Towards General Rely-Guarantee Reasoning for Concurrent, Value-Dependent Noninterference

Toby Murray

University of Melbourne &
Data61 (formerly NICTA), CSIRO

Confidentiality (Information Flow) Proofs

Confidentiality (Information Flow) Proofs

seL4 (Murray et al., S&P 2013)

- General intransitive noninterference over functional spec, transferred to C code by refinement; binary by TV
- seL4: IPC, shared memory, cap transfer, revocation, men allocation, thread creation, notifications, pre-emptive partition scheduling etc.
- approx. 9,600 C SLOC (and ~600 asm, unverified)

Confidentiality (Information Flow) Proofs

seL4 (Murray et al., S&P 2013)

- General intransitive noninterference over functional spec, transferred to C code by refinement; binary by TV
- seL4: IPC, shared memory, cap transfer, revocation, men allocation, thread creation, notifications, pre-emptive partition scheduling etc.
- approx. 9,600 C SLOC (and ~600 asm, unverified)

mCertiKOS (Costanzo et al., PLDI 2016)

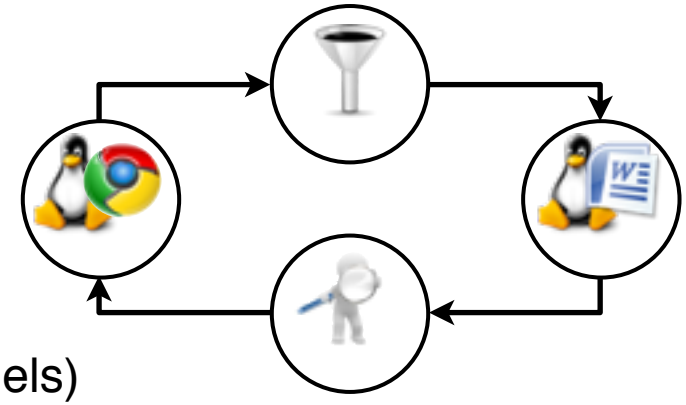
- Strict noninterference between user processes over functional spec, transferred to binary via CompCert
- mCertiKOS: process creation, page fault handling, cooperative scheduling; no IPC
- approx 3,000 C SLOC and (unlike seL4) ~350 asm

Kernel-Only vs Whole-System

Kernel-Only vs Whole-System

Kernel-Only Confidentiality

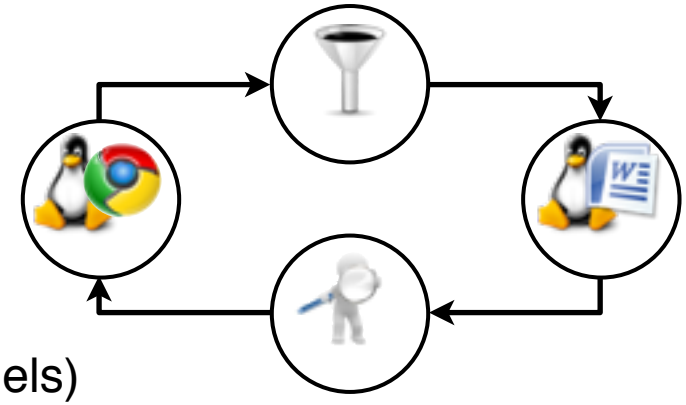
- assumes all application components are maximally hostile
- proved security only by reasoning about kernel enforced abstractions
  - no concurrency (for single-threaded kernels)
- simple static policies only



Kernel-Only vs Whole-System

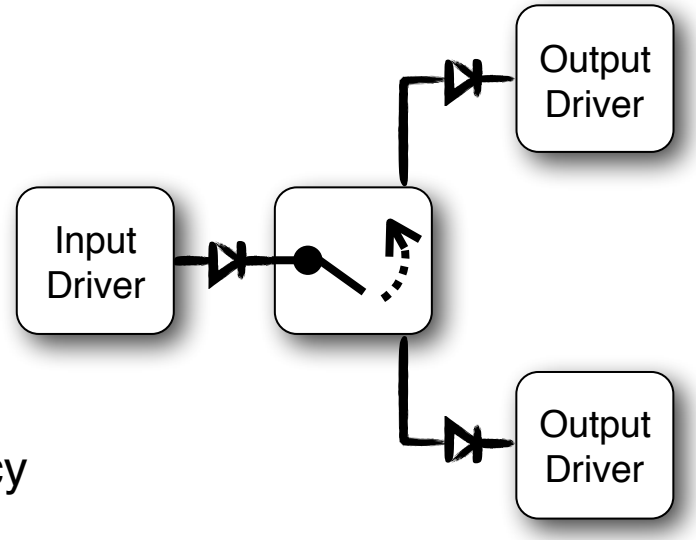
Kernel-Only Confidentiality

- assumes all application components are maximally hostile
- proved security only by reasoning about kernel enforced abstractions
  - no concurrency (for single-threaded kernels)
- simple static policies only



Whole-System Confidentiality

- reasons about behaviour of application components
- proves that their composition yields a secure system
  - time to start worrying about concurrency



The Cross Domain Desktop Compositor

Australian Government

Department of Defence

Defence Science and Technology Group

Mark Beaumont

CDC

The Cross Domain Desktop Compositor

Australian Government
Department of Defence
Defence Science and Technology Group

DATA 61

CSIRO

Mark Beaumont

CDC

The Cross Domain Desktop Composit

Toby Murray

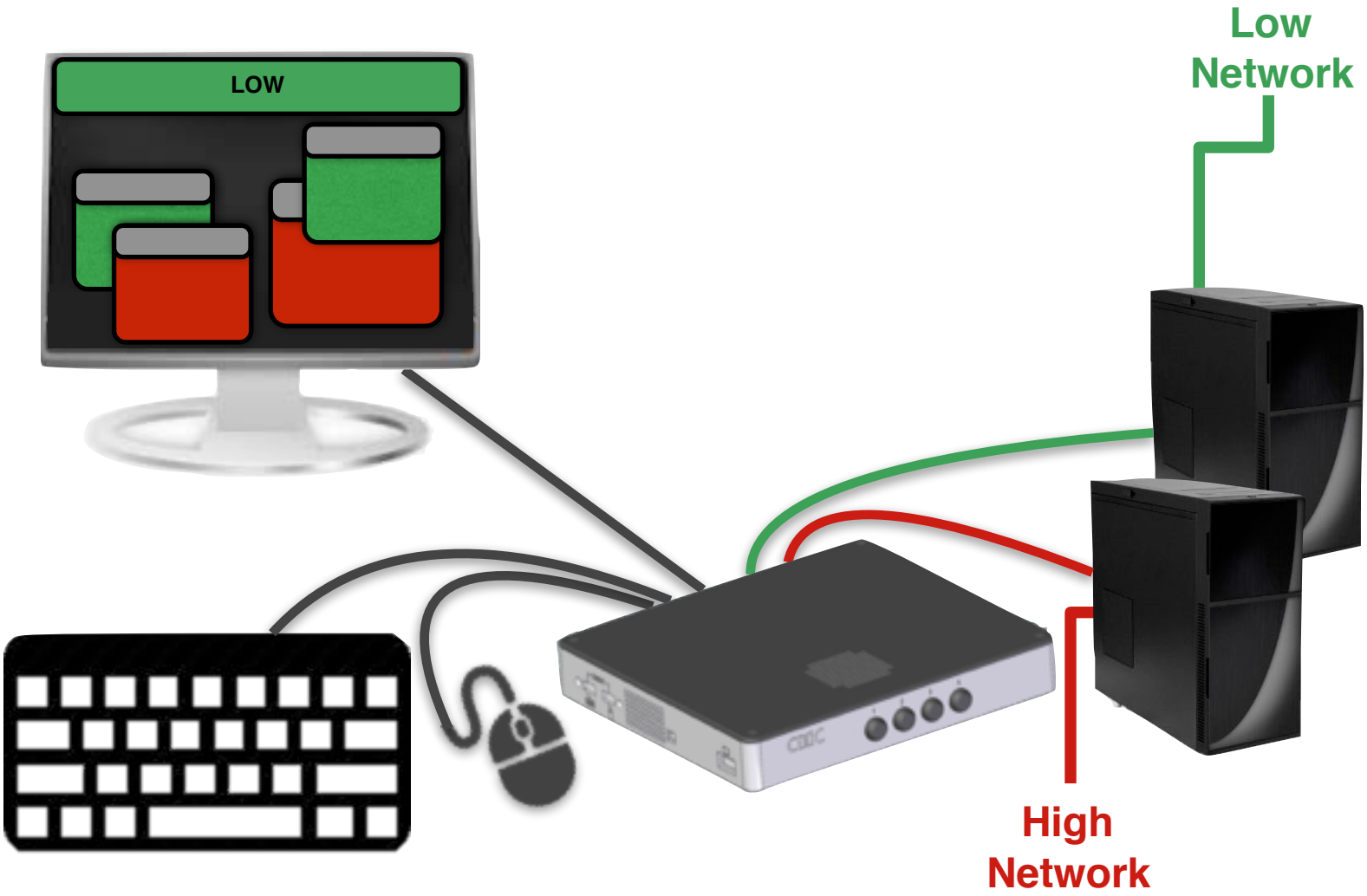
Kevin Elphinstone

Australian Government
Department of Defence
Defence Science and Technology Group

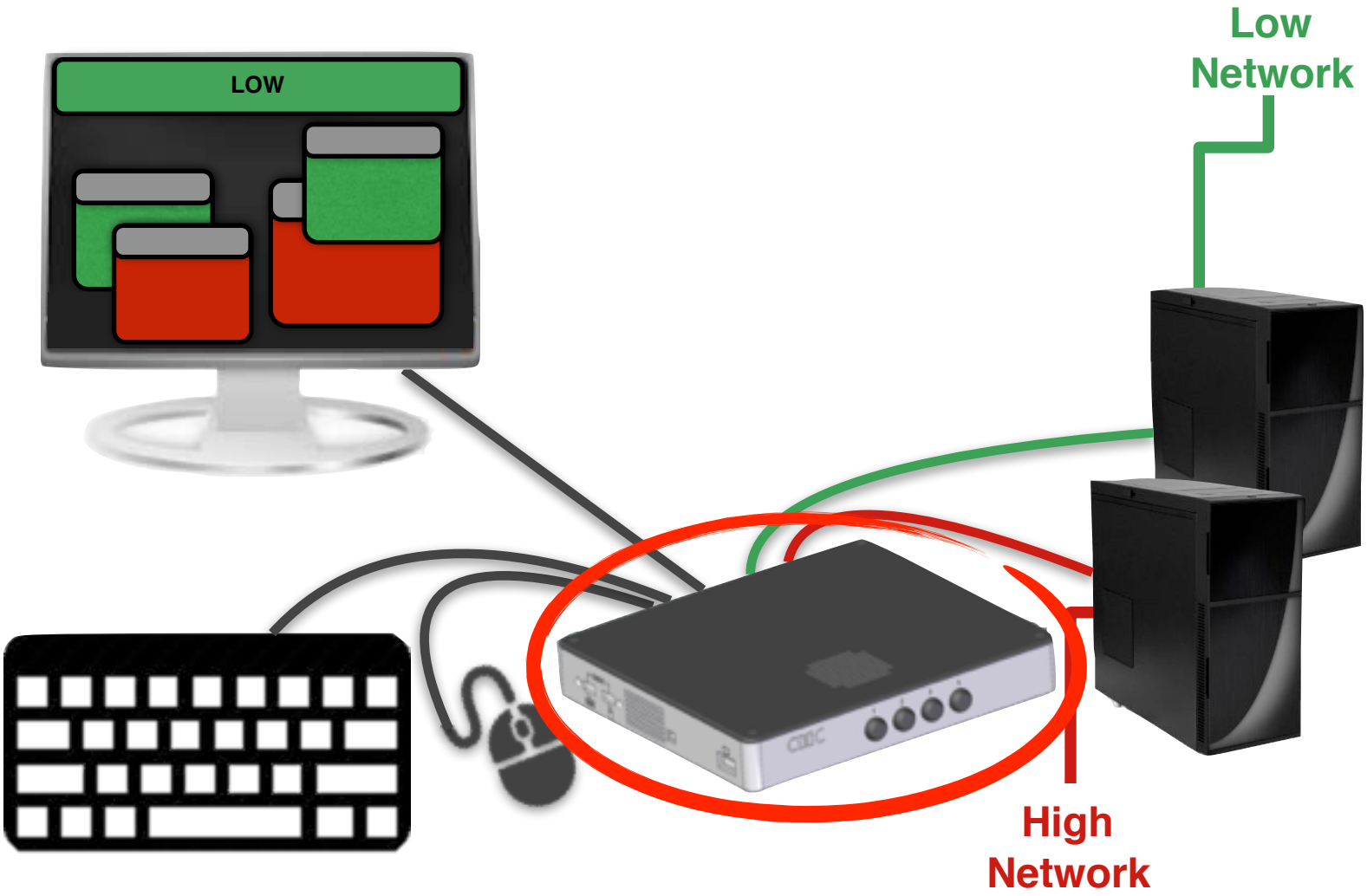
DATA 61

CSIRO

Cross Domain Desktop Compositor

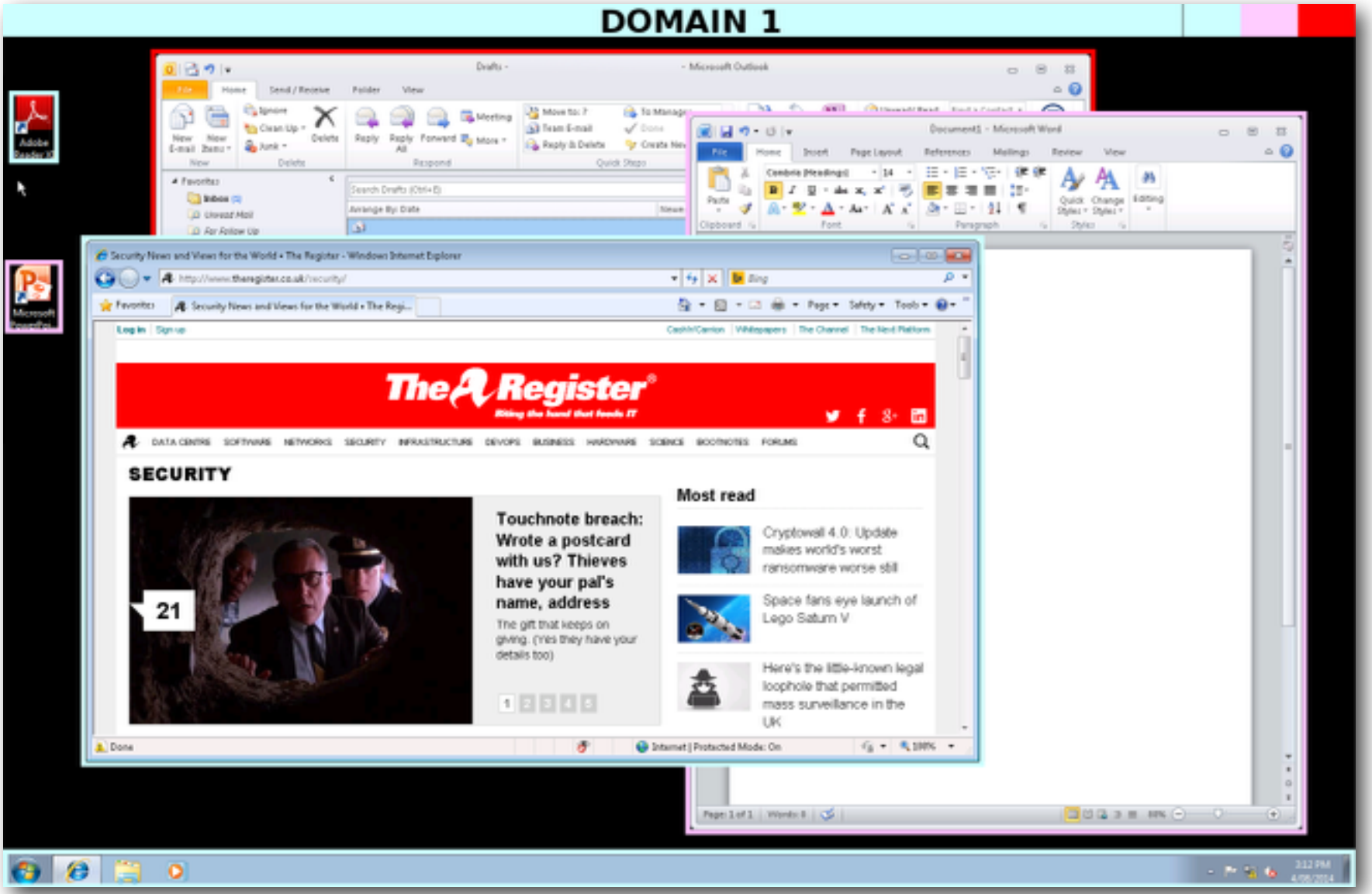


Cross Domain Desktop Compositor

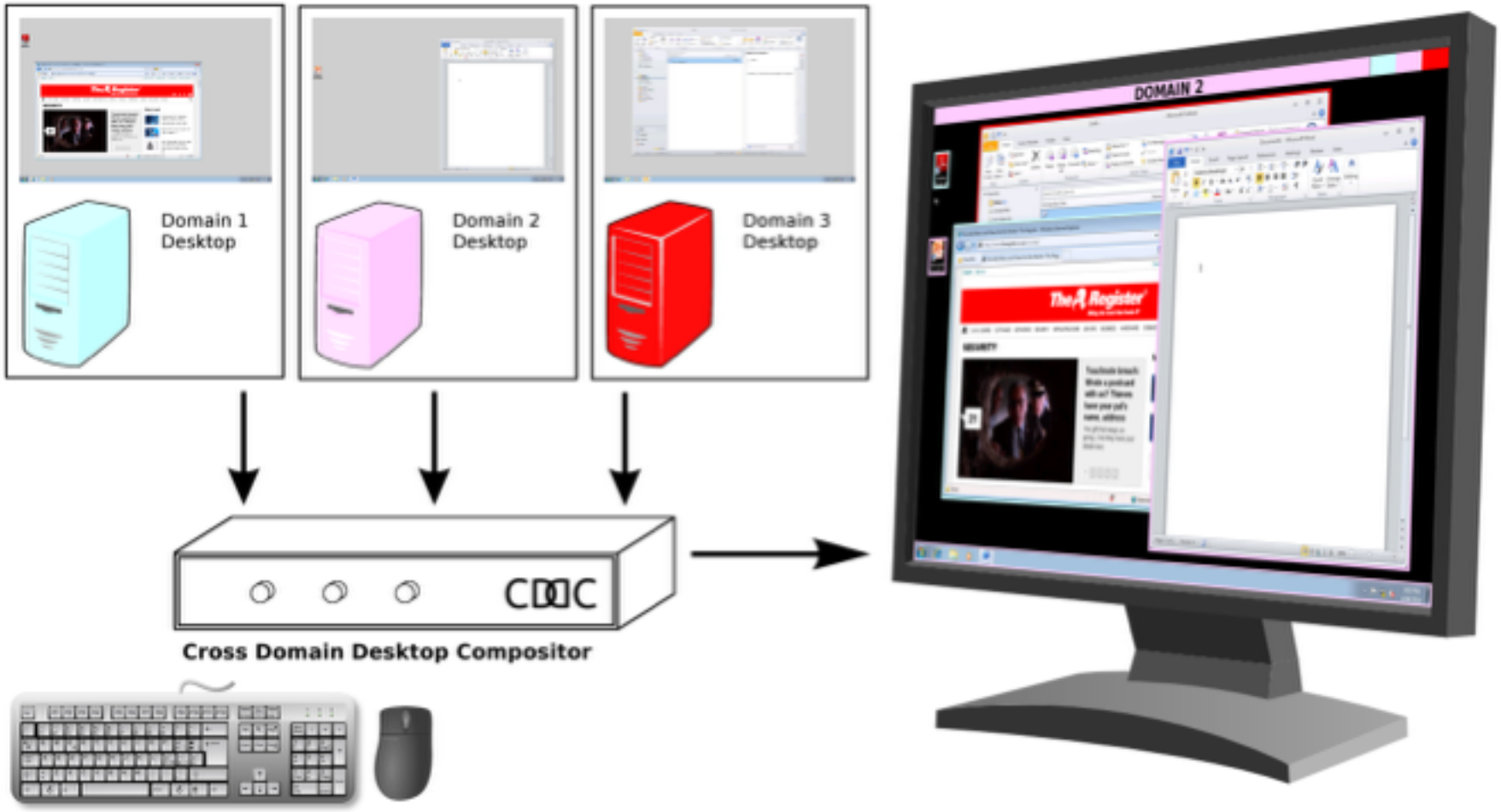


CDDC

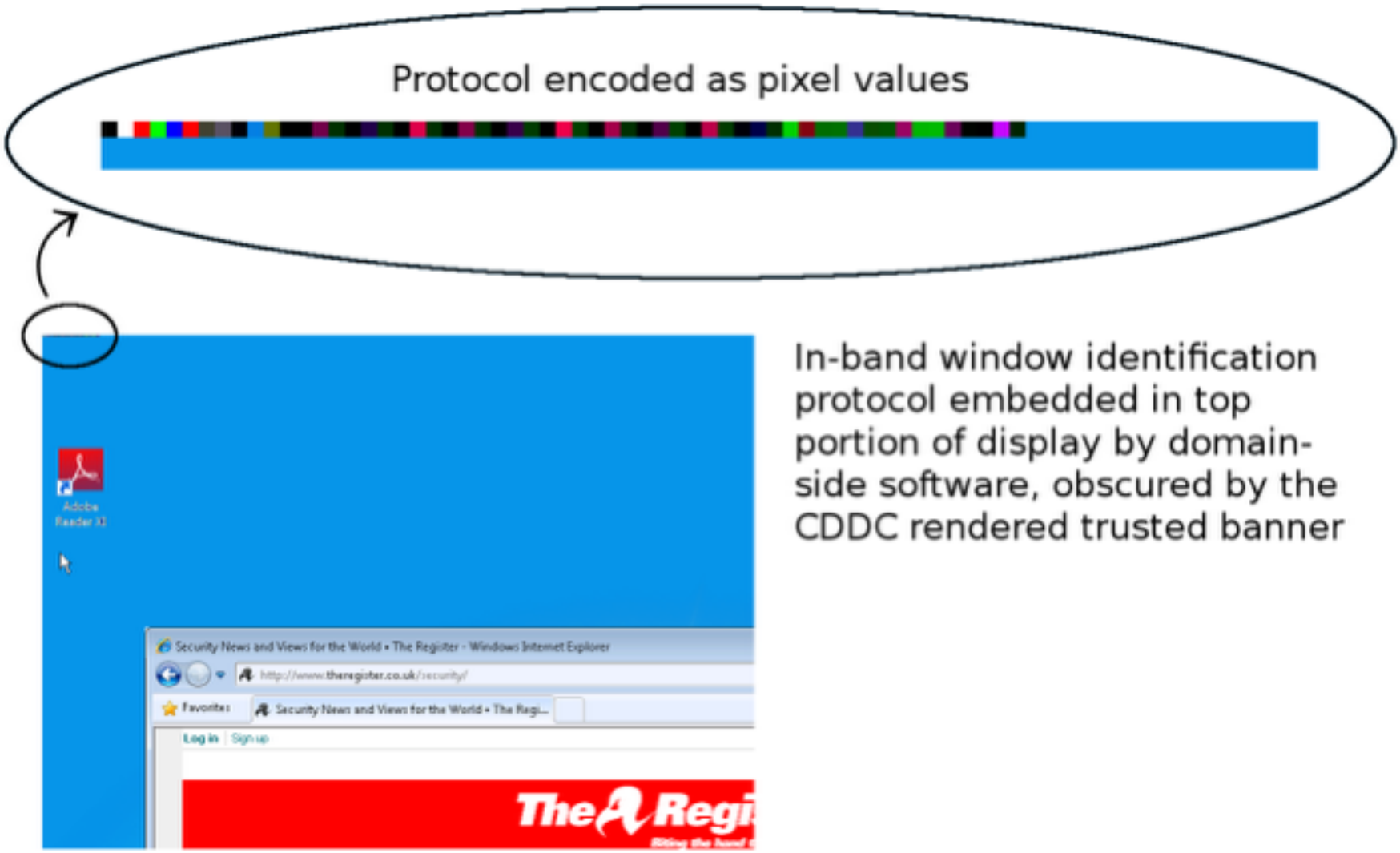
CDDC User Experience



Display Composition

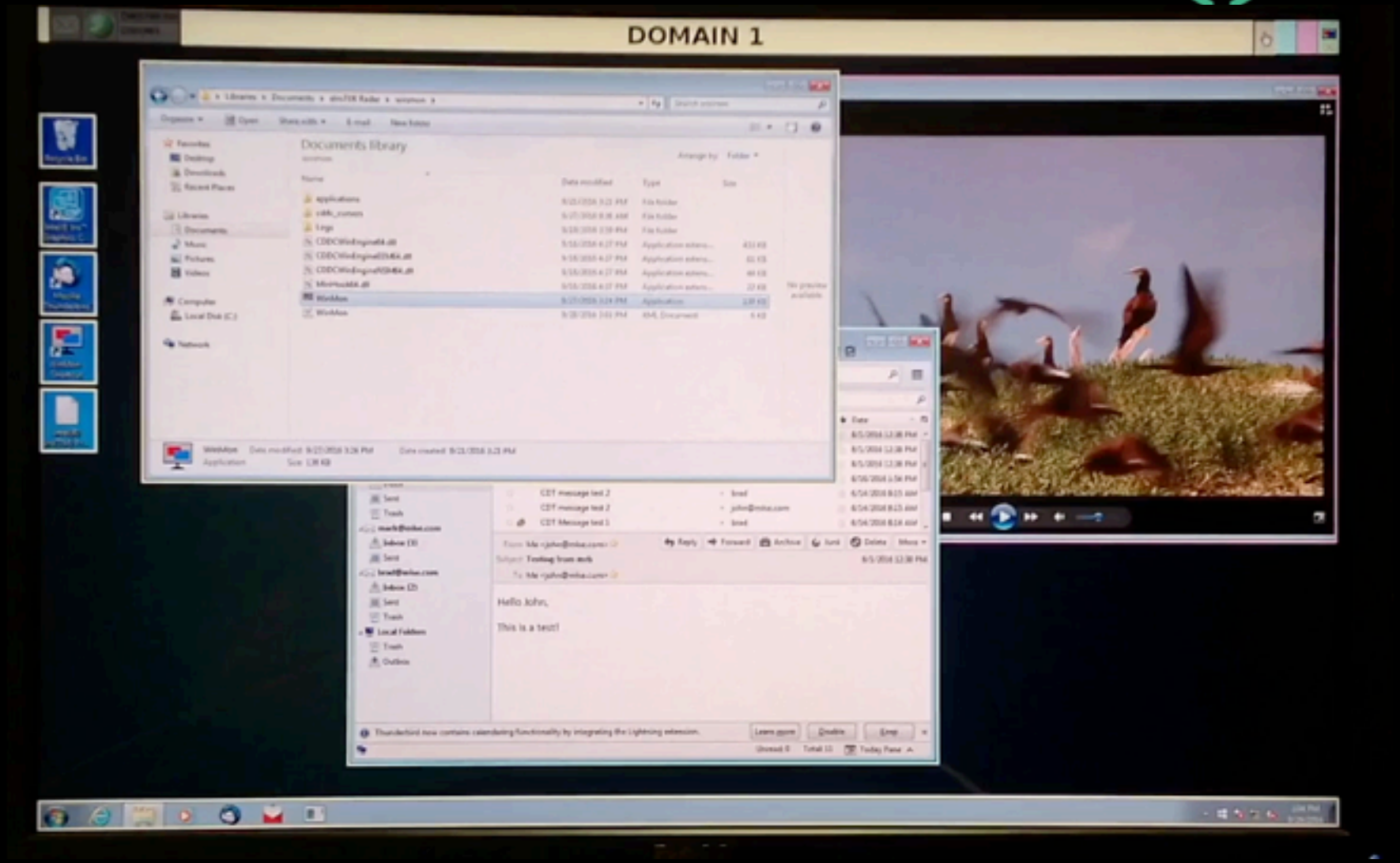


In-Band Composition Protocol

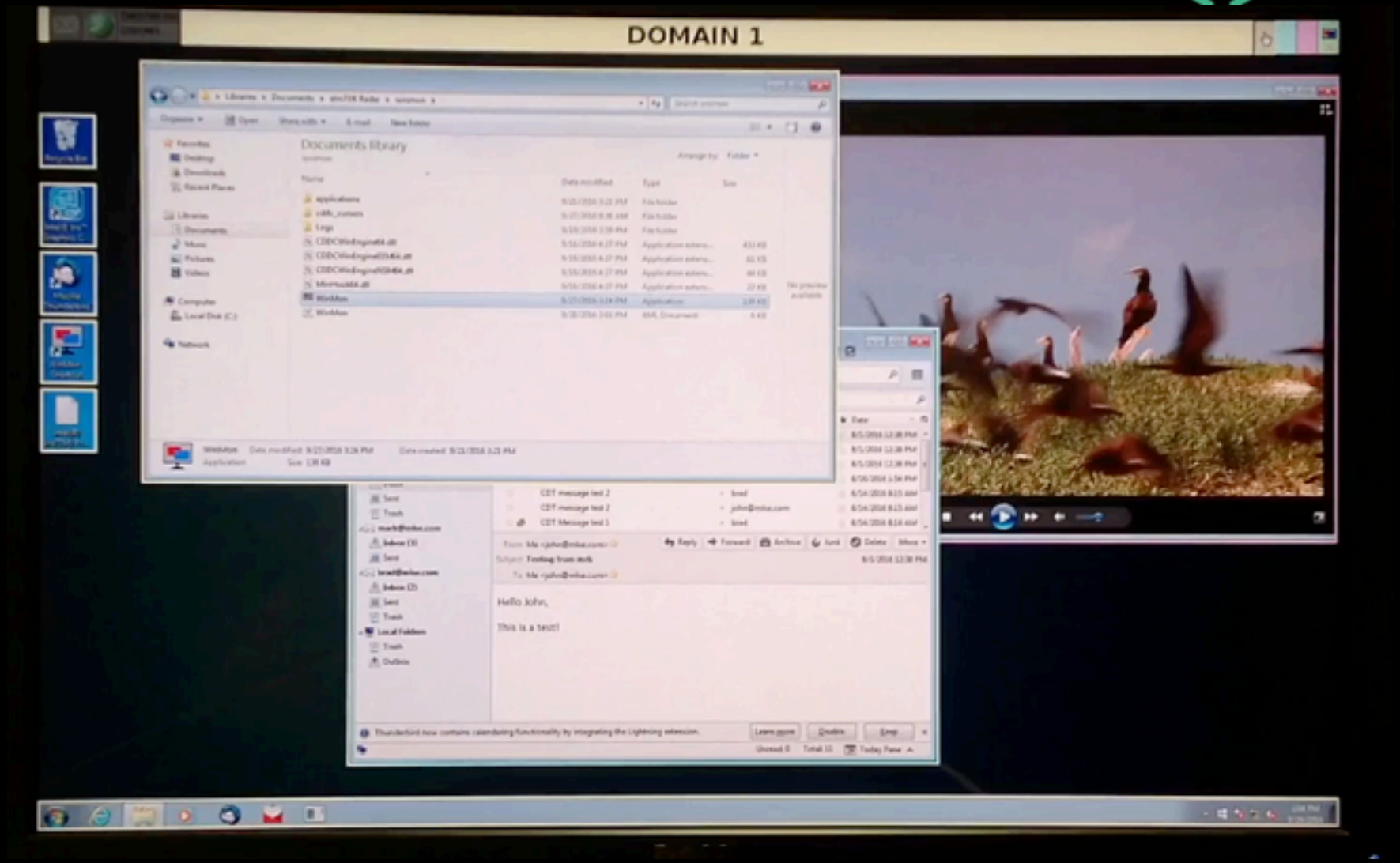


In-band window identification protocol embedded in top portion of display by domain-side software, obscured by the CDDC rendered trusted banner

Demo



Demo



CDDC Functionality

CDDC Functionality

Video Compositing

- Very high bandwidth
- Fixed protocol and policy

CDDC Functionality

Video Compositing

- Very high bandwidth
- Fixed protocol and policy

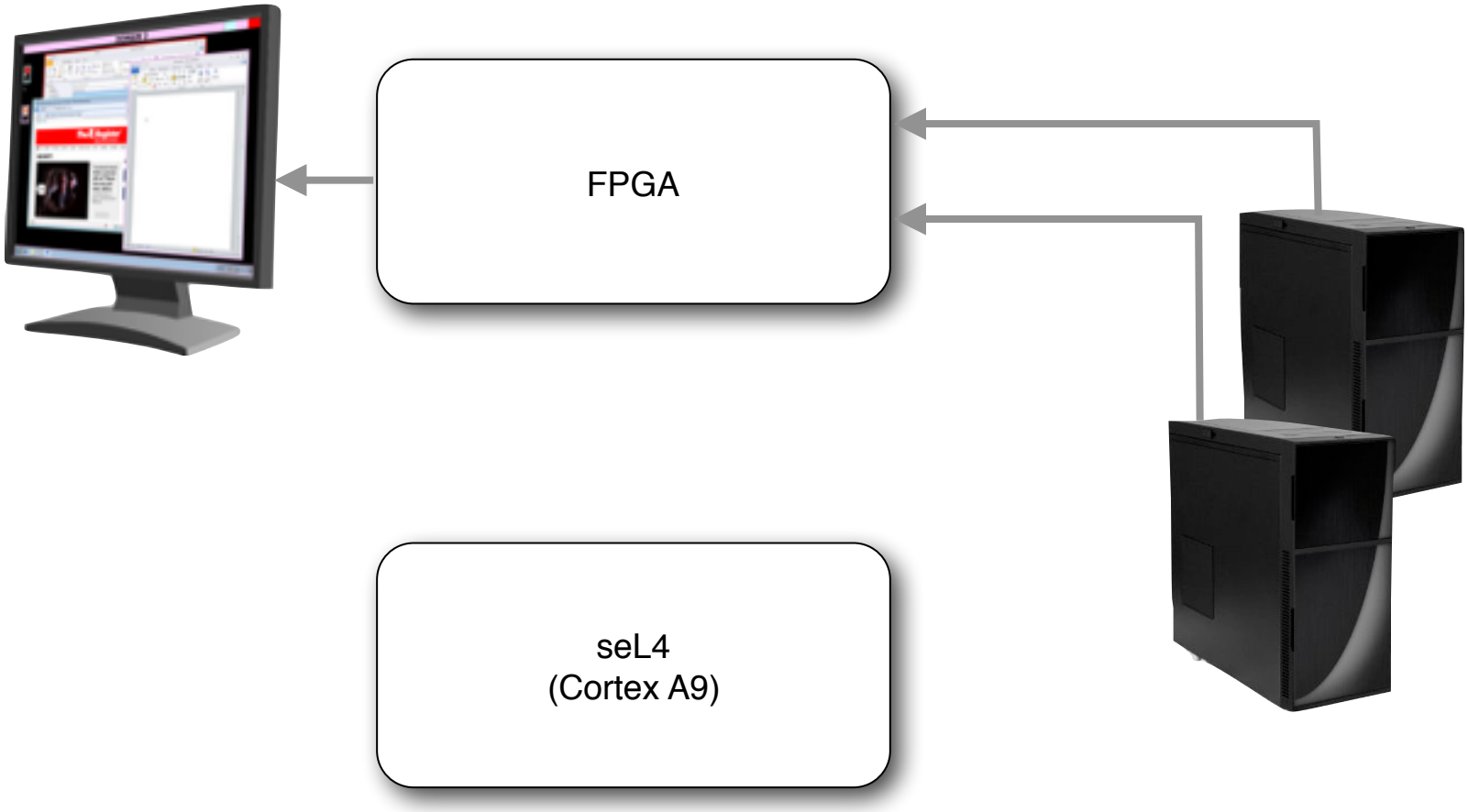
Input Processing

- Low bandwidth
- Policy dependent
  - mouse hover passthrough
  - keyboard shortcut domain switching
  - keystroke broadcast (e.g. Windows-D)
- Runtime configuration dependent
  - keyboard shortcuts for trusted path actions

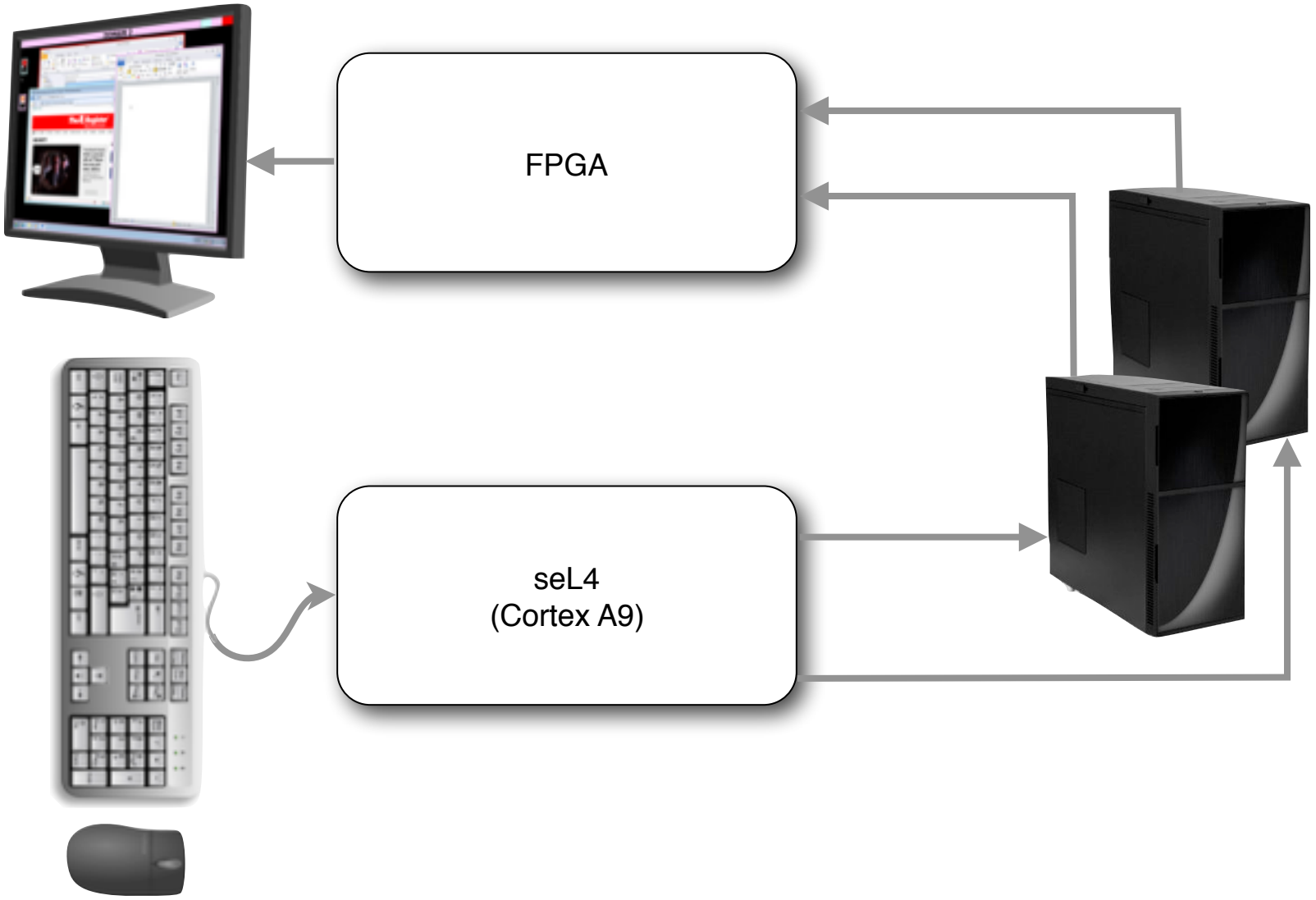
CDDC Architecture

CDDC Architecture

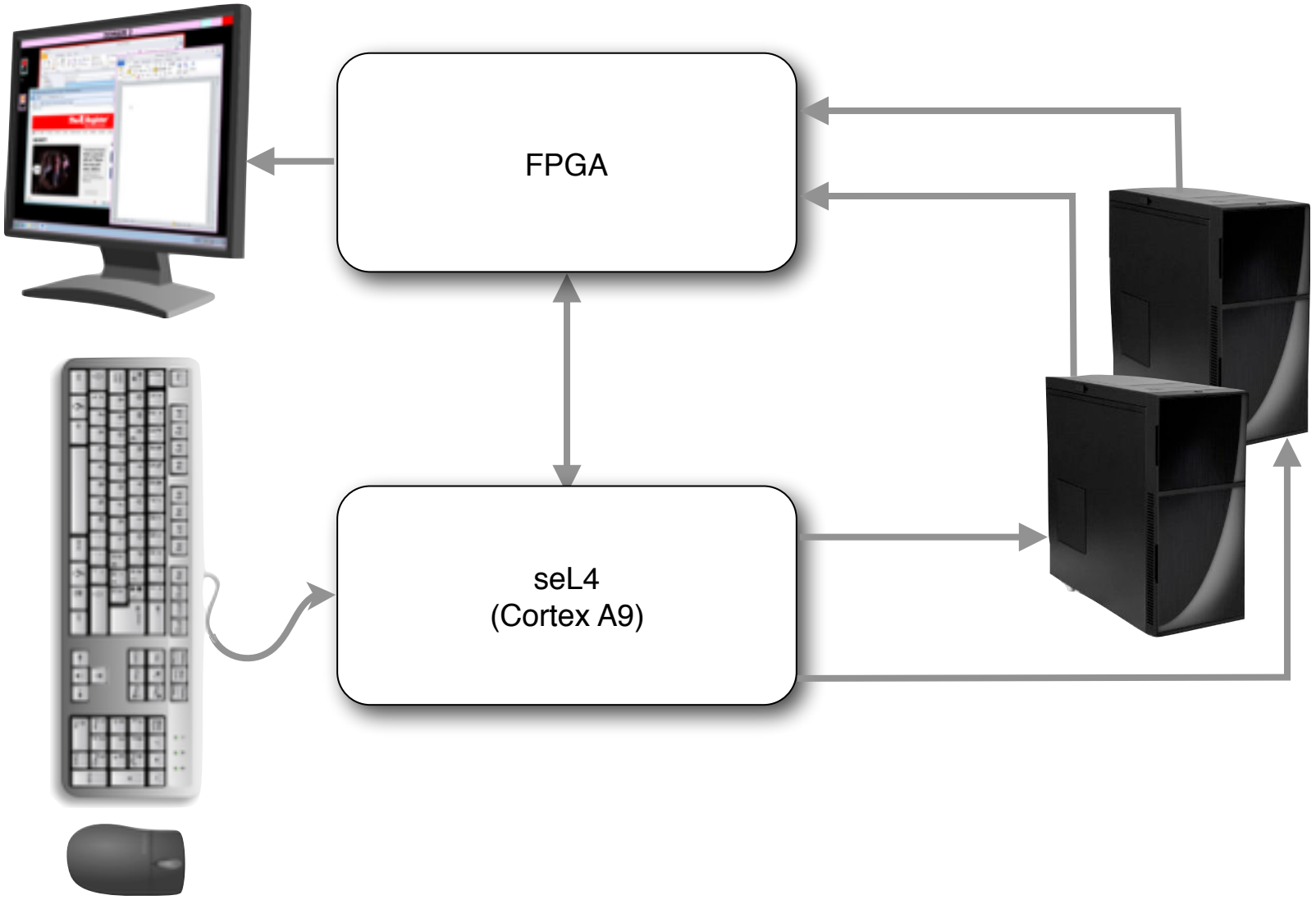
CDDC Architecture



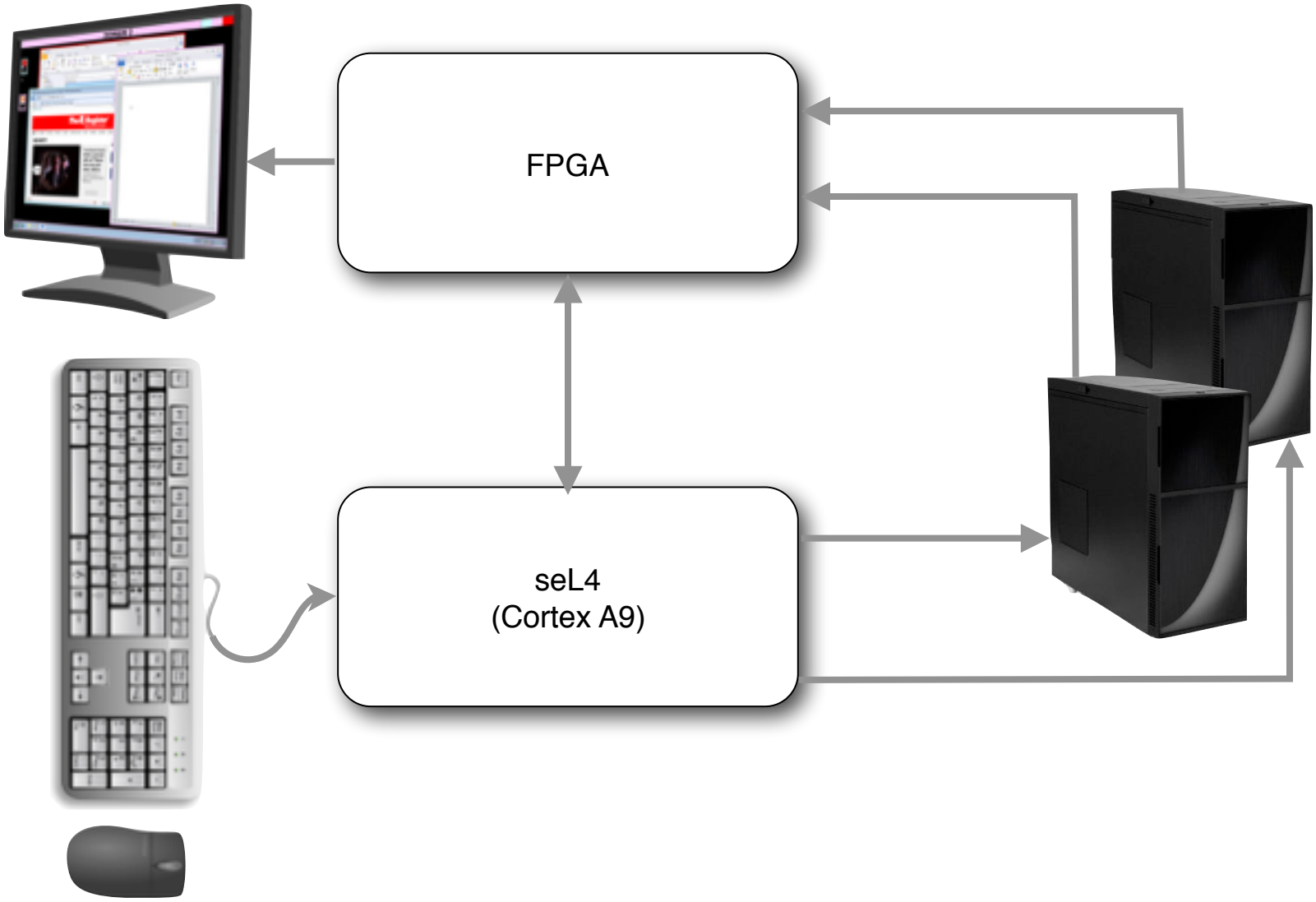
CDDC Architecture



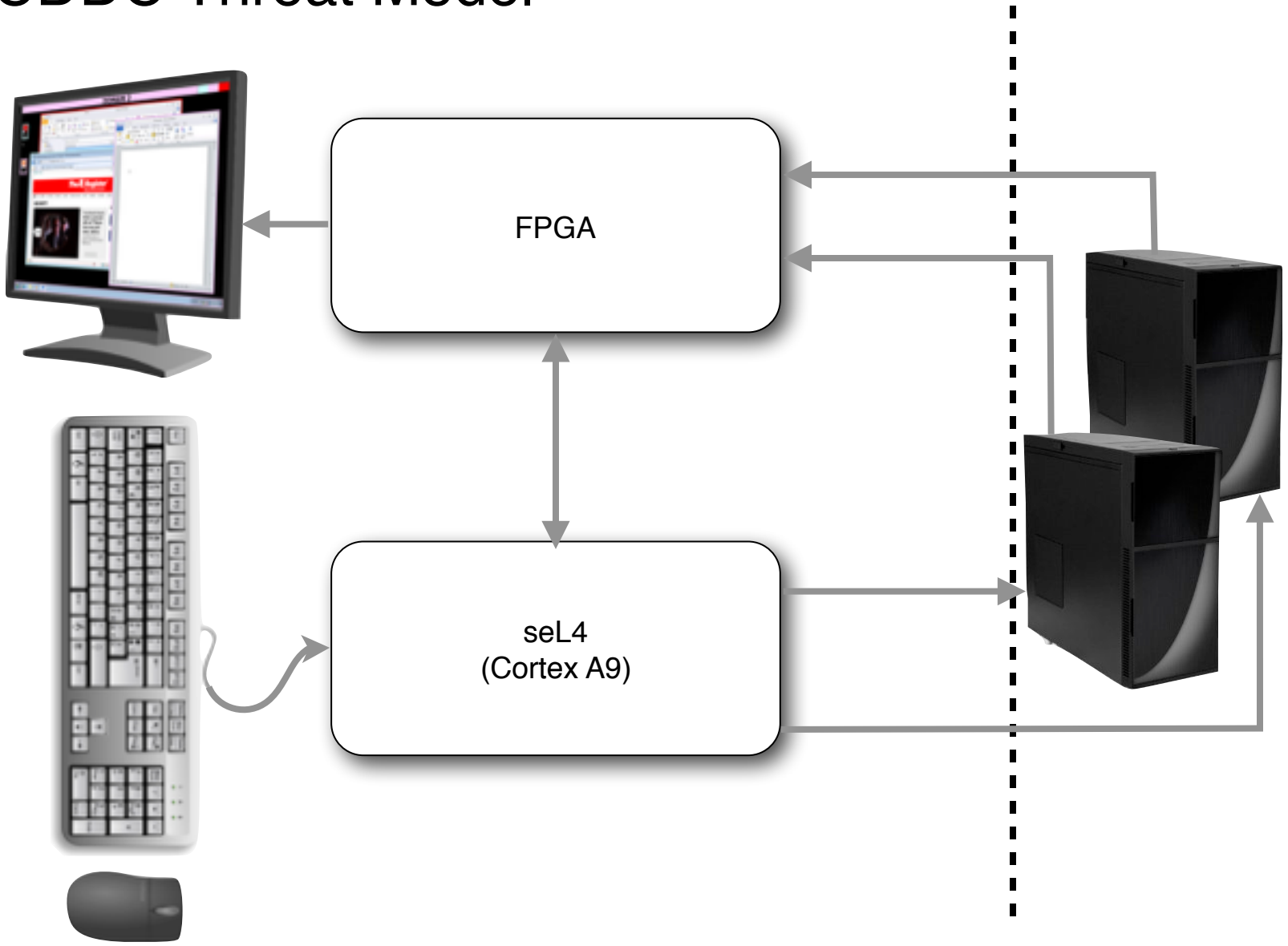
CDDC Architecture



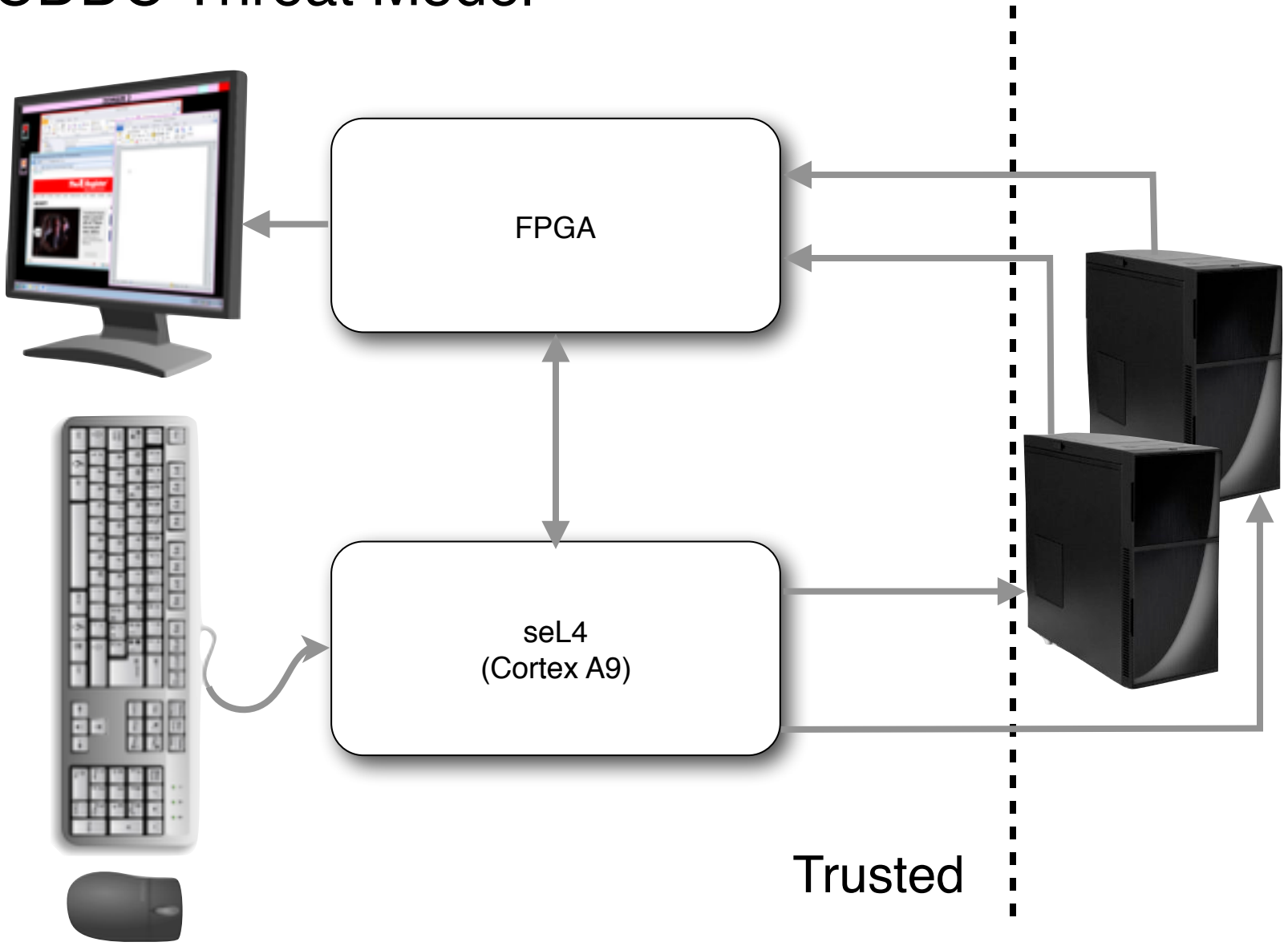
CDDC Threat Model



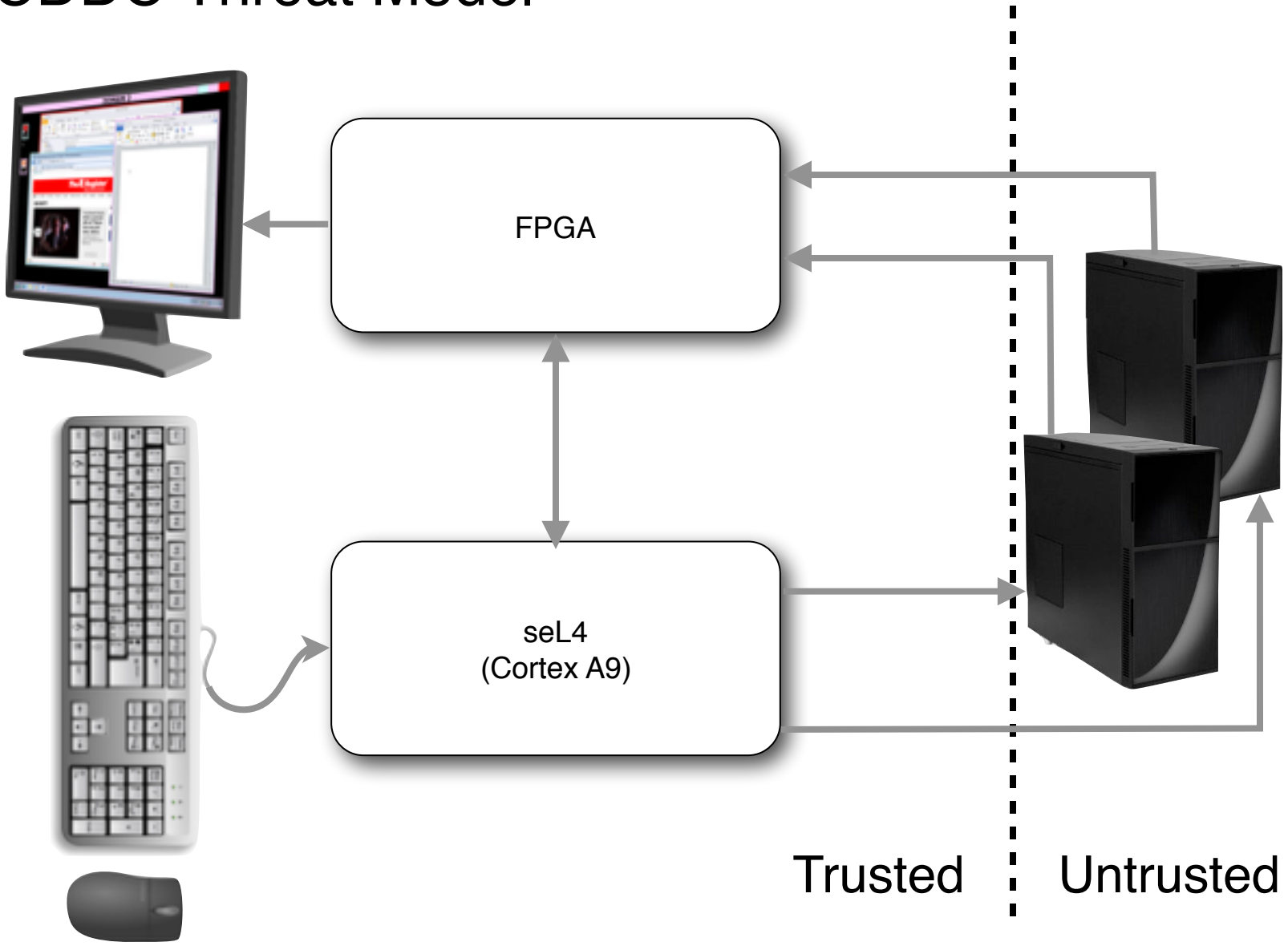
CDDC Threat Model



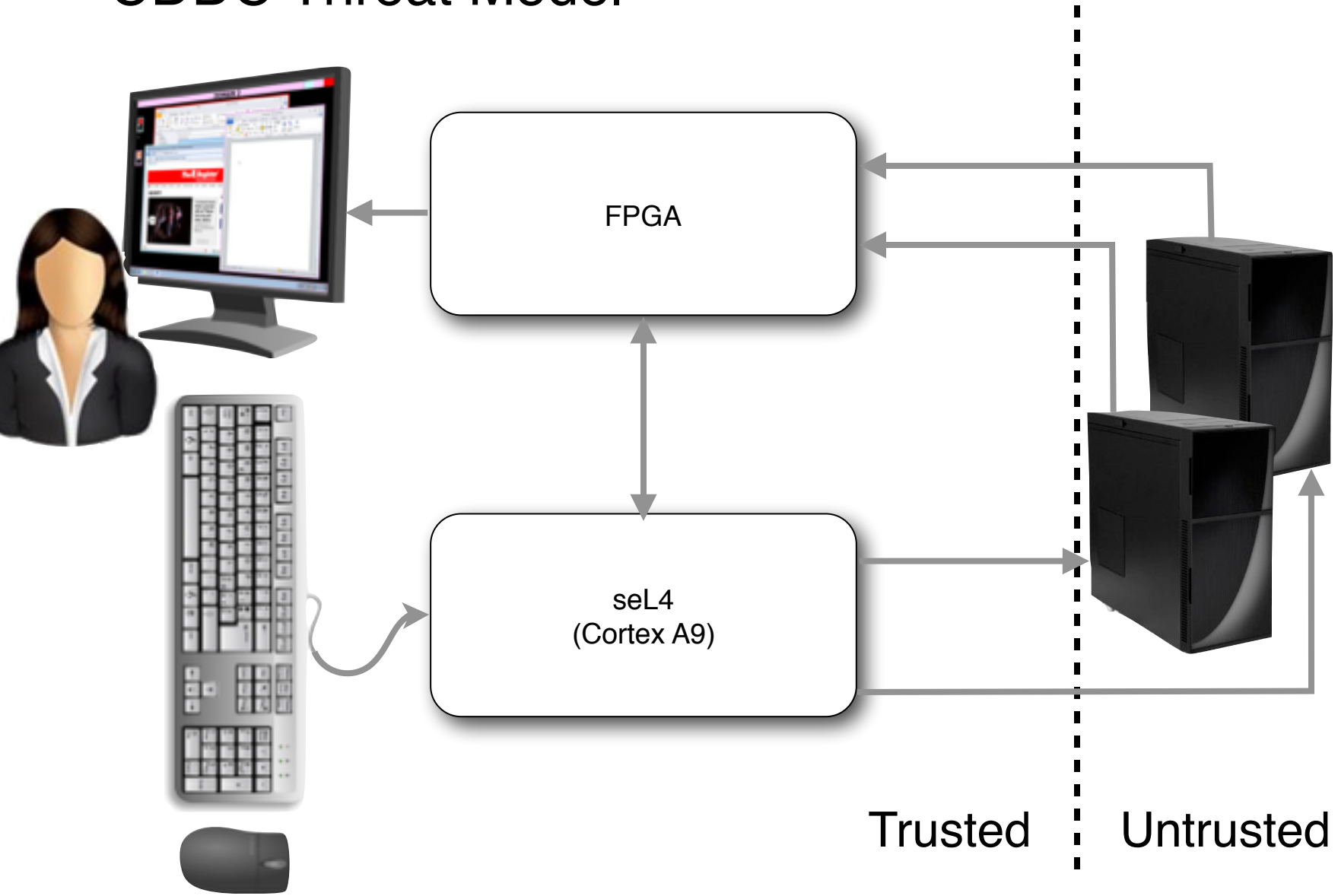
CDDC Threat Model



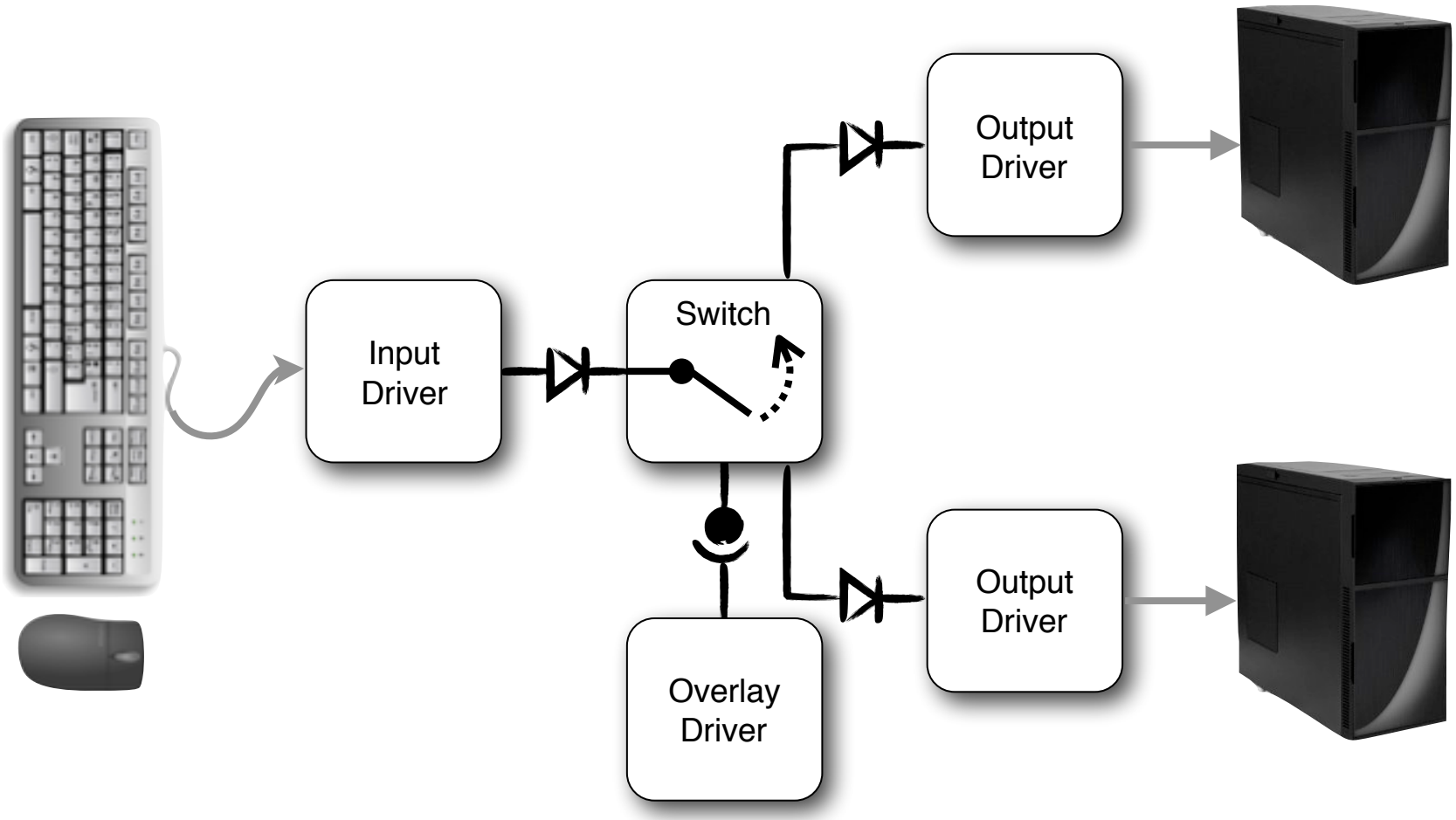
CDDC Threat Model



CDDC Threat Model

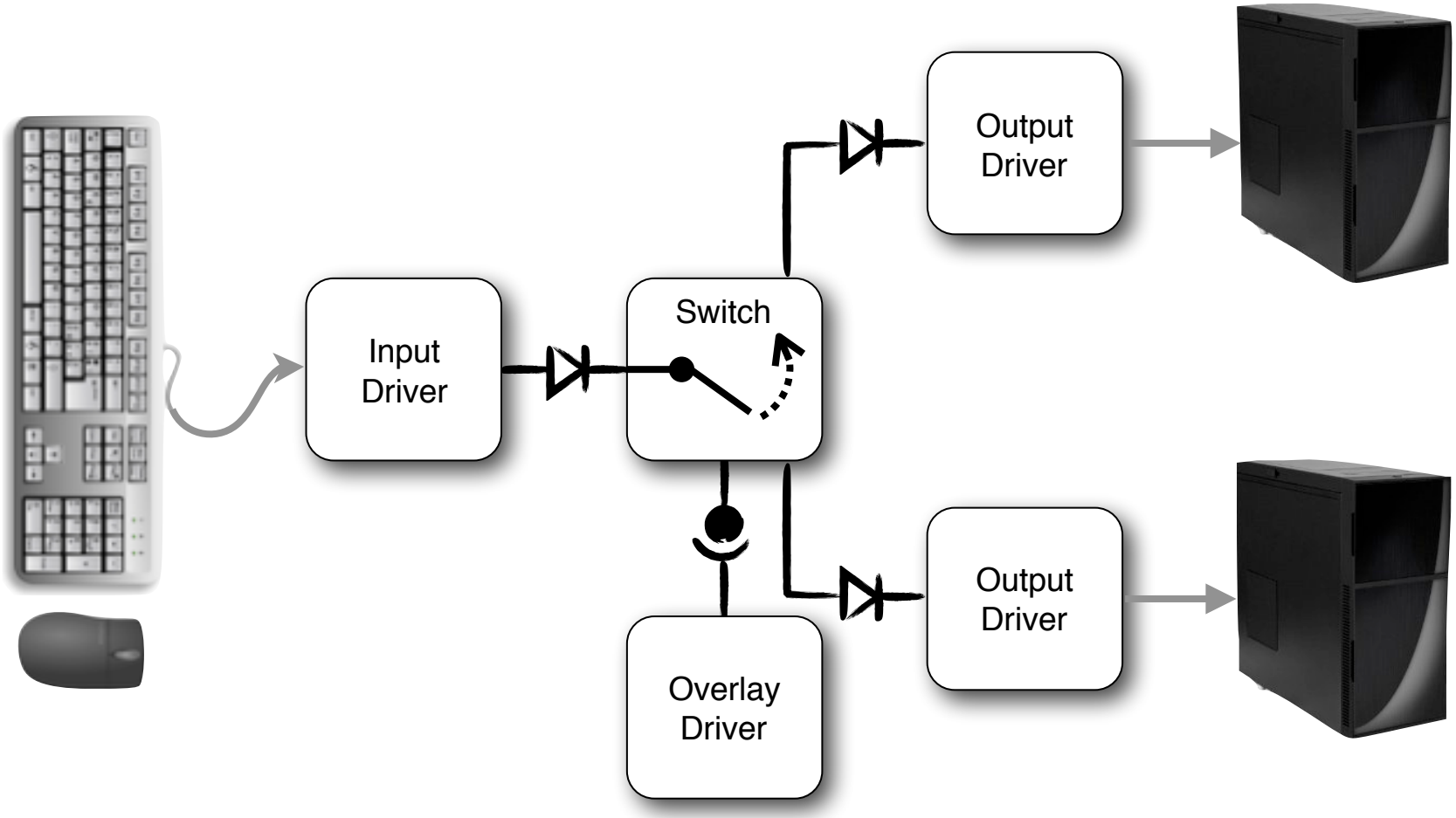


seL4-Based Software Architecture

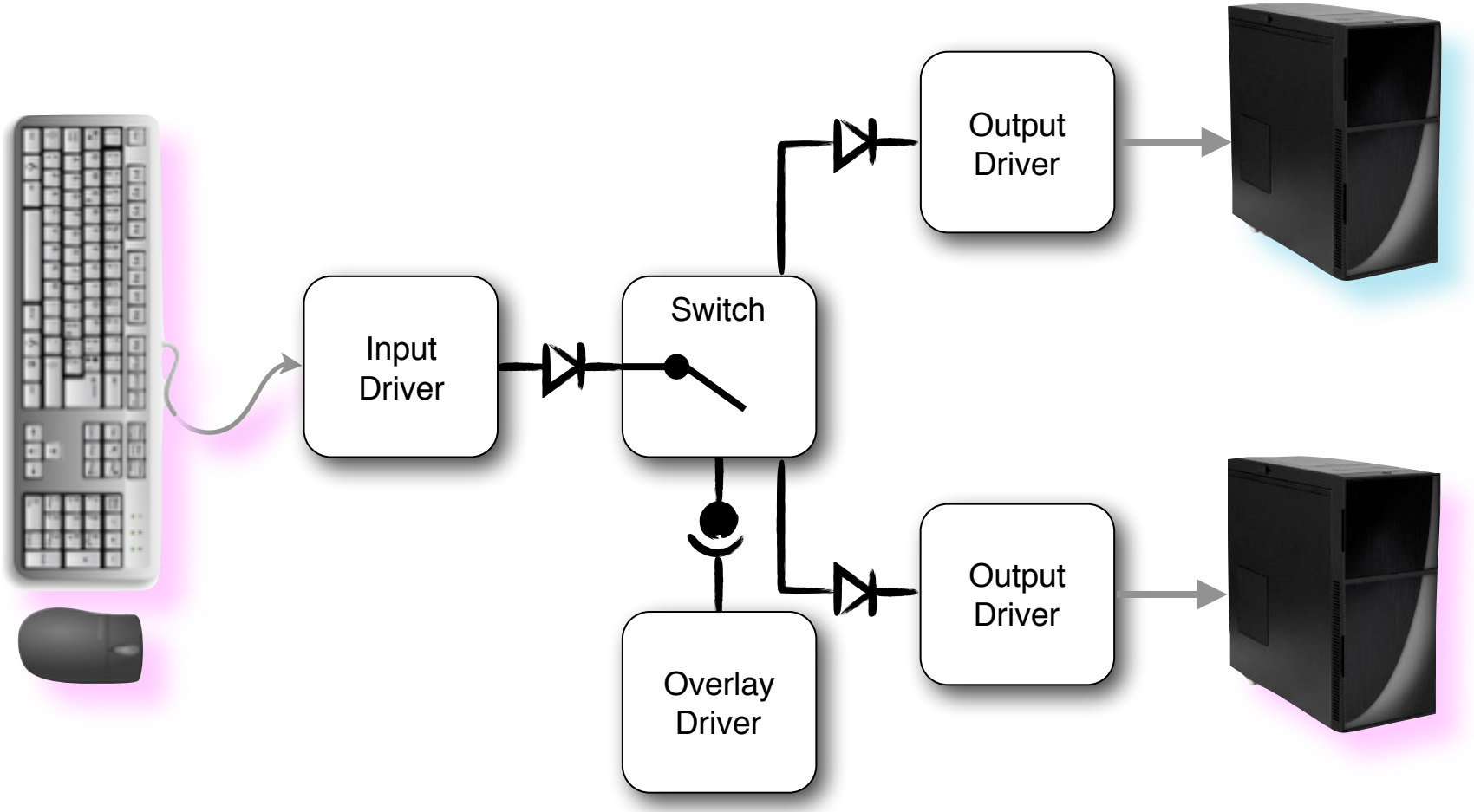


seL4-Based Software Architecture

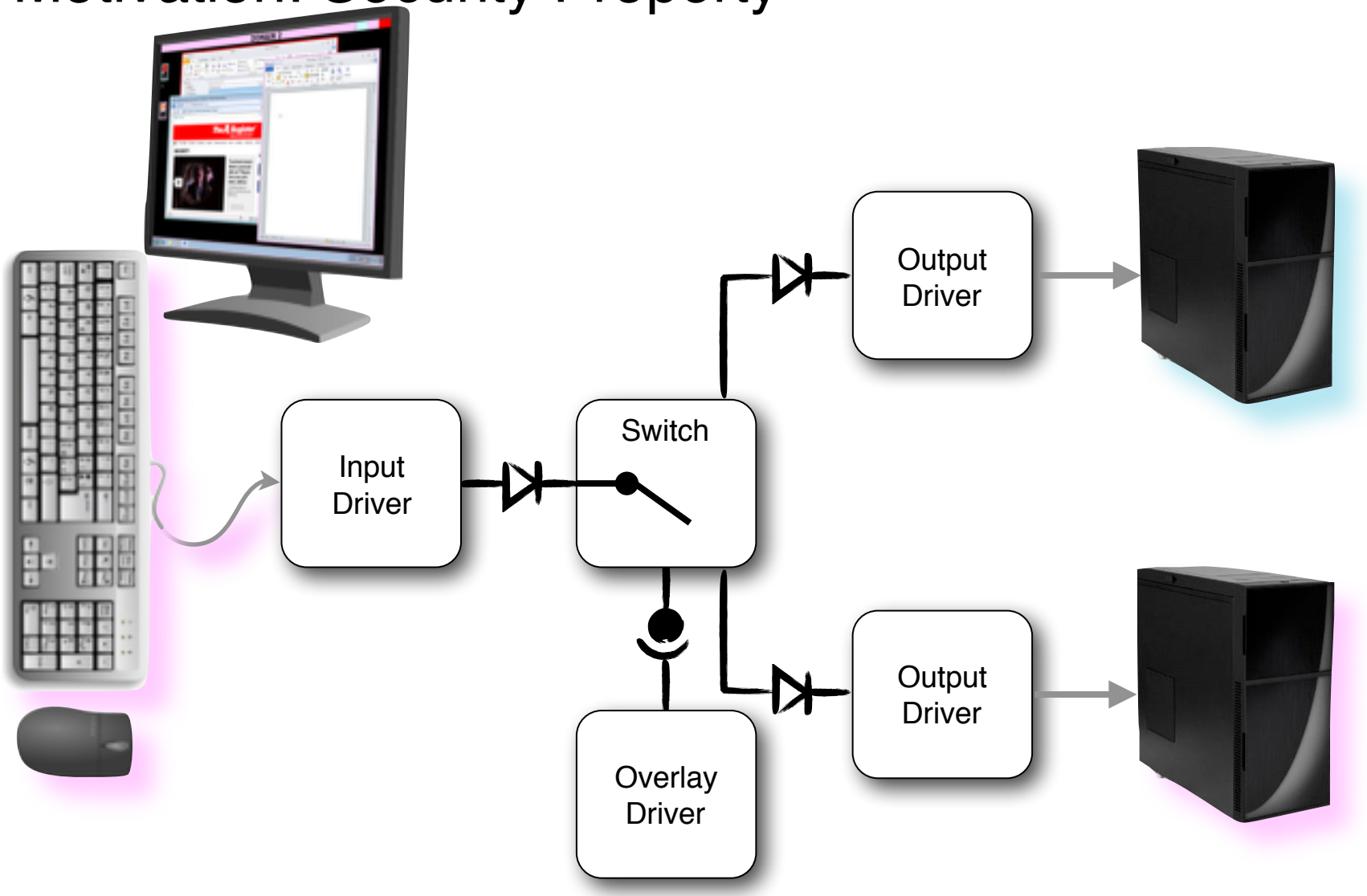
(ignoring device administration and configuration, plus keyboard LED control)



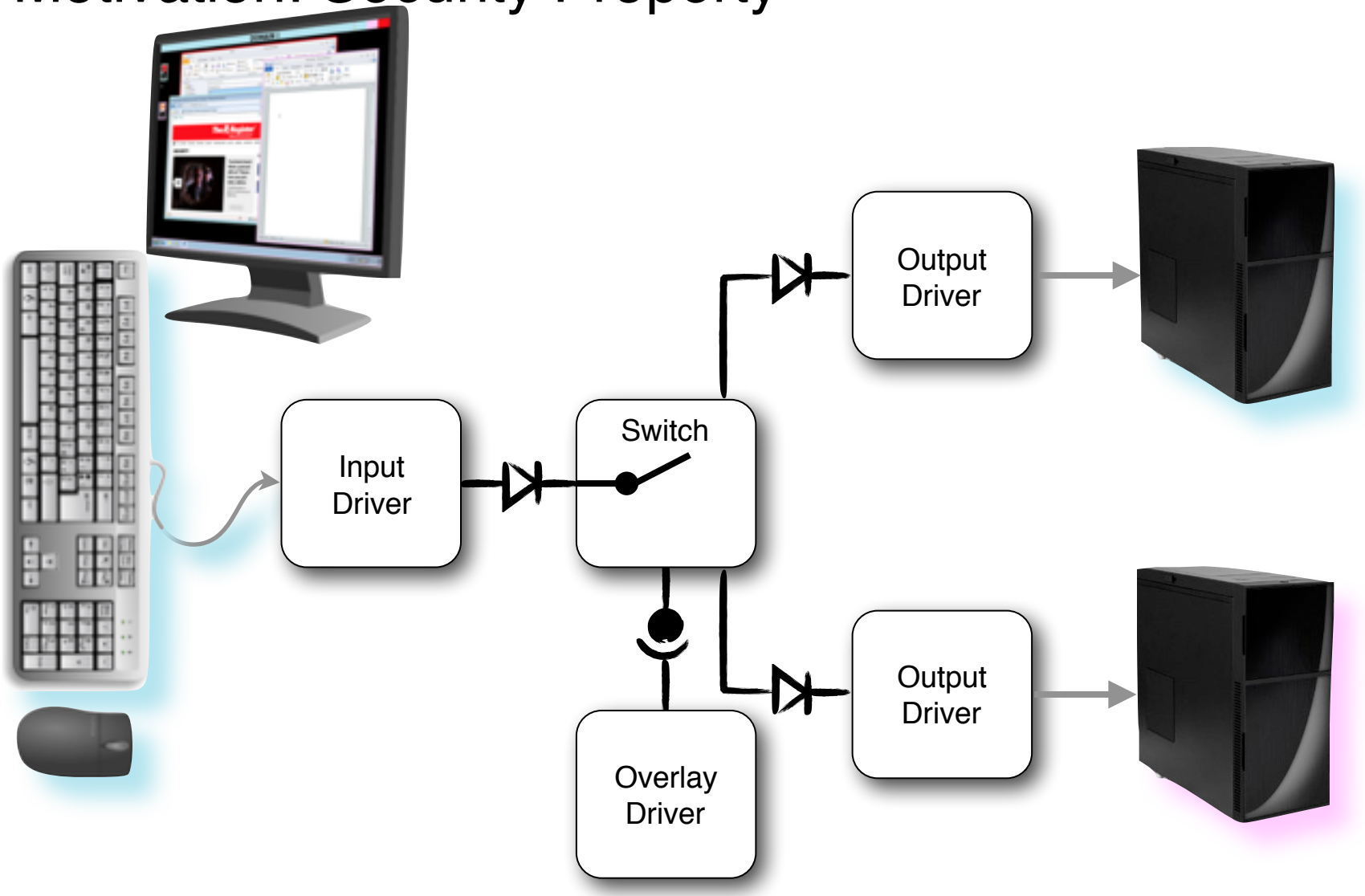
Motivation: Security Property



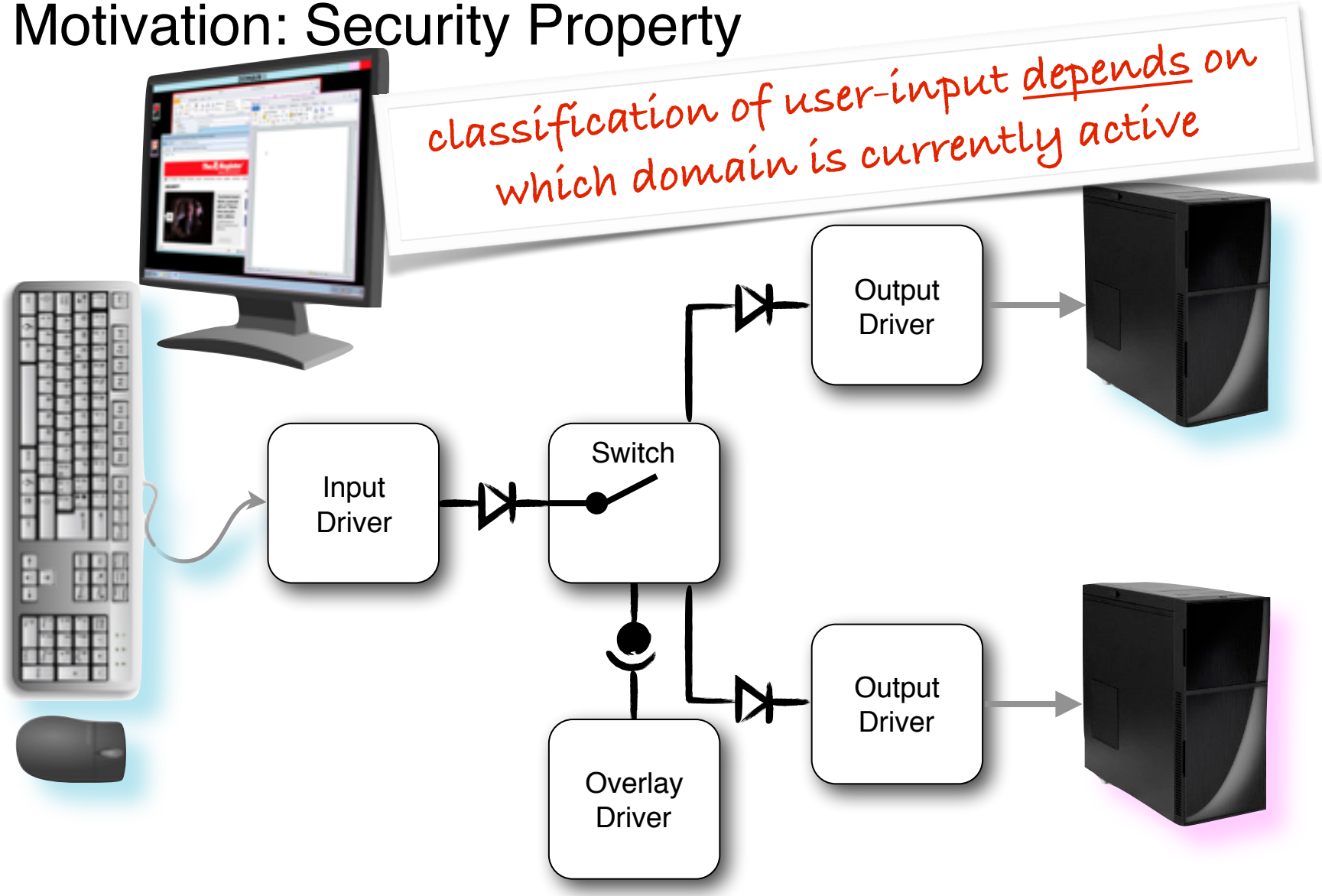
Motivation: Security Property



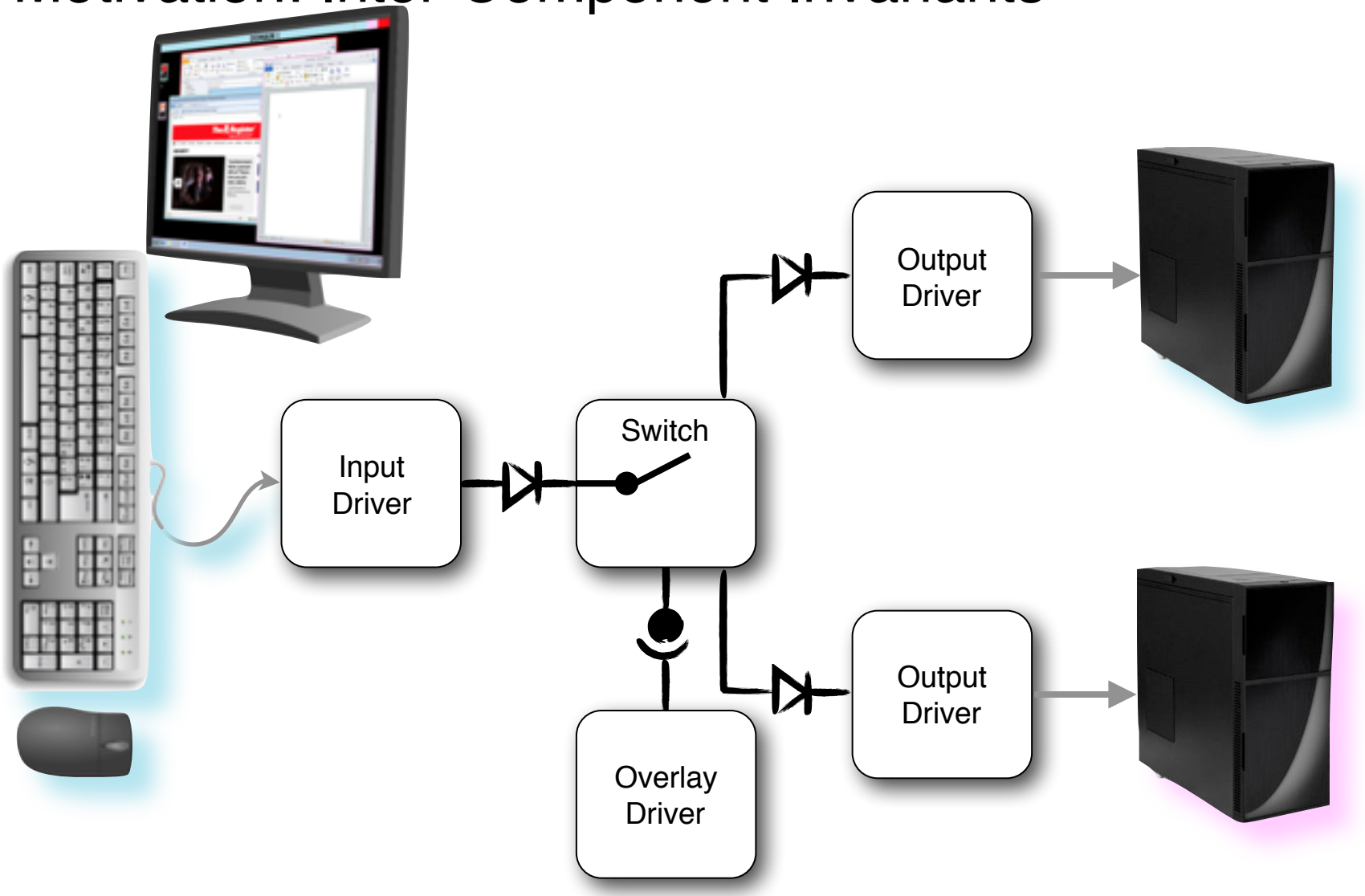
Motivation: Security Property



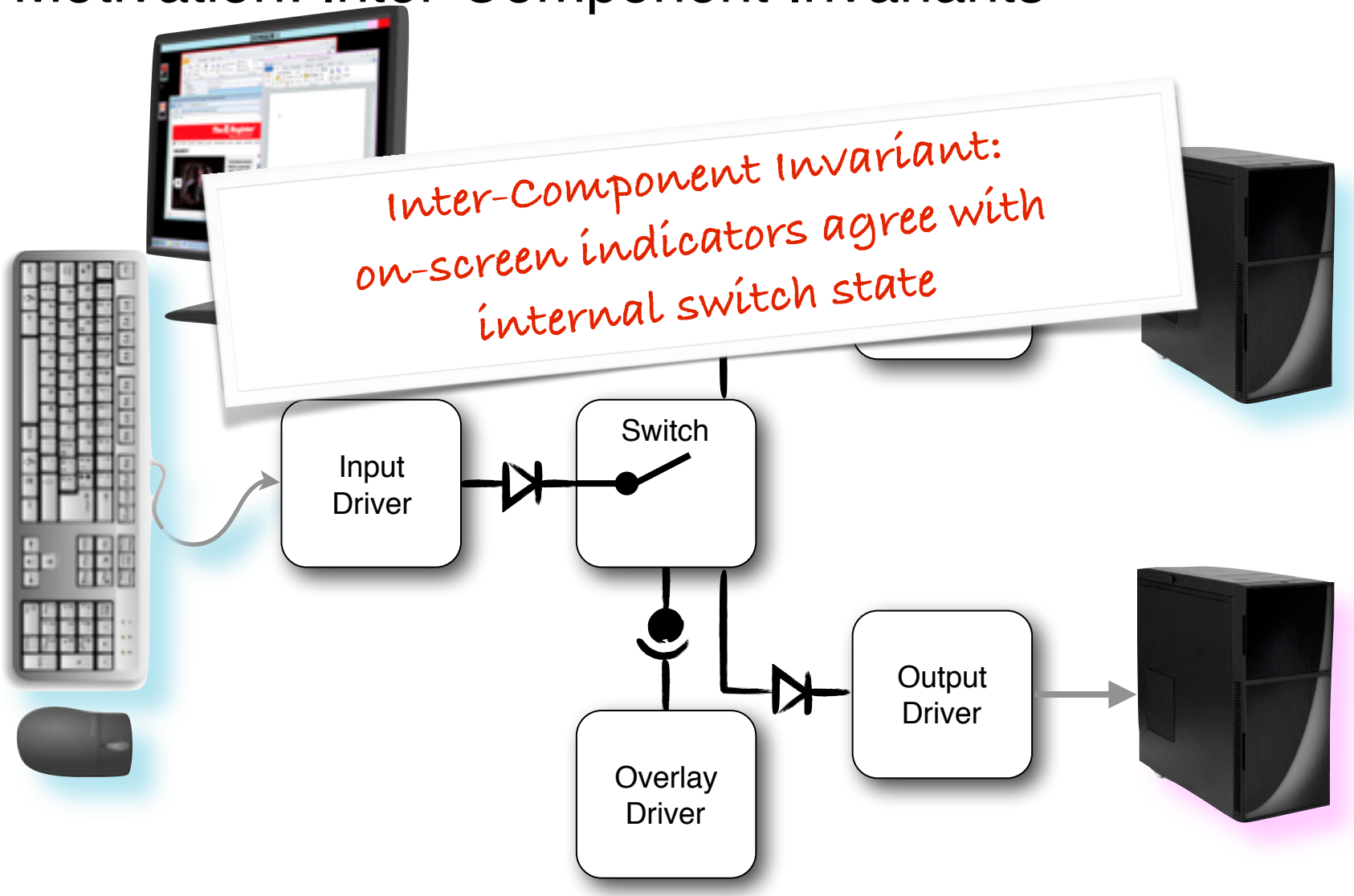
Motivation: Security Property



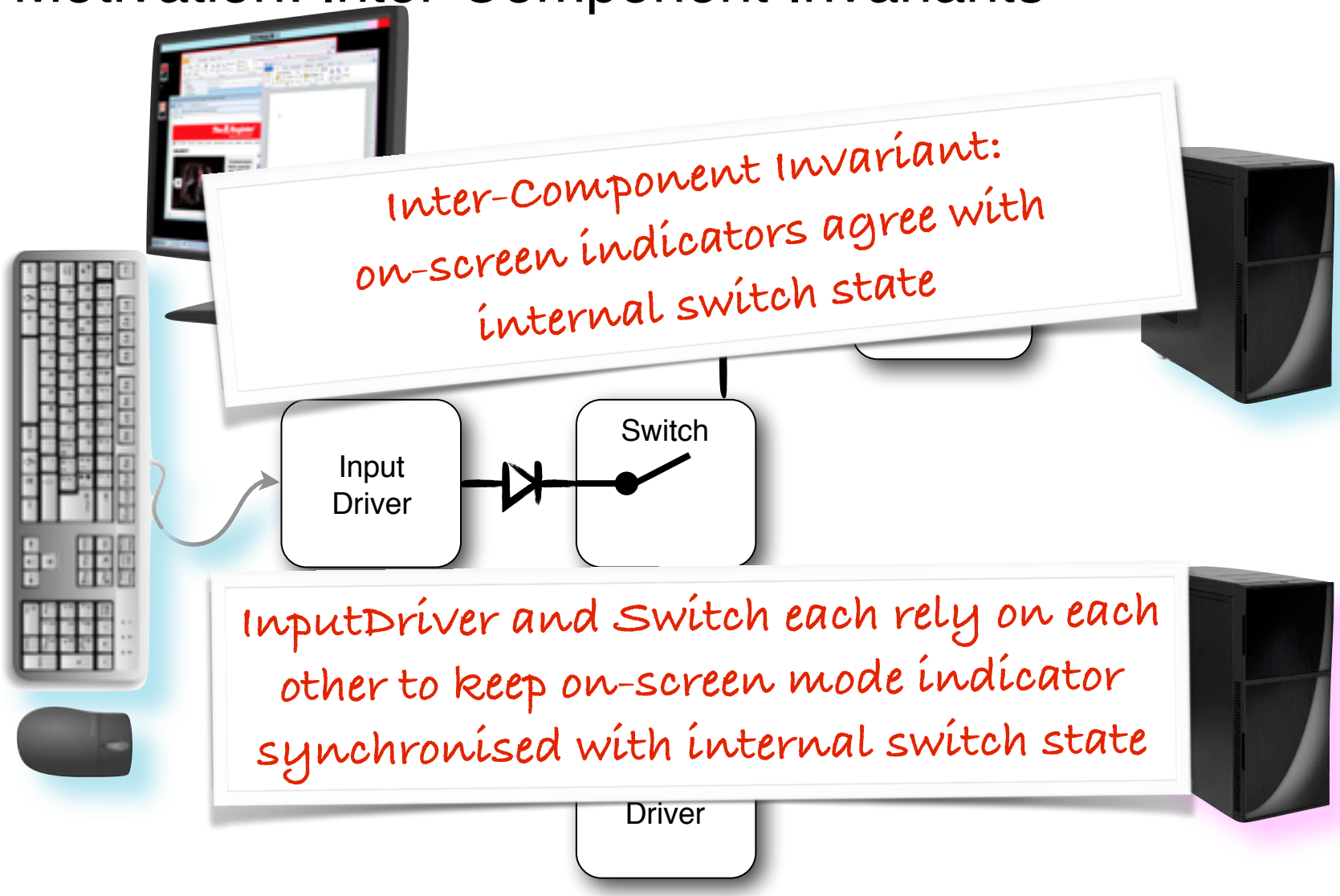
Motivation: Inter-Component Invariants



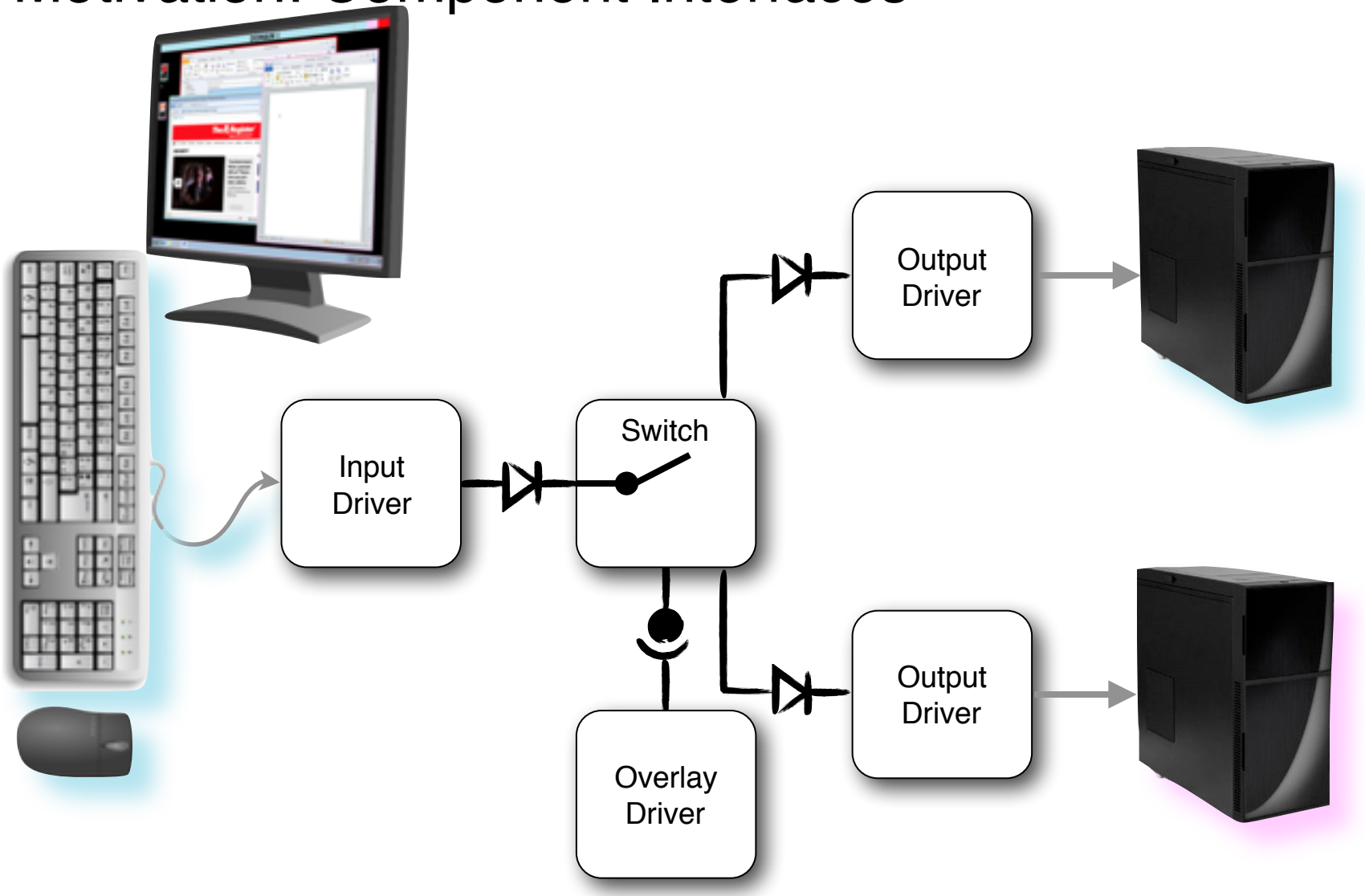
Motivation: Inter-Component Invariants



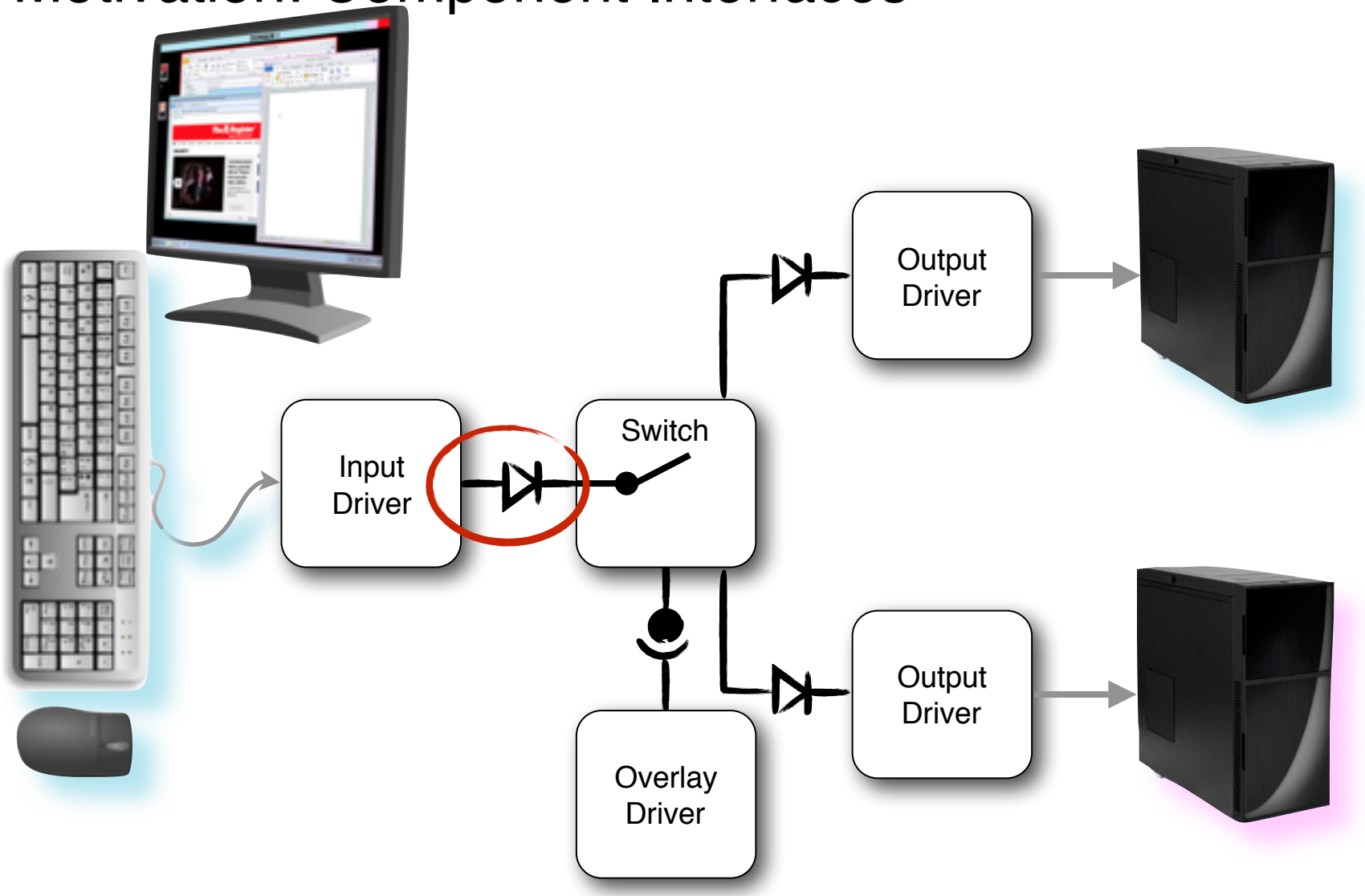
Motivation: Inter-Component Invariants



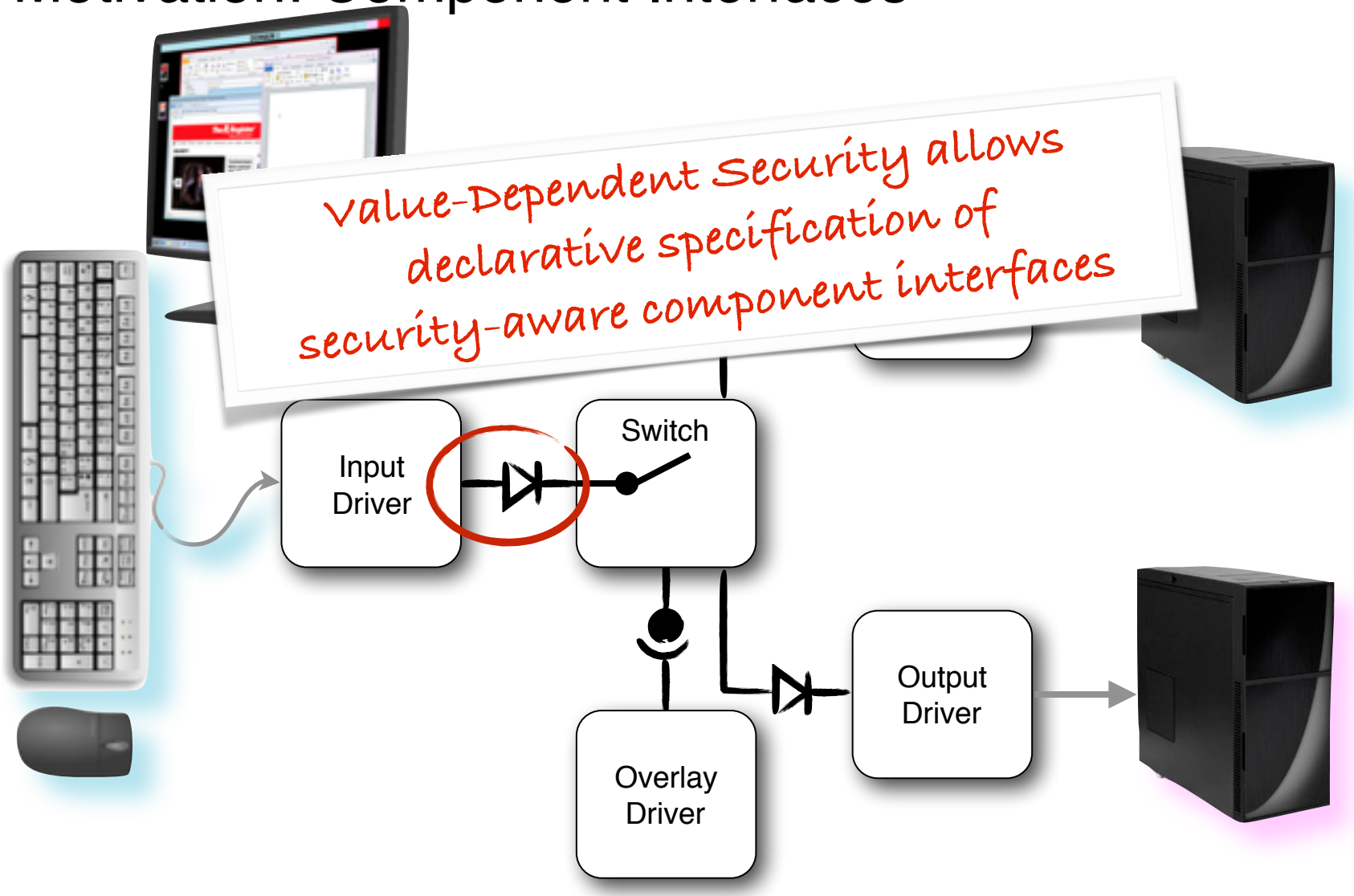
Motivation: Component Interfaces



Motivation: Component Interfaces

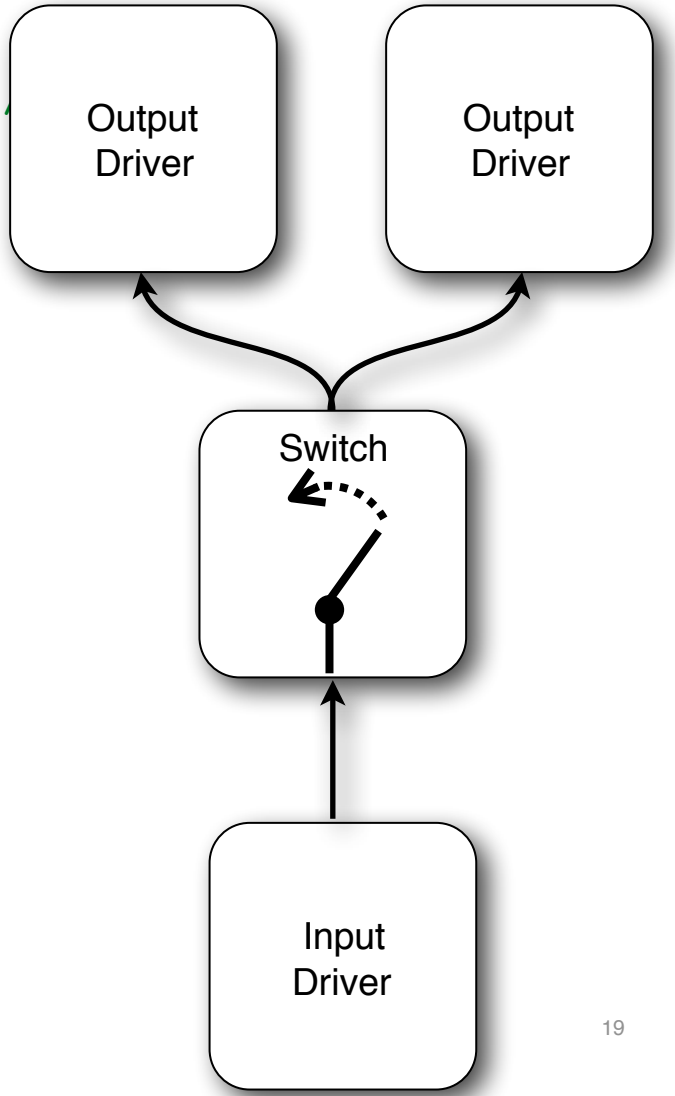


Motivation: Component Interfaces



Motivation: Shared Memory Concurrency

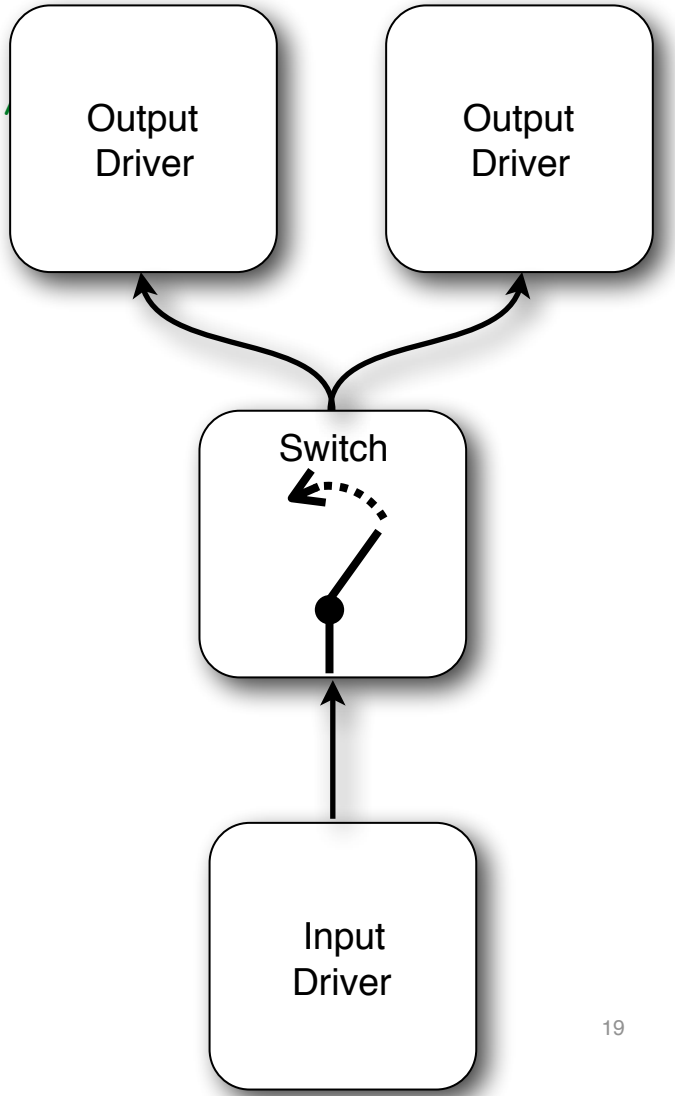
```
temp := buffer;
if sMode == 0 then
  low-var := temp
else
  high-var := temp
endif;
temp := 0
```



Motivation: Shared Memory Concurrency

Switch (idealised, obviously):

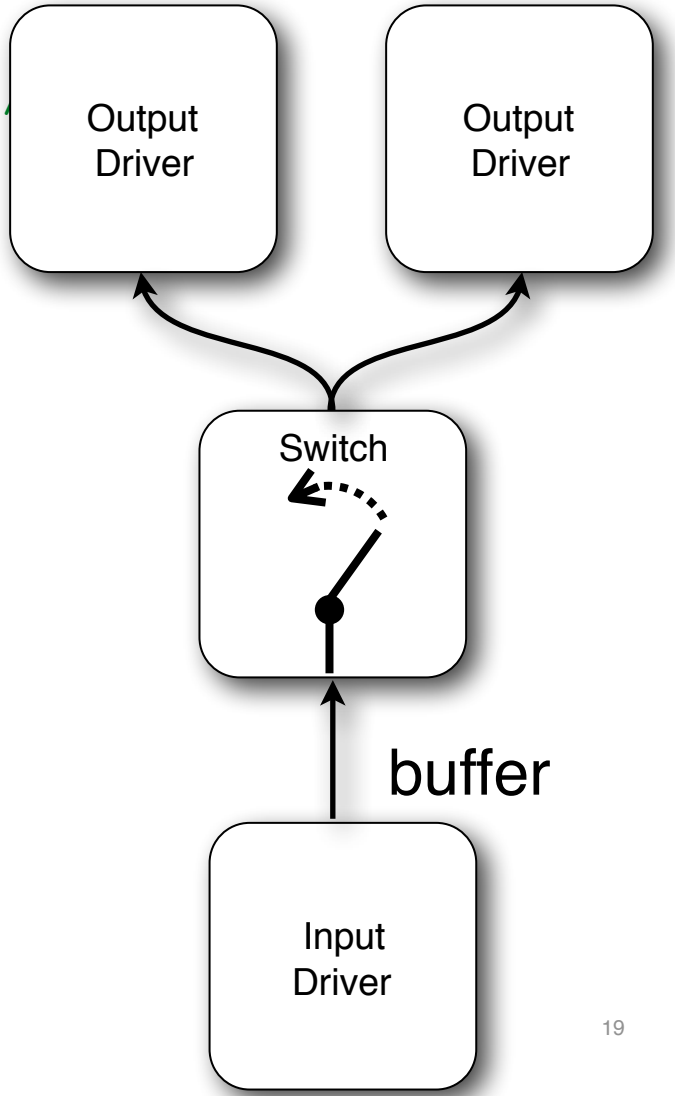
```
temp := buffer;
if sMode == 0 then
  low-var := temp
else
  high-var := temp
endif;
temp := 0
```



Motivation: Shared Memory Concurrency

Switch (idealised, obviously):

```
temp := buffer;
if sMode == 0 then
  low-var := temp
else
  high-var := temp
endif;
temp := 0
```

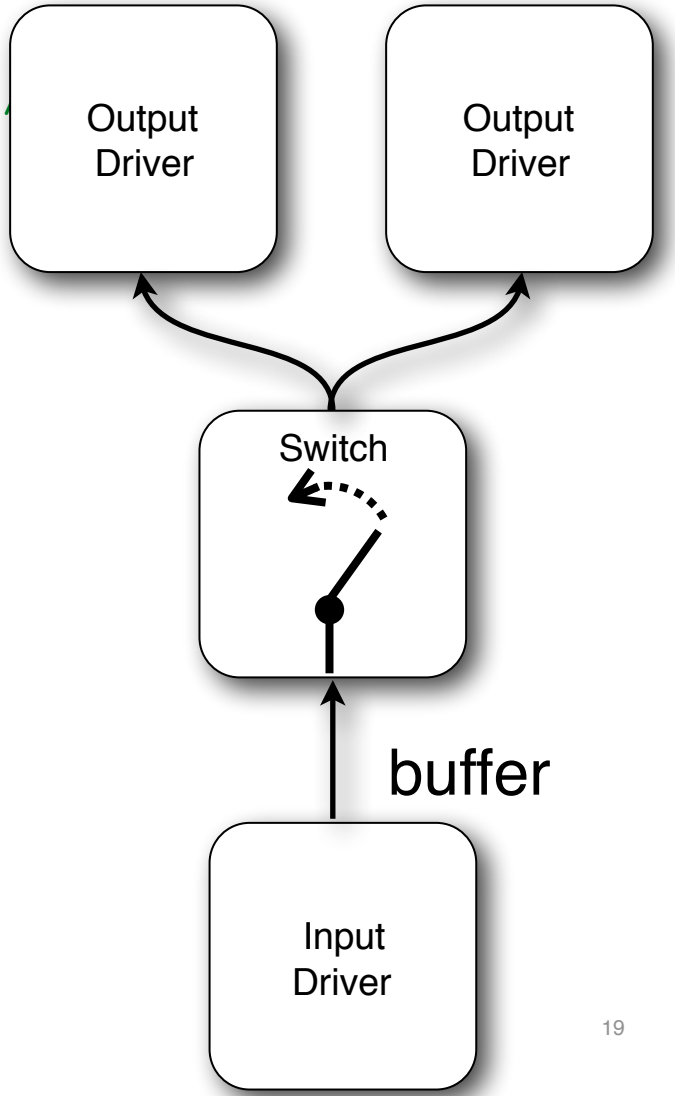


Motivation: Shared Memory Concurrency

Switch (idealised, obviously):

```
temp := buffer;
if sMode == 0 then
  low-var := temp
else
  high-var := temp
endif;
temp := 0
```

sMode *is switch's mode*

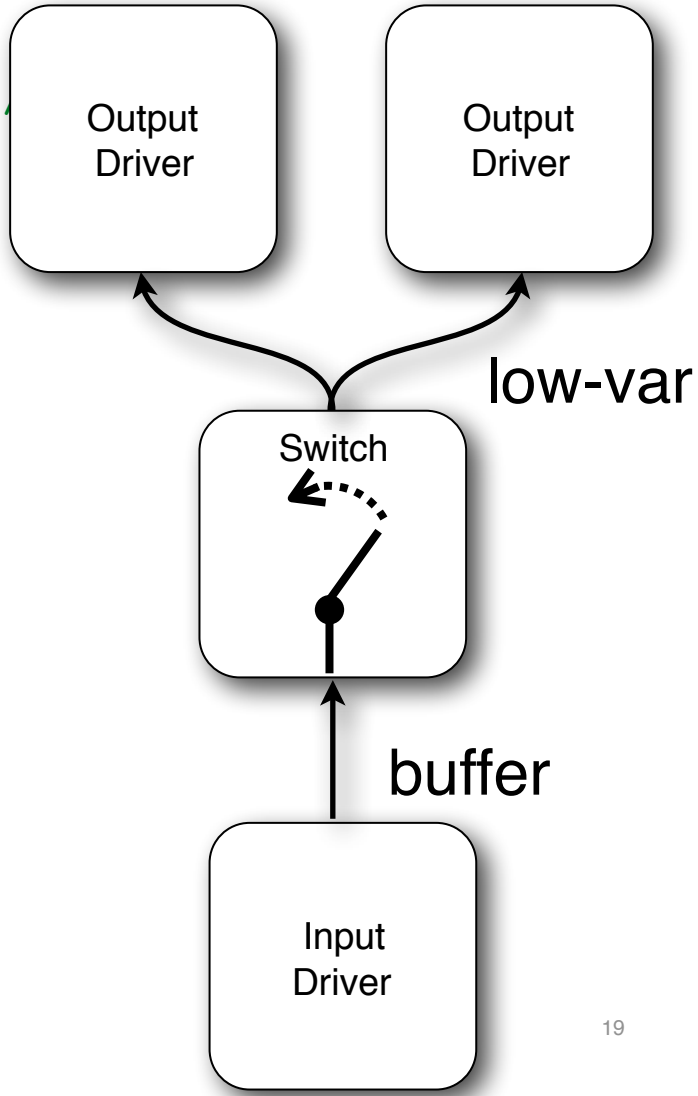


Motivation: Shared Memory Concurrency

Switch (idealised, obviously):

```
temp := buffer;
if sMode == 0 then
  low-var := temp
else
  high-var := temp
endif;
temp := 0
```

sMode is switch's mode

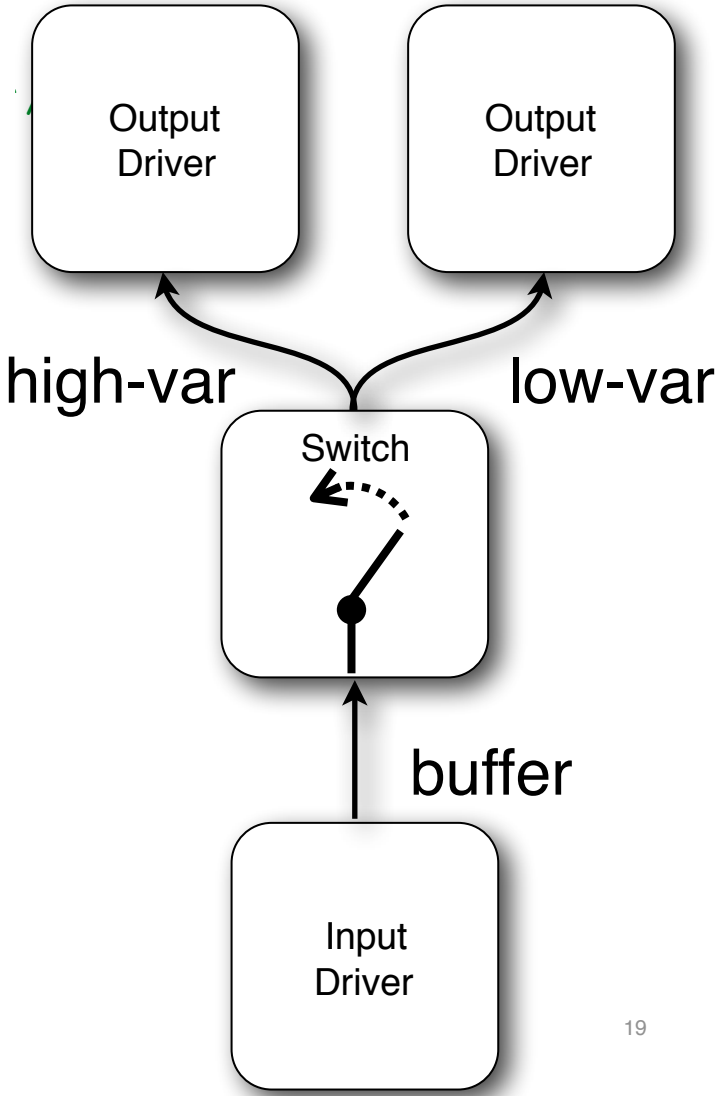


Motivation: Shared Memory Concurrency

Switch (idealised, obviously):

```
temp := buffer;
if sMode == 0 then
  low-var := temp
else
  high-var := temp
endif;
temp := 0
```

sMode is switch's mode

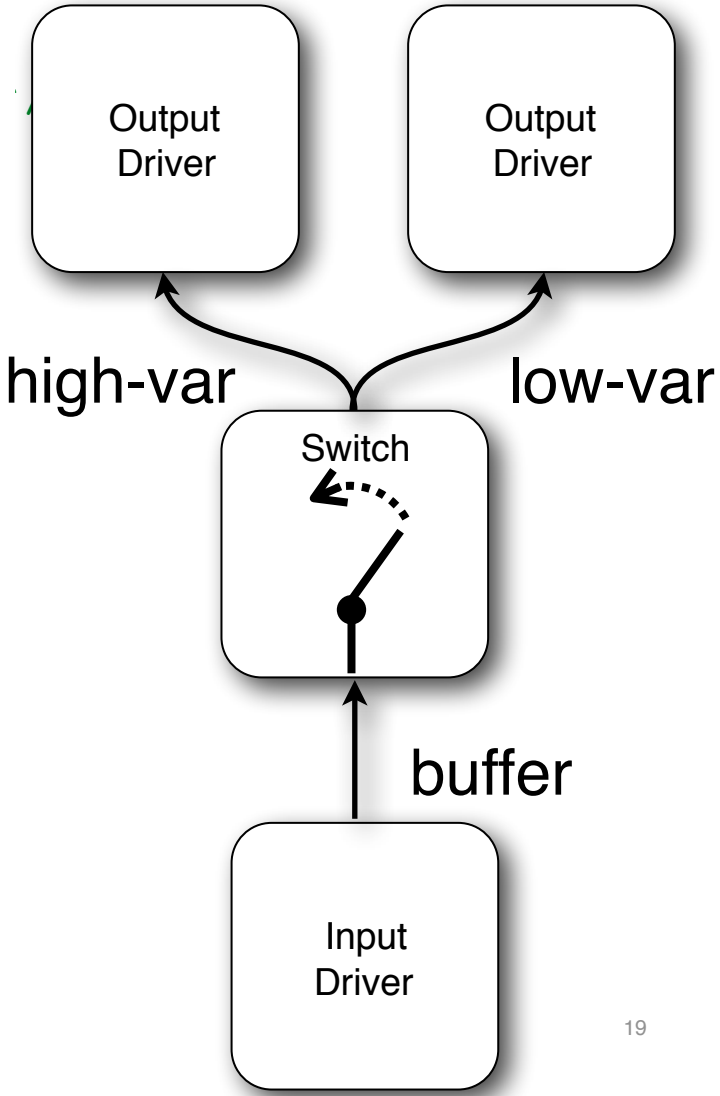


Motivation: Shared Memory Concurrency

Switch (idealised, obviously):

```
temp := buffer;
if sMode == 0 then
  low-var := temp
else
  high-var := temp
endif;
temp := 0
```

sMode *is switch's mode*



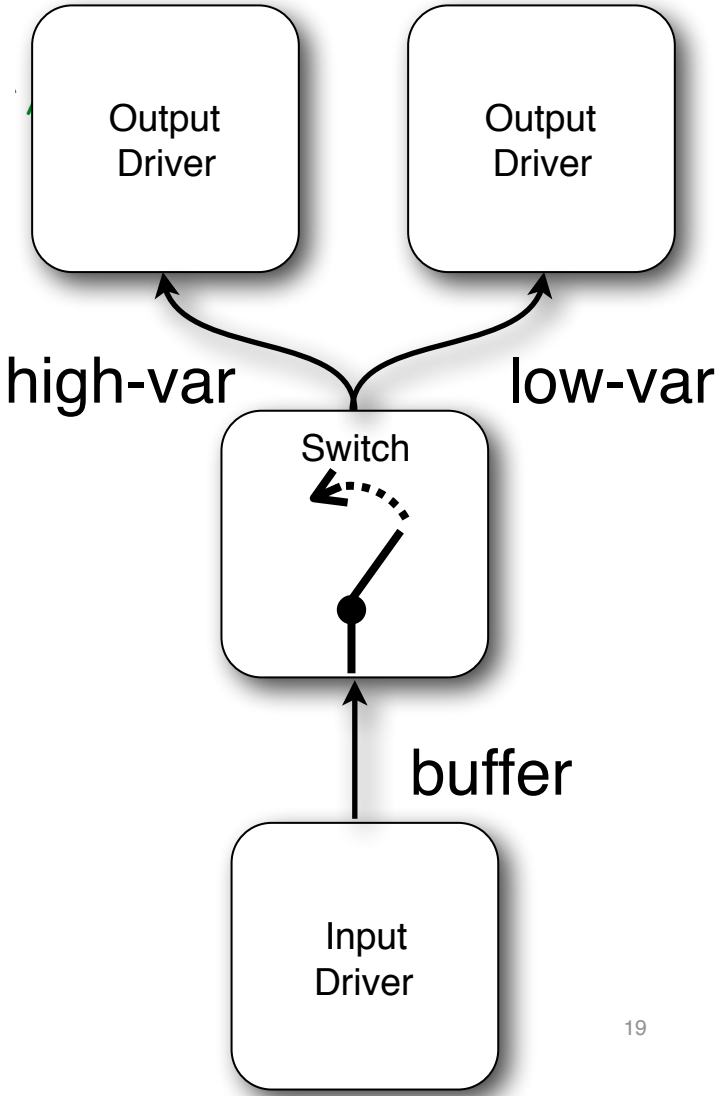
Motivation: Shared Memory Concurrency

Switch (idealised, obviously):

```
temp := buffer;
if sMode == 0 then
  low-var := temp
else
  high-var := temp
endif;
temp := 0
```

what if somebody else modifies **sMode** here?

sMode *is switch's mode*

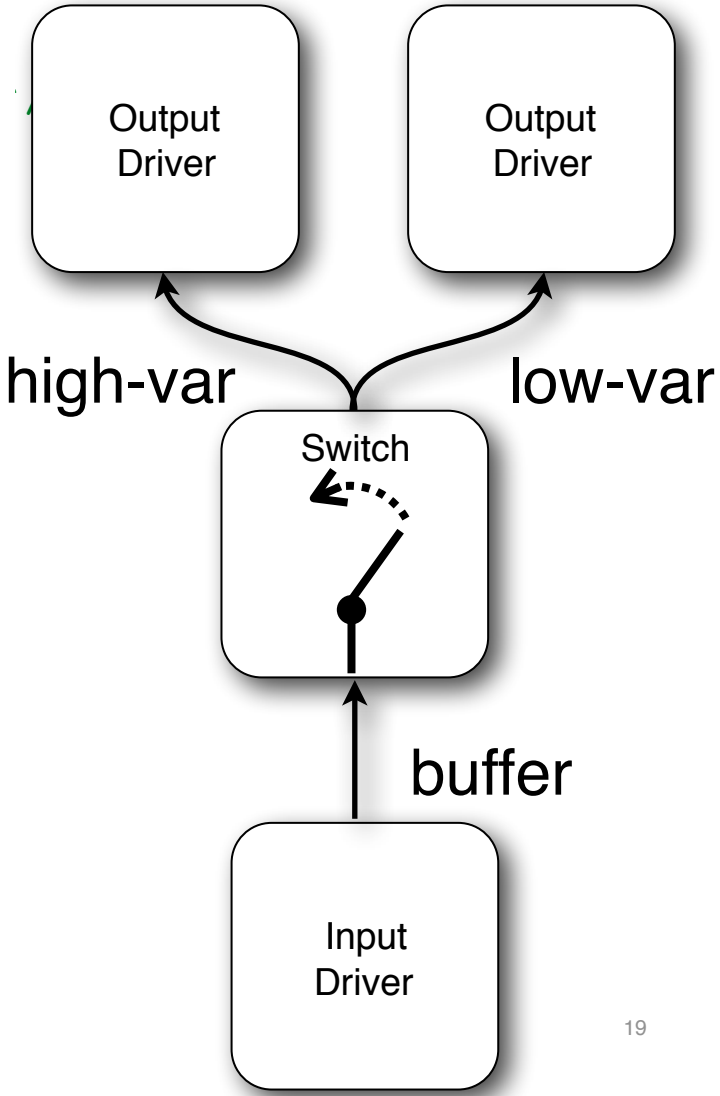


Motivation: Shared Memory Concurrency

Switch (idealised, obviously):

```
temp := buffer;
if sMode == 0 then
  low-var := temp
else
  high-var := temp
endif;
temp := 0
```

sMode *is switch's mode*



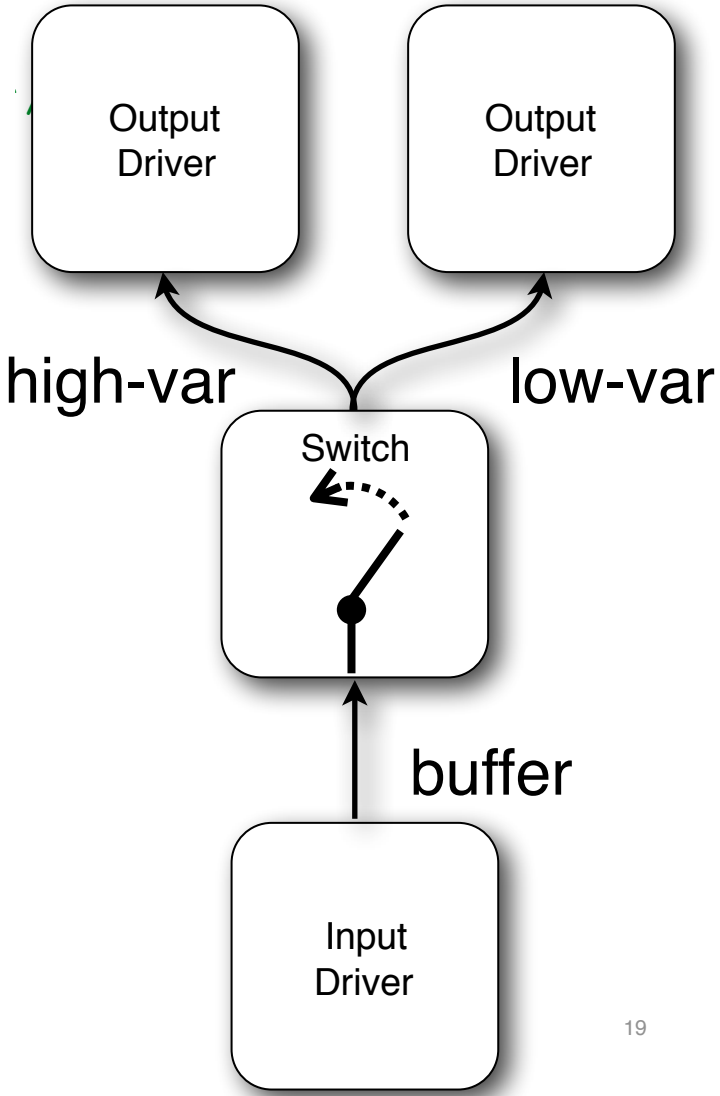
Motivation: Shared Memory Concurrency

Switch (idealised, obviously):

```
temp := buffer;
if sMode == 0 then
  low-var := temp
else
  high-var := temp
endif;
temp := 0
```

what if **temp** holds High data here and somebody else reads it?

sMode *is switch's mode*

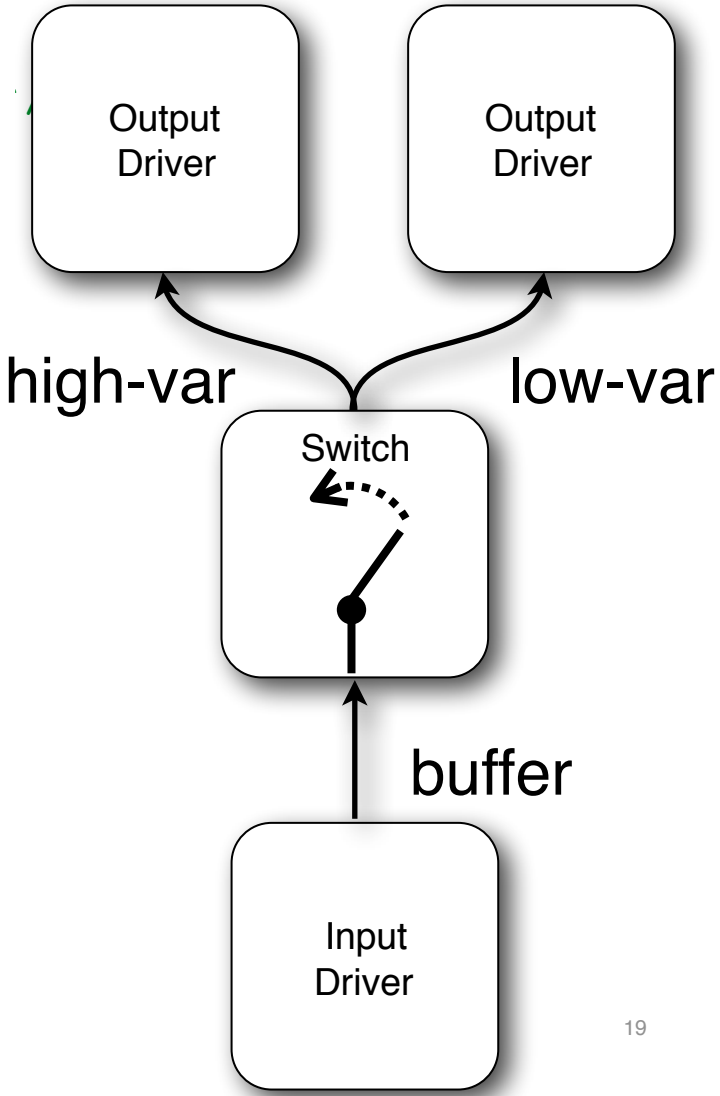


Motivation: Shared Memory Concurrency

Switch (idealised, obviously):

```
temp := buffer;
if sMode == 0 then
  low-var := temp
else
  high-var := temp
endif;
temp := 0
```

sMode *is switch's mode*

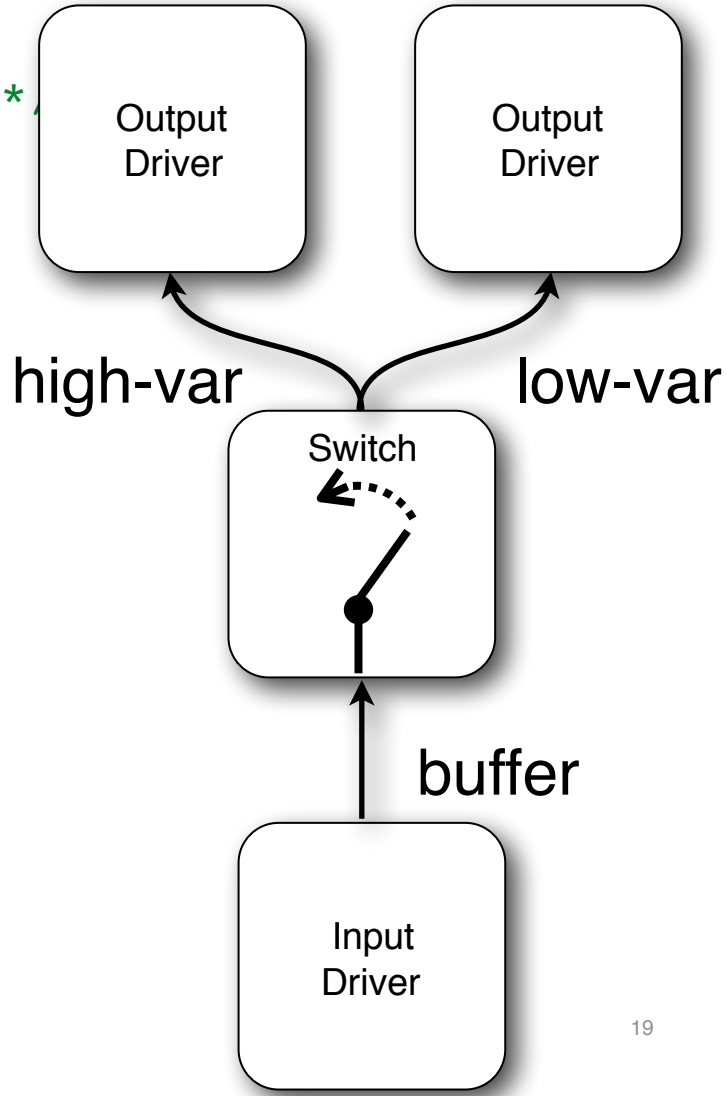


Motivation: Shared Memory Concurrency

Switch (idealised, obviously):

```
/* {dMode,sMode,temp} += AsmNoRW */
/* acquire (dMode == sMode) */
temp := buffer;
if sMode == 0 then
  low-var := temp
else
  high-var := temp
endif;
temp := 0
```

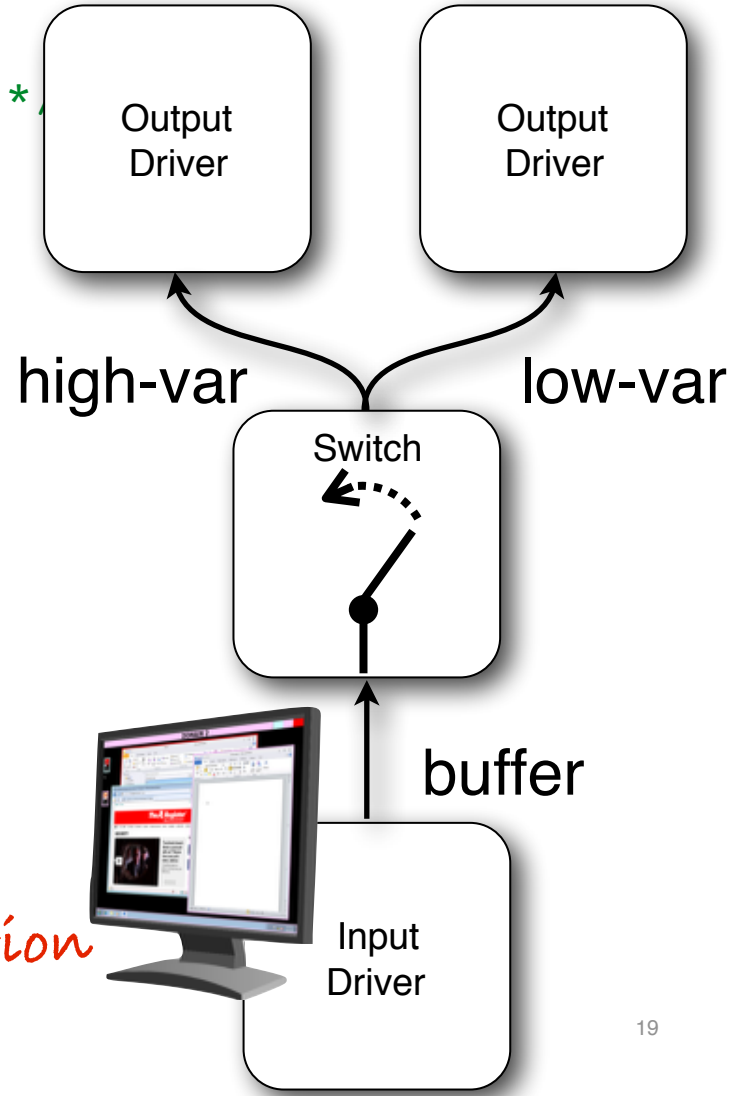
sMode *is switch's mode*



Motivation: Shared Memory Concurrency

Switch (idealised, obviously):

```
/* {dMode,sMode,temp} += AsmNoRW */
/* acquire (dMode == sMode) */
temp := buffer;
if sMode == 0 then
  low-var := temp
else
  high-var := temp
endif;
temp := 0
```



sMode *is switch's mode*

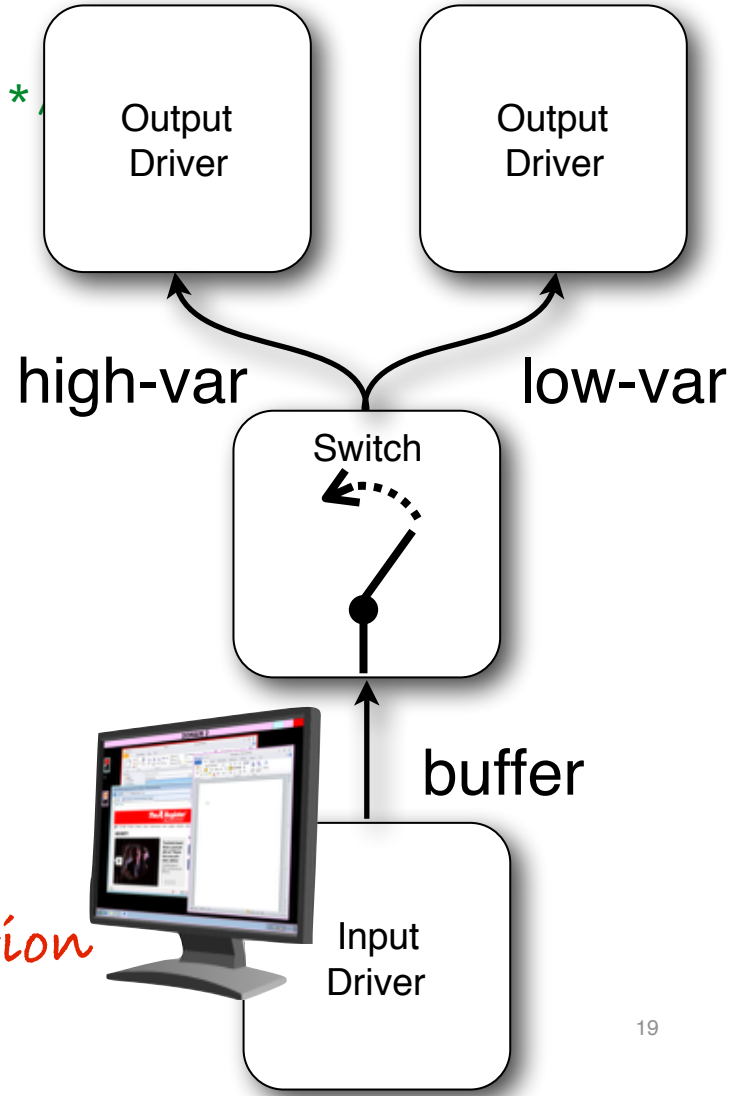
dMode *defines* buffer*'s classification*

Motivation: Shared Memory Concurrency

Switch (idealised, obviously):

```
/* {dMode,sMode,temp} += AsmNoRW */
/* acquire (dMode == sMode) */
temp := buffer;
if sMode == 0 then
  low-var := temp
else
  high-var := temp
endif;
temp := 0
```

what if **dMode** and **sMode** disagree?



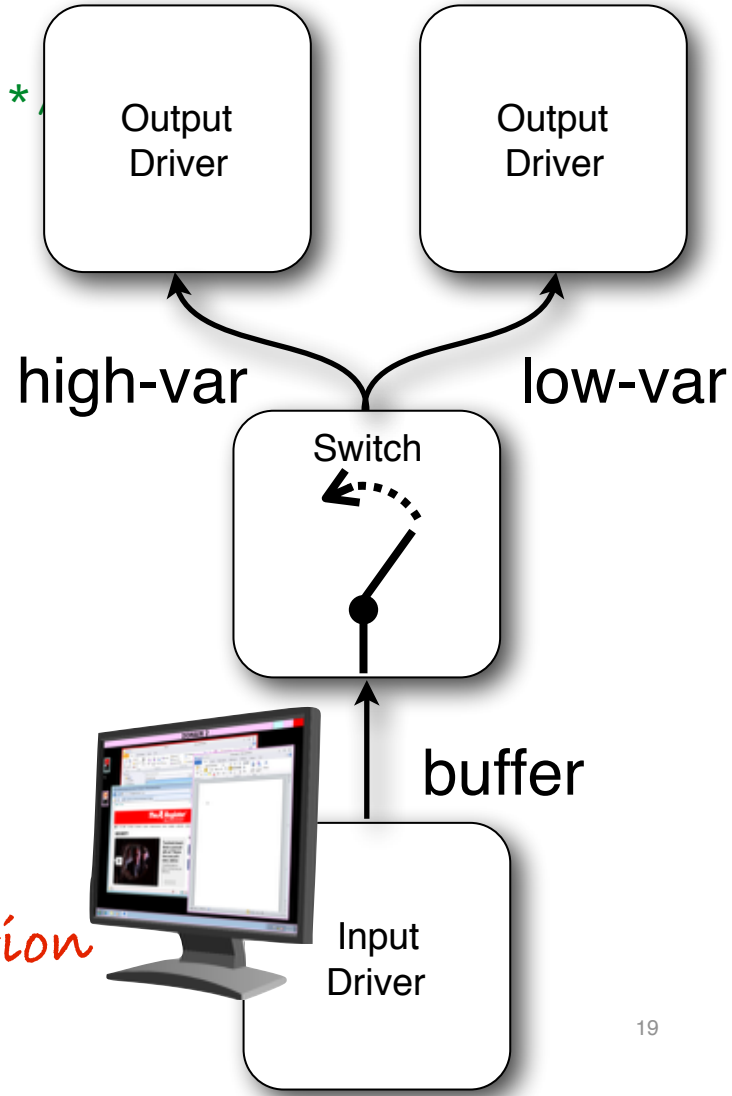
sMode *is switch's mode*

dMode *defines* buffer*'s classification*

Motivation: Shared Memory Concurrency

Switch (idealised, obviously):

```
/* {dMode,sMode,temp} += AsmNoRW */
/* acquire (dMode == sMode) */
temp := buffer;
if sMode == 0 then
  low-var := temp
else
  high-var := temp
endif;
temp := 0
```



sMode *is switch's mode*

dMode *defines* buffer*'s classification*

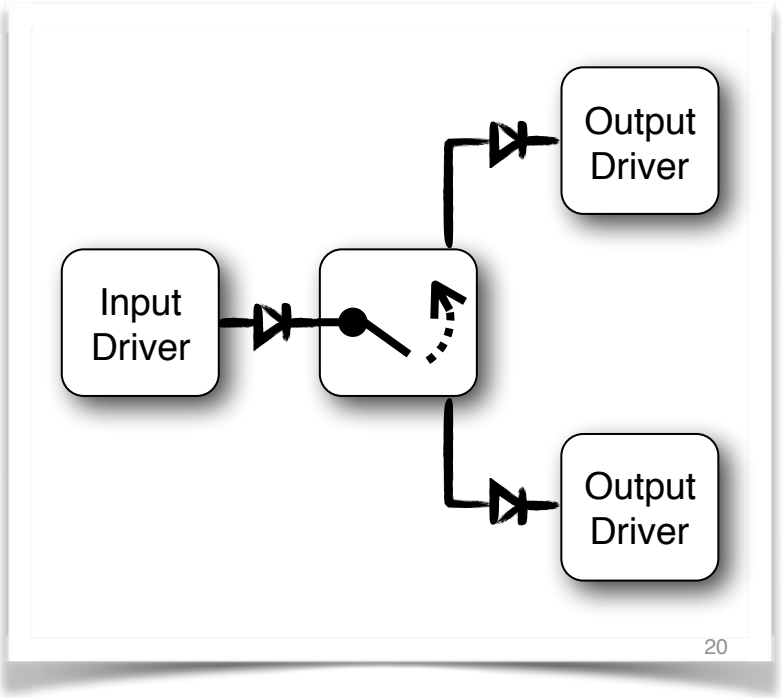
Verification Framework: Requirements

Verification Framework: Requirements

Compositional Security Property

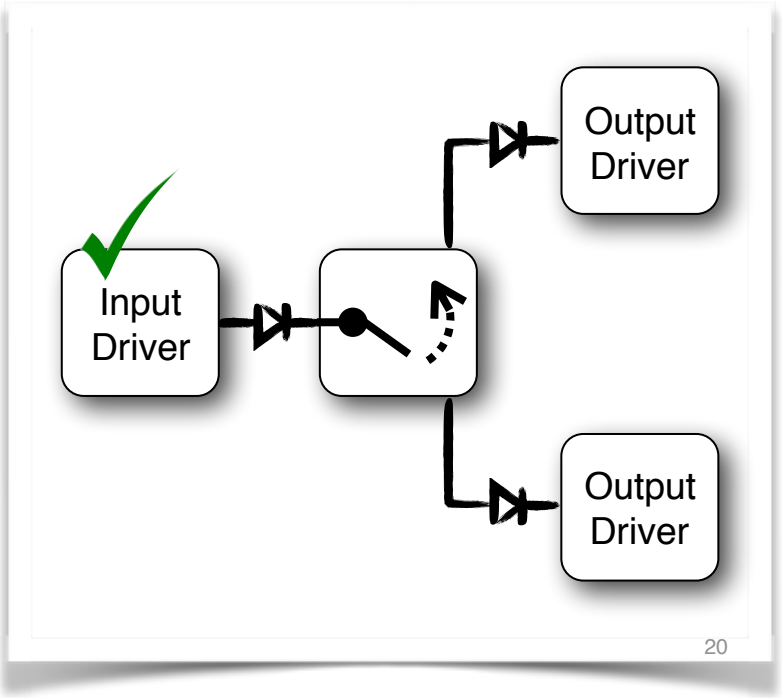
Verification Framework: Requirements

Compositional Security Property



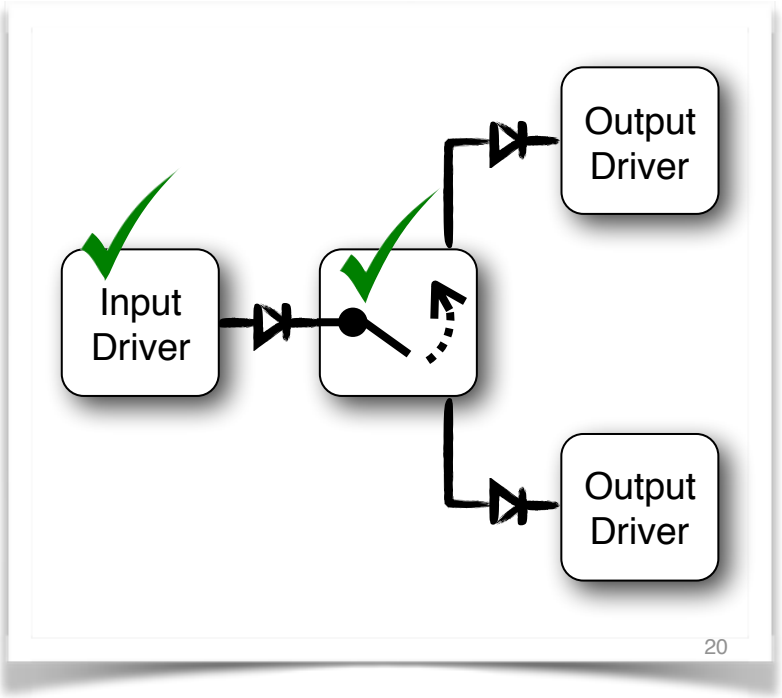
Verification Framework: Requirements

Compositional Security Property



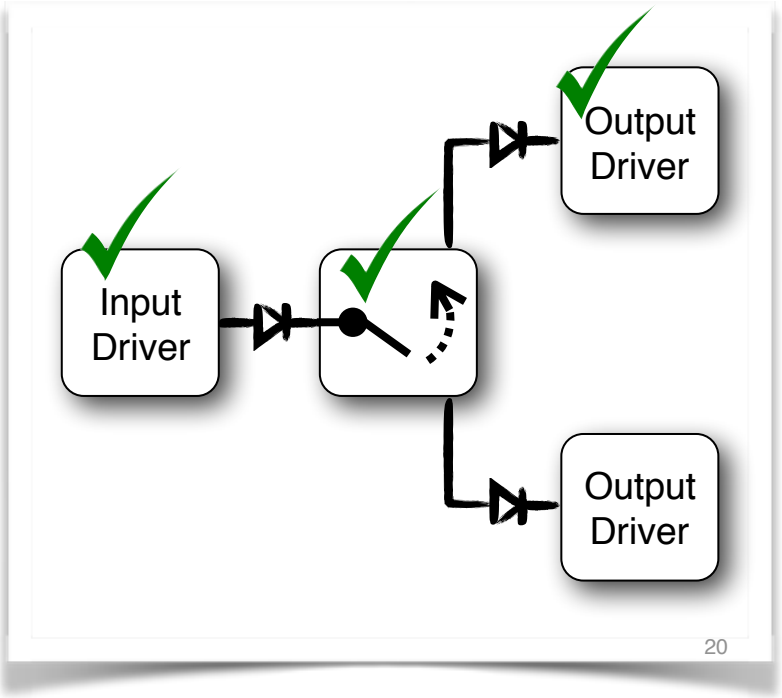
Verification Framework: Requirements

Compositional Security Property



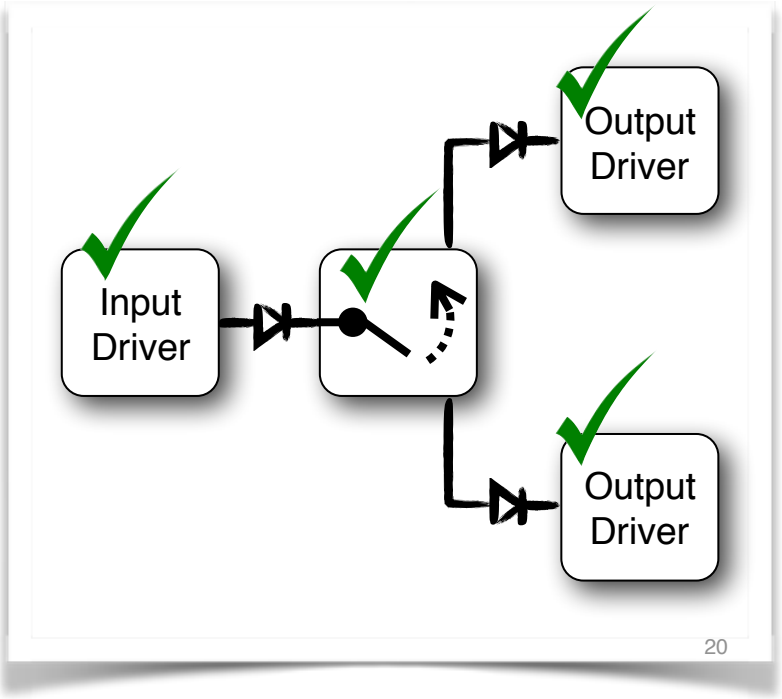
Verification Framework: Requirements

Compositional Security Property



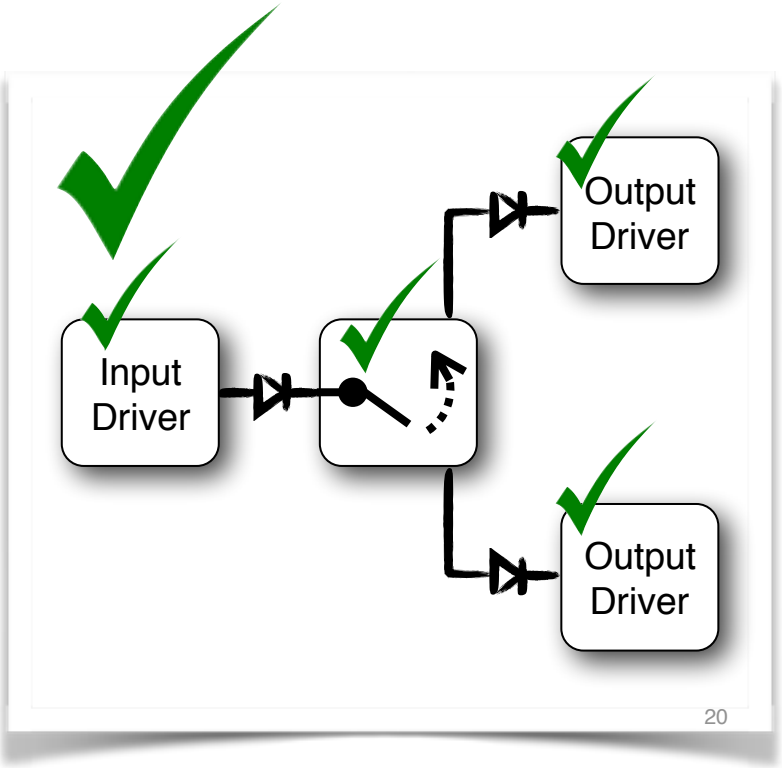
Verification Framework: Requirements

Compositional Security Property



Verification Framework: Requirements

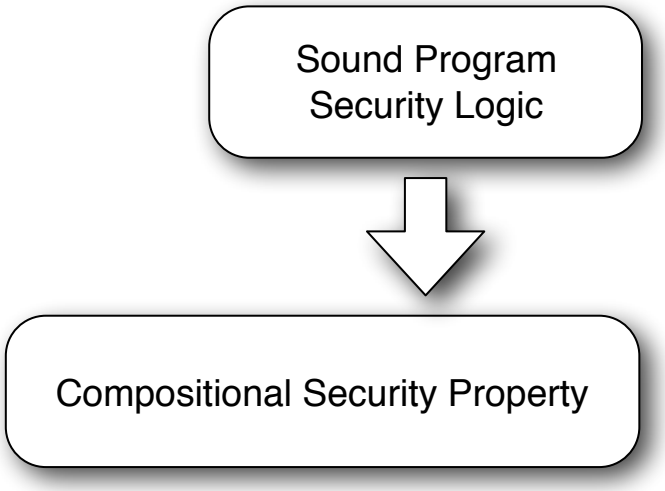
Compositional Security Property



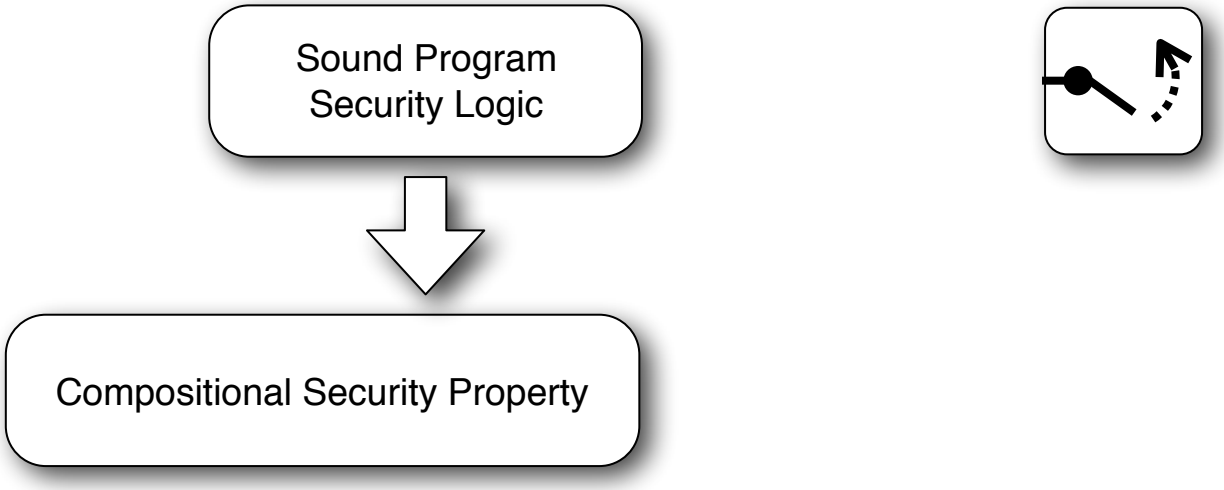
Verification Framework: Requirements

Compositional Security Property

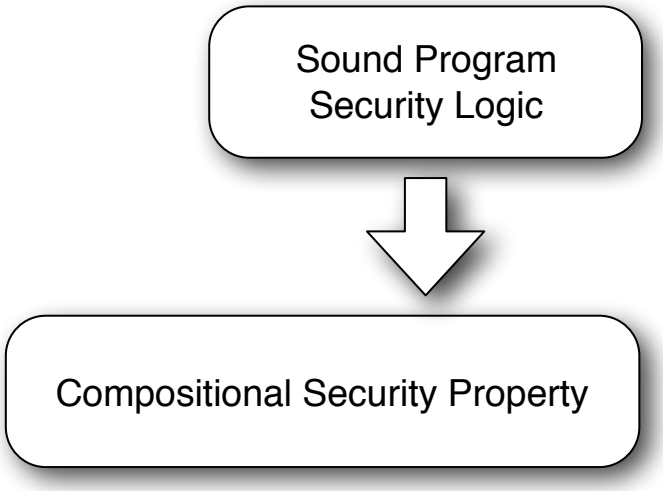
Verification Framework: Requirements



Verification Framework: Requirements

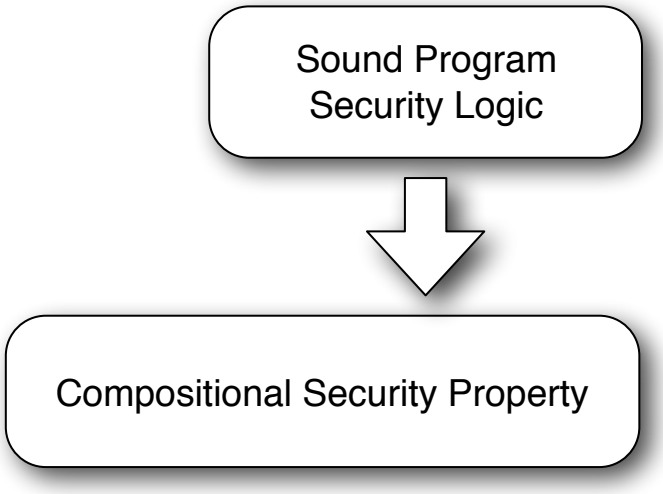


Verification Framework: Requirements



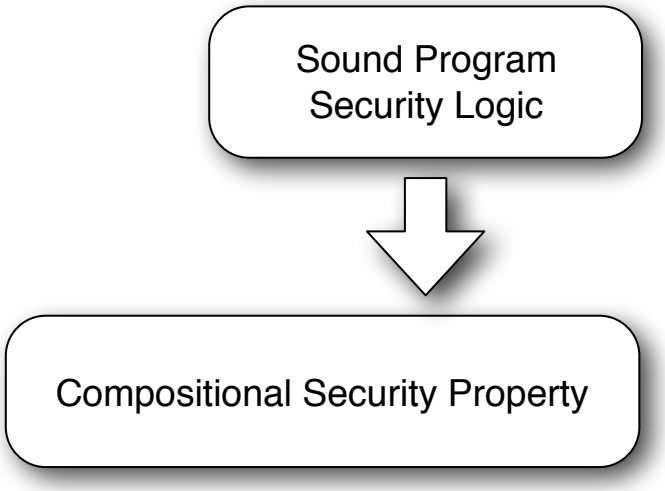

```
skip   /* control +=_m AsmNoW */;
skip   /* temp +=_m AsmNoRW */;
temp := buffer;
if control == 0 then
  low-var := temp
else
  high-var := temp
endif;
temp := 0;
skip   /* temp -=_m AsmNoRW */
```

Verification Framework: Requirements

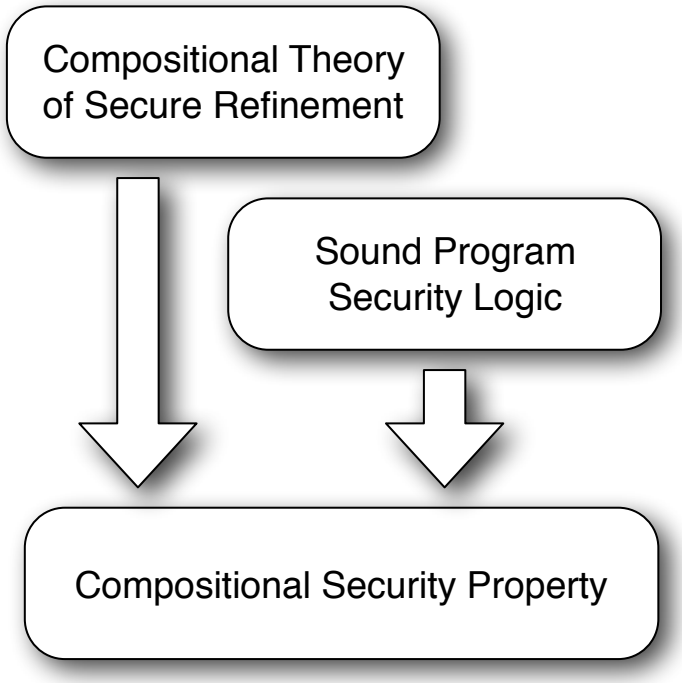



```
skip   /* control +=_m AsmNoW */;
skip   /* temp +=_m AsmNoRW */;
temp := buffer;
if control == 0 then
  low-var := temp
else
  high-var := temp
endif;
temp := 0;
skip   /* temp -=_m AsmNoRW */
```

Verification Framework: Requirements

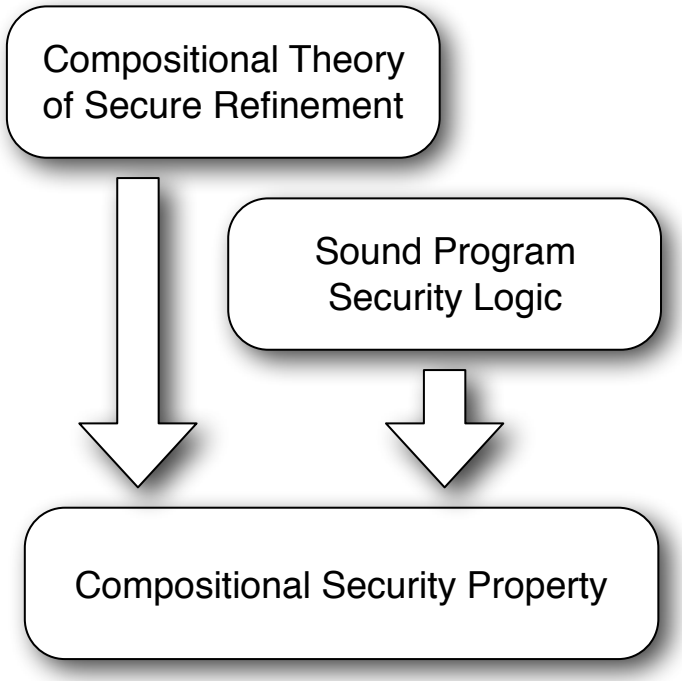


Verification Framework: Requirements

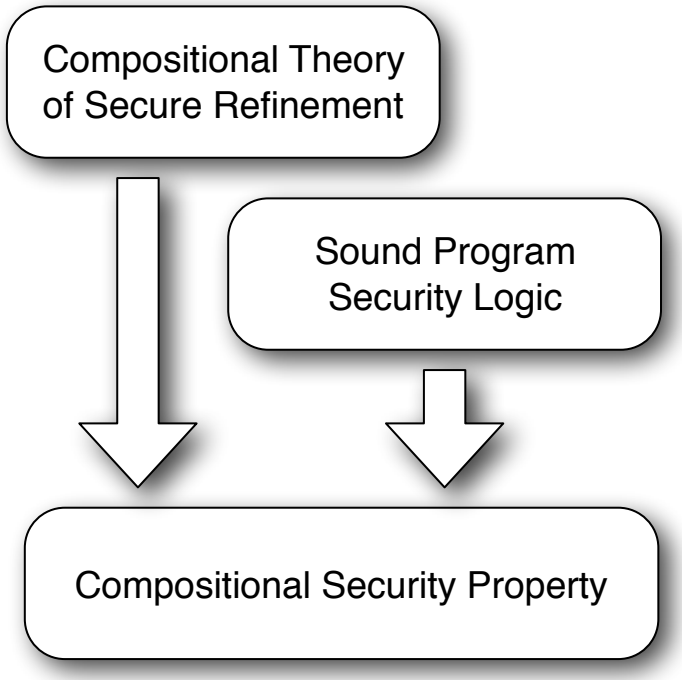


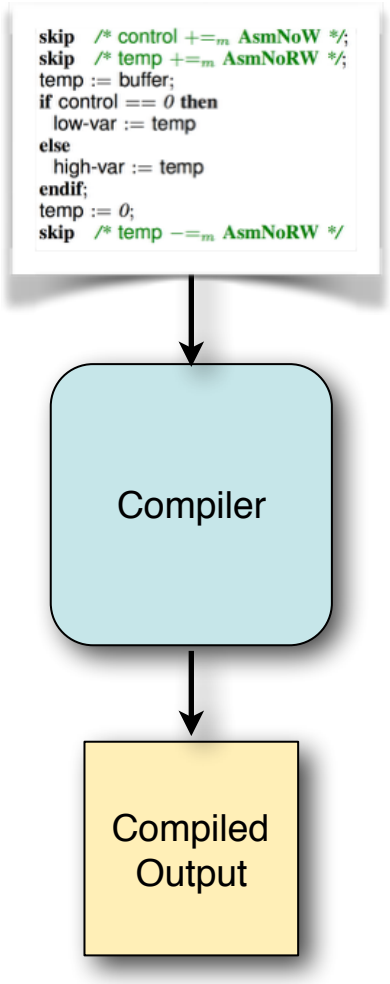
Verification Framework: Requirements

```
skip  /* control +=_m AsmNoW */;
skip  /* temp +=_m AsmNoRW */;
temp := buffer;
if control == 0 then
  low-var := temp
else
  high-var := temp
endif;
temp := 0;
skip  /* temp -=_m AsmNoRW */
```

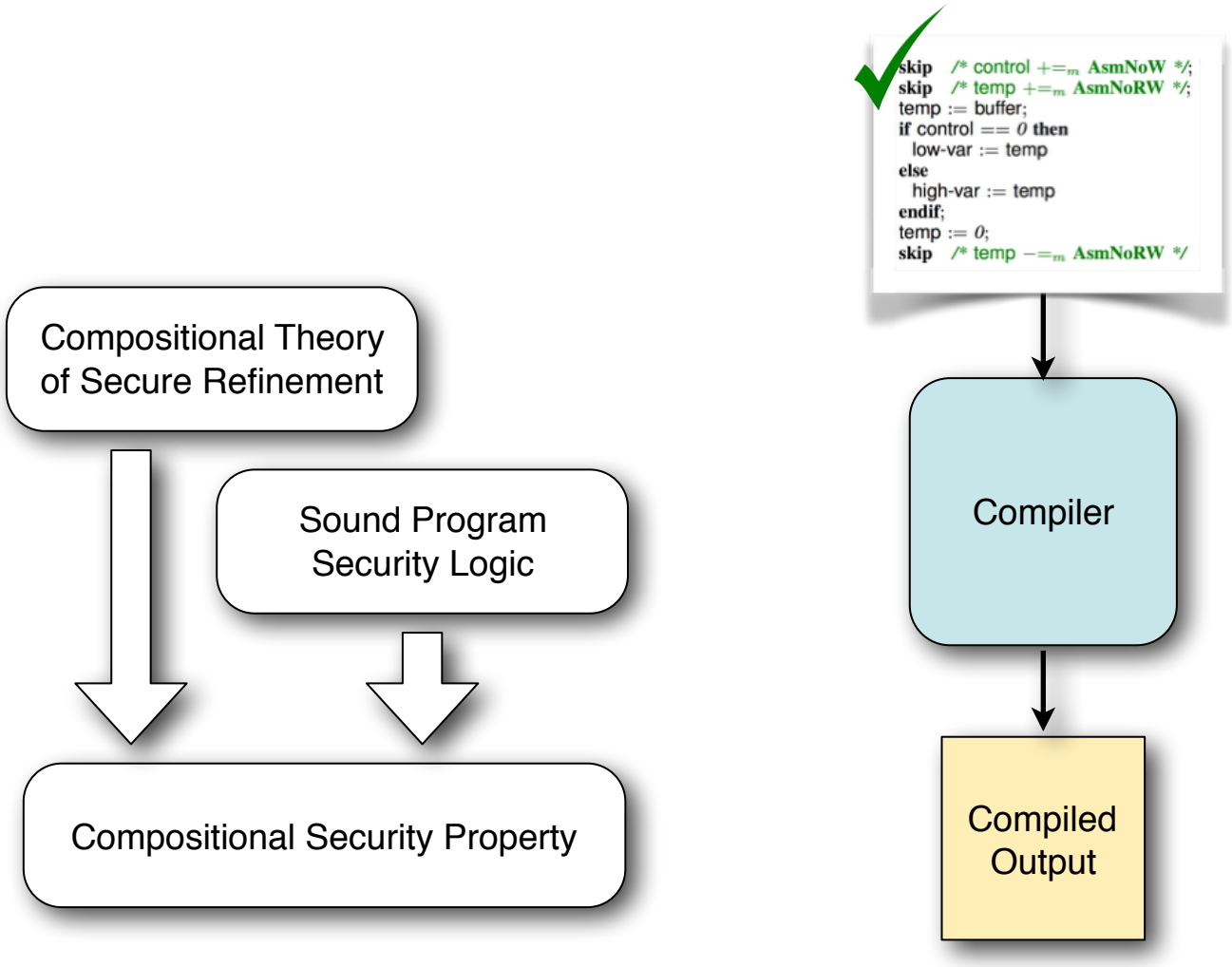


Verification Framework: Requirements

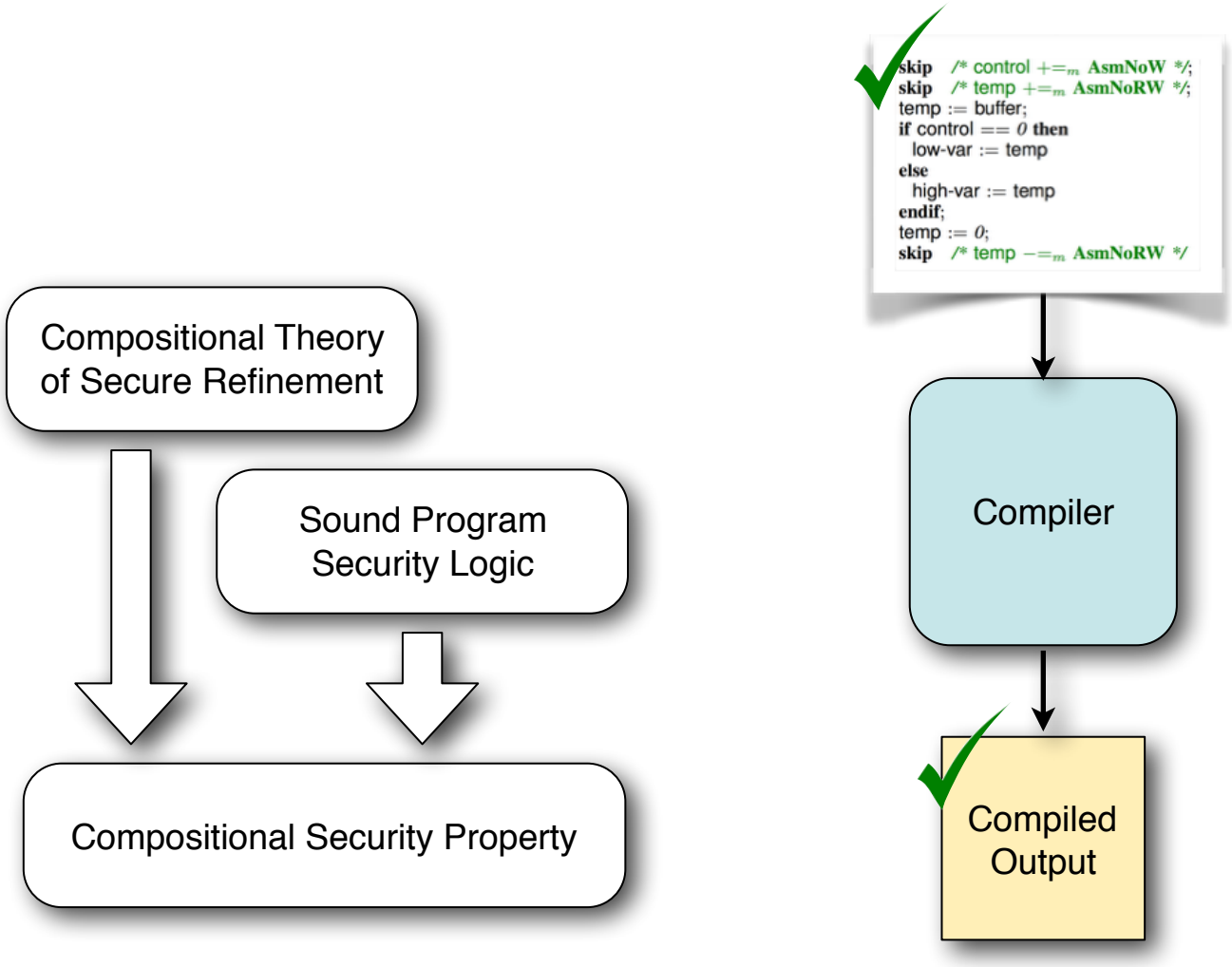




Verification Framework: Requirements

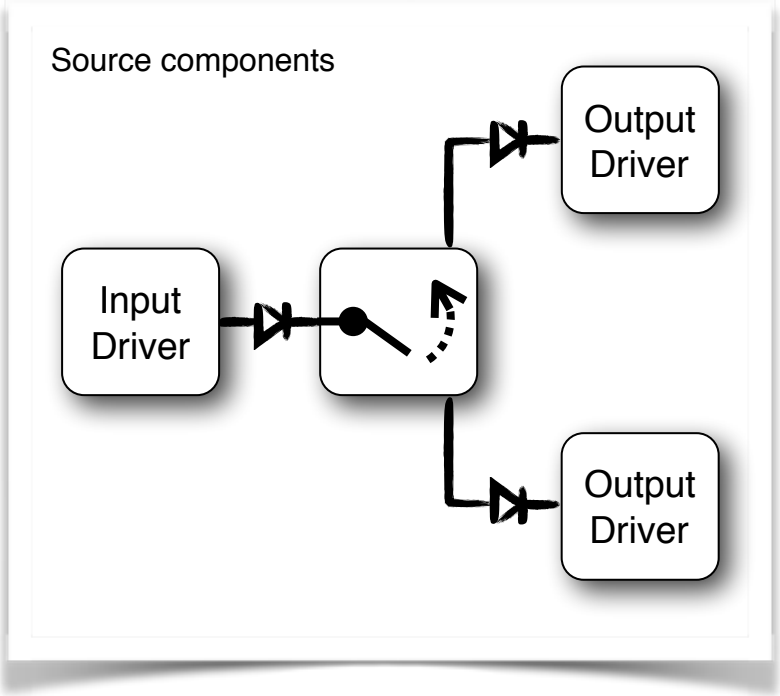


Verification Framework: Requirements

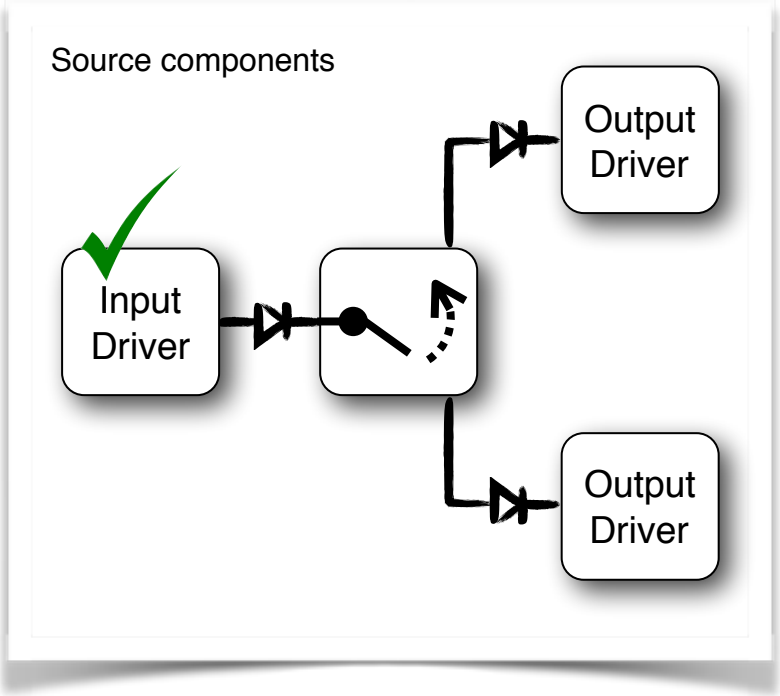


Verification Framework: Operation

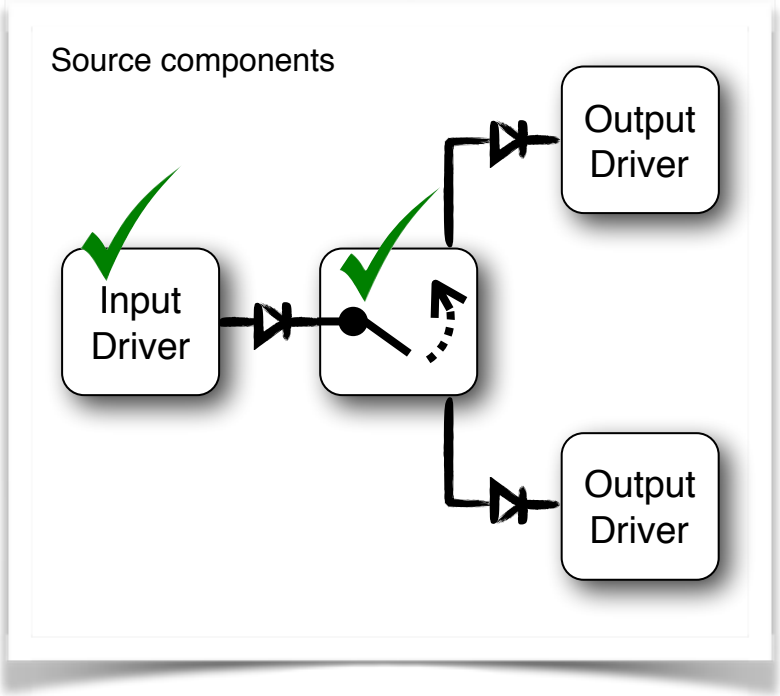
Verification Framework: Operation



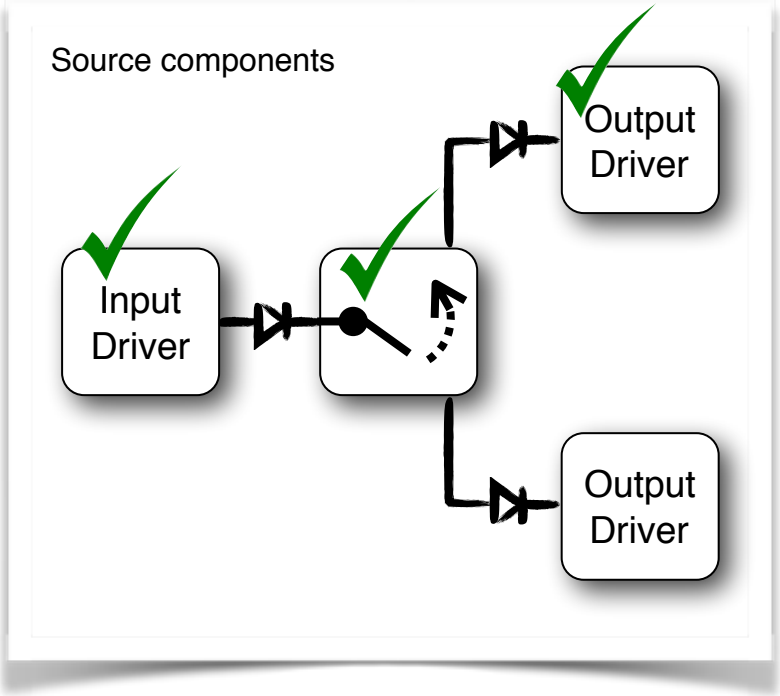
Verification Framework: Operation



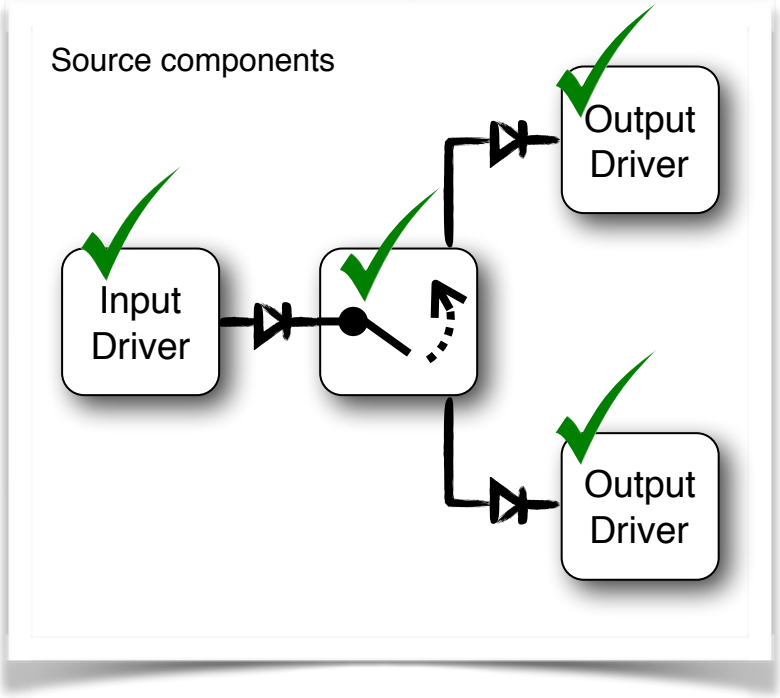
Verification Framework: Operation



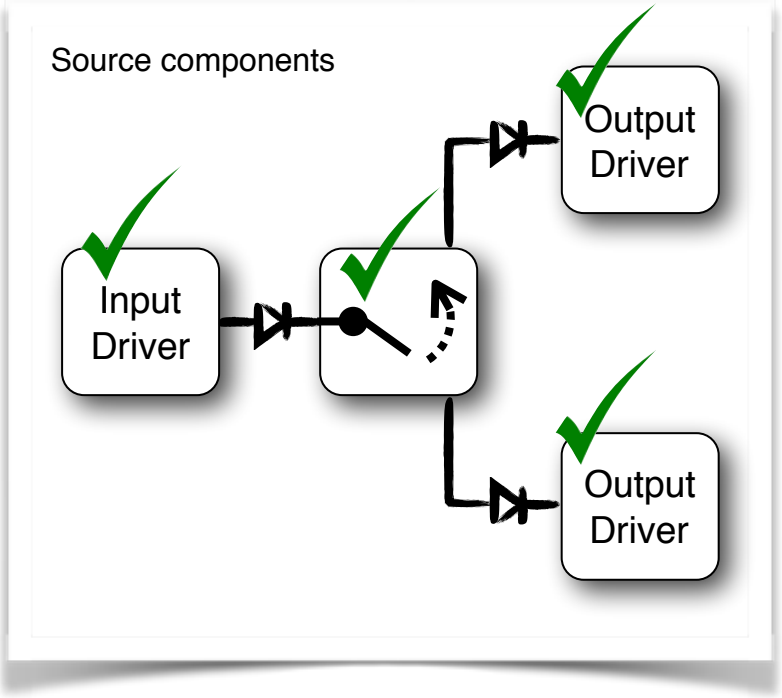
Verification Framework: Operation

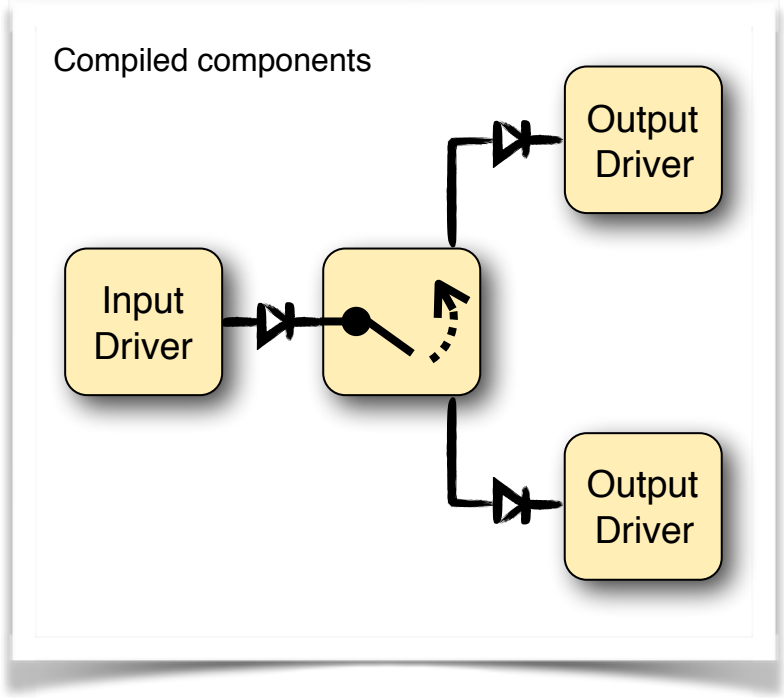


Verification Framework: Operation

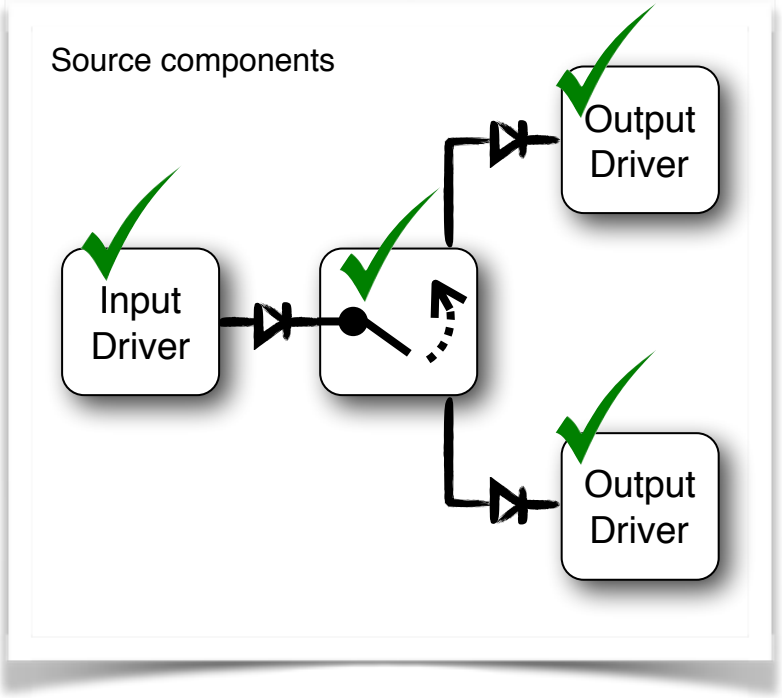


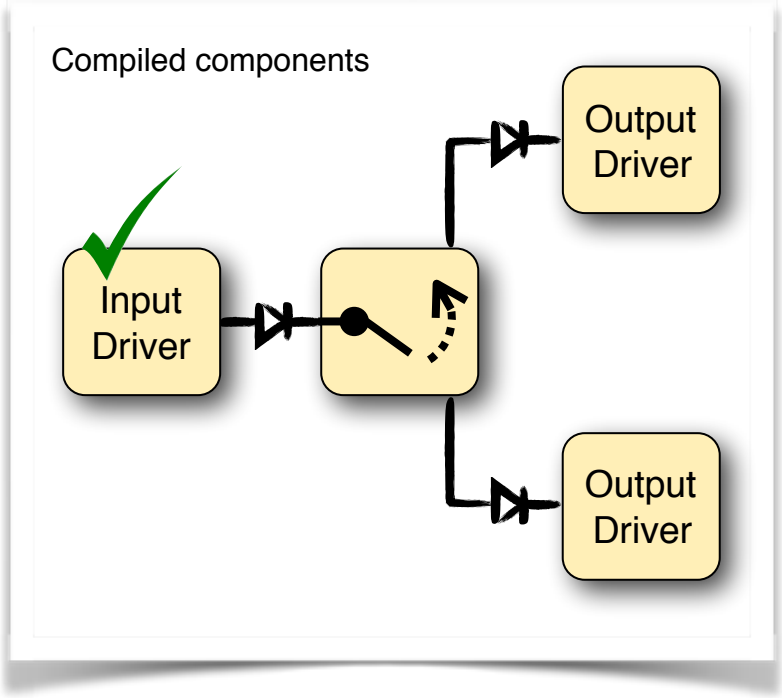
Verification Framework: Operation



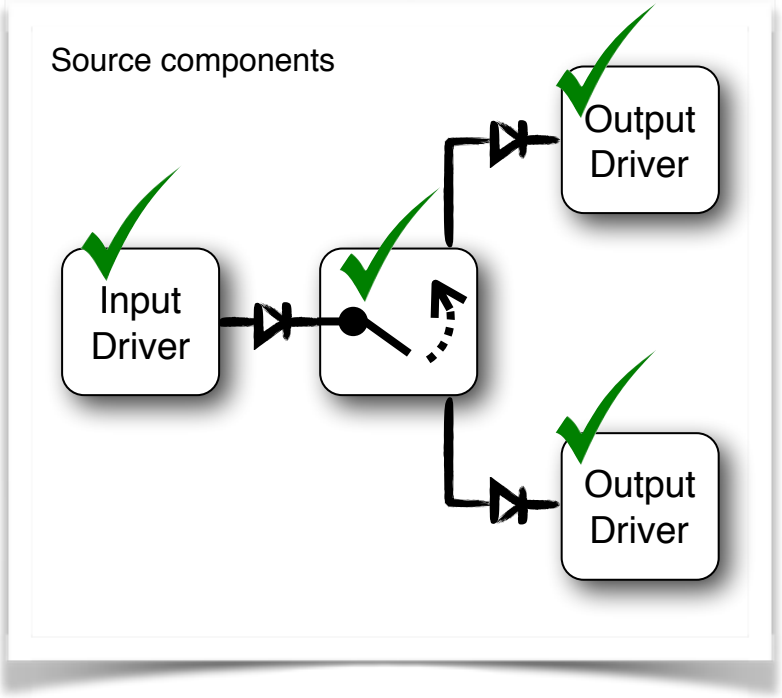


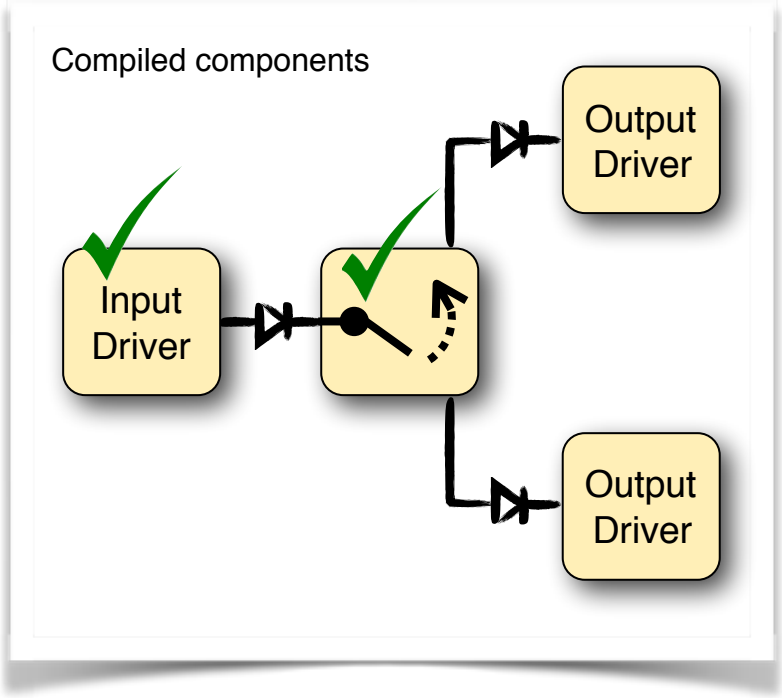
Verification Framework: Operation



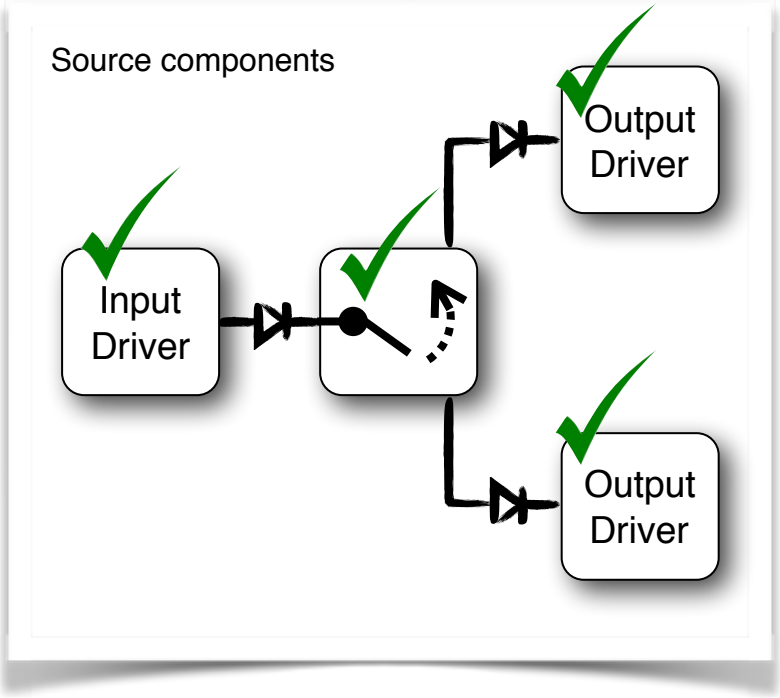


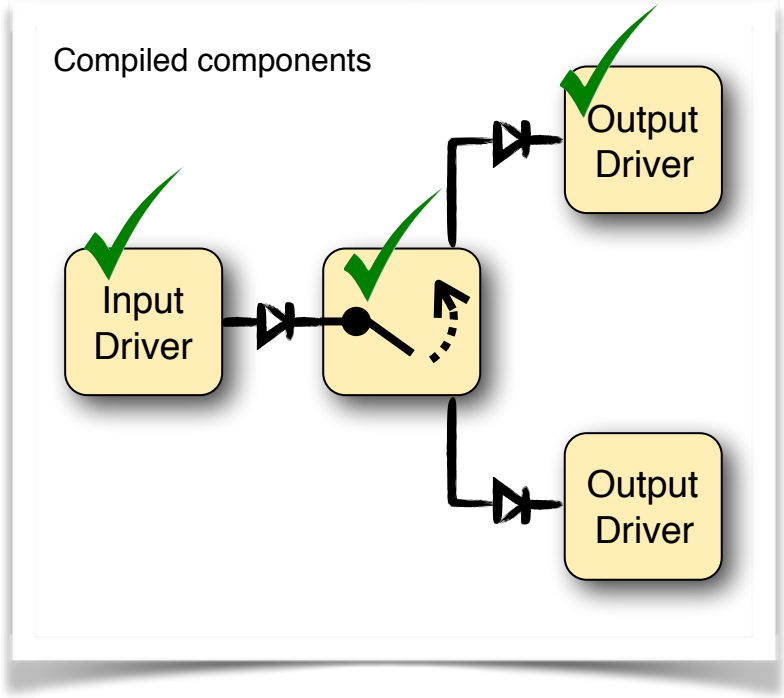
Verification Framework: Operation



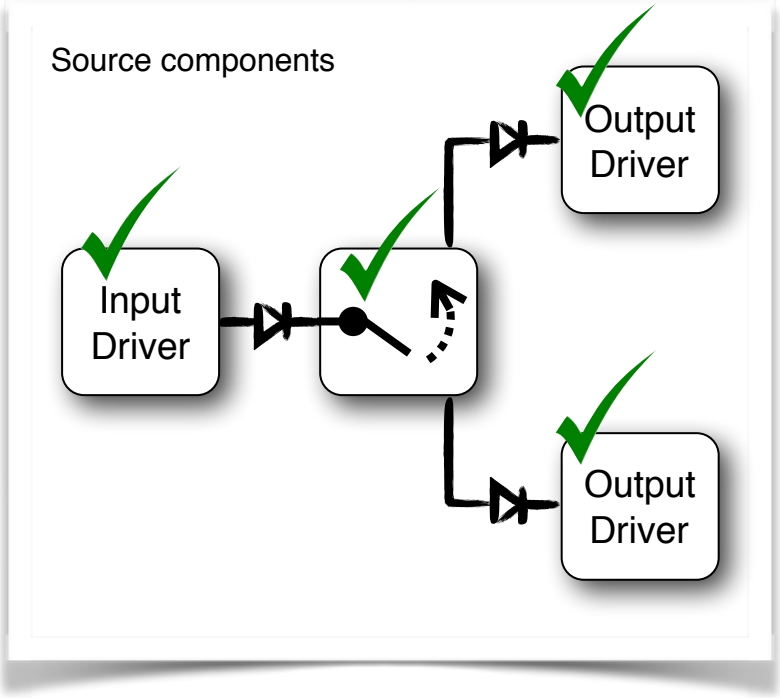


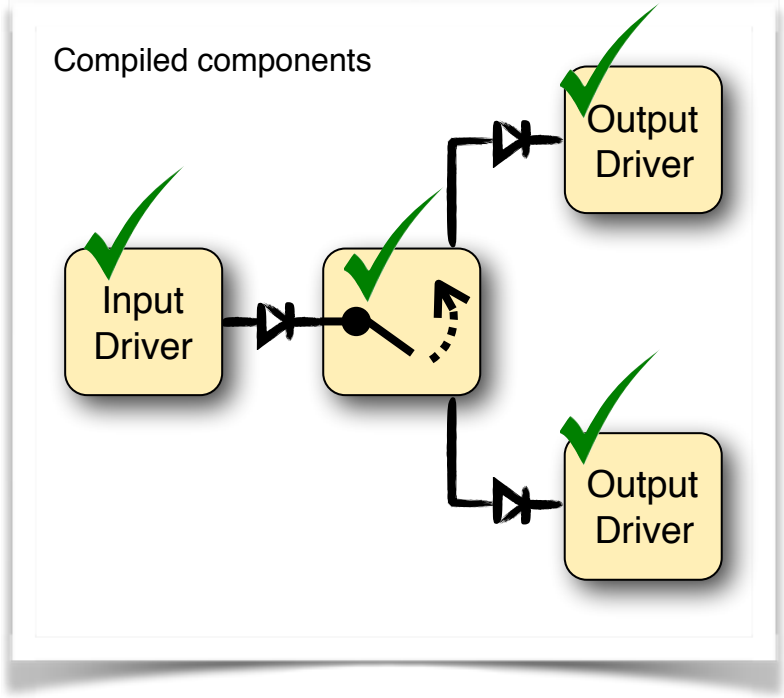
Verification Framework: Operation



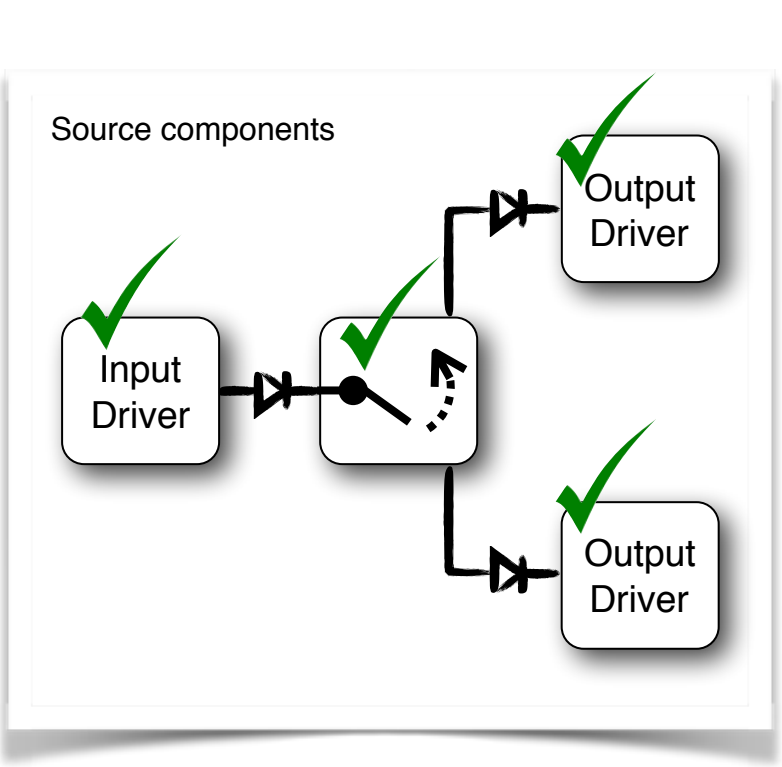


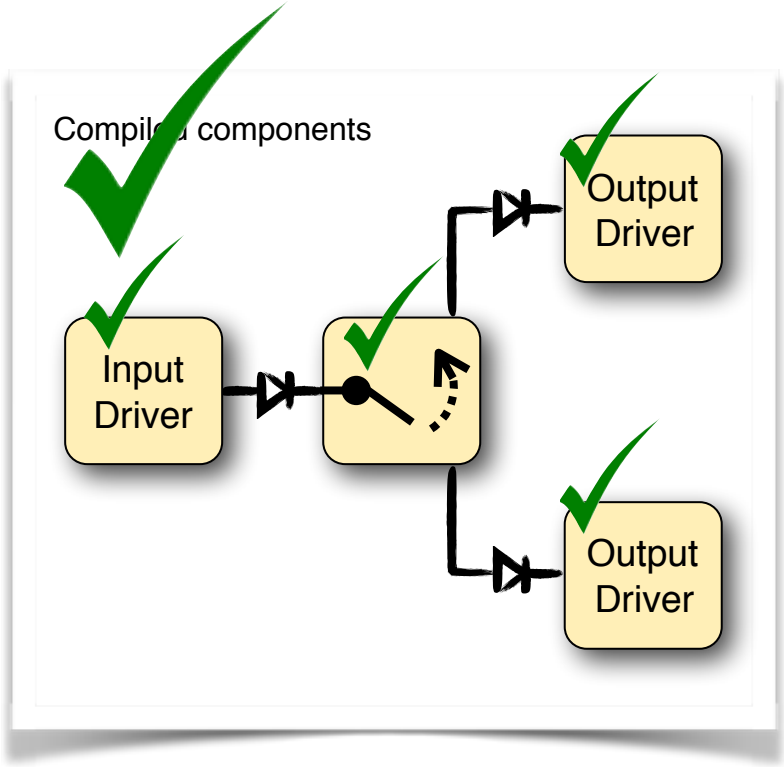
Verification Framework: Operation



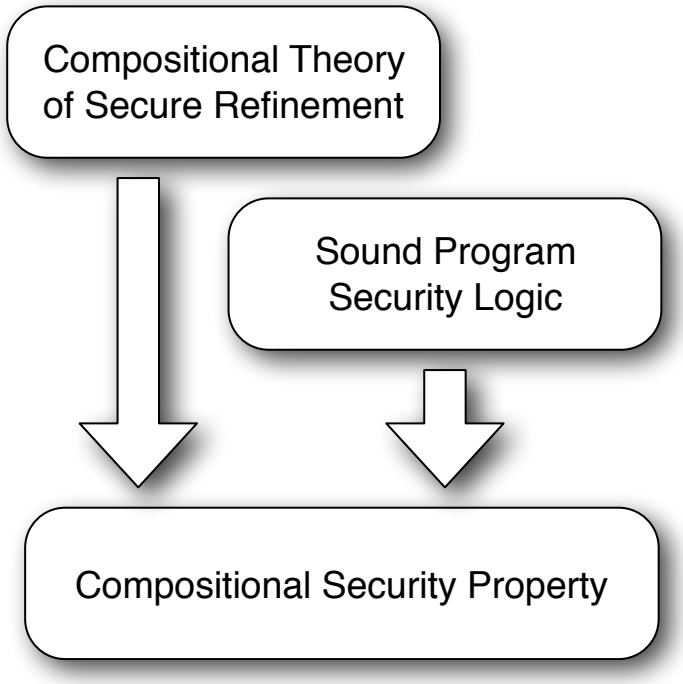


Verification Framework: Operation

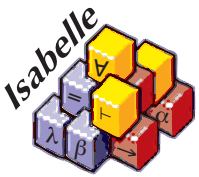


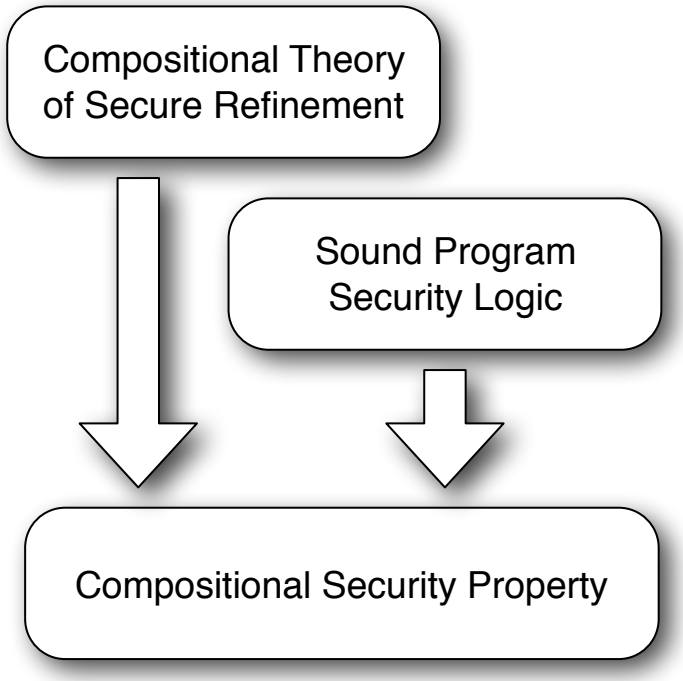


Verification Framework: Status

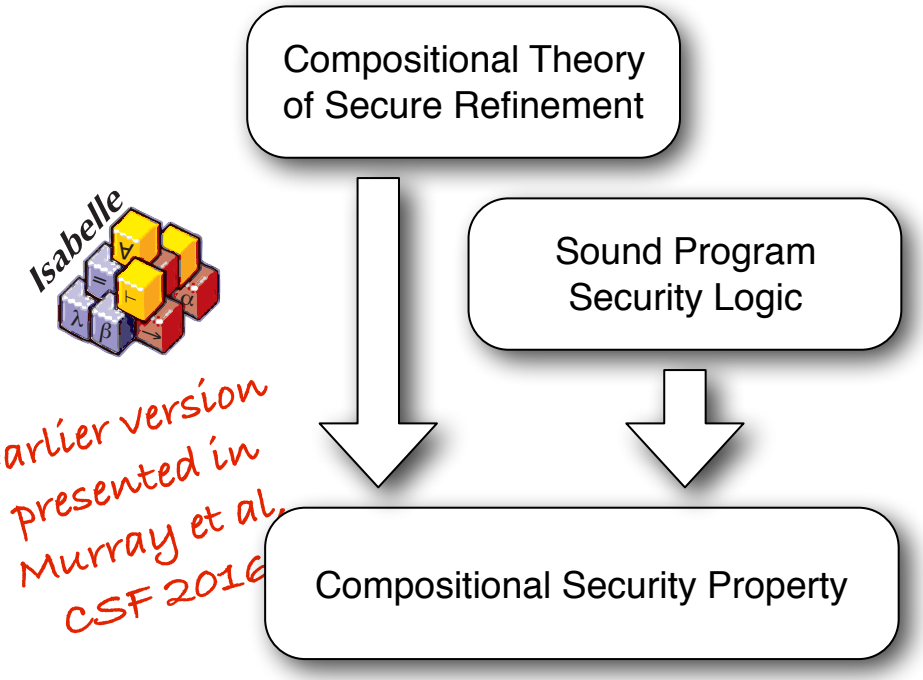


Verification Framework: Status

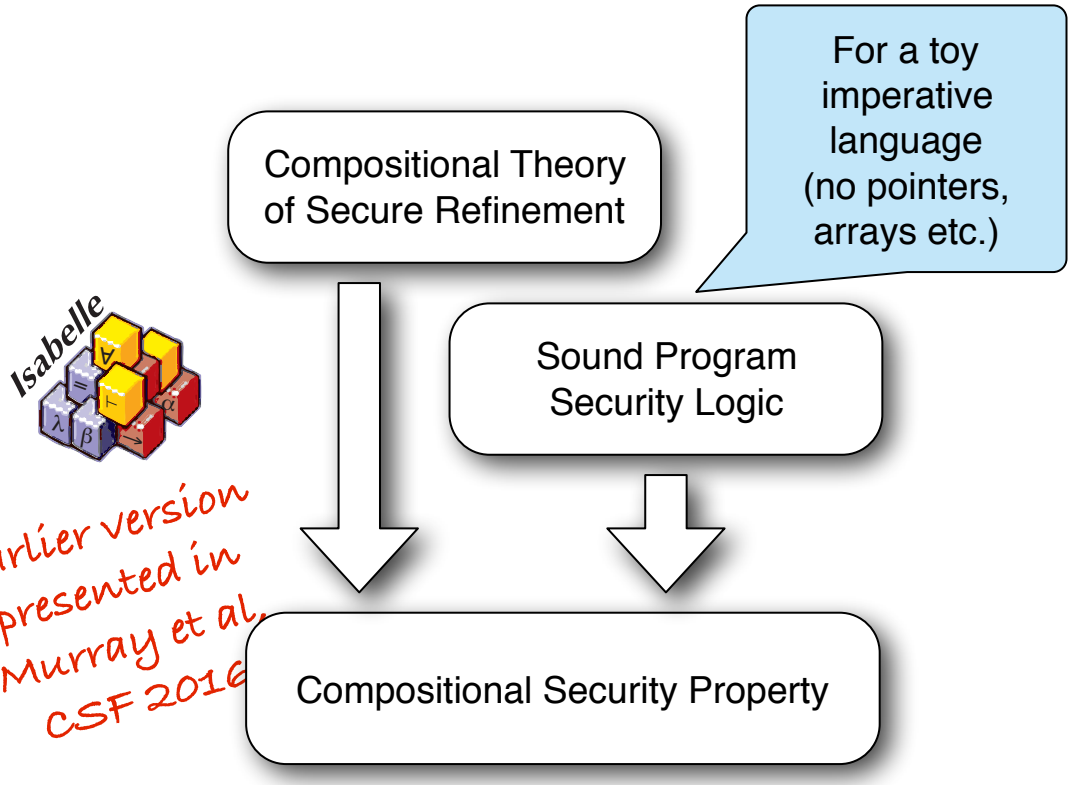




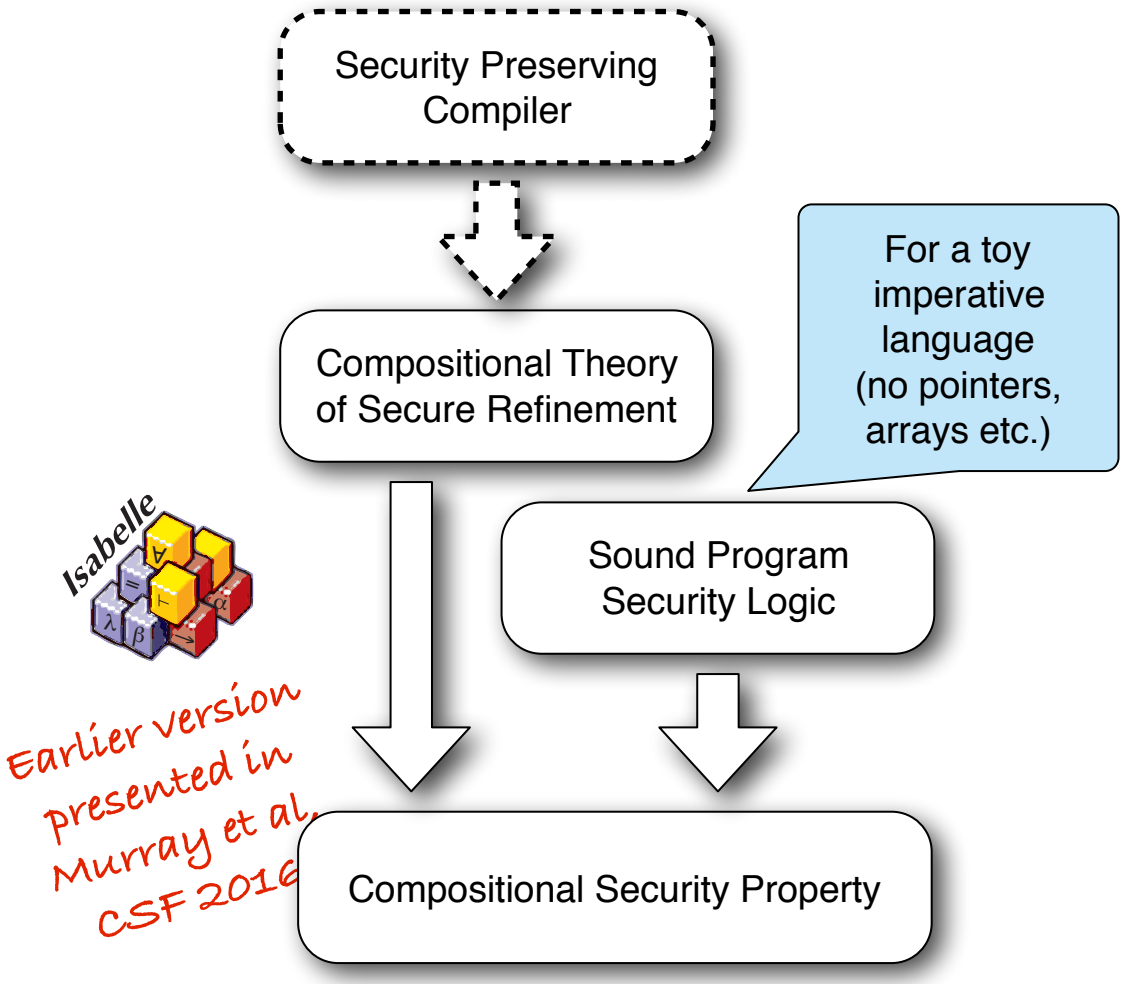
Verification Framework: Status



Verification Framework: Status

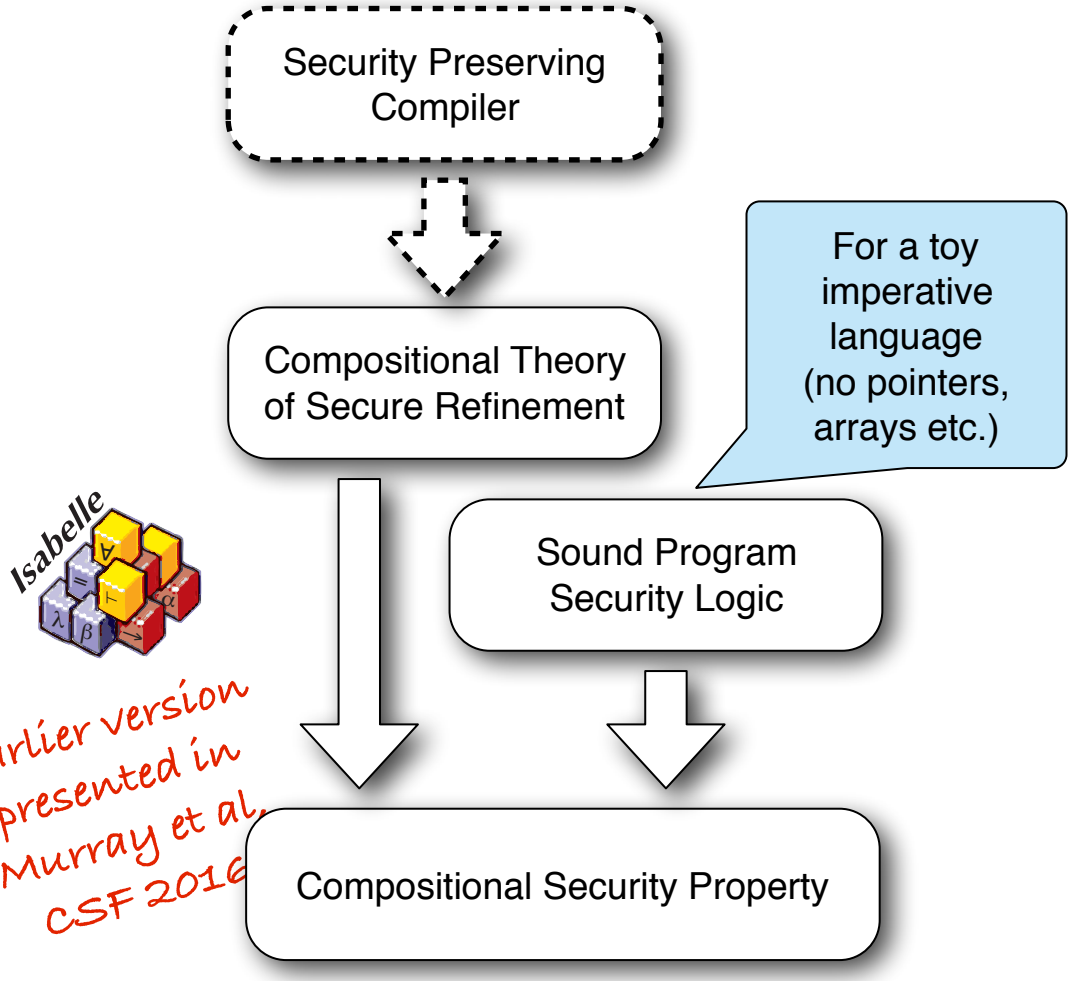


Verification Framework: Status

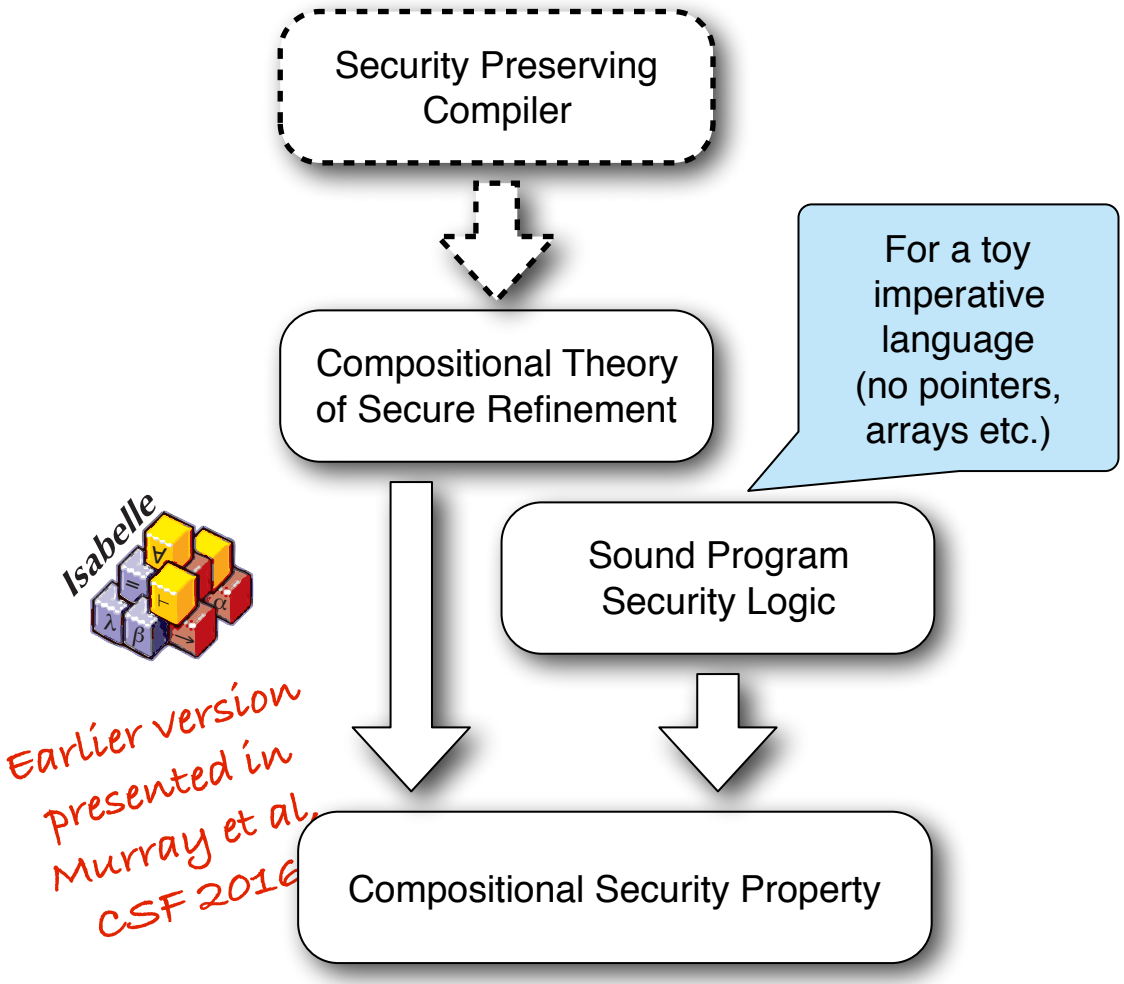


Verification Framework: Status

Robert Sison



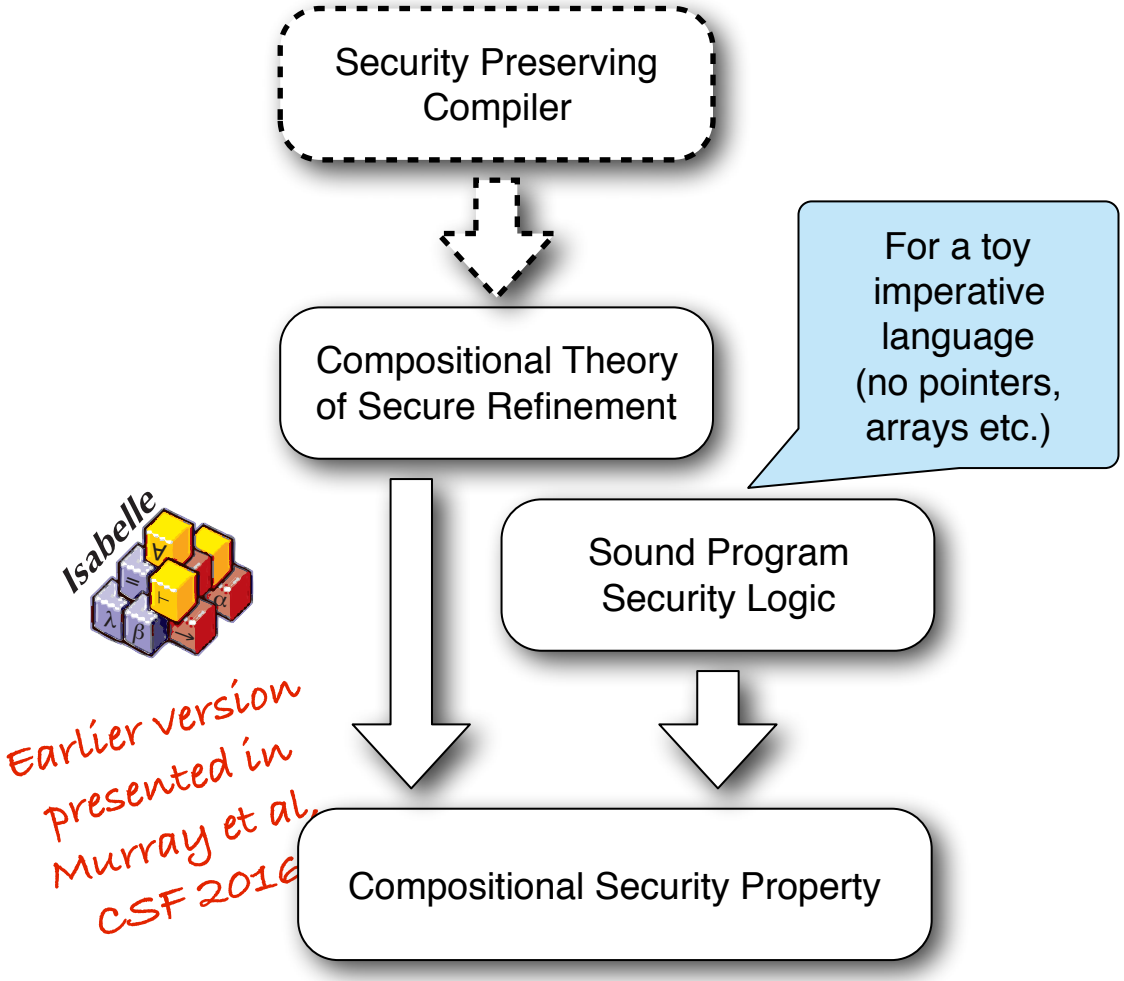
Verification Framework: Status



Robert Sison

Edward Pierzchalski

Verification Framework: Status

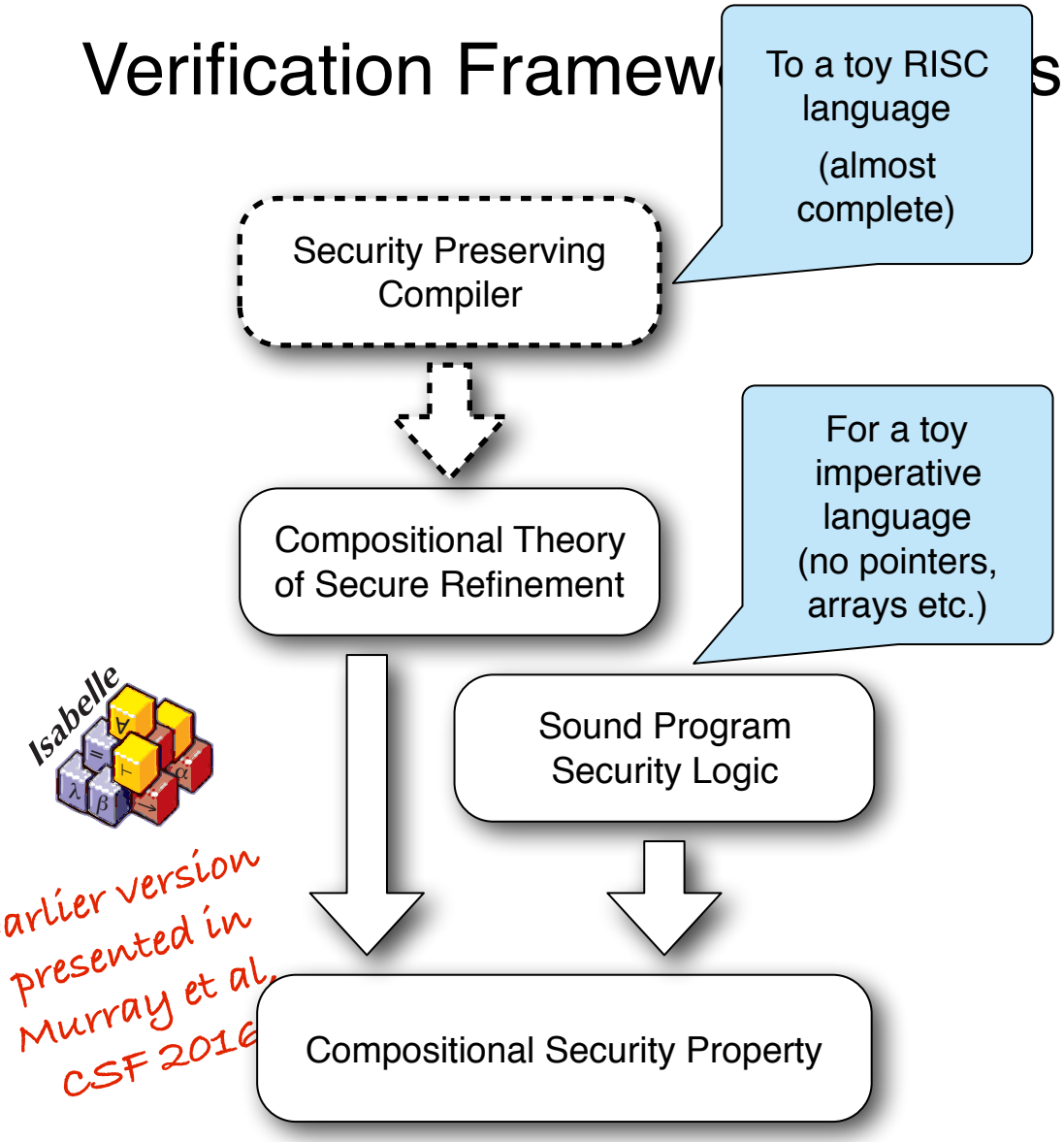


Robert Sison

Edward Pierzchalski

Christine Rizkallah

Verification Framew s

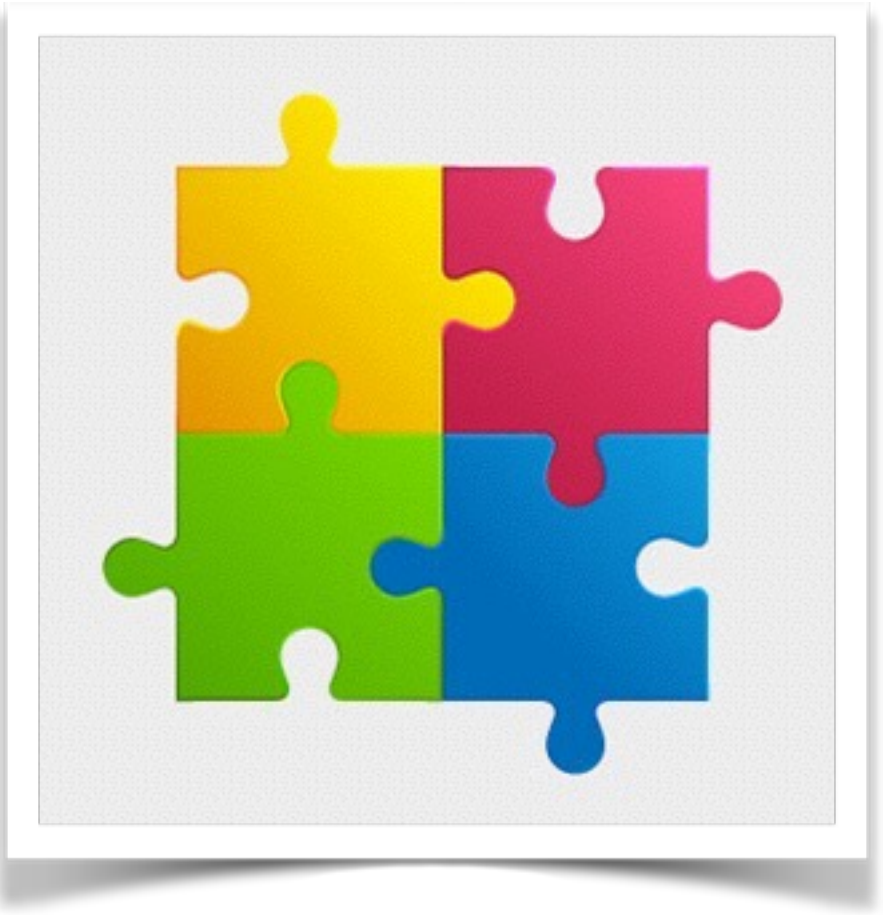


Robert Sison

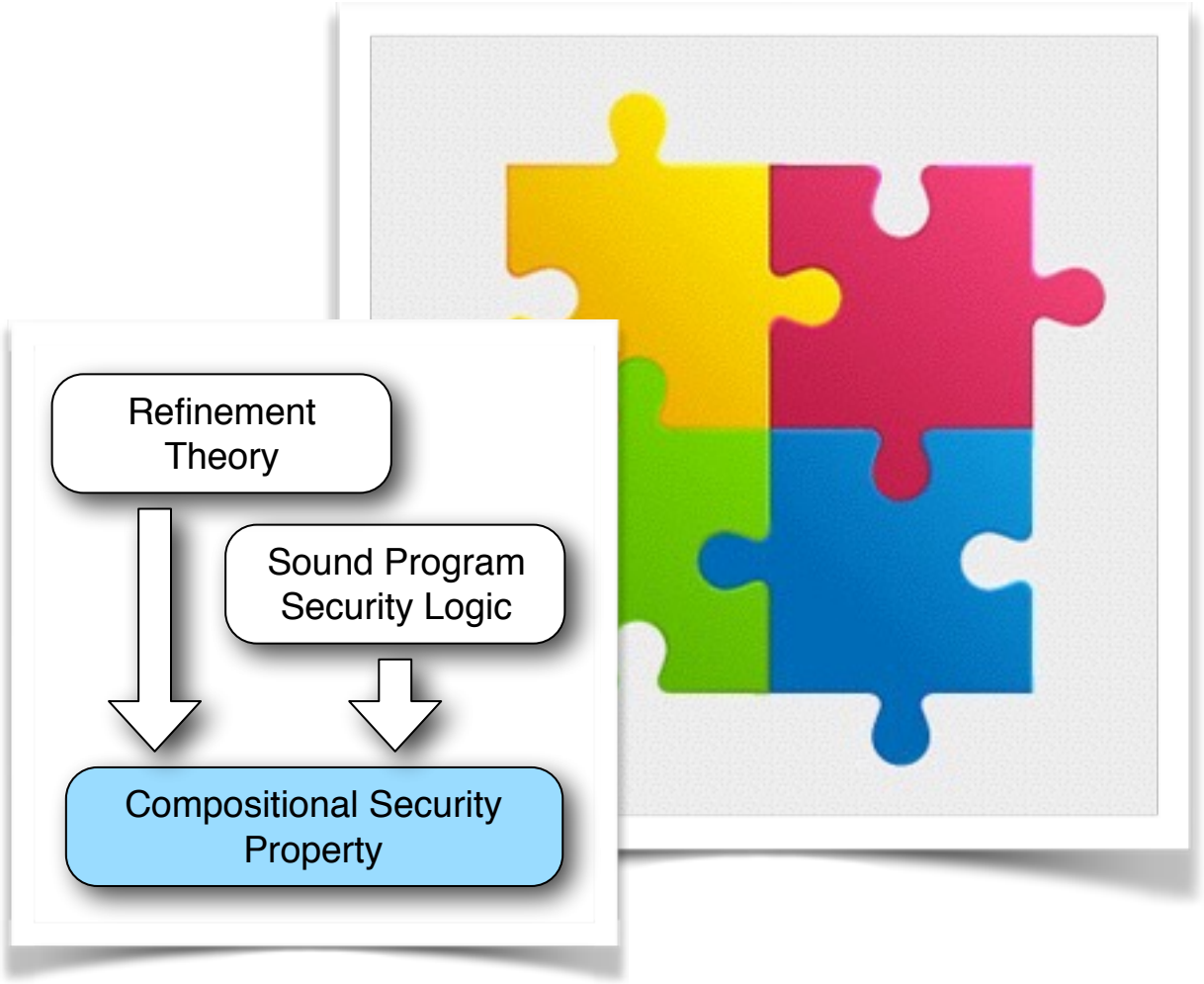
Edward Pierzchalski

Christine Rizkallah

COMPOSITIONAL SECURITY



COMPOSITIONAL SECURITY



Minimum Requirements

```
l = 0;
if (h){
  h = 0;
}
l = 1;
```

Minimum Requirements

```
l = 0;
if (h){
  h = 0;
}
l = 1;
```

Secure?

Minimum Requirements

```
l = 0;
if (h){
  h = 0;
}
l = 1;
```

Secure?

Timing-Insensitive Security:

Minimum Requirements

```
l = 0;
if (h){
  h = 0;
}
l = 1;
```

Secure?

Timing-Insensitive Security: **YES**

Minimum Requirements

```
l = 0;
if (h){
  h = 0;
}
l = 1;
```

Secure?

Timing-Insensitive Security: **YES**

Timing-Sensitive Security:

Minimum Requirements

```
l = 0;
if (h){
  h = 0;
}
l = 1;
```

Secure?

Timing-Insensitive Security: **YES**

Timing-Sensitive Security: **NO**

Minimum Requirements

```
l = 0;
if (h){
  h = 0;
}
l = 1;
```

```
print(l);
```

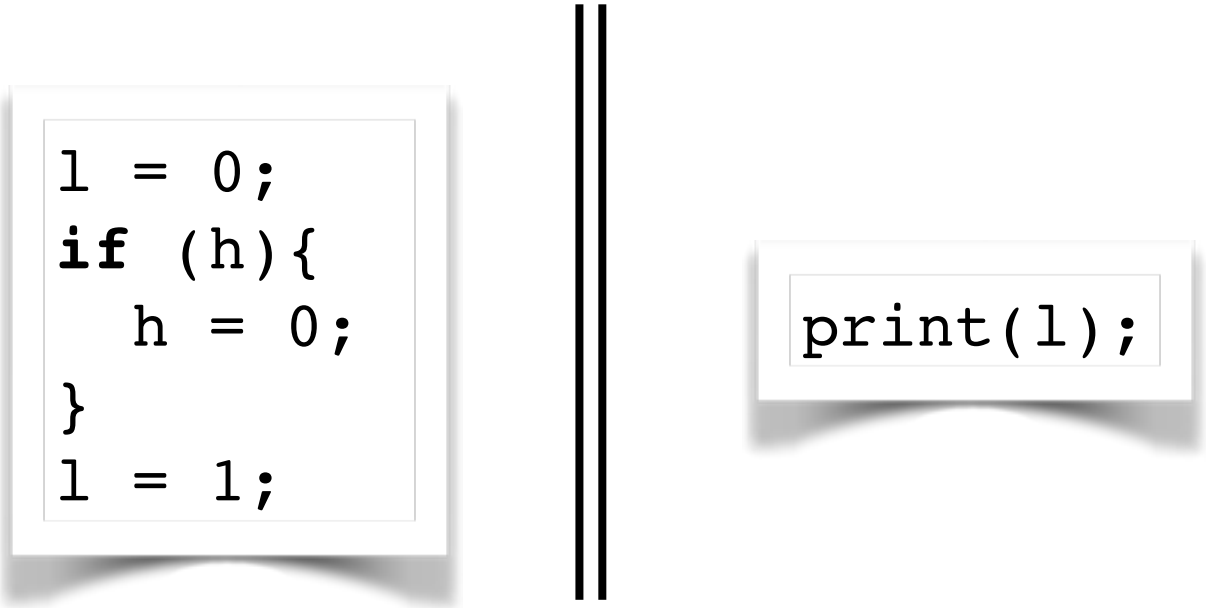
Minimum Requirements

```
l = 0;
if (h){
  h = 0;
}
l = 1;
```

```
print(l);
```

Secure?

Minimum Requirements



Secure?

Timing-Insensitive Security:

Minimum Requirements

(h = 0)

pc

```
l = 0;
if (h){
  h = 0;
}
l = 1;
```

≡

```
print(l);
```

Secure?

Timing-Insensitive Security:

Minimum Requirements

(h = 0)

pc

```
l = 0;
if (h){
  h = 0;
}
l = 1;
```

||

```
print(l);
```

Secure?

Timing-Insensitive Security:

Minimum Requirements

(h = 0)

pc

```
l = 0;
if (h){
  h = 0;
}
l = 1;
```

≡

```
print(l);
```

Secure?

Timing-Insensitive Security:

Minimum Requirements

(h = 0)

```
l = 0;
if (h){
  h = 0;
}
l = 1;
```

pc

≡

```
print(l);
```

Secure?

Timing-Insensitive Security:

Minimum Requirements

(h = 0)

```
l = 0;
if (h){
  h = 0;
}
l = 1;
```

pc

≡

```
print(l);
```

Secure?

Timing-Insensitive Security:

Minimum Requirements

(h = 0)

```
l = 0;
if (h){
  h = 0;
}
l = 1;
```

pc

||

```
print(l);
```

Secure?

Timing-Insensitive Security:

Minimum Requirements

(h = 0)

```
l = 0;
if (h){
  h = 0;
}
l = 1;
```

pc

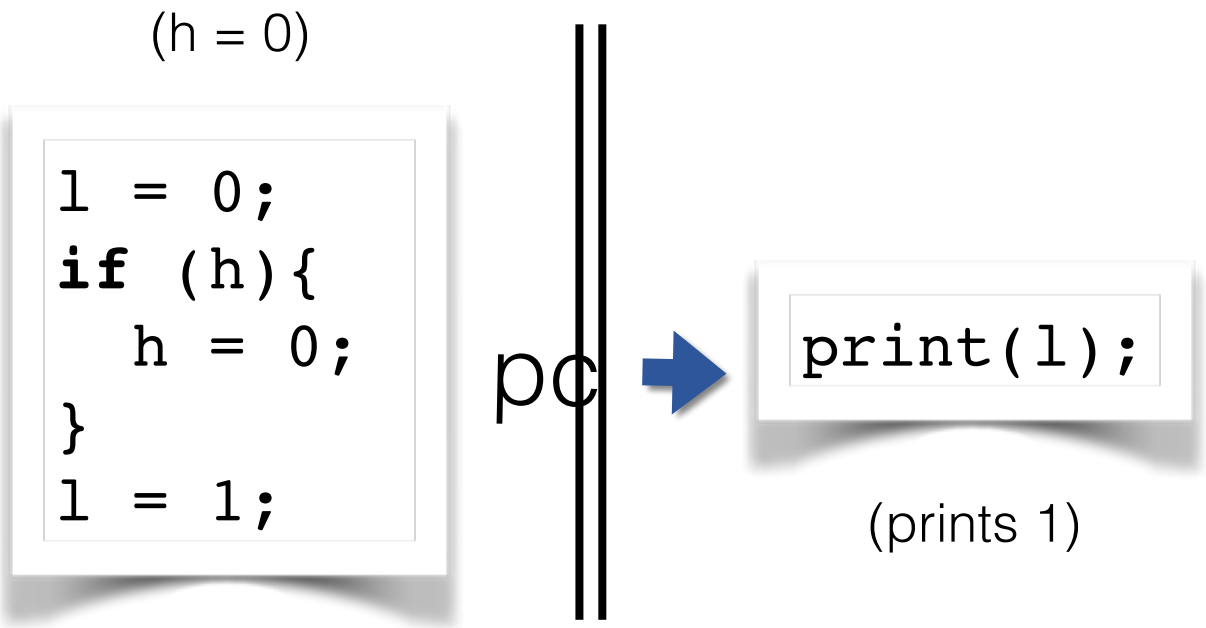
≡

```
print(l);
```

Secure?

Timing-Insensitive Security:

Minimum Requirements



Secure?

Timing-Insensitive Security:

Minimum Requirements

(h = 1)

pc

```
l = 0;
if (h){
  h = 0;
}
l = 1;
```

||

```
print(l);
```

Secure?

Timing-Insensitive Security:

Minimum Requirements

(h = 1)

pc

```
l = 0;
if (h){
  h = 0;
}
l = 1;
```

≡

```
print(l);
```

Secure?

Timing-Insensitive Security:

Minimum Requirements

(h = 1)

pc

```
l = 0;
if (h){
  h = 0;
}
l = 1;
```

≡

```
print(l);
```

Secure?

Timing-Insensitive Security:

Minimum Requirements

(h = 1)

pc

```
l = 0;
if (h){
  h = 0;
}
l = 1;
```

||

```
print(l);
```

Secure?

Timing-Insensitive Security:

Minimum Requirements

(h = 1)

pc

```
l = 0;
if (h){
  h = 0;
}
l = 1;
```

||

```
print(l);
```

Secure?

Timing-Insensitive Security:

Minimum Requirements

(h = 1)

```
l = 0;
if (h){
  h = 0;
}
l = 1;
```

pc

||

```
print(l);
```

Secure?

Timing-Insensitive Security:

Minimum Requirements

(h = 1)

pc

```
l = 0;
if (h){
  h = 0;
}
l = 1;
```

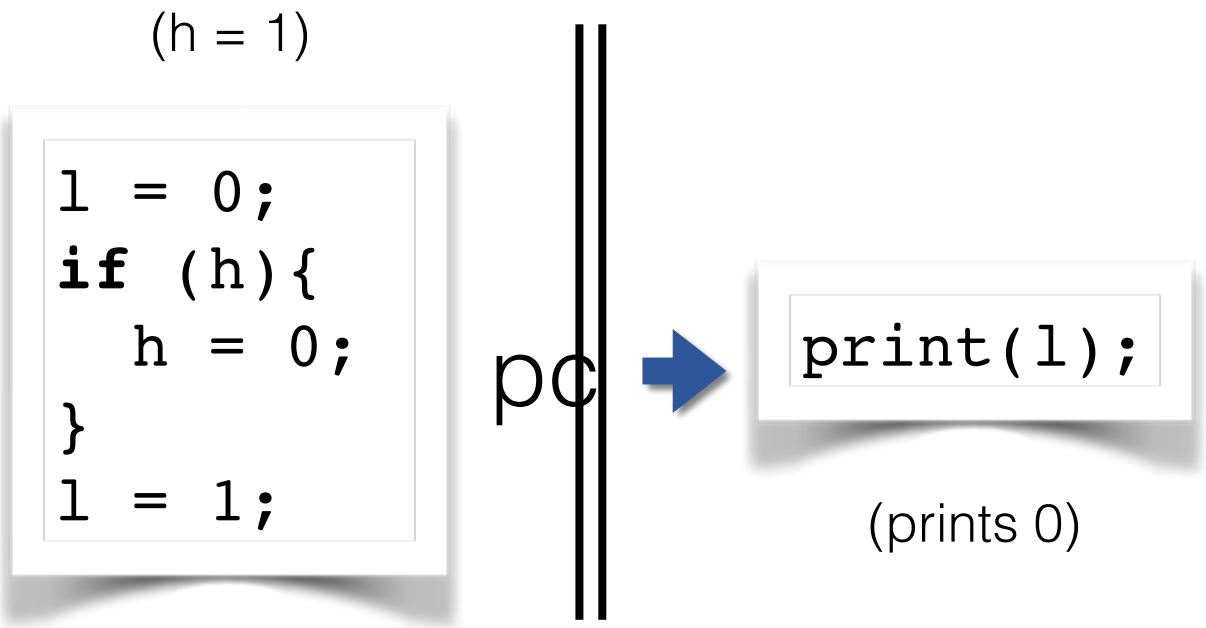
≡

```
print(l);
```

Secure?

Timing-Insensitive Security:

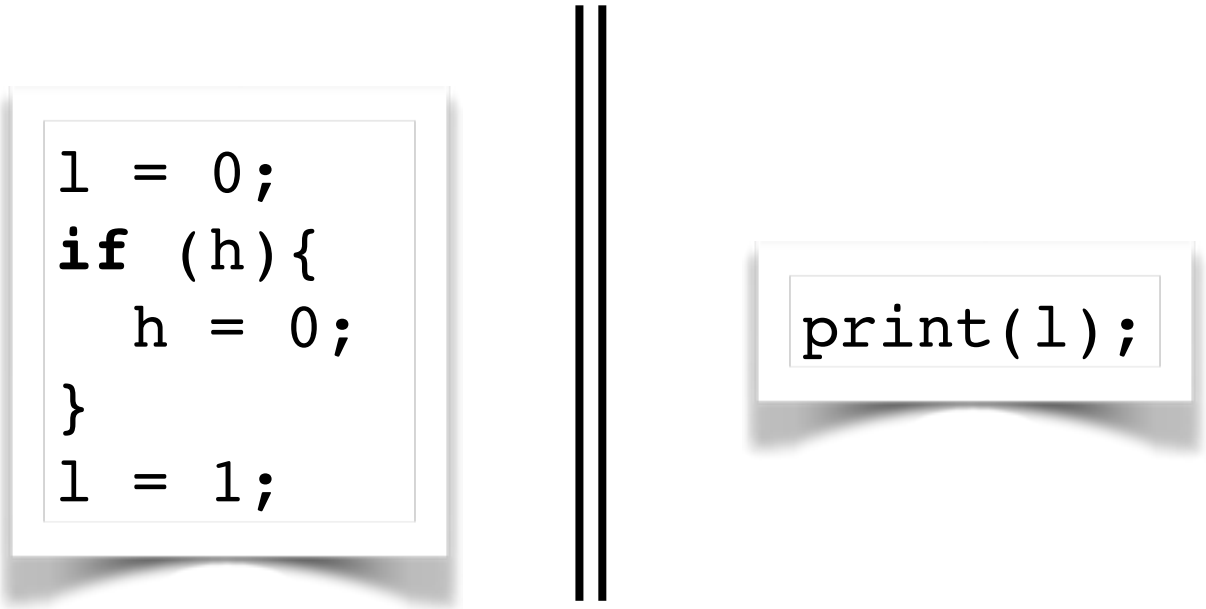
Minimum Requirements



Secure?

Timing-Insensitive Security:

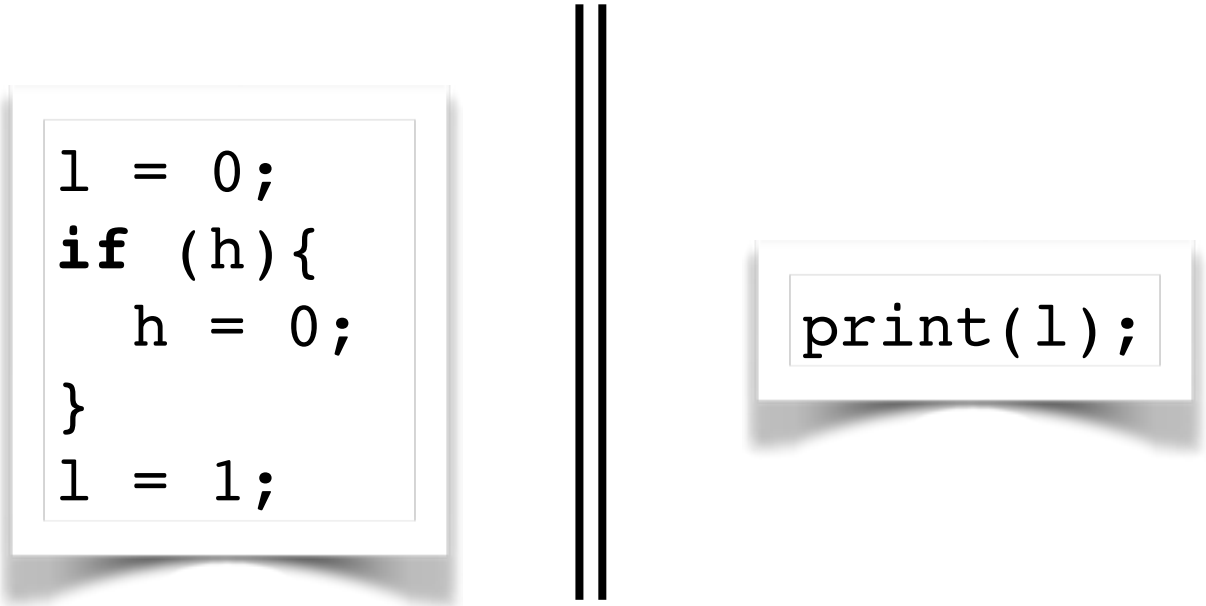
Minimum Requirements



Secure?

Timing-Insensitive Security:

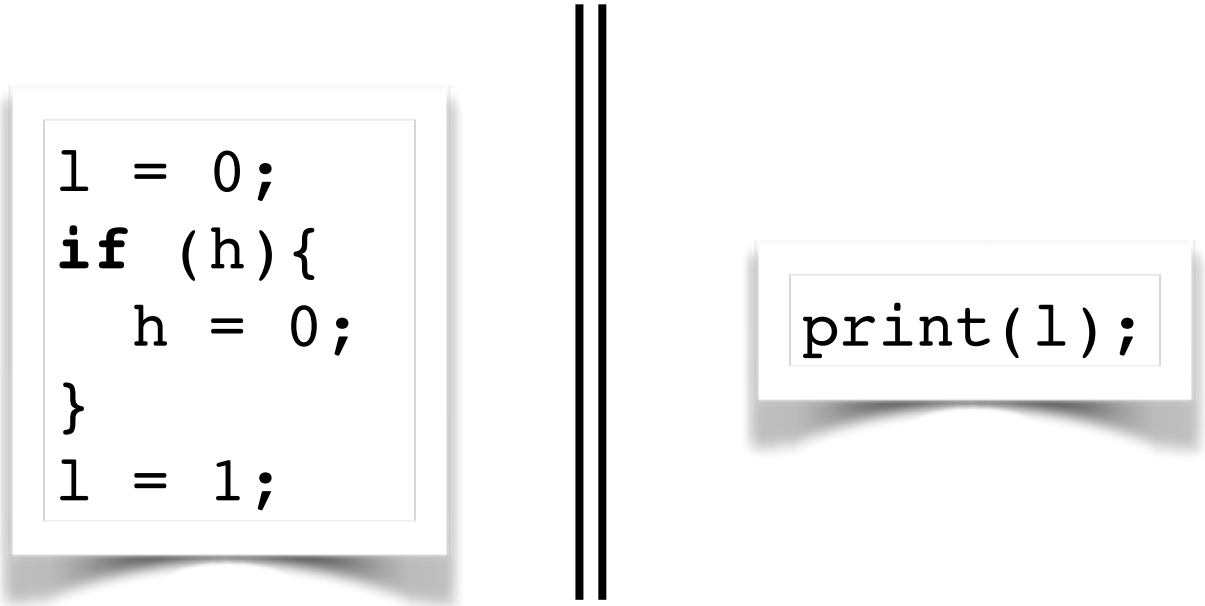
Minimum Requirements



Secure?

Timing-Insensitive Security: **NO**

Minimum Requirements



Secure?

Timing-Insensitive Security: **NO**

Timing-Sensitive Security:

Minimum Requirements

```
l = 0;
if (h){
  h = 0;
}
l = 1;
```

||

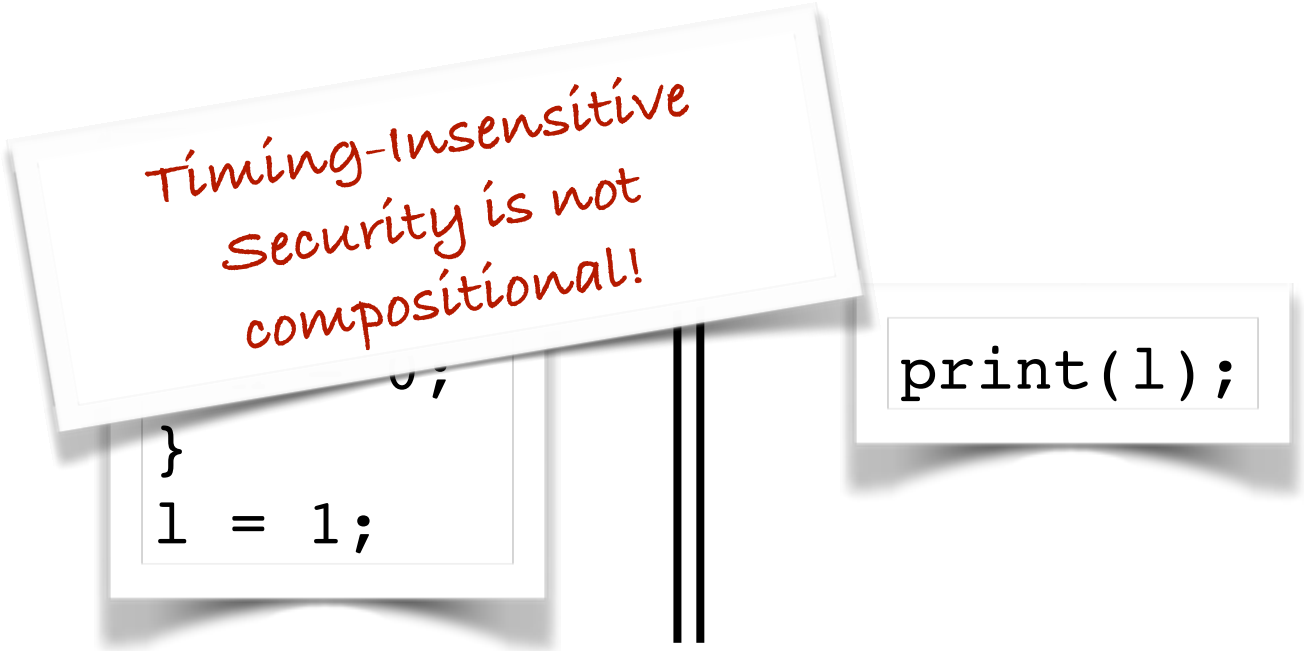
```
print(l);
```

Secure?

Timing-Insensitive Security: **NO**

Timing-Sensitive Security: **NO** (trivially)

Minimum Requirements

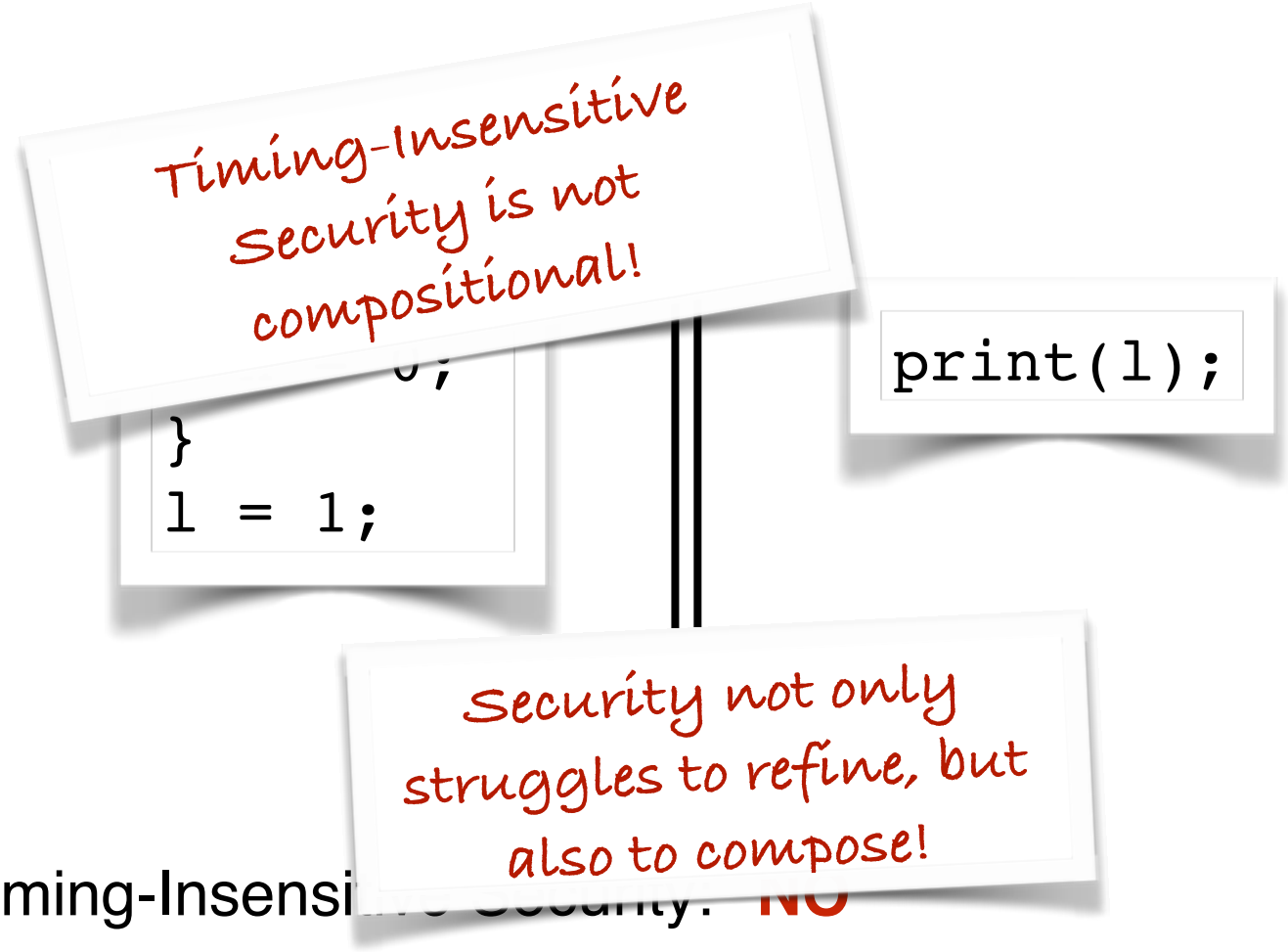


Secure?

Timing-Insensitive Security: **NO**

Timing-Sensitive Security: **NO** (trivially)

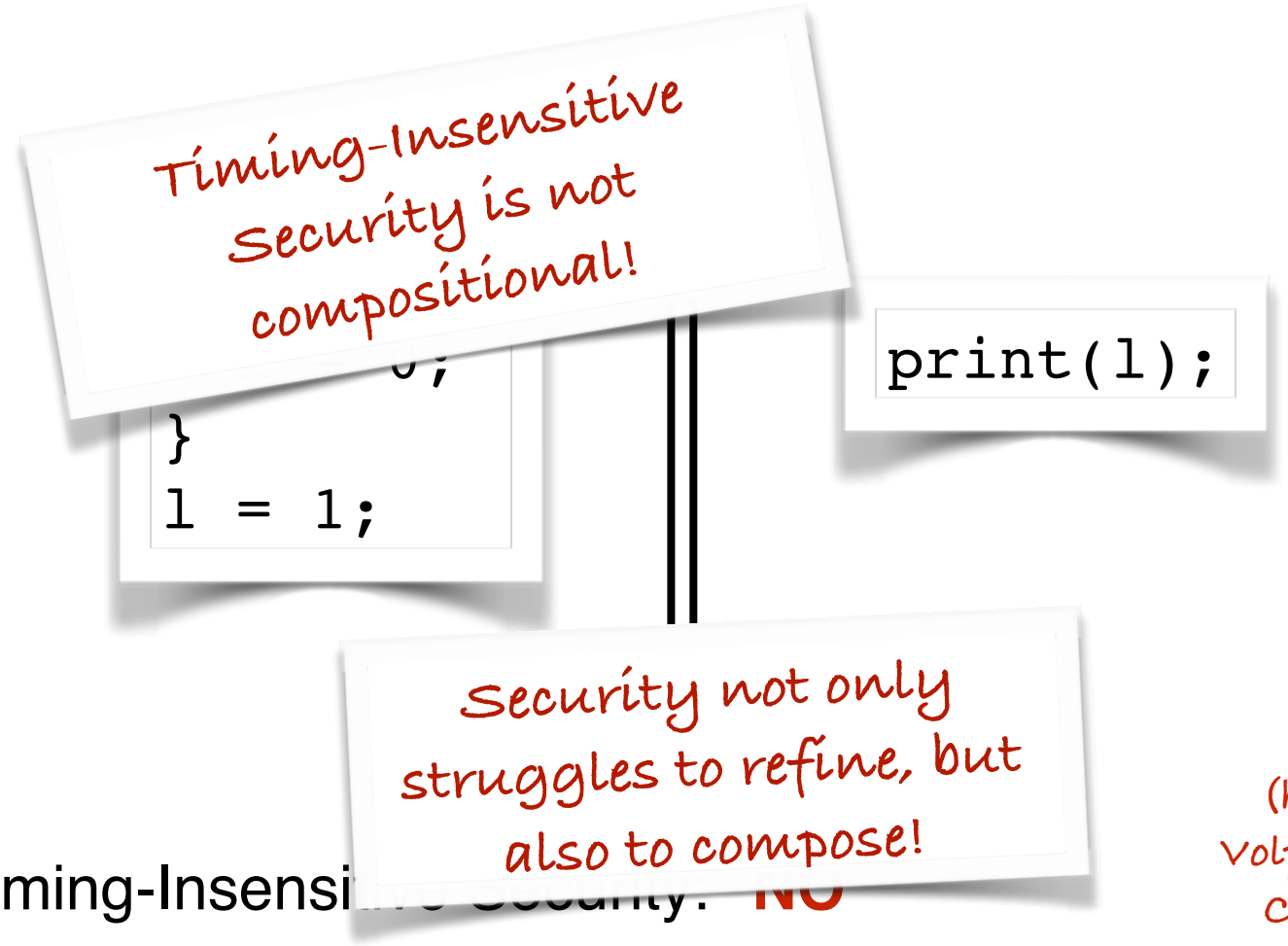
Minimum Requirements



Timing-Insensi… Security: **NO**

Timing-Sensitive Security: **NO** (trivially)

Minimum Requirements

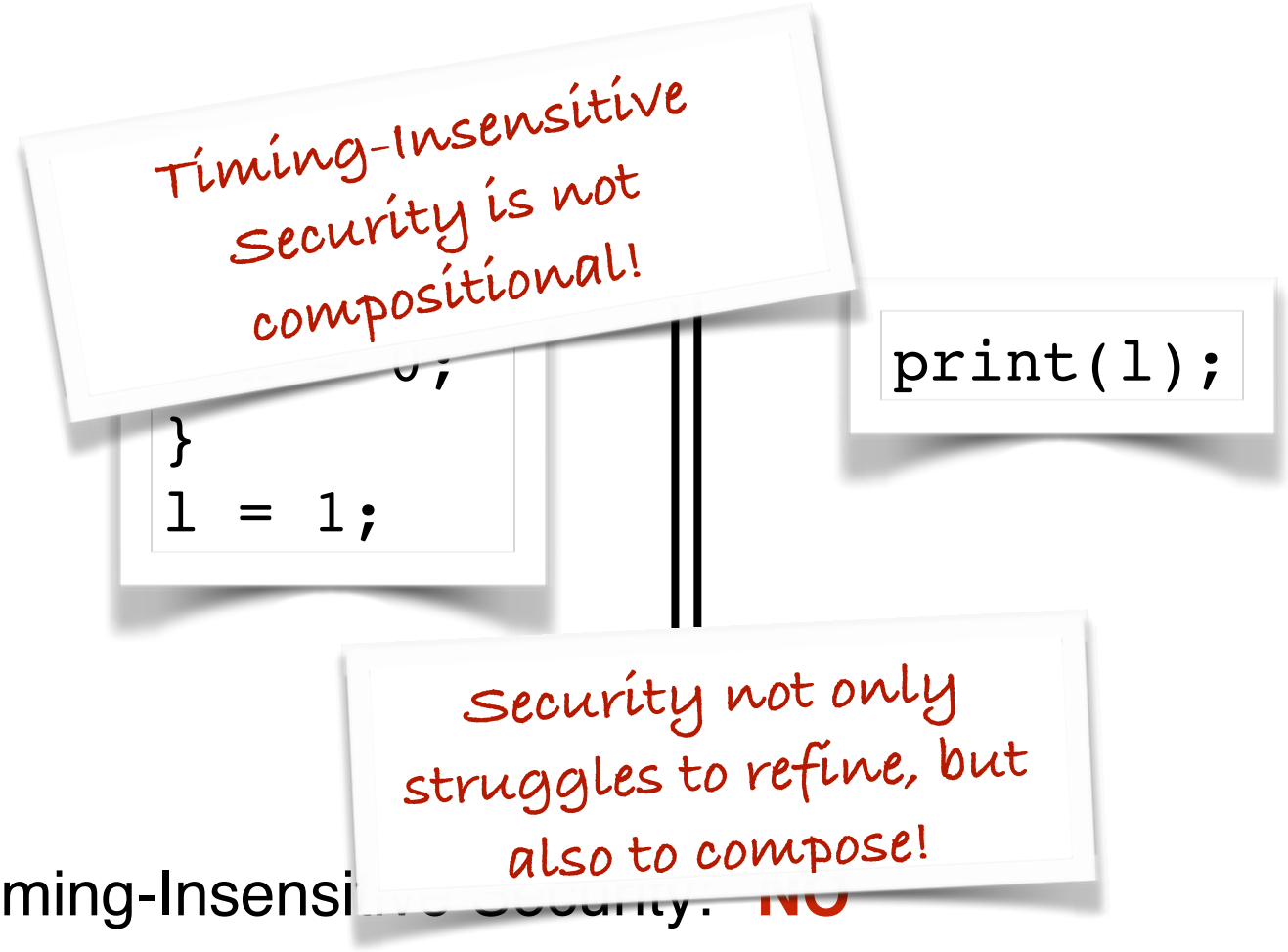


(known since Volpano & Smith, CSFW 1998)

Timing-Insensi… Security: **NO**

Timing-Sensitive Security: **NO** (trivially)

Minimum Requirements



Timing-Insensi… Security: **NO**

Timing-Sensitive Security: **NO** (trivially)

Compositional, Value-Dependent Security

```
/* {dMode,sMode,temp} += AsmNoRW */
/* acquire (dMode == sMode) */
temp := buffer;
if sMode == 0 then
  low-var := temp
else
  high-var := temp
endif;
temp := 0
```

Compositional, Value-Dependent Security

Extension of Mantel, Sands & Sudbrock (CSF 2011)

```
/* {dMode,sMode,temp} += AsmNoRW */
/* acquire (dMode == sMode) */
temp := buffer;
if sMode == 0 then
  low-var := temp
else
  high-var := temp
endif;
temp := 0
```

Compositional, Value-Dependent Security

```
var dMode              // clas: Low
var high_var           // clas: High
var low_var, sMode     // clas: Low
var buffer             // clas: dMode == 0 ? Low : High
```

```
/* {dMode,sMode,temp} += AsmNoRW */
/* acquire (dMode == sMode) */
temp := buffer;
if sMode == 0 then
  low-var := temp
else
  high-var := temp
endif;
temp := 0
```

Compositional, Value-Dependent Security

```
var dMode                 // clas: Low
var high_var              // clas: High
var low_var, sMode        // clas: Low
var buffer                // clas: dMode == 0 ? Low : High
```

value-dependent classification

```
/* {dMode,sMode,temp} += AsmNoRW */
/* acquire (dMode == sMode) */
temp := buffer;
if sMode == 0 then
  low-var := temp
else
  high-var := temp
endif;
temp := 0
```

Compositional, Value-Dependent Security

control variable (must be statically Low)

```
var dMode                 // clas: Low
var high_var              // clas: High
var low_var, sMode        // clas: Low
var buffer                // clas: dMode == 0 ? Low : High
```

value-dependent classification

```
/* {dMode,sMode,temp} += AsmNoRW */
/* acquire (dMode == sMode) */
temp := buffer;
if sMode == 0 then
  low-var := temp
else
  high-var := temp
endif;
temp := 0
```

Compositional, Value-Dependent Security

```
var dMode              // clas: Low
var high_var           // clas: High
var low_var, sMode     // clas: Low
var buffer             // clas: dMode == 0 ? Low : High
```

```
/* {dMode,sMode,temp} += AsmNoRW */
/* acquire (dMode == sMode) */
temp := buffer;
if sMode == 0 then
  low-var := temp
else
  high-var := temp
endif;
temp := 0
```

Compositional, Value-Dependent Security

```
var dMode                 // clas: Low
var high_var              // clas: High
var low_var, sMode        // clas: Low
var buffer                // clas: dMode == 0 ? Low : High
```

```
/* {dMode,sMode,temp} += AsmNoRW */
/* acquire (dMode == sMode) */
temp := buffer;
if sMode == 0 t
  low-var := te
else
  high-var := t
endif;
temp := 0
```

dynamic, local assumptions about environment, for compositional reasoning

Compositional Reasoning

Compositional Reasoning

Dynamic Modes:

Compositional Reasoning

Dynamic Modes:

```
R :: state rel
```

Compositional Reasoning

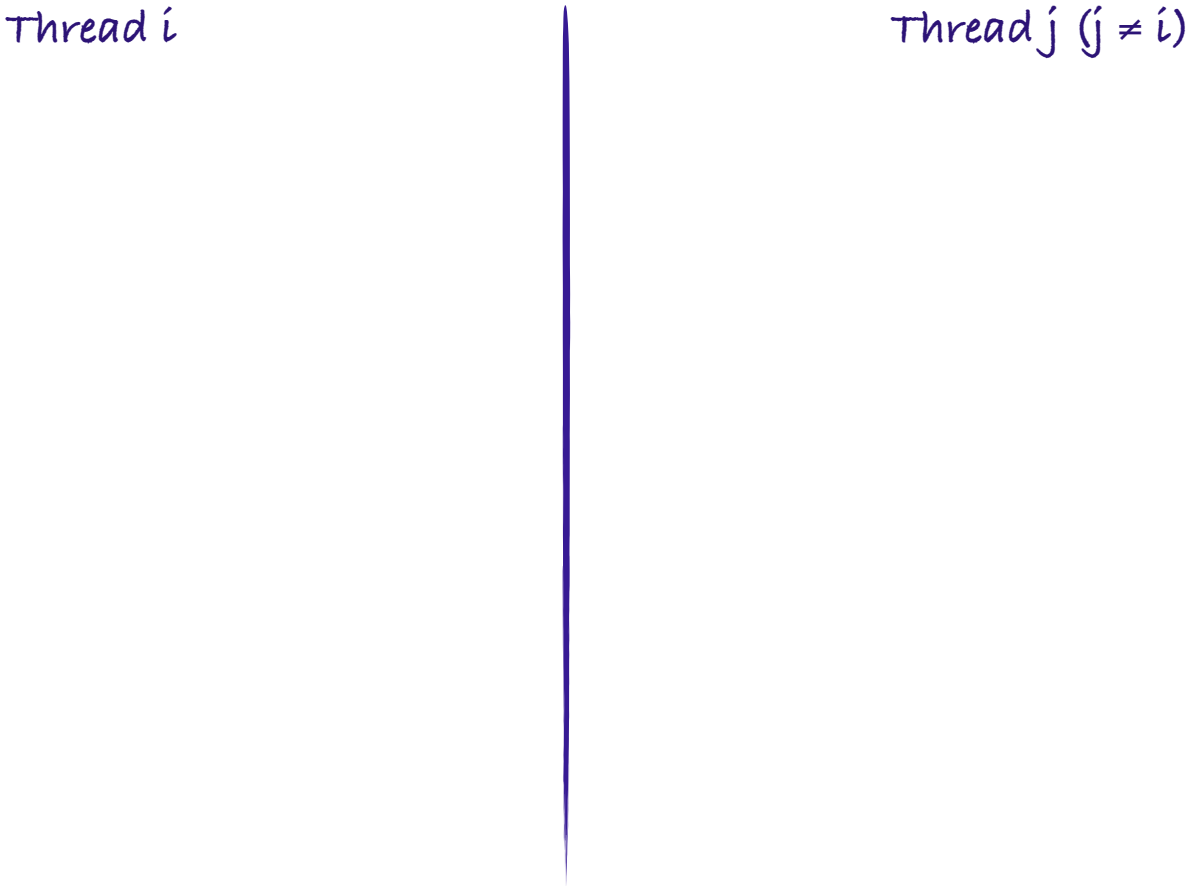
Dynamic Modes: `R :: state rel`

`NoReadOrWrite x`

Compositional Reasoning

Dynamic Modes: `R :: state rel`

`NoReadOrWrite x`



Compositional Reasoning

Dynamic Modes: `R :: state rel`

`NoReadOrWrite x`

Thread i

Thread j ($j \neq i$)

```
// assume: mode
```

Compositional Reasoning

Dynamic Modes: `R :: state rel`

`NoReadOrWrite x`

Thread i

```
// assume: mode

...
```

Thread j ($j \neq i$)

Compositional Reasoning

Dynamic Modes: `R :: state rel`

`NoReadOrWrite x`

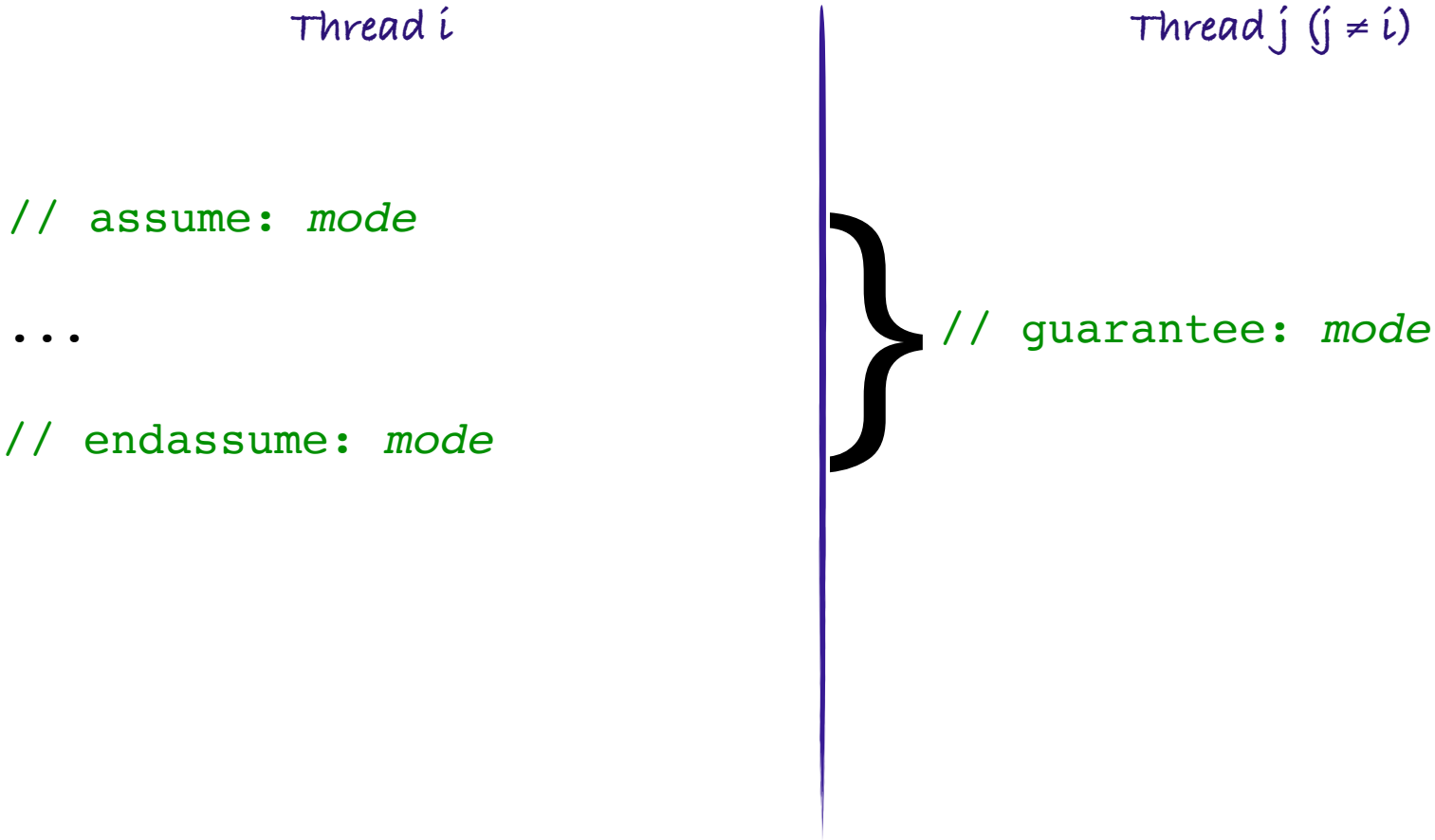
Thread i

```
// assume: mode

...

// endassume: mode
```

Thread j (j ≠ i)

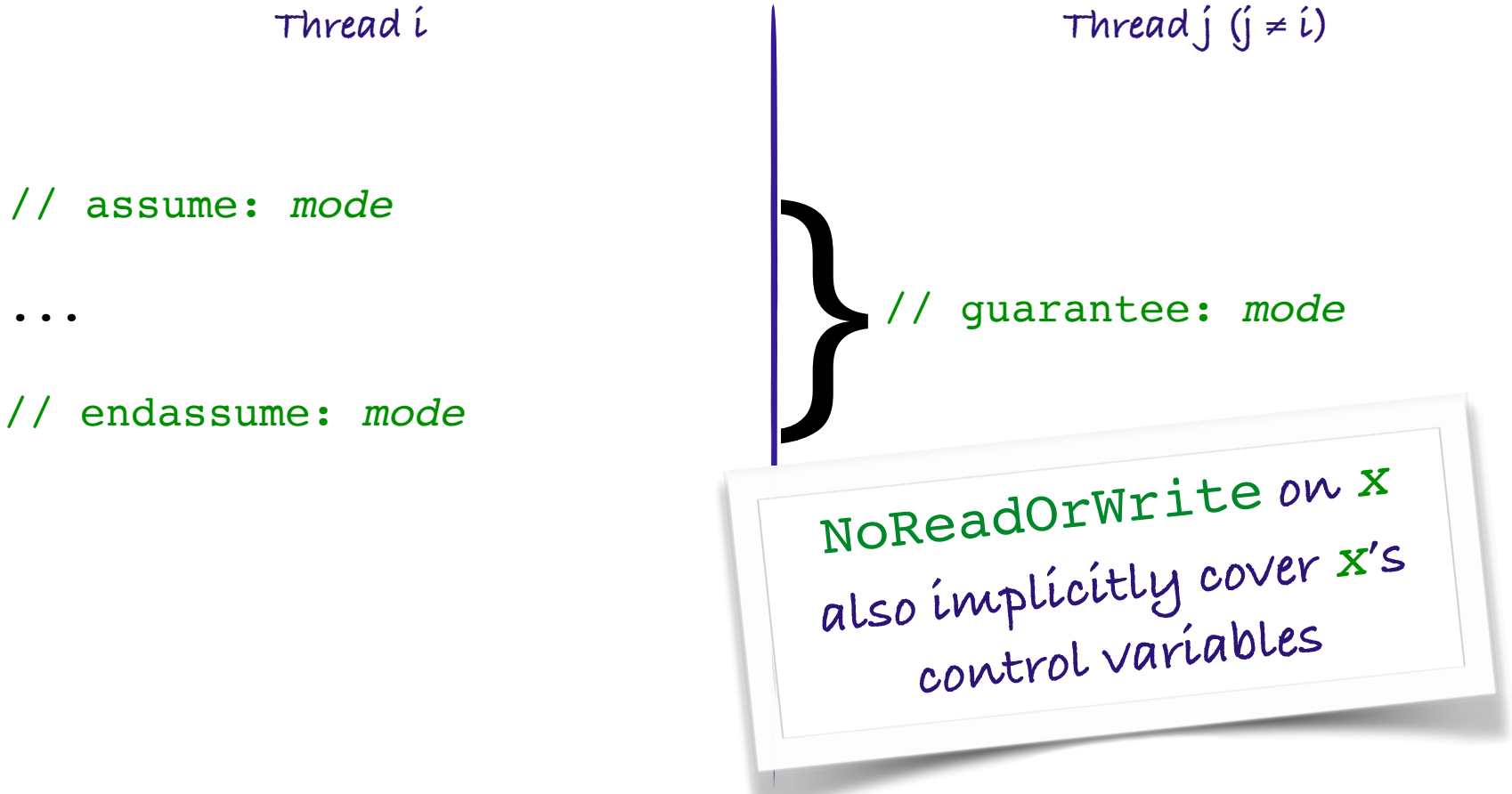
Compositional Reasoning

Dynamic Modes: `R :: state rel` `NoReadOrWrite x`



Compositional Reasoning

Dynamic Modes: `R :: state rel` `NoReadOrWrite x`



Compositional Reasoning

Dynamic Modes: `R :: state rel`

`NoReadOrWrite x`

Thread i

```
// assume: mode
...
// endassume: mode
```

Thread j (j ≠ i)

```
// guarantee: mode
```

If you `GuarNoReadOrWrite x` then your `assume R` can't mention `x`

`NoReadOrWrite` on `x` also implicitly cover `x`'s control variables

Semantics

Semantics

Local Configurations: (c, mds, mem)

Semantics

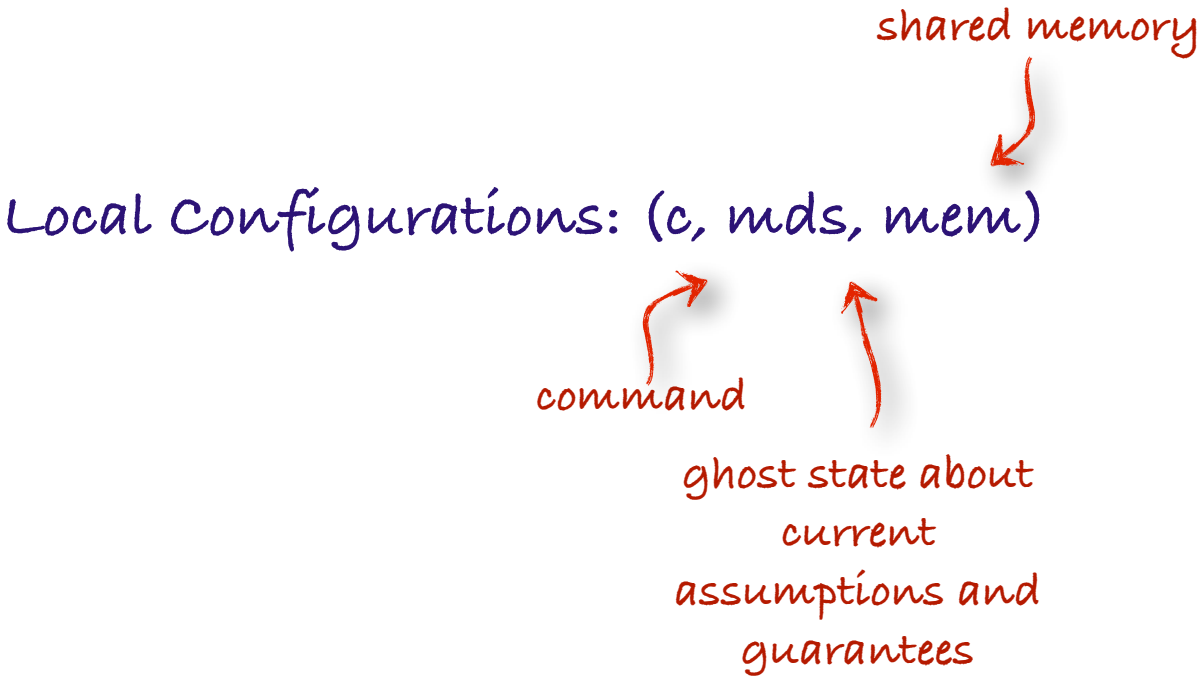
Local Configurations: (c, mds, mem)

Semantics

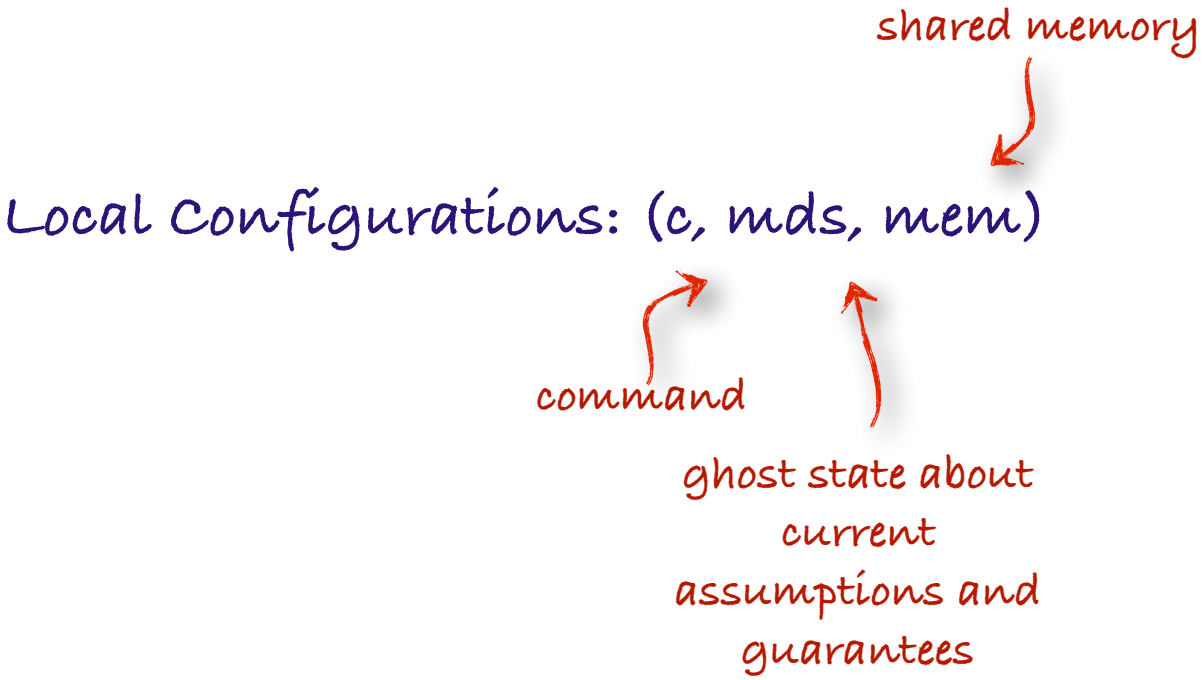
Local Configurations: (c, mds, mem)

ghost state about current assumptions and guarantees

Semantics


shared memory
Local Configurations: (c, mds, mem)
command
ghost state about current assumptions and guarantees

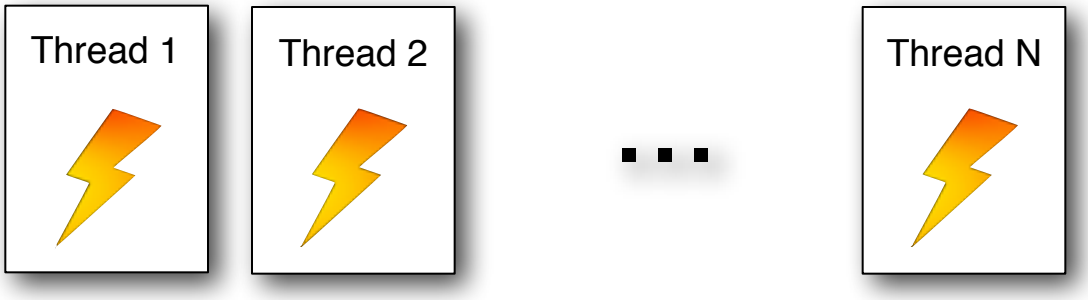
Semantics



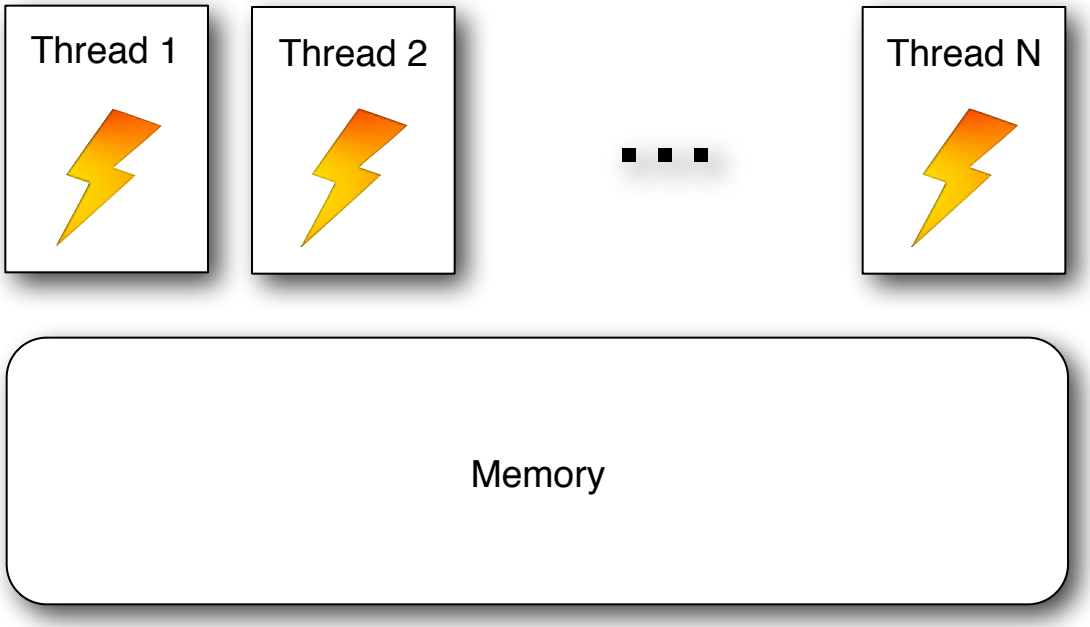
Global Configurations: ((c, mds) list, mem)

System Model

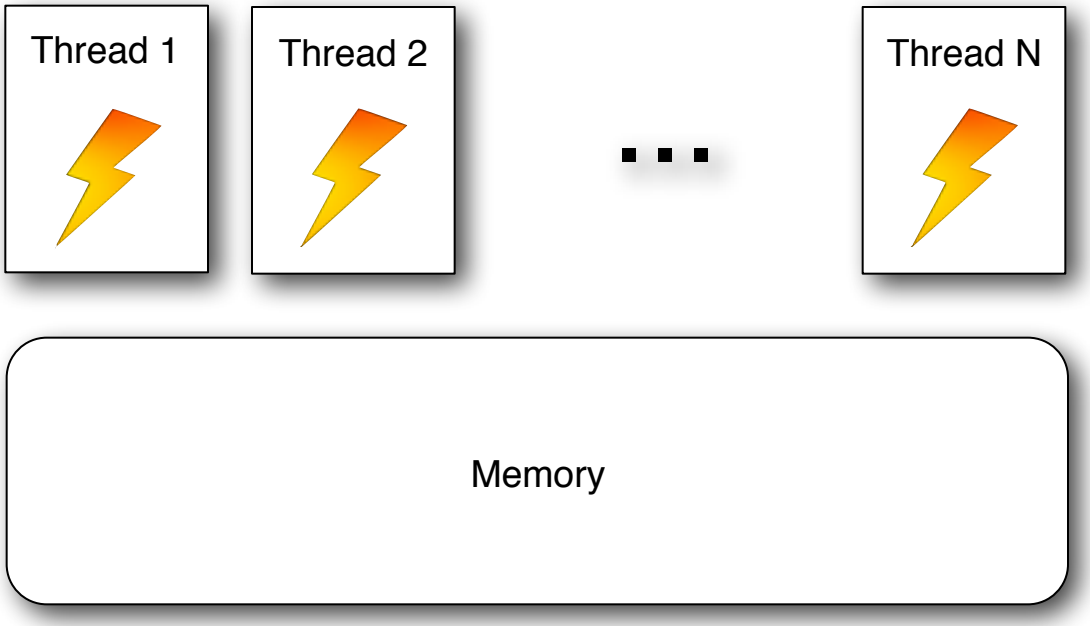
System Model



System Model

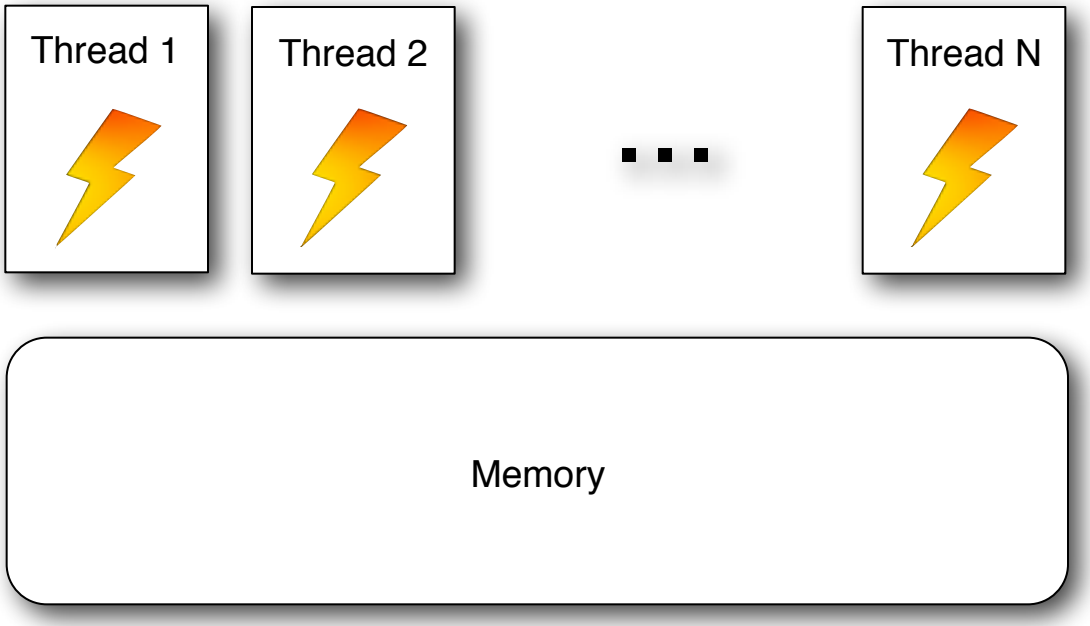


System Model

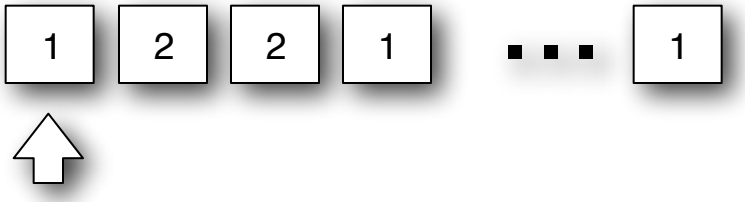


Arbitrary, Static Schedule:

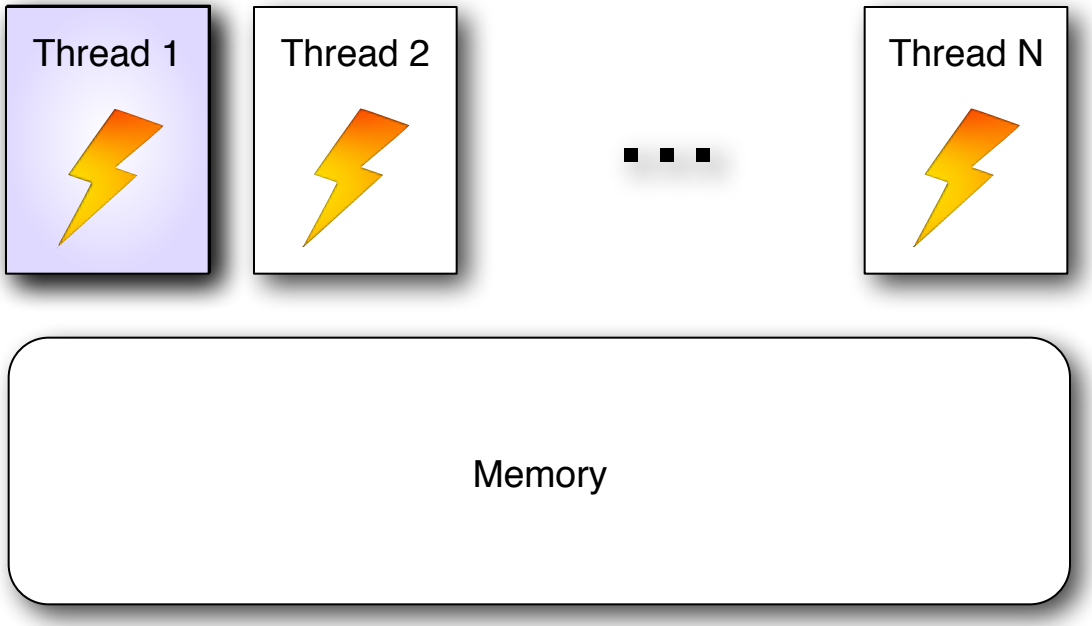
System Model



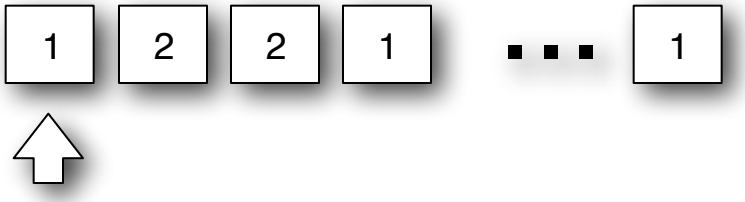
Arbitrary, Static Schedule:



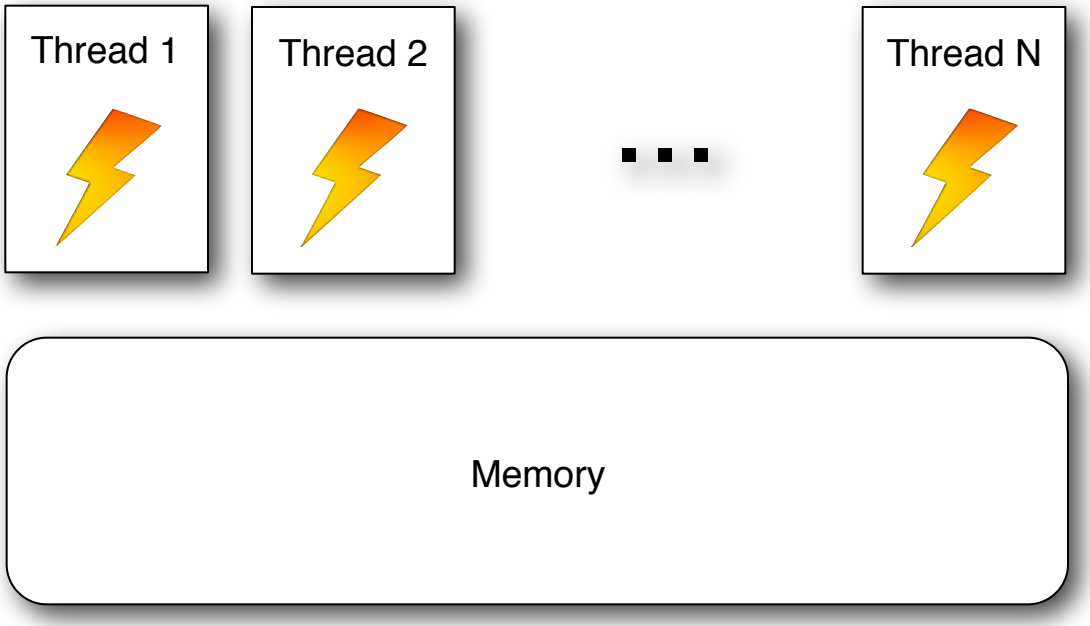
System Model



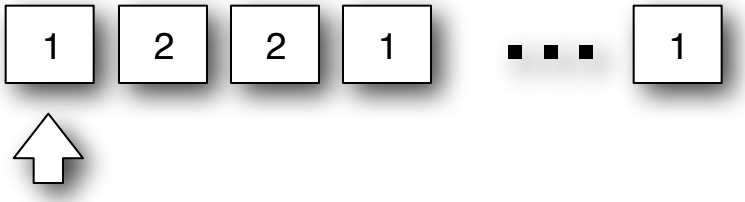
Arbitrary, Static Schedule:



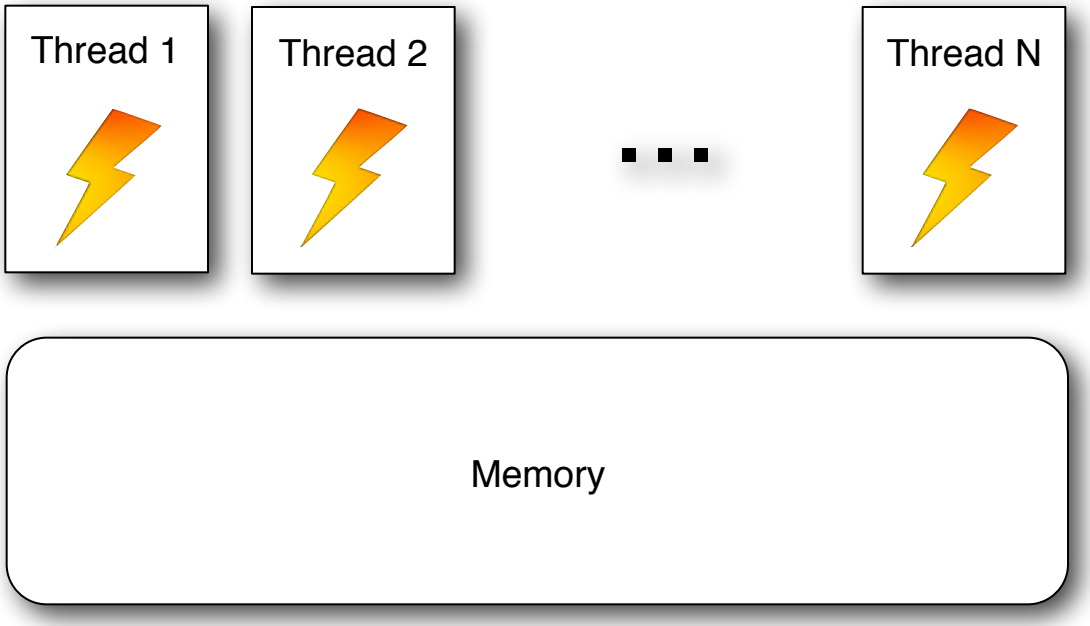
System Model



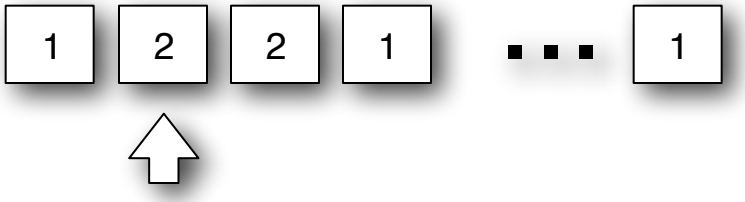
Arbitrary, Static Schedule:



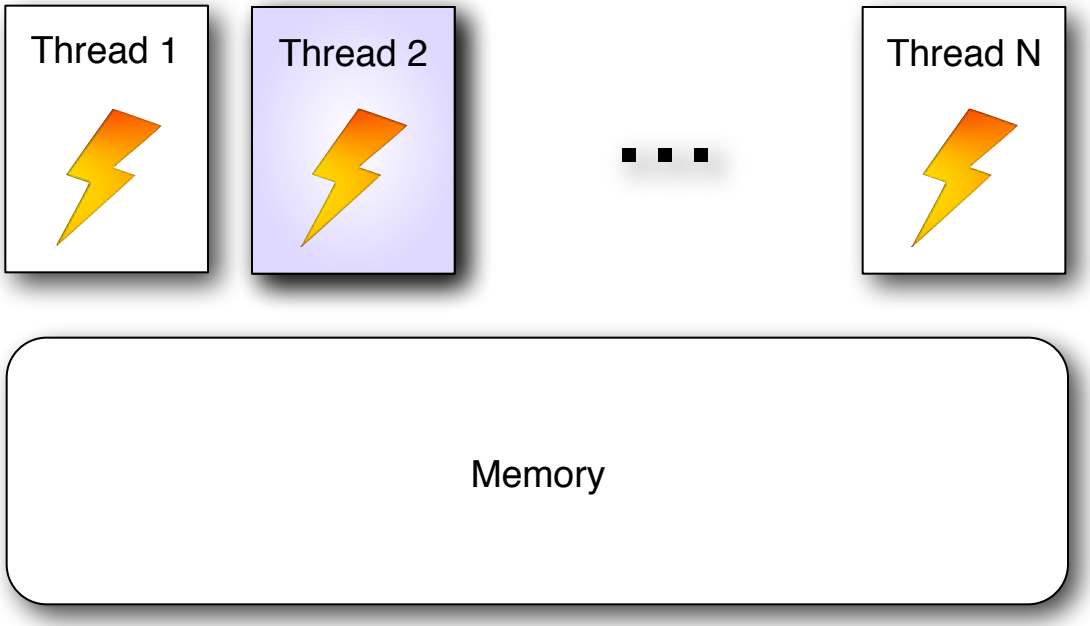
System Model



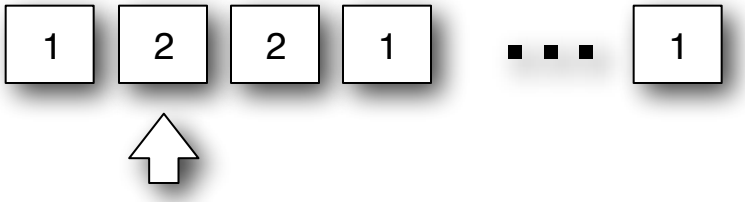
Arbitrary, Static Schedule:



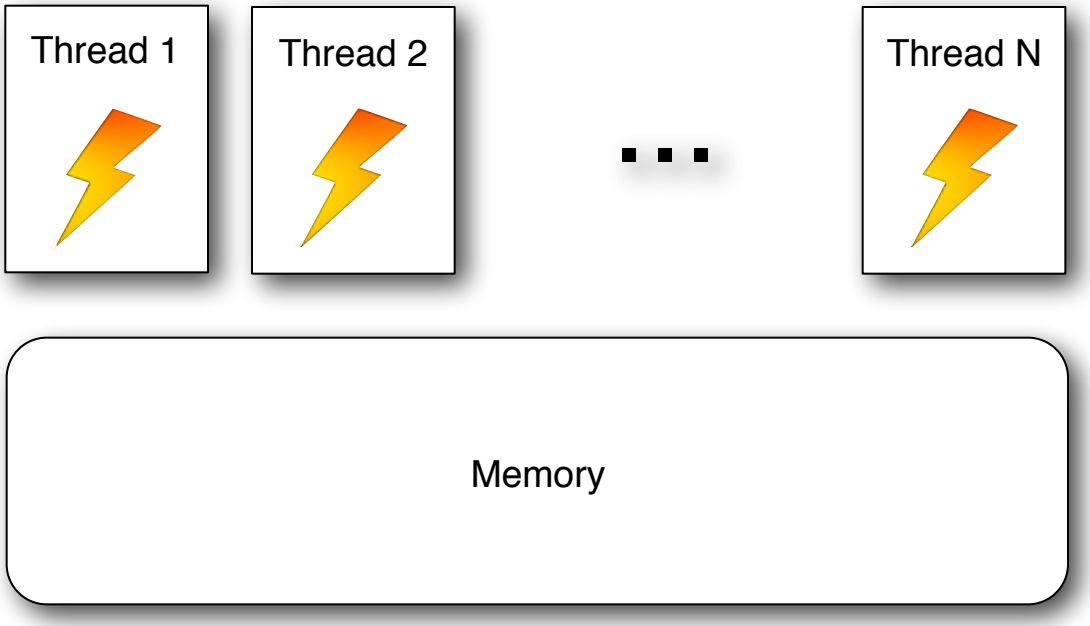
System Model



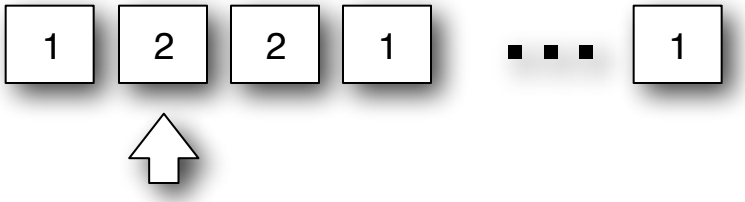
Arbitrary, Static Schedule:



System Model



Arbitrary, Static Schedule:

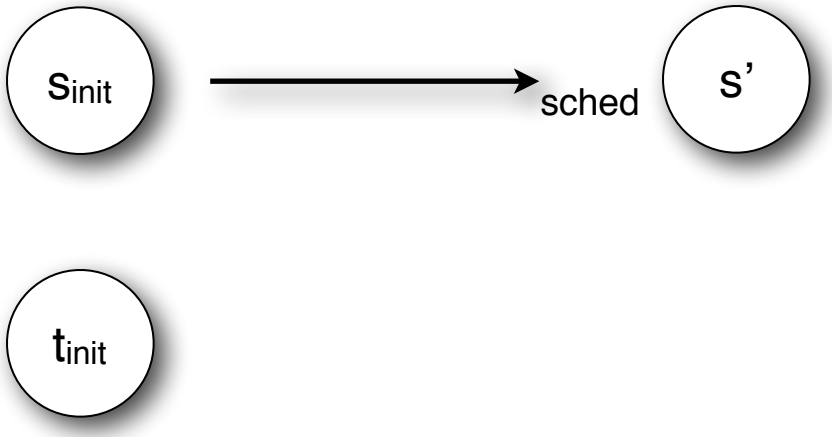


Global Security

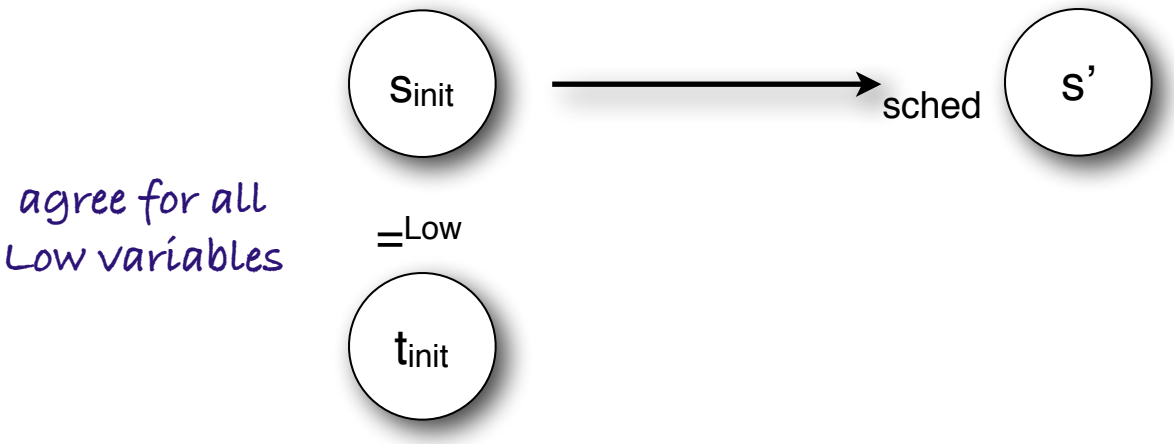
Global Security

Global Security

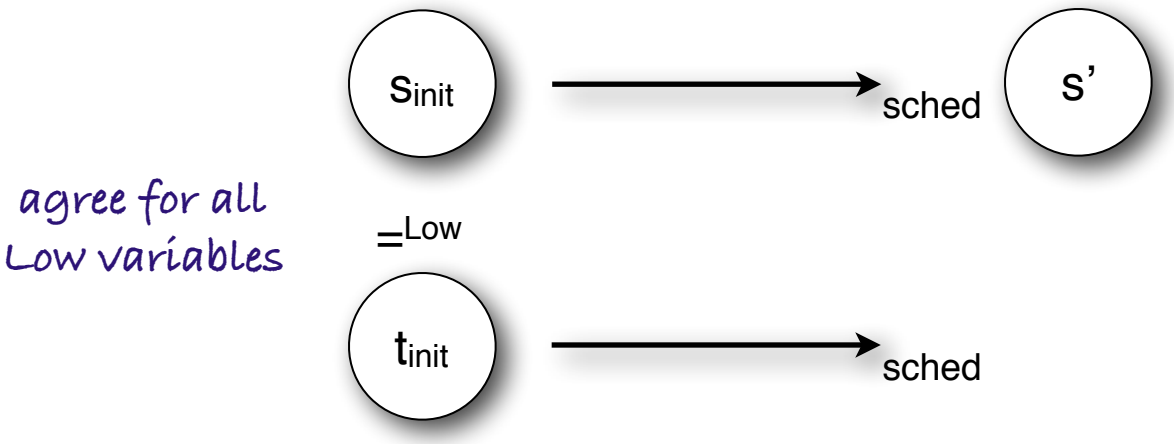
Global Security



Global Security

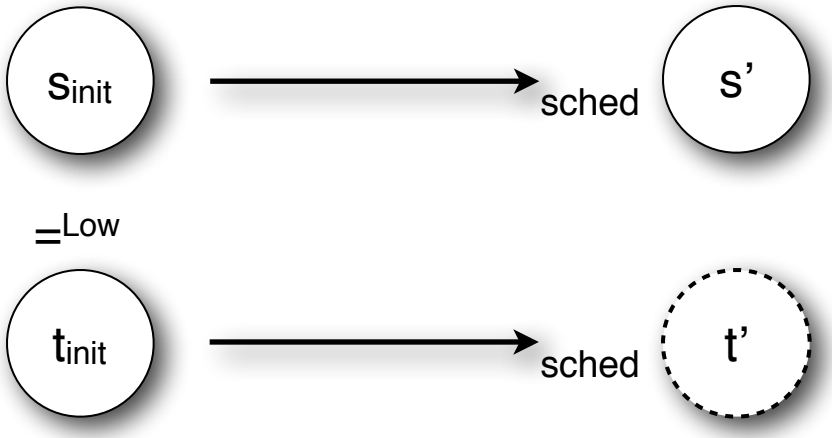


Global Security



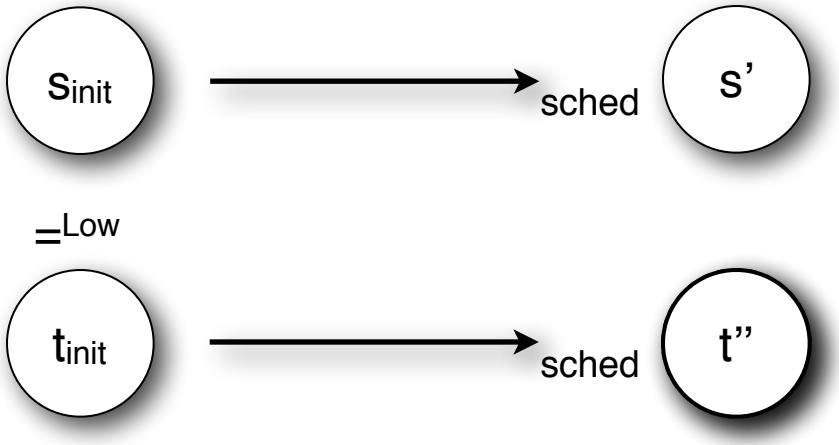
Global Security

agree for all Low variables

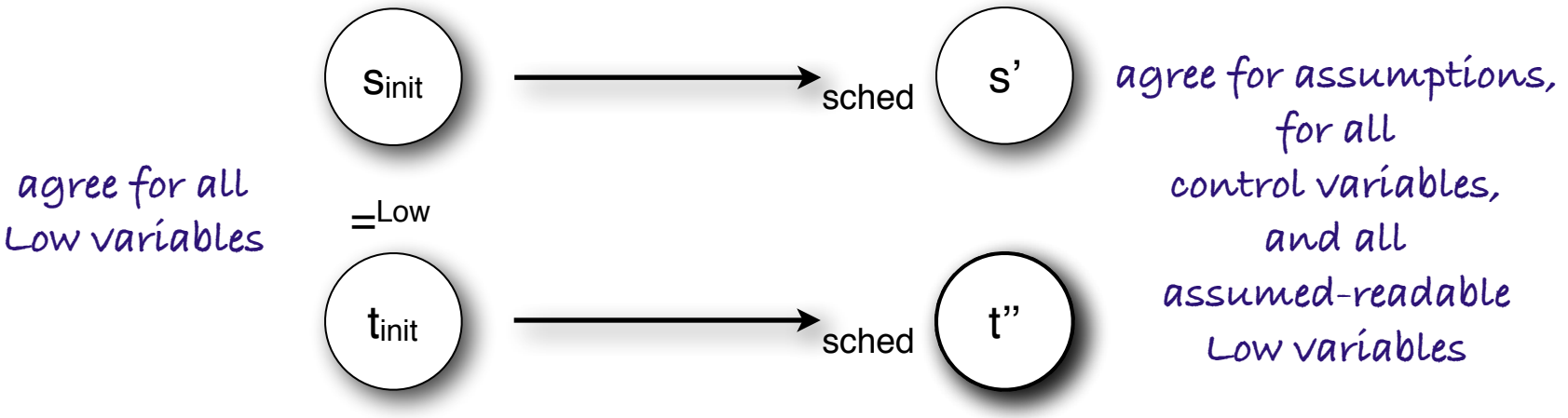


Global Security

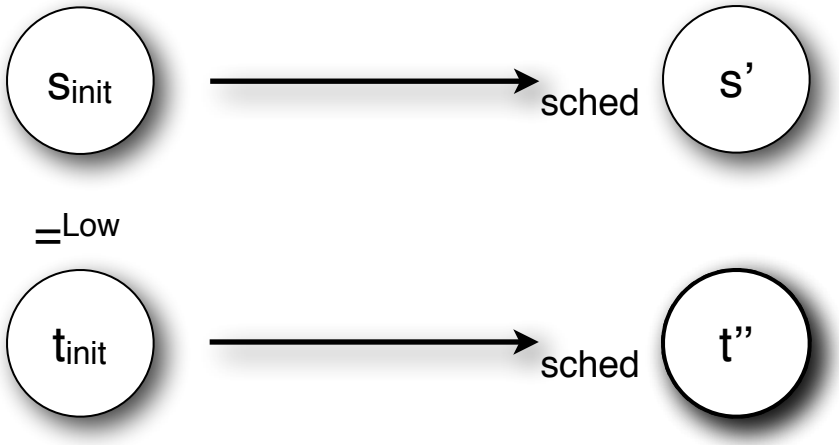
agree for all Low variables



Global Security



Global Security

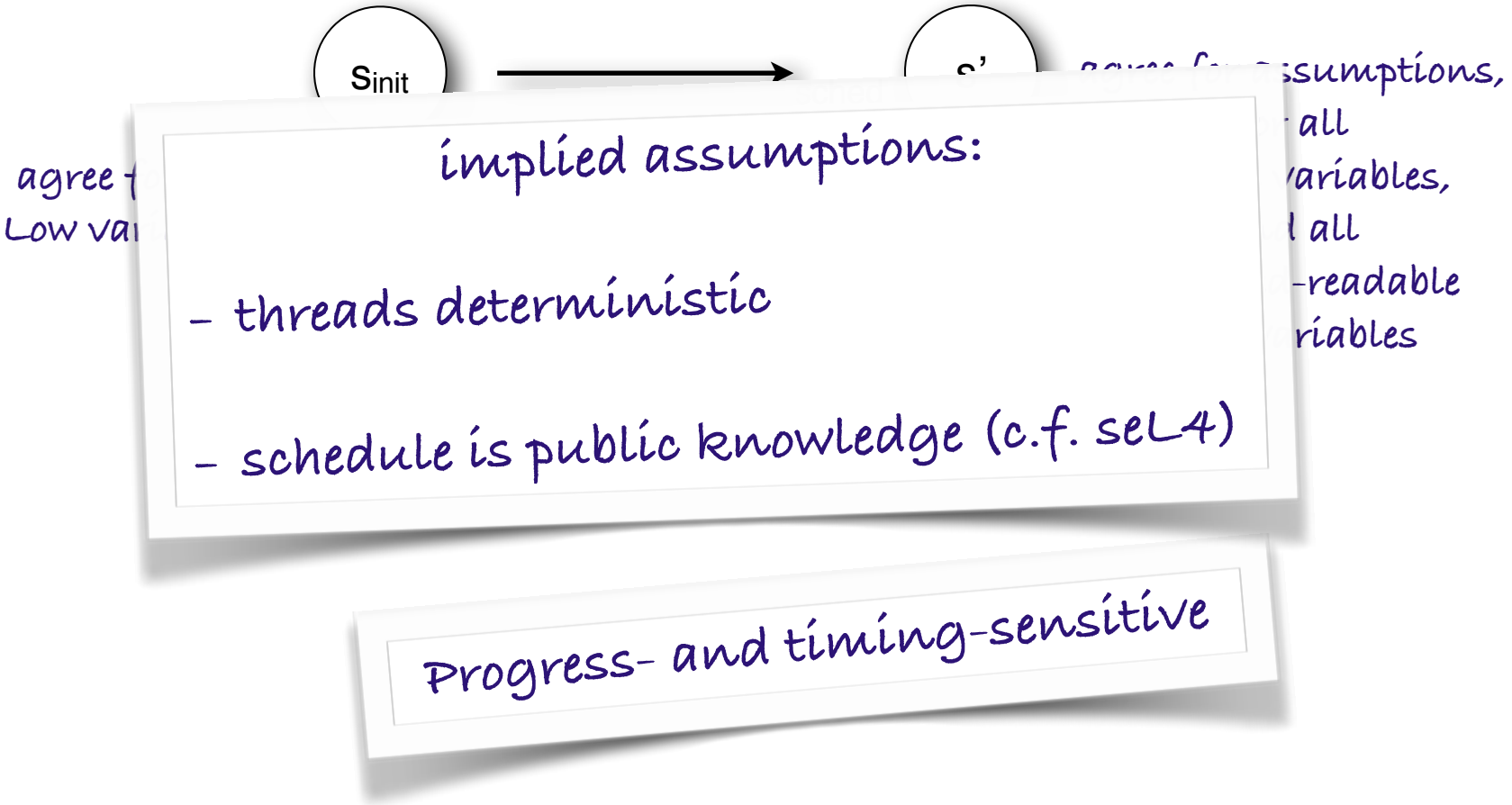


agree for all Low variables

agree for assumptions, for all control variables, and all assumed-readable Low variables

Progress- and timing-sensitive

Global Security



Low Equivalence

Noninterference is essentially preservation of Low-equivalence between all pairs of executions.

Low Equivalence

Noninterference is essentially preservation of Low-equivalence between all pairs of executions.

$$mem_1 =_{mds}^{l} mem_2 \equiv$$
$$\forall x.\ x \in \mathcal{C} \vee \mathcal{L}_{mem_1}\ x = \mathsf{Low} \wedge x \notin mds\ \mathbf{AsmNoRW} \longrightarrow$$
$$mem_1\ x = mem_2\ x$$

Low Equivalence

Noninterference is essentially preservation of Low-equivalence between all pairs of executions.

$$mem_1 =_{mds}^{l} mem_2 \equiv$$
$$\forall x.\ x \in \mathcal{C} \vee \mathcal{L}_{mem_1}\ x = \mathsf{Low} \wedge x \notin mds\ \mathbf{AsmNoRW} \longrightarrow$$
$$mem_1\ x = mem_2\ x$$

Two states are low equivalent when they agree on:

- All control variables (necessarily all Low)
- All currently-Low variables that might be read across a component boundary

Non-Readable Variables

Our security definition (and toy program language semantics, **with atomic expression evaluation**) would say this program is secure:

```
// {x,y,z} += AsmNoRW
if (h) {
  x := y + z;
  h := 0
} else {
  x := y;
  skip
}
x := 0;
y := 0;
z := 0;
// {x,y,z} -= AsmNoRW
```

Local Security: Intuition

Local Security: Intuition

Thread i

Local Security: Intuition

We need to prove:

Thread i

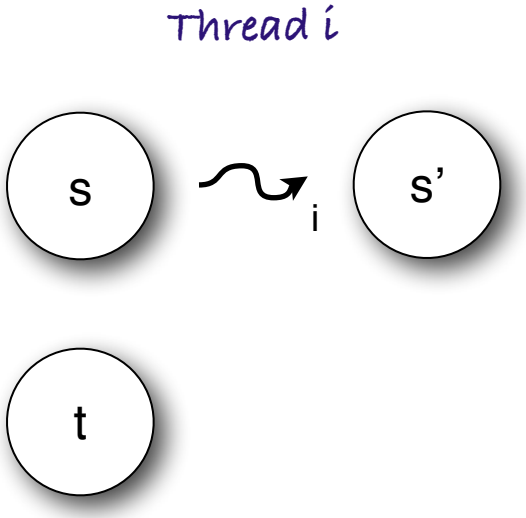
Local Security: Intuition

We need to prove:

Thread i

Local Security: Intuition

We need to prove:

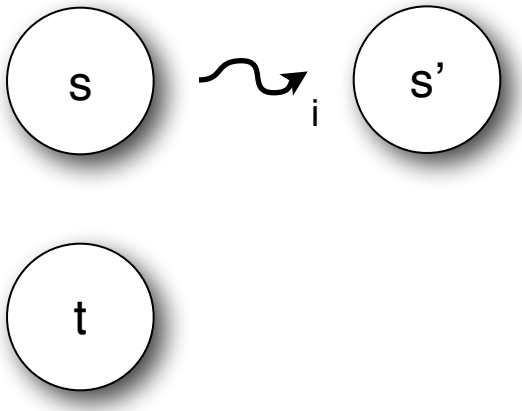


Local Security: Intuition

We need to prove:

Thread i

agree for assumptions,
for all
control variables,
and all
assumed-readable
Low variables

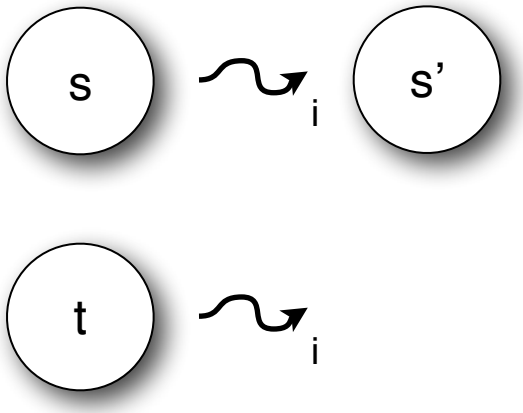


Local Security: Intuition

We need to prove:

Thread i

agree for assumptions,
for all
control variables,
and all
assumed-readable
Low variables

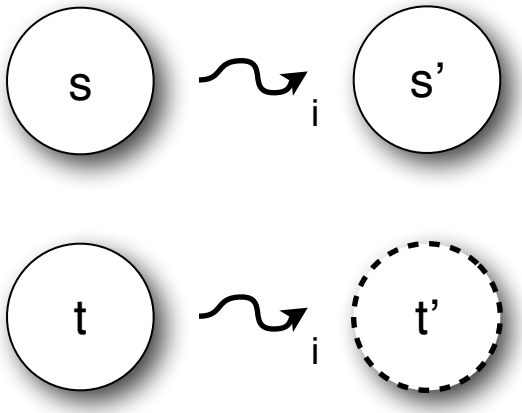


Local Security: Intuition

We need to prove:

Thread i

agree for assumptions,
for all
control variables,
and all
assumed-readable
Low variables

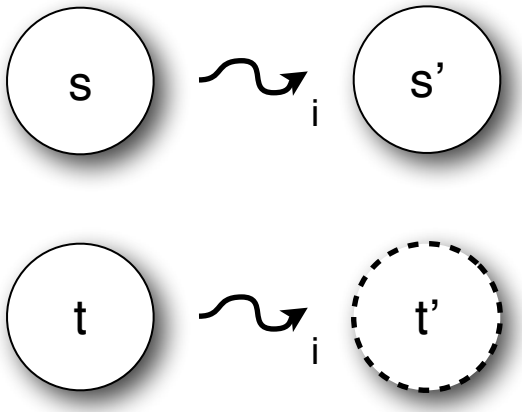


Local Security: Intuition

We need to prove:

Thread i

agree for assumptions, for all control variables, and all assumed-readable Low variables



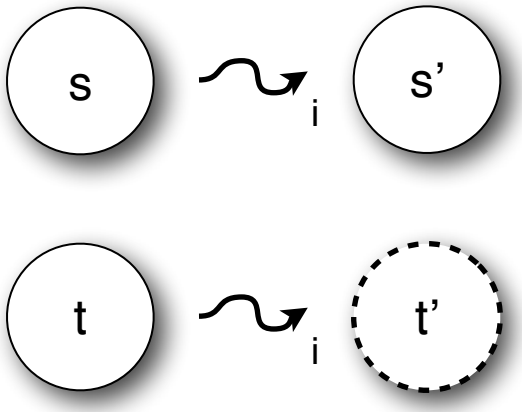
agree for assumptions, for all control variables, and all assumed-readable Low variables

Local Security: Intuition

We need to prove:

Thread i

agree for assumptions, for all control variables, and all assumed-readable Low variables



agree for assumptions, for all control variables, and all assumed-readable Low variables

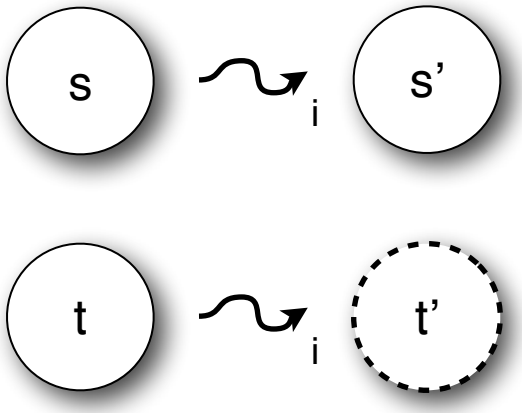
(keeping in mind that other threads could be interleaved)

Local Security: Intuition

We need to prove:

Thread i

agree for assumptions, for all control variables, and all assumed-readable Low variables



agree for assumptions, for all control variables, and all assumed-readable Low variables

(keeping in mind that other threads could be interleaved)

Proof technique:
(Rely-Guarantee
(bi)simulation

Local Security

Local Security

Find a relation $\mathcal{R}_i$ such that:

Local Security

Find a relation $\mathcal{R}_i$ such that:

Thread i

Local Security

Find a relation $\mathcal{R}_i$ such that:

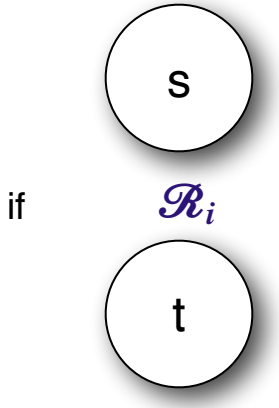
Thread i

if

Local Security

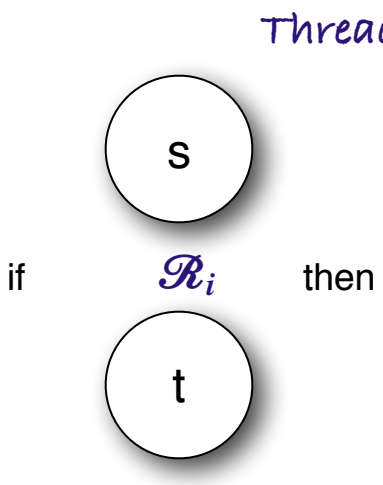
Find a relation $\mathcal{R}_i$ such that:

Thread i



Local Security

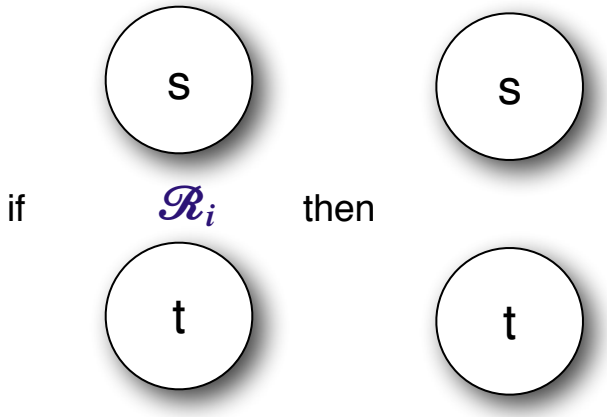
Find a relation $\mathscr{R}_i$ such that:



Local Security

Find a relation $\mathcal{R}_i$ such that:

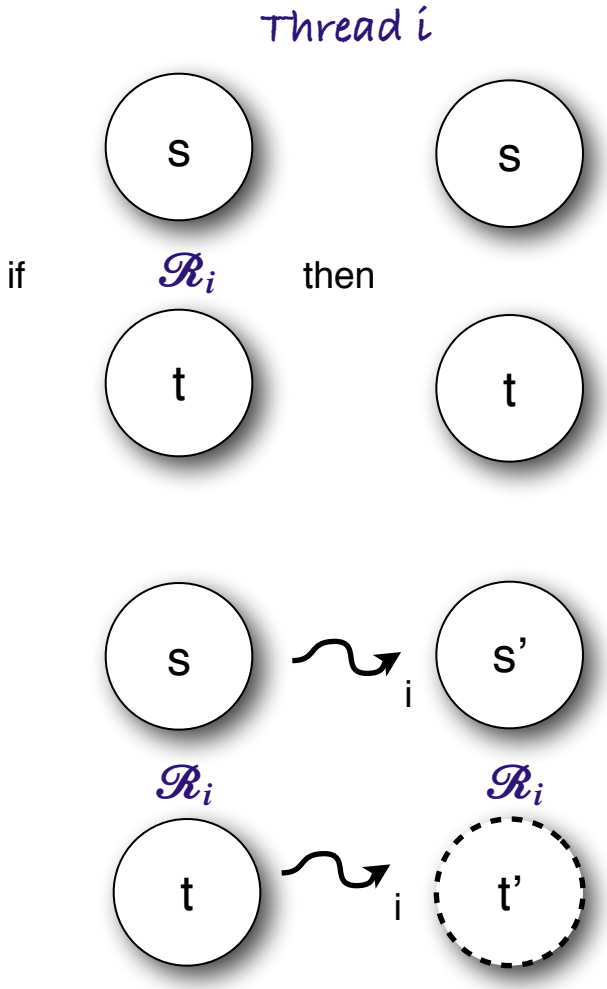
Thread i



agree for assumptions,
for all
control variables,
and all
assumed-readable
Low variables

Local Security

Find a relation $\mathcal{R}_i$ such that:

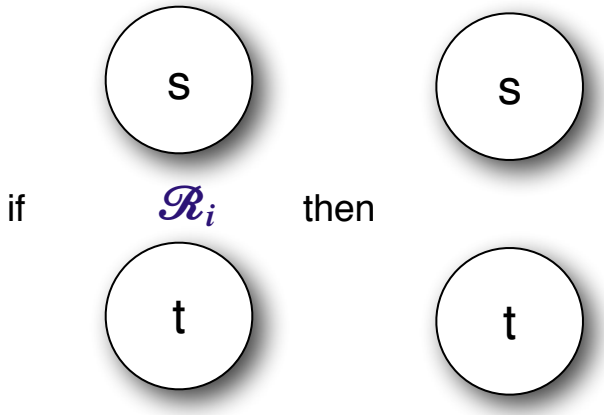


agree for assumptions,
for all
control variables,
and all
assumed-readable
Low variables

Local Security

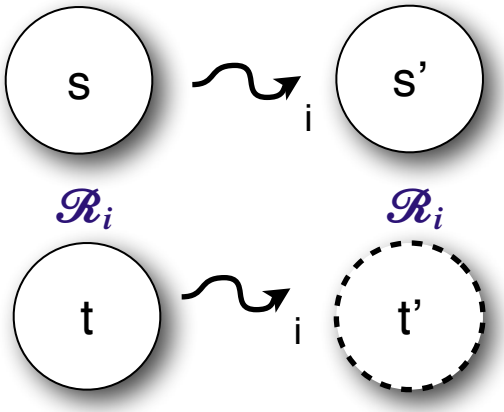
Find a relation $\mathcal{R}_i$ such that:

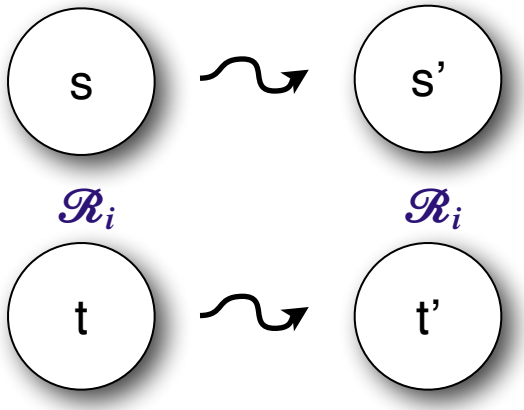
Thread i



agree for assumptions,
for all
control variables,
and all
assumed-readable
Low variables

Environment

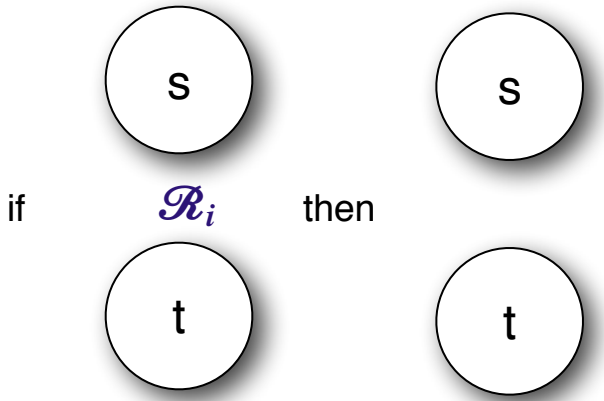




Local Security

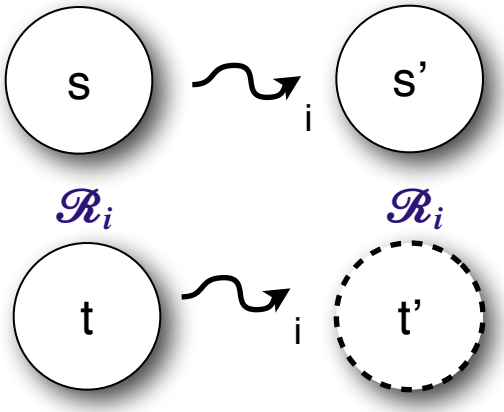
Find a relation $\mathcal{R}_i$ such that:

Thread i

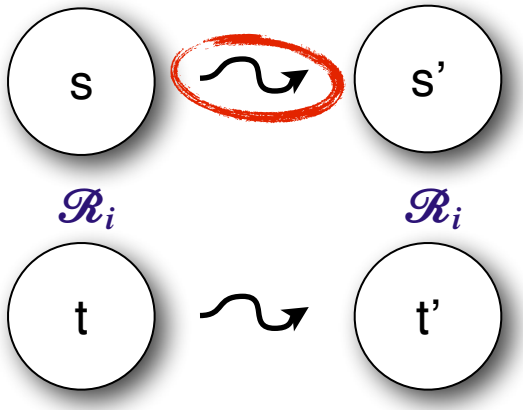


agree for assumptions,
for all
control variables,
and all
assumed-readable
Low variables

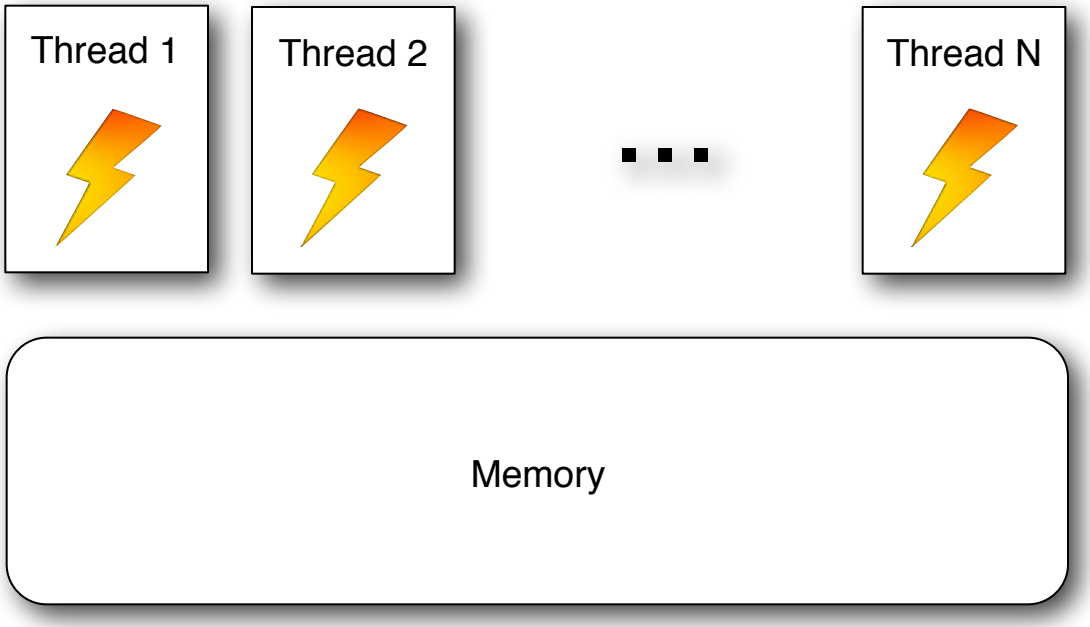
security-preserving
steps that
respect i's current
assumptions



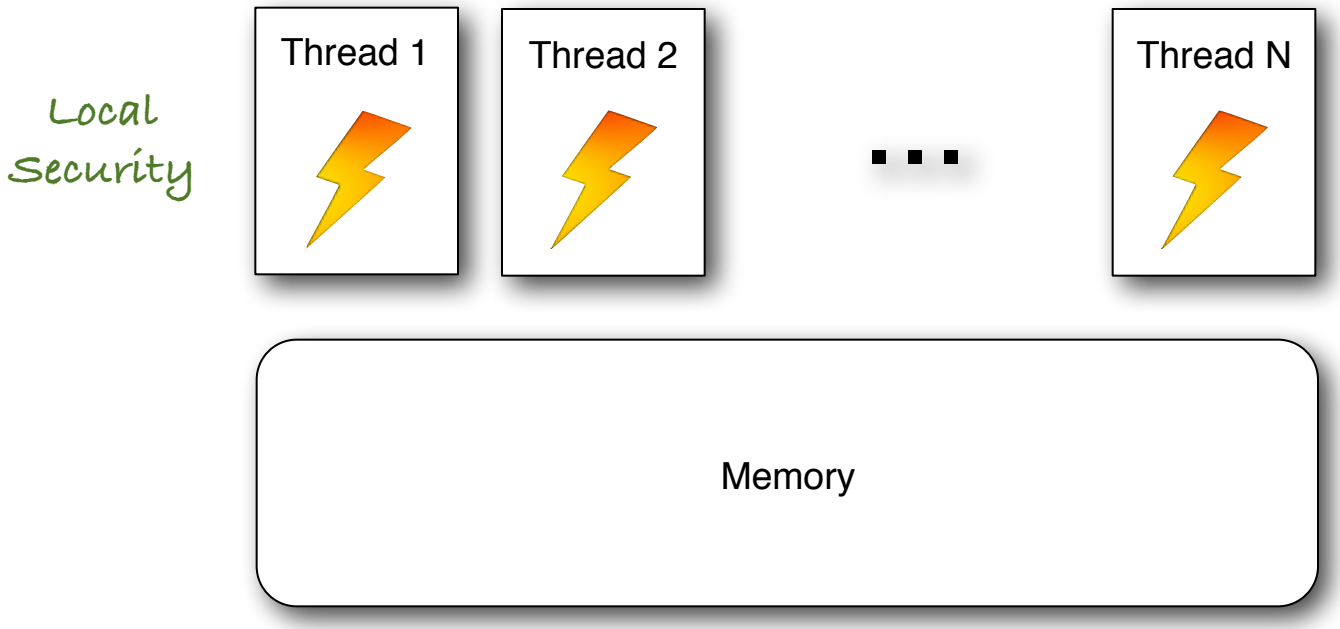
Environment



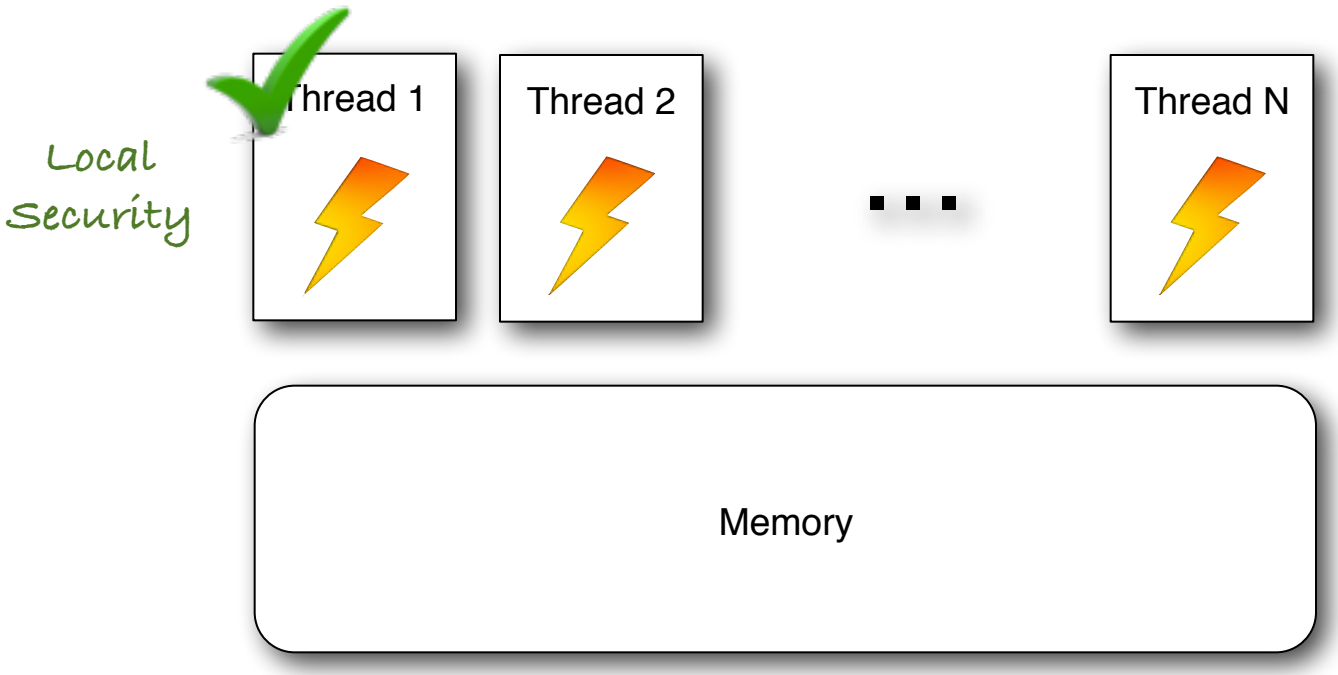
Compositionality Theorem



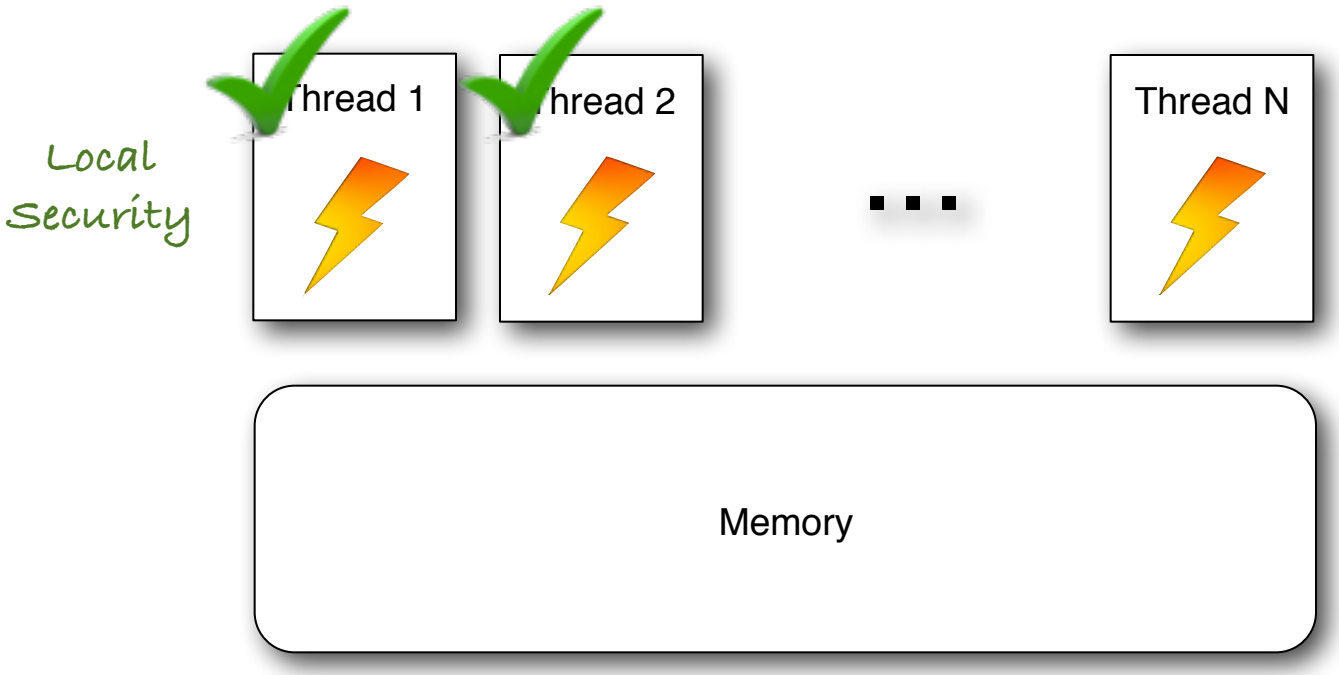
Compositionality Theorem



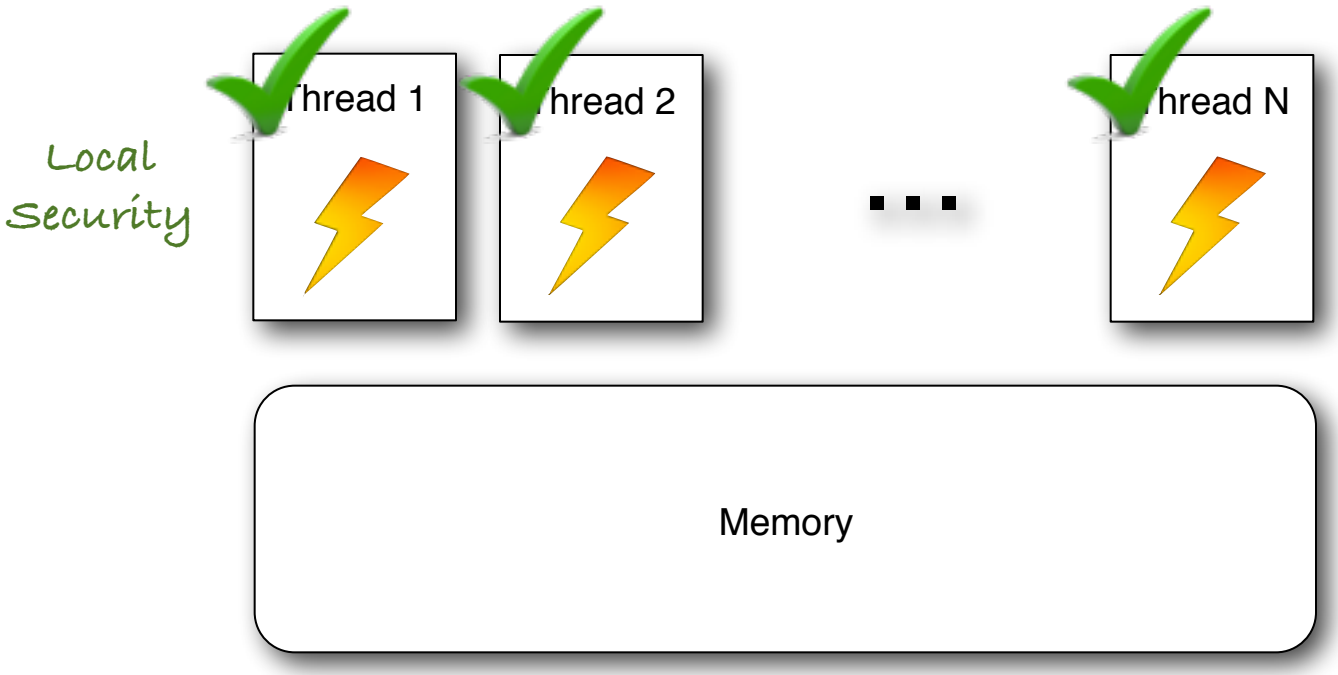
Compositionality Theorem



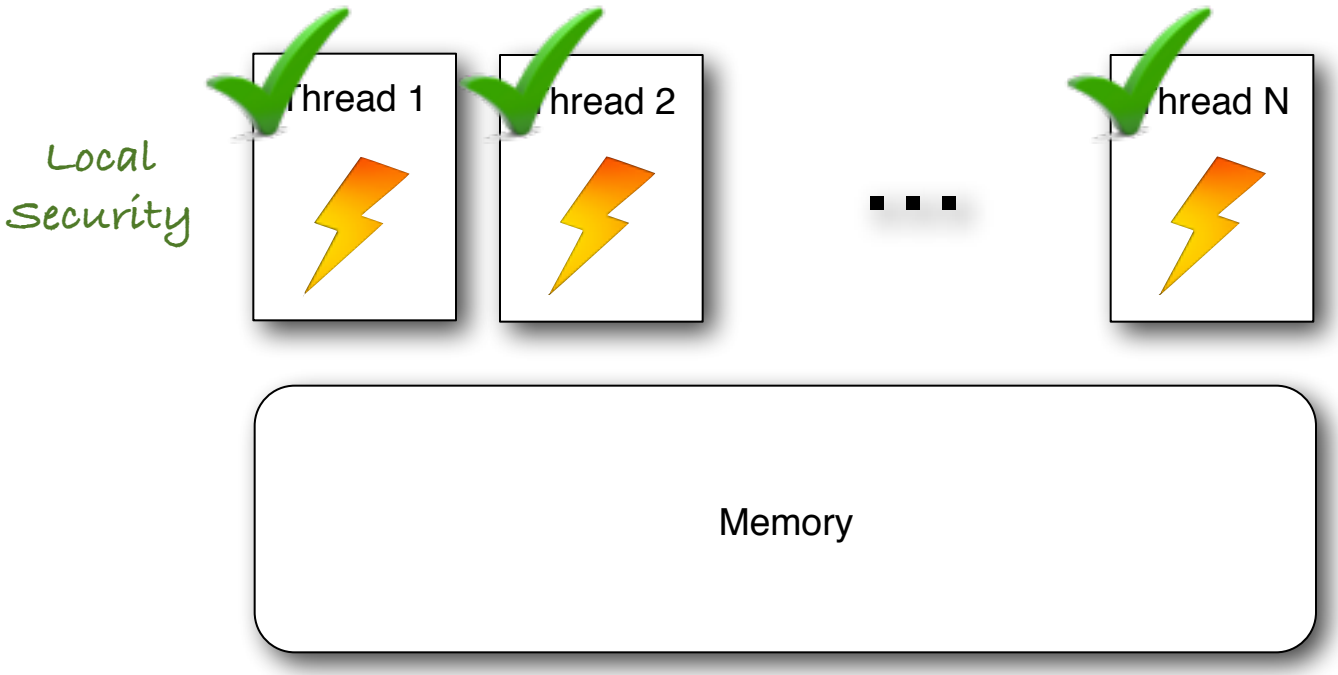
Compositionality Theorem



Compositionality Theorem

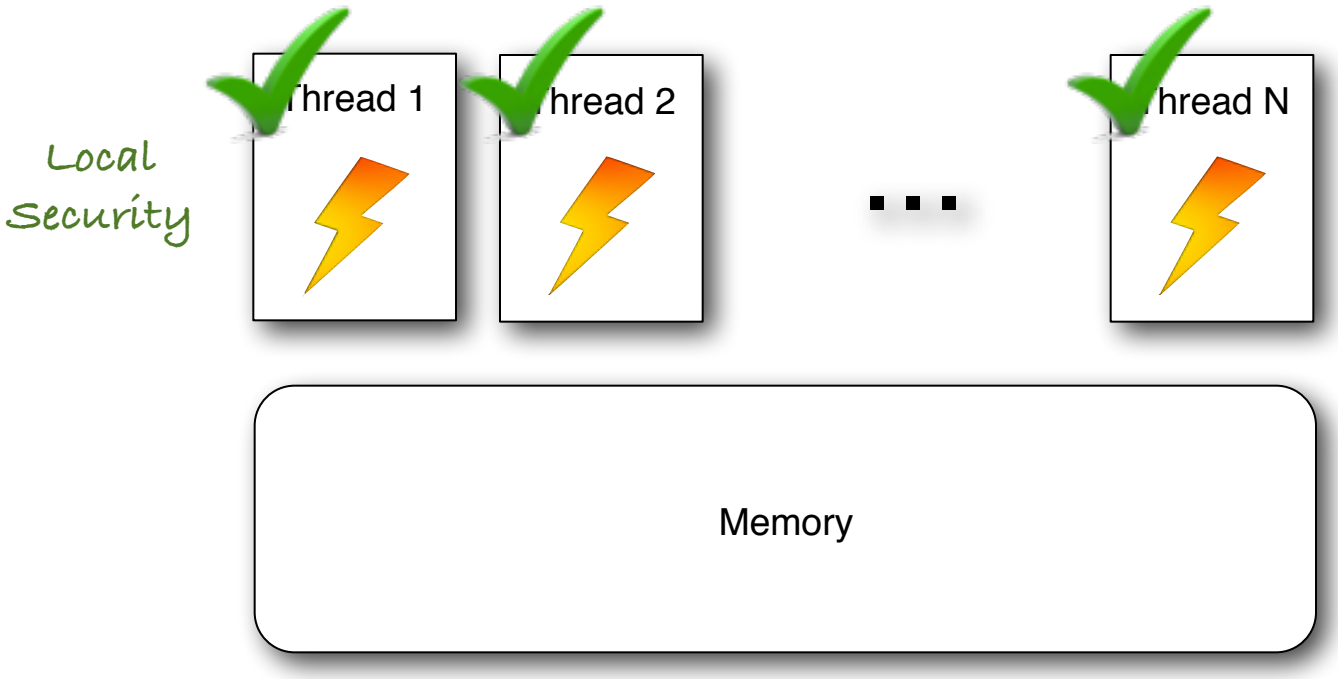


Compositionality Theorem



Each thread adheres to its own guarantees

Compositionality Theorem



Each thread adheres to its own guarantees

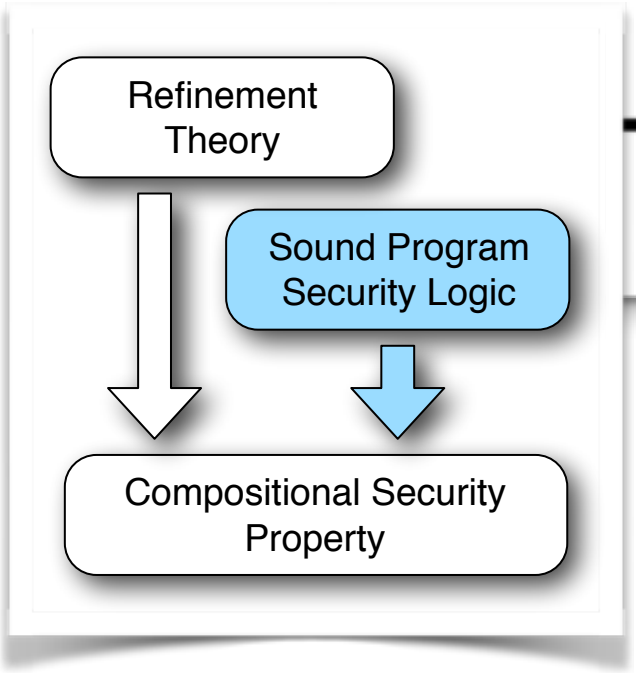
Each thread guarantees the assumptions of all others, at the right times

Compositionality Theorem

DEPENDENT SECURITY LOGIC

$$\Gamma \vdash e : t$$

DEPENDENT SECURITY LOGIC



$$\vdash e : t$$

Toy Language

cmd ::= **skip**
| x := E
| *cmd* ; *cmd*
| **if** E **then** *cmd* **else** *cmd*
| **while** E **do** *cmd* **done**
| **lock** l
| **unlock** l

Toy Language

cmd ::= **skip**
| x := E
| *cmd* ; *cmd*
| **if** E **then** *cmd* **else** *cmd*
| **while** E **do** *cmd* **done**
| **lock** l
| **unlock** l

lock_inv :: lock ⇒ (invariant x var set)

Toy Language

cmd ::= **skip**
| x := E
| *cmd* ; *cmd*
| **if** E **then** *cmd* **else** *cmd*
| **while** E **do** *cmd* **done**
| **lock** l
| **unlock** l

lock's footprint (set of shared variables that it protects)

lock_inv :: lock ⇒ (invariant x var set)

Toy Language

cmd ::= **skip**
| x := E
| *cmd* ; *cmd*
| **if** E **then** *cmd* **else** *cmd*
| **while** E **do** *cmd* **done**
| **lock** l
| **unlock** l

lock's footprint (set of shared variables that it protects)

lock_inv :: lock ⇒ (invariant x var set)

acquire invariant on acquiring l, must establish it when releasing l

Toy Language

cmd ::= **skip**
| x := E
| *cmd* ; *cmd*
| **if** E **then** *cmd* **else** *cmd*
| **while** E **do** *cmd* **done**
| **lock** l
| **unlock** l

lock's footprint (set of shared variables that it protects)

lock_inv :: lock ⇒ (invariant x var set)

acquire invariant on acquiring l, must establish it when releasing l
also acquire AsmNoRW on l's footprint

Toy Language

cmd ::= **skip**
| x := E
| *cmd* ; *cmd*
| **if** E **then** *cmd* **else** *cmd*
| **while** E **do** *cmd* **done**
| **lock** l
| **unlock** l

lock's footprint (set of shared variables that it protects)

lock_inv :: lock ⇒ (invariant x var set)

acquire invariant on acquiring l, must establish it when releasing l
also acquire AsmNoRW on l's footprint

global_invariant = ∀ l : lock, ¬ acquired l → (lock_inv l)

Toy Language

cmd ::= **skip**
| x := E
| *cmd* ; *cmd*
| **if** E **then** *cmd* **else** *cmd*
| **while** E **do** *cmd* **done**
| **lock** l
| **unlock** l

lock's footprint (set of shared variables that it protects)

lock_inv :: lock ⇒ (invariant x var set)

acquire invariant on acquiring l, must establish it when releasing l
also acquire AsmNoRW on l's footprint

global_invariant = ∀ l : lock, ¬ acquired l → (lock_inv l)

global, never changing rely condition:
R = {(s,s'). global_invariant s → global_invariant s'}

Toy Language

cmd ::= **skip**
| x := E
| *cmd* ; *cmd*
| **if** E **then** *cmd* **else** *cmd*
| **while** E **do** *cmd* **done**
| **lock** l
| **unlock** l

lock's footprint (set of shared variables that it protects)

lock_inv :: lock ⇒ (invariant x var set)

acquire invariant on acquiring l, must establish it when releasing l
also acquire AsmNoRW on l's footprint

global_invariant = ∀ l : lock, ¬ acquired l → (lock_inv l)

global, never changing rely condition:
R = {(s,s'). global_invariant s → global_invariant s'}

this is also the global, never changing guarantee condition for everyone

Types and Typing Judgements

$$\vdash \Gamma, \mathcal{S}, P \ \{c\} \ \Gamma', \mathcal{S}', P'$$

Types and Typing Judgements

$$\vdash \Gamma, \mathcal{S}, P \ \{c\} \ \Gamma', \mathcal{S}', P'$$

typing context

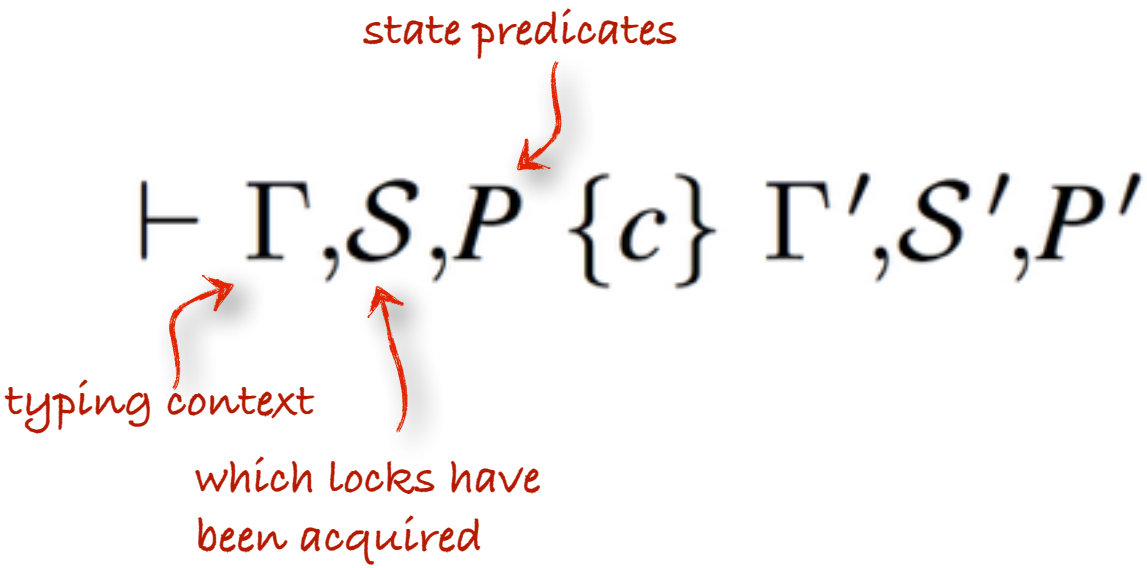
Types and Typing Judgements

$$\vdash \Gamma, \mathcal{S}, P \; \{c\} \; \Gamma', \mathcal{S}', P'$$

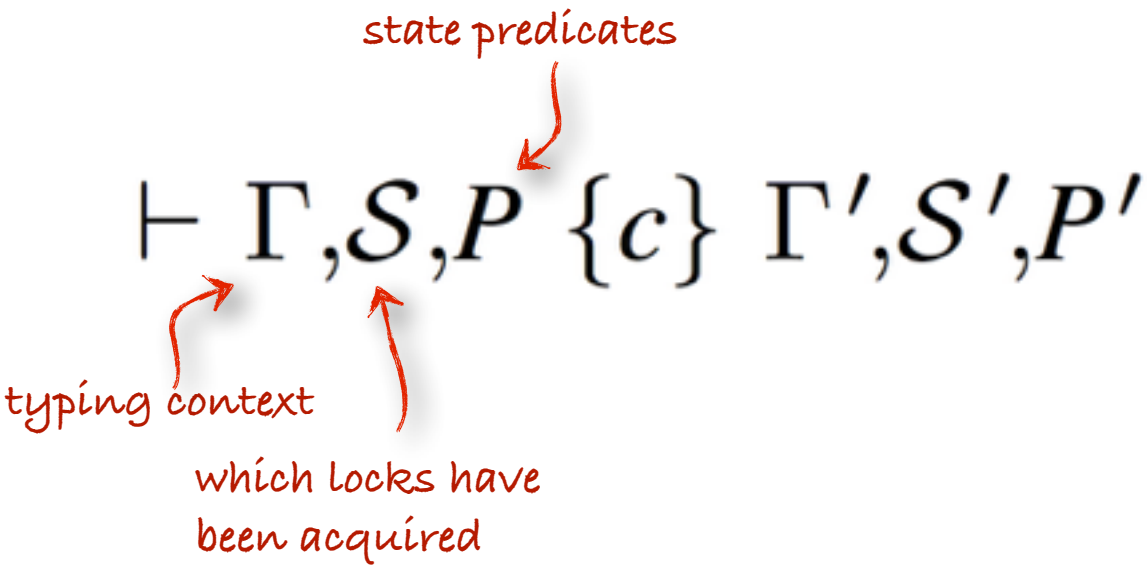
typing context

which locks have been acquired

Types and Typing Judgements

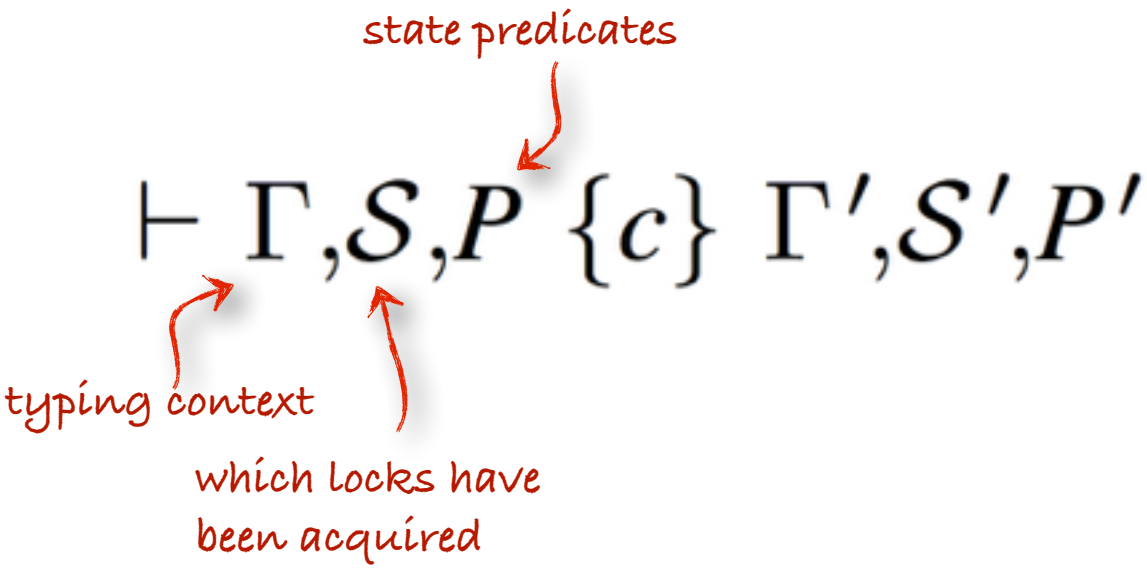


Types and Typing Judgements



types now depend on memory

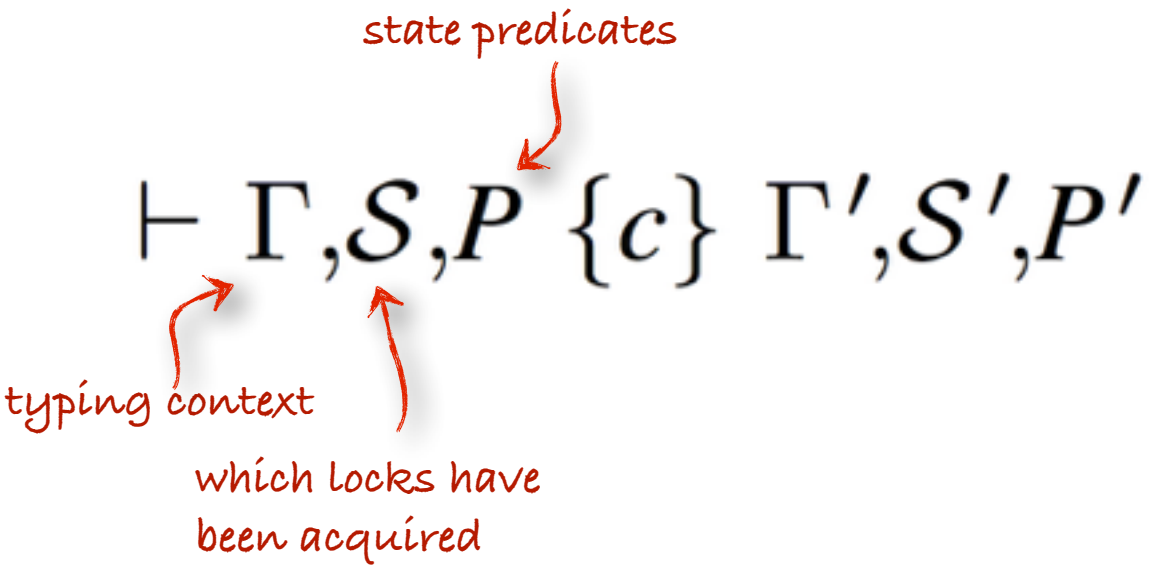
Types and Typing Judgements



types now depend on memory

we encode them as (sets of) state predicates

Types and Typing Judgements

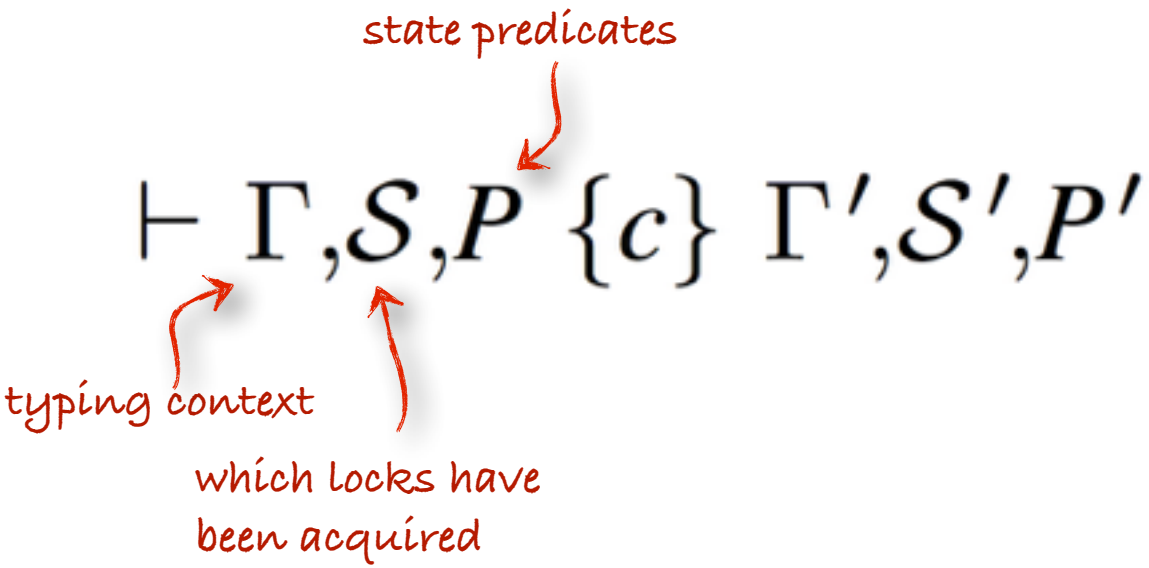


types now depend on memory

we encode them as (sets of) state predicates

type interpretation: $[\![t]\!]_{mem} \equiv \mathit{if}\ \lfloor t \rfloor_{mem}\ \mathit{then}\ \mathsf{Low}\ \mathit{else}\ \mathsf{High}$

Types and Typing Judgements



types now depend on memory

we encode them as (sets of) state predicates

type interpretation: $[\![t]\!]_{mem} \equiv \mathit{if}\ \lfloor t \rfloor_{mem}\ \mathit{then}\ \mathsf{Low}\ \mathit{else}\ \mathsf{High}$

subtype: $t \leq :_P t' \equiv (P \cup t') \vdash t$

Type System: Example in Action

```
lock(l);

temp := buffer;

if sMode == 0 then

  low-var := temp

else

  high-var := temp

endif;

temp := 0
```

lock l protects `sMode`, `dMode` and `temp`, with invariant `sMode` = `dMode`

Type System: Example in Action

Γ	$\mathcal{S}$	P	
[]	{}	{}	`lock(l);`
			`temp := buffer;`
			`if sMode == 0 then`
			`  low-var := temp`
			`else`
			`  high-var := temp`
			`endif;`
			`temp := 0`

lock l protects `sMode`, `dMode` and `temp`, with invariant `sMode` = `dMode`

Type System: Example in Action

[t : {}] {l} {sM = dM}

Γ $\mathcal{S}$ P

```
lock(l);
temp := buffer;
if sMode == 0 then
  low-var := temp
else
  high-var := temp
endif;
temp := 0
```

lock l protects `sMode`, `dMode` and `temp`, with invariant `sMode` = `dMode`

Type System: Example in Action

[t : {dM = 0}] {l} {sM = dM}

Γ $\mathcal{S}$ P

```
lock(l);
temp := buffer;
if sMode == 0 then
  low-var := temp
else
  high-var := temp
endif;
temp := 0
```

lock l protects `sMode`, `dMode` and `temp`, with invariant `sMode` = `dMode`

Type System: Example in Action

[t : {dM=0}] {l} {sM=dM, sM=0}

$\Gamma \quad \mathcal{S} \quad P$

```
lock(l);
temp := buffer;
if sMode == 0 then
  low-var := temp
else
  high-var := temp
endif;
temp := 0
```

lock l protects `sMode`, `dMode` and `temp`, with invariant `sMode` = `dMode`

Type System: Example in Action

[t : {dM=0}] {l} {sM=dM, sM=0}

Γ $\mathcal{S}$ P

```
lock(l);

temp := buffer;

if sMode == 0 then

  low-var := temp

else

  high-var := temp

endif;

temp := 0
```

Is {dM=0} a subtype of {}, under {sM=dM, sM=0} ?

lock l protects `sMode`, `dMode` and `temp`, with invariant `sMode` = `dMode`

Type System: Example in Action

[t : {dM=0}] {l} {sM=dM, sM=0}

Γ $\mathcal{S}$ P

```
lock(l);

temp := buffer;

if sMode == 0 then

  low-var := temp

else

  high-var := temp

endif;

temp := 0
```

{sM=dM,sM=0},{} ⊢ {dM=0} ?

lock l protects `sMode`, `dMode` and `temp`, with invariant `sMode` = `dMode`

Type System: Example in Action

```
lock(l);

temp := buffer;

if sMode == 0 then

  low-var := temp

else

  high-var := temp

endif;

temp := 0
```

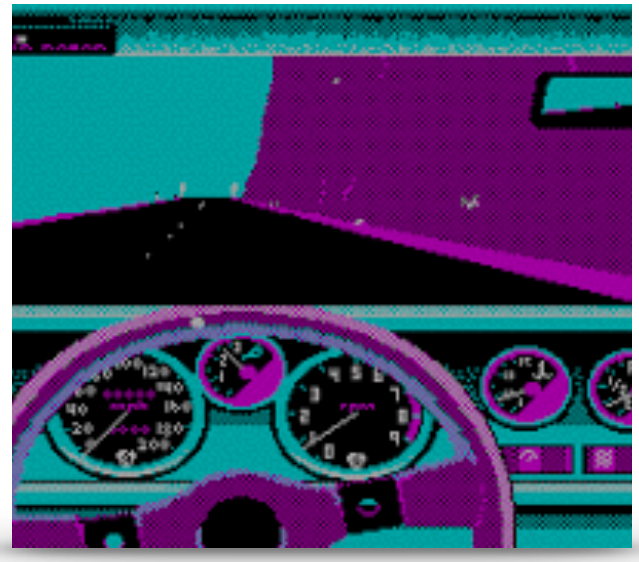
{sM=dM,sM=0},{} ⊢ {dM=0} ?

lock l protects `sMode`, `dMode` and `temp`, with invariant `sMode` = `dMode`

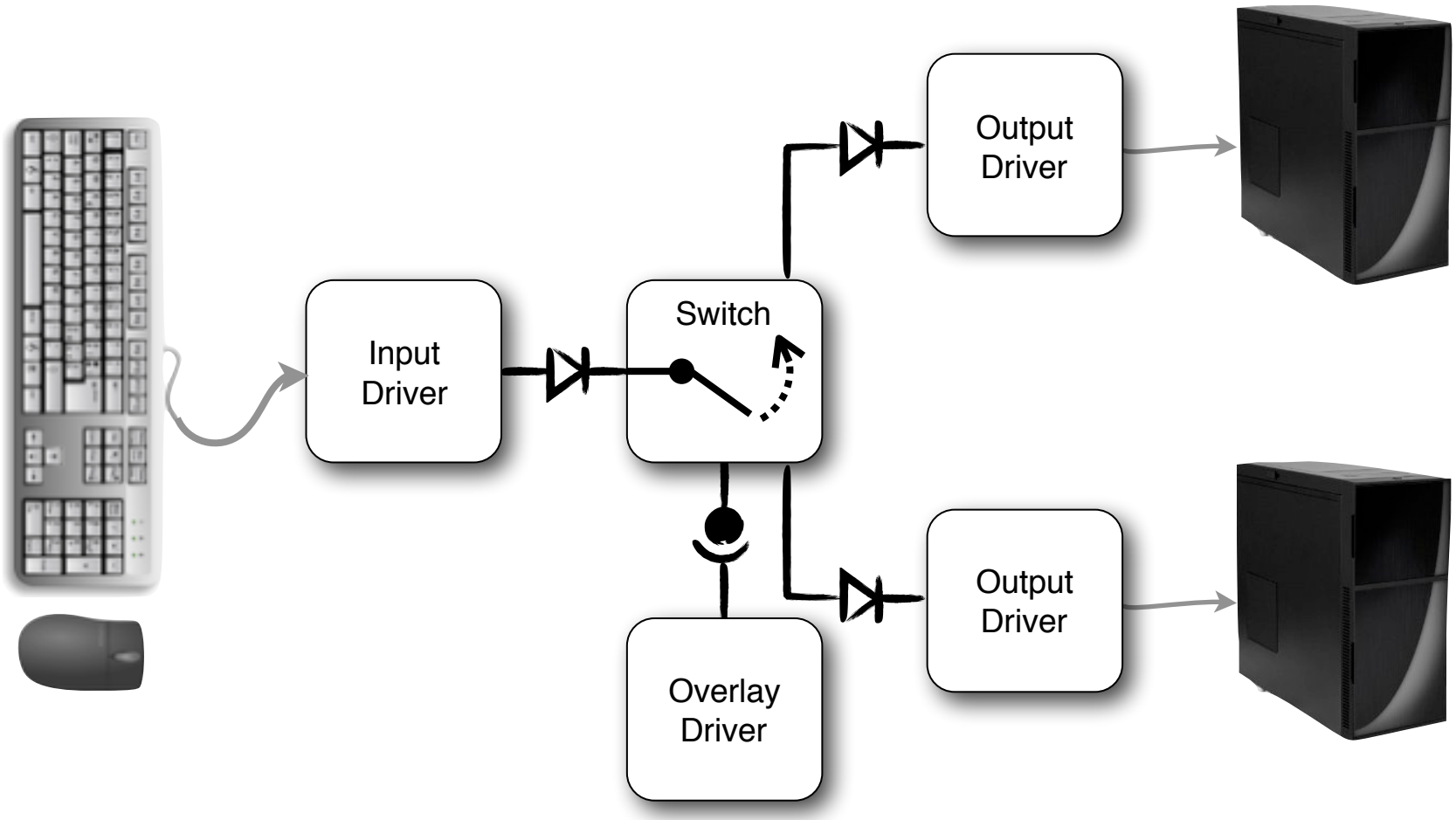
Status

- Soundness proof: almost complete
- Application to CDDC model: in progress

TEST-DRIVING THE FRAMEWORK

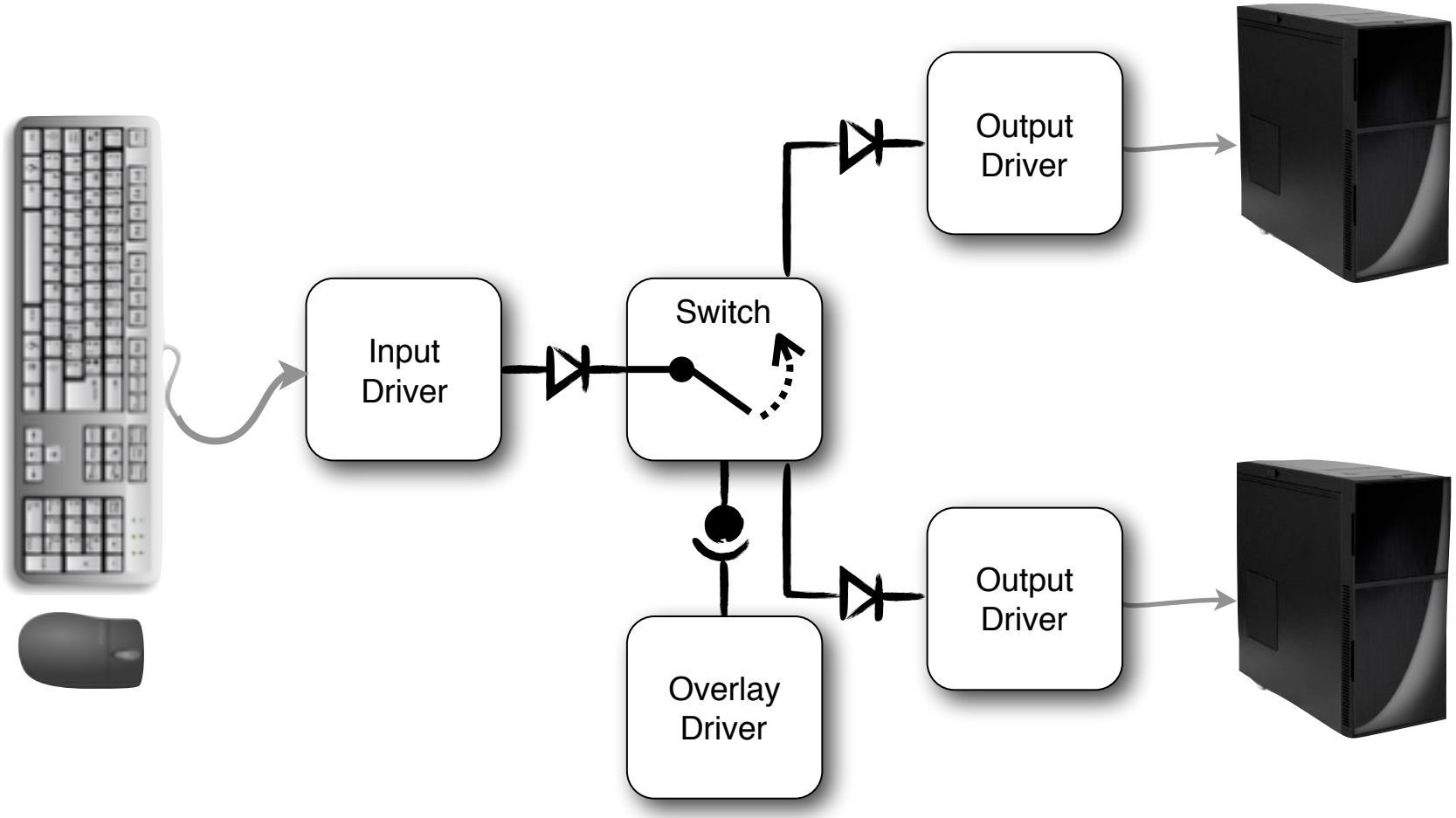


CDDC seL4-Based Software Architecture

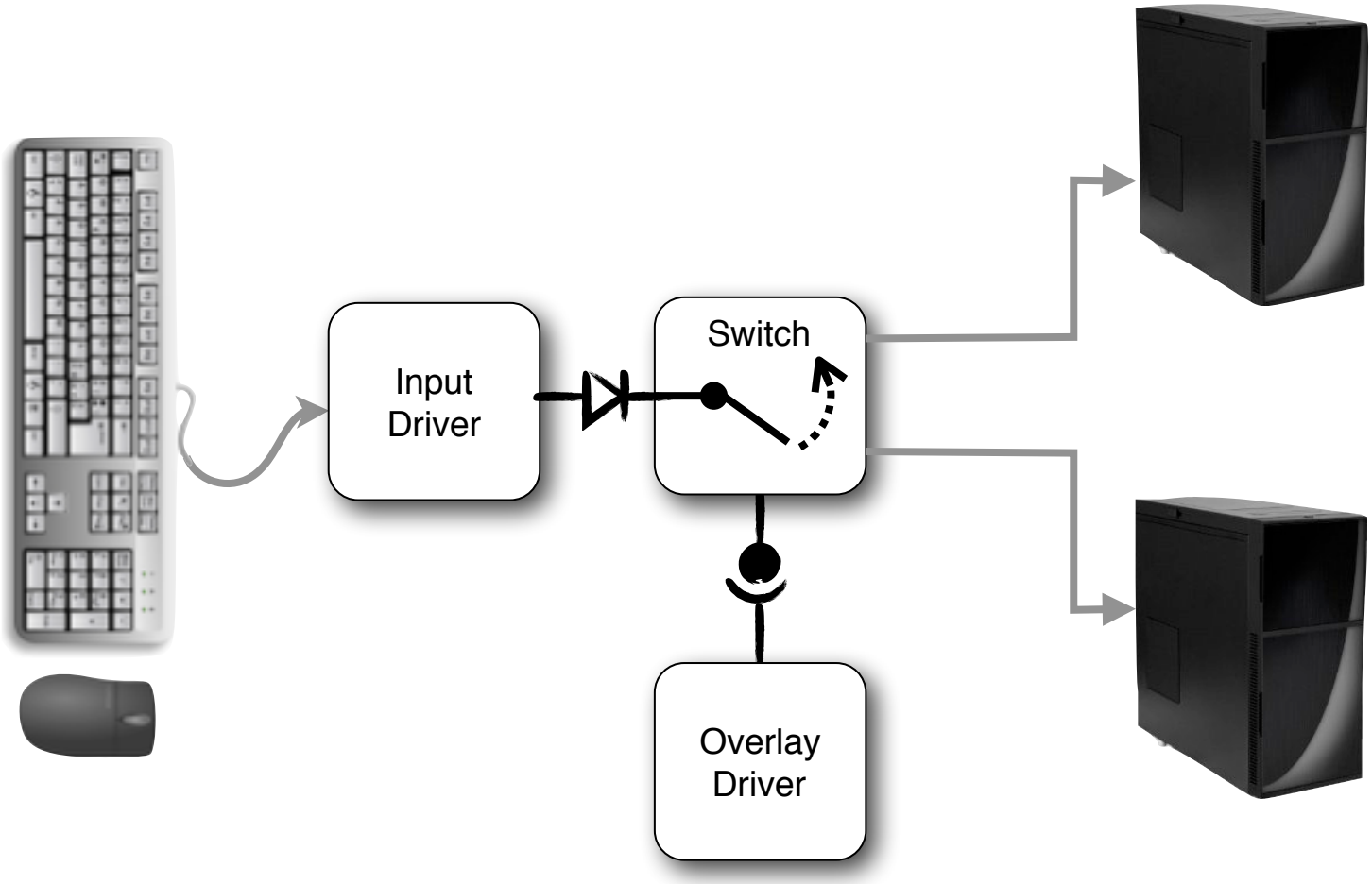


CDDC seL4-Based Software Architecture

(ignoring device administration and configuration, plus keyboard LED control)

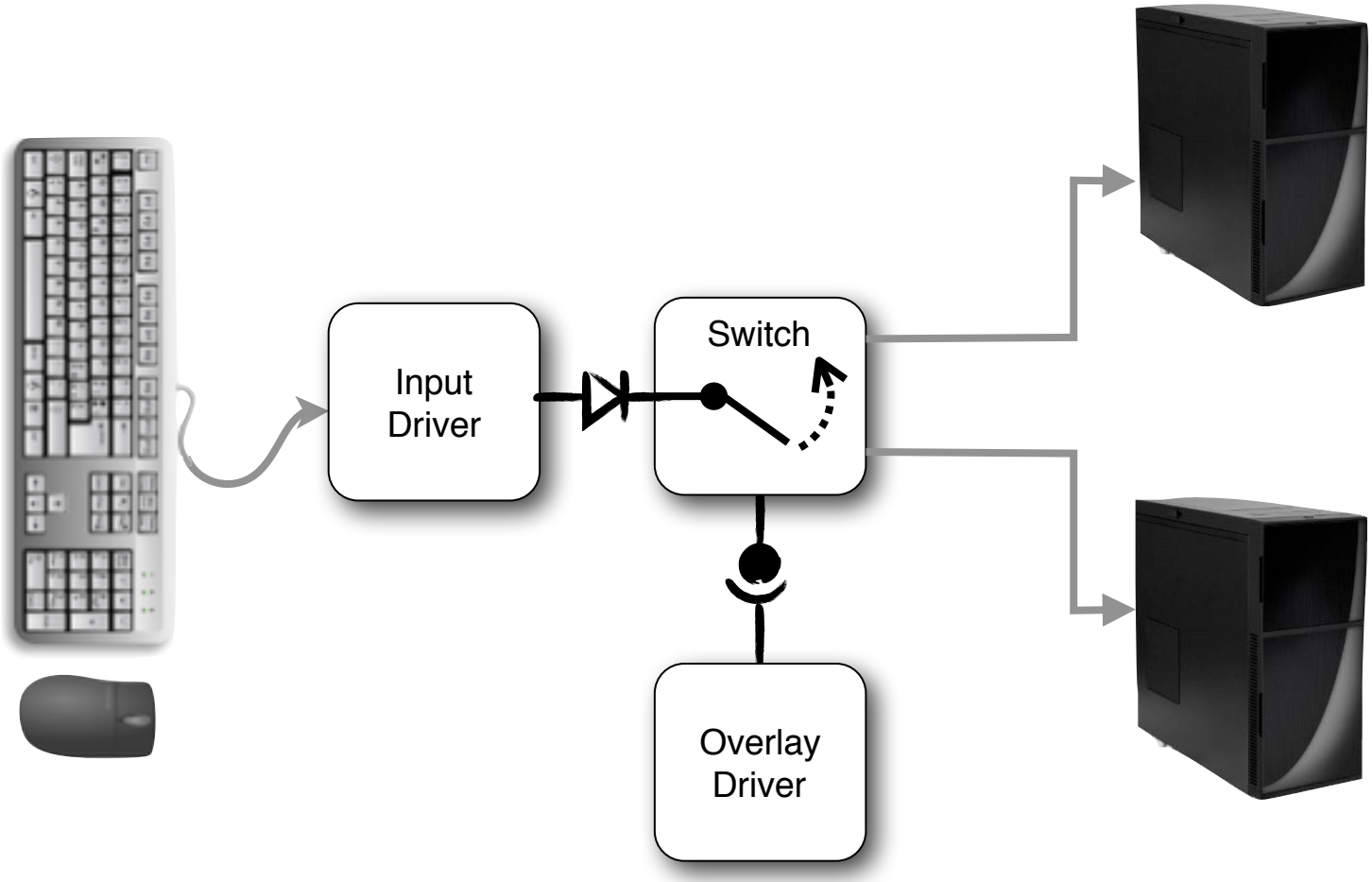


(Almost) Verified CDDC Software



(Almost) Verified CDDC Software

(static config, no hotkeys)

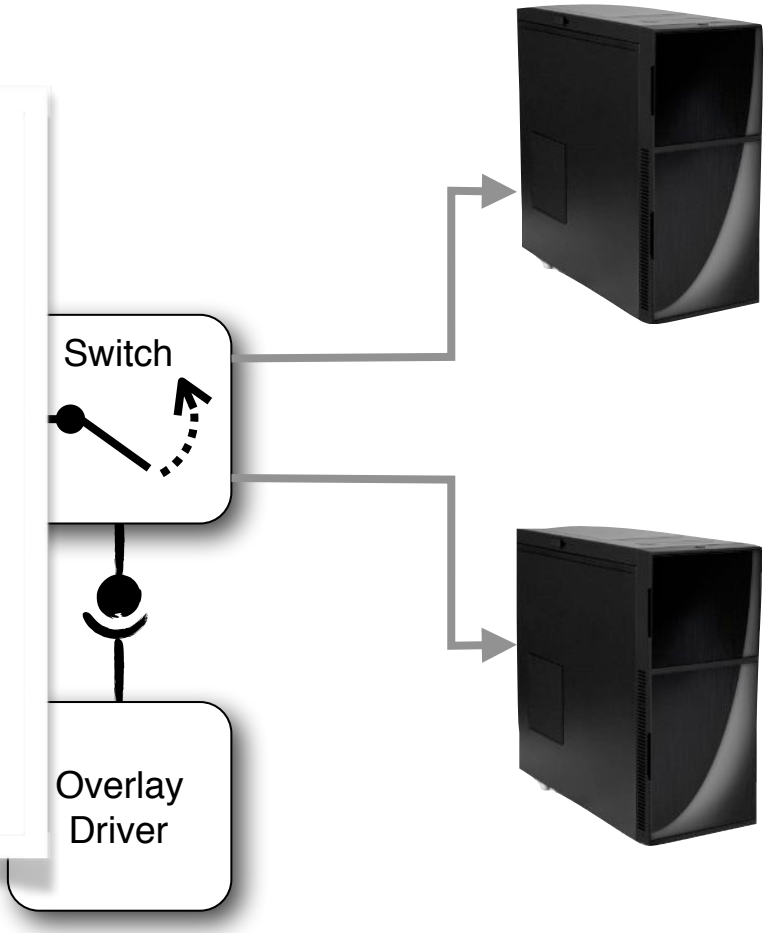


(Almost) Verified CDDC Software

(static config, no hotkeys)

Makes use of:

- Value-dependent classification
- Shared-memory concurrency
- Lock-based synchronisation
- Inter-component invariants

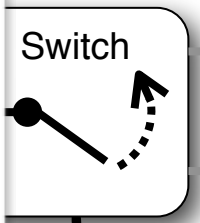


(Almost) Verified CDDC Software

(static config, no hotkeys)

Makes use of:

- Value-dependent classification
- Shared-memory concurrency
- Lock-based synchronisation
- Inter-component invariants



Avoids:

- Arrays & Pointer arithmetic
- Complex data structures

Pointers etc.

```
void write_labeled_val(int v, bool b,
                       int* highptr, int* lowptr) {
  if (b) {
    *highptr = *highptr | v;
  } else {
    *lowptr = *lowptr | v;
  }
}
```

Security Aware Separation Logic (c.f. Costanzo & Shao, POST 2014)

Pointers etc.

$$\{\texttt{highptr} \overset{\text{Hi}}{\longmapsto} h * \texttt{lowptr} \overset{\text{Lo}}{\longmapsto} l * [\texttt{b} = b, \texttt{v} = v, \texttt{b} :: \text{Lo}, \texttt{v} :: (b\ ?\ \text{Hi} : \text{Lo})]\}$$

```
void write_labeled_val(int v, bool b,
                       int* highptr, int* lowptr) {
  if (b) {
    *highptr = *highptr | v;
  } else {
    *lowptr = *lowptr | v;
  }
}
```

Security Aware Separation Logic (c.f. Costanzo & Shao, POST 2014)

Pointers etc.

$$\{\texttt{highptr} \overset{\text{Hi}}{\longmapsto} h \;*\; \texttt{lowptr} \overset{\text{Lo}}{\longmapsto} l \;*\; [\texttt{b} = b, \texttt{v} = v, \texttt{b} :: \text{Lo}, \texttt{v} :: (b \;?\; \text{Hi} : \text{Lo})]\}$$

```
void write_labeled_val(int v, bool b,
                       int* highptr, int* lowptr) {
  if (b) {
    *highptr = *highptr | v;
  } else {
    *lowptr = *lowptr | v;
  }
}
```

$$\{\texttt{highptr} \overset{\text{Hi}}{\longmapsto} (b \;?\; h|v : h) \;*\; \texttt{lowptr} \overset{\text{Lo}}{\longmapsto} (b \;?\; l : l|v)\}$$

Pointers etc.

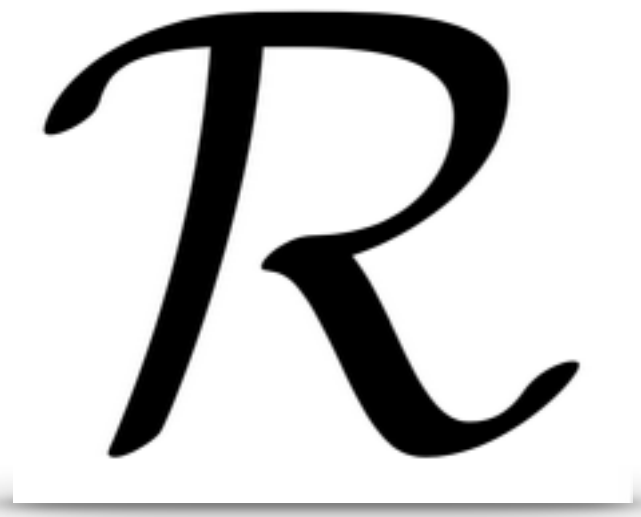
$$\{\texttt{highptr} \overset{\text{Hi}}{\longmapsto} h \;*\; \texttt{lowptr} \overset{\text{Lo}}{\longmapsto} l \;*\; [\texttt{b} = b, \texttt{v} = v, \; \texttt{b} :: \text{Lo}, \texttt{v} :: (b \;?\; \text{Hi} : \text{Lo})]\}$$

```
void write_labeled_val(int v, bool b,
                       int* highptr, int* lowptr) {
  if (b) {
    *highptr = *highptr | v;
  } else {
    *lowptr = *lowptr | v;
  }
}
```

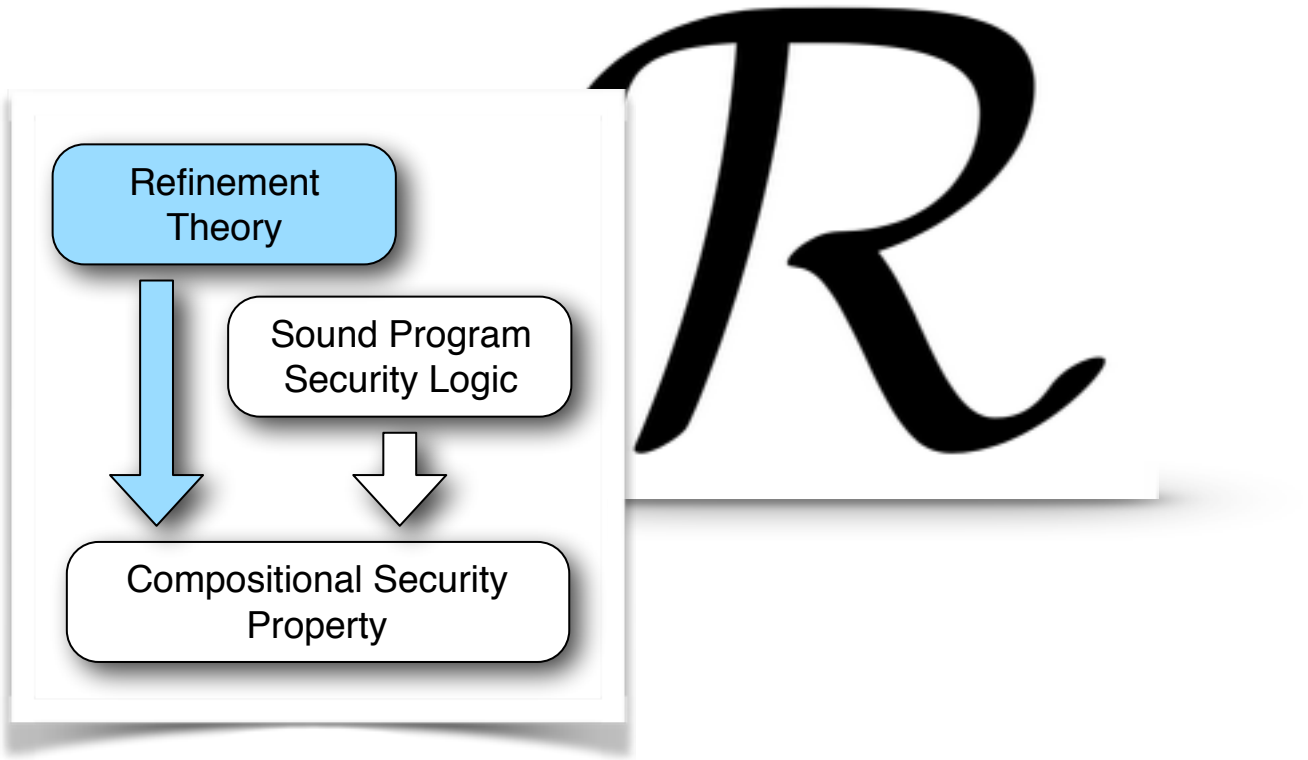
Samuel Gruetter

$$\{\texttt{highptr} \overset{\text{Hi}}{\longmapsto} (b \;?\; h|v : h) \;*\; \texttt{lowptr} \overset{\text{Lo}}{\longmapsto} (b \;?\; l : l|v)\}$$

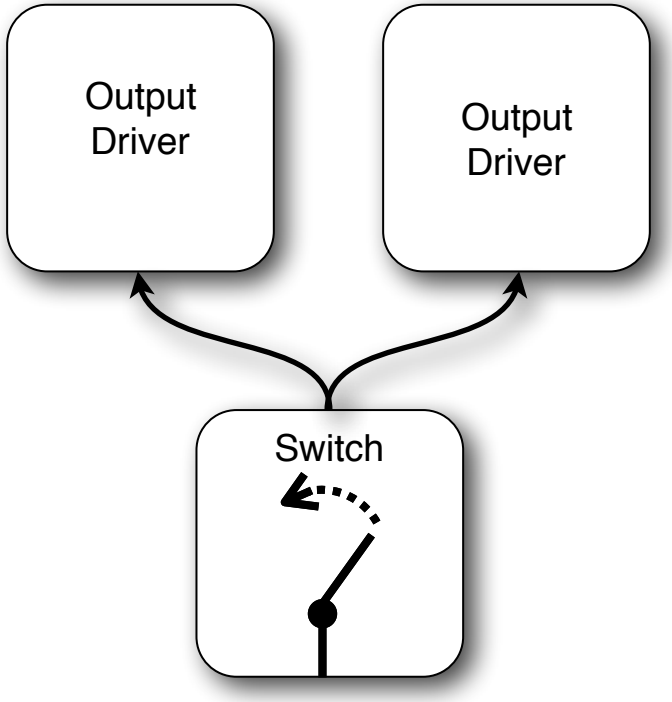
COMPOSITIONAL REFINEMENT



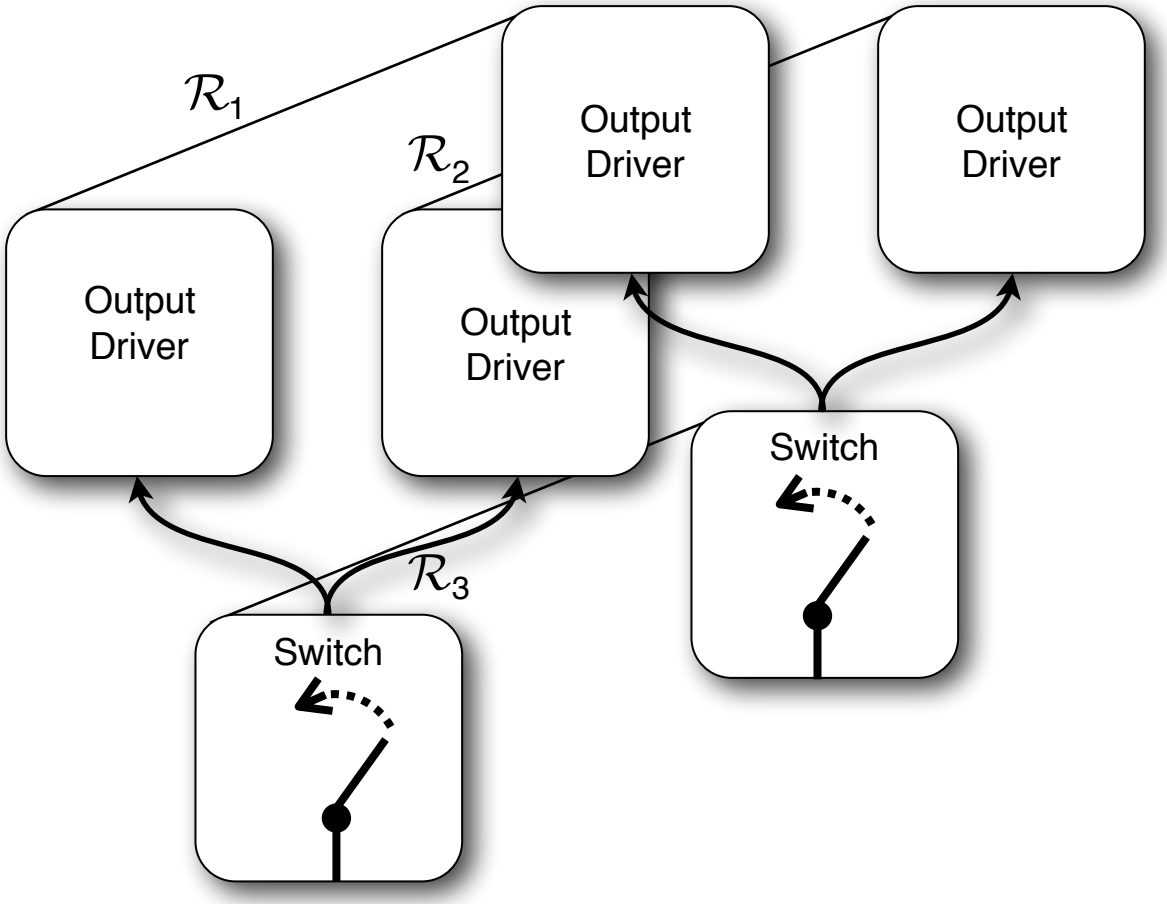
COMPOSITIONAL REFINEMENT



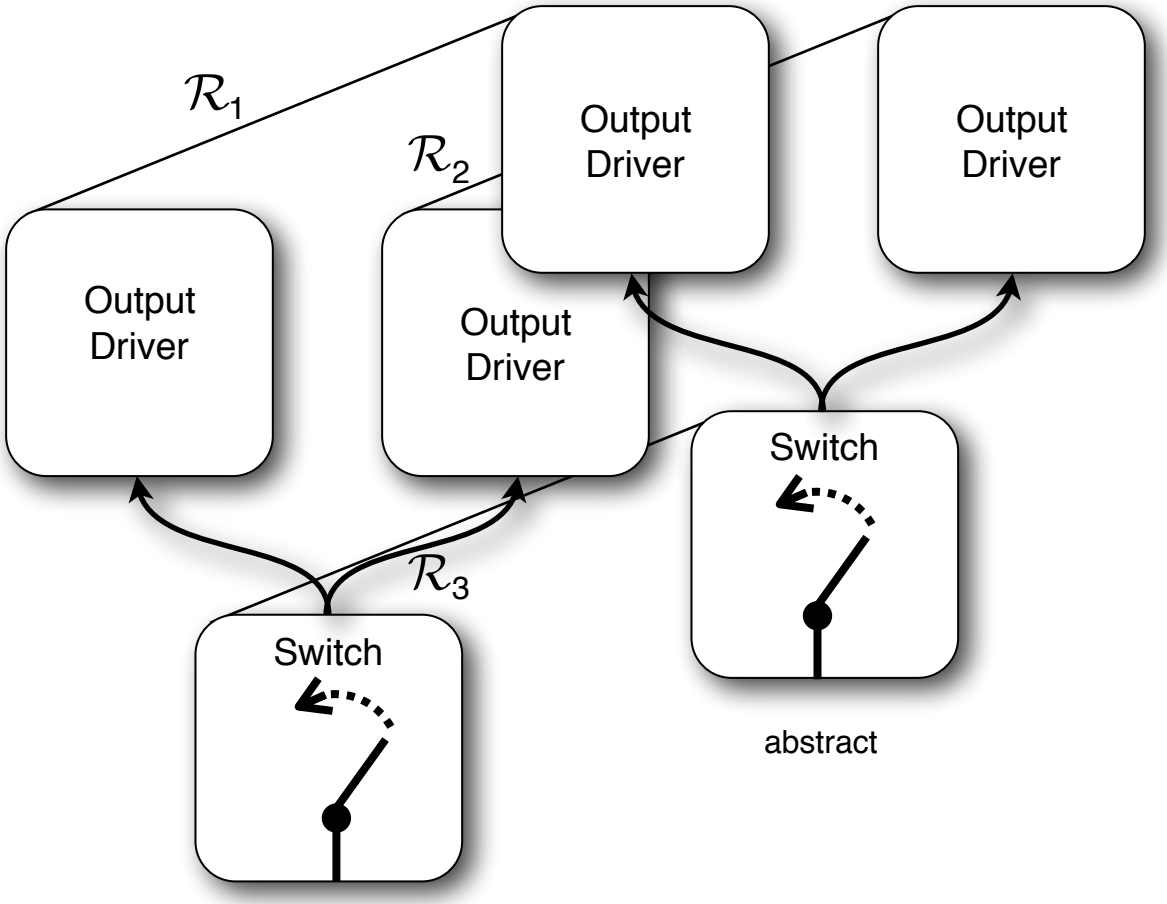
Secure Componentwise Refinement



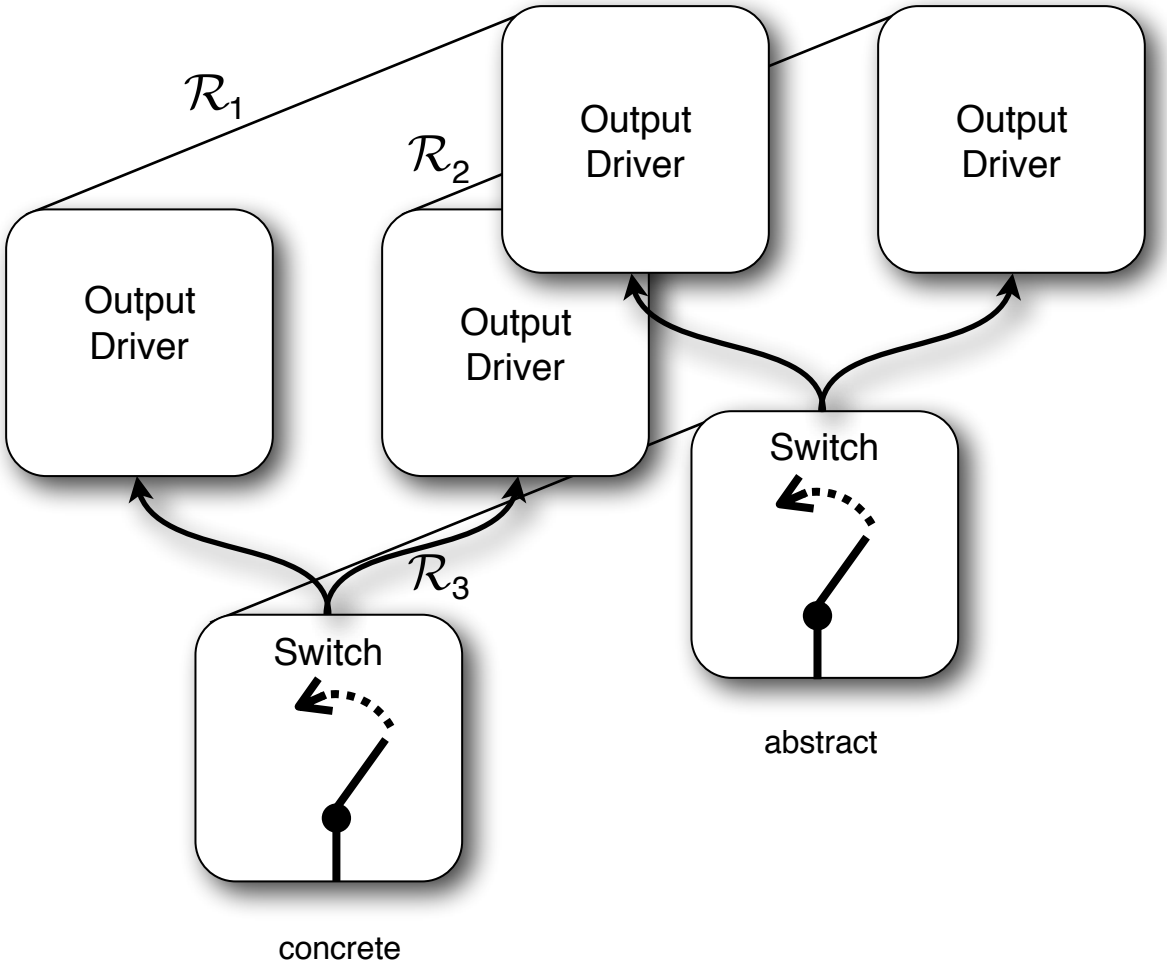
Secure Componentwise Refinement



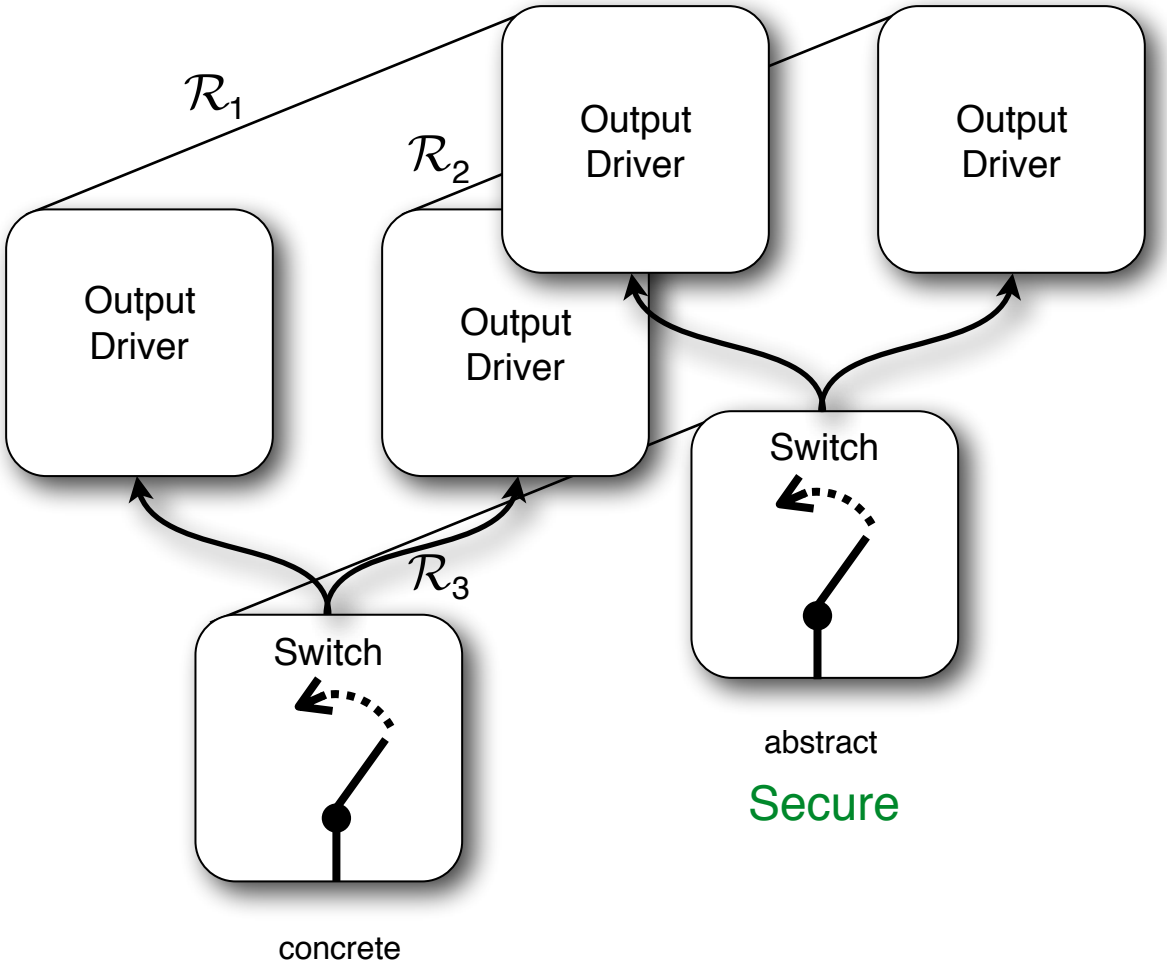
Secure Componentwise Refinement



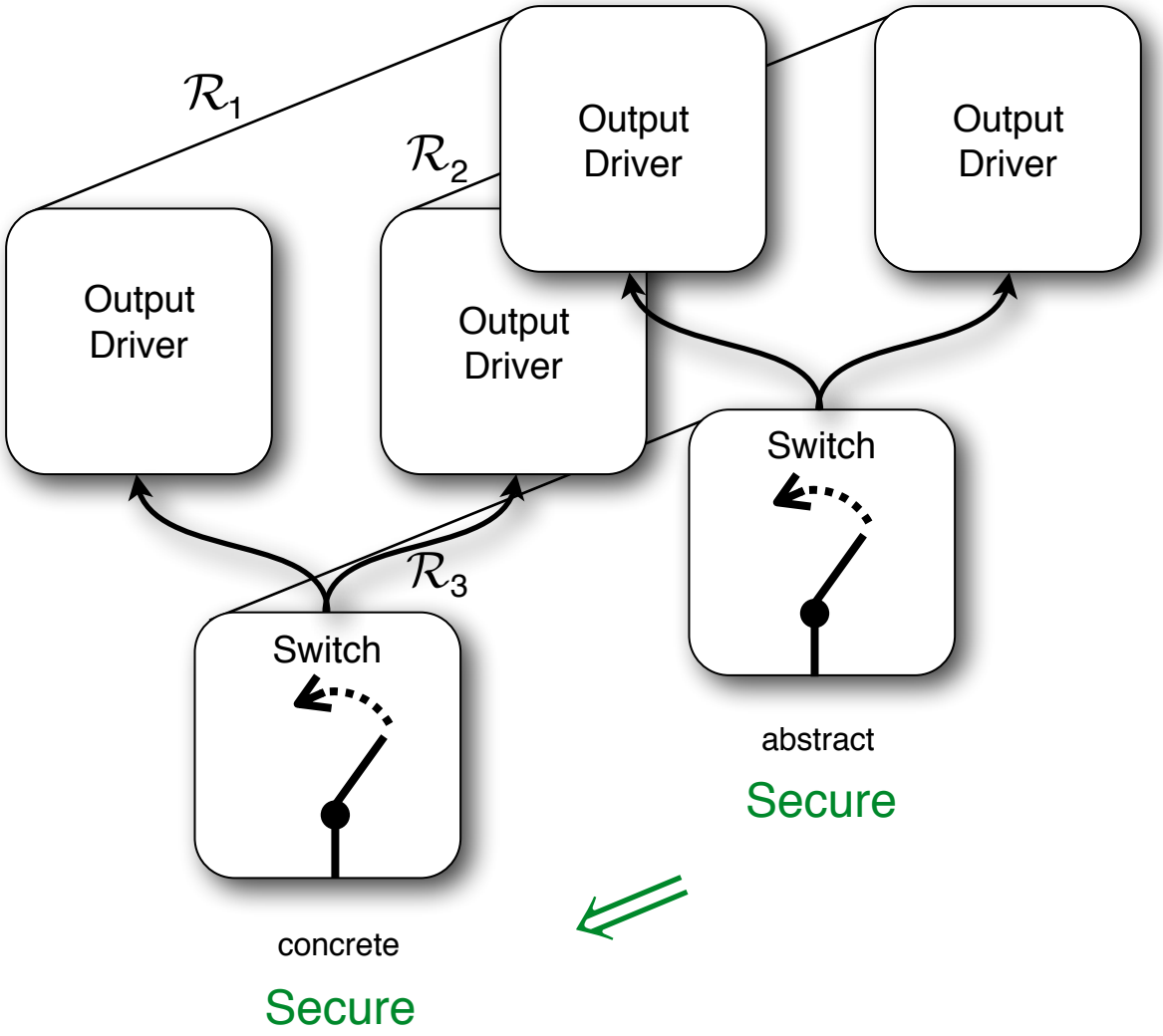
Secure Componentwise Refinement



Secure Componentwise Refinement



Secure Componentwise Refinement



Component Refinement

```
skip   /* control +=_m AsmNoW */;
skip   /* temp +=_m AsmNoRW */;
temp := buffer;
if control == 0 then
  low-var := temp
else
  high-var := temp
endif;
temp := 0;
skip   /* temp -=_m AsmNoRW */
```

```
skip   /* control_C +=_m AsmNoW */;
skip   /* temp_C +=_m AsmNoRW */;
temp_C := buffer_C;
reg_C := control_C;
if reg_C == 0 then
  low-var_C := temp_C
else
  high-var_C := temp_C
endif;
temp_C := 0;
skip   /* temp_C -=_m AsmNoRW */
```

Component Refinement

Original Program

```
skip   /* control +=_m AsmNoW */;
skip   /* temp +=_m AsmNoRW */;
temp := buffer;
if control == 0 then
  low-var := temp
else
  high-var := temp
endif;
temp := 0;
skip   /* temp -=_m AsmNoRW */
```

```
skip   /* control_C +=_m AsmNoW */;
skip   /* temp_C +=_m AsmNoRW */;
temp_C := buffer_C;
reg_C := control_C;
if reg_C == 0 then
  low-var_C := temp_C
else
  high-var_C := temp_C
endif;
temp_C := 0;
skip   /* temp_C -=_m AsmNoRW */
```

Component Refinement

Original Program

```
skip   /* control +=_m AsmNoW */;
skip   /* temp +=_m AsmNoRW */;
temp := buffer;
if control == 0 then
  low-var := temp
else
  high-var := temp
endif;
temp := 0;
skip   /* temp -=_m AsmNoRW */
```

A Trivial Refinement

```
skip   /* control_C +=_m AsmNoW */;
skip   /* temp_C +=_m AsmNoRW */;
temp_C := buffer_C;
reg_C := control_C;
if reg_C == 0 then
  low-var_C := temp_C
else
  high-var_C := temp_C
endif;
temp_C := 0;
skip   /* temp_C -=_m AsmNoRW */
```

Component Refinement

concrete variables implementing abstract ones

Original Program

```
skip   /* control +=_m AsmNoW */;
skip   /* temp +=_m AsmNoRW */;
temp := buffer;
if control == 0 then
  low-var := temp
else
  high-var := temp
endif;
temp := 0;
skip   /* temp -=_m AsmNoRW */
```

A Trivial Refinement

```
skip   /* control_C +=_m AsmNoW */;
skip   /* temp_C +=_m AsmNoRW */;
temp_C := buffer_C;
reg_C := control_C;
if reg_C == 0 then
  low-var_C := temp_C
else
  high-var_C := temp_C
endif;
temp_C := 0;
skip   /* temp_C -=_m AsmNoRW */
```

Component Refinement

Original Program

```
skip  /* control +=_m AsmNoW */;
skip  /* temp +=_m AsmNoRW */;
temp := buffer;
if control == 0 then
  low-var := temp
else
  high-var := temp
endif;
temp := 0;
skip  /* temp -=_m AsmNoRW */
```

concrete variables implementing abstract ones

concrete variables with no abstract counterpart

A Trivial Refinement

```
skip  /* control_C +=_m AsmNoW */;
skip  /* temp_C +=_m AsmNoRW */;
temp_C := buffer_C;
reg_C := control_C;
if reg_C == 0 then
  low-var_C := temp_C
else
  high-var_C := temp_C
endif;
temp_C := 0;
skip  /* temp_C -=_m AsmNoRW */
```

Component Refinement

Original Program

```
skip   /* control +=_m AsmNoW */;
skip   /* temp +=_m AsmNoRW */;
temp := buffer;
if control == 0 then
  low-var := temp
else
  high-var := temp
endif;
temp := 0;
skip   /* temp -=_m AsmNoRW */
```

A Trivial Refinement

```
skip   /* control_C +=_m AsmNoW */;
skip   /* temp_C +=_m AsmNoRW */;
temp_C := buffer_C;
reg_C := control_C;
if reg_C == 0 then
  low-var_C := temp_C
else
  high-var_C := temp_C
endif;
temp_C := 0;
skip   /* temp_C -=_m AsmNoRW */
```

Component Refinement

Original Program

```
skip   /* control +=_m AsmNoW */;
skip   /* temp +=_m AsmNoRW */;
temp := buffer;
if control == 0 then
  low-var := temp
else
  high-var := temp
endif;
temp := 0;
skip   /* temp -=_m AsmNoRW */
```

A Trivial Refinement

```
skip   /* control_C +=_m AsmNoW */;
skip   /* temp_C +=_m AsmNoRW */;
temp_C := buffer_C;
reg_C := control_C;
if reg_C == 0 then
  low-var_C := temp_C
else
  high-var_C := temp_C
endif;
temp_C := 0;
skip   /* temp_C -=_m AsmNoRW */
```

Component Refinement

Original Program

```
skip   /* control +=_m AsmNoW */;
skip   /* temp +=_m AsmNoRW */;
temp := buffer;
if control == 0 then
  low-var := temp
else
  high-var := temp
endif;
temp := 0;
skip   /* temp -=_m AsmNoRW */
```

A Trivial Refinement

```
skip   /* control_C +=_m AsmNoW */;
skip   /* temp_C +=_m AsmNoRW */;
temp_C := buffer_C;
reg_C := control_C;
if reg_C == 0 then
  low-var_C := temp_C
else
  high-var_C := temp_C
endif;
temp_C := 0;
skip   /* temp_C -=_m AsmNoRW */
```

Abstract Program Security: Bisimulation

```
if (h) {
    h = 0;
    l = 1;
} else {
    skip;
    l = 1;
}
```

Abstract Program Security: Bisimulation

```
if (h) {
    h = 0;
    l = 1;
} else {
    skip;
    l = 1;
}
```

```
if (h) {
    h = 0;
    l = 1;
} else {
    skip;
    l = 1;
}
```

Abstract Program Security: Bisimulation

```
if (h) {
    h = 0;
    l = 1;
} else {
    skip;
    l = 1;
}
```

$\mathcal{B}$

```
if (h) {
    h = 0;
    l = 1;
} else {
    skip;
    l = 1;
}
```

Abstract Program Security: Bisimulation

```
if (h) {
    h = 0;
    l = 1;
} else {
    skip;
    l = 1;
}
```

$\mathcal{B}$

```
if (h) {
    h = 0;
    l = 1;
} else {
    skip;
    l = 1;
}
```

Important:

$\mathcal{B}$ must guarantee that the two memories are always Low-equivalent

Abstract Program Security: Bisimulation

```
if (h) {
    h = 0;
    l = 1;
} else {
    skip;
    l = 1;
}
```

$\mathcal{B}$

```
if (h) {
    h = 0;
    l = 1;
} else {
    skip;
    l = 1;
}
```

Important:

$\mathcal{B}$ must guarantee that the two memories are always Low-equivalent

Abstract Program Security: Bisimulation

```
if (h) {
    h = 0;
    l = 1;
} else {
    skip;
    l = 1;
}
```

$\mathcal{B}$

```
if (h) {
    h = 0;
    l = 1;
} else {
    skip;
    l = 1;
}
```

Important:

$\mathcal{B}$ must guarantee that the two memories are always Low-equivalent

Abstract Program Security: Bisimulation

```
if (h) {
     h = 0;
     l = 1;
} else {
     skip;
     l = 1;
}
```

$\mathcal{B}$

```
if (h) {
     h = 0;
     l = 1;
} else {
     skip;
     l = 1;
}
```

Important:

$\mathcal{B}$ must guarantee that the two memories are always Low-equivalent

Abstract Program Security: Bisimulation

```
if (h) {
     h = 0;
     l = 1;
} else {
     skip;
     l = 1;
}
```

$\mathcal{B}$

```
if (h) {
     h = 0;
     l = 1;
} else {
     skip;
     l = 1;
}
```

(the soundness proof for the type system constructs an appropriate $\mathcal{B}$)

Important:

$\mathcal{B}$ must guarantee that the two memories are always Low-equivalent

Refining High Conditionals

```
if (h) {
    h = 0;
    l = 1;
} else {
    skip;
    l = 1;
}
```

Refining High Conditionals

A Secure Program

```
if (h) {
    h = 0;
    l = 1;
} else {
    skip;
    l = 1;
}
```

Refining High Conditionals

A Secure Program

branches need to do the same Low things at the same time

```
if (h) {
     h = 0;
     l = 1;
} else {
     skip;
     l = 1;
}
```

Refining High Conditionals

branches need to do the same Low things at the same time

A Secure Program

```
if (h) {
    h = 0;
    l = 1;
} else {
    skip;
    l = 1;
}
```

A Trivial Refinement

```
if (h) {
    h = 0;
    l = 1;
} else {
    l = 1;
}
```

Refining High Conditionals

branches need to do the same Low things at the same time

A Secure Program

```
if (h) {
    h = 0;
    l = 1;
} else {
    skip;
    l = 1;
}
```

A Trivial Refinement

```
if (h) {
    h = 0;
    l = 1;
} else {
    l = 1;
}
```

insecure

Refining High Conditionals

branches need to do the same Low things at the same time

A Secure Program

```
if (h) {
    h = 0;
    l = 1;
} else {
    skip;
    l = 1;
}
```

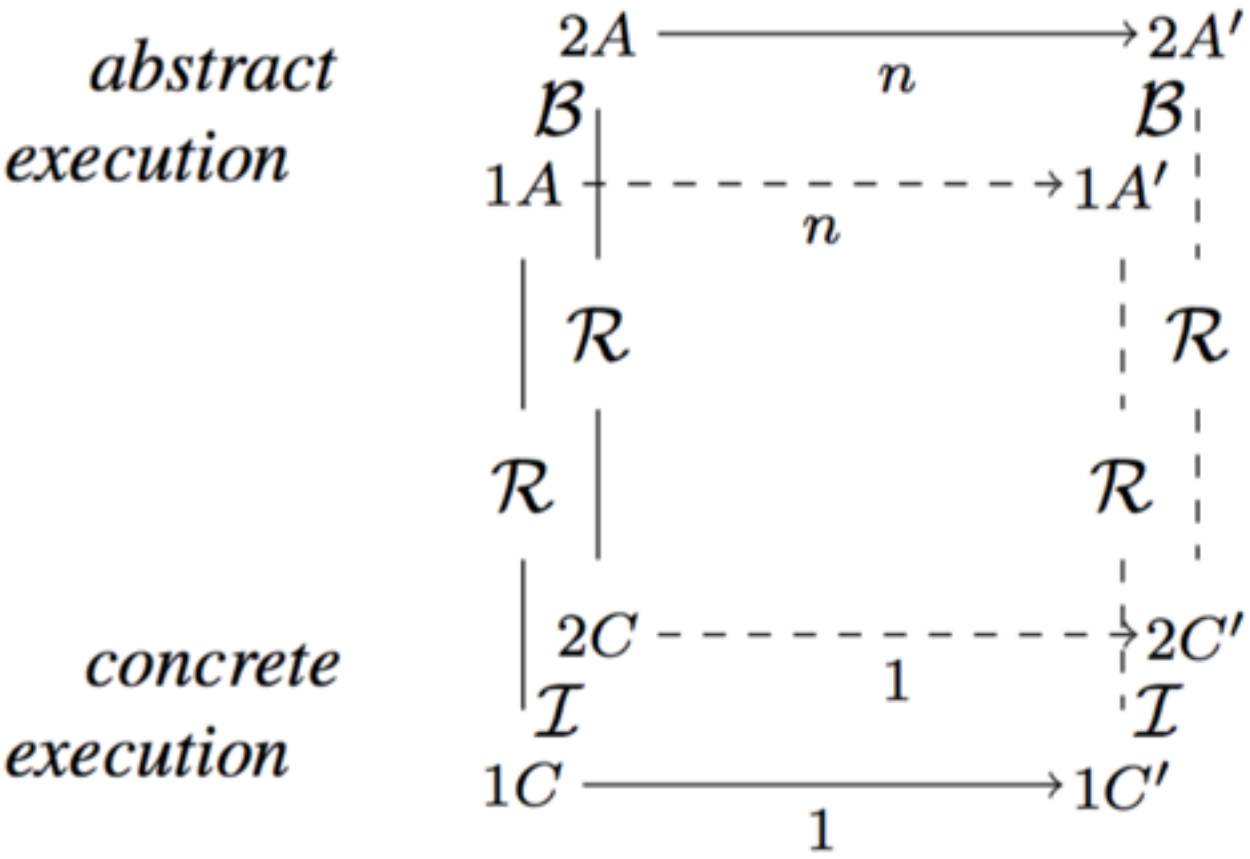
A Trivial Refinement

```
if (h) {
    h = 0;
    l = 1;
} else {
    l = 1;
}
```

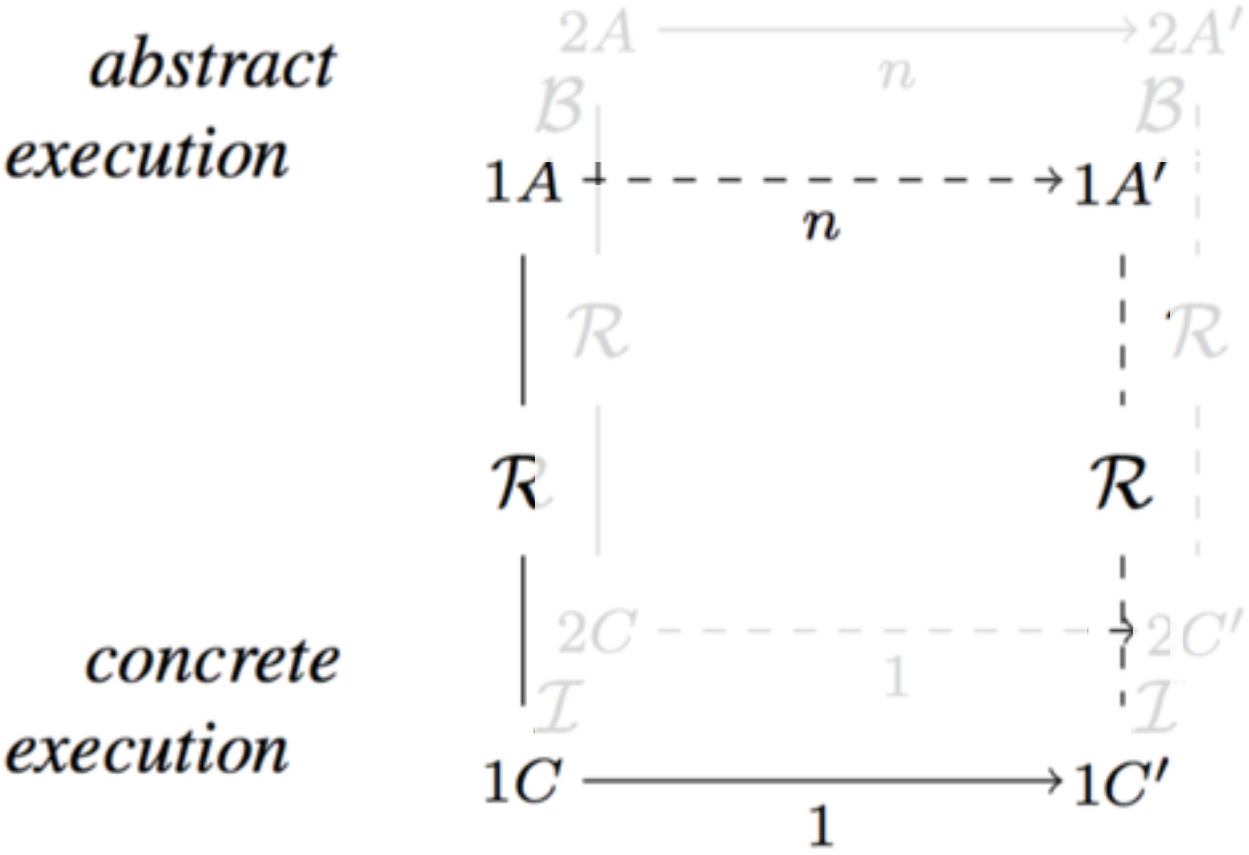
insecure

Need to make sure that the concrete executions stay in sync.

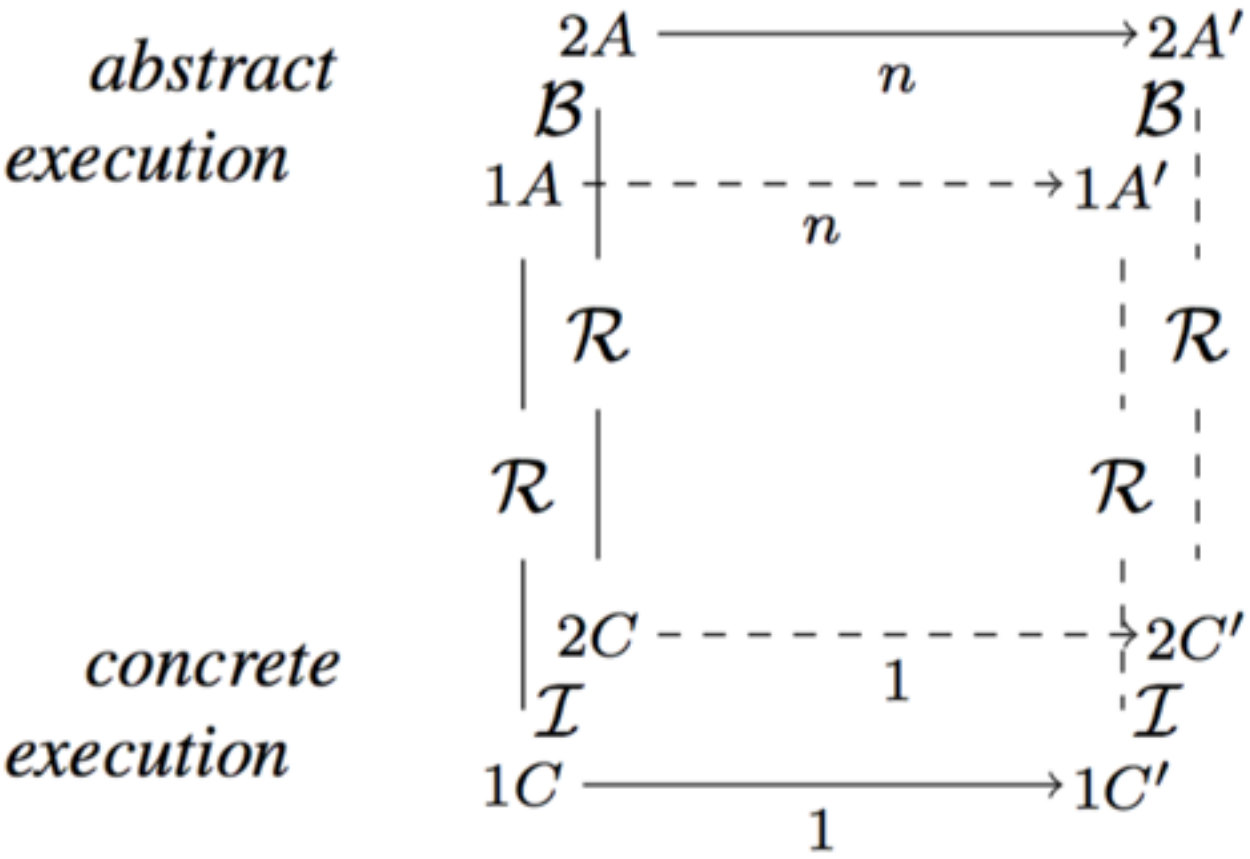
Secure Refinement via Coupling



Secure Refinement via Coupling



Secure Refinement via Coupling



Coupling Invariant: $\mathcal{I}$

```
skip   /* h +=_m AsmNoW */;
skip   /* y +=_m AsmNoW */;
skip   /* z +=_m AsmNoW */;
y := 0;
z := 0;
x := y;
if h != 0 then
  x := y
else
  x := y + z
endif
```

Coupling Invariant: $\mathcal{I}$

```
skip   /* h_C +=_m AsmNoW */;
skip   /* y_C +=_m AsmNoW */;
skip   /* z_C +=_m AsmNoW */;
y_C := 0;
z_C := 0;
x_C := y_C;
reg3_C := h_C;
if reg3_C != 0 then
  skip;
  skip;
  reg0_C := y_C;
  x_C := reg0_C
else
  reg1_C := y_C;
  reg2_C := z_C;
  reg0_C := reg1_C + reg2_C;
  x_C := reg0_C
endif
```

Coupling Invariant: $\mathcal{I}$

```
skip   /* h_C +=_m AsmNoW */;
skip   /* y_C +=_m AsmNoW */;
skip   /* z_C +=_m AsmNoW */;
y_C := 0;
z_C := 0;
x_C := y_C;
reg3_C := h_C;
if reg3_C != 0 then
  skip;
  skip;
  reg0_C := y_C;
  x_C := reg0_C
else
  reg1_C := y_C;
  reg2_C := z_C;
  reg0_C := reg1_C + reg2_C;
  x_C := reg0_C
endif
```


```
skip   /* h_C +=_m AsmNoW */;
skip   /* y_C +=_m AsmNoW */;
skip   /* z_C +=_m AsmNoW */;
y_C := 0;
z_C := 0;
x_C := y_C;
reg3_C := h_C;
if reg3_C != 0 then
  skip;
  skip;
  reg0_C := y_C;
  x_C := reg0_C
else
  reg1_C := y_C;
  reg2_C := z_C;
  reg0_C := reg1_C + reg2_C;
  x_C := reg0_C
endif
```

Coupling Invariant: $\mathcal{I}$

```
skip   /* h_C +=_m AsmNoW */;
skip   /* y_C +=_m AsmNoW */;
skip   /* z_C +=_m AsmNoW */;
y_C := 0;
z_C := 0;
x_C := y_C;
reg3_C := h_C;
if reg3_C != 0 then
  skip;
  skip;
  reg0_C := y_C;
  x_C := reg0_C
else
  reg1_C := y_C;
  reg2_C := z_C;
  reg0_C := reg1_C + reg2_C;
  x_C := reg0_C
endif
```

```
skip   /* h_C +=_m AsmNoW */;
skip   /* y_C +=_m AsmNoW */;
skip   /* z_C +=_m AsmNoW */;
y_C := 0;
z_C := 0;
x_C := y_C;
reg3_C := h_C;
if reg3_C != 0 then
  skip;
  skip;
  reg0_C := y_C;
  x_C := reg0_C
else
  reg1_C := y_C;
  reg2_C := z_C;
  reg0_C := reg1_C + reg2_C;
  x_C := reg0_C
endif
```

Coupling Invariant: $\mathcal{I}$

```
skip   /* h_C +=_m AsmNoW */;
skip   /* y_C +=_m AsmNoW */;
skip   /* z_C +=_m AsmNoW */;
y_C := 0;
z_C := 0;
x_C := y_C;
reg3_C := h_C;
if reg3_C != 0 then
  skip;
  skip;
  reg0_C := y_C;
  x_C := reg0_C
else
  reg1_C := y_C;
  reg2_C := z_C;
  reg0_C := reg1_C + reg2_C;
  x_C := reg0_C
endif
```

```
skip   /* h_C +=_m AsmNoW */;
skip   /* y_C +=_m AsmNoW */;
skip   /* z_C +=_m AsmNoW */;
y_C := 0;
z_C := 0;
x_C := y_C;
reg3_C := h_C;
if reg3_C != 0 then
  skip;
  skip;
  reg0_C := y_C;
  x_C := reg0_C
else
  reg1_C := y_C;
  reg2_C := z_C;
  reg0_C := reg1_C + reg2_C;
  x_C := reg0_C
endif
```

Coupling Invariant: $\mathcal{I}$

```
skip   /* h_C +=_m AsmNoW */;
skip   /* y_C +=_m AsmNoW */;
skip   /* z_C +=_m AsmNoW */;
y_C := 0;
z_C := 0;
x_C := y_C;
reg3_C := h_C;
if reg3_C != 0 then
  skip;
  skip;
  reg0_C := y_C;
  x_C := reg0_C
else
  reg1_C := y_C;
  reg2_C := z_C;
  reg0_C := reg1_C + reg2_C;
  x_C := reg0_C
endif
```

```
skip   /* h_C +=_m AsmNoW */;
skip   /* y_C +=_m AsmNoW */;
skip   /* z_C +=_m AsmNoW */;
y_C := 0;
z_C := 0;
x_C := y_C;
reg3_C := h_C;
if reg3_C != 0 then
  skip;
  skip;
  reg0_C := y_C;
  x_C := reg0_C
else
  reg1_C := y_C;
  reg2_C := z_C;
  reg0_C := reg1_C + reg2_C;
  x_C := reg0_C
endif
```

Coupling Invariant: $\mathcal{I}$

```
skip  /* h_C +=_m AsmNoW */;
skip  /* y_C +=_m AsmNoW */;
skip  /* z_C +=_m AsmNoW */;
y_C := 0;
z_C := 0;
x_C := y_C;
reg3_C := h_C;
if reg3_C != 0 then
  skip;
  skip;
  reg0_C := y_C;
  x_C := reg0_C
else
  reg1_C := y_C;
  reg2_C := z_C;
  reg0_C := reg1_C + reg2_C;
  x_C := reg0_C
endif
```

```
skip  /* h_C +=_m AsmNoW */;
skip  /* y_C +=_m AsmNoW */;
skip  /* z_C +=_m AsmNoW */;
y_C := 0;
z_C := 0;
x_C := y_C;
reg3_C := h_C;
if reg3_C != 0 then
  reg0_C := reg1_C + reg2_C;
  x_C := reg0_C
endif
```

Important:

$\mathcal{I}$ need not talk at all about memory contents being secure

Concurrency-Aware Refinement

$$x := y + z$$

Concurrency-Aware Refinement

$$x := y + z$$

Source language semantics evaluates y + z atomically

Concurrency-Aware Refinement

$$x := y + z$$

Source language semantics evaluates y + z atomically

Imagine we compile to a simple stack machine:

```
PUSH y;
PUSH z;
ADD;
POP x;
```

Concurrency-Aware Refinement

$$x := y + z$$

Source language semantics evaluates y + z atomically

Imagine we compile to a simple stack machine:

```
PUSH y;
PUSH z;
ADD;
POP x;
```

This compilation is valid only if no other thread concurrently modifies y and z

Concurrency-Aware Refinement

$$x := y + z$$

Source language semantics evaluates y + z atomically

Imagine we compile to a simple stack machine:

```
PUSH y;
PUSH z;
ADD;
POP x;
```

This compilation is valid only if no other thread concurrently modifies y and z

Idea: require y and z to be assumes not-writable by others

Concurrency-Aware Refinement

$$x := y + z$$

Source language semantics evaluates y + z atomically

Imagine we compile to a simple stack machine:

```
PUSH y;
PUSH z;
ADD;
POP x;
```

This compilation is valid only if no other thread concurrently modifies y and z

Idea: require y and z to be assumes not-writable by others

i.e. that the refinement relation is preserved by potential actions of other threads

Concurrency-Aware Refinement

$$x := y + z$$

Source language semantics evaluates y + z atomically

Imagine we compile to a simple stack machine:

```
PUSH y;
PUSH z;
ADD;
POP x;
```

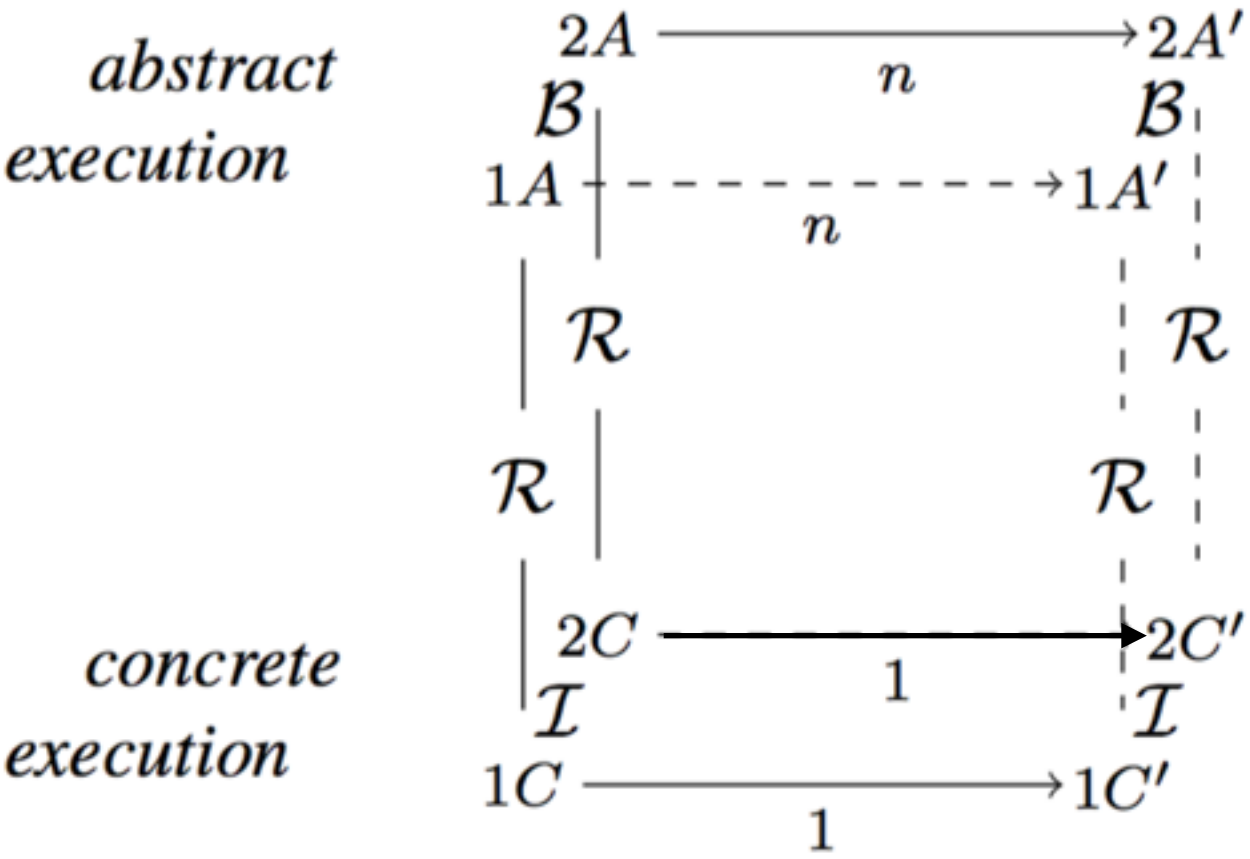
This compilation is valid only if no other thread concurrently modifies y and z

Idea: require y and z to be assumes not-writable by others

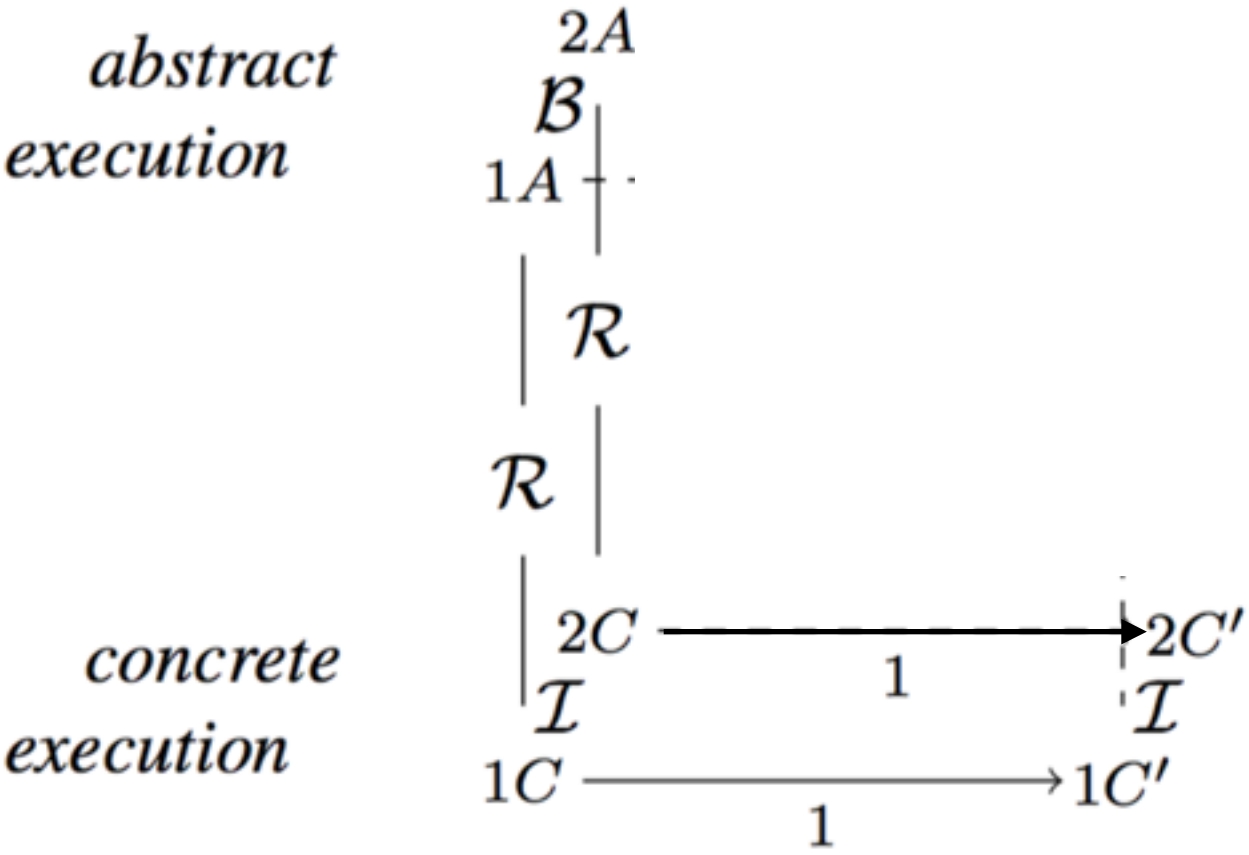
(see Liang et al., TOPLAS 2014)

i.e. that the refinement relation is preserved by potential actions of other threads

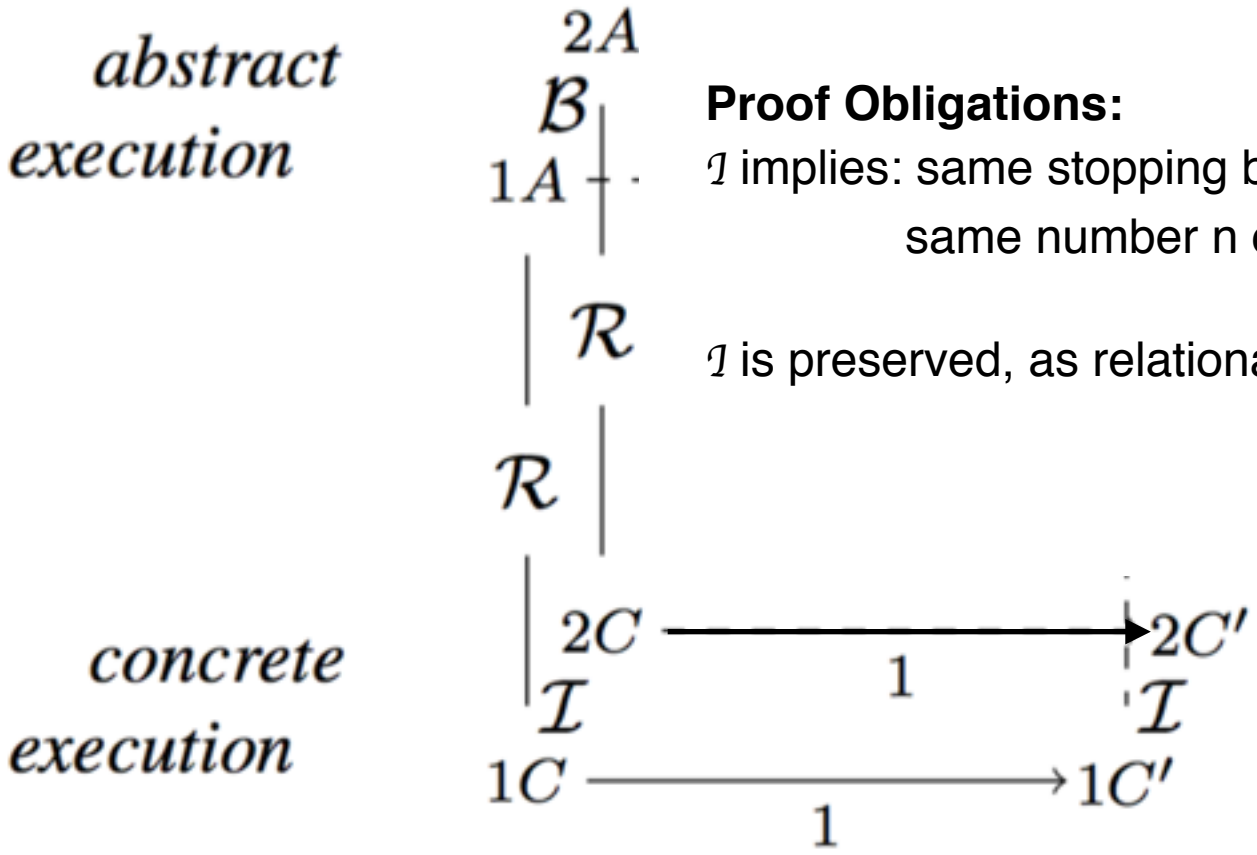
Leveraging Determinism



Leveraging Determinism



Leveraging Determinism



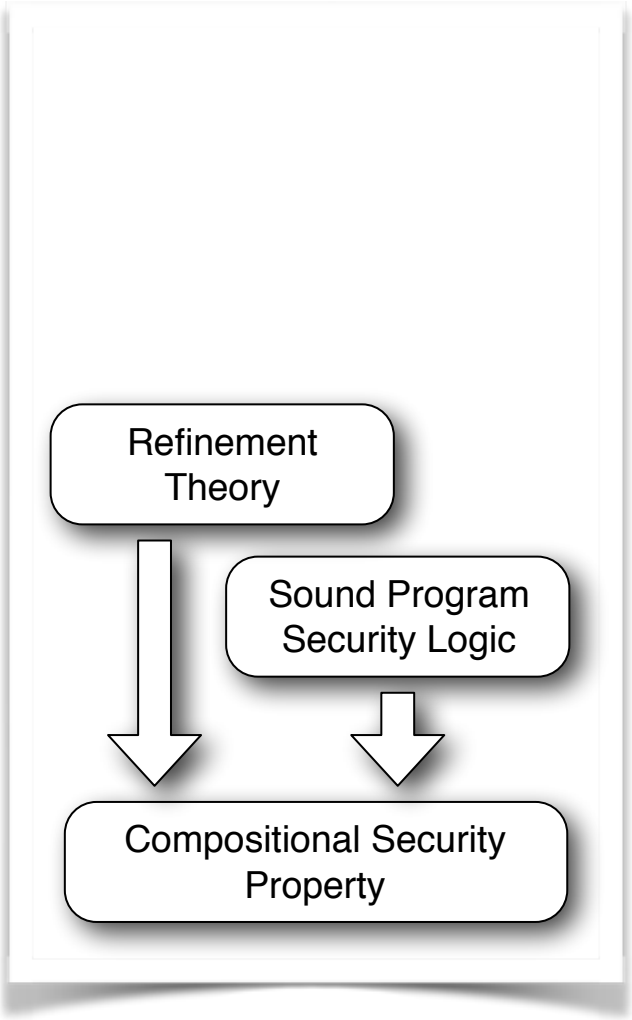
Proof Obligations:

$\mathcal{I}$ implies: same stopping behaviour
same number n of abstract steps

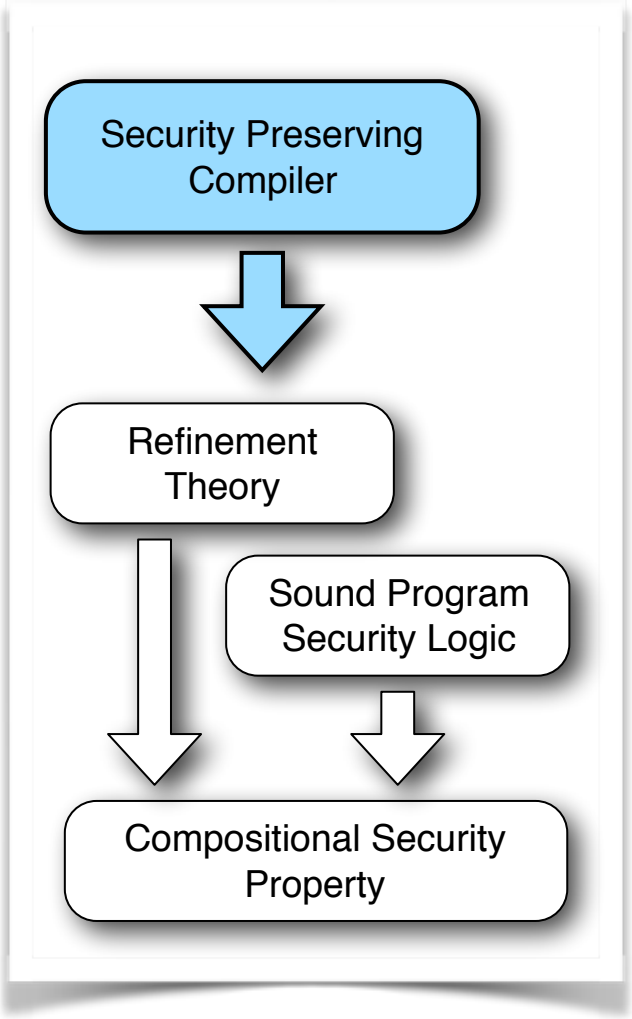
$\mathcal{I}$ is preserved, as relational invariant

SECURITY-PRESERVING COMPILATION

SECURITY-PRESERVING COMPILATION



SECURITY-PRESERVING COMPILATION



Case Study: Compilation to Idealised RISC

instr ::= **lock** l
| **release** l
| **load** reg x
| **store** x reg
| **Jump** label
| **Jz** label
| **Noop**
| **MoveK** reg w
| **Mov** reg reg
| **Binop** op reg reg

Case Study: Compilation to Idealised RISC

instr ::= **lock** l
release l
load reg x
store x reg
Jump label
Jz label
Noop
MoveK reg w
Mov reg reg
Binop op reg reg

Robert Sison

Current Status

- Proof finished for source programs that are secure and never branch on High data
  - This is the simple case where the compiler doesn't have to do anything except compile the program properly to preserve noninterference
- Sufficient for the CDDC model we have verified
- First example compiler for shared-memory concurrent programs proved to preserve noninterference
- Next steps: High conditionals, weak memory, …

How to leverage it?

- Constant-time RISC simulator for concurrent programs
  - A mini RTOS/VM

CONCLUSION

CONCLUSION

Moving from verified secure kernels to verified secure systems:

CONCLUSION

Moving from verified secure kernels to verified secure systems:

- concurrency ~> compositionality ~>
  timing-sensitivity, assume-guarantee reasoning
- dynamic policies (value-dependence)
- compositional secure refinement ~>
  coupling invariants for preservation of timing-sensitive security

CONCLUSION

Moving from verified secure kernels to verified secure systems:

- concurrency ~> compositionality ~>
  timing-sensitivity, assume-guarantee reasoning
- dynamic policies (value-dependence)
- compositional secure refinement ~>
  coupling invariants for preservation of timing-sensitive security

Still lots more to be done.

Thank You


toby.murray@unimelb.edu.au


http://people.eng.unimelb.edu.au/tobym


@tobycmurray

Dependent Security Logic: Intuition

Dependent Security Logic: Intuition

inspired by Mantel et al. CSF 2011

Dependent Security Logic: Intuition

inspired by Mantel et al. CSF 2011

```
skip   /* control +=_m AsmNoW */;
skip   /* temp +=_m AsmNoRW */;
temp := buffer;
if control == 0 then
  low-var := temp
else
  high-var := temp
endif;
temp := 0;
skip   /* temp -=_m AsmNoRW */
```

Dependent Security Logic: Intuition

inspired by Mantel et al. CSF 2011

buffer's classification depends on control: is Low iff control == 0

```
skip   /* control +=_m AsmNoW */;
skip   /* temp +=_m AsmNoRW */;
temp := buffer;
if control == 0 then
  low-var := temp
else
  high-var := temp
endif;
temp := 0;
skip   /* temp -=_m AsmNoRW */
```

Dependent Security Logic: Intuition

inspired by Mantel et al. CSF 2011

buffer's *classification depends on* control: *is* Low *iff* control == 0

assume nobody else can modify control

```
skip   /* control +=_m AsmNoW */;
skip   /* temp +=_m AsmNoRW */;
temp := buffer;
if control == 0 then
  low-var := temp
else
  high-var := temp
endif;
temp := 0;
skip   /* temp -=_m AsmNoRW */
```

Dependent Security Logic: Intuition

inspired by Mantel et al. CSF 2011

buffer's *classification depends on* control: *is* Low *iff* control == 0

assume nobody else can read or modify temp

```
skip   /* control +=_m AsmNoW */;
skip   /* temp +=_m AsmNoRW */;
temp := buffer;
if control == 0 then
  low-var := temp
else
  high-var := temp
endif;
temp := 0;
skip   /* temp -=_m AsmNoRW */
```

Dependent Security Logic: Intuition

inspired by Mantel et al. CSF 2011

buffer's *classification depends on* control: *is* Low *iff* control == 0

is secure (only) because temp not readable

```
skip   /* control +=_m AsmNoW */;
skip   /* temp +=_m AsmNoRW */;
temp := buffer;
if control == 0 then
  low-var := temp
else
  high-var := temp
endif;
temp := 0;
skip   /* temp -=_m AsmNoRW */
```

Dependent Security Logic: Intuition

inspired by Mantel et al. CSF 2011

buffer's *classification depends on* control: *is* Low *iff* control == 0

since control is not writable, its value now is unchanged from when buffer was read

```
skip   /* control +=_m AsmNoW */;
skip   /* temp +=_m AsmNoRW */;
temp := buffer;
if control == 0 then
  low-var := temp
else
  high-var := temp
endif;
temp := 0;
skip   /* temp -=_m AsmNoRW */
```

Dependent Security Logic: Intuition

inspired by Mantel et al. CSF 2011

buffer's *classification depends on* control: *is* Low *iff* control == 0

secure because buffer was Low when read, so temp holds Low data

```
skip   /* control +=_m AsmNoW */;
skip   /* temp +=_m AsmNoRW */;
temp := buffer;
if control == 0 then
  low-var := temp
else
  high-var := temp
endif;
temp := 0;
skip   /* temp -=_m AsmNoRW */
```

Dependent Security Logic: Intuition

inspired by Mantel et al. CSF 2011

buffer's *classification depends on* control: *is* Low *iff* control == 0

```
skip   /* control +=_m AsmNoW */;
skip   /* temp +=_m AsmNoRW */;
temp := buffer;
if control == 0 then
  low-var := temp
else
  high-var := temp
endif;
temp := 0;
skip   /* temp -=_m AsmNoRW */
```

temp is now guaranteed Low

Dependent Security Logic: Intuition

inspired by Mantel et al. CSF 2011

buffer's *classification depends on* control: *is* Low *iff* control == 0

```
skip   /* control +=_m AsmNoW */;
skip   /* temp +=_m AsmNoRW */;
temp := buffer;
if control == 0 then
  low-var := temp
else
  high-var := temp
endif;
temp := 0;
skip   /* temp -=_m AsmNoRW */
```

allow others to read temp: secure (only) because temp guaranteed Low

Dependent Security Logic: Intuition

inspired by Mantel et al. CSF 2011

buffer's classification depends on control: is Low iff control == 0

```
skip   /* control +=_m AsmNoW */;
skip   /* temp +=_m AsmNoRW */;
temp := buffer;
if control == 0 then
  low-var := temp
else
  high-var := temp
endif;
temp := 0;
skip   /* temp -=_m AsmNoRW */
```

Dependent Security Logic: Intuition

inspired by Mantel et al. CSF 2011

buffer's classification depends on control: is Low iff control == 0
temp is classified statically Low

```
skip   /* control +=_m AsmNoW */;
skip   /* temp +=_m AsmNoRW */;
temp := buffer;
if control == 0 then
  low-var := temp
else
  high-var := temp
endif;
temp := 0;
skip   /* temp -=_m AsmNoRW */
```

Type System: Stability and Rewriting

```
skip   /* control +=_m AsmNoW */;
skip   /* temp +=_m AsmNoRW */;
temp := buffer;
if control == 0 then
  skip   /* control -=_m AsmNoW */
  low-var := temp
else
  high-var := temp
endif;
temp := 0;
skip   /* temp -=_m AsmNoRW */
```

Type System: Stability and Rewriting

[t : {c=0}] ({c},{t}) {c=0}

Γ $\mathcal{S}$ P

```
skip   /* control +=_m AsmNoW */;
skip   /* temp +=_m AsmNoRW */;
temp := buffer;
if control == 0 then
  skip   /* control -=_m AsmNoW */
  low-var := temp
else
  high-var := temp
endif;
temp := 0;
skip   /* temp -=_m AsmNoRW */
```

Type System: Stability and Rewriting

[t : {c=0}] ({c},{t}) {c=0}

Γ $\mathcal{S}$ P

```
skip   /* control +=_m AsmNoW */;
skip   /* temp +=_m AsmNoRW */;
temp := buffer;
if control == 0 then
  skip   /* control -=_m AsmNoW */
  low-var := temp
else
  high-var := temp
endif;
temp := 0;
skip   /* temp -=_m AsmNoRW */
```

Type System: Stability and Rewriting

would make this type meaningless

[t : {c=0}] ({c},{t}) {c=0}

Γ $\mathcal{S}$ P

```
skip   /* control +=_m AsmNoW */;
skip   /* temp +=_m AsmNoRW */;
temp := buffer;
if control == 0 then
skip   /* control -=_m AsmNoW */
  low-var := temp
else
  high-var := temp
endif;
temp := 0;
skip   /* temp -=_m AsmNoRW */
```

Type System: Stability and Rewriting

would make this type meaningless

[t : {c=0}] ({c},{t}) {c=0}

Γ $\mathcal{S}$ P

```
skip   /* control +=_m AsmNoW */;
skip   /* temp +=_m AsmNoRW */;
temp := buffer;
if control == 0 then
  skip   /* control -=_m AsmNoW */
  low-var := temp
else
  high-var := temp
endif;
temp := 0;
skip   /* temp -=_m AsmNoRW */
```

Type System: Stability and Rewriting

would make this type meaningless

[t : {c=0}] ({c},{t}) {c=0}

Γ $\mathcal{S}$ P

but we know already that temp is Low

```
skip   /* control +=_m AsmNoW */;
skip   /* temp +=_m AsmNoRW */;
temp := buffer;
if control == 0 then
  skip   /* control -=_m AsmNoW */
  low-var := temp
else
  high-var := temp
endif;
temp := 0;
skip   /* temp -=_m AsmNoRW */
```

Type System: Stability and Rewriting

would make this type meaningless

[t : {c=0}] ({c},{t}) {c=0}

Γ $\mathcal{S}$ P

but we know already that temp is Low

i.e. under *P*, its type is equivalent to {}, which doesn't mention control

```
skip   /* control +=_m AsmNoW */;
skip   /* temp +=_m AsmNoRW */;
temp := buffer;
if control == 0 then
  skip   /* control -=_m AsmNoW */
  low-var := temp
else
  high-var := temp
endif;
temp := 0;
skip   /* temp -=_m AsmNoRW */
```

Type System: Stability and Rewriting

would make this type meaningless

[t : {c=0}] ({c},{t}) {c=0}

Γ $\mathcal{S}$ P

but we know already that temp is Low

```
skip   /* control +=_m AsmNoW */;
skip   /* temp +=_m AsmNoRW */;
temp := buffer;
if control == 0 then
  skip   /* control -=_m AsmNoW */
  low-var := temp
else
  high-var := temp
endif;
temp := 0;
skip   /* temp -=_m AsmNoRW */
```

i.e. under *P*, its type is equivalent to {}, which doesn't mention control

Solution: allow types to be rewritten to equivalent ones under *P*

Type System: Stability and Rewriting

[t : {c=0}] ({c},{t}) {c=0}

Γ $\mathcal{S}$ P

```
skip   /* control +=_m AsmNoW */;
skip   /* temp +=_m AsmNoRW */;
temp := buffer;
if control == 0 then
  skip   /* control -=_m AsmNoW */
  low-var := temp
else
  high-var := temp
endif;
temp := 0;
skip   /* temp -=_m AsmNoRW */
```

Solution: allow types to be rewritten to equivalent ones under P

Type System: Stability and Rewriting

```
                                  skip   /* control +=_m AsmNoW */;
                                  skip   /* temp +=_m AsmNoRW */;
                                  temp := buffer;
                                  if control == 0 then
[t : {}]  ({c},{t})  {c=0}        skip   /* control -=_m AsmNoW */
                                    low-var := temp
                                  else
                                    high-var := temp
                                  endif;
                                  temp := 0;
                                  skip   /* temp -=_m AsmNoRW */
```

Γ $\mathcal{S}$ P

Solution: allow types to be rewritten to equivalent ones under P

Type System: Stability and Rewriting

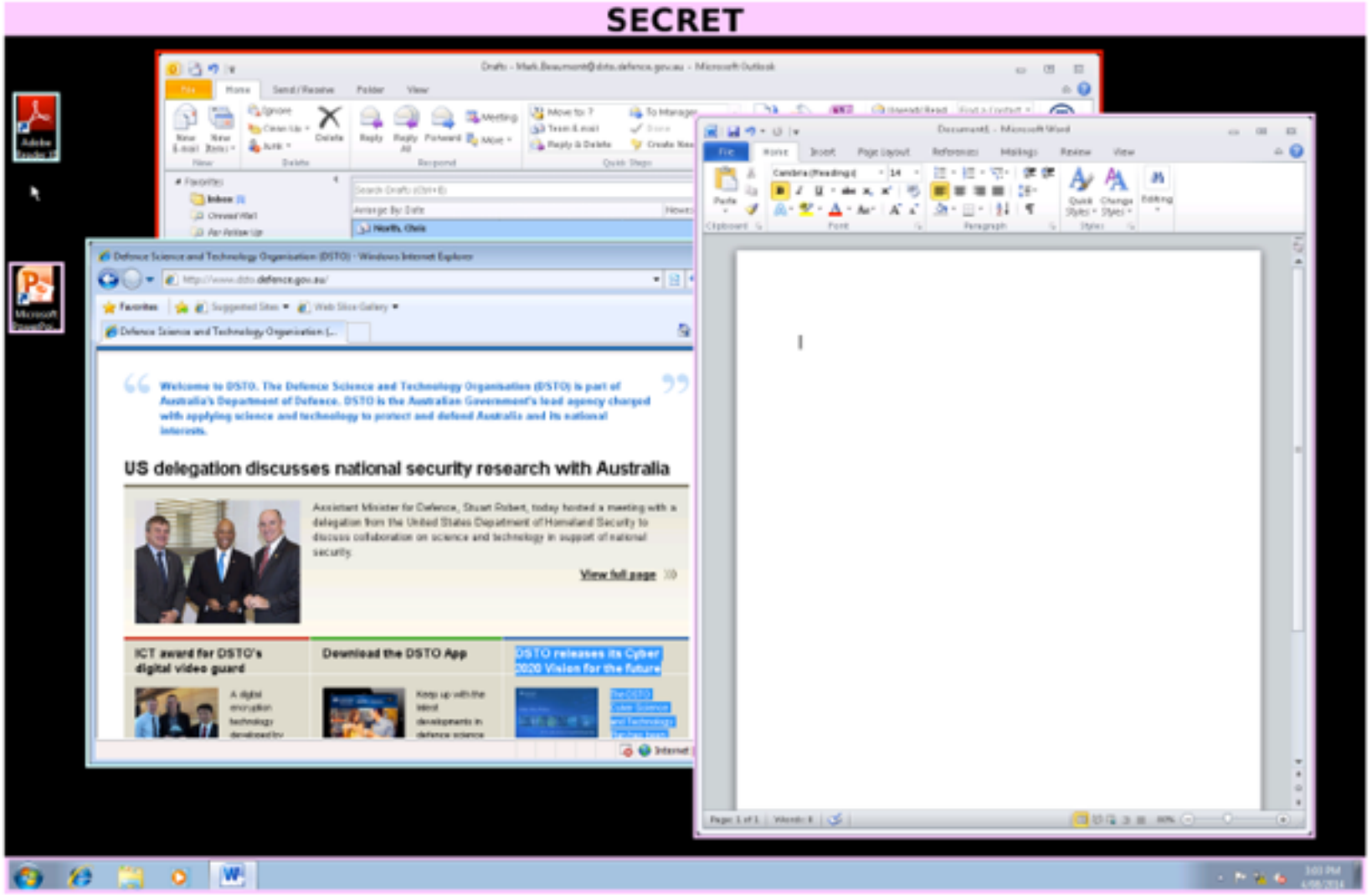
[t : {}] ({},{t}) {}

Γ $\mathcal{S}$ P

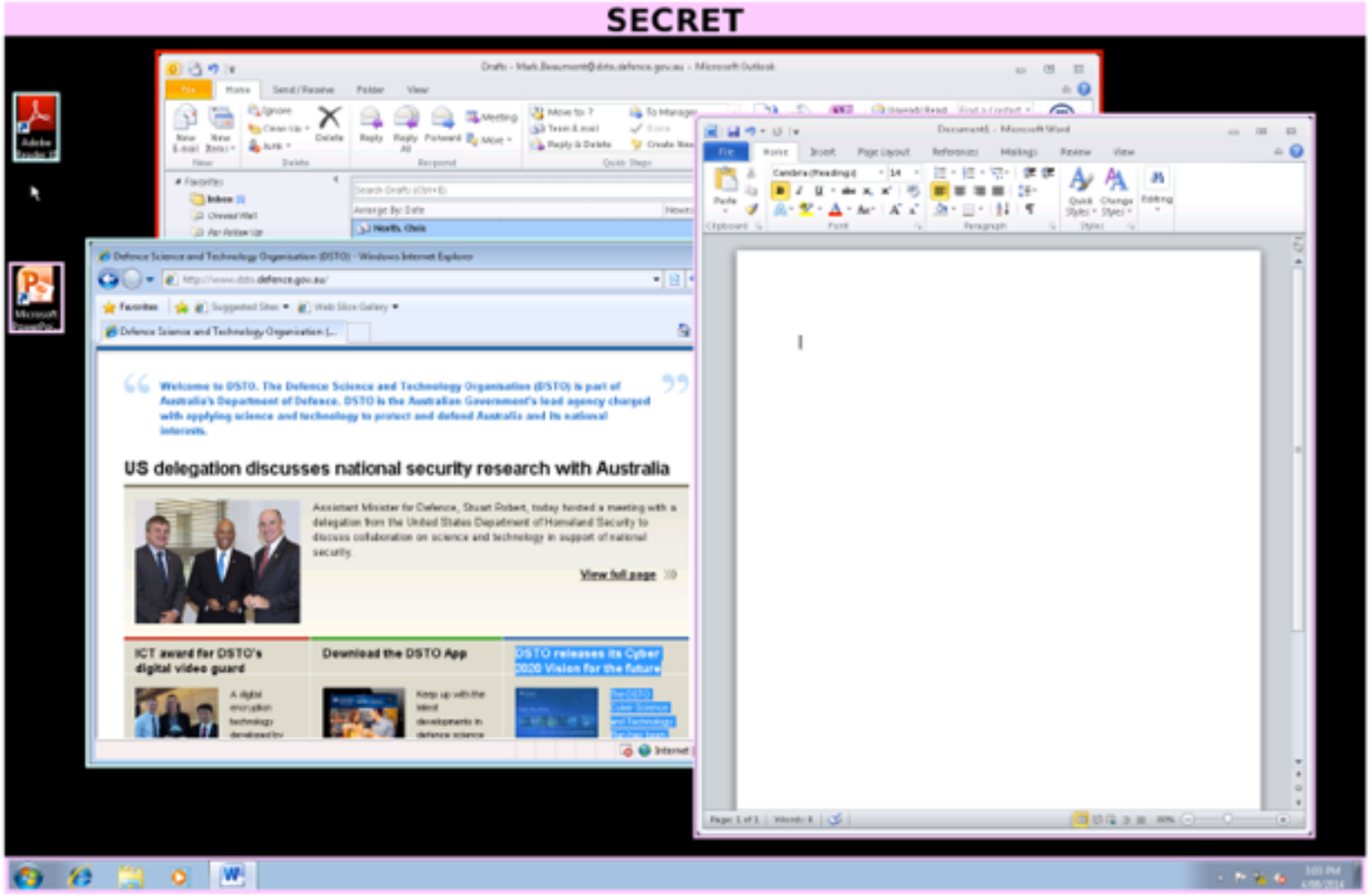
```
skip   /* control +=_m AsmNoW */;
skip   /* temp +=_m AsmNoRW */;
temp := buffer;
if control == 0 then
  skip   /* control -=_m AsmNoW */
  low-var := temp
else
  high-var := temp
endif;
temp := 0;
skip   /* temp -=_m AsmNoRW */
```

Solution: allow types to be rewritten to equivalent ones under *P*

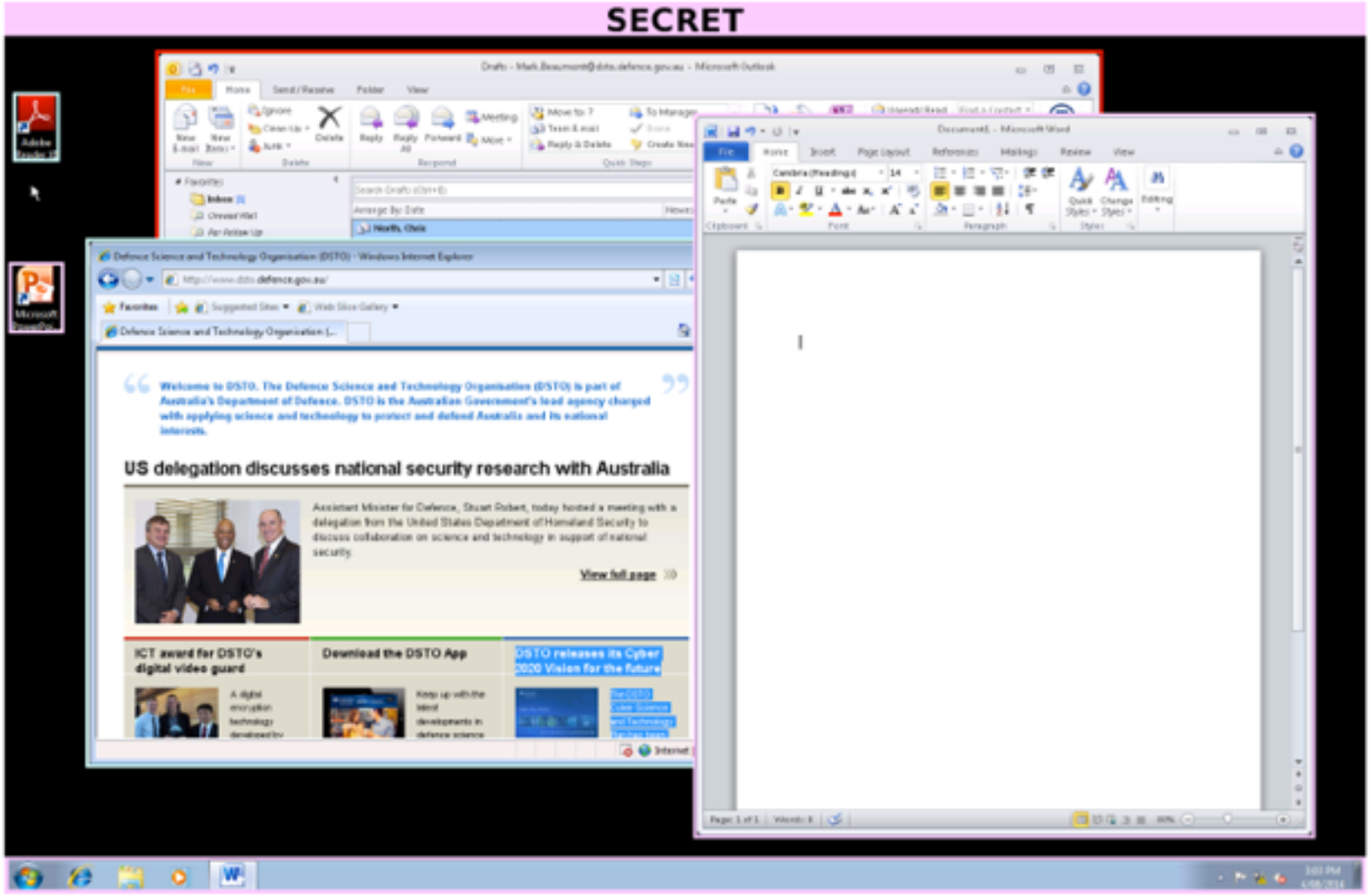
Display Composition: Untrusted Geometry



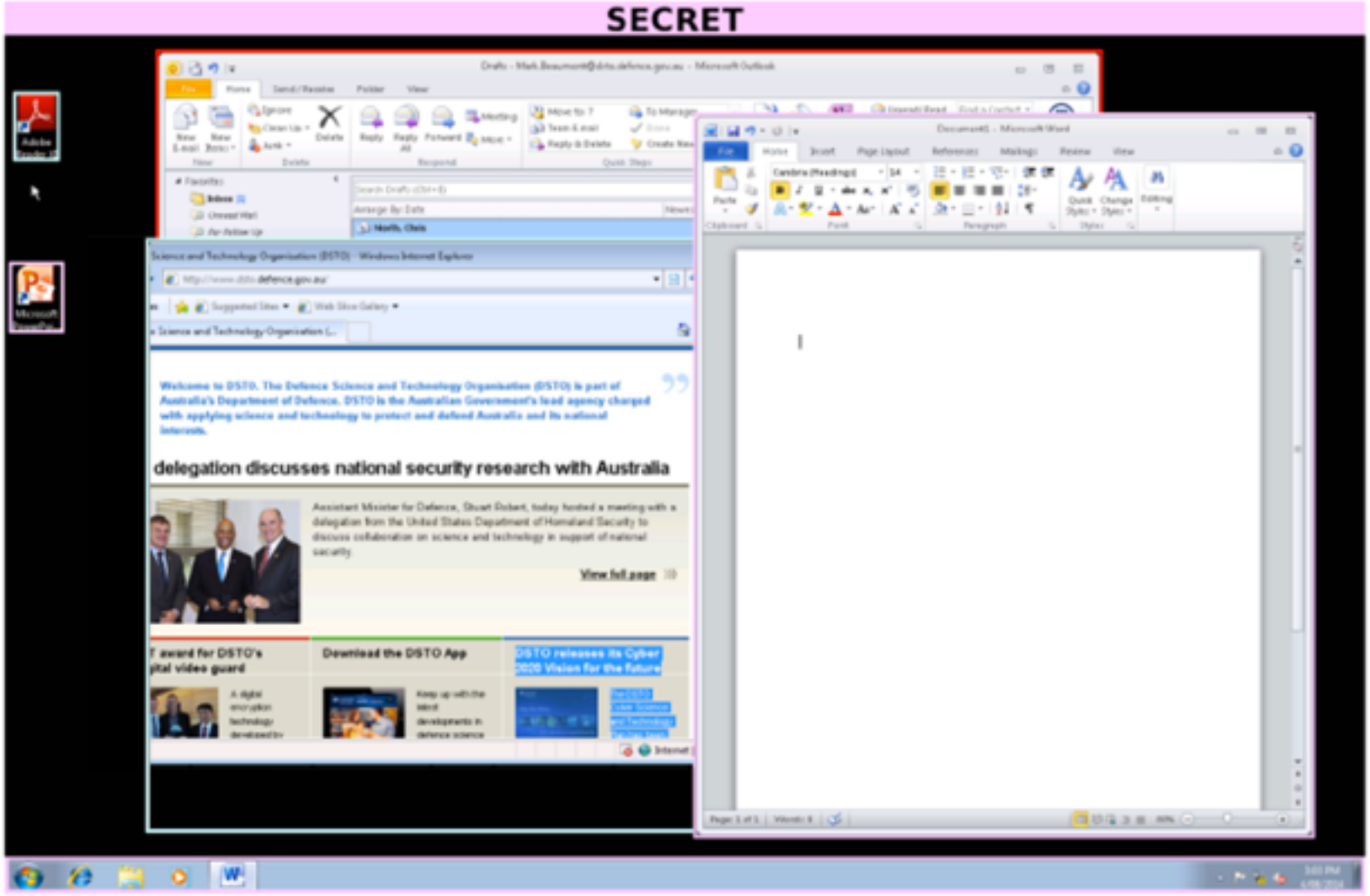
Dis[redacted]omposition: Untrusted Geometry



Dis  mposition: Untrusted Geometry



Dis  mposition: Untrusted Geometry



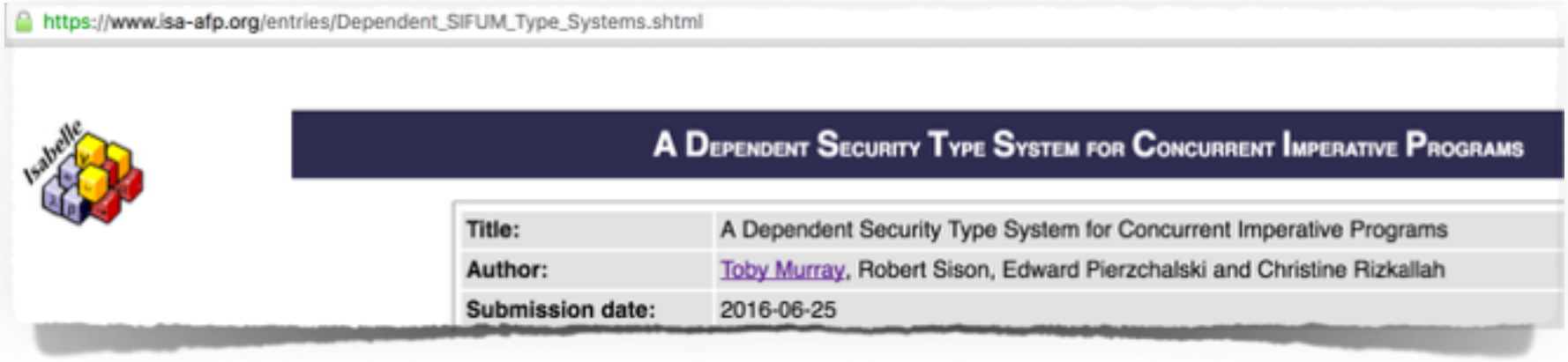
Formalisation

In Isabelle/HOL, available on Archive of Formal Proofs: http://isa-afp.org

Formalisation

In Isabelle/HOL, available on Archive of Formal Proofs: http://isa-afp.org

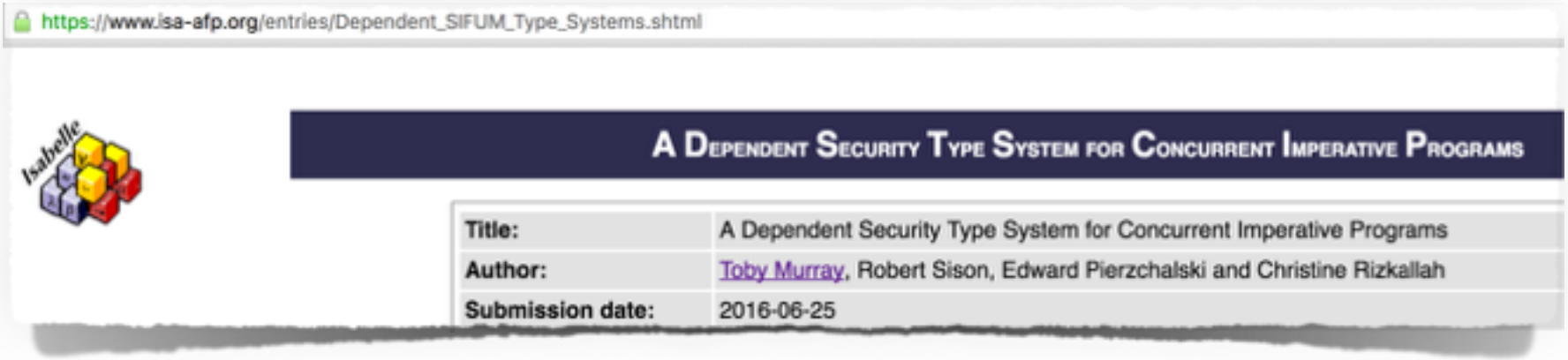
Compositional Security Property and Program Logic:



Formalisation

In Isabelle/HOL, available on Archive of Formal Proofs: http://isa-afp.org

Compositional Security Property and Program Logic:



Compositional Theory of Security Preserving Refinement: