Modeling Soft-Error Propagation in Programs

Guanpeng Li, Karthik Pattabiraman, Siva Kumar Sastry Hari, Michael Sullivan, Timothy Tsai

University of British Columbia, NVIDIA

HAILONG JIANG

Sept. 23rd, 2019

BACKGROUND

Resilience Research: Find the resilient/not-resilient part of the code and Protect

Do Fault Injection (FI) → to estimate the SDC probabilities of programs. (Monte Carlo)

Thousands of FIs → too much time to be practical

Modeling error propagation model to identify vulnerable instructions

- Accuracy
  - data-dependencies of an instruction, but also to the
  - subsequent branches (i.e., control flow) and
  - memory locations
- Scalability
  - tracking the deviation in control-flow and memory locations due to a fault often leads to state space explosion.

MOTIVATION

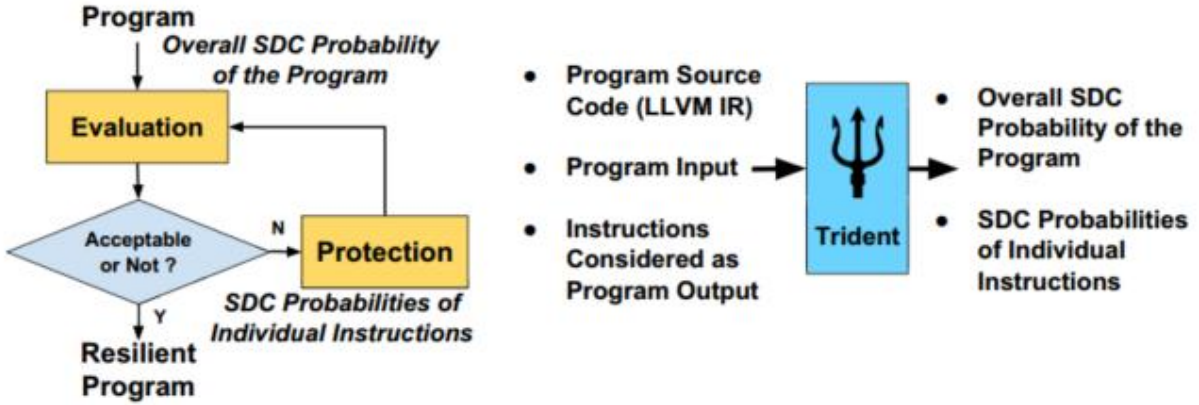
- To estimate the SDC probability of a program – both in the aggregate, and on an individual instruction basis - to decide whether protection is required
- TRIDENT, predict both the overall SDC probability of a program and the SDC probability of individual instructions based on dynamic and static analysis of the program without performing FI.
- The key insight: error propagation in dynamic execution can be decomposed into a combination of individual modules, each of which can be abstracted into probabilistic events.

BACKGROUND

- Fault model:
  - Crash
  - Silent Data Corruption (SDC)
  - Benign Faults
- Challenges in modeling error propagation:
  - (1) Statically modeling error propagation in dynamic program execution requires a model that abstracts the program data-flow in the presence of faults.
  - (2) control-flow divergence . In any divergence, there are numerous possible execution paths, leads to state space explosion.
  - (3) Faults may corrupt memory locations and hence continue to propagate through memory operations. enormous numbers of store and load instructions in a typical program execution, and tracing error propagations among these memory dependencies requires constructing a huge data dependency graph, which is very expensive.

TRIDENT

Input and Output



(a) Development Cycle (b) Workflow of TRIDENT

Fig. 1: Development of Fault-Tolerant Applications

OVERVIEW AND INSIGHTS

error propagation follows program data-flow at runtime

model data-flow at three levels:

(1) Static-instruction level, corresponding to the execution of a static data-dependent instruction sequence and the transfer of results between registers. (f_s)

(2) Control-flow level, when execution jumps to another program

location. (f_c)

(3) Memory level, when the results need to be transferred back to memory. (f_m)

CORE ALGORITHM

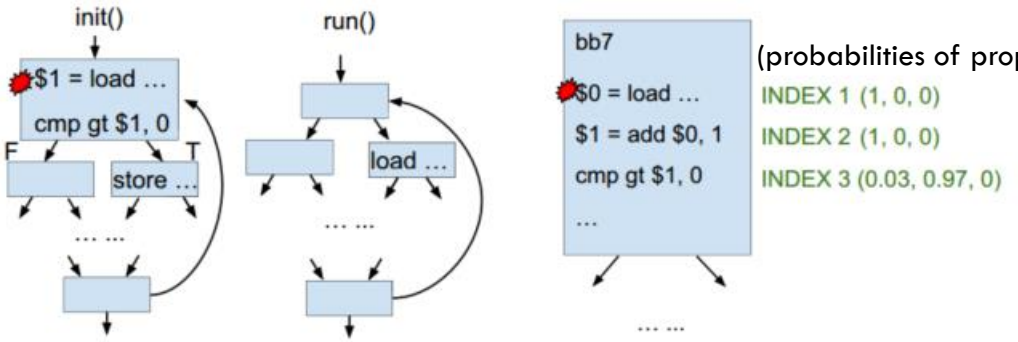
Algorithm 1: The Core Algorithm in TRIDENT

sub-models f_s , f_c , and f_m ;
Input : *I: Instruction where the fault occurs*
Output: P_{SDC}*: SDC probability*
$p_s = f_s\ (I)$;
if *inst. sequence containing I ends with branch* I_b **then**
    // Get the list of stores corrupted and their prob.
    [<I_c, p_c>, ...] = $f_c\ (I_b)$;
    // Maximum propagation prob. is 1
    Foreach(<I_c, p_c>): P_{SDC} += p_s * p_c * $f_m\ (I_c)$;
else if *inst. sequence containing I ends with store* I_s **then**
    $P_{SDC} = p_s * f_m\ (I_s)$;

STATIC-INSTRUCTION SUB-MODEL

Faults propagate on its static data-dependent instruction sequence.

f_s computes the probability of error propagation when the execution reaches the end of the static computation sequence of the instruction.



(a) Example of Propagation (b) Propagation in f_s

Fig. 2: Running Example

only a few types of instructions that have masking probabilities: comparisons (e.g., CMP), logic operators (e.g., XOR) and casts (e.g., TRUNC)

1/(32 bits)= 0.03

We consider crashes that are caused by program reading or writing out-of-bound memory addresses. Their probabilities can be approximated by profiling memory size allocated for the program

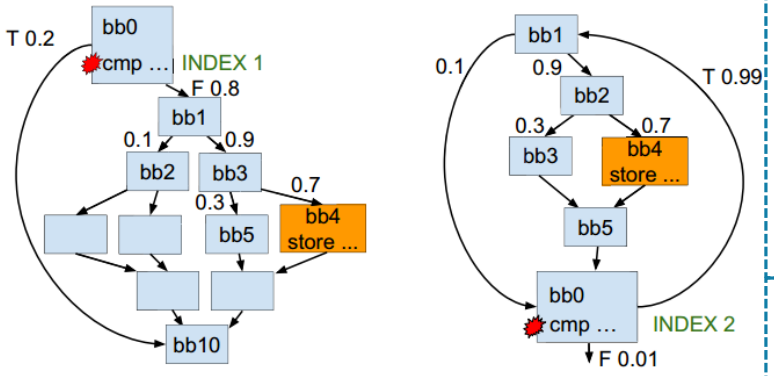
CONTROL-FLOW SUB-MODEL (FC)

The goal of f_c is to figure out which dynamic store instructions will be corrupted and at what probabilities, if a conditional branch is corrupted.

branch conditions into two types:

(1) Non-Loop-Terminating cmp (NLT)

(2) Loop-Terminating cmp (LT).



(a) Example of NLT (b) Example of LT

Fig. 3: NLT and LT Examples of the CFG

$$P_c = P_e / P_d \quad (1)$$

In the equation, P_c is the probability of the store being corrupted, P_e is the execution probability of the store instruction in fault-free execution, and P_d is the branch probability of which direction dominates the store.

P_e: 0.8 * 0.9 * 0.7 (bb0-bb1-bb3-bb4)
P_d: is 0.8 (bb0-bb1),
thus P_c is 0.8 * 0.9 * 0.7/0.8 = 0.63.

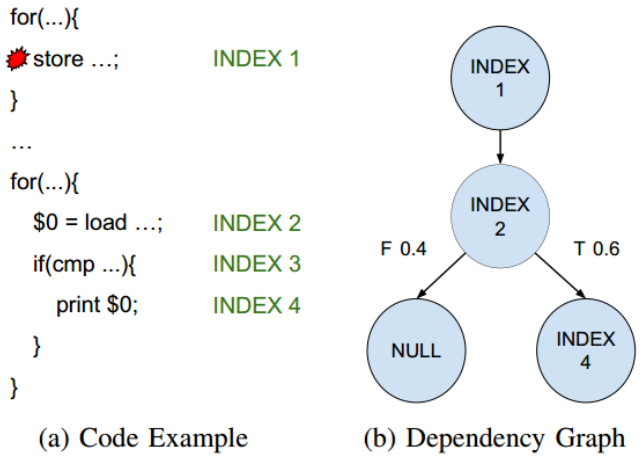
$$P_c = P_b * P_e \quad (2)$$

P_c is the probability that a dynamic store instruction is corrupted if the branch is modified, P_b is the execution probability of the back-edge of the branch, and P_e is the execution probability of the store instruction dominated by the back-edge.

P_b is 0.99 (bb0-bb1)
P_e is 0.7*0.9 (bb1-bb2-bb4),
thus P_c is 0.99 * 0.7 * 0.9 = 0.62.

MEMORY SUB-MODEL (FM)

- f_m reports the probability for the error to propagate from the corrupted memory locations to the program output.
- The idea is to represent memory data dependencies between the load and store instructions in an execution, so that the model can trace the error propagation in the memory



(a) Code Example (b) Dependency Graph

Fig. 4: Examples for Memory Sub-model

the propagation probability to the program output (INDEX 4),
if one of the store (INDEX 1) in the loop is corrupted, is
$1*1*1*0.6/(0.4+0.6)+1*1*0*0.4/(0.4+0.6) = 0.6$.

EVALUATION

Accuracy

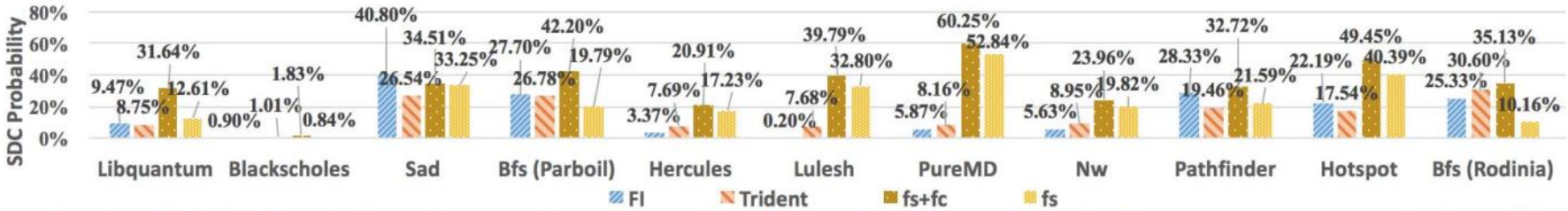


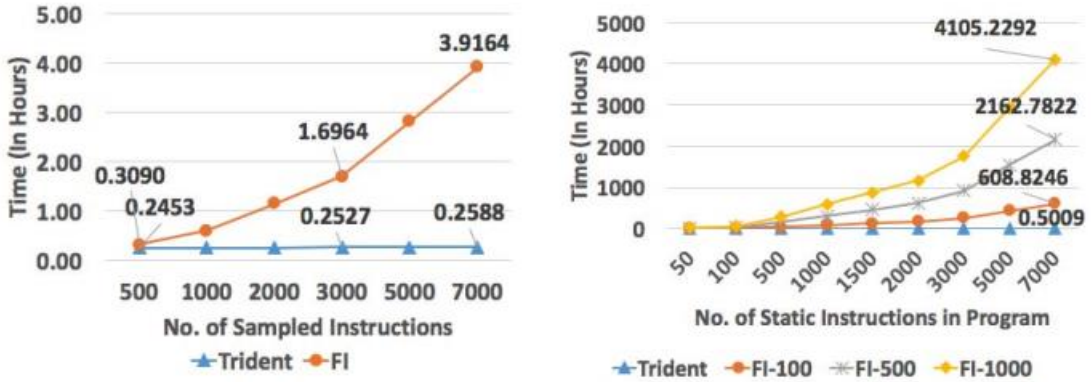
Fig. 5: Overall SDC Probabilities Measured by FI and Predicted by the Three Models (Margin of Error for FI: ±0.07% to ±1.76% at 95% Confidence)

TABLE II: p-values of T-test Experiments in the Prediction of Individual Instruction SDC Probability Values ($p > 0.05$ indicates that we are not able to reject our null hypothesis – the counter-cases are shown in red)

Benchmark	TRIDENT	fs+fc	fs
Libquantum	0.602	0.000	0.000
Blackscholes	0.392	0.173	0.832
Sad	0.000	0.003	0.000
Bfs (Parboil)	0.893	0.000	0.261
Hercules	0.163	0.000	0.003
Lulesh	0.000	0.000	0.000
PureMD	0.277	0.000	0.000
Nw	0.059	0.000	0.000
Pathfinder	0.033	0.130	0.178
Hotspot	0.166	0.000	0.000
Bfs (Rodinia)	0.497	0.001	0.126
No. of rejections	**3/11**	**9/11**	**7/11**

EVALUATION

Scalability



(a) Overall SDC Probability (b) Instruction SDC Probability

Fig. 6: Computation Spent to Predict SDC Probability

COMPARED WITH PVF&EPVF

- ePVF [1] (enhanced PVF) and PVF: modeling technique for error propagation
- Same goal with TRIDENT: predicting the SDC probability of a program (aggregate level and instruction level)
- PVF [2], (*Program Vulnerability Factor*). PVF does not distinguish between crash-causing faults and SDCs
- ePVF cannot distinguish between benign faults and SDCs
- ePVF only models error propagation in static data dependent instruction sequence and in memory, ignoring error propagation to control-flow and other parts of memory.

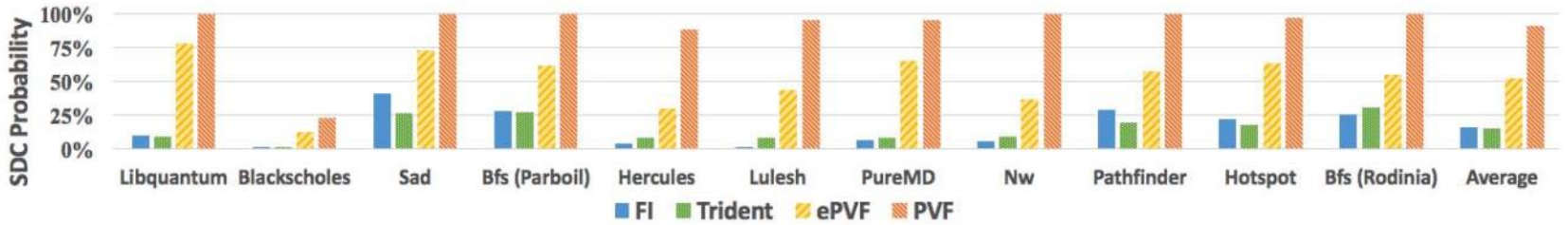


Fig. 9: Overall SDC Probabilities Measured by FI and Predicted by TRIDENT, ePVF and PVF (Margin of Error: ±0.07% to ±1.76% at 95% Confidence)

DISCUSSION

1. Is the error propagation probability correct? 1/32 in Figure 1.
2. How did they know the branch probability of each branch? Like in Figure 3.
3. Is that computation of f_c correct?
4. This may give us some idea and inspiration, but I am not really clear now.

PRIMARY PAPER INCLUDED

[1] Vilas Sridharan and David R Kaeli. Eliminating microarchitectural dependency from architectural vulnerability. In 15th International Symposium on High Performance Computer Architecture.

[2] Bo Fang, Qining Lu, Karthik Pattabiraman, Matei Ripeanu, and Sudhanva Gurumurthi. ePVF: An enhanced program vulnerability factor methodology for cross-layer resilience analysis. In Proceedings of the International Conference on Dependable Systems and Networks (DSN), pages 168–179. IEEE, 2016.