IEEE Int. Symp. on High-Assurance Systems Engineering (HASE '14), pp. 33–40. IEEE, Miami, Florida, USA (2014). DOI 10.1109/HASE.2014.14
16. Kanawati, G.A., Kanawati, N.A., Abraham, J.A.: Ferrari: A flexible software-based fault and error injection system. IEEE Transactions on Computers **44**, 248–260 (1995)
17. Lawton, K.P.: Bochs: A portable PC emulator for Unix/X. Linux Journal **1996**(29es), 7 (1996)
18. Li, X., Shen, K., Huang, M.C., Chu, L.: A memory soft error measurement on production systems. In: 2007 USENIX ATC, pp. 1–14. USENIX, Berkeley, CA, USA (2007)
19. Maiz, J., Hareland, S., Zhang, K., Armstrong, P.: Characterization of multi-bit soft error events in advanced SRAMs. In: Intern. Electron Devices Meeting (IEDM '03). IEEE Press, New York, NY, USA (2003). DOI 10.1109/IEDM.2003.1269335
20. Mandelbaum, D.: Arithmetic codes with large distance. IEEE Transactions on Information Theory **13**(2), 237–242 (1967). DOI 10.1109/TIT.1967.1054015
21. Massey, J.L.: Survey of residue coding for arithmetic errors. International Computation Center Bulletin **3**(4), 3–17 (1964)
22. Medwed, M., Schmidt, J.M.: Coding schemes for arithmetic and logic operations - how robust are they? In: H. Youm, M. Yung (eds.) Information Security Applications, *Lecture Notes in Computer Science*, vol. 5932, pp. 51–65. Springer, Heidelberg (2009). DOI 10.1007/978-3-642-10838-9_5
23. Oh, N., Mitra, S., McCluskey, E.: Ed4i: Error detection by diverse data and duplicated instructions. IEEE Transactions on Computers **51**(2), 180–199 (2002). DOI 10.1109/12.980007
24. Peterson, W.W., Weldon, E.J.: Error-correcting codes, 2 edn. MIT Press, Cambridge, MA, USA (1972)
25. Rao, T.R.N.: Error Coding for Arithmetic Processors, 1 edn. Academic Press, Orlando, FL, USA (1974)
26. Reis, G., Chang, J., Vachharajani, N., Rangan, R., August, D., Mukherjee, S.: Software-controlled fault tolerance. ACM Transactions on Architecture and Code Optimization (TACO '05) **2**(4), 366–396 (2005). DOI 10.1145/1113841.1113843
27. Schiffel, U.: Hardware error detection using AN-codes. Ph.D. thesis, Technische Universität Dresden, Fakultät Informatik (2011)
28. Schiffel, U., Schmitt, A., Süßkraut, M., Fetzer, C.: ANB- and ANBDmem-encoding: detecting hardware errors in software. In: E. Schoitsch (ed.) 29th Int. Conf. on Comp. Safety, Reliability, and Security (SAFECOMP '10), pp. 169–182. Springer, Heidelberg, Germany (2010). DOI 10.1007/978-3-642-15651-9_13
29. Schirmeier, H., Hoffmann, M., Kapitza, R., Lohmann, D., Spinczyk, O.: FAIL*: Towards a versatile fault-injection experiment framework. In: 25th Intern. Conf. on Architecture of Comp. Systems, *Lecture Notes in Informatics*, vol. 200. Gesellschaft für Informatik (2012)

30. Shye, A., Moseley, T., Reddi, V.J., Blomstedt, J., Connors, D.A.: Using process-level redundancy to exploit multiple cores for transient fault tolerance. In: 37th Int. Conf. on Dep. Systems & Networks (DSN '07), pp. 297–306. IEEE, Washington, DC, USA (2007). DOI 10.1109/DSN.2007.98
31. Steindl, M., Mottok, J., Meier, H.: Ses-based framework for fault-tolerant systems. In: Proceedings of the 8th Workshop on Intelligent Solutions in Embedded Systems (WISES '10), pp. 12–16 (2010). DOI 10.1109/WISES.2010.5548427
32. Ulbrich, P., Hoffmann, M., Kapitza, R., Lohmann, D., Schröder-Preikschat, W., Schmid, R.: Eliminating single points of failure in software-based redundancy. In: 9th Eur. Dep. Computing Conf. (EDCC '12), pp. 49–60. IEEE, Washington, DC, USA (2012). DOI 10.1109/EDCC.2012.21
33. Ulbrich, P., Kapitza, R., Harkort, C., Schmid, R., Schröder-Preikschat, W.: I4Copter: An adaptable and modular quadrotor platform. In: 26th ACM Symp. on Applied Computing (SAC '11), pp. 380–396. ACM, New York, NY, USA (2011)
34. Wappler, U., Fetzer, C.: Software encoded processing: Building dependable systems with commodity hardware. In: F. Saglietti, N. Oster (eds.) 26th Int. Conf. on Comp. Safety, Reliability, and Security (SAFECOMP '07), pp. 356–369. Springer, Heidelberg, Germany (2007). DOI 10.1007/978-3-540-75101-4_34