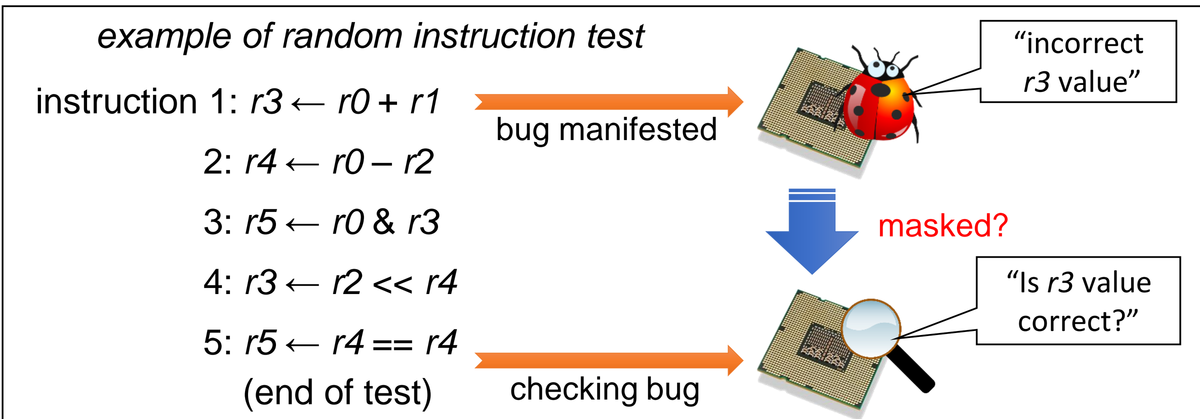
Probabilistic Bug-Masking Analysis for Post-Silicon Tests in Microprocessor Verification

Doowon Lee*, Tom Kolan†, Arkadiy Morgenshtein†, Vitali Sokhin†, Ronny Morad†, Avi Ziv† and Valeria Bertacco*

* University of Michigan, † IBM Research – Haifa

Post-silicon validation and challenge

- **Post-silicon validation** is crucial to stimulate corner cases that are unverified in pre-silicon validation
  - Pseudo-random instruction tests
  - Self-checking techniques
- Post-silicon validation challenge: **limited observability** to microprocessor internals
  - Assumption: only architectural state is observable at the end of test
- Bug-masking can happen due to limited observability



Baseline: information-flow tracking

- Adopting information-flow tracking used in software security analysis
  - Treat each instruction's target (e.g., register, memory) as taint source
  - Checking for information reachability
- Analyzing use-def chains along instruction sequence

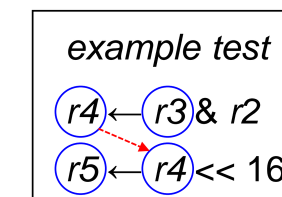
no	instruction	taint status: r0	r1	r2	r3	r4	r5
1	r3 ← r0 + r1	ϕ	ϕ	ϕ	{1}	ϕ	ϕ
2	r4 ← r0 – r2	ϕ	ϕ	ϕ	{1}	{2}	ϕ
3	r5 ← r0 & r3	ϕ	ϕ	ϕ	{1}	{2}	{1,3}
4	r3 ← r2 << r4	ϕ	ϕ	ϕ	{2,4}	{2}	{1,3}
5	r5 ← r4 == r4	ϕ	ϕ	ϕ	{2,4}	{2}	{2,5}

union set = {2, 4, 5}

"Bugs manifested in other two instructions (1 and 3) could go undetected"

Limitation of information-flow tracking

Buggy information can be masked by subsequent instructions depending on how instructions propagate the information



Information-flow tracking result:
"r5 might contain buggy information propagated from r3."
➔ Actual results depends on register r2 value…

e.g.: Suppose r3 gets a buggy value 0xBAADBAAD

(1) When r2 = 0x0000FFFF
r4 ← 0xBAADBAAD & 0x0000FFFF
r5 ← 0x0000BAAD << 16
(= 0xBAAD0000)

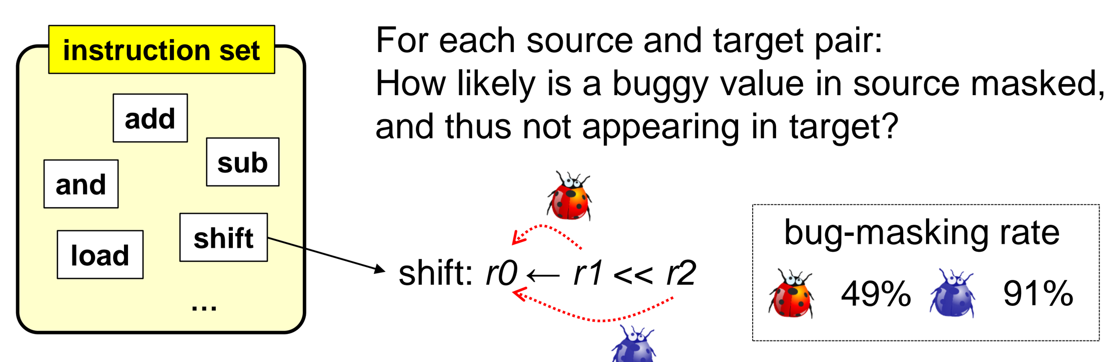
(2) When r2 = 0xFFFF0000
r4 ← 0xBAADBAAD & 0xFFFF0000
r5 ← 0xBAAD0000 << 16
(= 0x00000000: **no trace**)

- Static information-flow tracking conservatively predicts bug-masking incidence
- Goal: improving accuracy of tracking by using probability model

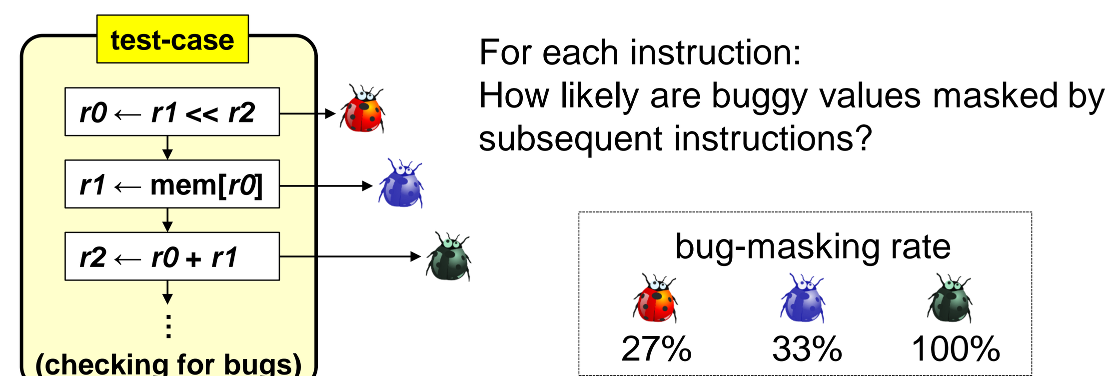
BugMAPI: Bug-Masking Analysis with Probabilistic Information-flow

BugMAPI overview

Step 1: Calculate bug-masking probability for each instruction type

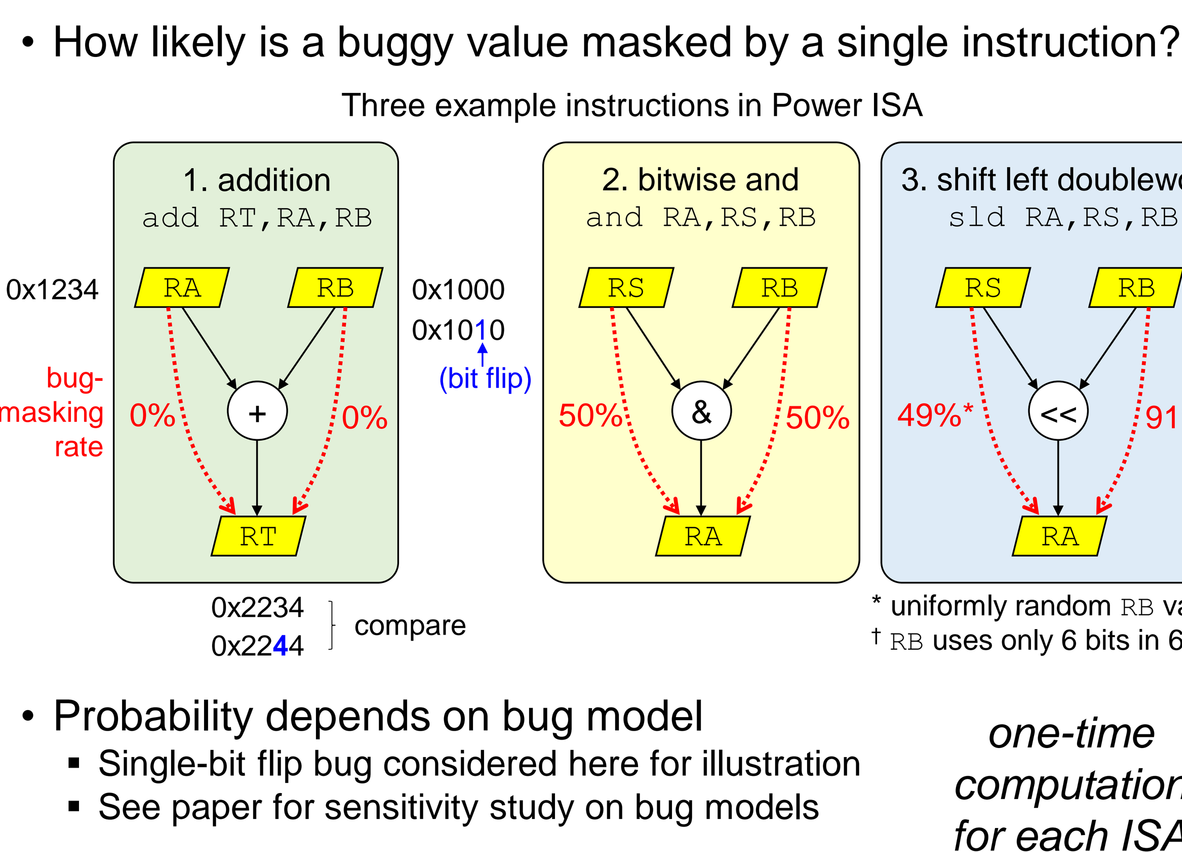


Step 2: Calculate bug-masking probability for instruction sequence (test-case)



Step 1: calculating instruction's probability

- How likely is a buggy value masked by a single instruction?



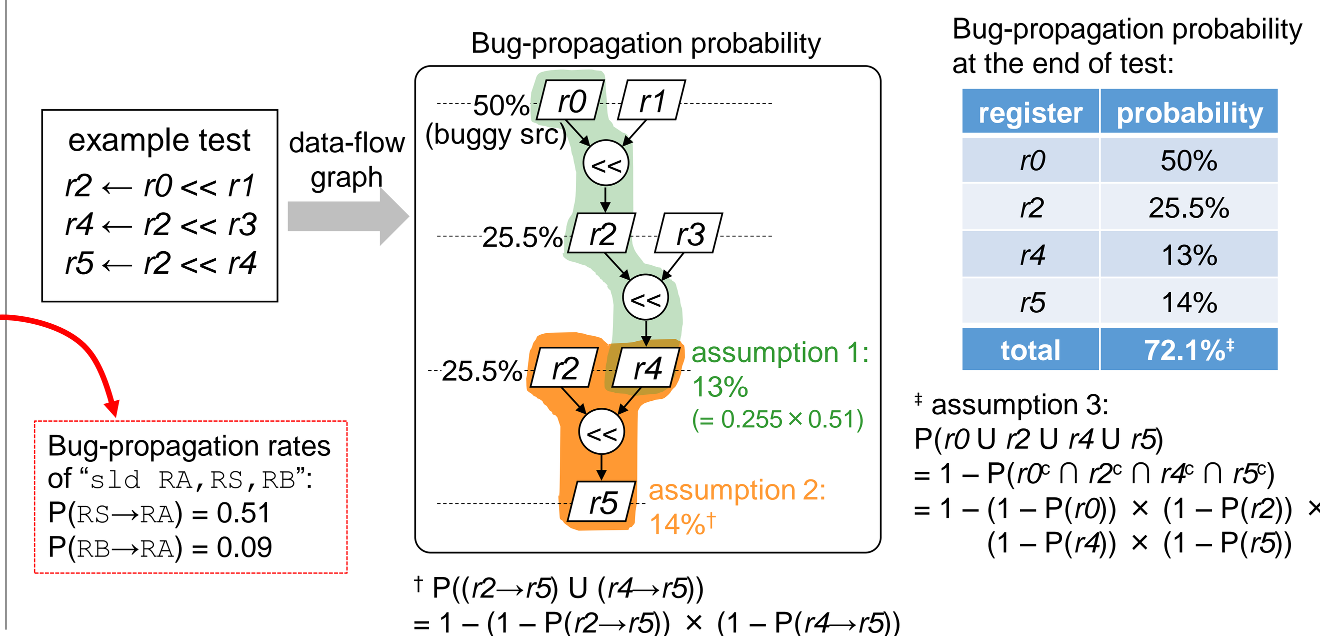
- Probability depends on bug model
  - Single-bit flip bug considered here for illustration
  - See paper for sensitivity study on bug models

one-time computation for each ISA

Step 2: calculating sequence's probability

Lightweight probability computation using three simplifying independence assumptions

1. Independence among instructions
2. Independence among inputs
3. Independence among resources



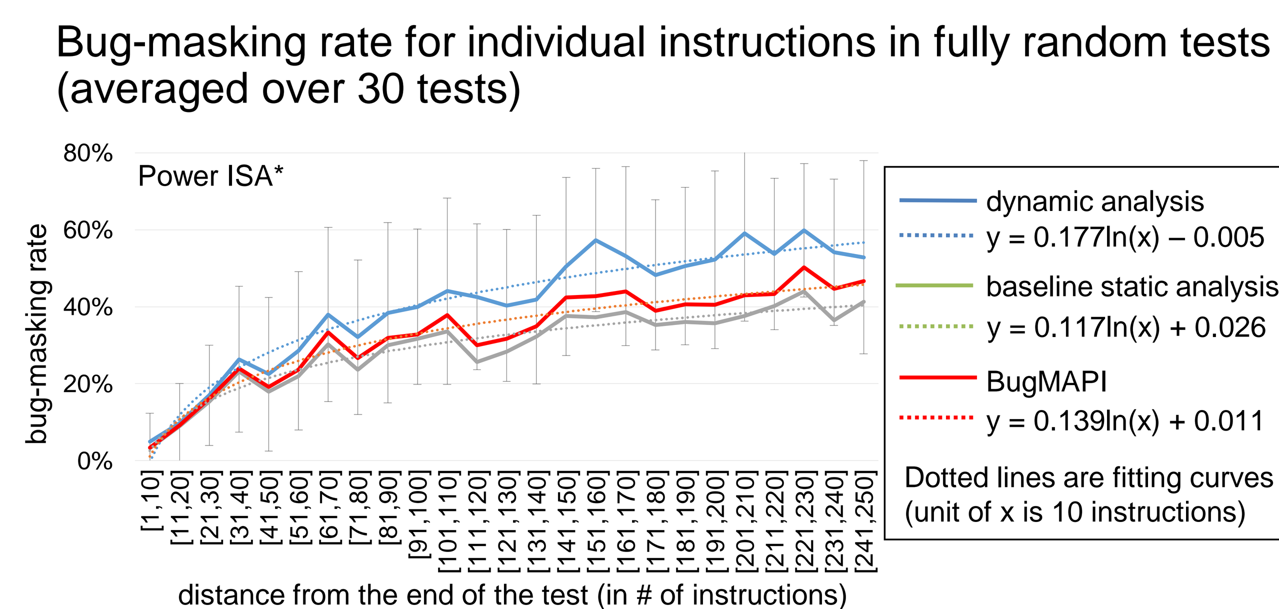
Experimental evaluation

Experimental setup

- Two ISAs evaluated: IBM Power and DEC Alpha
  - IBM Power: approximately 4,000 source-target pairs
    - Test generator: Threadmill [DAC'11]
  - DEC Alpha: 218 source-target pairs (subset of instructions)
    - Test generator: in-house fully-random test generator
- Bug model
  - Mimicking micro-architectural bugs by slightly altering architectural state
  - Random erroneous value update for architectural state (register / memory location)
- BugMAPI execution time
  - BugMAPI (11.3 seconds) vs. dynamic analysis (~25,000 seconds): 3 orders-of-magnitude speedup

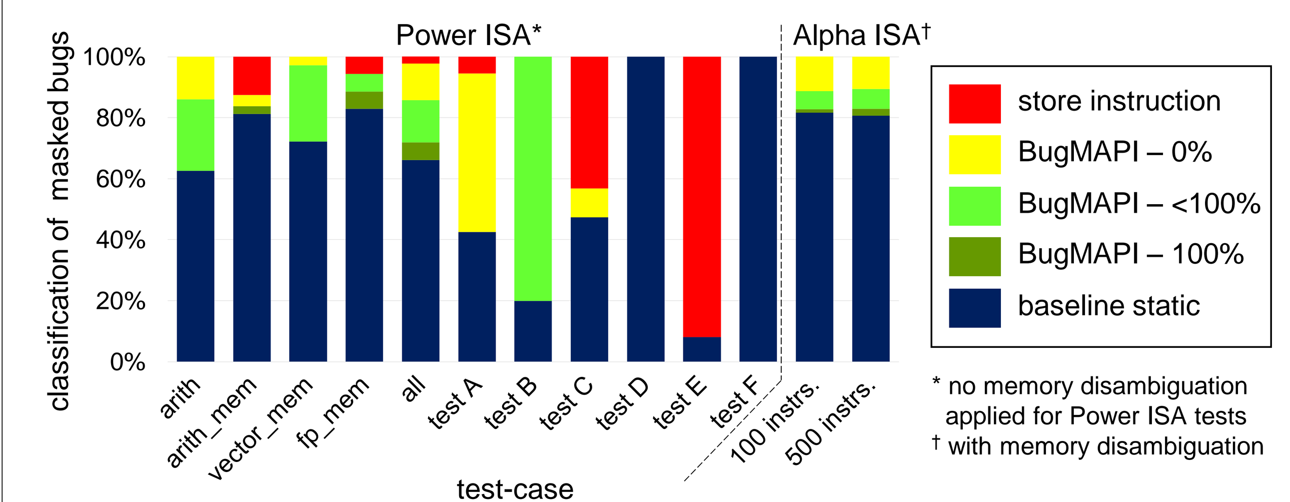
Bug-masking rate vs. distance from end of test

Bug-masking rate for individual instructions in fully random tests (averaged over 30 tests)



Observations:
1. The farther checking point, the higher bug-masking rate (but slower increment)
2. BugMAPI recognizes 79% of masked instruction

BugMAPI accuracy



Observations:
1. BugMAPI can detect masked bugs **15% more** than baseline static analysis : BugMAPI (77%) vs. baseline static analysis (62%) (Power ISA)
2. Bug-masking assessment remains stable **regardless of length** (Alpha ISA)

Application: bug-masking reduction

- Reducing bug-masking incidence in random instruction tests
- Two steps: identification and patching

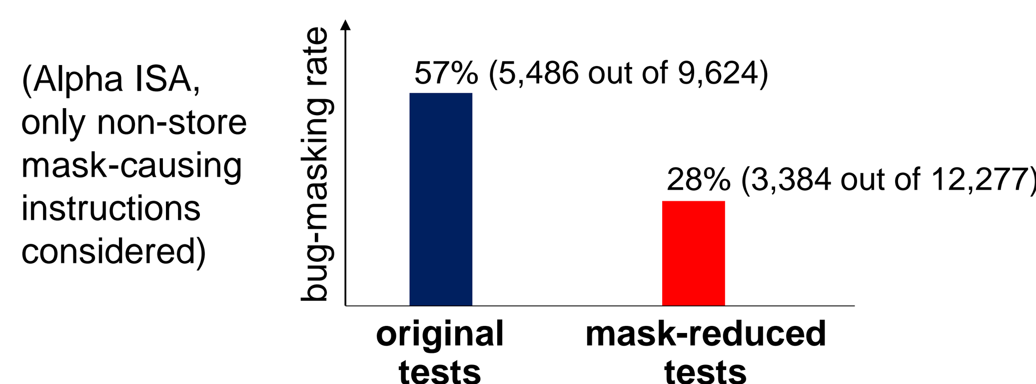
original test
r3 ← r0 + r1
r4 ← r0 – r2
r5 ← r0 & r3
r3 ← r2 << r4
r5 ← r4 == r4

mask-reduced test
r3 ← r0 + r1
r4 ← r0 – r2
r5 ← r0 & r3
r3 ← r2 << r4
r4 ← r5 XOR r4
r5 ← r4 == r4

(1) Identifying mask-causing instructions by using BugMAPI

(2) Patching mask-reducing instructions
- ✓ XOR instruction (0% masking rate)
- ✓ No extra register required

(Alpha ISA, only non-store mask-causing instructions considered)



Results:
- **Masking rate reduced to half**
- Can further reduce masking by considering mask-causing store instructions

Discussions

- Sources of inaccuracy
  - Coarse granularity of information-flow tracking (register-level tracking)
  - Approximated probability computation (dependency ignorance)
  - Lack of consideration of instruction blocks (e.g., address calculation)
- Bug-model dependency
  - 5 models considered (random value overwrite, single-bit flip, binary complementation, missing update, spurious update)
  - Comparable detection rates, except for spurious update (false positives)
- Multi-thread programs
  - Non-deterministic execution, inter-thread data dependency
  - Profiling frequency of inter-thread data dependency

Austin, TX, USA
June 5–9, 2016