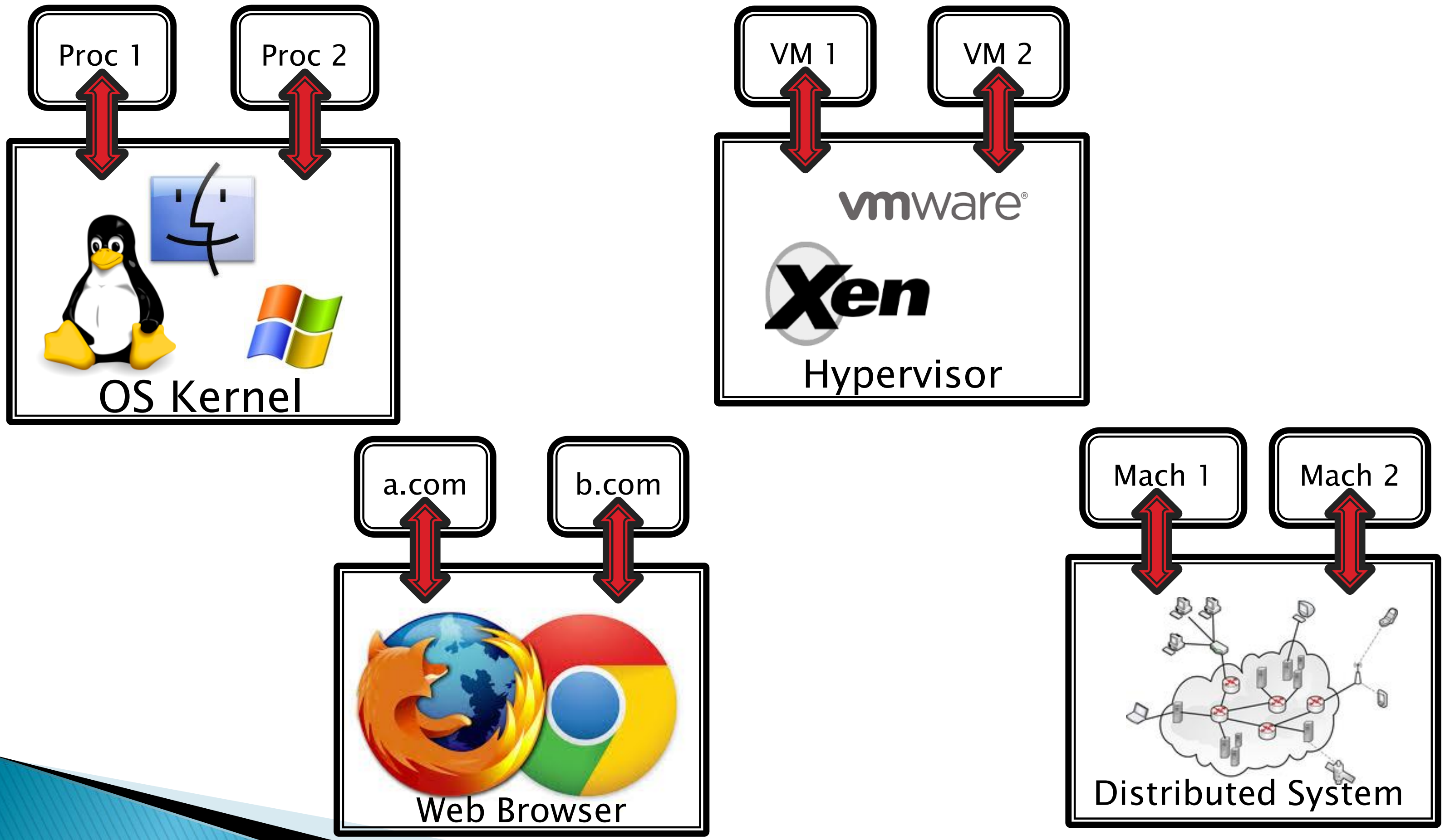
End-to-End Verification of Information-Flow Security for C and Assembly Programs

David Costanzo, Zhong Shao, Ronghui Gu
Yale University

PLDI 2016
June 17, 2016

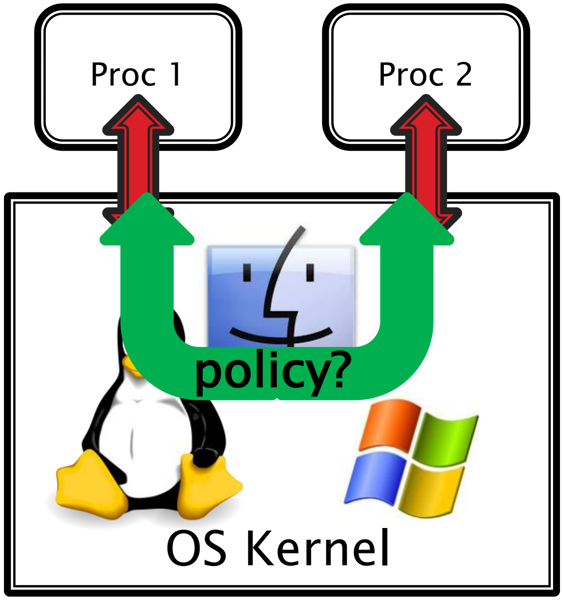
Information-Flow Security

Goal: formally prove an end-to-end information-flow policy that applies to the low-level code of these systems



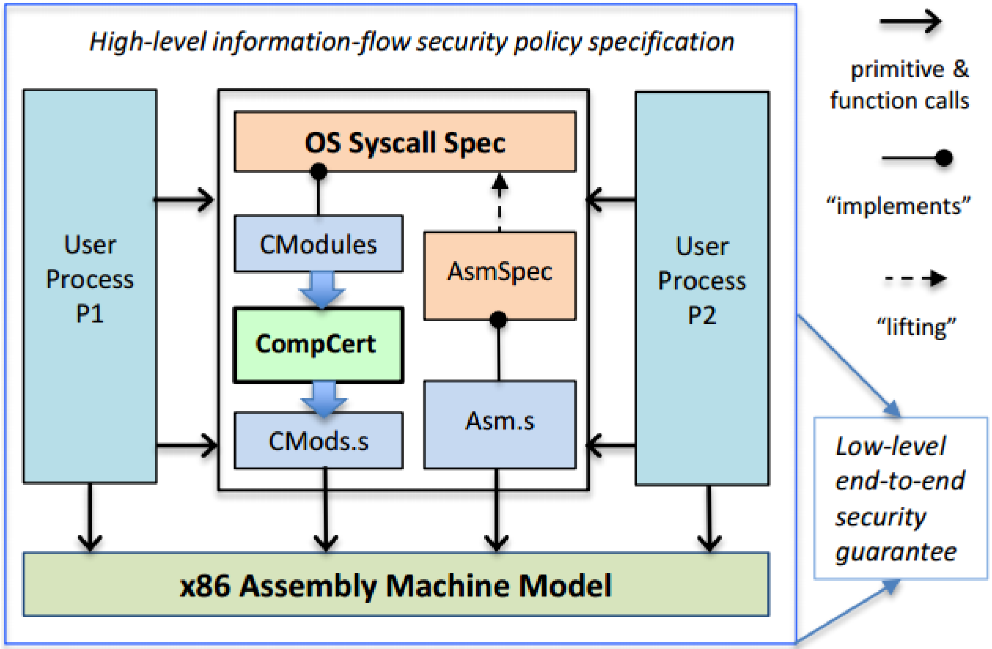
Challenges

- How to specify the information flow policy?
  - ideally, specify at high level of abstraction
  - allow for some well-specified flows (e.g., declassification)



Challenges

- Most systems are written in both C and assembly
  - must deal with low-level assembly code
  - must deal with compilation
    - even *verified* compilation may not preserve security



Challenges

- How to prove security on low-level code?
  - Security type systems (e.g., JIF) don't work well for weakly-typed languages like C and assembly
  - How do we deal with declassification?
  - Systems may have "internal leaks" hidden from clients

- How to prove security for all components in a unified way that allows us to link everything together into a system-wide guarantee?

Contribution 1

New methodology to specify, prove, and propagate IFC policies with a single unifying mechanism: the observation function

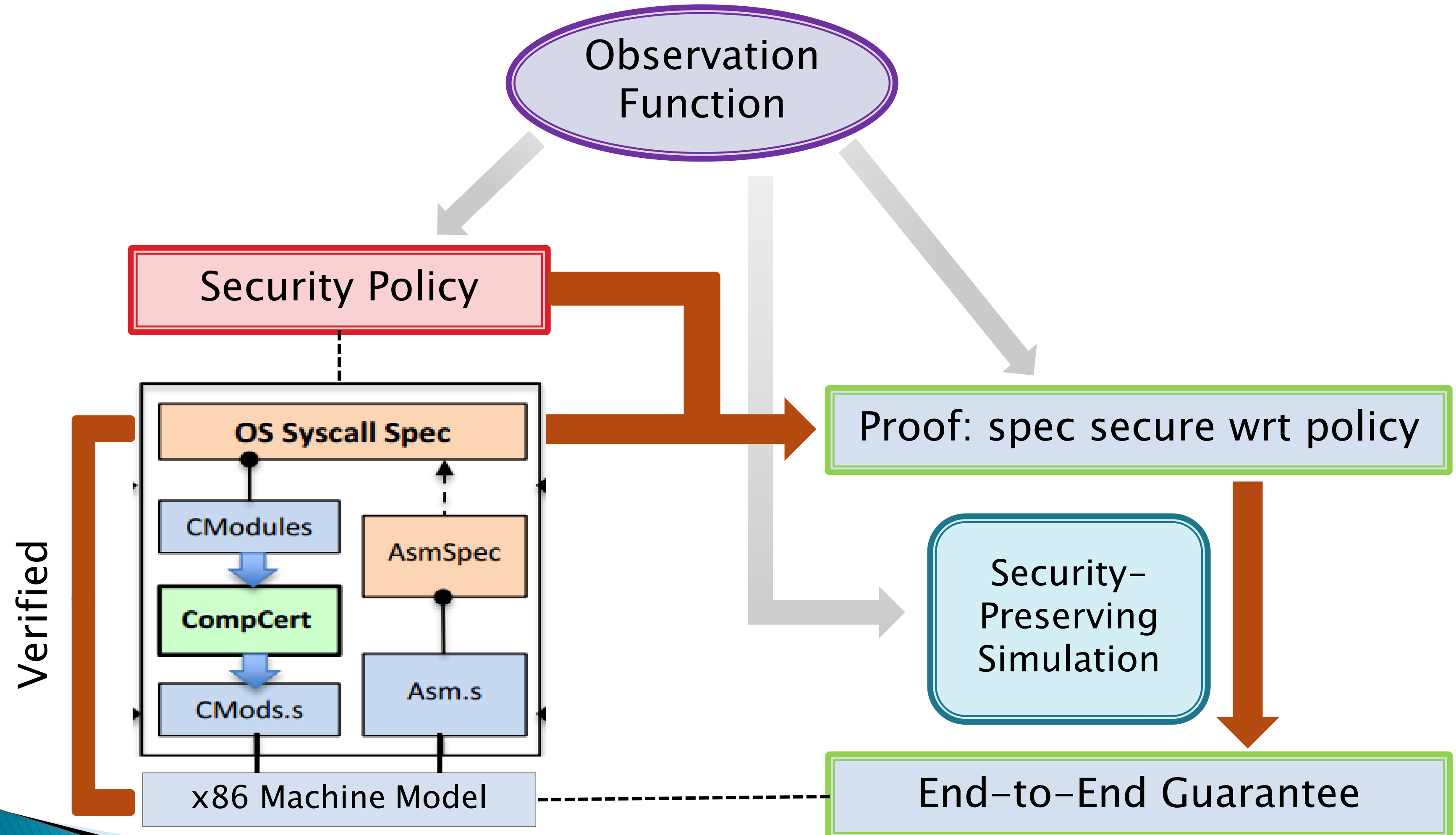
- specify – expressive generalization of classical noninterference
- prove – general proof method that subsumes both security label proofs and information hiding proofs
- propagate – security-preserving simulations

Contribution 2

Application to a real OS kernel (CertiKOS [POPL15])

- First fully-verified secure kernel involving C and assembly, including compilation
- Verification done entirely within Coq
- Fixed multiple bugs (security leaks)
- **Policy**: user processes running over CertiKOS cannot influence each other in any way (IPC disabled)

Our Solution

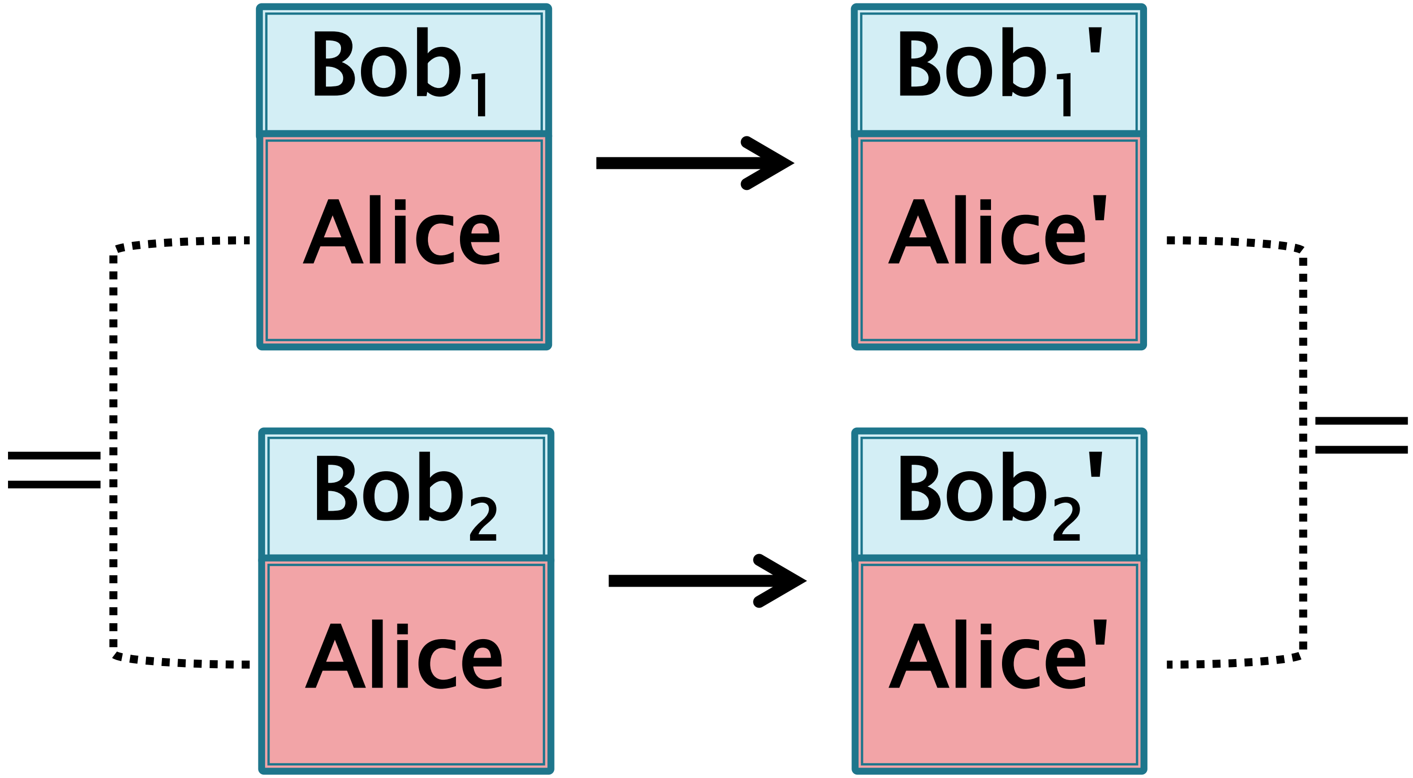


Rest of Talk

1. Specifying security
2. Proving security (example)
3. Propagating security across simulations
4. Experience with CertiKOS security proof

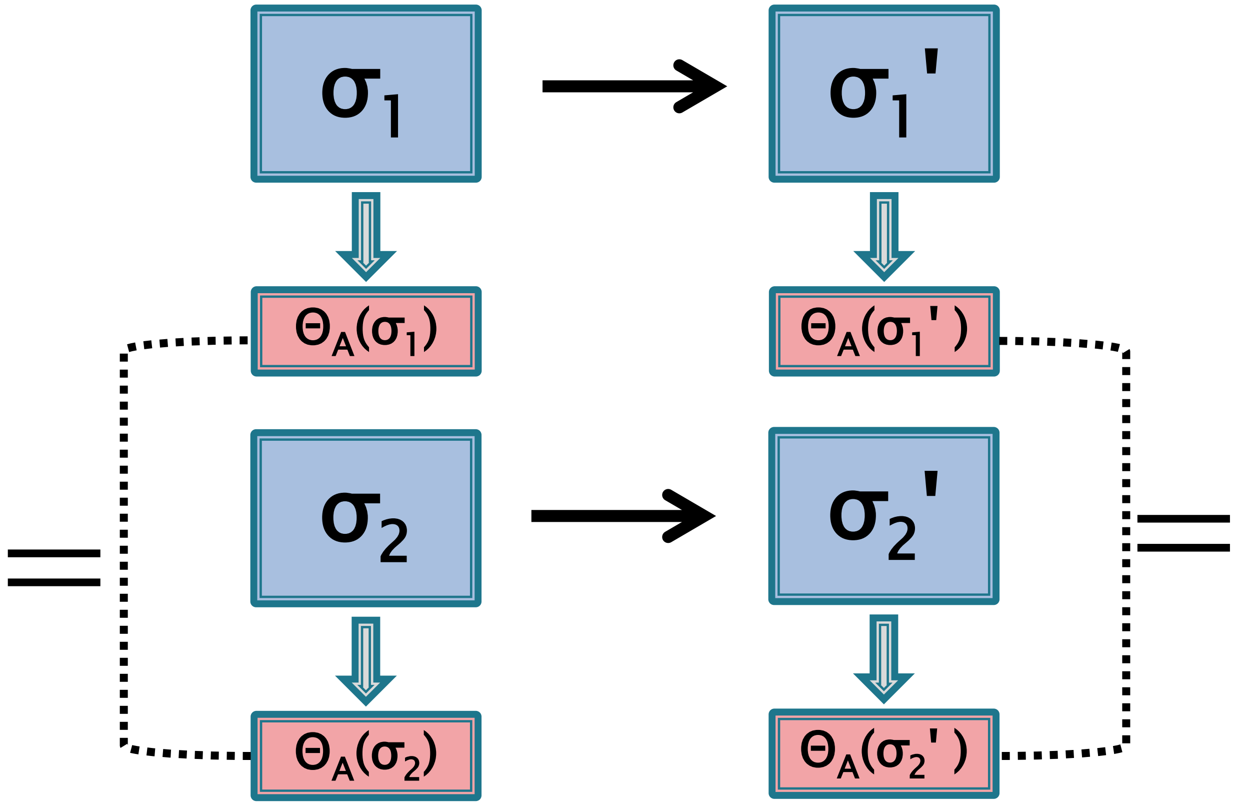
Pure Noninterference

"Alice's behavior is influenced only by her own data."



Generalized Noninterference

"Alice's behavior is influenced only by her own observation."



Observation Function

Θ : principal → program state → observation
(can be any type)

S : program state → program state → prop

"spec S is secure for principal p"

$$\forall\, \sigma_1, \sigma_2, \sigma'_1, \sigma'_2\,.$$

$$\Theta_p(\sigma_1) = \Theta_p(\sigma_2) \;\wedge\; S(\sigma_1, \sigma'_1) \;\wedge\; S(\sigma_2, \sigma'_2)$$

$$\Longrightarrow$$

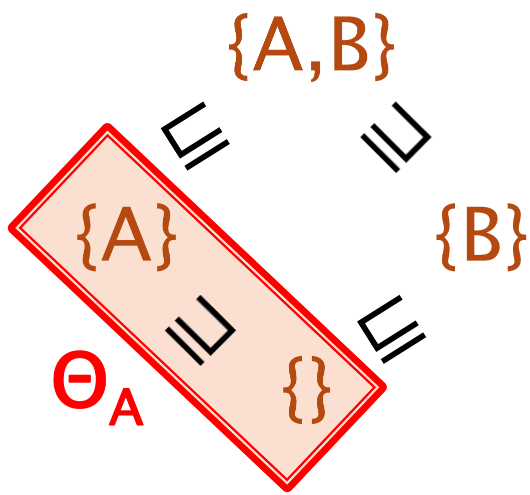
$$\Theta_p(\sigma'_1) = \Theta_p(\sigma'_2)$$

Example Observation Functions

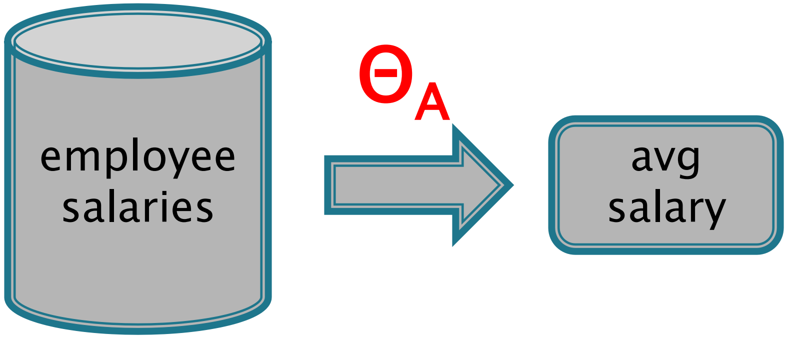
w	(5, {A})
x	(17, {A,B})
y	(42, {B})
z	(13, {})

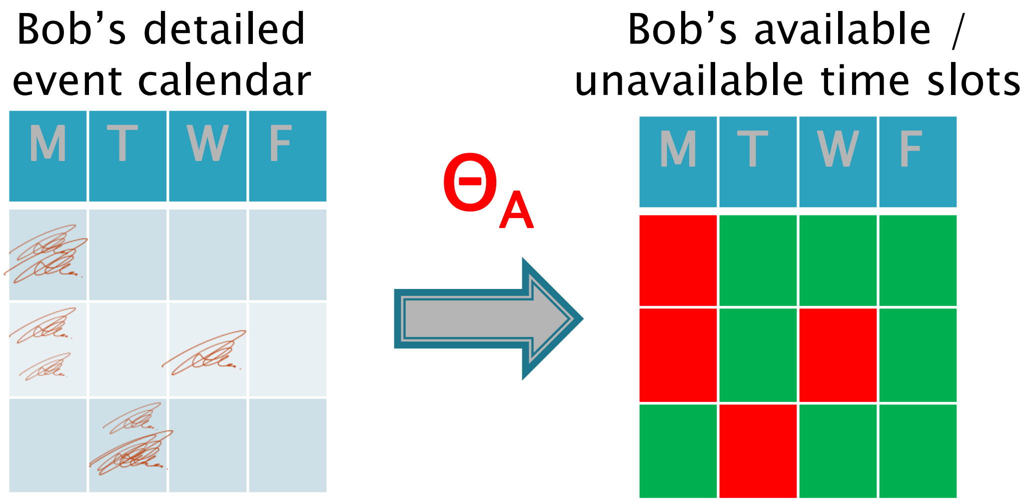
$\xrightarrow{\Theta_A}$

w	(5, {A})
x	(?, {A,B})
y	(?, {B})
z	(13, {})



Example Observation Functions



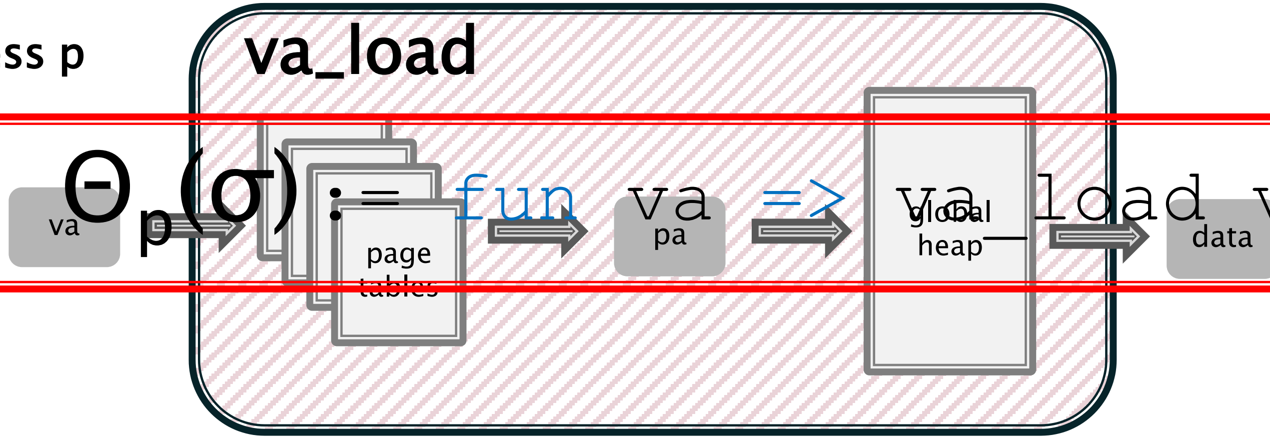


Rest of Talk

1. Specifying security
2. Proving security (example)
3. Propagating security across simulations
4. Experience with CertiKOS security proof

Virtual Address Translation

Process p



$\Theta_p(\sigma)$:= fun va => va_load va σ

```
Definition va_load va σ rs rd :=
  match ZMap.get (PDX va) (ptpool σ) with
    PDEValid _ pte =>
      match ZMap.get (PTX va) pte with
        | PTEValid pg _ =>
           Next (rs # rd <-
             FlatMem.load (HP σ) (pg*PGSIZE + va%PGSIZE))
        | PTEUnPresent => exec_pagefault σ va rs
      end
  end.
```

Declassify?

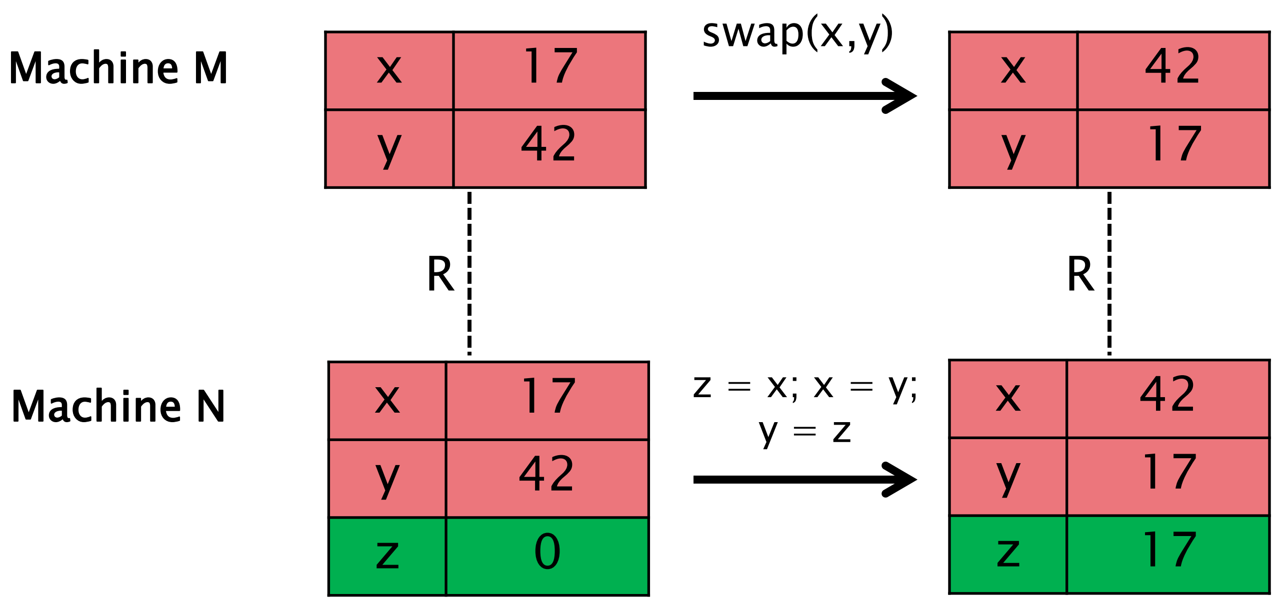
High Security

Rest of Talk

1. Specifying security
2. Proving security (examples)
3. Propagating security across simulations
4. Experience with CertiKOS security proof

Insecure Simulation

- OS and compiler refinement proofs use simulations
- Simulations may not preserve security!



$$R(\sigma_M, \sigma_N) \coloneqq (\sigma_M(x) = \sigma_N(x) \wedge \sigma_M(y) = \sigma_N(y))$$

Propagating Security

- Define an observation function for **each** machine, Θ^M and Θ^N
- Require that the simulation is security-preserving

Security-Preserving Simulation (for principal p)

$$\forall\, \sigma_1, \sigma_2, s_1, s_2 \,.$$

$$\Theta^M{}_p(\sigma_1) = \Theta^M{}_p(\sigma_2) \;\wedge\; R(\sigma_1, s_1) \;\wedge\; R(\sigma_2, s_2)$$

$$\Longrightarrow$$

$$\Theta^N{}_p(s_1) = \Theta^N{}_p(s_2)$$

- No significant changes to CompCert were needed

Rest of Talk

1. Specifying security
2. Proving security (examples)
3. Propagating security across simulations
4. Experience with CertiKOS security proof

CertiKOS Overview

- Certified functionally correct OS kernel with 32 layers
- 354 lines of assembly code, ~3000 lines of C code
  - CompCert compiles C to assembly
- Each layer has primitives that can be called atomically
- Bottom layer MBoot is the x86 machine model
- Top layer TSysCall contains 9 system calls as primitives
  - init, vmem load/store, page fault, memory quota, spawn child, yield, print

CertiKOS Observation Function

- For a process p, the observation function is:
  - registers, if p is currently executing
  - the output buffer of p
  - the **function** from p’s virtual addresses to values
  - p’s available memory remaining (quota)
  - the number of children p has spawned
  - the saved register context of p
  - the spawned status and currently-executing status of p

CertiKOS Security Property

TSysCall → Θ^S_p = (as described)

R

MBoot → Θ^I_p = p's current output buffer

B^I_p = p's "final" output buffer (whole-execution behavior)

Generalized Noninterference:

$$\forall \sigma_1, \sigma_2, \sigma_1', \sigma_2' \,.$$
$$\Theta_p^S(\sigma_1) = \Theta_p^S(\sigma_2) \wedge (\sigma_1, \sigma_1') \in S \wedge (\sigma_2, \sigma_2') \in S$$
$$\Rightarrow \Theta_p^S(\sigma_1') = \Theta_p^S(\sigma_2')$$

End-to-End Security:

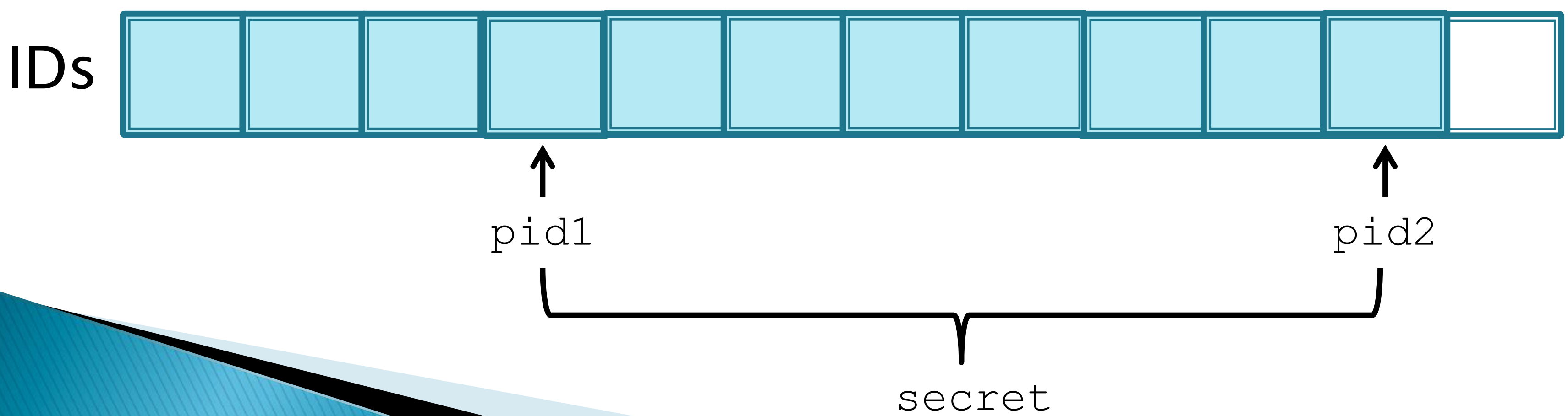
$$\forall \sigma_1, \sigma_2, s_1, s_2 \,.$$
$$\Theta_p^S(\sigma_1) = \Theta_p^S(\sigma_2) \wedge (\sigma_1, s_1) \in R \wedge (\sigma_2, s_2) \in R$$
$$\Rightarrow B_p^I(s_1) = B_p^I(s_2)$$

CertiKOS Security Leak

```
function alice {
  int pid1 = proc_spawn();
  yield();
  int pid2 = proc_spawn();
  print(pid2 - pid1 + 1);
}
```

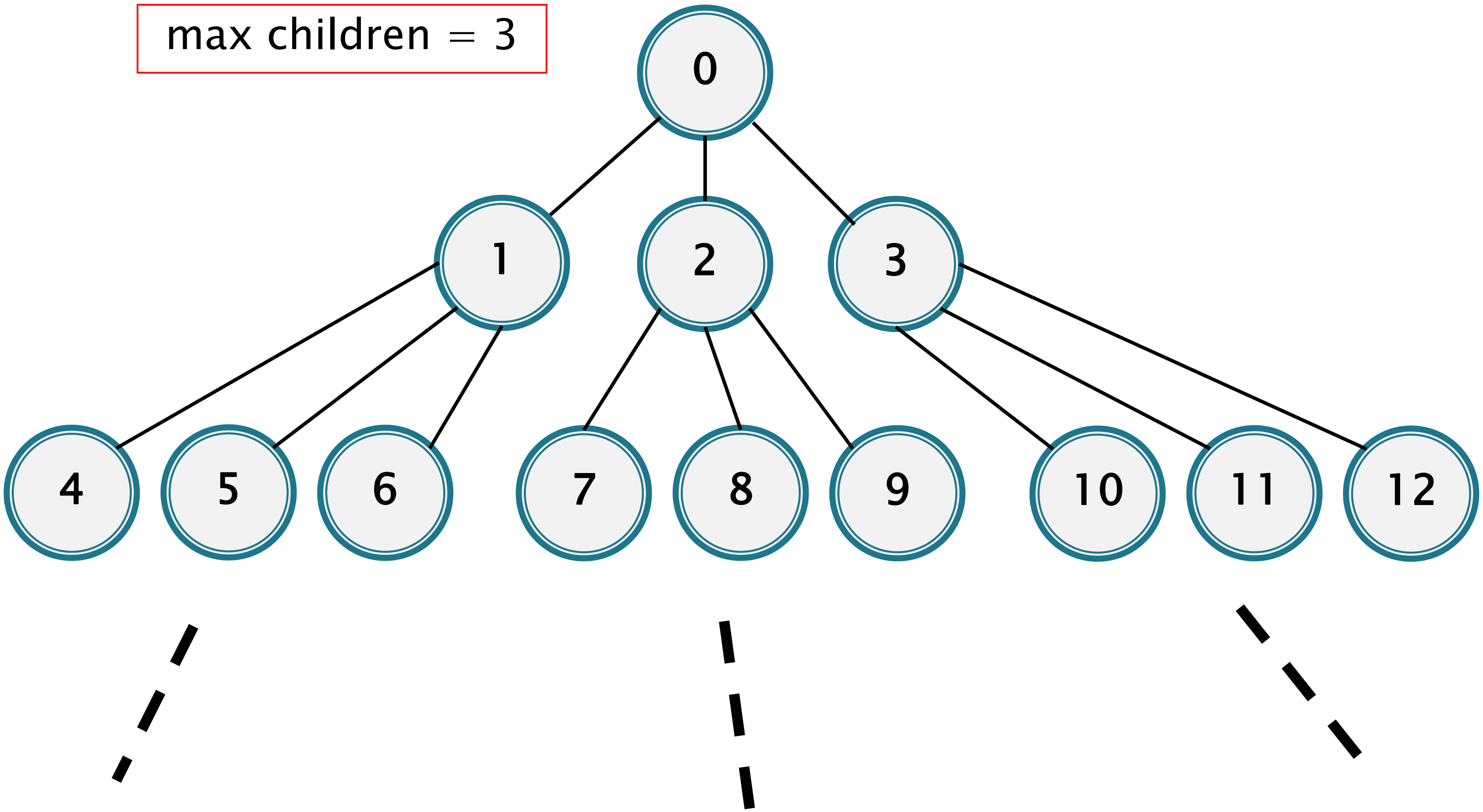
||

```
function bob {
  int secret = 42;
  for i = 0 to secret {
    proc_spawn();
  }
  yield();
}
```



Solution to Leak

max children = 3



Conclusion

- New methodology using observation function to specify, prove, and propagate IFC policies
  - applicable to all kinds of real-world systems!
- Verification of secure kernel done fully within Coq
  - machine-checked proofs!
- **Future Work**: virtualized time (already done), more realistic x86 model, preemption, concurrency

Thank You!

CertiKOS info – http://flint.cs.yale.edu/certikos/
PLDI certified artifact – ask me for link