Per-Thread Compositional Compilation for Confidentiality-Preserving Concurrent Programs

Robert Sison

13 Jan 2018

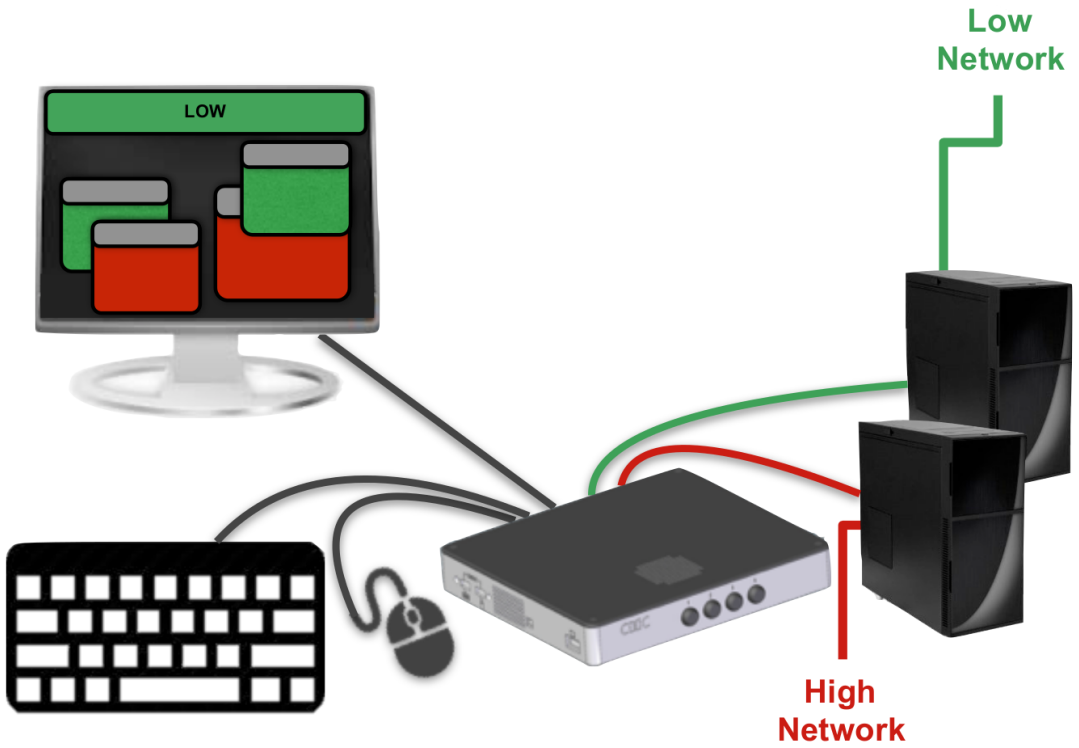
THE UNIVERSITY OF NEW SOUTH WALES

CSIRO

www.data61.csiro.au

A confidentiality-preserving program

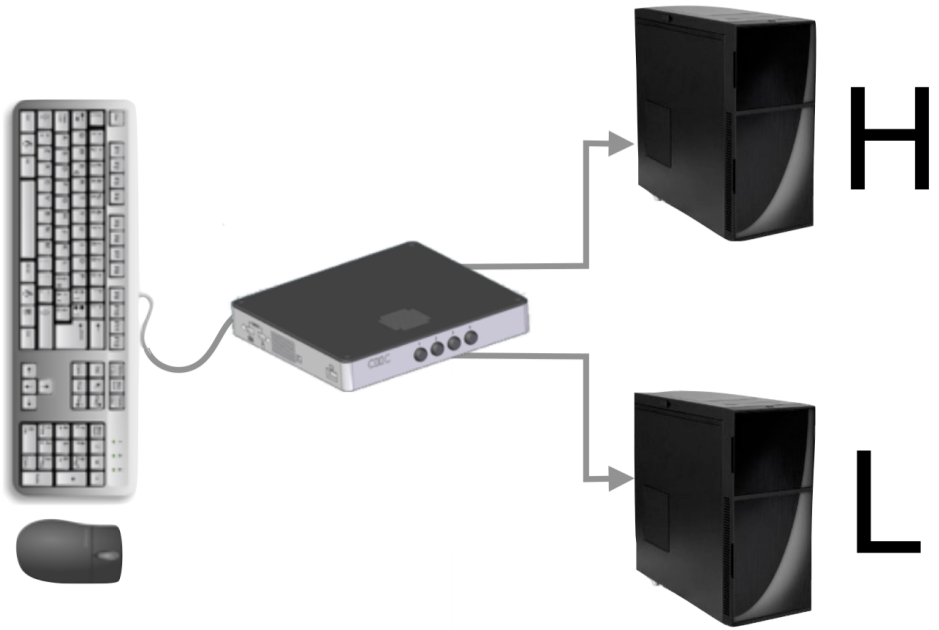
Cross Domain Desktop Compositor (CDDC)
[Beaumont et al, 2016]



Data61/DSTG project for de-duplicating user-facing hardware.

A confidentiality-preserving program

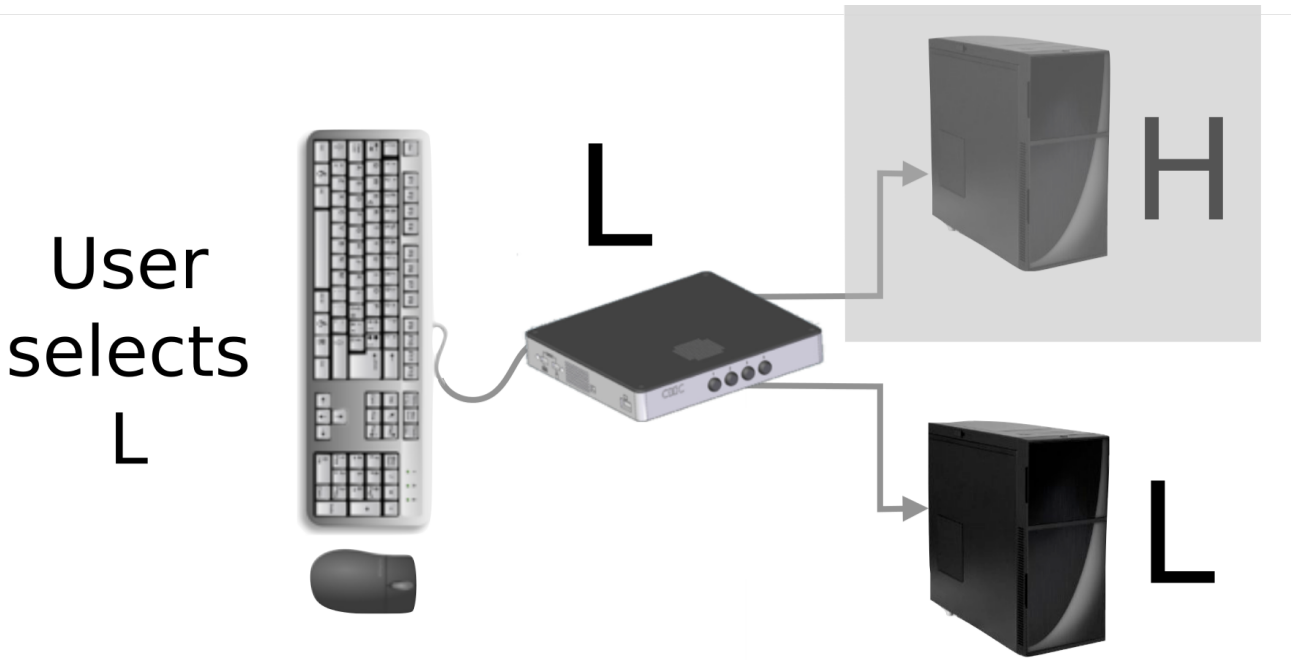
Cross Domain Desktop Compositor (CDDC)
[Beaumont et al, 2016]



Challenge #1: value-dependent security classifications

A confidentiality-preserving program

Cross Domain Desktop Compositor (CDDC)
[Beaumont et al, 2016]

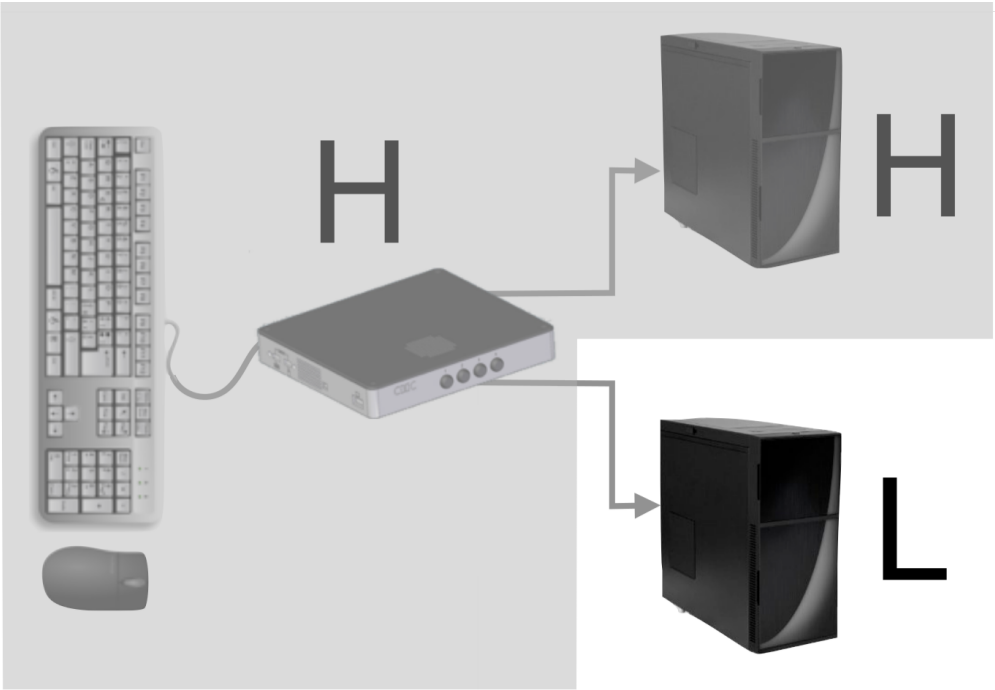


Challenge #1: value-dependent security classifications

A confidentiality-preserving program

Cross Domain Desktop Compositor (CDDC)
[Beaumont et al, 2016]

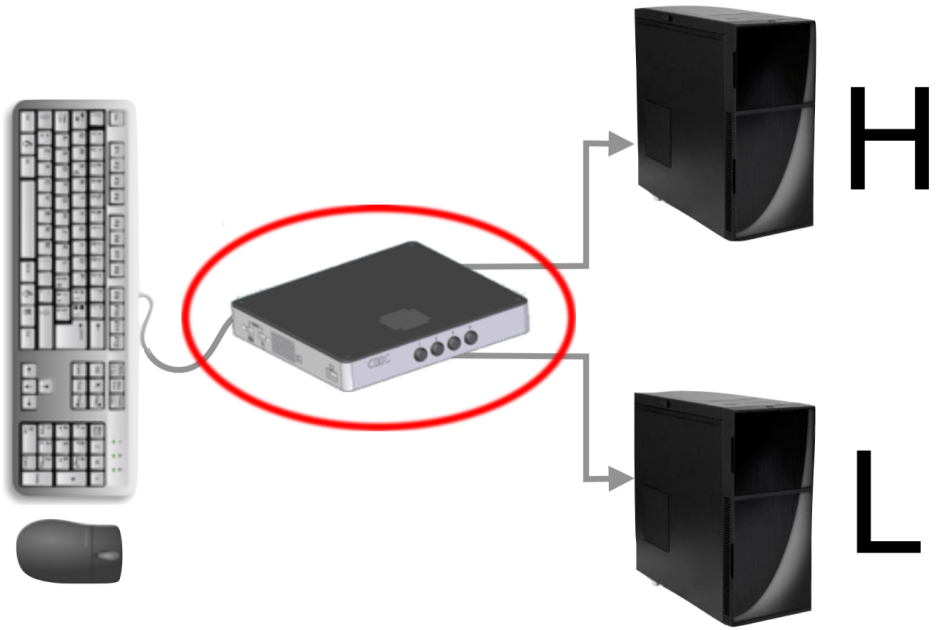
User selects H



Challenge #1: value-dependent security classifications

A confidentiality-preserving concurrent program

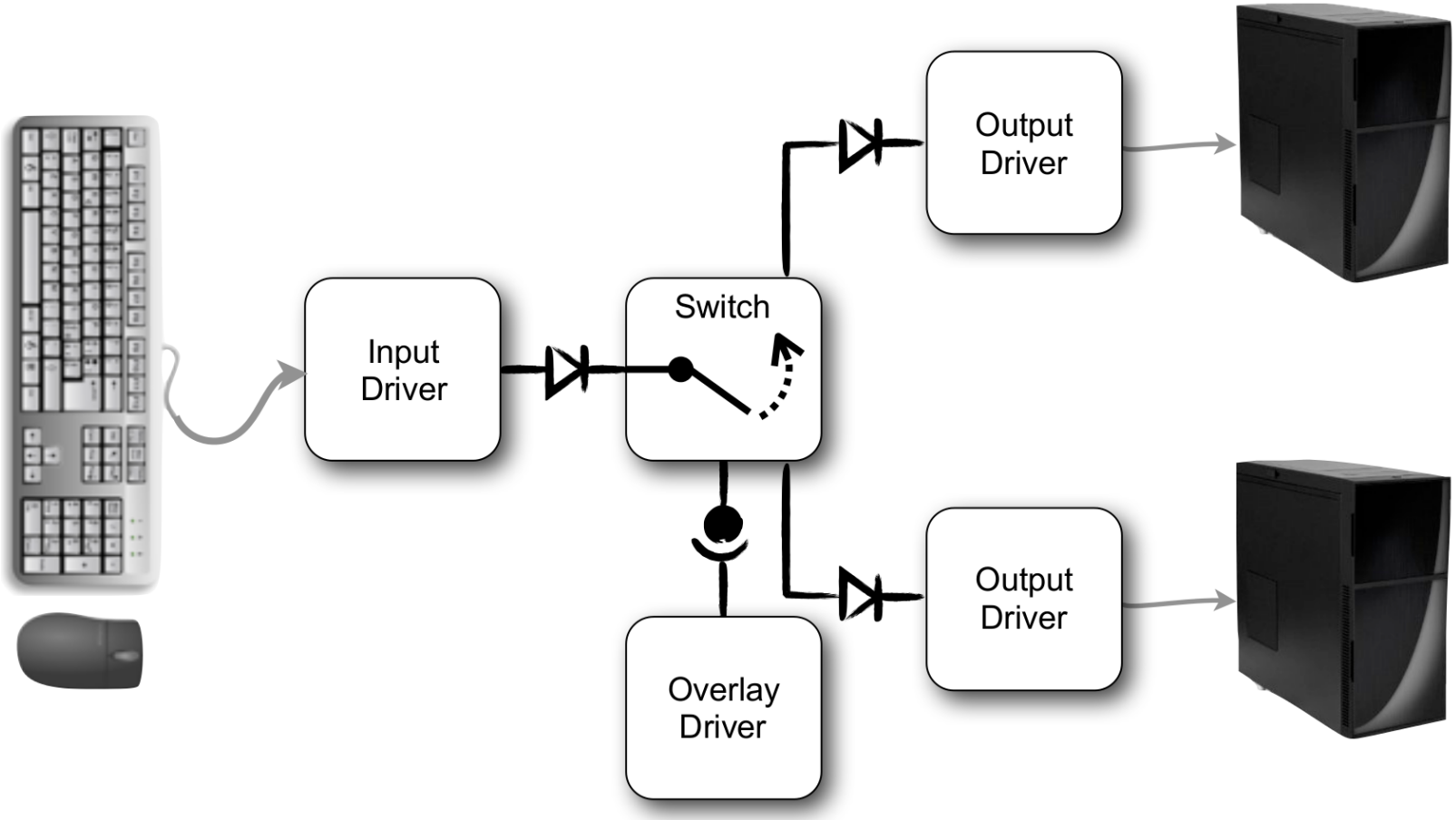
Cross Domain Desktop Compositor (CDDC)
[Beaumont et al, 2016]



Challenge #2: shared-variable concurrency

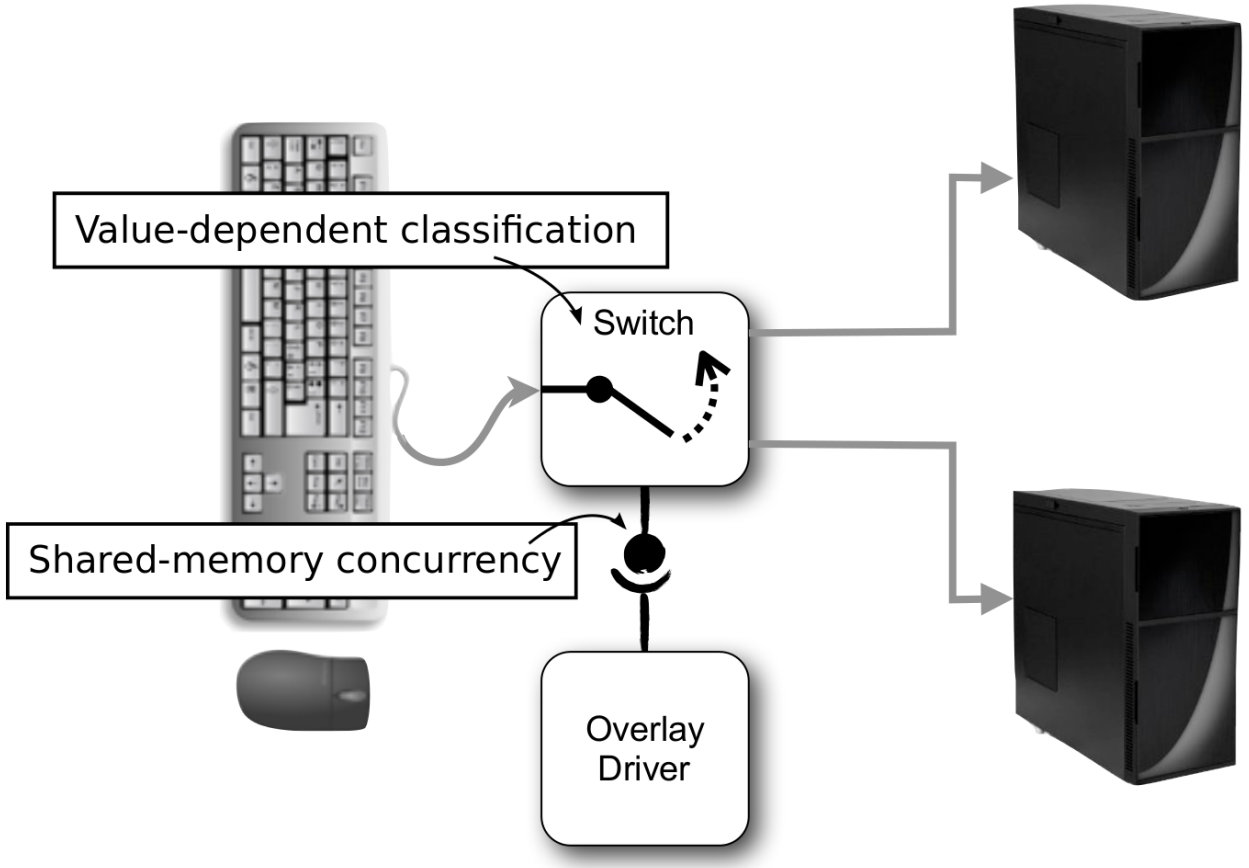
A confidentiality-preserving concurrent program

CDDC seL4-based software architecture:



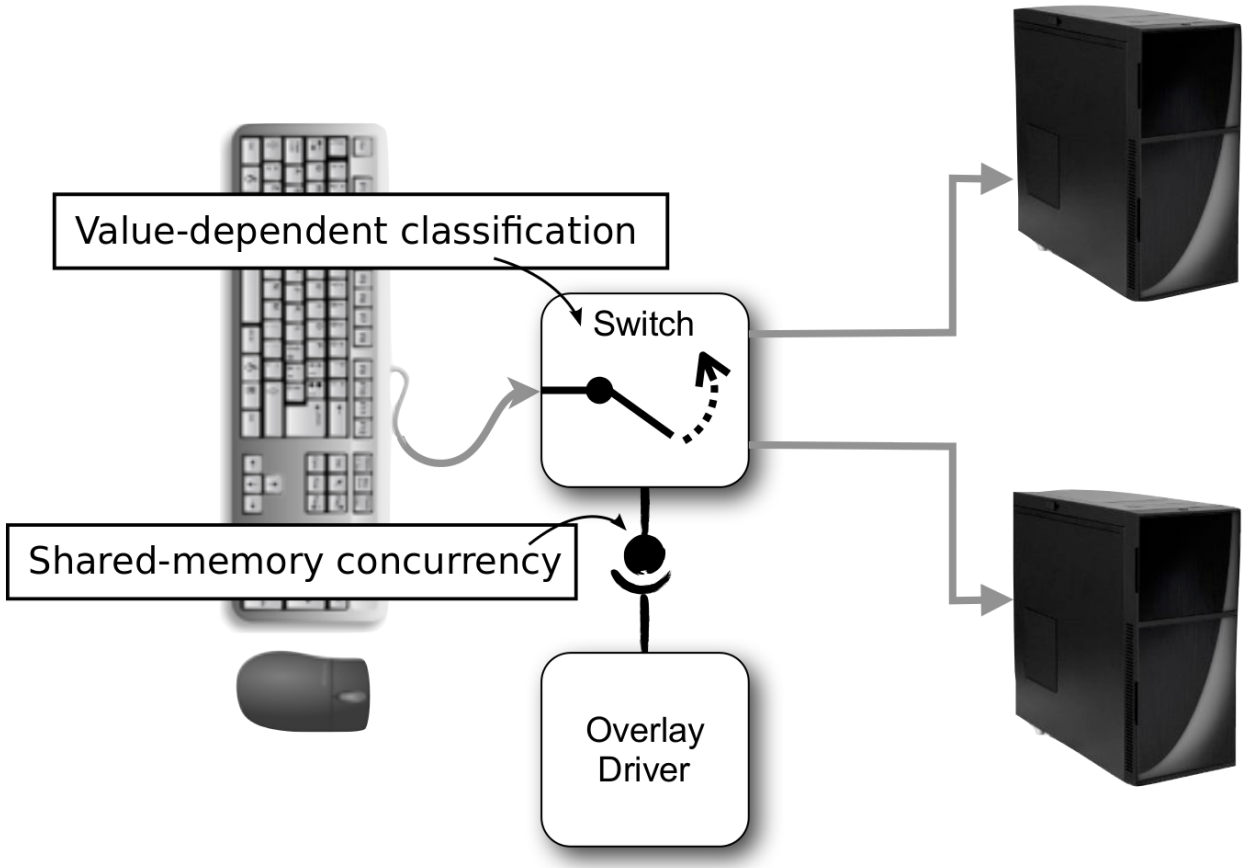
A confidentiality-preserving concurrent program

CDDC seL4-based software architecture (simplified model):



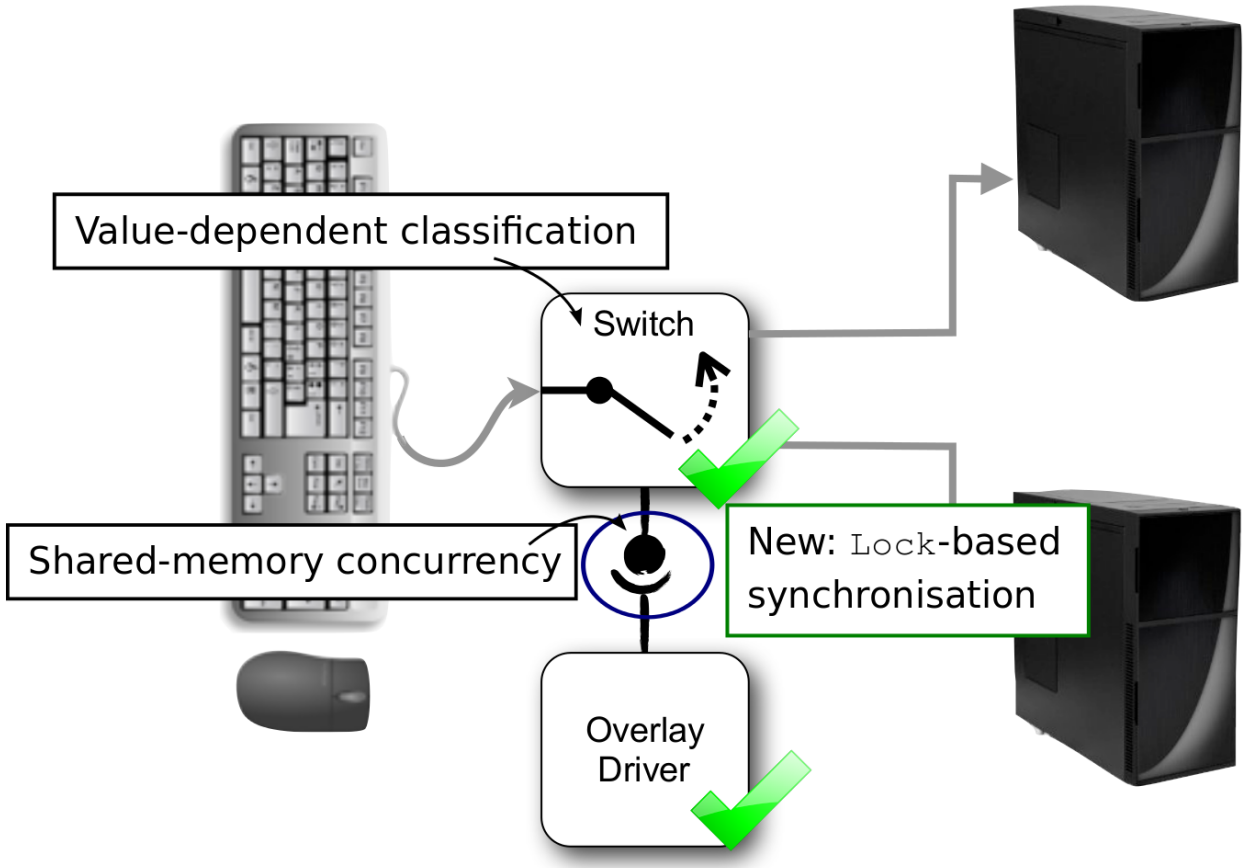
Per-thread compositional verification

Challenge #3: per-thread compositionality of proofs



Per-thread compositional verification

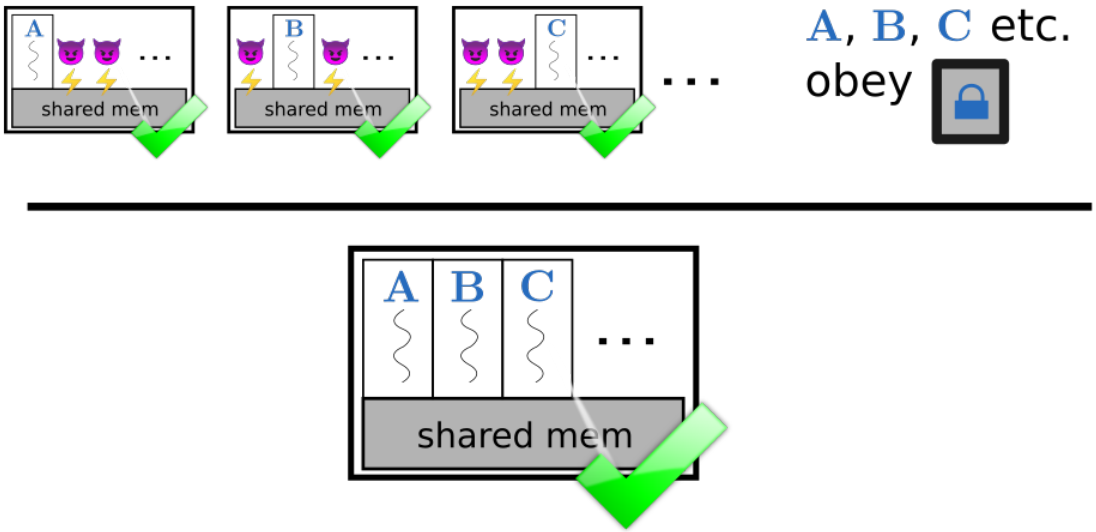
Challenge #3: per-thread compositionality of proofs



Mechanized in Isabelle/HOL. (More to appear: EuroS&P'18.)

Per-thread compositional verification

Challenge #3: per-thread compositionality of proofs

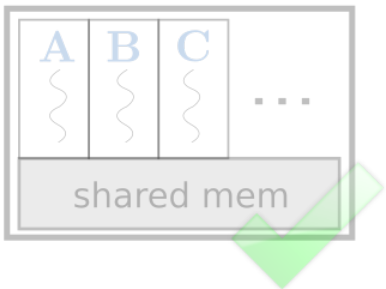


Per-thread compositional compilation

Challenge #3: per-thread compositionality of proofs

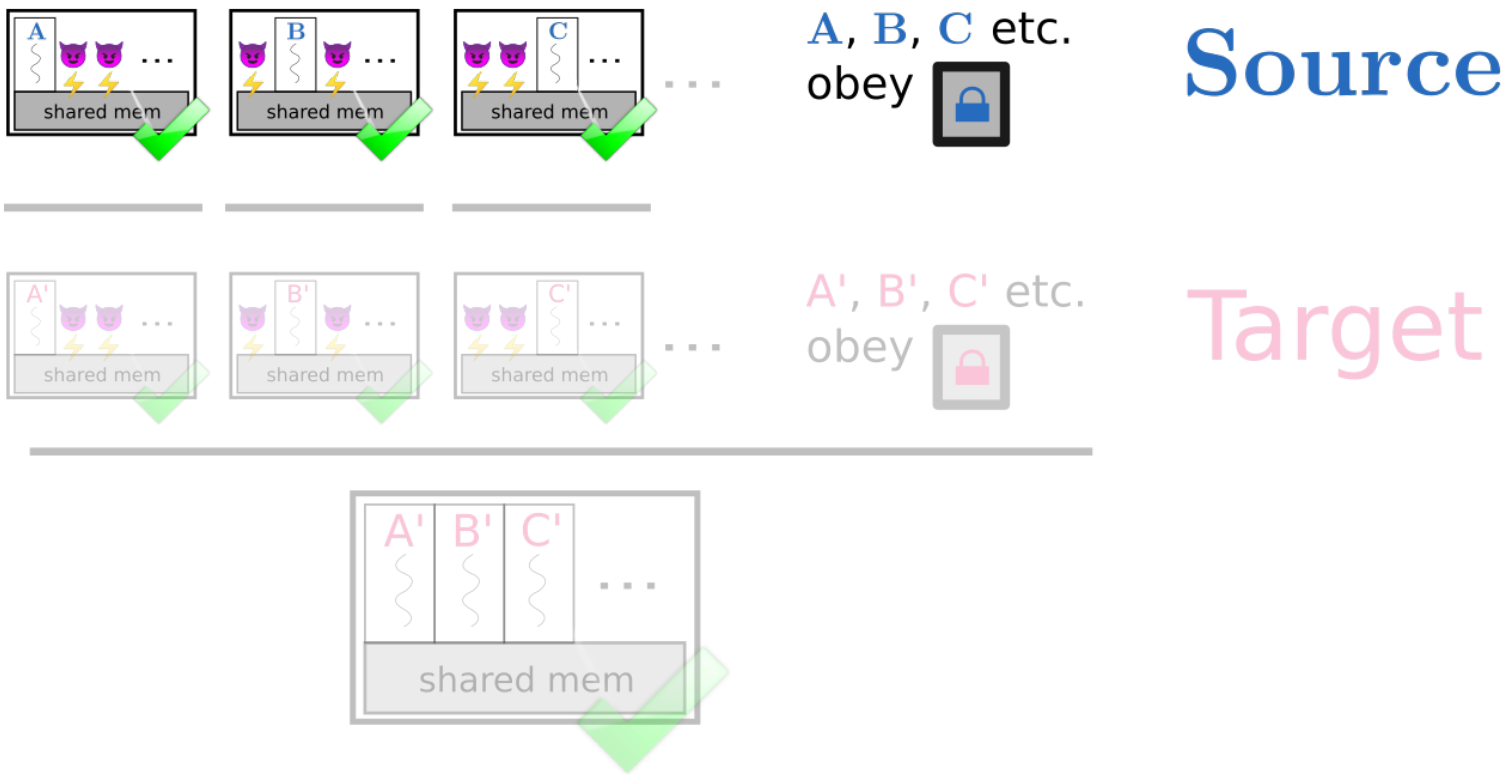
A, B, C etc. obey

Source



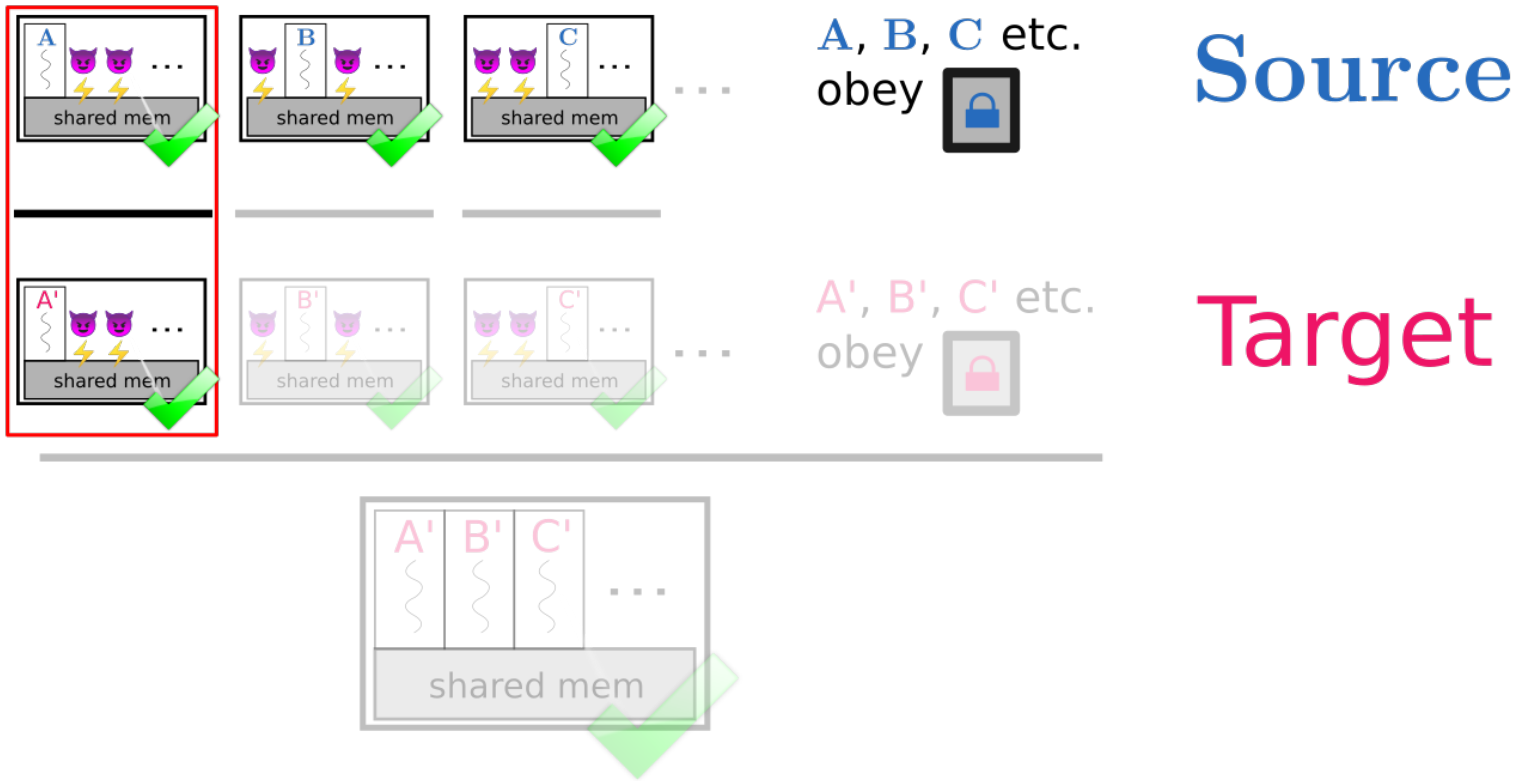
Per-thread compositional compilation

Challenge #3: per-thread compositionality of proofs



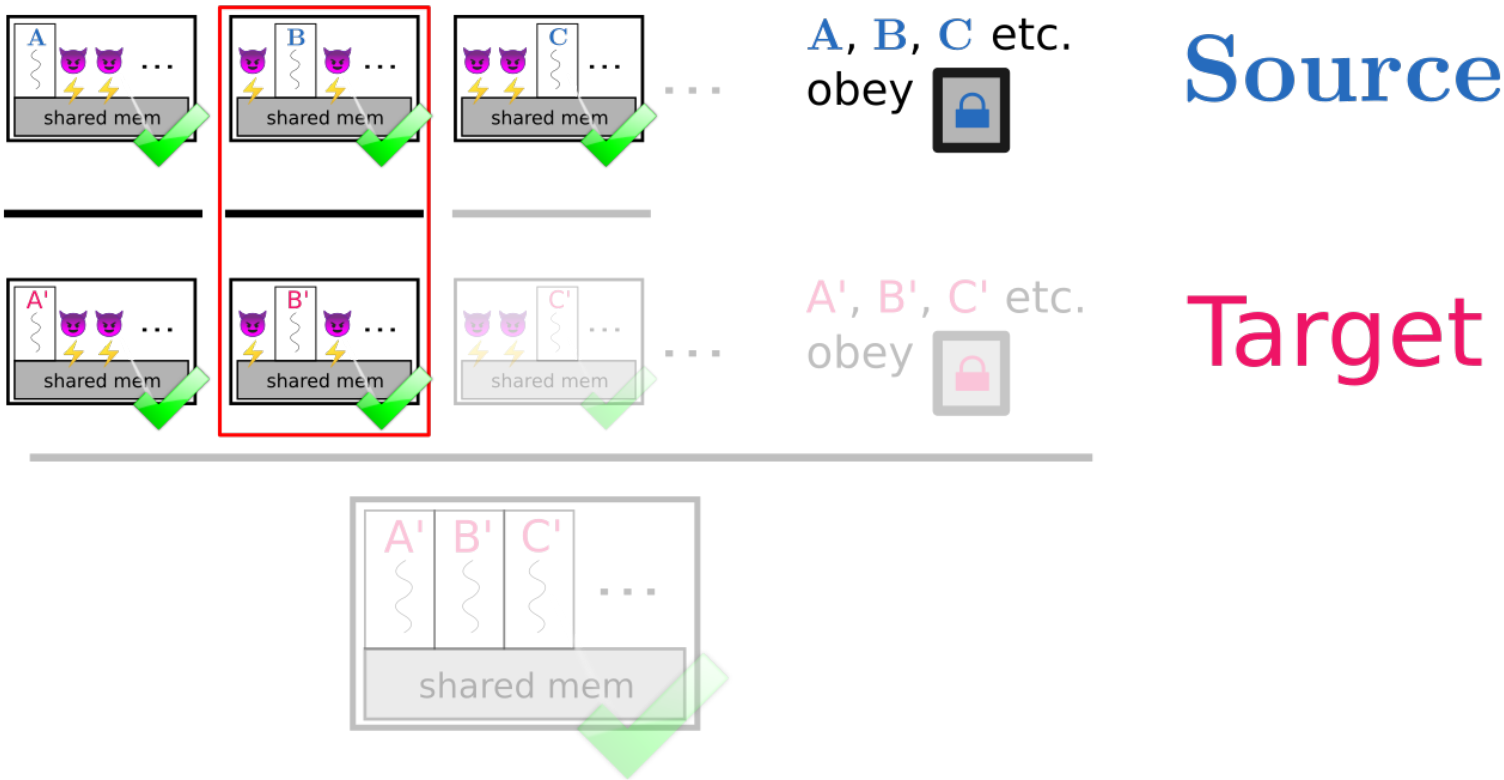
Per-thread compositional compilation

Challenge #3: per-thread compositionality of proofs



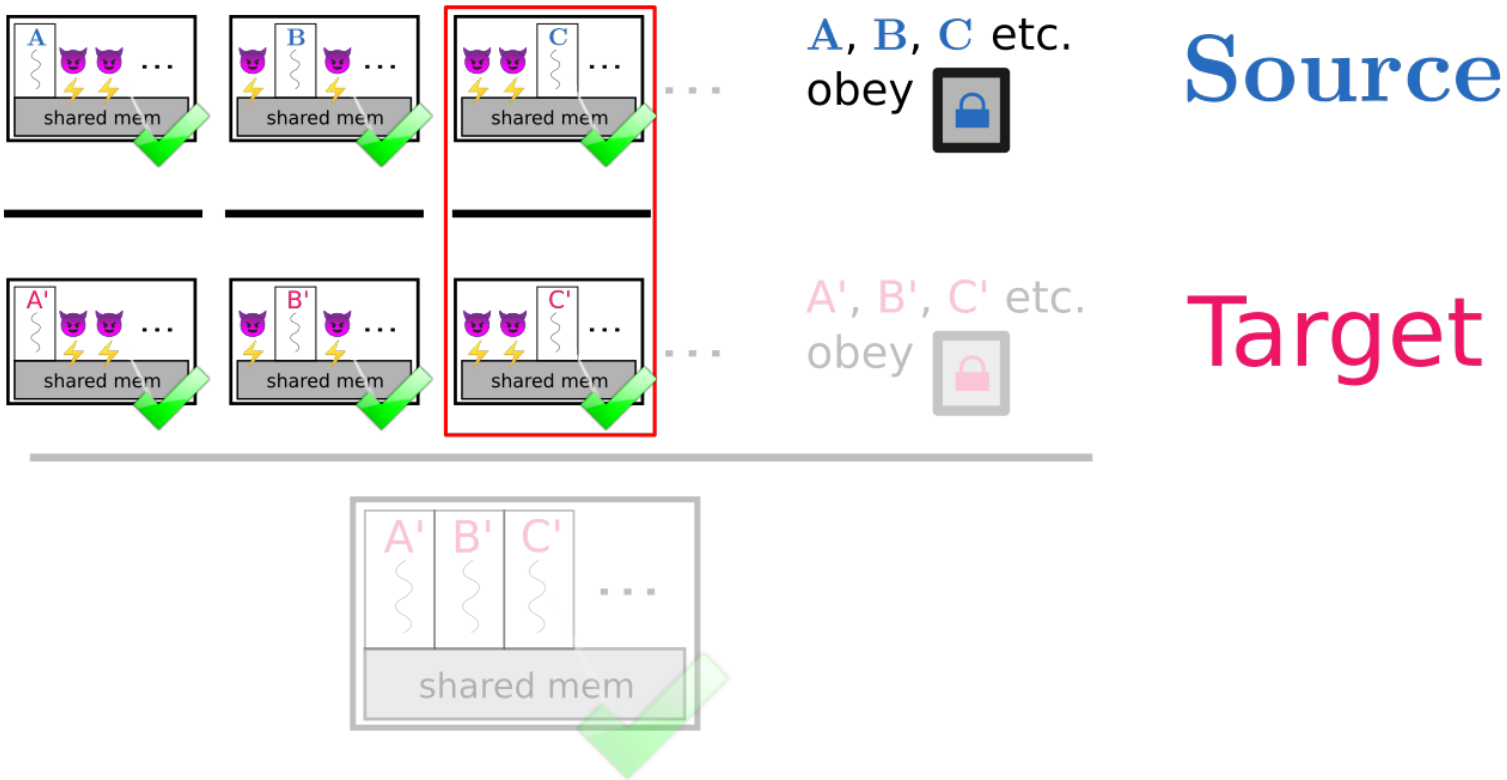
Per-thread compositional compilation

Challenge #3: per-thread compositionality of proofs



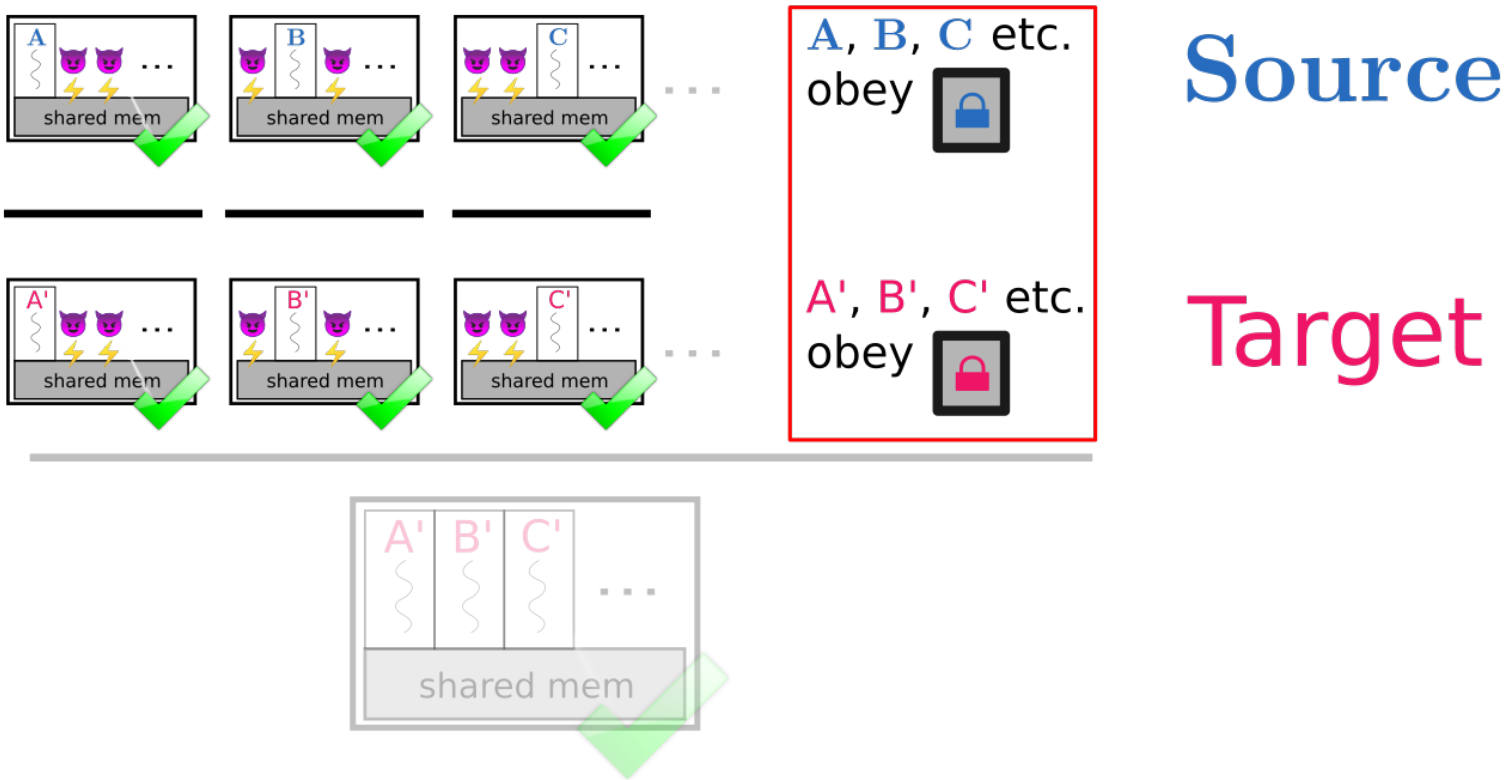
Per-thread compositional compilation

Challenge #3: per-thread compositionality of proofs



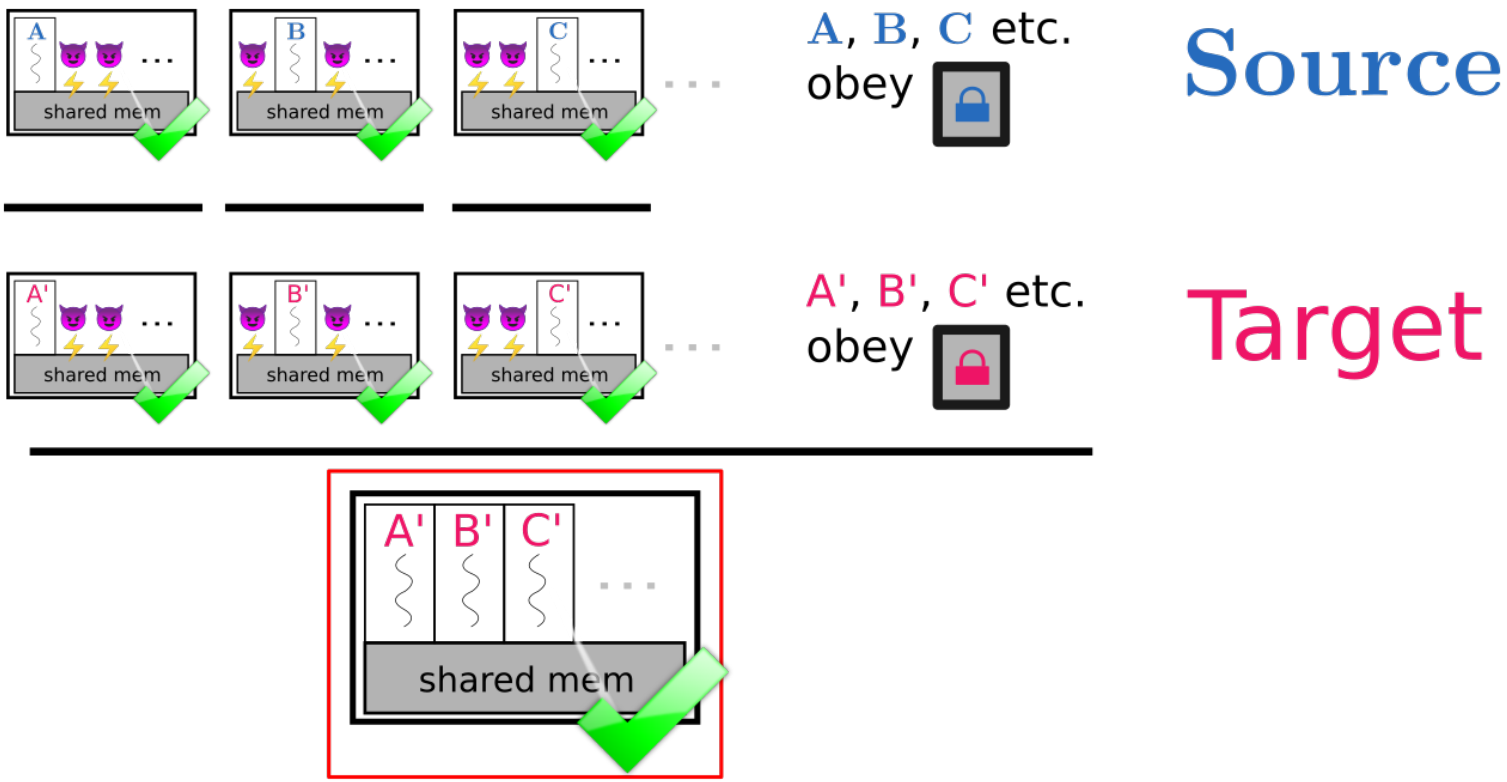
Per-thread compositional compilation

Challenge #3: per-thread compositionality of proofs



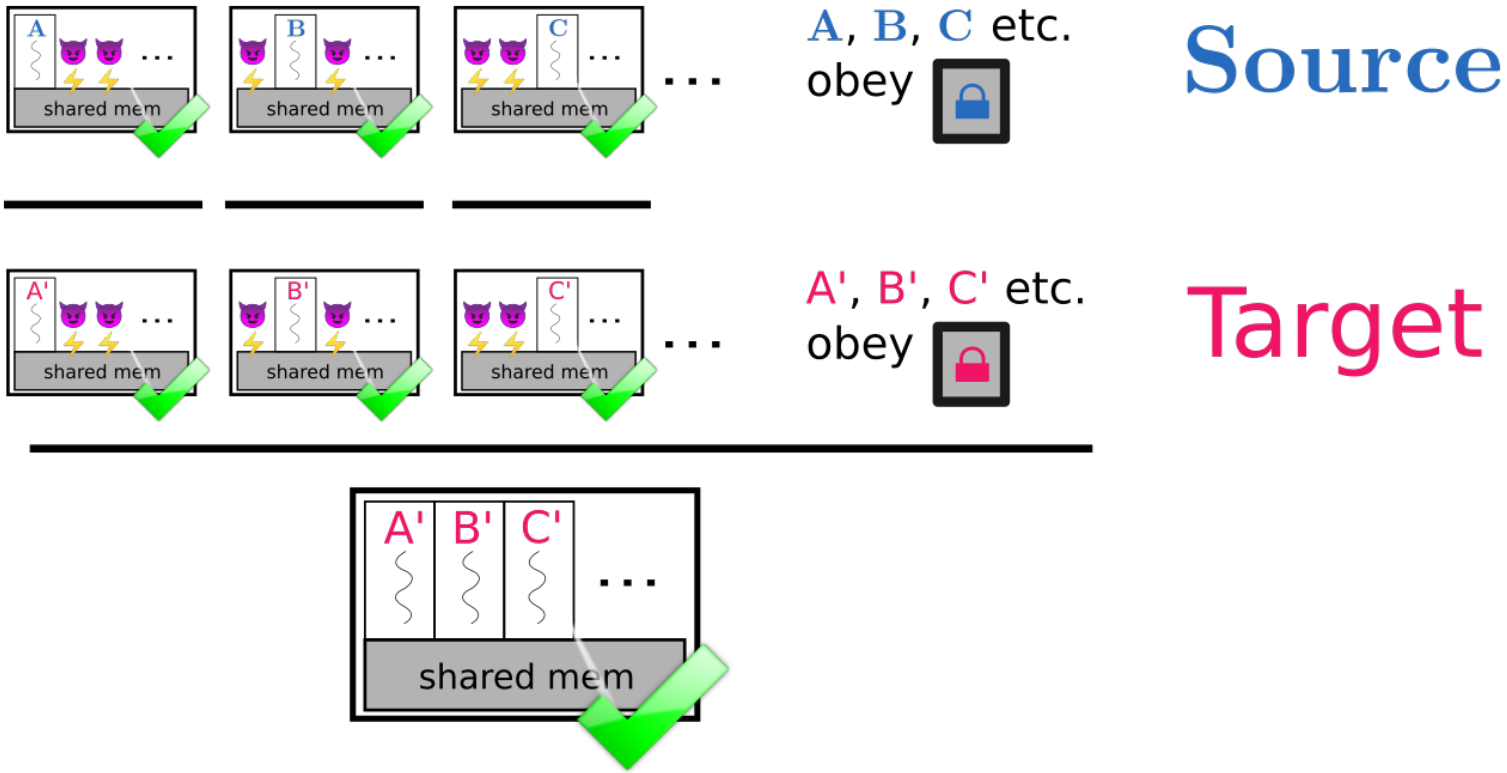
Per-thread compositional compilation

Challenge #3: per-thread compositionality of proofs



This talk

Part 1: Concurrent value-dependent noninterference
Part 2: Per-thread compositional refinement
Part 3: While-to-RISC compiler verification

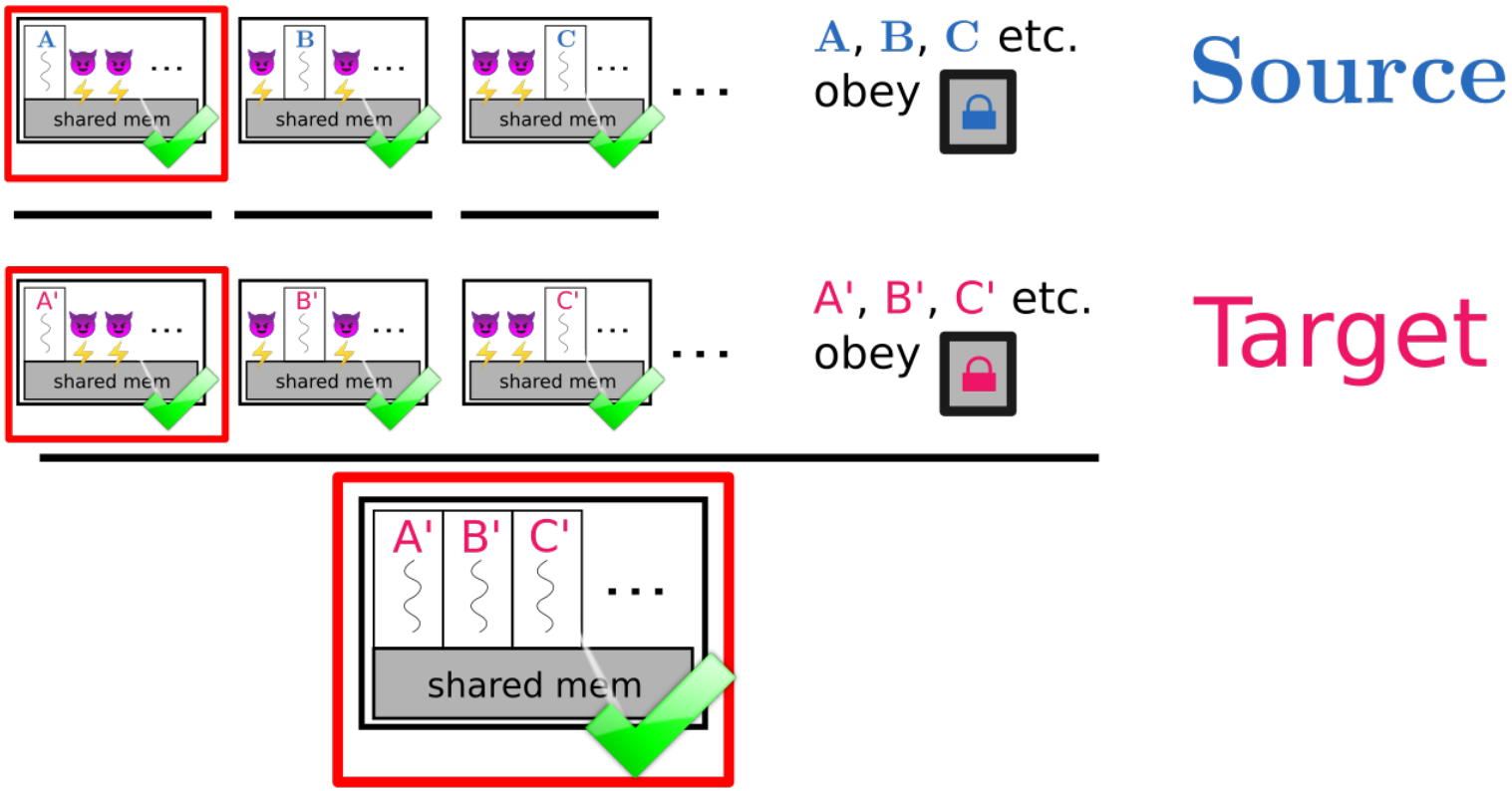


This talk: a preview

Part 1: Concurrent value-dependent noninterference

Part 2: Per-thread compositional refinement

Part 3: While-to-RISC compiler verification

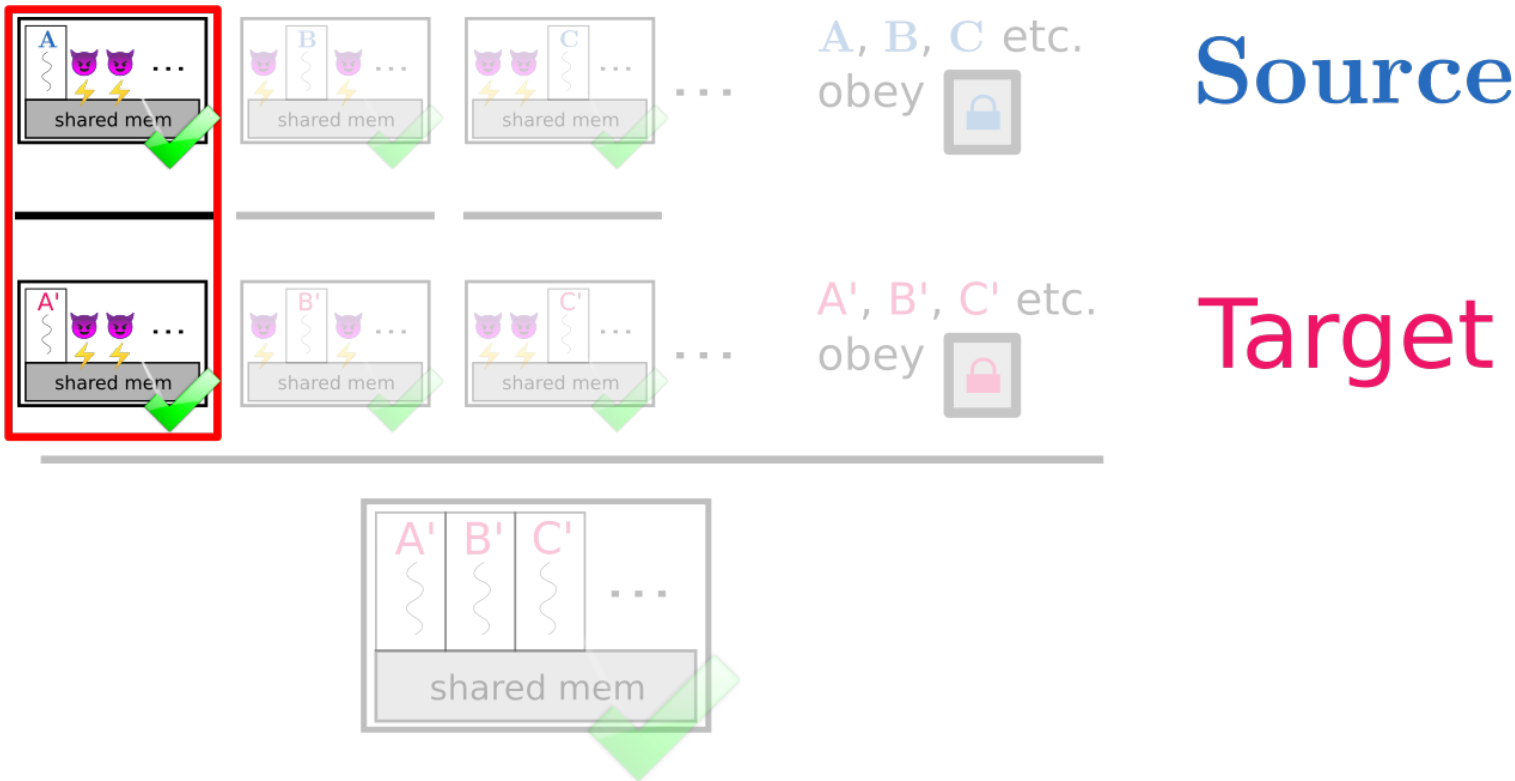


This talk: a preview

Part 1: Concurrent value-dependent noninterference

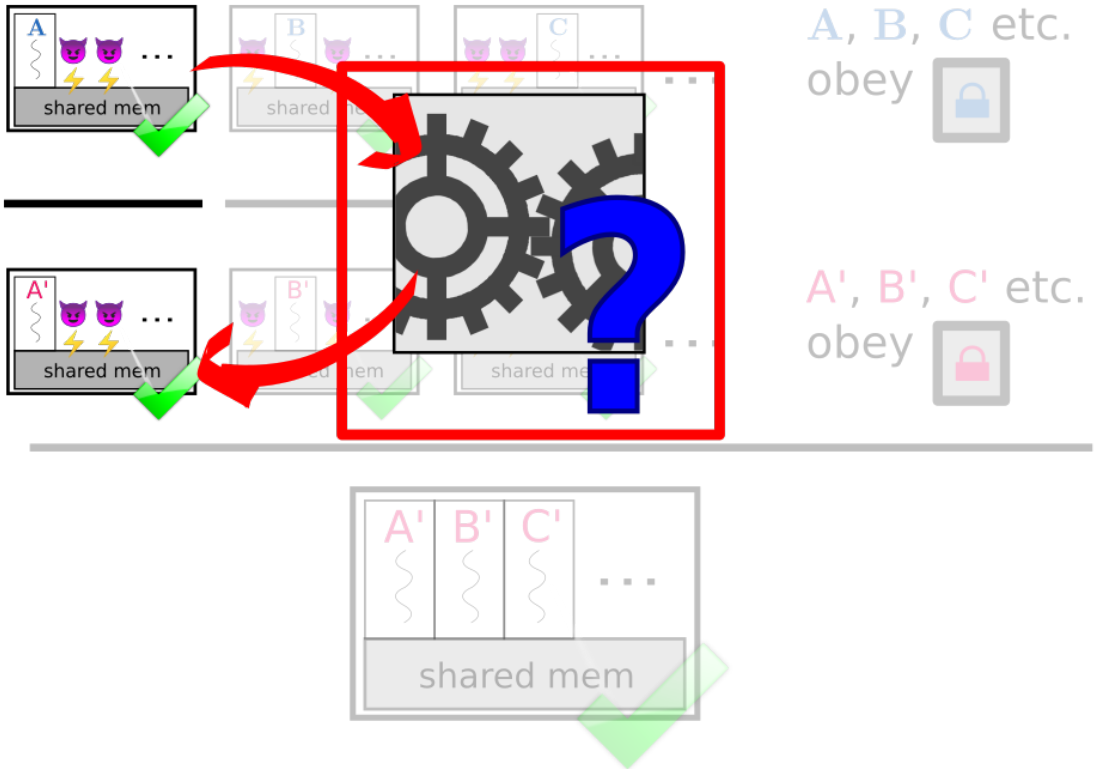
Part 2: Per-thread compositional refinement

Part 3: While-to-RISC compiler verification



This talk: a preview

Part 1: Concurrent value-dependent noninterference
Part 2: Per-thread compositional refinement
Part 3: While-to-RISC compiler verification

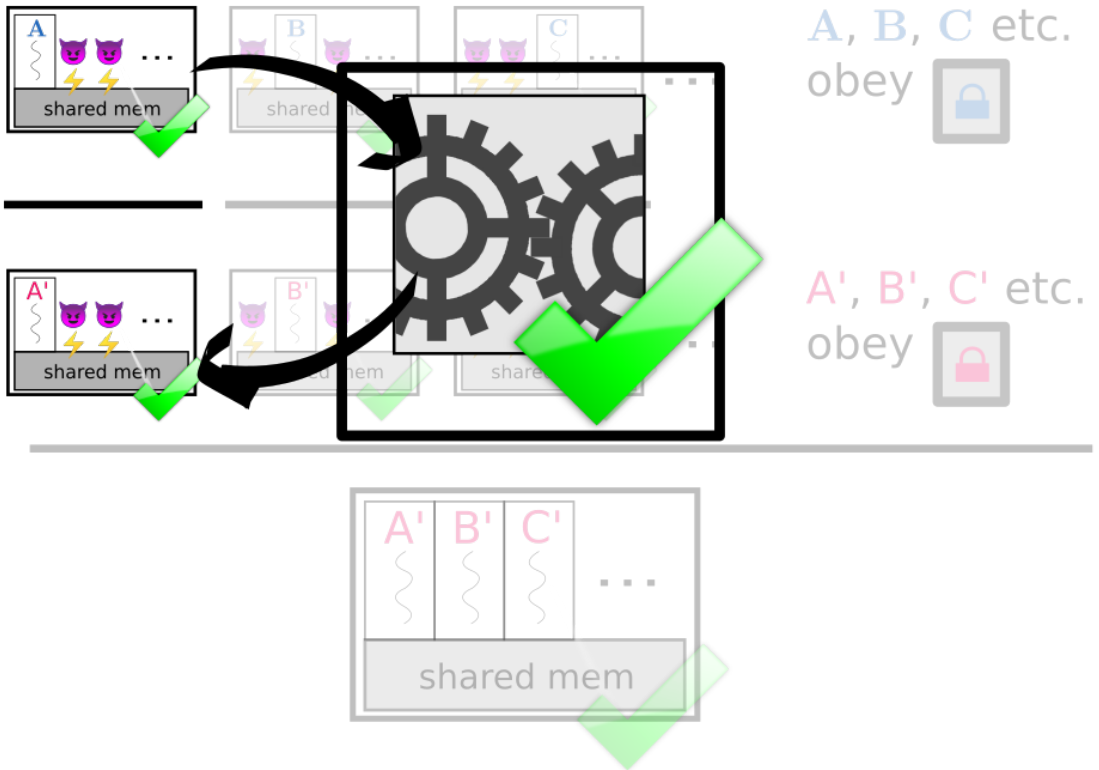


A, B, C etc.
obey

Source

This talk: a preview

Part 1: Concurrent value-dependent noninterference
Part 2: Per-thread compositional refinement
Part 3: While-to-RISC compiler verification

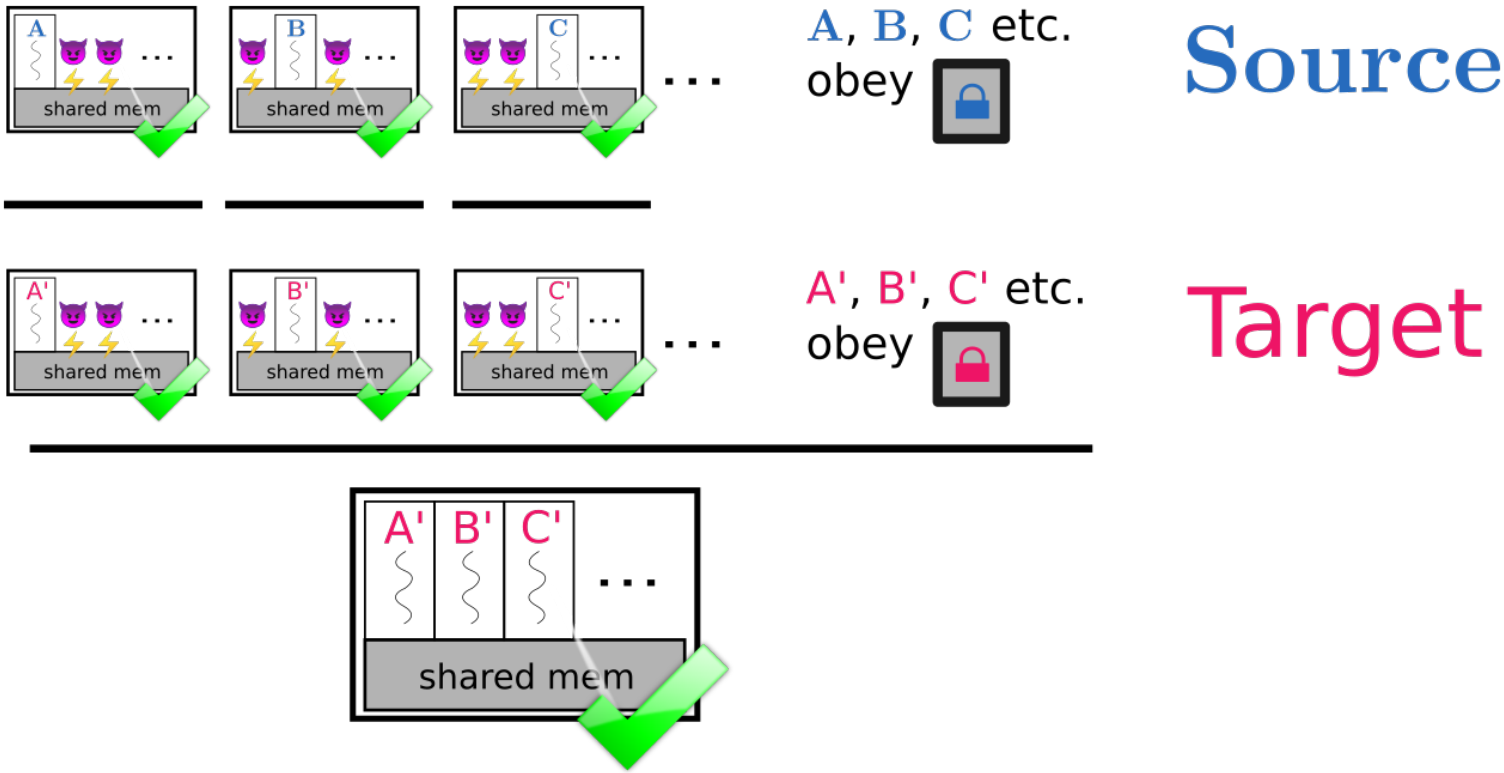


Source

Target

This talk

Part 1: Concurrent value-dependent noninterference
Part 2: Per-thread compositional refinement
Part 3: While-to-RISC compiler verification

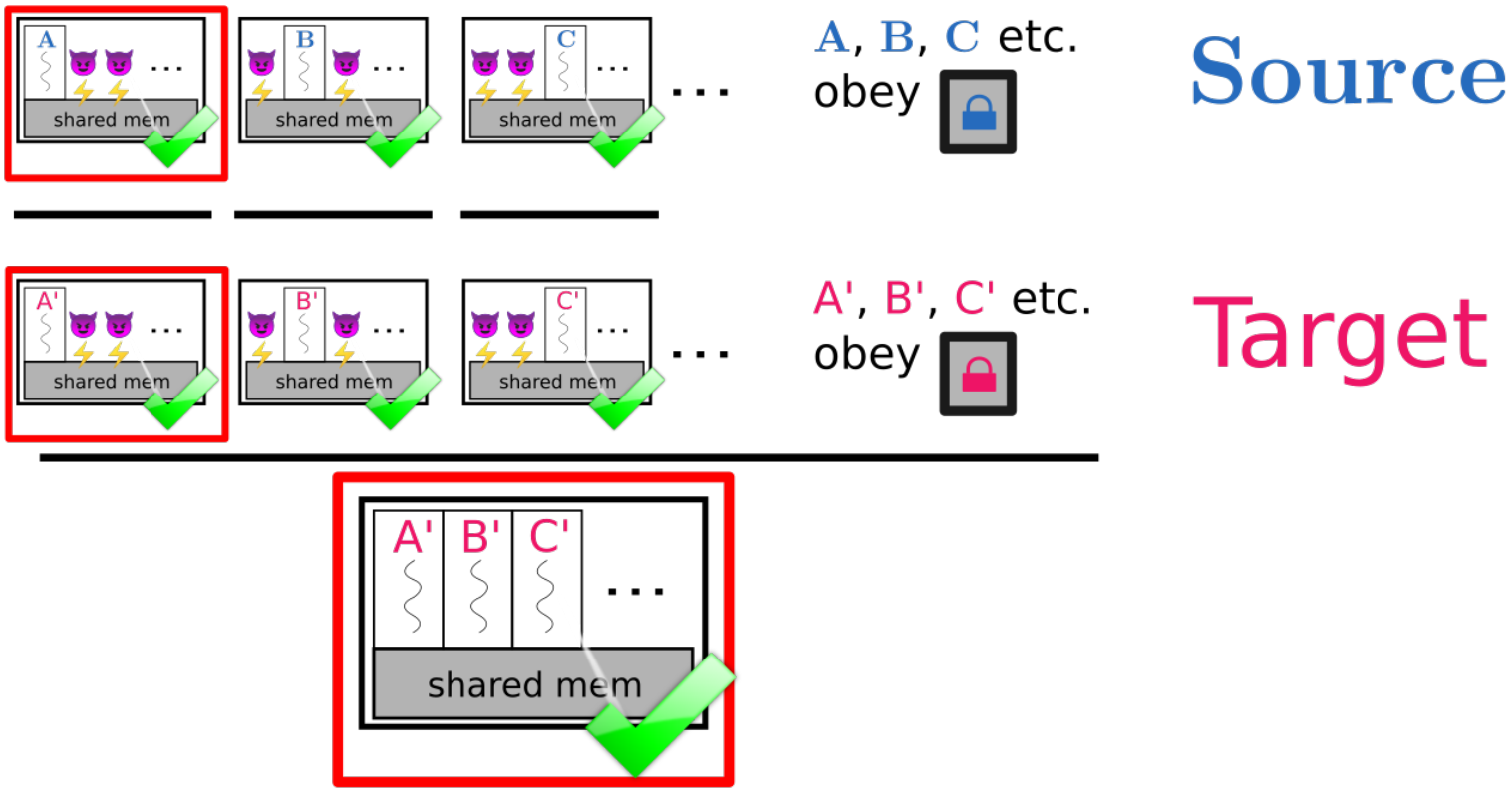


This talk

Part 1: Concurrent value-dependent noninterference

Part 2: Per-thread compositional refinement

Part 3: While-to-RISC compiler verification



The confidentiality property

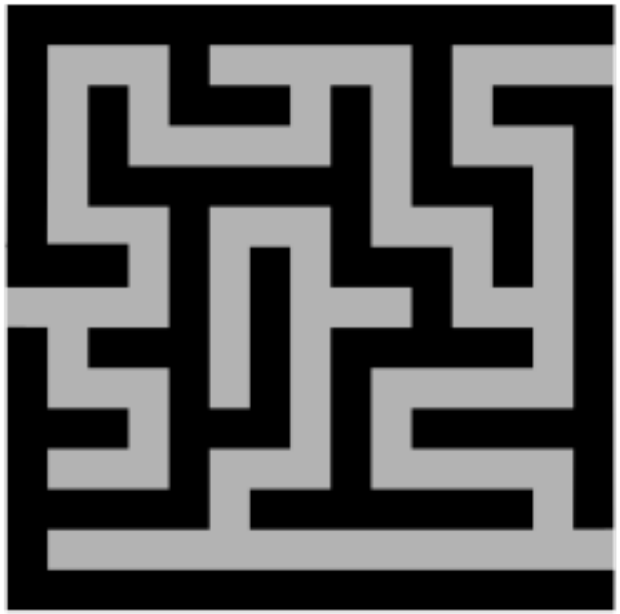
Concurrent value-dependent noninterference.

The confidentiality property

Concurrent value-dependent noninterference.

Simplest policy: **H**igh $\not\rightarrow$ **L**ow

Low part of state must remain indistinguishable.



The confidentiality property

Concurrent value-dependent noninterference.

Simplest policy: **H**igh $\not\rightarrow$ **L**ow

Low part of state must remain indistinguishable.

The confidentiality property

Concurrent value-dependent noninterference.

Simplest policy: **H**igh $\not\rightarrow$ **L**ow

Low part of state must remain indistinguishable.

Reflects the *attacker model*.

The confidentiality property

Concurrent value-dependent *noninterference*.

Simplest policy: **H**igh $\not\rightarrow$ **L**ow

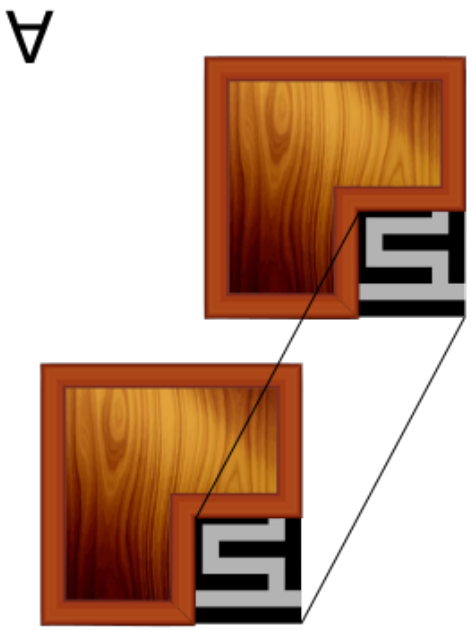
Low part of state must remain indistinguishable.

The confidentiality property

Concurrent value-dependent noninterference.

Simplest policy: **H**igh $\not\rightarrow$ **L**ow

Low part of state must remain indistinguishable.

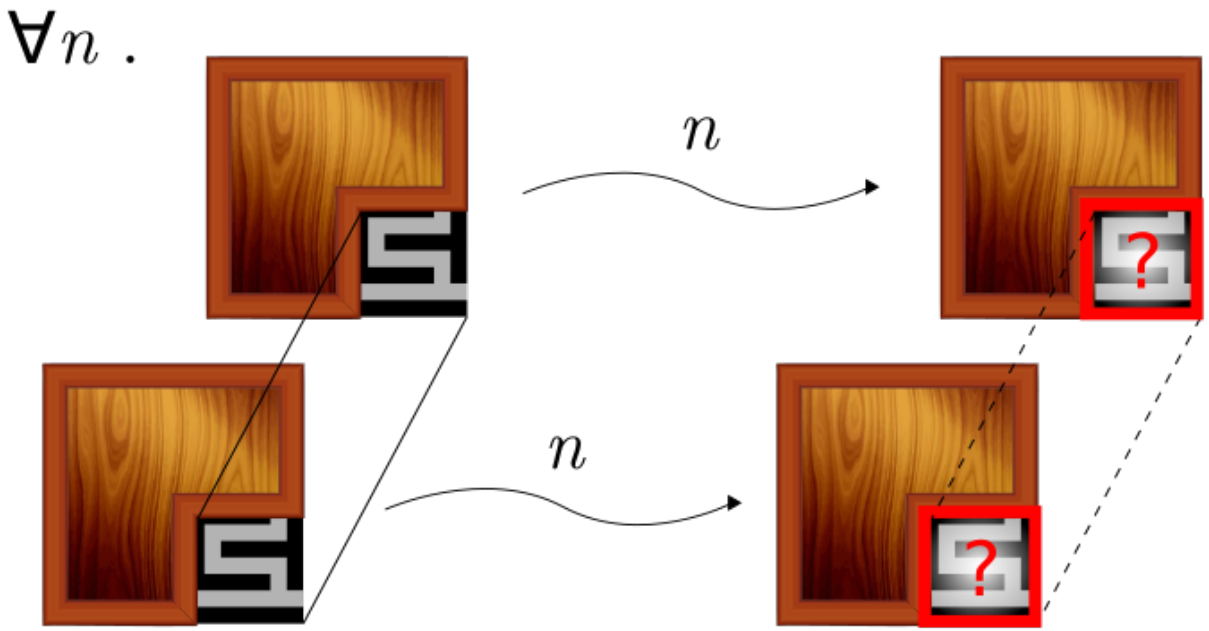


The confidentiality property

Concurrent value-dependent noninterference.

Simplest policy: **H**igh $\not\rightarrow$ **L**ow

Low part of state must remain indistinguishable.



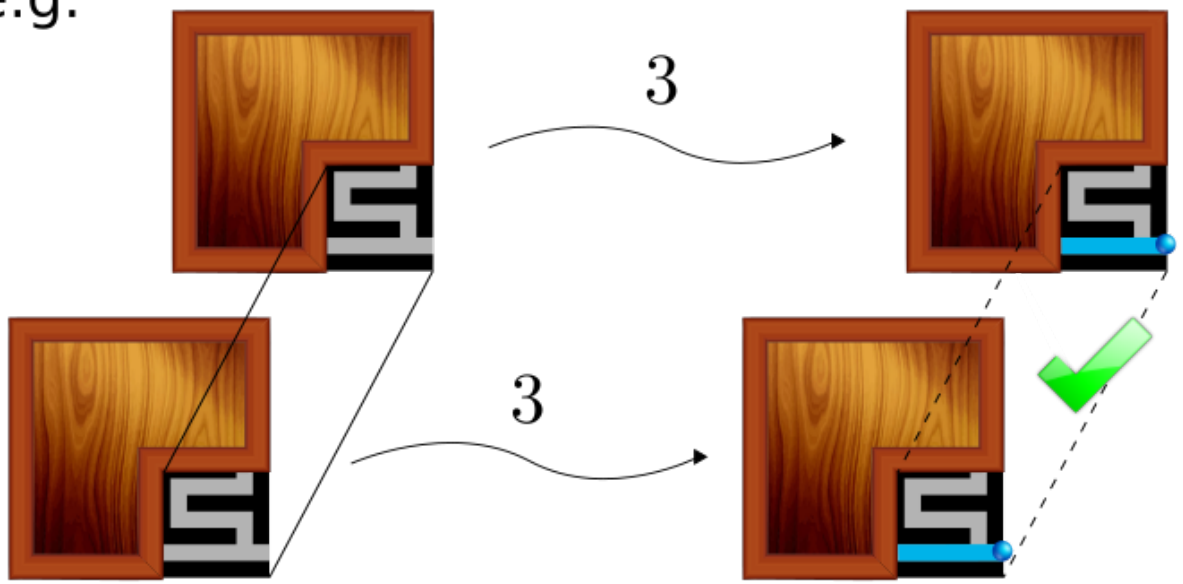
The confidentiality property

Concurrent value-dependent noninterference.

Simplest policy: **H**igh $\not\rightarrow$ **L**ow

Low part of state must remain indistinguishable.

e.g.



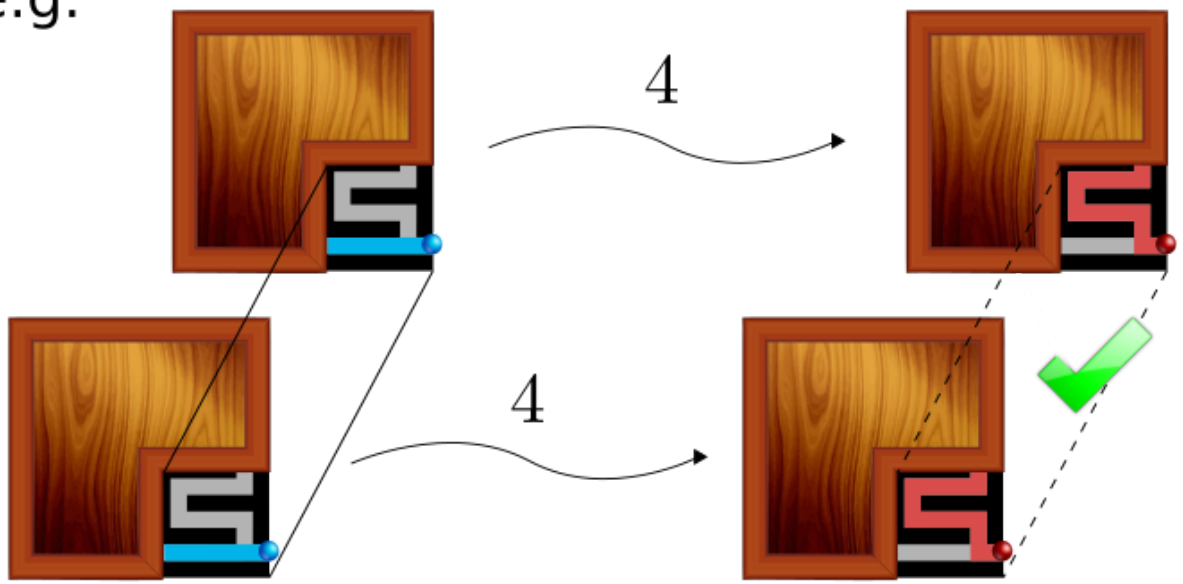
The confidentiality property

Concurrent value-dependent ***noninterference****.*

Simplest policy: **H**igh $\not\rightarrow$ **L**ow

Low part of state must remain indistinguishable.

e.g.



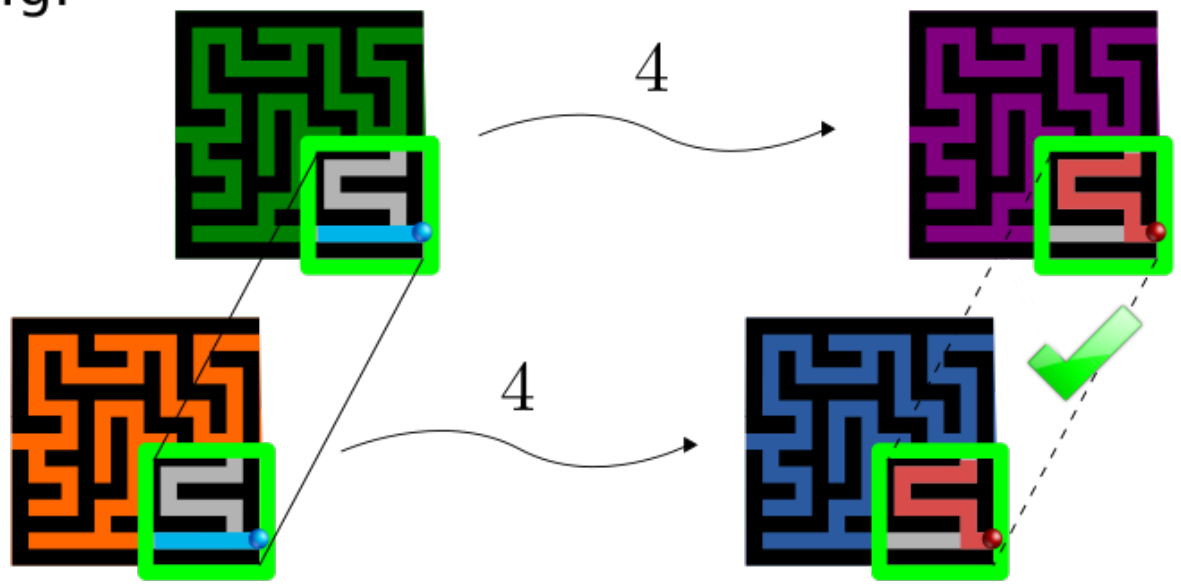
The confidentiality property

Concurrent value-dependent noninterference.

Simplest policy: **H**igh $\not\rightarrow$ **L**ow

Low part of state must remain indistinguishable.

e.g.



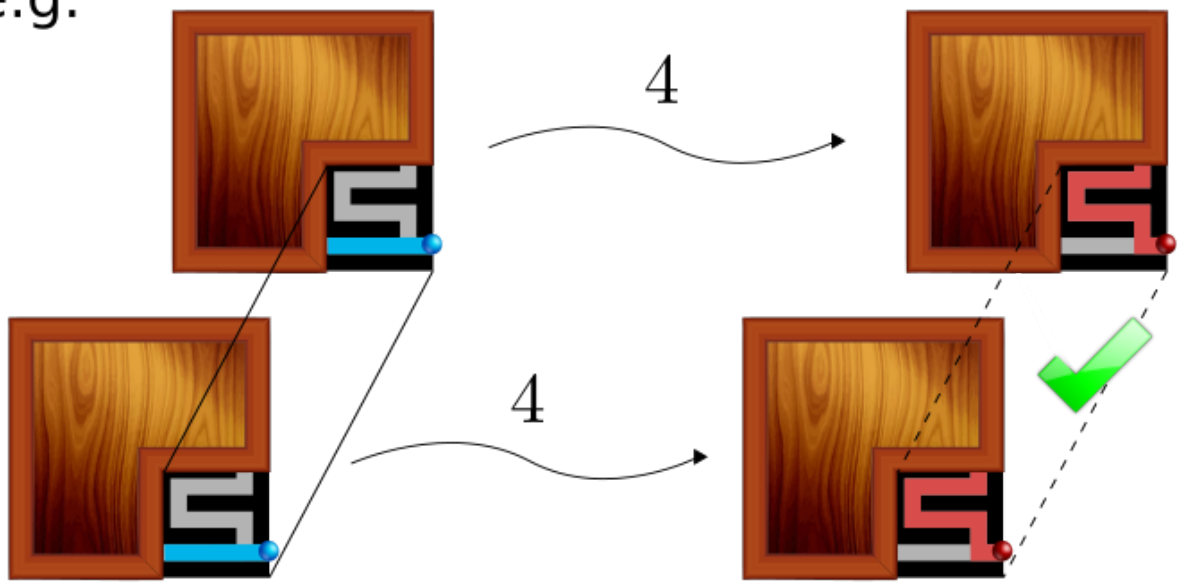
The confidentiality property

Concurrent value-dependent ***noninterference****.*

Simplest policy: **H**igh $\not\rightarrow$ **L**ow

Low part of state must remain indistinguishable.

e.g.



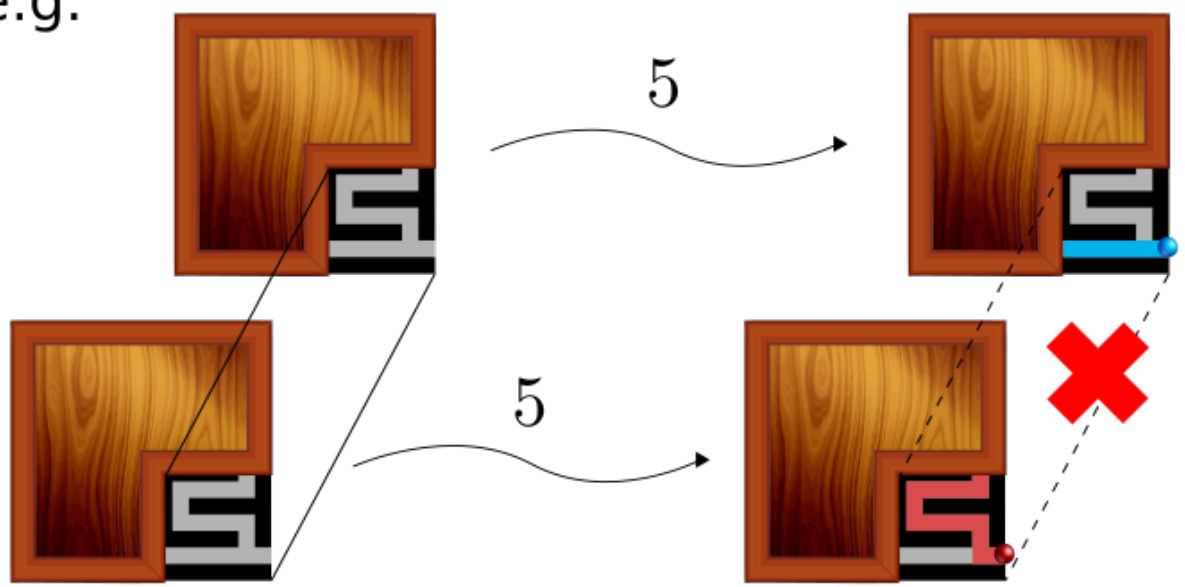
The confidentiality property

Concurrent value-dependent noninterference.

Simplest policy: **H**igh $\not\rightarrow$ **L**ow

Low part of state must remain indistinguishable.

e.g.



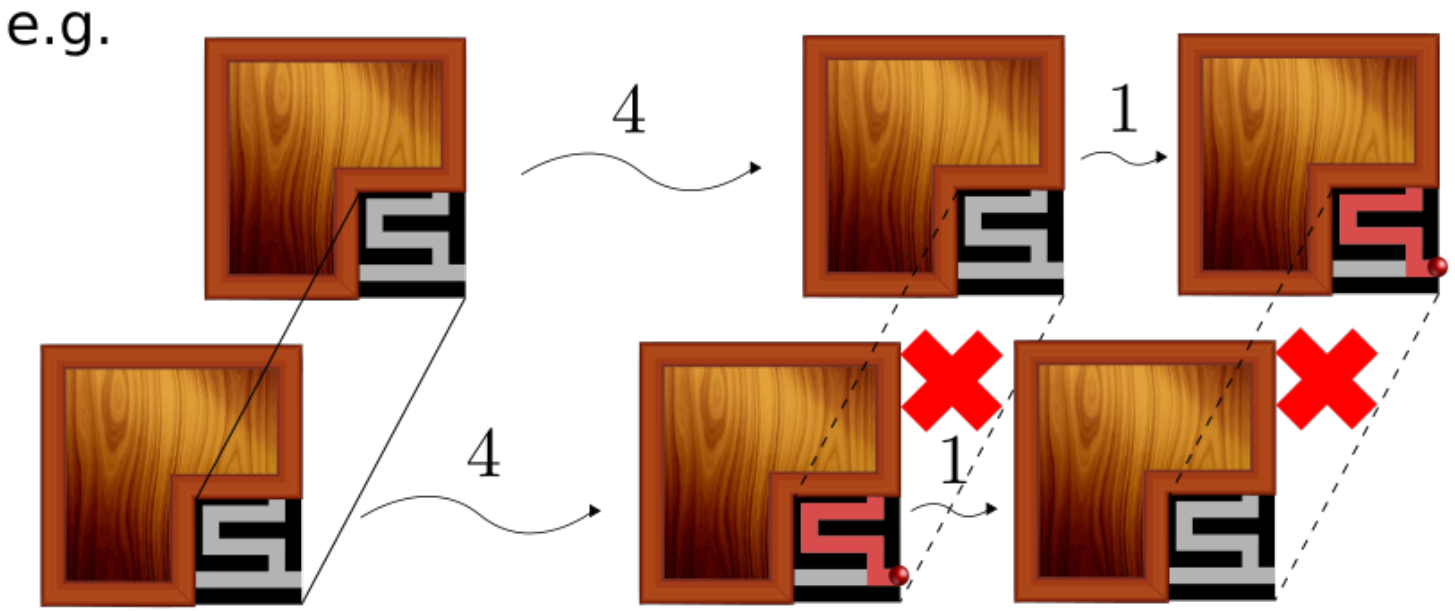
- A 2-safety hyperproperty.

The confidentiality property

Concurrent value-dependent noninterference.

Simplest policy: **H**igh $\not\to$ **L**ow

Low part of state must remain indistinguishable.



- A 2-safety hyperproperty.
- Timing-sensitive. (Want this for concurrency reasons.)

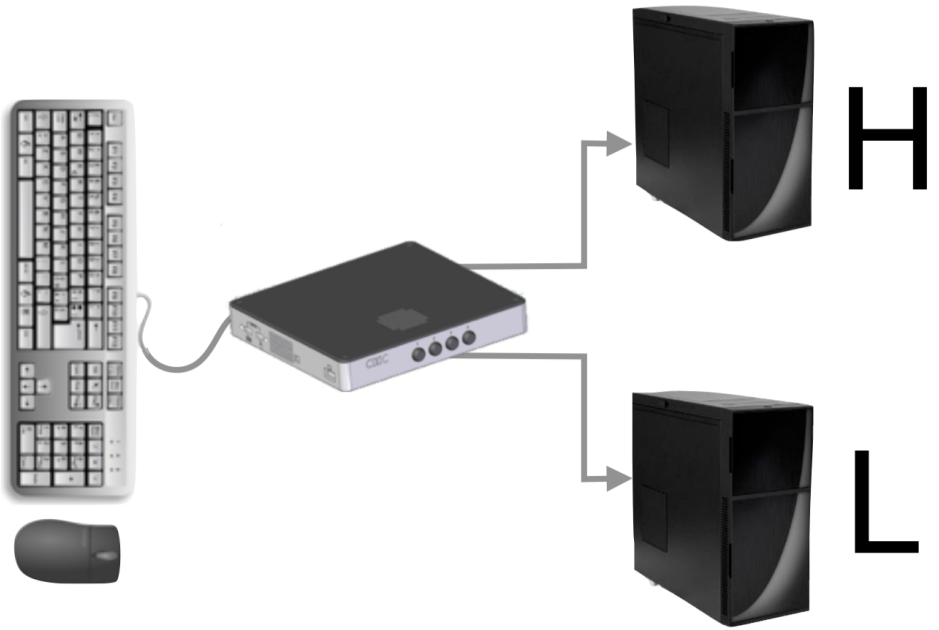
The confidentiality property

Concurrent value-dependent noninterference.

Simplest policy: **H**igh $\not\to$ **L**ow

Low part of state must remain indistinguishable.

Classification of state as H or L can vary over time.



The confidentiality property

Concurrent value-dependent noninterference.

Simplest policy: **H**igh $\not\to$ **L**ow

Low part of state must remain indistinguishable.

Classification of state as H or L can vary over time.

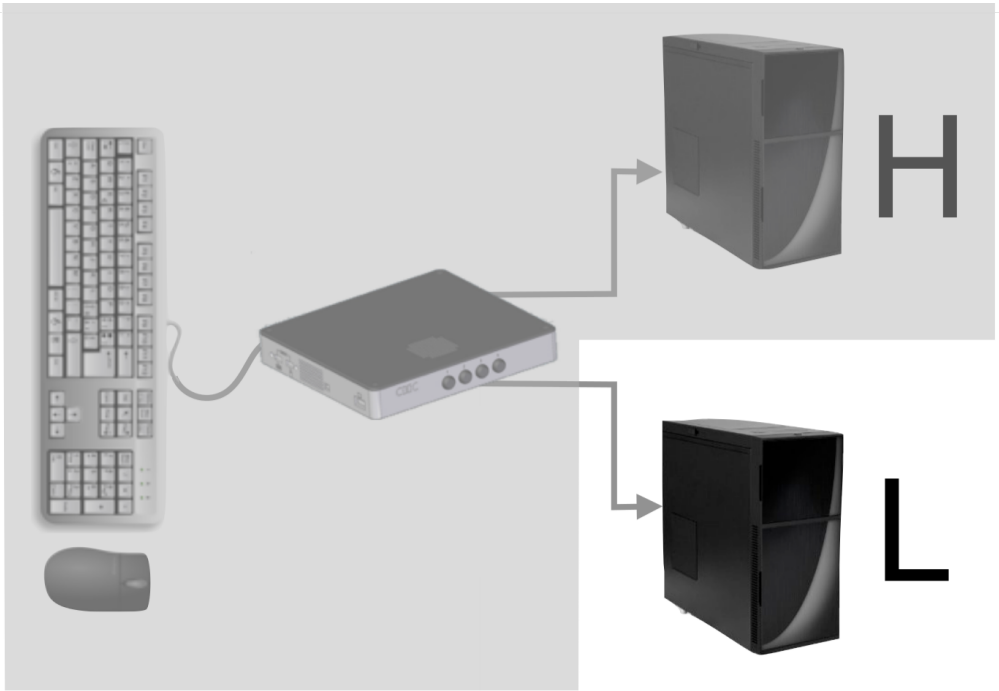
The confidentiality property

Concurrent value-dependent noninterference.

Simplest policy: **H**igh $\not\to$ **L**ow

Low part of state must remain indistinguishable.

Classification of state as H or L can vary over time.



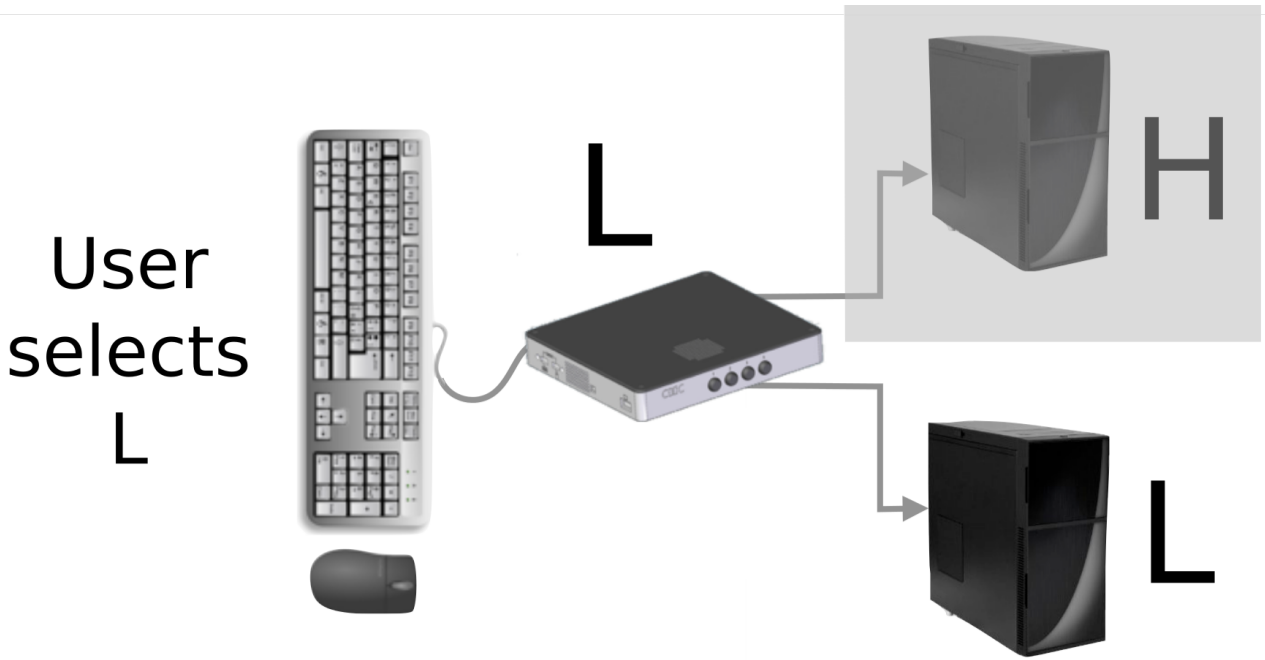
The confidentiality property

Concurrent value-dependent noninterference.

Simplest policy: **H**igh $\not\to$ **L**ow

Low part of state must remain indistinguishable.

Classification of state as H or L can vary over time.



The confidentiality property

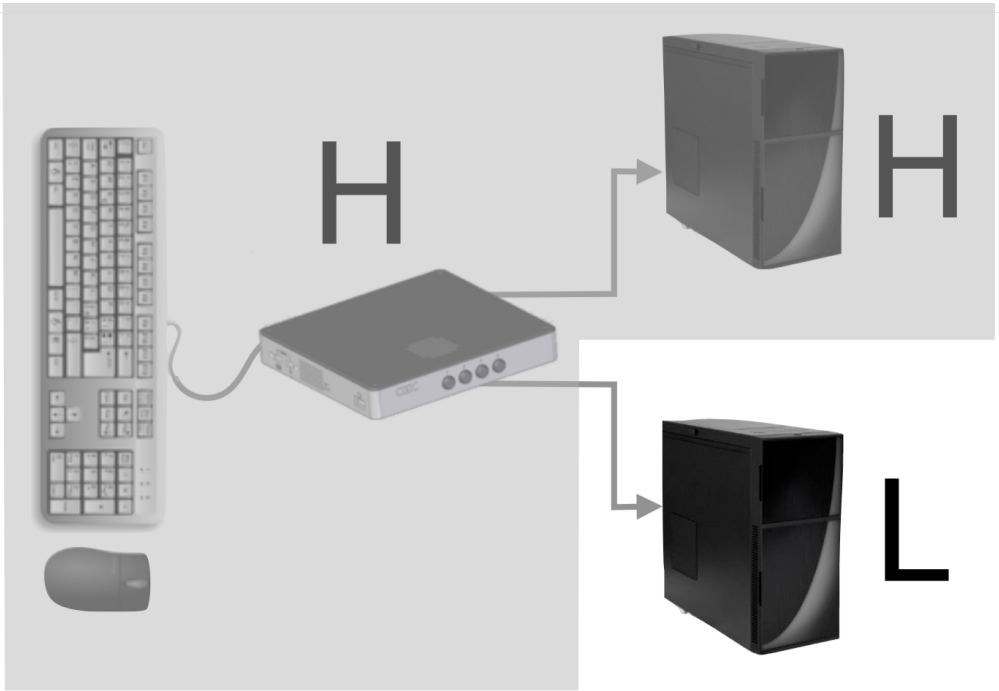
Concurrent value-dependent noninterference.

Simplest policy: **H**igh $\not\to$ **L**ow

Low part of state must remain indistinguishable.

Classification of state as H or L can vary over time.

User selects H



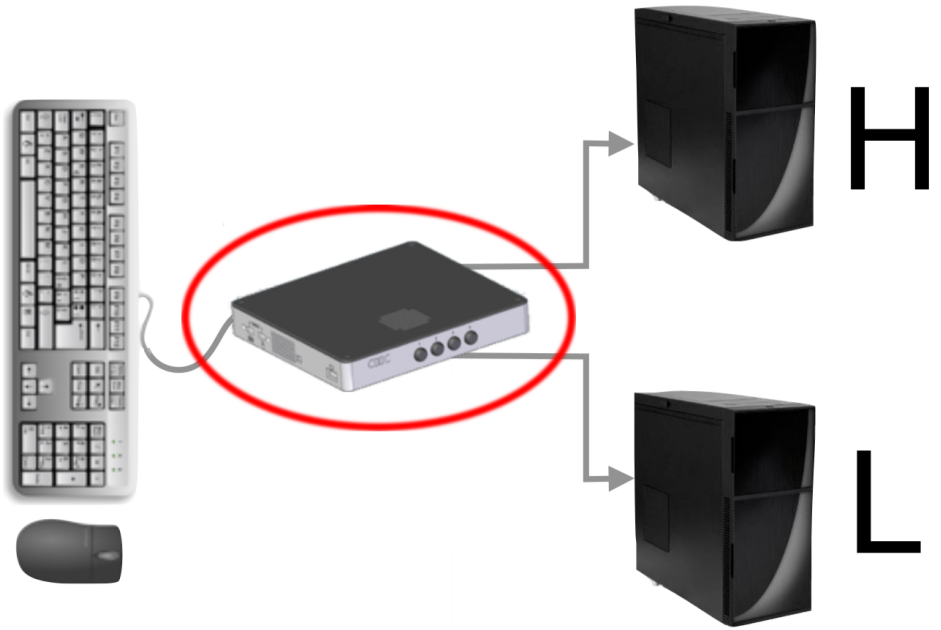
The confidentiality property

Concurrent value-dependent noninterference.

Simplest policy: **H**igh $\not\rightarrow$ **L**ow

Low part of state must remain indistinguishable.

Classification of state as H or L can vary over time.



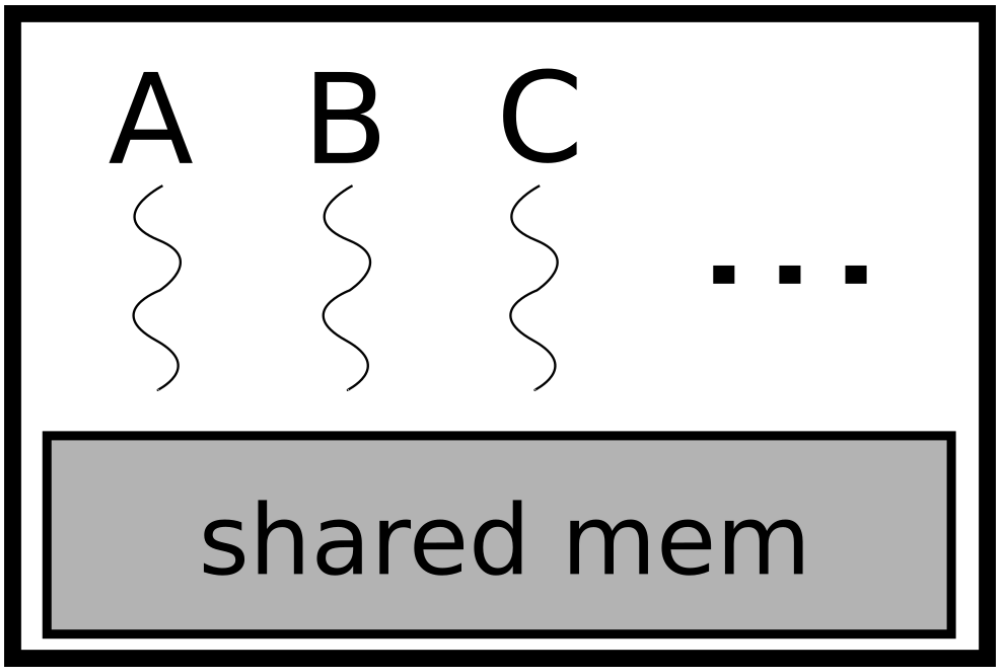
The confidentiality property

Concurrent value-dependent noninterference.

Simplest policy: **H**igh $\not\to$ **L**ow

Low part of state must remain indistinguishable.

Classification of state as H or L can vary over time.



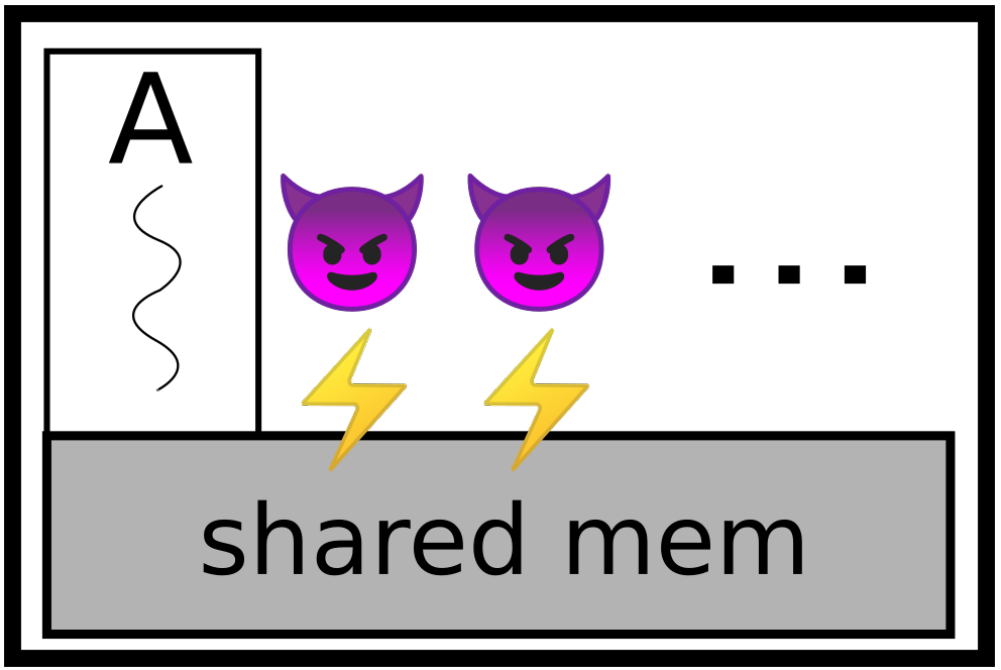
The confidentiality property

Concurrent value-dependent noninterference.

Simplest policy: **H**igh $\not\to$ **L**ow

Low part of state must remain indistinguishable.

Classification of state as H or L can vary over time.



Per-thread, subject to havoc.

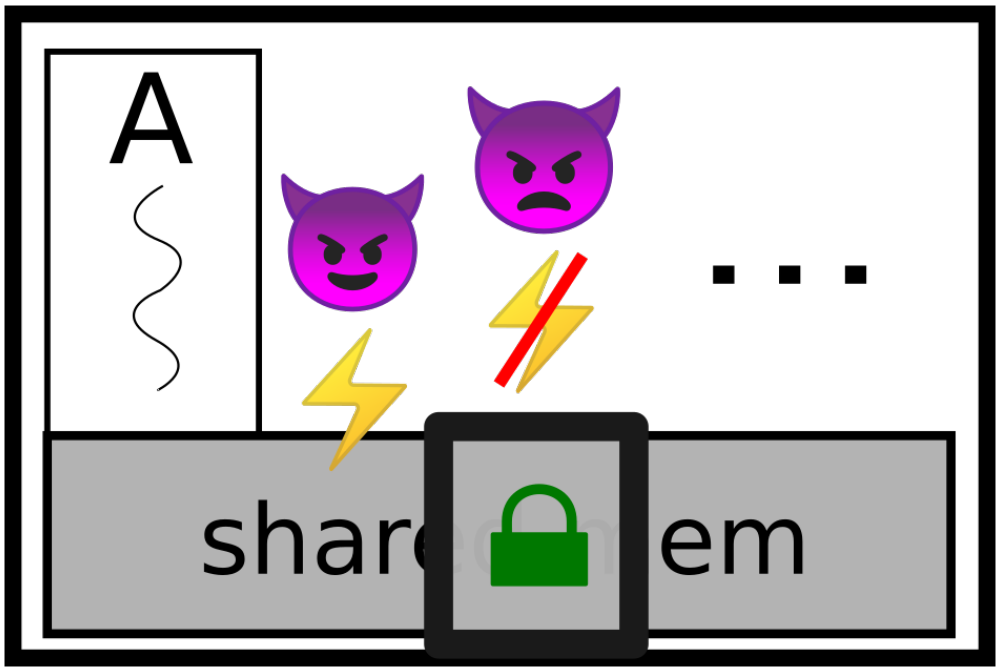
The confidentiality property

Concurrent value-dependent noninterference.

Simplest policy: **H**igh $\not\to$ **L**ow

Low, unlocked part of state must remain indistinguishable.

Classification of state as H or L can vary over time.



Per-thread, subject to havoc that obeys locking discipline.

The confidentiality property

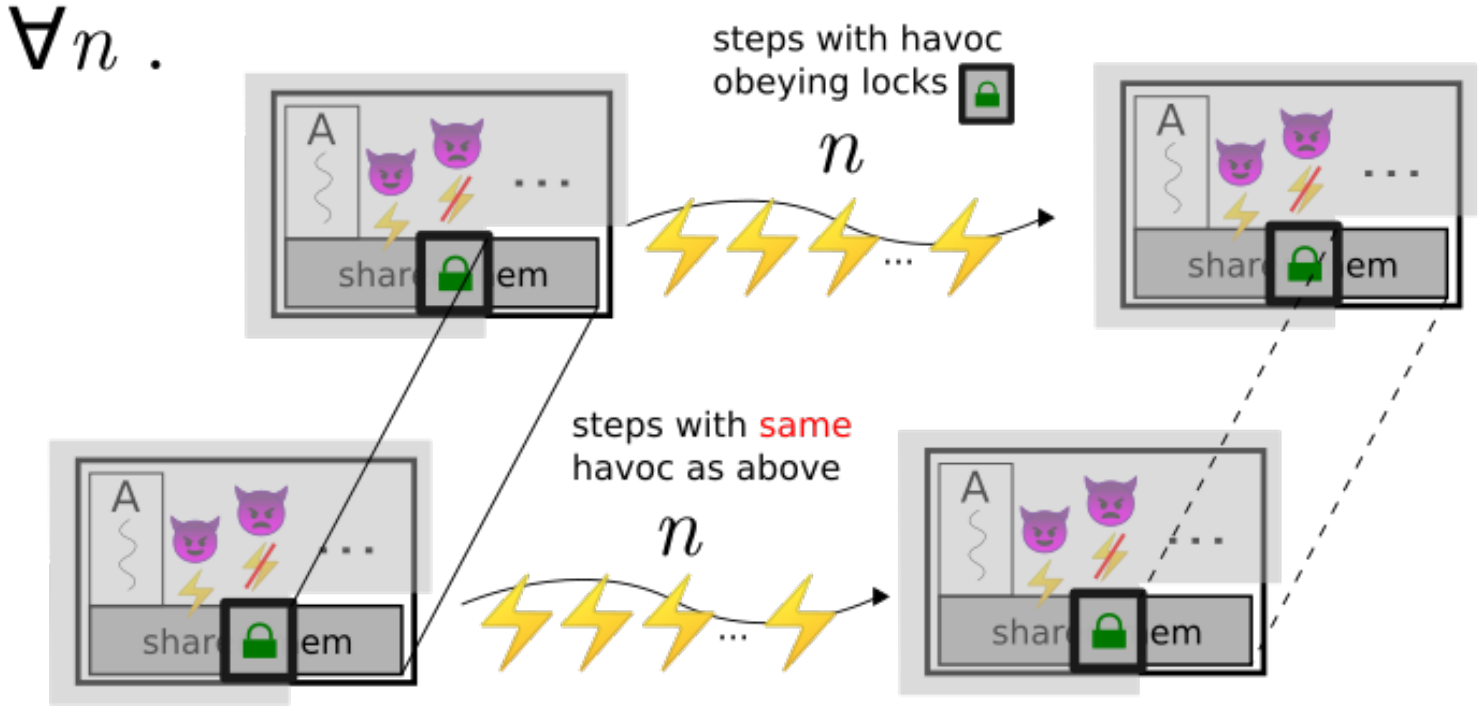
Concurrent value-dependent noninterference.

Simplest policy: **H**igh $\not\to$ **L**ow

Low, unlocked part of state must remain indistinguishable.

Classification of state as H or L can vary over time.

Per-thread compositional property:



The confidentiality property

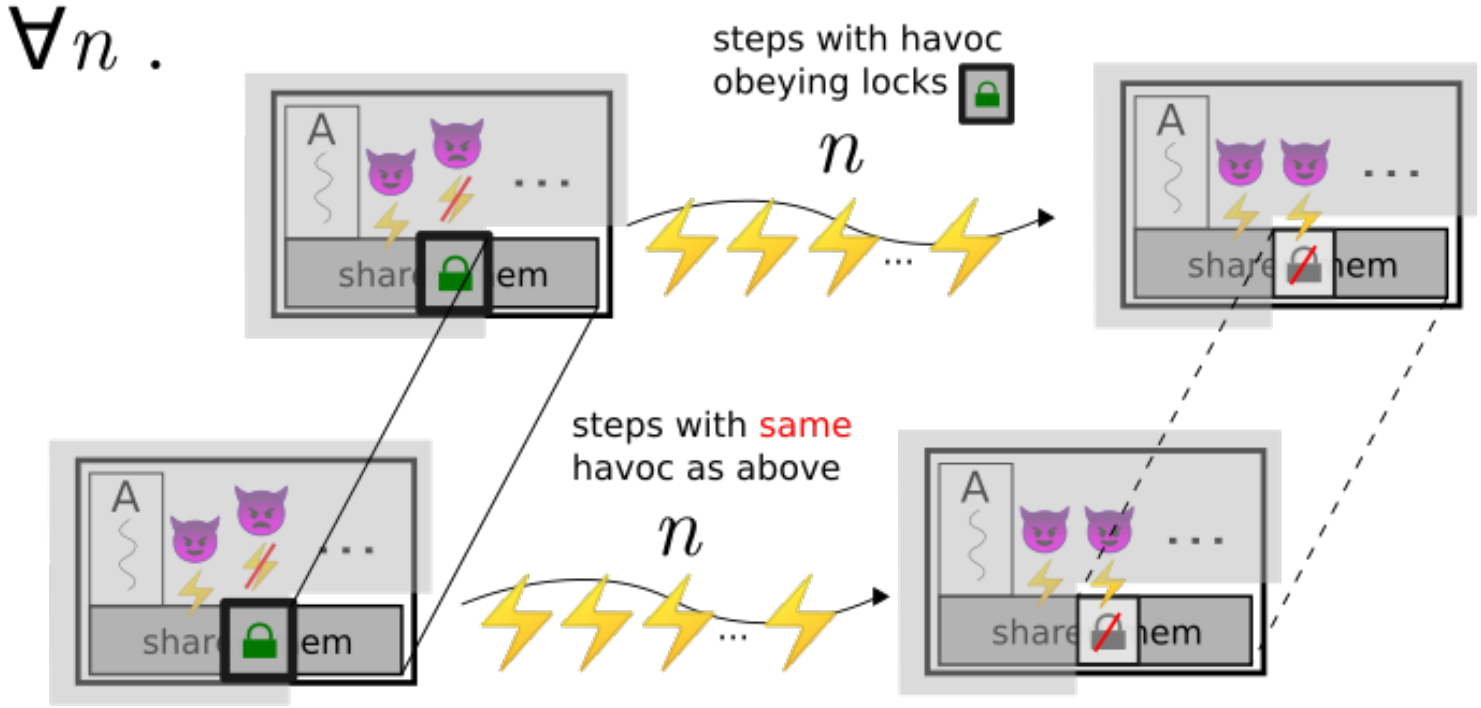
Concurrent value-dependent noninterference.

Simplest policy: **H**igh $\not\to$ **L**ow

Low, unlocked part of state must remain indistinguishable.

Classification of state as H or L can vary over time.

Per-thread compositional property:



The confidentiality property

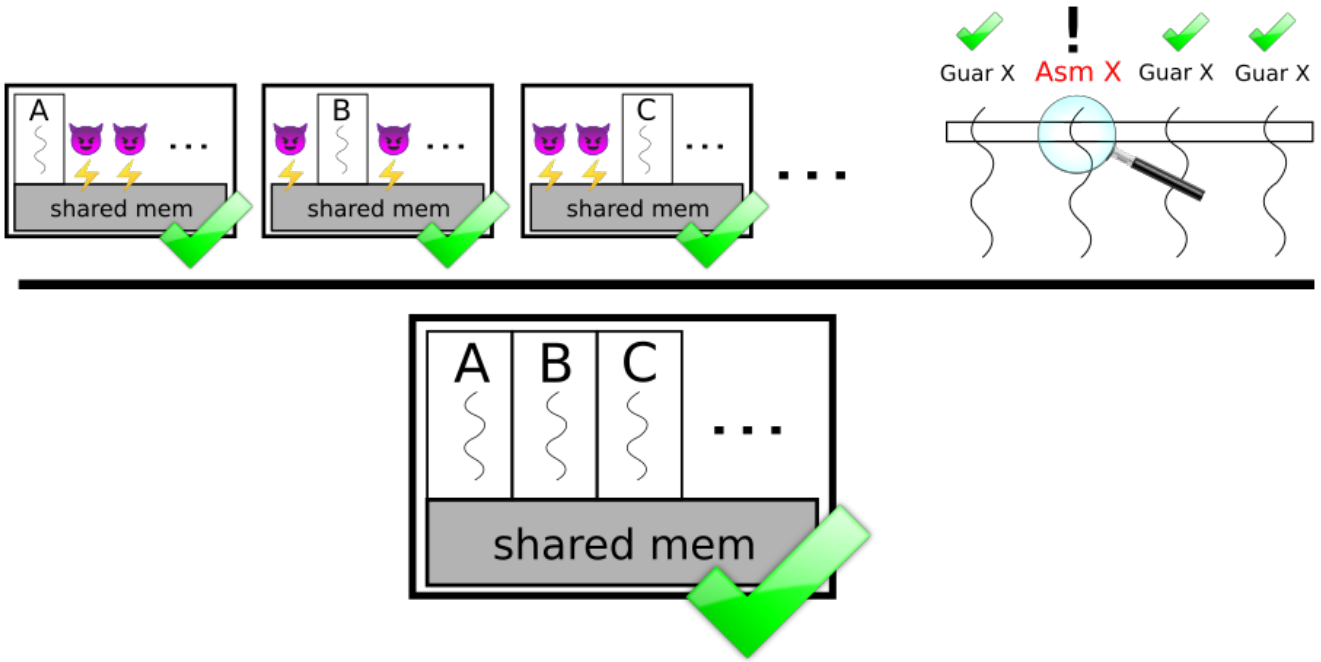
Concurrent value-dependent noninterference.

Simplest policy: **H**igh $\not\to$ **L**ow

Low, unlocked part of state must remain indistinguishable.

Classification of state as H or L can vary over time.

Per-thread compositionality theorem [Murray+, CSF'16]:



Under the hood: *assume-guarantee* on variable access.

The confidentiality property

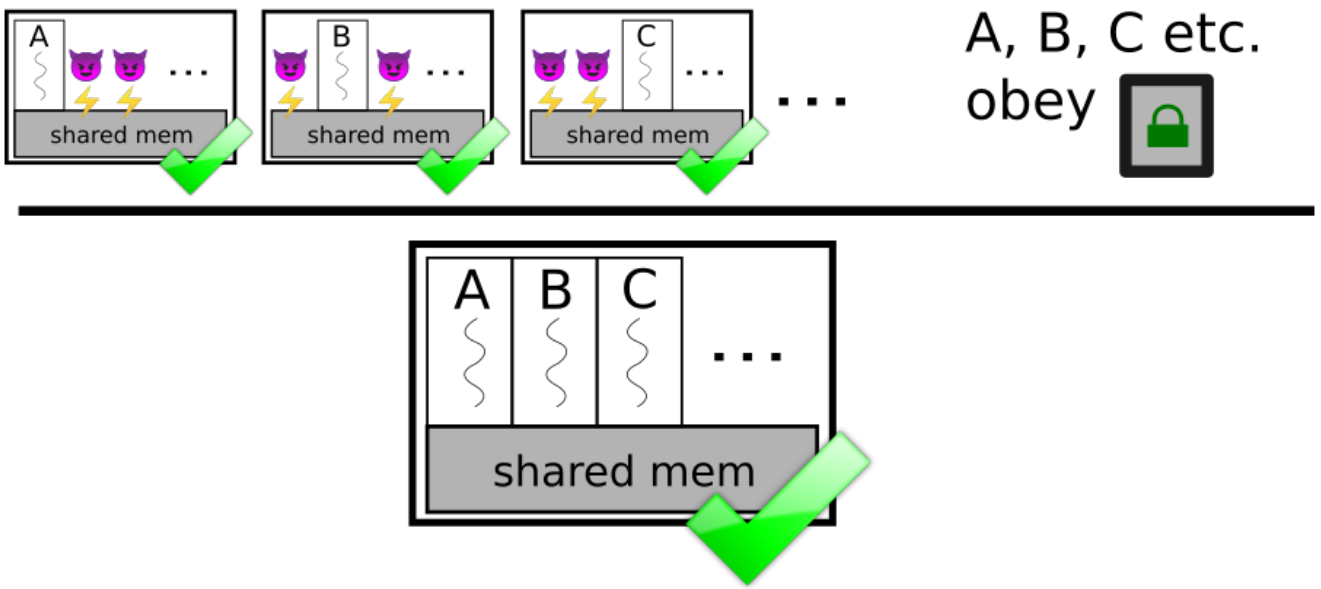
Concurrent value-dependent noninterference.

Simplest policy: **H**igh $\not\to$ **L**ow

Low, unlocked part of state must remain indistinguishable.

Classification of state as H or L can vary over time.

Per-thread compositionality theorem:



[Murray+, CSF'16] instantiated with locking primitives.

The confidentiality property

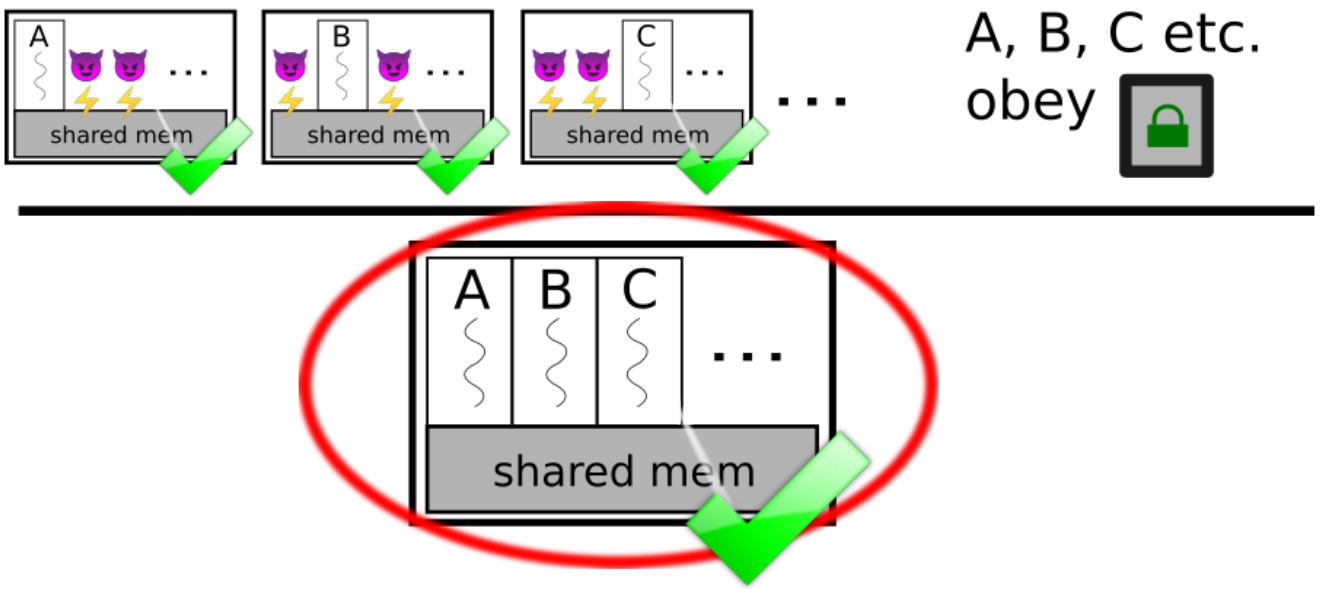
Concurrent value-dependent noninterference.

Simplest policy: **H**igh $\not\to$ **L**ow

Low, unlocked part of state must remain indistinguishable.

Classification of state as H or L can vary over time.

Whole-system property:



[Murray+, CSF'16] instantiated with locking primitives.

The confidentiality property

Concurrent value-dependent noninterference.

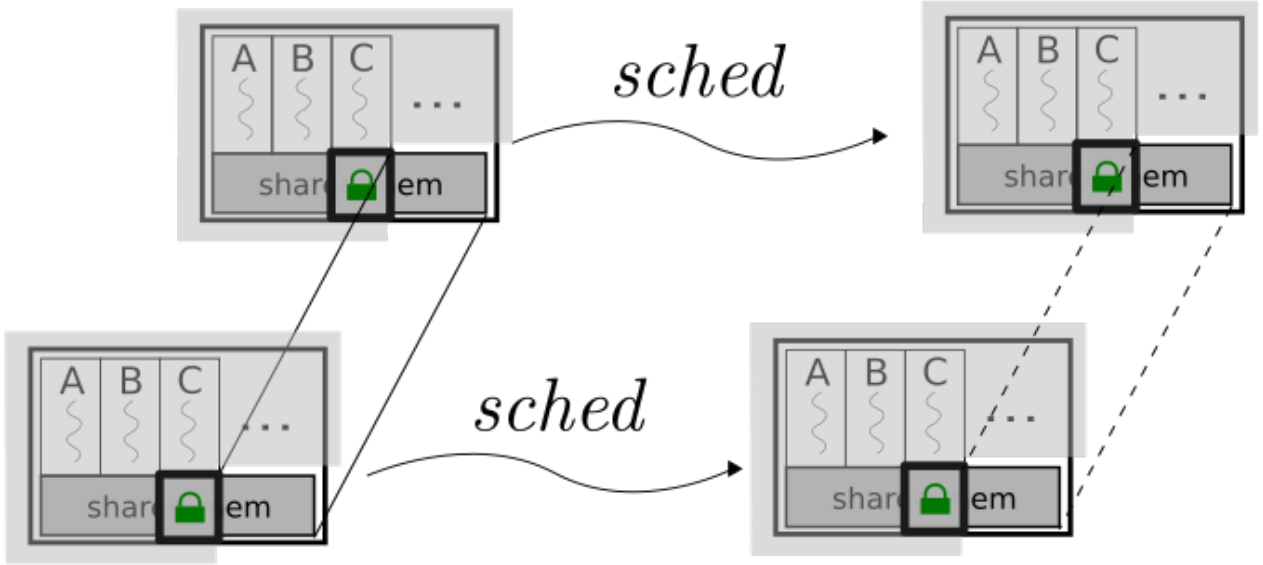
Simplest policy: **H**igh $\not\to$ **L**ow

Low, unlocked part of state must remain indistinguishable.

Classification of state as H or L can vary over time.

Whole-system property:

$\forall sched\ .$



The confidentiality property

Concurrent value-dependent noninterference.

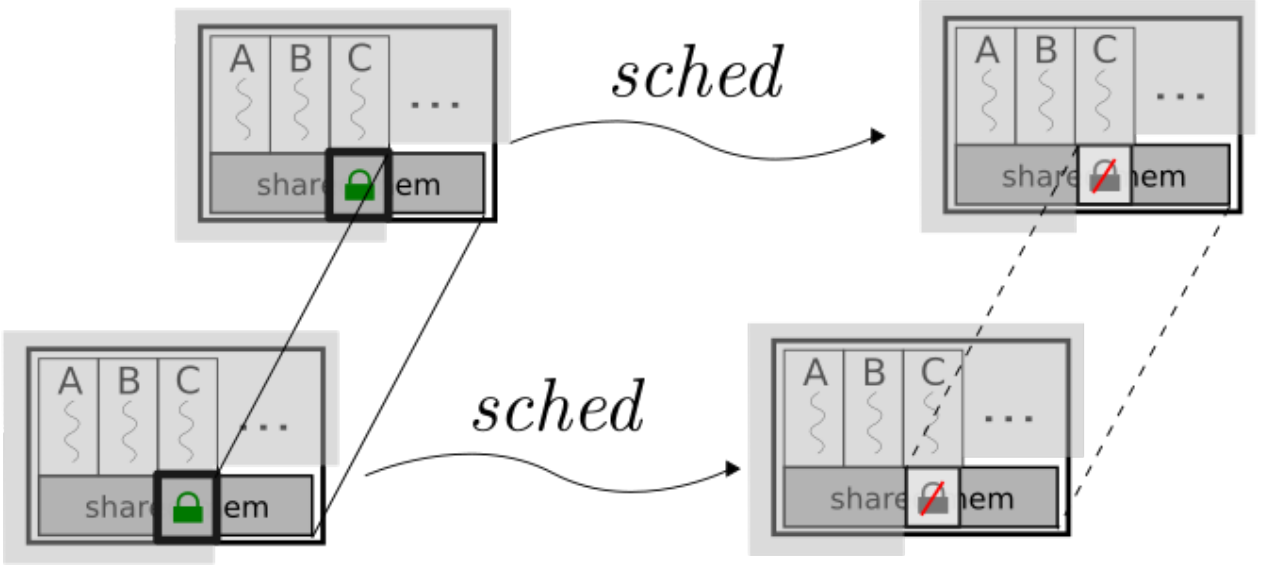
Simplest policy: **H**igh $\not\to$ **L**ow

Low, unlocked part of state must remain indistinguishable.

Classification of state as H or L can vary over time.

Whole-system property:

$$\forall sched\ .$$



The confidentiality property

Concurrent value-dependent noninterference.

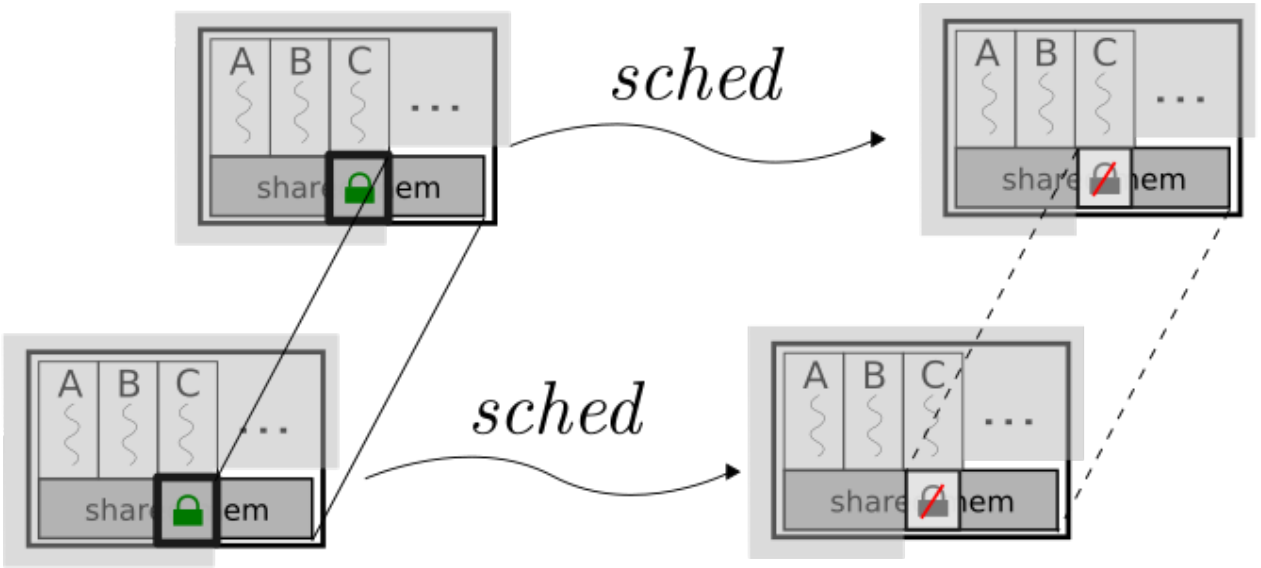
Simplest policy: **H**igh $\not\to$ **L**ow

Low, unlocked part of state must remain indistinguishable.

Classification of state as H or L can vary over time.

Whole-system property:

$\forall sched\ .$



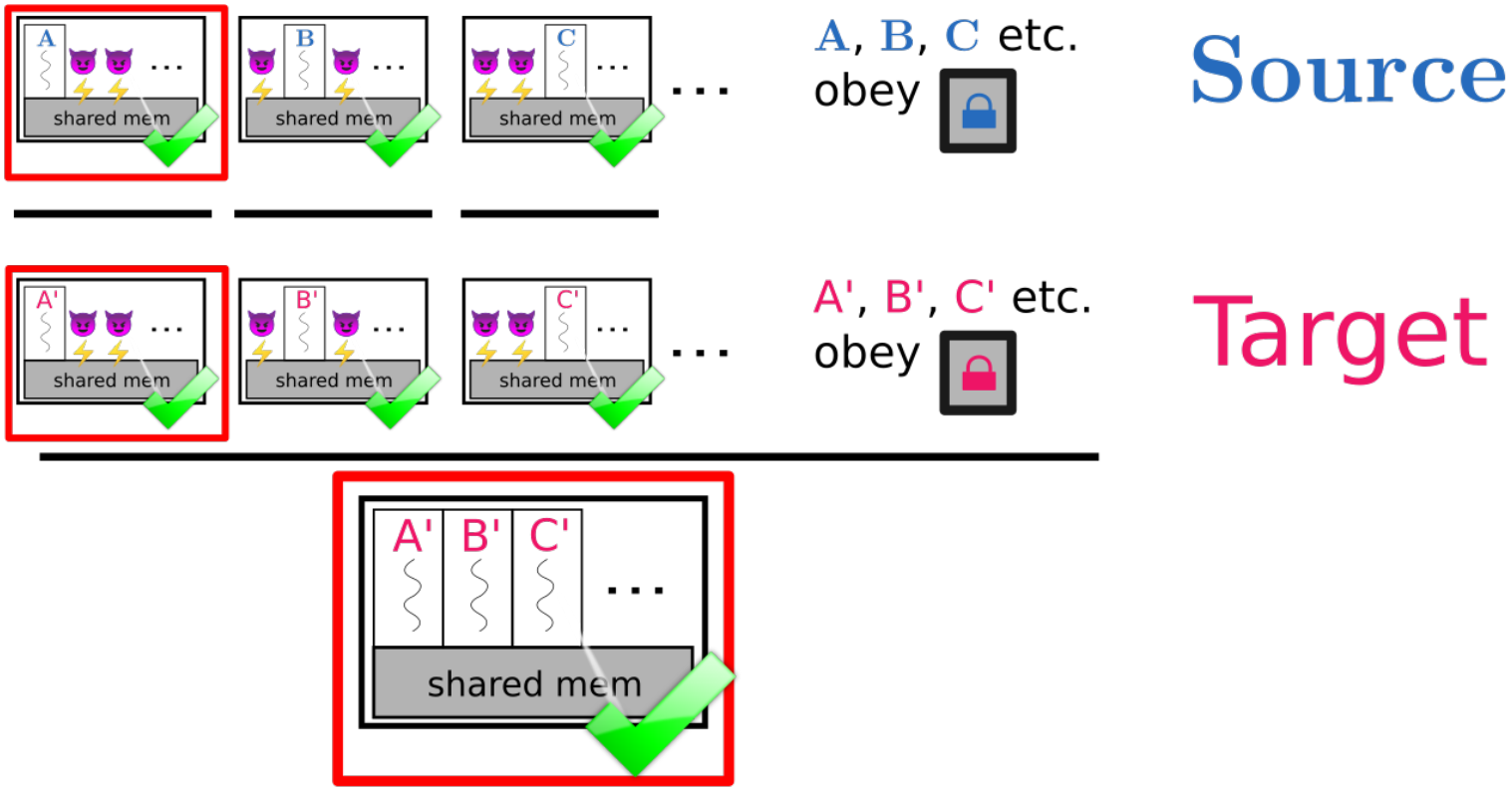
i.e. Locked state still not considered to be observable.

This talk

Part 1: Concurrent value-dependent noninterference

Part 2: Per-thread compositional refinement

Part 3: While-to-RISC compiler verification

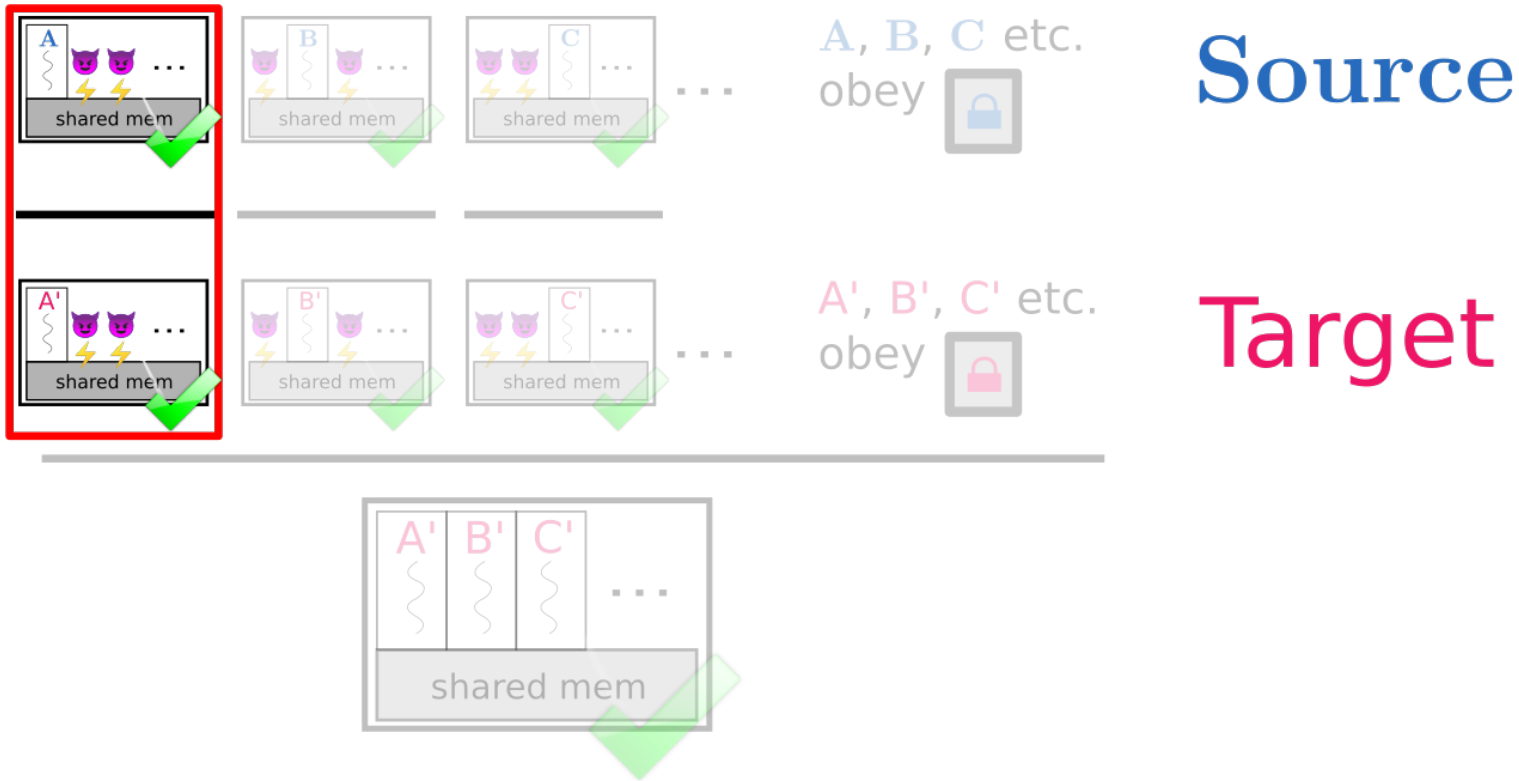


This talk

Part 1: Concurrent value-dependent noninterference

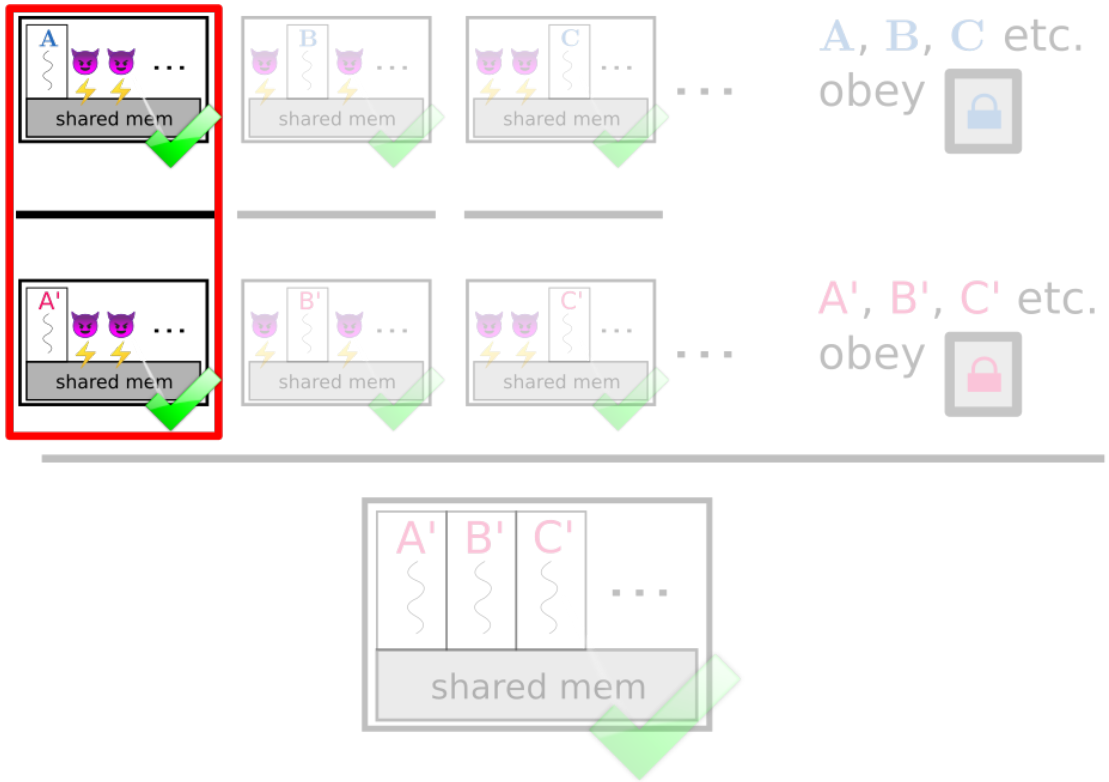
Part 2: Per-thread compositional refinement

Part 3: While-to-RISC compiler verification



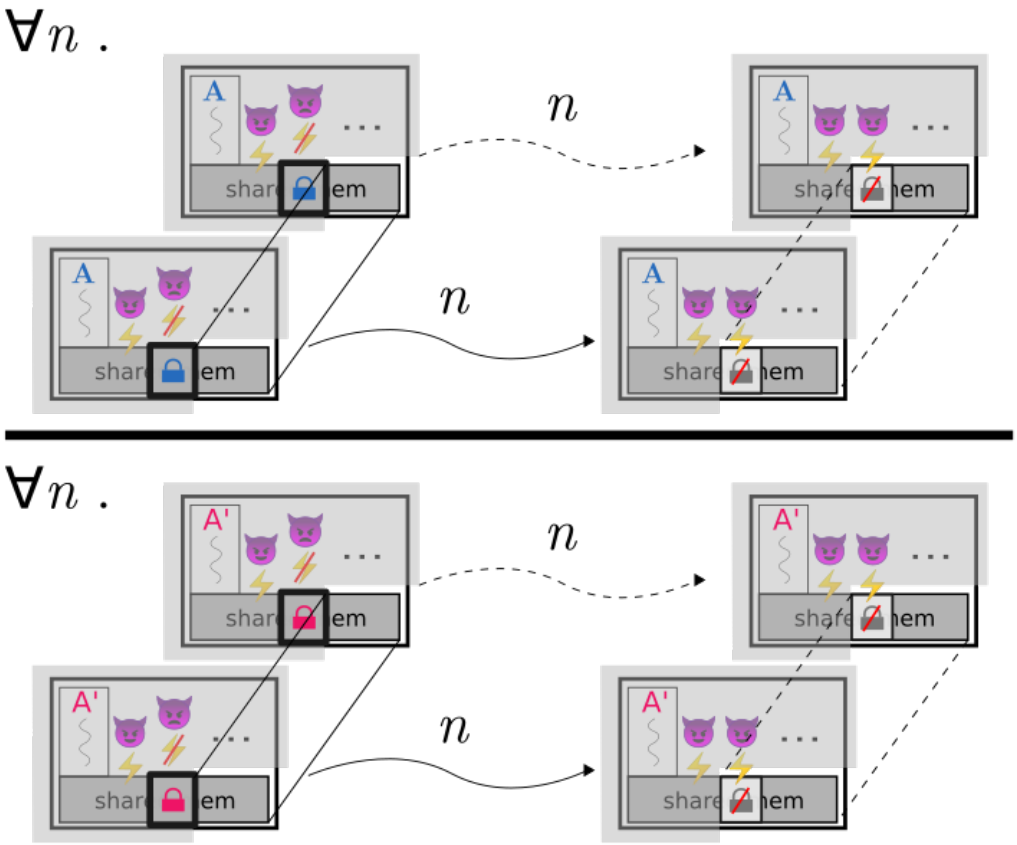
Proof technique for compilation

Per-thread *compositional refinement* [Murray+, CSF'16]



Proof technique for compilation

Per-thread *compositional refinement* [Murray+, CSF'16]

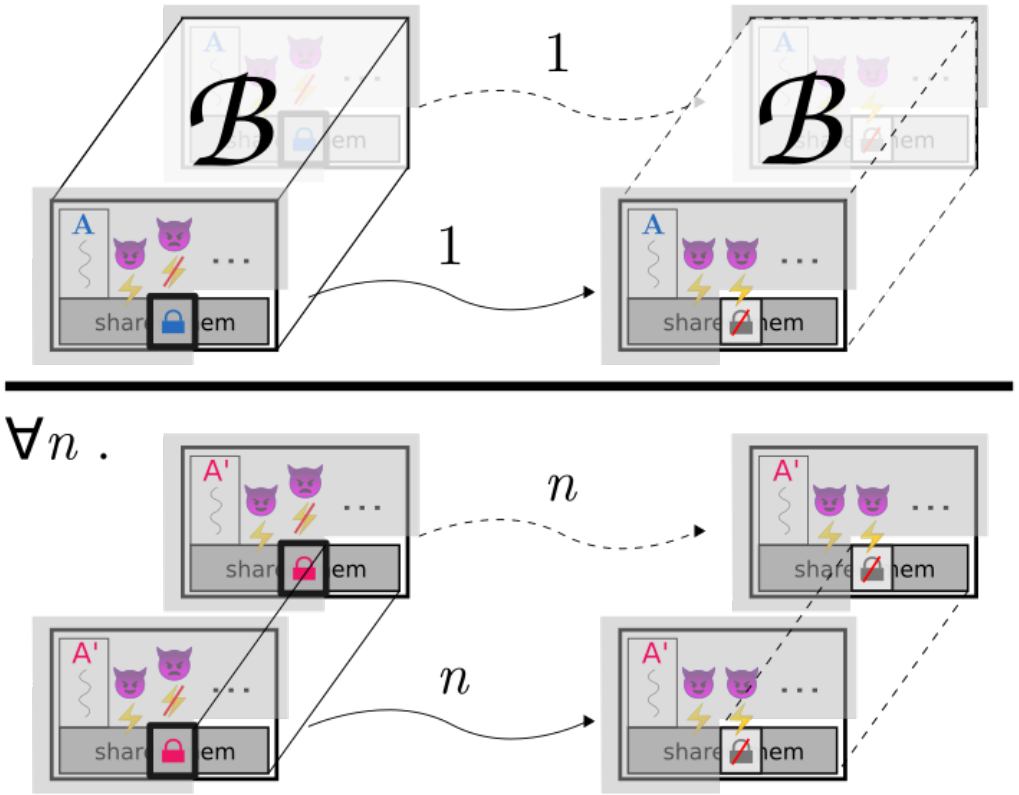


Source

Target

Proof technique for compilation

Per-thread *compositional refinement* [Murray+, CSF'16]
Given bisimulation $\mathcal{B}$ establishing the property,

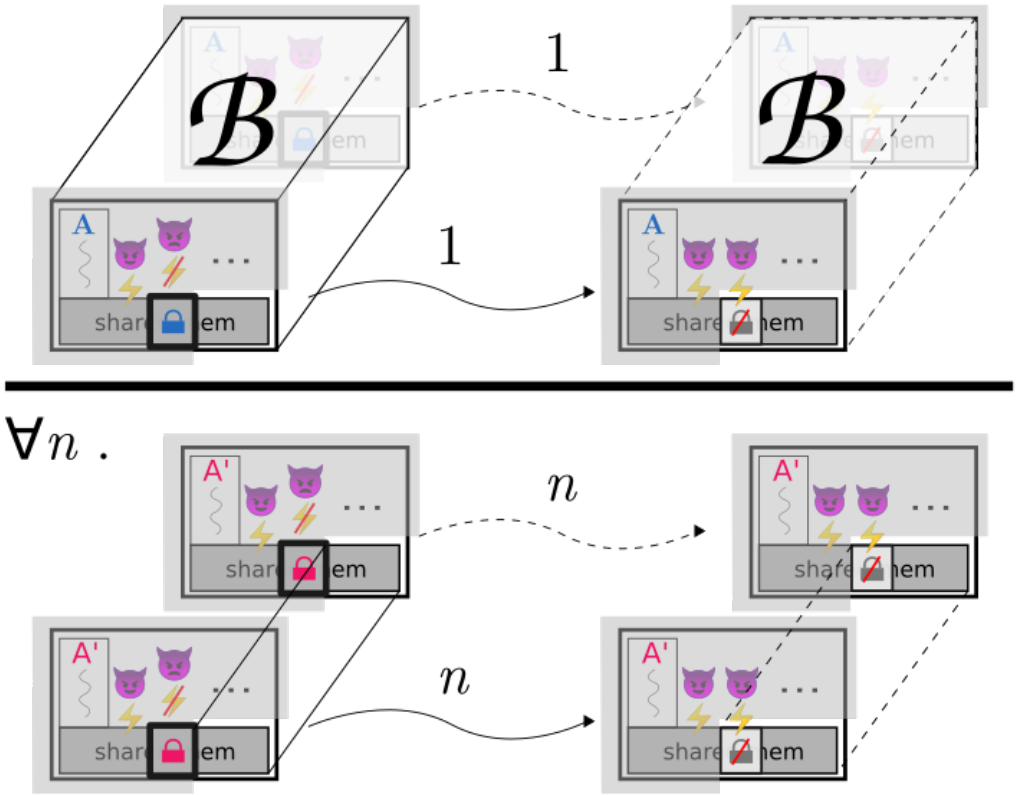


Source

Target

Proof technique for compilation

Per-thread *compositional refinement* [Murray+, CSF'16]
Given bisimulation $\mathcal{B}$ establishing the property, nominate $\mathcal{R}$, $\mathcal{I}$ s.t.:

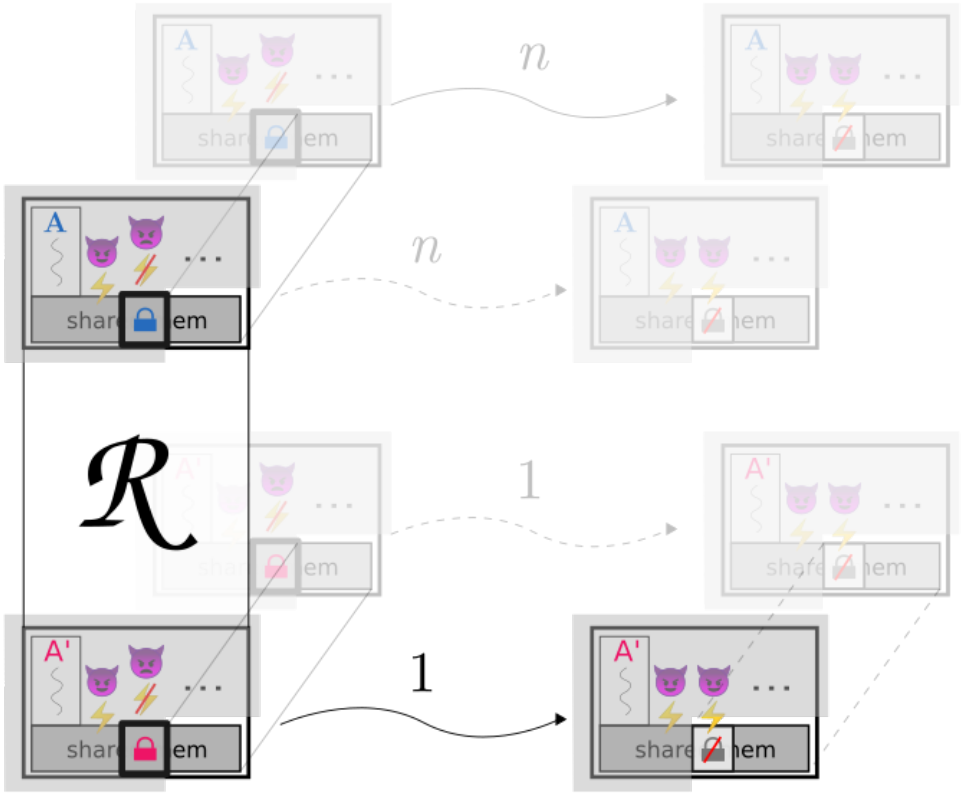


Source

Target

Proof technique for compilation

Per-thread *compositional refinement* [Murray+, CSF'16]
Given bisimulation $\mathcal{B}$ establishing the property, nominate $\mathcal{R}$, $\mathcal{I}$ s.t.:

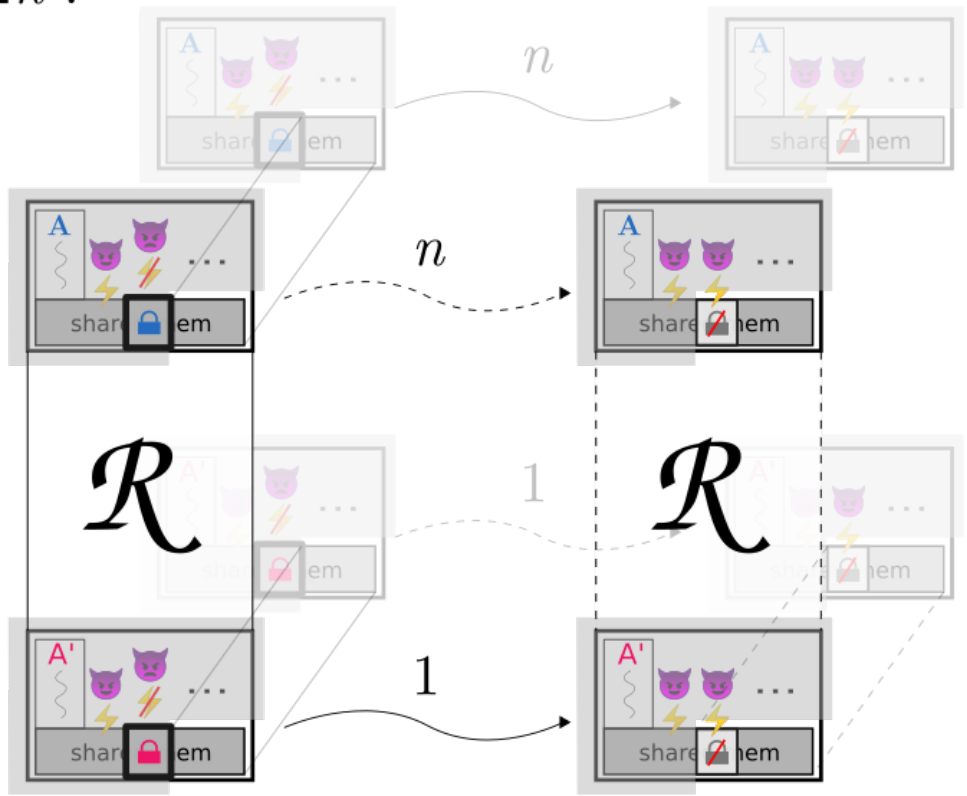


Proof technique for compilation

Per-thread *compositional refinement* [Murray+, CSF'16]

Given bisimulation $\mathcal{B}$ establishing the property, nominate $\mathcal{R}$, $\mathcal{I}$ s.t.:

$\exists n$.

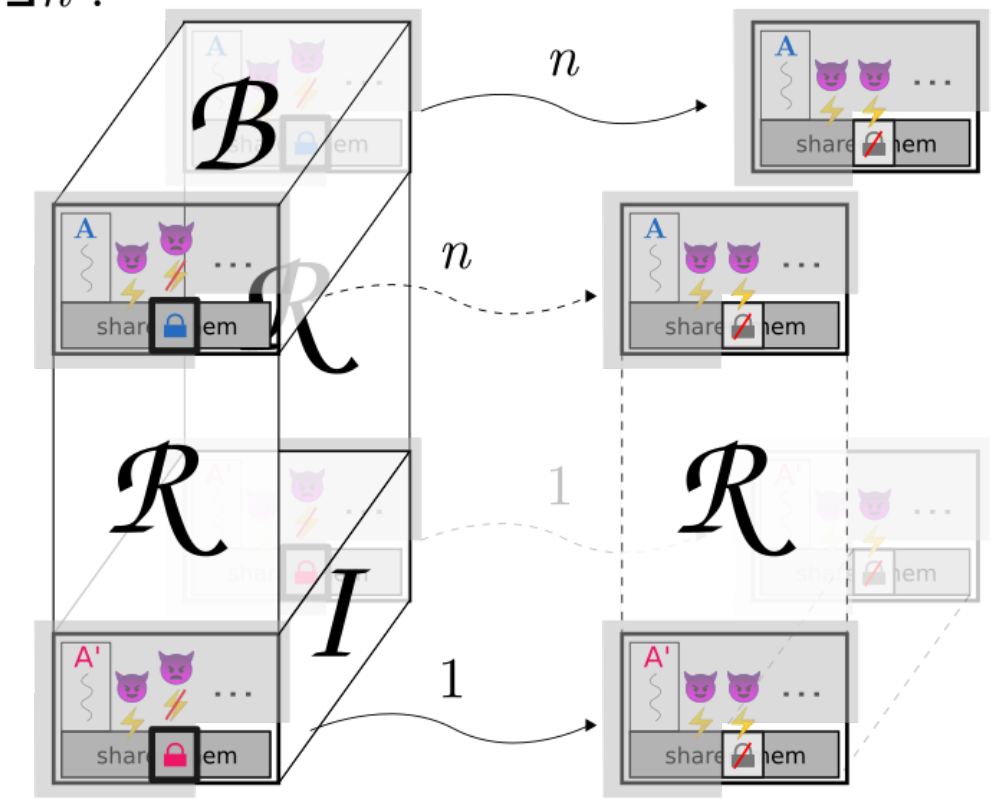


Proof technique for compilation

Per-thread *compositional refinement* [Murray+, CSF'16]

Given bisimulation $\mathcal{B}$ establishing the property, nominate $\mathcal{R}$, $\mathcal{I}$ s.t.:

$\exists n$.

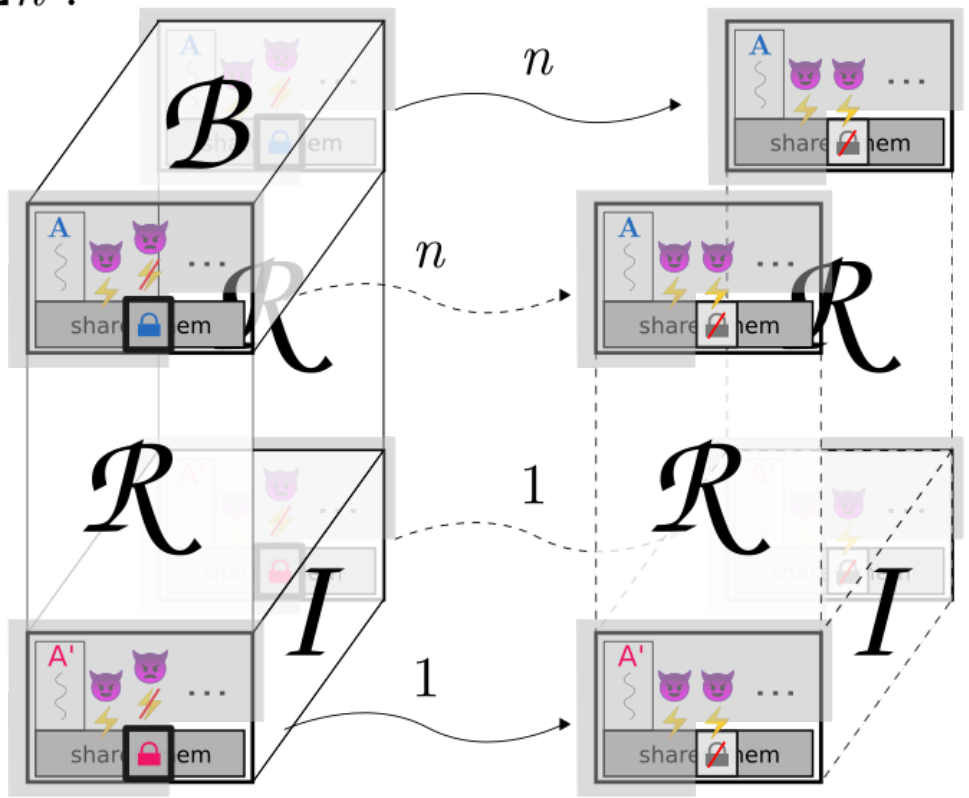


Proof technique for compilation

Per-thread *compositional refinement* [Murray+, CSF'16]

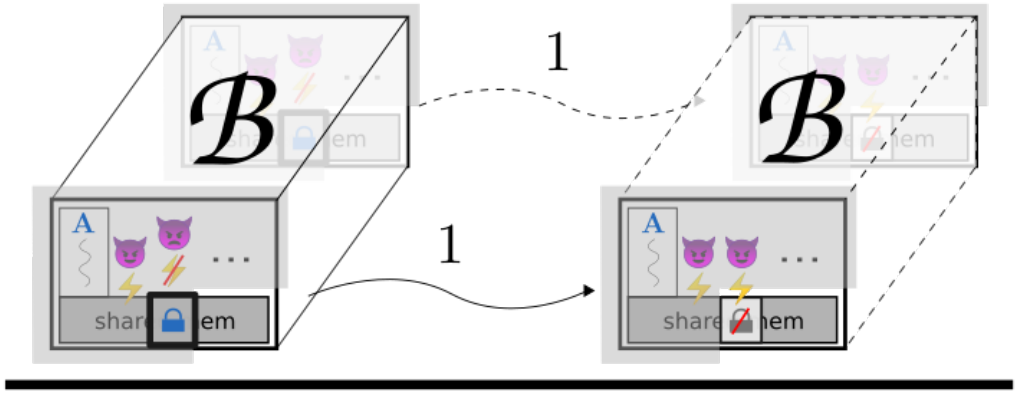
Given bisimulation $\mathcal{B}$ establishing the property, nominate $\mathcal{R}$, $\mathcal{I}$ s.t.:

$\exists n$.

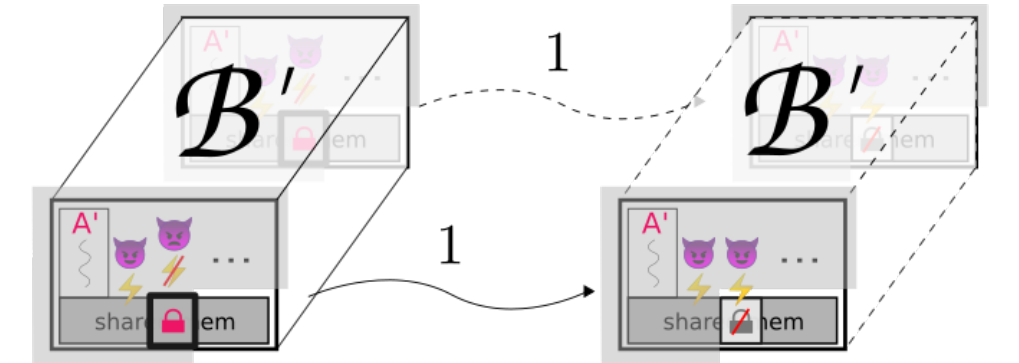


Proof technique for compilation

Per-thread *compositional refinement* [Murray+, CSF'16]
Then $\mathcal{B}'$ $(= \mathcal{B}_T\_of\ \mathcal{B}\ \mathcal{R}\ \mathcal{I})$ establishes the target-level property:



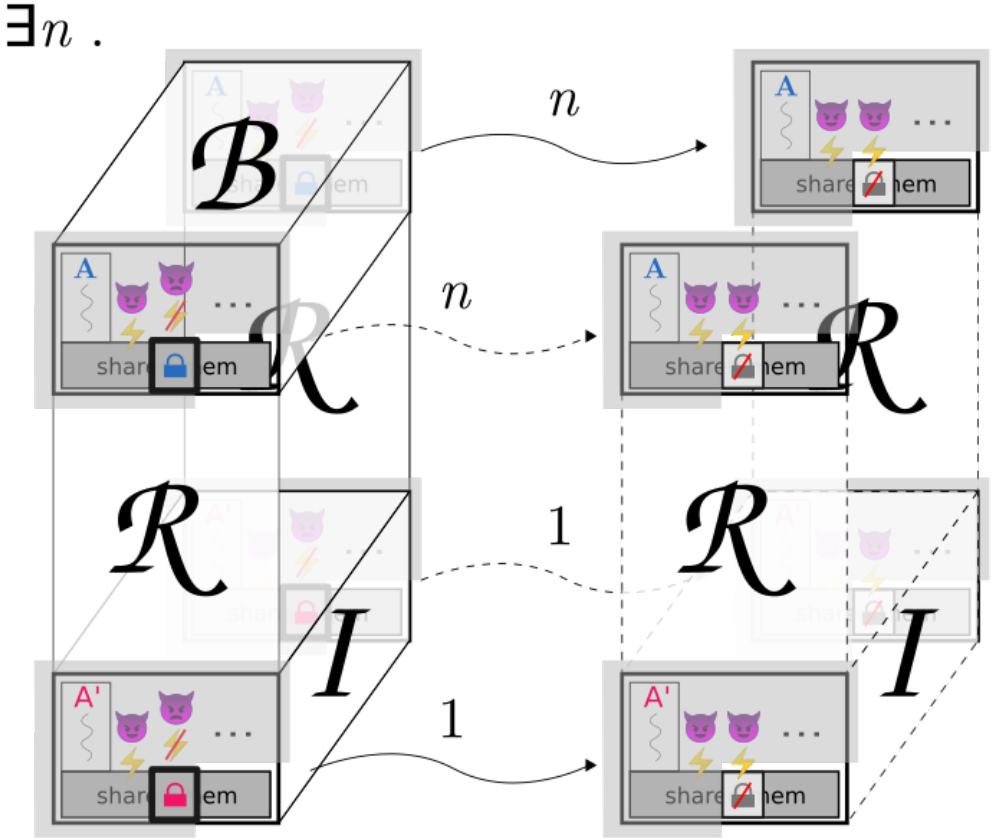
Source



Target

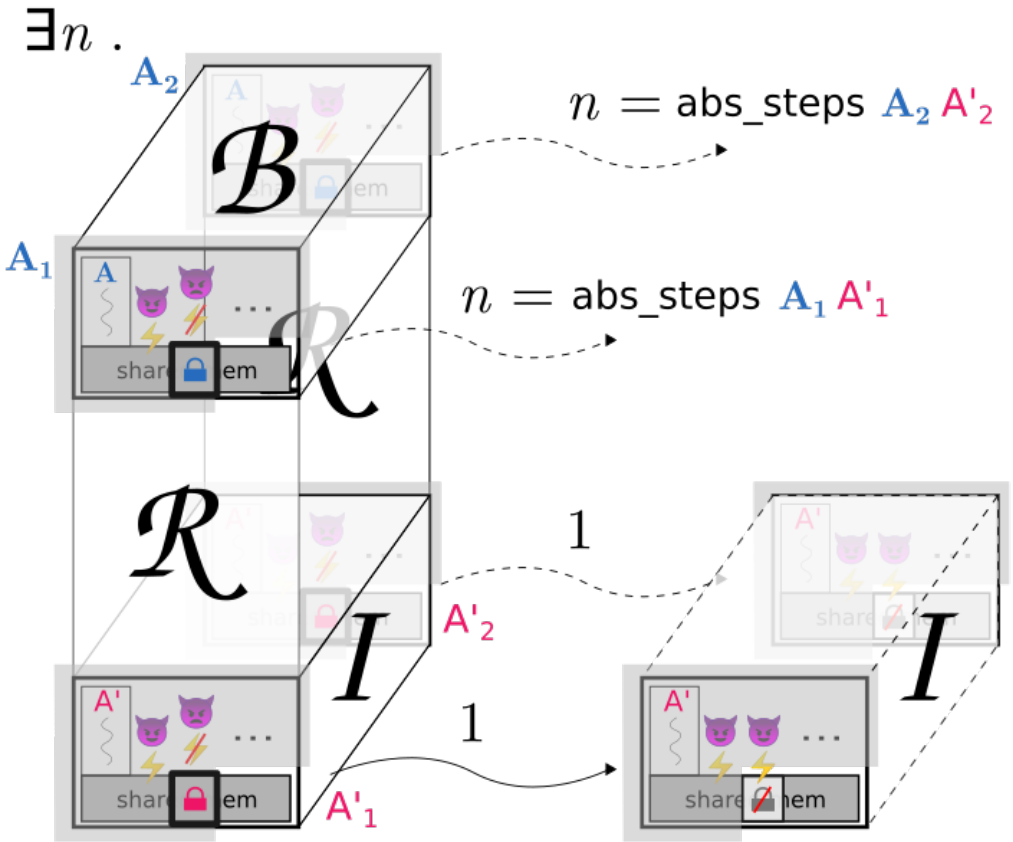
Proof technique for compilation

Simpler proof technique than this!



Proof technique for compilation

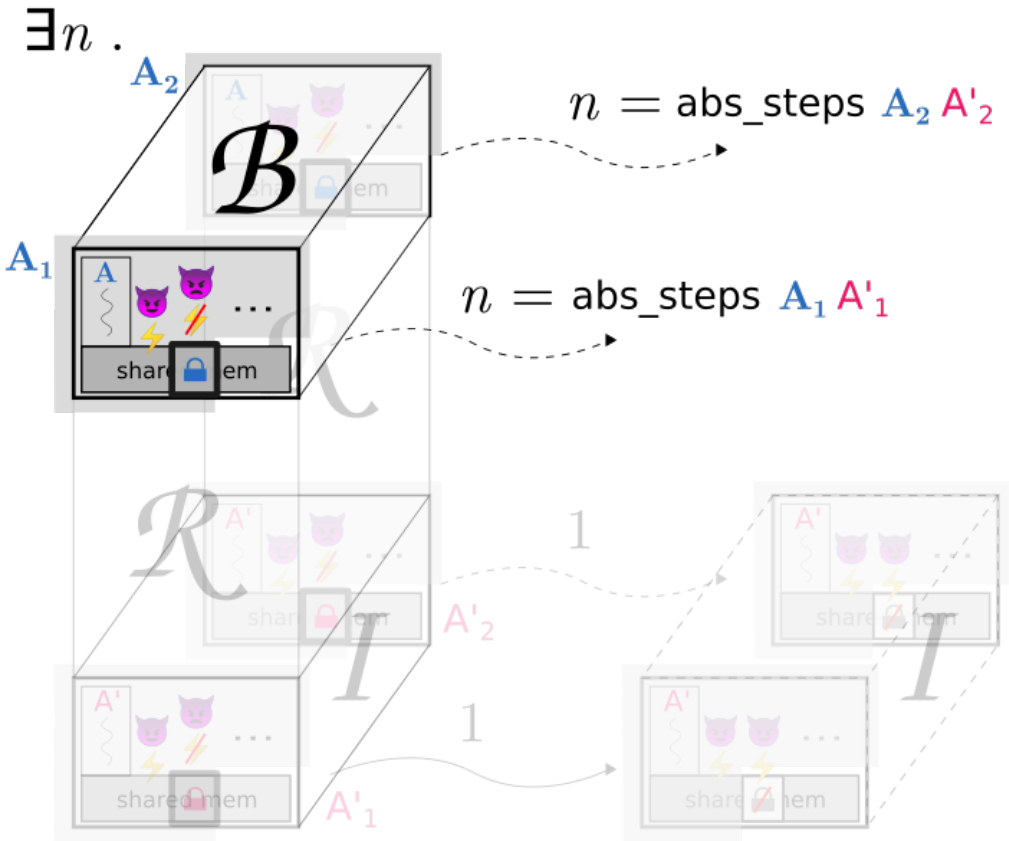
Simpler proof technique! Nominate $\mathcal{R}$, $\mathcal{I}$, abs_steps s.t.



(See: `https://www.isa-afp.org/entries/Dependent_SIFUM_Refinement.html`)

Proof technique for compilation

Simpler proof technique! Nominate $\mathcal{R}$, $\mathcal{I}$, abs_steps s.t.

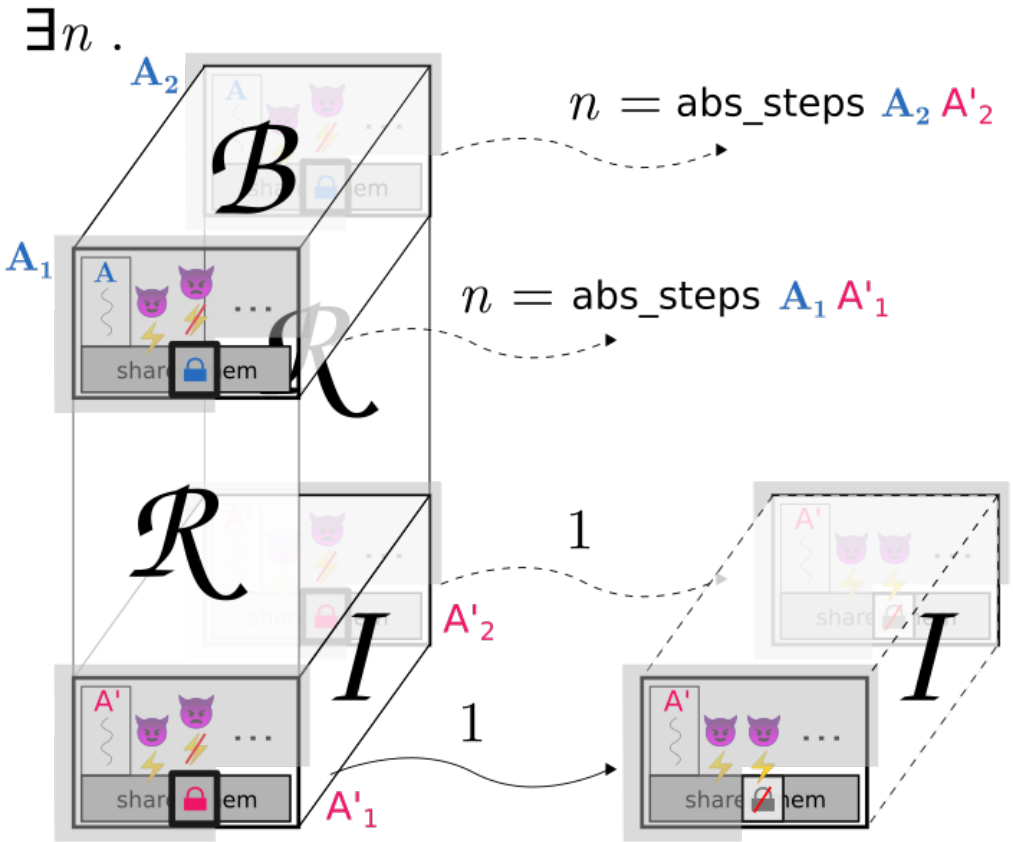


Easy to prove if no H-branching in $\mathbf{A}$

(See: `https://www.isa-afp.org/entries/Dependent_SIFUM_Refinement.html`)

Proof technique for compilation

Simpler proof technique! Nominate $\mathcal{R}$, $\mathcal{I}$, abs_steps s.t.



($\mathcal{I}$ as pc-security)

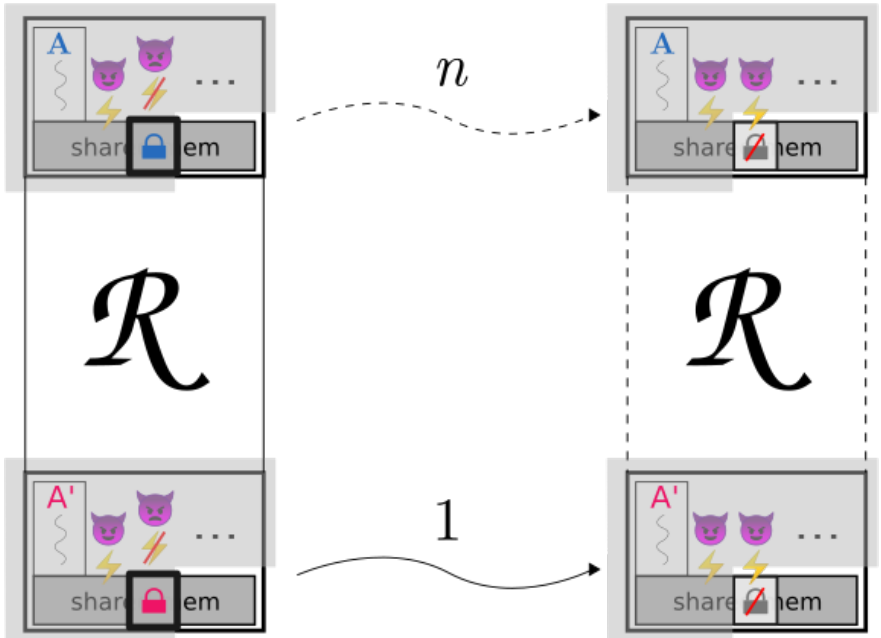
Easy to prove if no H-branching in $\mathbf{A}$, and no new H-branching.

(See: `https://www.isa-afp.org/entries/Dependent_SIFUM_Refinement.html`)

Proof technique for compilation

Simpler proof technique! Nominate $\mathcal{R}$, $\mathcal{I}$, abs_steps.
Then it suffices to prove:

$\exists n \;.\; n =$ abs_steps **A** A'

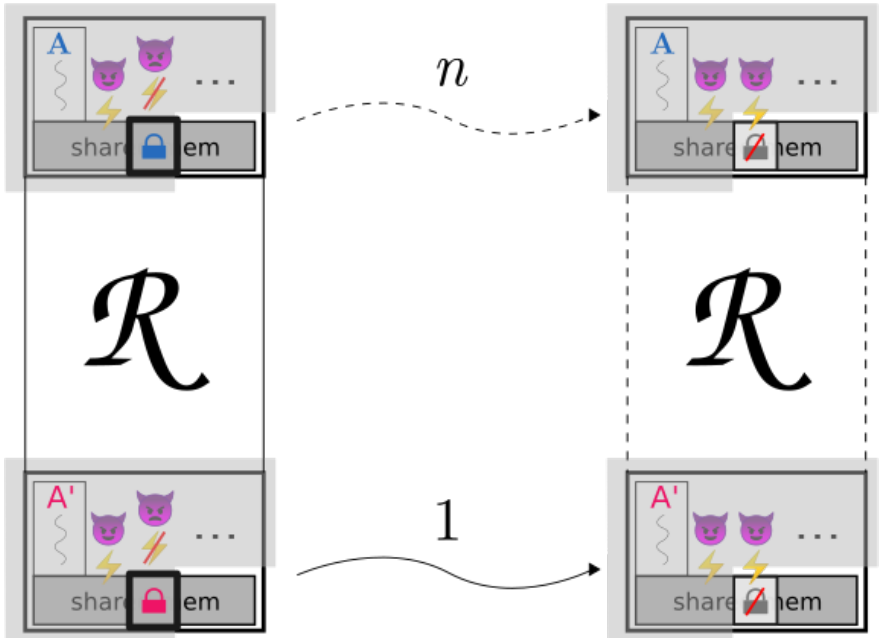


i.e. $\mathcal{R}$ a *simulation* of A' by **A**.

Proof technique for compilation

Simpler proof technique! Nominate $\mathcal{R}$, $\mathcal{I}$, abs_steps.
Then it suffices to prove:

$\exists n \;.\; n =$ abs_steps **A** A'

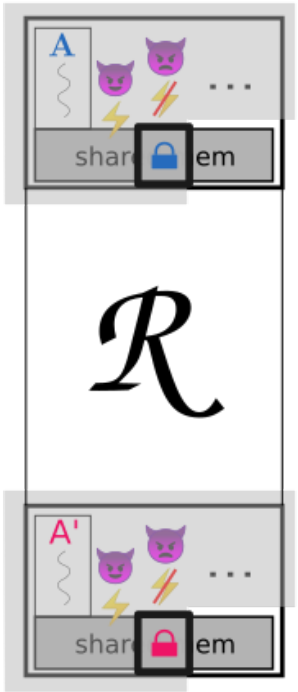


i.e. $\mathcal{R}$ a *simulation* of A' by **A**, with provisos...

Proof technique for compilation

Provisos for $\mathcal{R}$, $\mathcal{I}$:

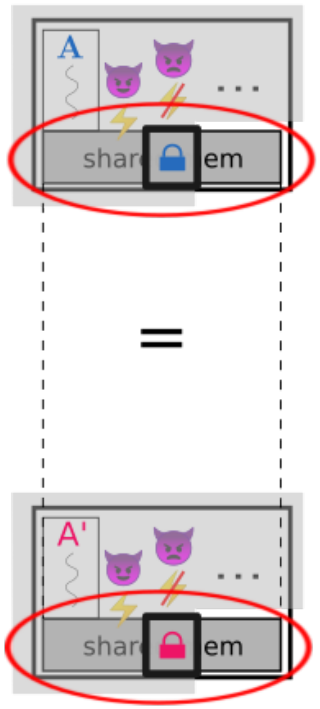
- $\mathcal{R}$ must preserve shared memory contents and locking state.
  - Under the hood: preserve assumptions and guarantees.



Proof technique for compilation

Provisos for $\mathcal{R}$, $\mathcal{I}$:

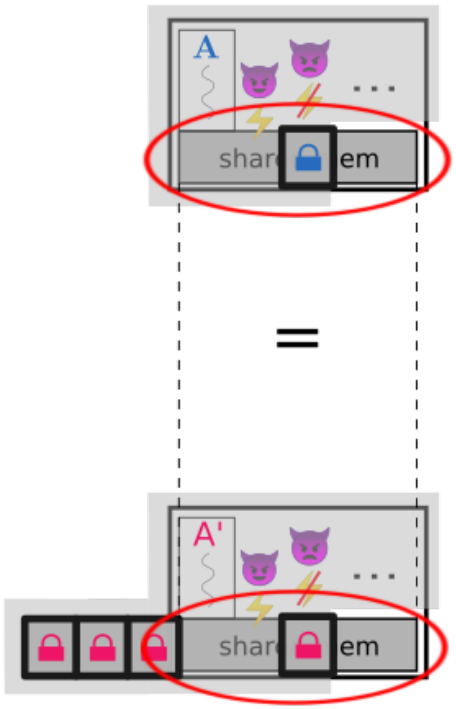
- $\mathcal{R}$ must preserve shared memory contents and locking state.
  - Under the hood: preserve assumptions and guarantees.



Proof technique for compilation

Provisos for $\mathcal{R}$, $\mathcal{I}$:

- $\mathcal{R}$ must preserve shared memory contents and locking state.
  - Under the hood: preserve assumptions and guarantees.



+ any new locations permanently locked.

i.e. No new shared state.

Proof technique for compilation

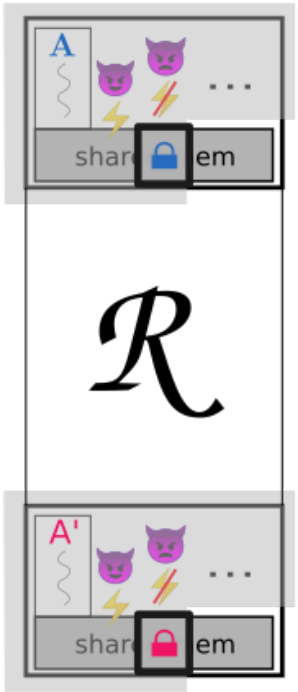
Provisos for $\mathcal{R}$, $\mathcal{I}$:

- $\mathcal{R}$ must preserve shared memory contents and locking state.
  - Under the hood: preserve assumptions and guarantees.
- $\mathcal{R}$ must be closed under lock-permitted shared memory havoc.

Proof technique for compilation

Provisos for $\mathcal{R}$, $\mathcal{I}$:

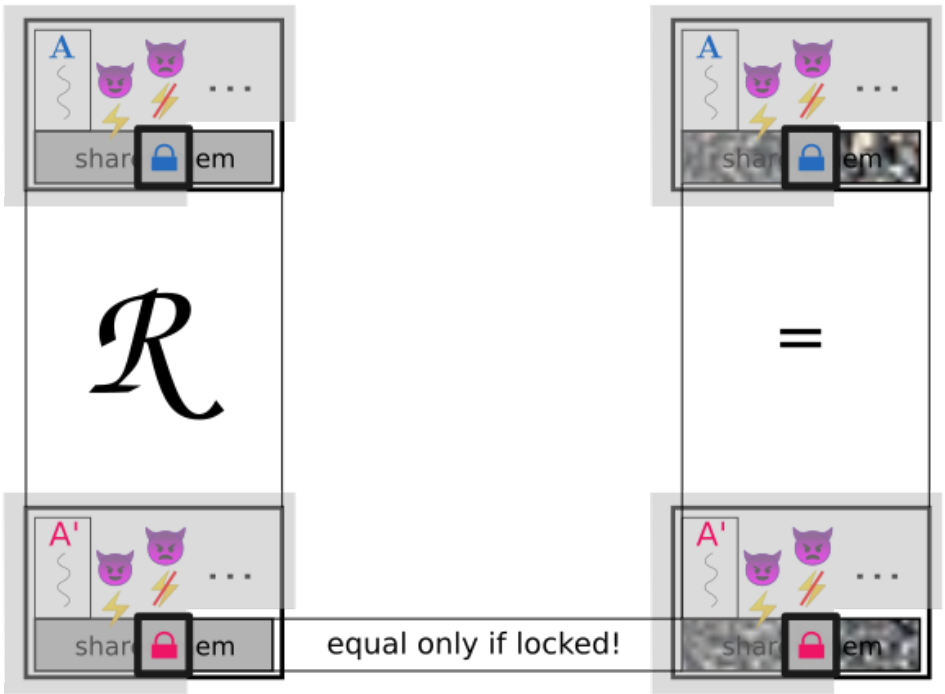
- $\mathcal{R}$ must preserve shared memory contents and locking state.
  - Under the hood: preserve assumptions and guarantees.
- $\mathcal{R}$ must be closed under lock-permitted shared memory havoc.



Proof technique for compilation

Provisos for $\mathcal{R}$, $\mathcal{I}$:

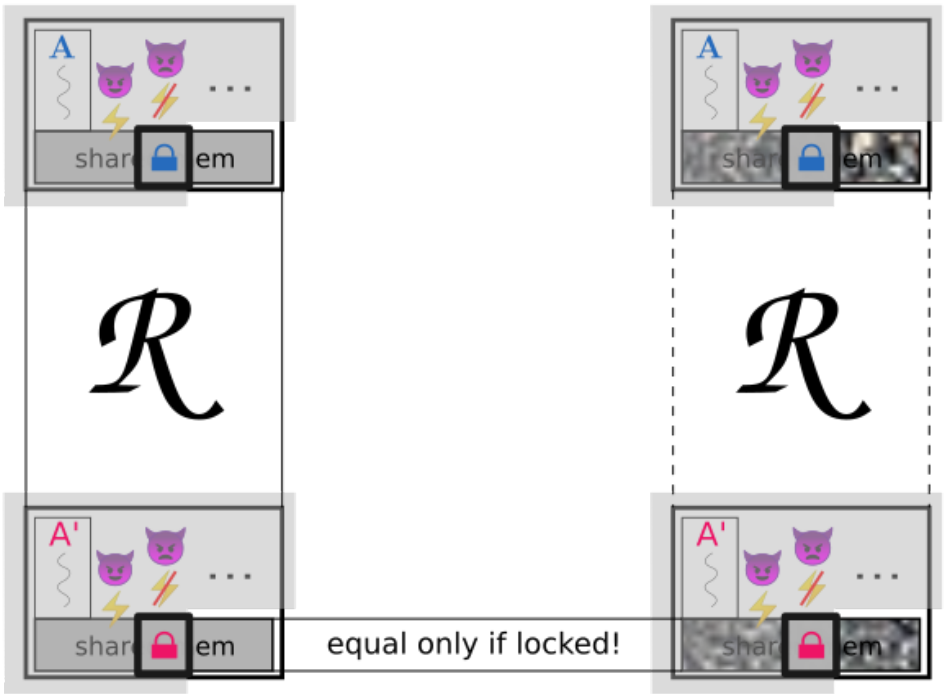
- $\mathcal{R}$ must preserve shared memory contents and locking state.
  - Under the hood: preserve assumptions and guarantees.
- $\mathcal{R}$ must be closed under lock-permitted shared memory havoc.



Proof technique for compilation

Provisos for $\mathcal{R}$, $\mathcal{I}$:

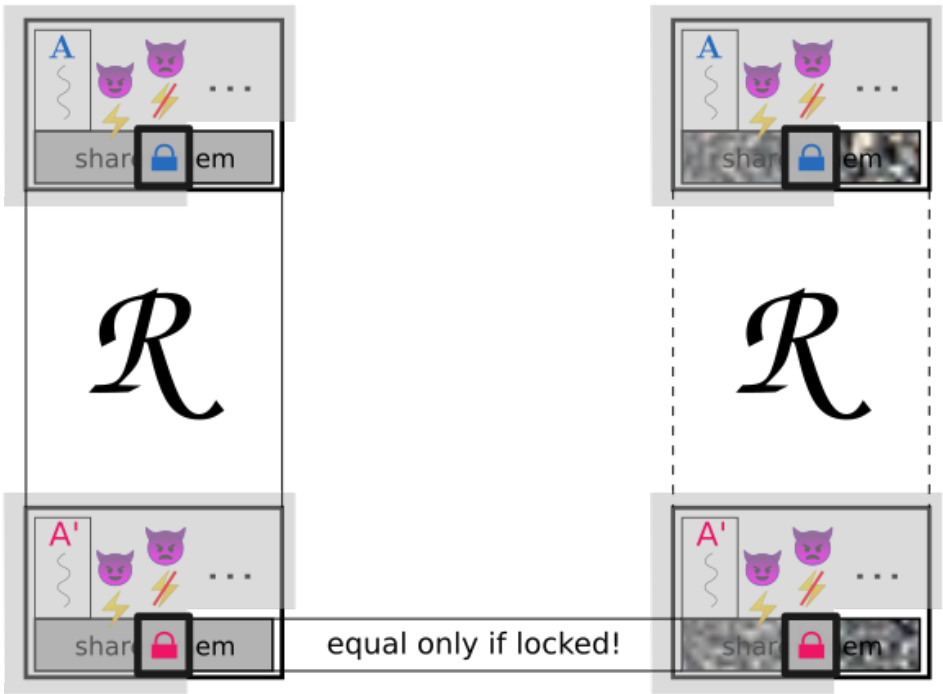
- $\mathcal{R}$ must preserve shared memory contents and locking state.
  - Under the hood: preserve assumptions and guarantees.
- $\mathcal{R}$ must be closed under lock-permitted shared memory havoc.



Proof technique for compilation

Provisos for $\mathcal{R}$, $\mathcal{I}$:

- $\mathcal{R}$ must preserve shared memory contents and locking state.
  - Under the hood: preserve assumptions and guarantees.
- $\mathcal{R}$ must be closed under lock-permitted shared memory havoc.



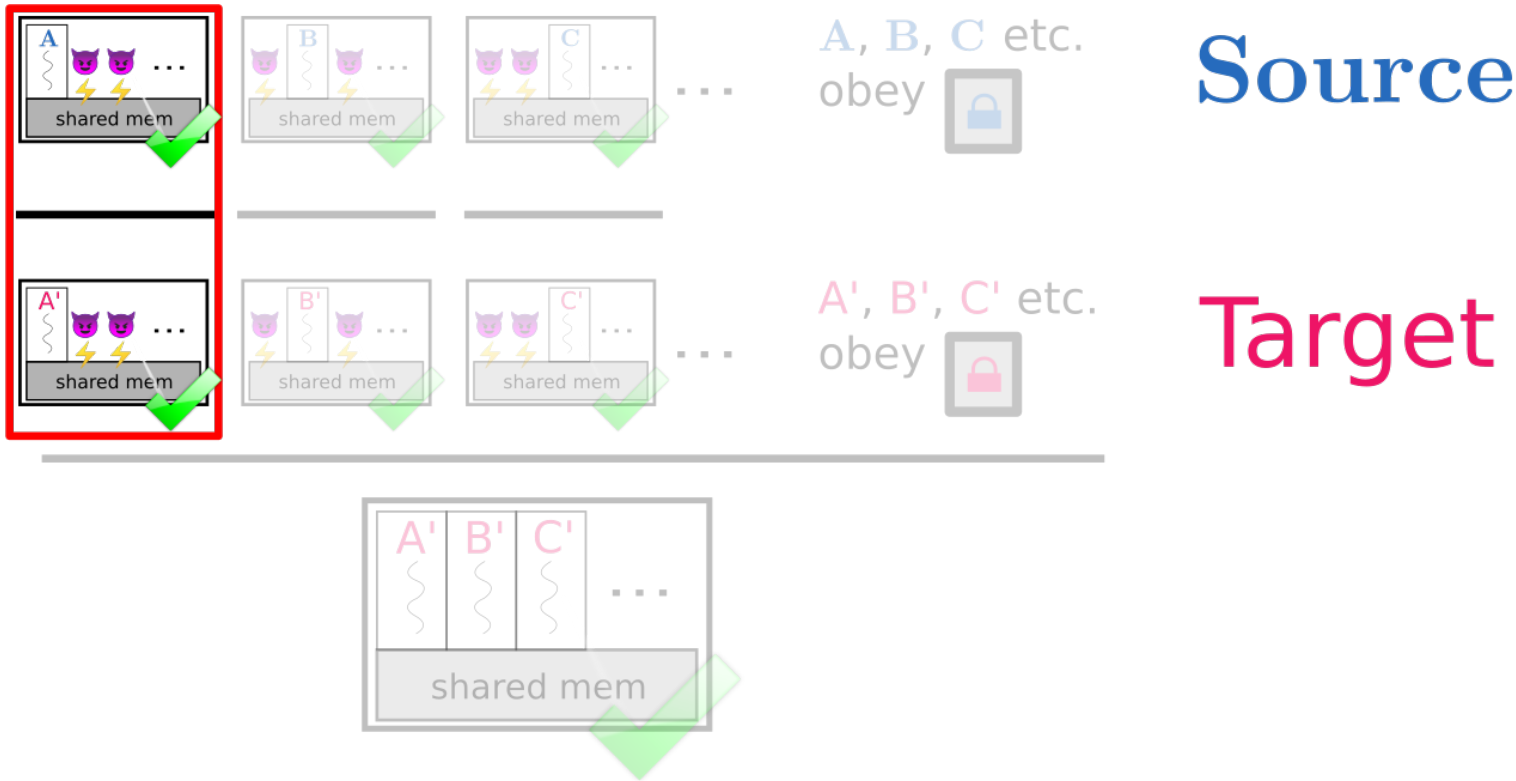
Similar for $\mathcal{I}$.

This talk

Part 1: Concurrent value-dependent noninterference

Part 2: Per-thread compositional refinement

Part 3: While-to-RISC compiler verification

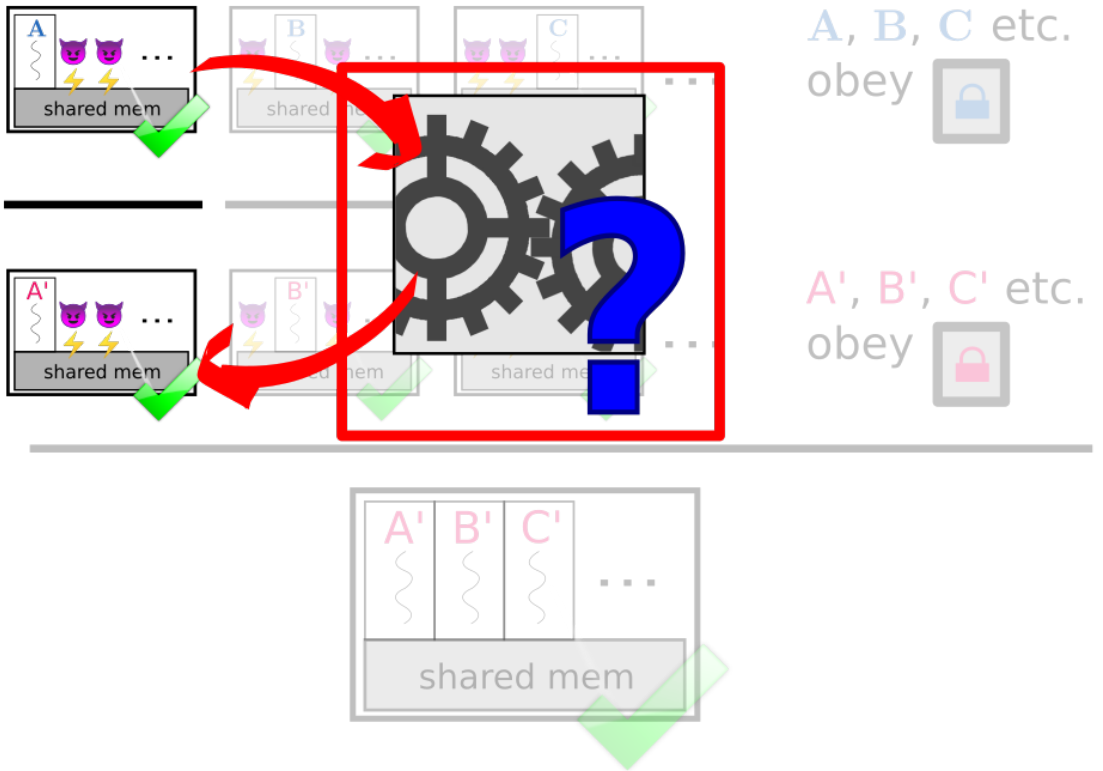


This talk

Part 1: Concurrent value-dependent noninterference

Part 2: Per-thread compositional refinement

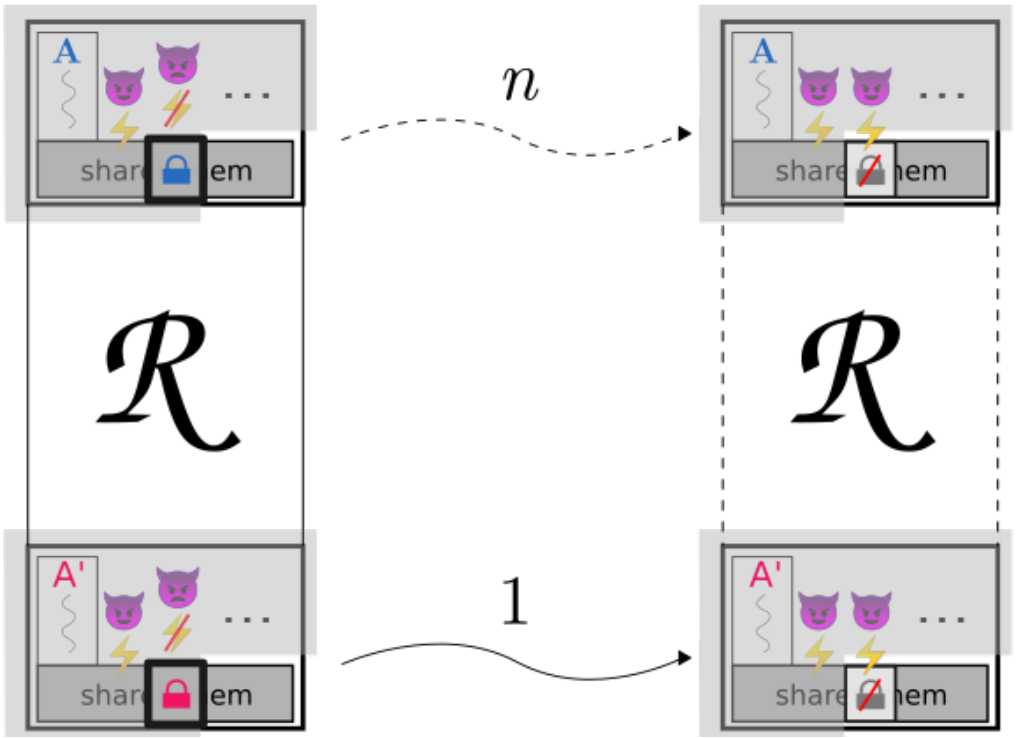
Part 3: While-to-RISC compiler verification



Compiler verification

Per-thread *simpler compositional refinement* [Murray+, AFP], instantiated with $\mathcal{R}$ characterising a compiler.

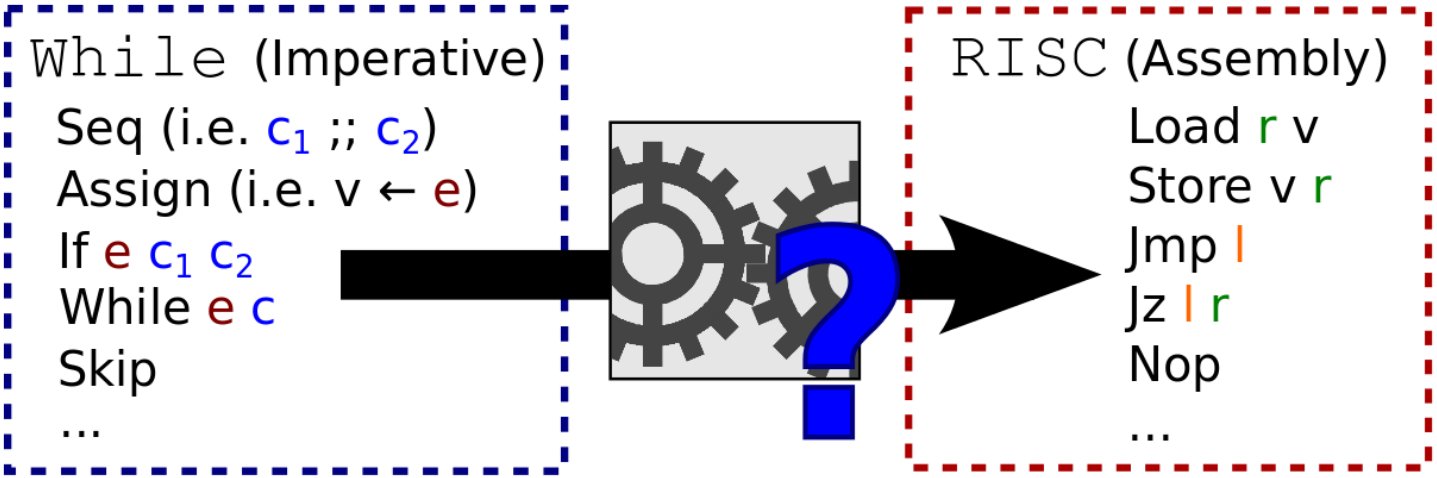
$\exists n \;.\; n =$ abs_steps A A'



Compiler verification

Per-thread *simpler compositional refinement* [Murray+, AFP], instantiated with $\mathcal{R}$ characterising a compiler.

Proof of concept: a While-to-RISC compiler



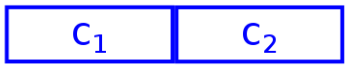
(Note: Constant-time execution steps, no cache effects)

Compiler verification

Per-thread *simpler compositional refinement* [Murray+, AFP], instantiated with $\mathcal{R}$ characterising a compiler.

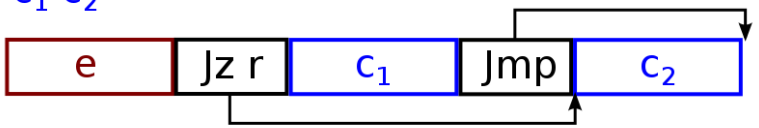
Proof of concept: a While-to-RISC compiler

- Seq (i.e. c_1 ;; c_2)

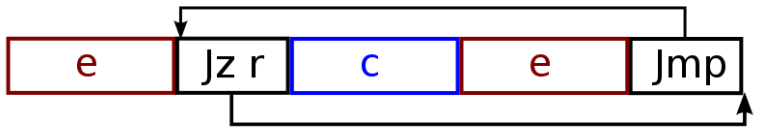


- Assign (i.e. v ← e)

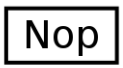
- If e c_1 c_2



- While e c



- Skip



~~- LockAcq l~~

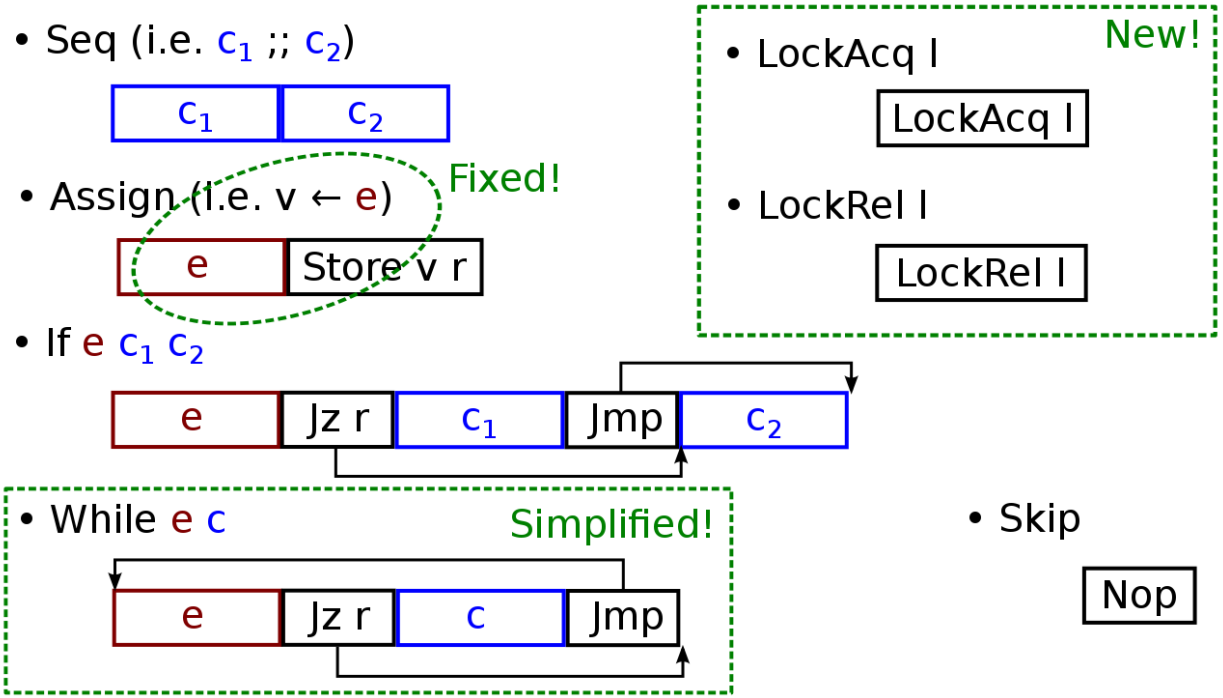
~~- LockRel l~~

Based on *Fault-Resilient Non-interference* [Tedesco et al, 2016].

Compiler verification

Per-thread *simpler compositional refinement* [Murray+, AFP], instantiated with $\mathcal{R}$ characterising a compiler.

Proof of concept: a While-to-RISC compiler



Based on *Fault-Resilient Non-interference* [Tedesco et al, 2016].
Implemented in Isabelle/HOL, executable, verified.

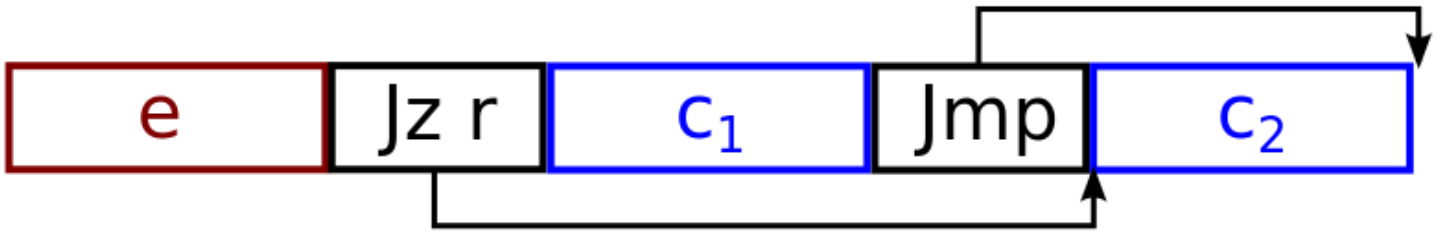
Compiler verification

Per-thread *simpler compositional refinement* [Murray+, AFP], instantiated with $\mathcal{R}$ characterising a compiler.

Proof of concept: a While-to-RISC compiler

e.g. $\mathcal{R}$ cases for If construct

If e c_1 c_2

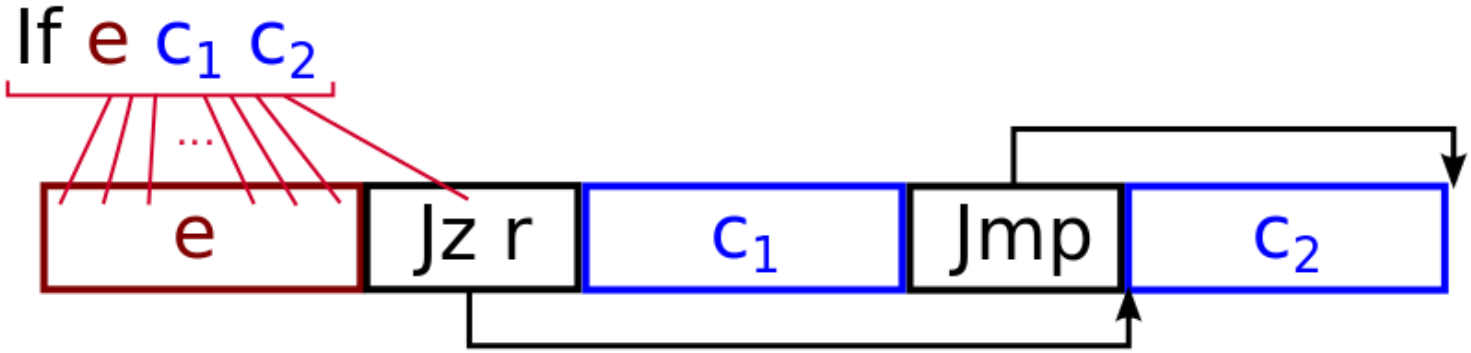


Compiler verification

Per-thread *simpler compositional refinement* [Murray+, AFP], instantiated with $\mathcal{R}$ characterising a compiler.

Proof of concept: a While-to-RISC compiler

e.g. $\mathcal{R}$ cases for If construct

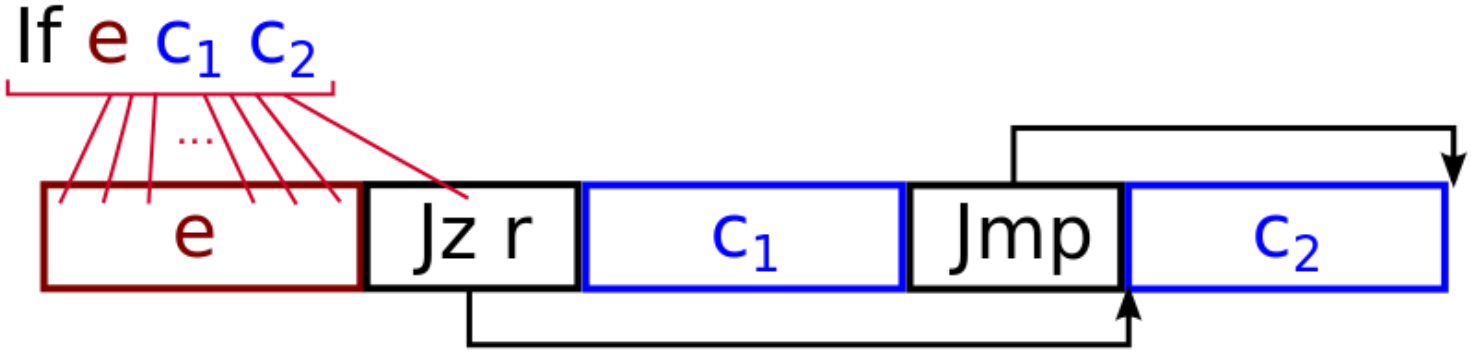


Compiler verification

Per-thread *simpler compositional refinement* [Murray+, AFP], instantiated with $\mathcal{R}$ characterising a compiler.

Proof of concept: a While-to-RISC compiler

e.g. $\mathcal{R}$ cases for If construct, c_1 case:

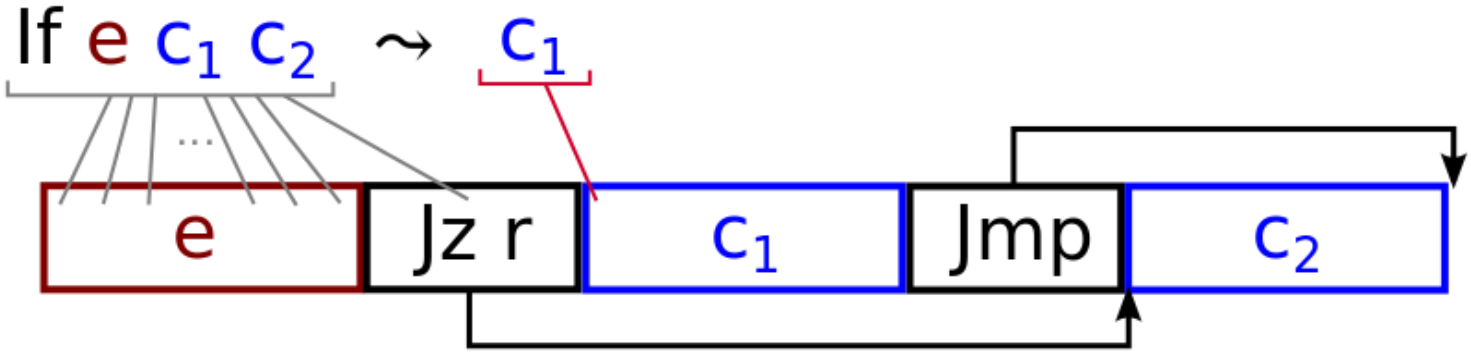


Compiler verification

Per-thread *simpler compositional refinement* [Murray+, AFP], instantiated with $\mathcal{R}$ characterising a compiler.

Proof of concept: a While-to-RISC compiler

e.g. $\mathcal{R}$ cases for If construct, c_1 case:



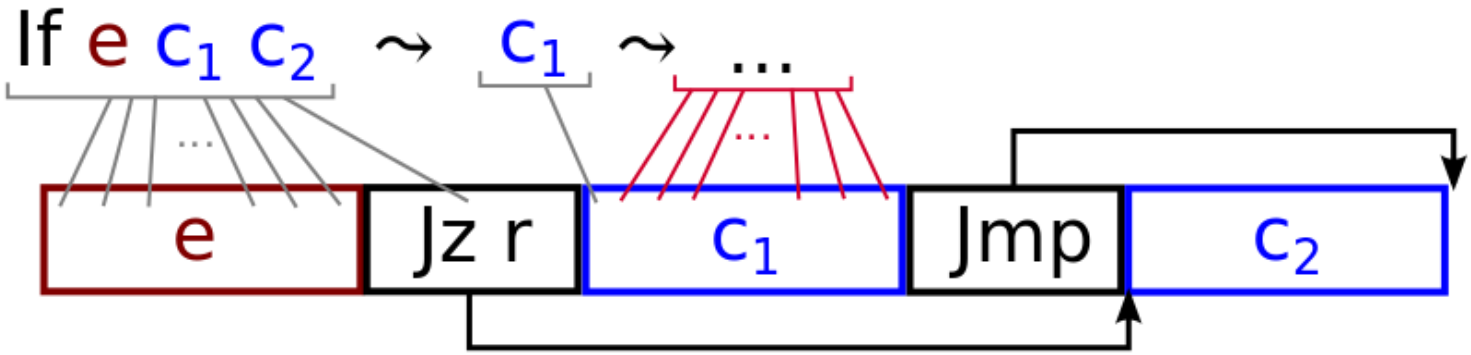
Relation is inductive for smaller program pairs c_1, c_2

Compiler verification

Per-thread *simpler compositional refinement* [Murray+, AFP], instantiated with $\mathcal{R}$ characterising a compiler.

Proof of concept: a While-to-RISC compiler

e.g. $\mathcal{R}$ cases for If construct, c_1 case:



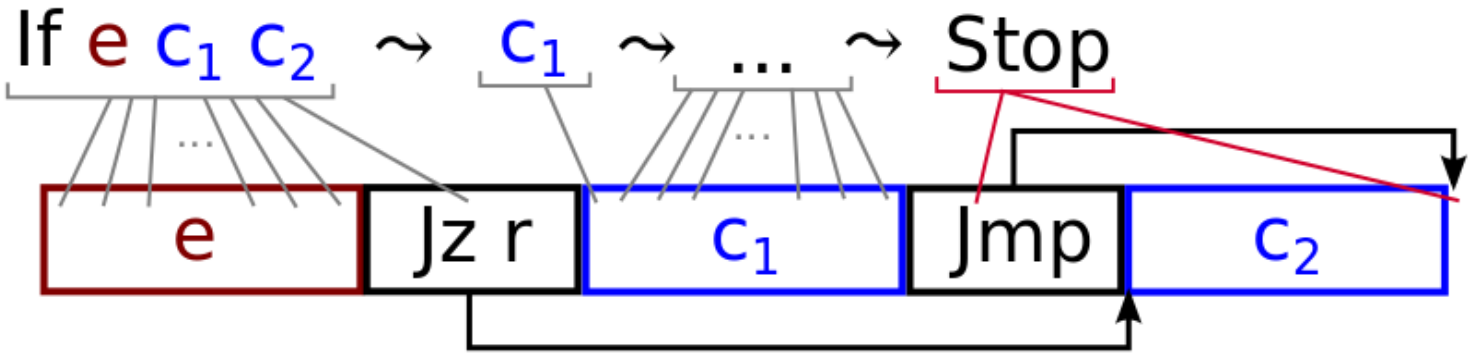
Relation is inductive for smaller program pairs c_1, c_2

Compiler verification

Per-thread *simpler compositional refinement* [Murray+, AFP], instantiated with $\mathcal{R}$ characterising a compiler.

Proof of concept: a While-to-RISC compiler

e.g. $\mathcal{R}$ cases for If construct, c_1 case:



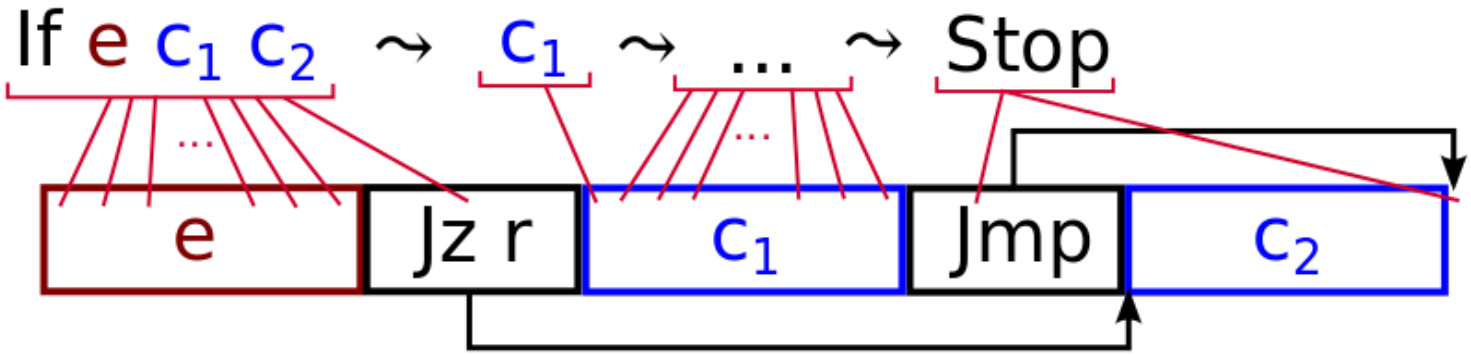
Relation is inductive for smaller program pairs c_1, c_2

Compiler verification

Per-thread *simpler compositional refinement* [Murray+, AFP], instantiated with $\mathcal{R}$ characterising a compiler.

Proof of concept: a While-to-RISC compiler

e.g. $\mathcal{R}$ cases for If construct, c_1 case:



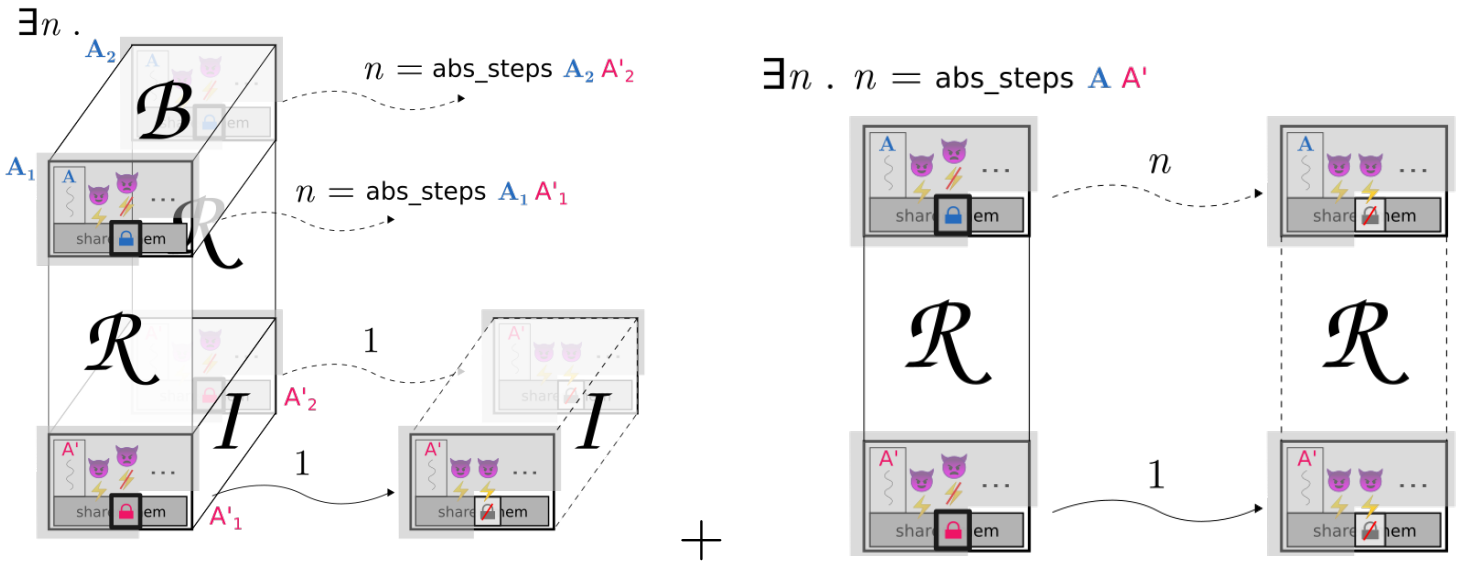
Relation is inductive for smaller program pairs c_1, c_2

Compiler verification

Per-thread *simpler compositional refinement* [Murray+, AFP], instantiated with $\mathcal{R}$ characterising a compiler.

Proof of concept: a While-to-RISC compiler

- Theorem: $\mathcal{R}$ preserves per-thread compositional value-dependent noninterference property
  - for $\mathcal{B}$ produced by our type system (no H-branching).
  - for $\mathcal{I}$ asserting equal pc and program text.

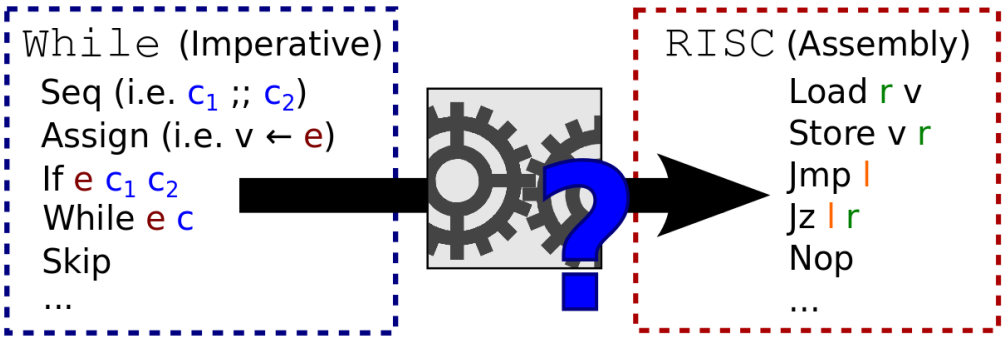


Compiler verification

Per-thread *simpler compositional refinement* [Murray+, AFP], instantiated with $\mathcal{R}$ characterising a compiler.

Proof of concept: a While-to-RISC compiler

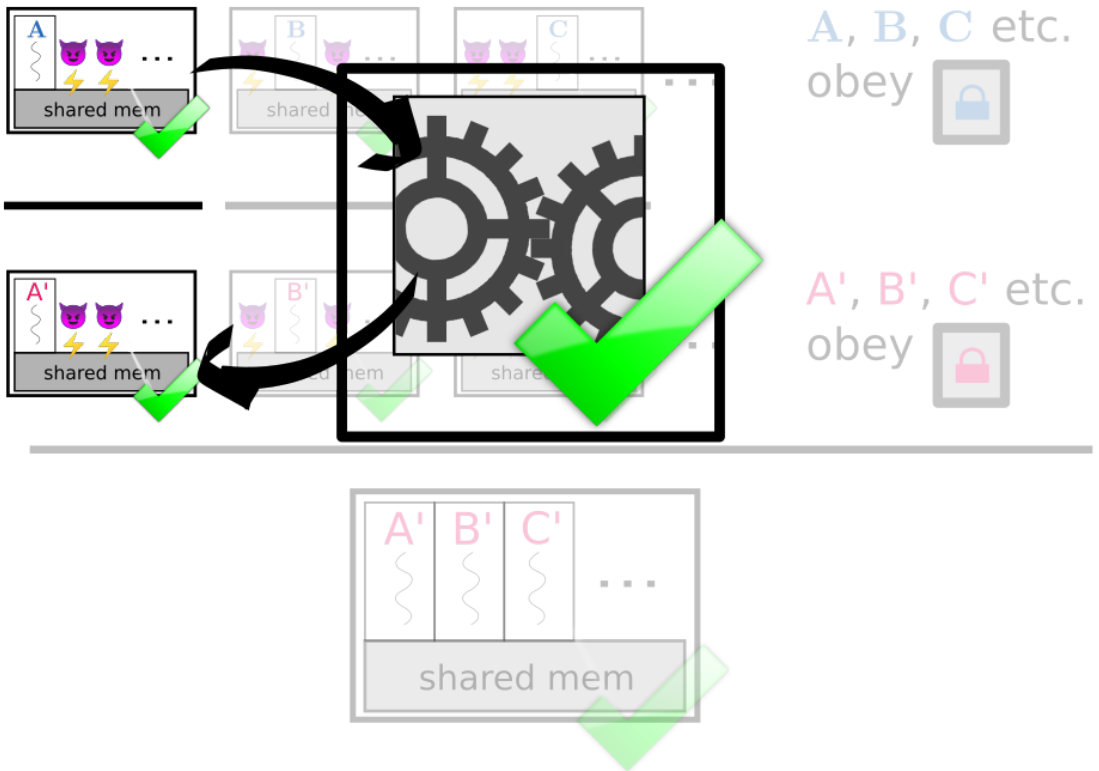
- Theorem: $\mathcal{R}$ preserves per-thread compositional value-dependent noninterference property
  - for $\mathcal{B}$ produced by our type system (no H-branching).
  - for $\mathcal{I}$ asserting equal pc and program text.
- Theorem: Compiler input is related to its output by $\mathcal{R}$
  - Started with same observable initial state.
  - No branching on H values. (Same as for type system.)



Compiler verification

Per-thread *simpler compositional refinement* [Murray+, AFP], instantiated with $\mathcal{R}$ characterising a compiler.

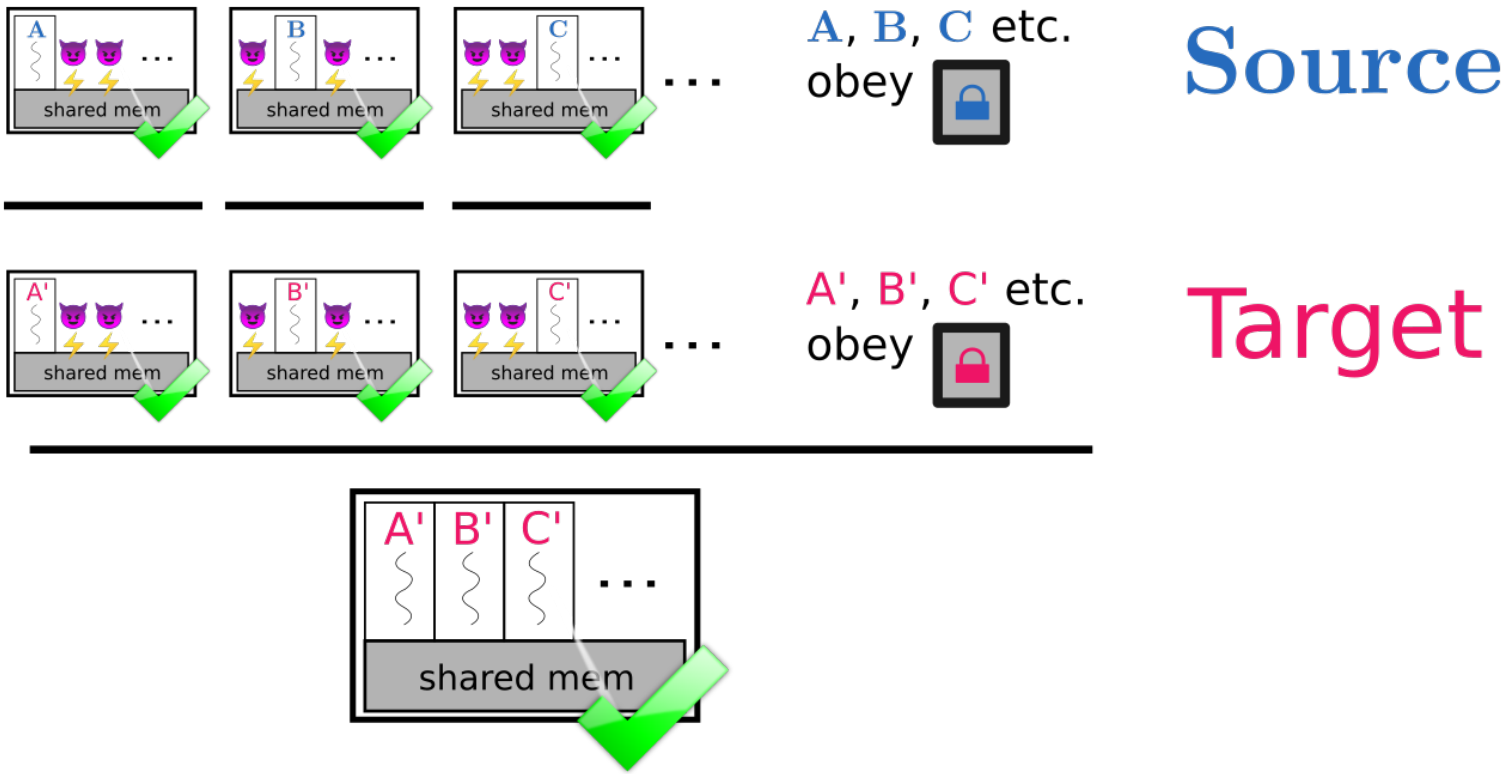
Proof of concept: a While-to-RISC compiler



Compiler verification

Per-thread *simpler compositional refinement* [Murray+, AFP], instantiated with $\mathcal{R}$ characterising a compiler.

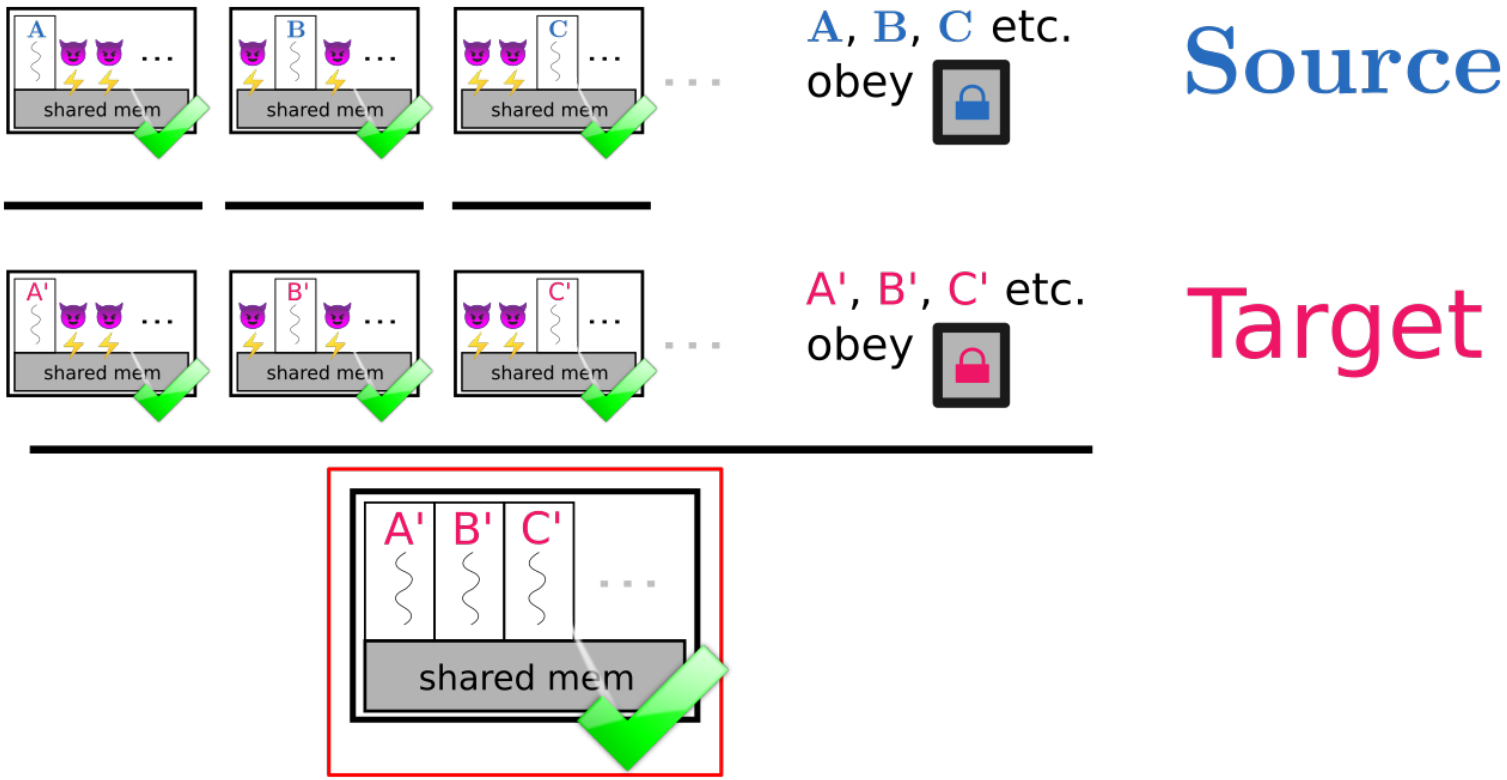
Proof of concept: a While-to-RISC compiler



Compiler verification

Per-thread *simpler compositional refinement* [Murray+, AFP], instantiated with $\mathcal{R}$ characterising a compiler.

Proof of concept: a While-to-RISC compiler



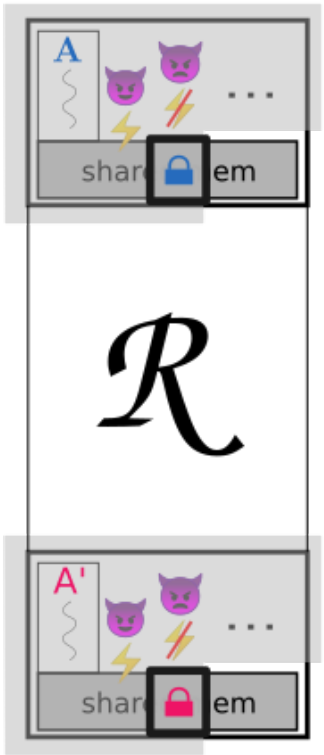
Exercised on verified Cross Domain Desktop Compositor model.

Limitations and future work ideas

- Optimisations to non-observable shared memory?

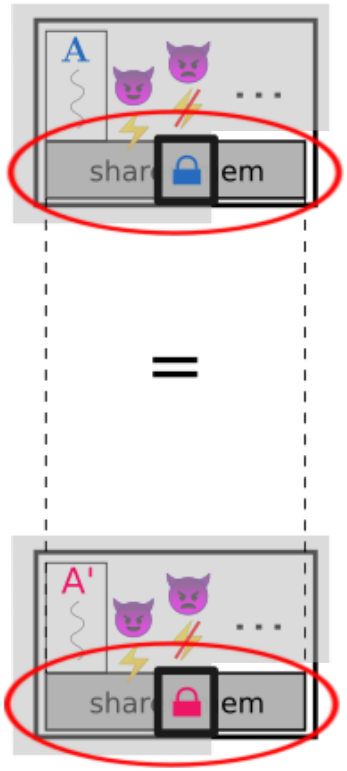
Limitations and future work ideas

- Optimisations to non-observable shared memory? Possibly too strict.



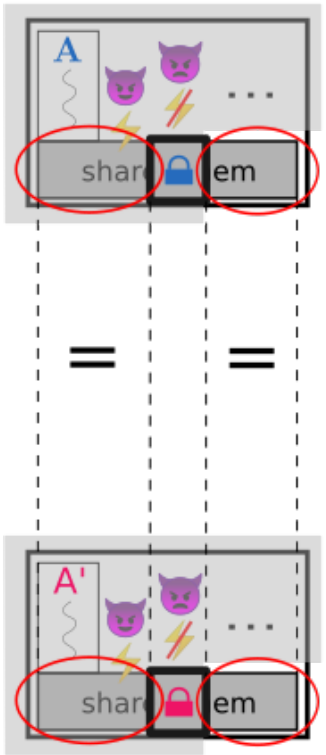
Limitations and future work ideas

- Optimisations to non-observable shared memory? Possibly too strict.



Limitations and future work ideas

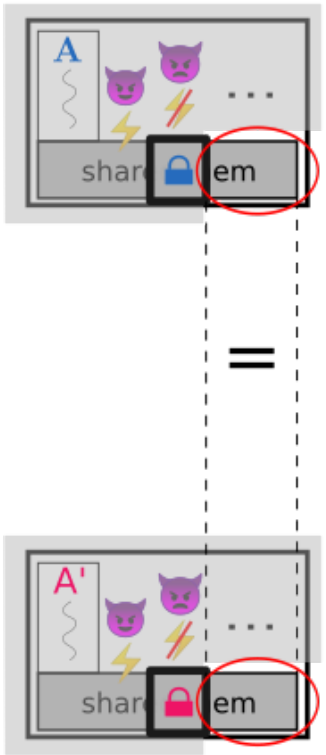
- Optimisations to non-observable shared memory? Possibly too strict.



Relax for shared memory out of reach of attacker model?

Limitations and future work ideas

- Optimisations to non-observable shared memory? Possibly too strict.



Relax for shared memory out of reach of attacker model?

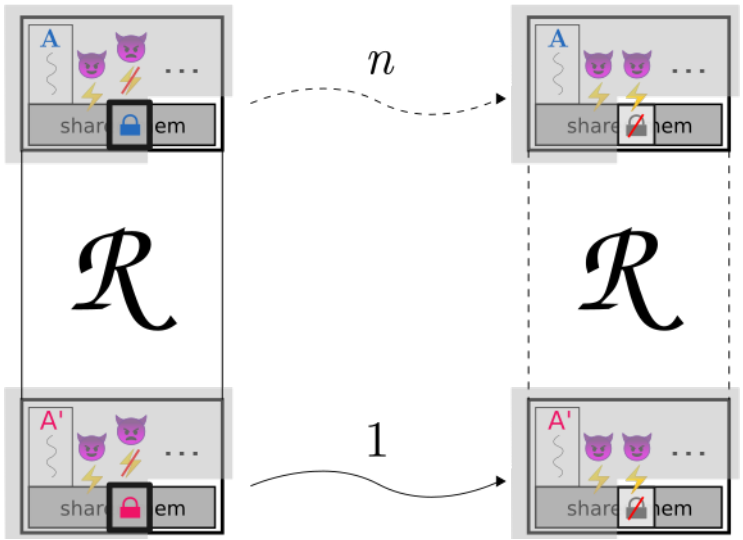
Limitations and future work ideas

- Optimisations to non-observable shared memory?
- Can existing compilers be proven to satisfy it?

Limitations and future work ideas

- Optimisations to non-observable shared memory?
- Can existing compilers be proven to satisfy it? CompCert?
  - small-step semantics, volatile R/W observable
  - simulation of target by source

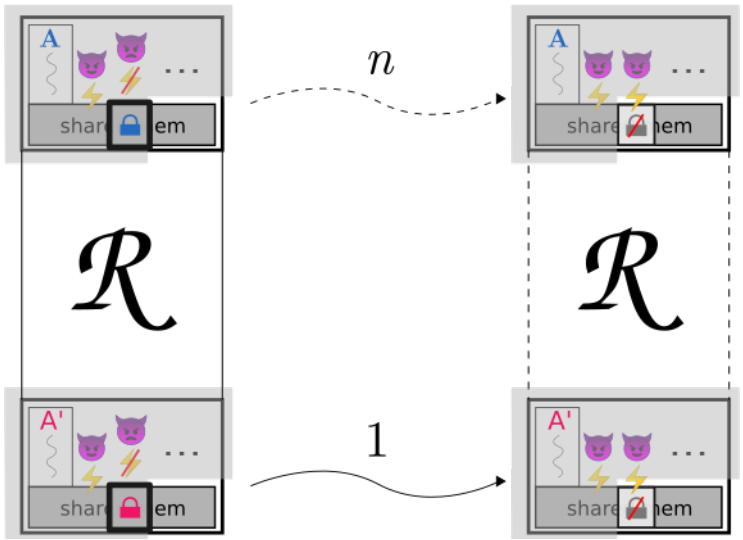
$\exists n\ .\ n =$ abs_steps A A'



Limitations and future work ideas

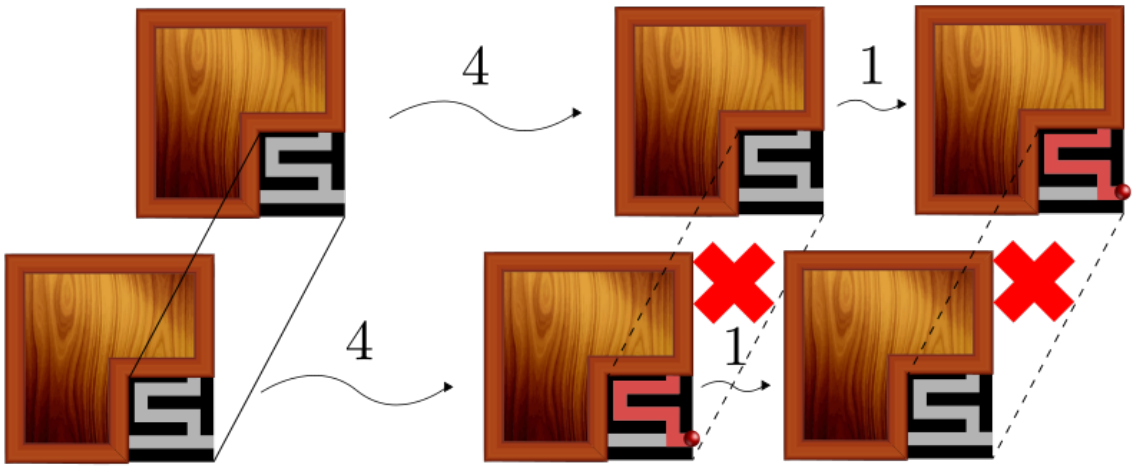
- Optimisations to non-observable shared memory?
- Can existing compilers be proven to satisfy it? CompCert?
  - small-step semantics, volatile R/W observable
  - simulation of target by source

$\exists n\ .\ n =$ abs_steps A A'



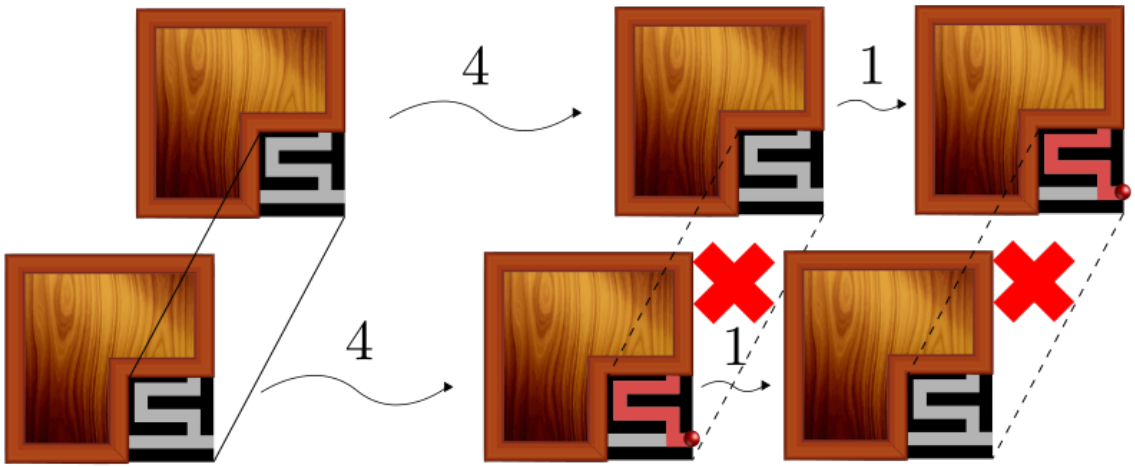
Limitations and future work ideas

- Optimisations to non-observable shared memory?
- Can existing compilers be proven to satisfy it? CompCert?
  - small-step semantics, volatile R/W observable
  - simulation of target by source
- Target models right for timing sensitivity?



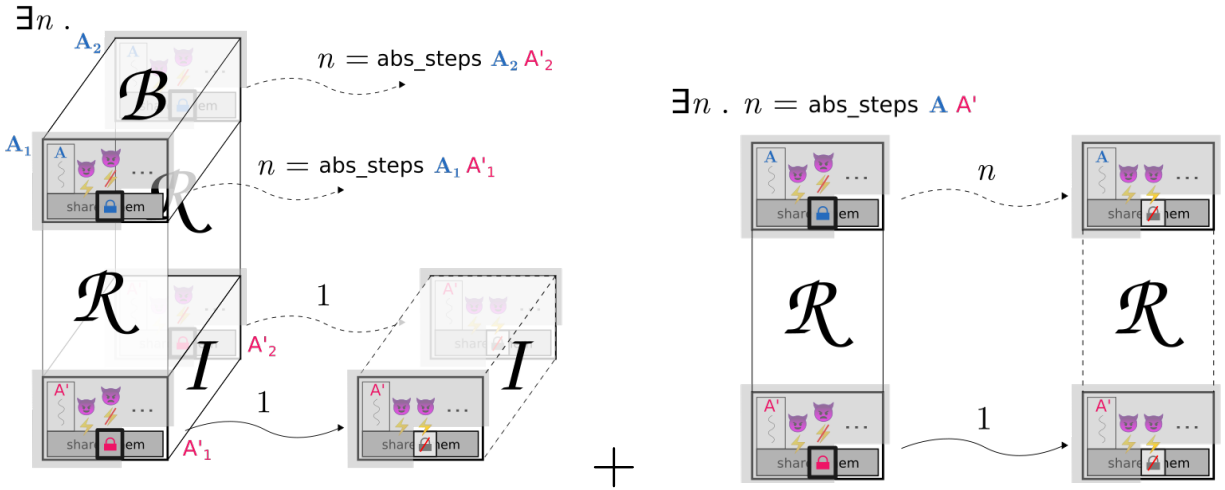
Limitations and future work ideas

- Optimisations to non-observable shared memory?
- Can existing compilers be proven to satisfy it? CompCert?
  - small-step semantics, volatile R/W observable
  - simulation of target by source
- Target models right for timing sensitivity? AVR, wasm?



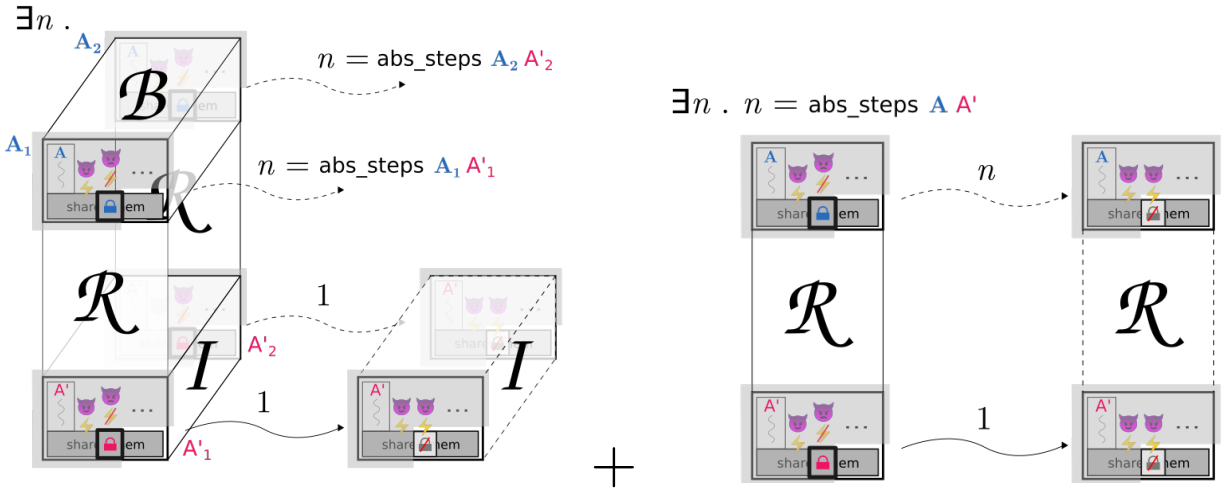
Limitations and future work ideas

- Optimisations to non-observable shared memory?
- Can existing compilers be proven to satisfy it? CompCert?
  - small-step semantics, volatile R/W observable
  - simulation of target by source
- Target models right for timing sensitivity? AVR, wasm?
- Branching on H values? Exercise with richer $\mathcal{B}$, $\mathcal{I}$:



Limitations and future work ideas

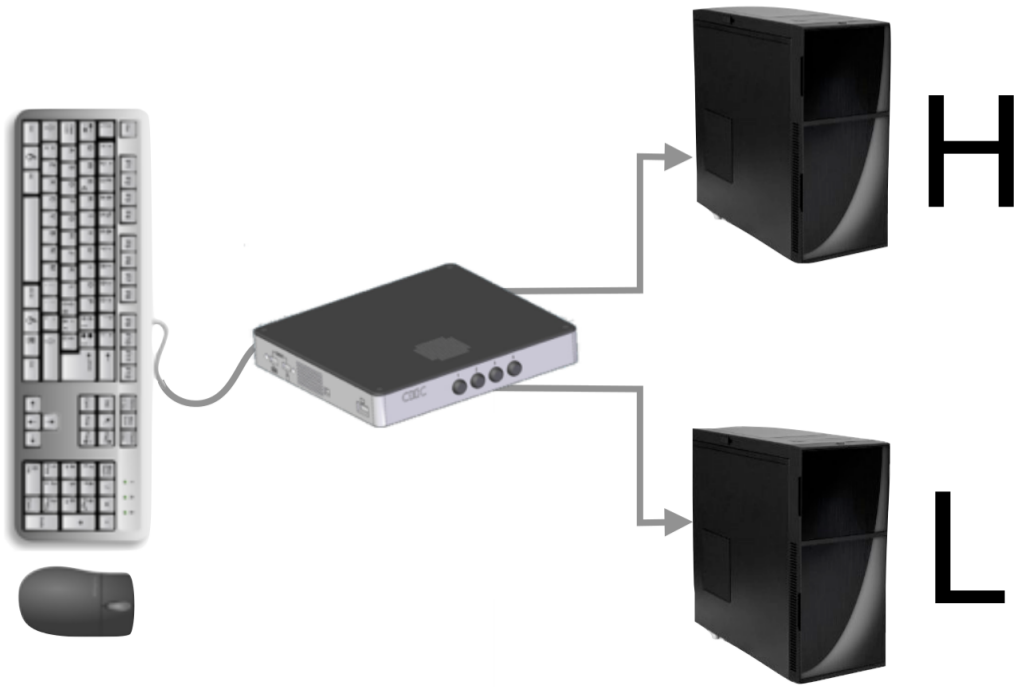
- Optimisations to non-observable shared memory?
- Can existing compilers be proven to satisfy it? CompCert?
  - small-step semantics, volatile R/W observable
  - simulation of target by source
- Target models right for timing sensitivity? AVR, wasm?
- Branching on H values? Exercise with richer $\mathcal{B}$, $\mathcal{I}$:



Thank you! Q & A

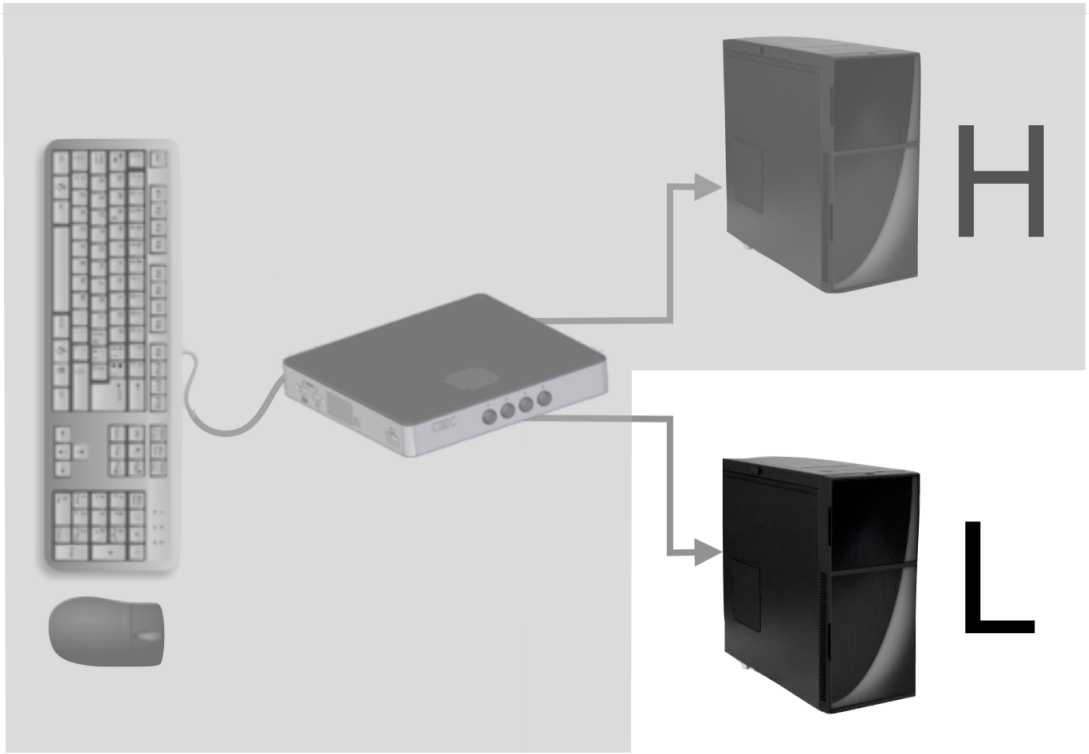
Appendix: Co-habiting attacker?

CDDC case study, again.



Appendix: Co-habiting attacker?

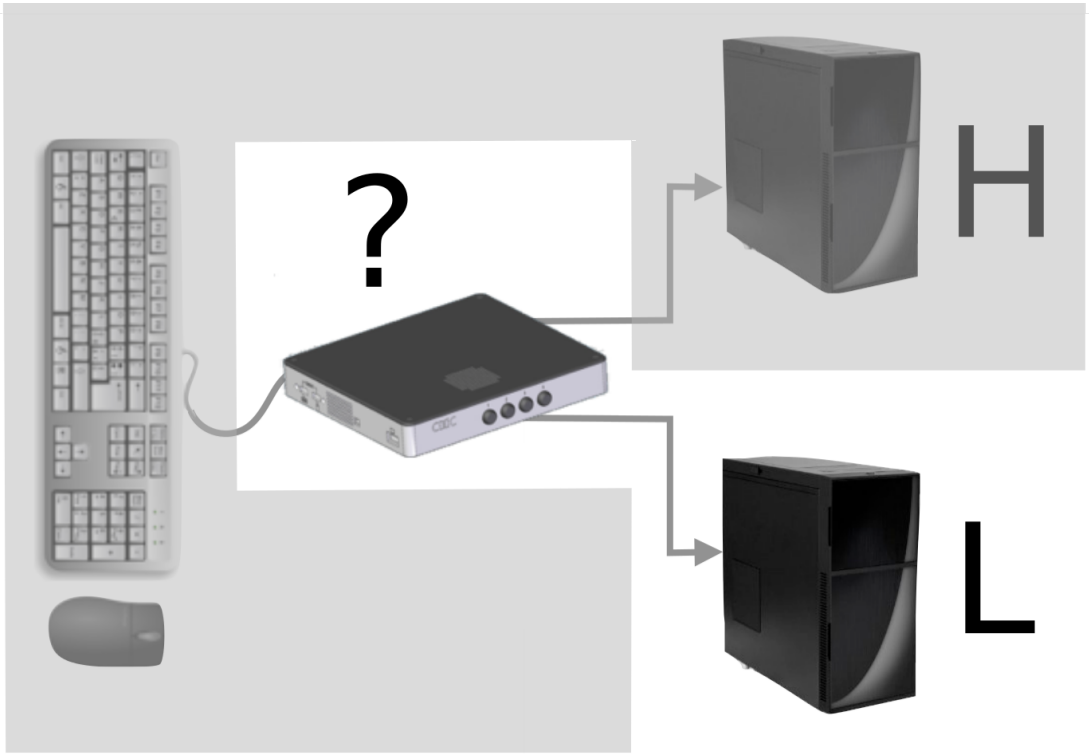
CDDC case study, again.



Untrusted sink: input device event stream out to Low machine.

Appendix: Co-habiting attacker?

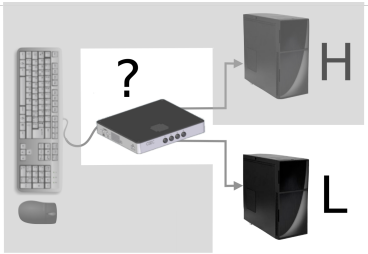
CDDC case study, again.



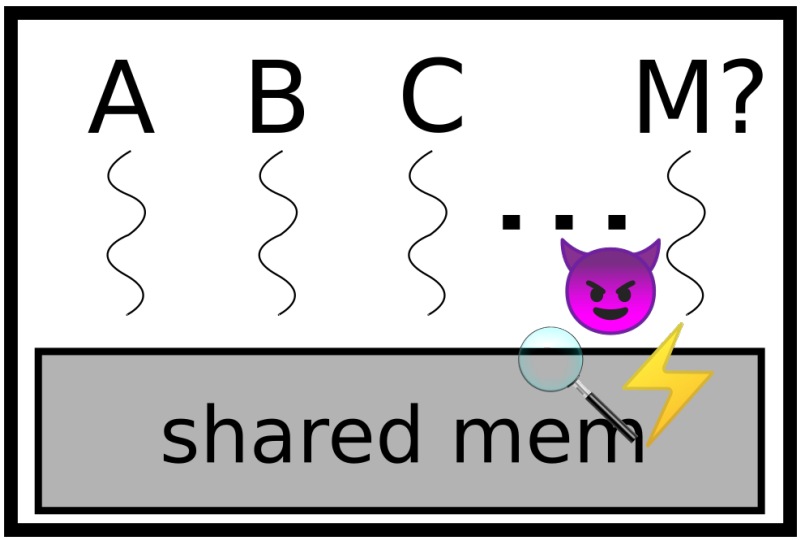
Untrusted sink: input device event stream out to Low machine.
What else can we afford to distrust?

Appendix: Co-habiting attacker?

CDDC case study, again.

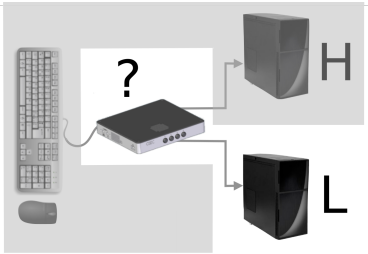


Hypothetically, a co-habiting "attacker"...?

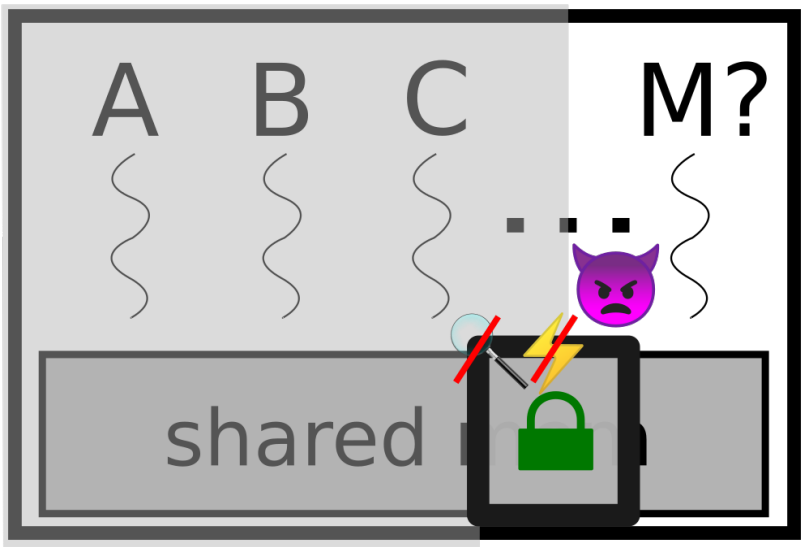


Appendix: Co-habiting attacker?

CDDC case study, again.



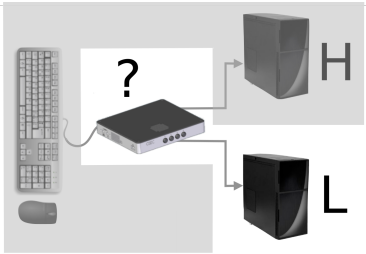
Hypothetically, a co-habiting "attacker"...



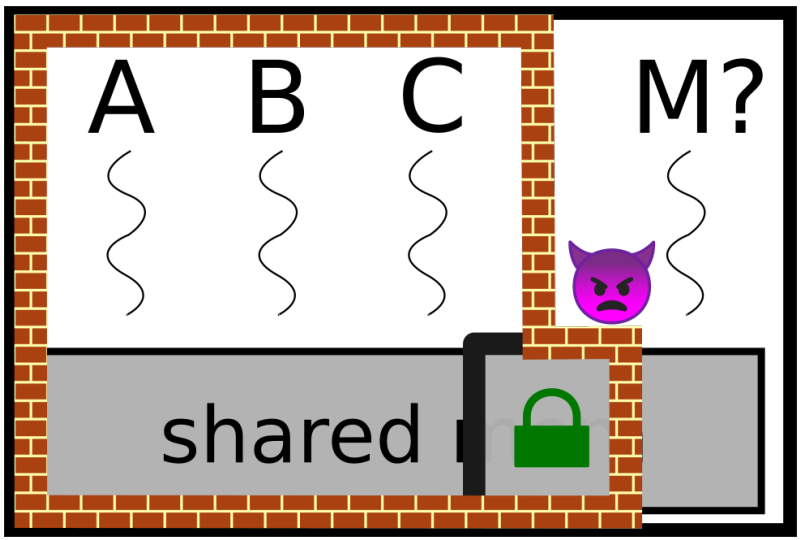
... if it in fact cannot see/touch High nor locked part of state.

Appendix: Co-habiting attacker?

CDDC case study, again.



Hypothetically, a co-habiting "attacker"...



... if it in fact cannot see/touch High nor locked part of state.
This may be reasonable in, e.g. a separation kernel environment.

Appendix: "Simpler" refinement

No H-branching ("L-shaped") obligation:

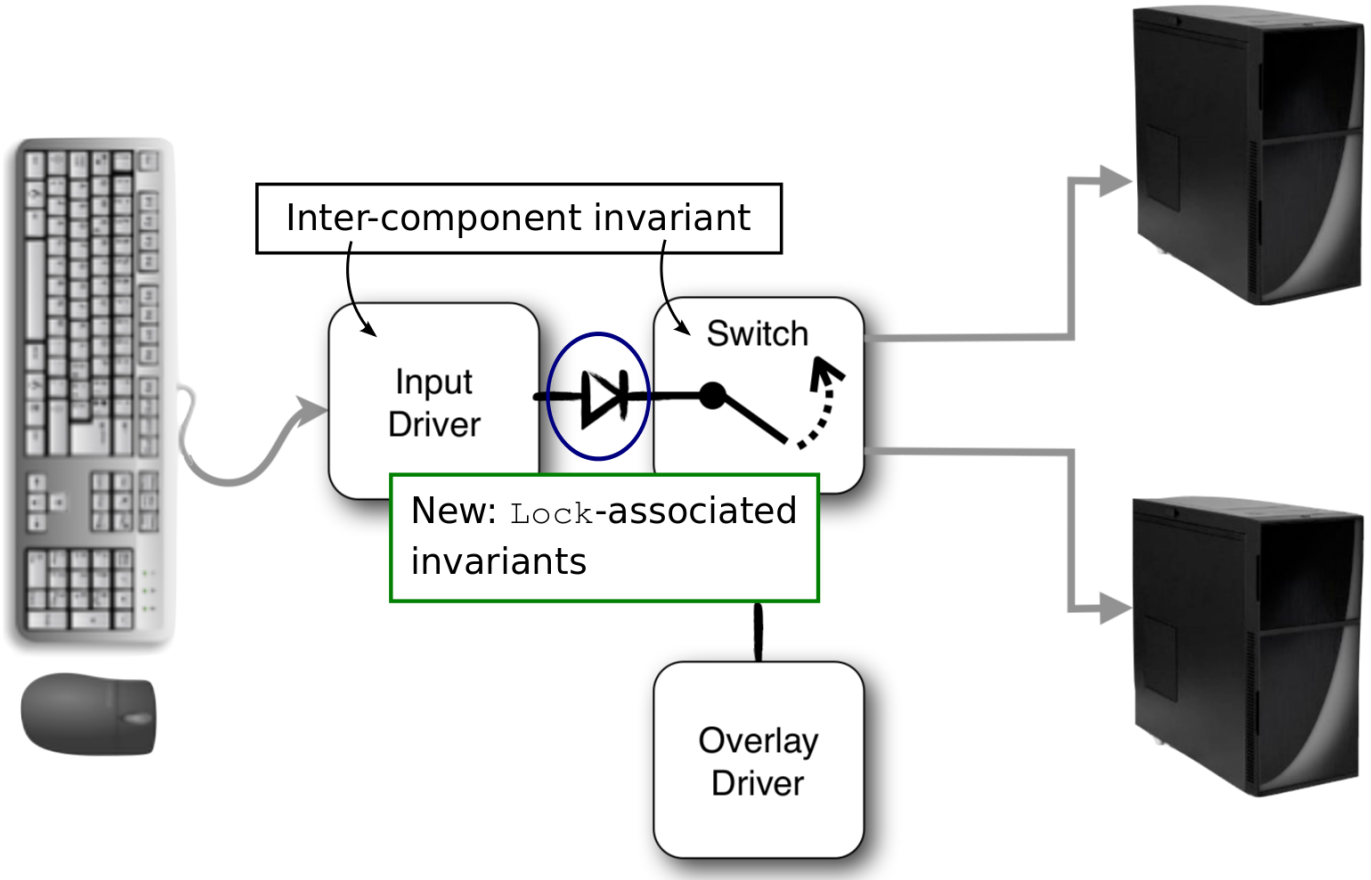
```
definition
  simpler_refinement_safe
where
  "simpler_refinement_safe R_A R P abs_steps ≡
  ∀c_1A mds_A mem_1A c_2A mem_2A c_1C mds_C mem_1C c_2C mem_2C. (⟨c_1A,mds_A,mem_1A⟩_A,⟨c_2A,mds_A,mem_2A⟩_A) ∈ R_A ∧
      (⟨c_1A,mds_A,mem_1A⟩_A,⟨c_1C, mds_C, mem_1C⟩_C) ∈ R ∧ (⟨c_2A,mds_A,mem_2A⟩_A,⟨c_2C, mds_C, mem_2C⟩_C) ∈ R ∧
       (⟨c_1C, mds_C, mem_1C⟩_C, ⟨c_2C, mds_C, mem_2C⟩_C) ∈ P  ⟶
           (stops_C ⟨c_1C, mds_C, mem_1C⟩_C = stops_C ⟨c_2C, mds_C, mem_2C⟩_C) ∧
           (abs_steps ⟨c_1A,mds_A,mem_1A⟩_A ⟨c_1C, mds_C, mem_1C⟩_C = abs_steps ⟨c_2A,mds_A,mem_2A⟩_A ⟨c_2C, mds_C, mem_2C⟩_C) ∧
           (∀mds_1C' mds_2C' mem_1C' mem_2C' c_1C' c_2C'. ⟨c_1C, mds_C, mem_1C⟩_C ⇝_C ⟨c_1C', mds_1C', mem_1C'⟩_C  ∧
                                                  ⟨c_2C, mds_C, mem_2C⟩_C ⇝_C ⟨c_2C', mds_2C', mem_2C'⟩_C ⟶
                                                  (⟨c_1C', mds_1C', mem_1C'⟩_C, ⟨c_2C', mds_2C', mem_2C'⟩_C) ∈ P ∧
                                                  mds_1C' = mds_2C')"
```

Provisos and simulation relation:

```
definition
  secure_refinement_simpler
where
  "secure_refinement_simpler R_A R P abs_steps ≡
  closed_others R ∧
  preserves_modes_mem R ∧
  new_vars_private R ∧
  simpler_refinement_safe R_A R P abs_steps ∧
  conc.closed_glob_consistent P ∧
  (∀ c_1A mds_A mem_1A c_1C mds_C mem_1C.
   (⟨ c_1A, mds_A, mem_1A ⟩_A, ⟨ c_1C, mds_C, mem_1C ⟩_C) ∈ R ⟶
    (∀ c_1C' mds_C' mem_1C'. ⟨ c_1C, mds_C, mem_1C ⟩_C ⇝_C ⟨ c_1C', mds_C', mem_1C' ⟩_C ⟶
     (∃ c_1A' mds_A' mem_1A'. abs.neval ⟨ c_1A, mds_A, mem_1A ⟩_A (abs_steps ⟨c_1A,mds_A,mem_1A⟩_A ⟨c_1C,mds_C,mem_1C⟩_C) ⟨ c_1A', mds_A', mem_1A' ⟩_A ∧
                   (⟨ c_1A', mds_A', mem_1A' ⟩_A, ⟨ c_1C', mds_C', mem_1C' ⟩_C) ∈ R)))"
```

(See: `https://www.isa-afp.org/entries/Dependent_SIFUM_Refinement.html`)

Appendix: CDDC 3-component architecture verification



Invariant on integrity of Switch's internal state w.r.t. indicator.
To appear: EuroS&P'18.