Path Sensitive Signatures for Control Flow Error Detection

Ze Zhang
University of Michigan
Ann Arbor, Michigan, USA
zezhang@umich.edu

Sunghyun Park
University of Michigan
Ann Arbor, Michigan, USA
sunggg@umich.edu

Scott Mahlke
University of Michigan
Ann Arbor, Michigan, USA
mahlke@umich.edu

Abstract

Transistors' performance has been improving by shrinking feature sizes, lowering voltage levels, and reducing noise margins. However, these changes also make transistors more vulnerable and susceptible to transient faults. As a result, transient fault protection has become a crucial aspect of designing reliable systems. According to previous research, it is about 2.5x harder to mask control flow errors than data flow errors, making control flow protection critical. In this paper, we present Path Sensitive Signatures (PaSS), a low overhead and high fault coverage software method to detect illegal control flows. PaSS targets off-the-shelf embedded systems and combines two different methods to detect control flow errors that incorrectly jump to both nearby and faraway locations. In addition, it provides a lightweight technique to protect inter-procedural control flow transfers including calls and returns. PaSS is evaluated on the SPEC2006 benchmarks. The experimental results demonstrate that with the same level of fault coverage, PaSS only incurs 15.5% average performance overhead compared to 64.7% overhead incurred by the traditional signature-based technique. PaSS can also further extend fault coverage by providing inter-procedural protection at an additional 3.6% performance penalty.

CCS Concepts: • **Computer systems organization → Reliability**.

Keywords: reliability, compiler, control flow error

ACM Reference Format:
Ze Zhang, Sunghyun Park, and Scott Mahlke. 2020. Path Sensitive Signatures for Control Flow Error Detection. In *Proceedings of the 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '20), June 16, 2020, London, United Kingdom.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3372799.3394360

1 Introduction

Transient faults, also called Single Event Upsets (SEUs) or soft errors, are caused by environmental effects such as electromagnetic interference (EMI), power fluctuations, and high energy particle strikes. With advancements of semiconductor technology, transistor size has reduced exponentially in the past decades. Aggressive voltage scaling and noise margin reduction have also emerged as effective methods to improve energy efficiency on microprocessors. However, this combination of techniques substantially weakens the architectural reliability, causing transient faults to happen more easily and frequently than ever before.

Transient faults do not cause permanent damage to the hardware, but they may silently corrupt an application's correctness during run time or even crash the whole system. For example, HP [38] stated that the frequent failures of its 2048-CPU system deployed at the Los Alamos National Laboratory were caused by high-energy cosmic rays. A study [12] even showed that the BlueGene/L machine installed in Lawrence Livermore National Labs suffered from soft errors in every four hours. Given the fact that the estimated reliability per bit drops roughly 8% per generation of processors [11], there is an urgent need to provide transient fault protection schemes on both current and future systems.

Transient fault detection techniques rely on different forms of redundant checking, either in hardware or software. Typical hardware solutions include DMR (dual-modular redundancy), TMR (triple-modular redundancy), and watchdog processors [34]. IBM Z-Series servers [6], HP NonStop systems [7], and Boeing 777 airplanes [59] are examples of systems incorporating hardware-based transient fault detection and recovery mechanisms. Even if these hardware-based solutions do not have a severe effect on performance, they introduce unavoidable area and energy costs. Therefore, they cannot be directly applied to commodity embedded systems.

Software-based redundant checking is more appealing for transient fault detection since it is free of production costs and offers more flexibility. Prior works [24, 56] report that the masking rate of control flow errors is significantly less than that of data flow errors. Consequently, securing control flows becomes a crucial aspect of transient fault protection. Traditional software methods [42, 55] perform verification on every *branch* instruction to ensure correctness. Although detailed-checking methods provide high fault coverage, a large number of validating instructions are injected into programs, resulting in moderate to large performance overhead. More recent studies [24, 62] try to reduce this validation overhead by injecting fewer instructions, but they sacrifice fault coverage to different extents due to their heuristic approaches. Given that transient fault protections aim to detect as many faults as possible, trading fault coverage for better performance has inherent weaknesses.

In this work, we focus on rethinking the *coverage versus overhead* trade-off to develop a new validation method that provides high fault coverage while keeping low performance overhead at the same time. With this motivation, we propose *Path Sensitive Signatures for Control Flow Error Detection* (PaSS), an efficient software method to detect illegal control flows caused by transient faults. PaSS creatively combines two different checking methods to detect control flow errors that incorrectly jump to both nearby locations and faraway locations. We observe that it is relatively easy to track runtime control flow history within a small region. Thus, instead of validating every *branch* instruction like previous methods, PaSS reduces the checking frequency by deferring the validation of this history until the control flow transfers to a new region. In this way, PaSS minimizes the number of required validation instructions without losing significant fault coverage. We also notice that previous works either cannot protect inter-procedural control flow transfers [2, 43, 55] or protect them with too much overhead [15, 24]. To solve this problem, PaSS introduces a novel, low cost technique to ensure inter-procedural control flow transfers. In the Commercial-Off-the-Shelf (COTS) embedded market, achieving high fault coverage with minimal overhead has the foremost importance. PaSS is designed to satisfy both constraints and makes the following contributions:

- Compared to traditional signature-based methods that suffer from 64.7% run-time overhead, PaSS only has an average performance overhead of **15.5%**, achieving **76% reduction**.
- PaSS can protect inter-procedural control flow transfers with a low cost scheme that incurs only an additional **3.6%** overhead.
- PaSS achieves **98.8%** fault coverage for illegal control flows on the SPEC2006 benchmark suite [22] (**99.0%** with inter-procedural protection), maintaining the same level of coverage as more detailed-checking methods.

2 Background and Motivation

2.1 Fault Detection Techniques

Fault detection is necessarily the first step to protect systems from transient faults. As we mentioned above, fault detection can be achieved through redundant checking in either hardware or software. Hardware-based redundant checking involves executing extra validating instructions in duplicated or specially designed hardware modules. These

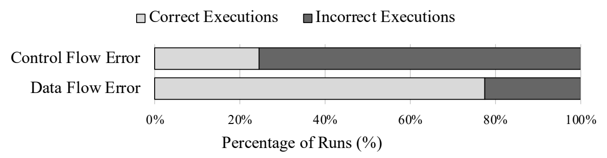

Figure 1. Percentage of incorrect executions caused by control flow errors and data flow errors.

solutions usually have broad fault coverage and low performance overhead, but incur high area and energy costs. Furthermore, they do not have any flexibility once the design is deployed on chips. Due to these limitations, hardware-based solutions are too expensive to be considered in embedded microprocessors. In contrast, software-based techniques are more appealing for embedded systems since they are flexible and free of the production cost, but they generally suffer from higher performance overheads.

Specifically, software-based transient fault detection techniques are classified into two categories: data flow protection and control flow protection. From our opportunity tests, we find most of data flow errors can be masked during program executions, but control flow errors are difficult to hide. Figure 1 shows the percentage of program executions for which incorrect behaviors result from injecting a single bit error in a random register (data flow error) or in the branch target address (control flow error) for the SPEC CPU 2006 benchmarks [22]. From the result, nearly 80% of errors in data flow have no effect on programs, but control flow errors cause more than 75% of executions to behave incorrectly. Although both control flow and data flow are equally important to be protected, this work focuses on detecting illegal control flows since control flow errors are more likely to cause programs to behave incorrectly.

2.2 Signature-based Control Flow Protection

To detect control flow errors, extra checking instructions are inserted into programs to make sure a control flow arrives at the correct target. We will use CFCSS [42] as an example to briefly introduce the fundamentals behind traditional control flow protection schemes, including their limitations.

At compile time, CFCSS assigns a unique integer, known as the signature S_i, to every basic block in a program. A basic block (BB) is a container for group of instructions with a single entry and a single exit. For each control flow from *srcBB* to *destBB*, a signature difference $d = S_{src}\ xor\ S_{dest}$ is also statically computed. Finally, a general purpose register G is allocated to hold the signature of executing BB. During run time, G is firstly initialized to the signature of entry BB. Whenever control flow transfers from *srcBB* to *destBB*, G is updated using $G = G\ xor\ d$. After this update, G should be equal to the signature of *destBB* unless an error has occurred. Therefore, a comparison between the updated G and S_{dest} is inserted to validate the transfer. Figure 2 shows the basic operation of CFCSS technique. For more complicated control flow patterns such as multiple predecessors, additional

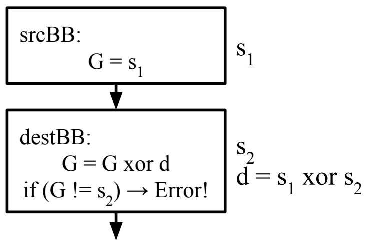

Figure 2. Signature-based control flow checking.

updating instructions are inserted to make sure *G* gets the correct value. Although CFCSS achieves high fault coverage (around 99%), every basic block is instrumented with signature updating and validating instructions, causing significant performance overhead (up to 130.8%, 64.7% on average).

More recent works [24, 62] try to cut down this huge overhead by injecting fewer instructions into programs. Even though these techniques bring overheads down to 25%, their fault coverage are all compromised, only ranging from 92% to 96%. For example, ACS [24] increments a simple counter every time a *branch* instruction gets executed. Whenever the control flow crosses two statically defined regions, the counter is checked against a pre-calculated value to make sure certain number of BBs has traversed. While achieving less overhead, ACS loses coverage if control flows incorrectly jump to any other path with the same counter value.

Since trading fault coverage for better performance is not an optimal solution, our work strives to minimize overhead without losing any coverage. Through experiments, we found signature updating instructions (*xor* instruction in CFCSS) only count for 30% of overhead. Because they are simple ALU operations, their effects can be easily hidden by the Instruction Level Parallelism (ILP). Hence, signature checking instructions (*cmp* followed by *bne*) are the primary source of overheads. To effectively reduce it, fewer checking instructions should be used. As carelessly deleting checking instructions hurts the coverage, a new validation method, PaSS, is proposed. In next section, we will explain how PaSS achieves low overhead while keeping high coverage.

3 Path Sensitive Signatures

Control flow errors can jump to both nearby locations (short-distance control flow errors) and faraway locations (long-distance control flow errors). To detect short-distance errors, we must distinguish between valid and invalid control flow paths that traverse a small code region. Fortunately, for small code regions, there are few valid control flow paths, which the compiler can statically enumerate. It is therefore possible for us to minimize the validation overhead by comparing the total executed path through a small region to this valid set only after execution of the region is complete. However, this method cannot be applied to long-distance control flow errors, because enumerating all valid control flow paths across large regions is not practical. Therefore, PaSS uses two different algorithms to effectively detect both short-distance and long-distance control flow errors. To clearly categorize two types of errors, we exploit the standard interval analysis. An interval (shown as dashed region in Figure 3) is a maximal group of basic blocks that satisfies the following properties:

- The header node of an interval dominates all other nodes in the same interval. In other words, an interval only has a single entry but can have multiple exits.
- Each interval contains at most one loop, with back edge pointing to interval's header node. Note that nested loops will be separated into multiple intervals.

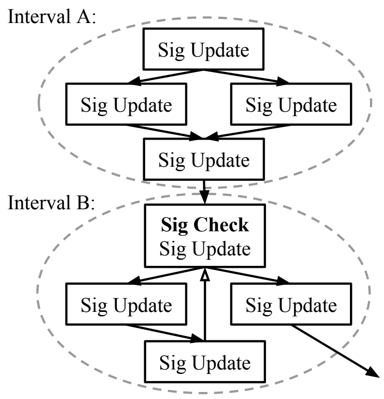

Figure 3. PaSS high-level operation (back edge is marked by the hollow arrow).

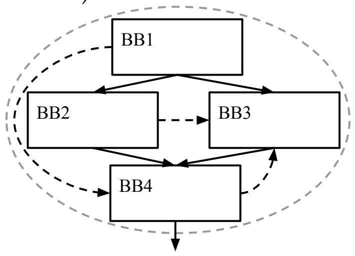

Figure 4. Intra-interval errors (shown in dashed arrows).

With an application divided into intervals, its control flows transfer either within the same interval or from the exit block of one interval to the entry block of another interval. Accordingly, control flow errors can be naturally classified into intra-interval errors (short-distance errors) and inter-interval errors (long-distance errors). As shown in Figure 3, PaSS constantly updates a signature to memorize current control flow path. Whenever the control flow arrives at a new interval, the signature is checked with a statically determined value to make sure no error happened in the previous interval. Simultaneously, the checking also validates the control flow transferring across intervals. Thus, PaSS is able to use a single checking instruction to detect both intra-interval and inter-interval errors at the same time.

3.1 Intra-Interval Error Detection

Intra-interval errors occur when control flows incorrectly transfer from *srcBB* to *destBB*, where both *srcBB* and *destBB* are in the same interval. Examples are shown with dashed arrows in Figure 4. Note that control flow errors that branch

Algorithm 1: Pseudocode for SV Assignments

```
cur_SV = 1;
path_history = [];
for cur_BB in interval_DFS_traversal do
    if !cur_BB.visited then
        cur_BB.visited = true;
        cur_BB.SV = cur_SV++;
        if cur_BB is interval_exit then
            while get_CS_value(cur_BB) in path_history do
                cur_BB.SV = cur_SV++;
            end
            path_history.push(get_CS_value(cur_BB));
        end
    end
end
```

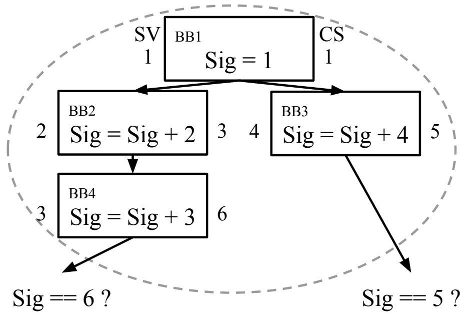

Figure 5. Basic operation for intra-interval protection.

back to interval's entry BB (from BB4 to BB1) are counted as inter-interval errors, which will be discussed in next section.

To detect intra-interval errors, PaSS firstly assigns a unique, non-negative integer to each basic block (BB) in the interval, known as the *Step Value (SV)*. Then, PaSS calculates the *Cumulative Sum(CS)* for each BB. The *Cumulative Sum* is computed by taking the maximum sum of all *SVs* from the interval entry BB to current BB along control flow edges. In other words, each BB's *CS* is the maximum path sum accumulated by *SVs* starting from the header node. Once these two values are determined, PaSS inserts signature update instructions at the beginning of each BB, which will increment the signature by corresponding BB's *SV*. Finally, a checking instruction is inserted when control flow leaves the current interval to verify the signature matching with the *CS* value of corresponding exit block. Considering an interval can have multiple exit BBs with different *CS* values (BB3 and BB4 in Figure 5), the signature is checked against different numbers depending on which exit BB is taken during run time.

We notice that the intra-interval's fault coverage is dependent on the assignment of *SV*s to BBs. To maximize the coverage, we decide to assign each BB with a unique *SV*, and make every exit BB with a different *CS*, because making each path sum unique can significantly reduce the probability that an invalid control path produces a valid control flow signature. To implement the proposed strategy, we use a depth-first-search (DFS) algorithm shown in Algorithm 1.

The basic operation for intra-interval error detection is shown in Figure 5. Numbers on the left side of blocks represent BBs' *SV* and numbers on the right side represent the *CS*. If an intra-interval error occurs during run time, one or more blocks will be skipped or re-executed, making the final

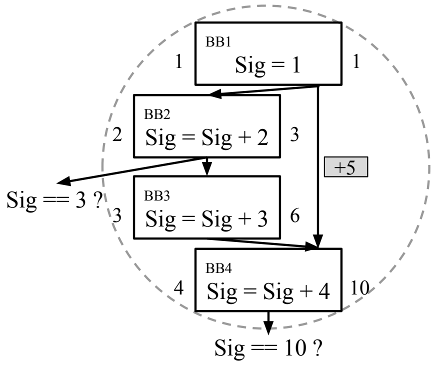

Figure 6. Intra-interval protection with the extra increment.

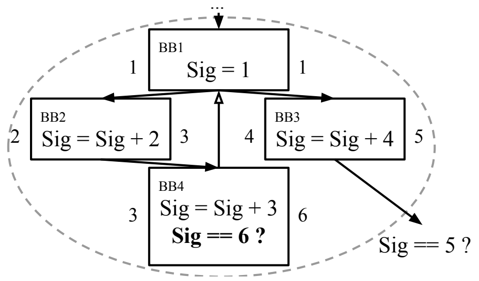

Figure 7. Extra checking instruction for loop interval.

signature value unequal to the *CS* value of the exiting BB. For instance, assuming a control flow is supposed to transfer from BB2 to BB4, but a transient fault causes the control flow to branch to BB3 instead. Upon exiting the interval, the control flow signature will be equal to 7 (1+2+4) rather than 6 (1+2+3) nor 5 (1+4). Thus, inserted checking instruction detects this error. Similarly, if control flows incorrectly jump to the middle of BBs, the signature updating instruction will be skipped, resulting a mismatch again at the checking point.

If multiple paths exist between two BBs, one extra increment will be inserted right before entering the *destBB* among other paths that are not the largest control flow path. The value of the increment is the difference between *destBB*'s *CS* (maximal path sum) and the path sums of other shorter paths. Example is shown in Figure 6, with the extra increment marked beside the target control flow edge. This technique adds negligible overhead since each critical path needs at most one extra increment to generate correct signature value.

The last special case we need to address is the loop interval, where the signature gets re-initialized every time the loop branches back to entry node, as shown in Figure 7. PaSS cannot detect control flow errors within the loop body since the signature value is not maintained across loop iterations. To solve this problem, we insert a checking instruction in the basic block that contains the back edge (bold instruction in BB4). Before the signature gets re-initialized, we compare it with the *CS* value of the latch block to make sure no error happened in the current loop iteration. Note that extra increments may also be inserted inside the loop if necessary.

To summary, PaSS ensures intra-interval control flow correctness by verifying the signature in following intervals'

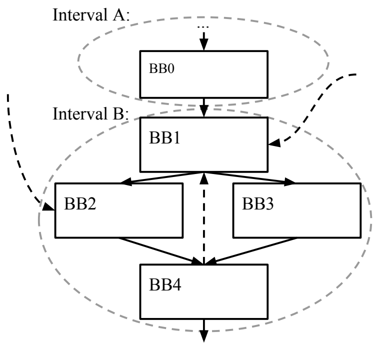

Figure 8. Inter-interval errors (shown in dashed arrows).

header node and in latch block, which greatly reduces the checking frequency compared to prior techniques that require the validation in every BB.

3.2 Inter-Interval Error Detection

Inter-interval errors are more likely to happen if control flows mistakenly jump to faraway locations. As shown in Figure 8, they either directly jump to the middle of an interval without traveling through the head node, or enter the interval with an illegal predecessor. Since inter-interval errors can jump to arbitrary locations, previously proposed method is no longer acceptable because tracking entire program's control flows is almost impossible. Thus, a more conservative checking algorithm is developed to guarantee every control flow that crosses two intervals behaves correctly.

To detect inter-interval errors, PaSS statically assigns an unique value, similar to each BB's *SV*, to each interval in a program, known as the *Region Value (RV)*. Whenever control flows enter a new interval, the control flow signature will be compared with the interval predecessor's *RV* to make sure the branch comes from one of the legal predecessors. After passing the checking instruction, current interval's *RV* gets assigned to the signature. Figure 9 shows inserted instructions for inter-interval error detection. Each interval's *RV* is represented by a letter on the upper left. If a branch arrives at interval *B* but from somewhere other than its legal predecessor, the error will be detected since the control flow signature is not set to *A*. In addition, if control flows erroneously jump to the middle of an interval, they will skip the signature updating instruction located in the entry block. Therefore, these errors will also be caught at the next signature check.

For intervals with more than one predecessors, the signature value needs to be checked against multiple *RVs* to ensure control flow correctness, which incurs unnecessary overheads. To simplify the checking process, we introduce a new static value called the *Region Difference (RD)* for each interval predecessor (excluding the first predecessor that the compiler processes). The *RD* of a predecessor interval is calculated by taking the *xor* value of the interval's *RV* and the first predecessor interval's *RV*. Once the *RD* is determined, an extra signature updating instruction is inserted in each predecessor's BB (except the first predecessor). The instruction

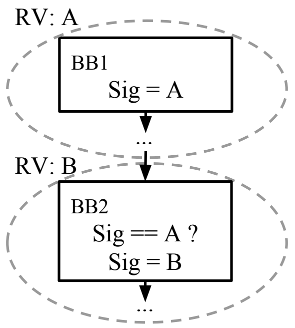

Figure 9. Basic operation for inter-interval protection.

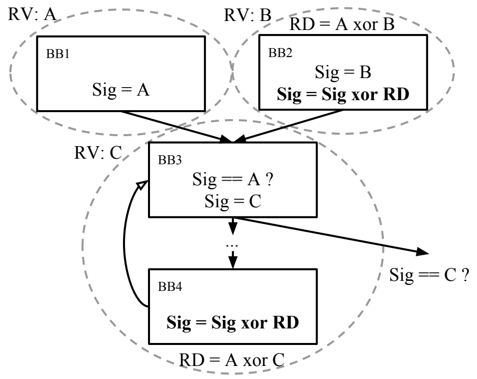

Figure 10. Extra updating instructions for fan-in interval.

simply updates the signature to be the *xor* value of *RD* and the signature itself, as shown in bold instruction of Figure 10. In this way, whichever predecessor the control flow comes from, the signature always equals to the first predecessor's *RV* upon reaching the destination. Consequently, a single checking instruction is enough to detect all inter-interval errors. In case one exit BB connects to multiple intervals where *RDs* are needed, the extra updating instruction is inserted along exiting edges to make sure the correct signature is used by different intervals. Lastly, if an interval contains a loop, the latch block will also branch to interval's header node. In this case, we consider the current interval to be one of its own predecessors and treat the interval same as the fan-in interval. Thus, the signature should also be updated with *RD* in the latch block, as illustrated in BB4 of Figure 10.

3.3 Integrating Two Checking Methods

Since both intra-interval and inter-interval error detection rely on signature checkings and both of them perform validations in interval's header node, we can combine these two checking methods together to further reduce performance overheads. To achieve this, we separate the control flow signature into two parts: the upper half is used for inter-interval checking and the lower half is used to detect intra-interval errors. The reason behind this partition scheme is because updating the signature with blocks' *SVs* will only change the signature's lower bits, having no effect on inter-interval checking. According to our study, using a 32-bit integer as the control flow signature is sufficient for error detection,

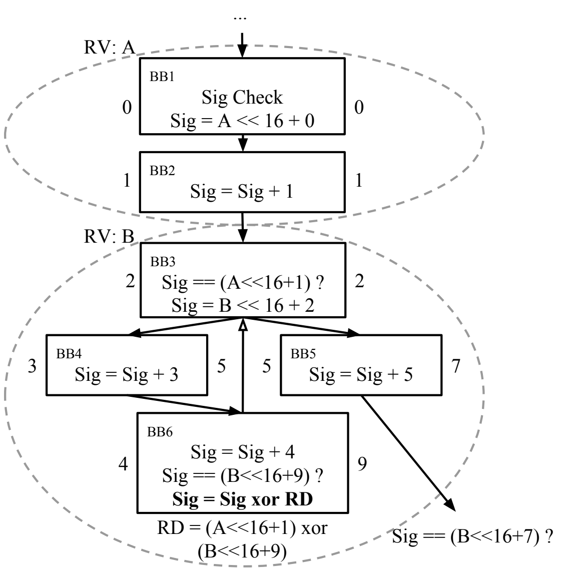

Figure 11. Integrated solution for PaSS.

where bits[15:0] are reserved for intra-interval protection and bits[31:16] store the information for inter-interval protection. Consequently, the final expected value at each interval's exit is equal to that interval's *RV* left shifted by 16 bits plus that interval exit's *CS* (*RV* << 16 + *CS(exitBB)*).

Figure 11 illustrates PaSS's integrated solution. Before a program starts running, the control flow signature is firstly initialized to *RV(entry_interval)* << 16 + *CS(entry_block)*. While traversing other blocks in the same interval, the signature is incremented by their corresponding *SVs*. Whenever the control flow transfers to a new interval, for example, from BB2 to BB3, the current signature will be checked against the expected value of the interval exit. One thing to point out is that with the integrated solution, the calculation for *Region Difference (RD)* is based on the expected value of two interval exits rather than their *RVs*, and the expected value for the latch block in a loop is *RV* << 16 + *CS(latch_block)*.

3.4 Inter-procedural Control Flow Protection

Transient faults can also happen to inter-procedural control flows (e.g. call and return instructions). According to our study, prior works either do not provide a solution [42, 43, 56] or spend too much costs (about 15% performance overhead) to protect it [15, 24]. To optimize this limitation, we propose a lightweight technique to detect errors that occur when control flows transfer between the calling and the callee site.

Instead of using the control flow signature alone, our proposed scheme relies on an additional signature, called the function signature (*F_Sig*) to detect inter-procedural control flow errors. At compile time, every function in a program will be statically assigned a unique ID. To protect transfers from caller to callee, PaSS updates *F_Sig* with the ID of the called function just before the *call* instruction. At the beginning of each function (except the *main* function), a checking instruction is also inserted to verify *F_Sig* matches with the

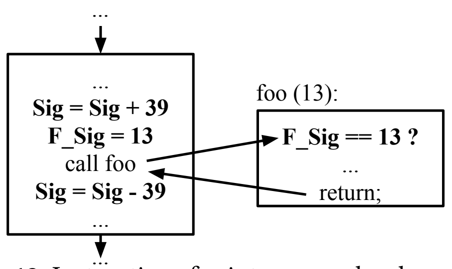

Figure 12. Instructions for inter-procedural protection.

static ID of the corresponding function. Thus, if a *call* instruction jumps to the wrong function, the checking instruction will detect a mismatch and report the error.

To protect transfers returning from callee to caller, two instructions are inserted around the *call* instruction on the caller side. Before the *call* instruction, the control flow signature will be incremented by a statically determined random value which is greater than the interval's largest *CS* to prevent aliasing with our intra-procedural protection scheme. After the *call* instruction, the control flow signature will be decremented by the same number to restore the original value. If a control flow returns to a wrong location, the control flow signature will not be restored correctly. Consequently, the error will be detected at next control flow signature check. Figure 12 shows inserted instructions for inter-procedural protection.

For the direct function call, the compiler can statically access the called function. In such cases, PaSS will insert the checking instruction in the callee function to make sure control flows arrive at the correct target. However, for indirect calls where targets are unresolved at compile time (e.g. call through pointers), PaSS cannot insert validating instructions. Nevertheless, the increment and decrement instructions are still inserted around the *call* instruction, guaranteeing that the called function will eventually return to the correct place. We choose this implementation strategy because the compiler has limited capability to analyze indirect function calls, and the probability of a single bit flipping to result in the start of a different function is nearly zero.

In our current design, the control flow signature is allocated as a local variable inside of each function. When a *call* instruction gets executed, the current signature will be saved on the function stack and thus will not affect the signature allocated in the callee function. On the other hand, the function signature needs to be a global variable since every function needs to access it.

3.5 Discussion

We use the example shown in Figure 13 to distinguish PaSS from CFCSS [42] and ACS [24]. In this instance, CFCSS needs to insert validating instructions in every basic block (6 in total), whereas both PaSS and ACS only requires a single checking instruction because all BBs are grouped into an interval. Since ACS only counts the number of traversed BBs, it loses the coverage if erroneous control flows jump

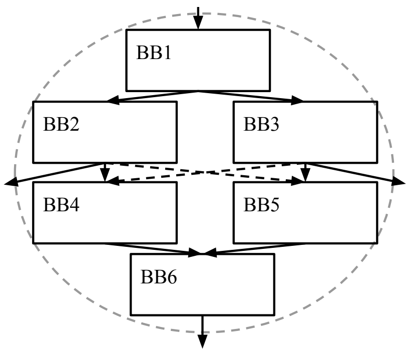

Figure 13. Control flow example.

to any other path with the same counter value (from BB3 to BB4 or from BB2 to BB5). However, PaSS still catches all errors as each control flow path gets accumulated differently (detailed evaluations are provided in Section 5). Given this, we conclude PaSS reduces the overhead while keeping the coverage same as the traditional methods.

4 Evaluation Methodology

4.1 PaSS Implementation

We implemented the PaSS scheme using the LLVM [28] compiler infrastructure. To apply PaSS, an application's source code is firstly compiled into LLVM Intermediate Representation (IR) with *-O3* option. At this step, common compiler optimizations are performed if necessary. For example, irreducible patterns will be converted to reducible structures through some code duplication. Then, intervals are internally formed based on the program's control flow graph (CFG). Exploiting the interval information, PaSS allocates signatures and instruments initializing, updating, and checking instructions in corresponding basic blocks of the program. Finally, binary executables are generated with the PaSS technique embedded. We notice that some optimizations such as the constant propagation may delay the signature initialization until the second BB, leaving the entry BB unprotected. To solve this, PaSS should be run as the last optimization step so that it will not be overwritten by others. Since instrumented instructions must be placed in exact BB, further compiler optimizations are not allowed.

4.2 Statistical Fault Injection Model

Statistical Fault Injection (SFI) has been extensively used [18, 24, 47, 61] to evaluate transient fault detection methods. In this fault model, Single-Event-Upset (SEU) is assumed, which means that an arbitrary bit will be flipped to simulate a transient fault at a random time during application's execution (the probability of happening more than one transient faults per program's execution is extremely small). Since PaSS targets the control flow error detection, we only consider faults that result illegal control flows. Note that similar to other signature-based techniques [2, 24, 42], PaSS cannot protect errors corrupting the value of a branch condition, which will cause legal but incorrect control flow transfers. However, these errors can be covered by combining PaSS with other data flow protection solutions.

Transient faults can cause illegal control flows in multiple ways, including but not limited to: 1) A transient fault converts a normal instruction into a control flow instruction. 2) Conversely, a transient fault can also convert a control flow instruction into a normal instruction. 3) A transient fault affects one of the registers used in the computation of a branch target address. 4) A transient fault directly changes the branch target address or the program counter (PC). However, no matter where errors happen, they all share the identical symptom: incorrect branch targets. Therefore, in our experiments, we choose to inject faults that directly change the branch target address to guarantee every fault will result a control flow error.

Errors are injected using GEM5 simulator [8]. First, GEM5 runs a program without any modification to collect the total simulation cycle. A random cycle is then selected as the fault injection point. Next, GEM5 will simulate the program again with the fault injection enabled. Once the simulated program reaches the fault injection point, the next control flow instruction's target address is chosen as the fault injection target. To complete the fault injection, one random bit of this address is flipped. The simulation continues until either the PaSS reports the error, or the program exits. Although we only inject faults to branch target addresses, our technique should be able to detect faults in other places as well, as long as the program's control flow gets corrupted.

5 Evaluation Results

To evaluate PaSS, we collect all C benchmarks (12 in total) from the SPEC CPU 2006 [22], including both integer and floating point suits, as our workloads. However, PaSS is completely programming-language independent and should have the same behavior no matter which language is used. To compare PaSS with prior techniques, we also implemented CFCSS [42] and ACS [24] as our baselines. As we mentioned in Section 2.2, CFCSS performs conservative control flow checking, which brings both high fault coverage and high overheads. On contrary, ACS is a counter-based scheme that tries to minimize the overhead by sacrificing some coverage.

5.1 Fault Coverage

To compare the fault coverage among evaluated methods, we inject 1000 faults per benchmark per technique using the method described in Section 4.2. The results of fault injection experiments are classified into following categories:

- Masked: The injected fault has no effect on the program due to masking. In this case, benchmarks will execute as normal and generate correct outputs.
- SW Detect: The inserted checking instructions successfully detect incorrect control flows and report errors.

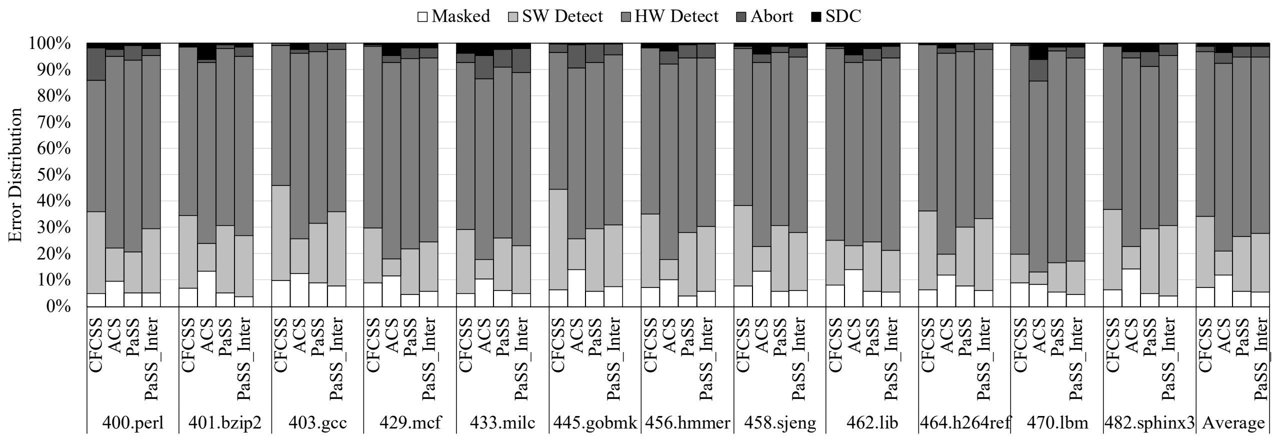

Figure 14. Fault coverage for CFCSS, ACS, PaSS as well as PaSS_Interprocedural techniques.

- HW Detect: The injected fault dumps the program by triggering a page fault or an invalid instruction etc.
- Abort: The incorrect control flow causes an infinite loop or jumps out of the memory address bound. In this category, simulation will terminate incorrectly.
- Silent Data Corruption (SDC): The program finishes execution without reporting any error but generates wrong outputs due to the injected fault. Generally, SDCs are the most harmful errors since the program produces incorrect outputs without any hint.

Figure 14 shows the distribution of injected faults for each technique. Like previous studies, we define the fault coverage of a technique to be the percentage of faults that do not result in SDCs, because errors in other categories can be observed by end users. The fault coverage of PaSS as well as other software-based techniques [2, 24, 42, 55] are not 100% because they cannot protect pre-built library functions due to lack of source codes. They will also miss the error if the control flow incorrectly jumps to a specific place to skip the signature check. As a result, these solutions should not be used on mission-critical systems.

Comparing results, ACS shows the lowest coverage rate among all evaluated methods. Even with its inter-procedural protection scheme, the fault coverage only achieves 96.4%. We observe that the share of *SW Detect* for ACS is significantly smaller than other techniques. According to our analysis, incapable of detecting inter-interval errors is one of the primary reasons resulting in ACS's low coverage. In addition, ACS will also miss some faults happen inside of loops and large intervals as its counter-based approach causes aliasing problems, like the example shown in Figure 13.

On the other hand, the fault coverage for CFCSS, PaSS and PaSS with inter-procedural protection is 99.0%, 98.8%, and 99.0%, respectively. This result confirms that PaSS does not hurt the coverage. We would like to point out although the *SW Detect* part for PaSS is 5.6% less than CFCSS, the fault coverage for both methods are identical because the control flow error may immediately trigger a page fault (contributes to *HW Detect* share) before reaching checking instructions instrumented by PaSS. In summary, PaSS can offer the same level of fault coverage as previously proposed conservative solutions, whereas ACS is less effective while detecting long-distance control flow errors.

5.2 Performance Overhead

The number of signature checking instructions inserted in a program plays a decisive role in overall overheads. CFCSS needs to insert the validation instruction in every basic block but PaSS only requires the instrumentation at the interval granularity. We collected the number of basic blocks and intervals from all benchmarks used in experiments. On average, there are 21502 basic blocks but only 4329 intervals per program. This result implies that PaSS will insert 4.97x fewer validation instructions into applications. In real, this number will be smaller considering the presence of loops. To accurately measure the performance overhead, we recorded the execution time for each benchmark using a desktop with an Intel i7-6700 CPU clocked at 3.4GHz and 8GB of DRAM. Figure 15 plots the performance overhead for each technique. The white part of the bar represents the overhead for intra-procedural protection, and the gray part represents the inter-procedural protection overhead. All experimental results are normalized to original programs.

The first bar in each benchmark illustrates the performance overhead of CFCSS technique. In this category, two-thirds of benchmarks experience more than 50% overhead, and three of them even have more than 100% overhead (130.8% highest). On average, the performance overhead for CFCSS is 64.7%. Since CFCSS does not support inter-procedural protection, all of its overhead comes from intra-procedural protection. The following two bars demonstrate the overheads for ACS and PaSS techniques, respectively. For intra-procedural protection, *403.gcc* suffers the highest overhead among all benchmarks (29.9% for ACS and 32.0% for PaSS), whereas *433.milc* and *470.lbm* only shows 2-3% overhead for both techniques because most of basic blocks

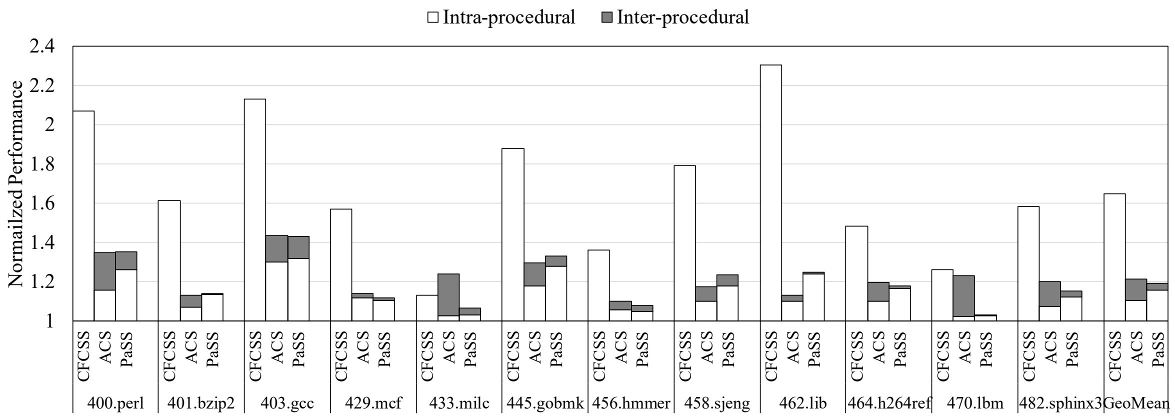

Figure 15. Performance overhead for CFCSS, ACS, and PaSS techniques.

in these programs are covered by large intervals. On average, the intra-procedural overhead for ACS and PaSS are 10.7% and 15.5%, respectively. Both methods achieve significant overhead reduction comparing to CFCSS. PaSS shows a 4.8% higher overhead than ACS because it needs to perform the signature check in every loop iteration, whereas ACS only verifies the counter after exiting the entire loop. Although ACS's lazy-check method reduces the overhead, it hurts the fault coverage as we described in the previous section. If a program does not contain any loop or only has small to medium iteration numbers, PaSS and ACS show the similar overhead (*429.mcf* and *456.hmmer*).

For inter-procedural protection, *400.perl* and *403.gcc* exhibit the highest additional overhead (around 10%) with PaSS technique, because both of workloads contain a lot of small functions and involve frequent function calls and returns. On average, PaSS incurs an additional 3.6% overhead, but it is much more efficient than ACS, which suffers from 10.8% overhead (3x higher than PaSS). The reason is because ACS uses a couple of expensive *modulo* operations, in both caller sides and callee sides, to catch inter-procedural errors. In contrast, PaSS only needs to verify the callee side without any costly instruction. Overall, the geomean performance overhead for CFCSS, ACS, and PaSS techniques are 64.7%, 21.5%, and 19.1%. This result shows PaSS achieves roughly 3.39x less performance overhead than CFCSS. Comparing to ACS, PaSS shows a slightly better performance improvement (2.4%), but PaSS is a lot more effective since it keeps the same level of fault coverage as conservative solutions.

5.3 Detection Latency

Although PaSS only detects control flow errors, it can easily be combined with other software recovery mechanisms such as Encore [19], Bolt [30], and InCheck [16]. Therefore, we also evaluate the detection latency for each technique since a longer latency generally increases the overhead for recovery schemes. We do expect PaSS to have a higher detection latency since it reduces the validation frequency. Figure 16 illustrates each method's detection latency. *Within 2K*, *Within 5K*, and *Within 10K* represent whether the detected error is reported within 2K, 5K, or 10K cycles after the fault injection. For example, the third column in *400.perl* shows that among all errors detected by PaSS, 10.4% are found within 2K cycles, 4.3% are found within 5K cycles, and the remaining 85.3% are detected within 10K cycles after the fault injection. Comparing all benchmarks, *400.perl* and *403.gcc* exhibit the longest detection latency among all evaluated methods because their basic blocks are larger than others. For remaining benchmarks, most of errors are detected under 5K cycles.

Specifically, CFCSS shows the lowest detection latency with 80.3% of errors detected under 2K cycles. This result meets our expectation as CFCSS validates every *branch* instruction. We expect ACS and PaSS have similar detection latency since both of them validate the signature at the interval granularity. However, experiment results demonstrate that ACS has a similar detection latency compared to CFCSS, where 73.5% of errors are reported within 2K cycles and 25.7% are reported within 5K cycles. This short detection latency confirms that ACS can only detect errors happened very close to checking instructions. If the error happens inside of a loop or in the middle of a large interval, ACS has very limited ability to catch the fault. This finding agrees with our previous analysis in Section 5.1.

PaSS has the highest detection latency among three evaluated methods. On average, 19.4% of errors are reported within 2K cycles, and 65.6% are reported within 5K cycles. The remaining 15% of errors takes more than 5K but less than 10K cycles to be detected (PaSS_Inter shows very similar results as well). Although PaSS experiences the longest detection latency, it can still report 85% of errors within a reasonable time (5K cycles). Since architectural recovery mechanisms generally perform the checkpoint per 100K instructions [53], we believe PaSS adds negligible overhead to these methods.

6 Related Work

Software-based techniques for control flow error detection have been well discussed in other works. In this paper, we reported a detailed comparison among CFCSS, ACS, and PaSS.

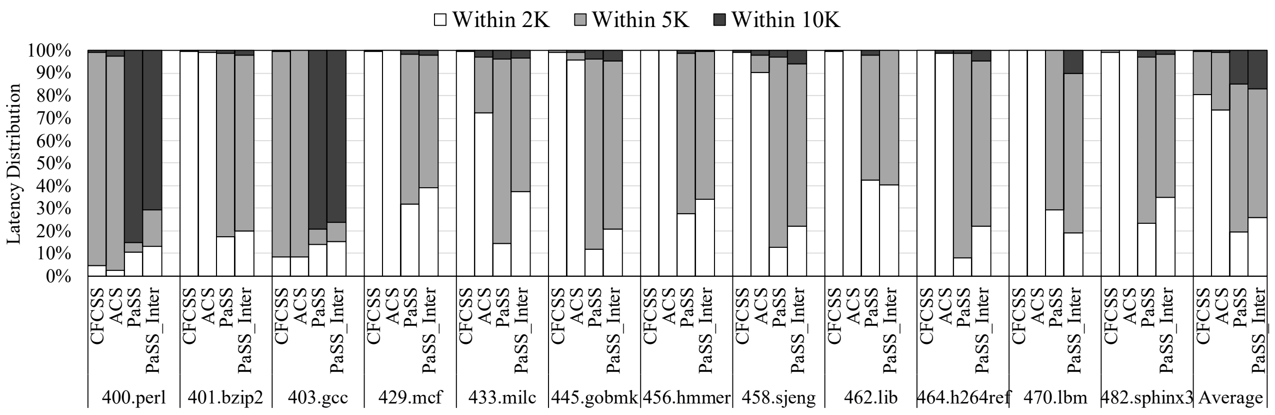

Figure 16. Detection latency for CFCSS, ACS, PaSS, and PaSS_Interprocedural techniques.

[2, 20, 55] are some classical works using run-time assertions to protect control flows. Other works [3, 41, 51, 52] have developed different algorithms to detect transient faults, but all of them need to validate the control flow correctness on every *branch* instruction, causing too much overhead. Similar to ACS, [10, 15, 29, 50, 54] are examples of low overhead techniques for control flow protection. However, they all sacrifice the fault coverage to achieve better performance, making them less desirable to the commodity embedded systems. Control flow integrity works [1, 9, 26, 58, 60] aim to protect control flows from malicious attacks. Although PaSS is not specifically designed from the security perspective, we believe it is still effective to detect some attacks causing illegal control flow transfers.

In addition to the control flow protection, previous works also proposed solutions for data flow protections. EDDI [43], SWIFT [47], and NEMESIS [17] check the data flow by duplicating program instructions. Later works such as PROFiT [48] improves the SWIFT by adding the architectural vulnerability factor (AVF) analysis [40]. Others [13, 25, 37, 44] focused on minimizing the overhead brought by instruction duplication. PaSS does not support data flow protection, but it can easily be combined with these techniques to further extend the fault coverage.

Redundant Multi-threading (RMT) is another approach to transient fault detection. AR-SMT [49] firstly introduce the idea of using simultaneous multi-threading for transient fault detection. In this work, an active thread runs the program, and a redundant thread checks the program's correctness. Following works [21, 39, 46] tried to optimize the overhead caused by RMT. Software-based redundant multi-threading for transient fault detection (SRMT) [57] is another software solution that achieves redundant checking with multiple threads. Using the adaptive multi-threading, [23, 61] successfully reduce the performance overhead. However, this type of works relies on an extra thread to validate the correctness, cutting processor's throughput by half.

Typical hardware-based solutions for control flow protection use watchdog processors [27, 33, 34, 36]. A watchdog processor is a simple co-processor alongside the main processor to perform concurrent system-level checking. Other hardware-assisted techniques [4, 5, 14, 45] suggest to modify part of the processor hardware or add special instructions for error detection. Argus [35] uses distributed checkers for various components in processors. More recent works [31, 32] deploy acoustic wave detectors to catch soft errors. Nevertheless, all hardware-based solutions are too expensive to be used in commodity embedded microprocessors since they have substantial design and area costs.

7 Conclusion

With developments in semiconductor technology, transistor size has reduced exponentially. In addition, increasing demands for energy efficiency have driven aggressive voltage scaling as well as noise margin reduction on microprocessors. All of these make current systems less reliable and more likely to get affected by transient faults. To keep control flow safe from transient faults with minimal overhead but maximal coverage, we propose *Path Sensitive Signatures for Control Flow Error Detection*. PaSS achieves its high efficiency by combining two different validation methods and reducing the checking frequency to the interval granularity. PaSS also offers a low overhead mechanism to protect inter-procedural control flow transfers. Experimental results demonstrate that PaSS brings down the performance overhead from 64.7% for traditional control flow signatures to 15.5% on average while maintaining the same level of fault coverage compared to the prior approach [42]. Inter-procedural protection is provided at the cost of an additional 3.6% overhead.

Acknowledgments

The authors would like to thank the anonymous reviewers for their constructive comments for improving this work. This research is supported in part by the U.S. Department of Energy, Office of Advanced Scientific Computing Research (ASCR), under the grant DE-SC0014134 and by the Office of Naval Research under the grant N00014-18-1-2020.

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)* 13, 1 (2009), 1–40.

[2] Zeyad Alkhalifa, VS Sukumaran Nair, Narayanan Krishnamurthy, and Jacob A. Abraham. 1999. Design and evaluation of system-level checks for on-line control flow error detection. *IEEE Transactions on Parallel and Distributed Systems* 10, 6 (1999), 627–641.

[3] Seyyed Amir Asghari, Hassan Taheri, Hossein Pedram, and Okyay Kaynak. 2014. Software-based control flow checking against transient faults in industrial environments. *IEEE Transactions on Industrial Informatics* 10, 1 (2014), 481–490.

[4] Todd M Austin. 1999. DIVA: A reliable substrate for deep submicron microarchitecture design. In *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture.* IEEE, 196–207.

[5] Saurabh Bagchi, Balaji Srinivasan, Keith Whisnant, Zbigniew Kalbarczyk, and Ravishankar K Iyer. 2000. Hierarchical error detection in a software implemented fault tolerance (sift) environment. *IEEE Transactions on Knowledge and Data Engineering* 12, 2 (2000), 203–224.

[6] Wendy Bartlett and Lisa Spainhower. 2004. Commercial fault tolerance: A tale of two systems. *IEEE Transactions on dependable and secure computing* 1, 1 (2004), 87–96.

[7] David Bernick, Bill Bruckert, Paul Del Vigna, David Garcia, Robert Jardine, Jim Klecka, and Jim Smullen. 2005. NonStop/spl reg/advanced architecture. In *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on.* IEEE, 12–21.

[8] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (2011), 1–7.

[9] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. 2011. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security.* 30–40.

[10] Edson Borin, Cheng Wang, Youfeng Wu, and Guido Araujo. 2006. Software-based transparent and comprehensive control-flow error detection. In *proceedings of the international symposium on Code generation and optimization.* IEEE Computer Society, 333–345.

[11] Shekhar Borkar et al. 2004. Microarchitecture and design challenges for gigascale integration. In *MICRO*, Vol. 37. 3–3.

[12] Greg Bronevetsky, B de Supinski, and Martin Schulz. 2009. *A foundation for the accurate prediction of the soft error vulnerability of scientific applications.* Technical Report. Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States).

[13] Zhi Chen, Alexandru Nicolau, and Alexander V Veidenbaum. 2016. SIMD-based soft error detection. In *Proceedings of the ACM International Conference on Computing Frontiers.* ACM, 45–54.

[14] Eric Cheng, Shahrzad Mirkhani, Lukasz G Szafaryn, Chen-Yong Cher, Hyungmin Cho, Kevin Skadron, Mircea R Stan, Klas Lilja, Jacob A Abraham, Pradip Bose, et al. 2016. CLEAR: C ross-L ayer E xploration for A rchitecting R esilience-Combining hardware and software techniques to tolerate soft errors in processor cores. In *Proceedings of the 53rd Annual Design Automation Conference.* ACM, 68.

[15] Kiho Choi, Daejin Park, and Jeonghun Cho. 2019. SSCFM: Separate Signature-Based Control Flow Error Monitoring for Multi-Threaded and Multi-Core Environments. *Electronics* 8, 2 (2019), 166.

[16] Moslem Didehban, Sai Ram Dheeraj Lokam, and Aviral Shrivastava. 2017. InCheck: An in-application recovery scheme for soft errors. In *Design Automation Conference (DAC), 2017 54th ACM/EDAC/IEEE.* IEEE, 1–6.

[17] Moslem Didehban, Aviral Shrivastava, and Sai Ram Dheeraj Lokam. 2017. NEMESIS: A software approach for computing in presence of soft errors. In *Computer-Aided Design (ICCAD), 2017 IEEE/ACM International Conference on.* IEEE, 297–304.

[18] Shuguang Feng, Shantanu Gupta, Amin Ansari, and Scott Mahlke. 2010. Shoestring: probabilistic soft error reliability on the cheap. In *ACM SIGARCH Computer Architecture News*, Vol. 38. ACM, 385–396.

[19] Shuguang Feng, Shantanu Gupta, Amin Ansari, Scott A Mahlke, and David I August. 2011. Encore: low-cost, fine-grained transient fault recovery. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture.* ACM, 398–409.

[20] Olga Goloubeva, Maurizio Rebaudengo, M Sonza Reorda, and Massimo Violante. 2003. Soft-error detection using control flow assertions. In *Defect and Fault Tolerance in VLSI Systems, 2003. Proceedings. 18th IEEE International Symposium on.* IEEE, 581–588.

[21] Mohamed A Gomaa and TN Vijaykumar. 2005. Opportunistic transient-fault detection. In *Computer Architecture, 2005. ISCA'05. Proceedings. 32nd International Symposium on.* IEEE, 172–183.

[22] John L Henning. 2006. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News* 34, 4 (2006), 1–17.

[23] Saurabh Hukerikar, Keita Teranishi, Pedro C Diniz, and Robert F Lucas. 2018. Redthreads: An interface for application-level fault detection/correction through adaptive redundant multithreading. *International Journal of Parallel Programming* 46, 2 (2018), 225–251.

[24] Daya Shanker Khudia and Scott Mahlke. 2013. Low cost control flow protection using abstract control signatures. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 3–12.

[25] Daya Shanker Khudia and Scott Mahlke. 2014. Harnessing soft computations for low-budget fault tolerance. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture.* IEEE Computer Society, 319–330.

[26] Volodymyr Kuznetzov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. 2018. Code-pointer integrity. In *The Continuing Arms Race: Code-Reuse Attacks and Defenses.* 81–116.

[27] Stanford University. Computer Systems Laboratory and David Jun Lu. 1980. *Watchdog processors and VLSI.* Center for Reliable Computing, Computer Systems Laboratory, Stanford University.

[28] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization.* IEEE Computer Society, 75.

[29] Liping Liu, Linlin Ci, Wei Liu, et al. 2016. Control-Flow Checking Using Branch Sequence Signatures. In *Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), 2016 IEEE International Conference on.* IEEE, 839–845.

[30] Qingrui Liu, Changhee Jung, Dongyoon Lee, and Devesh Tiwari. 2016. Compiler-directed lightweight checkpointing for fine-grained guaranteed soft error recovery. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* IEEE Press, 20.

[31] Qingrui Liu, Changhee Jung, Dongyoon Lee, and Devesh Tiwari. 2016. Low-cost soft error resilience with unified data verification and fine-grained recovery for acoustic sensor based detection. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture.* IEEE Press, 25.

[32] Qingrui Liu, Changhee Jung, Dongyoon Lee, and Devesh Tiwari. 2017. Compiler-directed soft error detection and recovery to avoid due and sdc via tail-dmr. *ACM Transactions on Embedded Computing Systems (TECS)* 16, 2 (2017), 32.

[33] David J. Lu. 1982. Watchdog processors and structural integrity checking. *IEEE Trans. Comput.* 31, 7 (1982), 681–685.

[34] Aamer Mahmood and Edward J McCluskey. 1988. Concurrent error detection using watchdog processors-a survey. *IEEE Trans. Comput.* 37, 2 (1988), 160–174.

[35] Albert Meixner, Michael E Bauer, and Daniel Sorin. 2007. Argus: Low-cost, comprehensive error detection in simple cores. In *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on.* IEEE, 210–222.

[36] Thierry Michel, Régis Leveugle, and Gabriele Saucier. 1991. A new approach to control flow checking without program modification. In *Fault-Tolerant Computing, 1991. FTCS-21. Digest of Papers., Twenty-First International Symposium.* IEEE, 334–341.

[37] Konstantina Mitropoulou, Vasileios Porpodas, and Marcelo Cintra. 2013. DRIFT: Decoupled compiler-based instruction-level fault-tolerance. In *International Workshop on Languages and Compilers for Parallel Computing.* Springer, 217–233.

[38] Shubu Mukherjee. 2011. *Architecture design for soft errors.* Morgan Kaufmann.

[39] Shubhendu S Mukherjee, Michael Kontz, and Steven K Reinhardt. 2002. Detailed design and evaluation of redundant multi-threading alternatives. In *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on.* IEEE, 99–110.

[40] 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 *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on.* IEEE, 29–40.

[41] Hamed Nikookar and Ahmad Patooghy. 2017. A New Control Flow Checking Method to Improve Reliability of Embedded Systems. *Journal of Advances in Computer Research* 8, 2 (2017), 1–11.

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

[43] 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.

[44] Sunghyun Park, Shikai Li, Ze Zhang, and Scott Mahlke. 2020. Low-cost prediction-based fault protection strategy. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization.* 30–42.

[45] Roshan G Ragel and Sri Parameswaran. 2006. Hardware assisted preemptive control flow checking for embedded processors to improve reliability. In *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis.* ACM, 100–105.

[46] Steven K Reinhardt and Shubhendu S Mukherjee. 2000. *Transient fault detection via simultaneous multithreading.* Vol. 28. ACM.

[47] George A Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I August. 2005. SWIFT: Software implemented fault tolerance. In *Proceedings of the international symposium on Code generation and optimization.* IEEE Computer Society, 243–254.

[48] George A Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, David I August, and Shubhendu S Mukherjee. 2005. Software-controlled fault tolerance. *ACM Transactions on Architecture and Code Optimization (TACO)* 2, 4 (2005), 366–396.

[49] Eric Rotenberg. 1999. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *Fault-Tolerant Computing, 1999. Digest of Papers. Twenty-Ninth Annual International Symposium on.* IEEE, 84–91.

[50] Mohammad Abdur Rouf and Soontae Kim. 2015. Low-Cost Control Flow Protection via Available Redundancies in the Microprocessor Pipeline. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 23, 1 (2015), 131–141.

[51] H Severínová, J Abaffy, and T Krajčovič. 2015. Control-Flow Checking Using Binary Encoded Software Signatures. In *Innovations and Advances in Computing, Informatics, Systems Sciences, Networking and Engineering.* Springer, 345–347.

[52] Aviral Shrivastava, Abhishek Rhisheekesan, Reiley Jeyapaul, and Carole-Jean Wu. 2014. Quantitative analysis of control flow checking mechanisms for soft errors. In *Proceedings of the 51st Annual Design Automation Conference.* 1–6.

[53] Daniel J Sorin, Milo MK Martin, Mark D Hill, and David A Wood. 2002. SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *ACM SIGARCH Computer Architecture News*, Vol. 30. IEEE Computer Society, 123–134.

[54] Jens Vankeirsbilck, Niels Penneman, Hans Hallez, and Jeroen Boydens. 2017. Random Additive Signature Monitoring for Control Flow Error Detection. *IEEE Transactions on Reliability* 66, 4 (2017), 1178–1192.

[55] Ramtilak Vemu and Jacob Abraham. 2011. CEDA: Control-flow error detection using assertions. *IEEE Trans. Comput.* 60, 9 (2011), 1233–1245.

[56] Rajesh Venkatasubramanian, John P Hayes, and Brian T Murray. 2003. Low-cost on-line fault detection using control flow assertions. In *On-Line Testing Symposium, 2003. IOLTS 2003. 9th IEEE.* IEEE, 137–143.

[57] Cheng Wang, Ho-seop Kim, Youfeng Wu, and Victor Ying. 2007. Compiler-managed software-based redundant multi-threading for transient fault detection. In *Proceedings of the International Symposium on Code Generation and Optimization.* IEEE Computer Society, 244–258.

[58] Zhi Wang and Xuxian Jiang. 2010. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *2010 IEEE Symposium on Security and Privacy.* IEEE, 380–395.

[59] Ying C Yeh. 1996. Triple-triple redundant 777 primary flight computer. In *Aerospace Applications Conference, 1996. Proceedings., 1996 IEEE*, Vol. 1. IEEE, 293–307.

[60] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. 2013. Practical control flow integrity and randomization for binary executables. In *2013 IEEE Symposium on Security and Privacy.* IEEE, 559–573.

[61] Yun Zhang, Jae W Lee, Nick P Johnson, and David I August. 2012. DAFT: decoupled acyclic fault tolerance. *International Journal of Parallel Programming* 40, 1 (2012), 118–140.

[62] Zhiqi Zhu, Joseph Callenes-Sloan, and Benjamin Carrion Schafer. 2018. Control Flow Checking Optimization Based on Regular Patterns Analysis. In *2018 IEEE 23rd Pacific Rim International Symposium on Dependable Computing (PRDC).* IEEE, 203–212.