ePVF: An Enhanced Program Vulnerability Factor Methodology for Cross-layer Resilience Analysis

Bo Fang*, Qining Lu*, Karthik Pattabiraman*, Matei Ripeanu*, Sudhanva Gurumurthi†
*Department of Electrical and Computer Engineering
University of British Columbia
Email: {bof, qining, karthikp, matei}@ece.ubc.ca
† Cloud Innovation Lab, IBM
Email: sgurumu@us.ibm.com

***Abstract*—The Program Vulnerability Factor (PVF) has been proposed as a metric to understand the impact of hardware faults on software. The PVF is calculated by identifying the program bits required for architecturally correct execution (ACE bits). PVF, however, is conservative as it assumes that all erroneous executions are a major concern, not just those that result in silent data corruptions, and it also does not account for errors that are detected at runtime, i.e., lead to program crashes. A more discriminating metric can inform the choice of the appropriate resilience techniques with acceptable performance and energy overheads. This paper proposes ePVF, an enhancement of the original PVF methodology, which filters out the crash-causing bits from the ACE bits identified by the traditional PVF analysis. The ePVF methodology consists of an error propagation model that reasons about error propagation in the program, and a crash model that encapsulates the platform-specific characteristics for handling hardware exceptions. ePVF reduces the vulnerable bits estimated by the original PVF analysis by between 45% and 67% depending on the benchmark, and has high accuracy (89% recall, 92% precision) in identifying the crash-causing bits. We demonstrate the utility of ePVF by using it to inform selective protection of the most SDC-prone instructions in a program.**

Keywords: PVF, Crash Model, Cross-layer Analysis

I. Introduction

Transient hardware faults, typically caused by particle strikes, are a major concern in modern computer systems. Current manufacturing trends (e.g., smaller feature sizes, manufacturing variations) suggest that transient faults will increase in the future [1], [2]. Further, stringent energy constraints make it challenging to deploy resilience techniques to protect the system from hardware faults [3]. The problem is exacerbated in high-performance computing (HPC) systems, where the large scale and long running time make applications more prone to hardware faults.

A hardware fault can affect an application in one of the following ways: (i) *crash*, i.e., an exception is raised and the application is terminated, (ii) *hang*, i.e., the application runs for a significantly longer time than normal, (iii) *silent data corruption* (SDC), i.e., the application finishes with an incorrect output, and (iv) *benign*, i.e., the application finishes with a valid output. The first three are failure outcomes. Among these, SDCs are considered the most severe, because users will trust the application's output in the absence of an error indication. A crash can be detected by monitoring the application, while hangs can be detected using timeouts. However, there is no generic method to detect SDCs without re-executing the entire application and checking for a mismatch, or without a significant amount of hardware redundancy, both of which are expensive.

Our long-term goal is to develop a systematic method to inform the design of software protection techniques (e.g., code transformations) to make applications resilient to SDCs. A first and essential step towards this goal is estimating the SDC rates of programs. SDCs are caused by a combination of application-specific and system-specific factors. In this paper, we focus on the system-specific factors that lead to SDCs. The main insight underlying this work is that a fault that leads to a crash cannot (by definition) lead to an SDC. Because crashes are caused by a combination of the hardware and Operating System (OS) features, they can be systematically reasoned about in an application-independent manner. By removing the crash-causing faults from the set of all faults, one can obtain a tighter estimate of the SDC rate. This is as important as crashes are often the dominant failure outcome, and hence significantly outnumber both SDCs and hangs [4]–[7].

This paper proposes a new method, ePVF (enhanced PVF), that builds on the original Program Vulnerability Factor (PVF) analysis methodology proposed by Sridharan et al. [8] to remove crash-causing faults from the set of all faults. PVF is a systematic method to efficiently evaluate the error resilience of software under hardware faults. PVF can also be used for predictive and comparative analysis studies to understand the effect of different protection techniques or code transformations on the error resilience. However, PVF does not distinguish between fault outcomes and, essentially, treats crashes, SDCs and hangs as equally severe. Therefore, using PVF to estimate application error resilience and inform the protection mechanisms often leads to overprotecting applications, thereby resulting in unnecessary performance and energy overheads. By distinguishing between crashes and other failures, ePVF allows protection techniques to better focus on the program's bits that if corrupted, can potentially cause SDCs.

There are two challenges in identifying crash-causing bits. Firstly, crashes are caused by OS and architecture-specific factors, which we need to understand and model. Secondly, the crash-related OS state varies during program execution

(e.g., segment boundaries for segmentation faults). Hence we need a dynamic model for predicting whether a particular fault will cause a crash. We find that the majority of crashes are caused by illegal memory addressing, and that by capturing and reasoning about the state of a program's memory segments in a platform-specific manner, we can accurately find almost all crash-causing bits. We therefore extend the original PVF estimation to estimate the ranges of values that may generate crashes, propagate them on the backward slices of the loads and stores, and efficiently compute the set of bits that can result in program crashes.

Contributions. To the best of our knowledge, our work is the first to consider the effects of different failure modes through a PVF-like analysis for the goal of analyzing a program's resilience to SDCs. Our work is also the first to close the gap between analytical models such as PVF, and experimental assessment techniques such as fault injections. This paper:

1) Develops a *crash model* (§III-D) to predict which faults in a program cause a crash, a *propagation model* (§III-C) to reason about propagating ranges of crash-causing bits in the program's dependence graph, and integrates them with the PVF methodology;
2) Implements the method in the LLVM compiler [9] and its intermediate representation (IR) which offers the ability to support multiple platforms and architectures;
3) Evaluates the accuracy of the proposed ePVF method vis-a-vis fault injection (§IV) at the same abstraction level using LLFI, an open-source fault injector [10]. It finds that ePVF estimates crash-causing bits with 89% recall and 92% precision, when evaluated over a set of ten benchmarks. More importantly, the number of vulnerable bits estimated by the ePVF analysis is lower than that estimated by the standard PVF analysis by 61% on average. Thus ePVF leads to a tighter estimate of the SDC rate, and a close estimate of the crash rate compared to fault injection.
4) Demonstrates the utility of the ePVF analysis through a case study involving selective instruction-level protection for SDC mitigation (§V). We find that the SDC rate reduction achieved using ePVF is, on average, 30% better than that achieved by hot-path duplication (i.e., duplicating the most frequently executed program paths), for the same performance overhead.

II. Background

This section offers background information on error resilience, the dependability metric we estimate (§II-A), past work on estimating it through fault-injection (§II-B) and vulnerability analysis techniques (§II-C), the abstraction level that our technique works at (§II-D) and our fault model (§II-E).

A. Dependability Metric: Error Resilience

Not all faults in a program result in failures due to masking at different layers of the system stack. As we focus on software resilience techniques, we do not consider hardware masking [11], but only take into account faults passing the hardware and seen by the software. This is in line with other work in this area [12]–[15].

In the context of this work, *we define error resilience as the probability that the application does not have an SDC after a transient hardware fault occurs and impacts the application state.* Note that error resilience does not take into account the probability of a fault occurring and affecting the software (which depends on the base fault rate in the hardware and the application execution time). In §V, we estimate the impact of protection techniques by taking into account their effect on application performance in addition to the resilience.

B. Fault Injection

Traditionally, program error resilience has been estimated through fault injection [16], that is, by introducing faults at various levels of the system stack and observing their outcome. This is a mature and well understood technique, and there are many tools to help automate the process [17]–[19]. Unfortunately, fault injection does not have predictive power in terms of determining the impact of code transformations on vulnerability - thus it is challenging to use it for guiding code optimizations. Further, fault injection campaigns are typically resource consuming, as thousands of faults need to be injected in complete executions of the program, to get statistically significant results. Hari et al. [20], [21] have proposed approaches to reduce the cost of fault injection campaigns by pruning the fault injection space. However, fault injection campaigns are still costly and cannot be used in situations where predictive power is needed to choose between the multiple options available for code optimization. Instead, we need an automated characterization of error resilience that does not use fault injections.

C. Program Vulnerability Factor (PVF)

The Architectural Vulnerability Factor (AVF) of a hardware component is the probability that a fault occurring in the component leads to a visible error in the final output of a set of executed instructions. Mukherjee et al. [22] introduced the Architecturally Correct Execution (ACE) analysis for estimating the AVF of processor structures (e.g., Reorder buffer) based on a dynamic execution trace on a specific microarchitecture. By combining ACE analysis with the raw error rate of a processor structure and its AVF, microprocessor designers can estimate the FIT (Failures In Time) of each processor structure and take appropriate action in the design stage. However, AVF is intricately tied to the microarchitectural design of a processor, and cannot be used to reason about software resilience in isolation.

Sridharan et al. [8] separate the hardware-specific component of AVF from the software-specific component: the Program Vulnerability Factor (PVF). They show that the PVF can be used to explain the error resilience behaviour of a program independent of the processor. Moreover, they show that by using techniques that are used in computing the PVF [8], programmers are able to pinpoint the vulnerability of different segments of the program, and gain insights for designing

application-specific fault tolerance mechanisms. However, the key drawback of PVF is that it does not distinguish between different kinds of failures, i.e. crashes and SDCs.

PVF abstracts out timing information and includes only the relative instruction order in the instruction flow. This makes the process of computing the PVF largely microarchitecture neutral, thus making it a function of the program and the architecture alone (when executed with a specific input). Sridharan et al. [8] estimate the PVF of an architectural resource 'R' as the ratio between the Architecturally Correct Execution (ACE) bits in the resource when executing the set of instructions I in a trace and the number of total bits involved in R (i.e., B_R) (Equation 1). The ACE bits are the bits in which a fault would potentially affect the correctness of the execution of the instructions in I.

$$PVF_R = \frac{\sum_{i=0}^{I} (ACE\ bits\ in\ R\ at\ instruction\ i)}{B_R \times |I|} \quad (1)$$

D. LLVM IR

LLVM is a compiler infrastructure that provides support for different hardware platforms [9]. The key component of LLVM is its intermediate representation (IR), an assembly-like language that abstracts out the hardware and ISA-specific information. Our methodology is built on LLVM IR level, and we choose to work at this abstraction level for the following reasons:

i) LLVM abstraction level (i.e., LLVM IR) offers an uniform representation of a program; hence the ePVF methodology is architecture neutral and easy to port across different architectures/ISAs, thereby eliminating the influence of architecture or ISA-specific factors.

ii) LLVM IR maps closely to constructs of a program and preserves source-level program properties, which makes it easy to help understand the inherent fault masking.

iii) LLVM infrastructure provides extensive support for program analysis and instrumentation. Prior work focusing on selective duplication techniques [13], [14] uses the LLVM compiler for both static and dynamic analysis, and program instrumentation.

To validate our methodology, we use fault injection experiments at the same abstraction level (LLVM IR) as our ePVF implementation using LLFI [10]. Cho et al. [23] have found that high-level fault injections can directly model a subset of system-level behaviors caused by transient faults. However, our focus (and the focus of the original PVF paper) is on the subset of faults that do manifest in architecturally visible state (e.g., registers) - these faults can be modeled by high-level fault injections.

E. The Fault Model

Hardware faults can be broadly classified as transient or permanent. Transient faults usually are "one-off" events and occur non-deterministically, while permanent faults persist at a given location. We consider transient faults that occur within the processor (i.e., register file, ALUs). We do not consider fault in main memory, since most servers used in HPC applications are protected via ECC and hence do not generally require additional software-based resilience mechanisms.

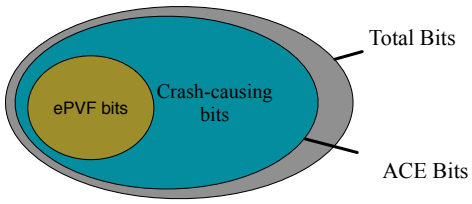


Fig. 1: Venn diagram that highlights the crash-causing and the ePVF bits that the ePVF methodology identifies as a subset of ACE bits. SDC-causing bits will be a subset of the ePVF bits.

We use the single-bit-flip model to represent transient faults as in other related work [4], [5], [13], [14], [24]. Our technique can be easily extended to multiple-bit flips. In recent work, Cho et al. [23] find that low-level hardware faults manifest as both single- and multiple-bit flips at the application level. However, recent work [25], [26] has shown that the difference between single- and multiple-bit flips occurring in program states is marginal in terms of their impact on SDCs. Therefore for this study, we stick to single bit flips, as SDCs are our main concern.

III. ePVF Methodology

Our goal is to obtain a comprehensive estimate of the program resilience that does not entail the full-blown costs of fault injection. We aim to adapt the PVF methodology (see §II) for this purpose. As pointed out earlier, PVF does not distinguish between crashes and SDCs, and hence is overly conservative, as SDCs are the main concern in practice.

Definition: We define ePVF by analogy to PVF as the ratio of *non-crashing* ACE bits over the total bits involved. Figure 1 shows the ePVF bits: they are a subset of all ACE-bits, and a superset of the SDC-causing bits. It is a superset because not all non-crashing bits cause SDCs.

$$ePVF_R = \frac{\sum_{i=0}^{I} (ACEBits\ -\ CrashBits\ in\ R\ at\ instruction\ i)}{B_R \times |I|} \quad (2)$$

Methodology Overview: At a high level, the ePVF methodology consists of three major components (Figure 2): (i) Base ACE analysis to estimate all the vulnerable bits of the program (§III-A); (ii) a *crash model* to identify the ranges of bit-level faults that cause crashes (§III-D); and (iii) a *propagation model* that propagates these ranges along the backward slices of each operand (§III-C). This methodology is supported by our identification of incorrect memory addressing as the most common cause for crashes (§III-B).

A. Base ACE Analysis

ACE analysis is used to determine the set of all bits in an architectural resource (e.g., a register file) that are not masked and can affect application's final state. The basic idea is to first identify the instructions that are responsible for the output of the program (called output instructions),

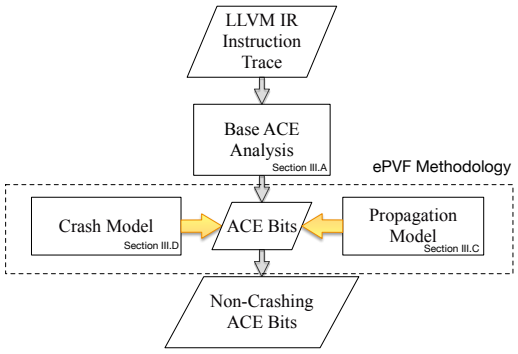


Fig. 2: The overall workflow of the ePVF methodology to compute the non-crashing ACE (ePVF) bits.

and then find all the instructions in their backward slice. We use the program's dynamic dependency graph (DDG) [27] to keep track of the data dependencies among the program's instructions. The DDG is a representation of data flow in the program, and is constructed based on the program's dynamic instruction trace [27]. In the DDG, a vertex can be a register, a memory address or even a constant value. An edge records the instruction (i.e., an operation) and links source operand(s) to destination operand(s).

We implement the DDG analysis at the LLVM compiler's intermediate representation (IR) level. Note that since the LLVM IR abstracts out the hardware/ISA-specific information, it contains an infinite number of virtual registers [1]. As a result, at this level there is no notion of a register file. We model the architectural resource as the set of virtual registers used in the IR of a program. This definition also matches our fault injection experiments as only activated faults are considered.

One further issue is deciding how to express register-based memory addressing instructions. This needs special care as it is common to have the same register to store many different memory addresses, or different registers store to the same address. Since we create new DDG nodes for each newly written memory address, it is also common for a register to store multiple uses of the same memory address. To handle these cases, we create an edge in the DDG to link the memory address used and the register. This edge is *virtual* to differentiate this case from direct data data dependencies.

Running example: Figure 3 shows a small portion of the DDG constructed from a dynamic IR instruction trace of the *pathfinder* benchmark [28]. We rename the IR registers for readability. Figure 3a presents the corresponding static instruction in the LLVM IR of the program[2]. Figure 3b illustrates the DDG obtained after executing the static instructions in Figure 3a. Nodes representing memory are labelled with the address values recorded during the run-time. Memory addresses that correspond to the output are highlighted.

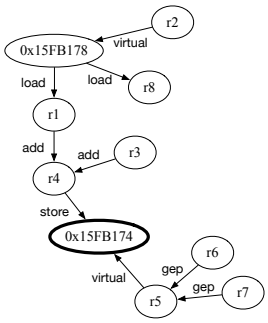
From each memory location that is part of the output, in this

[1]The register allocator will take the physical register file size into account when mapping the virtual registers to physical ones in the compiler backend.

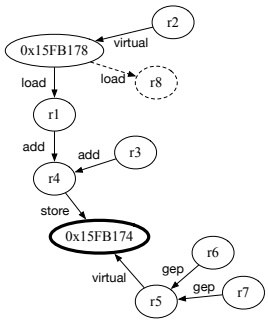
[2]LLVM has a special IR instruction named *getelementptr* (gep) to abstract memory address computations, which corresponds to a combination of several instructions in a assembly language such as MOV and ADD instructions.

```
r1 = load i32* r2, align 4
r4 = add nsw i32 r1, r3
r5 = getelementptr inbounds i32* r6, i64 r7
store i32 r4, i32* r5, align 4
r8= load i32* r2, align 4
```

(a) A small portion of static IR-level representation of the pathfinder benchmark



(b) The DDG constructed based on the dynamic trace resulting from executing the code presented in (a).



(c) The ACE graph used to capture the ACE bits. Note that dynamically dead code is eliminated.

Fig. 3: An example of computing PVF for the register file

case *0x15FB174* (highlighted in Figure 3b), we run a reverse breadth-first search on the DDG that contains all the dependent vertices of *0x15FB174*. This step will exclude the node *r8* as it does not contribute to the output. We call the resulting graph the ACE graph (Figure 3c). Then the total ACE bits are calculated as (the size of each operand is defined in IR):

$$ACE\ Bits_{used\ registers} = \sum_{i=1}^{7} Bits\ in\ R_i$$
$$= 32 + 64 + 32 + 32 + 64 + 64 + 64 = 352$$

To compute the PVF we also need to compute the total bits for used registers, summing the total bits used in operations within this sequence of instructions.

$$Total\ Bits_{used\ registers} = \sum_{i=1}^{8} Bits\ in\ R_i$$
$$= 32 + 64 + 32 + 32 + 64 + 64 + 64 + 64 = 416$$

Then, the PVF of used registers for this example is:

$$PVF_{used\ registers} = \frac{ACEBits_{used\ registers}}{TotalBits_{used\ registers}} = 0.846$$

B. Finding the Crash-causing Bits

We aim to identify the bits that cause the program to crash (i.e., lead to hardware exceptions), and subtract these bits from the overall ACE bits. To this end, it is important to determine the types of crashes we observe in practice and their relative frequencies. We perform a fault injection experiment by injecting faults into ten benchmark applications (§IV describes our fault injection methodology and benchmarks).

We observe four types of exceptions resulting in crashes (Table I). Table II shows their relative frequencies. Our results show that *segmentation faults are the predominant source of*

TABLE I: Types of exceptions resulting in crashes

Type	Description
Segmentation fault (SF)	Memory access that exceeds the legal boundary of a memory segment
Abort (A)	Programs aborted by themselves or OS
Misaligned memory access (MMA)	Memory accesses are not aligned at four bytes
Arithmetic errors (AE)	Division by 0, Overflow etc.

TABLE II: Relative crash frequency analysis for each benchmarks

Benchmark	Types of crashes (%)			
	SF	A	MMA	AE
hotspot	97.6%	0.0%	2.3%	0.1%
bfs	98.8%	0.0%	0.7%	0.5%
kmeans	100.0%	0.0%	0.0%	0.0%
nw	99.6%	0.0%	0.4%	0.0%
pathfinder	99.9%	0.1%	0.0%	0.0%
lud	100.0%	0.0%	0.0%	0.0%
srad	96.0%	0.0%	4.0%	0.0%
mm	99.8%	0.1%	0.1%	0.0%
particlefilter	100.0%	0.0%	0.0%	0.0%
lulesh	99.0%	1.0%	0.0%	0.0%

crashes with a 99% average frequency and a 96% minimum over all benchmarks. This observation suggests that, for the class of workloads with similar properties as these benchmarks, we only need to model the mechanisms that generate segmentation faults to identify almost all crash-causing bits. We note that other workloads, architectures, or operating systems may change these precise findings, but a similar methodology can be followed.

Segmentation faults result from memory access violations. Although different operating systems may implement violation detection mechanisms in different ways, segmentation faults are determined based on checking memory accesses against segment boundaries.

There are two main challenges in determining which bit flips would lead to a segmentation fault: first, we need to find the ranges of the bits that, if flipped, would result in an out-of-bounds memory access. This includes both faults in the memory instructions themselves (i.e., load and store), and faults in their backward slices used for memory addressing. Second, we need to predict if an incorrect memory access will generate an access violation. To this end, all segment boundaries at the time of the memory access need to be known.

To overcome the first challenge, we implement an algorithm that propagates the ranges of crash-causing bits along the backward slice of the memory access operation (§III-C). To overcome the second challenge, we instrument the program to embed a probe for each memory access and capture all the dynamic segment boundaries (§III-D).

C. The Propagation Model

We model fault propagation for crash-causing faults starting from a memory addressing operation and going backwards in the DDG. This analysis is triggered each time a load/store instruction is encountered during the iteration over the ACE graph (i.e., the subgraph that contains all ACE nodes in the DDG) to compute ACE bits. The aim is to find all bits that can generate an out-of-bound address on the backward slice of the memory address calculation. The model assumes that only one fault happens during the course of a program. (§II-E).

The propagation model consists of two algorithms. Algorithm 1 describes when and how the propagation model is triggered. It consists of two procedures: ITERATE_OVER_ACE_GRAPH and CRASH_CALC. The ITERATE_OVER_ACE_GRAPH procedure takes the ACE graph as input and iterates over the vertices in the ACE graph. When it reaches a load/store instruction (line 3), the backward slice for the address used in the load/store instruction is calculated (line 5) and passed to the procedure CRASH_CALC (line 6). Inside CRASH_CALC, all the instructions along the backward slice are visited and, for each instruction, the ranges for crash-causing bits in operands are computed by invoking GET_RANGE_FOR_CRASH_BITS.

Algorithm 1 Iterates over the ACE graph and invokes CRASH_CALC whenever a load or store instruction is encountered

```
1: procedure ITERATE_OVER_ACE_GRAPH(ACEGraph)
2:     for all inst in ACE Graph do
3:         if inst.opcode == load/store then
4:             backwardslice =
5:             CALCULATE_BACKWARD_SLICE(inst)
6:             CRASH_CALC(backwardslice)
7:         end if
8:     end for
9: end procedure
10: procedure CRASH_CALC(backwardslice)
11:     for all inst in backwardslice do
12:         GET_RANGE_FOR_CRASH_BITS(inst)
13:     end for
14: end procedure
```

Algorithm 2 Calculates the range of the crash-causing bits for memory access instructions based on the backward slice of the address used

```
1: procedure GET_RANGE_FOR_CRASH_BITS(inst)
2:     crashing_bits ← 0
3:     global crash_bits_list
4:     oplist ← inst.source_operands
5:     if inst == load/store then
6:         (max, min) = CHECK_BOUNDARY(inst.addres)
7:         crash_bits_list[inst.address] = (max, min)
8:     else
9:         (max, min) = crash_bits_list[inst.dest_opernd]
10:    end if
11:    for all op in oplist do
12:        (new_max, new_min) = lookup_table(op, inst)
13:        crash_bits_list[op] = (new_max, new_min)
14:        crashing_bits += bits that make the value of op outside (new_max, new_min)
15:    end for return crashing_bits
16: end procedure
```

The second algorithm (Algorithm 2) consists of the procedure GET_RANGE_FOR_CRASH_BITS that models the execution of each instruction along the backward slice to calculate the range for crash-causing bits. Specifically, for a load/store instruction, the crash model is invoked (CHECK_BOUNDARY) to determine the range of bits that generate an out-of-bound memory access (at line 6) to obtain a range of crash-causing bits for the destination register (line 9). Then, for each source operand, the procedure calculates the range for the

crash-causing bits by taking into account the range of the corresponding destination operand and the semantics of that instruction (line 11 to line 15). The semantics of the instruction are determined by the *lookup_table* function in line 13.

Table III shows the common instruction types encountered on the backward slice of a memory address calculation and how the *lookup_table* is used to compute the range for each operand. We assume that all values of operands are positive integers. The algorithm propagates these ranges along the backward slice by storing them in the **CRASHING_BIT_LIST** for further reference by the next instructions, as shown in line 7.

We explain the details of these algorithms using our running example. In Figure 3b, *r5* stores the address *0x15FB174* for the instruction *store i32 r4, i32* r5, align 4*. Suppose our bound-determination technique (described in the next subsection) returns the bound (0x15FB800, 0x15FA000), meaning that addressing outside this bound will generate a segmentation fault. The ACE graph indicates that *r5*, *r6* and *r7* are used in addressing (or computing the address). Together, these three registers belong to the instruction *getelementptr* in LLVM. The instruction semantics are based on row 6 of the Table III - ranges are obtained by applying the corresponding equations.

$$r5_{.value} = r6_{.value} + sizeof(r6)_{.type}) \times (r7)$$
$$max_{r6} = max_{r5} - sizeof((r6)_{.type}) \times (r7)$$
$$min_{r6} = min_{r5} - sizeof((r6)_{.type}) \times (r7)$$

The range of *r6* can be computed as follows: min: *0x15FB800* - 4*1 = 0x15FB7FC; max: *0x15FA000* - 4*1 = 0x15F9FFC. The resulting range (0x15FB7FC, 0x15F9FFC) of r6 is stored into **CRASHING_BIT_LIST** as the reference for operands on the backward slice of *r6*, if any. Similarly, we can compute the range for register *r7* and for other registers in the backward slice.

Algorithm 3 Obtains the boundary of the segment

```
1: procedure CHECK_BOUNDARY(inst.address)
2:     global crash_bits_list
3:     max ← 0
4:     min ← 0
5:     vma_start = locate_segment_start(inst.address)
6:     if inst.address ⊂ stack&&&vma_start < ESP − 65536 − 128 then
7:         min ← ESP − 65536 − 128
8:     else
9:         min ← vma_start
10:    end if
11:    vma_end = locate_segment_end(inst.address)
12:    max ← vma_end
13:    crash_bits_list[inst.address] = (max, min)
       return (max, min)
14: end procedure
```

D. Crash Model

The goal of the crash model is to determine the ranges of the addresses for which a memory access will trigger a segmentation fault. While this is platform-specific i.e., specific to the hardware and operating system (OS) on which the program is running, the technique described here can be adapted to any architecture that uses memory segmentation. This includes most modern architectures such as x86 and ARM.

Algorithm 3 describes how to compute the range of allowable addresses for a memory segment. It undertakes two main tasks: (i) obtains the boundary of the memory segment through the underlying OS interface (see line 5 and line 11) and, (ii) determines the run-time range of valid memory addresses for each load/store instruction (from line 6 to line 10). We explain the steps below.

Obtaining the segment boundaries. Modern operating systems organize process memory over multiple segments (e.g., text, data, heap, stack). To identify the boundary of each segment we instrument the program to embed a run-time probe that probes the "/proc" system of Linux to record the segment boundaries at each load and store instruction.

Determining allowed ranges. Once we determine segment boundaries at the time of each load or store, we need to determine which accesses would result in segmentation faults. We initially hypothesized that all accesses outside segment boundaries will trigger a segmentation fault. Unfortunately, this is not the case, as we found through a fault injection experiment: a segmentation fault occurred only for about 85% of the out-of-segment accesses we generated. The remaining 15% of accesses did not result in a segmentation fault even though they were outside the segment boundaries, suggesting that our hypothesis was incorrect.

To better understand this behaviour, we studied the source code of the Linux kernel in our platform, an x86-based machine (Figure 4)[3]. The "vma_start" and "vma_end" indicate the start and end addresses of Linux **v**irtual **m**emory **a**rea (vma) and the "addr" indicates the memory address used. In the code, the label "common case" shows the kernel code for when *addr* is within the valid bound. Note that if *addr* is within the last page of the stack, Linux will add one page below the current last page until the 8 megabyte limit is reached. The label "case I" is when *addr* is smaller than the "vma_start" and is still bigger than the $(ESP - 64KB - 128B)$. Linux treats such an address as valid and will expand the stack for it. However, if *addr* is smaller than $(ESP - 64KB - 128B)$, a segmentation fault occurs. The label "case II" occurs when *addr* is greater than the"vma_end", and will result in a segmentation fault.

Thus, for a non-stack segment, Linux determines the boundaries using its "vma_start" and "vma_end", while for stack segments, it compares the "vma_start" with the current stack pointer plus an offset to determine the lower bound of the stack. If *addr* is inside an invalid memory region, a segmentation fault occurs.

We implement our crash model to mirror the handling of these different cases. *Upon re-evaluating the accuracy of the model through the same fault injection experiment, we find that we can now accurately predict crashes for over* 99.5% *of the accesses, pointing to the correctness of the crash model.* We use this crash model in our experiments.

[3]Similar code can be found for both x86 and PowerPC kernel versions.

TABLE III: Range calculation operations that commonly occur on the backward slice of memory addresses

No.	Opcode	Operand	Semantic	Range Calculation for operands
1	add	dest, op1, op2	dest = op1 + op2	Max(op1) = Max(dest) - op2 ; Min(op1) = Min(dest) - op2
				Max(op2) = Max(dest) - op1; Min(op2) = Min(dest) - op1
2	sub	dest, op1, op2	dest = op1 - op2	Max(op1) = Max(dest) + op2 ; Min(op1) = Min(dest) + op2
				Max(op2) = op1 - Min(dest) ; Max(op2) = op1 - Max(dest)
3	mul	dest, op1, op2	dest = op1 * op2	Max(op1) = Max(dest)/op2 ; Min(op1) = Min(dest)/op2 (if op2 != 0)
				Max(op2) = Max(dest)/op1; Min(op2) = Min(dest)/op1
4	div	dest, op1, op2	dest = op1 / op2	Max(op1) = Max(dest)*op2 ; Min(op1) =Min(dest)*op2
				Max(op2) =Max(op1)/dest ; Min(op2) = Min(op1)/dest
5	getelementptr	dest, op1, op2	dest = op1 + sizeof(op1.type)*op2	Max(op1) = Max(dest) - sizeof(op1.type)*op2 ; Min(op1) = Min(dest) - sizeof(op1.type)*op2
				Max(op2) = (Max(dest) - op1)/sizeof(op1.type) ; Min(op2) = (Min(dest) - op1)/sizeof(op1.type)
6	srem	dest, op1, op2	dest = op1 % op2	Max(op1) = Max(for all bits in op1: bitflip(op1)%op2 >Max(dest) and bitflip(op1)%op2 <Min(dest))
				Min(op1) = Min(for all bits in op1: bitflip(op1)%op2 >Max(dest) and bitflip(op1)%op2 <Min(dest))
				Max(op2) = Max(for all bits in op2: op1%bitflip(op2) >Max(dest) and op1%bitflip(op2) <Min(dest))
				Min(op1) = Min(for all bits in op2: op1%bitflip(op2) >Max(dest) and op1%bitflip(op2) <Min(dest))
7	bitcast	dest, op1	dest = op1	Max(op1) = Max(dest); Min(op1) = Min(dest)

```
/* "vma" means virtual memory area it is an abstraction
    used in Linux for memory segments. */
/* addr represents the accessed memory address */
/* regs->sp stores the current stack pointer */
Common case:
    if vma_start <= addr <= vma_end:
        // everything is fine
        addr &= page_mask;
        // for stack:
        if addr == the last page of the vma:
            expand_stack(vma, addr)
Case I:
    // only for stack:
    if vma_start > addr:
        if addr + 65536 + 32 * sizeof(unsigned long)
           < regs->sp:
            // SEGFAULT
        else:
            expand_stack(vma, addr)
Case II:
    if vma_end < addr
        // SEGFAULT
```

Fig. 4: Linux kernel implementation for determining which memory accesses result in segmentation faults. Linux kernel version: 3.15. File locations: mm/ and arch/x86/mm.

IV. Evaluation

Our evaluation is guided by the following four questions:

Q1 How accurate is the ePVF methodology when predicting the bits in which faults lead to program crashes?

Q2 How close are estimated crash rates to the actual crash rates obtained through fault injection?

Q3 Can the methodology be used to obtain a significantly tighter estimate for the SDC rate than the conventional PVF methodology?

Q4 How fast and scalable is the ePVF analysis?

A. Experimental Setup

Benchmarks. We evaluate the ePVF methodology on ten HPC benchmarks (Table IV): these include eight OpenMP-based scientific applications picked from the Rodinia benchmark suite [28], our basic implementation of the matrix multiplication kernel, and Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (i.e. lulesh) [29], [30], a DOE proxy application. The applications range from 100 lines of code (*mm*) to 3000 lines of code (*lulesh*). Note that we target HPC applications and hence we do not consider SPEC programs.

Platform. All of our experiments are conducted on a machine with a x86 CPU running at 2.67GHz and Linux v3.15.

TABLE IV: Benchmarks used and their complexity (lines of C code).

Benchmark	Domain	LOC
LULESH (lulesh)	Physics Modelling	3,000
Particle Filter (particlefilter)	Medical Imaging	602
Speckle Reducing Anisotropic Diffusion (srad)	Image Processing	388
Needleman-Wunsch (nw)	Bio informatics	285
HotSpot (hotspot)	Physics Simulation	272
LAVA Molecular Dynamics (lavaMD)	Molecular Dynamics	218
Breadth-First Search (bfs)	Graph Algorithm	203
LU Decomposition (lud)	Linear Algebra	174
PathFinder (pathfinder)	Grid Traversal	135
Matrix Multiplication (mm)	Linear Algebra	100

Fault injection. To build a ground truth, we use the publicly available, open-source LLFI fault injector [10] to inject faults at the LLVM Intermediate Code (IR) level. We inject faults into the source registers for the executed instructions to emulate faults in the used registers of the instructions, and hence all faults are activated as they are used in the instruction. Only one fault is injected in each run. We perform over 3,000 fault injection runs for each benchmark. The 95% confidence levels are reported as error bars for statistical significance.

B. Q1: What is the Accuracy of ePVF Methodology?

To answer this question, we evaluate ePVF *recall* and *precision*. We use fault injection experiments to obtain the ground-truth, and compare the outcome of each fault injection experiment with the outcome predicted by the ePVF methodology. Figure 5 shows the outcome (i.e., SDC, crash, hang and benign fault) frequency for each benchmark: crashes are the dominant outcome, on average, 63% of injections result in crashes, while 12% result in SDCs, and less than 1% in hangs. The dominance of crashes highlights the importance of separating the crash-causing bits from the other failure bits.

Recall. We define *recall* as the ratio of crash runs that our model predicts correctly to be crashes, to all fault injection runs that lead to crashes in reality. To estimate recall, for each fault injection run that leads to a crash, we record the instruction counter and the register that the fault is injected into, as well as the bit that was flipped. We then run the crash and propagation models for the entire program and check whether the location appears in the final $crash\_bits\_list$ (described in Algorithm 2) that stores the bits that lead to a crash if the bit is corrupted.

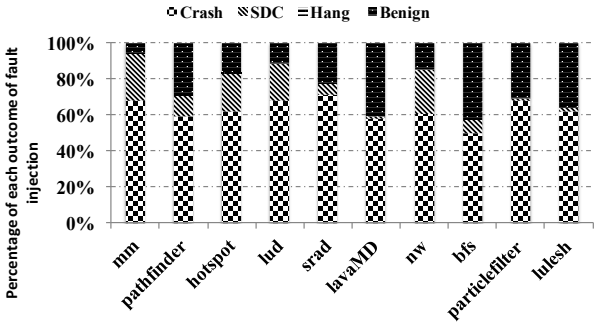


Fig. 5: Fault injection results for each benchmark.

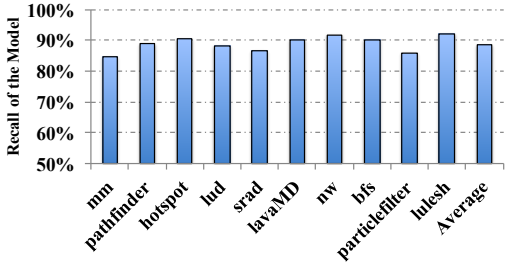


Fig. 6: Recall for the crash bits predicted using the ePVF methodology.

Figure 6 presents the recall for each benchmark. Overall, our methodology achieves an average of 89% recall across the ten benchmarks (ranging from 85% to 92%). We manually analyzed the crash-causing bits that were not identified as crashes by ePVF methodology. The main reason is that our validation technique introduces approximations due to non-determinism in execution environment: the segment boundaries may be slightly shifted. As a result, it cannot be guaranteed to execute fault injection runs with exactly the same environment, particularly the same memory allocation and the profiling. Through manual verification we found that, depending on benchmark, this factor accounts for 92% to 99% of incorrect predictions.

Precision. We define *precision* as the ratio of the number of *correctly* predicted crash-causing bits to the total number of predicted crash-causing bits. To estimate precision, we randomly choose over 1,200 different bits from those identified by the model as crash-causing (i.e., appear in the CRASHING_BIT_LIST), and perform a targeted fault injection experiment. Similar to the recall study, this time for each bit, we specify the dynamic instruction and the register to inject the fault into, as well as the bit that should be flipped. Precision is calculated as the number of observed crashes over the total number of fault injections performed.

Figure 7 shows the results of the evaluation. The average precision across all benchmarks is 92% (ranges from 86% to 98%). As in the case of recall, after manual inspection we have confirmed that the main reason for not hitting 100% precision is the difference between the run-time and modeled environments, i.e., non-deterministic memory allocation.

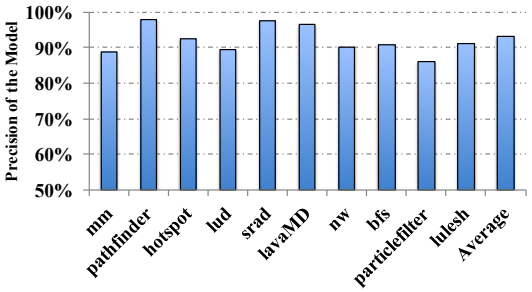


Fig. 7: Precision for the crash bits predicted using the ePVF methodology.

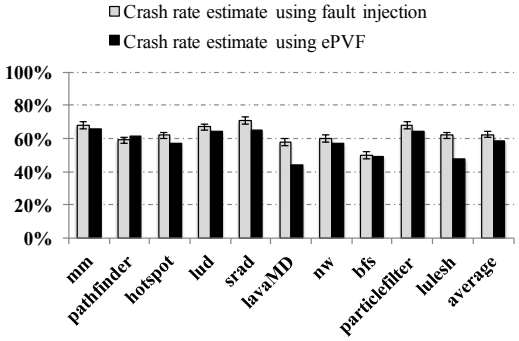


Fig. 8: The crash rates estimates using ePVF (right bars) and using fault injection experiments (left bars) are close. For fault injection experiments, the error bars indicate the 95% confidence intervals.

C. Q2: How close are the crash rates estimated using ePVF and fault injection?

The ePVF methodology is able to identify crash-causing bits with high accuracy. This can be used to estimate the crash rate of a program as the fraction of crash-causing bits over the total number of bits in an application. Such an estimate can be important for techniques that use crash rates to determine the level of protection to be provided, e.g., choosing the checkpoint interval.

Figure 8 shows that estimating crash rates this way is a good approximation for crash rates obtained through fault injection experiments. The differences are within or close to the 95% confidence interval bounds, except for *lavaMD* and *lulesh*. The reason the crash rate predictions are off for these two applications is that ePVF calculates the crash bits only based on the ACE graph, which contains 70% and 80% of the whole DDG for *lavaMD* and *lulesh* respectively. On the other hand, the fault injection uses the full program execution corresponding to the whole DDG.

D. Q3: Does ePVF lead to a tighter estimate of SDC rate than the original PVF?

We have shown that the ePVF methodology can accurately estimate the crash bits of an application. We now ask whether it can lead to better SDC rate estimates. As explained earlier, ePVF provides an upper bound (i.e., overestimate) for the SDC rate like PVF does. We compare the tightness of these two upper bounds.

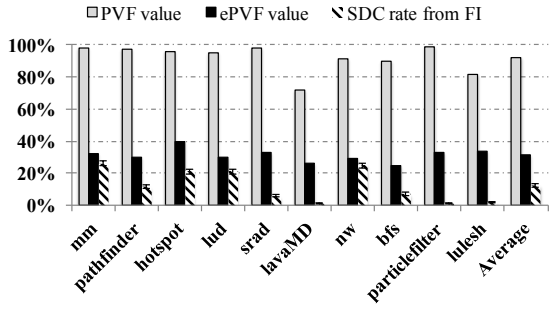


Fig. 9: ePVF (center bars) offers a much better upper bound estimate for the SDC rate (right bars) than the original PVF methodology (left bars). For SDC rates, error bars represent 95% confidence intervals.

TABLE V: Number of nodes in the ACE graph and time taken by the ePVF analysis for each benchmark

Benchmarks	# of Dynamic IR instructions	ACE nodes	Modelling time (s)
hotspot	954,920	1,102,265	14,400
pathfinder	839,163	967,836	18,000
mm	464,438	597,604	3,987
particlefilter	352,866	479,994	3,956
nw	376,022	453,998	3,800
lulesh	322,738	319,253	953
bfs	274,170	269,019	900
lud	75,543	93,089	205
srad	72,041	91,385	172
lavaMD	17,814	16,779	30

Figure 9 shows the original PVF and the ePVF values for the ten benchmarks. The original PVF ranges from 71% to 98%, with an average of 92%. In contrast, the ePVF estimate ranges from 25% to 40%, with an average value of 31%. The average difference between PVF and ePVF is 61%, ranging from 45% to 67% depending on the benchmarks.

Figure 9 also shows, for each benchmark, the SDC rate obtained through the fault injection experiments described earlier. The SDC rate ranges from 1 to 25% depending on the benchmark, with an average value of about 12% across benchmarks. *ePVF significantly lowers the upper bound of estimated SDC vulnerability of a program.* The above evaluation suggests that our technique has higher predictive power than the original PVF analysis to understand the SDC behaviour of a program (we demonstrate that this can be used in practice in §V). That said, there is still room for a tighter bound as we will discuss in §VI.

E. Q4: How fast is the ePVF analysis?

Table V shows, for each benchmark, the number of dynamic LLVM IR instructions, the number of nodes in the ACE graph, and the total time to compute ePVF. The running time ranges from less than a minute (*lavaMD*) to 5 hours (*pathfinder*). As expected, the time taken correlates with the ACE graph size. We also measured the time spent by various parts of the ePVF analysis: most time is spent in the crash and propagation models.

We discuss scalability in detail in Section §VI. Here we propose an optimization to reduce the time to compute ePVF, based on sampling the ACE graph. This approach is based on

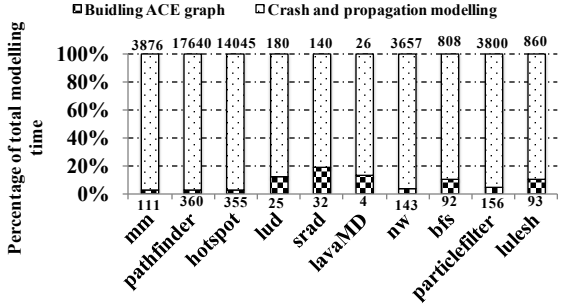


Fig. 10: Breakdown of execution time between graph construction (bottom bar) and running the crash and propagation models (top bar). Labels on bars present absolute time in seconds.

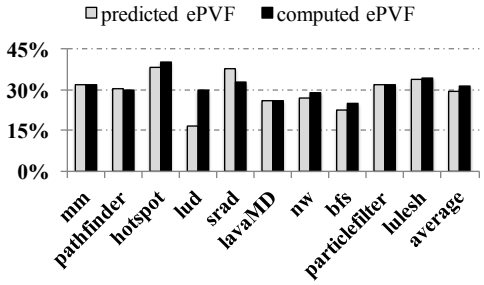


Fig. 11: The predicted ePVF value based on sampling only 10% of the ACE graph and ePVF computed based on the entire graph are close.

the intuition that many HPC applications consist of repetitive program states and patterns, and hence a small sample of the ACE graph will be representative of the overall application behaviour. Since a dynamic instruction trace preserves the temporal ordering of the instructions executed by the program, the output nodes in the ACE graph can be ordered based on their presence in the trace. To validate if the sampling works, we pick the first $p\%$ of the output nodes, and based on the resulting partial ACE graph we estimate ePVF. For regular applications, we find that there is a strong linear relationship and we can linearly extrapolate the partial ePVF to the entire application and thus estimate the overall ePVF accurately.

Figure 11 shows the extrapolated ePVF values based on analyzing only 10% of the ACE graph. As can be observed, for most benchmarks, the extrapolated ePVF values are a good approximation for the overall ePVF: on average the error is less than 1%, suggesting that these programs exhibit repetitive behaviors as we expected.

Importantly, we can also estimate whether an application displays repetitive behaviours and thus whether the ACE-graph sampling be useful, without completing the full ACE analysis. To demonstrate this, we randomly select multiple small subsample of the ACE graph nodes (each 1%) and compute for each of them the ePVF estimates. The normalized variance is relatively low for benchmarks with repetitive behaviours (e.g., 0.6 for *lavaMD* and 0.04 for *particlefilter*), but high for applications where the ACE-graph sampling technique does not offer high accuracy (e.g., 1.9 for *lud*).

V. Case Study: Selective Duplication

To demonstrate the practical usability of the ePVF methodology to improve application resilience, we use ePVF to guide a selective instruction duplication technique to protect against SDCs. The intuition is that a technique that prioritizes protecting instructions with high ePVF values will offer good SDC protection as the faults occurring in crashing bits are unlikely to lead to SDCs. To establish a baseline, we compare the SDC rate of a program protected by duplicating the high ePVF instructions, with that protected by duplicating the hot paths of the program. Prior studies [25], [31] have shown that protecting hot paths is an effective technique (i.e., instructions on the top 20% of most executed paths are responsible for most of the SDCs - these constitute the hot paths).

We also attempted to use PVF to drive the choice of instructions to duplicate. However, we found that the PVF values of most instructions are clustered around 1, which means that PVF has little discriminative power to inform the choice of which instructions to protect. As an example, we plot the CDF (Cumulative Distribution Function) of the PVF and ePVF values of every instruction for two benchmark programs, namely *nw* and *lud* in Figure 12. As can be seen in the figure, the CDF for PVF has a sharp spike near 1, while the ePVF values are distributed more evenly throughout the range. Therefore, we did not consider PVF-informed duplication as a comparison point in this study.

To make the comparison fair, we control the performance overhead incurred by both techniques we compare (by controlling the number of instructions we protect and measuring execution time). *Our hypothesis is that for a given performance overhead bound, the ePVF based duplication scheme can offer higher SDC coverage than hot-path duplication*. A full-duplication technique (i.e., duplication of every instruction) would offer 100% detection coverage, but incur significant performance overheads [13], [25]. Hence we do not consider full-duplication technique in this work.

An ePVF-informed protection heuristic We first compute the ePVF value of each dynamic instruction using the equation 3. Then, we compute the ePVF value of all static instructions in the program by averaging the ePVF values of all their dynamic instances, and rank the static instructions in descending order of their ePVF values. We then select the static instruction at the top of the list, and extract its backward slice. Finally, we selectively duplicate the instructions in the slice, and insert a comparison of the duplicated value with the original value following the chosen instruction. Because we need to limit protection within the overhead budget, we measure the performance overhead incurred by duplication. If the performance overhead bound is not exceeded, we choose the next instruction on the list and repeat the procedure. Thus this is a greedy algorithm for choosing instructions to duplicate. We use the same process for hot-path instructions, with the difference that the instructions are ranked in a decreasing order of their execution frequencies instead of their ePVF values.

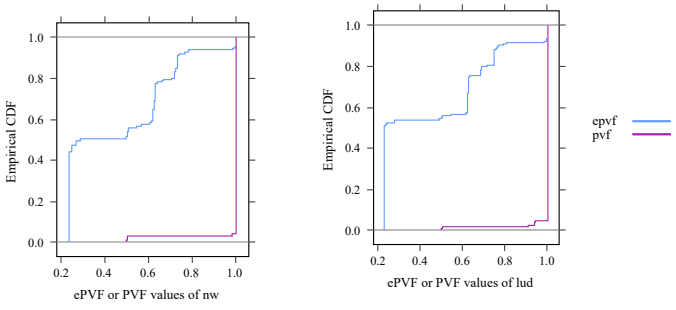


Fig. 12: The figure presents the CDF for the ePVF and PVF values of registers used in each instruction of *nw* (left) and *lud* (right) benchmarks. PVF values for most instructions are clustered around 1 and thus can not inform protection mechanisms based on instruction level protection.

$$ePVF_{inst} = \frac{\sum_{register\ in\ inst} (ACEBits\ -\ CrashBits)}{Total\ bits\ in\ inst} \quad (3)$$

Evaluation Methodology. We evaluate the coverage of the above schemes through fault injection experiments. We only consider the five benchmarks whose SDC rates were higher than 10% in Figure 9 (i.e. *mm, pathfinder, hotspot, lud and nw*) so as to discriminate the effects of the two schemes better. Further, we run the fault injection campaigns with different inputs than the ones we used to get the ePVF values (these are much larger in size) to get stable performance numbers.

Evaluation Results. Figure 13 shows the SDC rate of the original application (no protection), the SDC rate when using hot-path protection, and the SDC rate when using ePVF-informed protection, given a performance overhead bound of 24%[4]. Overall, we find that ePVF based protection does better than hot-path based protection, and reduces the SDC rate from 20% to 7% (geometric mean), while hot-path based protection reduces it to about 10%. Thus, ePVF based protection does 30% better than hot-path based protection, on average across benchmarks, showing that it has better discriminative power than execution frequencies for protection. Further, we find that ePVF-based protection outperforms hot-path based protection for all benchmarks except *hotspot*. This is due to the presence of many control-flow structures in *hotspot* all of which are marked as sensitive by ePVF though they do not cause SDCs.

VI. Discussion

A. Scalability is an important issue as most applications will likely generate ACE graphs with billions of vertices. The ACE-graph sampling technique we describe in §IV-E offers a significant speedup for applications that contain repetitive patterns. We believe that scaling to handle larger applications is a matter of good engineering and not a fundamental barrier for the following reasons. First, the current ePVF infrastructure (including building/processing the DDG) is implemented in Python. A tuned C/C++ implementation will likely be orders

[4]We considered performance overheads of 8%, 16% and 24% as well. We report here only the results using 24% overhead due to space constraints - the qualitative results were similar all cases.

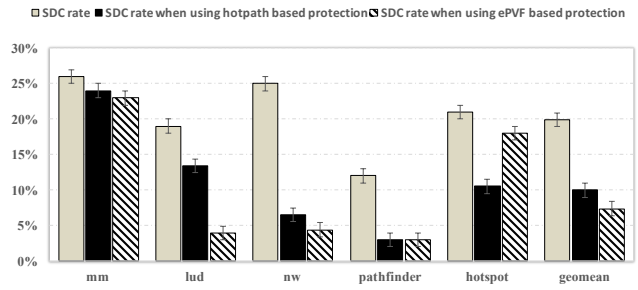


Fig. 13: SDC rates for the original application (left bars) and when using hotpath (center) and ePVF-based protection (right) for an overhead bound of 24%. Error bars present 95% confidence intervals.

of magnitude faster and consume less memory. Second, the most time-consuming portion of the ePVF analysis is running the crash and propagation models. These start from each load/store individually, and search along their backward slices. This process is trivially parallelizable (threads can be assigned to one backward slice each with minimum coordination required). Additionally the work allocated for each thread (i.e., the size of the backward slice) scales sub-linearly with the size of the graph. Third, if the DDG does not fit in memory, it can be partitioned to support the parallel backward slice exploration suggested above.

B. Sources of Inaccuracy. §IV-C shows that ePVF is a much closer upper bound than PVF for the SDC rate of an application. However, ePVF still overestimates the SDC rate, in some cases by a significant amount. This overestimate is generated mainly by the following three factors:

1. Lucky loads: ePVF assumes that any fault that causes a load to deviate from its intended source address (but still stays within the bounds of the program's allocated memory) will lead to an SDC. However, as prior work has found [32], this is not always true. For example, the value loaded from the incorrect address may still be correct, and hence have no effect on the program. The likelihood of this case increases if the value loaded is 0, as memory typically has large areas initialized to zeroes [32].

2. Y-branches: Y-branches are branches that do not affect the outcome of the application even when the program executes the wrong part of a branch due to a fault [33]. The ePVF analysis assumes that all branches lead to SDCs if flipped. However, only about 20% of branch flips lead to SDCs in practice, as prior work has found [33].

3. Application-specific correctness checks: Similar to PVF, the ePVF model, considers as ACE bits all bits that lead to visible changes to the application output. Some of these faults, however, may be characterized as benign by application-specific correctness checks (e.g., based on precision thresholds for floating-point computations).

C. Conservativeness: While ePVF may overestimate the SDC rate, it will never underestimate it (barring the case below). This is because ePVF conservatively labels every non-crash causing operation as potentially leading to an SDC. Being conservative is important as it can drive decisions about how much state to protect in the worst-case for the application. However, in Section IV-B, we found that our implementation of the ePVF methodology may yield false positives i.e., it may identify a failure as a crash when in fact it is an SDC. This occurs because of differences between the program's memory structures in the golden run (on which the ePVF analysis is based) and the fault injected runs. However, the differences are very small in practice (at most 8% in our experiments). A more robust implementation can address this difference.

VII. Related Work

There has been a considerable amount of work on estimating the error resilience of a program either through fault injection, or through vulnerability analysis techniques. The main advantage of fault-injection is that it is simple and allows distinguishing between different failure outcomes, yet has limited predictive power and is slow. The main advantage of vulnerability analysis is that it has predictive power and is faster, but does not distinguish between different kinds of failures. The main question we ask in this paper is whether it is possible to combine the advantages of the two approaches by building an architecture-neutral vulnerability analysis technique to distinguish different failure outcomes, and especially SDCs. Therefore, we use fault injection to gather the ground truth of the error resilience characteristics of an application and compare it with the result of the ePVF methodology.

Biswas et al. [34] separate the overall AVF of processor structures into SDC AVF and Detected Unrecoverable Errors (DUE) AVF by considering whether bit-level error protection mechanisms such as ECC or parity are enabled in those structures. While DUE is similar to the notion of crash in this paper, DUE is defined at the hardware level only and does not consider software-level mechanisms. Further, like AVF, the DUE-AVF is highly hardware dependent.

Bronovetsky et al. [35] use standard machine learning algorithms to predict the vulnerability profiles of different routines under soft errors, to understand the vulnerability of the full applications. However, their technique is confined to linear algebra applications. Lu et al. [36] and Laguna et al. [37] identify SDC-causing code regions through a combination of static analysis and machine learning. However, their technique does not provide foundational understanding behind why some faults cause SDCs and others do not. A common issue with machine learning techniques is that they require extensive training with representative data, which analytical techniques do not.

Finally, Yu et al. [38] introduce a novel resilience metric called *data vulnerability factor* (DVF) to quantify the vulnerability of individual data structures. By combining the DVF of different data structures, the vulnerability of an application can be evaluated. While useful, this technique requires the program to be written in a domain specific language, that is restricted in terms of its expressiveness. Further, DVF does not distinguish between crash-causing errors and other errors.

VIII. Summary

This paper presents ePVF, a methodology to extend the PVF analysis by distinguishing crash-causing bits from the ACE bits as as to get a tighter bound on SDC rate. Our methodology consists of two models: (1) a propagation model to predict the dependent bits of memory address calculations based on a range propagation analysis, and (2) a crash model to predict the platform-specific behaviour of program crashes. We implement the ePVF methodology in the LLVM compiler, and evaluate its accuracy. The results show that ePVF can predict crashes with high confidence (89% recall and 92% precision on average). Further, ePVF significantly lowers the upper bound of the estimated SDC rate of a program (61% on average), compared to the original PVF. Finally, we present a use-case for this methodology: an ePVF-informed selective duplication technique, which leads to 30% lower SDCs than hot-path instruction duplication.

While we have focused on using ePVF methodology for SDC rate estimation and reduction in software, there are two other uses in the future. First, it can be used to determine which architectural structures are more likely to cause SDCs, and selectively protect these structures through hardware techniques such as selective ECC. Second, the ePVF methodology can be used to determine the total number of crash-causing bits in the program and inform a fault-tolerance mechanism for crash-causing faults (e.g. checkpointing).

Acknowledgement

We thank Vilas Sridharan for his insightful feedback on this work. We also thank the reviewers of DSN 2016 for their feedback which helped improve this work. This work was funded in part by Discovery grants from the Natural Science and Engineering Research Council (NSERC).

References

[1] C. Constantinescu, "Trends and challenges in vlsi circuit reliability," in *IEEE MICRO*, 2003.

[2] T. Karnik and P. Hazucha, "Characterization of soft errors caused by single event upsets in cmos processes," *Dependable and Secure Computing, IEEE Transactions on*, vol. 1, no. 2, April 2004.

[3] L. Tan, S. L. Song, P. Wu, Z. Chen, R. Ge, and D. J. Kerbyson, "Investigating the interplay between energy efficiency and resilience in high performance computing," in *IPDPS*. IEEE, 2015, pp. 786–796.

[4] K. S. Yim, C. Pham, M. Saleheen, Z. Kalbarczyk, and R. Iyer, "Hauberk: Lightweight silent data corruption error detector for gpgpu," in *IPDPS*, 2011.

[5] W. Gu, Z. Kalbarczyk, and R. Iyer, "Error sensitivity of the linux kernel executing on powerpc g4 and pentium 4 processors," in *DSN 2003*.

[6] B. Atkinson, N. DeBardeleben, Q. Guan, R. Robey, and W. M. Jones, "Fault injection experiments with the clamr hydrodynamics mini-app," in *2014 ISSREW*.

[7] C. da Lu and D. Reed, "Assessing fault sensitivity in mpi applications," in *Supercomputing, 2004. Proceedings of the ACM/IEEE SC2004*.

[8] V. Sridharan and D. Kaeli, "Eliminating microarchitectural dependency from architectural vulnerability," in *HPCA 2009.*, 2009, pp. 117–128.

[9] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis & transformation," in *CGO*, ser. CGO '04, 2004.

[10] J. Wei, A. Thomas, G. Li, and K. Pattabiraman, "Quantifying the accuracy of high-level fault injection techniques for hardware faults," in *DSN*, June 2014.

[11] A. Meixner, M. Bauer, and D. Sorin, "Argus: Low-cost, comprehensive error detection in simple cores," in *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, Dec 2007.

[12] M. de Kruijf, S. Nomura, and K. Sankaralingam, "Relax: An architectural framework for software recovery of hardware faults," in *ISCA 14*, pp. 497–508.

[13] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, "Shoestring: Probabilistic soft error reliability on the cheap," *SIGPLAN Not.*, vol. 45, no. 3, Mar.

[14] G. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. August, "SWIFT: Software implemented fault tolerance," in *CGO*, 2005, pp. 243–254.

[15] D. S. Khudia and S. A. Mahlke, "Harnessing Soft Computations for Low-Budget Fault Tolerance." *MICRO*, pp. 319–330, 2014.

[16] M.-C. Hsueh, T. Tsai, and R. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75–82, 1997.

[17] D. Stott, B. Floering, D. Burke, Z. Kalbarczpk, and R. Iyer, "Nftape: a framework for assessing dependability in distributed systems with lightweight fault injectors," in *IPDPS 2000*, 2000, pp. 91 –100.

[18] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson, "Goofi: generic object-oriented fault injection tool," in *DSN*, 2001, pp. 83–88.

[19] J. Carreira, H. Madeira, and J. Silva, "Xception: a technique for the experimental evaluation of dependability in modern computers," *Software Engineering, IEEE Transactions on*, Feb 1998.

[20] S. K. S. Hari, S. V. Adve, H. Naeimi, and P. Ramachandran, "Relyzer: exploiting application-level fault equivalence to analyze application resiliency to transient faults," in *ASPLOS 2012*.

[21] S. Hari, R. Venkatagiri, S. Adve, and H. Naeimi, "Ganges: Gang error simulation for hardware resiliency evaluation," in *ISCA 2014*, 2014.

[22] S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, and T. Austin, "Measuring architectural vulnerability factors," in *IEEE MICRO*, vol. 23, no. 6.

[23] H. Cho, S. Mirkhani, C.-Y. Cher, J. Abraham, and S. Mitra, "Quantitative evaluation of soft error injection techniques for robust system design," in *DAC, 2013 50th ACM/EDAC/IEEE*, May, pp. 1–10.

[24] J. Wei and K. Pattabiraman, "BLOCKWATCH: Leveraging similarity in parallel programs for error detection," in *DSN*, 2012.

[25] Q. Lu, K. Pattabiraman, M. S. Gupta, and J. A. Rivers, "Sdctune: A model for predicting the sdc proneness of an application for configurable protection," in *CASE 2014*. New York, New York, USA: ACM Press, 2014, pp. 1–10.

[26] F. Ayatolahi, B. Sangchoolie, R. Johansson, and J. Karlsson, "A study of the impact of single bit-flip and double bit-flip errors on program execution," in *Computer Safety, Reliability, and Security*, 2013, vol. 8153, pp. 265–276.

[27] H. Agrawal and J. R. Horgan, "Dynamic program slicing," in *Proceedings of Programming Language Design and Implementation (PLDI)*, New York, NY, USA, 1990.

[28] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC*, ser. IISWC '09.

[29] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. DeVito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C. Still, "Exploring traditional and emerging parallel programming models using a proxy application," in *IEEE IPDPS 2013*, Boston, USA.

[30] I. Karlin, J. Keasler, and R. Neely, "Lulesh 2.0 updates and changes," Tech. Rep. LLNL-TR-641973, August 2013.

[31] S. K. S. Hari, S. V. Adve, and H. Naeimi, "Low-cost program-level detectors for reducing silent data corruptions," in *2012 42nd DSN*. IEEE, 2012, pp. 1–12.

[32] J. Cook and C. Zilles, "A characterization of instruction-level error derating and its implications for error detection," in *DSN*, 2008.

[33] N. J. Wang, A. Mahesri, and S. J. Patel, "Examining ace analysis reliability estimates using fault-injection," in *ISCA '07*, 2007.

[34] A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S. S. Mukherjee, and R. Rangan, "Computing architectural vulnerability factors for address-based structures," *SIGARCH Comput. Archit. News*, vol. 33, no. 2, May.

[35] G. Bronevetsky, B. de Supinski, and M. Schulz, "A foundation for the accurate prediction of the soft error vulnerability of scientific applications," in *SELSE*, 2009.

[36] Q. Lu, K. Pattabiraman, M. S. Gupta, and J. A. Rivers, "Sdctune: A model for predicting the sdc proneness of an application for configurable protection," in *CASE*, 2014.

[37] I. Laguna, M. Schulz, D. F. Richards, J. Calhoun, and L. Olson, "Ipas: Intelligent protection against silent output corruption in scientific applications," ser. CGO 2016, 2016.

[38] L. Yu, D. Li, S. Mittal, and J. S. Vetter, "Quantitatively modeling application resilience with the data vulnerability factor," in *SC*, 2014.