Peppa-X: Finding Program Test Inputs to Bound Silent Data Corruption Vulnerability in HPC Applications

Md Hasanur Rahman
University of Iowa
Iowa City, IA, USA
mdhasanur-rahman@uiowa.edu

Aabid Shamji
University of Iowa, USA
Iowa City, IA, USA
aabid-shamji@uiowa.edu

Shengjian Guo
Baidu Security
Sunnyvale, CA, USA
sjguo@baidu.com

Guanpeng Li
University of Iowa, USA
Iowa City, IA, USA
guanpeng-li@uiowa.edu

ABSTRACT

Transient hardware faults have become prevalent due to the shrinking size of transistors, leading to silent data corruptions (SDCs). Therefore, HPC applications need to be evaluated (e.g., via fault injections) and protected to meet the reliability target. In the evaluation, the target programs exercise with a set of given inputs which are usually from program benchmark suite. However, these inputs rarely manifest the SDC vulnerabilities, leading to over-optimistic assessment and unexpectedly higher failure rates in production. We propose Peppa-X, which efficiently identifies the test inputs that estimate the bound of program SDC resiliency. Our key insight is that the SDC sensitivity distribution in a program often remains stationary across input space. Thereby, we can guide the search of SDC-bound inputs by a sampled distribution. Our evaluation shows that Peppa-X can identify the SDC-bound input of a program that existing methods cannot find even with 5x more search time.

KEYWORDS

Error Resilience, Fault Injection, Silent Data Corruption, Software Testing, Input Fuzzing, Program Analysis, Error Propagation, High Performance Computing

ACM Reference Format:
Md Hasanur Rahman, Aabid Shamji, Shengjian Guo, and Guanpeng Li. 2021. Peppa-X: Finding Program Test Inputs to Bound Silent Data Corruption Vulnerability in HPC Applications. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21), November 14–19, 2021, St. Louis, MO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3458817.3476147

1 INTRODUCTION

Transient hardware faults (e.g., soft errors) have been predicted to increase in future processors due to growing system scales, progressive technology scaling, and lowering operating voltages [49]. In the past, those faults were typically masked through hardware-only solutions like redundancy and voltage guard bands. Nowadays, deploying such solutions is increasingly challenging concerning the significant energy cost, whereas energy is becoming a first-class constraint in microprocessor design [10]. The problem is exacerbated in High Performance Computing (HPC) systems because of the large scale, and it has been one of the top 10 challenges in HPC [37]. As a result, researchers expect that software needs to tolerate hardware faults with low overheads in HPC [3, 5, 31, 48, 54, 55, 59].

Hardware faults can cause programs to fail by crashing, hanging, or producing incorrect program outputs, also known as silent data corruptions (SDCs). SDCs are severe concerns in practice because there is no noticeable symptom that the program failed, and the contaminated data continues propagating in program execution, resulting in the wrong final output. Hence, developers have to evaluate the SDC probability of their applications before deployment. If the assessment result fails to meet the resiliency target, developers must selectively add extra protection to the most vulnerable parts of the application until satisfying the target, without causing much performance and energy overheads [1, 15, 20, 28, 29, 32, 42, 53].

Statistical fault injection is the common approach for evaluating program resilience. In the evaluation, the target program repeats executions with a test input for thousands of fault injection (FI) campaigns to achieve statistical significance in the measurement [17, 20, 41, 46, 51], thus obtaining the program's overall SDC probability. In each FI campaign, a single fault is injected into a randomly sampled instruction during the execution. Existing studies primarily use the test inputs provided in the benchmark suites of the program [17, 28, 32, 38, 46]. However, those provided inputs are often for performance and functionality testing. Evaluating with the inputs rarely exploits the vulnerabilities caused by hardware faults, leading to over-optimistic results for SDC evaluation and protection (we show that in Section 5). Consequently, HPC applications, which run with cumulatively diverse inputs in practice, may suffer from much higher-than-expected failure rates in the production environment. Those unexpected failures seriously compromise the HPC reliability and cause the applications fail to meet the reliability target. Moreover, a recent paper has disclosed persuasive findings in its computation infrastructures and confirms the observation, revealing that applications are often under-evaluated for their resiliency in the evaluation phase and reporting more pervasive SDCs

SC '21, November 14–19, 2021, St. Louis, MO, USA

ACM ISBN 978-1-4503-8442-1/21/11.
https://doi.org/10.1145/3458817.3476147

at scale in production [14]. The erroneous application outputs and loss of data can cost months of debugging efforts [14]. Therefore, it is crucially important to bound program SDC probability in the evaluation and so the program can be properly protected in order to meet the reliability target. We call such an input *SDC-bound input*, which can help developers stress-test HPC applications in the evaluation and hence protect them accordingly. Unfortunately, such test inputs for SDC resiliency are not available in any HPC benchmark suites.

A strawman method to find the SDC-bound input is to evaluate every possible program input via statistical FIs. This approach appears to be unrealistic in terms of the vast input search space. Also, the FIs can be prohibitively time-consuming – even a single HPC program execution may take several hours. Repeating thousands of FI campaigns for each input is by no means practical. Looking into the software engineering community, researchers have drawn upon code coverage to guide the search of the inputs that exhibit software bugs [25]. However, we find a negligible correlation between the code coverage of an input and the SDC vulnerability (Section 3); thereby, one cannot simply use existing metrics like code coverage to search for SDC-bound inputs.

In this paper, we propose Peppa-X, a compiler-based technique that approximates the upper bound of the SDC probability in a program without conducting extensive FIs. Peppa-X leverages static and dynamic program analysis methods to identify program SDC-bound input candidates that iteratively update the SDC probability bound. The key insight is that the SDC sensitivity distribution across program regions often remains stationary in program input space. Therefore, we can guide the search of program input towards the explorations of more vulnerable program paths based on the distribution. The guided search helps maximize the program SDC probability and identifies the corresponding SDC-bound input. Besides, we develop two pruning heuristics that can efficiently estimate the SDC sensitivity to avoid the unnecessary search. Furthermore, we design a genetic-algorithm-based dynamic fuzzing method to help locate SDC-bound input, which drastically speeds up the entire process of Peppa-X. *To our best knowledge, we are the first one who designs automated program analysis techniques to identify test inputs that bound the SDC probability in HPC applications.*

We outline the main contributions of this work as follows:

- We propose Peppa-X, an automated technique that efficiently identifies the SDC-bound input set that exploits the SDC vulnerability and depicts the upper bound of program SDC probability in a timely manner.
- We implement Peppa-X with compiler-based static analysis and fuzzing-based dynamic search. The found inputs lead to as much as 32x higher SDC probability than the baseline result. Meanwhile, the baseline method cannot find a comparable result even given 5x more search time.
- We apply the SDC-bound input to stress test applications protected by the popular selective instruction duplication. Experiments show that the protections are often compromised by the SDC-bound input, resulting in significantly lower-than-expected SDC coverage (around 2.59x coverage losses on average).

2 BACKGROUND

In this section, we first present our fault model, then define the terms we use, followed by a brief description of the LLVM compiler used in Peppa-X. Finally, we provide a brief overview of the genetic algorithm used in our method.

2.1 Fault Model

In this paper, we consider transient hardware faults in the processor's computing components, including pipeline stages, flip-flops, and functional units. We do not consider faults in the memory or caches, as we assume that ECC protects these. Likewise, we do not consider faults in the processor's control logic. Further, we ignore faults in the instruction's encoding as they can be detected through other means, such as error-correcting codes. Finally, we assume that the program does not jump to arbitrary, illegal addresses due to execution faults, as this problem can be mitigated by control-flow checking techniques [44]. However, the program may take a legal but wrong branch. That is, the execution path is legal, but the branch selection may be wrong due to propagated faults. Our fault model is in line with other work in the area [1, 4, 11, 15, 17, 23**?**].

2.2 Terms and Definitions

- **Fault Occurrence:** The occurrence of a transient hardware fault in the processor. The fault may or may not result in an error in the running program.
- **Fault Activation:** The event corresponding to the manifestation of the fault to the software, i.e., the fault becomes an error and corrupts some portion of the execution state (e.g., register, memory location). The error may or may not result in a failure (i.e., SDC, crash or hang).
- **Crash:** The raising of a hardware trap or exception due to the error, because the program attempted to perform an action it should not have (e.g., read outside its memory segments). The OS terminates the program as a result.
- **Silent Data Corruption (SDC):** A mismatch appears between the outputs of a program's faulty execution and error-free execution of the same program.
- **Benign Faults:** Program output matches that of the error-free execution even though a fault occurred during its execution. This fact means either the fault was masked or overwritten by the program.
- **SDC Probability:** We refer to the SDC probability as the probability of an SDC given that the fault was activated – other work uses a similar definition [15, 22, 32, 41, 51, 56].
- **Static Instruction:** The instruction in the program code.
- **Dynamic Instruction:** An instance of static instruction that is executed by the CPU in a program execution.

2.3 LLVM Compiler

We perform the program analysis, FI experiments, and the tool implementation based on the LLVM compiler [30]. Our choice of LLVM is motivated by three reasons. First, LLVM uses a typed intermediate representation (IR) that can easily represent source-level constructs. In particular, it preserves the names of variables and functions, which makes source mapping feasible. This feature allows us to perform a fine-grained analysis of which program

locations cause the specific failures by mapping the error sites back to the source code. Secondly, LLVM IR is a platform-neutral representation that abstracts away low-level assembly language and hardware details. This feature greatly aids in porting our analysis to various architectures; it also simplifies the handling of different cases of assembly language formats. Finally, LLVM IR has been accurate for doing FI studies [46, 51], and a set of fault injectors for LLVM [3, 33, 47, 51] already exists. Several studies in the area of resilience also utilize the LLVM infrastructure [3, 15, 22, 28, 32]. Therefore, in this paper, we mean an instruction at the LLVM IR level when we say instruction. However, our methodology is general rather than tied to LLVM.

2.4 Genetic Algorithm

Genetic algorithm (GA) [24] is a meta-heuristic search algorithm inspired by natural evolution. The algorithm starts with an initial set of candidate solutions, which are collectively called the *population*. Then, a *fitness function* that computes the fitness score of a candidate drives forward the algorithm. The fitness score considers how good the candidate is at solving the problem. At each phase, the algorithm chooses some candidate solutions from the population for *recombination operations*. There are two types of recombination operations — *crossover* and *mutation*. *Crossover* tends to narrow the search and move toward an optimal solution. Two randomly chosen candidates get exchanged in the *crossover* operation to generate a better solution from a good one. By comparison, *mutation* only randomly selects one candidate. It flips a bit or an entity in the candidate solution, which expands the search exploration. In general, recombination operations give rise to new, better-performing candidates, which contribute to population growth. In contrast, members that have poor fitness scores are gradually eliminated. Each such process is called a *generation* and is repeated until either a population member has the desired fitness score (hence a solution is found) or the algorithm terminates after the time threshold.

3 INITIAL FAULT INJECTION STUDY

In this section, we design experiments to show how the overall SDC probability of a program correlates to program input, code coverage, and per-instruction SDC probability, and discuss the implications of the observations.

Table 1: Characteristics of Benchmarks

Benchmark	Suite/Author	Description	No. of Static Instructions
Pathfinder	Rodinia	Use dynamic programming to find a path on a 2-D grid	372
Needle	Rodinia	A nonlinear global optimization method for DNA sequence alignments	1069
Particlefilter	Rodinia	Statistical estimator of the location of a target object given noisy measurements of that target's location in a Bayesian framework	1869
CoMD	Mantevo	Molecular dynamics algorithms and workloads	11457
Hpccg	Mantevo	A simple conjugate gradient benchmark code for a 3D chimney domain on an arbitrary number of processors	1975
Xsbench	CESAR	A mini-app representing a key computational kernel of the Monte Carlo neutronics application	2366
FFT	SPLASH-2	1D fast Fourier transform using six-step FFT method	2138

3.1 Experiment Setup

3.1.1 Benchmarks. We use seven applications from four standard benchmark suites for this study. We search from recent works in the area of HPC resilience [4, 29, 36, 40, 46, 57], and choose applications based on two criteria: (1) Compatibility with our toolset (i.e., we can compile them into LLVM IR to work with our fault injector); and (2) Ability to generate diverse inputs for our experiments. For the latter criteria, we choose applications that take numeric values as their program inputs rather than binary files or files of unknown formats for the ease of input generations. As a result, seven benchmarks meet the criteria, they are listed in Table 1. These benchmarks are frequently used in HPC resilience studies, and we consider them as representative applications in HPC domains.

3.1.2 Input Generation. Since all the chosen benchmarks take numerical values as their inputs, we randomly generate values and preserve the useful ones to run the benchmark programs. The selection of the generated inputs performs on two rules. First, the input should not lead to any reported errors or exceptions that halt the program execution, as the error-introducing input may not represent the ordinary application behavior in production. Second, the number of dynamically executed instructions by the input should not exceed 40 billion, as such input can make the experiment time reasonable. After the selection, we keep 30 random inputs for each benchmark to perform the initial FI study. The average number of executed instructions per input is around 4.43 billion, which is in line with what has been used in prior works [32, 38**?**].

3.1.3 Fault Injection Tool. We use LLVM Fault Injector (LLFI) [51] to perform FI experiments. LLFI has been accurate in studying SDCs [46, 51]. As we only consider transient errors in the computing components, we use LLFI to inject single bit flips into a random instruction's return value. We consider single bit flips since it is the de-facto fault model for simulating transient faults in the literature [1, 15, 20, 23, 38**?**]. Despite the concerns about the usefulness of using single-bit flip faults for FI to model soft errors [9], a recent study [47] has shown that there is little difference in SDC probabilities between the single and multiple bit flips at the application level. Therefore, we adopt single bit flips in our SDC-related evaluation.

3.1.4 FI Methodology. To measure the overall SDC probability of a program, we inject 1,000 random faults to each benchmark for each input. To derive the per-instruction SDC probabilities of programs, we inject 100 random faults to each static instruction of each benchmark on each input in order to balance our experiment time. Our FI measurement yields an error bar from 0.26% to 3.10% for the 95% confidence intervals. This error range is comparable with other related works [15, 17, 20, 32, 46].

3.2 Results and Observations

After the FI experiments with random inputs, we measure the overall SDC probabilities of each benchmark. Then, we examine the variance of code coverage and per-instruction SDC probabilities against different inputs.

3.2.1 Overall SDC Probability. Figure 1 compares the overall SDC probability of each benchmark with the 30 random inputs. Each blue bar represents the range of SDC probability. The slim red mark

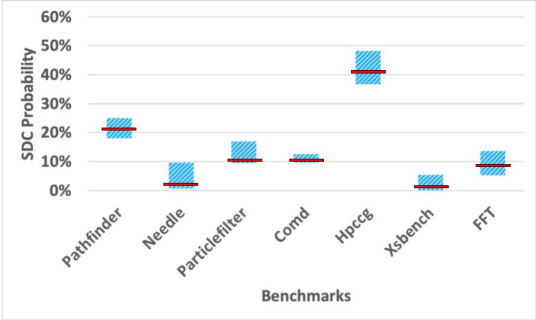


Figure 1: Range of Overall Program SDC Probability across Multiple Inputs; Red marks indicate SDC Probability measured from default reference inputs

in each blue bar indicates the SDC probability measured with the provided test inputs from the benchmark suite or existing works' dataset [17, 32, 46]. We call such input the *default reference input* of the benchmark. As seen, the bottom and top bounds of each blue bar vary in a wide range and the range is highly application-dependent. For example, in Hpccg, the range spans from 36.75% to 48.20%, whereas it is only from 9.55% to 12.58% in CoMD. Note these ranges are observed based on the random inputs. The ranges are even more expansive when running the benchmarks with the SDC-bound inputs found by PEPPA-X. We will show this later in Section 5. The reason why a program SDC probability changes under different inputs is that running different inputs may change the data-flow and control-flow in program executions. The changed execution flows consequently lead faulty values (originated from a single bit flip) to go through different combinations of program regions where masking (or propagation) effects are different. Thus, program input changes may result in different program SDC probabilities. Similar observations have been made in [13, 16, 32]

We also observe that the red marks are all in the lower half of the blue bars. For example, the highest SDC probability with the random inputs is 5.50% in Xsbench, whereas it is only 1.17% with its default reference input. We observe similar situations in other benchmarks too. This observation implies that the default reference inputs cannot sufficiently expose the SDC vulnerabilities in software programs. Even worse, there are no existing dedicated test inputs for resilience evaluation purposes. The default reference inputs are more for the performance assessment or functionality testing rather than for resilience evaluation.

3.2.2 Code Coverage and SDC Vulnerability. Finding test inputs that exhibit software bugs has been studied for a long time in software engineering research [25]. Researchers use code coverage as a metric to evaluate whether the test inputs will likely reveal more software bugs [26, 27]. While studies have shown code coverage can effectively guide the search of bug-triggering program inputs that expose software bugs, it is unclear whether code coverage can help exploit hardware fault vulnerability.

To answer the question, we profile the code coverage (based on static instructions) on running the random inputs for each benchmark and then compare the coverage data with the program SDC probabilities of the inputs. Table 2 shows Spearman's ranking correlation coefficient between code coverage and program SDC probability regarding the inputs in each benchmark. As seen, the correlation coefficients are notably low, with an average value of only 0.01. This result proves that code coverage alone is unlikely to guide an efficient search of inputs to exploit SDC vulnerability in programs.

Table 2: Correlation between Code Coverage and Program SDC Probability across Different Inputs

Pathfinder	Needle	Particlefilter	CoMD	Hpccg	Xsbench	FFT
0.00	-0.29	0.17	-0.18	0.00	0.38	0.00

3.2.3 Per-Instruction SDC Probabilities. We now examine the per-instruction SDC probabilities across multiple inputs. We use per-instruction SDC probabilities as a proxy to the SDC sensitivity distribution in a program. Here the term instruction refers to static instruction of a program. Figure 2 shows the range of per-instruction SDC probabilities across different inputs. For brevity, we use CoMD as an example for illustration purposes. Due to the enormous static instructions in the benchmarks, it is impractical to present the results of all instructions in the figure. Instead, we choose to sample 10 static instructions in CoMD for the illustration purposes.

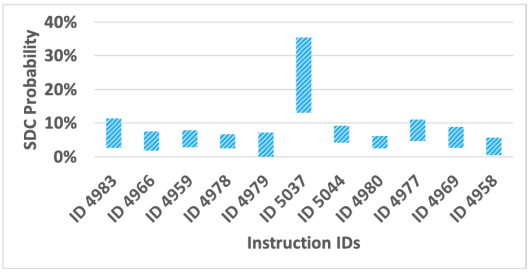


Figure 2: Range of Per-Instruction SDC Probabilities in CoMD across Multiple Inputs

Figure 2 shows that the per-instruction SDC probability differs a lot. The range of the per-instruction SDC probability varies as well. For example, *Instruction ID 5037* in CoMD has the highest SDC probability of 35.38%, whereas *Instruction ID 4958* has no more than 5.66% SDC probability across all inputs.

We notice that some instructions in a program are always vulnerable to SDCs regardless how inputs are changed and vice versa. For example, *Instruction ID 5037* in CoMD always produces a higher SDC probability over different inputs than other program instructions. Similar situations can be observed in the rest of the benchmarks. Here we consider this finding is crucial because it leads us to hypothesize that the ranking of each instruction's SDC vulnerability, or the SDC sensitivity distribution, is stable in a program regardless of the changing inputs.

To further investigate the SDC sensitivity distribution against changing inputs, we measure the Spearman's Ranking coefficient for each benchmark when comparing the ranking of per-instruction SDC probability among different inputs. Given an input for a benchmark, we measure the SDC probability of every instruction, then

Table 3: Correlation Coefficient between the Rankings of Per-Instruction SDC Probabilities with Different Inputs

Pathfinder	Needle	Particlefilter	CoMD	Hpccg	Xsbench	FFT
0.92	0.79	0.90	0.90	0.96	0.59	0.77

rank instructions by the SDC probabilities. So there is a rank list of instructions regarding an input. We then compute Spearman's Ranking correlation pairwise between all the rank lists, and take an average of them as the coefficient for the benchmark. Table 3 shows the results. A higher value of the coefficient indicates a stable ranking of the per-instruction SDC probability. To be specific, the instruction with higher per-instruction SDC probability retains its high value relative to that of other instructions on the changing inputs.

That is, *no matter how program input changes, the SDC sensitivity distribution of the program tends to remain stationary.* This observation allows us to design a search technique that finds proper inputs to explore more vulnerable program regions based on the distribution, thus obtaining higher program SDC probability.

4 METHODOLOGY

In this section, we first present the overall design of Peppa-X and then explain the details in each core component.

4.1 Overall Design of Peppa-X

Figure 3 shows the overall workflow of Peppa-X. Users of Peppa-X only need to provide the source code of the target program. Peppa-X will automatically search and try to generate the best SDC-bound input within the assigned time budget. The entire search process is automatic without requiring any interventions from the user. We have made Peppa-X publicly available[1].

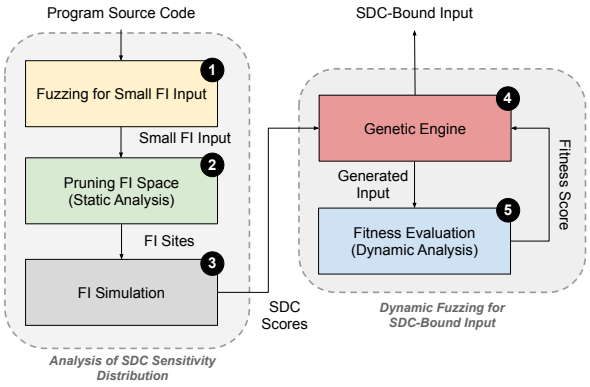


Figure 3: Workflow of Peppa-X

Since the SDC sensitivity distribution in a program is stable over different inputs, we design the search method that drives the program execution towards exploring more vulnerable program regions to maximize program SDC probability and generate the SDC-bound input accordingly. We utilize Genetic Algorithm (GA) as the search method to make optimization decisions based on the dynamic analysis feedback with the generated inputs. Note that our technique does not tie to GA; other search-based optimization algorithms can be adopted into our technique as well.

We have to address two main challenges in Peppa-X:

- **(C1):** In order to guide the exploration towards vulnerable program regions, we need to identify the distribution of SDC sensitivity in the program. A strawman method is to conduct extensive FI simulations for every program instruction. However, this method is extremely time-consuming. For example, in CoMD, it takes about 5030 hours to complete the FI evaluation on the default reference input in order to derive the SDC sensitivity distribution, as we observed in Section 3.
- **(C2):** To make optimization decisions during the dynamic search, we have to know the program SDC probability of each generated input relative to that of other inputs. Statistical FIs can be used here to measure the SDC probability with each input, and hence rank all the inputs. However, this method is inevitably costly as thousands of FI trials need to run for each input. Due to the vast input space, numerous input candidates will be generated during the search; using statistical FIs to evaluate candidates significantly slows down the search progress.

To address (C1), we propose a set of pruning techniques based on the static dataflow dependencies, thus reducing the required FI trials for identifying the program SDC sensitivity distribution. This step is shown in ❷ in Figure 3. Our pruning strategy leverages program static dataflow analysis to group the instructions along with the same static data dependency. The SDC probabilities of instructions in the same group share similar SDC sensitivity as errors directly propagate through immediate data dependency. In this way, Peppa-X performs only a handful of FI trials on the representative instructions in each group to derive the program's SDC sensitivity distribution.

To further save time in FI evaluation, we strategically select an input with a small workload in the FIs to speed up the evaluation. We call such input a *small FI input* of the program. We fuzz the target program starting from a limited workload range of inputs until reaching a specified code coverage, hence achieving the *small FI input* (❶). Recall that the SDC sensitivity distribution remains stable over the inputs. Thereby, we use the obtained small FI input in the FI simulations to derive SDC sensitivity distribution (❸). These developed heuristics significantly speed up the process of deriving SDC sensitivity distribution. The next step is to fuzz the program using GA for SDC-bound inputs (❹). GA is responsible for the input candidate generation and selection based on the SDC vulnerability potentials.

To address (C2), we have to find a way to avoid using statistical FI for estimating each generated candidate input during the GA fuzz in ❹. In Peppa-X, we design a novel dynamic program analysis technique that tracks the accumulated SDC vulnerability potentials on the explored execution paths during the program execution (❺). We compare the accumulated on-the-fly potential values on executing inputs generated by the GA recombination operation. The comparison winner directs GA to make optimization decisions. This process prevents us from using the slow offline statistical FI

[1] https://github.com/hasanur-rahman/Peppa-X

evaluation for the inputs generated in the GA recombination. It also incrementally steers the GA optimization search towards the SDC-bound input in the given time budget.

4.2 Design Details

In this section, we explain the core steps in Figure 3 in details.

4.2.1 Fuzzing for Small FI Input ❶. This step aims to find an input used in the FI evaluation to obtain the SDC sensitivity distribution. One can use the default reference input in each benchmark suite to compute the distribution. However, as mentioned, it is usually for performance testing, and can be significantly slow when used in FI runs. Thereby, we look for a new input with a small workload yet provides enough code coverage for evaluating the SDC sensitivity distribution. Instead of directly using the default reference input, we collect the code coverage from running it. Then, we use the coverage as the metric to find a new input from fuzzing since the default input often exercises representative parts of a program. In this way, the new input we find will cover the distribution of representative program regions.

We start the procedure from a small numerical range that only triggers a light computation workload in the target program. Then, we randomly generate an input within the range, and track its code coverage. If the new coverage meets the target coverage, we have found a small FI input. Otherwise, we have to repeat the process by increasing the numerical range until reaching the target coverage. On average, our fast fuzzing procedure only needs about 3 seconds to find a small FI input.

4.2.2 Static Analysis for Pruning FI Space ❷. In order to get the SDC sensitivity distribution of a program, one can use statistical FIs to evaluate the SDC probability of every static instruction in the program. However, blindly injecting random faults to every static instruction brings intolerable cost in practice. We observe that most instructions in its static dataflow (e.g., the instructions that have static data dependency) in a program share very similar SDC probabilities except for certain types of instructions. Based on this observation, we develop a heuristic that leverages static program analysis to group the program instructions based on the data dependencies. In each group, most instructions share similar SDC probabilities, and we only need to conduct FIs for the representative instructions and use them to approximate the SDC probabilities for the rest of the instructions in the same group.

```
BB167
...
%168 = load i32* %k...          ; ID1562; SDC: 0.8%
%169 = add nsw i32 %168, 1...   ; ID1563; SDC: 0.4%
...
%171 = icmp eq i32 %169...      ; ID1565; SDC: 100.0%
...
```

Figure 4: Code Example of Pruning FI Space in CoMD

The example in Figure 4 shows a basic block from the CoMD code. There are three data-dependent instructions in the basic block, namely *ID1562*, *ID1563*, and *ID1565*. We also measure the SDC probability of each instruction and annotate it in the figure. The three instructions are within the same static data dependency; they are in the same group based on our pruning method. We observe that the per-instruction SDC probabilities are very similar between the instructions *ID1562* and *ID1563* except for the *ID1565* instruction (CMP instruction). Therefore, we further divide the group into two subgroups based on the similarity of per-instruction SDC probabilities: one includes the former two, and the other subgroup has the *ID1565* instruction.

We find that a CMP instruction consistently differentiates the SDC probability with previous data-dependent instructions and separates a group of data-dependent instructions into subgroups. We observe some other types of instructions have a similar situation: they include all the logic operators (e.g., AND, OR, XOR, etc.), bit manipulation instructions (e.g., TRUNC, SEXT, etc.), and pointer operations. Therefore, in the example, we need to select either the instruction *ID1562* or *ID1563* and the instruction *ID1565* for FI evaluation, pruning the FI space from 3 instructions to 2.

Table 4 shows the pruning ratios we obtain after applying our pruning heuristic. The pruning ratio is calculated between the number of pruned instructions and the number of all instructions. As seen, the pruning ratio is application-specific, ranging from 25.49% in Pathfinder to 58.69% in Hpccg. The average pruning ratio is 49.32% over all the seven benchmarks.

Table 4: FI-Space Pruning Ratio for All Benchmarks

Pathfinder	Needle	Particlefilter	CoMD	Hpccg	Xsbench	FFT	Avg
25.49%	51.40%	46.35%	58.44%	58.69%	49.22%	55.64%	49.32%

4.2.3 Deriving SDC Scores with reduced FI Simulations ❸. After the pruning, we conduct FIs on the program with the small FI input to derive the program's SDC sensitivity distribution. We inject 30 random faults over the pruned number of static instructions and normalize the measured SDC probabilities. The normalized measurement helps assign an SDC score for each static instruction, indicating the SDC sensitivity relative to other instructions in the program. Note that one can choose to inject more faults for each instruction for a more accurate measurement as [1, 28, 32] do in their studies. However, this step aims to identify the relative ranking of per-instruction SDC probability of the program and estimate the SDC sensitivity distribution. Therefore, we choose to reduce the number of FI trials for the sake of performance.

Table 5: Time for the Analysis of SDC Sensitivity Distribution

Time (hrs)	Pathfinder	Needle	Particlefilter	CoMD	Hpccg	Xsbench	FFT	Avg.
With Heuristics	0.08	0.33	0.80	59.67	1.08	10.84	0.33	10.45
Without Heuristics	0.13	20.76	2.78	5029.76	775.11	58.71	1.14	841.20

We now show the efficiency and the effectiveness of our pruning heuristics. Table 5 displays the time taken to obtain the SDC sensitivity distribution with and without the proposed heuristics in ❶ and ❷. With our heuristics, the average analysis time decreases from 841.20 hours to 10.45 hours, exhibiting a speed-up of about 84 times.

4.2.4 Fuzzing with Genetic Engine ❹. After getting a program's SDC sensitivity distribution, we feed this information to the Genetic Engine, an input search engine driven by GA. Recap that the SDC sensitivity distribution is stable over different program inputs (Section 3); thereby, the genetic engine can search for the execution based on the obtained SDC sensitivity distribution. We design a fitness function inside the genetic engine that continuously reports the accumulated SDC vulnerability in each explored program execution. Next, following the heuristics proposed for GA in [24], we choose 0.4 and 0.05 for the mutation rate and crossover rate respectively. A program input consists of a set of input arguments. The mutation operation makes a slight value change to one of the arguments each time. The range of the value change is between -10% and +10% of the current argument value. In each mutation, we randomize a value within the range and add it to the original value. For the crossover operation, we randomly select two program inputs generated in the current GA generation, and swap one argument value between both inputs. Finally, we adopt the roulette selection algorithm [24] for the GA selection.

4.2.5 Dynamic Analysis for Guiding the Search ❺. For GA to make optimization decisions during the search, the fitness function assesses the GA-generated inputs. To avoid statistical FI for every candidate input, we track program execution with the input and accumulate the SDC vulnerability on the executed program path to estimate an alternative SDC vulnerability potential.

$$P_{overall} = N_{SDC}/N_{total} \tag{1}$$

$$= (\sum_{i=1}^{n} P_i * N_i)/N_{total} = \sum_{i=1}^{n} P_i * (N_i/N_{total}) \tag{2}$$

Equation 1 shows the overall SDC probability calculation in statistical FIs. $P_{overall}$ is the overall SDC probability of a program, N_{SDC} represents the number of trials observed as SDCs, and N_{total} is the total number of trials sampled in the FI experiment. Equation 2 is an equivalent form of Equation 1, where we introduce two terms P_i (per-instruction SDC probability) and N_i (the execution count of a static instruction during a dynamic program run). Since the relative ranking of P_i is stable over different inputs, we can use the SDC scores attached with static instructions (❸) to approximate the respective P_i in the equation. Note that both N_i and N_{total} rely on the specific input; we need to measure them at runtime. We track the executed times of each static instruction at runtime, then accumulate them to get N_i/N_{total} in each program execution with a candidate input. We use the accumulated value as the fitness score for the candidate input since the fitness score is an alternative to the program's SDC probability on that input. Finally, GA uses the fitness score to make optimization decisions — the candidate input with a higher fitness score will survive. It is worth mentioning that we only need to execute the program once on a candidate input in this dynamic analysis. In contrast, in statistical FIs, one need to execute the program once per FI trial while there are always thousands of FI trials to complete the SDC probability measurement for a candidate input [17, 32, 46]. Therefore, the dynamic fitness score calculation significantly speeds up the search process for SDC-bound input.

5 EVALUATION OF PEPPA-X

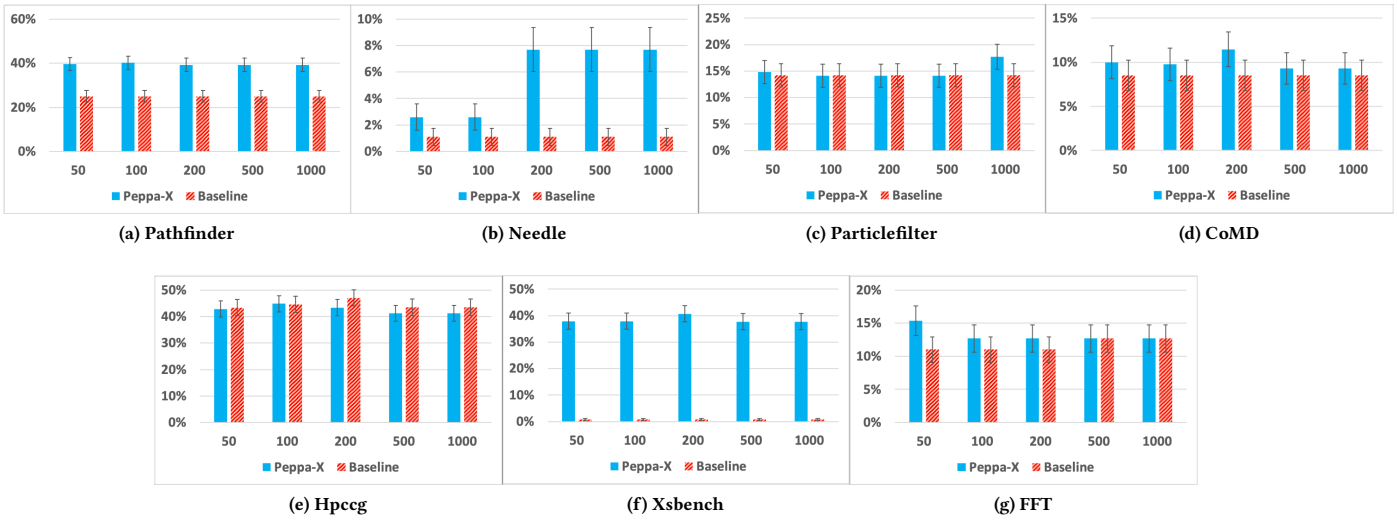
In this section, we first examine the bound of program SDC probability depicted by the SDC-bound inputs from Peppa-X. Then we evaluate the performance of running Peppa-X.

5.1 Bounding SDC Probability of Programs

We compare the program SDC probabilities bounded by the generated inputs from Peppa-X and the baseline technique, respectively, given the same search time. Our baseline method is the random input generation method with statistical FIs as it is the only currently available approach that searches for the SDC-bound input in a program. In both techniques, we inject 1,000 faults to the target program when it is required to be evaluated for program SDC probability. Our FI measurements are in line with many prior works in the area [15, 17, 22, 32, 46, 51**?**]. Note that we do not need to conduct any FI evaluation for the program SDC probability in Peppa-X until the end of the search once an SDC-bound input is reported (Section 4.2). Whereas in the baseline method, FI measurement is required for each generated input in the search to know whether or not the input can lead to a higher program SDC probability.

Figure 5 shows the results. We draw the highest SDC probabilities in each benchmark bounded by the inputs from both techniques, given time budgets of 50, 100, 200, 500, and 1,000 generations. As seen, Peppa-X can find an input that brings higher program SDC probability in most of cases. For example, at the time budget of 50 generations, Peppa-X can find an input that leads to a program SDC probability of 37.9% in Xsbench, whereas it is merely 0.7% in the baseline. Similar situations happen in Pathfinder and Needle, where Peppa-X identifies a program SDC probability of 39.2% and 7.7% respectively at 200 generations while the baseline can only find 25.0% and 1.1%.

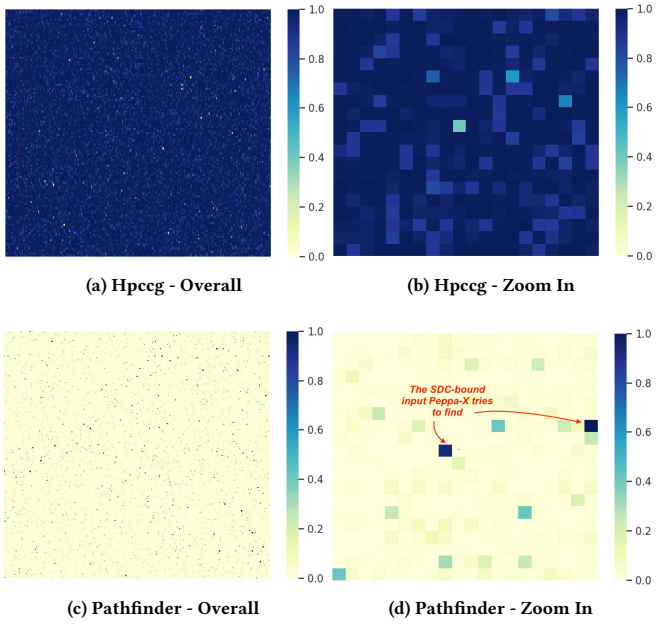
We observe that in some benchmarks, such as Hpccg, Particlefilter, and FFT, the baseline could perform as good as Peppa-X and finds inputs that lead to comparable SDC probabilities. We use Figure 6 to investigate the reason. The heat map draws the distribution of the program SDC probabilities in the input space of a program. For illustration purposes, we present the heat maps of Hpccg and Pathfinder. Each dot in the heat map represents an input, and the color of the dot denotes the program SDC probability of running the benchmark with the input. All the observed SDC probabilities are normalized to between 0 and 1, the darker color the higher SDC probability. The two sub-figures on the top row display the overall and the zoom-in situations of Hpccg, where we can find that most inputs are dark-colored. This distribution means a randomly sampled input may easily bring a high SDC probability. On average, a randomly sampled input in the space will lead to a program SDC probability that is above 96th percentile in Hpccg measurement. Thereby, the baseline approach, which randomly samples inputs, tends to perform as good as Peppa-X. We make similar observations in Particlefilter and FFT. In contrast, we draw two sub-figures at the bottom row in Figure 6 for Pathfinder benchmark in which Peppa-X performs much better. The two sub-figures significantly differ from those of Hpccg, where most dots have very light colors – on average, a randomly sampled input will lead to a program SDC probability that is only about 2nd percentile, making it extremely hard for the baseline to identify those SDC-bound inputs in Pathfinder.



(a) Pathfinder (b) Needle (c) Particlefilter (d) CoMD

(e) Hpccg (f) Xsbench (g) FFT

Figure 5: The Result of Bounding SDC Probability (Y-Axis: SDC Probability, X-Axis: No. of Generations in GA)



(a) Hpccg - Overall (b) Hpccg - Zoom In

(c) Pathfinder - Overall (d) Pathfinder - Zoom In

Figure 6: Heat Maps of SDC Vulnerability Distribution in the Input Space of Hpccg and Pathfinder (Normalized between 0 and 1.)

Note that the program SDC probability distribution in the input space depends on the application characteristics – it reflects how data-flow and control-flow (and hence error propagation) in the program execution are sensitive to input changes. Peppa-X performs much better when the inputs that lead to higher SDC probabilities are sparsely distributed (and hard to find) in the input space.

Further, we observe that in all the benchmarks, the SDC-bound inputs found by Peppa-X always outperform the default reference inputs provided in the benchmark suites (Figure 1) within any of the time budgets, showing that the default reference inputs often lead to over-optimistic SDC estimates and hence are not suitable for the evaluations of program SDC resiliency.

Next, we examine the program SDC probability the baseline method can reach if given 5x more time for the search. We compare the 5x-time result with the result of Peppa-X at 200 generations. The reason for using 200 generations as the cut-off time budget is that we observe that the program SDC probabilities mostly saturate after 200 generations. The average time taken for 200 generations in Peppa-X is nearly 40 hours, now we extend the baseline search by 5 times for each benchmark, to an average of 200-hour search for the comparison.

Figure 7 shows the result. As can be seen, the results are quite consistent with our prior evaluation in Figure 5. That is, if the baseline under-performs at 200 generations of Peppa-X, it still cannot find the SDC-bound inputs even given 5x more search time. In fact, the SDC probabilities bounded by the baseline in the benchmarks such as Xsbench are still far lower, showing Peppa-X is highly efficient in finding SDC-bound inputs.

Finally, we run Peppa-X for another 4,000 generations in the search and reach 5,000 generations for all the benchmarks. We observe that the search often converges within the first 200 generations except for Particlefilter (bumped up at 754 generations), indicating a saturation after 200 generations in most cases.

5.2 Performance

In this section, we show the performance evaluation in Peppa-X and the baseline technique. Then we present the breakdown time measurement in each step. To recap, Peppa-X consists of three parts that take time. They are (1) analysis of SDC sensitivity distribution, (2) dynamic fuzzing with GA, and (3) FI evaluation for SDC-bound

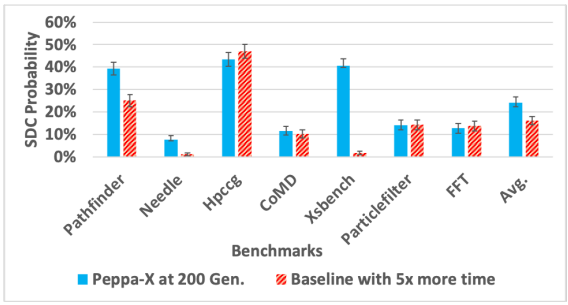


Figure 7: Program SDC Probability Bounded by Peppa-X at 200 Generations and Baseline with 5x More Search Time

input reported at the end of the search. The baseline contains FI evaluations for every randomly generated inputs. The FI evaluation for a given input in both Peppa-X and the baseline consists of instrumentation, profiling, and 1000-trial FIs using LLFI [51].

Note that our technique can be easily parallelized to improve the performance of the search, so does the baseline. However, for a fair comparison, we report the performance of both Peppa-X and the baseline measured without any parallelization.

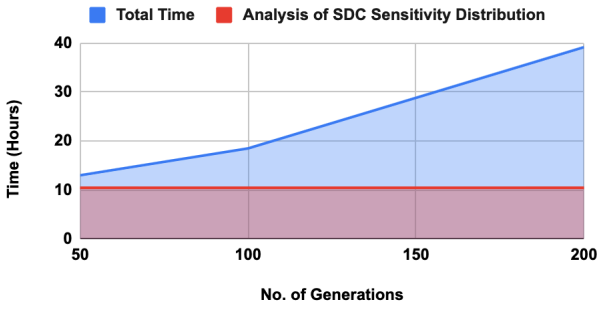


Figure 8: Total Time Taken by Peppa-X at Different Generations

Figure 8 shows the average time taken to complete Peppa-X at 50, 100, 200 generations over the seven benchmarks. As seen, the analysis of SDC sensitivity distribution is an one-time and fixed cost, and the total time of running Peppa-X is proportional to the number of generations in GA. Note that in Peppa-X, it only requires FI evaluation for the SDC-bound input reported at the end of the search. Thereby, it is a fixed cost too included in the total time.

Table 6: Performance Breakdown in Peppa-X and baseline

Time (s)	Peppa-X Per-Input Evaluation	Baseline Per-Input Evaluation
Pathfinder	1.06	9326.91
Needle	1.02	7497.40
Particlefilter	0.45	865.27
CoMD	3.99	110218.25
HPCCG	2.09	45325.39
Xsbench	18.63	222248.48
FFT	0.36	80.19
Average	3.94	56508.84

As a point of comparison, we show the time taken for assessing the SDC vulnerability of an input in Peppa-X and the baseline. To recap, Peppa-X leverages the dynamic analysis and fitness score to estimate the SDC vulnerability of an input relative to that of other inputs during the search (Section 4). In contrast, the baseline uses only FI evaluation to measure the program SDC probability with the input and hence to update the upper bound of the measurement. Table 6 shows the time taken for both Peppa-X and the baseline in the evaluation of an input. As can be seen, it takes only a small fraction of the time by Peppa-X compared with the baseline. On average, Peppa-X has a speedup of more than 4 orders of magnitude compared with the baseline in the evaluation of one input, allowing many more inputs to be assessed in Peppa-X given same amount of search time.

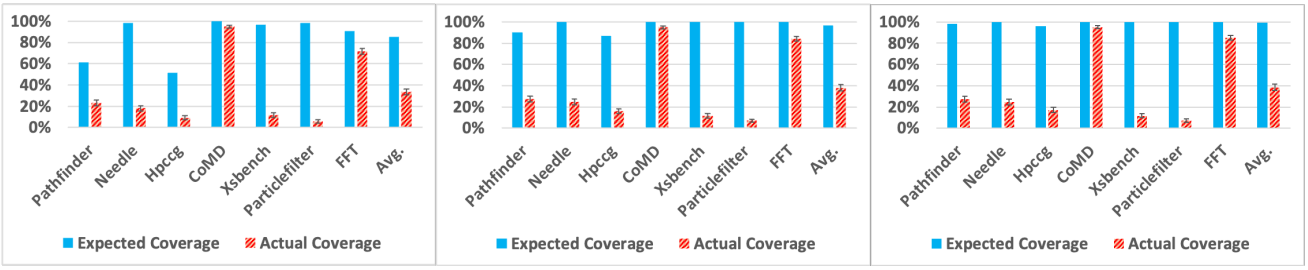
6 CASE STUDY: STRESS TESTING SELECTIVE INSTRUCTION DUPLICATION

In this section, we demonstrate the usefulness of our technique in stress testing a popular SDC protection technique – selective instruction duplication. The protection technique has been proposed and used in many other studies in the area of HPC resiliency, and is believed to be cost-effective in protecting programs from SDCs caused by transient hardware faults [1, 18, 22, 28, 29]. The selective instruction duplication relies on the assumption that only a small amount of instructions in a program are responsible for the majority of the program SDC probability. So develpers can gain a high SDC coverage by protecting those specific instructions with low overhead. In the selection, the technique formulates the SDC coverage and protection overhead as a classical 0-1 knapsack optimization problem [39].

For a given allowance of performance overhead incurred by the protection, the protection technique identifies the best set of instructions for the protection in order to maximize the SDC coverage. The protection is added through duplicating the selected instructions at compile-time. So once a transient hardware fault occurs at any of the protected instructions at runtime, the error will be detected by matching the computation results between the original and the duplicated copy of the instruction. In the optimization of a knapsack setting, the *cost* refers to the performance overhead of an instruction if duplicated, whereas the *benefit* is the SDC coverage as the result of duplicating the instruction.

In order to get the SDC coverage of every instruction, the SDC probability of the instruction needs to be measured. Therefore, in the selective instruction duplication technique, one needs to first evaluate the SDC probability of every instruction before making the optimization decisions in the knapsack problem. In this step, FIs are used, and hence an input needs to be provided for the FIs. In the past, researchers have been using the default reference input of a program when deploying selective instruction duplication and deriving an expected SDC coverage from the protection [1, 18, 22, 28]. However, we observe that the protection is often compromised if we use the SDC-bound inputs found by Peppa-X to stress test the protected programs, and hence the actual SDC coverages are often significantly lower than the expected ones.

Our stress-test experiment is conducted as follows: We adopt the selective instruction duplication technique proposed in [1, 18, 28], evaluate the per-instruction SDC probabilities using the default reference input in each benchmark (as all the existing works did),



(a) 30% Protection Level (b) 50% Protection Level (c) 70% Protection Level

Figure 9: Using Peppa-X to Stress Test Selective Protection Techniques (Y-Axis: SDC Coverage; X-Axis: Benchmarks)

then run the knapsack optimization algorithm to select a subset of instructions for the protections at 30%, 50%, and 70% levels of performance overhead respectively. We then measure the SDC coverages provided at each protection level via FIs – this is the expected SDC coverages by the protection at runtime and hence developers make their reliability decisions based on the information [1, 18, 22, 28, 29]. Finally, we execute the protected binary of the program with the SDC-bound input found by Peppa-X and examine the protection with FIs. We then compare the observed SDC coverages with the expected ones.

Figure 9 shows the stress-test results. On average, the expected SDC coverages are 85.23%, 96.63%, and 99.18% at 30%, 50% and 70% protection levels respectively. In contrast, they are only 33.52%, 38.02%, and 38.28% when stress-tested with the SDC-bound inputs. As seen, the SDC coverages tested with SDC-bound inputs are dramatically lower than the expected ones at all protection levels. For example, at 70% protection level, on average the actual coverage is measured 2.59 times lower that the expected coverage.

The exceptions are CoMD and FFT, where the SDC coverages tested with SDC-bound input are only slightly lower than the expected ones at all protection levels, but still, the expected coverages cannot be reached when stress-tested with their SDC-bound inputs. The most likely reason behind this is that when deciding SDC coverages, besides a stationary P_i in Eq 2, the dynamic footprints of instructions, N_i/N_{total} in Eq 2 in CoMD and FFT, may also have relatively smaller variances across inputs. Thereby, the Knapsack algorithm selects the subsets of instructions that are more globally optimal, protecting the instructions even if there are more SDC vulnerabilities revealed by the SDC-bound inputs in the program executions. We refer the improvement of selective instruction duplication technique to our future work.

Overall, our results in the case study show that developers likely underestimate the failure rates of their applications when using current selective instruction duplication techniques for protections. In resilience, being conservative is important in order to ensure the systems do not fail unexpectedly. Therefore, before deployment, SDC protection techniques shall be stress-tested for a conservative estimate, for example, with program SDC-bound inputs which can be efficiently found by Peppa-X.

7 DISCUSSION

We first discuss two other potential use cases of Peppa-X, then present the threats to validity in this study.

7.1 Other Potential Use Cases

7.1.1 Data Generation in Modeling Error Propagation. There exists a large body of works that characterized and modeled error propagation in software programs. These studies rely on FI experiments to manifest SDCs in programs before investigating the individual propagation cases [3, 19, 21, 32, 36]. Recent studies use FIs to generate a large amount of SDC campaigns, then feed the corpus to machine learning models in order to model error propagation [20, 29, 41]. Typically, large-scale FI experiments are necessary for these studies. The sets of SDC-bound inputs from Peppa-X can complement existing studies to increase the FI efficiency since those inputs can reveal significantly more SDCs. For example, in Xsbench, the probability of generating a FI campaign of SDC by a statistical FI technique is around 32x lower than Peppa-X. That is, Peppa-X may need merely 1/32 time to generate the same amount of corpus compared with the statistical FI methods, allowing more efficient data collection in their studies.

7.1.2 Integration into Software Development Cycle. In HPC and cloud software development, developers need to assess their applications for SDC resiliency before release. The code of applications needs to be frequently updated due to various reasons such as bug fixing, code refactoring etc after the initial release. Unfortunately, the SDC probability of the program need to be re-evaluated every time the code is updated since it is program-dependent. To this end, a fast and efficient test case generation technique is needed in order to save time in the evaluation. In an agile software development cycle, one can quickly run Peppa-X for generating SDC-bound input for a conservative evaluation of SDC probability upon the code changes. Moreover, the widely-used continuous integration (CI) technology can seamlessly leverage Peppa-X as a backend compiler toolchain to run automatically upon code commits.

7.2 Threats To Validity

7.2.1 Benchmarks. As stated in Section 3, we consider benchmarks used in recent publications in the related studies. Unlike performance evaluation, there is no standard benchmark suite for reliability evaluation. Our results may be specific to the choice of

benchmarks, though we have not observed this to be the case. Other work in this domain makes similar decisions [17, 20, 32, 36, 38].

7.2.2 Fault Injection Methodology. We use LLFI, a fault injector that works at the LLVM IR level, to inject single-bit flips. While this method is accurate for estimating SDC probabilities of programs [46, 51], it remains an open question of how accurate it is for other failure types. That said, our focus in this paper is SDCs, and so this is an appropriate choice for us.

7.2.3 Input Format. As mentioned in Section 3 , our evaluation is based on the benchmarks that take numerical inputs for the ease of generating a large amount of diverse inputs. Other related works in the area of HPC resilience made the same choice when generating diverse inputs [12, 13, 16, 32]. We believe that there is no fundamental limitation in our technique in terms of working with other types of input formats. All the program inputs must be converted to its numerical counterparts before it can be computed in the program execution, and our technique tracks the control-flow and data-flow changes in the program execution when inputs are changed (Section 4). Therefore, we believe that our technique can be extended to work with the programs that take a different initial format of inputs, if the formats are known or properly documented in the benchmark suites.

8 RELATED WORK

SDC Evaluation and Application Resiliency. There have been many works investigating the SDC resiliency of applications. Feng et al. [15] proposed Shoestring, a model of error propagation that identifies SDC-prone instructions in programs. Ashraf et al. [3] designed a framework that analyzes error propagation in MPI applications. Hari et al. [23] developed a pruning technique that evaluates the SDC resiliency of programs. Li et al. [36] proposed a FI sampling technique that obtains an approximate fault-tolerance threshold value for FI sites in HPC programs. While these works advanced the knowledge of application-level SDC resiliency, they focused on one or very few inputs in target programs. As we show in this work, there is often a significant variance in programs' SDC resiliency and error propagation characteristics with different inputs. Hence, programs need to be evaluated and studied across multiple inputs.

Other researchers have investigated program error detection and protection techniques. Laguna et al. [29] trained machine learning models to select and protect vulnerable instructions in programs. Kalra et al. [28] developed a compiler-based technique that selectively protects GPU programs in order to reduce SDCs. Anwer et al. [1] extended the Trident technique to GPU programs and protected error-prone instructions. Li et al. [34] built techniques to detect SDCs in HPC applications equipped with lossy compression. While these techniques can mitigate SDCs in programs, their techniques may not be optimized for programs' multiple inputs, especially for the inputs, such as SDC-bound inputs, that expose higher program SDC probabilities (as we show in our stress tests).

Multiple Inputs and Resiliency Characterization. In recent years, there have been studies investigating how program inputs may affect error propagation and resiliency evaluations. Di Leo et al. [13] characterized the relationship between program workloads and the failure distribution model. Folkesson et al. [16] analyzed different workloads and program failure rates. Li et al. [32] modeled input-dependent error propagation in programs. Yang et al. [55] proposed SUGAR to speed up the GPU evaluation process via input sizing. Mahmoud et al. [38] adopted software testing methods to evaluate program's SDC resiliency by prioritizing test cases based on PC coverage. While those works have presented insightful observations and characterizations in the program resiliency of multiple inputs, they rarely provide means to identify program test cases, such as SDC-bound inputs, to estimate the upper bound of program SDC probability. Our work is the first one that proposes an efficient search technique to find program test inputs that can be in conservative SDC evaluations.

Software Fuzzing. Grey-box fuzz testing, or fuzzing for short, has proved to be highly effective in software testing and vulnerability exploitation [6–8, 26, 35, 45]. Recent advances from industry [25, 27] also boost the wide applications of fuzzing to various domains like performance evaluation [50, 52], game deep state exploration [2], side-channel detection [43], voice assistant semantic fuzzing [58], database management system testing [60], etc. These methods, however, heavily rely on the coverage-guided search, which inherently lacks practicability in finding SDC vulnerability. Also, many existing fuzzers primarily focus on the abnormal symptoms in program execution, and none could track the silent but dangerous SDC vulnerability. Instead, Peppa-X develops an innovative fitness feedback-based search against the accumulated SDC potentials. Furthermore, the general design of Peppa-X can port to scenarios where black-box behavioral fuzzing may play an important role.

9 CONCLUSION

To conclude, we propose Peppa-X, which efficiently identifies the test inputs that estimate the bound of program SDC resiliency. The key insight of Peppa-X is that the SDC sensitivity distribution in a program often remains stationary across input space. Thereby, we can guide the search of SDC-bound inputs by the sampled distribution. Our evaluation shows that Peppa-X can identify the SDC-bound input of a program that existing methods cannot find even with 5x more search time.

REFERENCES

[1] Abdul Rehman Anwer, Guanpeng Li, Karthik Pattabiraman, Michael Sullivan, Timothy Tsai, and Siva Kumar Sastry Hari. Gpu-trident: efficient modeling of error propagation in gpu programs. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2020.

[2] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. Ijon: Exploring deep state spaces via fuzzing. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1597–1612. IEEE, 2020.

[3] Rizwan A Ashraf, Roberto Gioiosa, Gokcen Kestor, Ronald F DeMara, Chen-Yong Cher, and Pradip Bose. Understanding the propagation of transient errors in hpc applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 72. ACM, 2015.

[4] Chao Chen, Greg Eisenhauer, and Santosh Pande. Near-zero downtime recovery from transient-error-induced crashes. *arXiv preprint arXiv:2103.05185*, 2021.

[5] Chao Chen, Greg Eisenhauer, Matthew Wolf, and Santosh Pande. Ladr: Low-cost application-level detector for reducing silent output corruptions. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, pages 156–167, 2018.

[6] Hongxu Chen, Shengjian Guo, Yinxing Xue, Yulei Sui, Cen Zhang, Yuekang Li, Haijun Wang, and Yang Liu. MUZZ: thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 2325–2342. USENIX Association, 2020.

[7] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: Towards a desired directed grey-box fuzzer. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 2095–2108. ACM, 2018.

[8] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. SAVIOR: towards bug-driven hybrid testing. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1580–1596. IEEE, 2020.

[9] Hyungmin Cho, Shahrzad Mirkhani, Chen-Yong Cher, Jacob A Abraham, and Subhasish Mitra. Quantitative evaluation of soft error injection techniques for robust system design. In *Proceedings of the 50th Annual Design Automation Conference*, page 101. ACM, 2013.

[10] Cristian Constantinescu. Intermittent faults and effects on reliability of integrated circuits. In *Reliability and Maintainability Symposium*, page 370. IEEE, 2008.

[11] Jeffrey J Cook and Craig Zilles. A characterization of instruction-level error derating and its implications for error detection. In *International Conference on Dependable Systems and Networks(DSN)*, pages 482–491. IEEE, 2008.

[12] Edward W. Czeck and Daniel P. Siewiorek. Observations on the effects of fault manifestation as a function of workload. *IEEE Transactions on Computers*, 41(5):559–566, 1992.

[13] Domenico Di Leo, Fatemeh Ayatolahi, Behrooz Sangchoolie, Johan Karlsson, and Roger Johansson. On the impact of hardware faults–an investigation of the relationship between workload inputs and failure mode distributions. In *International Conference on Computer Safety, Reliability, and Security*, pages 198–209. Springer, 2012.

[14] Harish Dattatraya Dixit, Sneha Pendharkar, Matt Beadon, Chris Mason, Tejasvi Chakravarthy, Bharath Muthiah, and Sriram Sankar. Silent data corruptions at scale. *arXiv preprint arXiv:2102.11245*, 2021.

[15] Shuguang Feng, Shantanu Gupta, Amin Ansari, and Scott Mahlke. Shoestring: probabilistic soft error reliability on the cheap. In *ACM SIGARCH Computer Architecture News*, volume 38, page 385. ACM, 2010.

[16] Peter Folkesson and Johan Karlsson. *The effects of workload input domain on fault injection results.* Citeseer, 1999.

[17] Giorgis Georgakoudis, Ignacio Laguna, Dimitrios S Nikolopoulos, and Martin Schulz. Refine: Realistic fault injection via compiler-based instrumentation for accuracy, portability and speed. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2017.

[18] Guanpeng Li, Karthik Pattabiraman, Siva Kumar Sastry Hari, Michael Sullivan and Timothy Tsai. Modeling soft-error propagation in programs. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2018.

[19] Luanzheng Guo and Dong Li. Moard: Modeling application resilience to transient faults on data objects. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 878–889. IEEE, 2019.

[20] Luanzheng Guo, Dong Li, and Ignacio Laguna. Paris: Predicting application resilience using machine learning. *Journal of Parallel and Distributed Computing*, 2021.

[21] Luanzheng Guo, Dong Li, Ignacio Laguna, and Martin Schulz. Fliptracker: Understanding natural error resilience in hpc applications. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 94–107. IEEE, 2018.

[22] Siva Kumar Sastry Hari, Sarita V Adve, and Helia Naeimi. Low-cost program-level detectors for reducing silent data corruptions. In *International Conference on Dependable Systems and Networks (DSN)*, pages 1–12. IEEE, 2012.

[23] Siva Kumar Sastry Hari, Sarita V Adve, Helia Naeimi, and Pradeep Ramachandran. Relyzer: exploiting application-level fault equivalence to analyze application resiliency to transient faults. In *ACM SIGARCH Computer Architecture News*, volume 40, page 123. ACM, 2012.

[24] Randy L Haupt. Optimum population size and mutation rate for a simple real genetic algorithm that optimizes array factors. In *IEEE Antennas and Propagation Society International Symposium. Transmitting Waves of Progress to the Next Millennium. 2000 Digest. Held in conjunction with: USNC/URSI National Radio Science Meeting (C*, volume 2, pages 1034–1037. IEEE, 2000.

[25] https://github.com/google/AFL. american fuzzy lop.

[26] https://google.github.io/clusterfuzz/. clusterfuzz.

[27] https://llvm.org/docs/LibFuzzer.html. libfuzzer.

[28] Charu Kalra, Fritz Previlon, Norm Rubin, and David Kaeli. Armorall: Compiler-based resilience targeting gpu applications. *ACM Transactions on Architecture and Code Optimization (TACO)*, 17(2):1–24, 2020.

[29] Ignacio Laguna, Martin Schulz, David F Richards, Jon Calhoun, and Luke Olson. Ipas: Intelligent protection against silent output corruption in scientific applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, pages 227–238. ACM, 2016.

[30] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization*, page 75. IEEE, 2004.

[31] Dong Li, Zizhong Chen, Panruo Wu, and Jeffrey S Vetter. Rethinking algorithm-based fault tolerance with a cooperative software-hardware approach. In *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2013.

[32] Guanpeng Li and Karthik Pattabiraman. Modeling input-dependent error propagation in programs. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 279–290. IEEE, 2018.

[33] Guanpeng Li, Karthik Pattabiraman, Chen-Yang Cher, and Pradip Bose. Understanding error propagation in GPGPU applications. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 240–251. IEEE, 2016.

[34] Sihuan Li, Sheng Di, Kai Zhao, Xin Liang, Zizhong Chen, and Franck Cappello. Towards end-to-end sdc detection for hpc applications equipped with lossy compression. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 326–336. IEEE, 2020.

[35] Yuekang Li, Yinxing Xue, Hongxu Chen, Xiuheng Wu, Cen Zhang, Xiaofei Xie, Haijun Wang, and Yang Liu. Cerebro: context-aware adaptive fuzzing for effective vulnerability detection. In Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo, editors, *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, pages 533–544. ACM, 2019.

[36] Zhimin Li, Harshitha Menon, Kathryn Mohror, Peer-Timo Bremer, Yarden Livant, and Valerio Pascucci. Understanding a program's resiliency through error propagation. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 362–373, 2021.

[37] Robert Lucas, James Ang, Keren Bergman, Shekhar Borkar, William Carlson, Laura Carrington, George Chiu, Robert Colwell, William Dally, Jack Dongarra, et al. Doe advanced scientific computing advisory subcommittee (ascac) report: top ten exascale research challenges. Technical report, USDOE Office of Science (SC)(United States), 2014.

[38] Abdulrahman Mahmoud, Radha Venkatagiri, Khalique Ahmed, Sasa Misailovic, Darko Marinov, Christopher W Fletcher, and Sarita V Adve. Minotaur: Adapting software testing techniques for hardware errors. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1087–1103, 2019.

[39] George B Mathews. On the partition of numbers. *Proceedings of the London Mathematical Society*, 1(1):486–490, 1896.

[40] Harshitha Menon and Kathryn Mohror. Discvar: Discovering critical variables using algorithmic differentiation for transient faults. *ACM SIGPLAN Notices*, 53(1):195–206, 2018.

[41] Bin Nie, Ji Xue, Saurabh Gupta, Tirthak Patel, Christian Engelmann, Evgenia Smirni, and Devesh Tiwari. Machine learning models for gpu error prediction in a large scale hpc system. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 95–106. IEEE, 2018.

[42] Bin Nie, Lishan Yang, Adwait Jog, and Evgenia Smirni. Fault site pruning for practical reliability analysis of gpgpu applications. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 749–761. IEEE, 2018.

[43] Shirin Nilizadeh, Yannic Noller, and Corina S. Pasareanu. Diffuzz: differential fuzzing for side-channel analysis. In Joanne M. Atlee, Tevfik Bultan, and Jon Whittle, editors, *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 176–187. IEEE / ACM, 2019.

[44] Nahmsuk Oh, Philip P Shirvani, and Edward J McCluskey. Control-flow checking by software signatures. *IEEE Transactions on Reliability*, 51(1):111–122, 2002.

[45] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. Specfuzz: Bringing spectre-type vulnerabilities to the surface. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 1481–1498. USENIX Association, 2020.

[46] Lucas Palazzi, Guanpeng Li, Bo Fang, and Karthik Pattabiraman. A tale of two injectors: End-to-end comparison of ir-level and assembly-level fault injection. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pages 151–162. IEEE, 2019.

[47] Behrooz Sangchoolie, Karthik Pattabiraman, and Johan Karlsson. One bit is (not) enough: An empirical study of the impact of single and multiple bit-flip errors. In *International Conference on Dependable Systems and Networks (DSN)*, pages 97–108. IEEE, 2017.

[48] Evgenia Smirni. Practical reliability analysis of gpgpus in the wild: From systems to applications. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, pages 3–3, 2019.

[49] Marc Snir, Robert W Wisniewski, Jacob A Abraham, Sarita V Adve, Saurabh Bagchi, Pavan Balaji, Jim Belak, Pradip Bose, Franck Cappello, Bill Carlson, Andrew A Chien, Paul Coteus, Nathan A DeBardeleben, Pedro C Diniz, Christian Engelmann, Mattan Erez, Saverio Fazzari, Al Geist, Rinku Gupta, Fred Johnson, Sriram Krishnamoorthy, Sven Leyffer, Dean Liberty, Subhasish Mitra, Todd Munson, Rob Schreiber, Jon Stearley, and Eric Van Hensbergen. Addressing failures in Exascale computing. *The International Journal of High Performance Computing*

Applications, 28(2):129–173, 2014.

[50] Jiayi Wei, Jia Chen, Yu Feng, Kostas Ferles, and Isil Dillig. Singularity: pattern fuzzing for worst case complexity. In Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu, editors, *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pages 213–223. ACM, 2018.

[51] Jiesheng Wei, Anna Thomas, Guanpeng Li, and Karthik Pattabiraman. Quantifying the accuracy of high-level fault injection techniques for hardware faults. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 375–382. IEEE, 2014.

[52] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. Memlock: memory usage guided fuzzing. In Gregg Rothermel and Doo-Hwan Bae, editors, *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, pages 765–777. ACM, 2020.

[53] Lishan Yang, Bin Nie, Adwait Jog, and Evgenia Smirni. Practical resilience analysis of gpgpu applications in the presence of single-and multi-bit faults. *IEEE Transactions on Computers*, 70(1):30–44, 2020.

[54] Lishan Yang, Bin Nie, Adwait Jog, and Evgenia Smirni. Enabling software resilience in gpgpu applications via partial thread protection. *arXiv preprint arXiv:2103.02825*, 2021.

[55] Lishan Yang, Bin Nie, Adwait Jog, and Evgenia Smirni. Sugar: Speeding up gpgpu application resilience estimation with input sizing. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 5(1):1–29, 2021.

[56] Keun Soo Yim, Cuong Pham, Mushfiq Saleheen, Zbigniew Kalbarczyk, and Ravishankar Iyer. Hauberk: Lightweight silent data corruption error detector for GPGPU. In *International Parallel & Distributed Processing Symposium (IPDPS)*, page 287. IEEE, 2011.

[57] Li Yu, Dong Li, Sparsh Mittal, and Jeffrey S Vetter. Quantitatively modeling application resilience with the data vulnerability factor. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 695–706. IEEE, 2014.

[58] Yangyong Zhang, Lei Xu, Abner Mendoza, Guangliang Yang, Phakpoom Chinprutthiwong, and Guofei Gu. Life after speech recognition: Fuzzing semantic misinterpretation for voice assistant applications. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

[59] Kai Zhao, Sheng Di, Sihuan Li, Xin Liang, Yujia Zhai, Jieyang Chen, Kaiming Ouyang, Franck Cappello, and Zizhong Chen. Ft-cnn: Algorithm-based fault tolerance for convolutional neural networks. *IEEE Transactions on Parallel & Distributed Systems*, 32(07):1677–1689, 2021.

[60] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. SQUIRREL: testing database management systems with language validity and coverage feedback. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, pages 955–970. ACM, 2020.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

We compiled source code of seven applications using llvm/clang v3.4 compiler to IR files, and ran our proposed technique with the compiled IR files as described in the paper. We also conducted large scale fault injection experiments with the IR files using LLFI for the initial fault injection study and the baseline results. Finally, in our case study, we ran Knapsack algorithm and used LLVM pass implemented to duplicate instructions at application level based on the proposed selective instruction duplication technique.

Author-Created or Modified Artifacts:

```
Persistent ID: 10.1145/3476480
Artifact name: Peppa-X workflow
```

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: i) CPU: Intel(R) Core(TM) i9-10900 CPU @ 2.80GHz, x86_64 architecture, 10 cores/20 threads , ii) Memory: 64GB

Operating systems and versions: Ubuntu 16.04.7 LTS running linux kernel 4.4.0-203-generic

Compilers and versions: llvm and clang v3.4

Applications and versions: Benchmarks: Pathfinder, Needle, Particlefilter, CoMD, Hpccg, Xsbench, FFT

Libraries and versions: LLVM Fault Injector (LLFI) v3.4

Key algorithms: Knapsack and Genetic algorithm

Input datasets and versions: i) llvm/clang v3.4 compiled files from the source code of seven applications from four standard benchmarks suites Rodinia, Mantevo , Cesar, SPLASH-2. ii) SDC bounding results and performance evaluation of our proposed technique and baseline for each of the seven applications.