Test-Driven Development of an Information-Flow ISA

A QuickCheck Adventure

Arthur Azevedo de Amorim, Catalin Hritcu, John Hughes, Leonidas Lampropoulos, Ulf Norell, **Benjamin C. Pierce**, Dimitrios Vytiniotis, Antal Spector-Zabusky

WG2.8
November 2012

Suppose…

1. … we wanted to design a machine architecture with dynamic information-flow tracking…
2. … and we wanted to use QuickCheck to help get it right.

Could that be done?

Let’s find out!

A *very!* Simple Stack-and-Memory Machine

- Values = integers
- Stack = list of values
- Memory = array of values
- PC = value
- Instructions…

Instruction	Stack before	Stack after	Memory
Push n	stk	n : stk	
Add	a : b : stk	(a+b) : stk	
Load	a : stk	mem[a] : stk	
Store	b : a : stk	stk	mem[b] := a

A *very!* Simple Information-Flow Machine

- Values = *labeled* integers (1@L, 2@H, …)
- Stack = list of values
- Memory = array of values
- PC = value
- Instructions…

Instruction	Stack before	Stack after	Memory
Push n@l	stk	n@**l** : stk	
Add	a@l : b@l’ : stk	(a+b)@**?** : stk	
Load	a@l : stk	mem[a]@**?** : stk	
Store	b@l : a@l’ : stk	stk	mem[b] := a@**?**

“Correctness”?

- A nice property: *noninterference*
  - “High inputs do not flow to low outputs”
- More formally:
  - If initial machine states differ only in high values, then “low observations” of execution traces are the same
- Yet more formally:
  - Forall s,s’ with s ~~~ s’,

    observe(trace s) ~~~ observe(trace s’)

If the adversary can’t tell the difference between starting states, they can’t tell the difference between executions

“Observe”?

- Design choice:
  - Introduce special “I/O events”?
  - Observe memory?
    - Values only?
    - Values and labels?
  - Stack?
  - PC?

observe (trace s)
=
sequence of memories as
s executes

~~~ in Haskell

```
class Observable a where
  (~~~) :: a -> a -> Bool


instance Observable a => Observable (Labeled a) where
  (Labeled L x) ~~~ (Labeled L y) = x ~~~ y
  (Labeled H _) ~~~ (Labeled H _) = True
  _ ~~~ _ = False


instance Observable a => Observable [a] where
  xs ~~~ ys  =  length xs == length ys
                && and (zipWith (~~~) xs ys)
```

QuickChecking Noninterference

- For arbitrary s,s’,

  s ~~~ s’

  ➔ observe(trace s) ~~~ observe(trace s’)

Rare!

Ask QC to look for counterexamples to this property…

- For arbitrary s,

  for an arbitrary ~~~ variation s’ of s,

  observe(trace s) ~~~ observe(trace s’)

Better!

Variation in Haskell

```
class Observable a where
  …
  vary :: a -> Gen a
```

Invariant: $\forall a' \in \text{vary } a. \; a \sim\sim\sim a'$

```
instance (Arbitrary a, Observable a) =>
         Observable (Labeled a) where
  …
  vary (Labeled H x) = Labeled H <$> arbitrary
  vary a             = return a
```

Ready for bugs!

Instruction	Stack before	Stack after	Memory
Push n@l	stk	n@l : stk	
Add	a@l : b@l' : stk	(a+b)**@L** : stk	
Load	a@l : stk	**mem[a]** : stk	
Store	b@l : a@l' : stk	stk	mem[b] := a**@l'**

Let's take them one at a time…

What if Add doesn't taint its result?

```
[Add,Push 0@L,Store]

1@L       M=[0@L]            S=[{0@H/1@H},1@L]
2@L       M=[0@L]            S=[{1@L/2@L}]
3@L       M=[0@L]            S=[0@L,{1@L/2@L}]
4@L       M=[{1@L/2@L}]      S=[]
```

What if Load doesn't taint its result?

```
[{Push 0@H/Push 2@H},Load,Store]

1@L      M=[0@L,0@L,1@L]                S=[1@L]
2@L      M=[0@L,0@L,1@L]                S=[{0@H/2@H},1@L]
3@L      M=[0@L,0@L,1@L]                S=[{0@L/1@L},1@L]
4@L      M=[{1@L/0@L},{0@L/1@L},1@L]    S=[]
```

What if Store doesn't taint the value stored?

```
[Store]

1@L     M=[0@H,0@H]                      S=[{1@H/0@H},0@L]
2@L     M=[{0@H/0@L},{0@L/0@H}]          S=[]


[Store]

1@L     M=[0@L,0@L]                      S=[{0@H/1@H},1@L]
2@L     M=[{1@L/0@L},{0@L/1@L}]          S=[]
```

How many tests are needed?

to find a counter-example

Bug	Information leak through memory			
Add fails to taint	506			
Load fails to taint	50582			
Store fails to taint	42855			

(Averaged over 10 runs of QuickCheck)

Optimisation

```
[Add,Push 0@L,Store]

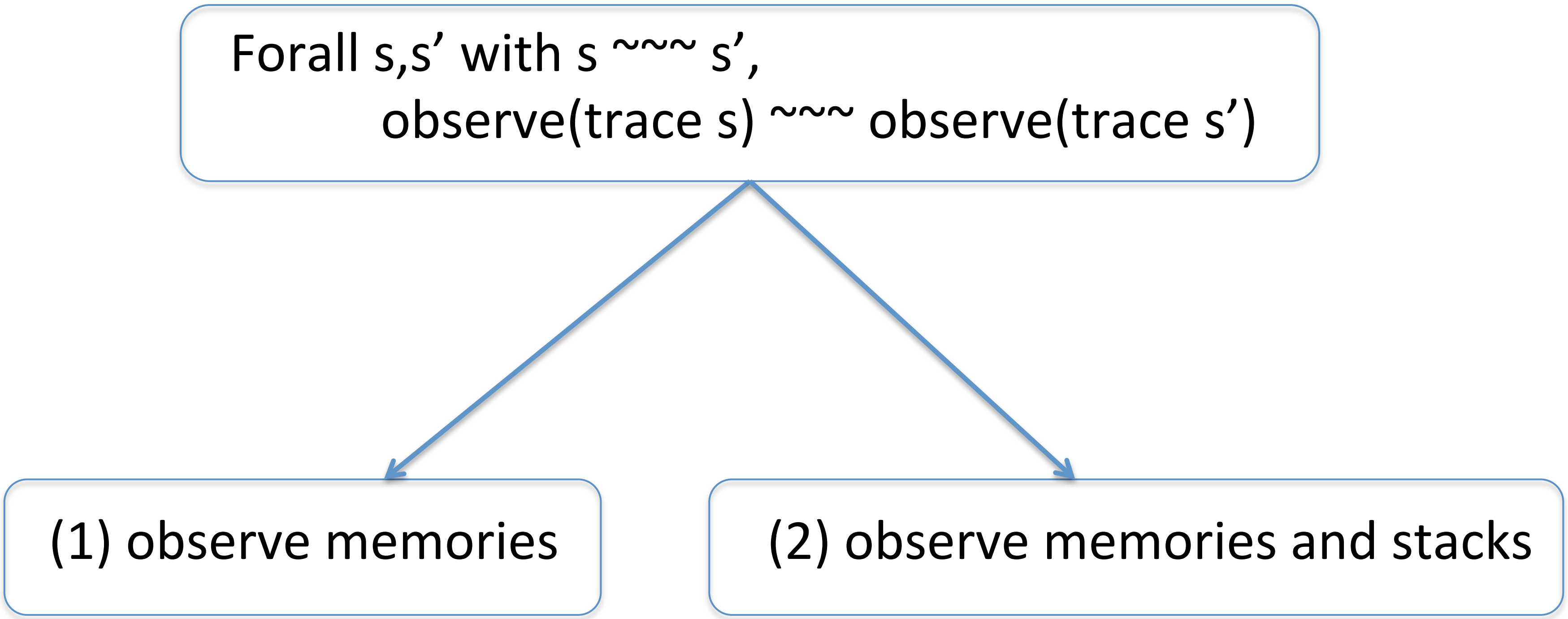
1@L      M=[0@L]          S=[{0@H/1@H},1@L]
2@L      M=[0@L]          S=[{1@L/2@L}]
3@L      M=[0@L]          S=[0@L,{1@L/2@L}]
4@L      M=[{1@L/2@L}]    S=[]
```

Notice:

- The bug in Add makes the *stacks* different at step 2!
- The need for a Store to make the bug visible *makes detection harder*

Idea:

- observe whole machine state (*stack and memory),* not just memory



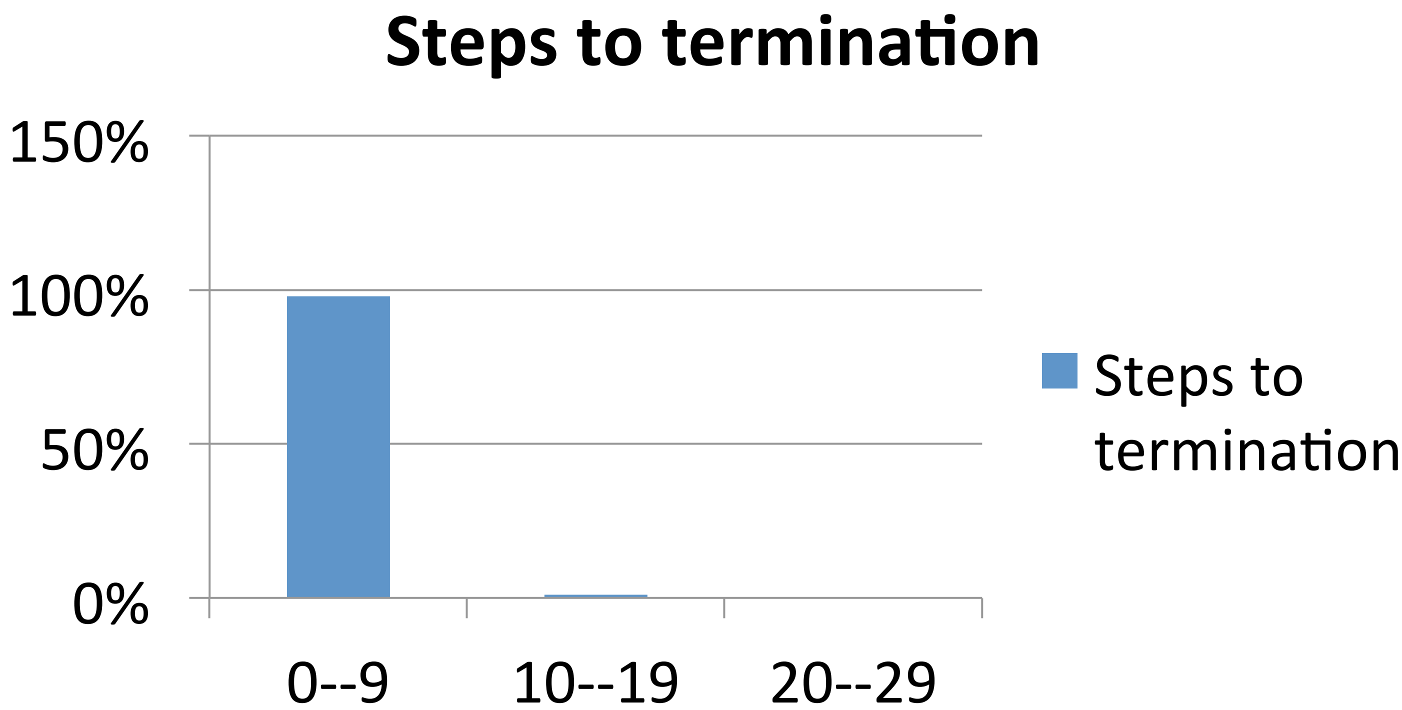
- What we really want

- Implies (1)
- Fails faster
- Expected to hold for “reasonable” machines

How many tests are needed?

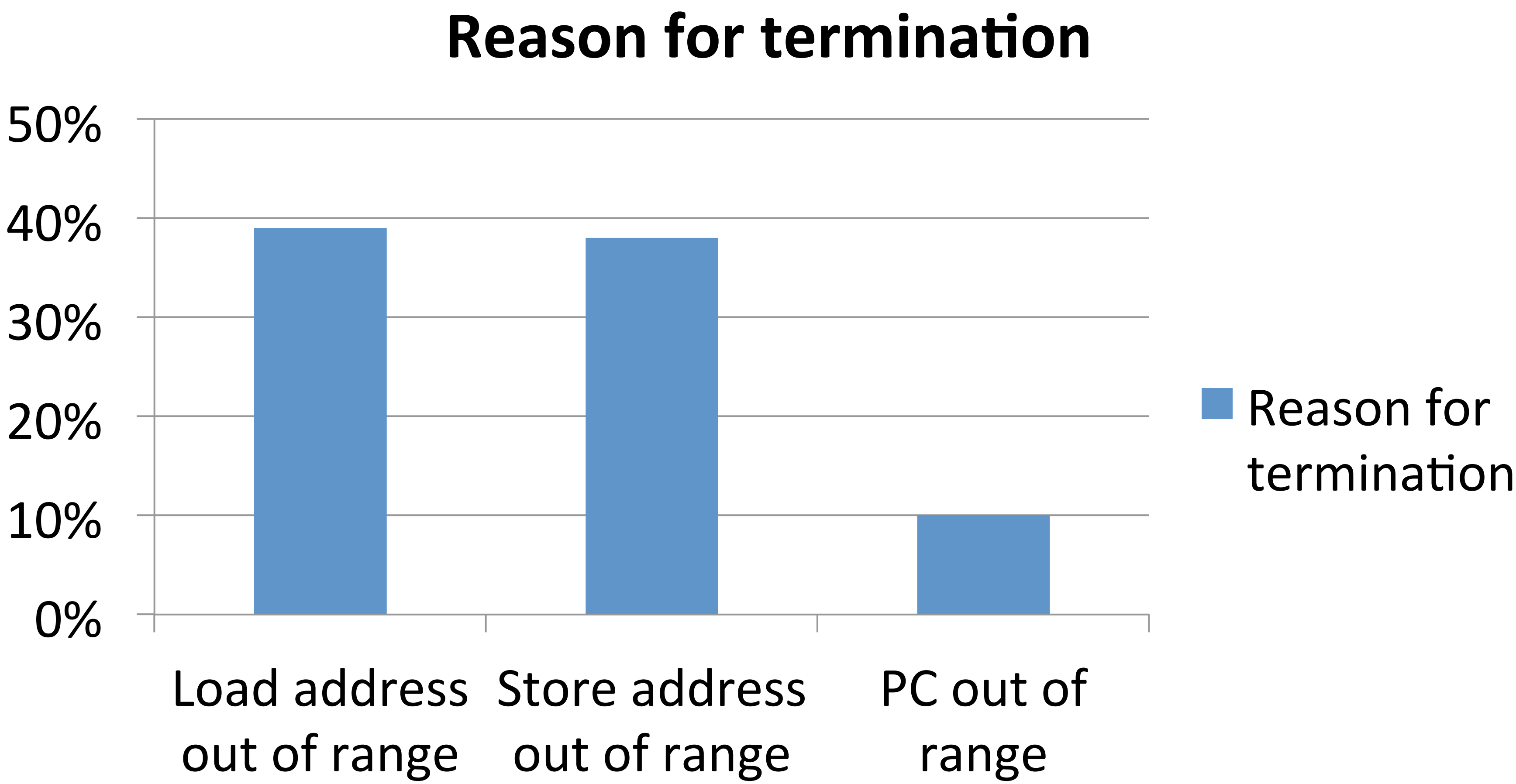
Bug	Information leak through memory	Information leak through stack or memory		
Add fails to taint	506	11		
Load fails to taint	50582	1904		
Store fails to taint	42855	52833		

How long do programs run?



- 98% of executions are <10 instructions

Why do executions terminate?



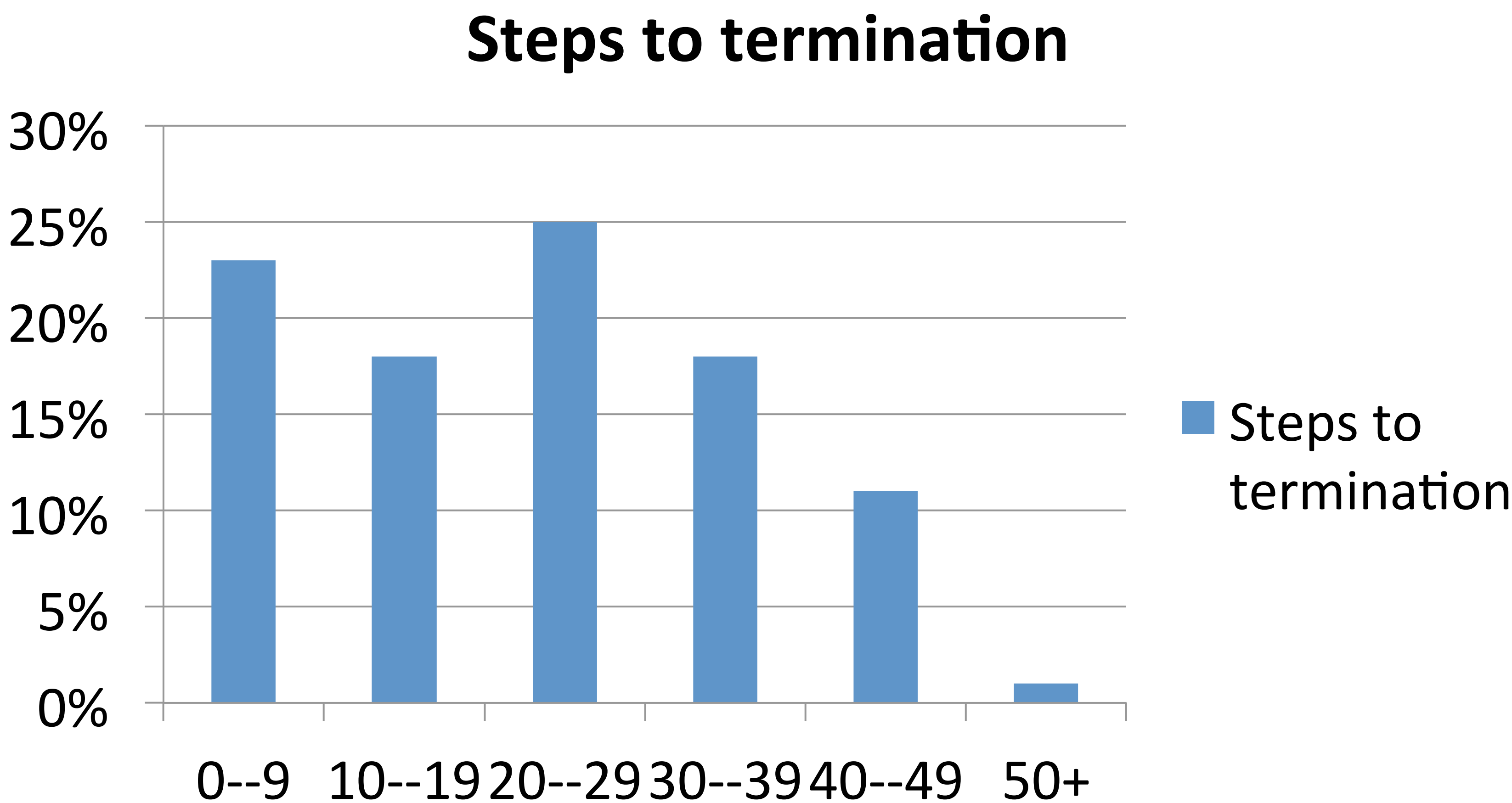
Smart program generation

- Track machine states as instruction sequences are generated
  - Don't generate instructions that fail in current state
    - e.g., don't generate Add when stack is empty

- Generate "sensible instruction pairs", as well as random instructions
    - Push *valid addr*; Load
    - Push *valid addr*; Store

  - Often generate *low* valid addresses (0, 1, 2)
    - so we reuse locations often

How long do programs run now?



How many tests are needed?

Bug	Information leak through memory	Information leak through stack or memory	Smart program generation, leak through memory	
Add fails to taint	506	11	26	
Load fails to taint	50582	1904	1242	
Store fails to taint	42855	52833	3383	

How many tests are needed?

Bug	Information leak through memory	Information leak through stack or memory	Smart program generation, leak through memory	Smart programs, leak through stack or memory
Add fails to taint	506	11	26	6
Load fails to taint	50582	1904	1242	179
Store fails to taint	42855	52833	3383	3031

Bugs squashed!

Instruction	Stack before	Stack after	Memory
Push n@l	stk	n@l : stk	
Add	a@l : b@l' : stk	(a+b)@**$(l \sqcup l')$** : stk	
Load	a@l : stk	mem[a]**$\sqcup l$** : stk	
Store	b@l : a@l' : stk	stk	mem[b] := a@**$(l \sqcup l')$**

What do counterexamples look like?

Program=[Push 4@L,Store,Push 0@L,Load,Push 5@L,Load,Store,Push -1@L,Push 6@L,Load,Push 5@L,Store,Push 1@L,Push 0@L,Store,Push -3@L,Add,Push 10@L,Store,Load,{Push 6@H/Push -16@H},Push 3@L,Store,Push -3@L,{Push 5@H/Push 2@H},Store,{Push -2@H/Push 12@H},Push 0@L]

Memory=[25@L,19@L,{18@H/4@H},-3@L,3@L,3@L,{29@H/13@H},6@L,17@L,24@L,15@L,8@L]

Stack=[1@L, 5@L, 22@L, 7@L]

Shrinking 101

- When a test fails, QC tries to replace it by a "shrunk" test — a similar input that also fails
  - goto 1

- Candidates are generated by a function

```
shrink :: a -> [a]
```

Details of shrinking

- We are working with pairs of ~~~ states
  - shrinking must preserve this invariant

```
data Variation a = Variation a a

class Observable a where
  …
  shrinkV :: Variation a -> [Variation a]
```

- Now define shrinkV for each kind of Observable…
  - Standard definitions for Int, lists, etc.
  - Domain-specific: Shrink H to L

Before:

Program=[Push 4@L,Store,Push 0@L,Load,Push 5@L,Load,Store,Push -1@L,Push 6@L,Load,Push 5@L,Store,Push 1@L,Push 0@L,Store,Push -3@L,Add,Push 10@L,Store,Load,{Push 6@H/Push -16@H},Push 3@L,Store,Push -3@L,{Push 5@H/Push 2@H},Store,{Push -2@H/Push 12@H},Push 0@L]

Memory=[25@L,19@L,{18@H/4@H},-3@L,3@L,3@L,{29@H/13@H},6@L,17@L,24@L,15@L,8@L]

Stack=[1@L, 5@L, 22@L, 7@L]

After:

Program=[Push 0@L,Store,Push 0@L,Load,Push 0@L,Load,Store,Push 0@L,Push 0@L,Load,Push 0@L,Store,Push 0@L,Push 0@L,Store,Push 0@L,Add,Push 0@L,Store,Load,Push 0@H,Push 3@L,Store,Push 0@L,{Push 3@H/Push 2@H},Store]

Memory=[0@L,0@L,0@H,0@L]

Stack=[0@L, 0@L]

Idea

- Try shrinking instructions to Noop

Before:

Program=[Push 0@L,Store,Push 0@L,Load,Push 0@L,Load,Store,Push 0@L,Push 0@L,Load,Push 0@L,Store,Push 0@L,Push 0@L,Store,Push 0@L,Add,Push 0@L,Store,Load,Push 0@H,Push 3@L,Store,Push 0@L,{Push 3@H/Push 2@H},Store]

Memory=[0@L,0@L,0@H,0@L]

Stack=[0@L, 0@L]

After:

Program=[Noop,Noop,Noop,Noop,Noop,Noop,Noop,Noop,Noop,Noop,Noop,Noop,Noop,Noop,Noop,Noop,Noop,Noop,Noop,Noop,Push 0@H,Push 3@L,Store,Noop,{Push 3@H/Push 2@H},Store]

Memory=[0@L,0@L,0@H,0@L]

Stack=[0@L]

Idea

- Try deleting Noop instructions

Before:

Program=[Noop,Noop,Noop,Noop,Noop,Noop,Noop,Noop,Noop,Noop,Noop,Noop,Noop,Noop,Noop,Noop,Noop,Noop,Noop,Noop,Push 0@H,Push 3@L,Store,Noop,{Push 3@H/Push 2@H},Store]

Memory=[0@L,0@L,0@H,0@L]

Stack=[0@L]

After:

[Push 0@H,Push 3@L,Store,{Push 3@H/Push 2@H},Store]

Memory=[0@L,0@L,0@H,0@L]

Stack=[0@L]

Another run of QC yields:

Program=[{Push 1@H/Push 0@H},Store]

Memory=[0@H,0@H]

Stack=[0@L]

Going further…

- Jumps
  - Complicates smart generation and shrinking
  - Raises possibility of branching on secrets

- Call/return
  - Much more interesting design issues
    - Not easy to achieve noninterference!

Surprises

- Not all bugs were planted :-)

- Subtleties in definition of noninterference
  1. Combining “private labels” with pointers doesn’t work
     - Should not permit 1@L ~~~ 2@H
  2. Data and return addresses on the stack must not be conflated (even when both are labeled high)

Going even further

- Ultimate goal:
  - Use QC to find bugs in implementation of SAFE operating system

Any (more) questions?