Mitigating Silent Data Corruptions in HPC Applications across Multiple Program Inputs

Yafan Huang*, Shengjian Guo†, Sheng Di‡, Guanpeng Li*, Franck Cappello‡
* Computer Science Department, University of Iowa, Iowa City, IA, USA
† Baidu Security, Sunnyvale, CA, USA
‡ Mathematics and Computer Science Division, Argonne National Laboratory, Lemont, IL, USA
yafan-huang@uiowa.edu, sjguo@baidu.com, sdi@anl.gov, guanpeng-li@uiowa.edu, cappello@mcs.anl.gov

***Abstract*—With the ever-shrinking size of transistors, silent data corruptions (SDCs) are becoming a common yet serious issue in HPC. Selective instruction duplication (SID) is a widely used fault-tolerance technique that can obtain high SDC coverage with low performance overhead. However, existing SID methods are confined to single program input in its assessment, assuming that error resilience of a program remains similar across inputs. Nevertheless, we observe that the assumption cannot always hold, leading to a drastic loss in SDC coverage across different inputs, compromising HPC reliability. We notice that the SDC coverage loss correlates with a small set of instructions - we call them incubative instructions, which reveal elusive error propagation characteristics across multiple inputs. We propose MINPSID, an automated SID framework that automatically identifies and re-prioritizes incubative instructions in a given program to enhance SDC coverage. Evaluation shows MINPSID can effectively mitigate the loss of SDC coverage across multiple inputs.**

***Keywords*—Silent Data Corruption, Error Resilience, Fault Injection, Instruction Duplication, Program Analysis, Software Testing, High Performance Computing**

I. INTRODUCTION

The shrinking transistor sizes have increased the susceptibility of modern high-performance computing (HPC) systems to hardware transient faults [1]. Though hardware protections like voltage guard bands and triple modular redundancy (TMR) can mask these faults, such hardware methods inevitably incur tremendous energy consumption overhead, whereas energy has become first-class constraint in modern system design [2]. The situation is particularly exacerbated in HPC due to the large computation scale and was classified as one of the top challenges in HPC [3]. As a result, researchers expect that future HPC applications should tolerate hardware faults with low performance and energy overheads.

Silent data corruption (SDC) is recognized as one of the most severe types of failures [4]. It silently propagates in program execution and finally corrupts program output without noticeable symptoms. Researchers have observed that a portion of program instructions are responsible for almost all the SDCs in a program [5]. Thus, protecting the most vulnerable parts of the program may be sufficient to achieve high SDC coverage with relatively low overhead. In light of this, *selective instruction duplication* (SID) has been proposed to mitigate SDCs in programs [2], [6], [7]. In general, SID chooses the most vulnerable program instructions and duplicates them in program execution against SDCs.

However, existing SID studies often confine themselves to one program input when assessing the effectiveness of the protection, assuming that the error resiliency of the program remains similar across different program inputs. Nevertheless, we observe that such an assumption does not always hold. The SDC coverage suffers from significant loss[1] when a protected program runs with different inputs, which may seriously compromise HPC reliability. Industry experiences with similar views have recently been released by Meta [4] and Google [8], disclosing much higher SDC rates than expected in their large-scale applications, even if the applications are protected.

To investigate the observed problem in-depth, we make four key contributions in this paper: (1) We conduct an extensive fault injection (FI) experiment to demonstrate that popular SID methods may suffer from a significant loss of SDC coverage when a protected program runs with different inputs. (2) Through a root cause analysis, we identify that there is a small set of instructions that reveal drastically different error propagation behaviors among different inputs. Such instructions result in the SDC coverage loss, and we name those instructions *incubative instructions* – the instructions that show resilient on some inputs but others. (3) Based on our analysis, we propose MINPSID (**M**ulti-**Inp**ut-Hardened **S**elective **I**nstruction **D**uplication), an automated SID framework that identifies *incubative instructions* of a given program through static analysis and dynamic input searching. MINPSID also re-prioritizes the *incubative instructions* according to their conservative error propagation behaviors, and improves the overall SDC coverage for SID across multiple program inputs. (4) We evaluate MINPSID versus existing SID method on a diverse set of HPC applications. Our experimental results are summarized as follows:

- We identify that the SDC coverage decreases from an average of 96.12% to 58.76% over 11 benchmarks when running existing SID method with multiple program inputs. In an extreme case, the loss of SDC coverage can be as significant as 100% in the Kmeans benchmark.

[1] We define SDC coverage loss as the reduction of SDC coverage provided by the protection technique.

This paper is accepted by SC'22.

On average, 37.58% of the inputs lead to SDC coverage loss in each benchmark, with a maximum of 72.00% in the FFT benchmark.
- We find that the number of *incubative instructions* varies in programs, from 6.20% (LU) to 32.09% (Needle) of the total instructions. On average, there are 15.79% *incubative instructions* in a program. They are responsible for at least 97% loss of SDC coverage.
- We propose MINPSID which combines program analysis and input searching techniques for identifying *incubative instructions* in a program. Our proposed technique can find 45.60% more *incubative instructions* than a random search within the given time budget. MINPSID also re-prioritizes *incubative instructions* in the instruction selection phase of SID to improve the SDC coverage. On average, MINPSID takes a one-time cost of 63.71 minutes to complete its execution.

The remaining of the paper is organized as follows. In Section II, we introduce the background. In Section III, we describe our experimental setup in details. Then, we demonstrate the serious drawback of the existing SID methods in Section III-B, and analyze the root-cause in Section IV. In Section V, we describe MINPSID in details. In Section VI, we present and analyze the evaluation results.

II. BACKGROUND KNOWLEDGE

In this section, we first present the fault model, followed by an introduction of the LLVM compiler. In the end, we review the SID technique in detail.

A. Fault Model

In this work we use a fault model that has been commonly studied in related fault tolerance works [5], [7], [9], [10]. Specifically, we focus on transient hardware faults in processor 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 the instruction encoding faults as they can be detected through other techniques like error-correcting codes. Finally, we do not take into account the situation that the program jumps to arbitrary, illegal addresses due to execution faults, as this problem can be resolved by control-flow checking techniques [11]. However, the program may take a legal but wrong branch, which means the executed path is legal, but the branch selection is not as expected due to propagated faults.

We focus on the faults that affect program executions. Such faults may cause a program to crash, hang, or produce an incorrect program output. In the case of an incorrect output, we say that the fault leads to an silent data corruption (SDC) which is our focus in this work. We define SDC probability as the probability of an SDC upon a manifested fault in program execution. We also define SDC coverage as the percentage of SDCs that has been mitigated by a used protection technique. These definitions have been commonly used in [5], [7], [12], [13].

B. LLVM Compiler

We perform the program analysis, fault injection, and the tool implementation based on the LLVM compiler [14] for the following reasons: First, LLVM uses a typed intermediate representation (IR) that can easily represent source-level semantics. 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 to locate the error sites in the source code that cause specific failures. Second, LLVM IR is a platform-neutral representation that abstracts away low-level hardware details. This feature eases porting our analysis to various architectures. It also simplifies the treatment of the assembly language formats. Finally, LLVM IR has been practical for doing FI studies [15], and there exists a set of fault injectors upon LLVM [6], [16]. With the unique advantages, the LLVM compiler has been utilized in the resilience research [2], [5], [7], [12], [16], [17]. For brevity, we refer the term *instruction* in this paper to the LLVM IR level instruction, although our methodology is general and not tied to LLVM.

C. Selective Instruction Duplication (SID)

SID has been extensively used in protecting programs from soft errors [2], [6], [12]. Figure 1 shows an example of SID. As shown in (c), if instruction D is vulnerable and needs to be protected, one can duplicate its execution by inserting an instruction D_{dup}, and compare the computation results between D and D_{dup}. Such comparisons in SID need to be placed before any program synchronization locations such as function calls and control-flow branches etc. If any faults occur at D or D_{dup}, the duplication will detect the mismatch at runtime, and hence detect SDC. SID incurs much less runtime overhead compared with a full duplication such as in (b).

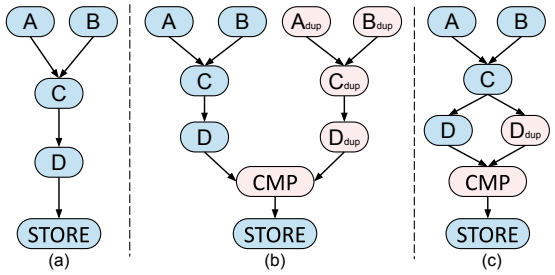


Fig. 1: An example of SID from Pathfinder benchmark. Each block represents an instruction. (a) Original program (b) Full duplication (c) Selective protection of instruction D

SID formulates the SDC coverage and protection overhead as a classic 0-1 knapsack optimization problem [18]. Within the given allowance of the performance overhead, SID identifies the set of instructions that maximize the SDC coverage. The protection is performed by duplicating the selected instructions in program compilation. Since we consider transient

hardware faults which will affect only one instruction at a time, the immediate repetition of the instruction will be fault-free. Thereby, once a transient hardware fault occurs at any protected instruction at runtime, the error will be detected by comparing the computation results between the original instruction and its duplicated copy. In the optimization of a knapsack setting, the *cost* refers to the performance overhead of the instruction duplication. In contrast, the *benefit* is the increased SDC coverage from the duplicated instructions.

$$\text{Cost}_i = (\text{Dynamic Cycle})_i \ / \ (\text{Total Cycle}) \quad (1)$$

$$\text{Benefit}_i = (\text{SDC Probability})_i \times \text{Cost}_i \quad (2)$$

Equation (1) and (2) give the details of the cost and benefit calculations for an instruction i. We compute the cost by profiling the fraction of dynamic cycles of an instruction against the total execution cycles of the program. The benefit is assessed by measuring the SDC probability of the instruction and its cost. The detailed formulation can be found in [2], [7]. The optimization step that chooses instructions for protection based on the cost and benefit profiles is known as *instruction selection* in SID.

In the *instruction selection* phase, the FI method is usually utilized in measuring the SDC probability of an instruction under an input. Once the cost and benefit of every instruction are obtained, SID uses the Knapsack algorithm for instruction selection, given a targeted protection level. The term *protection level* indicates the amount of dynamic instructions need to be duplicated in SID, which is also a proxy to the performance overhead of the protection. The most critical instructions (per unit cost) will be selected for instruction duplication in this phase. Upon finishing the selection, SID provides an *expected SDC coverage* by aggregating all the measured SDC probabilities of the selected instructions. Developers will leverage the SDC coverage from SID to gauge whether or not their protected applications meet the reliability target before deployment.

III. Preliminary Study

In this section, we manifest the loss-of-SDC-coverage issue in existing SID techniques through FI experiments. We first present our experimental setup before discussing the results. We also use the same experimental settings later to evaluate our proposed solution in Section VI.

A. Experimental Setup

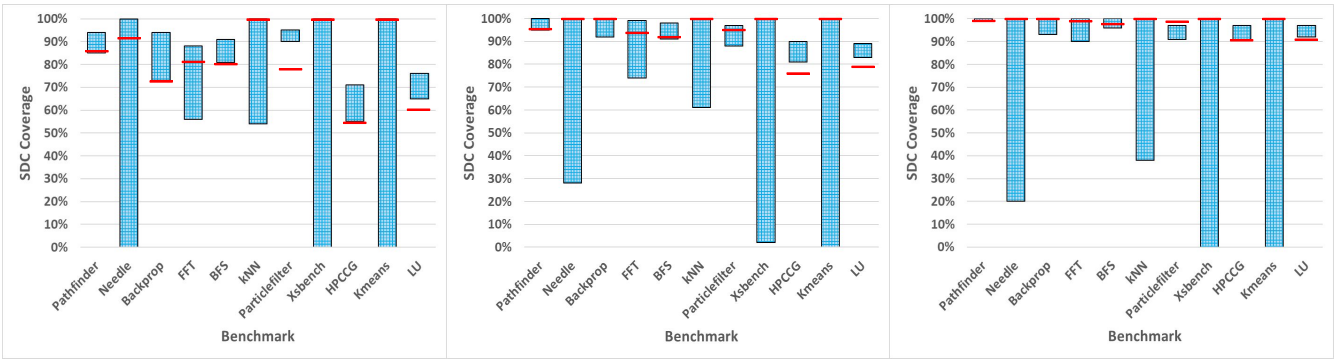
1) Our Benchmarks: We collect a total of 11 benchmarks in our evaluation. They are shown in Table II. All the benchmarks are CPU programs since they are what we focus in this paper. In our study, we choose benchmarks that are free of the following problems: (1) The benchmarks cannot be compiled with LLVM so we cannot perform subsequent program analysis and code transformation; (2) The benchmarks do not work with the fault injector (LLFI) that we use; (3) The input of the benchmarks is undocumented (e.g., unknown binaries), so we cannot easily generate new inputs for the benchmarks. Among these benchmarks, 8 are from the Rodinia benchmark suite [19] since they are commonly used in related studies [20]–[22]. Besides Rodinia, we include the three HPC kernel applications (HPCCG, FFT, and Xsbench) that are used in a recent related work in HPC resilience [22]. Note that the original CG code is incompatible with the instrumentation of LLFI, so we choose HPCCG [23], the HPC version of CG developed by the same group, as an alternative in our study. All the selected benchmarks have been commonly used in HPC resilience studies [9], [15], [20], [24]–[28]. The study [25], for example, used LU, FFT, and CG as their benchmarks in a recent HPC resilience work – we have included all of the three in our evaluation.

TABLE I: Our Benchmarks

Benchmark	Suite	Description
Xsbench	CESAR	Key computational kernel of the Monte Carlo neutronics application
HPCCG	Mantevo	A simple conjugate gradient benchmark code for a 3D chimney domain on an arbitrary number of processors
FFT	SPLASH-2	1D fast Fourier transform using six-step FFT method
kNN	Rodinia	Find the k-nearest neighbours from an unstructured data set
Pathfinder	Rodinia	Use dynamic programming to find a path in grid
Backprop	Rodinia	A machine-learning algorithm that trains the weights of connected nodes on a layered neural network
BFS	Rodinia	Breadth-first search all connected components in a graph
Particlefilter	Rodinia	Statistical estimator of the location of a target object given noisy measurements of that target's location in a Bayesian framework
Kmeans	Rodinia	A clustering algorithm used extensively in data-mining and elsewhere
LU	Rodinia	An algorithm calculating the solutions of a set of linear equations
Needle	Rodinia	A nonlinear global optimization method for DNA sequence alignments

2) Input Generation Method: To conduct our experiment without loss of generality, we generate random inputs based on two important rules. First, the input should not produce any reported errors or exceptions that halt the program execution. The error-introducing input may not represent the ordinary application behavior in production. Second, the number of executed instructions by an input should not exceed 40 billion, since we have to balance experiment time in our study. After the filtering, we keep 50 can random inputs for each benchmark to perform the FI experiments. The average number of executed instructions per input is around 1.49 billion.

Our method to generate random inputs is as follows: If the input of a benchmark contains numeric values, we randomize each with a valid random value. For non-numeric types (e.g., string etc), we randomize each parameter among its legit value domain (following its documentation). On the other hand, if the benchmarks come with scripts which are used



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

Fig. 2: The loss of SDC coverage in existing SID method (Red bars indicate expected SDC coverage provided by SID)

to randomize inputs in the suite, we will utilize the scripts to generate random inputs for these benchmarks. For example, for structured inputs such as graph etc, the BFS, kNN, and kmeans benchmarks provide such scripts in the benchmark suite. Overall, our input generation method is inline with prior work in the area [12], [22], [29].

3) FI Process: To perform FI experiments, we utilize the LLVM Fault Injector (LLFI) [16] which has been used in studying SDCs and SID in other recent works [2], [7], [30], [31]. As stated in our fault model, we only consider transient faults in the computing components. Hence, we use LLFI to inject single-bit flips into a random instruction's return value. We consider single-bit flips since it is a common fault model for simulating transient faults in the literature [5], [9], [20], [29], [32], [33]. Therefore, we adopt single-bit flips in our SDC-related evaluation.

To measure a program's overall SDC probability (for either protected or unprotected programs), we inject 1,000 random faults to each program for each input. Also, to derive the per-instruction SDC probabilities in each program, we inject 100 random faults to each static instruction on each input to maintain a reasonable experiment time. Our FI measurement yields an error bar from 0.26% to 3.10% for the 95% confidence intervals. The FI methodology is comparable with related works [5], [10], [12], [15], [33].

4) SID Technique: The used SID technique in our study is in line with state-of-the-art SID works [2], [7], [30]. Similar to these studies, we use LLVM to implement SID to protect the programs. Specifically, we first apply SID to each benchmark, setting the protection level with a performance overhead budget of 30%, 50%, and 70%, respectively. For each benchmark, the reference input from the benchmark suite is chosen to measure the benefit and cost of each instruction. Then, we perform the instruction selection for duplication. After setting up the protection, we inject faults to the protected programs with 30 random inputs and measure the SDC coverage at different protection levels. To conclude, we conduct the SID instruction selection based on reference input, whereas we use random inputs to execute the protected programs and evaluate the SDC coverage. Note that existing studies [2], [12], [20], [30], [34]–[36] use the same program input for both instruction selection and SID evaluation. In contrast, our setup tries to mock up a practical environment that an application may run with arbitrary inputs. Finally, we compare the measured SDC coverage across inputs against the expected SDC coverage provided by the SID method.

TABLE II: Percentage of Random Coverage-loss Inputs

Benchmark	30% Level	50% Level	70% Level
Pathfinder	4.00%	6.00%	2.00%
kNN	58.00%	78.00%	74.00%
BFS	2.00%	22.00%	26.00%
Backprop	0.00%	52.00%	66.00%
Needle	34.00%	56.67%	44.00%
Kmeans	56.00%	68.00%	62.00%
LU	0.00%	0.00%	0.00%
Particlefilter	0.00%	74.00%	100.00%
HPCCG	0.00%	0.00%	0.00%
Xsbench	52.00%	40.00%	48.00%
FFT	62.00%	70.00%	84.00%
Average	24.36%	42.36%	46.00%

B. The Witnessed SDC-Coverage-Loss Issue in SID

1) Key Observations: Figure 2 presents the results of our preliminary study. The red bars indicate the expected SDC coverage provided by the existing SID method. The candlesticks represent the ranges of the SDC coverage measured among 50 generated inputs for each program. If the bottom of a candlestick is lower than the red bar, then the measured SDC coverage is lower than the expected coverage under some inputs, implying the loss of SDC coverage in practice. We make three important observations as follows.

First, the actual SDC coverage measured over different inputs is highly application-specific. The range of the coverage varies at different protection levels in the benchmarks. For example, the SDC coverage ranges from 0.00% to 100.00% at all 30%, 50%, and 70% protection levels in Kmeans. However, the range is lower than 11.20% in LU and Pathfinder.

Second, the actual SDC coverage can be much lower than the expected value provided in the existing SID method. That means a protected program may suffer from significantly

degraded SDC coverage when running with different inputs. For example, the SID expects a 100% coverage at the 50% protection level in Xsbench, whereas the measured coverage can be as low as 2.17% when running with different inputs. Similar situations also appear in Needle, kNN, and Kmeans.

Third, we witness a decrease in SDC coverage when increasing the protection level in SID if different inputs are used in the evaluation. This finding is surprising since SID assumes that the higher overhead budget, the better SDC coverage. For example, in kNN, the minimum SDC coverage decreases from 61.45% to 38.33% when increasing protection level from 50% to 70%. We observe similar situations in Needle, FFT, Particlefilter, Xsbench, and Kmeans when running the protected instances of them with different inputs. The root cause will be discussed in Section IV.

2) Percentage of Coverage-Loss Inputs: Table II lists the percentage of the inputs that fail to meet the expected SDC coverage in the experiment. We see that most of the applications have a significant amount of inputs that cause the loss of SDC coverage. For example, Particlefilter always fails to meet the target coverage at the protection level of 70%. FFT owns 62.00%, 70.00%, and 84.00% of the inputs that miss the expected SDC coverage at all three protection levels. On the other hand, the existing SID method performs well in LU, HPCCG, and Pathfinder. Specifically, no inputs are causing the coverage loss for LU and HPCCG according to Table II. The percentage of coverage-loss inputs in Pathfinder is only 4.00%, 6.00%, and 2.00% at the three protection levels, respectively. We will further study the reasons in Section IV.

Overall, the important takeaway is that the loss of SDC coverage in existing SID method could be severe and common over multiple inputs. Putting into context, if one uses existing SID method to protect an application, the actual SDC coverage in practice could be significantly lower than the expected coverage. As a result, the application will likely fail to meet the reliability target in the production environment since programs essentially run with different inputs in practice. Even worse, developers might be unaware of this issue. Instead, they may blindly trust the protected applications to produce correct computations.

IV. Incubative Instructions

In this section, we describe the root cause of the observed SDC-coverage-loss problem. We investigate this issue from analyzing the instructions that lead to SDCs in FI experiments. First, given a protection level of SID, we conduct FIs upon the protected program with the reference input since it is the default input used in SID to measure the cost and benefit of each instruction. We measure the SDC coverage and collect all the instructions that result in SDCs under the reference input. Next, we conduct FIs to the protected program with different inputs and get all the instructions which result in SDCs. We repeat the process with all 30 inputs of each benchmark and obtain the non-overlapping instructions in terms of reference input and other inputs. That is, we identify the instructions that do not result in SDCs with reference input, but lead to SDCs when using different inputs. Intuitively, in a protected program, those instructions will not cause SDCs when running with reference input since the implied protection is based on the reference input. However, they would cause *new* SDCs when running with other inputs, thus leading to the loss of SDC coverage in SID – they are the target instructions we want to characterize.

Further, we repeat the above procedure at three different SID protection levels of 30%, 50%, and 70%. We observe that some target instructions always exist among those levels. For example, 54.4% of those target instructions at 30% protection level still appear at the 50% protection level and the result is 41.3% after elevating the protection level from 50% to 70%. This is because those instructions are always not prioritized for protection even increasing the protection level to a very high position (70%). Such instructions lead to the SDC coverage loss on changed inputs. Note that if we keep moving the protection level to near full protection, the target instructions found at lower protection levels would gradually disappear – because SID eventually protects all the instructions at full protection.

Oriented from such observation, we now examine the reasons why these instructions are not prioritized by SID. Based on the analysis of Knapsack algorithm used in the instruction selection of SID (Section II-C), there are two possible reasons: (1) The cost, which is the dynamic cycles of the instruction in program execution, is too high; or (2) The benefit, which is the cost times the SDC probability of the instruction, is too small to be considered.

In our analysis, we do not observe certain patterns in their costs. Instead, we find that the benefits of these instructions are very small (near zeros) when using reference inputs, indicating tiny SDC probabilities in these instructions measured with reference inputs. However, the instructions could reveal much higher SDC probabilities when measured with other inputs. We name these instructions as *incubative instructions*.

Figure 3 shows an example of *incubative instruction* in FFT benchmark. As seen, when using the reference input, *icmp* instruction has an SDC probability of near 0%. This is because *%11* has a small negative value and it is difficult for a bit-flip to modify it to a positive value that is greater than 50. Therefore, all the faults occur at the *icmp* will be masked. However, when executing with a different input, *%11* can be a small positive value that is less than 50, and hence a bit-flip (say at a higher bit position) may modify it to a much larger value and thereby invert the comparison result, causing an SDC.

The existence of *incubative instructions* shows that when the program input changes, the error propagation behavior in a program execution may also change, resulting in different SDC probabilities. Based on our observations, we place instructions into *incubative instructions* if their benefits fall into the last 1% of the overall results with one input, but move out of the last 30% of the overall results when using different inputs.

In summary, existing SID method only runs with single reference input in the protection for a program, and *incubative instructions* have negligible SDC probabilities under the input,

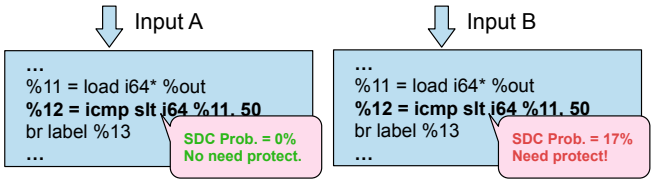


Fig. 3: Example of *incubative instructions* (FFT Benchmark)

making them difficult to be prioritized by SID. However, those instructions may reveal higher SDC probabilities when inputs are changed, and eventually cause the loss of SDC coverage.

V. MINPSID: HARDENING SID ACROSS INPUTS

In this section, we first explain the high-level design of MINPSID, and then present the details of each component.

A. Design Overview

Figure 4 shows the overall workflow of MINPSID. Users only need to provide the program source code and the protection level. MINPSID will automatically select instructions for protection, compute an expected SDC coverage of the protection, and generate the protected binary of the program. The entire process is fully automated without any user interventions. We implement MINPSID as LLVM passes and release it as open-source software on GitHub[2].

The main target of MINPSID is to efficiently identify as many *incubative instructions* as possible for a given program and re-prioritize them in the instruction selection phase of SID. A bruteforce method may consist of the following steps: (1) randomly generate program inputs; (2) conduct FI measurement over each instruction; (3) compare the measurement between inputs for identifying *incubative instructions*; and (4) repeat steps (1)-(3) until reaching time threshold or no new *incubative instructions* can be found. This approach, however, is impractical as the per-instruction FI measures over each input can be extremely time-consuming.

We propose an input searching technique in MINPSID to tackle this challenge, which performs a guided search for inputs that reveal *incubative instructions* in a program. The key insight is that different program executions flows can lead to different error propagation behaviors in the program, hence more likely revealing *incubative instructions* between the inputs. Based on this point, we leverage the static analysis in MINPSID to generate the control-flow graph (CFG) for a given program (③). With an input search engine (④) and dynamic profiling (⑤) of the program execution, MINPSID computes a weighted CFG list for input. The weighted CFG list represents the unique execution paths under the input and is used by the search engine to differentiate the program execution from all other inputs seen in the past search. In this way, MINPSID can identify *incubative instructions* with fewer inputs, hence requiring much fewer FIs to be performed (⑦).

[2]Download link: https://github.com/hyfshishen/SC22-MINPSID

After identifying the *incubative instructions*, MINPSID re-prioritizes those instructions by updating their benefits with the upper bounds measured among the generated inputs (⑧), to let SID prioritize them in the instruction selection phase (⑨). Finally, MINPSID completes instruction duplication and generates the protected binary of the program.

B. Design Details

We now present the details of each component in Figure 4.

1) SID Preparation (①, ②): We first measure the cost and benefit for each instruction with the reference input (①), and then construct a profile of both benefit and cost for the program (②). The benefit measurement consists of per-instruction FIs, and the cost measurement relies on dynamic profiling. This step is similar to the preparation in the existing SID method.

2) Input Search Engine (③-⑦): In this module, we design an input search engine that uses a genetic algorithm (GA) (④) to guide the search of inputs that help identify *incubative instructions* efficiently.

GA is a meta-heuristic search algorithm inspired by natural evolution [37]. In the following, we describe how we apply GA to identifying *incubative instructions*.

Mutation and Crossover Within each input generation in GA, we mutate inputs in the population. The idea of the mutation operation is to make a small change to the existing input. Our mutation operation is designed as follows: we first randomly select one argument from the input. If the argument is numerical, we modify the value with a random number between $\pm 10\%$ of the current value. If the argument is non-numerical, we randomly enumerate a possible value for the argument. For the crossover operation in GA, we first randomly select two program inputs generated in the current GA generation, randomly choose an argument in both inputs, and swap the arguments between the two inputs. Following common heuristics used in GA, we select 0.4 and 0.05 for the mutation and crossover rates, respectively [37].

Weighted Control-flow Graph The intuition behind our fitness function design is that if an input runs a different execution path, it likely leads to new error propagation behaviors in the path exploration, thus contributing to new *incubative instructions*. Thereby, the fitness function should quantitatively differentiate control-flows in program executions. We use dynamic program analysis to construct a weighted control-flow graph (CFG) for a generated input and compare the differences of the weighted CFGs with those of the generated inputs in history. Specifically, we first construct a static CFG for the program at compilation (③). In this static CFG, each node donates a basic block, and each edge represents a possible execution path. Note that all possible inputs of the program share the same static CFG. Next, for each generated input in GA, we execute the target program with the input, dynamically profiling the number of executions for each CFG edge in the program execution (⑤) – the count becomes the weight of the edge. Following the steps, we generate a weighted CFG for the input. Finally, we convert the weighted CFG to an indexed CFG list for calculating the fitness score. The list

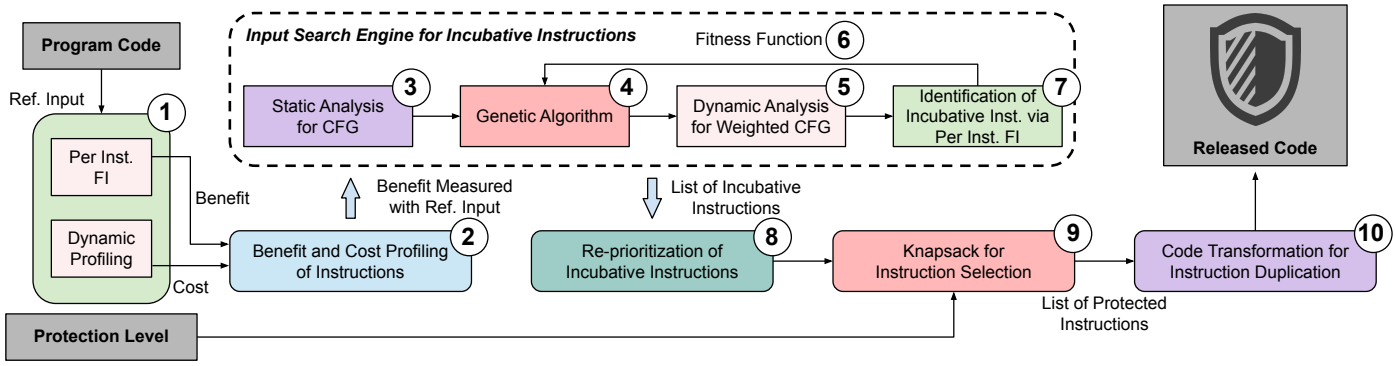


Fig. 4: Workflow of MINPSID

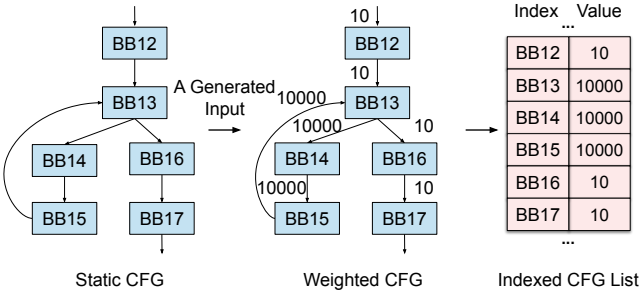


Fig. 5: A running example of weighted CFG construction (Code fragment from Pathfinder benchmark)

index is the index of each basic block, and the value of the list is the total number of execution counts in the edges that point to each basic block. Figure 5 shows an example of weighted CFG construction and its conversion. The static CFG (each BB denotes a basic block) in Pathfinder is obtained at program compilation. By executing the program under a generated input, the dynamic cycle of each basic block can be recorded, named as weighted CFG. Thus the indexed CFG list can be obtained by combining the dynamic cycle of each basic block.

Fitness Score After obtaining the indexed CFG list for a generated input, we evaluate the fitness score for the input (⑥) by checking the difference of their indexed CFG lists. The evaluation scheme is as follows: We denote indexed CFG list of the current input in the search as $L = \{i_1, i_2, ..., i_N\}$, where i_n denotes the number of executions of $n-$th basic block in the program execution with the input, and N is the total number of basic blocks. Let $B_j = \{b_{j1}, b_{j2}, ..., b_{jN}\}$ denote the indexed CFG lists of the $j-$th inputs that are recorded in GA search history. Let b_{jn} represent the number of executions of the $n-$th basic block in the program execution with the $j-$th input. We compute the average Euclidean distance, S_L, between L and every B_j. The formula is shown in Eq. 3, where the additional notation $|M|$ denotes the total number of inputs recorded in the GA search history.

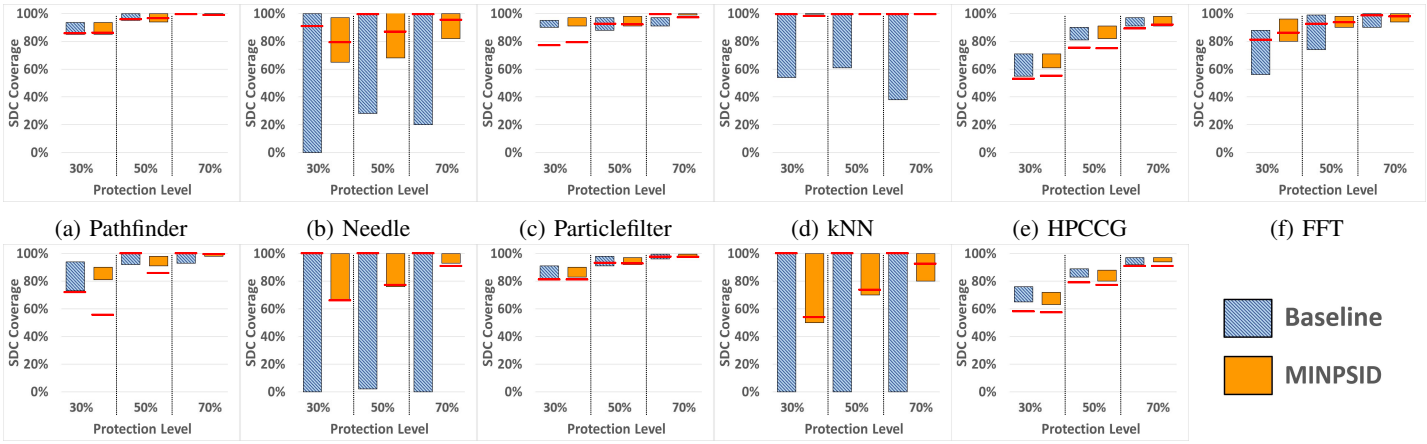
$$S_L = \frac{1}{|M|+1} \sum_{j=0}^{M} \sqrt{\sum_{n=1}^{N} |i_n - b_{jn}|^2} \quad (3)$$

The GA uses the fitness score to make optimization decisions - the candidate input with a higher fitness score will survive to next generation. The current GA search of inputs will terminate when fitness score no longer improves. After the new input is obtained in the current GA search, MINPSID will then perform per-instruction FI to obtain the SDC probabilities of every instruction. By comparing the per-instruction SDC probabilities of the input with those of historical inputs, a list of *incubative instructions* can be identified (⑦). The entire search process will terminate once the number of *incubative instructions* no longer increases.

3) Re-prioritization and Code Transformation (⑧, ⑨, ⑩): Once distinguished *incubative instructions*, MINPSID updates the benefits of those instructions with the highest values observed in the prior FI measurement (⑧). For the rest of the instructions that are not determined as *incubative instructions*, MINPSID adopts the cost and benefit profiles of the instructions (measured in ①). In this way, the new benefits of *incubative instructions* weigh higher in the Knapsack algorithm and will be prioritized in the instruction selection. The instructions for the protection under a target protection level are selected by running the Knapsack algorithm based on the updated cost and benefit profile (⑨). Finally, the program is transformed for instruction duplication as in the existing SID method (⑩), and a new protected binary executable is generated as the output of MINPSID.

VI. EVALUATION

In this section, we present the evaluation of MINPSID. We first show the effectiveness of MINPSID in mitigating the loss of SDC coverage in SID, then measure the efficiency of our heuristics in MINPSID to identify *incubative instructions*. Finally, we evaluate the time taken in executing MINPSID. All our experiments are conducted on 4 Debian servers, each has two 20-core Intel CPU processors.



(a) Pathfinder (b) Needle (c) Particlefilter (d) kNN (e) HPCCG (f) FFT

(g) Backprop (h) Xsbench (i) BFS (j) Kmeans (k) LU

Fig. 6: Mitigation of the loss of SDC coverage by MINPSID, compared with the existing SID method (Red bars indicate expected SDC coverage provided by each technique)

A. Mitigation of the Loss of SDC Coverage

Figure 6 demonstrates the effectiveness of MINPSID in mitigating the loss of SDC coverage. The baseline we use in the comparison is the state-of-the-art SID method described in Section II-C. As shown in the figure, the ranges of SDC coverage provided by MINPSID are much shorter in almost all the benchmarks at every protection level, compared with the baseline. The lower bounds of the coverage are much higher than those in the baseline. For example, in Needle, at 30%, 50%, and 70% protection levels, we observe that MINPSID provides at least 64.91%, 68.41%, and 82.45% SDC coverage respectively, whereas they are only 0.00%, 27.64%, and 19.60% in the baseline. The only exception is Backprop at the protection level of 50%. The minimum coverage observed is 91.12% in MINPSID, which is 0.47% lower than that of the baseline (91.59%). Overall, averaging over all the benchmarks at the 3 protection levels, MINPSID mitigates 97% of the loss of SDC coverage in the existing SID.

Another important observation is that, unlike the baseline, the minimum coverage provided by MINPSID always increases as the protection level increases in each benchmark. For example, in kNN, the minimum coverage provided by MINPSID is 99.54% at 30% protection level, 100.00% at 50% and 70.00% protection level. In contrast, the coverage ability of the baseline may decrease unexpectedly with increasing protection levels (e.g., from 61.45% to 38.33% at protection levels from 50% to 70%). That is, MINPSID provides more predictable SDC coverage as the protection level elevates, which is critical for developers to improve the software resilience and meet the reliability target.

Table III examines the percentage of the inputs that lead to loss of SDC coverage after using MINPSID. Note that the baseline result is shown in Table II in Section III-B, where we reveal the loss-of-SDC-coverage problem in existing SID method. By comparing the two tables, we observe that the percentage of the inputs that lead to the loss of coverage turns much lower in MINPSID than that in the baseline, with an average of 8.36% in MINPSID and 37.58% in the baseline. This shows that when running with different inputs, by applying MINPSID, the protected applications will less likely experience the loss of SDC coverage.

TABLE III: Percentage of Inputs that Results in the Loss of SDC Coverage in MINPSID

Benchmark	30% Level	50% Level	70% Level
Pathfinder	14.00%	4.00%	4.00%
kNN	0.00%	0.00%	0.00%
BFS	0.00%	10.00%	12.00%
Backprop	0.00%	0.00%	16.00%
Needle	2.00%	2.00%	2.00%
Kmeans	4.00%	12.00%	52.00%
LU	0.00%	0.00%	0.00%
Particlefilter	0.00%	56.00%	0.00%
HPCCG	0.00%	0.00%	2.00%
Xsbench	0.00%	6.00%	0.00%
FFT	12.00%	60.00%	6.00%
Average	2.91%	13.64%	8.55%

Finally, we observe that the expected SDC coverage (the red bars in the figure) provided by MINPSID is more conservative than those provided by existing SID method. That is, the expected coverage by MINPSID is lower in most of the cases. The only exception is Backprop, the expected coverage is lower than minimum coverage by 24.85% in 30% protection level. This is because the benefits of *incubative instructions* in Backprop vary much larger than those of other benchmarks, which lead to over-conservation in the protection. More importantly, in most of the benchmarks, MINPSID tends to bound the minimum coverage it can provide. This allows the users of MINPSID to make more accurate plan for the protection. The expected coverage tends to be conservative and so the SDC probabilities of their protected applications in production are closer to their reliability target. Note that the resulted conservative protection in MINPSID is as expected,

since we re-prioritize *incubative instructions* with their highest benefits measured in the instruction selection phase.

B. Efficiency of Identifying Incubative Instructions

In this section we show the efficiency of identifying *incubative instructions* by MINPSID. For comparison purpose, we replace the input search engine (③ - ⑥ in Figure 4) in MINPSID with a random searcher as the baseline. For every benchmark, the baseline randomly search inputs for identifying *incubative instructions*. Unlike our input search engine in MINPSID, the baseline has no fitness function and performs blind search.

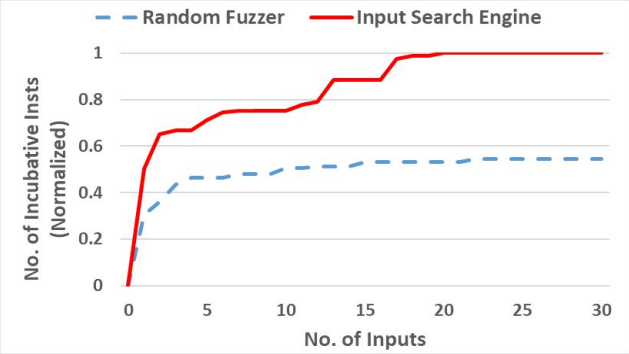


Fig. 7: Number of incubative instructions (normalized) identified by our input search engine in MINPSID and a random searcher

Figure 7 presents the results. As can be seen, our input search engine identifies more *incubative instructions* than the baseline at all time. MINPSID and the baseline techniques converge at 21 and 23 inputs on average respectively. Our input search engine can identify 45.60% more *incubative instructions* than the baseline upon the convergence. As a result, MINPSID mitigates additional 34% loss of SDC coverage in SID compared with the one that uses a random searcher.

C. Time Taken to Run MINPSID

In this section we measure the time taken to execute MINPSID for each benchmark. Figure 8 shows the results. We parallelize all the FIs in both MINPSID and the existing SID on our machines. *Per-Inst-FI (Ref Input)* indicates the time to finish FI and profiling with the reference input in each benchmark (① in Figure 4). *Per-Inst-FI (For Incubative Insts.)* represents the time to complete FI in identifying *incubative instructions* (⑦ in Figure 4). Together with the portion of running the input search engine, the three components constitute more than 98% of execution time on average in each benchmark.

Measured on our machine, the FI operations for identifying *incubative instructions* in MINPSID take an average of 26.42 minutes. Besides, the execution time of these FI operations ranges from 0.88 minutes in kNN to 101.25 minutes in Xsbench. The average time for running FIs with the reference input and input search engine is 3.87 and 33.41 minutes respectively. Specifically, the time for running FIs with the reference input ranges from 0.21 minutes (Pathfinder) to 21.07 minutes (HPCCG), while the time for input search engine is beteween 0.56 minute (Backprop) and 158.97 minutes (Xsbench). Overall, MINPSID takes an average of 63.71 minutes to complete the entire analysis for each benchmark. In comparison, running existing SID method consumes around 3.87 minutes. Note that though MINPSID takes longer time than the existing SID method, it is a one-time cost at the compilation before the deployment. Moreover, the FIs process can be easily parallelized with more computation resources to further speed up MINPSID.

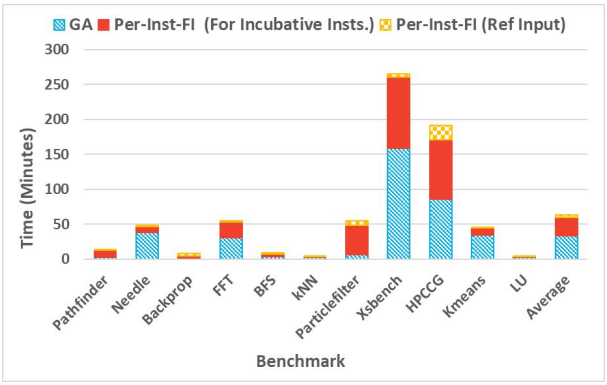


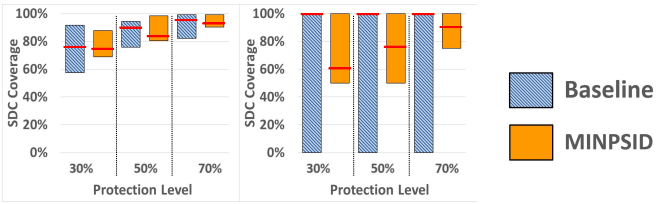
Fig. 8: MINPSID Execution Time

VII. Case Study: MINPSID with Real-World Program Inputs

As mentioned in Section III, we use randomly generated inputs for evaluating MINPSID. While randomly generating program inputs is an acceptable way to sample realistic inputs seen in other related works [12], [22], [29], we want to investigate the effectiveness of MINPSID when executing with real-world program inputs. We realize that public execution logs or datasets are not always available for the benchmarks we use, thus it may not be possible for us to test every benchmark we have with real-world program inputs. However, we find KONECT [38] and Kaggle [39] which provide graph and clustering datasets that are collected from real-world problems. We use these datasets to execute our BFS and Kmeans benchmarks. KONECT is a graph collection, containing real-world social networks, citation networks etc. [38], whereas Kaggle is a data science community that provides datasets from real-world problems [39]. More specifically, we select top 30 graph datasets in KONECT as the program inputs to run BFS benchmark, and 10 available datasets in clustering problems from Kaggle to execute Kmeans in our case study.

As usual, we first protect the program using MINPSID at 30%, 50%, and 70% protection levels, then run the protected program with the chosen real-world program inputs. For each input, we conduct FI evaluation to the protected program as we did in our evaluation, in order to measure the loss of SDC coverage. The results are presented in Figure 9.

As can be seen, in both benchmarks, MINPSID can effectively reduce the loss of SDC coverage. The range of SDC



(a) BFS (b) Kmeans

Fig. 9: MINPSID with Real-World Program Inputs

coverage provided by MINPSID are shorter in every protection levels compared with the baseline SID. As well, the lower bounds of the coverage are much higher than those in the baseline, suggesting that MINPSID mitigates the loss of SDC coverage among inputs. For example, in BFS, at 30%, 50%, and 70% protection levels, MINPSID provides at least 68.87%, 80.38%, and 90.19% SDC coverage respectively, whereas they are only 57.46%, 75.82%, and 82.09% in the baseline. Similar observations can be made in Kmeans.

TABLE IV: Percentage of Inputs that Results in the Loss of SDC Coverage in MINPSID

Benchmark	30% Level	50% Level	70% Level
BFS (Baseline)	60.00%	76.67%	86.67%
BFS (MINPSID)	10.00%	23.33%	16.67%
Kmeans (Baseline)	50.00%	60.00%	60.00%
Kmeans (MINPSID)	10.00%	30.00%	10.00%

Table IV shows the percentage of the inputs that results in the loss of SDC coverage. We find that the percentage of the inputs that lead to the loss of coverage is much lower in MINPSID than that in the baseline, with an average of 16.67% in MINPSID and 65.56% in the baseline, indicating that the protected applications will less likely experience the loss of SDC coverage when applying MINPSID. The above observations are consistent with what we have analyzed in the evaluation section.

VIII. DISCUSSION

A. Performance Overhead across Multiple Inputs

In this work, we investigate the loss of SDC coverage problem in existing SID, which imposes an immediate reliability threat to HPC community. Meanwhile, we observe that there also exists variance in the performance overheads when running protected programs with different inputs. Recall that the protection level indicates the expected amount of dynamic instructions that are duplicated by SID (Section II). We observe that on average, existing SID method duplicates only 15.61%, 28.63%, 46.31% of dynamic instructions at 30%, 50%, and 70% protection levels, respectively, when running with different inputs in each benchmark. That is, at all levels, the actual numbers of duplicated dynamic instructions are lower than the said target protection levels by 14.39%, 21.37%, 23.69%, respectively. On the other hand, we observe a similar shortage in MINPSID as well, they are lower than the target protection levels by 15.20%, 22.68%, 22.05%. This indicates that MINPSID experiences a similar variance in performance overhead to the existing SID method under multiple inputs. We leave the mitigation of the performance overhead variance in our future work.

B. Implication to Parallel Programs

SID is applicable to parallel programs (e.g., multi-threaded applications), as studied in prior works [2], [35]. Recall that the entire process of SID is done at compile-time. In the studied applications, each thread of the program runs the same copy of the protected code, and the mismatch (due to error, if any) must be detected through the checking before program synchronization locations such as function calls (hence before any threads interleave, Section II). Thereby, our technique (as well as the existing SID method) detects SDCs the same way on each individual thread of a multi-threaded program compared with that in a single-threaded program. Therefore, we believe MINPSID also works on multi-threaded programs. We experiment MINPSID on a multi-threaded version of FFT with three settings of threads. We observe that existing SID method has the SDC coverage loss of 7.52%, 12.13%, and 6.00% in terms of 1, 2 and, 4 threads. In contrast, MINPSID effectively decreases the loss of the SDC coverage to 2.50%, 5.50% and 1.46% respectively. The preliminary results match our expectations.

C. Threats To Validity

The first threat to validity we identify in this work is the method we used to conduct FIs. As mentioned, we conduct our FI experiments at LLVM level using an open-source fault injector, LLFI. There has been debate on whether LLVM-level FI is accurate or not in studying hardware faults. However, recent works have shown that LLVM-level FI is accurate in measuring SDCs [15], [16], so we adopt it as we focus on SDCs in this work. Another motivation for us to use LLVM-level FI is that many other SID studies in the literature are conducted using LLVM-level FIs [2], [6], [7], [30], so we consider our choice is appropriate. On the other hand, we try to adopt the benchmarks that are used in other recent works in the area. Our results may be specific to the selection of the benchmarks, although this is not what we have observed. Our decisions on benchmark selections are similar to what other closely related studies have practiced [2], [7], [10], [30], [33].

IX. RELATED WORK

Instruction duplication techniques have been proposed for more than two decades [40]–[42]. Soon after that, researchers have observed that not all the program states are equally likely vulnerable, hence one can selectively protect a fraction of instructions in a program to achieve high coverage with low overheads, which have become the prototypes of SID [5], [43], [44]. Along with the direction, there has been a large body of works investigating the efficiency and effectiveness of SID in a broad spectrum of applications [2], [6], [7], [26], [35]. However, these studies confine themselves to single input when studying SID, which lead to loss of coverage issues when

the protected programs run with different inputs, as we show in this work.

Recently, there have been studies investigating how program inputs may affect error propagation and resiliency evaluations. Di Leo et al. [45] characterized the relationship between program workloads and the failure distribution model. Folkesson et al. [46] analyzed different workloads and program failure rates. Li et al. [12] modeled input-dependent error propagation in programs. Yang et al. [47] proposed SUGAR to speed up the GPU evaluation via input sizing. Mahmoud et al. [29] 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 do not investigate multiple input scenarios in SID. Rahman et al. [22] proposed Peppa-X to identify the upper-bound of program SDC probability. Unlike our work which studies SID protection across different inputs, their work focused on estimating the highest program SDC probability given a set of inputs. Our work is the first one that investigates the root causes of the loss-of-SDC-coverage issue in SID across multiple inputs, and proposes an efficient and effective mitigation technique for SID.

Originated from an operating system academia project [48], input search has been well studied and widely recognized in test case generation and software vulnerability detection in the past years [49]. Practical supports from industry [50] has led to the boosting applications of search to various areas [51]–[53]. Recently, software issues correlated with hardware microarchitectural designs have attracted intensive attentions from the program analysis community [49], [54], [55]. The techniques in MINPSID in general belongs to this category. However, MINPSID owns innovative differences compared to other methods. Existing search tools typically works under the coverage-guided mechanism, which guides the input mutation to reach better coverage of certain metrics. Meanwhile, they primarily focus on observable program execution issues like crash, hang, etc. Unfortunately, the classic search schema fails to test SDC issues which present no apparent problems but corrupted data values. By comparison, MINPSID absorbs the general idea of input search and realizes a new analysis framework. It leverages the genetic algorithm and control-flow-based fitness function to identify critical instructions through mutated inputs, hence hardening the SID protection with the precise set of *incubative instructions*.

X. Conclusion and Future Work

In conclusion, existing SID may suffer the loss of SDC coverage issue when the protected program runs with different inputs. We observe that the problem is due to *incubative instructions* in the program. We propose MINPSID, which leverages an input search technique to identify *incubative instructions* and re-prioritizes the *incubative instructions* in selective protection. The evaluation shows that MINPSID effectively mitigates the loss of SDC coverage issue across multiple inputs in SID. In the future, we plan to extend MINPSID in two possible directions: (1) Accelerating input search process by exploring more efficient fuzzing algorithms and heuristics, and (2) Extending our work to other upcoming platforms such as GPUs and special-purpose accelerators.

Acknowledgments

This research was supported in part by the National Science Foundation (NSF) under Grant No. 2211538 and 2211539, and the U.S. Department of Energy, Office of Science under contract DE-AC02-06CH11357. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF, the U.S. Department of Energy, or Baidu.

References

[1] K. Lee, A. Shrivastava, I. Issenin, N. Dutt, and N. Venkatasubramanian, "Mitigating soft error failures for multimedia applications by selective data protection," in *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, 2006, pp. 411–420.

[2] C. Kalra, F. Previlon, N. Rubin, and D. Kaeli, "Armorall: Compiler-based resilience targeting gpu applications," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 17, no. 2, pp. 1–24, 2020.

[3] R. Lucas, J. Ang, K. Bergman, S. Borkar, W. Carlson, L. Carrington, G. Chiu, R. Colwell, W. Dally, J. Dongarra *et al.*, "Doe advanced scientific computing advisory subcommittee (ascac) report: top ten exascale research challenges," USDOE Office of Science (SC)(United States), Tech. Rep., 2014.

[4] H. D. Dixit, S. Pendharkar, M. Beadon, C. Mason, T. Chakravarthy, B. Muthiah, and S. Sankar, "Silent data corruptions at scale," *arXiv preprint arXiv:2102.11245*, 2021.

[5] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, "Shoestring: probabilistic soft error reliability on the cheap," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 1. ACM, 2010, p. 385.

[6] I. Laguna, M. Schulz, D. F. Richards, J. Calhoun, and L. Olson, "Ipas: Intelligent protection against silent output corruption in scientific applications," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. ACM, 2016, pp. 227–238.

[7] G. Li, K 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.

[8] P. H. Hochschild, P. Turner, J. C. Mogul, R. Govindaraju, P. Ranganathan, D. E. Culler, and A. Vahdat, "Cores that don't count," in *HotOS '21: Workshop on Hot Topics in Operating Systems, Ann Arbor, Michigan, USA, June, 1-3, 2021*, S. Angel, B. Kasikci, and E. Kohler, Eds. ACM, 2021, pp. 9–16.

[9] 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 *ACM SIGARCH Computer Architecture News*, vol. 40, no. 1. ACM, 2012, p. 123.

[10] G. Georgakoudis, I. Laguna, D. S. Nikolopoulos, and M. 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*, 2017, pp. 1–14.

[11] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Control-flow checking by software signatures," *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 111–122, 2002.

[12] G. Li and K. Pattabiraman, "Modeling input-dependent error propagation in programs," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2018, pp. 279–290.

[13] B. Nie, J. Xue, S. Gupta, T. Patel, C. Engelmann, E. Smirni, and D. 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)*. IEEE, 2018, pp. 95–106.

[14] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization*. IEEE, 2004, p. 75.

[15] L. Palazzi, G. Li, B. Fang, and K. 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)*. IEEE, 2019, pp. 151–162.

[16] J. Wei, A. Thomas, G. Li, and K. 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)*. IEEE, 2014, pp. 375–382.

[17] G. Li, K. Pattabiraman, C.-Y. Cher, and P. Bose, "Understanding error propagation in GPGPU applications," in *International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 240–251.

[18] G. B. Mathews, "On the partition of numbers," *Proceedings of the London Mathematical Society*, vol. 1, no. 1, pp. 486–490, 1896.

[19] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *International Symposium on Workload Characterization (IISWC 2009)*. IEEE, 2009, pp. 44–54.

[20] A. R. Anwer, G. Li, K. Pattabiraman, M. Sullivan, T. Tsai, and S. K. S. Hari, "Gpu-trident: efficient modeling of error propagation in gpu programs," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–15.

[21] L. Yang, B. Nie, A. Jog, and E. Smirni, "Practical resilience analysis of gpgpu applications in the presence of single-and multi-bit faults," *IEEE Transactions on Computers*, vol. 70, no. 1, pp. 30–44, 2020.

[22] M. H. Rahman, A. Shamji, S. Guo, and G. Li, "Peppa-x: finding program test inputs to bound silent data corruption vulnerability in hpc applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–13.

[23] M. A. Heroux, "Hpccg solver package," Sandia National Laboratories, Tech. Rep., 2007.

[24] C.-K. Chang, S. Lym, N. Kelly, M. B. Sullivan, and M. Erez, "Evaluating and accelerating high-fidelity error injection for hpc," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 577–589.

[25] Z. Li, H. Menon, K. Mohror, P.-T. Bremer, Y. Livant, and V. Pascucci, "Understanding a program's resiliency through error propagation," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021, pp. 362–373.

[26] S. K. S. Hari, R. Venkatagiri, S. V. Adve, and H. Naeimi, "Ganges: Gang error simulation for hardware resiliency evaluation," in *International Symposium on Computer Architecture (ISCA)*. IEEE, 2014, pp. 61–72.

[27] L. Guo, D. Li, I. Laguna, and M. Schulz, "Fliptracker: Understanding natural error resilience in hpc applications," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 94–107.

[28] X. Wei, N. Jiang, X. Wang, and H. Yue, "Detecting sdcs in gpgpus through an efficient instruction duplication mechanism," in *International Conference on Knowledge Science, Engineering and Management*. Springer, 2021, pp. 571–584.

[29] A. Mahmoud, R. Venkatagiri, K. Ahmed, S. Misailovic, D. Marinov, C. W. Fletcher, and S. 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*, 2019, pp. 1087–1103.

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

[31] S. K. S. Hari, T. Tsai, M. Stephenson, S. W. Keckler, and J. Emer, "Sassifi: Evaluating resilience of gpu applications," in *Proceedings of the Workshop on Silicon Errors in Logic-System Effects (SELSE)*, 2015.

[32] B. Fang, Q. Guan, N. Debardeleben, K. Pattabiraman, and M. Ripeanu, "Letgo: A lightweight continuous framework for hpc applications under failures," in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, 2017, pp. 117–130.

[33] L. Guo, D. Li, and I. Laguna, "Paris: Predicting application resilience using machine learning," *Journal of Parallel and Distributed Computing*, 2021.

[34] Q. Lu, K. Pattabiraman, M. S. Gupta, and J. A. Rivers, "Sdctune: a model for predicting the sdc proneness of an application for configurable protection," in *Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, 2014, pp. 1–10.

[35] A. Mahmoud, S. K. S. Hari, M. B. Sullivan, T. Tsai, and S. W. Keckler, "Optimizing software-directed instruction replication for gpu error detection," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 842–854.

[36] C.-K. Chang, G. Li, and M. Erez, "Evaluating compiler ir-level selective instruction duplication with realistic hardware errors," in *2019 IEEE/ACM 9th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS)*. IEEE, 2019, pp. 41–49.

[37] R. 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*, vol. 2. IEEE, 2000, pp. 1034–1037.

[38] J. Kunegis, "KONECT – The Koblenz Network Collection," in *Proc. Int. Conf. on World Wide Web Companion*, 2013, pp. 1343–1350. [Online]. Available: http://dl.acm.org/citation.cfm?id=2488173

[39] https://www.kaggle.com/, "Kaggle."

[40] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Error detection by duplicated instructions in super-scalar processors," *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 63–75, 2002.

[41] P. P. Shirvani, N. Oh, E. J. Mccluskey, D. Wood, M. N. Lovellette, and K. Wood, "Software-implemented hardware fault tolerance experiments: Cots in space," in *International Conference on Dependable Systems and Networks (FTCS-30 and DCCA-8), New York (NY)*, 2000.

[42] P. P. Shirvani, N. Saxena, N. Oh, S. Mitra, S.-Y. Yu, W.-J. Huang, S. Fernandez-Gomez, N. A. Touba, and E. J. McCluskey, "Fault-tolerance projects at stanford crc," in *MAPLD 1999- Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference, 2 nd, Johns Hopkins Univ, APL, Laurel, MD*. Citeseer, 1999.

[43] A. Pillai, *Improving performance for dual instruction execution by selective instruction duplication*. Southern Illinois University at Carbondale, 2006.

[44] K. Pattabiraman, G. P. Saggese, D. Chen, Z. Kalbarczyk, and R. K. Iyer, "Dynamic derivation of application-specific error detectors and their implementation in hardware," in *2006 Sixth European Dependable Computing Conference*. IEEE, 2006, pp. 97–108.

[45] D. Di Leo, F. Ayatolahi, B. Sangchoolie, J. Karlsson, and R. 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*. Springer, 2012, pp. 198–209.

[46] P. Folkesson and J. Karlsson, *The effects of workload input domain on fault injection results*. Citeseer, 1999.

[47] L. Yang, B. Nie, A. Jog, and E. Smirni, "Sugar: Speeding up gpgpu application resilience estimation with input sizing," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 5, no. 1, pp. 1–29, 2021.

[48] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[49] O. Oleksenko, B. Trach, M. Silberstein, and C. Fetzer, "Specfuzz: Bringing spectre-type vulnerabilities to the surface," in *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, S. Capkun and F. Roesner, Eds. USENIX Association, 2020, pp. 1481–1498.

[50] https://github.com/google/AFL, "american fuzzy lop."

[51] C. Wen, H. Wang, Y. Li, S. Qin, Y. Liu, Z. Xu, H. Chen, X. Xie, G. Pu, and T. Liu, "Memlock: memory usage guided fuzzing," in *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, G. Rothermel and D. Bae, Eds. ACM, 2020, pp. 765–777.

[52] J. Wei, J. Chen, Y. Feng, K. Ferles, and I. Dillig, "Singularity: pattern fuzzing for worst case complexity," in *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*, G. T.

Leavens, A. Garcia, and C. S. Pasareanu, Eds. ACM, 2018, pp. 213–223.

[53] C. Aschermann, S. Schumilo, A. Abbasi, and T. 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*. IEEE, 2020, pp. 1597–1612.

[54] M. Wu, S. Guo, P. Schaumont, and C. Wang, "Eliminating timing side-channel leaks using program repair," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, F. Tip and E. Bodden, Eds. ACM, 2018, pp. 15–26.

[55] S. Guo, Y. Chen, P. Li, Y. Cheng, H. Wang, M. Wu, and Z. Zuo, "Specusym: speculative symbolic execution for cache timing leak detection," in *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, G. Rothermel and D. Bae, Eds. ACM, 2020, pp. 1235–1247.