Demystifying Soft-Error Mitigation by Control-Flow Checking – A New Perspective on its Effectiveness

SIMON SCHUSTER, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)
PETER ULBRICH, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)
ISABELLA STILKERICH, Schaeffler Technologies AG & Co. KG
CHRISTIAN DIETRICH, Leibniz Universität Hannover (LUH)
WOLFGANG SCHRÖDER-PREIKSCHAT, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)

Soft errors are a challenging and urging problem in the domain of safety-critical embedded systems. For decades, checking schemes have been investigated and improved to mitigate soft-error effects for the class of control-flow faults, with current industrial standards strongly recommending their use.

However, reality looks different: Taking a systems perspective, we implemented four representative Control-Flow Checking (CFC) schemes and put them through their paces in 396 fault-injection campaigns. In contrast to previous work, which typically relied on probability-based vulnerability metrics, we accounted for the influence of memory and time overheads on the fault-space dimensions and applied those in full-scan fault injections. This change in procedure alone severely degraded the perceived effectiveness of CFC.

In addition, we expanded the perspective to data-flow faults and their influence on the overall susceptibility, an aspect that so far has been largely ignored. Our results suggest that, without accompanying measures, any improvement regarding control-flow faults is dominated by the increase in data faults caused by the increased attack surface in terms of memory and runtime overhead. Moreover, CFC performance less depended on the detection capabilities than on general aspects of the concrete binary compilation and execution.

In conclusion, incorporating CFC is not as straightforward as often assumed and the vulnerability of systems with hardened control-flow may in many cases even be increased by the schemes themselves.

CCS Concepts: • **Hardware** → **Transient errors and upsets**; • **Software and its engineering** → **Software reliability**; **Software fault tolerance**; *Empirical software validation*; Translator writing systems and compiler generators; • **General and reference** → *Reliability*; *Empirical studies*;

ACM Reference format:
Simon Schuster, Peter Ulbrich, Isabella Stilkerich, Christian Dietrich, and Wolfgang Schröder-Preikschat. 2017. Demystifying Soft-Error Mitigation by Control-Flow Checking – A New Perspective on its Effectiveness. *ACM Trans. Embedd. Comput. Syst.* 1, 1, Article 1 (July 2017), 19 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

This article was presented in the International Conference on Embedded Software 2017 and appears as part of the ESWEEK-TECS special issue.

This work is supported by the German Research Foundation (DFG) under grants no. SCHR 603/9-2, the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR89, Project C1), the Bavarian Ministry of State for Economics under grant no. 0704/883 25 (EU EFRE funds) and the German Federal Ministry of Education and Research with the funding ID 01IS16025 (ARAMiS II).

Authors' addresses: Simon Schuster: schuster@cs.fau.de; Peter Ulbrich: ulbrich@cs.fau.de; Isabella Stilkerich: isabella@stilkerich.eu; Christian Dietrich: dietrich@sra.uni-hannover.de; Wolfgang Schröder-Preikschat: wosch@cs.fau.de
.

1 INTRODUCTION

Current hardware designs for embedded systems offer more performance and parallelism. Still, this development is associated with shrinking structure sizes and operating voltages and thus comes at the price of being less reliable. As a consequence, mitigation of soft errors (syn.: transient hardware faults, single event upsets) is one of the major challenges [5] for safety-critical applications and systems. In general, soft errors are induced by ionizing particles that affect and corrupt data stored in SRAM memories as well as computation logics without permanently impairing the functionality [4]. Besides adding costly hardware redundancy, virtually sacrificing the technology gain, software-based fault-tolerance offers a selective and thus potentially resource-efficient alternative.

To tackle this problem, as systems engineers[1] we focus on software approaches that either engage with the language or the instruction level and which furthermore can be applied automatically to a given application as an independent development step.

In general, soft errors can manifest in both data or control-flow faults (i.e., a change in the intended program execution). For the domain of control-flow faults, several *control-flow checking* (CFC) schemes strive to provide a fine-grained, orthogonal hardening of individual control flows, with industry standards such as the automotive ISO 26262 [18] safety standard recommending their use.

1.1 Problem Statement

Recommendation or not, at the end of the day, the only thing that matters is the overall system reliability. Unfortunately, this is a complex issue that reaches well beyond a single perspective, such as theoretical detection capabilities or specific design considerations. Instead, there are more influencing factors to it, such as the hardware platform, toolchain, operating system, application, configurations, and so on. Yet, such a whole-system perspective seems to be underrepresented in related literature. One reason might be that judging CFC effectiveness in this context is rather difficult as an easy to compute metric—such as the Hamming distance [16] for data faults—is missing. In many evaluations, either fault coverage or probability-based vulnerability metrics are used for quantifying the effectiveness.

However, recent studies [12, 27] suggest that these contemporary evaluation approaches are overly simplistic as they ignore, among other things, the increase in the attack surface through software measures. For the domain of control-flow faults a structured, quantitative effect-analysis is overdue that accounts for attack surface increase, hardware properties as well as other fault types, such as data faults.

1.2 Our Contribution

In this paper, we put CFC fault-detection capabilities as well as the commonly used probabilistic reliability metric to the test. We claim the following key contributions to improve our insights on language- and instruction-level CFC efficacy:

- Integration of four representative CFC schemes with various configurations into the KESO compiler framework. Reevaluation in four application use cases, including a real-world UAV flight-control [37], amounting to 396 *fault injection* (FI) campaigns. Reproduction and validation of previous reliability experiments based on the residual failure metric.
- Novel quantitative whole-system evaluation based on full-scan experiments that incorporate time and memory overheads. Extension of the conventional control-flow-centric fault model to also cover data faults as an important factor.

[1]That is *(real-time) operating system* and *systems-software* engineers.

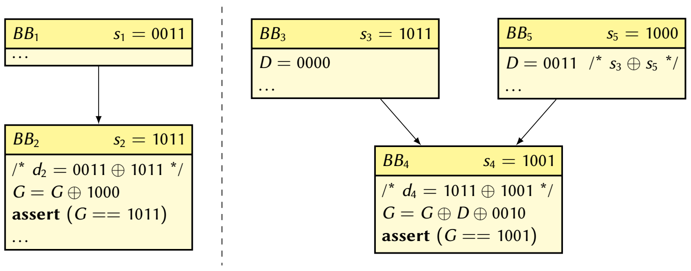


Fig. 1. Signature-based CFC by the example of CFCSS for single and a multiple predecessors on the left and right, respectively.

- Disclosure and discussion of consistent qualitative effects, deficiencies and flaws of CFC schemes and their evaluation.

Starting with a background section on CFC schemes, our contributions outline the coarse structure of the remaining paper.

2 BACKGROUND

Before immersing into the actual study, we give a brief overview of CFC techniques and their underlying mechanics: In the last decades, substantial work has been done on those techniques, resulting in numerous variants. First of all, there are combined data- and control-flow integrity schemes such as arithmetic encoding [11, 26], which tackle the problem implicitly by mapping it onto the data integrity domain. In this paper, we focus on explicit encoding and verification of the control-flow by signature-based CFC schemes. The common principle is to derive a *control-flow graph* (CFG) from the program's structure and to unify actual execution paths with the corresponding ones in the CFG at runtime. Usually, this is accomplished by associating each node in the CFG with a signature as a representative for the respective execution progress.

Even with the focus on signature-based CFC, there are various degrees of implementation freedom that amount to a considerable number of variants. Variations include node sizes, ranging from individual instructions over basic blocks (i.e., branch-free intervals) to groups of basic blocks (e.g., single-entry, single-exit or dominance regions in the CFG). Furthermore, the signature associated with a given node can either be assigned or be derived from the nodes' properties itself [22]. Additionally, it is not specified whether the actual control-flow checking is embedded into the control flow to be monitored or executed concurrently, for example on a dedicated watchdog processor [22]. Some CFC schemes, especially concurrent ones, are hybrid hardware-assisted implementations while others are fully implemented in software. Finally, the actual operations used for signature verification, assignment and transition are further design choices.

Nonetheless, signature-based CFC techniques considered in this paper share the same basic principle, which we will showcase using the example of *Control Flow Checking by Software Signatures* (CFCSS) [24]. It operates on the basic-block level with each basic block being referenced by a uniquely assigned *signature* s_i. The global *signature variable* G is used to track the execution at run-time; in a fault-free run, it always corresponds to the signature of the basic block currently being executed. G is updated at the beginning of each basic block using the XOR-operation on G and the *bit-wise difference* d_i between the predecessors signature s_h and s_i:

$$G = G \oplus d_i = s_h \oplus (s_h \oplus s_i) = s_i$$

While this approach is straightforward for single predecessors (cf. Figure 1, $BB_1 \rightarrow BB_2$), precautions have to be taken for multiple predecessors. As illustrated in Figure 1 (right), the basic idea is to declare one of the predecessors as the canonical one (BB_3) and to use its signature in all successors for the computation of the respective signature difference d_i. To incorporate the remaining predecessors (BB_5), a global *signature adjusting variable D* is introduced. Whenever a basic block is left, the individual signature difference with the canonical block is stored in D. Subsequently, the successor (BB_4) will use both D and the bitwise difference d_4 to complete the signature transition:

$$\begin{aligned} G &= G \oplus d_3 \oplus D = G \oplus s_3 \oplus s_4 \oplus s_3 \oplus s_x \\ &= G \oplus s_4 \oplus s_x \end{aligned}$$

In general, control-flow faults are detectable by a mismatch of the global signature G and the block-local signature s_i.

This basic principle is common to virtually all CFC schemes with only details varying: Differences can mainly be found in the operations used for signature transition, the frequency of signature checks as well as the granularity and the handling of multi predecessor situations. All in all, these commonalities allow for a uniform and structured analysis of signature-based CFC scheme efficacy.

3 CASE STUDY

In this paper, we examined the following signature-based software-CFC schemes: (1) *Plain Interblock*[2] (PI), a simple basic block fencing approach inspired by [6]. (2) *Control Flow Checking by Software Signatures* (CFCSS) [24], which we already described in Section 2. (3) *Yet Another Control flow Checking Approach* (YACCA) [13], an adapted and enhanced version of CFCSS. (4) DOM [8], a dominator-region based signature checking approach. For reference, Table 1 holds a list of the most important abbreviations used in this paper. We selected these four CFC schemes as they are representative of the work that has been done in the field. They furthermore span the opposing design objectives of overhead/granularity and detection capabilities.

PI, for example, assigns a static signature to every basic block ahead of time. At the beginning of the basic block, the signature is dynamically stored in a global register, which is validated at the block exit. As return addresses on the stack are considered to be particularly vulnerable (they are stored in data memory), a special method signature is set before returning and subsequently validated at the call site.

In contrast to PI, the other CFC schemes verify predecessor/successor information when the control-flow migrates between basic blocks. As described in Section 2, CFCSS verifies signatures by using the XOR-operation. YACCA assigns a unique signature to each basic-block entry and exit. This method enables the detection of both erroneous branches from within a basic block to a valid successor and from the final branch instruction back into the basic block itself. Furthermore, to eliminate wrong-successor control-flow errors for conditional branches, the condition is reevaluated at the branch destination. We also incorporated this improvement as the $\mathrm{YACCA_d}$ variant to determine its effectiveness.

The granularity of CFG nodes adds another dimension to the design space. Typically, an inverse relationship between check granularity (i.e., node size) and the susceptibility to soft errors is assumed [2]. Therefore, coarse granularity (e.g., single-entry-single-exit regions instead of basic blocks) should weaken the scheme's detection capabilities. DOM makes this trade-off by operating on groups of basic blocks, namely dominance regions. These hold a single dominator block that is executed before any other block of the region (e.g., a function entry block dominates all basic

[2]Naming based on [15] as no name was given in the original publication

Abbrev	Description
PI	Plain Interblock [6]
CFCSS	Control Flow Checking by Software Signatures [24]
YACCA $_{[d]}$	Yet Another Control flow Checking using Assertions with opt. <u>d</u>uplication [13, 14]
DOM $_{\{c,s\}[l]}$	Dominator based CFC using <u>c</u>overage or <u>s</u>mallest region selection strategy with opt. <u>l</u>oop optimization [8]
MEM	Fault injection into memory
IP	Fault injection into program counter
ALL	Combination of mem, ip and registers

Table 1. Abbreviations used in this paper.

blocks within a function). DOM selects a fixed number of dominance regions and assigns to it a single bit in the global signature. The region-entry node sets the bit, whose presence can then be enforced either within the region or when leaving the dominance region.

3.1 Software Infrastructure

The next step is the actual implementation and, even more important, the automated employment of the selected CFC schemes to a given application. This employment is achieved by an automated program transformation using compiler techniques, which may engage from instruction level to high-level programming language—the schemes originally used diverse approaches and toolchains.

However, to be admissible, each transformation requires a set of static guarantees, in this case for global control-flow analysis and refinement. The language C, as the de facto standard in the embedded domain, is somewhat difficult in this sense: Its permissive, low-level memory model and type system emphasizes aggressive local performance optimizations over accurate global analysis. Consequently, either some language features have to be omitted (e.g., function pointers, computed gotos) to again facilitate global analysis or one can resort to type-safe languages, such as Java, in the first place.

As there was no common ground between all CFC scheme's toolchains, we opted for the later and implemented the schemes in the context of the *KESO Multi-JVM for OSEK-based Systems* (KESO) [33, 36]. This source-to-source compiler takes type-safe JVM bytecode and outputs C. At the same time, it provides the required global control-flow information, including both, strong guarantees for corner cases like function pointers as well as fine-grained control-flow refinement. Since KESO is a transpiler to C, the output remains comparable as the actual CFC is woven in the C code. Subsequently, the output equals a traditional C implementation and relies, for example, on the GNU toolchain. Thereby, we completely circumvented the complexity of a conventional JVM.

KESO's global analysis furthermore allows for transformation of non-functional properties such as timeliness, overhead, and most notably dependability (e.g., replication [35], array bounds checking, and parity encoding of pointers [32]), when paired with contextual information (e.g., static system configuration). To obtain a pristine evaluation of CFC, we disabled all those additional hardening mechanisms, even those inherent to the language, such as null and bounds checking. KESO was further configured to use the JDK7 bootclasses (OpenJDK 1.7.0_111) and to perform a full escape analysis [7, 31]. The generated C code was then compiled with GCC 5.4.0[3] (`-O2 -mpreferred-stack-boundary=4 -g`).

[3]GCC Bug #78580, unresolved during the study, prevented a more recent compiler version

Stress	**Control-flow**		**Data-flow**	
Use Case	Sequence Detector	Quicksort	Matrix Multiplication	Flight Control
Instruction Count	663	499	96	3299
Function Calls	67	8	2	27
Branches (cond/total)	58/98	27/58	8/15	173/301
Basic Blocks (BB)	165	74	20	427
Opcodes per BB (max/∅)	22/4.02	29/6.74	13/4.80	135/7.73
Predecessors (max/∅)	16/1.32	6/1.39	2/1.05	6/1.34
Successors (max/∅)	16/1.39	4/1.42	2/1.10	5/1.35

Table 2. Structural properties of the use cases.

3.2 Use Cases

We have selected two of our applications from those typically used in the original work on the selected CFC schemes: Matrix multiplication [6, 13, 14, 24] and quicksort [6, 24]. However, we consider the relevance of these algorithms to the domain of embedded systems as rather limited. Furthermore, both algorithms resemble heavily repetitive workloads. Therefore, we introduced two additional use cases: a sequence-detector state machine and a flight attitude controller, which mimic real-world production code as close as possible. We used quicksort (1) from the GNU classpath (v0.99) and implemented matrix multiplication (2) ourselves. The sequence detector (3) was generated using the State Machine Compiler (v6.3.0). We took the flight attitude controller (4) as a MATLAB/Simulink model from [37], with C code generated by Simulink's Embedded Coder (v8.10, R2016a) and verbatim ported to Java. As inputs, we used: (1) an unsorted list of 42 random integers, (2) two 5×5 integer matrices, (3) a sequence of 31 binary inputs that produces two matches in the detector and (4) a control step of the flight controller with 13 (simulated) sensor signals.

The structural properties of all use cases are given in Table 2. Due to the rigid structure of matrix multiplication, its maximum and the average number of pre- and successors is comparatively low, and the overall use case is rather small with 96 instructions. To the other extreme, the state machine implementation in the sequence detector provides a higher cyclomatic complexity with multiple pre- and successors and a large number of function calls. The latter induced additional vulnerabilities due to the necessary return addresses pushed on the stack. The attitude controller represents a rather data-driven, compute-intensive workload with rather large basic blocks. Compared to the other applications, it is also rather large with 3299 instructions. Quicksort's properties usually lie midways between those extremes and provide a highly data-dependent, conditional control-flow. Overall, we consider this set of use cases as adequate to cover a large set of application classes.

3.3 System and Fault Model

The effectiveness of a given hardening method can only be evaluated with respect to a given fault model. From a software point-of-view, transient hardware faults manifest as observable bit flips in memory and CPU registers [4, 20, 38]. The resulting fault patterns range from single-bit over multi-bit flips up to a complete inversion of data words. Radiation experiments with 90–130 nm SRAM memory [23] indicate, however, that 95 % of soft errors lead to single-bit flips, with the remaining 5 % spreading up to five-bit flips at maximum. These observations have been corroborated by and extended to more recent hardware designs as well as to computation logic by further studies [4, 21, 30]. In contrast, non-volatile flash memory shows a comparatively high robustness regarding transient hardware faults [17] and is therefore considered to be reliable.

The fault hypothesis used in this paper is, therefore, the widely accepted single-bit single-fault model with uniform fault distribution, which we in particular share with the original work on the CFC schemes discussed in this paper. Furthermore, we assume that spatial and temporal isolation is provided by the operating system's partial virtualization. The *memory protection or management unit* (MPU/MMU) further confines execution to the text segment, which further resides in read-only memory, e.g., flash memory. Therefore, soft errors leading to code-section writes are detected and only faulty writes to data may become permanent.

3.4 Fault-Injection Methodology

We evaluated the effectiveness of the CFC schemes using the FI framework FAIL* [28]. FAIL* records a full trace of a program's execution (i.e., the golden run) by single-stepping it in a simulator and extracting all referenced data, registers as well as memory locations and values at *instruction set architecture* (ISA) level. The trace is then used to determine the liveness intervals between a write access and the last subsequent read access to the corresponding data. This information is used to perform targeted FI at ($cpucycle$, $datalocation$) in the trace for all cycles in which live data is stored in a certain data location, memory or register, providing a *full scan* of the fault space. After the injection, normal execution is resumed. When the actual computation has completed, the result is verified and recorded. For our study, we opted to check just the result (i.e., we verified the integrity of the given data structures storing the result) but not the integrity of the whole address space. This method matches the way a further execution would access the results. If a trap (e.g. an arithmetic exception or illegal memory access) occurs between injection and verification, the execution will be terminated and the event recorded. Finally, we used a timeout that mimics the OS's timing protection to catch execution-time budget overruns (e.g., fault-induced non-terminating loops).

4 EXPERIMENTS AND ANALYSIS

In this section, we move on to the evaluation of the CFC schemes by extensive FI campaigns. After detailing the experimental setup in Section 4.1, we start by reproducing and revalidating the comparative results from the literature using the *residual failure rates* metric in Section 4.2. Subsequently, we change perspective and glean the shortcomings of this metric in comparison with the more recent *absolute failure counts* metric in Section 4.3. Based on this, we assess the impact of ISA-level micro-effects on CFC efficacy in Section 4.4. Finally, to achieve a more complete, whole-system evaluation of CFC, we extend our view to data faults and their momentous impact in Section 4.5.

4.1 Experimental Setup

The experiment setup covers the configuration of the system regarding hardware and OS, as well as the CFC implementation variants and the FAIL* campaign design.

To account for the large design-space described in Section 2, we made several aspects of the CFC implementations configurable: (1) Signatures and other CFC-related variables can either be stored in *memory* or dedicated *registers.* (2) Consecutive basic blocks can optionally be *compress*ed into larger regions (e.g., at the locations of potential array bound checks) to increase the granularity. (3) For YACCA *duplication* of the branch condition can be enabled as YACCA_d. (4) Either the 32 dominance regions in DOM are equally distributed with uniform *coverage* as DOM_c or a heuristic selects the *smallest* regions first as DOM_s. (5) Optionally, marker checks can be reduced to one per *loop* if the body does not cross the region boundary as DOM_l. Along with the four CFC schemes, this resulted in 33 configurations per use case.

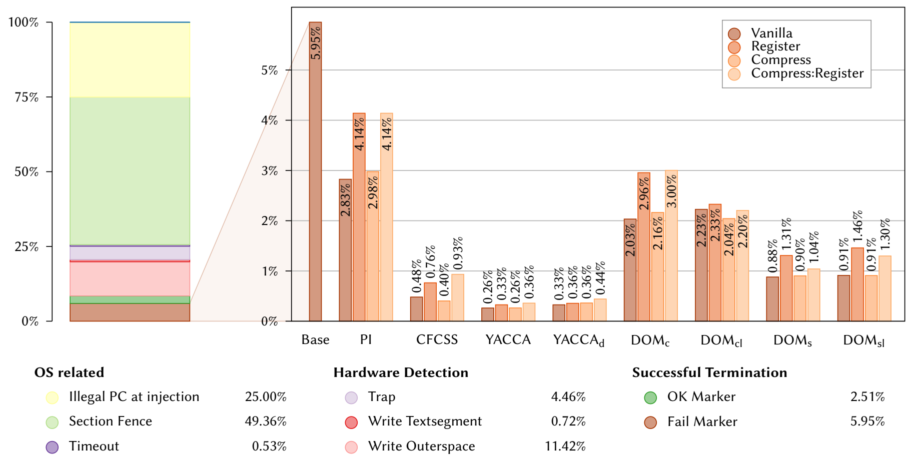


Fig. 2. Control–flow-only FI campaign on the quicksort use case: On the left the overall fault distribution with the software-visible failures magnified to the right. Thereof, the residual failure rates of the unhardened (base) versus the various CFC schemes (relative scale).

The actual FI was performed on the IA-32 architecture as this is the only one available for both the FAIL* FI framework and the AUTOSAR-compliant real-time operating system used in KESO. We provided the simulated system with 16 MB of memory, mapped at the lower end of the address space. As accurate worst-case execution times were only available for the actual hardware, we used a safe timeout parameter in the range of seconds. In FAIL* individual injections are grouped as dedicated *benchmarks* by their respective fault location: Program-counter register (IP), other registers (REGS), and memory locations (MEM).

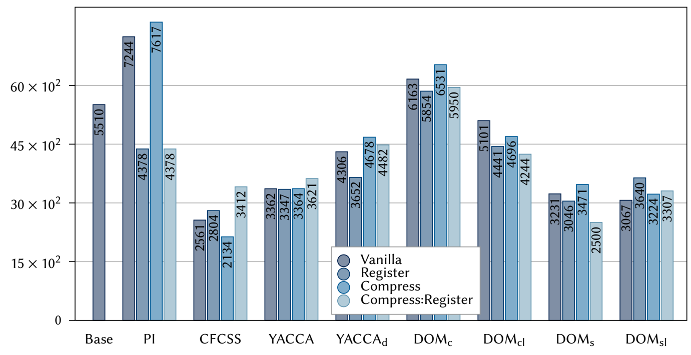
In total, our experiments amount to 703,485,376 individual injections and 47 GB of raw data, grouped by use cases, implementation variants and benchmarks into 396 aggregated result sets. In the brevity of this paper, it is impossible to present such a multidimensional dataset in its entirety. Therefore, we will focus on the most notable, general effects and explain those by representative examples. The source code along with the complete dataset is, however, publicly available on our website[4] for further exploration and investigation.

4.2 First Step – Residual Failure Rates

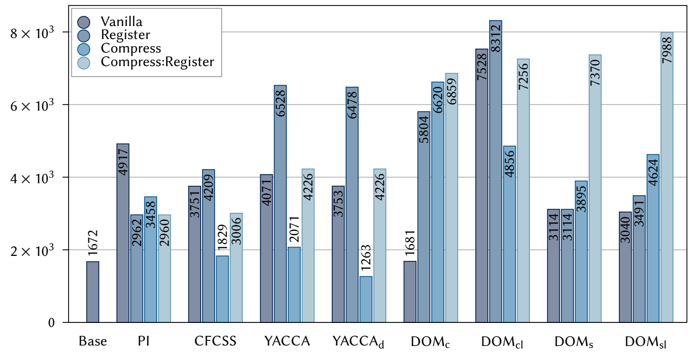
First, we evaluate the effectiveness of CFC schemes to detect control-flow faults by injecting bit flips into the program-counter register (IP benchmark). This defined approach ensures an isolated injectivity, observability, and traceability of control-flow faults. While there are in general other more subtle fault sources—for example, faults affecting address and conditional operands, return addresses or instruction memory—these can either be classified as data faults, which we address separately in Section 4.5 or mapped to a time-shifted program-counter fault. In the following, we discuss the impact of control-flow faults on unhardened programs before evaluating the efficiency of CFC schemes under study using the commonly used *residual failure rate* metric.

4.2.1 General Fault Distribution. Before investigating the qualities of the various CFC schemes, we assessed the overall relevance of CFC by examining the effects of control-flow faults on the

[4]www4.cs.fau.de/Research/Soft-Errors/



(a) Arraysort for comparison with Figure 2.



(b) Matrix multiplication as the worst-case example.

Fig. 3. Control-flow-only FI evaluating the absolute failure count (linear scale).

unhardened programs. Figure 2 displays the distribution of different injection results for the quicksort application.

The most dangerous event is the fail-marker event, which represents an undetected failure (silent data corruption). It's surprising that, depending on the use case, only 0.74–5.95 % of all injected control-flow faults resulted in undetected failures. As noted earlier, disregarding latent errors in local data structures that did not manifest as failures shifts the balance between the fail and OK markers in favor of the latter. Still, even combining OK and fail markers only $\approx$ 8.5 % of the experiments result in successful program termination. So more than 91 % of the faulty computations terminate prematurely due to hardware mechanisms. Here, memory-related events dominate: Either execution was attempted on unmapped memory (illegal PC) or outside of the executable code area (section fence), or a write access to unmapped memory was performed (write outerspace). Certainly, the effectiveness of spatial isolation (i.e., section fencing) benefits from small-size micro benchmarks. However, we, also with regard to our example, consider our results to be representative for the domain of safety-critical embedded systems, which typically are designed and engineering with rather small protection domains in mind.

Our conservative choice of the timeout value resulted in a rather small number of timeout events. We assume that tight execution budgets, again typical in safety-critical embedded systems, would, therefore, reduce the number of undetected failures of even further.

► *The vast majority of more than* 91 % *of control-flow faults was caught by hardware and OS mechanisms. In the end, no more than 6 % of failures were subject to CFC actions at all.*

4.2.2 Residual Failure Rates. Soft error mitigation mechanisms are employed to improve the system's reliability. A commonly used metric to quantify reliability is the *residual failure rate*: the ratio of undetected wrong results.

The right-hand side of Figure 2 displays the residual failure rates for the IP FI of the quicksort application. While our injection heuristic for control-flow faults deviates from earlier injection experiments [6, 13, 14, 24], the results still show the same qualitative effects: All implemented CFC schemes reduced the residual failure rates, predecessor-successor encoding schemes like CFCSS and YACCA outperformed the region-based approaches like PI and DOM. YACCA represented an improvement upon already good results of CFCSS and reached figures equal or even slightly above those found in [13] with a fault-coverage of up to 99.74 %. In comparison to CFCSS and YACCA, the poor performance of DOM variants seems to further support the often acclaimed notion of a tradeoff between check granularity and efficiency.

► *We consistently observed residual failure rates that match with literature. All variants were effective in reducing the failure rate; the effectiveness correlates with granularity and redundancy.*

4.3 Changing Perspective – Absolute Failure Counts

Recently, however, the usage of residual failure rates in quantitative evaluations of hardening mechanisms has been proven to be problematic [27]. The key point of criticism is the lack of a common base when failure rates are used to compare variants. Our evaluation just presented in Section 4.2.2 is affected by the very same problem: Fault probabilities are always expressed in relation to both space and time. As all CFC techniques induce (non-functional) overhead, both regarding size and execution time, their fault space increases compared to the unhardened application.

To resolve this problem, the *absolute failure count* was proposed as an alternative metric [27], which rectifies this issue by weighting all injection results with the size of the associated fault space. In the case of a full fault-space scan, as performed in this paper, this metric can be easily obtained by counting the absolute number of failures (i.e., fail marker events).

Figure 3a depicts the same experiment as Figure 2 using the absolute-failure-count metric. This direct comparison yields some interesting results: First of all, not all CFC schemes were effective. The memory variants of PI and all variants of DOM lead to an increased failure count compared to the baseline. For all other variants, relative improvements fell significantly short compared with the observed failure rates: While YACCA shrunk failure rates by 95.6 %, it reduced absolute failure counts by mere 39.3 %. Furthermore, the best performing algorithm for this use case switched from YACCA to CFCSS, with the improvements peaking at only 61.3 %.

Moreover, while we observed consistent improvements for all CFC schemes across the board with failure rates, this is not the case with failure counts. For comparison, Figure 3b provides the failure counts for the matrix multiplication. Note that all but one CFC variant show degraded failure counts. The only improvement was achieved by YACCA_d (compressed), albeit only by 24.5 %. A probable explanation for this effect is the difference in the application structure. Matrix multiplication typically consists of a couple of small basic blocks that represent the loop's indexing and conditional branches, while the entire computation resides in one large basic block in the innermost loop body, where most of the execution time is spent. This structure benefits intra–basic-block control-flow faults, which are undetectable by all implemented CFC schemes.

► *Switching to the more realistic absolute failure count metric, the overall efficiency degraded substantially for all CFC schemes. Even worse, in many cases, CFC even reduced the overall reliability.*

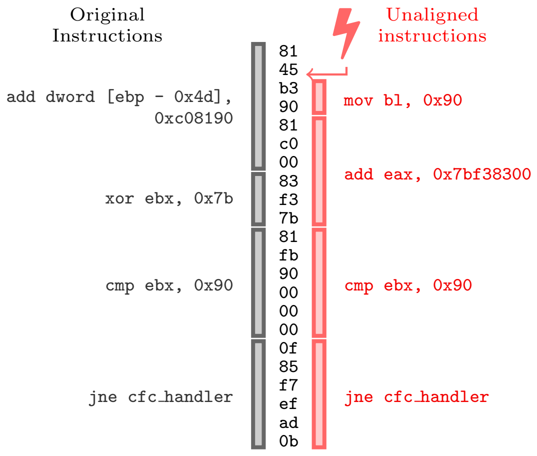


Fig. 4. Micro-effect: Fault-induced non-aligned jump is bypassing the CFC.

4.4 Micro-effect Analysis

In addition to the metrics issue, we observed a high variance within individual use cases, in particular with the matrix multiplication (Figure 3b). For DOM, to some extent, fluctuation are intrinsic to the scheme: Even small changes to the CFG and therefore the dominator tree directly influenced the region selection and signature-check-placement strategies, resulting in substantially different executables. This variation, however, did not apply to the deviations between register and memory variants that we observed in other cases.

For the register variants of YACCA and CFCSS, we permanently allocated two registers to the CFC scheme. These, however, represent a notoriously scarce resource, especially in the IA-32 architecture that only provides eight general-purpose registers. Therefore, the deprivation of two registers leads to more variables being spilled to the stack, which consequently leads to a larger fault space and bigger basic blocks. The latter in turn causes additional and again undetectable intra–basic-block control-flow faults. The effect of this is, however, highly application specific: As the set of live variables is higher for the matrix multiplication, the impact is comparably higher or even inverse to the one observed with array sort.

► *Overall, the reliability impact of CFC variants is not only highly implementation specific but also strongly correlated with the application structure. Suboptimal combinations can even harm.*

Given this relationship between structure and implementation variants, it might be tempting to infer similar correlations between application specifics and well performing CFC schemes. Indeed, the best performing configurations differed between use cases: CFCSS with CFG compression for array sort, YACCA$_d$ with CFG compression for the matrix multiplication, DOM$_c$ with compression and signature storage in registers for the sequence detector and YACCA$_d$ with CFG compression for the flight control.

However, it would be ill-conceived to arrive at conclusions regarding the suitability of schemes for certain application classes from only those individual data points. To make this clear, we have to reconsider two requirements for failures in our injection: First, the fault causing the failure has to be possible. Given our single-bit fault model, this means that control-flow faults may only misdirect the control flow to addresses with a Hamming distance of one to any given fault-free address in the program. Second and even more important, the subsequent, erroneous execution has to terminate successfully, without causing any hardware traps and within the timeout. This effect is tightly coupled with the actual hardware context at the fault location, that is register and memory content. The context is reinterpreted by the resulting faulty instruction stream with potentially completely different semantics. To exemplify this effect: If a given register represents an array index in the fault-free case but is used as a pointer variable in the erroneous control flow, it will most likely trigger an MPU/MMU trap. If it were used as an arithmetic operand instead, it would most likely silently corrupt the subsequent calculation and lead to a wrong result. That way, it remains undetected by hardware measures, actually causing a failure.

Consequently, each variant exhibits an entirely different set of possible control-flow faults: The additional instructions introduced by CFC at the very least cause a relocation of all symbols in the binary and therefore a different set of addresses within Hamming distance one. When integrating the hardening process into a source-level compiler, as we did, these extra instructions further interact with the instruction scheduling, register allocation and code generation features of the backend, which will in most cases again result in significant changes to the generated binary.

► *In contrast to evaluation approaches often presented in literature, CFC should be assessed under careful consideration of the actual code generation process as the application's binary representation heavily impacts possible fault patterns.*

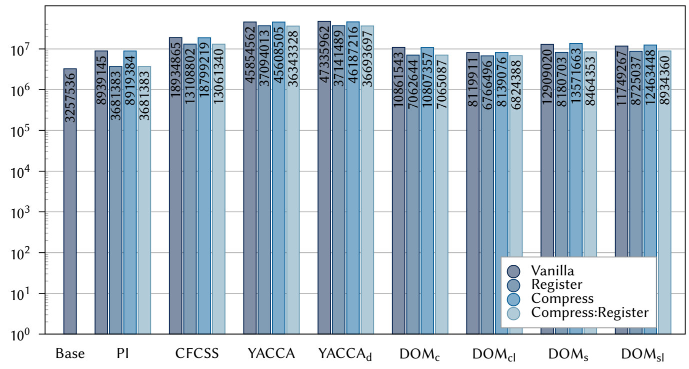
Even with this knowledge, it is hard to provide conclusive predictions on a schemes effectiveness. For instance, differences between YACCA und DOM in the matrix-multiplication use case (cf. Figure 3b) seem to be significant from a quantitative point of view. However, even tiny changes may have a tremendous impact on the absolute failure count. Even more, especially with iterative, loop-heavy code, those effects may be amplified by repetitive execution. For example, in matrix multiplication, we observed that for some configuration, faults targeting a single instruction lead to 823 failure events, which translates to 51.6 % of the baselines total fail marker events.

► *Coincidental hot spots in the binary representation may ultimately change quantitative evaluation results and thus even reverse qualitative conclusion on the suitability of individual CFC schemes.*

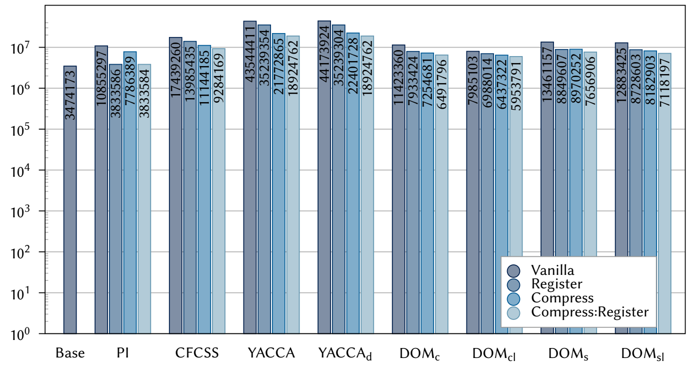
4.4.1 Unaligned Jumps. On architectures, such as IA-32, with variable-length instruction encoding, a particular incarnation of control-flow related micro-effects can be observed regarding unaligned jumps. For example, the average instruction length of our use cases is ≈ 3.25 bytes. As a result, we observed a considerable number of control-flow faults to lead to errant executions starting somewhere in the middle of another instruction. Again, even small changes in the underlying assembly may result in considerable variations in the execution of the faulty instruction stream and therefore either aggravate or alleviate the CFC's effect. In the worst case, this could even zero out the CFC code as illustrated in Figure 4.

In our experiments we observed 59.6 % of the failures to be unaligned control flow, posing a serious threat to reliability. Admittedly, given that the overall rate of unaligned control flows among all injected faults was 86.1 %, this fault class was not particularly effective and triggered the hardware detection mechanisms (i.e., illegal instruction traps) above average.

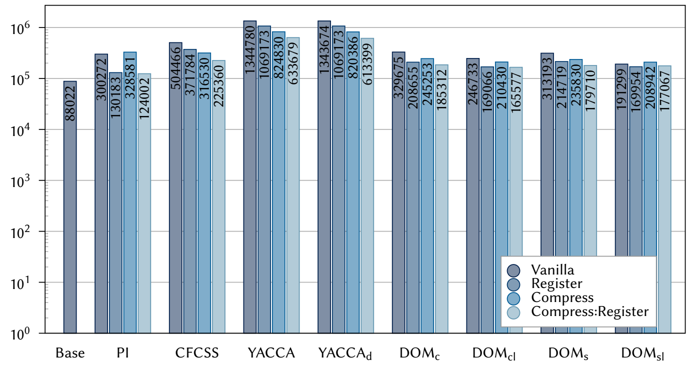
4.4.2 Interim conclusion. So overall, the ratio of control-flow faults that result in actual failures is not substantial. While failure rates suggest a high improvement for CFC, a switch to the absolute failure count metric, which accounts for overhead, shows only an impaired, very application specific performance, with even degraded reliability in many scenarios. Overall, the high influence of micro-effects renders general propositions on scheme efficiency infeasible and mandates for a binary-level evaluation.



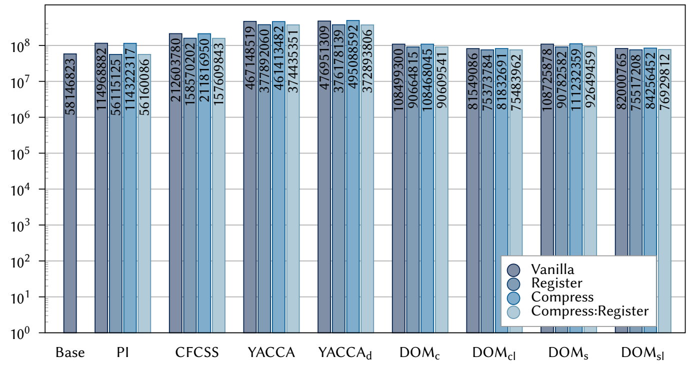
(a) Arraysort



(b) Matrix multiplication



(c) Sequence detector



(d) Flight control

Fig. 5. Control-flow and data FI evaluating the absolute failure count of all applications. Figures show the drastic increase of the failure count due to overhead-induced fault-space explosion (logarithmic scale).

4.5 General Fault Injection

As noted earlier, transient faults in the program counter register are one, but not the only source of control-flow errors in a system. Actually, the original publications of both CFCSS and YACCA did not even model this particular fault type at all and focused on memory based control-flow faults instead. They performed injections into the immediate operand of branch instructions [13, 14, 24] and in the case of CFCSS [24] branch deletion and insertion.

However, these attempts failed to cover all possible single bit faults. Therefore, we extended our experiment to also include all single-bit upsets in all general-purpose registers and memory locations to obtain a full scan of the underlying fault space including data faults.

The results of this *general fault injection* are shown for all four applications in Figure 5. Once again, the outcome shifts and worsens the perception of CFC efficiency: In virtually all cases, the overall reliability was degraded by the introduction of CFC schemes, with PI with CFG compression in the flight-control use case being the only exception. Notably, the schemes complexity and implementation overhead now strictly dominates their efficiency: The simple PI and the coarse-granular DOM scheme consistently outperform CFCSS, which in turn puts the more complex YACCA and YACCA_d into place. Regarding implementation variants, both register and CFG compression reduces the overhead and thus excel their heavier counterparts.

Overall, the degraded reliability is caused by a widening of liveness intervals due to the CFC's overhead: During an execution flow, each data location is associated with a series of liveness intervals, during which it stores information relevant to the computation. Each instruction inserted between definition and use represents a new vulnerability option that has to be added to the fault space. As a large set of memory locations is usually active and relevant to the computation at any given point in time and as no data integrity measures were taken, any fault to such a data point will result in a failure, unless it triggers a trap during the continued execution. For reference, **??** shows a modeled injection of 1000 FI campaigns, with and without weighting of the results by overhead. While a comparison with Section 4.4.2 indicates that this effect is not linear in runtime as the affected dataset varies in time, it still dominates any other effects that could be introduced by the individual properties of the different CFC schemes.

► *Also considering data faults, CFC dangerously fails the reality check with the reliability decreased in virtually all test scenarios. The reason being the overhead induced, which substantially enlarges the fault space especially for the typically neglected data faults.*

As hinted earlier, the impact of the CFC's overhead depends on the number of live memory and thus on the size of the application state: Both quicksort and matrix multiplication process large arrays in memory without aggregation and subsequently pose a huge attack surface. Similarly, the internal state of the sequence detector is maintained during the complete execution of the application. The flight-control use case, on the other hand, is data-flow driven and represents a rather sequential computation that continually transforms the same small data set and is, therefore, less vulnerable to the fault-space explosion. But even given such a beneficial application structure, the best performing scheme, PI storing its signature in registers, only improved the reliability by mere $\approx 3.5\,\%$.

► *The fault space's impact is directly related to the software's structure and state size. Consequently, data-intensive are more vulnerable than control–flow-oriented applications.*

Interim conclusion. Overall, it is imperative to account for data faults when implementing CFC schemes. Otherwise, reliability may be even degraded in comparison to the unprotected system, with potentially catastrophic effects in safety-critical settings. Ultimately, given the small fraction of control–flow-related failures in comparison to the significant overheads induces by CFC, their use is to be questioned at all. Unless the deficits are solved, we have only the recommendation to resort to proven techniques such as replication or arithmetic codes [26].

5 THREATS TO VALIDITY

First of all, it is important to recall that this paper is neither designed nor intended as a shootout between CFC schemes. Instead, it focuses on general, qualitative design constraints that impact such techniques. In this section, we perform a structured analysis on the threats to validity following and adapting the structure commonly used in statistical analysis: conclusion, internal, construct, and external validity along with a short discussion on dependability aspects [9].

As a form of empirical *conclusion validity*, we attempted to address and eliminate different types of bias by means of traditional study design. While a full scan of the fault space effectively eliminates any selection or sampling bias in the FI itself, its subjects (i.e., the four test applications) might suffer a slight selection bias: We used matrix multiplication and quicksort because they are very common benchmarks (e.g., used in [6, 13, 14, 24]), yet they are data driven and seem somewhat unrepresentative for the domain of embedded systems. We addressed this issue by adding both a control-flow intensive workload and a real-world example. Having said that, application-specific micro-effects (cf. Section 4.4) may adversely impact the FI results of individual binaries. Still, as we have only drawn conclusions from consistent, indicative effects observable in all benchmark, the law of large numbers should mitigate those artifacts.

Internal validity raises the question of cause and effect, focusing on alternate factors that could contribute to the observed result. Across all experiments, we used the same hardware platform, toolchain, and FI framework. Consequently, variants differ only in the CFC configuration. Furthermore, FAIL*'s hardware simulation and FI are entirely deterministic and controllable. As we rely on and advocate for binary-level FI, we consider this to be a common ground for the evaluation free of unanticipated side effects.

Construct validity traditionally considers whether the experiment is fit to measure the cause–effect relationship, which corresponds to modeled versus actual behavior in our case. In general, we share the commonly accepted and used fault model of uniformly distributed single-bit faults on ISA level. While this might not accurately reflect the actual susceptibility of the individual hardware components, it is first of all paramount to maintain at least qualitative comparability of our results with previous work. At the same time, there is a notable variance in hardware technologies as well as architectures especially in the embedded domain and thus an equally large variance of susceptibility. However, a definite, representative and commonly agreed fault model at that level is missing to the best of our knowledge. Nonetheless, practical experiences [4, 20, 21, 23, 25, 30, 38] suggest that evaluations using the simplified model still provide sufficient qualitative indicators to assess the vulnerability of systems and therefore substantiate our evaluation.

Further issues arise from the use of FAIL* and its hardware simulator, which simplifies the timing of individual instructions to a single-cycle behavior. As this deviates from the actual behavior of contemporary hardware, any assessment of the practical impact directly translates to *external validity*. As we have seen in Section 4.5, the size of liveness intervals and therefore the duration of instructions is a central concern for vulnerability. The single-cycle model absorbs execution-time variations caused for example by pipeline hazards, memory access timings, load-store architectures or complex instructions. In all cases, the model is overly optimistic and thus the execution times and liveness intervals on the actual hardware can only become larger. This simplification benefits the CFC schemes in the sense that it underestimates the increase in attack surface.

KESO internals further fosters small basic blocks as a result of the runtime check implementation, which in turn reduces granularity and amplifies the CFC overhead. Once more, this will result in quantitative changes when ported to other implementations. However, this is completely justified in the context of this work, as it permits easy pinpointing of problems in CFC and equates to changes in the application structure.

We conclude the list of experiment specific threats with a discussion of *dependability*. As we performed a full scan of the fault space, the individual FI experiments are deterministic and repeatable, which eliminates measurement related artifacts. All conclusions drawn from the experiments conducted in Section 4 and summarized in Section 7 are based on significant effects observed across all applications, which makes our propositions dependable.

So overall, the threats to validity might impact the quantitative portions of this paper, mainly due to the choice of our simulation platform, toolchain, and use cases. All these cross-cutting concerns naturally have a considerable influence on reliability and therefore the quantitative results might vary, which is part of our message. Even though, we are confident that the qualitative effects shown in Section 7 remain observable across all platforms and therefore consider our model to be as good as any other.

6 RELATED WORK

Some of the individual effects we addressed in this paper have already been discussed in isolated contexts in the past. The described influence of compiler specifics has already been studied in different implementation and evaluation settings: Using *Vulnerability Factor of Control Flow* (VFCF), a port of the architectural vulnerability metric to the domain of control-flow faults [19], in the context of aspect-oriented fault tolerance implementations including a signature based control-flow checking scheme (DS-CFC) [1], and in the evaluation of the ACCE signature based control-flow checking technique, which was implemented on the level of LLVM intermediate representation [10], a level similar to the implementation on JVM level used in this publication. In all cases, a high impact of different compiler optimizations on the reliability was observed. Furthermore, variations of reliability due to the interaction of several compiler optimizations with different application types have been reported in this context [10, 19], which corresponds to the findings on implementation details presented here.

The influence of memory and runtime overhead has repeatedly been revisited in the past. In [3], an evaluation factor is proposed and evaluated, that tries to integrate the overhead as a performance characteristic into CFC comparisons, both in an experimental as well as an analytical setting. The impact on susceptibility to faults was discussed in [12], where the static memory and runtime overhead is used to determine a normalized fault coverage metric. Recently, the absolute failure count metric has been proposed for data faults, which, while conceptually similar, additionally uses the dynamic memory overhead in the comparison [27]. This metric was applied to control-flow faults and used in this work.

Overall, the use of complex fault hardening schemes has been questioned. Due to the influence of overhead on architectural vulnerability [29], costly schemes were found to degrade the overall vulnerability. While we believe that precisely predicting the peculiar effect of individual control-flow faults on the result in theoretical vulnerability calculations is hard to impossible, and therefore resorted to simulated FI for our work, this paper supports our conclusions from a lower level perspective. From a security viewpoint [34], FI experiments similar to our setup were performed for a variety of control-flow and data hardening schemes. Although not directly comparable, the conclusion that simple schemes often produced better results due to the lower overhead matches with our findings.

7 CONCLUSIONS AND LESSONS LEARNED

Overall, our study disclosed various latent deficiencies of both the *residual failure rate* metric as well as software-implemented CFC which potentially compromise their general use.

First of all, tentative experiments with the injection of control-flow faults suggest that contemporary hardware protection mechanisms already catch the vast majority of control-flow faults, with only a small fraction of less than 6 % manifesting themselves as failures in our experiments.

While the reliability improvements were significant regarding residual *failure rates* that are in line with the literature, the transition to absolute *failure counts*, which account for memory and execution-time overheads, resulted only in moderate improvements at best and even degraded reliability at worst. This disparity between the two metrics suggests that ignoring overheads has led to ill-guided optimization and evaluation of CFC schemes in the past.

The absolute-failure-count metric furthermore exposes the liability of CFC to coincidental micro-effects and artifacts caused by compilation and linking steps, coupling the effectiveness to attributes of the binary that are unrelated to the CFC's implementation. Therefore, the overall reliability can only be assessed from individual, concrete binaries, jeopardizing the ease of use.

Finally, soft errors do cause not only control-flow faults but also a considerable amount of data faults. Their impact is typically neglected in the literature, which spoils the evaluation from a systems perspective. However, our experiments conclusively emphasize that, when unhandled, data faults dominate the overall reliability. Even worse, their count is tightly coupled to the residence time of data in memory and therefore to the overhead of the CFC itself, which means that scheme complexity had an adverse effect on reliability.

The bottom line is that CFC schemes are not only mostly ineffective but even dangerous when used without further data hardening measures. The scheme's difficult handling, the moderate performance and the incurred overhead in relation to the effectiveness of hardware protection mechanisms, we found that, in our setting, the usage of CFC schemes was not appealing at all. Given the apparentness of our quantitative results, we consider the qualitative conclusions drawn in this paper to be relevant to most applications in the embedded domain.

8 OUTLOOK

Our implementation in the context of KESO leads to comparably small basic blocks, and therefore to a high overhead of the hardening schemes, which certainly amplified the effects discussed above. It should be explored how the qualitative effects quantify on other toolchains as well as target platforms. In a similar vein, we plan to study the quantification of multi-bit faults, caused for instance by error bursts, on the effects discussed.

Furthermore, we observed significant differences between the benchmarking applications, which we traced back to certain application characteristics. It would be interesting to identify more application categories that respond well to certain implementation variants of software based CFC beyond the notion of data versus control-flow intensive applications.

REFERENCES

[1] R. Alexandersson and J. Karlsson. 2011. Fault Injection-Based Assessment of Aspect-Oriented Implementation of Fault Tolerance. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN)*. 303–314.

[2] Z. Alkhalifa, V. S. S. Nair, N. Krishnamurthy, and J. A. Abraham. 1999. Design and Evaluation of System-Level Checks for On-Line Control Flow Error Detection. *IEEE Trans. Parallel Distrib. Syst.* 10, 6 (June 1999), 627–641.

[3] S. A. Asghari, H. Taheri, H. Pedram, and O. Kaynak. 2014. Software-Based Control Flow Checking Against Transient Faults in Industrial Environments. *IEEE Transactions on Industrial Informatics* 10, 1 (Feb. 2014), 481–490.

[4] R. Baumann. 2005. Soft errors in advanced computer systems. *IEEE Design Test of Computers* 22, 3 (May 2005), 258–266.

[5] S. Y. Borkar. 2005. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *IEEE Micro* 25, 6 (2005), 10–16.

[6] P. Cheynet, B. Nicolescu, R. Velazco, M. Rebaudengo, M. S. Reorda, and M. Violante. 2000. Experimentally Evaluating an Automatic Approach for Generating Safety-Critical Software with Respect to Transient Errors. *IEEE Transactions on Nuclear Science* 47 (2000), 2231–2236.

[7] J.-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. 2003. Stack Allocation and Synchronization Optimizations for Java Using Escape Analysis. *ACM Trans. Program. Lang. Syst.* 25, 6 (Nov. 2003), 876–910. DOI: https://doi.org/10.1145/945885.945892

[8] C. Dietrich, M. Hoffmann, and D. Lohmann. 2017. Global Optimization of Fixed-Priority Real-Time Systems by RTOS-Aware Control-Flow Analysis. *ACM Trans. Embed. Comput. Syst.* 16, 2 (Jan. 2017), 35:1–35:25.

[9] R. Feldt and A. Magazinius. 2010. Validity Threats in Empirical Software Engineering Research-An Initial Survey.. In *SEKE*. 374–379.

[10] R. R. Ferreira, R. B. Parizi, L. Carro, and Á. F. Moreira. 2013. Compiler Optimizations Impact the Reliability of the Control-Flow of Radiation-Hardened Software. *Journal of Aerospace Technology and Management* 5, 3 (Aug. 2013), 323–334.

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

[12] P. Gawkowski, J. Sosnowski, and B. Radko. 2005. Analyzing the Effectiveness of Fault Hardening Procedures. In *11th IEEE International On-Line Testing Symposium*. 14–19.

[13] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante. 2003. Soft-Error Detection Using Control Flow Assertions. In *18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 2003. Proceedings*. 581–588.

[14] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante. 2005. Improved Software-Based Processor Control-Flow Errors Detection Technique. In *Annual Reliability and Maintainability Symposium, 2005. Proceedings*. 583–589.

[15] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante. 2006. *Software-Implemented Hardware Fault Tolerance*. Springer US.

[16] R. W. Hamming. 1950. Error detecting and error correcting codes. *Bell System technical journal* 29, 2 (1950), 147–160.

[17] F. Irom and D. Nguyen. 2007. *IEEE Transactions on Nuclear Science* 54, 6 (Dec 2007), 2547–2553.

[18] ISO 26262-9. 2011. *ISO 26262-9:2011: Road vehicles – Functional safety – Part 9: Automotive Safety Integrity Level (ASIL)-oriented and safety-oriented analyses*. ISO, Geneva, Switzerland.

[19] S. Kim and M. A. Rouf. 2010. Modeling and Evaluation of Control Flow Vulnerability in the Embedded System. In *18th IEEE/ACM International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS 2010)*. IEEE Computer Society, Los Alamitos, CA, USA, 430–433.

[20] V. Kleeberger, C. Gimmler-Dumont, C. Weis, A. Herkersdorf, D. Mueller-Gritschneder, S. Nassif, U. Schlichtmann, and N. Wehn. 2013. A Cross-Layer Technology-Based Study of How Memory Errors Impact System Resilience. *IEEE Micro* 33, 4 (July 2013), 46–55.

[21] X. Li, K. Shen, M. C. Huang, and L. Chu. 2007. A Memory Soft Error Measurement on Production Systems. In *Proceedings of the USENIX Annual Technical Conference (ATC '07) (ATC'07)*. USENIX Association, Berkeley, CA, USA, Article 21, 6 pages. http://dl.acm.org/citation.cfm?id=1364385.1364406

[22] A. Mahmood and E. J. McCluskey. 1988. Concurrent Error Detection Using Watchdog Processors-A Survey. *IEEE TC* 37 (February 1988), 160–174. Issue 2.

[23] J. Maiz, S. Hareland, K. Zhang, and P. Armstrong. 2003. Characterization of multi-bit soft error events in advanced SRAMs. In *Intern. Electron Devices Meeting (IEDM '03)*. IEEE Press, New York, NY, USA, 21.4.1–21.4.4.

[24] N. Oh, P. Shirvani, and E. McCluskey. 2002. Control-flow checking by software signatures. *IEEE Transactions on Reliability* 51, 1 (2002), 111–122.

[25] T. Santini, C. Borchert, C. Dietrich, H. Schirmeier, M. Hoffmann, O. Spinczyk, D. Lohmann, F. R. Wagner, and P. Rech. 2017. Effectiveness of Software-Based Hardening for Radiation-Induced Soft Errors in Real-Time Operating Systems. *Lecture Notes in Computer Science (LNCS)* (2017), 3–15. http://dx.doi.org/10.1007/978-3-319-54999-6_1

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

[27] H. Schirmeier, C. Borchert, and O. Spinczyk. 2015. Avoiding Pitfalls in Fault-Injection Based Comparison of Program Susceptibility to Soft Errors. In *45th Int. Conf. on Dep. Systems & Networks (DSN '15)*. IEEE, Washington, DC, USA, 12.

[28] H. Schirmeier, M. Hoffmann, C. Dietrich, M. Lenz, D. Lohmann, and O. Spinczyk. 2015. FAIL*: An Open and Versatile Fault-Injection Framework for the Assessment of Software-Implemented Hardware Fault Tolerance. In *12th Int. Conf. on Eur. Dep. Computing Conf. (EDCC '15)*, Pierre Sens (Ed.). 245–255.

[29] A. Shrivastava, A. Rhisheekesan, R. Jeyapaul, and C. J. Wu. 2014. Quantitative Analysis of Control Flow Checking Mechanisms for Soft Errors. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6.

[30] V. Sridharan, N. DeBardeleben, S. Blanchard, K. B. Ferreira, J. Stearley, J. Shalf, and S. Gurumurthi. 2015. Memory Errors in Modern Systems: The Good, The Bad, and The Ugly. In *20th Int. Conf. on Arch. Support for Programming Languages & Operating Systems (ASPLOS '15)*. ACM, New York, NY, USA.

[31] I. Stilkerich, C. Lang, C. Erhardt, C. Bay, and M. Stilkerich. 2017. The Perfect Getaway: Using Escape Analysis in Embedded Real-Time Systems. *ACM Trans. Embed. Comp. Syst.* 16, Article 99 (2017), 99:1–99:30 pages. Issue 4. DOI: https://doi.org/10.1145/3035542

[32] I. Stilkerich, M. Strotz, C. Erhardt, M. Hoffmann, D. Lohmann, F. Scheler, and W. Schröder-Preikschat. 2013. A JVM for Soft-Error-Prone Embedded Systems. In *2013 ACM SIGPLAN/SIGBED Conf. on Languages, Compilers and Tools for Embedded Systems (LCTES '13)*. ACM, New York, NY, USA, 21–32.

[33] M. Stilkerich, I. Thomm, C. Wawersich, and W. Schröder-Preikschat. 2012. Tailor-Made JVMs for Statically Configured Embedded Systems. *Concurrency and Computation: Practice and Experience* 24, 8 (2012), 789–812. DOI: https://doi.org/10.1002/cpe.1755

[34] N. Theißing, D. Merli, M. Smola, F. Stumpf, and G. Sigl. 2013. Comprehensive Analysis of Software Countermeasures Against Fault Attacks. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '13)*. EDA Consortium, San Jose, CA, USA, 404–409.

[35] I. Thomm, M. Stilkerich, R. Kapitza, D. Lohmann, and W. Schröder-Preikschat. 2011. Automated Application of Fault Tolerance Mechanisms in a Component-Based System. In *JTRES '11: 9th Int. W'shop on Java Technologies for real-time & embedded systems*. ACM, New York, NY, USA, 87–95.

[36] I. Thomm, M. Stilkerich, C. Wawersich, and W. Schröder-Preikschat. 2010. KESO: An Open-Source Multi-JVM for Deeply Embedded Systems. In *JTRES '10: 8th Int. W'shop on Java Technologies for real-time & embedded systems*. ACM, New York, NY, USA, 109–119.

[37] P. Ulbrich, R. Kapitza, C. Harkort, R. Schmid, and W. Schröder-Preikschat. 2011. I4Copter: An Adaptable and Modular Quadrotor Platform. In *26th ACM Symp. on Applied Computing (SAC '11)*. ACM, New York, NY, USA, 380–396.

[38] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. patel. 2004. Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline. In *34th Int. Conf. on Dep. Systems & Networks (DSN '04)*. IEEE, Washington, DC, USA, 61–70.

Received March 2017; revised June 2017; accepted June 2017