QuickCheck your language design*

Cătălin Hriţcu, **John Hughes**, Benjamin C. Pierce,
Antal Spector-Zabusky, Dimitrios Vytiniotis,
Arthur Azevedo de Amorim, Leonidas Lampropoulos

*based on *Testing noninterference, quickly* (ICFP 2013)

- **Suppose you want to guarantee…**
  - Type safety
  - Confidentiality
  - …other *global* properties
- Based on…
  - Static checks
  - Run-time checks
  - …
- **How can you get your design right?**
  - And do it *quickly*—for design exploration

- **Dynamic Information Flow Control**
  - *Taints* secrets at run-time
  - *Interrupts* execution if a secret is about to leak
- Guarantees confidentiality
  - A ”low observer” can infer nothing about high values

- **Our experiment**
  - A low-level abstract machine with dynamic IFC
  - Show how to find all of a reasonable class of bugs quickly

A very! simple stack-and-memory machine

- values = integers
- stack = list of values
- memory = list of values

instruction	stack before	stack after	memory
Push n	stk	n : stk	
Pop	n : stk	stk	
Add	n : m : stk	(n+m) : stk	
Load	a : stk	mem[a] : stk	
Store	a : n : stk	stk	mem[a] := n
Halt	stk	----	

A very! simple information-flow machine

- values = labeled integers
- stack = list of values
- labels = L and H
- memory = list of values

instruction	stack before	stack after	memory
Push n@X	stk	n@X : stk	
Pop	n@X : stk	stk	
Add	n@X : m@Y :stk	(n+m)@? : stk	
Load	a@X : stk	mem[a] : stk	
Store	a@X : n@Y : stk	stk	mem[a] := n@?
Halt	stk	----	

A very! simple wrong information-flow machine

- values = labeled integers
- stack = list of values
- labels = L and H
- memory = list of values

instruction	stack before	stack after	memory
Push n@X	stk	n@X : stk	
Pop	n@X : stk	stk	
Add	n@X : m@Y :stk	(n+m)@L : stk	
Load	a@X : stk	mem[a] : stk	
Store	a@X : n@Y : stk	stk	mem[a] := n@L
Halt	stk	----	

Noninterference (EENI)

- "secret inputs don't affect public outputs"
  - secret inputs = numbers labeled H in initial state
    - initial state = empty stack, memory all 0@L, instructions can contain secrets (Push 0@H)
  - public outputs = memory labeled L in halted state
- more precisely:
  - forall i_1 i_2, if $i_1 \approx i_2$ and $i_1 \rightarrow^* h_1$ and $i_2 \rightarrow^* h_2$ then $mem(h_1) \approx mem(h_2)$
  - $n_1@L \approx n_2@L$ iff $n_1=n_2$ $\quad$ $n_1@H \approx n_2@H$ always

Bugs

Counterexample #1

memory	stack	next instruction
[0@L]	[]	Push {0/1}@H
[0@L]	[{0/1}@H]	Push 0@L
[0@L]	[0@L,{0/1}@H]	Store
[{0/1}@L]	[]	Halt

Fixing bug in Store

instruction	stack before	stack after	memory
Store	a@X : n@Y : stk	stk	mem[a] := n@Y

Counterexample #2

memory	stack	next instruction
[0@L,0@L]	[]	Push 1@L
[0@L,0@L]	[1@L]	Push {0/1}@H
[0@L,0@L]	[{0/1}@H,1@L]	Store
[{1/0}@L,{0/1}@L]	[]	Halt

Fixing 2nd bug in Store

instruction	stack before	stack after	memory
Store	a@X : n@Y : stk	stk	mem[a] := n@X⌊⌋Y

Counterexample #3

memory	stack	next instruction
[0@L,0@L]	[]	Push 0@L
[0@L,0@L]	[0@L]	Push {0/1}@H
[0@L,0@L]	[{0/1}@H,0@L]	Store
[{0@H/0@L},{0@L/0@H}]	[]	Halt

Fixing 3nd bug in Store

stack before	side condition	stack after	memory
a@X : n@Y : stk	X ≤ labOf(mem[a])	stk	mem[a] := n@X⊔Y

No sensitive upgrade [Steve Zdancewic's PhD, 2002]

Counterexample #4

memory	stack	next instruction
[0@L]	[]	Push 0@L
[0@L]	[0@L]	Push {0/1}@H
[0@L]	[{0/1}@H,0@L]	Add
[0@L]	[{0/1}@L]	Push 0@L
[0@L]	[0@L,{0/1}@L]	Store
[{0/1}@L]	[]	Halt

Fixing bug in Add

instruction	stack before	stack after	memory
Add	n@X : m@Y :stk	(n+m)@(X⊔Y) : stk	

Counterexample #5

memory	stack	next instruction
[0@L,0@L]	[]	Push 1@L
[0@L,0@L]	[1@L]	Push 0@L
[0@L,0@L]	[0@L,1@L]	Store
[1@L,0@L]	[]	Push {1/0}@H
[1@L,0@L]	[{1/0}@H]	Load
[1@L,0@L]	[{0/1}@L]	Push 0@L
[1@L,0@L]	[0@L,{0/1}@L]	Store
[{0/1}@L,0@L]	[]	Halt

Fixing bug in Load

instruction	stack before	stack after	memory
Load	a@X : stk	mem[a]@X : stk	

How did we do this?

- QuickCheck: random testing tool for Haskell
- Property ~= Boolean Haskell expression
  - QC generates random instances for variables
  - implications treated specially
    - failing preconditions cause test case to be discarded
  - Failed tests are shrunk to minimal counterexamples

- Out of the box: complete failure! ☹
  - couldn't find any bug; astronomic discard rate

(Re)phrasing noninterference

Original ☹

for random i_1,
for random i_2,
 if $i_1 \approx i_2$ ← **Rare**
  and $i_1 \rightarrow^* h_1$
  and $i_2 \rightarrow^* h_2$
 then
  $mem(h_1) \approx mem(h_2)$

Much better ☺

for random i_1,
for random
 ≈ variation i_2 of i_1,
 if $i_1 \rightarrow^* h_1$
  and $i_2 \rightarrow^* h_2$
 then
  $mem(h_1) \approx mem(h_2)$

Naive generation

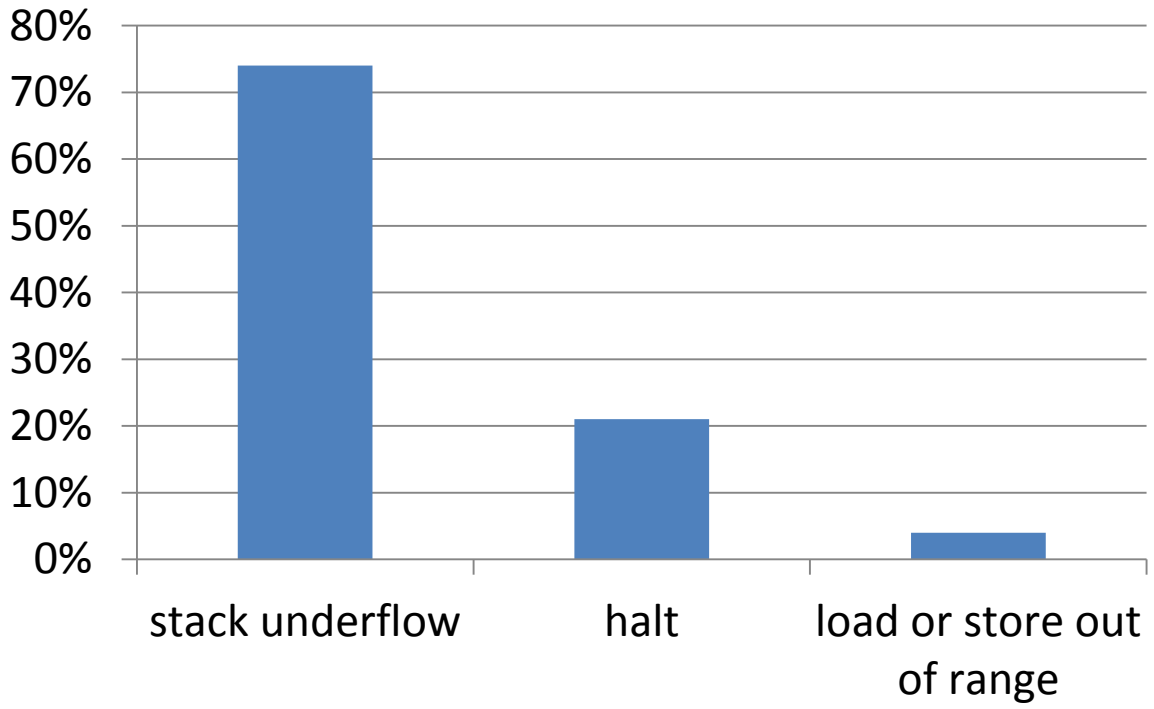
- How can we evaluate how ~~good~~ bad our testing is?
  - add bugs one at a time and see how fast they're found
  - Mean Time to Find (MTTF)

	Bug	MTTF
from before	1nd for Store	8s
	2st for Store	∞*
	3rd for Store	47s
	Add	83s
	Load	∞*
new	Push	4s

*not found in 300s

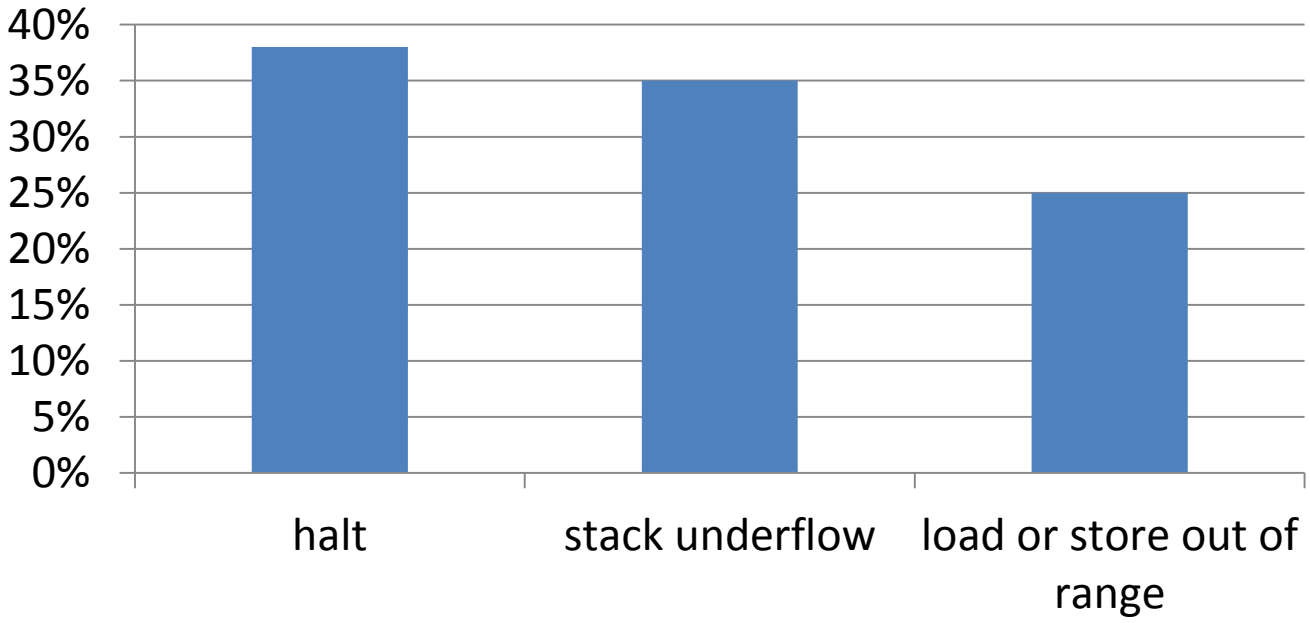
Some statistics

- discard rate: 79% (not reaching halted states)
- average number of execution steps: 0.47
- reasons for termination



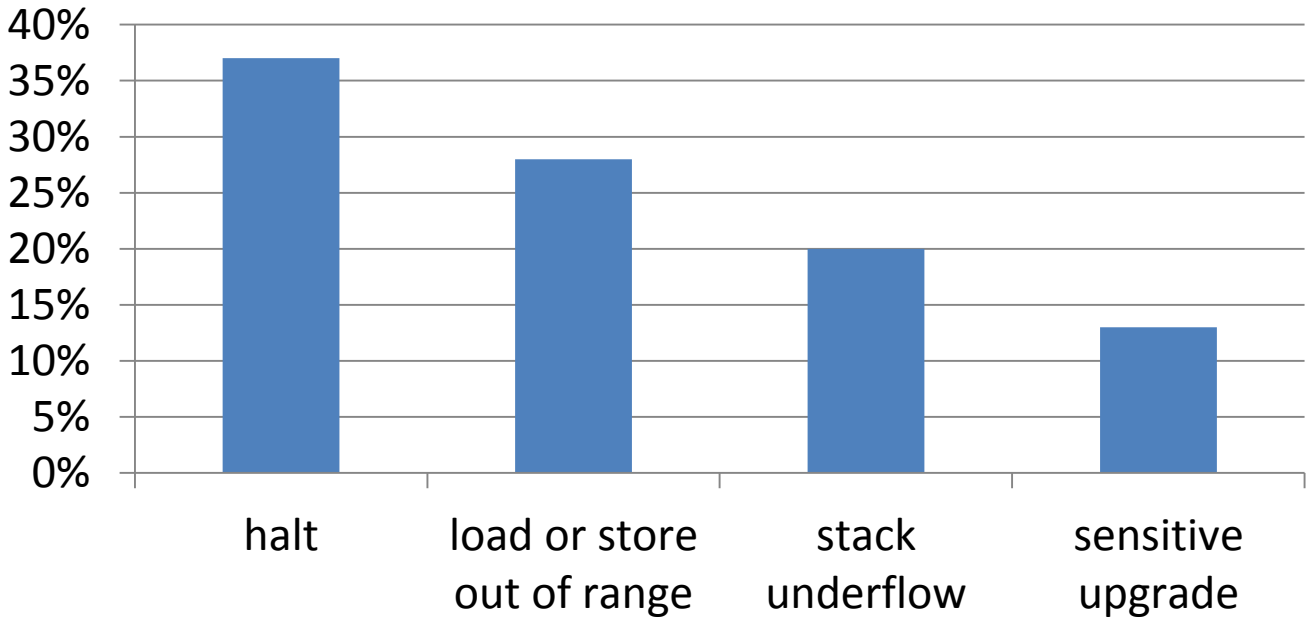
Weighted distribution on instructions

- increased chance of getting Push or Halt
- average number of execution steps: 2.69
- reasons for termination



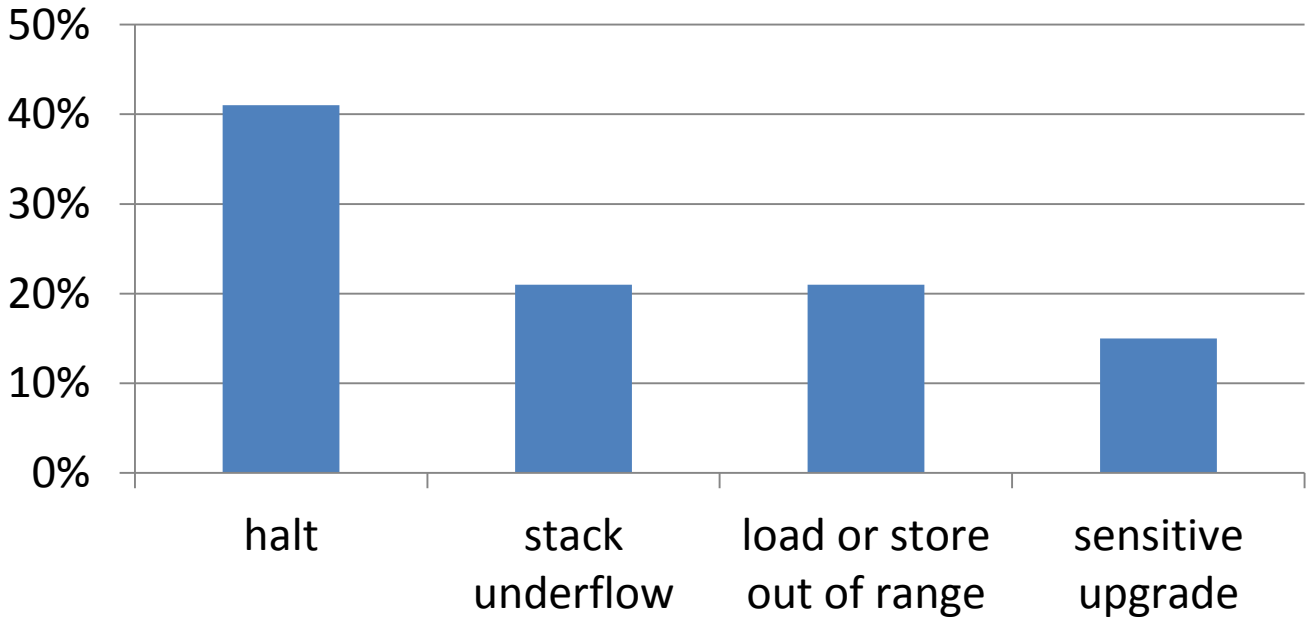
Instruction sequences

- generating useful instruction sequences more often (e.g. Push a; Store, where a is valid addr)
- average number of execution steps: 3.86
- reasons for termination



Smart integers

- generating valid code and data addr. more often
  - varying valid addr with high probability to other addr
- average number of execution steps: 4.22
- reasons for termination



They don’t just run longer ...

- Smarter generation finds bugs much faster
- Mean Time to Find (MTTF)

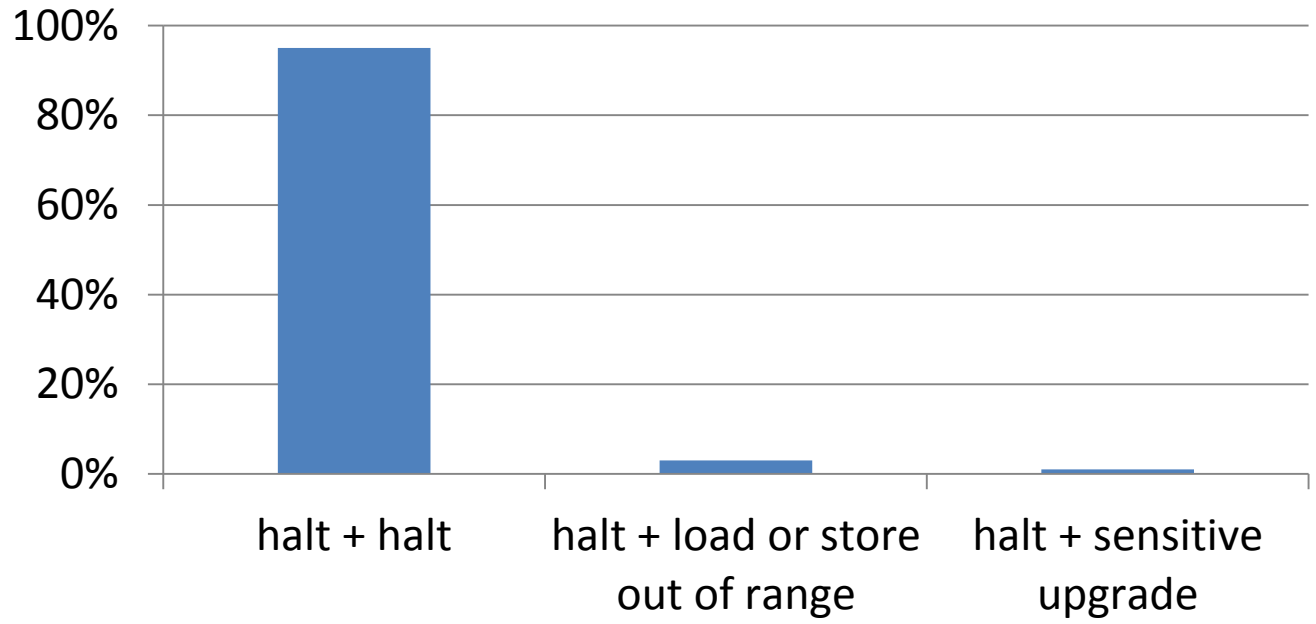
Bug	Naive	Smarter
1st for Store	7660.07ms	0.31ms
2nd for Store	∞	32227.10ms
3rd for Store	47365.97ms	0.12ms
Add	83247.01ms	30.05ms
Load	∞	2258.93ms
Push	3552.54ms	0.07ms

Generation by execution

- try to generate instruction seq that doesn't crash
- maintain a current state
  - generate instr(s) that make sense in current state
  - run instr(s) to obtain new current state
  - fully precise for straight-line code
- jumps forward easy, jumps backward harder
  - look ahead 2 steps before committing to jump
  - current state still not always accurate
- give Halt more weight as execution gets longer

Statistics for generation by execution

- average number of execution steps:
  - 11.6 for original program, 11.26 for variation
- reasons for termination (original + variation)



Generation by execution finds bugs faster

Bug	Naive	Smarter	By Exec	
1st for Store	7660.07ms	0.31ms	0.02ms	
2nd for Store	∞	32227.10ms	1233.51ms	
3rd for Store	47365.97ms	0.12ms	0.25ms	
Add	83247.01ms	30.05ms	0.87ms	
Load	∞	2258.93ms	4.03ms	
Push	3552.54ms	0.07ms	0.01ms	
Arith. mean	∞	5752.76ms	206.45ms	28x
Geom. mean	∞	13.33ms	0.77ms	17x
tests / second	24129	7915	3284	
discard rate	79%	59%	4%	

Adding control flow

- jumps & procedures
  - New program counter taken from the stack
- 14 bugs = 6 old bugs + 8 new bugs
- GenByExec
  - finds 13 of them in 0.22ms to 69s
  - misses one completely ☹

Improving the property

Counterexample to Load bug

takes 155ms to find now; 433 tests (average)

	memory	stack	next instruction	
setting up	[0@L,0@L]	[]	Push 1@L	
	[0@L,0@L]	[1@L]	Push 0@L	
	[0@L,0@L]	[0@L,1@L]	Store	
	[1@L,0@L]	[]	Push {1/0}@H	
	[1@L,0@L]	[{1/0}@H]	Load	bug
observing	[1@L,0@L]	[{0/1}@L]	Push 0@L	
	[1@L,0@L]	[0@L,{0/1}@L]	Store	
	[{0/1}@L,0@L]	[]	Halt	

Stronger noninterference

Current ☹

for random i_1,
for random
 ≈ variation i_2 of i_1,
 if $i_1 \rightarrow^* h_1$
  and $i_2 \rightarrow^* h_2$
 then
  $mem(h_1) \approx mem(h_2)$

Better ☺

for random q_1,
for random
 ≈ variation q_2 of q_1,
 if $q_1 \rightarrow^* h_1$
  and $q_2 \rightarrow^* h_2$
 then
  $h_1 \approx h_2$

q - quasi initial = arbitrary, but labOf(pc)≠H
(control not affected by secrets) ≈ equates all H states

Counterexamples to Load bug

used to take 155ms to find; 433 tests
now it takes 6ms to find; 12 tests (average)

memory	stack	next instruction
[0@L,1@L]	[]	Push {0/1}@H
[0@L,1@L]	[{0/1}@H]	Load
[0@L,1@L]	[{0/1}@L]	Halt

memory	stack	next instruction
[0@L,1@L]	[{1/0}@H]	Load
[0@L,1@L]	[{1/0}@L]	Halt

This finds all bugs, including ...

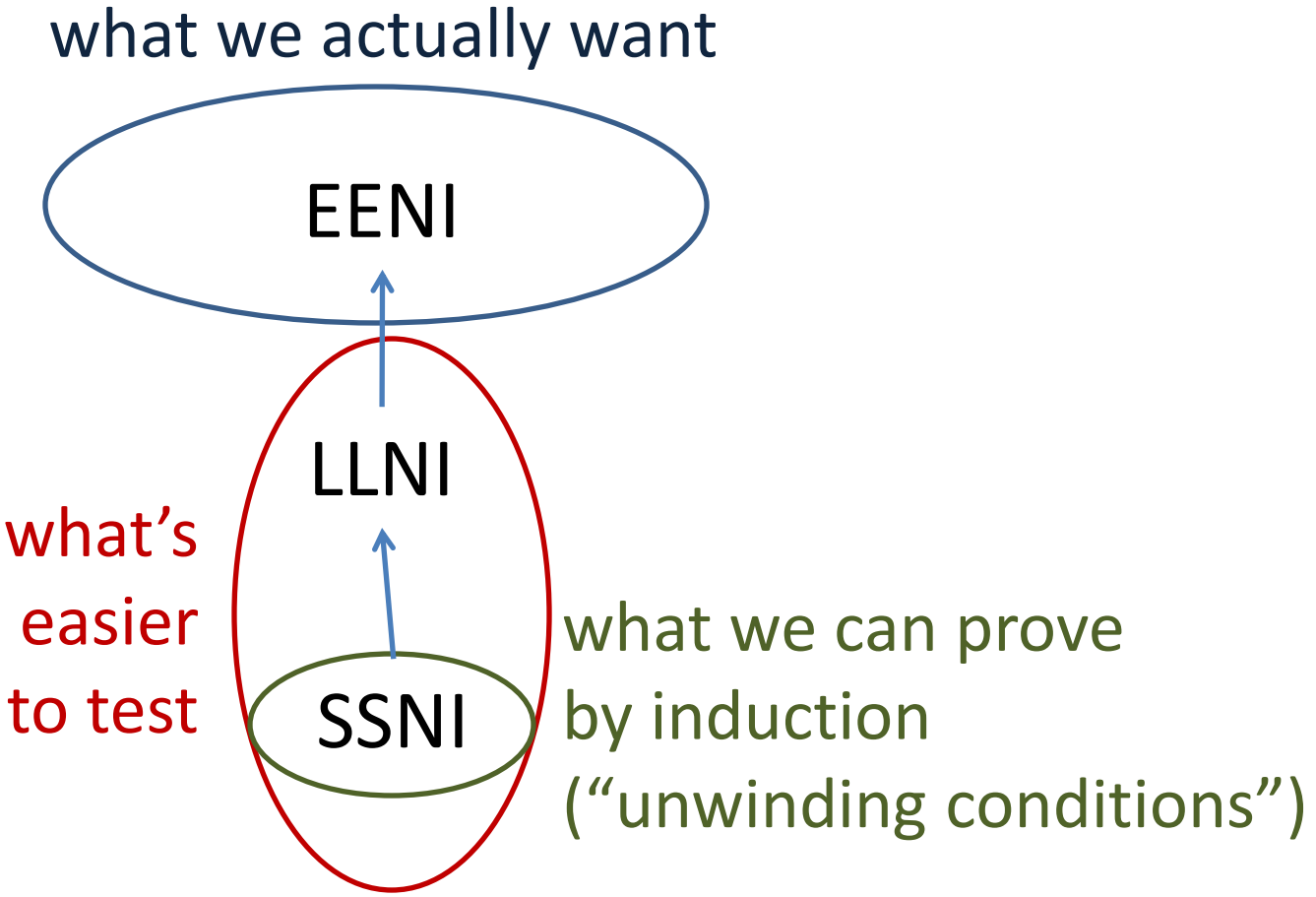
it takes 16s to find this one (average)

memory	stack	next instruction
[]	[ARet (3,False)@L,0@L,ARet (4,True)@L]	Push {3/2}@H
[]	[{3/2}@H,ARet (3,False)@L,0@L,ARet (4,True)@L]	Jump
execution 1 continues ...		
[]	[ARet (3,False)@L,0@L,ARet (4,True)@L]	Return
[]	[0@L,ARet (4,True)@L]	Return
[]	[0@L]	Halt
execution 2 continues ...		
[]	[ARet (3,False)@L,0@L,ARet (4,True)@L]	Pop
[]	[0@L,ARet (4,True)@L]	Return
[]	[0@H]	Halt

Surprises

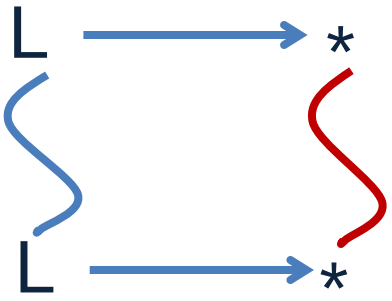
- Jump cannot return the PC from high to low
  - Can only be achieved by call/return
- High return addresses cannot ≈ high integers
  - They can be distinguished by Return!
- The *number* of return values must be specified at a call
  - Or else a secret can be leaked by choosing different Return instructions

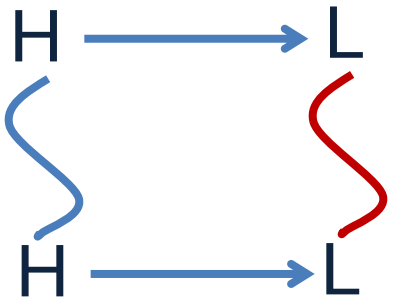
Even stronger noninterference

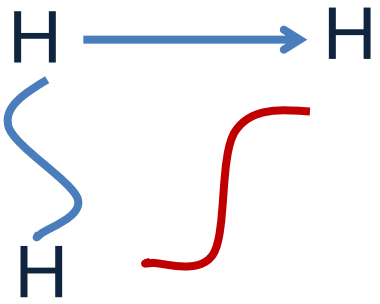


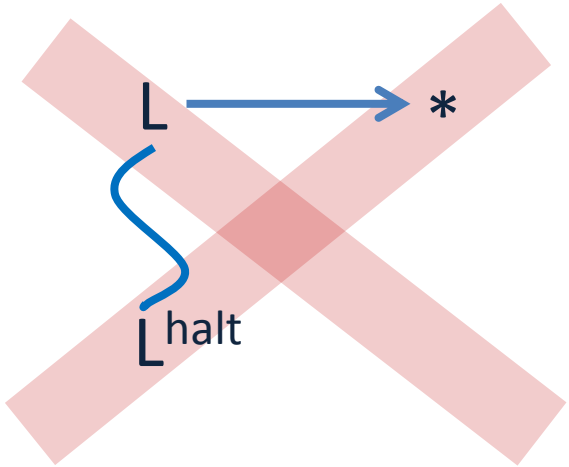
Single-step noninterference (SSNI)

easiest to test and suitable for proof ("unwinding conditions")









SSNI finds each bug in under 17ms

	EENI (with all improvements)	SSNI
Arith. mean MTTF	1526.75ms	2.01ms
Geom. mean MTTF	46.48ms	0.47ms
tests/s	2391	18407
discard rate	69%	9%

Tradeoff:

SSNI requires discovering stronger invariants

invariants of real SAFE machine are very complicated

Why shrink counterexamples?

memory	stack	next instruction
[0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L]	[]	Push {0/15}@H
[0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L]	[{0/15}@H]	Load
[0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L]	[0@L]	Pop
[0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L]	[]	Push -5@L
[0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L]	[-5@L]	Push 17@L
[0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L]	[17@L,-5@L]	Push 0@L
[0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L]	[0@L,17@L,-5@L]	Store
[17@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L]	[-5@L]	Push 1@L
[17@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L]	[1@L,-5@L]	Store
[17@L,-5@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L]	[]	Push {21/3}@H
[17@L,-5@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L]	[{21/3}@H]	Push 2@L
[17@L,-5@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L]	[2@L,{21/3}@H]	Load
[17@L,-5@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L]	[0@L,{21/3}@H]	Pop
[17@L,-5@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L]	[{21/3}@H]	Push 1{/0}@H
[17@L,-5@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L]	[1{/0}@H,{21/3}@H]	Push 8@L
[17@L,-5@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L]	[8@L,1{/0}@H,{21/3}@H]	Store
[17@L,-5@L,0@L,0@L,0@L,0@L,0@L,0@L,1{/0}@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L]	[{21/3}@H]	Push {9/17}@H
[17@L,-5@L,0@L,0@L,0@L,0@L,0@L,0@L,1{/0}@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L]	[{9/17}@H,{21/3}@H]	Push {3/0}@H
[17@L,-5@L,0@L,0@L,0@L,0@L,0@L,0@L,1{/0}@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L]	[{3/0}@H,{9/17}@H,{21/3}@H]	Load
[17@L,-5@L,0@L,0@L,0@L,0@L,0@L,0@L,1{/0}@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L]	[{0/17}@L,{9/17}@H,{21/3}@H]	Store
[{9/17}@L,-5@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,1{/0}@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,{0/17}@L,0@L,0@L,0@L]	[{21/3}@H]	Push 3@L
[{9/17}@L,-5@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,1{/0}@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,{0/17}@L,0@L,0@L,0@L]	[3@L,{21/3}@H]	Push 1@H
[{9/17}@L,-5@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,1{/0}@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,{0/17}@L,0@L,0@L,0@L]	[1@H,3@L,{21/3}@H]	Load
[{9/17}@L,-5@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,1{/0}@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,{0/17}@L,0@L,0@L,0@L]	[-5@L,3@L,{21/3}@H]	Pop
[{9/17}@L,-5@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,1{/0}@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,{0/17}@L,0@L,0@L,0@L]	[3@L,{21/3}@H]	Push 1@L
[{9/17}@L,-5@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,1{/0}@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,{0/17}@L,0@L,0@L,0@L]	[1@L,3@L,{21/3}@H]	Push 19@L
[{9/17}@L,-5@L,0@L,0@L,0@L,0@L,0@L,0@L,1{/0}@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,0@L,{0/17}@L,0@L,0@L,0@L]	[19@L,1@L,3@L,{21/3}@H]	Halt

Shrinking

- Greedy search for smaller counterexample
- Default QuickCheck shrinking, + we specify "shrinking candidates"
  - Lots of tricks to shrink fast and effectively
- Reported counterexamples are almost always minimal

Summary

- With careful generation, a well-formulated property, and good shrinking…

  we ***CAN*** find minimal counterexamples to non-interference fast and reliably

- …in a toy (but interesting) situation
  - Now applying the same techniques in anger to the real CRASH/SAFE abstract machine