CS428/528 Lecture 13: Information-Flow Security for mCertiKOS

Zhong Shao

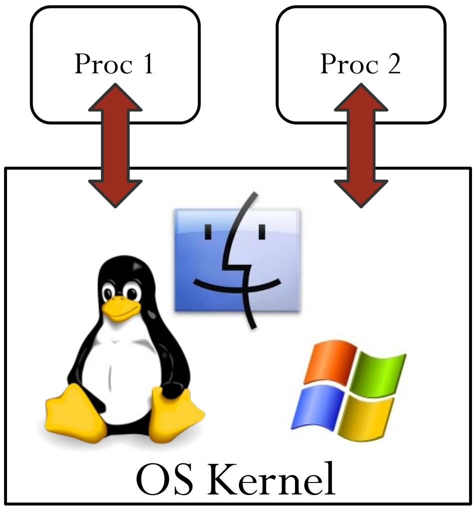
Yale University

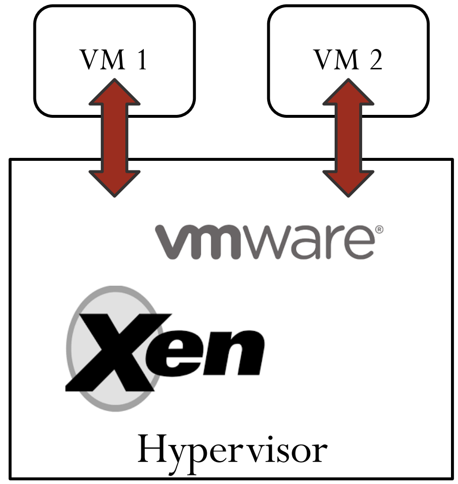
February 27, 2024

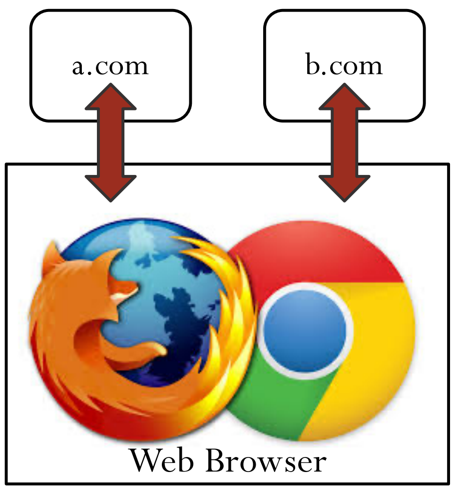
Based on the PLDI 2016 paper by Costanzo et al.

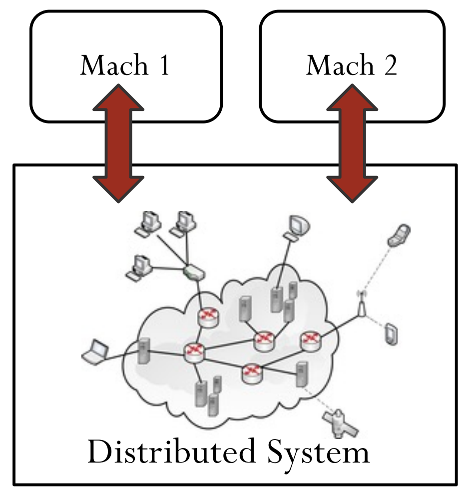
Information-Flow Security

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



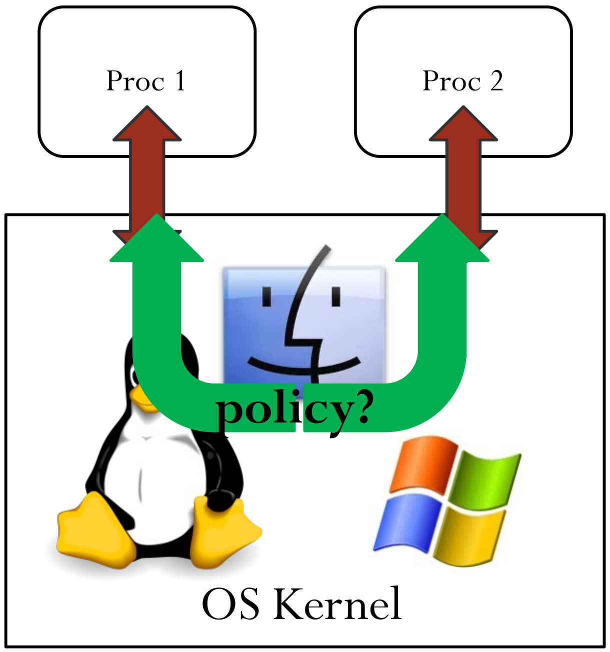






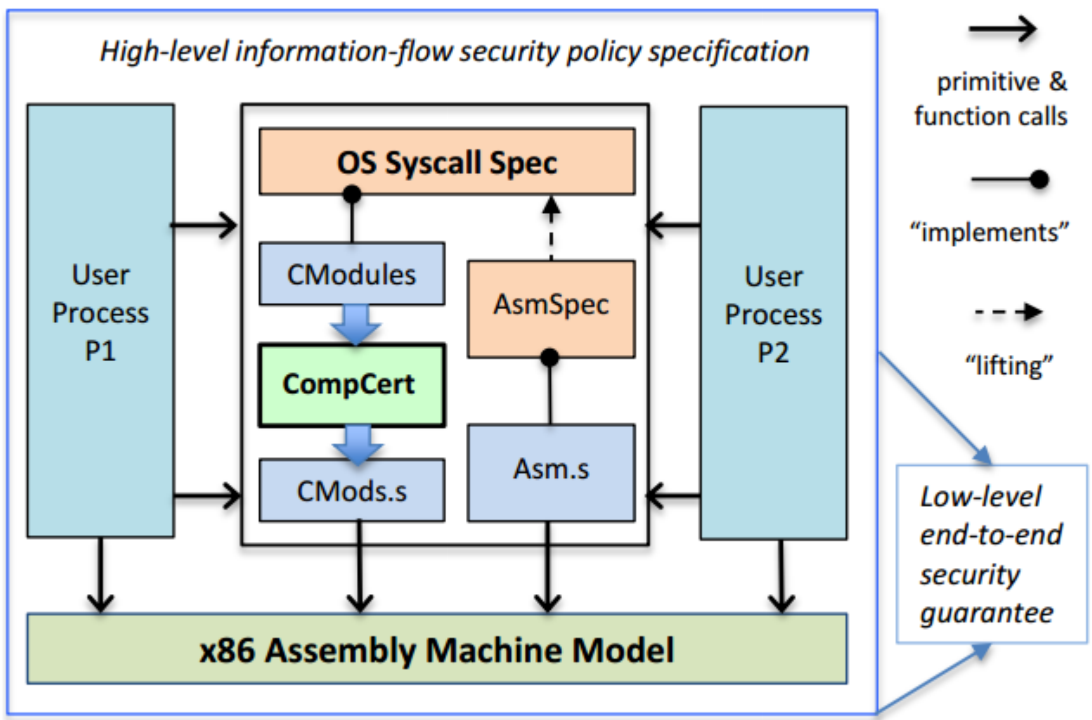
Challenges

- 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?

No existing system solves all of these challenges!

Related Work

- Practical languages with security labels: JIF [1], FlowCaml [2]
  - Typed languages only, no C or assembly
  - No formal end-to-end guarantees

[1] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. ACM Trans. Softw. Eng. Methodol., 9(4):410–442, 2000.

[2] Vincent Simonet and Inria Rocquencourt. Flow Caml in a Nutshell. Proceedings of the first APPSEM-II workshop. 2003

Related Work

- Dynamic label tracking and label checks (e.g., [1], [2])
  - Runtime exceptions can leak information
  - Declassifications are particularly problematic
  - Necessarily incomplete
    - dynamic label checks may disallow safe "internal leaks"
  - Execution overhead

[1] Thomas H. Austin and Cormac Flanagan. Efficient purely-dynamic information flow analysis. In PLAS, pages 113–124, 2009.

[2] Catalin Hritcu, Michael Greenberg, Ben Karel, Benjamin C. Pierce, and Greg Morrisett. All your ifcexception are belong to us. In IEEE Symposium on Security and Privacy, pages 3–17, 2013.

Related Work

- seL4 (NICTA) end-to-end security proof [1]
  - no assembly code verification
  - everything verified w.r.t. a C-level machine model
    - ignores many intricacies of virtual memory address translation, page fault handling, and context switching
  - no guarantee that the C compiler maintains security

[1] Toby C. Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. sel4: From general purpose to a proof of information flow enforcement. In IEEE Symposium on Security and Privacy, pages 415–429, 2013.

Contribution 1

New methodology to solve all of these challenges!

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

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

Contribution 2

Application to a real OS kernel (our group's CertiKOS [1])

- 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)

[1] Ronghui Gu, Jeremie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep specifications and certified abstraction layers. In Proc. 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), Mumbai, India, pages 595–608, 2015.

Program Logic Basics

Program C

```
i := 0;
while (i < 64) do
  x := [A+i];
  if (x = 0)
    then
      output i;
    else
      skip;
  i := i+1;
```

derive

Hoare Triple

$$\{P\}\ C\ \{Q\}$$

soundness

1. C doesn't crash when P holds
2. C always takes P states to Q states
3. C satisfies the **security policy** specified by P

Language

$E ::= x \mid n \mid E + E \mid \ldots$

$B ::= E = E \mid \mathsf{true} \mid \mathsf{false} \mid B \wedge B \mid \ldots$

$C ::= x := E \mid x := [E] \mid [E] := E \mid \text{output } E \mid \text{skip}$

$\mid C; C \mid \text{if } B \text{ then } C \text{ else } C \mid \text{while } B \text{ do } C$

Example Program

```
i := 0;

while (i < 64) do

  x := [A+i];

  if (x = 0)
    then

      output i;

    else

      skip;


  i := i+1;
```

Example Program Verification

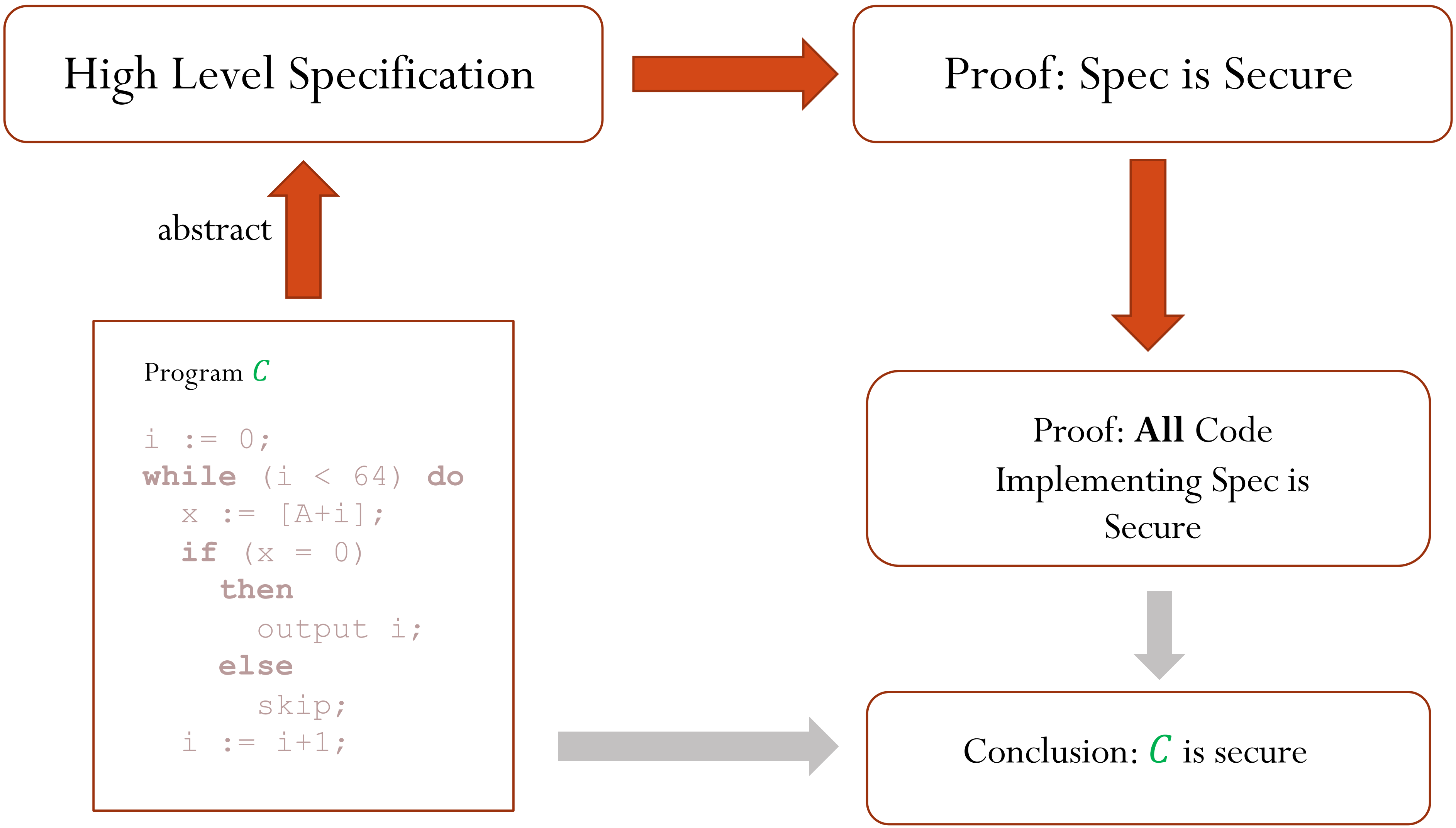
$$P = \circledast_{i \in [0,63]} A{+}i \mapsto (n_i, l_i) \wedge ((n_i = 0 \wedge l_i = Lo) \vee (n_i \neq 0 \wedge l_i = Hi))$$

```
Lo ⊢ {P}
i := 0;
Lo ⊢ {P ∧ (i ≥ 0 ∧ lbl(i) = Lo)}
while (i < 64) do
  Lo ⊢ {P ∧ (0 ≤ i < 64 ∧ lbl(i) = Lo)}
  x := [A+i];
  Lo ⊢ {P ∧ (lbl(i) = Lo ∧ ((x = 0 ∧ lbl(x) = Lo) ∨ (x ≠ 0 ∧ lbl(x) = Hi)))}
  if (x = 0)
    then
      Lo ⊢ {P ∧ (lbl(i) = Lo ∧ x = 0 ∧ lbl(x) = Lo)}
      output i;
      Lo ⊢ {P ∧ (lbl(i) = Lo)}
    else
      Hi ⊢ {P ∧ (lbl(i) = Lo ∧ x ≠ 0 ∧ lbl(x) = Hi)}
      skip;
      Hi ⊢ {P ∧ (lbl(i) = Lo)}
  Lo ⊢ {P ∧ (lbl(i) = Lo)}
  i := i+1;
  Lo ⊢ {P ∧ (lbl(i) = Lo)}
Lo ⊢ {P}
```

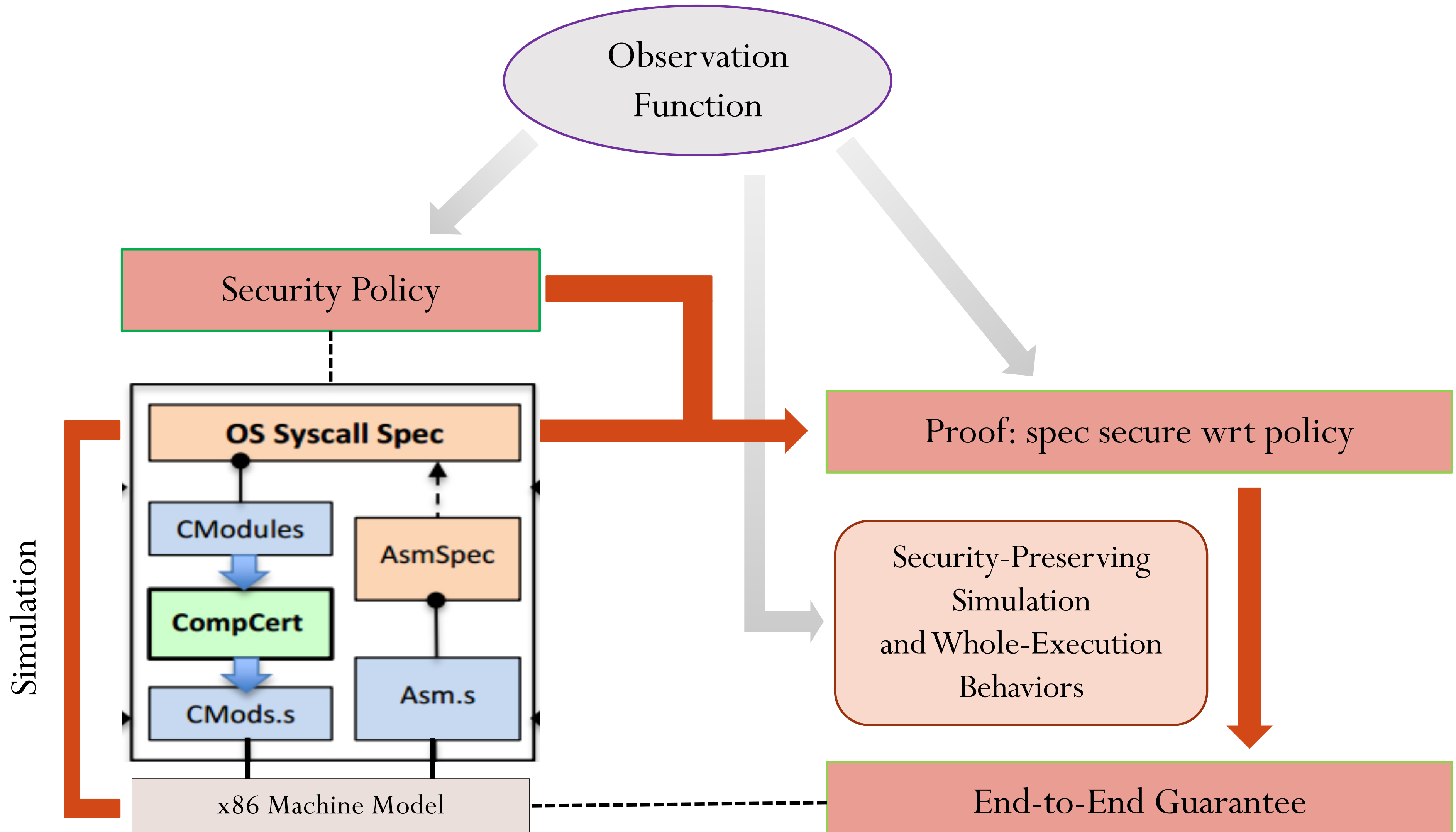
Problems with this Approach

- **Language-specific**
  - bound to C-level reasoning and control flow constructs
- **Depends on specific code details**
  - any change in the system's code would require reverification
- **Overlaps functional correctness with security concerns**
  - which aspects of P are important for safety, and which for security?
- **Incomplete**
  - some programs are secure but cannot be verified in the logic
  - informal observation: all such programs can be rewritten to become verifiable

Ideal Solution



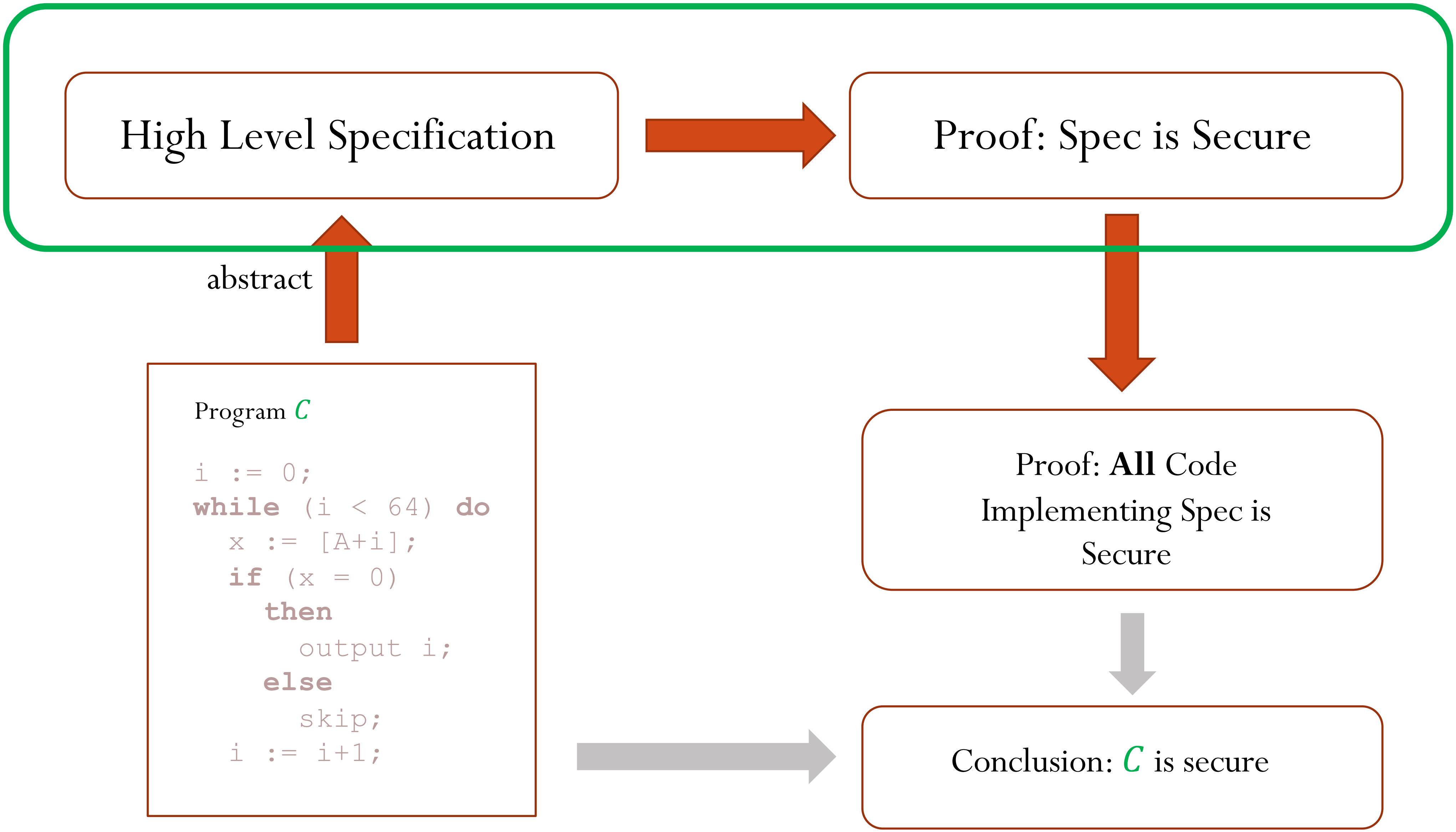
Ideal Solution – Achievable!



Rest of Talk

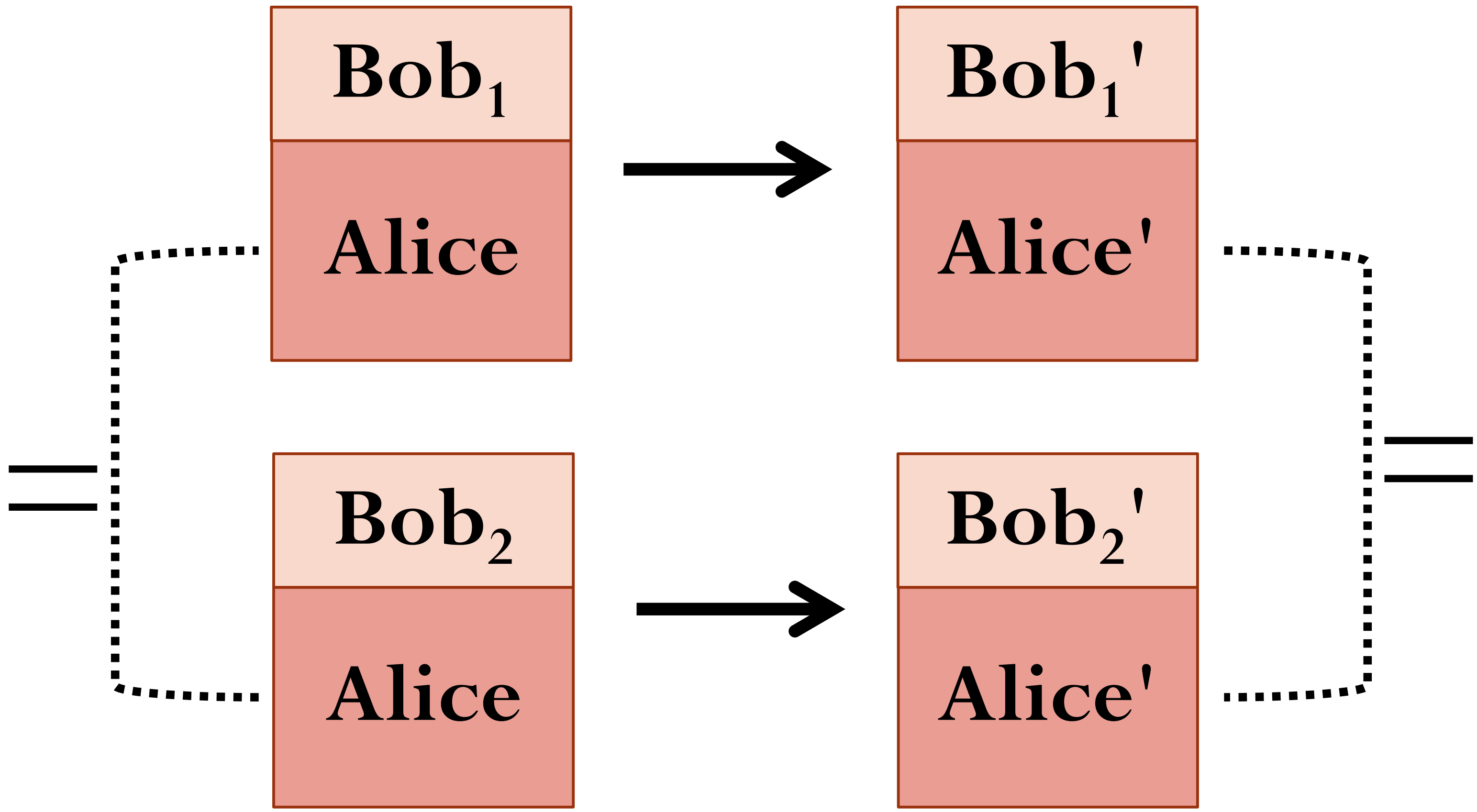
1. Specifying and proving security
2. Propagating security across simulations
3. CertiKOS security proof
4. Limitations and extensions

Ideal Solution



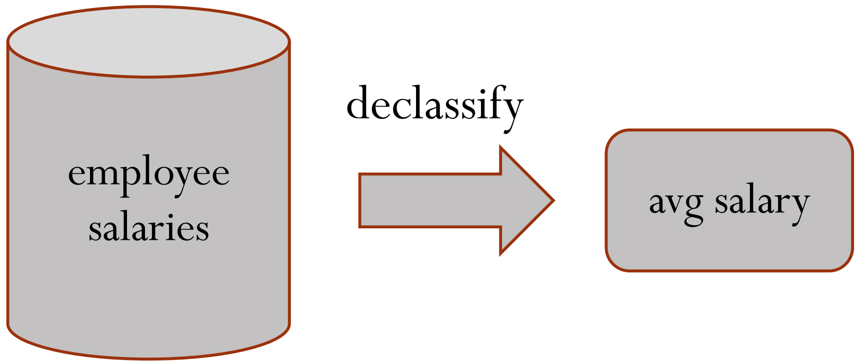
Pure Noninterference

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



Common end-to-end security property for systems using security-label reasoning.

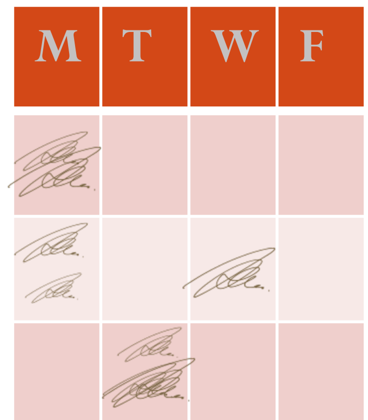
More Complex Policies



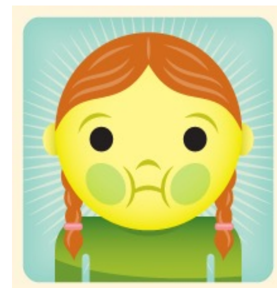

```
void printAvg() {
  int sum = 0;
  for int i = 0 to db.size-1
     sum += db[i];

  double avg = double(sum) / (db.size-1);
  print(avg);
}
```

More Complex Policies

Bob's detailed event calendar

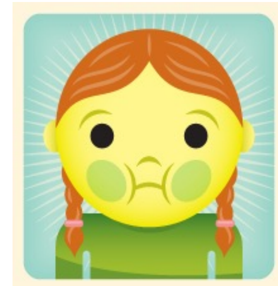


schedule meeting with Bob

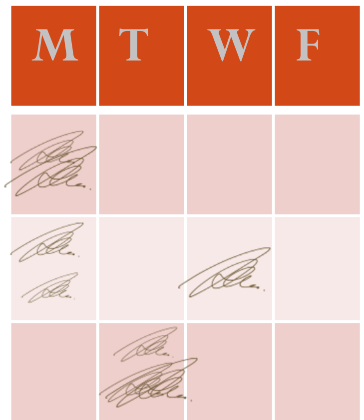


Bob says: Alice can see only whether a day is free or not free

More Complex Policies



Bob's detailed event calendar



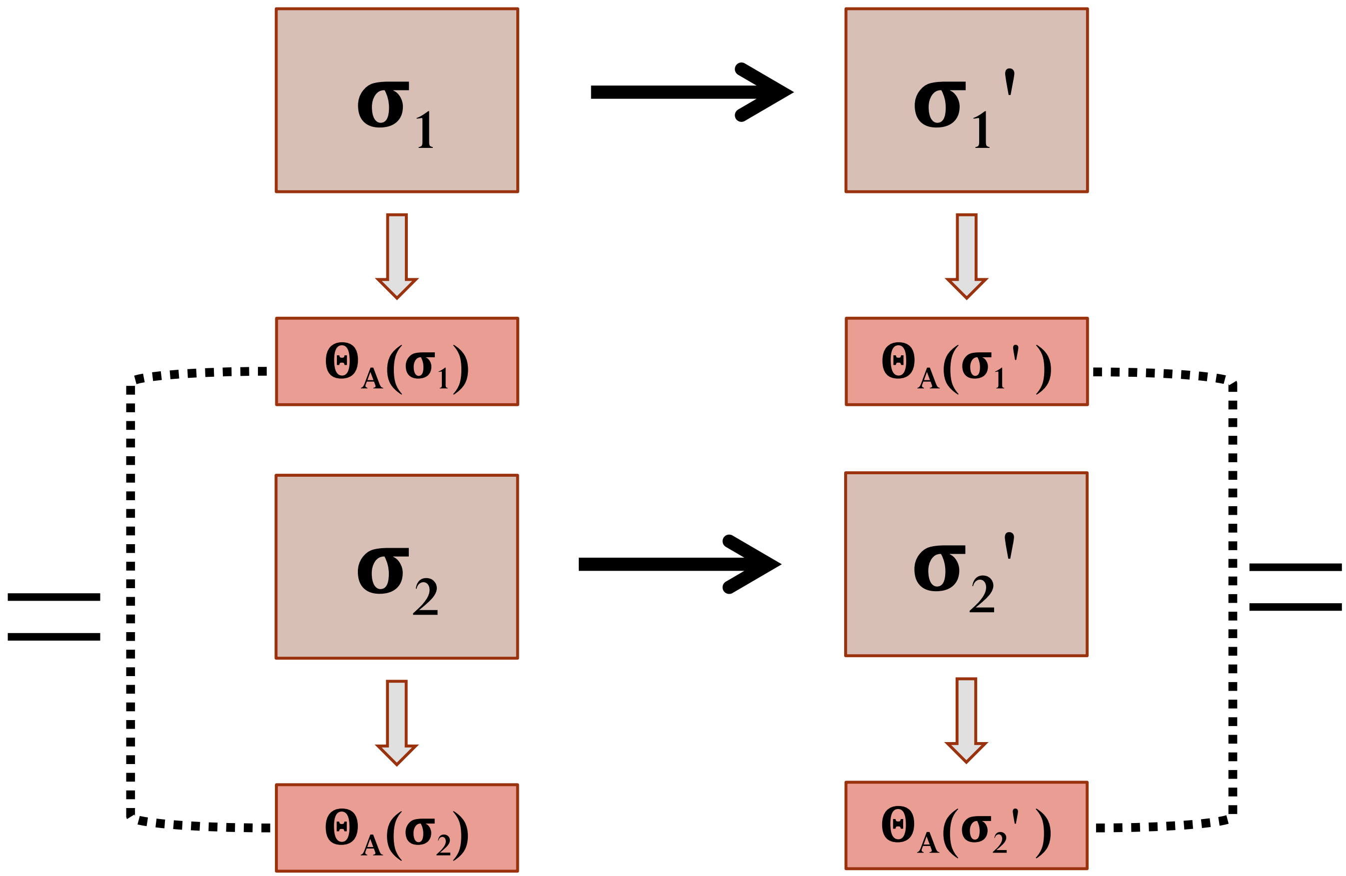
Bob says: Alice can see only whether a day is free or not free

```
void sched(event e) {
  for int i = 0 to cal.size-1 {
    int day = -1;
    if cal[i] == None {
      day = i;
      break;
    }
  }
  if day != -1
    cal[day] = Some e;
}
```

Requires conditional labels, as the security levels depend on the values themselves

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"

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

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

$$\Longrightarrow$$

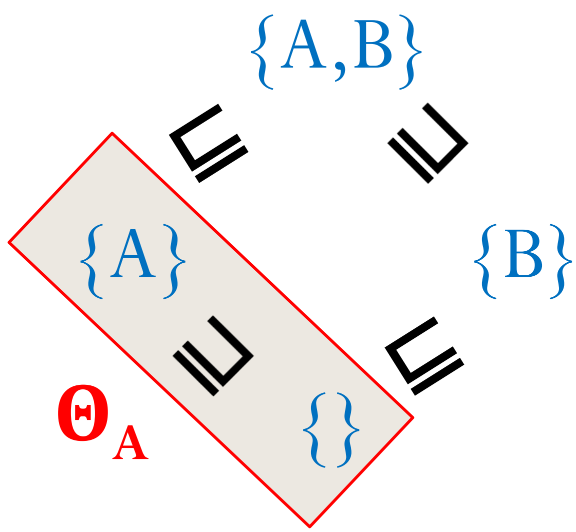
$$\Theta_p(\sigma'_1) = \Theta_p(\sigma'_2)$$

Example Observation Functions

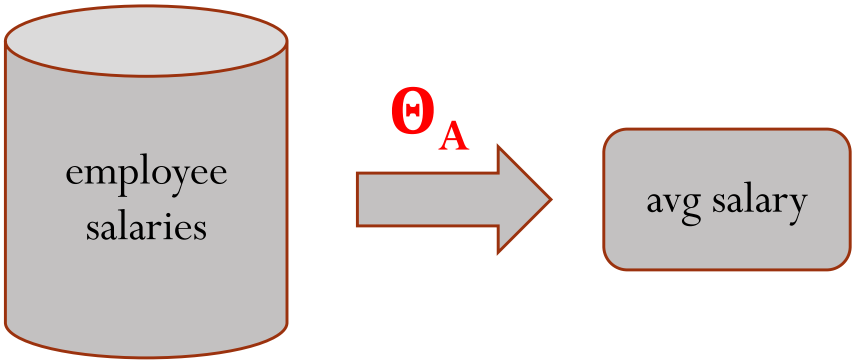
w	(5, {A})
x	(17, {A,B})
y	(42, {B})
z	(13, {})

Θ_A ⇒

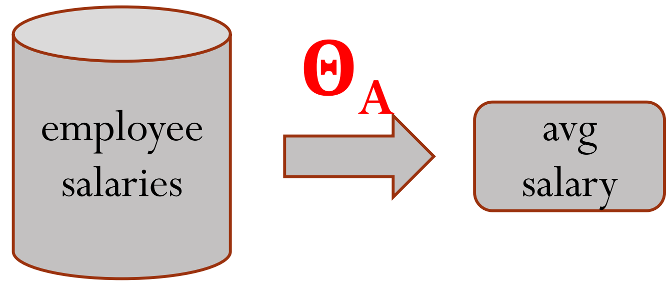
w	(5, {A})
x	(?, {A,B})
y	(?, {B})
z	(13, {})



Average Salary



Average Salary



0	5
1	17
2	42
3	13

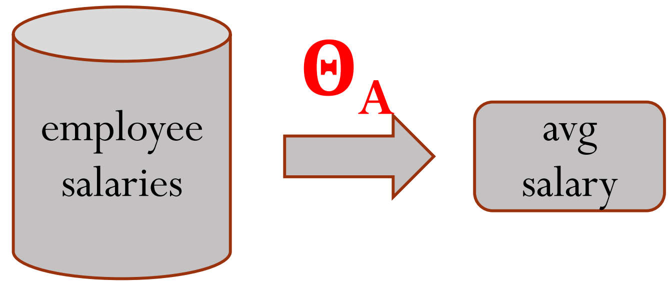
19.25

same behavior

0	35
1	8
2	22
3	12

19.25

Average Salary



$$avg(\sigma) = (\sigma(0) + \sigma(1) + \ldots + \sigma(size\text{-}1)) \;/\; (size\text{-}1)$$

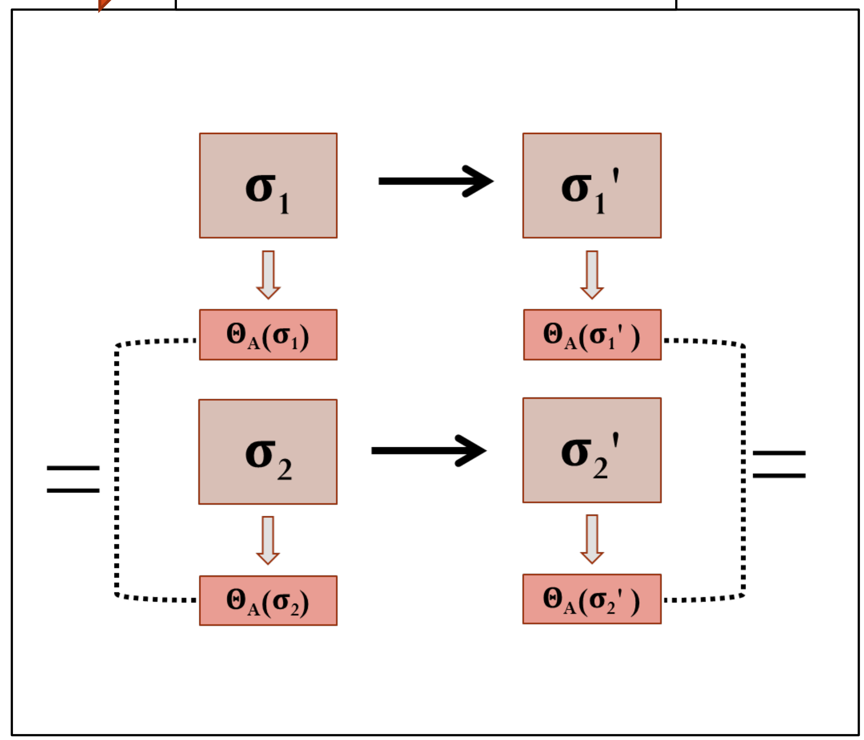
$$printAvgSpec(\sigma) = \sigma\{out \to out(\sigma) \mathbin{++} [avg(\sigma)]\}$$

$$\Theta_A(\sigma) = (avg(\sigma), out(\sigma))$$

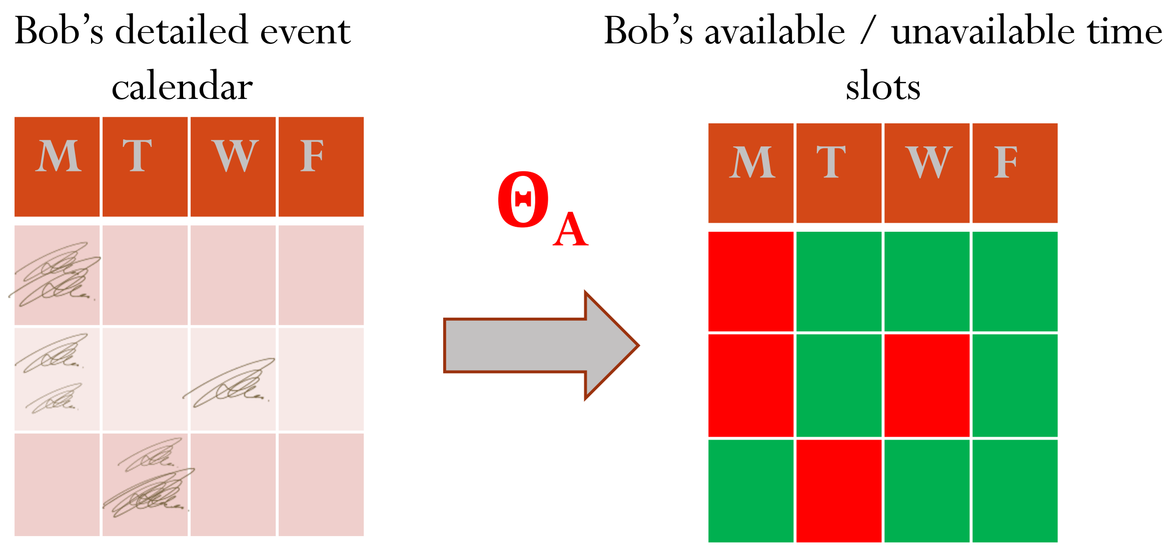
abstract

```
void printAvg() {
  int sum = 0;
  for int i = 0 to db.size-1
     sum += db[i];

  double avg = double(sum) / (db.size-1);
  print(avg);
}
```

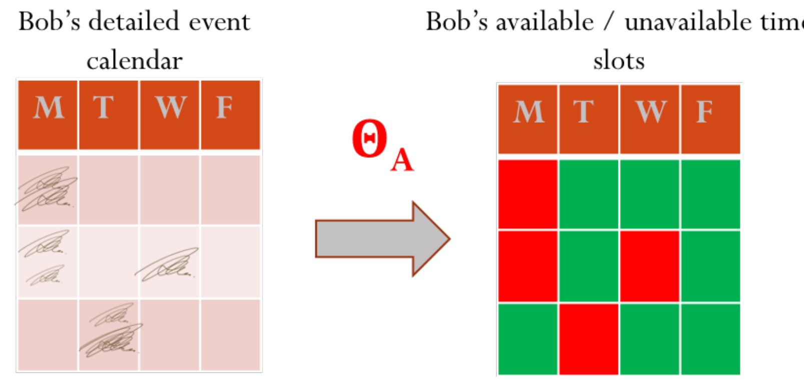
Proof: Generalized Noninterference



Event Calendar



Bob says: Alice can see only whether a day is free or not free



Event Calendar

first(σ) = Some <first available slot>, if an empty slot exists
OR None, otherwise

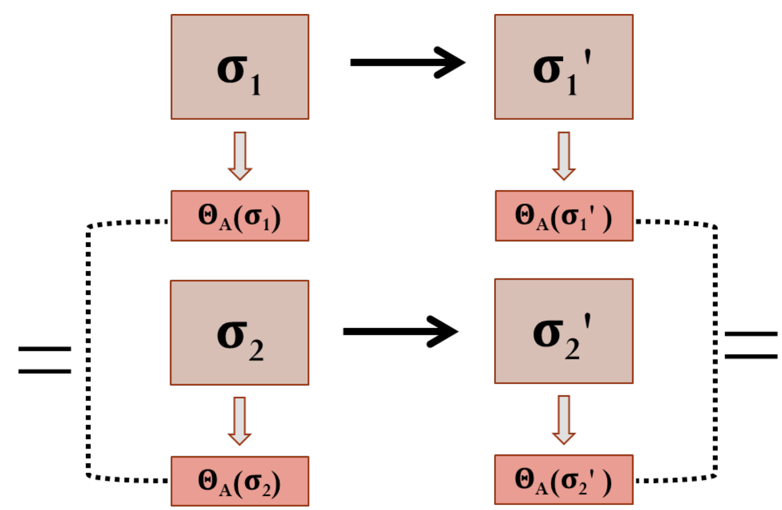
schedSpec(e, σ) = $\sigma\{f \rightarrow \text{Some } e\}$, if first($\sigma$) = Some f
OR σ, if first(σ) = None

$\Theta_A(\sigma) = \lambda\, i\ .\ \text{true, if } \sigma(i) = \text{None}$
OR false, otherwise

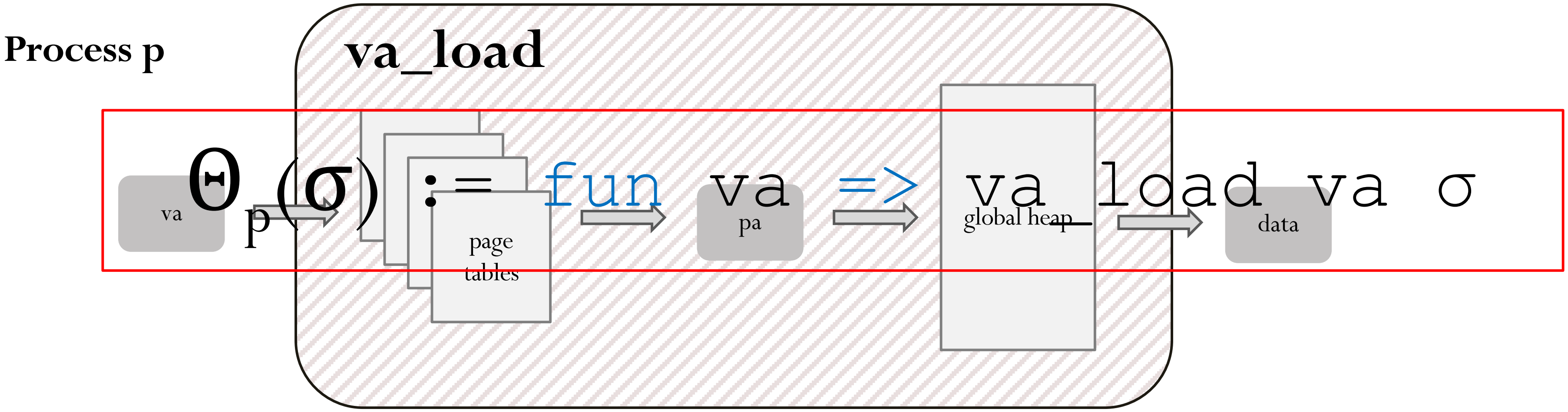
abstract

```
void sched(event e) {
  for int i = 0 to cal.size-1 {
    int day = -1;
    if cal[i] == None {
      day = i;
      break;
    }
  }
  if day != -1
    cal[day] = Some e;
}
```

Proof: Generalized Noninterference



Virtual Address Translation



$\Theta_p(\sigma)$:= 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.
```

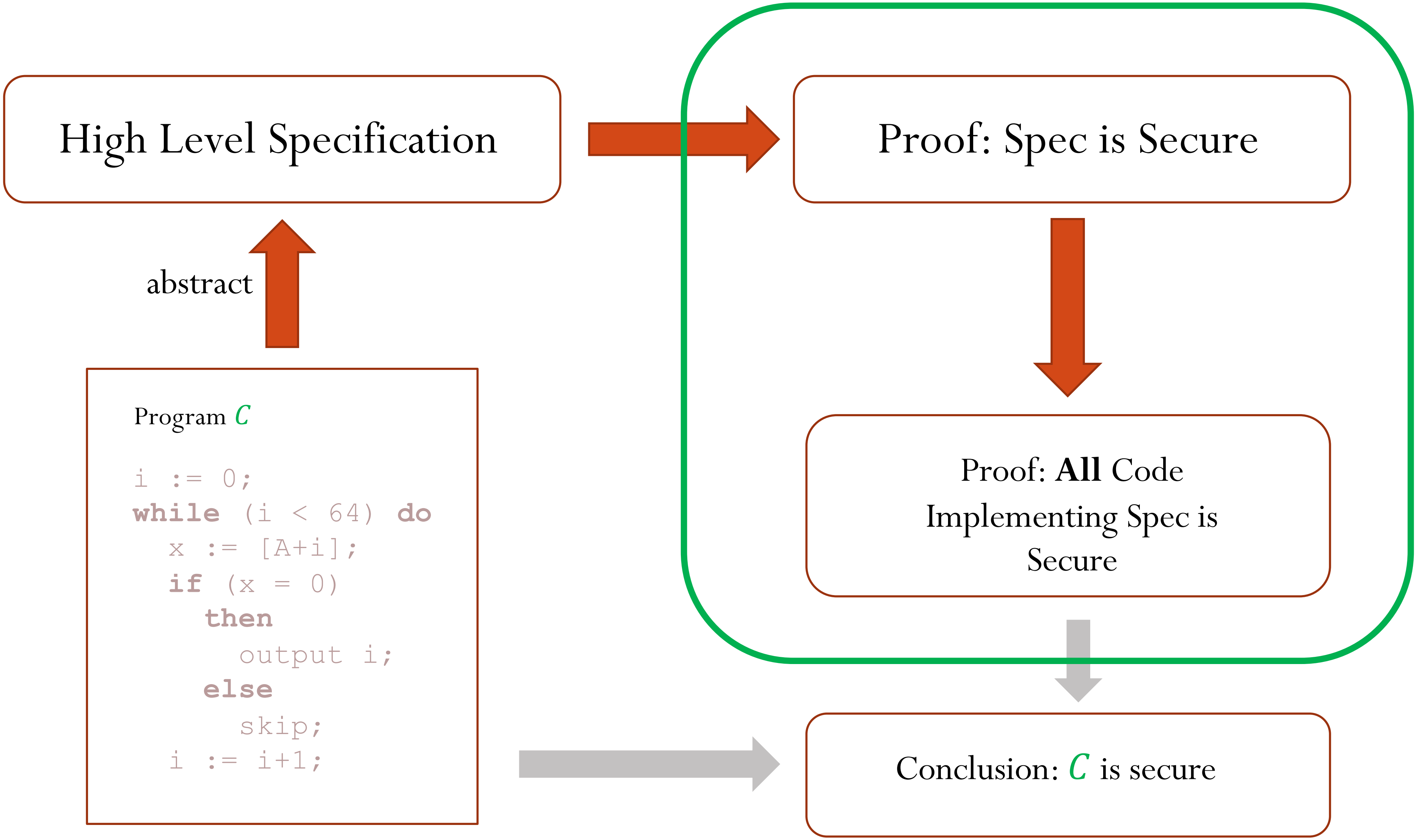
Declassify?

High Security

Rest of Talk

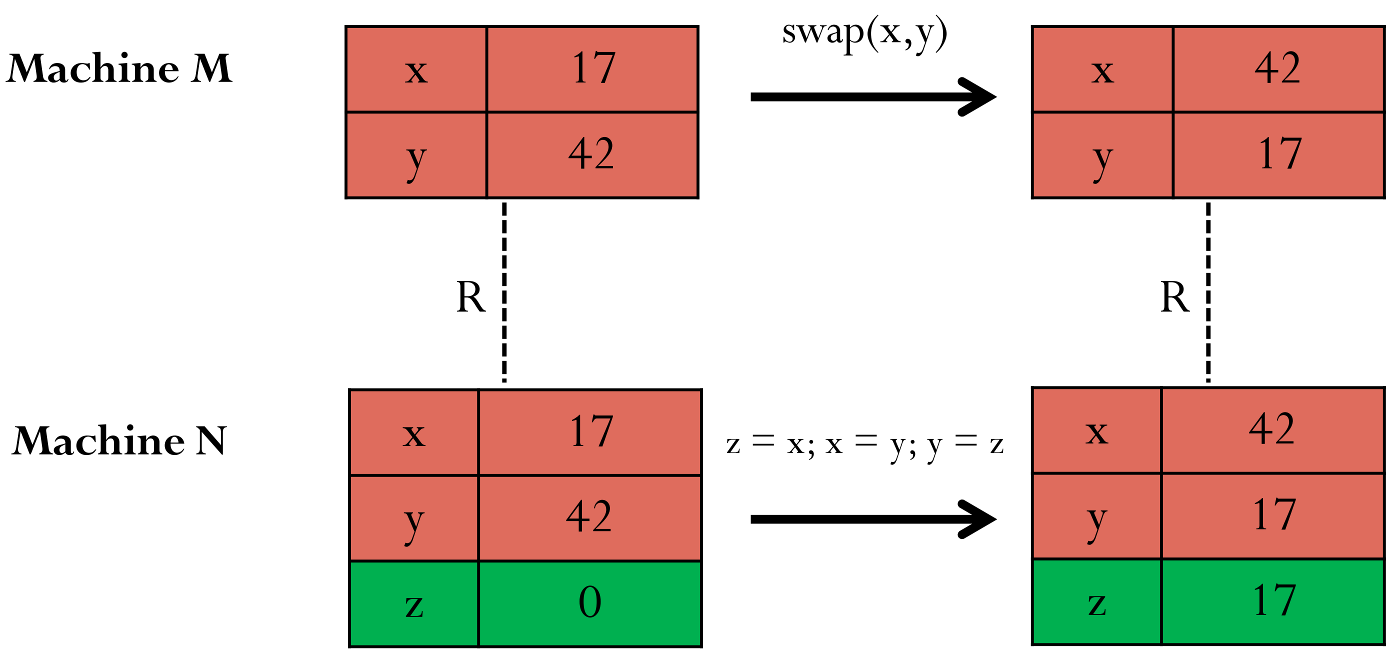
1. Specifying and proving security
2. Propagating security across simulations
3. Experience with CertiKOS security proof
4. Limitations and extensions

Ideal Solution



Insecure Simulation

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



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

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)

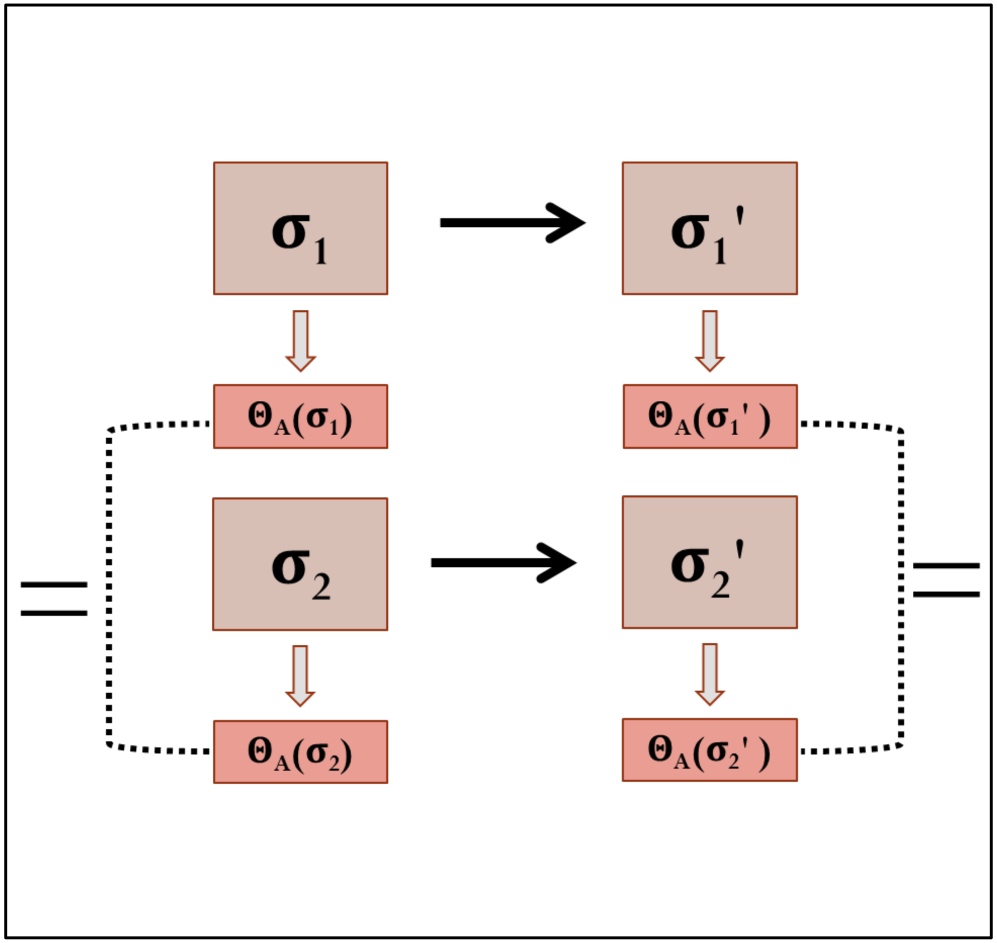
$$\forall\, \boldsymbol{\sigma}_1, \boldsymbol{\sigma}_2, s_1, s_2 \,.$$

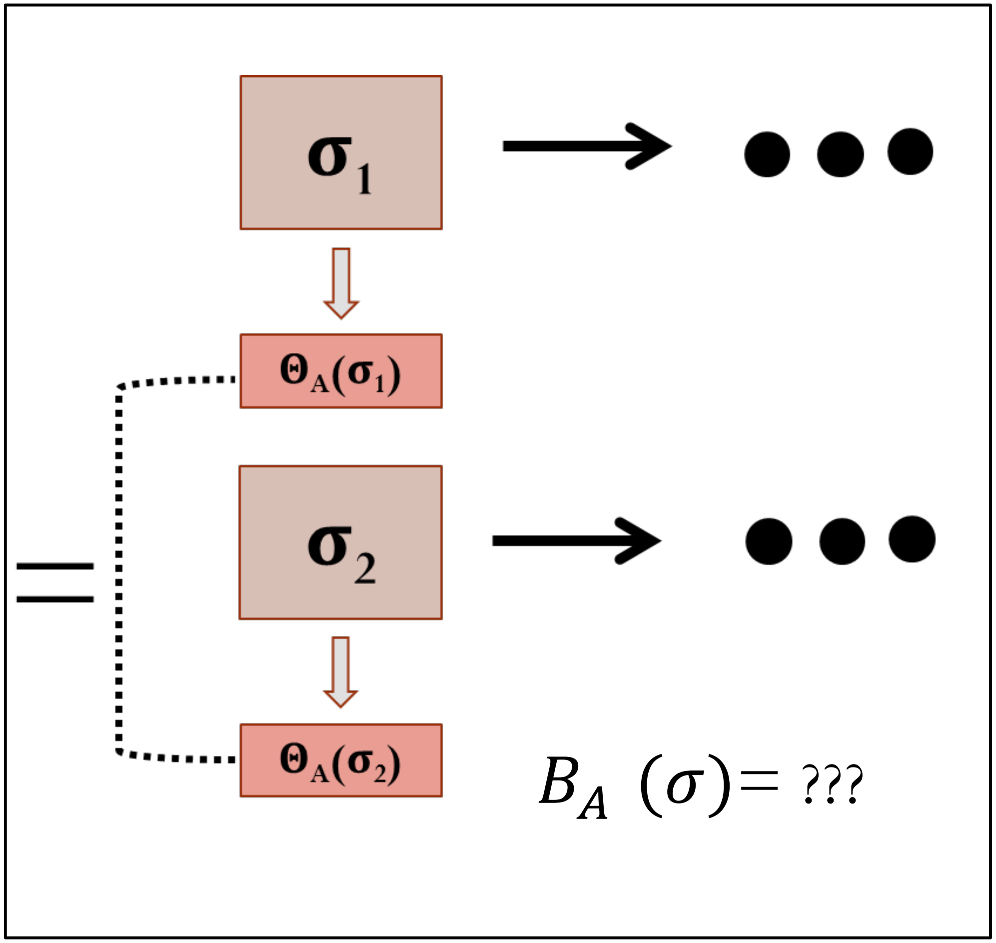
$$\Theta_p^M(\boldsymbol{\sigma}_1) = \Theta_p^M(\boldsymbol{\sigma}_2) \wedge R(\boldsymbol{\sigma}_1, s_1) \wedge R(\boldsymbol{\sigma}_2, s_2)$$

$$\Longrightarrow$$

$$\Theta_p^N(s_1) = \Theta_p^N(s_2)$$

Whole-Execution Behaviors

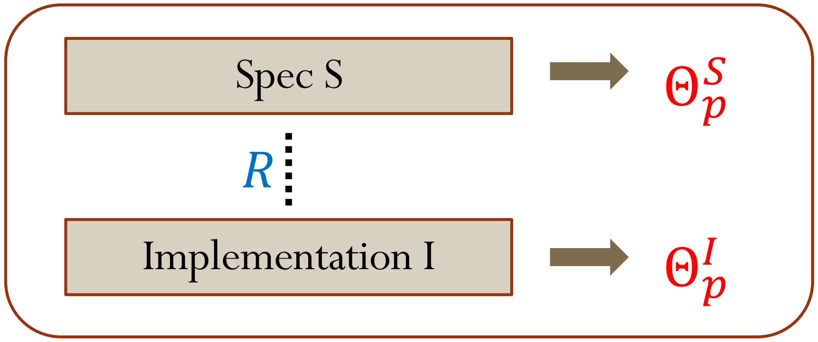




Can define $B_A\ (\sigma)$ if Θ_A is "monotonic" (behaves like an output buffer)

- *only* required for low-level implementation
- see PLDI2016 paper for technical details

End-to-End Security



If R is a **security-preserving simulation** and Θ_p^I is **monotonic**, then:

Generalized Noninterference:

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

End-to-End Security:

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

Rest of Talk

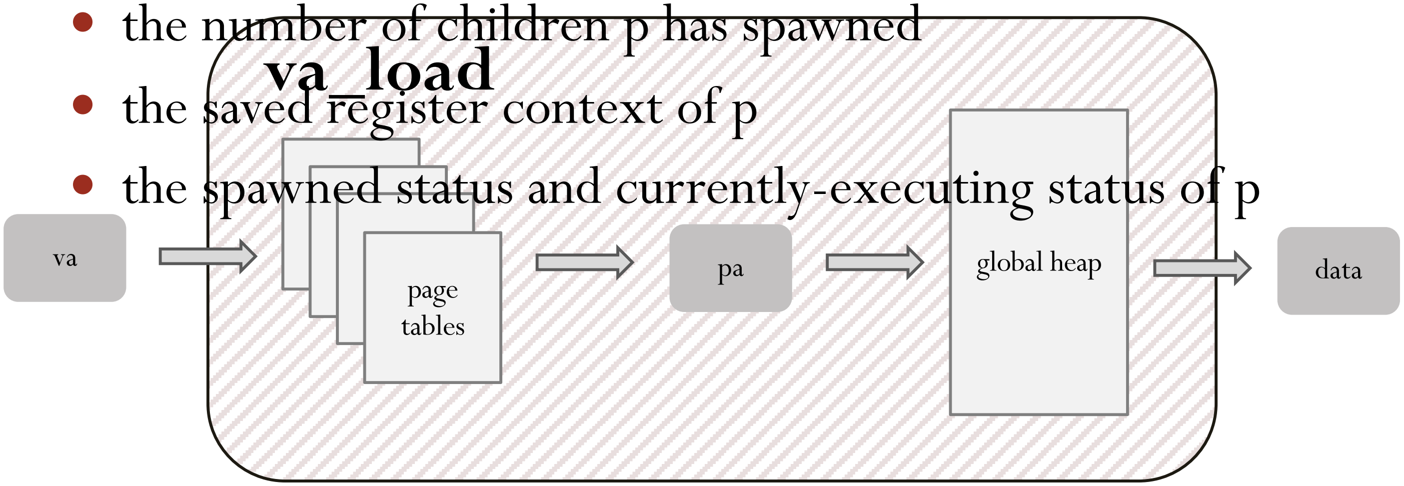
1. Specifying and proving security
2. Propagating security across simulations
3. Experience with CertiKOS security proof
4. Limitations and extensions

CertiKOS Overview

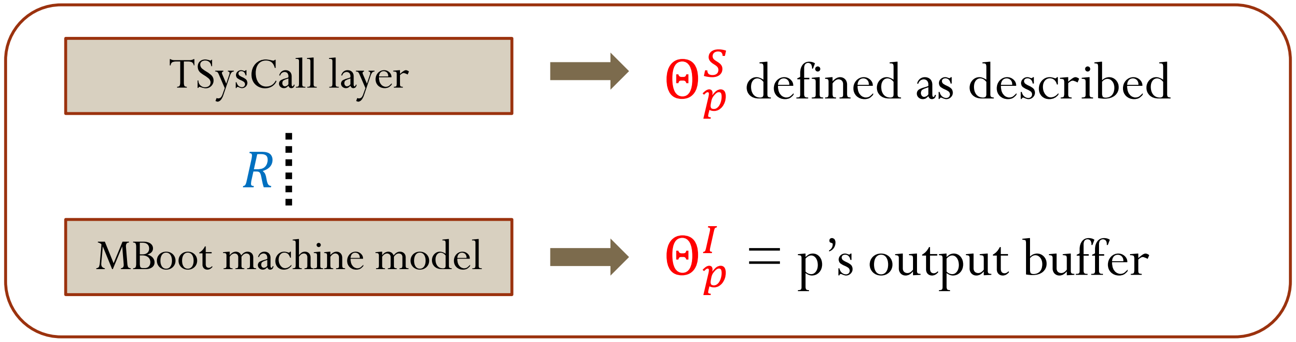
- Certified functionally correct OS kernel with 32 layers
- 354 lines of assembly code, ~3000 lines of C code
  - CompCert compiles C to assembly
- Each layer has primitives that can be called atomically
- Bottom layer MBoot is the x86 machine model
- Top layer TSysCall contains 9 system calls as primitives
  - 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



R is a **security-preserving simulation**

Θ_p^I is **monotonic**

Generalized Noninterference:

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

End-to-End Security:

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

Evaluation

Security of Primitives (LOC)

Load	147
Store	258
Page Fault	188
Get Quota	10
Spawn	30
Yield	960
Start User	11
Print	17
Total	**1621**

Security Proof (LOC)

Primitives	1621
Glue	853
Framework	2192
Invariants	1619
Total	**6285**

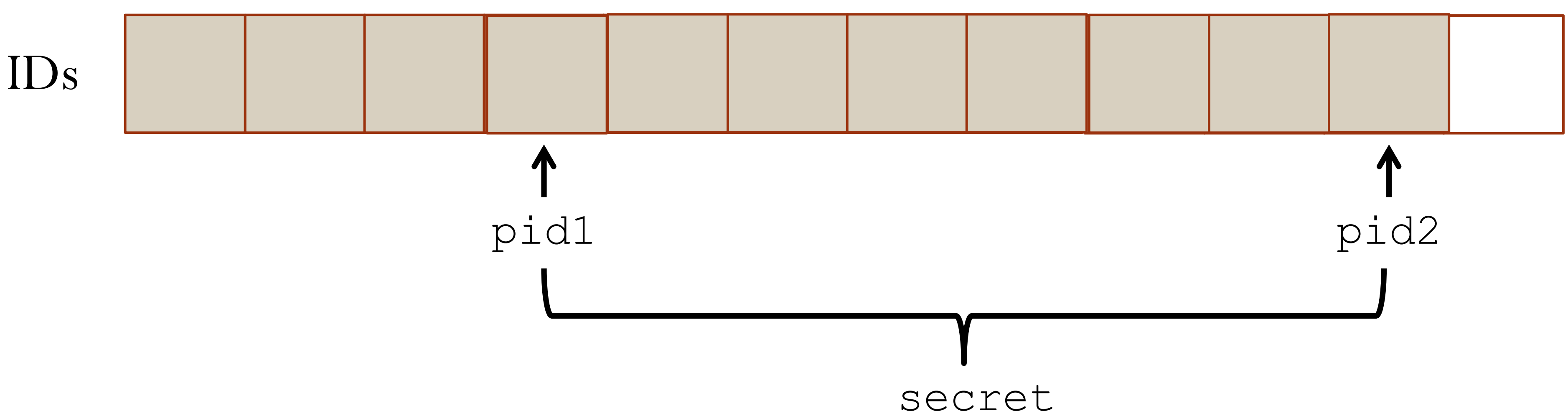
Time needed for Coq proof effort: ~ 6 months

CertiKOS Security Leak

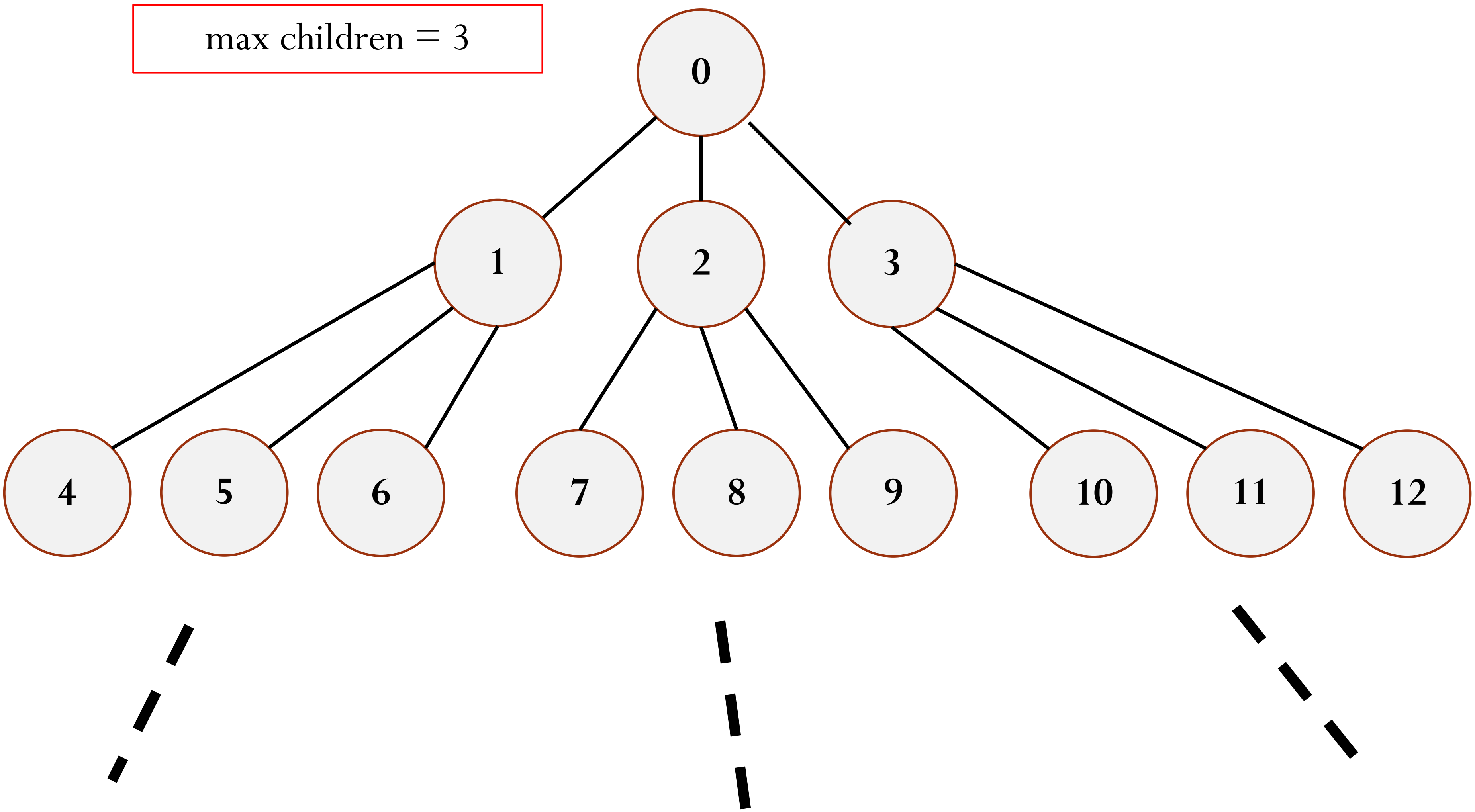
```
function alice {
  int pid1 = proc_spawn();
  yield();
  int pid2 = proc_spawn();
  print(pid2 - pid1 + 1);
}
```

||

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



Solution to Leak



Rest of Talk

1. Specifying and proving security
2. Propagating security across simulations
3. Experience with CertiKOS security proof
4. Limitations and Extensions
   a. Model Fidelity
   b. Virtualized Time
   c. Top-Level CertiKOS Theorem

Machine Model Fidelity

- Gaps between MBoot machine model and the physical x86 hardware
  - **Completeness** – some unmodeled assembly instructions (e.g., RDTSC)
  - **Soundness** – must trust that we modeled x86 instructions faithfully
  - **Safety** – must assume that users never execute code modeled as undefined behavior

Future plans to deal with safety gap:

- Define a user-level machine model with three types of instructions
  - **Interrupt** – trap into the kernel to handle a privileged instruction or syscall
  - **Load/Store** – access global heap according to the kernel's load/store specs
  - **Other** – other user-level instructions, which are only allowed to use local registers

  - Instructions of first two types are proved to be safe
  - Instructions of third type are safe due to restriction to local registers

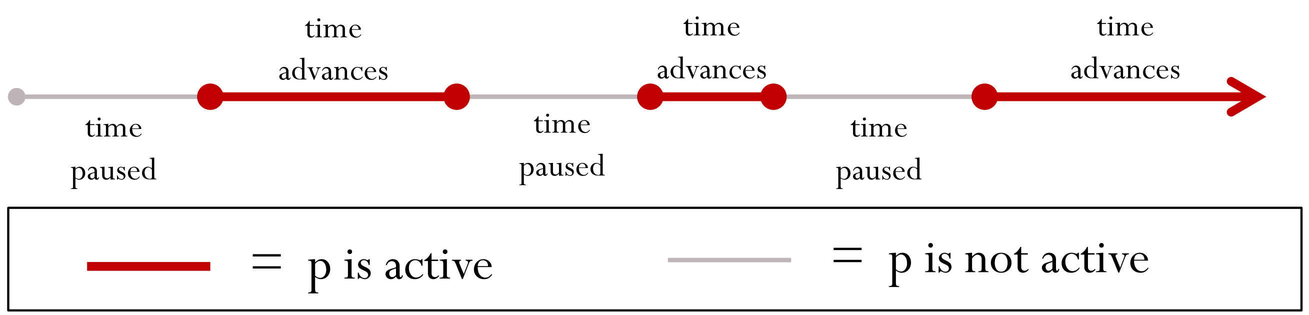
New Feature: Virtualized Time

```
function alice {
  int t0 = gettime();
  while (true) {
    for i = 0 to 10^6 {
       // do some work...
    }
    int t = gettime();
    print(t - t0);
    yield();
  }
}
```

||

```
function bob {
  int t0 = gettime();
  while (true) {
    for i = 0 to 10^6 {
       // do some work...
    }
    int t = gettime();
    print(t - t0);
    yield();
  }
}
```

New Feature: Virtualized Time



```
int gettime() {
  int p = get_cid();
  int t = rd_tsc();
  return (sum_p + (t - cur));
}
```

```
void stoptime() {
  int p = get_cid();
  int t = rd_tsc();
  sum_p += t - cur;
}
```

```
void starttime() {
  cur = rd_tsc();
}
```

New Feature: Virtualized Time

```
Hacker: The current time is 65735500.
Hacker: Ok, yielding now to let Alice execute her program. See you later.

Alice: I did something secret, the time is now 88014576.
Alice: I did something secret, the time is now 116917548.
Alice: I did something secret, the time is now 146445524.
Alice: I did something secret, the time is now 203650560.
Alice: I did something secret, the time is now 205546124.
Alice: I did something secret, the time is now 300386953.
Alice: I did something secret, the time is now 427359527.
Alice: I did something secret, the time is now 429350439.
Alice: I did something secret, the time is now 456707395.
Alice: I've finished my top secret computation!
Alice: It took me 396460583 cycles. I sure hope no one was able to learn anything
Alice: about what I did. Goodbye!

Hacker: And we're back! Let's see what we can figure out about Alice's secret computation.
Hacker: The time is now 104580368. That's only 38844868 cycles since last time.
Hacker: I guess Alice's execution had no effect on my view of time. Oh well.
dsc5@fromage:~/mycertikos-secure-tsc/certikos/kernel$
```

End-to-End Security in CertiKOS

End-to-End Security:

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

Requires understanding and trusting the observation function.

But CertiKOS enforces pure isolation on processes; can we do better?

Proposed solution (not yet completed):

1. Define $Spawned(p)$ = process p was *just spawned* by the kernel
2. Prove: $\forall \sigma_1, \sigma_2 \in Spawned(p) \,.\, \Theta_p^S(\sigma_1) = \Theta_p^S(\sigma_2)$

$\Rightarrow$ End-to-end security theorem is independent from choice of observation function!

Conclusion

- New methodology using observation function to specify, prove, and propagate IFC policies
  - applicable to all kinds of real-world systems!
- Verification of secure ke... ully within Coq
  - machine-checked pr ofs!
- Future W...
  - high r-... n theorem (independent of observation function choice)
  - more re...
  - preemp...
  - concurr...