Program-Structure–Guided Approximation of Large Fault Spaces

Oskar Pusz, Daniel Kiechle, Christian Dietrich, Daniel Lohmann
Leibniz Universität Hannover, Germany
{pusz,kiechle,dietrich,lohmann}@sra.uni-hannover.de

***Abstract*—Due to shrinking structure sizes and operating voltages, hardware becomes more susceptible to transient faults. Fault injection campaigns are a common approach to systematically assess the resilience of a system and the effectiveness of software-based counter measures. However, experimentally injecting all possible faults to achieve full fault-space coverage is infeasible in practice. While precise pruning techniques, such as def/use pruning, already provide a significant reduction of the campaign size, the number of injections remains still challenging for even medium-sized systems.**

We propose *fault-space regions (FSRs)* as a method to approximately cover the complete fault space with a significantly lower number of required injections. Instead of probabilistic subsampling of the fault space, our approximation exploits the actual program structure and execution trace (e.g., flow of basic blocks) to identify injection points that are representatives for a larger set of faults. We identify such data-flow regions and inject only data values that flow across region boundaries. Thereby, we can further reduce the number of injections by up to 76 percent, while the results divert only by less than 2.7 percent from those of a complete and precise fault-injection campaign. Furthermore, we keep the locality of the results regarding silent data corruptions to a deviation of less than 6.9 percent.

***Index Terms*—reliability, functional correctness, single event upset, bit flip, fault injection, fault space approximation**

I. Introduction

Due to shrinking transistor sizes and operating voltages, transient hardware faults are an emerging challenge for safety-critical systems [8]. Functional safety standards, such as the automotive ISO 26262 standard [20, 21], take up this fact and recommend explicit measures to assess (and possibly mitigate) the effects of *single-event upsets (SEUs)* causing transient hardware faults (soft errors) [26] to the functional safety of the system. This is commonly done by performing extensive *fault injection (FI)* campaigns on the target system [2, 6] that try to mimic either the physical *causes* for SEUs (by exposing the system to, e.g., heat or radiation [14, 29]) or their *effects* (by changing logic signals). In this paper, we focus on injections of logic faults on ISA-level (i.e., bit flips in all software-accessible registers and memory), but the proposed concepts could be applied to other levels as well.

Every single injection within a FI campaign mimics an SEU and its impact on a concrete system's behavior. To reach full *fault space (FS)* coverage of all possible single-bit faults, one has, in principle, to inject each bit at every cycle of the application's execution. The resulting error-rate function would be complete and precise, but the FS to examine is prohibitively huge. However, it is possible to reduce the number of required injection experiments by pruning techniques, such as the well-known *def/use-pruning (DUP)* [34, 16, 13, 17]. In the paper, we present an approximative approach that, applied on top of DUP, can reduce the number of necessary injections even further at only very little costs regarding precision.

A. About This Paper

We present a program–structure–guided FI method to calculate, approximately, the resilience of a given program execution against bit flips. In a nutshell, we use the program's control-flow structure (e.g., jumps) in the fault-free execution trace to form *fault-space regions (FSRs)* as consecutive instruction sequences. For each region, we inject those data flows that cross region boundaries as they transfer computation results from region to region. By assigning appropriate weights to each region and each performed injection, we derive an approximation of the actual error-rate function for the given program, which would include inter- and intra-region data flows.

Figure 1 illustrates our idea and shows the control- and data-flow, as well as a fragment of the fault space, for the inner loop of a CRC32 checksum. Here, the jump at the end of the loop acts as a region boundary such that each loop iteration becomes one FSR. In the fault space, we see the fault-equivalence sets and their representative faults as they are produced by DUP, which here would require ten FIs. With our method, we avoid the injection of the intermediate results within the loop body by executing only three FIs in values that

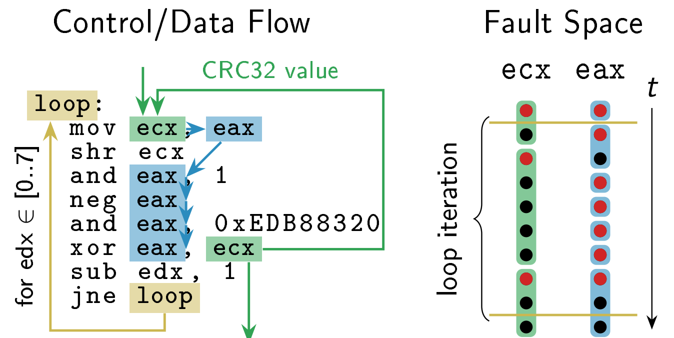


Fig. 1: Program structure for the inner loop of a CRC32 implementation. The corresponding fault-space fragment is partitioned with def/use pruning into equivalence sets, and we have to inject all representative (red) faults for a complete coverage.

cross the region boundaries. Since we assume that a fault within a FSR manifests in an erroneous region-crossing data flow, we thereby concentrate our injection efforts on the neuralgic spots of the program execution without jeopardizing the precision of the resilience investigation.

In particular, we claim the three following contributions:

- We introduce the *fault-space region (FSR)* concept, which uses the program structure and data flow in an execution trace, to drastically reduce the number of fault injections.
- We present two instantiations of the FSRs concept using basic blocks as well as function calls, and extend the available FAIL* [32] FI framework to support FSR.
- We evaluate our approach with established benchmarks (MiBench [15]) and demonstrate that our approximation of the error rates is comparable to a precisely-covered fault space.

In the following, we first describe our fault- and system model in Section II and then present FSRs, which is our core contribution, in Section III. We evaluate our approach with a set of benchmarks from the MiBench suite in Section IV and discuss some general findings in Section V. In Section VI we summarize the related work and finally conclude the paper in Section VII.

II. Fault Model

We want to investigate on the resilience of a program in the presence of transient hardware faults (soft errors) [26], which arise from SEUs caused by radiation, electromagnetic interferences or other environmental influences. These influences manifest within the system, in combinatorial logic, registers, and memory cells [5, 19, 33] and surface, at some point, as bit flips at the hardware/software boundary.

The *fault model (FM)* determines the type, location, time, and probability of the considered faults. In this work, we use three FMs with uniformly-distributed single-bit flips that happen in between instructions in (1) the processor's general purpose registers, (2) the main memory, and (3) both registers and memory. In combination with a concrete, fault-free execution (golden run) of the program-under-test, we derive the rectangular *fault space (FS)*, with time and location axis, as the set of all possible faults. In Figure 2, we show a fault space that spans over 12 instructions and 4 bits, and contains 48 possible faults (squares), which are equally likely.

We use these FMs as examples, although our approach is applicable to a wider range of models and spaces as long as: (1) the time axis is synchronized with the program execution. (2) the number of fault locations is fixed and known. (3) the probability p_f of each fault is known.

With a single FI, we flip a bit of data at the given time and in the given location in a deterministic re-execution of the program and classify the subsequent error and failure behavior of the program by using the golden run as a reference. Therefore, fault injection is a function that maps faults to *error classes*, like "benign error", "detected error", or "silent data corruption".

$$\mathcal{E}_T(C) = \sum\{p_f | f \in FS, \quad FI(T, f) = C\}$$

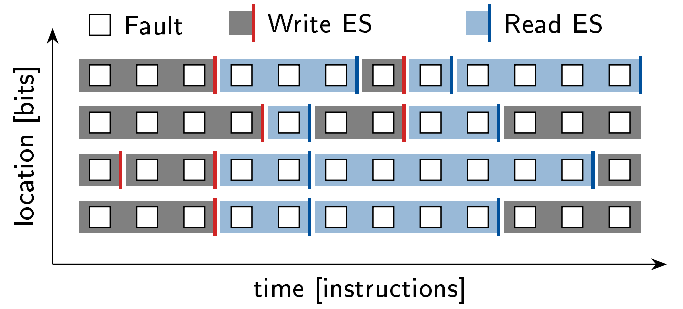


Fig. 2: Fault space with DUP equivalence sets

Goal of a FI campaign is to deduce the combined *error-rate function* $\mathcal{E}$, which gives a probability for each error class C that such an error occurs during a program execution T. For our uniformly-distributed single-bit fault model, it is sufficient to give and compare error-classification counts instead of accumulated probabilities [30] as they differ only in a constant multiplicative probability factor (p_f).

If a FI method is *complete*, it derives an error classification for each member of the fault space. This can either be done by injecting every fault or by interpolating classifications from a partial covering. If a FI method is *precise*, the error function perfectly represents the error rates for a concrete fault space. In this classification, our method is complete but not precise as we only provide an approximation $\widetilde{\mathcal{E}}$ of the error-rate function, but at a much lower number of required fault injections. For this, we sample faults and weight the results according to the program structure.

III. Program-Structure-Guided Approximation

We introduce the concept of *fault-space regions (FSRs)*, a program-structure-guided fault-space approximation method to reduce the number of required FIs.

The standard technique to reduce the required FIs is *def/use-pruning (DUP)* [34, 16, 13, 17], which reduces the FS up to five orders of magnitude [4]. This operation principle is shown in Figure 2: In the rectangular FS (location $\times$ time), we consider each location (rows) in isolation and split the time-axis aligned sequence of faults at read/write accesses into partitions (see Figure 2). These partitions are *equivalence sets (ESs)* with regard to their error classification as a fault can only become effective at the next accessing instruction. ESs that end in a write access will always result in a benign error (gray), because every fault within an ES will be overwritten by the write access. Read ESs can lead to an actual erroneous behavior and manifest in the system. It does not matter in which instruction of the ES a fault is injected, because the data is not used until the read access and every fault within a read ES lead to an equivalent behavior. So, for each read ES, we have to inject only one representative fault. We call the set of all read ESs the *effective fault space*. The DUP is a precise and complete FI method if we inject all representative faults.

Despite using DUP, a FI campaign can still be infeasible due to a too large number of representative FIs. Therefore,

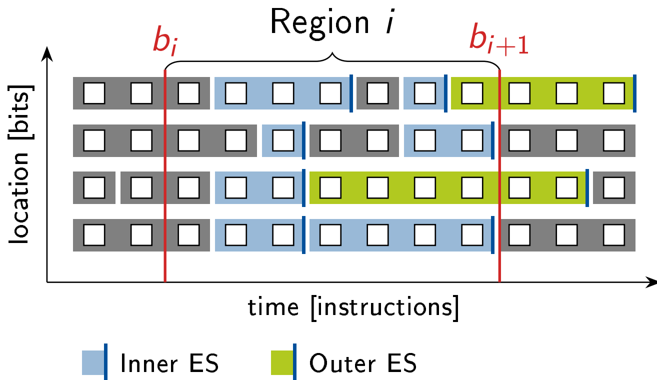


Fig. 3: Fault space with a fault space region.

different heuristics [17, 24] were proposed to reduce the number of necessary FIs even further. While our method also falls in this category, we base our approximation of error-rate function on the DUP technique i.e., its ES and we deduce the required injections using program structures from the program trace. Thereby, the FSRs concept is application-tailored, based on a precise pruning method, and provides a complete but imprecise FI method. We start out by explaining the FSRs concept in general, before we present two possible instantiations that use basic blocks and call/return instructions as the guiding program structures.

A. The Fault-Space–Region Concept

For our approach, we partition the time axis of the FS into continuous sequences of instructions. The fault model's information about the available locations or bits define the formation of FSRs. We derive the FSR border from the fault-free execution trace, which is the recorded sequence of *instruction pointers (IPs)* for the investigated program path. From the trace, we extract program structures (e.g., basic blocks, function scopes or module boundaries) and use them as a guide to portion the time-axis. Figure 3 shows exemplary two FSR borders at b_i and b_{i+1}.

DUP results in the previously mentioned effective ESs of different lengths. With the FSRs, we select all of the effective ESs, that start within a given FSR. We decide for every element of an FSR if it is an *inner* ES, which does not extend beyond the next FSR border (b_{i+1}), or if it is an *outer* ES, which transport computation results from this FSR into a following one. In Figure 3, all ineffective (write) ESs were removed (gray) and only the effective ESs for the FSR i are shown; inner ESs are filled in blue, the two outer ones are drawn green.

For the FI campaign, we only execute injections for the outer ESs and weight their results to get an approximation of the error-rate function. With this weighting, we compensate for the influence of the omitted inner injections and interpolate from the classification results of the outer ES to a classification for all effective ESs. Our rationale for this focus on the outer injections is that faults in inner sets have to manifest in an outer ES, which carries the error to other FSRs where it can become a failure. In essence, we treat each FSR as a black box and only inject faults into its outputs.

One key component of the FSR approach is the weighting function for the injection results of outer ESs. First, we define the weight w_r of an FSR$_r$ as the sum of all cardinalities of covered (inner and outer) ESs. Please note that the sum over all FSR weights is equal to the sum of all ES weights if we only apply DUP. We distribute the FSR weight over its outer ESs set E_r and their injection results j_e of every outer ES $e \in E_r$. In our experiments, we developed two different methods that distribute the weights uniformly (w_{mean}) or rather according to the relative DUP weight within the FSR (w_{wmean}):

$$w_{\text{mean}}(j_e) = \frac{1}{|E_r|} \cdot w_r \quad e \in E_r$$

$$w_{\text{wmean}}(j_e) = \frac{|e|}{w_{r,\text{outer}}} \cdot w_r \quad e \in E_r \quad w_{r,\text{outer}} = \sum_{e_k \in E_r} |e_k|$$

The first one is the mean-like function $w_{\text{mean}}(j_e)$ that uniformly distributes the FSR weight w_r uniformly over all $|E_r|$ outer ESs. As a refinement, we developed $w_{\text{wmean}}(j_e)$ as a second weighting method, which considers the weight $|e|$ of the outer ES e with respect to the other outer ESs. For this weighted mean, we sum up ($w_{r,\text{outer}}$) the cardinalities of the region's outer ESs E_r. Thereby, injections with a higher DUP weight also get a higher weight with the w_{wmean} function, where a higher weight with the w_{mean} function is evenly distributed over all outer injections.

With this FSR concept, we have the tools to inject only a subset, namely the outer ESs, of all representative faults determined by DUP and calculate an approximation of the error-rate function by weighting the results. Without the ESs of DUP, each instruction is considered as an isolated ES and none of the ESs would ever cross a border between FSRs, which means that not a single FI will be executed. Thus, the formation of ESs is a necessary condition for the usage of FSRs. However, we still have defined how we derive FSR borders from the execution trace. While the FSR concept can be combined with different FSR-forming methods, we will present two instantiations based on the control-flow structure of the program in the following.

B. Basic-Block and Call Regions

The (control- or data-flow) structures of a program are static properties that are most relevant in the context of compilers or static-program analysis. We take this term from the static world and apply it to the execution trace, which is one control-flow path through the program. By analyzing the trace, we deduce *basic-block regions (BBRs)* and *call regions (CRs)* as two different granularities of FSRs.

a) Basic block regions: For a compiler, a *basic block (BB)* is a single-entry–single-exit region of machine instructions that is only entered at the first instruction and only left at the last

one. Thereby, the inner instructions of a BB are branch free, and they are never target of a jump or call instruction. As BBs are the building block for the control-flow graph, they are one of the smallest structuring elements of programs that is larger than an individual instruction.

Instead of extracting the BBs from the binary itself or getting it from the compiler, we deduce *dynamic* BBs from the program trace. First, we want to find all instructions that start a new BB. For this, we iterate over the trace and find all changes to the control flow where instructions are not executed directly in sequence (i.e., jumps, calls). All instructions that follow such a change in the trace and the static successor of the jump instruction start a new BB and thus an FSR. In a second run over the program trace, we use the set of all FSRs and determine the outer ESs as well as the resulting FIs.

Please note that our dynamic BBs are sometimes larger (but never smaller) than the actual BB that are produced by the compiler. This happens, if the execution from a basic block directly falls through to the successor block and the trace contains no explicit jump to the successor (e.g., a while loop with one loop iteration).

b) Call regions: For *call regions (CRs)*, we use the BBR construction but only consider control-flow changes that stem from call or return instructions as FSR borders. Thereby, we end up with larger FSRs, which increases the number of inner ES even further. For example, the execution of a leaf function, which calls no other functions, becomes a single FSR. The resulting CRs also is the compound of all BBRs between call and/or return instructions.

IV. Evaluation

For the evaluation, we apply *basic-block regions (BBRs)* and *call regions (CRs)* to seven benchmarks from the MiBench [15] benchmark suite. We use the classical DUP technique as a baseline and compare the FSR method in terms of required fault injections and the relative deviation we get for the error-rate function $\mathcal{E}_T$. This comparison is done in three different FMs (memory only, register only, both) and with both presented weighting functions. At the end of the evaluation, we examine whether the regions can keep the locality of the results despite deviations.

A. Experimental setup

For the fault injection, we use the simulation-based FI framework FAIL* [32], which extracts program traces, performs the DUP, and simulates the representative faults using the Intel IA-32 simulator Bochs. We extended tooling to support the BBR and the CR instances of the FSR concept.

Due to its client-server-architecture and the independence of the injections, we performed the necessary fault injections on a cluster system using 17 Intel Xeon X5650 @ 2.67 GHz (12 cores each). The DUP and the FSR calculation were performed on an Intel i5-7400 @ 3 GHz (4 cores).

B. Evaluation scenario

To evaluate FSRs we use seven selected programs of the automotive and security branch of the benchmark suite MiBench [15] and executed them with a reduced input size. We choose the automotive branch, since FIs are recommended to be performed for safety analyses and certifications in the automotive sector [20]. For this branch, we use *bit count (BC)* and *quick sort (QSORT)* and omitted the *basicmath* benchmark, since FAIL* does not yet support injections into floating-point registers. Furthermore, we also skipped the *susan* benchmarks (image processing), since failure behavior of image recognition is rather gradually, due the amount of redundancy within the image, than classifiable with a few error classes. From MiBench's security branch, we choose the encryption algorithms *blowfish encode (BFE)*, *blowfish decode (BFD)*, *rijndael encode (RDE)*, *rijndael decode (RDD)* and *sha1 (SHA)*. These benchmarks are interesting since every single bit is important for the correct execution of the encryption and therefore it is also challenging for the FSRs concept. We consider these benchmarks to be sufficient and decided to omit the benchmark *PGP* as it has similar properties in comparison to the other security benchmarks.

For the error classification after the fault injection, we used five error classes: (1) benign error; the program recovered and produced the correct result. (2) silent-data corruption; the program terminated but produced an erroneous output. (3) timeout; the program took significantly longer than the fault free execution. (4) trap; the processor reported a trap during the execution. (5) write text segment; the program tried to write to the read-only code section.

We evaluate the FSR concept with three uniformly-distributed single-bit fault models as described in Section II: (1) Memory; each bit in memory can flip between two instructions. (2) Register; each bit in each of the eight general-purpose registers of an IA-32 processor can flip. (3) Combined; combination of memory-only and register-only FM where the rows of both are stacked to form a combined FS. By using these models, we can make predications about injections into register and memory data separately as well as about the interplay of the combined usage patterns.

For the Register FM, we excluded the control-flow–related registers (i.e., program counter and flags) on purpose: Since the PC register changes with every instruction, its ESs have always only one element and therefore can never cross a FSR border. Similar to this, during a normal program execution, the flags register on IA-32 is mainly used to store comparison results shortly before the final branch instruction of a basic block. Therefore, ESs of the flags register end with the FSR border. Furthermore, we excluded floating-point registers, since the FAIL* platform is currently unable to inject the floating-point unit (FPU). Nevertheless, with the Memory FM, we still cover all FPU input and output values when they are read from or written to the main memory.

fault model		Memory	Register	Combined
#injections	BBR[%]	−9.79	−83.73	−69.91
	CR [%]	−38.03	−95.44	−83.87
Deviation δ	BBR[%]	19.80	14.01	31.92
[mean]	CR [%]	20.54	37.73	20.93
Deviation δ	BBR[%]	0.04	16.87	0.91
[wmean]	CR [%]	1.79	31.74	3.66

TABLE I: Overview of the reduced number of required fault injections and the deviations δ [geo. mean] in the classification for both presented weighting functions w_{mean} and w_{wmean}.

C. Reduced number of FIs

In the following, we characterize our benchmarks, quantify the reduction of required FIs, and calculate the approximation error introduced by our FSR concept if compared with a precise and complete DUP injection campaign. Due to different run times and error-classification distributions of the benchmarks, we calculate the relative deviations for each error class separately and use the geometric mean δ over these error-classification deviations as the metric for the FSR accuracy. Thereby, we avoid that a long-running benchmark or a dominating classification distort the results. Calculating the FSRs themselves took end-to-end no more than 4.1 seconds.

1) Overview: The number of instructions, BBRs, and CRs are presented in Table II; they are valid for all FSs, because the same program trace was used for all FSs. Table I shows the percental reduction of required FIs and the deviations δ for the precise error-rate function $\mathcal{E}_T$. We present the results for the two possible weight functions w_{mean} and w_{wmean} for FSRs. Overall, we could reduce the number of FIs by up to 95 percent with a deviation of 38 percent (register-only FM, CR, w_{mean}). The lowest deviation of only 0.04 percent was achieved with BBRs for the memory FM, but resulted only in a FI reduction of nearly 10 percent. In total, the w_{mean} function is more inaccurate and results in at least 14 percent deviation. Therefore, in the following, we only show results for the weighted-mean function w_{wmean} in our detailed examination of the results for the individual benchmarks and the three FSs.

2) Memory-only FM: Table II shows the results of our approximation method for the memory-only FM. For SHA and RDE, BBRs causes its maximal deviations of 0.19 and 0.17 percent under a reduction of FIs of 22.72 and 34.55 percent; the other deviations caused by BBRs are negligible. Furthermore, the #injection reduction of BBRs ranges from a little under 1 percent (BFE) to 35 percent (RDE). The minor reduction observed for the blowfish benchmarks (BFD, BFE) is caused by many large ESs that cross region borders. In such cases, where memory is not used for intermediate results, BBR injection converges to the DUP method.

For CRs, we see the largest deviation of around 10 percent for the SHA benchmark accompanied by the drastic FI reduction of 23 percent. This is caused by the small number of functions calls, which leads to large but few FSRs. It shows that CR are not practical if a program contains a small amount of function calls. Besides SHA, CRs reduce the number of FIs at least 25 percent (up to $44\,\%$) with deviations ranging between 0.15 percent and 6.27 percent.

Summarized, for the memory-only FM, CRs (except SHA) and especially BBRs provide significant reductions at moderate approximation errors.

3) Register-only FM: For the register-only FM, the results paint a different picture (Table III). First of all, we see that the register FS is three orders of magnitude smaller than the memory FS and it is much more densely packed with short read ES, as can see from the minor FS reduction by DUP. This is caused by the small number of registers, compared to the number of touched memory cells, and the short retention time of register data.

Over both methods, we can save at least 72 percent of all injections at the price of much higher deviations up to 41 percent for BBRs (RDE) and up to 109 percent for CRs (SHA). Only for the arithmetic intense BC benchmark we get a good approximation. These huge deviations are caused by a common register-usage pattern that conflicts with our assumption that we capture FSR computation results by selecting the border-

Memory-only FM			BC	BFD	BFE	QSORT	RDD	RDE	SHA
FS Size	time axis	[instr.]	112 848	56 141	55 444	46 240	71 318	71 411	41 113
	total	[$\cdot 10^9$ faults]	2.23	3.07	3.04	2.77	6.25	5.59	0.99
	effective	[$\cdot 10^9$ faults]	0.68	1.93	1.93	1.66	3.56	3.56	0.26
#regions	BBR		21 405	5850	5031	7436	4049	4215	5644
	CR		8818	434	436	1010	415	504	67
#injections	DUP	[$\cdot 10^6$]	1.04	0.57	0.57	0.52	1.10	1.10	0.50
	BBR	[%]	−14.66	−1.51	−0.91	−15.49	−35.10	−34.55	−22.72
	CR	[%]	−24.75	−25.21	−25.42	−44.26	−42.55	−41.40	−92.98
Deviation δ	BBR	[%]	0.117	0.004	0.002	0.062	0.123	0.168	0.186
	CR	[%]	0.147	2.475	2.655	0.998	6.270	0.942	10.232

TABLE II: Memory-only FM. Detailed statistic for the memory-only FM for seven MiBench benchmarks. With DUP, we reduced the total fault space to the effective fault space by excluding FIs directly before write accesses. The number of injections refer to the pruning method *def/use-pruning (DUP)* as well as our approximations *basic-block region (BBR)* and *call region (CR)*. The deviation δ is the geometric mean over the relative error-classification deviations. The number of instructions (time axis) and the number of regions are valid for all fault spaces.

Register-only FM			BC	BFD	BFE	QSORT	RDD	RDE	SHA
FS Size	total	$[\cdot 10^6]$	32.50	16.17	15.97	13.32	20.54	20.57	11.84
	effective	$[\cdot 10^6]$	15.68	11.90	11.32	8.38	14.35	13.15	7.83
#injections	DUP	$[\cdot 10^6]$	4.00	2.70	2.65	2.19	2.83	2.86	2.01
	BBR	[%]	−72.18	−86.60	−89.43	−73.96	−92.21	−92.46	−81.87
	CR	[%]	−82.32	−97.82	−97.80	−95.79	−97.89	−98.04	−99.62
Deviation δ	BBR	[%]	3.67	25.41	12.59	12.33	23.76	41.30	27.39
	CR	[%]	3.17	34.28	56.78	24.59	40.45	48.57	108.77

TABLE III: Detailed statistic for the register-only FM

Combined FM			BC	BFD	BFE	QSORT	RDD	RDE	SHA
FS Size	total	$[\cdot 10^9]$	2.26	3.08	3.05	2.79	6.27	5.61	1.00
	effective	$[\cdot 10^9]$	0.70	1.95	1.94	1.67	3.57	3.58	0.26
#injections	DUP	$[\cdot 10^6]$	5.04	3.27	3.22	2.71	3.93	3.96	2.51
	BBR	[%]	−60.31	−71.71	−73.75	−62.75	−76.28	−76.37	−70.02
	CR	[%]	−70.43	−85.11	−84.97	−85.91	−82.45	−82.30	−98.29
Deviation δ	BBR	[%]	0.17	1.16	1.14	0.33	2.66	1.73	1.48
	CR	[%]	0.18	7.34	6.43	2.88	9.37	3.33	11.48

TABLE IV: Detailed statistic for the combined FM (memory + register)

crossing ESs. While intermediate results are calculated in registers, they are written to memory before the region is left (e.g., caller-saved registers for the CRs). Thereby, the relevant data flows happen within the FSR and the results bypass our method in memory. We therefore conclude that FSRs are unsuitable for a register-only FM if intermediate results are mainly communicated through main memory.

4) Combined FM: Caused by our experience for the register-only FM, we decided to apply FSRs also to the combined FM. This FM provides the resilience investigator with a more realistic picture for a whole system in a harsh environment as it considers faults in all relevant storage and computation components (processor, memory, and indirectly also caches). Table IV shows the results of the combined FM.

With BBRs, we end up with a maximum deviation of 2.7 percent (1.24 % on average). At this cost, we get a reduction of required injections that ranges between 60 percent and 76 percent (70.17 % on average). For the coarser CR method, the average deviation increases to around 5.9 percent but the FI reduction is at least 70 percent and goes up (excluding SHA) to 86 percent. Again SHA proves difficult because of the low number of function calls. Summarized, we see that BBRs well suited for the combined FM and CRs when a bit higher deviation is accepted.

D. Locality of the Results

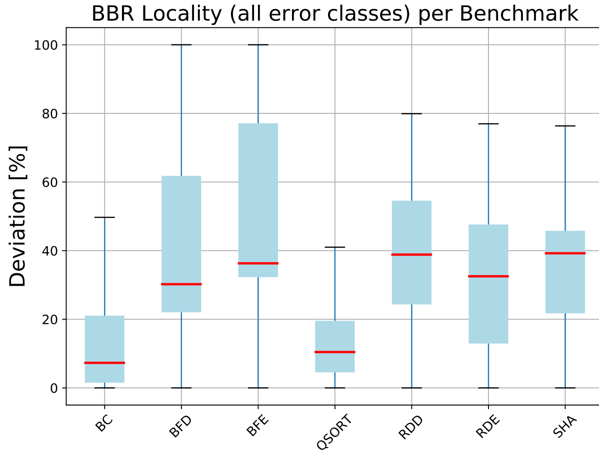
For a resilience assessment, often not only the end-to-end error rate is of interest, but also a comparison of different program phases or sections. For example, with detailed FI results, we can make an informed choice which function requires additional software-based hardening measures, like triple-modular redundancy.

With FSRs, we coarsen the focus point of the FI from individual instructions to larger code regions. Therefore, we are no longer able to distinguish between two neighboring instructions, but we can only calculate local error rates for each FSR. This is still possible since we approximate, for each region individually, from the injection results of the outer ESs to all ESs in the region. Furthermore, since FSRs are formed according to the program structure and since they partition the execution trace on the time axis, we can still distinguish between different program phases and code blocks. For example, with BBRs, we can still judge if the first execution of a loop body is more vulnerable than the last one. In the following, we will investigate how well the FSRs method is able provide local results and if this locality is sensitive for a specific benchmark, fault model, or error class. Since CRs already showed a higher approximation error on the whole-program level, we focus only on BBRs.

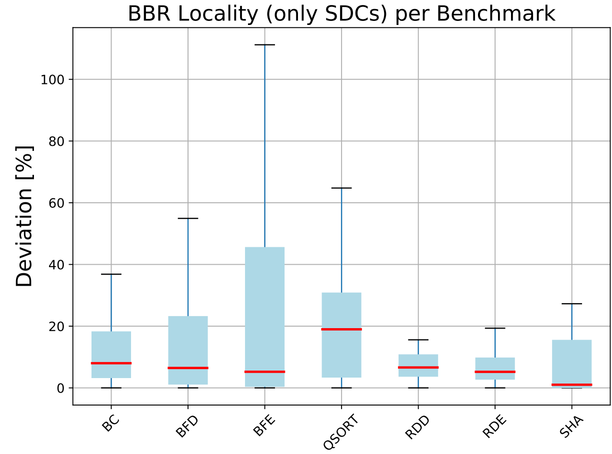
For each BBR, we calculate two error-rate vectors, a precise and an approximated one, that hold the results for each of our five error classes: As precise baseline, we use the FI results for the inner *and* the outer ESs and weight them according to their ES size. For the approximated one, we weight the results of the outer ESs according to w_{wmean}. We calculate the relative deviations, element by element, and use the geometric mean to summarize the BBR-local precision.

1) Benchmark- and Fault-Model Sensitivity: First, we want to look at the sensitivity of the local results with respect to different benchmarks and the three used fault models. In general, we expect that a higher precision on the whole-program level (Table II-IV) also translates into better local results.

For the register FM, which already proved to be already difficult on the whole-program level, our method shows large deviations if looking at individual BBRs: We end up with median deviations that range from 20.46 percent (BC) up to 88.82 percent (SHA). The 75-percent quantile even goes up to 100 percent deviation from the actual result. Again we see, that our FSR approach is less suited for a register-only FM as



(a) Boxplots of all benchmarks over all error classes. Each data point is the geometric mean over the relative deviations of the local error-rate vector for one BBR.



(b) Boxplots of all benchmarks over the SDC error class. Each data point is the relative deviation of the local SDC error rate for one BBR.

Fig. 4: Local-result deviations for the combined FM. he red line is the median, the boxes represent the 0.25 and 0.75 quantiles, and the whiskers refer to the last value within 1.5 times the interquartile distance. We omitted the outliers for clarity reasons.

local results can bypass the region borders in memory.

For the combined FM, we show the boxplot over the BBR-result deviations in Figure 4a. Overall, we see that the median deviations always stay below 40 percent with a 75-percent quantile up to 77 percent (BFE). Thereby, good results in the whole-program approximation (Table IV) also translate into more precise results for the local error-rate function: BC and QSORT show a median deviation of 7.3 percent and 10.44 percent.

For the Memory FM, we observed, as already predicted by the whole-program approximation, the most precise local results: Here, all benchmarks showed a median relative error of zero and only three benchmarks had a 75-percent quantile unequal to zero: BC and QSORT stayed below 0.33 percent and only SHA had a 13 percent deviation in 0.75-quantile. Therefore, we conclude that our BBRs is not only well suited for whole-program approximation but also provides precise local results for the Memory FM.

2) Error-Class Sensitivity: Besides benchmark and fault-model sensitivity, we were also interested into the sensitivity of BBR-local results towards different error classes, since our evaluation metrics were not error-class separated until now. However, in reality, some error classes, especially the *silent-data corruption (SDC)* class, are more important to judge the resilience of a program phase or code block. Therefore, we took a deeper look into the local deviation results for the combined FM, since this model provides the highest coverage of the program behavior.

In Figure 5, we show the boxplots over the relative error-rate deviations over all benchmarks and grouped by error class. We see that benign faults, where the system recovered from the injection, and invalid text-segment writes, show the highest amount of imprecision (median: $> 20\,\%$). From this higher

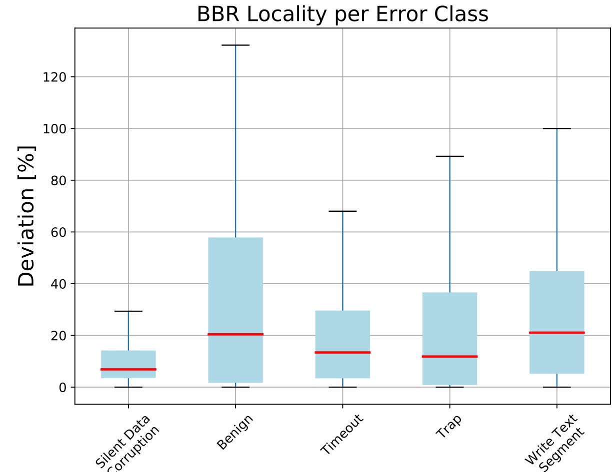


Fig. 5: Boxplots of all error classes containing data points from all benchmarks for the combined FS.

imprecision, we can speculate that such outcomes arise more often from injections within a BBR. On the other hand, we see that our BBR approximation provides the best results for the most important error class: SDC. Here, the median is 6.88 percent and the 75-percent quantile keeps below 12 percent.

With this in mind, Figure 4b focuses only on SDCs and shows the per-benchmark quality of BBR local results. In essence, this figure is the same boxplot as Figure 4a but filtered for the SDC class. We see that the median precision for local SDC rates is higher for all benchmarks, with the exception of QSORT. The medians range between 1.03 percent for SHA, 8 percent for BC, and 18.98 percent for QSORT; 0.75 quantiles

are between 9.65 percent (RDE) and 45 percent(BFE). Since local results for the SDC class are (most of the time) more precise than for other error classes, we conclude that especially those data flows that result in a SDC cross a BBR border and are captured by our method.

From our investigation on local error rates, we see that BBRs approximation is able to provide a high degree of locality. Especially for SDCs, which are the most critical error class, our approximation provides low deviations from a precise FI that covers the entire FS, but at much lower injection costs.

V. Discussion

1) Influence of the ISA: In our experiments, the FSR concept led to a large approximation error for the register-only FM because intermediate results do not cross the region border in a register, but in main memory. This conflicts with our premise that all values that transport data from region to region are visible at the region border. For the combined FM, we could avoid this as the bypass channel was also covered.

This bypass effect is highly dependent on the ISA of the executing processor. In our evaluation, we used the Intel IA-32 architecture, which is known for its high *register pressure* as it has only 8 general-purpose registers. This forces the compiler to spill intermediate results to memory instead of keeping them in a data register. Hence, the applicability of FSRs on register-only FMs is dependent on the ISA. We can assume it will improve with a higher number of register if the compiler exploits them properly. Using architectural properties to improve the FSR's accuracy for the register-only FM is a topic of further research.

2) Determining Borders: In the paper, we proposed to extract the program structures that guide the FSR construction from the program trace. However, other sources for the region borders are possibly also available. For example, as the compiler produces all jump targets in a program, we could use this static information instead of extracting dynamic basic blocks from the trace. However, such a static method would have to cope with dynamic branch targets or memory-indirect jumps that are unknown at compile time.

Therefore, we think that our approach of extracting the necessary information from the fault-free execution trace, which definitely covers all required program structures, is preferable. Nevertheless, it might be beneficial to enrich the trace-derived region formation by static information from the compiler (or a static analysis tool).

3) Limitations of the Approach: Our approach is limited to FI scenarios for which a fault-free execution trace is available and DUP is applicable. For example, if faults should be injected in a continuously running system, we cannot prepare a list of representative faults as no deterministic re-execution of the system is possible. However, for such scenarios, the general undertaking to calculate the error-rate function by a complete coverage of the fault space is impossible.

The applicability of our approach is limited, if the chosen program structures are not present. This was the case for the SHA benchmark with CR, as a call-region program structure is mainly absent in the binary. Similar situations can arise for BBRs, if the compiler emits code using instruction predicates or straight-line code. Here, we cannot detect jumps, although some kind of control flow is still performed.

In a same direction, the FSR approach is sensitive to some compiler optimizations that reduce the number of regions. For example, with loop unrolling, a small loop body is concatenated several times into one large basic block. In this case, our BBR method would only inject data that flow in and out of the loop, but no data-flow between loop iterations is considered. To mitigate this, static information about the program structure could be used (e.g., from the compiler-generated debug information).

4) Generalizability: We performed the formation of FSRs on the ISA level, where faults (virtually) only happen between instructions in user-visible registers and memory. However, we could also apply the concept of FSRs to fault models on other abstraction levels. For example, we could apply FSRs to a flip-flop–level fault model, where we inject the processor on the gate level. Flip-flops define the location axis of the FS and cycles the time axis. For FSRs to work, we only need to form ES for each flip-flop and define region borders from the trace of flip-flop values.

We could also generalize FSRs to higher abstraction levels, such as a program executed by a language interpreter. Here, the set of all language variables become the location axis and the executed statements the time axis. In this scenario, we also could extract information about the program-structure directly from the interpreter as it has to be available anyway.

VI. Related Work

As covering large FS is often infeasible in practice, several methods were proposed to reduce the number of required fault injections. The methods can be categorized in terms of *completeness* and *precision* as defined in Section II.

Using the precise and complete DUP, which was proposed several times [7, 17], already reduces the number of injections significantly. Most similar to our FSR approach are methods that group faults using structural characteristics with *similar*, instead of equal error, behavior by data-structure dependencies [11], address bounds [28], or memory states [18]. While these methods also use program properties, they only consider and prune a special section of the FS in comparison to the FS relevant for the complete program.

Without using coarser-grained program structures, sampling heuristics are used to approximately cover the FS [27, 22] or concentrate on the most important faults [10]. However, these pruning techniques are coarser grained and make no complete statement about the FS and are, as every sampling method, not precise. The *Relyzer* tool [17] as like as *SmartInjector* [24] provide heuristics that combine multiple ESs into compounds and inject one FI of this compound only. The disadvantage is the inflexibility regarding the result accuracy and the complexity of the analysis steps, as they mainly focus on *silent data corruptions*. However, in many cases, other error classes become more relevant [9, 25]. With the *fault-similarity*

heuristic [31], machine-learning techniques, with the recorded machine state at the fault location as training data, were used to avoid injections of similar faults, but is not reproducible at all. In contrast, forming FSRs is flexible, reproducible, the accuracy depends on the weight function only, and it works independent of the error classification.

Besides the injection-based resilience assessments, different methods [1, 3, 35, 11, 12] estimate the program reliability by determining different vulnerability factors. For this, they statically combine information about the program execution, about the program structure, and even about the processor architecture. However, these vulnerability factors provide no quantitative classification of actual occurring errors, neither on the whole-program nor on a region-local level. Furthermore, since these methods analyze the program without executing the program, they provide neither a complete nor precise picture of the fault-space. The tool Trident [23] also uses sequences of instructions to split a program for determining the SDC probability. They provide an injection-free heuristic based on the propagation of faults between these sequences, their branches and memory dependencies. However, this tool is similar to (e)PVF [35, 12] and is, due to branch probabilities and pruned data dependencies, also neither complete nor precise.

VII. Conclusion

With FI, we can quantify a program's resilience against transient hardware faults in the process of assessing the functional safety of a critical system. For this, we inject different faults into a large number of program re-executions and classify the following erroneous behavior. However, the number of required faults quickly rises and even with precise reduction methods, like *def/use-pruning (DUP)*, complete and precise *fault space (FS)* coverage is not feasible.

Therefore, we presented the *fault-space region (FSR)* concept as an approximation method that reduces the number of required FIs while the deviation in the error classification keeps relatively low. We form these FSRs from program structures (i.e., basic blocks and call instructions) by analyzing the program trace and inject faults only in those data flows that cross a region boundary. In the evaluation, we compare the number of required faults between the DUP technique and our FSR concept and quantify the approximation error between both if applied to three different fault models (general-purpose registers, main memory and both at once). For seven MiBench benchmarks from the automotive and the security category, we could reduce the number required faults, on average, for the combined FM, by 76 percent at a relative approximation error of under 2.7 percent. Furthermore, we keep the locality of the results regarding silent data corruptions to a deviation of less than 6.9 percent.

References

[1] J. Aidemark, P. Folkesson, and J. Karlsson. "Path-based error coverage prediction". In: *Proceedings Seventh International On-Line Testing Workshop*. 2001, pp. 14–20. DOI: 10.1109/OLT.2001.937811.

[2] Jean Arlat, Martine Aguera, Louis Amat, Yves Crouzet, Jean-Charles Fabre, Jean-Claude Laprie, Eliane Martins, and David Powell. "Fault Injection for Dependability Validation: A Methodology and Some Applications". In: *IEEE Transactions on Software Engineering* 16.2 (Feb. 1990), pp. 166–182. ISSN: 0098-5589. DOI: 10.1109/32.44380.

[3] Ghazanfar Asadi and Mehdi Baradaran Tahoori. "An analytical approach for soft error rate estimation in digital circuits". In: *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on*. IEEE. 2005, pp. 2991–2994.

[4] Raul Barbosa, Jonny Vinter, Peter Folkesson, and Johan Karlsson. "Assembly-Level Pre-injection Analysis for Improving Fault Injection Efficiency". In: *Dependable Computing - EDCC 5*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 246–262. ISBN: 978-3-540-32019-7. DOI: 10.1007/11408901_19.

[5] Robert C Baumann. "Radiation-induced soft errors in advanced semiconductor technologies". In: *IEEE Transactions on Device and Materials Reliability* 5.3 (2005), pp. 305–316.

[6] Alfredo Benso and Paolo Ernesto Prinetto. *Fault injection techniques and tools for embedded systems reliability evaluation*. Frontiers in electronic testing. Boston, Dordrecht, London: Kluwer Academic Publishers, 2003. ISBN: 1-4020-7589-8.

[7] L. Berrojo, I. Gonzalez, F. Corno, M. S. Reorda, G. Squillero, L. Entrena, and C. Lopez. "New techniques for speeding-up fault-injection campaigns". In: *Design, Automation & Test in Europe Conference & Exhibition 2002 (DATE '02)*. Washington, DC, USA: IEEE Computer Society Press, 2002, pp. 847–852. DOI: 10.1109/DATE.2002.998398.

[8] C. Constantinescu. "Trends and challenges in VLSI circuit reliability". In: *Micro, IEEE* 23.4 (July 2003), pp. 14–19. ISSN: 0272-1732. DOI: 10.1109/MM.2003.1225959.

[9] Björn Döbel, Horst Schirmeier, and Michael Engel. "Investigating the Limitations of PVF for Realistic Program Vulnerability Assessment". In: *Proceedings of the 5th HiPEAC Workshop on Design for Reliability (DFR '13)*. Berlin, Germany, Jan. 2013.

[10] Mojtaba Ebrahimi, Nour Sayed, Maryam Rashvand, and Mehdi B Tahoori. "Fault injection acceleration by architectural importance sampling". In: *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2015 International Conference on*. IEEE. 2015, pp. 212–219.

[11] Mojtaba Ebrahimi, Mohammad Hadi Moshrefpour, Mohammad Saber Golanbari, and Mehdi B Tahoori. "Fault injection acceleration by simultaneous injection of non-interacting faults". In: *Proceedings of the 53rd Annual Design Automation Conference*. ACM. 2016, p. 25.

[12] B. Fang, Q. Lu, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi. "ePVF: An Enhanced Program Vulnerability Factor Methodology for Cross-Layer Resilience Analysis". In: *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2016, pp. 168–179. DOI: 10.1109/DSN.2016.24.

[13] Johannes Grinschgl, Armin Krieg, Christian Steger, Reinhold Weiss, Holger Bock, and Josef Haid. "Efficient fault emulation using automatic pre-injection memory access analysis". In: *SOC Conference (SOCC), 2012 IEEE International*. IEEE. 2012, pp. 277–282.

[14] Ulf Gunneflo, Johan Karlsson, and Jan Torin. "Evaluation of Error Detection Schemes Using Fault Injection by Heavy-ion Radiation". In: *Proceedings of the 19th International Symposium on Fault-Tolerant Computing (FTCS-19)*. IEEE Computer Society Press, June 1989, pp. 340–347. DOI: 10.1109/FTCS.1989.105590.

[15] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. "MiBench: A free, commercially representative embedded benchmark suite". In: *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*. 2001, pp. 3–14. DOI: 10.1109/WWC.2001.990739.

[16] Jens Güthoff and Volkmar Sieh. "Combining software-implemented and simulation-based fault injection into a single fault injection method". In: *Proceedings of the 25rd International Symposium on Fault-Tolerant Computing (FTCS-25)*. IEEE Computer Society Press, June 1995, pp. 196–206. DOI: 10.1109/FTCS.1995.466978.

[17] Siva Kumar Sastry Hari, Sarita V Adve, Helia Naeimi, and Pradeep Ramachandran. "Relyzer: Application resiliency analyzer for transient faults". In: *IEEE Micro* 33.3 (2013), pp. 58–66.

[18] Siva Kumar Sastry Hari, Sarita V Adve, Helia Naeimi, and Pradeep Ramachandran. "Relyzer: Exploiting application-level fault equivalence to analyze application resiliency to transient faults". In: *ACM SIGPLAN Notices*. Vol. 47. 4. ACM. 2012, pp. 123–134.

[19] IEC 61508-3. *IEC 61508-3: - Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 3: Software requirements*. Geneva, Switzerland: International Electrotechnical Commission, Dec. 1998.

[20] ISO 26262-6. *ISO 26262-6:2018: Road vehicles – Functional safety – Part 6: Product development at the software level*. Geneva, Switzerland: International Organization for Standardization, 2018.

[21] ISO 26262-9. *ISO 26262-9:2018: Road vehicles – Functional safety – Part 9: Automotive Safety Integrity Level (ASIL)-oriented and safety-oriented analyses*. Geneva, Switzerland: International Organization for Standardization, 2018.

[22] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert. "Statistical Fault Injection: Quantified Error and Confidence". In: *Proceedings of the Conference on Design, Automation and Test in Europe*. DATE '09. Nice, France: European Design and Automation Association, 2009, pp. 502–506. ISBN: 978-3-9810801-5-5. URL: http://dl.acm.org/citation.cfm?id=1874620.1874743.

[23] G. Li, K. Pattabiraman, S. K. S. Hari, M. Sullivan, and T. Tsai. "Modeling Soft-Error Propagation in Programs". In: *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2018, pp. 27–38. DOI: 10.1109/DSN.2018.00016.

[24] Jianli Li and Qingping Tan. "SmartInjector: Exploiting Intelligent Fault Injection for SDC Rate Analysis". In: *Proceedings of the International Conference on Defect and Fault Tolerance in VLSI and Nanotechnology Systems(DFT '13)*. IEEE Computer Society Press, Oct. 2013, pp. 236–242.

[25] Qining Lu, M. Farahani, Jiesheng Wei, A. Thomas, and K. Pattabiraman. "LLFI: An Intermediate Code-Level Fault Injection Tool for Hardware Faults". In: *Software Quality, Reliability and Security (QRS), 2015 IEEE International Conference on*. 2015, pp. 11–16. DOI: 10.1109/QRS.2015.13.

[26] Shubu Mukherjee. *Architecture Design for Soft Errors*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008. ISBN: 978-0-12-369529-1.

[27] P. Ramachandran, P. Kudva, J. Kellington, J. Schumann, and P. Sanda. "Statistical Fault Injection". In: *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*. 2008, pp. 122–127. DOI: 10.1109/DSN.2008.4630080.

[28] Behrooz Sangchoolie, Roger Johansson, and Johan Karlsson. "Light-Weight Techniques for Improving the Controllability and Efficiency of ISA-Level Fault Injection Tools". In: *Dependable Computing (PRDC), 2017 IEEE 22nd Pacific Rim International Symposium on*. IEEE. 2017, pp. 68–77.

[29] Thiago Santini, Christoph Borchert, Christian Dietrich, Horst Schirmeier, Martin Hoffmann, Olaf Spinczyk, Daniel Lohmann, Flávio Rech Wagner, and Paolo Rech. "Effectiveness of Software-Based Hardening for Radiation-Induced Soft Errors in Real-Time Operating Systems". In: *Proceedings of the 2017 Conference on Architecture of Computing Systems (ARCS '17)* (Karlsruhe, Germany). Heidelberg, Germany: Springer-Verlag, Apr. 2017. DOI: 10.1007/978-3-319-54999-6_1.

[30] Horst Schirmeier, Christoph Borchert, and Olaf Spinczyk. "Avoiding Pitfalls in Fault-Injection Based Comparison of Program Susceptibility to Soft Errors". In: *Proceedings of the 45th International Conference on Dependable Systems and Networks (DSN '15)* (Rio de Janeiro, Brazil). Washington, DC, USA: IEEE Computer Society Press, June 2015.

[31] Horst Schirmeier, Christoph Borchert, and Olaf Spinczyk. "Rapid Fault-Space Exploration by Evolutionary Pruning". In: *International Conference on Computer Safety, Reliability, and Security*. Ed. by Andrea Bondavalli and Felicita Di Giandomenico. Cham: Springer International Publishing, 2014, pp. 17–32. ISBN: 978-3-319-10506-2.

[32] Horst Schirmeier, Martin Hoffmann, Christian Dietrich, Michael Lenz, Daniel Lohmann, and Olaf Spinczyk. "FAIL*: An Open and Versatile Fault-Injection Framework for the Assessment of Software-Implemented Hardware Fault Tolerance". In: *Proceedings of the 11th European Dependable Computing Conference (EDCC '15)*. Ed. by Pierre Sens. Paris, France, Sept. 2015, pp. 245–255. DOI: 10.1109/EDCC.2015.28.

[33] Premkishore Shivakumar, Michael Kistler, Stephen W. Keckler, Doug Burger, and Lorenzo Alvisi. "Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic". In: *Proceedings of the 32nd International Conference on Dependable Systems and Networks (DSN '02)* (Bethesda, MD, USA). Washington, DC, USA: IEEE Computer Society Press, June 2002, pp. 389–398. DOI: 10.1109/DSN.2002.1028924.

[34] D Todd Smith, Barry W Johnson, Joseph A Profeta, and Daniele G Bozzolo. "A method to determine equivalent fault classes for permanent and transient faults". In: *Reliability and Maintainability Symposium, 1995. Proceedings., Annual*. IEEE. 1995, pp. 418–424.

[35] V. Sridharan and D. R. Kaeli. "Eliminating microarchitectural dependency from Architectural Vulnerability". In: *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. 2009, pp. 117–128. DOI: 10.1109/HPCA.2009.4798243.

LIST OF ACRONYMS

Acronym		Benchmark	
BB	basic block	**BC**	bit count
BBR	basic-block region	**BFE**	blowfish encode
CR	call region	**BFD**	blowfish decode
DUP	def/use-pruning	**QSORT**	quick sort
ES	equivalence set	**RDE**	rijndael encode
FI	fault injection	**RDD**	rijndael decode
FS	fault space	**SHA**	sha1
FM	fault model		
FSR	fault-space region		
IP	instruction pointer		
SEU	single-event upset		
SDC	silent-data corruption		