KEYSIGHT riscure SECURITY SOLUTIONS

Improving CPU Fault Injection Simulations

Insights from RTL to Instruction-Level Models

Jasper van Woudenberg, Rajesh Velegalati,
Cees-Bart Breunesse, Dennis Vermoen

FDTC 2024

Problem statement

Programmers have a hard time developing FI resistant code

Can we help programmers create more resistant code through simulation?

Countermeasure refs:

https://riscureprodstorage.blob.core.windows.net/production/2017/08/Riscure_Whitepaper_Side_Channel_Patterns.pdf

http://hardwarehack.ing

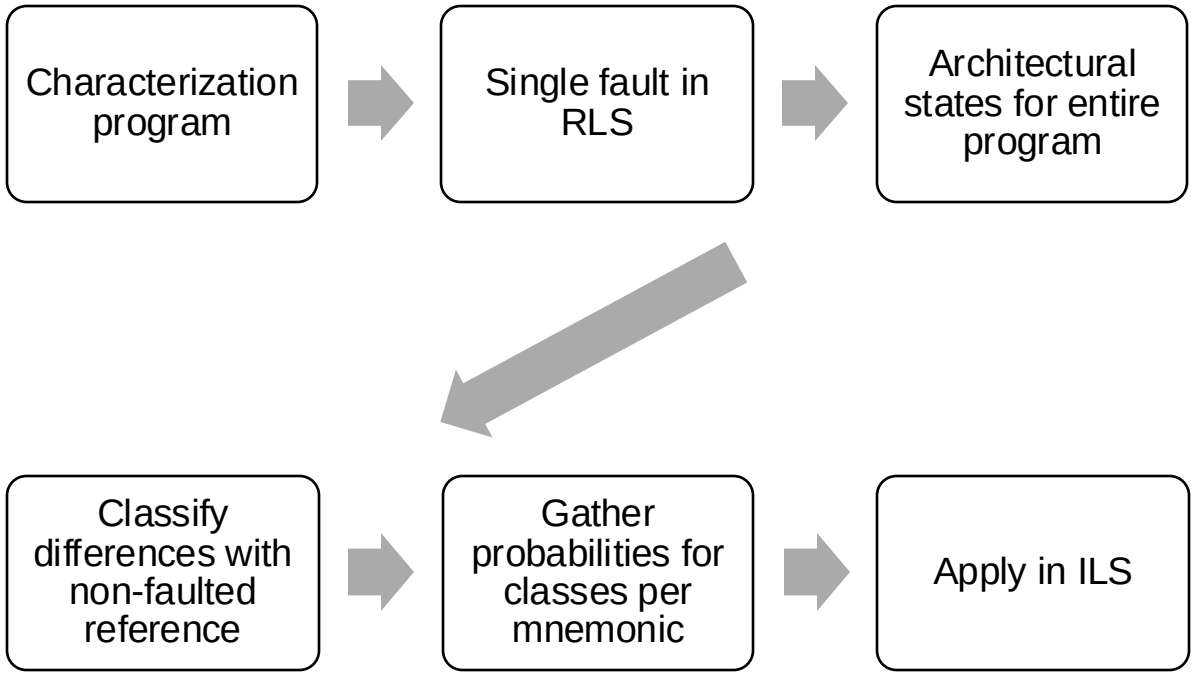
FI approach comparison

Rankings; higher is better

Can we make this “2”?

	RTL level sim (RLS)	Instruct. level sim (ILS)	Post-si fault injection
Accuracy	1	0	2
Runtime	0 (but scales with compute)	1 (also scales)	2 (scales poorly)
Ease of triage/fixing for sw dev	0	2	1
ROM fixing possibilities	1	2	0

Research overview

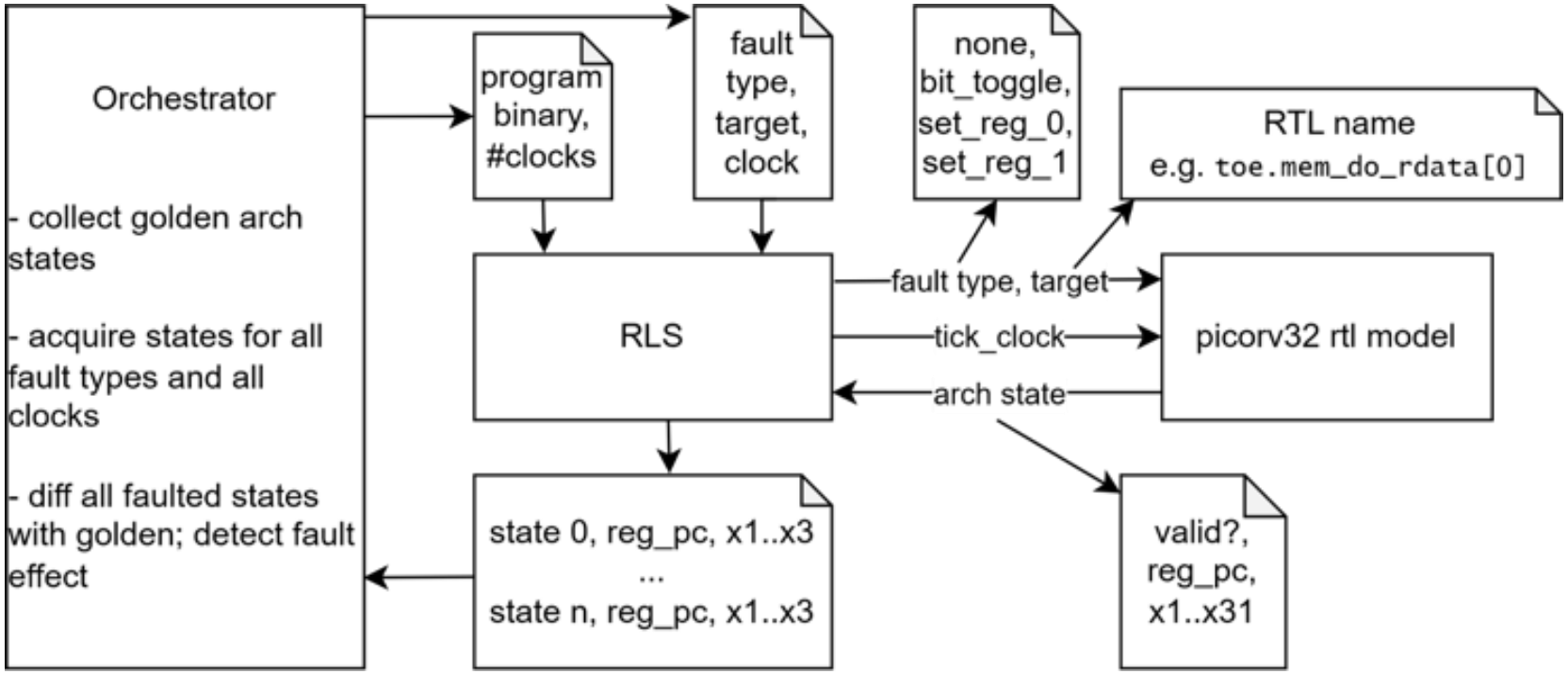


Characterization program

- We want fault statistics per mnemonic, presumably different statistics
- Put in (almost) all rv32imc instruction (mnemonics)
- Ensure each instruction changes the architectural state
  - Arch state we look at is registers + PC only

```
…
 sltu x13, x1, x2
    lui x1, 0x12345
    sra x14, x1, x2
    srl x15, x1, x2
    sub x16, x1, x2
    xor x17, x1, x2
    xori x18, x1, 0xff
    beq x1, x1, label_beq
    .word 0x00000013 // NOP in 32 b
label_beq:
    bge x1, x2, label_bge
    nop
label_bge:
    bgeu x1, x2, label_bgeu
    nop
label_bgeu:
    blt x17, x1, label_blt
    nop
label_blt:
    bltu x1, x2, label_bltu
    nop
label_bltu:
    bne x1, x2, label_bne
…
```

RTL level simulator

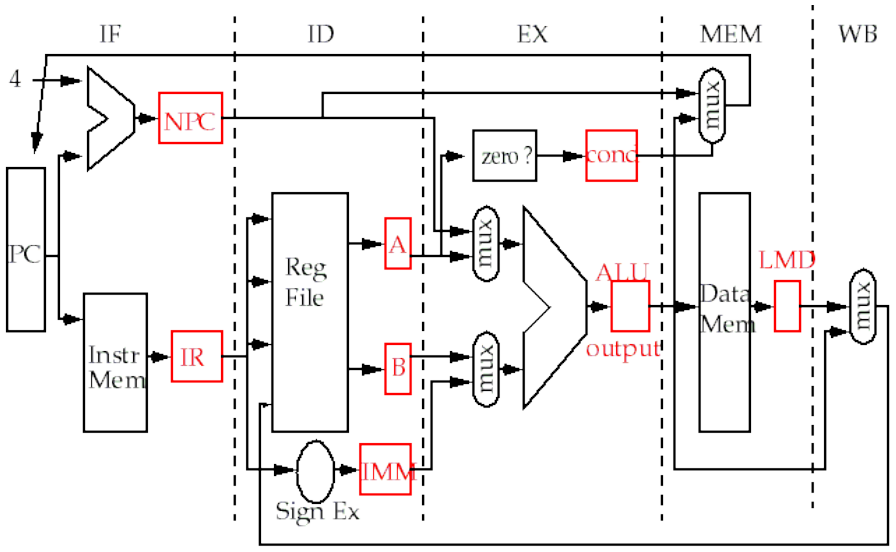


- Verilated Picorv32 core (configured for rv32imc), enhanced with FI capabilities
- Not faulting memories, only logic.
- Generates #clock cycles x #targets architectural state traces

Mapping faults to architectural states

- There is no 1:1 mapping between clock cycles and program counter (even in picorv32!)
- But, we can detect which state gets affected, and map back to that PC

http://ece-research.unm.edu/jimp/611/slides/chap3_1.html



Golden run

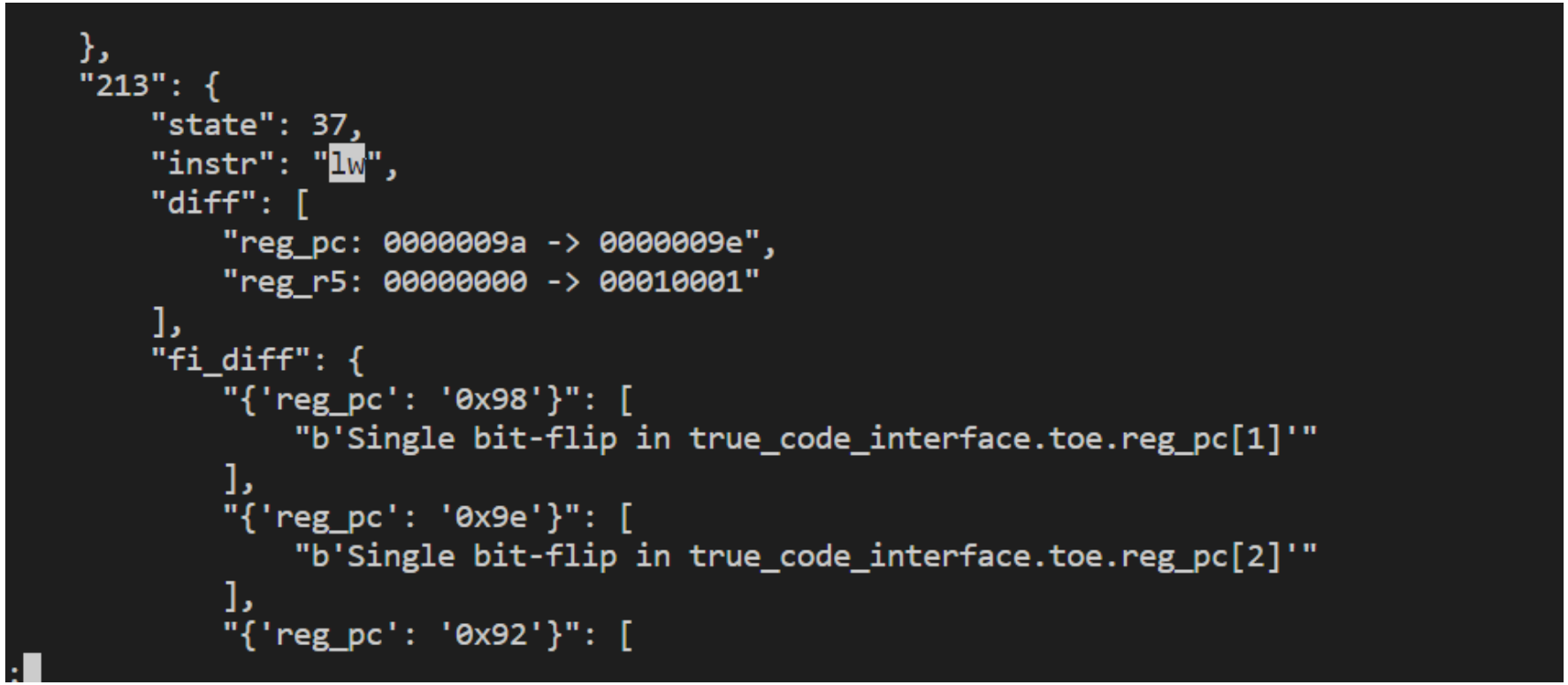
State	X1	PC
0	ab	00
1	cd	02
2	38	06

Faulted run

State	X1	PC
0	ab	00
1	cd	02
2	39	06

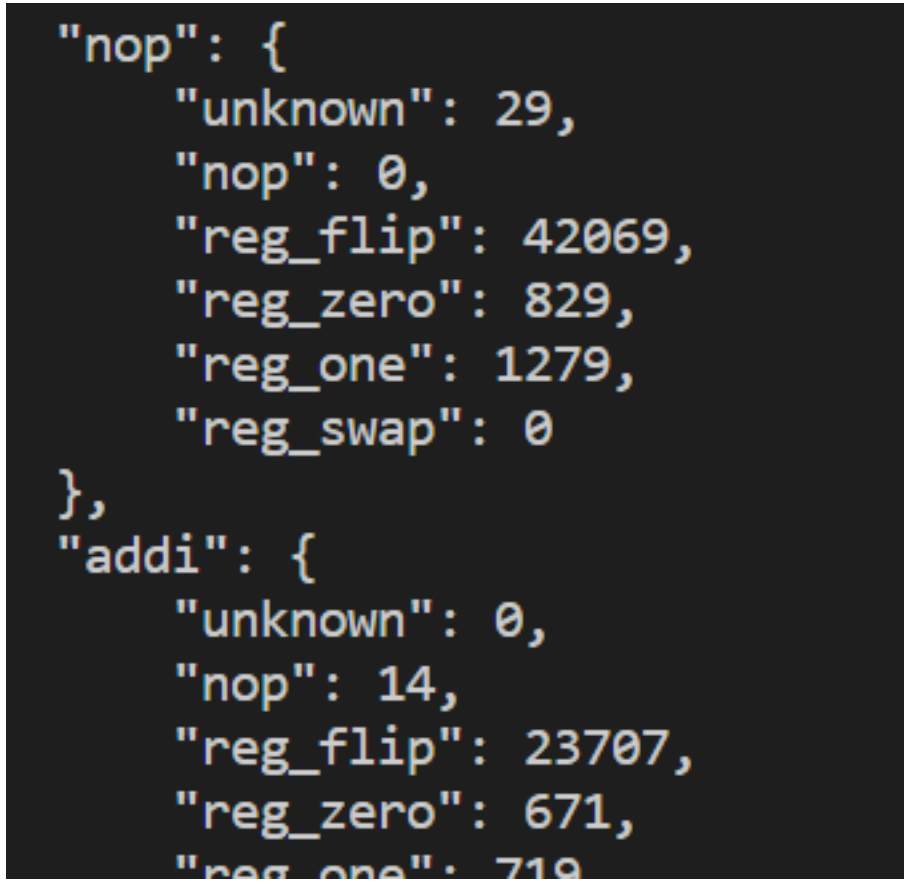
Fault differences

- For each RLS fault, we now know the instruction it was caused on and the arch state diff



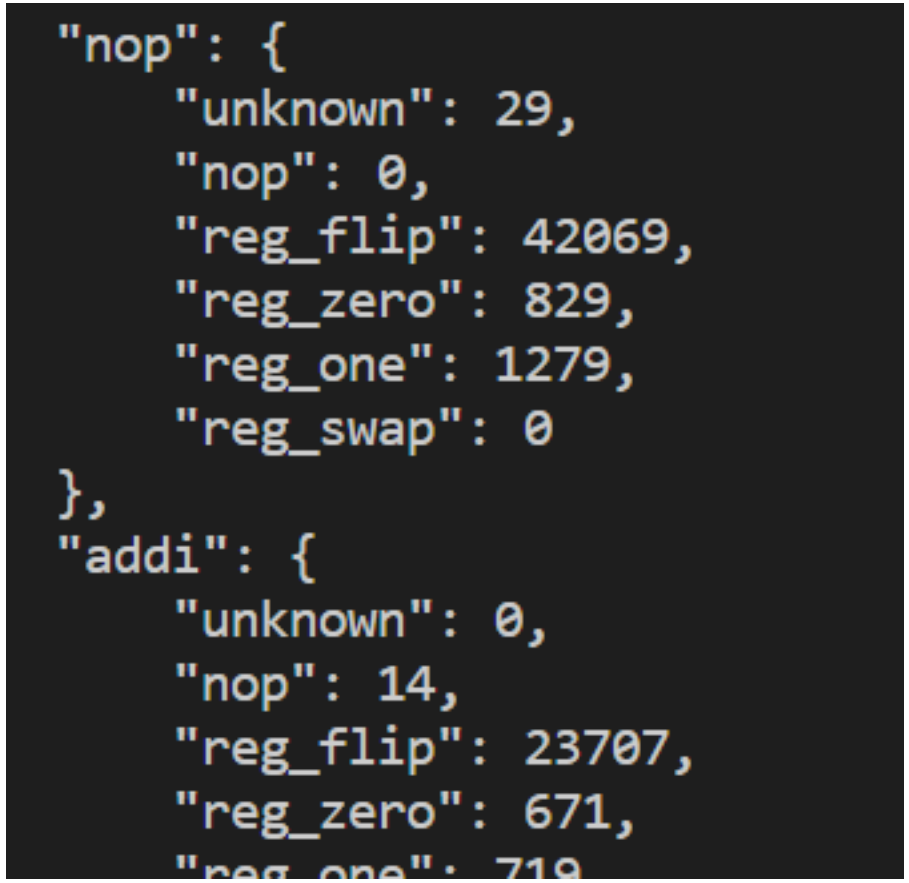
Classify the fault differences

- Characterize the effect into 6 classes
- Aggregate counts per mnemonic
- The resulting model we call Architectural Fault Effect Model (AFEM)
  - We ignore “unknown”



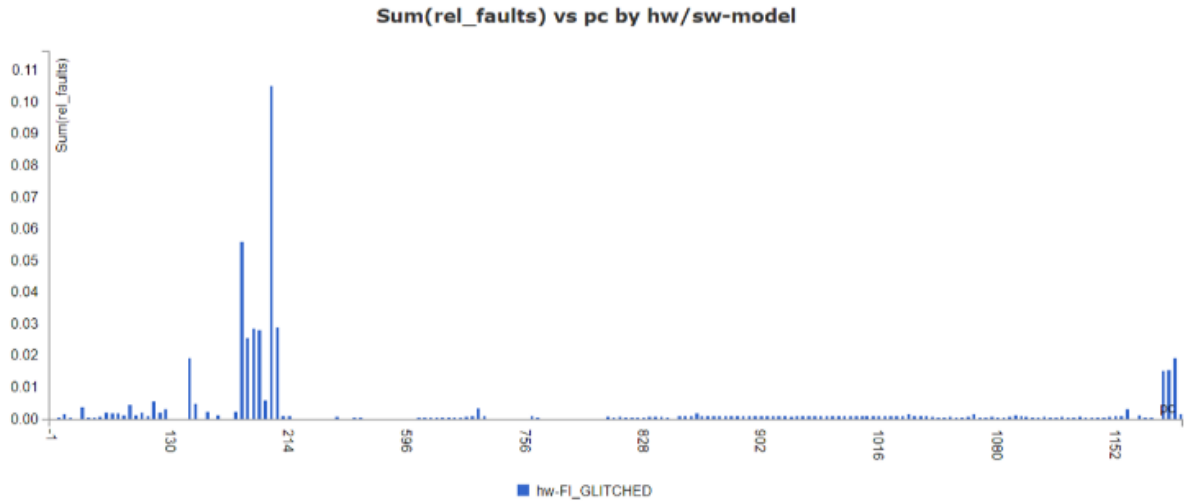
Instruction Level Simulator using AFEM

- Obtain an architectural state to fault
- Flip weighted coin to choose a class
- Further flip coin where applicable to select a register and/or bits therein



Evaluation for accuracy and runtime

- Compare ILS AFEM $x samples to RLS, NOP, NOP2, instruction bitflip
  - $x = how often to do a program run with fault per instruction
- Test program from FIRM*: branch test, memory test, register test
- Correlation between vectors PC -> successful fault probability; RLS baseline

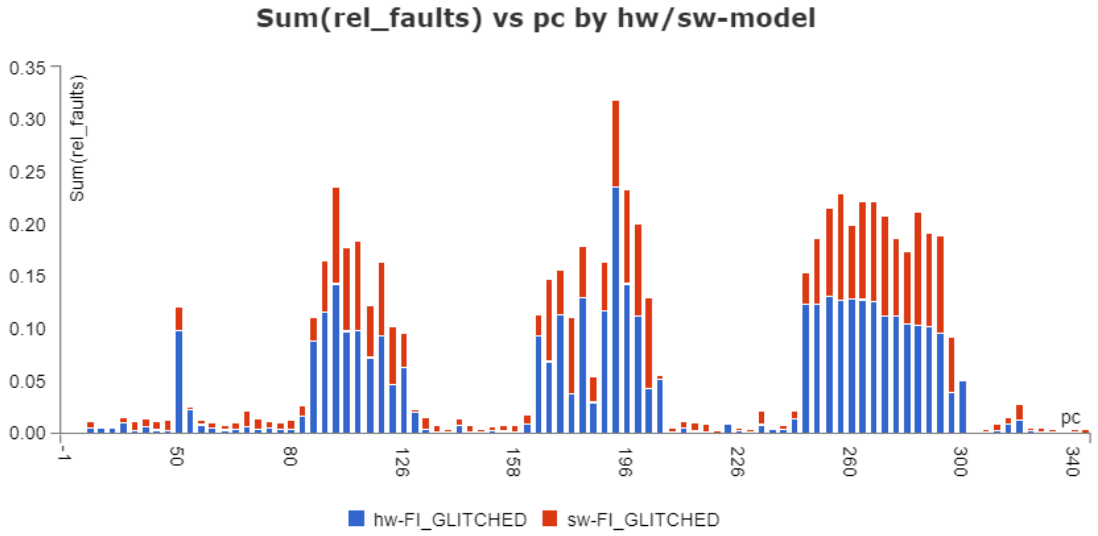


* FI Resistance Metric developed at Riscure by Carlo Maragno, Praveen Vadnala, Pierre-Yves Peneau, Chris Berg, Nisrine Jafri, Marc Witteman

Correlation rationale

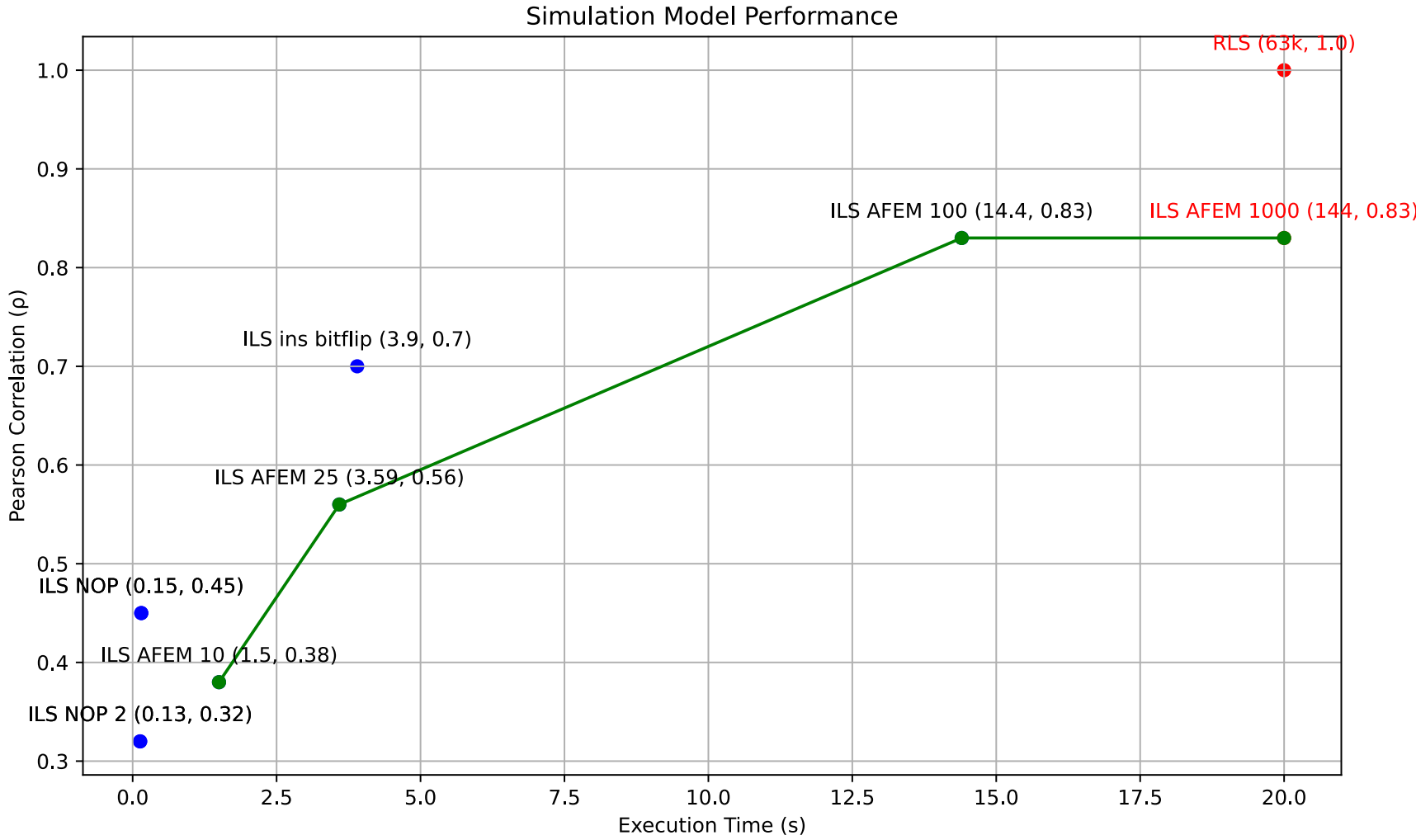
- We don’t care about absolute height of peaks (hw prob << sw prob)
- We do care about relative heights (sensitive areas in program)
- Per PC, prediction on hw prob given sw prob
- Pearson correlation fits this

Correlate ‘red’ and ‘blue’



Accuracy vs time tradeoffs

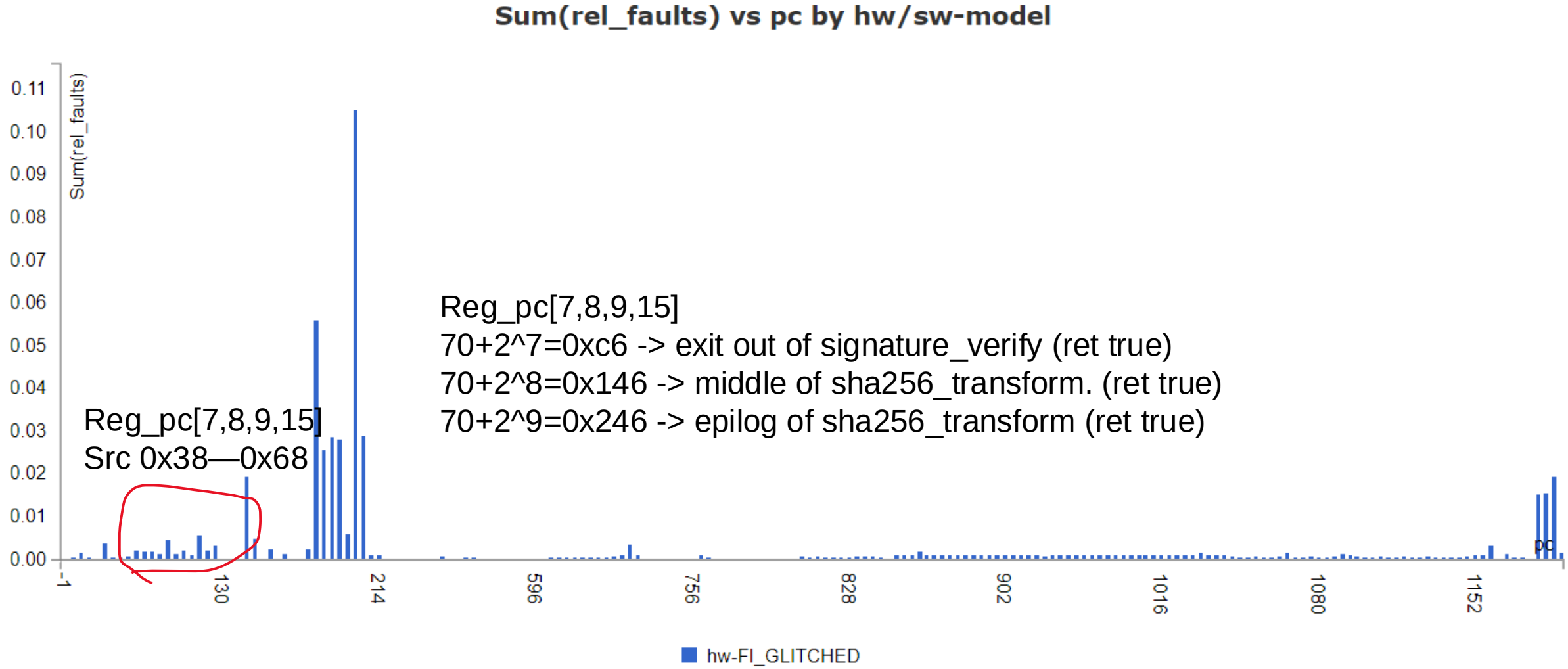
Simulation model	ρ	sim time (s)
RLS	1	63k
ILS AFEM 1000	0.83	144
ILS AFEM 100	0.83	14.4
ILS ins bitflip	0.70	3.9
ILS AFEM 25	0.56	3.59
ILS NOP	0.45	0.15
ILS AFEM 10	0.38	1.5
ILS NOP 2	0.32	0.13



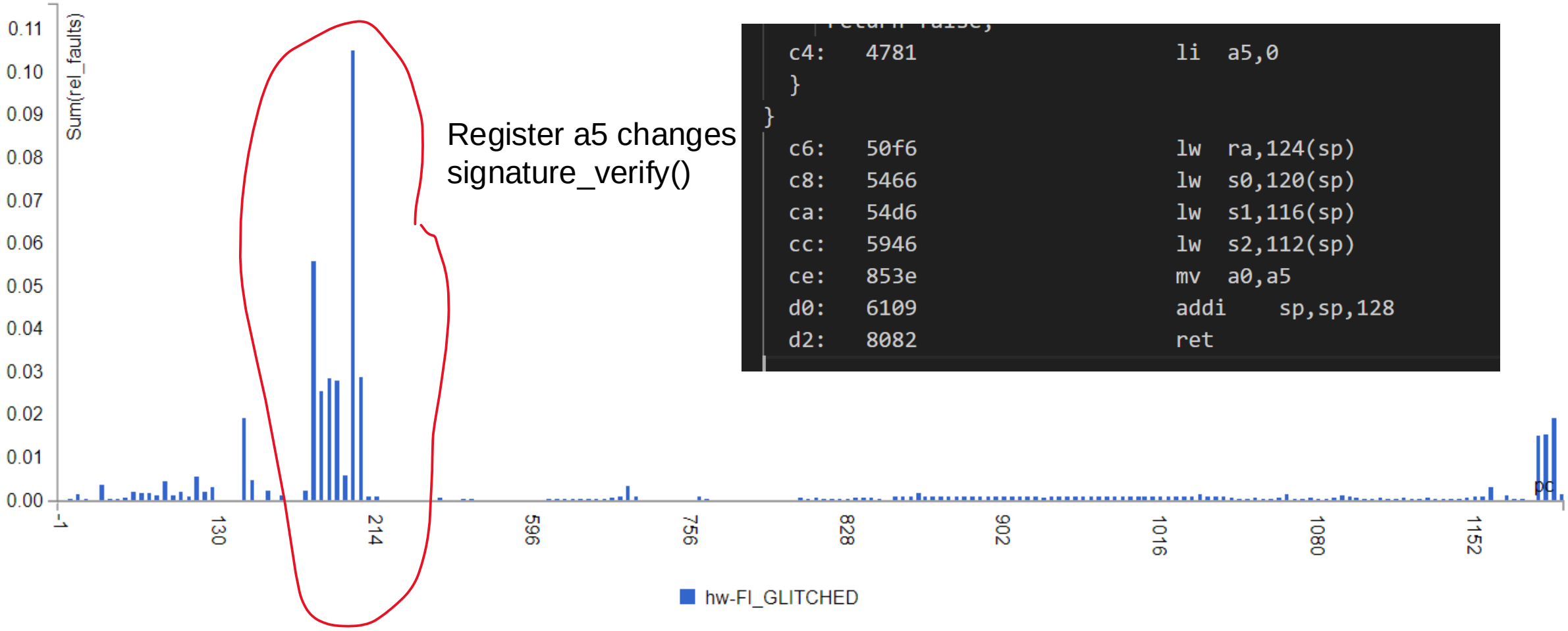
Application to boot code

- Test code is test code and may not represent real applications
- Try boot code (… also test code, admittedly)

```
int main(void) {
  if (!signature_verify()) {
    result[0] = 0xCAFEBABE;
  } else {
    application_entry_point();
  }
  // Do not return from this main function!
  while(1) {}
}
```


Sum(rel_faults) vs pc by hw/sw-model
Sum(rel_faults)
0.11
0.10
0.09
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0.00
-1
130
214
596
756
828
902
1016
1080
1152
pc
hw-FI_GLITCHED
Reg_pc[7,8,9,15]
Src 0x38—0x68
Reg_pc[7,8,9,15]
70+2^7=0xc6 -> exit out of signature_verify (ret true)
70+2^8=0x146 -> middle of sha256_transform. (ret true)
70+2^9=0x246 -> epilog of sha256_transform (ret true)

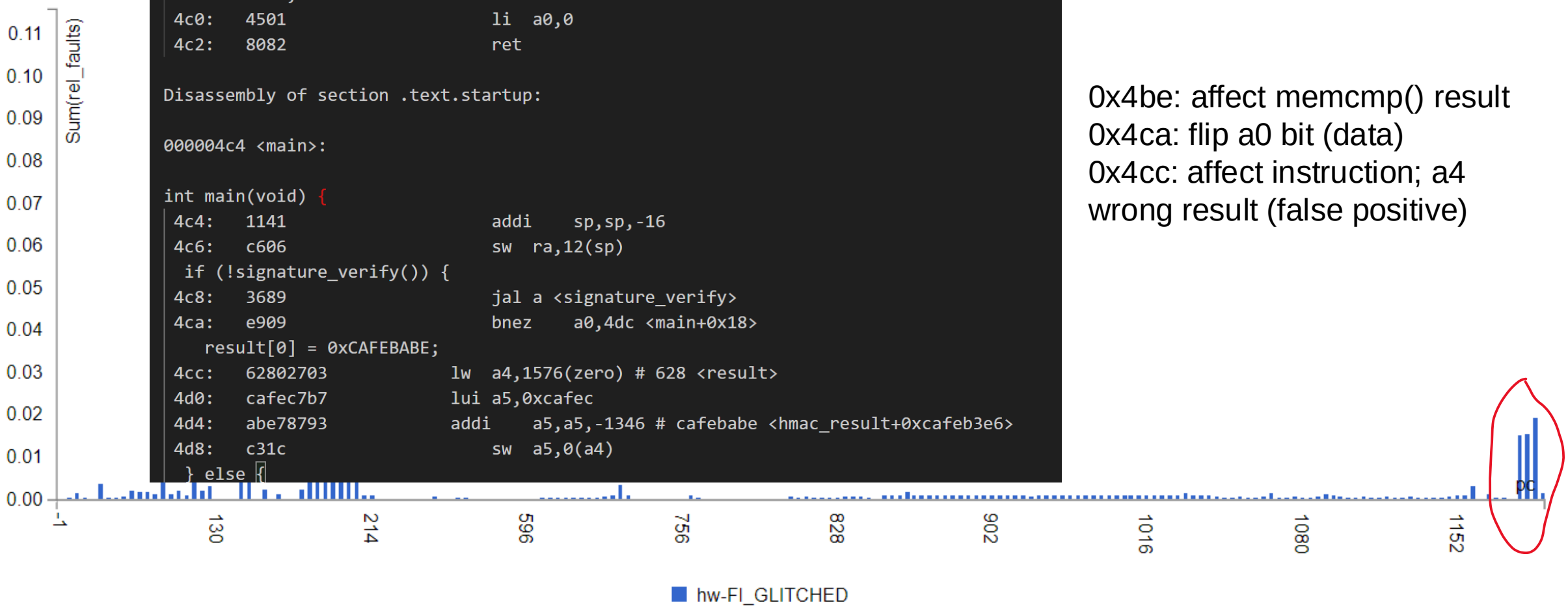
Sum(rel_faults) vs pc by hw/sw-model



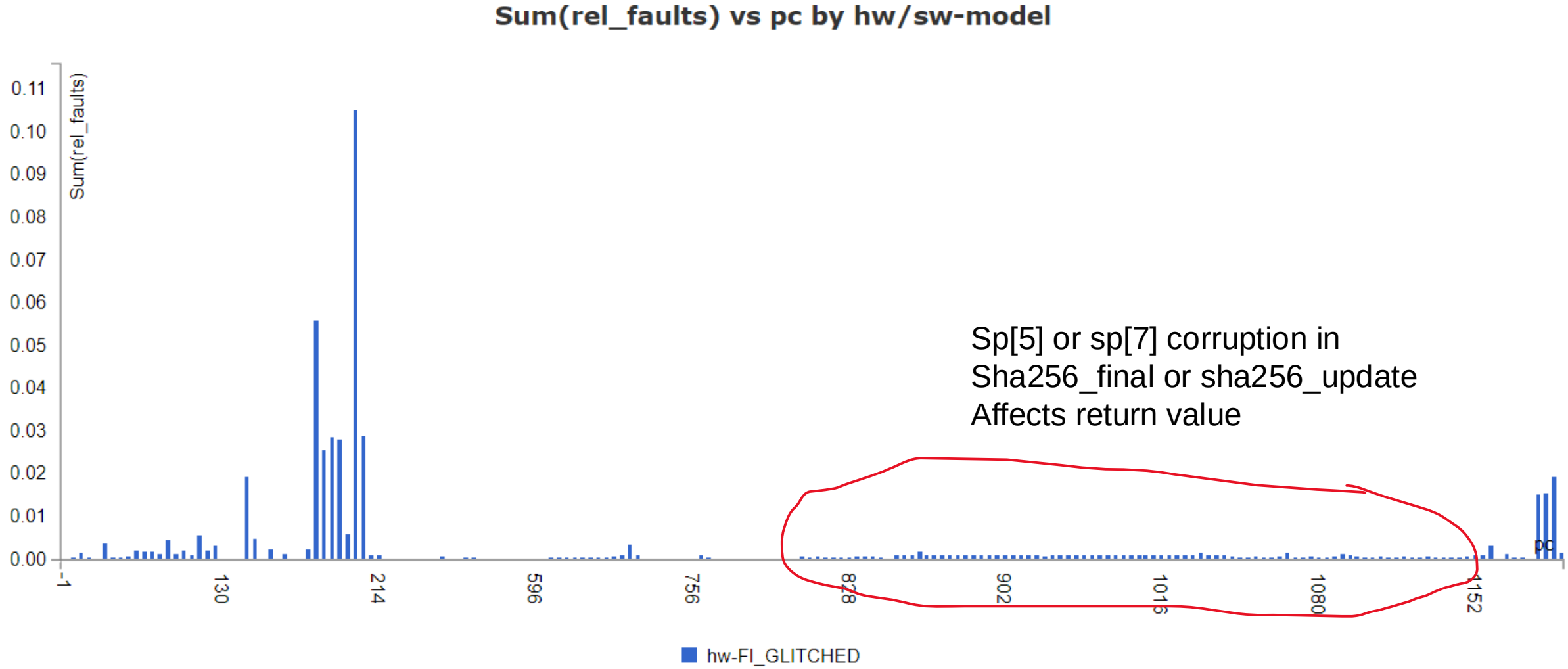
Register a5 changes signature_verify()

```
    c4:   4781                  li  a5,0
    }
}
    c6:   50f6                  lw  ra,124(sp)
    c8:   5466                  lw  s0,120(sp)
    ca:   54d6                  lw  s1,116(sp)
    cc:   5946                  lw  s2,112(sp)
    ce:   853e                  mv  a0,a5
    d0:   6109                  addi    sp,sp,128
    d2:   8082                  ret
```

```
}
 4be:   8082                    ret
   return 0;
 4c0:   4501                    li  a0,0
 4c2:   8082                    ret

Disassembly of section .text.startup:

000004c4 <main>:

int main(void) {
 4c4:   1141                    addi    sp,sp,-16
 4c6:   c606                    sw  ra,12(sp)
  if (!signature_verify()) {
 4c8:   3689                    jal a <signature_verify>
 4ca:   e909                    bnez    a0,4dc <main+0x18>
   result[0] = 0xCAFEBABE;
 4cc:   62802703            lw  a4,1576(zero) # 628 <result>
 4d0:   cafec7b7            lui a5,0xcafec
 4d4:   abe78793            addi    a5,a5,-1346 # cafebabe <hmac_result+0xcafeb3e6>
 4d8:   c31c                    sw  a5,0(a4)
  } else {
```



0x4be: affect memcmp() result
0x4ca: flip a0 bit (data)
0x4cc: affect instruction; a4 wrong result (false positive)



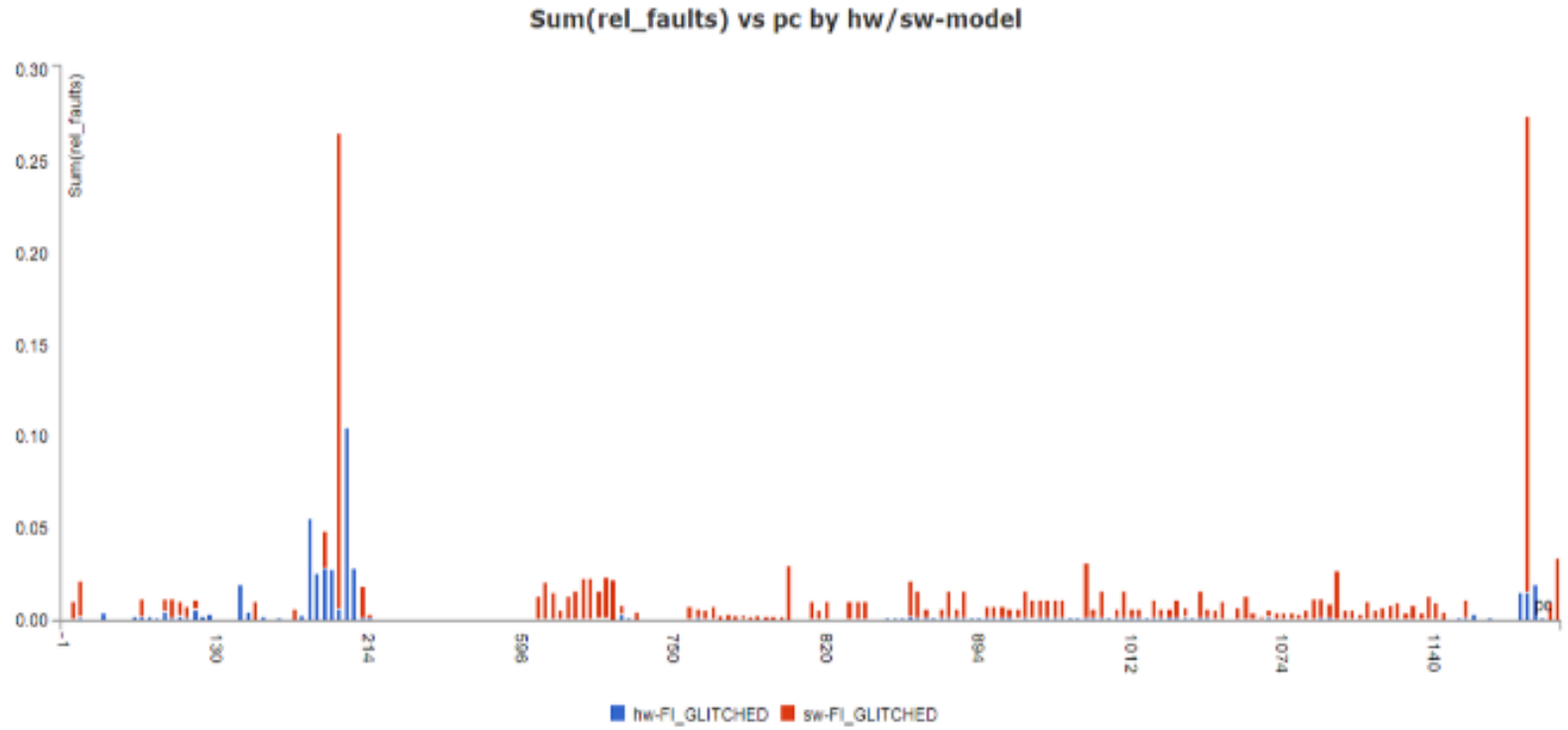
Sp[5] or sp[7] corruption in
Sha256_final or sha256_update
Affects return value

Improved AFEM: ‘regonly’

- Insight: most faults directly affect the register state
  - Processor registers are a significant chunk of the RTL registers
  - Most relevant: src register, the PC, return address (RA), or Stack Pointer (SP).
- Try next model: randomly select one of these CPU registers and flip a random bit within the chosen register.
  - No longer conditional probability on the instruction mnemonic

Regonly results

- Identifies very similar regions to the RLS
- ρ = 0.06; ‘regonly’ predicts vulnerable locations, not accurately assigns probability



Discussion

- Generalization beyond picorv32 / rv32imc to larger designs
- Simple programs used; most bit flips cause relevant faults
- Did not test programs / CPUs with countermeasures
- RLS as baseline, instead of post-si testing
- Simple input fault models (no complex multi-bit faults)
- Larger designs / software require significant #CPUs

Conclusions

- Predicting relevant code paths for FI is non-trivial, simulation/testing can help
- We can learn FI models from arbitrary CPU implementations
  - Faster than RLS
  - More accurate than existing instruction skip/bitflip models
- For our CPU, we can improve by manually finetuning the model

Future work

Future:

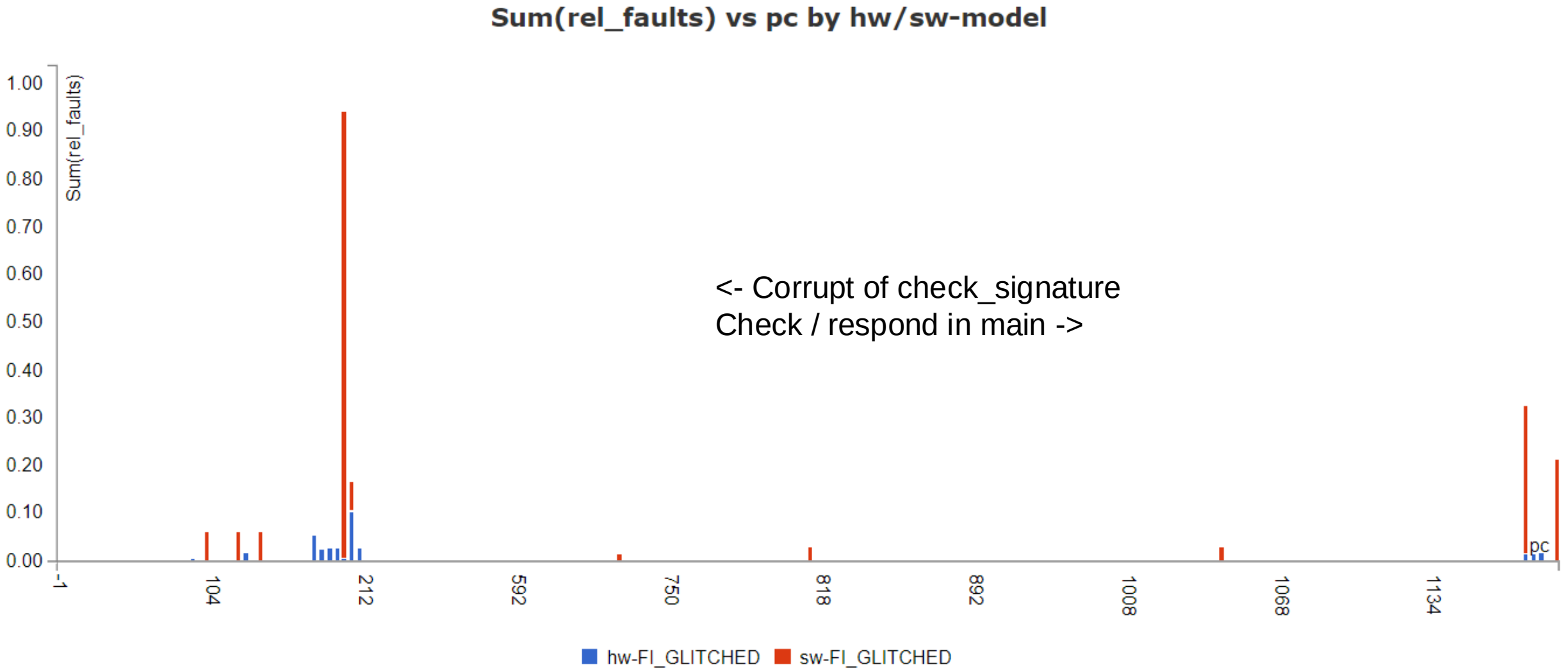
- Metrics that reflect indicating relevant code parts (as opposed to predicting the fault effects)
- Post-si evaluation of the models accuracy in sensitive code
- Validate how generic the models are

We would like to thank Google for their funding of this work

KEYSIGHT

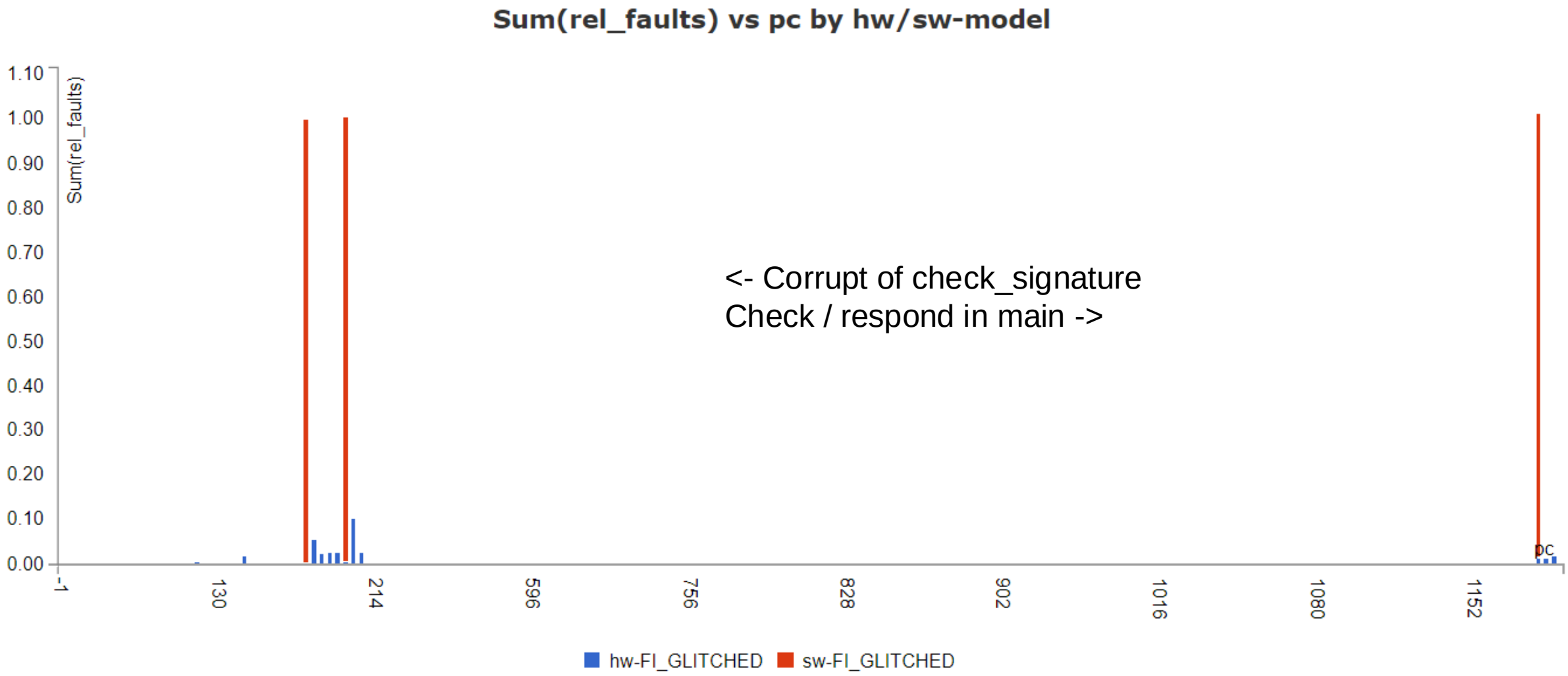
Thank you

INStruction bitflip

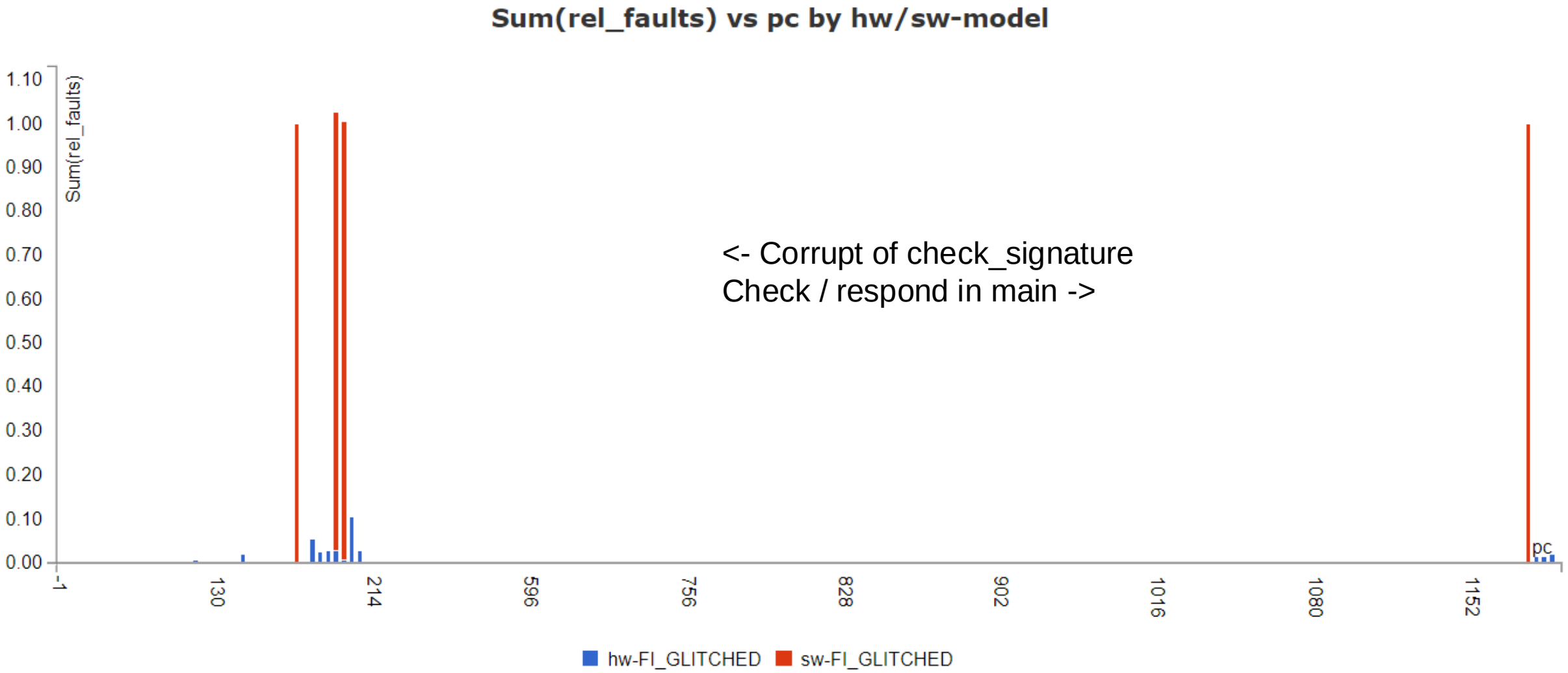


<- Corrupt of check_signature
Check / respond in main ->

NOP


Sum(rel_faults) vs pc by hw/sw-model
Sum(rel_faults)
1.10
1.00
0.90
0.80
0.70
0.60
0.50
0.40
0.30
0.20
0.10
0.00
-1
130
214
596
756
828
902
1016
1080
1152
pc
<- Corrupt of check_signature
Check / respond in main ->
hw-FI_GLITCHED
sw-FI_GLITCHED

nop2


Sum(rel_faults) vs pc by hw/sw-model
Sum(rel_faults)
1.10
1.00
0.90
0.80
0.70
0.60
0.50
0.40
0.30
0.20
0.10
0.00
-1
130
214
596
756
828
902
1016
1080
1152
pc
<- Corrupt of check_signature
Check / respond in main ->
hw-FI_GLITCHED
sw-FI_GLITCHED

Metric

Property	Desire	Metric
Relevance	rank most sensitive parts of code	correlation hw and sw
Actionable	highlight lines of code Indicate fault model	no metric needed, straight output from sw simulator
Performance	run sim in reasonable time	instructions/sec sec/full test