Demystifying and Mitigating Cross-Layer Deficiencies of Soft Error Protection in Instruction Duplication

Zhengyang He
University of Iowa
Iowa City, IA, USA
zhengyang-he@uiowa.edu

Yafan Huang
University of Iowa
Iowa City, IA, USA
yafan-huang@uiowa.edu

Hui Xu
Fudan University
Shanghai, China
xuh@fudan.edu.cn

Dingwen Tao
Indiana University
Bloomington, IN, USA
ditao@iu.edu

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

ABSTRACT

Soft errors are prevalent in modern High-Performance Computing (HPC) systems, resulting in silent data corruptions (SDCs), compromising system reliability. Instruction duplication is a widely used software-based protection technique against SDCs. Existing instruction duplication techniques are mostly implemented at LLVM level and may suffer from low SDC coverage at assembly level. In this paper, we evaluate instruction duplication at both LLVM and assembly levels. Our study shows that existing instruction duplication techniques have protection deficiency at assembly level and are usually over-optimistic in the protection. We investigate the root-causes of the protection deficiency and propose a mitigation technique, FLOWERY, to solve the problem. Our evaluation shows that FLOWERY can effectively protect programs from SDCs evaluated at assembly level.

CCS CONCEPTS

• **Computer systems** → **High-performance computing**; • **Dependable and fault-tolerant systems and networks**;

KEYWORDS

System Reliability, Hardware Transient Faults, Instruction Duplication, Compiler Transformation, Architecture, Fault Injection

ACM Reference Format:
Zhengyang He, Yafan Huang, Hui Xu, Dingwen Tao, and Guanpeng Li. Demystifying and Mitigating Cross-Layer Deficiencies of Soft Error Protection in Instruction Duplication. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3581784.3607078

1 INTRODUCTION

The prevalence of transient hardware faults, also known as soft errors, is expected to rise in high-performance computing (HPC) systems, owing to factors such as system scaling, technology advance, and voltage reduction [22, 23, 28]. These faults may affect program instructions being executed in the systems, and corrupt program output. We call this silent data corruption or SDC. Traditionally, HPC systems were protected using hardware-based solutions such as hardware redundancy and circuit hardening methods. However, they impose significant overheads in performance and energy consumption, thereby are challenging to deploy in practice.

To overcome these challenges, researchers have proposed software solutions. Prior research has demonstrated that a small percentage of instructions are responsible for almost all the SDCs in a program [11, 12, 18]. In order to achieve low-overhead protection against SDCs, developers have proposed instruction duplication techniques to selectively protect vulnerable instructions with priority. The technique has been demonstrated to be efficient, and widely applied in HPC to reduce SDC rate [12, 21, 24].

The instruction duplication technique makes a copy of original instruction sequence and compares the computation results of both. If the computations mismatch between the two copies, errors are detected. The entire technique can be implemented using compiler techniques. A vast number of existing instruction duplication techniques are implemented at LLVM intermediate representation (IR) level [2, 10–12, 15, 25]. This is because, at IR level, it allows both error sensitivity analysis (e.g., fault injection and characterization, etc) and handy code transformation as per analysis result using rich LLVM compiler libraries at compile time before the deployment of the program [2, 11, 12, 15]. In contrast, instruction duplication technique implemented at lower level such as assembly instructions has limited means for a comprehensive analysis and flexible transformation. For instance, Intel PIN [19], by far the most commonly sought assembly-level tool comes with only dynamic instrumentation at program runtime, restricting the possibility to implement a program-specific selective protection scheme which can be practically done only at compile time.

While LLVM is commonly used in implementing selective instruction duplication, the fault coverage at assembly instruction level is unknown given the faults essentially occur at lower layer and it is assembly instructions that read the faults at runtime. Prior studies evaluate the technique at only LLVM level and claim based on that [10–12, 18, 25], thereby it is unclear whether LLVM-based instruction duplication techniques are effective and by how much

SC '23, November 12–17, 2023, Denver, CO, USA

ACM ISBN 979-8-4007-0109-2/23/11.
https://doi.org/10.1145/3581784.3607078

if faults are injected from assembly instruction level – we aim to answer this research question in this paper.

In this work, we quantitatively evaluate the LLVM-based instruction duplication technique by injecting faults at lower level, assembly instructions of programs. We examine the measured effectiveness of the protection and compare it end-to-end with those claimed in prior work. We have two main findings: (1) There is *non-negligible gap* between the effectiveness and efficiency of the protection between the evaluation at LLVM level and assembly level. The results are particularly application-specific. In addition, the fault coverage often falls short at assembly instruction level compared with LLVM level evaluation, indicating an over-optimistic estimate of protection in the existing studies; (2) The *good news* is that the root-causes leading to the shortfalls in the protection fall into strong identifiable patterns which can be characterized using compiler program analysis techniques. With the knowledge of the root-causes, we propose Flowery, a set of compiler patches that transparently harden the cross-layer deficiencies in the existing LLVM-based instruction duplication techniques, closing the gap. *To the best of our knowledge, we are the first ones who quantify and characterize the cross-layer protection deficiency of instruction duplication, investigate the root-causes of it, and propose solutions to mitigate the deficiency.*

Our main results are as follows:

- We assess the protection effectiveness of existing LLVM-based instruction duplication in 16 benchmarks. We measure the SDC coverage at both LLVM instruction level and assembly instruction level, then compare their differences. Our results show a considerable disparity between the two level results, implying that instruction duplication technique often fails to strike a satisfactory balance protection between LLVM and assembly levels. The average SDC coverage gap between LLVM level and assembly level is 31.21%, while the highest SDC coverage gap reaches up to 82% in Stringsearch benchmark.
- We analyze the root-causes of the deficiencies and summarize them into five categories: store penetration, branch penetration, comparison penetration, call penetration, and mapping penetration.
- We propose a novel technique called Flowery to mitigate the protection deficiency based on the analysis of the root-causes. Our technique achieve an average of 31.21% improvement in SDC coverage compared with existing instruction duplication technique while incurring only 2.71% additional runtime performance overhead.

2 FAULT MODEL AND TERMINOLOGY

In this section, we first define the terms we use, followed by a brief description of the fault model used in the study.

2.1 Technical Terminology

We first define the terms we will use in the study:

Silent Data Corruption (SDC): A fault occurs and affect program execution. The program completes its execution, but the output differs from its error-free execution.

SDC Coverage: SDC coverage is defined as the proportion of all SDCs occurring in a program that are detected by a given protection technique such as instruction duplication. SDC coverage can be calculated by $(SDC_{raw} - SDC_{prot})/SDC_{raw}$, where SDC_{prot} and SDC_{raw} denote program SDC probability with and without protection respectively.

DUE: Detectable unrecoverable errors (DUEs) means the program execution terminates early or crashes due to faults. For example, operating system may throw an exception (e.g., segmentation fault) and terminate the program execution.

Static Instruction: The instructions in the program code seen from compile time.

Dynamic Instruction: An instance of static instruction that is executed in a program execution.

2.2 Fault Model

A fault model describes *what, when,* and *where* a fault happens in the target system under study. It then abstracts to a model that guides the simulations of fault occurrence. Every resilience study has to assume a fault model which sets the scope of the study. When it comes the protection technique in resilience, there rarely exists a single technique that can mitigate faults occurred in all possible hardware components in a system. That is, each mitigation technique in the literature ties to a fault model that the technique is designed for. Thereby, their evaluation should be subjected to the same fault model.

In the research of soft errors, there are two main fault models that are commonly studied in the literature. They are classified based on the locations of fault occurrence: (1) memory faults and (2) datapath faults. Memory faults focus on faults occurred in large storage such as maim memory and cache etc, and can be mitigated by deploying techniques such as Error Correction Code (ECC) [6]. On the other hand, datapath faults are mainly for the faults happened in individual latches in processor pipeline and load/store units etc. These faults can be mitigated by applying instruction duplication techniques [11, 12, 15, 18].

On the other front, the number of bit-flips may also vary. While there have been some recent insights showing that multiple-bit-flip faults may be possible in the future [30], vast majority of current studies in the literature assume single-bit-flip in their fault models [11, 12, 18], so our study adopts single-bit-flips in our experiments. Since our study focuses on instruction duplication techniques, our fault model is set to the common datapath fault model that the technique aims for. In this paper, we adopt the datapath fault model with single bit-flips, which is commonly studied in prior work in the area of instruction duplication.

Another important note is that the fault model used in a cross-layer study needs to be consistent. It makes nonsense if a fault model is used at one level but a different one at the other level. In our case, we consider the evaluation at LLVM and assembly levels, so our datapath fault model needs to be consistent at both levels.

In this study, we focus on SDCs rather than DUEs among the failure types, as SDCs are the most insidious type of failures in HPCs [12, 15, 16].

3 INSTRUCTION DUPLICATION TECHNIQUE

Instruction duplication technique has been proposed in recent decades [24, 26, 27]. The technique can be used to protect programs from soft errors occurred in datapath [8, 12, 15]. In short, instruction duplication makes a copy of computation sequence in the program and compares the results between the two copies. If any mismatch is observed, errors are detected. Figure 1 shows how instruction duplication works in more details. In the example, instruction *D* is a synchronization point (e.g., store, function call, or control-flow branch.) at the end of a data dependency sequence, instruction *A*, *B*, and *C*. The technique duplicates instructions by inserting *A'*, *B'*, and *C'* along with a checker before the synchronization point *D* (say a store instruction) at the compile time. If any faults occur in either copy, the checker will detect the mismatch at runtime, and hence detect the error.

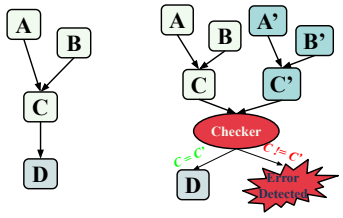

Figure 1: Example of instruction duplication; (a) Original program. (b) After instruction duplication.

The instruction duplication technique can be highly configurable [11, 12, 15, 18]. That is, developers can selectively choose which instructions shall be protected based on the vulnerability analysis and reliability target. Duplicating an instruction at compile time will incur performance overhead at runtime since the instructions are doubled. At the same time, instructions have different probabilities of causing SDCs in a program. Therefore, the trade-off between SDC coverage and performance overhead presents an optimization opportunity for developers to selectively protect the most beneficial instructions with priority in order to achieve high SDC coverage while incurring low performance overhead. The optimization problem can be formulated as a classic 0-1 knapsack optimization problem [7, 17], with the SDC coverage provided as benefit and the performance overhead as cost.

There have been a number of studies that show instruction duplication techniques can provide high SDC coverage with low performance overhead [11, 12, 15, 18, 21]. Since SDC distribution is highly application-specific among instructions in a program, developers need to measure the SDC probability of each instruction before they can choose which instruction to duplicate given a *protection level*. Here, protection level refers to the maximum amount of additional dynamic instructions generated due to the duplication of static instructions in the program. As a result, fault injection analysis is often used to assess the SDC probabilities of each instruction as well as the overall SDC coverage a protection strategy provides. LLVM compiler infrastructure [13] presents a unique opportunity to allow developers to conduct fault injection analysis per each LLVM IR instruction while being able to duplicate the one needed at the same IR level based on the analysis. The procedure is at compile time before program execution.

On the other front, it is possible to conduct assembly level instruction duplication. For example, Intel PIN [19], a popular dynamic instrumentation tool for assembly code can duplicates assembly instructions at runtime. While the technique is capable to conduct a full duplication (e.g., duplicate all the executed instructions unconditionally), it is impractical to be selective at runtime due to the large runtime overhead that the condition checking incurs. In addition, due to the lack of a handy assembly code compiler that is freely available to the public for vulnerability analysis and code transformation at compile time, developers prefer LLVM level implementation of instruction duplication. Therefore, most existing instruction duplication techniques are implemented at LLVM IR level [1, 4, 5, 15]. This largely motivates our study to conduct evaluation at assembly level to examine the protection effectiveness of LLVM-based instruction duplication techniques.

4 EXPERIMENTAL SETUP

In this section, we describe the experimental setup we have for the cross-layer evaluation of instruction duplication.

4.1 Program Benchmarks

Table 1: Details of Benchmarks; DI Count represents the number of dynamic instructions in million.

Benchmark	Suite	Domain	DI Count
Backprop	Rodinia	Machine Learning	148.20
BFS	Rodinia	Graph Algorithm	527.92
Pathfinder	Rodinia	Dynamic Programming	0.6
LUD	Rodinia	Linear Algebra	59.16
Needle	Rodinia	Dynamic Programming	593.39
kNN	Rodinia	Machine Learning	206.44
EP	NPB	Parallel Computing	4904.50
CG	NPB	Gradient Algorithm	721.95
IS	NPB	Sort Algorithm	43.97
FFT2	MiBench	Signal Processing	3.24
Quicksort	MiBench	Sort Algorithm	1.98
Basicmath	MiBench	Mathematical Calculations	2.80
Susan	MiBench	Image Recognition	42.30
CRC32	MiBench	Error Detection	21.90
Stringsearch	MiBench	Comparison Algorithm	2.60
Patricia	MiBench	Data Structure	4.96

We use 16 applications drawn from three benchmark suites which are commonly used in HPC research [10, 12, 16]. We choose the ones we are able to compile to both LLVM IR and binary for the fault injectors we use in the experiment. Table 1 provides a summary of the benchmarks used. For each benchmark, our compilation is without any standard optimization. The benchmarks will be used with instruction duplication technique as well as in fault injection experiments in the study.

4.2 Implementation of Instruction Duplication

We follow the design of instruction duplication technique used in [2, 11, 12, 15, 18] and described in Section 3. Similar to these related studies, we use LLVM to implement the instruction duplication and validate the correctness of the implementation in Section 5.

The implementation can be downloaded from our GitHub repository[1]. We use the instruction duplication technique to protect each target benchmark with 30%, 50%, 70%, and 100% protection levels respectively for the cross-layer evaluation.

4.3 Fault Injection Methodology

In order to conduct fault injection evaluation on both LLVM IR and assembly levels, we have to inject faults at each level respectively. The details of the fault injection process are described as follows.

We inject faults at LLVM level by implementing a set of LLVM compiler passes to do so. On the other hand, we use Intel PIN tool to inject faults at assembly level. At either case, there are three parts when conducting fault injection experiments: (1) Instrumentation; (2) Profiling; (3) Fault injection. In the instrumentation phase, the injectors add necessary code for implementing the profiling and fault injection mechanism. In the profiling phase, the injectors need to figure out the total number of dynamic instructions (either at LLVM or assembly levels). Finally, in the fault injection phase, each campaign will select a dynamic instruction among all the executed ones for injecting a fault.

We configure both fault injectors to simulate fault occurrence as per our fault model (Section 2.2). As we focus on faults occurred in datapath, we inject faults into the destination register of a chosen instruction. In more detail, in each fault injection campaign, we randomly select an executed dynamic instruction, then we randomly choose a bit position in its destination register to flip. We repeat the process for 3,000 campaigns for each benchmark at each protection level at LLVM and assembly levels in order to achieve statistical significance in the measurement. The method is standard and commonly used in study the datapath fault model and hence inline with prior work in the related area [11, 12, 15, 16, 18, 25, 28, 33]. In particular, the existing studies on instruction duplication all adopt the same fault model and injection method as instruction duplication is design for the fault model [2, 10–12, 15, 18]. Thereby, our fault injection simulation largely reproduces what prior work on instruction duplication has been done - we also confirm this by comparing fault injection results in Section 5.

4.4 Hardware Platform and ISA

To conduct our experiment, we use a Ubuntu 20.04 machine with an Intel Xeon processors. The machine has X86 ISA, which is the most common ISA in HPC systems, thereby it is our focus in this paper.

5 CROSS-LAYER EVALUATION

In this section, we conduct a large-scale fault injection experiment to evaluate the effectiveness of SDC detection by instruction duplication at LLVM IR and assembly instruction levels. We first describe the observations we make from the experiment results, then investigate the potential root-causes that are responsible for the deficiency of the protection.

[1]Code: https://github.com/hyfshishen/SC23-FLOWERY

5.1 Experiment Results

We present the fault injection results in Figure 2. From the experiment results, we make three major observations. They are described as follows:

Observation 1: The SDC coverages obtained in each benchmark at the same protection levels are very application-specific. For example, Pathfinder benchmark reveals a steeper curve compared with others such as BFS benchmark at both LLVM and assembly levels. At 30% protection level, Pathfinder benchmark reaches an SDC coverage of 94.26% evaluated at LLVM level, whereas in EP benchmark, it is only 63.72%. Similar observations can be made at assembly level. For instance, at 70% protection level, Susan benchmark has an SDC coverage of 85.76%, while Backprop benchmark reveals only 51.66%. The main reason is that error propagation is program-specific, hence SDCs are distributed differently across programs. The trade-offs brought by instruction duplication between SDC coverage and performance overhead vary in programs.

Observation 2: There is a clear gap between LLVM and assembly level protection in SDC coverage. In more detail, the SDC coverage measured at assembly level often falls short compared with that at LLVM level evaluation. For example, at 30% protection level, Quicksort benchmark reaches an SDC coverage of 85.02% at IR level, while it has only 74.45% at assembly level. Furthermore, at 50% protection level, Basicmath benchmark has 87.26% at IR level while it is only 53.46% at assembly level. The only exception is Susan benchmark at 30% protection level. Both IR and assembly levels show rather similar coverage of about 76.09%.

The observation we make is concerning since the assembly level evaluation represents a more realistic measurement because it is closer to the fault occurrence. That is, the SDC coverage measured and claimed at LLVM level in prior studies [10–12, 15, 18] tends to be over-optimistic – the real coverage provided by the instruction duplication at assembly level can be much lower than expected.

Observation 3: Instruction duplication technique will rarely reach 100% protection even all the instructions are protected. This is a surprising result as it shows LLVM-based instruction duplication may have intrinsic incapability to eliminate SDCs from a program even at full protection. In other words, even though all the instructions at LLVM level are duplicated, there are still assembly instructions that are skipped from the protection. For example, at assembly level, Quicksort benchmark has only 56.20% SDC coverage under full protection where it is measured as 100% at LLVM level. Similarly, BFS benchmark has only 53.33% at assembly level, suffering from the same issue. Susan benchmark, on the other hand, has 86.46%, the highest coverage with full protection among the benchmarks, but still far from achieving 100% coverage as expected from an evaluation at LLVM level. Note that at LLVM level fault injection, similar to what prior work has reported [10–12, 15, 18], the instruction duplication we use with full protection can effectively detect all the SDCs, indicating the instruction duplication mechanism implemented in the study are correct and inline with prior work.

In summary, the observations we make clearly illustrate the deficiency in the instruction duplication technique that is popularly used in existing literature. It raises concerns to HPC community as the technique is commonly applied and used in HPC systems as well as other mission-critical systems for ensuring reliability.

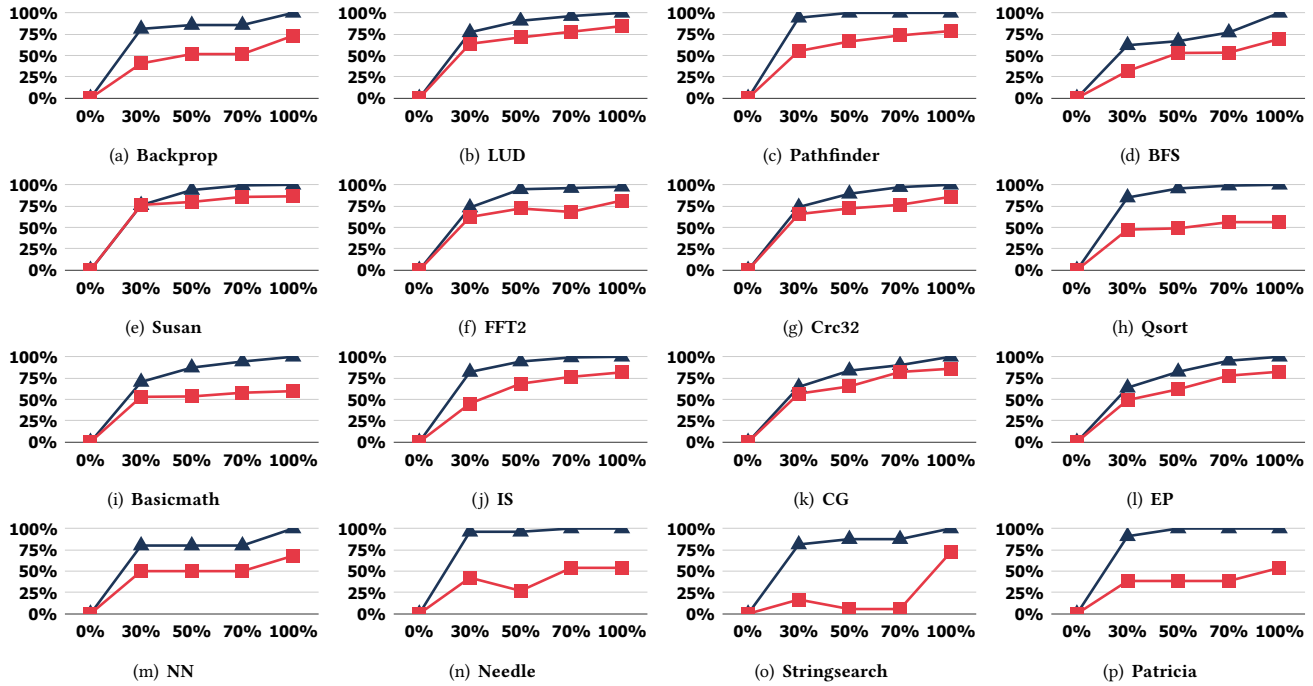

Figure 2: SDC coverage evaluation at LLVM and assembly level; X-axis denotes "protection level", Y-axis denotes "SDC coverage"; Blue line and red line represent LLVM level and assembly level evaluation respectively.

5.2 Root-Causes of Deficiencies

In this section, we investigate the root-causes that lead to the protection deficiency revealed at assembly level, and characterize the patterns of them in order to develop a technique that identifies and mitigates the deficiency.

We first go through every fault injection case which leads to the deficiency in our experiment results and analyze the patterns. As a result, we classify all the problematic cases into five categories. They are store penetration, branch penetration, comparison penetration, call penetration and mapping penetration. Among total, store penetration, comparison penetration, and branch penetration take up to about 94.50% whereas call penetration and mapping penetration cases occupy only 5.50%. Their distribution is shown in Figure 3. Next, we explain each category in details.

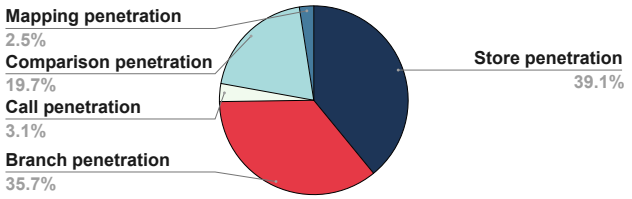

Figure 3: Percentage of Different Deficiency Cases

Store Penetration: It is due to the difference in store instructions between LLVM IR and assembly instructions when backend compilation is applied. Recall that in instruction duplication, a checker must be added before any synchronization locations such as store instructions. At LLVM level, store instructions are implemented with single IR instruction. However, at assembly level, the counterpart of the store instruction may involve the transfer of the stored value of the operand to a general-purpose register before writing to the target memory address.

```
1 %92 = load i32* @rows, align 4
2 ; During register allocation, the value of 92%
3 ; could be spilled into stack.
4 store float %92, float* %sum
```

Figure 4: Store Instruction at LLVM Level

```
1 mov  -0x40(%rbp),%rax
2 mov  %rax,-0x1c(%rbp)
```

Figure 5: Store Instruction at Assembly Level

Figure 4 shows an example of a store instruction at LLVM level and the corresponding assembly code. Note that LLVM IR introduces temporary value identifiers (e.g., %1) which are similar to registers. Ideally, all such temporary values should be mapped to registers via register allocation algorithms during compilation. However, due to the limited number of general-purpose registers, such temporary values should be spilled to the memory and reloaded back into the register when needed. In X86, the mov instruction

cannot take two memory addresses as the parameters. Therefore, moving one spilled value to a named variable *e.g.,%92* involves two moving operations, memory to register and register to memory, leading to the "non-atomic" issue of the IR store instruction.

At LLVM IR level, store instructions are not considered as a fault injection site whereas the additional instruction at assembly level becomes one. In the assembly level fault injections, we observe faults injected into the first instruction as shown in the figure 5 and lead to SDC. On average, the store penetration cases occupy a total of 39.10% across all the deficiency cases across benchmarks in our experiments. In the individual benchmarks, we observe store penetration cases vary. For example, in kNN benchmark, 15.67% cases are store penetrations while it is 56.10% in BFS benchmark, depending on whether a program is memory-bound or not.

Branch Penetration: In LLVM IR, a conditional branch instruction accepts a Boolean condition and two destination labels as its parameters. When translated to assembly code, the condition is generally already well-stored in the FLAGS register. Hence the conditional jump instruction can directly refer to the register for the condition. However, this is true only if the previous consecutive IR is an `icmp` instruction. Otherwise, the translated assembly code should set the EFLAG/RFLAG register first before executing the conditional jump. In other words, the conditional branch instruction is also "non-atomic" if the previous IR instruction is not `icmp`. Such cases widely exist in the protected IRs.

```
BB0:
br i1 %139, label %142, label %267
```

Figure 6: Branch Instruction at LLVM Level

```
test   $0x1,%al
jne    400ea7 <lud_omp+0x2f7>
jmpq   4010e7 <lud_omp+0x537>
```

Figure 7: Branch Instruction at Assembly Level

Figure 6 shows the example. The branch instruction is at the head of a basic block and does not have an `icmp` instruction as its previous consecutive IR. This leads to the code at assembly level in Figure 7 introducing a `test` instruction to set the EFLAG/RFLAG register before jumping to the destination address.

Therefore, at LLVM IR level, branch instructions are not considered as a fault injection site while it becomes one at assembly level. In the assembly level fault injections, we detect faults injected into status register after the test instruction as shown above and lead to SDC. On average, the store penetration cases occupy a total of 35.70% across all the deficiency cases in our experiments. And this category also varies a lot across different benchmarks. For example, in FFT2 benchmark, the branch penetration cases occupy only 27.30% across all its deficiency cases, but in kNN benchmark the number increases to 57.10%.

Comparison Penetration: This pattern typically concerns the situation when trying to validate the result of a comparison instruction, and it will lead to multiple consecutive `icmp` instructions at the IR level. Recall that in instruction duplication, if we are going to validate the result of an `icmp` instruction, the protected code should run the two `icmp` instructions and a third `icmp` to check whether they produce the same results. However, such code may be optimized by the compiler and invalidate the protection, *e.g.,* as a constant condition. In practice, the optimization result depends on the characteristics of the dependent instructions. LLVM has implemented dozens of powerful optimization passes to optimize IR, such as dead code elimination and constant propagation. It applies these optimization passes iteratively on a code snippet. For example, if the compiler knows two temporary values are computed based on the same expression (available expression analysis), it may eliminate one redundant expression and solve the data dependency of related instructions. This optimization result may enable further optimizations afterwards.

```
; <label>:0
  %1 = load i32* %a, align 4
  %2 = load i32* %a, align 4
  %3 = load i32* %b, align 4
  %4 = load i32* %b, align 4
  %5 = icmp slt i32 %1, %3
  %6 = icmp slt i32 %2, %4
  %check_cmp = icmp eq i1 %5, %6
  br i1 %check_cmp, label %7, label %checkBb
checkBb:
  call void @check_flag()
  br label %7
; <label>:7
  br i1 %5, label %8, label %9,
```

Figure 8: Comparison Penetration at LLVM Level

```
mov    %eax,-0xfc(%rbp)
mov    -0xb0(%rbp),%eax
mov    -0xac(%rbp),%ecx
sub    %ecx,%eax
setl   %dl
mov    $0x1,%sil
test   %sil,%sil
mov    %eax,-0x100(%rbp)
mov    %dl,-0x101(%rbp)
jne    400a07 <main+0x237>
jmpq   400a02 <main+0x232>
callq  402760 <check_flag>
mov    -0x101(%rbp),%al
test   $0x1,%al
jne    400a1a <main+0x24a>
jmpq   400aa9 <main+0x2d9>
```

Figure 9: Comparison Penetration at Assembly Level

Figure 8 shows an example. The *icmp* instruction is duplicated and checked at the end of a data dependency sequence, while the

assembly code in figure 9 shows that the function of the code is optimized and only runs comparison instruction *setl* once and replace the third *icmp* instruction with a constant condition.

Consequently, at LLVM IR level, protection techniques used for comparison instructions work but they fail at assembly level. At assembly level fault injections, we observe that faults injected into the setl instruction as shown above and lead to SDC. On average, the comparison penetration cases occupy a total of 19.70% across all the deficiency cases across benchmarks. In individual benchmarks, we see such cases vary depending on the control-flow properties of programs. In BFS benchmark, for example, the comparison penetration only occupies 6.1% of all deficiency cases, but in Needle benchmark, the number reaches 37.5%.

Call Penetration: The different ways to run a function call between LLVM IR and assembly instructions are also one of the root-causes. According to the calling convention in X86, all parameters should be placed on specific registers following an order before making a function call. Therefore, one such parameterized function call should be translated into a set of several assembly instructions, *i.e.,* register preparation and call or jump to the destination code. However, register preparation is not required in LLVM IR because a call instruction in LLVM IR can directly take function parameters. Therefore, it also suffers similar "non-atomic" issues.

```
call void @_Z3runiPPc(i32 %4, i8** %7)
```

Figure 10: Call Instruction at LLVM Level

```
mov    -0x14(%rbp),%edi
mov    -0x20(%rbp),%rsi
callq  400f70 <_Z3runiPPc>
```

Figure 11: Call Instruction at Assembly Level

Figure 10 shows an example. The `call` instruction does not have processes to transfer the function parameters to make a function call. However, at assembly level code shown in figure 11, two `mov` instructions are introduced before making this function call which becomes fault injection sites.

As a consequence, at LLVM IR level, call instructions are not considered as a fault injection site whereas it becomes one at assembly level. In the assembly level fault injections, we observe faults injected into `mov` instructions as shown above and lead to SDC. On average, the call penetration cases are a total of 3.10% among all. In Needle benchmark, the call penetration share of 12.50% of the penetration cases across all the deficiency cases but in BFS benchmark it only has 2.43%. So this prevalence of the category also varies from program to program.

Mapping Penetration: We attribute the last root-cause to the mapping problem that certain instructions and operations may not be mappable between the IR level and the assembly level. For example, when using a callee-saved register (e.g., rbp), the callee should back up its existing value on the stack via the push command and restore its value via pop. Such semantics do not exist in the IR code. Figure 12 shows an example. When running a function call, the program first *push* the target value into the stack, and execute all the instructions inside the function. Then, before *retq* is executed, the target value will be popped out. These two instructions do not have corresponding IR instructions at the LLVM level.

```
push   %rbp
... ; function body
pop    %rbp
retq
```

Figure 12: Mapping Penetration at Assembly Level

As a result, in the assembly level fault injection, we observe faults injected into the instructions that are not mappable between two levels and cause SDCs. However, the mapping penetration cases only have 2.50% among all the deficiency cases on average. The highest mapping penetration rate is 9.1% in FFT2 benchmark, in contrast, it is 0% in LUD benchmark.

5.3 Summary of Root-Causes

The key results are summarized as follows: (1) Some instructions do not have a fault injection site at the IR level, but when converted to the assembly level, there are unprotected areas that can be penetrated. These are store penetration, branch penetration and call penetration, they occupy 39.1%, 35.7%, and 3.1% of the total number of penetrations, respectively. (2) Some cases are due to the nullification of the original IR-level protection mechanism by the mapping process of the two layers. For example, comparison penetration and partial mapping penetration account for 19.7% and 2.5% of the total number of penetrations, respectively. Based on the results, it can be observed that the existing instruction duplication technique implemented at the IR level lacks the ability to provide adequate protection against fault occurred at the assembly level. In order to address this issue, we propose various mitigating methods at the IR level to enhance its protection in Section 6.

6 OUR SOLUTION

In this section, we propose a set of compiler patches that fix the deficiency of the protection without incurring much performance overhead. Our technique is fully automated, and transparent to users, bridging the gaps between LLVM and assembly level coverage in the protection provided by instruction duplication. The proposed technique, named Flowery, consists of three parts, each of which is described below.

6.1 Eager Mode of Store

As mentioned earlier, the reason for store penetration is the scarcity of general-purpose registers, which results in temporary values being stored in memory and later retrieved back into the registers during the execution of store instructions at the assembly level. Existing IR-based instruction duplication techniques employ a lazy mode, *i.e.,* a value must be checked before being stored. Such lazy mode is especially vulnerable to the register spilling issue. Note that when a checker is added, the branch instruction in the checker will separate the following synchronization point (in this case, the store

instruction) into an individual basic block. Since the temporary value to be stored is not immediately used, it is prone to be spilled.

To overcome the spilling issue, we propose to employ an eager mode for store, *i.e.,* store before being checked. Figure 13 illustrates the idea. As seen, if we move the problematic store instruction to the location before the checker and connect it to the end of one of the computation copies in the instruction duplication, the target store instruction will be used right before itself (e.g., A instruction in the example) within its current basic block thus move the temporary value into a register. This way we avoid introducing any additional computations. However, this may come with a problem that we already stored error data before it has been detected. However, if the error data has been detected, we don't need to keep running this program, so there is no extra loss.

Therefore, by swapping all the problematic store instructions in respect to their checkers as mentioned above, the store instructions will not bring any additional assembly instructions after the back-end compilation. We expect to eliminate the deficiency caused by the store penetration in the protection. The proposed method can be implemented at LLVM IR level as a compiler pass after instruction duplication and thereby mitigate issue at assembly level.

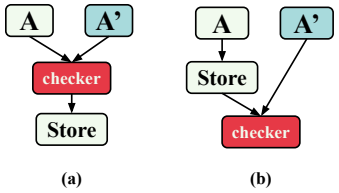

Figure 13: Example of Eager Mode of Store. (a) Original Checker Position. (b) Eager Mode Checker Position.

6.2 Postponed Branch Condition Check

Recall that one of the root-causes is the difference in branch instructions between LLVM IR and assembly instructions. At the IR level, the branch instruction directly changes the address of the program at the end of each run and has no return value. At the assembly level, if the branch instruction's previous consecutive IR is not an *icmp* instruction, it will set the FLAGS register by itself thus introducing a fault injection site. This, in particular, causes problems in instruction duplication because when a checker is added, the branch instruction in the checker will separate the following synchronization point (in this case, the branch instruction) into an individual basic block without an *icmp* instruction before it, thereby causing the issue.

Therefore, in order to protect branch instructions, we have developed a patch in Flowery, which can provide effective protection for branch instructions at low overhead. Since the branch instruction cannot be duplicated, we cannot directly determine whether a bit flip has occurred in the status register or not, but we can place the error detection after the execution of the branch instruction. In Flowery, we store the value of the branch instruction in a global variable before the branch instruction is executed, and after executing the branch instruction we insert two checkers in the two possible destinations of the branch instruction separately. Inside each checker, we will detect if the global variable value matches the destination. If the value of the global variable does not correspond to the basic block that is jumped, the program will detect the error for protection purposes. Figure 14 illustrates how this patch works to a branch instruction. By using this patch, we expect to solve the deficiency caused by the branch penetration in the protection. The method we showed above can be implemented at LLVM IR level as a compiler pass after instruction duplication and to solve the problem in assembly level.

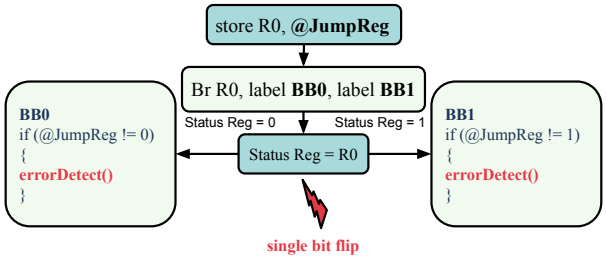

Figure 14: Example of Postponing Branch Condition Check

6.3 Anti-Comparison Duplication Optimization

As we analyzed in section 5, one of the root-causes is the compiler optimization when comparison instruction is located at the end of a data dependency sequence. This almost happens in every comparison instruction, meaning that the instruction duplication technique used for comparison instruction fails.

To solve this problem, one possible way is to avoid the optimization that targets the comparison instructions. To do this, our idea is to move the `cmp` instructions into another new basic block and complicate the optimization problem. As shown in Figure 15, A and B are two operands of the `cmp` instruction. If we directly duplicate the instruction sequence of def(A)-def(B)-cmp(A,B), it is not difficult for the compiler to recognize def(A) is equivalent to def(A'), and cmp(A,B) is equivalent to cmp(A',B'). Our anti-optimization first separates the `cmp` instruction and the definition of A and B into different basic blocks. Furthermore, we add another conditional check before reaching the `cmp` block and thus complicate the reachability analysis from the def(A)-def(B) to cmp(A,B). Figure 15 presents the idea. As seen, if we force a comparison instruction as a single independent dataflow, it will be protected and checked individually. By making all the comparison instructions independently check and duplicated, we expect to avoid the optimization, thus eliminating the deficiency caused by the comparison penetration in the protection. The proposed method can also be implemented at LLVM IR level as a compiler pass after instruction duplication and thereby mitigate the issue at assembly level.

For the rest of deficiency cases (call and mapping penetrations), we do not come up with LLVM-level solutions. However, they can be mitigated at assembly level if the corresponding compiler for transformation and analysis is available. With that said, the total covered cases (store, comparison and branch penetrations) by the proposed patches above already reach 94.4% among the total reported deficiency cases (Figure 3).

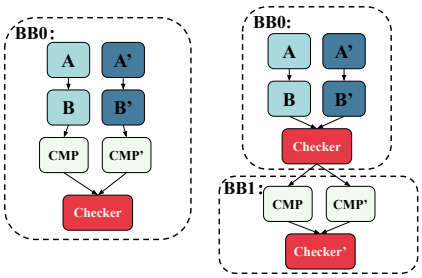

Figure 15: Example of Anti-Comparison Duplication Optimization. (a) Original `cmp` position. (b) After Avoiding Optimization `cmp` Position.

6.4 Workflow of Flowery

Figure 16 shows the workflow of Flowery. All the patches in Flowery are implemented as a set of LLVM compiler passes. During the compile time, the user runs existing LLVM-based instruction duplication as normal, then Flowery is applied after the instruction duplication to mitigate the protection deficiency. Finally, the protected binary will be generated for execution. The entire process is fully automated and transparent to the user.

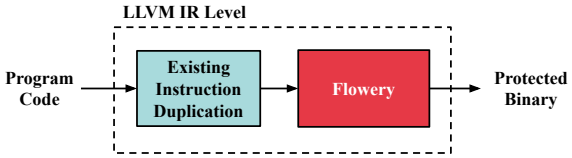

Figure 16: Workflow of Flowery.

7 EVALUATION OF OUR TECHNIQUE

In this section, we evaluate our technique in protecting programs from SDCs. The evaluation of Flowery is at assembly level. We compare the results with original instruction duplication technique measured at LLVM and assembly level respectively – we use it as our baseline. There are three metrics we consider in our evaluation: (1) SDC coverage provided, (2) runtime performance overhead, and (3) time taken to execute our technique. In the explanation, we use ID-IR and ID-Assembly to denote the original instruction duplication evaluated at LLVM level and assembly level respectively.

7.1 SDC Coverage

Figure 17 demonstrates the SDC coverage provided by Flowery measured at assembly level as well as the original instruction duplication measured at LLVM and assembly levels. We make three main observations from the results.

Our first observation is that the coverage provided by Flowery is always higher than that by ID-Assembly. Note that both are evaluated at assembly level. For example, in Susan, at 30%, 50%, and 70% protection levels, we observe that Flowery provides at least 92.01%, 95.49%, and 98.26% SDC coverage respectively, whereas they are only 76.39%, 79.86%, and 85.76% in ID-Assembly. This shows that our proposed mitigate technique repairs the protection deficiency of instruction duplication at assembly level, and only improves the coverage of original instruction duplication technique.

In individual cases, Flowery provides relatively higher coverage in some benchmarks such as Crc32 and EP compared with others such as Stringsearch and Patricia. This is because the proportion of penetrations is application-specific. In Crc32 and EP benchmarks, their penetration is mainly occupied by branch, comparison, and store penetrations, and the proportion of these three penetrations is 90.8% in Crc32 and 94.6% in EP. However, in Stringsearch and Patricia, the function call numbers and instructions with mapping issues are very high, the proportion of these two even reaches 23% in Patricia, which leads to Flowery does not perform as well as it on other benchmarks.

On the other hand, we observe that the coverage provided by Flowery is much closer to that by ID-IR. Unlike ID-Assembly, Flowery closes the gap between LLVM and assembly level evaluation in most of the benchmarks. This shows that developers can trust their protection estimated when using instruction duplication technique, and expect the coverage to be similar to what they aim at.

We notice that the gap is relatively wider between Flowery and ID-IR in Stringsearch and Patricia benchmarks. We speculate that this is because these two benchmarks have a great number of function calls and mapping penetrations that make the gap not fixable by Flowery in each protection level.

Finally, we observe that Flowery provides much higher coverage at full protection. Recall that ID-Assembly provides an average coverage of only 76.74% at full protection. In contrast, with Flowery, the average coverage reaches 93.72%. From individual benchmark perspective, the highest coverage Flowery provides is 99.31% in Susan whereas it is 86.46% by ID-Assembly. The worst case in Flowery is Basicmath benchmark as it provides only 82.3% coverage, but still much higher than ID-Assembly (59.58%).

The reason Flowery cannot reach 100% protection at full protection is because Flowery aims to mitigate the gap between LLVM level and Assembly level with low overhead, and Flowery is implemented in LLVM level. Since some of the penetrations can hardly be fixed at LLVM level e.g., call and mapping penetrations, we can not mitigate all of them with a simple and low-cost technique.

7.2 Performance Overhead

We evaluate the runtime performance overhead incurred by deploying Flowery. Since Flowery is on top of original instruction duplication technique, we are interested in understanding the additional overhead our technique brings to the instruction duplication. To gauge this, we measure the runtime overheads (in wall-clock time) incurred by instruction duplication before and after applying Flowery. As measured, the additional overheads by Flowery are 1.93%, 1.63%, 3.72%, 3.74% on average at 30%, 50%, 70%, and 100% protection levels respectively. Each time measurement is taken as an average of three executions of a program to minimize the noise in the measurement. The results indicate that our technique incurs very runtime low performance overhead compared with original instruction duplication technique, showing that Flowery is practical.

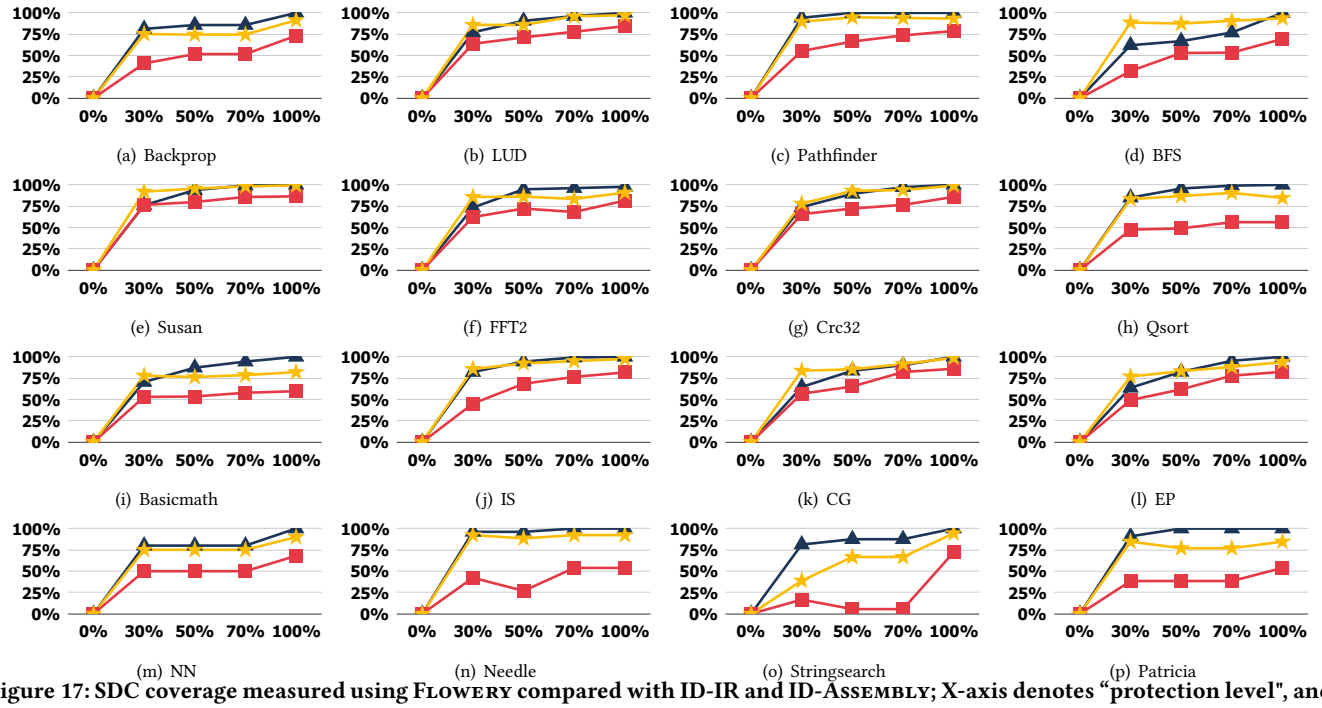

Figure 17: SDC coverage measured using Flowery compared with ID-IR and ID-Assembly; X-axis denotes "protection level", and Y-axis denotes "SDC coverage" measured; Blue line represents ID-IR, red line represents ID-Assembly, yellow line represents Flowery measured at assembly level.

7.3 Execution Time

We now report the time taken to execute Flowery. After applying instruction duplication technique to a program, Flowery can be applied as an LLVM compiler transformation pass - all these happen at compile-time before the deployment of the protected application. Our measurement shows that Flowery takes only 0.12 seconds on average for each benchmark, with a maximum of 0.51 seconds (CG benchmark) and a minimum of 0.08 seconds (Quicksort benchmark). We find the time taken depends on the number of static instructions in a program, as Flowery needs to linearly scan the code and generate transformations. For example, in CG benchmark, the number of static instructions is 2290 while it is 92 in Quicksort benchmark. Overall, Flowery takes almost negotiable amount of time at compile-time.

8 DISCUSSION

Implication to Existing Instruction Duplication Techniques As we show in Section 5, existing instruction duplication technique is over-optimistic on SDC coverage, and often suffers from low SDC coverage at assembly level. In Section 7, we show that Flowery mitigates the deficiency. After applying Flowery on top of existing instruction duplication technique, the protection shows similar SDC coverage measured at both LLVM and assembly levels (Figure 17). The entire process of Flowery is automated and transparent to the user and incurs only minimal performance overhead. With Flowery, HPC developers can now confidently apply LLVM-based instruction duplication techniques that are popular in HPC, and protect their applications from SDCs.

Other Implementation Options We implement Flowery at LLVM level for patching protection deficiency. One of the reasons is that LLVM is largely supported as open-source tools across research communities and industries. Leveraging IR-based infrastructure allows developers to easily pinpoint their reliability analysis to the protection in given programs. With that said, it is also possible to implement the patches at assembly level. We do not choose this way since one rarely has a convenient backend compiler to do so.

ISA As mentioned, we focus on X86 ISA at assembly level because it is the most commonly seen ISA in HPC systems. Hence, our results may be ISA-specific. With that said, we believe that the conjectures we report should also be insightful to other ISA platforms as we explore both the common background of ISAs and the IR issues. For example, RISC-V and ARM may both suffer from store penetration issues because it also has limited registers; Comparison penetration cases may also be observed as well because the root-cause lies in IR optimization and is irrelevant to ISAs.

9 RELATED WORK

Instruction Duplication Techniques Instruction duplication has been proposed for more than two decades. [24, 26, 27] Soon after that, it becomes a popular technique for detecting soft errors at the program level [11, 12, 18, 21, 33] Laguna et al. [12] utilized machine learning techniques to selectively duplicate the most vulnerable instructions using instruction duplication technique, in order to detect soft errors at a low cost. Li et al. [15] proposed an analytical model to identify the most vulnerable instructions for protection and used instruction duplication to mitigate SDCs. Others have

focused on exploring SDC coverage variations in instruction duplication technique [9, 10, 14, 33]. These studies do not investigate cross-layer protection effectiveness of instruction duplication.

Fault Injection Study For over 50 years, fault injection techniques have been proposed as a crucial element in assessing software protection. Numerous researchers have developed diverse fault injection tools at various levels to replicate and simulate fault occurrence [20, 29, 32] Wei et al. [32] proposed LLFI, a configurable fault injector for the LLVM IR level, and compare its performance with PINFI fault injector. NFTAPE [29] is a fault injection tool at the assembly level for emulating hardware faults. NFTAPE utilizes machine code-based break-point injection, which permits users to create their own injectors operating at the source code level. The work demonstrated the usefulness of the injector for conducting resilience analysis of programs. G-SWiFT[20] aims to simulate software defects by detecting clusters of assembly code instructions that correspond to high-level software constructs, and then introducing faults in these clusters to emulate software deficiency at the machine code level. None of them focuses the protection coverage of instruction duplication techniques.

Soft Error Cross-Layer Evaluation Vallero et al. [31] proposed a scalable, cross-layer method and a supporting suite of tools for accurate and fast estimation of reliability. Ebrahimi et al. [3] explained the significance of cross-layer soft error modeling and mitigation, and demonstrates how it can lead to a low-cost design for soft error reliability by combining existing soft error modeling techniques. The most related work is [2]. The report first mentioned that existing instruction duplication technique may suffer from protection deficiency. However, neither analysis nor solutions were provided in the report. In contrast, our work quantitatively shows the protection deficiency and analyzes the root-causes of it. Moreover, we propose solutions, Flowery, and demonstrate the effectiveness of the technique.

10 CONCLUSION

In conclusion, we investigate the effectiveness of LLVM-based instruction duplication technique at assembly level, and discover the root-causes of the inconsistency between LLVM level and assembly level protection. We observe that existing instruction duplication technique often suffer from low SDC coverage if measured at assembly level. To mitigate the issues, we propose Flowery, a set of compiler passes that mitigate the protection deficiency on top of existing instruction duplication technique. Our evaluation shows that Flowery is effective in mitigating protection deficiency in instruction duplication.

REFERENCES

[1] Rizwan A. Ashraf, Roberto Gioiosa, Gokcen Kestor, Ronald F. DeMara, Chen-Yong Cher, and Pradip Bose. 2015. Understanding the propagation of transient errors in HPC applications. In *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12. https://doi.org/10.1145/2807591.2807670

[2] Chun-Kai Chang, Guanpeng Li, and Mattan Erez. 2019. 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)*. 41–49. https://doi.org/10.1109/FTXS49593.2019.00010

[3] Mojtaba Ebrahimi and Mehdi B. Tahoori. 2016. Invited - Cross-Layer Approaches for Soft Error Modeling and Mitigation *(DAC '16)*. Association for Computing Machinery, New York, NY, USA, Article 32, 6 pages. https://doi.org/10.1145/2897937.2905007

[4] Shuguang Feng, Shantanu Gupta, Amin Ansari, and Scott Mahlke. 2010. Shoestring: Probabilistic Soft Error Reliability on the Cheap. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Pittsburgh, Pennsylvania, USA) *(ASPLOS XV)*. Association for Computing Machinery, New York, NY, USA, 385–396. https://doi.org/10.1145/1736020.1736063

[5] Giorgis Georgakoudis, Ignacio Laguna, Dimitrios S. Nikolopoulos, and Martin Schulz. 2017. REFINE: Realistic Fault Injection via Compiler-based Instrumentation for Accuracy, Portability and Speed. In *SC17: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[6] R. W. Hamming. 1950. Error detecting and error correcting codes. *The Bell System Technical Journal* 29, 2 (1950), 147–160. https://doi.org/10.1002/j.1538-7305.1950.tb00463.x

[7] Siva Kumar Sastry Hari, Sarita V. Adve, and Helia Naeimi. 2012. Low-cost program-level detectors for reducing silent data corruptions. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*. 1–12. https://doi.org/10.1109/DSN.2012.6263960

[8] Jie Hu, Feihui Li, Vijay Degalahal, Mahmut Kandemir, Vijaykrishnan Narayanan, and Mary Irwin. 2005. Compiler-Directed Instruction Duplication for Soft Error Detection. 1056–1057. https://doi.org/10.1109/DATE.2005.98

[9] Yafan Huang, Shengjian Guo, Sheng Di, Guanpeng Li, and Franck Cappello. 2022. Hardening selective protection across multiple program inputs for HPC applications. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 437–438.

[10] Yafan Huang, Shengjian Guo, Sheng Di, Guanpeng Li, and Franck Cappello. 2022. Mitigating Silent Data Corruptions in HPC Applications across Multiple Program Inputs. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14. https://doi.org/10.1109/SC41404.2022.00022

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

[12] Ignacio Laguna, Martin Schulz, David F Richards, Jon Calhoun, and Luke Olson. 2016. Ipas: Intelligent protection against silent output corruption in scientific applications. In *2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 227–238.

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

[14] Guanpeng Li and Karthik Pattabiraman. 2018. Modeling Input-Dependent Error Propagation in Programs. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 279–290. https://doi.org/10.1109/DSN.2018.00038

[15] Guanpeng Li, Karthik Pattabiraman, Siva Kumar Sastry Hari, Michael Sullivan, and Timothy Tsai. 2018. Modeling Soft-Error Propagation in Programs. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 27–38. https://doi.org/10.1109/DSN.2018.00016

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

[17] Qining Lu, Guanpeng Li, Karthik Pattabiraman, Meeta S. Gupta, and Jude A. Rivers. 2017. Configurable Detection of SDC-Causing Errors in Programs. 16, 3, Article 88 (mar 2017), 25 pages. https://doi.org/10.1145/3014586

[18] Qining Lu, Karthik Pattabiraman, Meeta S Gupta, and Jude A Rivers. 2014. 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*. 1–10.

[19] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. *SIGPLAN Not.* 40, 6 (2005), 190–200.

[20] H. Madeira, D. Costa, and M. Vieira. 2000. On the emulation of software faults by software fault injection. In *Proceeding International Conference on Dependable Systems and Networks. DSN 2000*. 417–426. https://doi.org/10.1109/ICDSN.2000.857571

[21] Abdulrahman Mahmoud, Siva Kumar Sastry Hari, Michael B Sullivan, Timothy Tsai, and Stephen W Keckler. 2018. Optimizing software-directed instruction replication for gpu error detection. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 842–854.

[22] Shubhendu S Mukherjee, Christopher Weaver, Joel Emer, Steven K Reinhardt, and Todd Austin. 2003. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.* IEEE, 29–40.

[23] Bin Nie, Ji Xue, Saurabh Gupta, Christian Engelmann, Evgenia Smirni, and Devesh Tiwari. 2017. Characterizing temperature, power, and soft-error behaviors in data center systems: Insights, challenges, and opportunities. In *2017 IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and*

Telecommunication Systems (MASCOTS). IEEE, 22–31.

[24] Nahmsuk Oh, Philip P Shirvani, and Edward J McCluskey. 2002. Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability* 51, 1 (2002), 63–75.

[25] 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 *SC21: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14. https://doi.org/10.1145/3458817.3476147

[26] Philip P Shirvani, Namsuk Oh, Edward J Mccluskey, DL Wood, Michael N Lovellette, and KS Wood. 2000. 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)*.

[27] Philip P Shirvani, Nirmal Saxena, Nahmsuk Oh, Subhasish Mitra, Shu-Yi Yu, Wei-Je Huang, Santiago Fernandez-Gomez, Nur A Touba, and Edward J McCluskey. 1999. 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.

[28] Premkishore Shivakumar, Michael Kistler, Stephen W Keckler, Doug Burger, and Lorenzo Alvisi. 2002. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proceedings International Conference on Dependable Systems and Networks*. IEEE, 389–398.

[29] D.T. Stott, B. Floering, D. Burke, Z. Kalbarczpk, and R.K. Iyer. 2000. NFTAPE: a framework for assessing dependability in distributed systems with lightweight fault injectors. In *Proceedings IEEE International Computer Performance and Dependability Symposium. IPDS 2000*. 91–100. https://doi.org/10.1109/IPDS.2000.839467

[30] Devesh Tiwari, Saurabh Gupta, James Rogers, Don Maxwell, Paolo Rech, Sudharshan Vazhkudai, Daniel Oliveira, Dave Londo, Nathan DeBardeleben, Philippe Navaux, Luigi Carro, and Arthur Bland. 2015. Understanding GPU errors on large-scale HPC systems and the implications for system design and operation. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 331–342. https://doi.org/10.1109/HPCA.2015.7056044

[31] A. Vallero, A. Savino, G. Politano, S. Di Carlo, A. Chatzidimitriou, S. Tselonis, M. Kaliorakis, D. Gizopoulos, M. Riera, R. Canal, A. Gonzalez, M. Kooli, A. Bosio, and G. Di Natale. 2016. Cross-layer system reliability assessment framework for hardware faults. In *2016 IEEE International Test Conference (ITC)*. 1–10. https://doi.org/10.1109/TEST.2016.7805863

[32] Jiesheng Wei, Anna Thomas, Guanpeng Li, and Karthik Pattabiraman. 2014. Quantifying the Accuracy of High-Level Fault Injection Techniques for Hardware Faults. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 375–382. https://doi.org/10.1109/DSN.2014.2

[33] Lishan Yang, Bin Nie, Adwait Jog, and Evgenia Smirni. 2021. Enabling software resilience in gpgpu applications via partial thread protection. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1248–1259.

Appendix: Artifact Description/Artifact Evaluation

ARTIFACT DOI

https://github.com/hyfshishen/SC23-FLOWERY

ARTIFACT IDENTIFICATION

Instruction duplication is a widely used software-based protection technique against silent data corruption (SDC), which is regarded one severe failure outcome of soft errors in modern High-Performance Computing (HPC) systems. However, existing instruction duplication techniques are mostly implemented at the LLVM level and are demonstrated not fully effective at the assembly code level. To tackle this issue, this work proposes Flowery, which is an LLVM IR-level patch for enhancing LLVM IR-level instruction duplication against cross-layer (i.e. assembly-level) soft errors.

We run the Flowery workflow on a Linux server, which equips two Intel Xeon CPUs (32-C, 64-T) and 64GB RAM. The experiments in this work can be divided into three parts: (1) All the 16 benchmarks are compiled to both LLVM IR (readable format) and assembly via clang/clang++ 3.4 compilers, (2) IR level and assembly level fault injection experiments to demonstrate the problem. (3) Flowery is conducted to mitigate the inconsistency of instruction duplication technique via LLVM code transformation at the IR level, and (4) IR level and assembly level fault injection experiments on 16 widely-used HPC benchmarks are conducted to verify the effectiveness of protection by Flowery.

REPRODUCIBILITY OF EXPERIMENTS

(1) Artifact name: Flowery Workflow
(2) Relevant hardware details: The Flowery workflow is executed on a server with Linux OS. This server is powered by two Intel Xeon CPUs (32-C, 64-T), and it is equipped with 64GB RAM. Other software: LLVM Infrastructure (version 3.4), LLVM level fault injector (LLFI), Intel PIN tool v-3.5 - assembly level fault injector(PINFI), Python 3.7.2.
(3) Benchmarks: Rodinia suite: kNN, Backprop, BFS, Pathfinder, LUD, and Needle. NPB suite: CG, EP, and IS. MiBench suite: Basicmath, Stringsearch, Quicksort, Susan, CRC32, FFT, and Patricia.
(4) Experiment workflow: The experiment workflow consists of four steps: (1) All the 16 benchmarks are compiled to both LLVM IR (readable format) and assembly via clang/clang++ 3+ compilers, (2) IR level and assembly level fault injection experiments to demonstrate the problem. (3) Flowery is conducted to mitigate the inconsistency of instruction duplication technique via LLVM code transformation at the IR level, and (4) IR level and assembly level fault injection experiments on 16 benchmarks are conducted to verify the effectiveness of protection by Flowery.
(5) Estimation execution time: Around several hours. This work requires fault injection experiments to validate the results, which takes lots of program executions.
(6) Expected results and evaluation: Flowery should satisfy: mitigating the inconsistency between IR level and assembly level fault injection, small runtime performance overhead compared with original instruction duplication technique, and tolerant time taken to execute Flowery workflow.
(7) Results between experiment workflow and paper: The results should be consistent with the data that is provided in the Evaluation section.

ARTIFACT DEPENDENCIES REQUIREMENTS

In this section, we present the *hardware dependencies* and provide *step-by-step instructions for Flowery environment configuration.* For setting up executing environments for Flowery workflow, we provide two methods: one is using the Docker image we prepared, and the other one is manually setting up environments on your local Ubuntu 16.04 machines. **We highly recommend you use the Docker image we prepared (can be found at DockerHub)**, because all the dependencies are already configured on that. Very easy to use. Also, please do not use “zsh" for running the scripts, since it may automatically shutdown fault injection campaigns in some cases.

0.1 Hardware Dependencies

Before configuring the environments, please **make sure you are using Intel CPU** and **your CPU supports at least 20 threads** (checking by "nproc" or hardware specifications). There are two reasons: (1) Our assembly-level fault injection tool is tied to Intel PIN, which is a dynamic instrumentation tool for assembly code for Intel CPU; and (2) fault injection experiments are usually time-consuming and should be accelerated in parallel.

0.2 Configuring Environments with Docker

To install Docker on your local Linux machine, you can follow the steps in this LINK. You may also want to use Docker without sudo access (like I did in the following commands), please check this LINK. The bash scripts for Docker image installation and running can be found below:

(1) Download our prepared image from Docker Hub:
docker pull hyfshishen/sc23-flowery-env
(2) Execute this image to a running container:
docker run -it hyfshishen/sc23-flowery-env /bin/bash

After you run this image as a running container, and change the directory to “/root", you can find LLFI, PINFI, and other dependencies have already been installed. And you are ready to run Flowery workflow.

0.3 Configuring Environments Mannully

To configure environments manually, there are several software dependencies, which are listed below:

- Ubuntu 16.04
- Python 2.7 and 3.5 (with PyYaml 4.2b4 installed)
- CMake (minimum v2.8)
- tcsh

- LLFI (LLVM-level Fault Injection Tool)
- PINFI (Assembly-level Fault Injection Tool)

Among those dependencies, LLFI and PINFI are the two most important tools to execute Flowery workflow.

LLFI, which contains LLVM 3.4 and its related software (such as Clang), is one key dependency for LLVM IR-level fault injection in Flowery workflow. THe commands for installing LLFI can be checked below.

(1) Download LLFI source code:
git clone https://github.com/DependableSystemsLab/LLFI.git
(2) Quick install LLFI and LLVM 3.4:
cd $PATH-TO-LLFI$/installer/
python3 InstallLLFI.py –noGUI
(3) Add LLVM/LLFI executable binary path to local environments:
echo "export PATH=$PATH:$PATH-TO-LLFI$/installer/llvm/bin/" » /.bashrc
echo "export PATH=$PATH:$PATH-TO-LLFI$/installer/llfi/bin/" » /.bashrc

PINFI, which builds on Intel PIN tool v-3.5, is made by Dependable Systems Lab@UBC. It can perform assembly-level fault injection in Flowery workflow. Here we use a PINFI version that has optimized Python scripts, and the installation commands can be checked below.

(1) Download forked PINFI source code along with PIN V3.5 toolkit:
git clone https://github.com/hyfshishen/pinfi.git
(2) Luckily, no building process is needed for PINFI. So skip this process.
(3) Add PIN executable binary path to your local environment:
echo "export PATH=$PATH:$PATH-TO-LLFI$" » /.bashrc

Then, you are ready to run the Flowery workflow.

ARTIFACT INSTALLATION DEPLOYMENT PROCESS

This section contains benchmark information (used in this work) and detailed steps to execute Flowery workflow. Note that after the environment is built, no building process for this project is needed, since the LLVM transformation passes are already compiled to ".so" files and ready to use.

0.4 Benchmark Information

This information can be found in the Table attached in the last part of this AD. The information it contains is shown as format **Name (in Paper) – Name (in Code)**:

- Backprop – backprop
- BFS – bfs
- Pathfinder – pathfinder
- LUD – lu
- Needle – needle
- kNN – nn
- EP – EP
- CG – CG
- IS – IS
- FFT2 – fft2
- Quicksort – qsort
- Basicmath – basicmath
- Susan – susan
- CRC32 – crc32
- Stringsearch – stringsearch
- Patricia – patricia

Name (in Paper): means the benchmark name shown in our SC'23 paper, whereas **Name (in Code)**: means the benchmark name used in code. To execute Flowery, please make sure you use **Name (in Code)** while running the code.

0.5 To Run Flowery Workflow

We provide step by step commands for running FLOWERY and result-generation scripts. There are two key evaluations in this paper:

(1) Figure 2 in Section V: This is showing the inconsistency of instruction duplication technique in IR level and assembly level.
(2) Figure 17 in Section VII: This is showing FLOWERY in mitigating the inconsistency of instruction duplication technique in IR level and assembly level.

The time-cost is different across different benchmarks, which can be found in Table above. Besides, please use Name in Code (also shown in above Table) to execute each benchmark. Before you start, please download the FLOWERY code by following commands:
git clone https://github.com/hyfshishen/SC23-FLOWERY.git

Section V - Preliminary Study: Results in this section refers to Figure 2 in paper.

(1) Change directory to the folder related to Section V:
cd AD-AE-evaluation/s5-cross-layer-evaluation
(2) Execute scripts for running and results colelction. This step contains massive fault-injection experiments and may take you some time. After that the results will be printed automatically:
python run.py BENCHMARK-NAME-IN-CODE

Section VII - Evaluation: Results in this section refers to Figure 17 in paper. We here use pathfinder as example, other benchmarks are totally the same.

(1) Change directory to the folder related to Section VII.:
cd AD-AE-evaluation/s7-evaluation
(2) Random fault injection experiments for evaluating the FLOWERY in assembly level. This step contains massive fault-injection experiments and may take you some time. After that the results will be printed automatically.:
python run.py pathfinder

Note that only FLOWERY results in Figure 17 will be printed here, the Baseline (i.e. original instruction duplication method) results can be found in Figure 2 (Section V).