The Effect of Input Data on Program Vulnerability

Vilas Sridharan and David R. Kaeli
Department of Electrical and Computer Engineering
Northeastern University
{vilas, kaeli}@ece.neu.edu

I. Introduction

The *System Vulnerability Stack* is a novel method to compute a system's vulnerability to transient faults (see Figure 1). The vulnerability stack observes that a system consists of multiple independent layers that interact through well-defined interfaces (e.g., the ISA). Therefore, the vulnerability stack quantifies fault masking within an individual layer by focusing on its interfaces: a fault that does not propagate to a layer's interfaces will be masked. These layer-level measurements can then be combined to yield a full-system vulnerability measure.

The vulnerability stack can have myriad benefits for architects interested in system vulnerability calculation. For instance, the vulnerability stack isolates hardware-level from software-level fault masking effects. This allows software-based reliability studies to report program-level fault masking (e.g., *Program Vulnerability Factor* [1]) instead of system-level fault masking (e.g., *Architectural Vulnerability Factor* [2]). For instance, Jones et al. examine the impact of compiler optimizations on AVF [3]. The authors report that compiler flag *-freorder-blocks* reduces AVF across a range of benchmarks. However, we do not know whether the optimization affected the benchmarks' *application derating* (software vulnerability) or the *machine derating* (hardware vulnerability) of the system under test [4]. If the latter, the compiler flag may have a different effect on different hardware.

In principle, the vulnerability stack can also eliminate much redundant computation currently required for design-time AVF analysis. The PVF of a program can be computed once, and the results used in all further simulations that use this program. Since benchmark programs typically have a useful life of many years, this can lead to a significant reduction in overall simulation time, potentially reducing the time-to-market for new products.

Finally, through fault abstraction, the vulnerability stack enables a much broader segment of the computer architecture and software engineering communities to participate in the vulnerability assessment process. Currently, these activities are performed by architects equipped with a microarchitecture simulator. By defining the effects of a fault on the ISA, the vulnerability stack allows a software architect to assess the vulnerability an application without the need for a microarchitecture model.

In this work, we first present an introduction to the System Vulnerability Stack. We then examine a limitation of the vulnerability stack: its dependence on input data. Specifically, program vulnerability calculations depend on the inputs to a program, but the extent of this dependence has yet to be determined. For example, the PVF of a program can be measured at compile time, when real inputs are not available. It is important to understand whether these compile-time vulnerability measurements will accurately reflect the behavior of the application when running with real input data. That is the goal of this work.

II. The System Vulnerability Stack

To introduce the System Vulnerability Stack, we first present some definitions. We define a *fault* as a raw failure event such as a single-event upset. A *visible fault* is observable by a particular system layer. For instance, a *microarchitecture-visible fault* is observable by the microarchitecture; and a *program-visible fault* is observable by a user program. A visible fault may be *masked* within a layer; a masked fault does not affect correct operation of that layer. A visible fault may also be *exposed* to another layer; this occurs when a visible fault in one layer creates a visible fault in another layer. For example, a fault within the microarchitecture will be exposed to a user program if it propagates to that program's architected state. A visible fault may be *activated* within a layer; this leads to a *visible error* in that layer. For example, a program-visible fault will be activated if it causes the program to generate incorrect output; and a microarchitecture-visible fault will be activated if it causes the hardware to deadlock. A visible error that is not corrected will lead to a *system failure*.

We define the *vulnerability* of a system layer as the fraction of visible faults in that layer that are either activated or exposed (i.e., the fraction of unmasked faults). For example, the *Program Vulnerability Factor* (PVF) is the vulnerability of a software layer [1]; and the *Hardware Vulnerability Factor* (HVF) is the vulnerability of a hardware layer. *System vulnerability* is the product of the vulnerability of all layers; that is, a fault will not cause a system error if it is masked by some layer. Note that an activated fault by definition causes an error. Therefore, for calculation purposes we treat a fault activated in one layer as if it is exposed by all other layers.

A. Computing System Vulnerability

In this section, we demonstrate how to compute system vulnerability using the vulnerability stack; for convenience, we use the terms *AVF* and *system vulnerability* interchangeably. Figure 2 shows a sequence of machine instructions and the corresponding events in physical register $P1$ of the microarchitecture. During cycles 4-6 and 12, $P1$ is not mapped to an architectural register. A microarchitecture-visible fault during

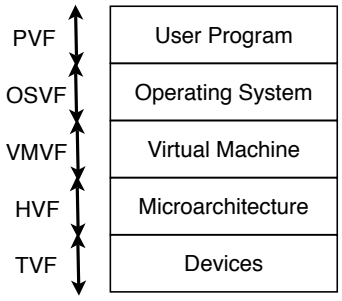


Fig. 1. The System Vulnerability Stack.

these cycles will be masked. In the remaining cycles, $P1$ is mapped to architectural registers $R1$ and $R2$. Therefore, a microarchitecture-visible fault during these cycles will propagate to the ISA and be exposed to the user program.

The resulting program-visible fault may or may not be masked. For instance, a fault during cycle 14 will be masked if it does not affect the compare operation in cycle 15. Program-visible masking can be given by the *Program Vulnerability Factor*, or PVF, of a bit [1]. If a program-visible fault in a bit will be masked, the PVF of that bit is defined as 0; otherwise, its PVF is 1. PVF assigns vulnerability values at moments in time marked by dynamic instructions; therefore, the AVF of bit b in cycle n is the PVF of the architectural state a_b contained in b at the time (instruction) i_n, the instruction that consumes the value stored in $P1$ during cycle n. For example, in cycle 10, the AVF of $P1$ is the PVF of $R2$ at time 5 (instruction $Stl\ R2,(R3)$).

We can calculate the vulnerability (AVF) of bit b in register $P1$ using PVF values:

$$\begin{aligned} AVF_b = \frac{1}{15} \times (&PVF_{R1_{b,i_1}} + PVF_{R1_{b,i_2}} + PVF_{R1_{b,i_3}} + \\ &PVF_{R2_{b,i_7}} + PVF_{R2_{b,i_8}} + PVF_{R2_{b,i_9}} + \\ &PVF_{R2_{b,i_{10}}} + PVF_{R2_{b,i_{11}}} + PVF_{R1_{b,i_{13}}} + \\ &PVF_{R1_{b,i_{14}}} + PVF_{R1_{b,i_{15}}}) \end{aligned}$$

To express this more concisely, we introduce the *Hardware Vulnerability Factor*. The HVF of a bit is 1 if the bit is mapped to architectural state, and 0 otherwise. Equation 1 can then be expressed as:

$$AVF_b = \frac{1}{N} \sum_{n=1}^{N} HVF_{b,n} \times PVF_{R1_b,i_n} \quad (1)$$

More generally, the AVF of a bit b in the microarchitecture during cycle n can be calculated as the HVF of that bit times the PVF of the architectural state a_b contained in that bit at time (program instruction) i_n:

$$AVF_b = \frac{1}{N} \sum_{n=1}^{N} HVF_{b,n} \times PVF_{a_b,i_n} \quad (2)$$

If multiple events to b occur simultaneously (e.g., two instructions read b in clock cycle n), we use the PVF of a_b at the earliest (in program order) instruction.

To determine the AVF of register $P1$ with size B_{P1}, we compute this over all bits in $P1$:

$$AVF_{P1} = \frac{1}{N \times B_{P1}} \sum_{b=1}^{B_{P1}} \sum_{n=1}^{N} HVF_{b,n} \times PVF_{a_b,i_n} \quad (3)$$

Equation 3 is a method to calculate AVF from independently-measurable quantities which correspond directly to system layers: HVF is the vulnerability of the hardware, and PVF is the vulnerability of the user program.

B. Computing the Vulnerability of Multiple Software Layers

The vulnerability stack can also quantify fault masking from software layers such as an operating system or virtual machine by focusing on the interfaces that these layers implement: a virtual machine implements an ISA, and an operating system implements an Application Binary Interface (ABI). Figure 3 depicts this process for register $R1$ on a system with a virtual machine and a (guest) user program. Instructions 1-5 and 15 are program instructions; instructions 6-14 are virtual machine instructions. From instructions 1-5, $R1$ is mapped to program register $V1$; a VM-visible fault in these cycles will be exposed to the user program. At instructions 6 and 7, the VM performs a context switch, saving $V1$ to memory and $V2$ to $R1$; $R1$ is mapped to guest register $V2$ from instructions 7-9. From instructions 10-13, the VM uses $R1$ to update its internal state; a fault in $R1$ will be masked during instructions 6, 10, and 13, and activated during instructions 11-12. Finally, in cycle 14, the VM restores the program context and the program continues execution in cycle 15.

To calculate the vulnerability of $R1$ over this period, we proceed similar to Equation 3: from instructions 1-5 and 14-15, a VM-visible fault is exposed to the user program, so $VF_{R1} = PVF_{V1}$. Similarly, from instructions 7-9, $VF_{R1} = PVF_{V2}$. For instruction 6 and instructions 10-13, the register is not mapped to program state; the vulnerability can be given by a *Virtual Machine Vulnerability Factor* (VMVF), which is 1 if a fault would be activated and 0 if it would be masked Therefore, the overall vulnerability of $R1$ can be calculated as follows:

$$VF_{R1} = \frac{1}{I \times B_{R1}} \sum_{b=1}^{B_{R1}} \sum_{i=1}^{I} VMVF_{b,i} \times PVF_{a_b,j_i} \quad (4)$$

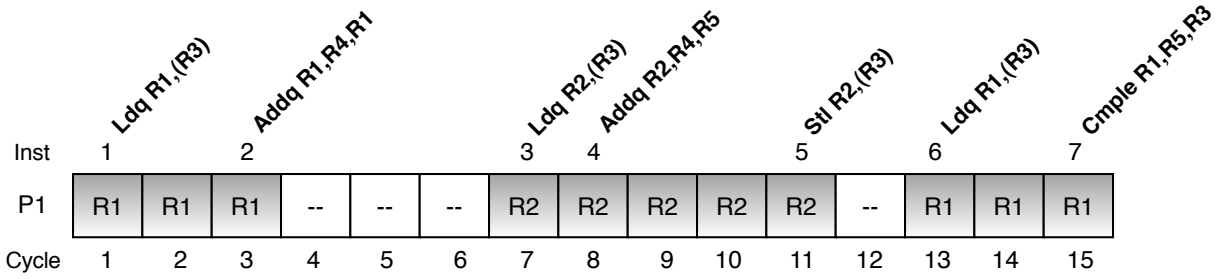


Fig. 2. Physical register P1 is mapped to architectural registers R1 and R2 over several cycles. Cycles 4-6 and 12 have microarchitecture-level masking (no bits are ACE). Cycles 1-3, 7-11 and 13-15 have program-level masking (a fraction of the bits are ACE).

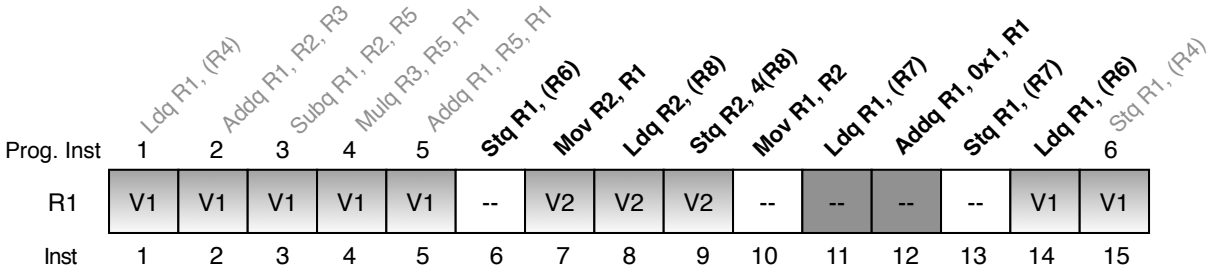


Fig. 3. The behavior of architectural register R1 with a VM and user program. VM instructions are shown in black; user program instructions are shown in gray. R1 is mapped to program register V1 during instructions 1-5 and 14-15; to program register V2 during instructions 7-9; and not mapped during instructions 6 and 10-13. The vulnerability of R1 can be calculated using the VMVF and PVF during each instruction.

Here, PVF_{a_b,j_i} denotes the PVF of the architectural state a mapped to bit b at time (instruction) j_i, the program instruction that consumes the value stored in b during VM instruction i. VF_{R1} is the vulnerability of architectural register $R1$ in the system under test; this value can be used in Equation 3 to calculate the AVF of the physical registers to which $R1$ is mapped.

C. The Effect of Input Data

The behavior of a system depends upon the inputs it is given. Therefore, the accuracy of vulnerability measurements depends on the ability to either: calculate vulnerability using real input data; or generate representative vulnerability behavior using synthetic inputs. In this paper, we explore the following question: can vulnerability measurements taken with one set of inputs predict the vulnerability behavior of a program with another set of inputs?

Input data affects program behavior in many ways. For example, input data may determine how many iterations of a loop are executed; or a program that accepts integer and floating-point data might call different functions based on the data type [5]. Many studies have examined how input data affects performance characteristics; however, most of the work has focused on the similarity of the execution profile of a binary with different inputs [6] [7] or on finding input-dependent branches [5] [8]. Typically, the goal of these studies is to determine whether a small input set can be used instead of a larger one. For fault tolerance, however, there are also other considerations: for example, the *value* of an input field which does not affect control flow can change the amount of logical masking present in the program. To our knowledge, there is no published study which examines the effect of input data on program reliability.

III. Methodology and Experimental Setup

Since the primary impact of input data is on program behavior, we examine the impact of input data on PVF. The PVF of a program can be measured by multiple means. For instance, a fault injection campaign can be performed: faults can be injected into a program's architectural state and tracked to determine whether they will be masked. Alternatively, PVF estimates can be generated using ACE Analysis: a single fault-free execution of a program that determines, at each instruction, whether a bit in an architectural resource is ACE. Neither fault-injection nor ACE Analysis require any microarchitectural simulation to measure PVF; they can be performed using an architectural simulator or a dynamic binary translator such as Pin [9]. As a result, complex real-world programs can be profiled to completion, a task that is impractical in many microarchitectural simulators.

In this work, we use ACE Analysis to measure the PVF of the Architectural Integer Register File. For our experiments, we use the Pin dynamic binary instrumentation tool and implement a Pintool to measure the PVF of a program's register file [9]. We perform our experiments on the SPEC CPU2006 benchmarks compiled for the x86_64 architecture. All benchmarks were simulated to completion. Our PVF instrumentation routines imposed an average slowdown of 500x, resulting in an average simulation speed of between 1 and 5 MIPS. This is a substantial slowdown relative to native execution, but remains orders of magnitude faster than microarchitectural simulation which can run at 1-10 KIPS in industrial-grade cycle-accurate simulators.

IV. Dependence on Input Data

Figure 4 shows that the PVF of *400.perlbench* varies substantially when run with different input data; this is one of

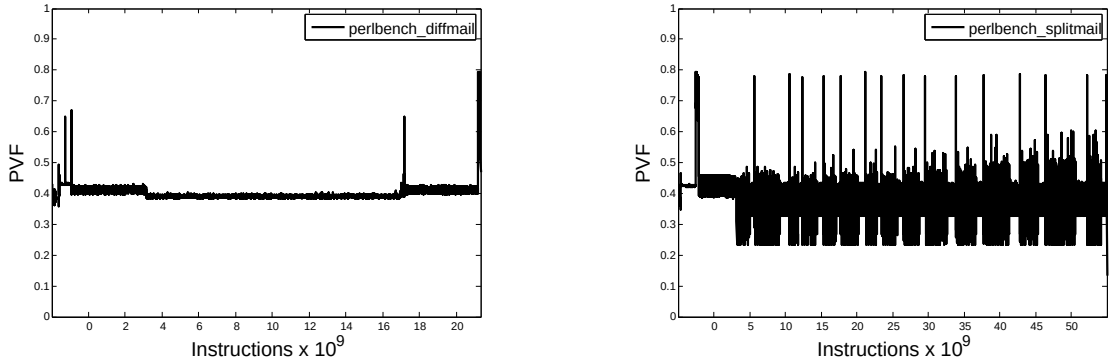


Fig. 4. Integer Register File PVF for *400.perlbench* using input files *diffmail* and *splitmail* from the *train* data set.

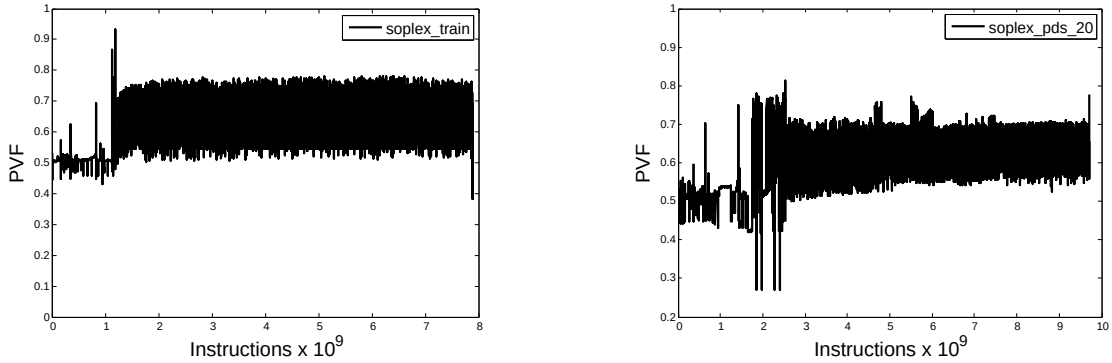


Fig. 5. Integer Register File PVF for *450.soplex* using input files *train* and *pds-20* from the *train* data set.

Benchmark	Input 1	Input 2	PVF Difference
400.perlbench	diffmail	splitmail	0.013
401.bzip2	program	combined	0.006
445.gobmk	nicklas2	blunder	0.016
450.soplex	pds-20	train	0.010
473.astar	BigLakes1024	rivers1	0.011

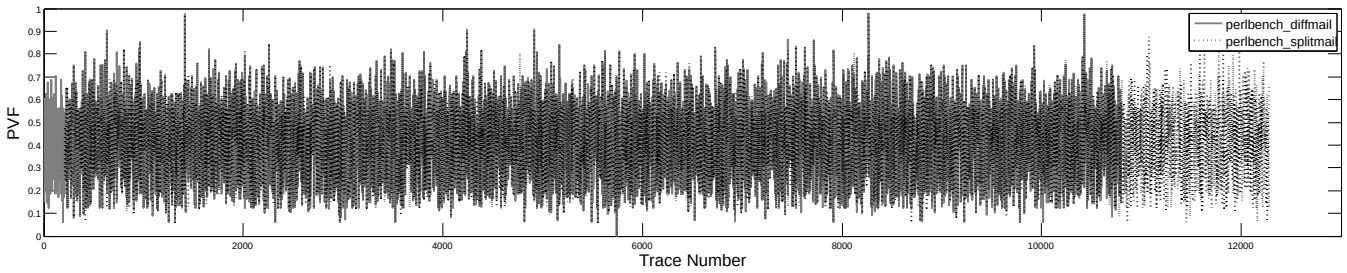
TABLE I
MEAN ABSOLUTE DIFFERENCE IN PVF FOR TRACES COMMON TO BOTH INPUT SETS.

the most noticeably input-dependent programs that we tested. Similarly, Figure 5 shows that the PVF of *450.soplex* varies when the input data is changed. However, our data show that these differences are predictable. Figure 6 presents the same data sorted by code trace (a single-entry, multiple-exit region of code). Each point on the x-axis represents a unique code trace as identified by Pin; the y-axis is the average PVF of that trace over the simulation. The center region of each figure contains traces that are touched by both inputs, while traces unique to one input are plotted on the left and right. The figure shows that most traces common to both inputs have approximately the same PVF regardless of the input; therefore, the PVF differences observed in Figures 4 and 5 primarily arise from differences in execution profile (i.e., traces unique to an input). Table I confirms this numerically for benchmarks with multiple inputs in the *train* dataset; there are only slight PVF differences between inputs for traces common to multiple inputs. Therefore, we hypothesize that representative inputs (e.g., inputs that exercise all code paths) will be able to accurately characterize program PVF and allow prediction of a program's PVF with unknown inputs.

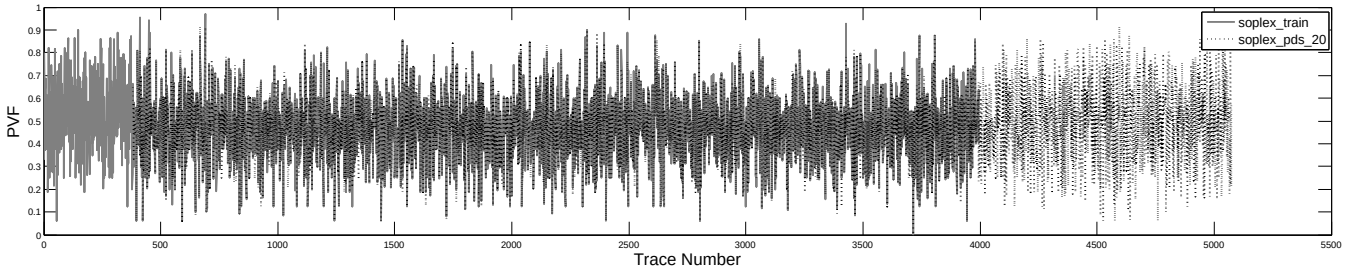
To test this hypothesis, we implement an algorithm to predict the PVF of a program that is given unknown input data. Our goal is to predict average PVF behavior rather than to calculate an exact PVF value. This is useful, for example, in systems that estimate vulnerability at runtime. Our current goal, however, is to demonstrate the feasibility of PVF prediction rather than to develop a low-overhead predictor. Therefore, we focus on accuracy at the expense of runtime considerations (e.g., performance, memory overhead). Our predictor uses a training run to generate a table of PVF predictions for each trace in a program. The table includes the start address of each trace and the average PVF of the trace over the training run. When the program is run with real inputs, we access the prediction table at the start of each new trace. If the table contains an entry for the trace, we use that value as the predicted PVF for the trace. If we do not find an entry, we retain the existing PVF prediction.

V. RESULTS

We use the *train* data set to train our predictor for each SPEC benchmark; Table II shows the input we use for benchmarks with multiple inputs (we always choose the input that touches the most traces). We then test our predictor on the benchmarks' *ref* data sets. Figure 7 shows the mean absolute error of our predictions for each benchmark. Overall, the predictor has a mean error of under 0.06 for all benchmarks except *450.soplex*. For example, the predictor generates predictions for over 99% of traces executed by *gcc_166*, shown in Figure 8(a), and accurately captures the PVF behavior of the application. *Gcc_g23* (Figure 8(b)), on the other hand,



(a) *400.perlbench*



(b) *450.soplex*

Fig. 6. Per-trace PVF for *400.perlbench* and *450.soplex* using multiple input files from their *train* data set.

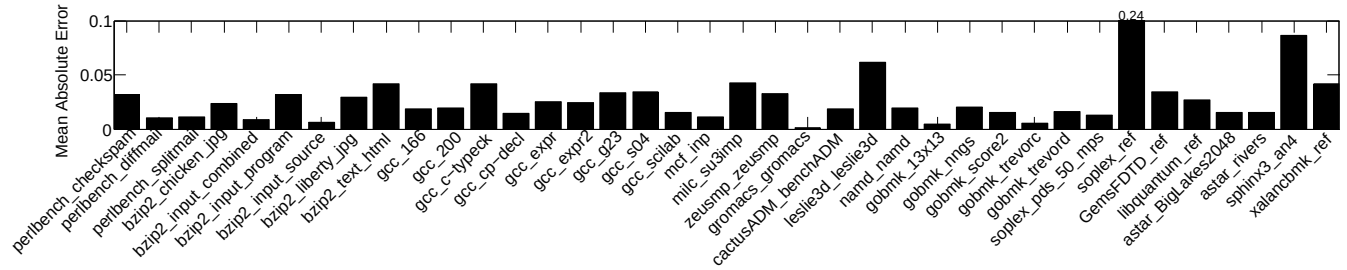


Fig. 7. Mean absolute error for our prediction algorithm for predicted traces and overall.

encounters more unique traces; the predictor only generates predictions for 90% of the traces, resulting in stretches of inaccuracy (e.g., the phase between instructions 14.2B and 17.5B). However, the policy of retaining the previous prediction works well, so the predictor has a mean absolute error of only 0.03 on average across the benchmark.

The predictor has an extremely high error on *450.soplex* (mean error = 0.24). This is because 67% of the traces encountered by the *ref* input were not encountered during training; the *pds-20.mps* input is not representative of the *ref* input. Figure 8(c) shows a portion of the execution of *450.soplex*. The predictor does well until approximately 27B instructions into execution, when *450.soplex* begins to encounter a high percentage of new traces. To recognize this condition, a runtime predictor can count the fraction of predicted traces. When this count dips below a pre-defined threshold, such as in this case or in the case of *gcc_g23*, the predictor can indicate a low confidence in its PVF estimates.

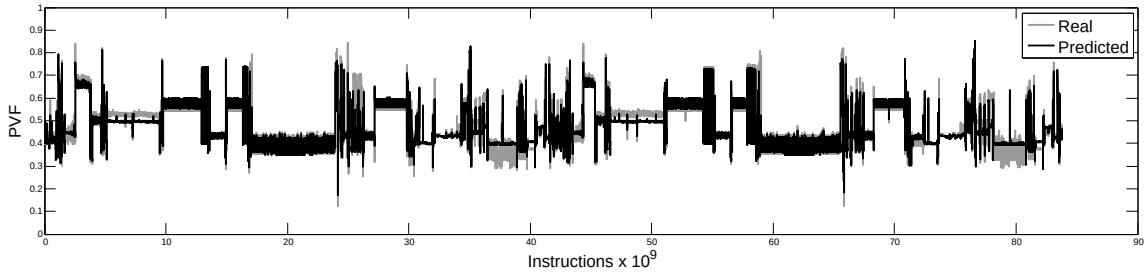
A post-hoc analysis of *450.soplex* (Figure 9) shows that the predictor's accuracy would have been significantly improved had it trained on *train.mps* rather than *pds-20.mps*. This again demonstrates the need for representative input data; but shows that given this input data, the PVF of an application can be accurately assessed.

Benchmark	Training Input
400.perlbench	splitmail
401.bzip2	combined
445.gobmk	nicklas4
450.soplex	pds-20
473.astar	BigLakes1024

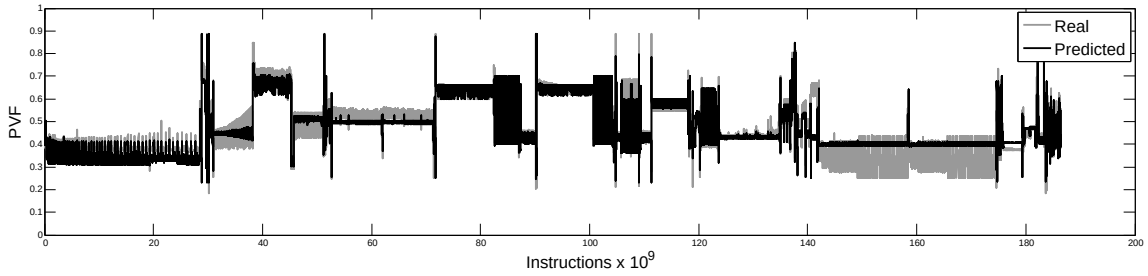
TABLE II
TRAINING INPUT USED FOR BENCHMARKS WITH MULTIPLE INPUTS IN THE *train* DATA SET.

VI. CONCLUSIONS

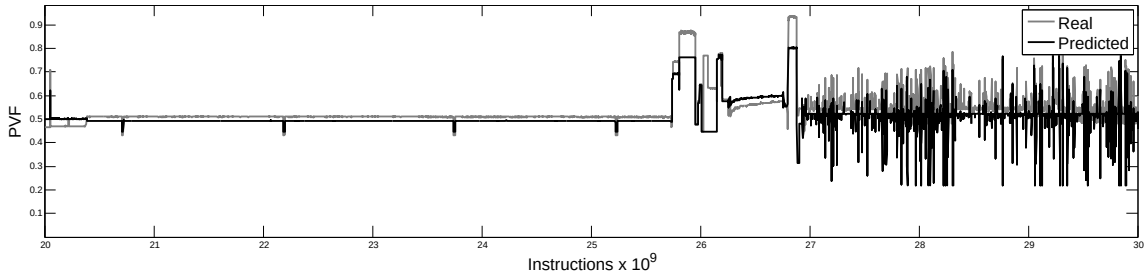
This work examined the dependence of program vulnerability on input data. Our data show that the primary cause of PVF difference between inputs is changes to a program's execution profile (i.e., which code is executed). Changes in the PVF of a single trace are only a small contributor to changes in PVF. As a result, an input file that causes a program to touch all code regions can be used to accurately characterize the PVF of a program. This can be used to predict the PVF of the program when run with arbitrary inputs. Our predictor generates highly accurate predictions when trained on representative inputs. In addition, this implies that software designers can use PVF at compile time to accurately assess (and potentially improve) the vulnerability of their application when executed on a real system. This result is significant, as compile-time vulnerability assessment is one of the primary benefits of PVF analysis.



(a) gcc_166



(b) gcc_g23



(c) soplex_ref

Fig. 8. Real versus predicted PVF for *gcc_166*, *gcc_g23*, and *soplex_ref*.

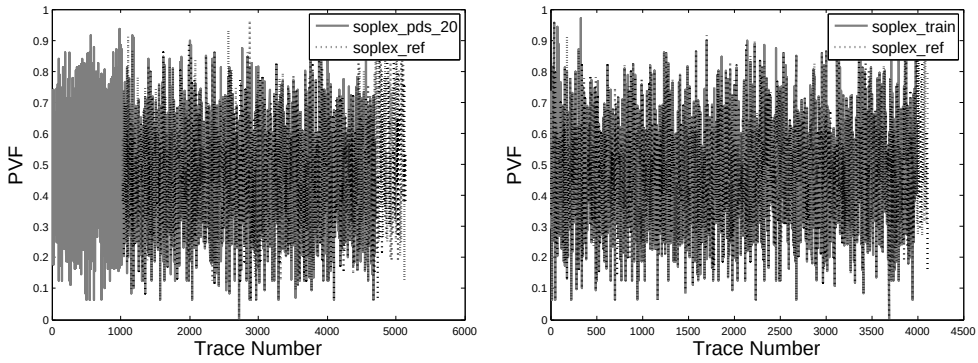


Fig. 9. The *train.mps* file covers more of the code traces touched by the *ref.mps* input file than does *pds-20.mps*; therefore, our predictor would perform better when trained with *train.mps*.

REFERENCES

[1] V. Sridharan and D. R. Kaeli, "Eliminating microarchitectural dependency from architectural vulnerability," in *International Symposium on High Performance Computer Architecture (HPCA-15)*, 14-18 Feb. 2009.

[2] S. S. Mukherjee *et al.*, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," in *International Symposium on Microarchitecture (MICRO-36)*, 2003.

[3] T. M. Jones *et al.*, "Evaluating the effect of compiler optimizations on AVF," in *Workshop on Interaction Between Compilers and Computer Architecture (INTERACT-12)*, 2008.

[4] P. N. Sanda *et al.*, "Soft-error resilience of the IBM POWER6 processor," *IBM Journal of Research and Development*, vol. 52, no. 3, pp. 275–284, 2008.

[5] H. Kim *et al.*, "2d-profiling: Detecting input-dependent branches with a single input data set," in *International Symposium on Code Generation and Optimization (CGO)*, 2006, pp. 159–172.

[6] W. C. Hsu *et al.*, "On the predictability of program behavior using different input data sets," in *Sixth Annual Workshop on Interaction between Compilers and Computer Architectures (INTERACT)*, 2002.

[7] L. Eeckhout *et al.*, "Workload design: Selecting representative program-input pairs," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2002, pp. 83–94.

[8] B. Calder *et al.*, "Corpus-based static branch prediction," in *Conference on Programming language design and implementation (PLDI)*, 1995, pp. 79–92.

[9] C.-K. Luk *et al.*, "Pin: building customized program analysis tools with dynamic instrumentation," in *Conference on Programming Language Design and Implementation (PLDI '05)*, 2005.