An Empirical Study of the Impact of Single and Multiple Bit-Flip Errors in Programs

Behrooz Sangchoolie, Karthik Pattabiraman, *Senior Member, IEEE,* and Johan Karlsson, *Member, IEEE*

Abstract—Recent studies have shown that technology and voltage scaling are expected to increase the likelihood that particle-induced soft errors manifest as multiple-bit errors. This raises concerns about the validity of using single bit-flips in fault injection experiments aiming to assess the program-level impact of soft errors. The goal of this paper is to investigate whether multiple-bit errors could cause a higher percentage of silent data corruptions (SDCs) compared to single-bit errors. Based on 2700 fault injection campaigns with 15 benchmark programs, featuring a total of 27 million experiments, our results show that single-bit errors in most cases either yield a higher percentage of SDCs compared to multiple-bit errors or yield SDC results that are very close to the ones obtained for the multiple-bit errors. Further, we find that only around 2% of the multiple-bit campaigns resulted in an SDC percentage that was more than 5 percentage points higher than that obtained for the corresponding single-bit campaigns. For most of these campaigns, the highest percentage of SDCs was obtained by flipping at most 3 bits. Based on our results, we also propose four techniques for error space pruning to avoid injection of multiple-bit errors that are either unlikely or infeasible to cause SDCs.

Index Terms—Fault injection, transient hardware faults, single/multiple bit-flip errors, error space pruning, error space clustering.

1 Introduction

Technology and voltage scaling are making integrated circuits increasingly susceptible to bit-flip errors caused by radiation-induced transient hardware faults [1], [2]. These errors, commonly known as soft errors, can degrade system reliability by causing silent data corruptions, or SDCs (i.e., non-detected output errors), which often lead to unacceptable or catastrophic system failures. A cost-effective way of reducing the risk that hardware faults cause such failures is to introduce software-implemented error handling mechanisms [3], [4]. A common approach for evaluating the effectiveness of these mechanisms is to use Software Implemented Fault Injection (SWiFI). In SWiFI, the impact of soft errors are emulated by injecting bit-flips in registers and memory locations accessible by software.

An important challenge in SWiFI techniques is the selection of the fault model, which needs to be both straightforward to implement, and representative of real hardware faults. The single bit-flip model has been a popular engineering approximation to mimic the impact of particles strikes in both combinational logic and storage elements (e.g., flip-flops and S-RAM cells). However, earlier studies have found that many soft errors that occur in microprocessors manifest as multiple-bit errors at the application level [5], [6], [7]. This observation has led researchers [5] to question the validity of the single bit-flip fault model in SWiFI-based fault injection experiments, for calculating measures such as *error coverage* [8], [9] and *error resilience* [10], [11]. We focus on error resilience as our dependability measure in this work.

- *Behrooz Sangchoolie is with RISE Research Institutes of Sweden, Box 857, 501 15 Borås, Sweden. Email: behrooz.sangchoolie@ri.se.*
- *Karthik Pattabiraman is with University of British Colombia, 2332 Main Mall, Vancouver, BC V6T1Z4, Canada. Email: karthikp@ece.ubc.ca.*
- *Johan Karlsson is with Chalmers University of Technology, SE-412 96 Göteborg, Sweden. Email: johan@chalmers.se.*

In this paper, we propose a generic fault model for emulating the effects of radiation-induced *multiple bit-flip errors* in SWiFI experiments. This fault model allows us to inject multiple bit-flips systematically to provide *evidence* of whether such errors have a higher or lower likelihood of resulting in SDCs compared to single bit-flip errors. Since our focus is on estimating the error resilience of executable programs by fault injection experiments, the main question we answer in the paper is: *Does the error resilience of a program differ significantly between our generic multiple bit-flip model and the classical single bit-flip model, and if so by how much?*

Prior work [11], [12], [13], [14] has studied the impact of double bit-flip errors on a program, i.e., injecting two errors in a single word or multiple words. However, few studies have investigated program-level effects of multiple bit-flip errors beyond double bit-flips. One challenge in studying the effects of multiple-bit errors is that the size of the error space grows exponentially with the number of bit-flips. To address this challenge, we introduce techniques for *error clustering* and *error space pruning*. Our error clustering technique divides the error space into several subsets of errors that can be explored separately in different fault injection campaigns. The error pruning technique aims to avoid injection of errors that are not relevant for the objective of our fault injection campaigns. Examples of such errors include errors that are always detected or always overwritten, or errors that have a very low likelihood of causing an SDC. *To the best of our knowledge, we are the first to study the effects of multiple bit-flip errors on programs beyond double bit-flips, using error clustering and pruning techniques.*

This paper makes the following contributions:

- Extends an LLVM (Low Level Virtual Machine)-based fault injector that injects single bit-flips at the LLVM compiler's intermediate code level [15] (§4.1), to inject multiple-bit errors (§3.3) in a single word (§4.2) as well as multiple words (§4.3).

- Performs more than *27 million* experiments (§3.5) on 15 benchmark programs (§3.4) and for 182 single/multiple bit error configurations (§3.3) using two different fault injection techniques (§3.1).
- Quantifies the maximum (upper bound) number of multiple bit-flip errors needed to cause pessimistic percentage of SDCs (§4.2 and §4.3.2).
- Derives new insights about how the results of single bit-flip experiments can be used to prune the multiple bit-flip error space by targeting only a fraction of these errors, which reveal weaknesses of the programs under test (in terms of the number of SDCs) that are not revealed by the single bit-flip model
- Identifies LLVM instructions, as well as bit positions within the registers used by these instructions, that if targeted by errors would cause a program crash, which are not impactful (in terms of the number of SDCs), and hence be removed from the error space.

2 Fault Model and Background

2.1 Fault Model

We use the *bit-flip* model to study the effects of soft errors occurring inside a processor's core, e.g., in the register file, ALU, or different pipeline registers, which eventually manifest as data corruptions in source/destination registers of an LLVM instruction. The bit-flip model has been used in other related work, e.g., [3], [16], [17] to model the effects of soft errors. Unlike these works, our model includes both *single* and *multiple bit-flip* errors.

Similar to prior work [5], [6], we do not consider faults in memory. However, we do consider memory values getting corrupted while they are being stored (i.e., storing the wrong value to memory and reading it back later), as well as corruption of the values in the memory bus during a load (and hence the load retrieves an incorrect value). Our fault model is in line with many other related work in this area [11], [18], [19], [20]. Note that ECC, which can correct single bit flips and detect double bit flips in memory, is incapable of protecting the memory against multiple bit-flips in the same word (unless Chipkill ECC is used).

2.2 Error Resilience

In this paper, we use *error resilience* [10], [11] as the dependability metric and define it as the conditional probability that the program does not produce an SDC after a transient hardware fault occurs and impacts the program state (i.e., similar to work such as [21], [22], [23] it deals with faults passing the hardware and seen by the software). Error resilience is suitable for comparing the soft error vulnerability of programs. In contrast to *error coverage*, it can be measured or estimated for any program regardless of whether the program is equipped with any fault tolerance mechanisms.

We focus on SDCs in this paper as they are the most insidious kind of failures; however, other failure types such as crashes may be just as important in some situation, e.g., when the system designer needs to consider the overall FIT (Failure in Time) rate, which includes both the SDC and crash rates.

2.3 Related Work

Traditionally, most fault injection studies at the program level have focused on the single bit-flip model, i.e., injecting single bit-flips into executing programs. However, recently, there have been some studies focusing on double bit-flip model [11], [12], [13]. Lu et al. [11] compare the results of injecting single bit-flip errors with injecting double bit-flip errors in a single word and in different words at the LLVM compiler's intermediate code level using the LLFI [24] fault injector. They find that there is not much variation between the error resilience of the different models. The main focus of the work is on the fault injection tool rather than a thorough study of the impact of multiple bit-flip errors. Ayatolahi et al. [12] compare the single bit-flip model with the double bit-flip model at the assembly-level code. In their study, double bit-flip errors are only injected into a single word (i.e., register or memory location). They also find that the SDC results obtained for the two fault models are only marginally different. Adamu-Fika and Jhumka [13] compare the results of injecting double bit-flip errors in a single word with those obtained when injecting two single bit-flips simultaneously into two different words at the LLVM compiler's intermediate code level. Similar to the other two studies, the results of their experiments show that, on average, the difference between the percentage of *data failures* for the two models is marginal. However, they do not consider the relative positions of the faults injected, nor do they generalize their findings beyond double bit-flips.

Compared to the above mentioned studies, in this paper, we go beyond the double bit-flip model by injecting up to 30 bit-flip errors in single words as well as different words in each program run. We conduct experiments with two fault injection techniques, *inject-on-read* and *inject-on-write* (see §3.1), and employ two parameters for controlling the multiple bit-flip injections (see §3.3) i) the maximum number of multiple bit-flips to be injected in one experiment (10 variants), and ii) the size of the dynamic window between consecutive injections (9 variants). In addition, our experiments are conducted on 15 programs from two benchmark suits (see §3.4). This yields a parameter space that include 2x10x9 = 180 variants of multiple bit-flip injections for each program. Since the multiple bit-flip error space is extremely large, we also derive insights on pruning the error space.

There has also been some work targeting multiple inputs of software modules to evaluate the robustness of the module to failures triggered by exceptional inputs [25], [26]. Jiantao Pan [25] introduces a model called *dimensionality* to pin-point the number of function call parameters that are responsible for a failure. The model is used in a subsequent work [26] to improve software robustness. However, compared to our fault model, the dimensionality model has two main limitations; *(i)* multiple errors are only introduced to the parameters of each interface, which may not be representative of errors that occur in variables used within the function; *(ii)* the number of errors that are introduced in each interface is limited by the number of parameters used.

There are also studies addressing intermittent faults, which could model multiple-bit errors. Intermittent faults are those that show up intermittently at the program level. Rashid et al. [27] build an intermittent fault model at the

microarchitectural level using stuck-at-last-value and stuck-at-zero/one models. However, they assume that *(i)* a microarchitectural unit may be affected by at most a single intermittent fault and *(ii)* at most one microarchitectural unit may be affected by an intermittent fault. These assumptions may not hold for transient faults, which is our focus.

Chang et al. [28] evaluate the effect of single bit-flip faults injected into a microarchitectural model of a processor at the program level. These faults can manifest as multiple bit flip errors at the program-level. Their study comes to a similar conclusion as ours, namely that the SDC probabilities do not change much regardless of whether the fault manifests as a single or multiple bit flip error. However, their technique is confined to a single architecture and hardware platform, while ours is more generic and can capture a wide variety of hardware platforms and architectures.

Chatzidimitriou et al. [2] analyze the effects of multi-bit upsets in modern microprocessors, using microarchitecture level fault injection on six hardware components of an ARM Cortex-A9 CPU modeled on a Gem5 microarchitectural simulator. Their study is limited to a maximum injection of three bit-flip errors in a limited set of spatially adjacent bits as opposed to the systematic error space exploration done in our study. Moreover, they use the architectural vulnerability factor (AVF) to illustrate the results, which does not focus on SDC as the comparison factor.

2.4 Error Clustering and Error Space Pruning

The size of the error space, (i.e. the total number of injectable errors) depends on the execution time of the program as well as the number of bit positions that are reachable for fault injection. Since even small programs consume and produce a large number of bits during execution, the error space becomes too large to explore exhaustively for almost all programs. This makes it infeasible to conduct exhaustive fault injection campaigns even for workloads with a fairly low number of instructions. Therefore, in this paper, we introduce techniques for *error clustering* and *error space pruning* in the context of multiple-bit errors.

The clustering technique we propose divides the error space systematically into several subsets that can be explored separately by fault injection. To define the clusters, we rely on two parameters: (i) the number of bit-flip errors that could occur during a program run; and (ii) the distance in time (in terms of the number of dynamic instructions) between consecutive injections (see §3.3). Using these parameters, we form 180 clusters for each program and conduct multiple bit-flip fault injection experiments for each cluster.

The clustering technique is complemented by error space pruning techniques aiming to avoid injection of errors that are not relevant for the objective of our fault injection campaigns, i.e., to compare the impact of multiple-bit and single-bit errors. A basic form of error space pruning is to exclude errors that have a known given impact on a program, e.g., errors that are always detected or always overwritten. In our case, we are interested in investigating differences in the percentage of errors that result in SDCs for multiple bit-flip errors Vs. single bit-flip errors. To this end, we propose an approach for step-wise error space pruning that aims to exclude errors that are "uninteresting", for answering a series of inter-related research questions (§3.6).

To address the issue of large error space, prior work has randomly sampled the error space or used error clustering [19], [29], [30]. However, the heuristics used are specific to the single bit-flip scenario. Fault pruning has also been explored for techniques in the CPU [18] and GPU domains [31]. These techniques leverage similarities in error propagation across multiple fault sites and/or multiple threads in the program to prune faults that result in similar executions or intermediate states. In contrast, our work leverages the similarities between single and multiple bit flips in different fault sites to prune the injection space. Yang et al. [32] extend the fault site pruning of GPU applications [31] to multi-bit faults. However, they limit themselves to GPU applications. Further, they do not consider heuristics for pruning multi-bit faults beyond those for single bit faults.

Finally, Kaliorakis et. al. [33] propose *Merlin*, that leverages the observation that faults injected in the same microarchitectural structure in an interval of time are likely to have similar effects on the program execution, and consequently groups these faults into a single entity. Unlike our work, Merlin is confined to single bit-flip faults, and works at the microarchitectural level rather than at the application level.

3 Experimental Setup

In this section, we first present the different fault injection techniques used in §3.1. Then in §3.2 and §3.3, we present the fault injection tool used in the paper and our extensions to it, respectively. In §3.4, we present the benchmark programs used in our experiments. In §3.5, we present the design and the outcome classification of the experiments. Finally, in §3.6, we present the Research Questions (RQs) we answer in this paper.

3.1 Fault Injection Techniques

In this paper, we conduct our fault injection experiments using *inject-on-read* and *inject-on-write* techniques. Using these techniques, faults are only injected in live registers, which eliminate injection of faults with no possibility of activation. Previous studies (e.g. [34], [35]), have shown that 80-90% of faults selected by random sampling are not activated. Examples are faults placed in a register just before the register is written into (and thus the error is overwritten), and faults that are injected into unused registers.

3.1.1 Inject-on-read

This technique injects bit-flip errors into registers just before they are read by an instruction [30], [36], [37]. Using this technique, Barbosa et al. [30] reduced the error space of workloads by two to five orders of magnitude compared to using random sampling in the time and space domains. The inject-on-read is well suited for emulating errors that occur when the target register is struck by an ionizing particle (as opposed to errors that propagate into the target register).

3.1.2 Inject-on-write

In inject-on-write, bit-flip errors are injected into a register when a new value is written into the register by a machine instruction [24], [36], [38]. Inject-on-write is suitable for

emulating soft errors that first occur in other parts of the CPU than the register file, such as the arithmetic logic units (ALU), cache memories, translation lookaside buffer (TLB), etc., and then propagate into a register in the register file.

3.1.3 Limitations

When using program-level error injection, it is desirable to apply a weight factor that reflects the number faults modelled by each injected error [30], [36], [39]. Unfortunately, determining such weight factors is a non-trivial task, and therefore most software-based error injection tools, including ours, lack support for deriving weight factors. To what extent the introduction of weight factors would influence the results of error injection experiments is an open question that we leave for future work.

3.2 LLFI Fault Injection Tool

In this paper, we use LLFI [24], an open source fault injector, that injects faults into the LLVM [15] framework's intermediate code of a program. LLVM is a collection of reusable compiler tools and components, and allows analysis and optimization of code written in multiple programming languages. The key component of LLVM is its intermediate representation (IR), an assembly-like language that abstracts out the hardware and ISA-specific information.

3.3 Extending LLFI for Multiple Bit-Flip Injections

In this work, we have extended LLFI[1] to facilitate the injection of multiple bit-flip errors. LLFI [24] defines single bit-flip errors as time-location pairs according to a fault-free execution of a program. The location is a bit position within an IR-register (i.e., a register defined in the intermediate representation), while the time corresponds to the execution of a dynamic IR instruction, which reads from or writes to the selected IR-register. Thus, we use a two-stage sampling of the error space, where we first select the target instruction (time) and then the bit-position to be flipped (location) within a register that is read or written by an instruction.

To model multiple bit-flip errors, we extend the time-location parameters by two additional parameters, namely *max-MBF* and *win-size*, which allow us to cluster the error space into different classes of errors to be able to explore the error space systematically. The max-MBF parameter controls the *maximum* number of bit-flip errors that could occur in one run of a program. Selecting a certain value, say 5, as the max-MBF does not necessarily mean that five errors will be injected in each fault injection experiment. This is because the program may crash prematurely (after the first injection, say), causing the remaining faults not to be injected. The win-size controls the number of dynamic instructions that should be executed between consecutive injections. For example, if the win-size is equal to 2, the dynamic instruction distance between each injection is 2.

As there are no commonly agreed values for these parameters in the literature, we consider a wide range of values when studying the impact of multiple bit-flip errors on programs. These value ranges cover various multiple bit-flip scenarios temporally, enabling us to perform sensitivity analysis. We use different values for the max-MBF (see Table 1) ranging from 2 to 30. We explain why later in §4.3.1.

1. Available at http://github.com/DependableSystemsLab/LLFI

Table 1
Values Selected for the Maximum Number of Multiple Bit-Flip Errors (max-MBF) and the Dynamic Window Size (win-size) Between Consecutive Injections.

max-MBF index	max-MBF value	win-size index	win-size value
m1	2	**w1**	0
m2	3	**w2**	1
m3	4	**w3**	4
m4	5	**w4**	random between 2-10
m5	6	**w5**	10
m6	7	**w6**	random between 11-100
m7	8	**w7**	100
m8	9	**w8**	random between 101-1000
m9	10	**w9**	1000
m10	30		

For the window size parameter, we select nine win-size values covering dynamic window sizes from zero to 1000 (see Table 1). A window size of zero implies that the injections, following the first one, will be performed into same register as the first one. The rationale behind limiting the maximum value of this parameter to 1000 is that we predominantly consider multiple bit-flip errors in software that are caused by a single transient fault in the processor. Such faults are likely to affect instructions that are "in-flight" in the processor's instruction window (i.e., the set of all instructions that have been decoded but not yet committed in a superscalar processor). Typical instruction windows in modern processors are a few hundred of instructions in size, and hence 1000 is a reasonable upper bound. Six of the values selected are constants (0, 1, 4, 10, 100, 1000). The remaining three values are randomly selected from a range of 2-10, 11-100, or 101-1000, for better representativeness.

The chosen win-size values could also represent multiple unconnected transient faults that cause errors in instructions that are apart from each other by less than 1000 dynamic instructions. However, it is unlikely that multiple transient faults occur in a single run of a program, especially within such short dynamic window sizes as we consider in Table 1.

3.4 Benchmark Programs

We conduct experiments with 15 programs from MiBench [40] and Parboil [41] benchmark suits, see Table 2. The programs are selected to enforce diversity in source code implementation and size, input type/size and functionality.

3.4.1 MiBench Benchmark Suite

This benchmark suite contains programs divided into six packages representing different application domain of embedded systems: automotive, consumer, network, office, security, and telecom. We selected 11 programs from these packages shown in Table 2. MiBench provides two inputs for each program, *small* and *large*. We use the small inputs in our experiments as the total number of candidate instructions for fault injection is already significantly large (see Table 2) allowing us to answer the research questions defined in §3.6.

3.4.2 Parboil Benchmark Suite

This benchmark suite contains a set of programs representing scientific and commercial applications. We selected two programs, bfs and histo, from its *base* implementation package, and two programs, sad and spmv, from its *CPU* implementation package, see Table 2. The table also shows the total number of dynamic instructions that are candidates for inject-on-read and inject-on-write, respectively. The number of instructions available for inject-on-read is higher than for inject-on-write. This is because some instructions, e.g., the `store` instruction, do not have a destination register in the LLVM IR, and thus inject-on-write cannot be performed.

3.5 Experimental Design and Outcome Classification

We conducted 182 *fault injection campaigns* for each benchmark program. A campaign refers to a set of fault injection experiments using the same fault model on a given *workload*, which is a program running with a given input. Half of the campaigns used inject-on-read and the other half inject-on-write. In addition to two single bit-flip campaigns (one inject-on-read and one inject-on-write), we performed 180 multiple bit-flip campaigns for each program by varying the parameters max-MBF and win-size shown in Table 1.

Each campaign consists of 10,000 fault injection experiments to obtain tight error bounds. Thus, we perform a total of $10,000 * 182 * 15 = 27,300,000$ experiments. The error bars shown in the results represent 95% confidence intervals. The outcome of each experiment is classified into:

- *Benign.* The program terminates normally and produces a correct output. Outcomes in this category reflect the inherent robustness of the target program.
- *Detected by Hardware Exceptions.* The injected error raises a hardware exception. Almost all these exceptions cause the program to *crash*. Outcomes in this category represent errors that can potentially be corrected by software-level or system-level fault tolerance mechanisms. These exceptions include *segmentation faults* (accessing memory words outside the legal memory boundary), *aborts* (programs aborted by themselves or the OS), *misaligned memory accesses* (memory accesses are not aligned at four bytes), and *arithmetic errors* such as division by zero.
- *Hang.* The program fails to terminate within a predefined time, which is set by LLFI to be one or two orders of magnitude greater than the execution time of the fault-free run. Errors producing this type of outcomes can be detected by watchdog timers.
- *NoOutput.* The program terminates without generating an output. Errors producing this type of outcomes can be potentially corrected by re-executing the program.
- *Silent Data Corruption (SDC).* The program terminates normally and there is no indication of failure, but the output is incorrect based on a bit-wise comparison.

The first four outcome categories contribute to a program's error resilience. Out of these, the Benign category represents the inherent robustness of the program, while Detected by hardware exceptions, Hang, and NoOutput categories represent errors that are detected, thus could potentially be corrected. In the remainder of this paper, we merge the results obtained for the latter three categories, and denote them as *Detection*.

3.6 Research Questions (RQs)

The RQs are motivated by four error pruning techniques we investigate. The first error space pruning technique deals with the selection of an upper bound for the max-MBF parameter, since there is no commonly agreed mapping model that could be used to reason about the number of software-level errors due to a hardware transient fault. So the first RQ deals with the errors that are activated when multiple errors are injected, and do not result in crashes.

- ***RQ1.*** When multiple errors are injected, how many errors are activated before the program crashes (if it crashes)?

In the second error space pruning technique, we classify fault injection results with respect to parameters such as the fault injection technique used (inject-on-read and inject-on-write), the maximum number of bit-flips injected (max-MBF), and the dynamic window size between consecutive injections (win-size), to investigate whether we could further prune the error space by finding parameter values that result in pessimistic percentage of SDCs i.e., conservative upper-bounds. Thus, we answer the following questions:

- ***RQ2.*** Does the single bit-flip error model result in pessimistic percentage of SDCs when compared with the multiple bit-flip error model?
- ***RQ3.*** Is there an upper bound to the maximum number of multiple bit-flips for pessimistic percentage of SDCs?
- ***RQ4.*** Is there a maximum dynamic window size that causes pessimistic percentage of SDCs?

In the third error space pruning technique, we answer the following question:

- ***RQ5.*** Is it possible to find fault injection locations that are insensitive to multiple bit-flip errors compared to single bit-flip errors, and exclude them from the multiple-injection error space?

We add a fourth error space pruning technique, for identifying instructions as well as bit positions that could be the target of the pruning. We then answer the following RQs.

- ***RQ6.*** Are there specific instructions that could be candidates of further pruning?
- ***RQ7.*** Are there specific bit positions within the registers used by the instructions that could be pruned?

4 Experimental Results

In this section, we present detailed classifications of fault injection results with respect to the parameters, max-MBF and win-size as well as the type of fault injection technique used. These classifications help us quantify the differences between candidate values that can be chosen for each parameter, which allows us to answer the RQs in §3.6.

4.1 Single Bit-Flip Model

Fig. 1 shows the outcome classification results with the single bit-flip model. Fig. 1a and Fig. 1b show the results for when inject-on-read and inject-on-write fault injection techniques are used, respectively. Recall that the Detection category is the sum of the *Hang*, *NoOutput* and *Detected by Hardware Exception* categories. The percentage of experiments classified as *Hang* and *NoOutput* is insignificant

Table 2
Selected Benchmark Programs

Benchmark	Package	Program (LoC)	Total number of candidate instructions for fault injection		Description & Input
			inject-on-read	inject-on-write	
MiBench	automotive	basicmath (178)	3,683,881	2,964,600	Performs mathematical calculations such as cubic equation calculation and square root calculation on a set of constants.
		qsort (35)	2,615,557	2,214,245	Implements the Quick Sort algorithm on a list of words.
		susan_corners(1700)	2,449,209	2,088,322	Finds corners of a black & white image of a rectangle.
		susan_edges(1700)	5,188,476	4,413,577	Finds edges of a black & white image of a rectangle.
		susan_smoothing(1700)	62,752,639	49,105,460	Smooths a black & white image of a rectangle.
	telecomm	FFT (215)	5,313,377	4,526,716	Performs Fast Fourier Transformation on an array of data.
		IFFT (215)	5,423,988	4,620,938	Performs reverse FFT on an array of data.
		CRC32 (107)	28,746,216	23,270,737	Implements the 32-bit Cyclic Redundancy Check on a sound file.
	network	dijkstra (133)	67,617,629	54,495,536	Uses Dijkstra's algorithm to find the shortest path between pairs of nodes constructed from an adjacency matrix representation graph.
	security	sha (188)	30,609,559	25,726,389	Implements the well known SHA (secure hash algorithm), generating a 160-bit digest from an ASCII text file.
	office	stringsearch (340)	161,533	114,835	Searches for words in phrases using case insensitive comparison.
Parboil	base	bfs (592)	113,582,521	94,021,100	Uses the breadth-first search algorithm to compute the shortest-path cost from a single node to every reachable node in an irregular graph of uniform edge weights derived from the map of New York.
		histo (610)	678,224,521	566,829,877	Computes a 2-D saturating histogram with a maximum bin count of 255 of the default input set.
	cpu	sad (944)	648,604,565	510,295,230	Calculates the sum of absolute differences in the default input set.
		spmv (619)	11,003,882	8,965,172	Computes the product of a sparse matrix with a dense vector. We select the small input, which is a sparse matrix in coordinate format.

(less than 0.3%), and hence most of the experiments in the Detection category were detected by hardware exceptions.

Fig. 1 shows that there is significant variations between the SDC results obtained for different programs, ranging from less than 5% for the susan_corner, susan_edge, dijkstra, stringsearch, histo and spmv programs to up to around 75% for sha. Fig. 1 also shows that overall, the SDC percentage when using the inject-on-write technique is higher than that when using the inject-on-read technique. A similar trend was also observed by Sangchoolie et al. [36]. The reasons for this difference are *(i)* the type of data-items stored in source/destination registers as well as *(ii)* the number of times that these registers are accessed throughout the execution of the program. Registers could hold data-items of different types such as memory addresses, data variables, and control information. Errors injected in memory addresses are mostly detected by hardware exception mechanisms, causing a higher percentage of crashes and hence lower percentage of SDCs [10], [36]. Both source registers and destination registers could hold a memory address, however, an address may be read multiple times after it is written into. This increases the probability of an error being injected into an address when using the inject-on-read technique, which would eventually result in a lower percentage of SDCs for the results obtained using the inject-on-read technique compared to the inject-on-write technique.

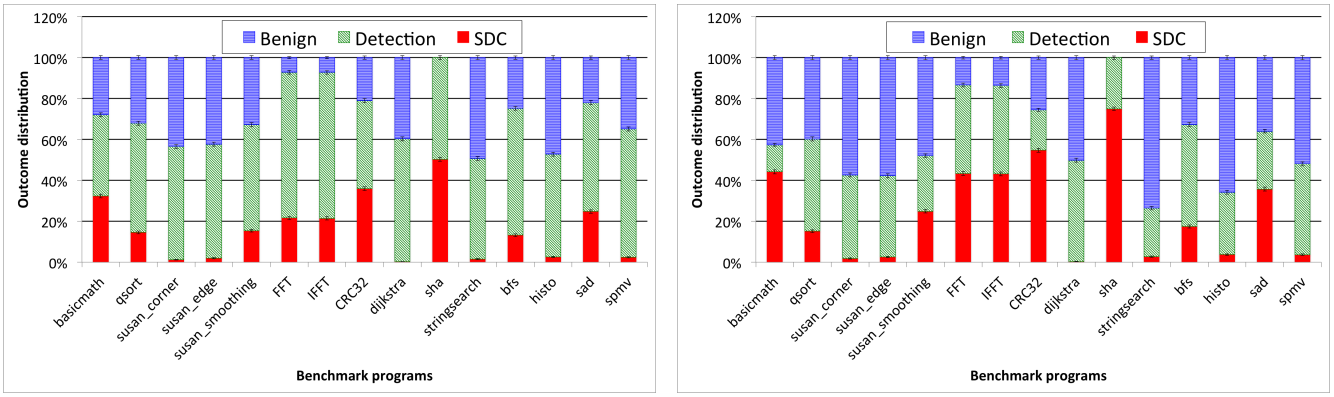
4.2 Multiple Bit Flips in the Same Register

Fig. 2 shows the classification of fault injection results when the multiple injections are performed into the same instruction (i.e., register). In other words, the dynamic window size (win-size) value is zero, and only the max-MBF parameter is varied from 1 (the leftmost bar) to 30 (the rightmost bar). The goal is to understand how much the max-MBF parameter alone contributes to the percentage of SDCs.

Fig. 2a and Fig. 2b show the results for when inject-on-read and inject-on-write fault injection techniques are used, respectively. The leftmost result bar for each benchmark program represents the percentage of SDCs when only a single-bit error is injected, while the other result bars correspond to the percentages of SDCs caused by different numbers of multiple bit-flip errors ranging from 2 to 30.

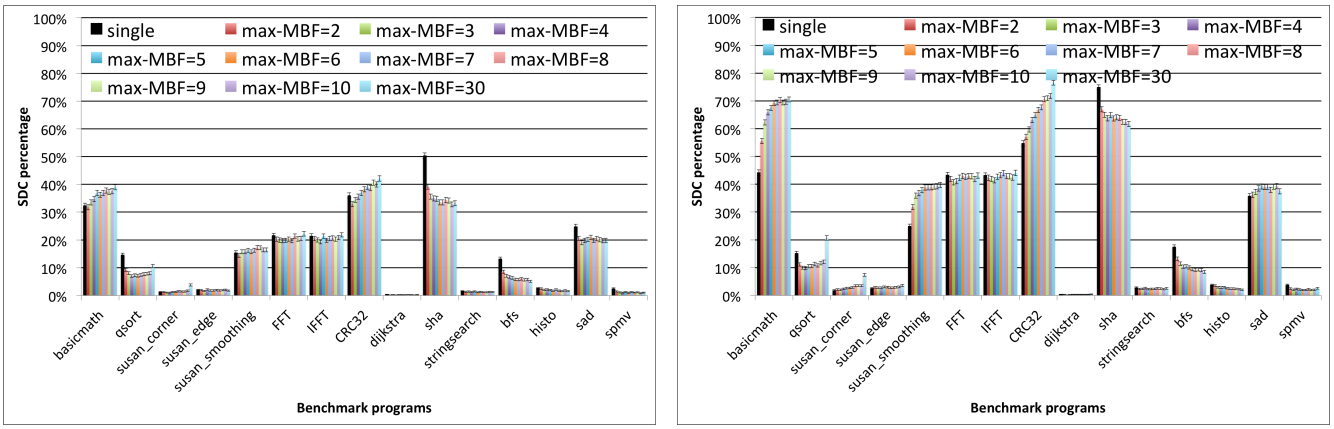
Fig. 2 shows that, similar to the results obtained in previous work [12], [42], for the majority of the programs, the SDC results obtained for the single bit-flip model is either pessimistic, or very close to the ones obtained for the multiple bit-flip model. This is because an increase in the number of injected bit-flips also results in an increase in the likelihood that detection mechanisms are raised, reducing the percentage of SDCs. This is explained further in §4.3.3.

However, for basicmath and CRC32 programs, the SDC percentages for the single bit-flip model are significantly lower (especially when using the inject-on-write) than for



(a) inject-on-read (b) inject-on-write

Figure 1. Fault injection outcome classification for campaigns using single bit-flip model. The Detection category refers to the sum of Detected by Hardware Exception, Hang and NoOutput categories. The error bars indicate 95% confidence intervals.



(a) inject-on-read (b) inject-on-write

Figure 2. Percentage of SDCs for injecting different number of errors into the same instruction/register (i.e., win-size = 0). The leftmost and rightmost bars, for each program, represents the percentage of SDCs when injecting 1 and 30 errors, respectively. The error bars indicate 95% confidence intervals.

those obtained for the multiple bit-flip model, and hence the single bit-flip model does not yield pessimistic SDC results for these programs. The reason for this is that single bit-flip errors injected into these programs results in a low percentage of Detections compared with the other programs (Fig. 1). This means that a high number of errors remained undetected and hence were classified as either SDC or Benign for these programs. Therefore, when targeting these programs with multiple bit-flip errors, it is more likely for the result of the experiment to be classified as SDC (as opposed to Detection), thereby causing a higher percentage of SDCs for the multiple bit-flip model.

For qsort and susan-corner programs, the single bit-flip model results in a higher and thereby a more pessimistic percentage of SDCs compared to the multiple bit-flip model, except for when max-MBF = 30. However, it is unlikely that these number of bits are affected by a single fault; this configuration (max-MBF = 30) is mainly selected for answering RQ1. Therefore, for qsort and susan-corner programs also, the single bit-flip model provides us with a pessimistic estimate of the percentage of SDCs caused due to multiple bit-flip injections.

***RQ2-Answer*:** For the majority of the benchmark programs, the results obtained for the single bit-flip model is either pessimistic or very close to the ones obtained for the multiple bit-flip model for bit-flips in the same register (i.e., $win - size = 0$).

4.3 Results When Targeting Bits of Multiple Registers

In this section, we consider multiple bit-flips in multiple registers accessed by different instructions. To control the distance between consecutive injections, we choose the dynamic window sizes (win-size) that are greater than zero from Table 1. We first attempt to bound max-MBF by studying the number of errors that are activated when max-MBF=30 (§4.3.1). We find that only a small fraction of these bit-flips are activated before the program crashes, making it unnecessary to select higher values for max-MBF. However, as the error space is still large, we search for max-MBF/win-size pairs in the space that cause pessimistic percentage of SDCs (§4.3.2). Finally, we investigate whether the single bit-flip fault injection results can help prune the multiple bit-flip error space (§4.3.3).

4.3.1 Number of Activated Errors

Fig. 3 shows the distribution of the number of activated bit-flips before causing a program to crash, up to 30 bit-flip errors. The reason for selecting such a high value (max-MBF=30) is to find the portion of errors that would remain undetected and can hence be pruned. The results presented here include all win-size values shown in Table 1.

Fig. 3 shows that 5 activated errors are enough to cause a program to crash in more than 96% (78%) of the experiments using inject-on-read (inject-on-write) techniques. Furthermore, 3% and 14% of the inject-on-read and inject-on-write experiments, respectively, managed to activate six to ten errors. And finally, only around 1% of the inject-on-read experiments and 8% of the inject-on-write experiments had more than 10 activated errors. Thus, we see that an upper bound of 10 errors for max-MBF is sufficient to capture the majority of fault injection outcomes, and hence can be used to bound the value of max-MBF. As expected, the errors that remain undetected would most likely result in SDCs.

***RQ1-Answer*:** Around 99% of inject-on-read and 92% of inject-on-write experiments had at most 10 activated errors.

4.3.2 Max-MBF/win-size Pairs that Cause Pessimistic Percentage of SDCs

Fig. 4 and Fig. 5 show the SDC results for the experiments targeting bits of multiple registers using the inject-on-read, and inject-on-write techniques, respectively. Both figures show that when increasing the number of errors, the general trend for the SDC results is declining, regardless of the value of win-size. We further study each technique in detail below.

4.3.2.1 Inject-on-read Technique: Fig. 4 shows the SDC results for the experiments targeting bits of multiple registers using the inject-on-read technique. The 95% confidence intervals for these results are ± 0.19 for dijkstra and ± 0.97 for sha. In 13 programs, the percentage of SDCs caused due to the single bit-flip model is higher than or almost the same as (i.e., difference less than one percentage point) the ones caused due to the multiple bit-flip model. However, for CRC32 and stringsearch, there are multiple bit-flip campaigns that result in a higher percentage of SDCs. Even for these 2 programs, the percentage of SDCs in the single bit-flip model is only 2 percentage points lower than the multiple bit-flip model that causes the highest percentage of SDCs. *Thus, the single bit-flip model provides a pessimistic upper-bound on SDCs for most of the programs*.

It is interesting to note that even when the single bit-flip model does not result in pessimistic SDC results, *two errors are enough to result in the highest (most pessimistic) percentage of SDCs*, regardless of the value of win-size selected (see Table 3). The value of this observation is that in the case of the inject-on-read technique, there is no need to perform experiments with more than two bit-flip injections to obtain pessimistic estimate of the error resilience of a system.

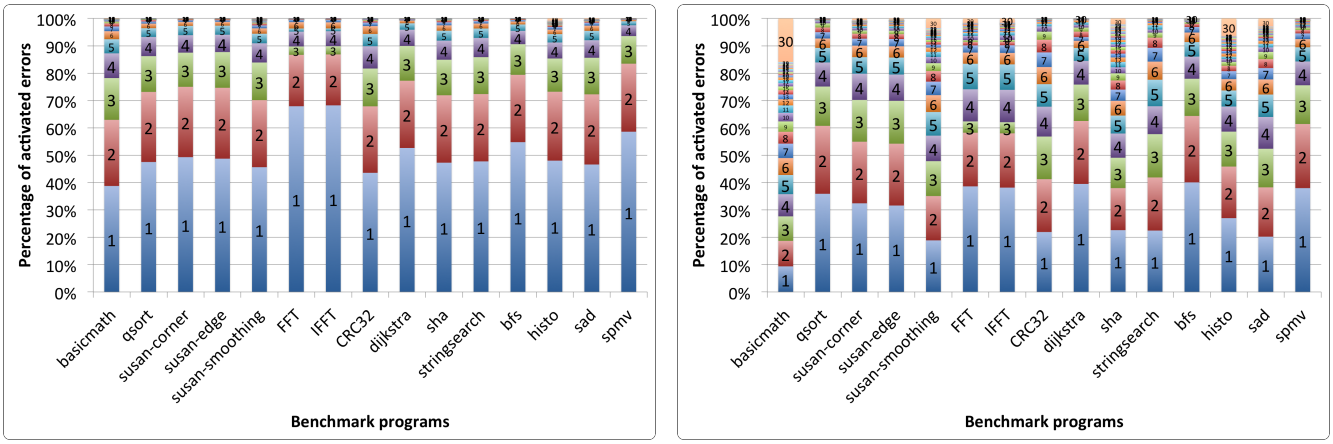
Fig. 4 also shows that except for a couple of programs such as CRC32 and susan-smoothing, there is not much variation between the percentage of SDCs obtained for different win-size configurations. In other words, when studying the impact of multiple bit-flip errors on programs, the win-size parameter does not matter much for the SDC percentages. However, Table 3 shows the win-size configurations that caused the highest percentage of SDCs, among all multiple-bit error campaigns. We can see that when using the inject-on-read technique, *higher window sizes are more likely to result in the highest percentage of SDCs*. This is because a high percentage of data-items targeted by errors when using the inject-on-read technique are memory addresses, which tend to be read more times than other types of data-items. Injecting errors into memory addresses are mostly detected by the exception mechanisms (see Fig. 1a). Thus, multiple injections into registers that are within a small window are more likely to result in an address corruption that raises an exception, thereby resulting in a higher percentage of Detections than when consecutive errors are injected into registers that are within a larger window.

Table 3
Configurations that Resulted in the Highest Percentages of SDCs, Among all Multiple Bit-Flip Error Campaigns.

Program	inject-on-read		inject-on-write	
	max-MBF	win-size	max-MBF	win-size
basicmath	2	100	3	1
qsort	2	100	3	1
susan_corner	2	1000	4	1
susan_edge	2	1000	3	1
susan_smoothing	2	1000	3	1
FFT	2	1	2	1
IFFT	2	1	2	1
CRC32	2	100	2	100
dijkstra	2	4	3	4
sha	2	10	2	1
stringsearch	2	RND(2-10)	2	4
bfs	2	1000	2	1000
histo	2	RND(2-10)	6	1
sad	2	1000	2	4
spmv	2	1000	2	RND(11-100)

Result summary (inject-on-read technique):
***RQ2-Answer*:** The single bit-flip model provides a pessimistic upper-bound on SDCs for most of the programs.
***RQ3-Answer*:** Two errors are enough to cause the highest percentage of SDCs.
***RQ4-Answer*:** Window size does not have much effect on the percentage of SDCs.

4.3.2.2 Inject-on-write Technique: Fig. 5 shows the SDC results for the experiments targeting bits of multiple registers using the inject-on-write technique. The 95% confidence intervals for the results presented here are between ± 0.26 for dijkstra and ± 0.97 for sha. From the figure, we can see that *the single bit-flip model results in a pessimistic estimate of the percentage of SDCs for only around half of the programs*. In the other half, the percentage of SDCs are 2 (dijkstra) to 17 (basicmath) percentage points lower for single bit-flip errors than for the multiple bit-flip configurations that cause the highest percentages of SDCs. The high percentage of



(a) inject-on-read (b) inject-on-write

Figure 3. Distribution of the number of activated errors before causing a program to crash, given that max-MBF is equal to 30.

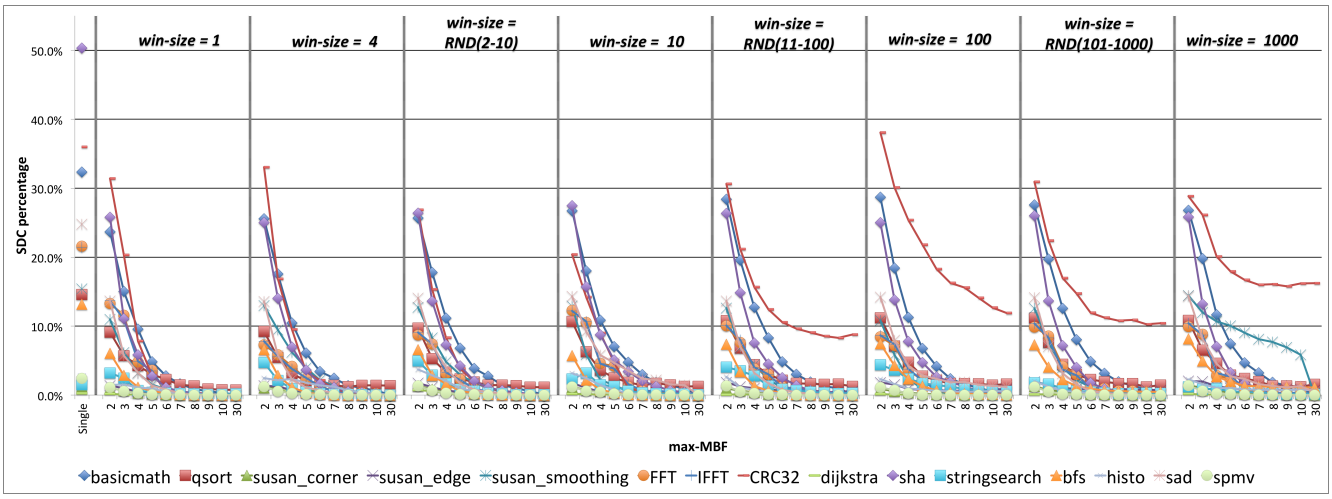


Figure 4. SDC results for experiments targeting bits (from 1 to 30) of multiple registers using the inject-on-read technique. Here the RND (α, β) refers to a randomly selected value between α and β.

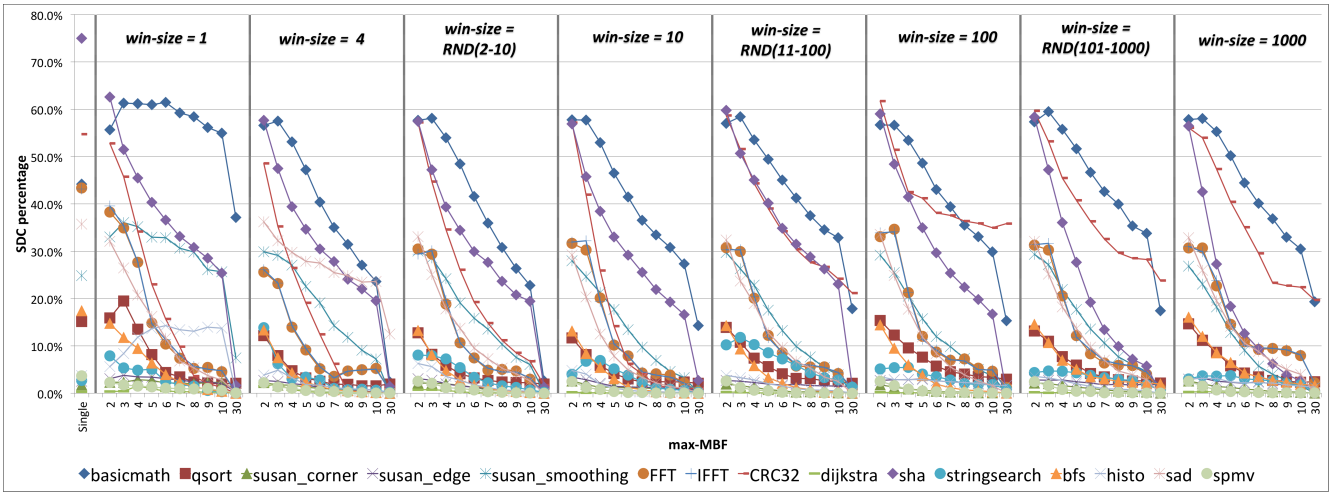


Figure 5. SDC results for experiments targeting bits (from 1 to 30) of multiple registers using the inject-on-write technique. Here the RND (α, β) refers to a randomly selected value between α and β.

difference for basicmath can again be explained using Fig. 1, where injecting single bit-flip errors in basicmath result in the lowest percentage of Detections. This in turn leads to higher percentage of SDCs.

The results of our multiple bit-flip campaigns show that *three errors are sufficient to cause the highest percentage of SDCs for 114 out of 120 program/win-size pairs* (corresponding to 95% of the pairs). Out of these 114 pairs, 93 and 21 of them correspond to when the max-MBF is equal to two and three, respectively. Out of the 120 program/win-size pairs, there are also five cases where four errors are needed to cause the highest percentage of SDCs; however, compared to when three errors are injected, these cases only result in at most one percentage point of higher percentage of SDCs, which is not a significant difference. The only exception is the histo program using the window size of one, where six errors are needed to cause the highest percentage of SDCs.

Comparing Fig. 4 and Fig. 5 suggests that depending on the fault injection technique used, different numbers of errors need to be injected into the system to produce a pessimistic estimate of the SDC percentages. *However, aggregating the results from both techniques suggest that injecting 3 errors is sufficient to result in the highest percentage of SDCs.*

Fig. 5 also shows that for many of the programs, the win-size parameter has a significant effect on the percentage of SDCs when using the inject-on-write technique (unlike the inject-on-read technique). Further, according to Table 3, *when using the inject-on-write technique, lower window sizes are more likely to result in the highest percentage of SDCs*, which is different from what we observed for the inject-on-read technique. This is because the inject-on-write technique targets a higher percentage of data variables, and thereby a lower percentage of address variables, compared to the inject-on-read technique. Errors injected into data variables mostly result in outcomes in the Benign or SDC categories. Thus, injecting multiple bit-flips within a small window size is more likely to cause an SDC, as there is less opportunity for an error to be masked before the next injection; thus, we can choose smaller window sizes for pruning.

Result summary (inject-on-write technique):
***RQ2-Answer*:** The single bit-flip model does not result in pessimistic percentage of SDCs for half of the programs.
***RQ3-Answer*:** Three errors are enough to cause the highest percentage of SDCs in 95% of the program/win-size pairs.
***RQ4-Answer*:** Lower window size values are more likely to result in the highest percentage of SDCs.

4.3.3 Sensitivity of Fault Injection Locations to Multiple Bit-Flip Errors

In this section, we investigate whether the results obtained for a single bit-flip campaign can be used to prune the multiple bit-flip error space. Our goal is to avoid injecting any multiple bit-flips errors where the first bit-flip results in an SDC (if it is injected as a single bit-flip error). The rationale for this type of pruning is that we want to focus the multiple bit-flip campaigns on errors that transform the outcome of the first bit-flip from Benign or Detected to an SDC. This process is explained in Fig. 6.

In Fig. 6, t_s, t_b and t_d correspond to when a single bit-flip error is injected into a specific location in the program, resulting in SDC, Benign or Detection, respectively. All other transitions correspond to when multiple bit-flip errors are injected starting from the same program location, and change the result of the single bit-flip outcome. For example, t_{b-s} refers to a change in the fault injection result from Benign to SDC due to the injection of a multiple bit-flip error. Fig. 6 shows two transitions where injecting additional bit-flip errors into the program changes its result from Benign or Detection to SDC, thereby decreasing its resilience:

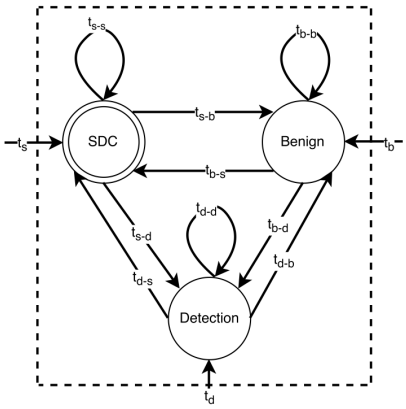


Figure 6. State diagram showing transitions between different outcome categories due to the injection of multiple-bit errors.

- **Transition I (t_{d-s}).** Injecting a single bit-flip error into a location results in the Detection category, but injecting multiple bit-flip errors changes the result to an SDC.
- **Transition II (t_{b-s}).** Injecting single bit-flip error into a location results in the Benign category, but injecting multiple bit-flip errors changes the results to an SDC.

To find the likelihood of the above transitions, we conduct two fault injection campaigns for each program, one for each fault injection technique. To get the worst-case estimates, we use the max-MBF/win-size pairs that caused the highest percentage of SDCs when conducting multiple bit-flip fault injection campaigns (see Table 3). We choose the location of the first error of each multiple bit-flip experiment from those chosen for the single bit-flip model. We do not consider the t_{s-s} transition as *we only consider cases that would add to the number of SDCs (i.e., pessimistic percentage)*.

Table 4 shows the results. From the table, we can see that Transition I is very unlikely (in most cases below 1%), especially with the inject-on-read technique. Therefore, we can prune the multiple bit-flip error space by excluding those locations that would result in the Detection category or an SDC under the single bit-flip model. In fact according to the results presented in Fig. 1, these locations include around 50-100% of the inject-on-read and 27-100% of the inject-on-write single bit-flip experiments, which is a significant reduction in the error space. However, there is much more variation when it comes to the likelihood of Transition II, and its value ranges from 0% to 81%, and hence these locations cannot be ignored.

***RQ5-Answer*:** We can prune the multiple bit-flip error space by injecting the first error of each experiment only into locations that if targeted by a single bit-flip error would result in Benign outcomes, as these are the locations that are likely to add to the number of SDCs under multiple bit-flips.

Table 4
Likelihood of Transition I and Transition II.

Program	inject-on-read		inject-on-write	
	Tran. I	Tran. II	Tran. I	Tran. II
basicmath	1.1%	31.9%	0.6%	58.3%
qsort	0.7%	13.4%	0.4%	29.5%
susan_corner	0.1%	1.2%	0.1%	4.1%
susan_edge	1.2%	0.6%	0.9%	0.8%
susan_smoothing	0.3%	14.6%	0.8%	34.6%
FFT	0.4%	25.6%	4.1%	23.4%
IFFT	0.5%	23.6%	3.6%	26.0%
CRC32	0.8%	48.1%	0.6%	81.8%
dijkstra	0.0%	3.1%	0.2%	2.9%
sha	1.0%	0.0%	2.2%	0.0%
stringsearch	0.1%	7.7%	0.2%	15.7%
bfs	0.2%	10.7%	0.8%	19.2%
histo	0.1%	5.2%	0.2%	19.6%
sad	12.9%	2.9%	14.9%	2.1%
spmv	0.1%	1.1%	0.1%	1.5%

4.3.4 Identification of Instructions and Bit Positions for Pruning

In this section, we describe how the multiple bit-flip error space can be pruned by analysis of the results of the multiple bit-flip campaigns. Pruning an error refers to the cases where the outcome of an error is classified as Detection without conducting an experiment. Our goal is to identify instructions that if targeted by an error would cause an *early crash*, which is when the program under test crashes before we are able to inject the number of injections specified in the experiment. The early crash cases correspond to around 20 million experiments, which is around 75% of the total number of experiments conducted. Therefore, pruning of the multiple bit-flip error space with regards to the early crash cases could significantly reduce the time and effort needed to test systems using fault injection campaigns.

Connecting the injection of an error in a specific instruction to the reason behind an early crash is challenging as there are several factors contributing to the early crash. We explain the challenge using Fig. 7. This figure shows a code Snippet from the FFT program where three errors are planned to be injected in the source registers of `shl`, `load` and `store` instructions. The program crashed before LLFI was able to inject the third error into the `store` instruction. However, it is not clear if the early crash is caused by the first error alone, the second error alone, or jointly by both.

An earlier study [27], has found that the dominant effect of an injected error is to cause a program crash soon after its injection, which indicates that large crash distances are infrequent. Therefore, in our analysis of early crash, we assume that the cause of the early crash is the injection of the error in the last instruction that precedes the crash. In other words, in the case of the example given in Fig. 7, it means that according to our assumption the injection of the error in the `load` instruction is the reason behind the early crash. This assumption allows us to study the results of the experiments, and identify the instructions that were responsible for these early crashes. These instructions then can be pruned.

```
; <label>:7
e1  %8 = load i32* %rev, align 4
    %9 = shl i32 %8, 1
    %10 = load i32* %1, align 4
    %11 = and i32 %10, 1
    %12 = or i32 %9, %11
e2  store i32 %12, i32* %rev, align 4
    %13 = load i32* %1, align 4
e3  %14 = lshr i32 %13, 1
    store i32 %14, i32* %1, align 4
    br label %15
```

Figure 7. A code Snippet of the FFT program where errors are planned to be injected in the marked instructions.

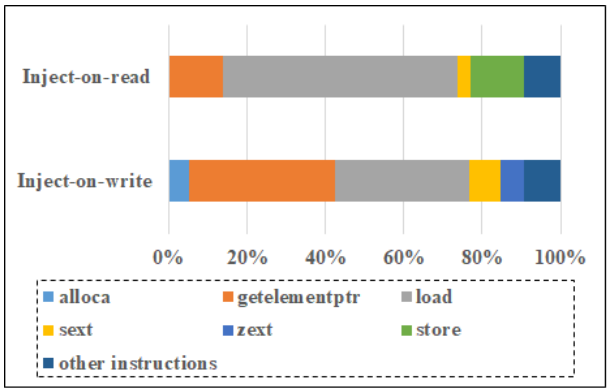


Figure 8. Instructions that contributed the most to the total number of early crashes.

Fig. 8 shows six LLVM IR instructions that caused the highest percentages of early crashes. In general, there are 39 and 37 types of instructions that, if subjected to a bit-flip, resulted in an early crash for the inject-on-read and inject-on-write campaigns, respectively. Fig. 8 illustrates that only a few of these instructions have significantly contributed to the total percentage of early crashes. In the case of the inject-on-read campaigns, more than 90% of the total number of early crashes are caused as a result of injections in four instructions, namely the `getelementptr`, `load`, `sext`, and `store` instructions. Similarly, in the case of the inject-on-write campaigns, more than 90% of the total number of early crashes are caused as a result of injections in five instructions, namely the `alloca`, `getelementptr`, `load`, `sext`, and `zext` instructions. *By leveraging these early crashes (90%), and combining it with the aggregate percentage of crashes (75%), one can prune about 67% of the injections overall for multiple bit-flip injections.* However, because the prediction is not perfect, there is a small amount of inaccuracy introduced.

***RQ6-Answer*:** Around 75% of the experiments resulted in early crashes; hence can be removed from the error space. For the inject-on-read campaigns, more than 90% of the early crashes are caused as a result of injections in four out of 39 instructions, namely `getelementptr`, `load`, `sext`, and `store`. As for the inject-on-write campaigns, more than 90% of the early crashes are caused as a result of injections in five out of 37 instructions, namely `alloca`, `getelementptr`, `load`, `sext`, and `zext`.

Table 5
LLVM IR instructions contributed the most to the number of early crashes. Note: these instructions may not have direct assembly code analogues.

Instruction	Overview of the instructions	Number of source registers targeted	Number of destination registers targeted
alloca	Allocates memory on the stack frame of the currently executing function, to be automatically released when this function returns to its caller. Sample instruction: `%2 = alloca i32, align 4`	1	1
getelementptr	Gets the address of a subelement of an aggregate data structure and performs address calculation. It can also generate a vector of such addresses and does not access memory. Sample instruction: `%2 = getelementptr inbounds i8** %1, i64 1`	1-3	1
load	Reads from memory. Sample instruction: `%2 = load i32* %1, align 4`	1	1
sext	Takes a value to cast, and a type to cast it to and sign extends the value to the type. Sample instruction: `%2 = sext i1 %1 to i32`	1	1
zext	Takes a value to cast, and a type to cast it to and zero extends the value to the type. Sample instruction: `%2 = zext i1 %1 to i32`	1	1
store	Writes to memory. Sample instruction: `store i32 0, i32* %i, align 4`	2	0

In the remainder of this subsection, we present data on how individual bits of these instructions contribute to the early crashes. To better understand the results presented, Table 5 shows high-level details about the structure of the six LLVM IR instructions under investigation. It also shows the number of source and destination registers targeted by the inject-on-read and inject-on-write campaigns respectively.

4.3.4.1 Identification of sensitive bits that lead to early crash in inject-on-read campaigns: We investigate four instructions as they contributed the most to the total number of early crashes: `load`, `getelementptr`, `store`, and `sext` contributed to around 60%, 14%, 13%, and 3% of the total number of early crashes, respectively.

Fig. 9 shows the likelihood of an early crash for bits targeted in `load` instructions. The figure shows the difference between the sensitivity of the lowest 17 bits (on average around 26%) of the register targeted compared with the remaining 47 bits (on average around 93%). This indicates that the error space can be significantly pruned by removing majority of the bits in `load` instructions from the error space while classifying them as *Detected* in the results.

The high sensitivity of the most significant 47 bits in resulting in early crashes is because the register holds a memory address. Injecting into these bits adds to the likelihood of a program crash or the injected error to be detected by mechanisms such as invalid memory access and illegal instruction. This is due to the format in which bits are stored in the register, where injections in higher significant bits result in memory addresses that are likely to be significantly large. Same conclusions are drawn in prior work [12].

Fig. 10–12 shows the likelihood of an early crash for different bits of the registers targeted in `getelementptr` instructions. Note that, in the inject-on-read campaigns, there are between one to three potential registers (depending on the program under evaluation) in a `getelementptr` instruction that could be targeted to faults (see Table 5). The graphs presented in Fig. 10–12 illustrate the results of injections in these registers separately. These figures show that except very few cases, higher significant bits are significantly more likely to result in early crashes. In other words, the sensitivity of the lowest 17 bits and the highest 47 bits to causing an early crash is on average around 24% and 72%.

In general, the `getelementptr` instruction is used to get the address of a subelement of an aggregate data structure. Errors in the higher significant bits of an address, say bit 16 and higher, has in general a high likelihood of being detected by a hardware exception and hence causing a crash. However, compared to the results presented for the `load` instruction (see Fig. 9), injections in source registers of the `getelementptr` are less likely to cause an early crash. This is because the `getelementptr` instruction only performs address calculation and does not access memory. Assuming that most errors are activated when the memory is accessed using the erroneous address calculated by the `getelementptr` instruction, it is less likely for injections in the `getelementptr` instruction to cause an early crash compared to injections in the `load` instruction. However, regardless of this comparison, Fig. 10–12 show that a significant number of bits in registers used by the `getelementptr` instruction could serve as indicators for pruning the multiple bit-flip error space. That is, any multiple bit-flip errors that target these bits are assumed to be detected, and hence there is no need to inject these errors.

Fig. 13–14 show the likelihood of an early crash for errors in the source registers of `store` instructions. For inject-on-read, a `store` instruction has two source registers that are potential injection targets (see Table 5). The 1st register holds an address or a data variable to be stored in memory, while the 2nd register holds a memory address at which to store the content of the 1st register.

Fig. 13–14 show that errors injected in the two registers have quite different outcomes. In fact, for the first potential register, Fig. 13 shows that the highest 32 significant bits only result in a slightly higher percentage of early crashes (on average around 28% as compared to around 7% for the lowest 32 significant bits). On the other hand, the results

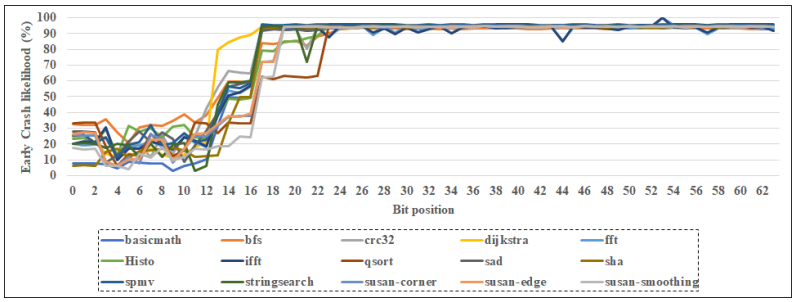


Figure 9. Likelihood of an early crash for different bit positions in the source register of `load` instructions (inject-on-read).

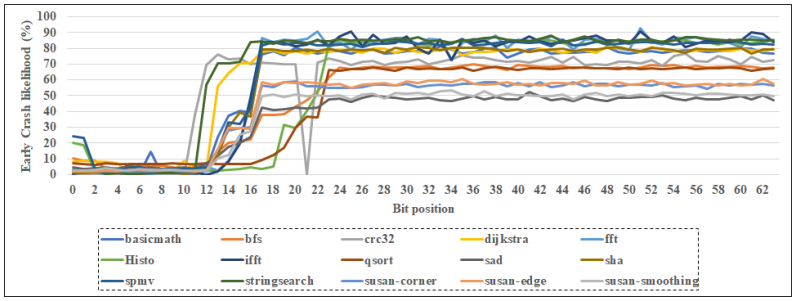


Figure 10. Likelihood of an early crash for different bit positions in the 1st source register of `getelementptr` instructions (inject-on-read).

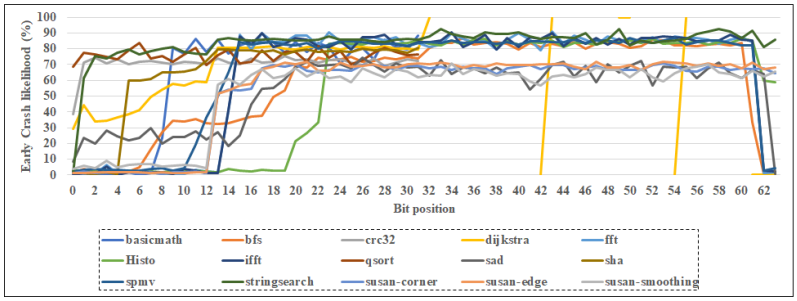


Figure 11. Likelihood of an early crash for different bit positions in the 2nd source register of `getelementptr` instructions (inject-on-read).

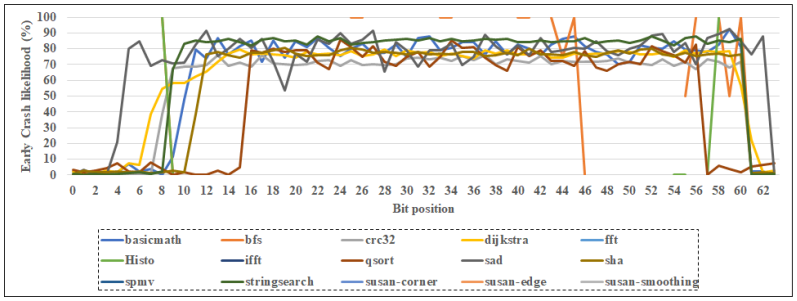


Figure 12. Likelihood of an early crash for different bit positions in the 3rd source register of `getelementptr` instructions (inject-on-read).

obtained for the second potential register (Fig. 14) illustrate that the 47 highest significant bits are significantly more sensitive to early crashes (on average, around 93% as opposed to around 12% for the lowest 32 significant bits). This indicates that the error space could be significantly pruned by removing the majority of the bits in the 2nd source register of `store` instructions from the error space (while classifying them as Detected in the outcome classification).

The variation between the results obtained for the two registers targeted in the `store` instruction is due to the type of data they hold. As discussed earlier (see §4.3.2.1) as well as in previous studies [36], [42], it is much more likely for an injected error to cause an early crash if the type of the data targeted is a memory address rather than a data variable.

Fig. 15 shows the likelihood of an early crash for different bits of the source register of the `sext` instruction. The figure shows that in general, the 10 highest significant bits are very sensitive in causing an early crash, whereas the lowest 6 insignificant bits are very insensitive in causing an early crash. It is interesting to note that similar to the results

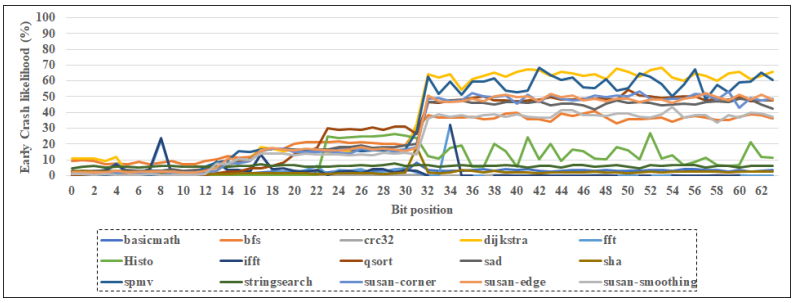


Figure 13. Likelihood of an early crash for different bit positions in the 1st source register of `store` instructions (inject-on-read).

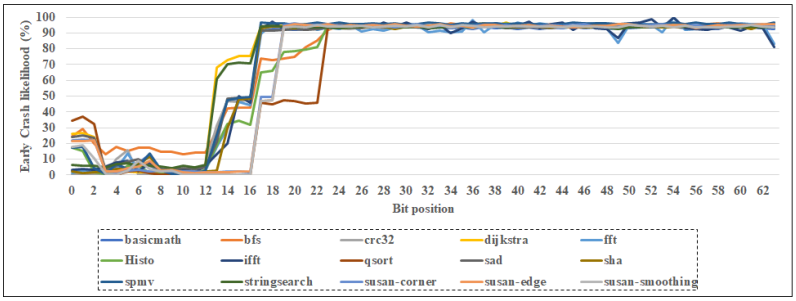


Figure 14. Likelihood of an early crash for different bit positions in the 2nd source register of `store` instructions (inject-on-read).

presented in Fig. 9 and Fig. 14, the sensitivity of the different bits of the `sext` instruction to causing an early crash follows a bi-modal distribution, i.e., a specific bit is either highly unlikely or highly likely to cause an early crash.

RQ7-answer (inject-on-read technique):
***`load` instruction*:** The 47 most significant bits of source registers show a high likelihood of causing an early crash (on average around 93%). These errors are therefore potential candidates for pruning.
***`getelementptr` instruction*:** Except few cases, higher significant bits are significantly more likely to cause early crashes. The sensitivity of the lowest 17 bits and the highest 47 bits to causing an early crash is on average around 24% and 72%, respectively.
***`store` instruction*:** The sensitivity of the 1st source registers in causing an early crash is significantly lower than the 2nd registers. Moreover, the likelihood of an error injected in the 47 most significant bits of the 2nd registers is on average around 93%; thus these bits could be removed from the error space while being classified as Detected.
***`sext` instruction*:** The sensitivity of the different bits of the `sext` instruction to causing an early crash follows a two-level step function, i.e., for each program under evaluation, a specific bit is either not sensitive or significantly sensitive to causing an early crash.

4.3.4.2 Identification of sensitive bits causing early crash in inject-on-write campaigns: We investigate 5 instructions as they contributed the most to the total number of early crashes: `getelementptr`, `load`, `sext`, `zext`, and `alloca` contributed to 37%, 34%, 8%, 6%, and 5% of the early crashes, respectively.

Fig. 16 shows the likelihood of an early crash for different bits of the destination register targeted in `getelementptr` instructions. The figure shows that the 17 highest significant bits are significantly more likely to result in early crashes (on average around 78% as opposed to 21% for the 17 lowest significant bits). As mentioned in §4.3.4.1, the `getelementptr` instruction is used to get the address of a sub-element of an aggregate data structure. Therefore, it performs address calculation, which is why errors in higher significant bits contributed the most to the early crashes.

Fig. 17 shows the likelihood of an early crash for different bits of the register targeted in `load` instructions. The figure shows that in most cases, the highest significant bits, starting from bit 32, are significantly more likely to result in early crashes (on average around 52% as opposed to around 15% for the 32 lowest significant bits). The increase in the higher percentage of early crashes starts from bit 16 and becomes more pronounced when moving from bit 31 to 32. In fact, the sensitivity of the 16 lowest significant bits to causing a crash is only around 5%, while the sensitivity is on average around 25% for bits 17 to 31.

The register targeted in inject-on-write campaigns may hold memory addresses, which are significantly sensitive in causing an early crash, especially if the bits targeted are one of the highest 47 significant bits. This is why the percentage of early crashes in Fig. 17 are higher for bits 17 to 63 (especially starting from bit 32). Moreover, the `load` instruction's register targeted in the inject-on-write campaigns may also hold a data variable, which is significantly less sensitive than a memory address. This is why the `load` instruction registers in inject-on-write campaigns are significantly less sensitive than those in inject-on-read campaigns.

Fig. 18 shows the likelihood of an early crash for different bits of the register targeted in the `sext` instructions. The figure shows that in most cases, the highest significant bits (starting from bit 17) are significantly more likely to result in early crashes (on average around 56% as opposed to 16% for the 17 lowest significant bits). Further, for some of the programs, the likelihood of the highest 3 significant

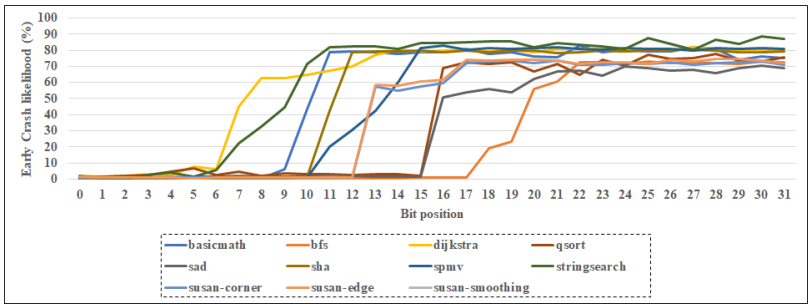


Figure 15. Likelihood of an early crash for different bit positions in the source register of `sext` instructions (inject-on-read). Note that fft, ifft, crc32, and histo programs are removed from the graph as they experienced no to insignificant injections in their `sext` instructions.

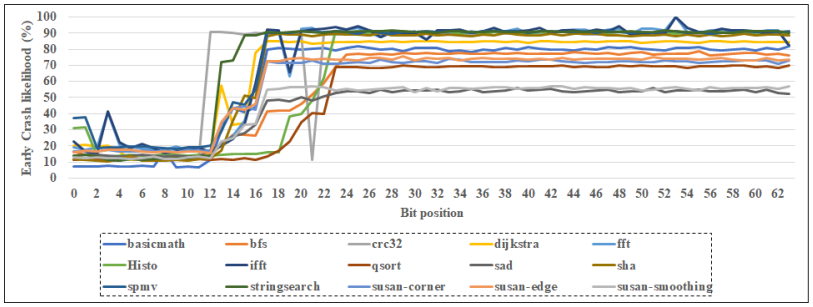


Figure 16. Likelihood of an early crash for different bit positions in the destination register of `getelementptr` instructions (inject-on-write).

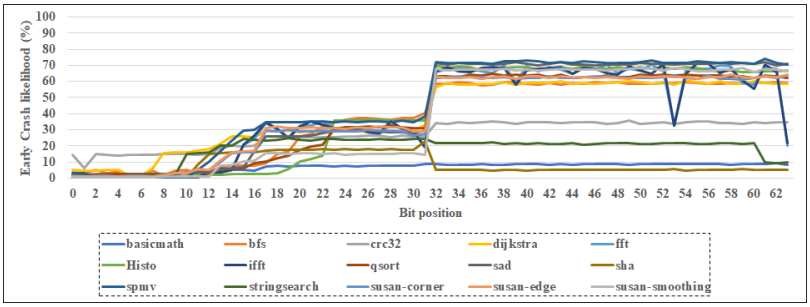


Figure 17. Likelihood of an early crash for different bit positions in the destination register of `load` instructions (inject-on-write).

bit of the register targeted causing an early crash is close to 0. This is likely because the highest significant bit is often a sign bit. The instruction performs a sign extension of a value to a specified type by copying the sign bit of the value until it reaches the bit size of the type specified.

Fig. 19 shows the likelihood of an early crash for different bits of the destination register targeted in `zext` instructions. Note that, basicmath, dijkstra, qsort, sha, and spmv programs are removed from the graph as there were either no or very few injections in the destination registers of `zext` in these programs. Also note that, in the case of the susan-corner, susan-edge and susan-smoothing programs, the target register was a 32-bit register. For these programs, the sensitivity of the 13 lowest significant bits is on average around 2%. For the other programs, although the general trend is that higher significant bits are more likely to cause early crashes, the bit position corresponding to the starting of the high sensitivity bits is program-dependent. The figure also shows that for these programs, the highest 2-3 significant bits have low likelihood of causing early crashes.

Fig. 20 shows the likelihood of an early crash for different bits of the register targeted in `alloca` instructions. The figure shows that in general, injections in higher significant bits starting from bit 17 are more likely to cause an early crash (on average, around 41% as opposed to around 11% for the 17 lowest significant bits). However, there are a few programs that exhibit fluctuations depending on the different bits targeted. Note that in this instruction, memory is allocated and a pointer is returned. Therefore, the register targeted holds a memory address. Depending on whether this register is read, the injected error may or may not be activated, which is the likely reason for the fluctuations.

Similar to the results from inject-on-read campaigns, the results obtained for the inject-on-write suggest that there is not much variation between the sensitivity of the higher significant bits to causing an early crash, for different instruction/program pairs. This means that by knowing the failure mode of one of these bits, the error space could be further pruned by removing the other high significance bits.

In summary,the inject-on-read campaigns are more likely to cause an early crash compared to inject-on-write campaigns. This is likely because in the case of inject-on-read,

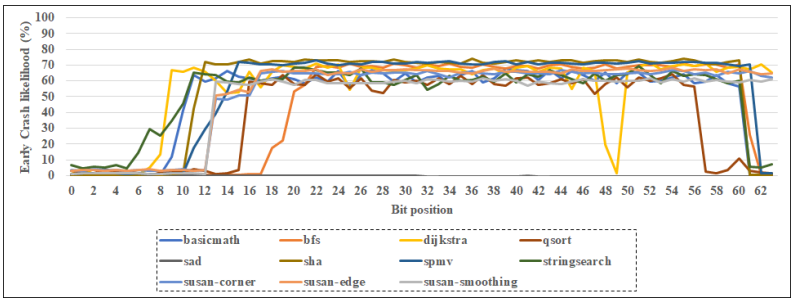


Figure 18. Likelihood of an early crash for different bit positions in the destination register of `sext` instructions (inject-on-write). Note that fft, ifft, crc32, and histo programs are removed from the graph as they experienced no or insignificant injections in their `sext` instruction.

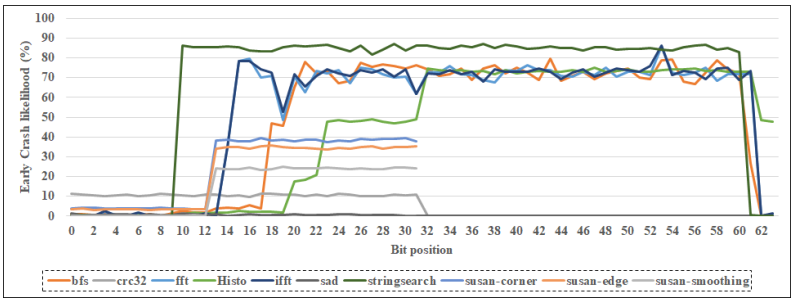


Figure 19. Likelihood of an early crash for different bit positions in the destination register of `zext` instructions (inject-on-write). Note that basicmath, dijkstra, qsort, sha, and spmv programs are removed from the graph as they experienced no or insignificant injections in their `zext` instruction.

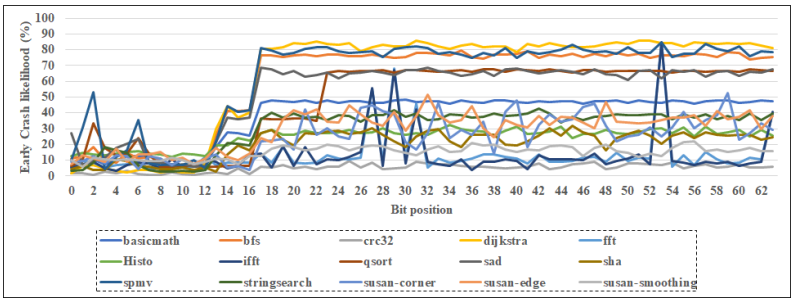


Figure 20. Likelihood of an early crash for different bit positions in the destination register of `alloca` instructions (inject-on-write).

the activation occurs at the same time as the injection, whereas for the inject-on-write campaigns, the activation only occurs when the erroneous register/content is read. Therefore, for inject-on-write campaigns, the erroneous register may not be read or be overwritten, which reduces the likelihood of the injected error to cause an early crash.

RQ7-answer (inject-on-write technique):
***`getelementptr` instruction*:** The highest 47 significant bits of source registers show on average around 78% of sensitivity to causing an early crash and could be removed from the error space while being classified as Detected.
***`load` instruction*:** The registers targeted could hold a memory address or a data variable. The 47 highest significant bits of the ones holding a memory address could be removed from the error space while being classified as Detected as they are significantly more sensitive in causing an early crash.
***`sext` instruction*:** Except few cases, the 47 highest significant bits are significantly more likely to result in early crashes (on average around 56% as opposed to 16% for the lowest 17 significant bits).
***`zext` instruction*:** Although the general trend is that higher significant bits are more likely in causing early crashes, the bit position corresponding to the starting of the high sensitivity bits is program-dependent.
***`alloca` instruction*:** In general, injections in the 47 highest significant bits are more likely to result in an early crash (on average around 41% as opposed to around 11% for the 17 lowest significant bits). The registers targeted hold a memory address and if this register is not read, the injected error will not be activated.

5 Summary and Conclusions

In this paper, our goal was to study the impact of multiple-bit errors in programs and to find ways to explore and reduce the multiple-bit fault injection space (error space). This is important as previous studies [5], [6], [7] have shown that soft errors often manifest as multiple-bit errors at the software level, and hence we need efficient methods to inject multiple-bit errors in software and evaluate their effects.

Prior work had considered at most two bit-flips, and did not cover the entire space of multiple-bit errors. We performed a comprehensive analysis of the parameter space of multiple bit-flips to identify which parameters affect the SDCs for a program. Our findings are:

- The SDC results of the single bit-flip model are close to the results for the multiple bit-flip model (except for 2% of the multiple bit-flip campaigns which result in an SDC percentage that was more than 5 percentage points higher than that obtained for the corresponding single bit-flip campaigns) across the majority of programs and parameter values, with a few exceptions. This holds regardless of whether the multiple bit-flip injections are in the same register or in different registers. This is because an increase in the number of injected faults also results in an increase in the likelihood that exceptions are raised (see Table 4), thus reducing the percentage of SDCs.
- With the above said, the single bit-flip model is not sufficient to establish conservative upper bounds on the SDC results (i.e., pessimistic percentages of SDCs) under multiple-bit errors. However, for most programs, the pessimistic percentage of SDCs for the multiple-bit error model is achieved under relatively few multiple-bit errors (2 errors with the inject-on-read technique, and 3 errors with the inject-on-write technique).
- The dynamic window size parameter value does not matter much when the inject-on-read technique is used, but it matters when the inject-on-write technique is used. In the latter case, the highest percentage of SDCs is achieved when the window size is low, i.e., less than 5 dynamic instructions in most cases.
- Only a very small fraction of single bit-flip errors that result in Detection lead to SDCs under multiple bit-flips in which the starting location is the same as the single bit-flip error. Therefore, to maximize the SDCs uncovered by multiple bit-flip injections, one needs to inject only into the program locations in which single bit-flip error injections led to benign outcomes.
- Around 75% of the total number of experiments conducted had resulted in early crashes, which can be removed from the error space as they do not lead to SDCs. More than 90% of the total number of early crashes are caused as a result of injections in registers used by instructions `getelementptr`, `load`, `sext`, `zext` `store`, and `alloca`. By leveraging these early crashes (90%), and combining this result with the aggregate percentage of crashes (75%), 67% of the injections can be pruned for multiple bit-flip injections. Further, most of the higher significant bits of registers used by these instructions can be pruned as injecting errors in them will likely cause early crashes.

ACKNOWLEDGMENTS

This work was partially supported by the NSERC Discovery Grants Programme, and the VALU3S project, which has received funding from the ECSEL Joint Undertaking (JU) under grant agreement No 876852. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Austria, Czech Republic, Germany, Ireland, Italy, Portugal, Spain, Sweden, Turkey.

REFERENCES

[1] S. Borkar, "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation," *IEEE Micro*, vol. 25, no. 6, pp. 10–16, 2005.

[2] A. Chatzidimitriou, G. Papadimitriou, C. Gavanas, G. Katsoridas, and D. Gizopoulos, "Multi-bit upsets vulnerability analysis of modern microprocessors," in *IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2019.

[3] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "SWIFT: Software implemented fault tolerance," in *Proceedings of the 3rd International Symposium on Code Generation and Optimization*, ser. CGO '05. IEEE Computer Society, 2005, pp. 243–254.

[4] M. Rebaudengo, M. S. Reorda, and M. Violante, "A new approach to software-implemented fault tolerance," *Journal of Electronic Testing*, vol. 20, no. 4, pp. 433–437, 2004.

[5] H. Cho, S. Mirkhani, C.-Y. Cher, J. A. Abraham, and S. Mitra, "Quantitative evaluation of soft error injection techniques for robust system design," in *Proceedings of the 50th ACM/EDAC/IEEE Design Automation Conference*, ser. DAC '13. ACM, 2013, pp. 1–10.

[6] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, "EMAX: An automatic extractor of high-level error models," in *Proceedings of the 9th AIAA Computing in Aerospace Conference*, 1993, pp. 1297–1306.

[7] J. F. Ziegler et al., "IBM experiments in soft fails in computer electronics (1978-1994)," *IBM Journal of Research and Development*, vol. 40, no. 1, pp. 3–18, 1996.

[8] W. G. Bouricius, W. C. Carter, and P. R. Schneider, "Reliability modeling techniques for self-repairing computer systems," in *Proceedings of the 24th National Conference*, ser. ACM '69. ACM, 1969, pp. 295–309.

[9] T. F. Arnold, "The concept of coverage and its effect on the reliability model of a repairable system," *IEEE Transactions on Computers*, vol. C-22, no. 3, pp. 251–254, 1973.

[10] B. Fang, Q. Lu, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, "ePVF: An enhanced program vulnerability factor methodology for cross-layer resilience analysis," in *46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2016, pp. 168–179.

[11] Q. Lu, M. Farahani, J. Wei, A. Thomas, and K. Pattabiraman, "LLFI: An intermediate code-level fault injection tool for hardware faults," in *2015 IEEE International Conference on Software Quality, Reliability and Security*, 2015, pp. 11–16.

[12] F. Ayatolahi, B. Sangchoolie, R. Johansson, and J. Karlsson, "A study of the impact of single bit-flip and double bit-flip errors on program execution," in *Proceedings of the 32nd International Conference on Computer Safety, Reliability, and Security*, ser. SAFECOMP 2013. Springer-Verlag New York, Inc., 2013, pp. 265–276.

[13] F. Adamu-Fika and A. Jhumka, "An investigation of the impact of double bit-flip error variants on program execution," in *Proceedings of the 15th International Conference on Algorithms and Architectures for Parallel Processing*. Springer International Publishing, 2015, pp. 799–813.

[14] E. Touloupis, J. A. F. Member, V. A. Chouliaras, and D. D. Ward, "Study of the effects of SEU-induced faults on a pipeline protected microprocessor," *IEEE Transactions on Computers*, vol. 56, no. 12, pp. 1585–1596, 2007.

[15] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization*, ser. CGO '04. IEEE Computer Society, 2004, pp. 75–86.

[16] W. Gu, Z. Kalbarczyk, and R. K. Iyer, "Error sensitivity of the linux kernel executing on PowerPC G4 and Pentium 4 processors," in *2004 IEEE/IFIP International Conference on Dependable Systems and Networks*, 2004, pp. 887–896.

[17] D. Skarin, R. Barbosa, and J. Karlsson, "GOOFI-2: A tool for experimental dependability assessment," in *2010 IEEE/IFIP International Conference on Dependable Systems Networks*, 2010, pp. 557–562.

[18] S. K. Sastry Hari, R. Venkatagiri, S. V. Adve, and H. Naeimi, "Ganges: Gang error simulation for hardware resiliency evaluation," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 61–72, 2014.

[19] S. K. S. Hari, S. V. Adve, H. Naeimi, and P. Ramachandran, "Relyzer: Exploiting application-level fault equivalence to analyze application resiliency to transient faults," in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII. ACM, 2012, pp. 123–134.

[20] 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)*. IEEE, 2018, pp. 27–38.

[21] M. de Kruijf, S. Nomura, and K. Sankaralingam, "Relax: An architectural framework for software recovery of hardware faults," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10. ACM, 2010, pp. 497–508.

[22] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, "Shoestring: Probabilistic soft error reliability on the cheap," in *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XV. ACM, 2010, pp. 385–396.

[23] D. S. Khudia and S. Mahlke, "Harnessing soft computations for low-budget fault tolerance," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47. IEEE Computer Society, 2014, pp. 319–330.

[24] A. Thomas and K. Pattabiraman, "LLFI: An intermediate code level fault injector for soft computing applications," in *Workshop on Silicon Errors in Logic System Effects (SELSE)*, 2013.

[25] J. Pan, "The dimensionality of failures - a fault model for characterizing software robustness," in *Proceedings of the International Symposium on Fault-Tolerant Computing*, 1999.

[26] J. Pan, P. Koopman, and D. Siewiorek, "A dimensionality model approach to testing and improving software robustness," in *Proceedings of the 1999 IEEE AUTOTESTCON*, 1999, pp. 493–501.

[27] L. Rashid, K. Pattabiraman, and S. Gopalakrishnan, "Characterizing the impact of intermittent hardware faults on programs," *IEEE Transactions on Reliability*, vol. 64, no. 1, pp. 297–310, 2015.

[28] C.-K. Chang, S. Lym, N. Kelly, M. B. Sullivan, and M. Erez, "Evaluating and accelerating high-fidelity error injection for hpc," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. IEEE Press, 2018, p. 45.

[29] R. Venkatagiri, A. Mahmoud, S. K. S. Hari, and S. V. Adve, "Approxilyzer: Towards a systematic framework for instruction-level approximate computing and its application to hardware resiliency," in *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–14.

[30] R. Barbosa, J. Vinter, P. Folkesson, and J. Karlsson, "Assembly-level pre-injection analysis for improving fault injection efficiency," in *Proceedings of the 5th European Dependable Computing Conference*. Springer Berlin Heidelberg, 2005, pp. 246–262.

[31] B. Nie, L. Yang, A. Jog, and E. Smirni, "Fault site pruning for practical reliability analysis of GPGPU applications," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 749–761.

[32] L. Yang, B. Nie, A. Jog, and E. Smirni, "Practical resilience analysis of GPGPU applications in the presence of single- and multi-bit faults," 2020, pp. 1–1.

[33] M. Kaliorakis, D. Gizopoulos, R. Canal, and A. Gonzalez, "Merlin: Exploiting dynamic instruction behavior for fast and accurate microarchitecture level reliability assessment," ser. ISCA '17, 2017.

[34] H. Madeira and J. G. Silva, "Experimental evaluation of the fail-silent behavior in computers without error masking," in *Proceedings of the 24th IEEE International Symposium on Fault-Tolerant Computing*, 1994, pp. 350–359.

[35] P. Yuste, J. C. Ruiz, L. Lemus, and P. Gil, "Non-intrusive software-implemented fault injection in embedded systems," in *Proceedings of the 1st Latin-American Symposium on Dependable Computing*. Springer Berlin Heidelberg, 2003, pp. 23–38.

[36] B. Sangchoolie, F. Ayatolahi, R. Johansson, and J. Karlsson, "A comparison of inject-on-read and inject-on-write in ISA-level fault injection," in *11th European Dependable Computing Conference*, 2015, pp. 178–189.

[37] H. Schirmeier, M. Hoffmann, R. Kapitza, D. Lohmann, and O. Spinczyk, "Fail*: Towards a versatile fault-injection experiment framework," in *ARCS Workshops*, 2012, pp. 1–5.

[38] P. Racunas, K. Constantinides, S. Manne, and S. S. Mukherjee, "Perturbation-based fault screening," in *Proceedings of the 13th IEEE International Symposium on High Performance Computer Architecture*, ser. HPCA '07. IEEE Computer Society, 2007, pp. 169–180.

[39] H. Schirmeier, C. Borchert, and O. Spinczyk, "Avoiding pitfalls in fault-injection based comparison of program susceptibility to soft errors," in *45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2015, pp. 319–330.

[40] 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 4th Annual IEEE International Workshop on Workload Characterization*, ser. WWC-4, 2001, pp. 3–14.

[41] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, vol. 127, 2012.

[42] B. Sangchoolie, R. Johansson, and J. Karlsson, "Light-weight techniques for improving the controllability and efficiency of ISA-level fault injection tools," in *2017 IEEE 22nd Pacific Rim International Symposium on Dependable Computing (PRDC)*, Jan 2017, pp. 68–77.

Behrooz Sangchoolie is a researcher in the Department of Electrification and Reliability at RISE Research Institutes of Sweden. He is the technical coordinator of National and European research projects in the area of dependable and secure computing and has served on many program committees for conferences and workshops in the area. His current research interests include the use of fault and attack injection experiments for dependability and security assessment of computer systems as well as to conduct interplay analyses between non-functional requirements such as safety and security.

Karthik Pattabiraman is an associate professor in the Department of Electrical and Computer Engineering at the University of British Columbia (UBC). He has served on the conference program committees of DSN, ISSRE and other dependability conferences. He was General Chair for the IEEE Pacific Rim International Symposium on Dependable Computing (PRDC) in 2013, and Program Co-Chair for the IEEE International Symposium on Software Reliability Engineering (ISSRE) in 2017, and IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) in 2019. His research interests are in dependable computing, software engineering and computer security. He is a senior member of the IEEE and ACM, and a vice-chair of the IFIP Working Group on Dependable Computing and Fault Tolerance (WG 10.4).

Johan Karlsson is a professor in the Department of Computer Science and Engineering at Chalmers University of Technology. He has served on numerous program committees for conferences and workshops in the area of dependable computing. He was General Chair for the 12th European Dependable Computing Conference (EDCC) in 2016, and Program Co-Chair for the IEEE/IFIP 45th International Conference on Dependable Systems and Networks (DSN) in 2015. He was Head of the Computer Science and Engineering department at Chalmers from 2015 to 2019. His research interests include the use of fault injection experiments and probabilistic models for assessing and evaluating dependability, robustness and safety of embedded computer systems. He is a member of the IEEE, the IEEE Computer Society, and the IFIP Working Group on Dependable Computing and Fault Tolerance (WG 10.4).