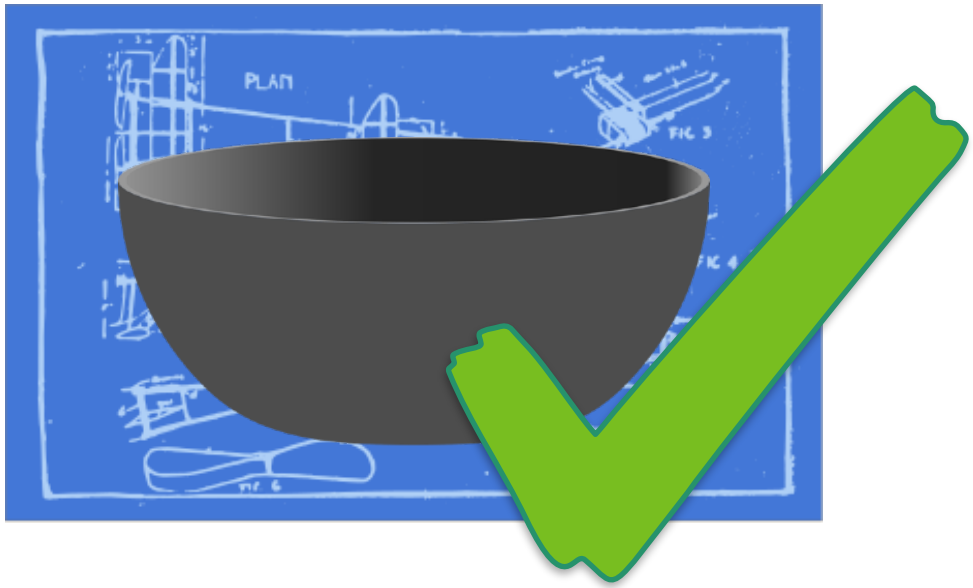
Verifying that a compiler preserves concurrent value-dependent information-flow security

Robert Sison (UNSW Sydney, Data61) and **Toby Murray** (University of Melbourne)

September 2019

www.data61.csiro.au

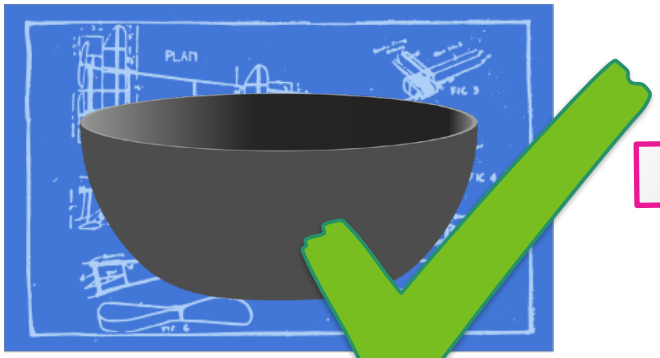
So you've proved your program *doesn't leak secrets*…



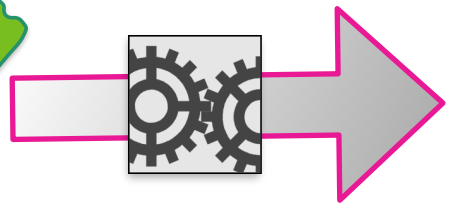
No leaks!

So you've proved your program *doesn't leak secrets*…

How do you know your compiler won't *introduce leaks*?

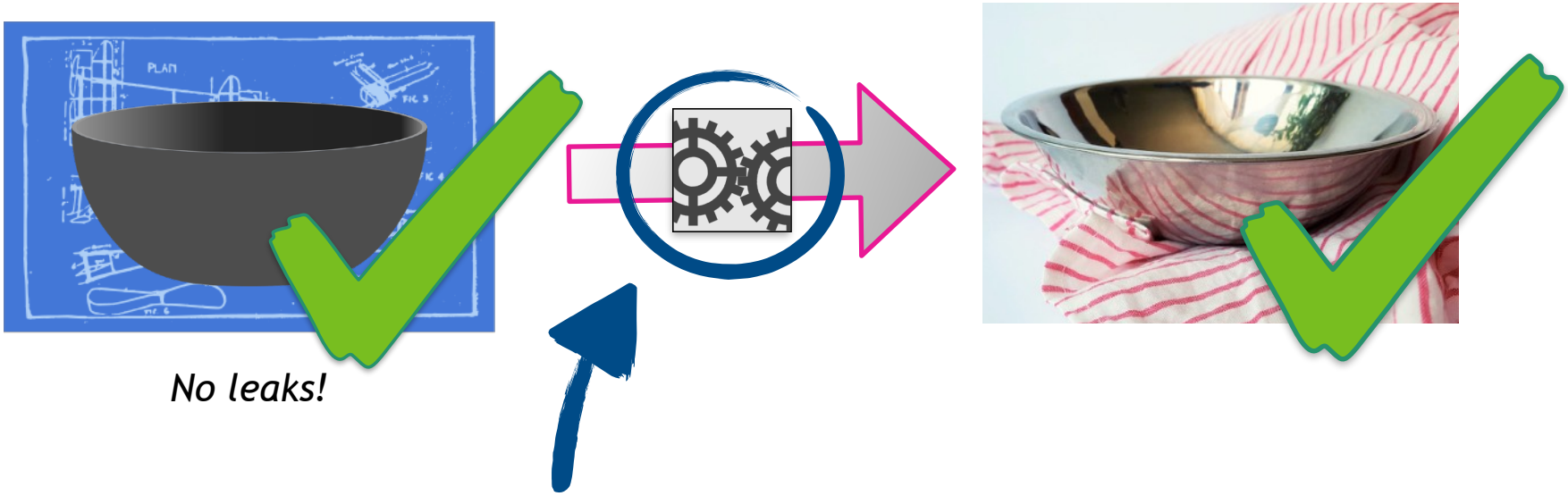


No leaks!



So you've proved your program *doesn't leak secrets*…

How do you know your compiler won't *introduce leaks*?



What if your compiler could be proved to *preserve* it?

So you've proved your program *doesn't leak secrets*…

What if your compiler could be proved to *preserve* it?

So you've proved your program *doesn't leak secrets*…

What if your compiler could be proved to *preserve* it?

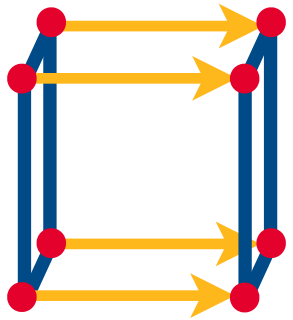
Here's how!

So you've proved your program *doesn't leak secrets*…

What if your compiler could be proved to *preserve* it?

Here's how!

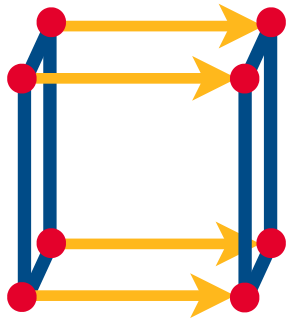
Using *confidentiality-preserving* refinement



So you've proved your program *doesn't leak secrets*…

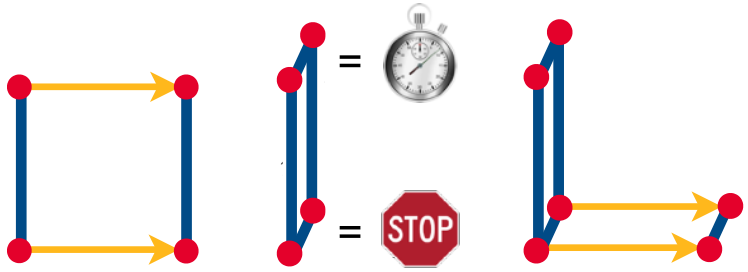
What if your compiler could be proved to *preserve* it?

Here's how!



Using *confidentiality-preserving* refinement

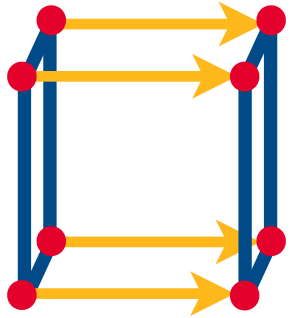
1. With a decomposition principle



So you've proved your program *doesn't leak secrets*…

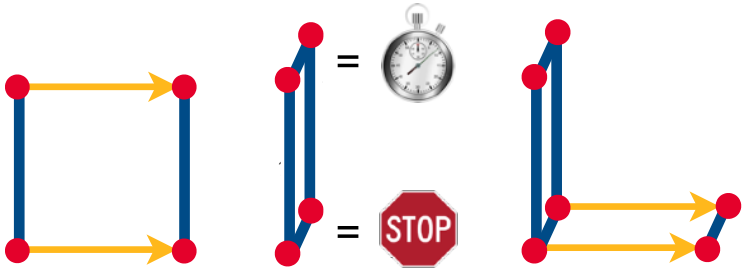
What if your compiler could be proved to *preserve* it?

Here's how!

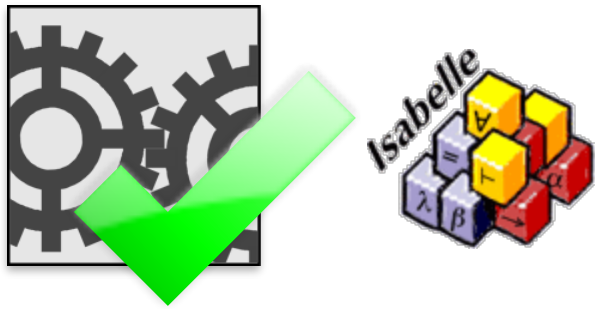


Using *confidentiality-preserving* refinement

1. With a decomposition principle



2. Applied to a compiler (in Isabelle/HOL)

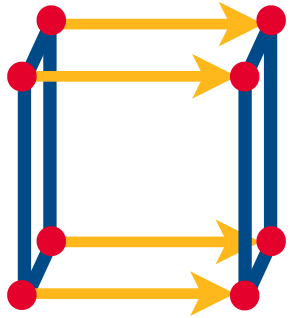


Our contributions

So you've proved your program *doesn't leak secrets*…

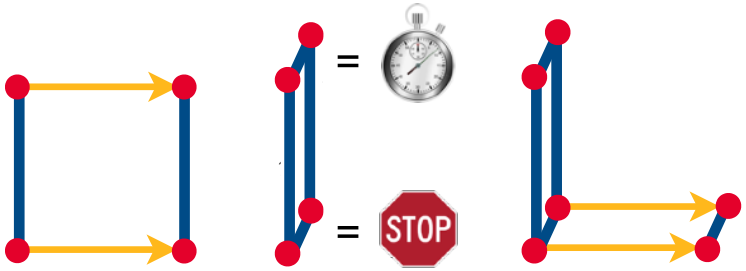
What if your compiler could be proved to *preserve* it?

Here's how!

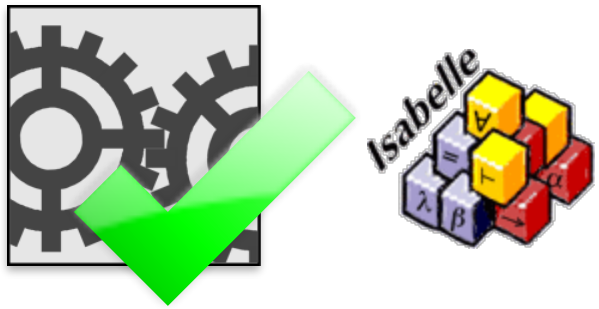


Using *confidentiality-preserving* refinement

1. With a decomposition principle



2. Applied to a compiler (in Isabelle/HOL)



Our contributions

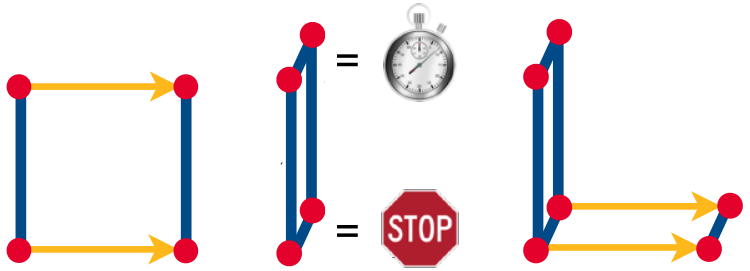
Goal

Prove a compiler *preserves proofs* of confidentiality — in an interactive theorem prover!

Results

1. Decomposition principle

for *confidentiality-preserving refinement*



(Technique)

2. Verified compiler

While-language to RISC-style assembly

(Proof-of-concept for technique)

(Formalisation: https://covern.org/itp19.html)

Our contributions

Goal

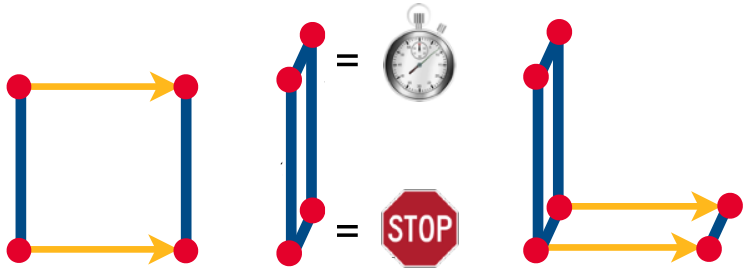
(Specifically…)

Prove a compiler *preserves proofs* of concurrent value-dependent information-flow security

Results

1. Decomposition principle

for *confidentiality-preserving refinement*



(Technique)

2. Verified compiler

While-language to RISC-style assembly

(Proof-of-concept for technique)

(Formalisation: `https://covern.org/itp19.html`)

Our contributions

Goal

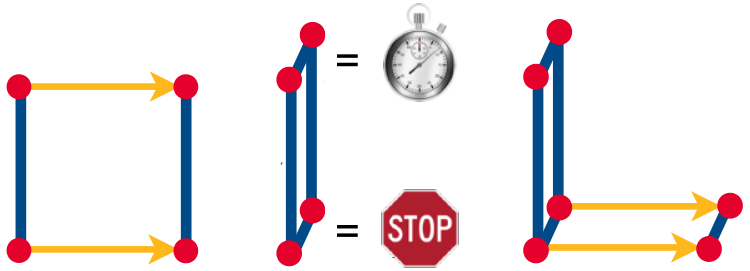
(Specifically…)

Prove a compiler *preserves proofs* of concurrent value-dependent information-flow security

Results

1. Decomposition principle

for *confidentiality-preserving refinement*



(Technique)

2. Verified compiler

While-language to RISC-style assembly

(Proof-of-concept for technique)

Impact

1st such proofs carried to assembly-level model by compiler

(Formalisation: `https://covern.org/itp19.html`)

Our contributions

Motivation

(Why all this?)

Goal

Prove a compiler *preserves proofs* of concurrent value-dependent information-flow security

Results

1. Decomposition principle

for *confidentiality-preserving refinement*

(Technique)

2. Verified compiler

While-language to RISC-style assembly

(Proof-of-concept for technique)

Impact

1st such proofs carried to assembly-level model by compiler

(Formalisation: https://covern.org/itp19.html)

Our contributions

Motivation

(Why it's hard!)

(Why all this?)

Goal

Prove a compiler *preserves proofs* of concurrent value-dependent information-flow security

Results

1. Decomposition principle

for *confidentiality-preserving refinement*

(Technique)

2. Verified compiler

While-language to RISC-style assembly

(Proof-of-concept for technique)

Impact

1st such proofs carried to assembly-level model by compiler

(Formalisation: https://covern.org/itp19.html)

Our contributions

Motivation

+

Background: Murray et al. (CSF'16)
(Why *still* hard?)

Goal

Prove a compiler *preserves proofs* of concurrent value-dependent information-flow security

Results

1. Decomposition principle

for *confidentiality-preserving refinement*

(Technique)

2. Verified compiler

While-language to RISC-style assembly

(Proof-of-concept for technique)

Impact

1st such proofs carried to assembly-level model by compiler

(Formalisation: https://covern.org/itp19.html)

Motivation

Confidentiality for modern software (CSF'16)

Concurrent value-dependent information-flow security

Motivation

Confidentiality for modern software (CSF'16)

Concurrent value-dependent **information-flow security**

Doesn't leak secrets
(storage channels)

Confidentiality

Motivation

Confidentiality for modern software (CSF'16)

Concurrent value-dependent **information-flow security**

Doesn't leak secrets
(storage channels)

Confidentiality

Motivation

Confidentiality for modern software (CSF'16)

1. Multiple moving parts
(well-synchronised)

Concurrent value-dependent information-flow security

Doesn't leak secrets
(storage channels)

Confidentiality

Motivation

Confidentiality for modern software (CSF'16)

1. Multiple moving parts
   (well-synchronised)

2. Mixed-sensitivity reuse
   (of devices, space, etc.)

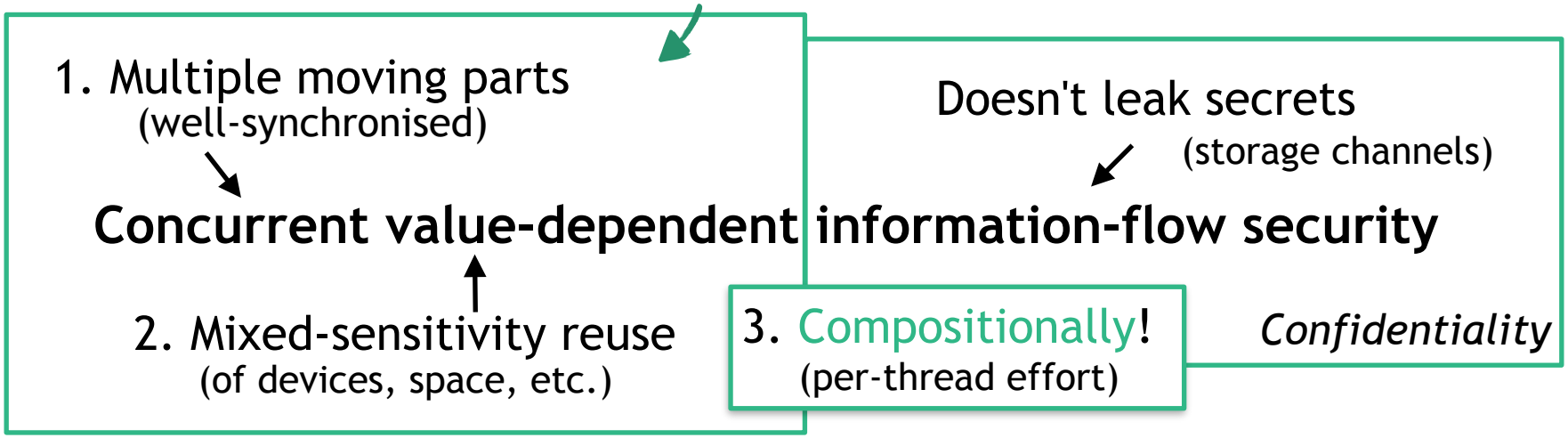
Concurrent value-dependent information-flow security

Doesn't leak secrets
(storage channels)

Confidentiality

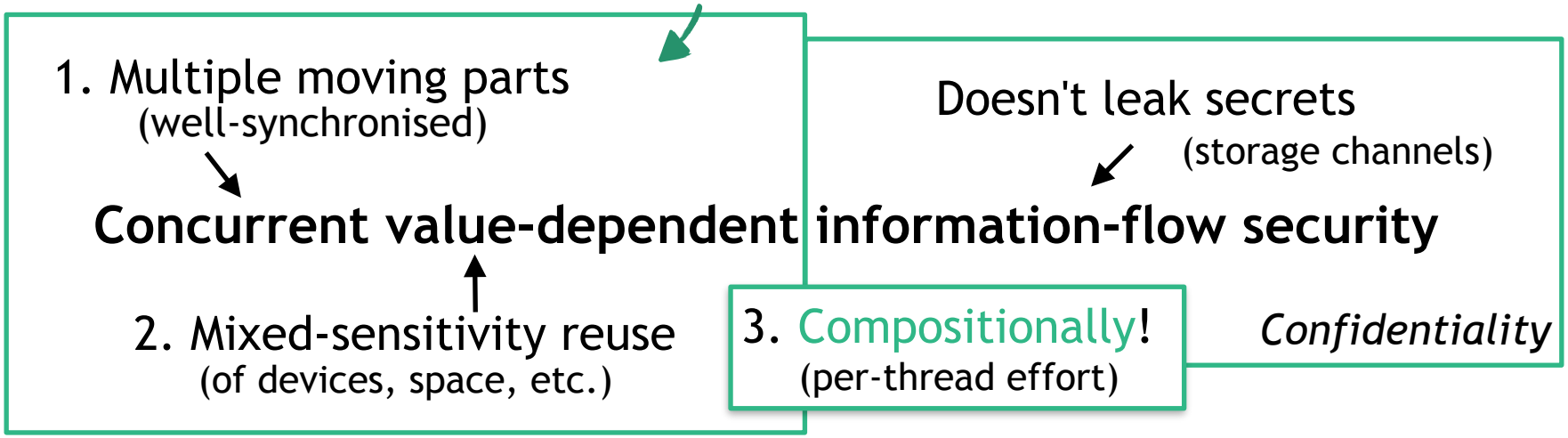
Motivation

Confidentiality for modern software (CSF'16)



Motivation

Confidentiality for modern software (CSF'16)



Beaumont et al.
(ACSAC'16)

Example

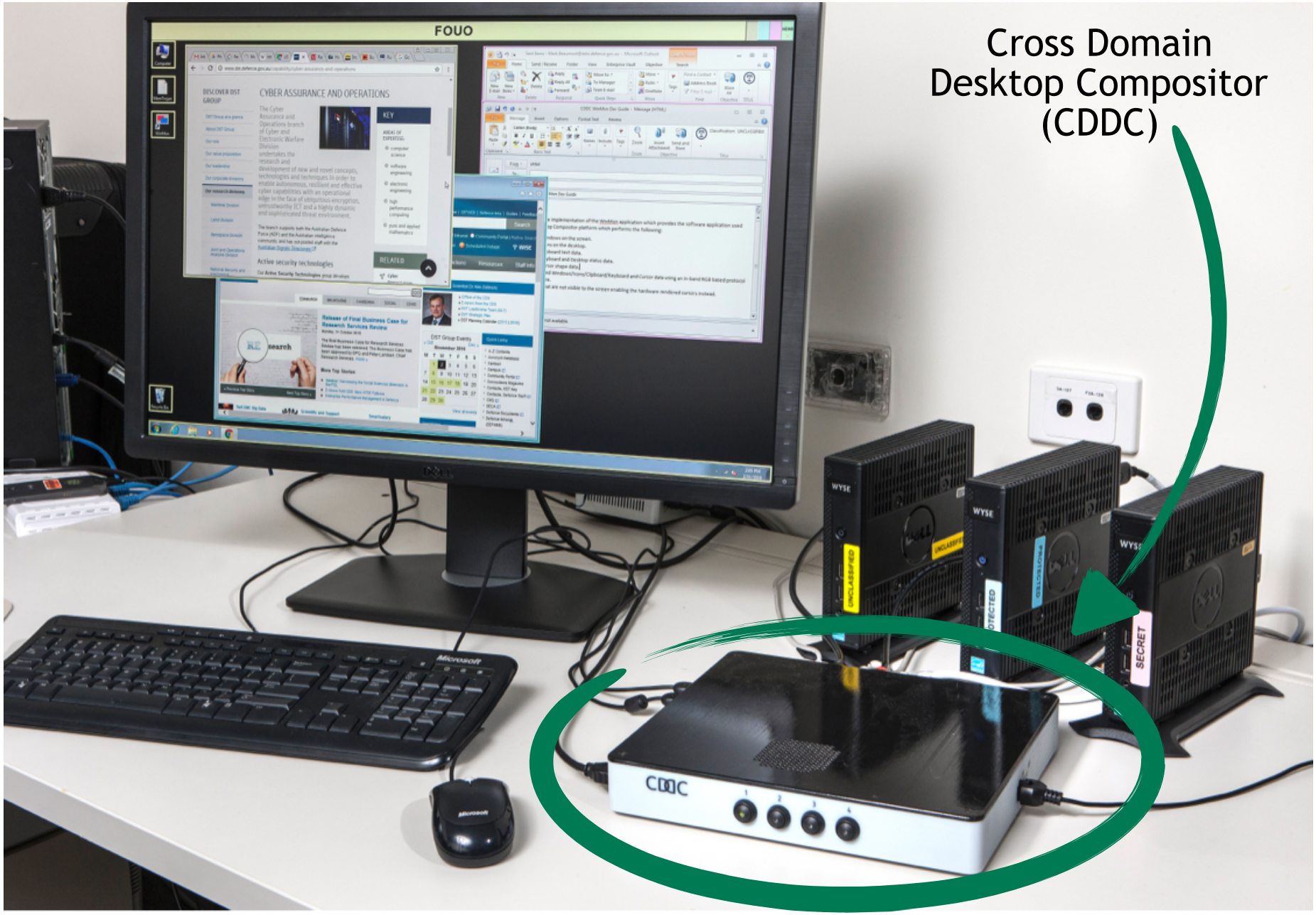
(DSTG + Data61 collaboration)

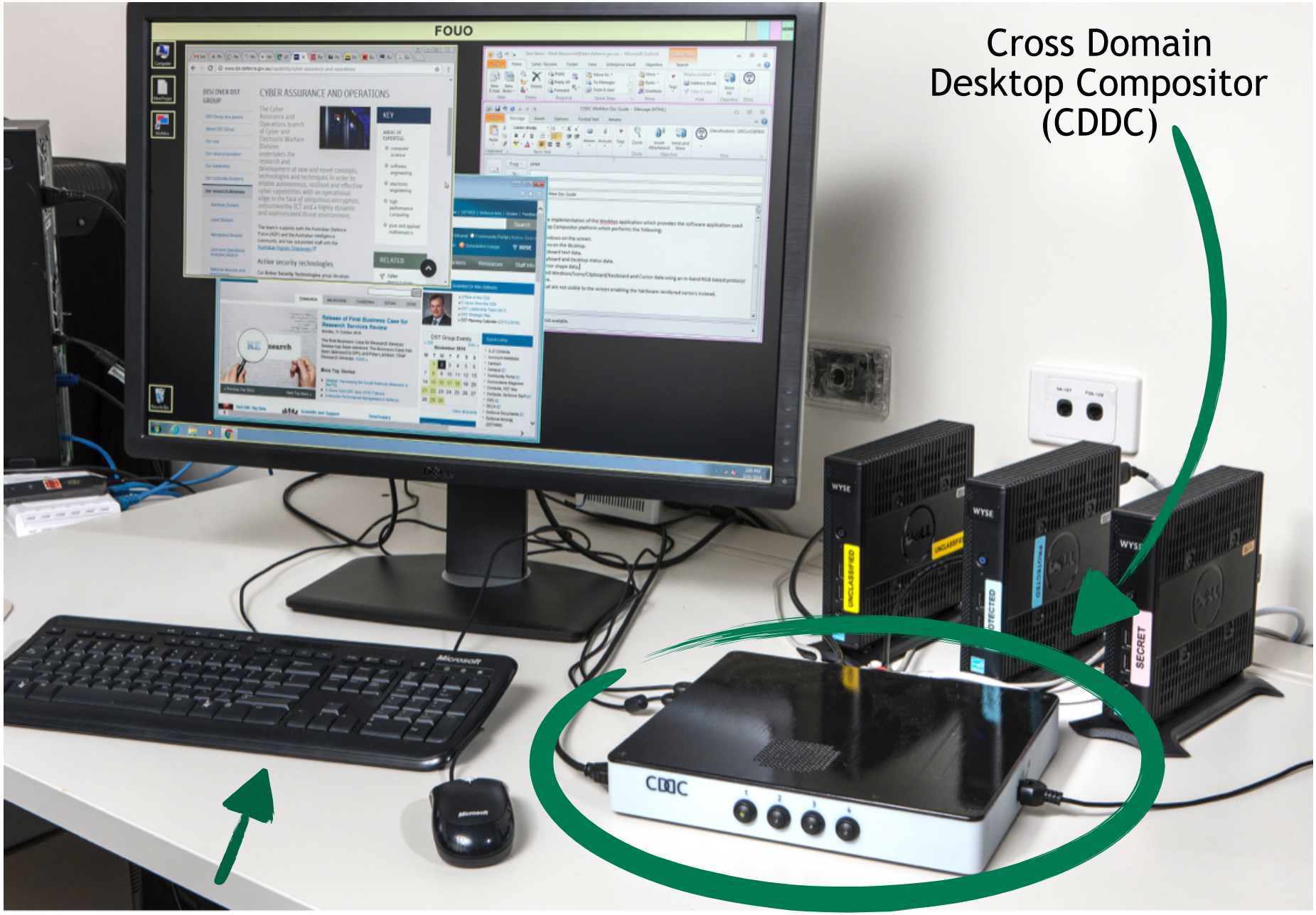
Motivation

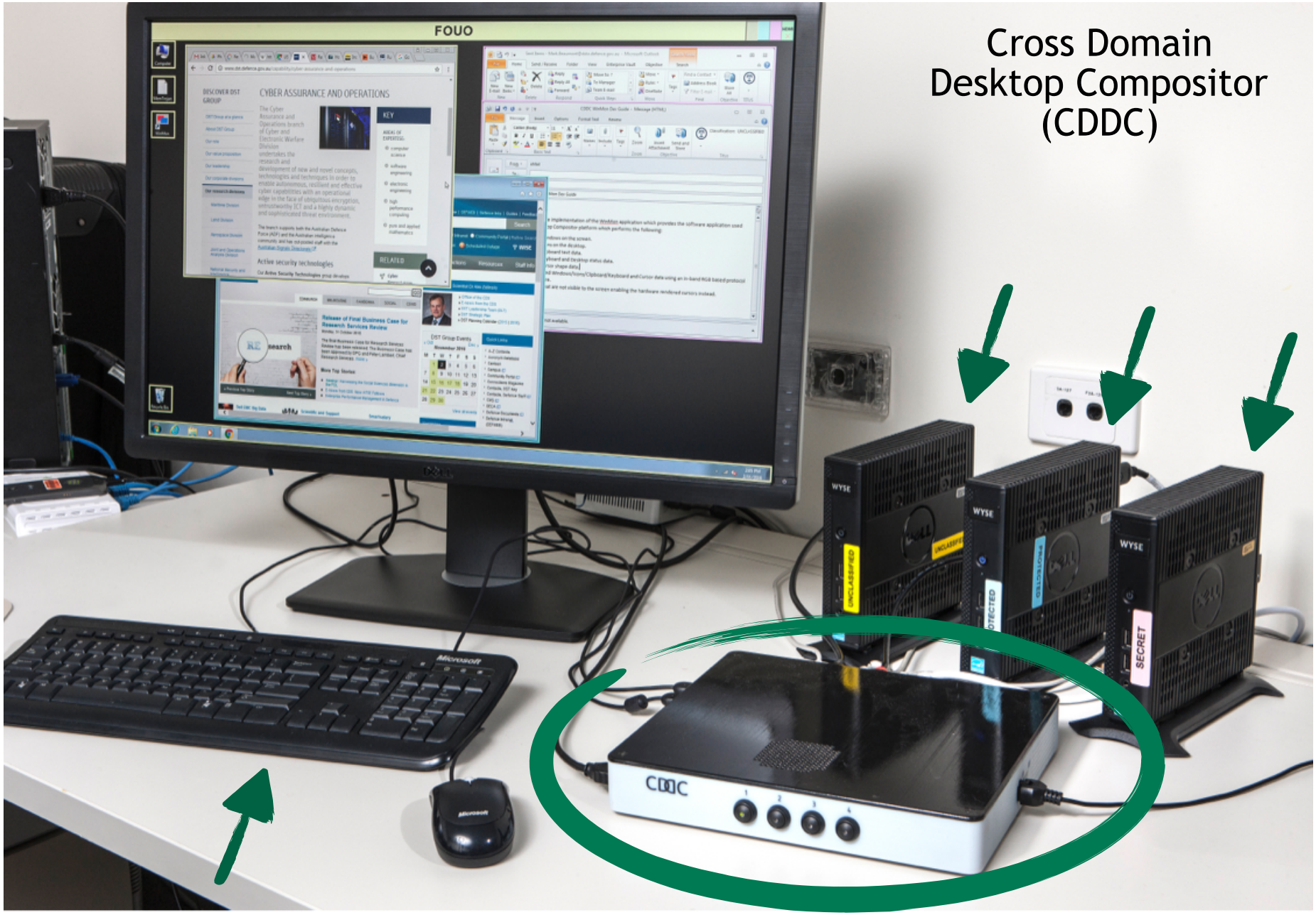
Confidentiality for modern software (CSF'16)

Motivation

Confidentiality for modern software (CSF'16)

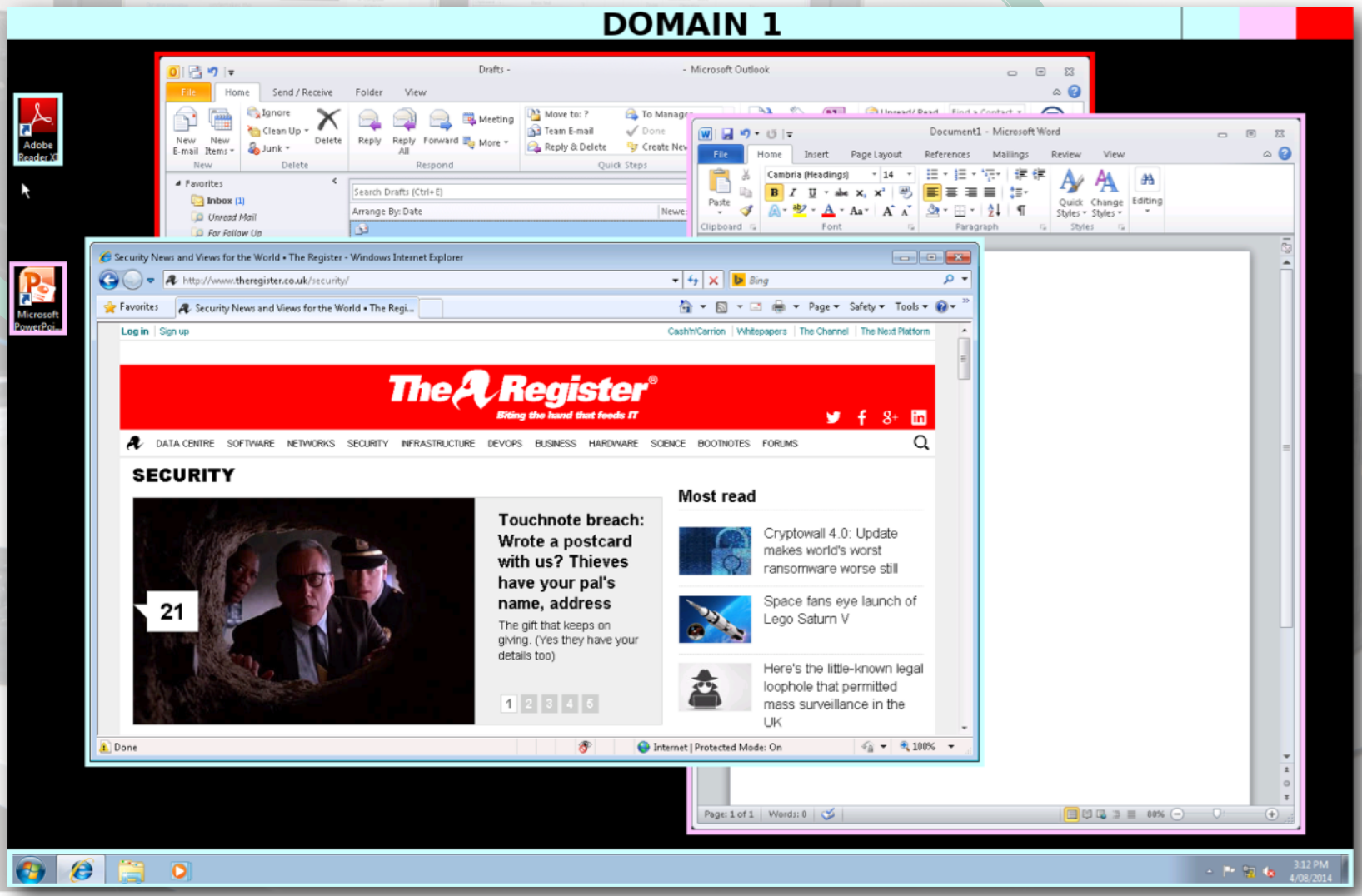

Cross Domain
Desktop Compositor
(CDDC)
FOUO
CYBER ASSURANCE AND OPERATIONS
CDDC
WYSE
SECRET


Cross Domain
Desktop Compositor
(CDDC)
FOUO
CYBER ASSURANCE AND OPERATIONS
Active security technologies
WYSE
UNCLASSIFIED
PROTECTED
SECRET
CDDC

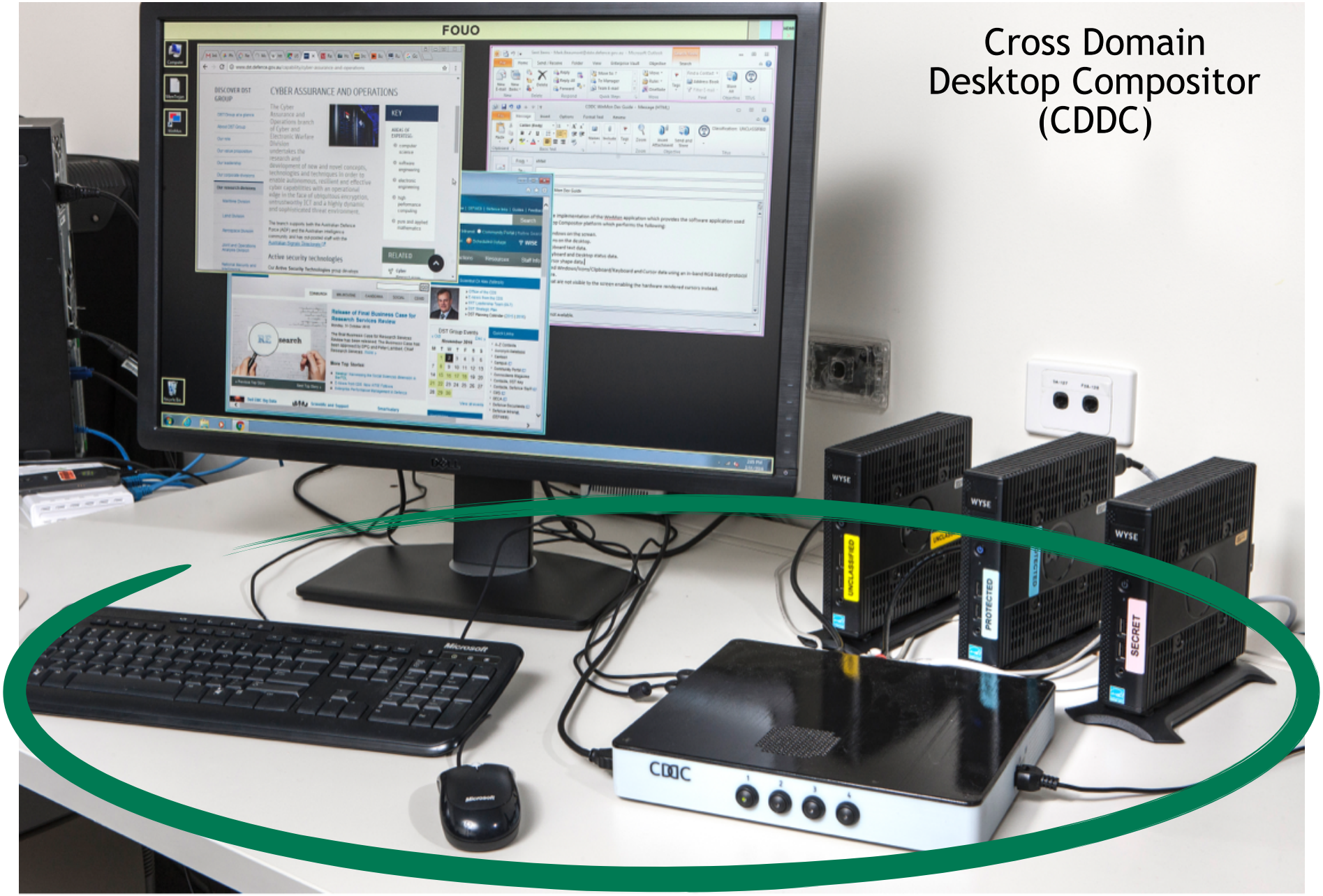

Cross Domain
Desktop Compositor
(CDDC)
FOUO
CYBER ASSURANCE AND OPERATIONS
WYSE
WYSE
WYSE
UNCLASSIFIED
PROTECTED
SECRET
CDDC


FOUO
Cross Domain
Desktop Compositor
(CDDC)
CYBER ASSURANCE AND OPERATIONS
DISCOVER DST GROUP
KEY
RELATED
Active security technologies
WYSE
UNCLASSIFIED
PROTECTED
SECRET
CDDC
Microsoft

Cross Domain Desktop Compositor (CDDC)



Cross Domain Desktop Compositor (CDDC)



Cross Domain Desktop Compositor (CDDC)

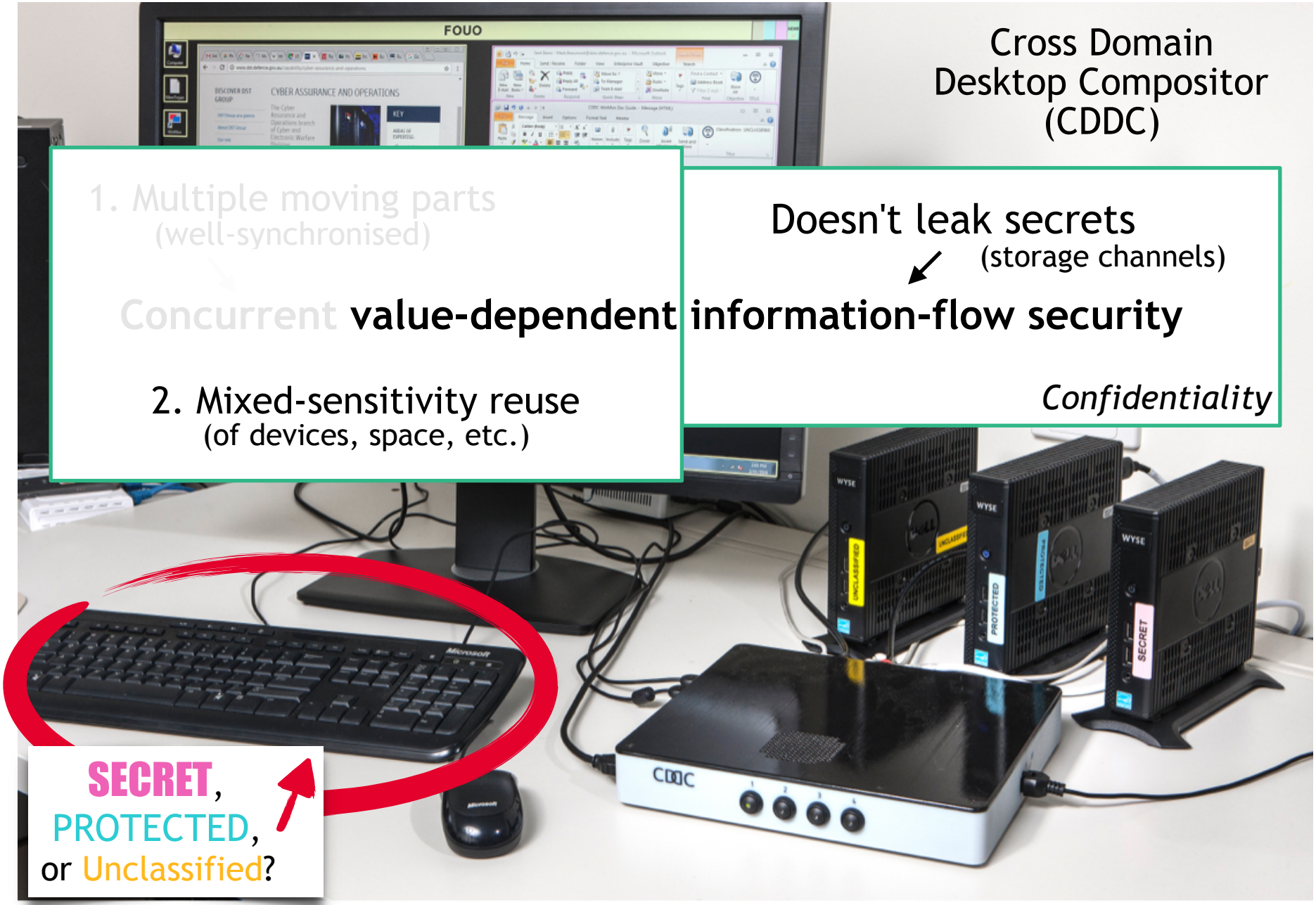
1. Multiple moving parts
(well-synchronised)

Concurrent **value-dependent information-flow security**

2. Mixed-sensitivity reuse
(of devices, space, etc.)

Doesn't leak secrets
(storage channels)

Confidentiality



SECRET, PROTECTED, or Unclassified?

Cross Domain Desktop Compositor (CDDC)

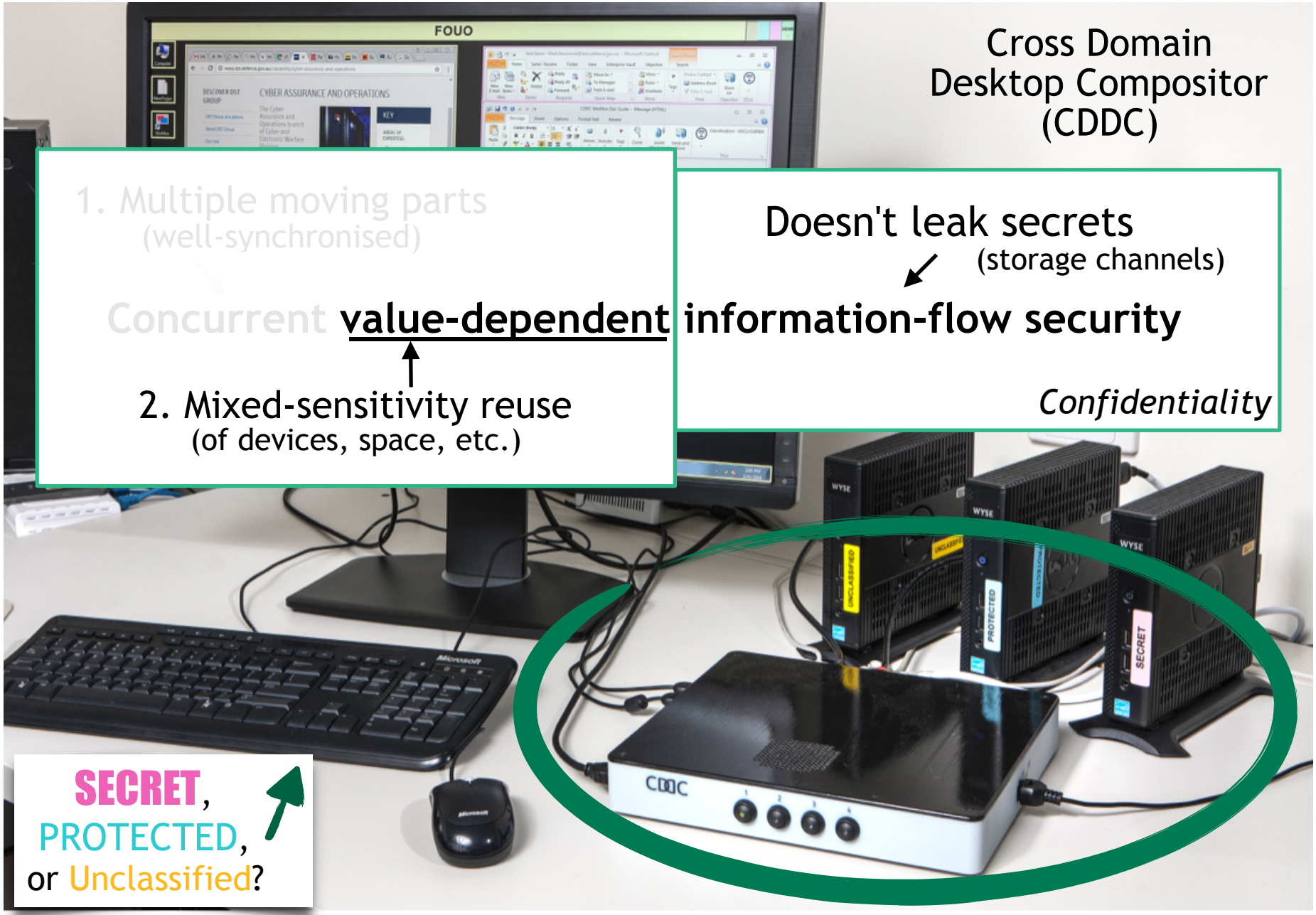
1. Multiple moving parts
(well-synchronised)

Concurrent **value-dependent** **information-flow security**

2. Mixed-sensitivity reuse
(of devices, space, etc.)

Doesn't leak secrets
(storage channels)

Confidentiality



SECRET, PROTECTED, or Unclassified?

Cross Domain Desktop Compositor (CDDC)

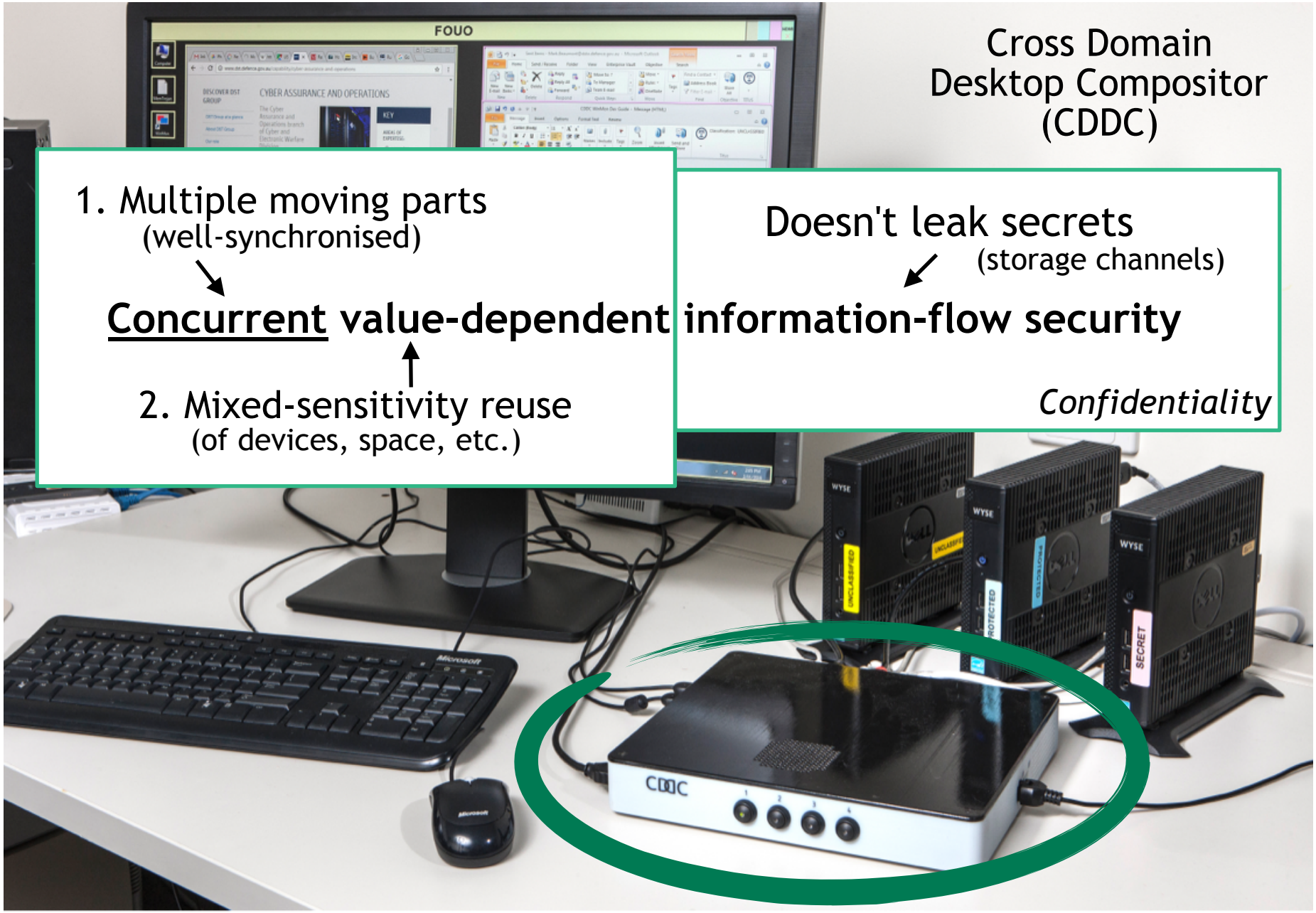
1. Multiple moving parts
(well-synchronised)

Concurrent value-dependent information-flow security

2. Mixed-sensitivity reuse
(of devices, space, etc.)

Doesn't leak secrets
(storage channels)

Confidentiality



Cross Domain Desktop Compositor (CDDC)

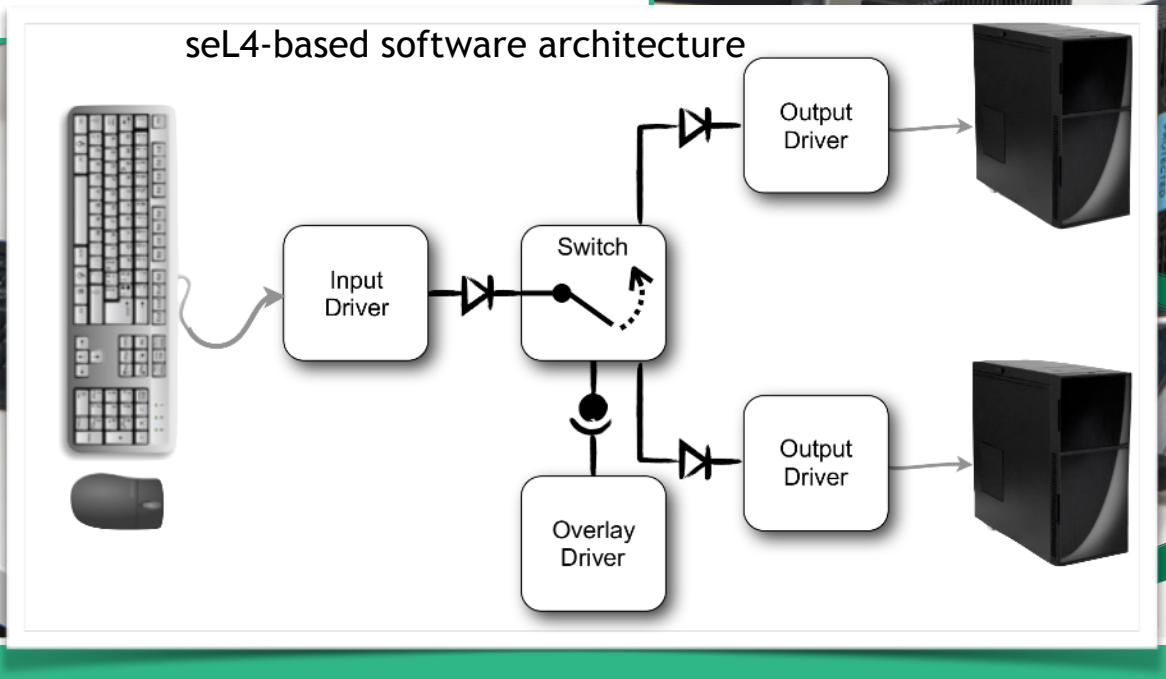
1. Multiple moving parts
(well-synchronised)

Concurrent value-dependent information-flow security

2. Mixed-sensitivity reuse
(of devices, space, etc.)

Doesn't leak secrets
(storage channels)

Confidentiality



Motivation

Confidentiality for modern software (CSF'16)

1. Multiple moving parts
   (well-synchronised)

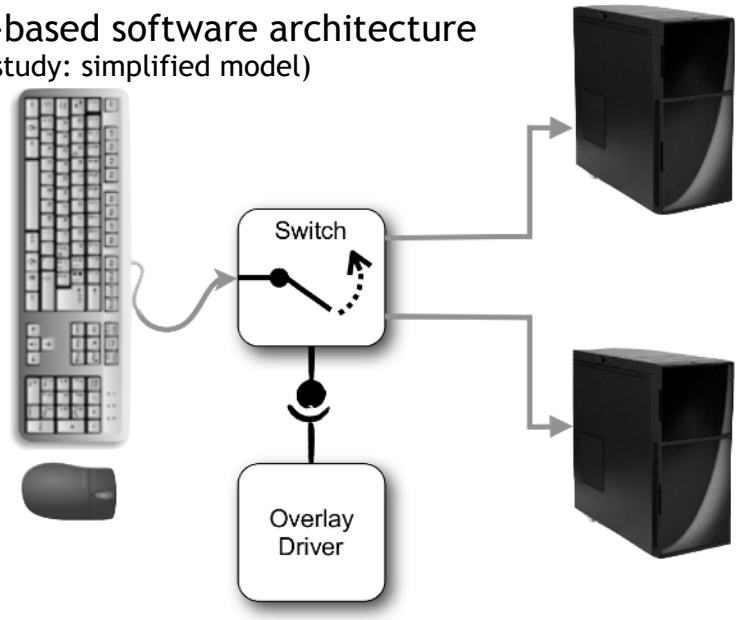
2. Mixed-sensitivity reuse
   (of devices, space, etc.)

Concurrent value-dependent information-flow security

Doesn't leak secrets
(storage channels)

Confidentiality

seL4-based software architecture
(Case study: simplified model)



Motivation

Confidentiality for modern software (CSF'16)

1. Multiple moving parts
   (well-synchronised)

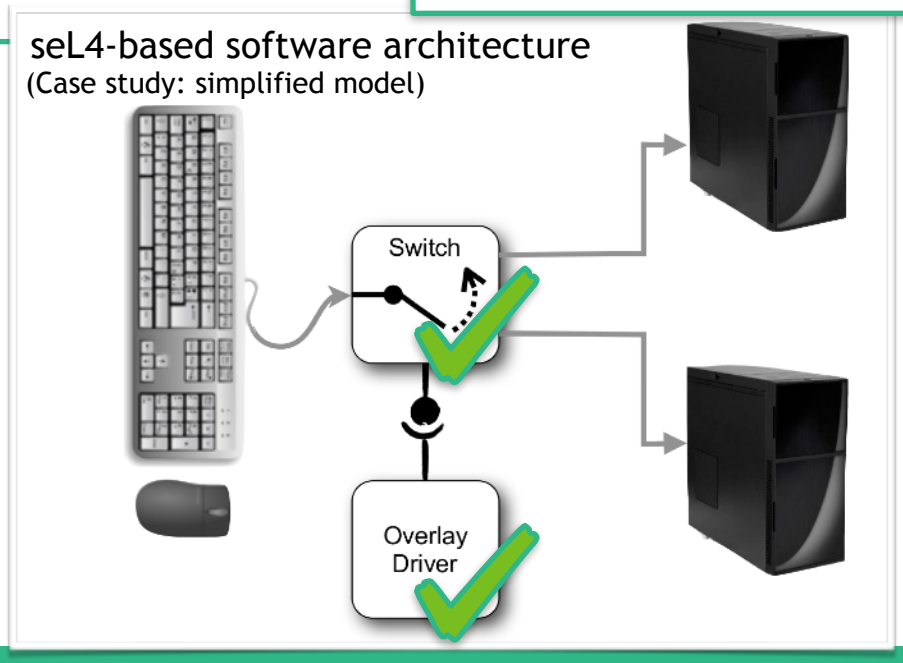
Doesn't leak secrets
(storage channels)

Concurrent value-dependent information-flow security

2. Mixed-sensitivity reuse
   (of devices, space, etc.)

3. Compositionally!
   (per-thread effort)

Confidentiality



Motivation

Confidentiality for modern software (CSF'16)

1. Multiple moving parts
(well-synchronised)

Concurrent value-dependent information-flow security

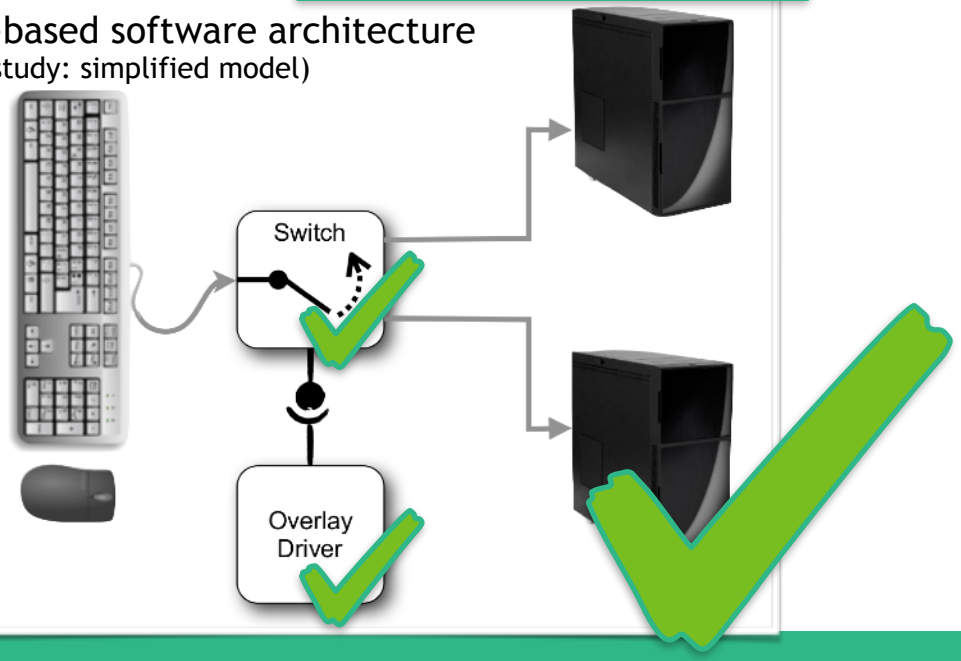
2. Mixed-sensitivity reuse
(of devices, space, etc.)

Doesn't leak secrets
(storage channels)

Confidentiality

3. Compositionally!
(per-thread effort)

seL4-based software architecture
(Case study: simplified model)



Motivation

Confidentiality for modern software (CSF'16)

1. Multiple moving parts
   (well-synchronised)

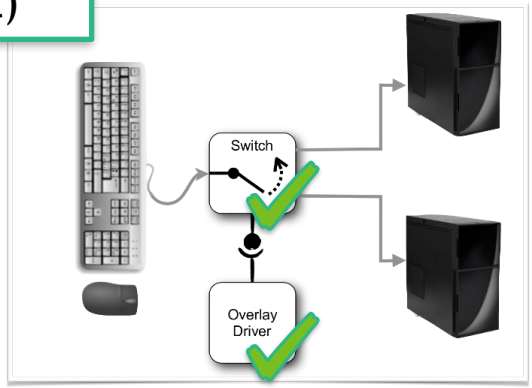
Doesn't leak secrets
(storage channels)

Concurrent value-dependent information-flow security

2. Mixed-sensitivity reuse
   (of devices, space, etc.)

3. Compositionally!
   (per-thread effort)

Confidentiality



Can a compiler preserve it?

Motivation

Why wouldn't a compiler preserve it? (CSF'16)

Concurrent value-dependent information-flow security

Motivation

Why wouldn't a compiler preserve it? (CSF'16)

Murray et al. CSF'16

control variable contents
(*sensitivity-switching*)

Concurrent value-dependent information-flow security

Some extra stuff to preserve
(not that hard)

Motivation

Why wouldn't a compiler preserve it? (CSF'16)

Murray et al. CSF'16

control variable contents
(*sensitivity-switching*)

Concurrent value-dependent information-flow security

Some extra stuff to preserve
(not that hard)

This particularly
makes it harder!

Motivation

Why wouldn't a compiler preserve it? (CSF'16)

Murray et al. CSF'16

control variable contents
(*sensitivity-switching*)

Concurrent value-dependent information-flow security

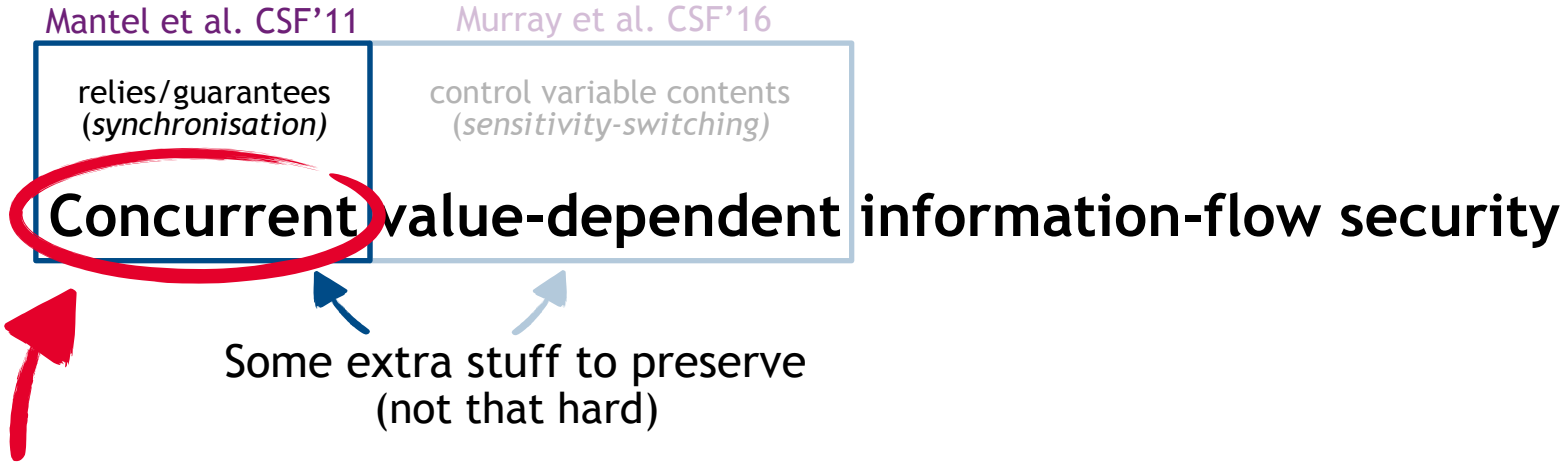
Some extra stuff to preserve
(not that hard)

This particularly
makes it harder!

Interference-resilience (tricky)

Motivation

Why wouldn't a compiler preserve it? (CSF'16)

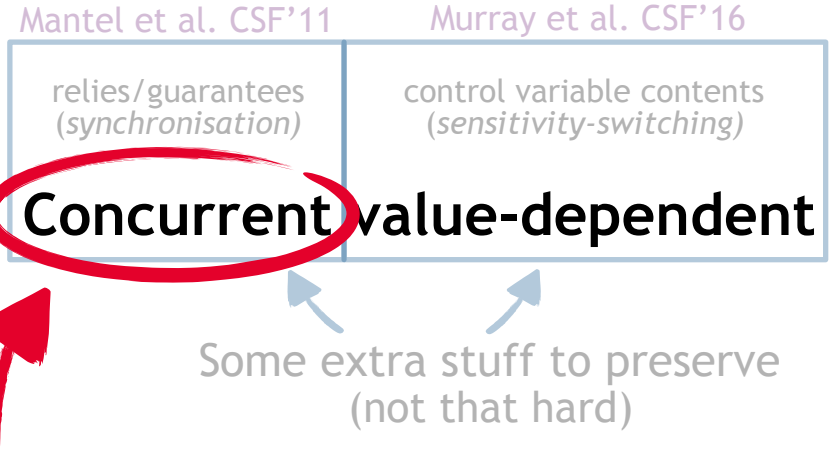


This particularly
makes it harder!

Interference-resilience (tricky)

Motivation

Why wouldn't a compiler preserve it? (CSF'16)



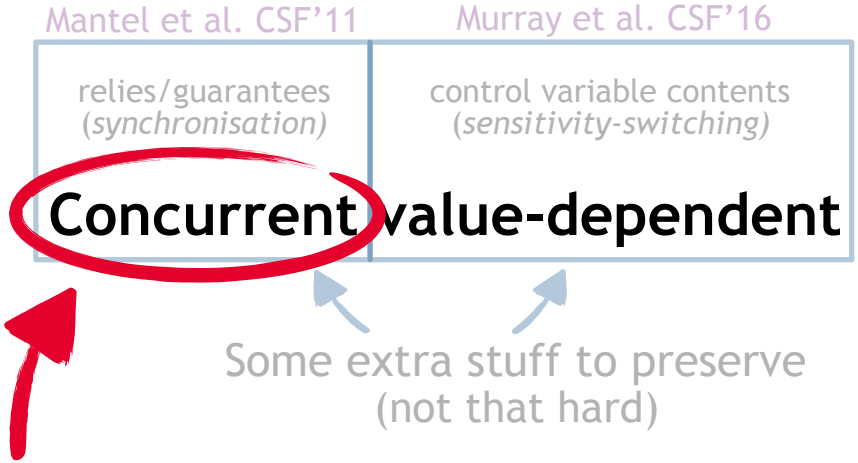
This particularly
makes it harder!

Interference-resilience (tricky)

+

Motivation

Why wouldn't a compiler preserve it? (CSF'16)



No storage leaks

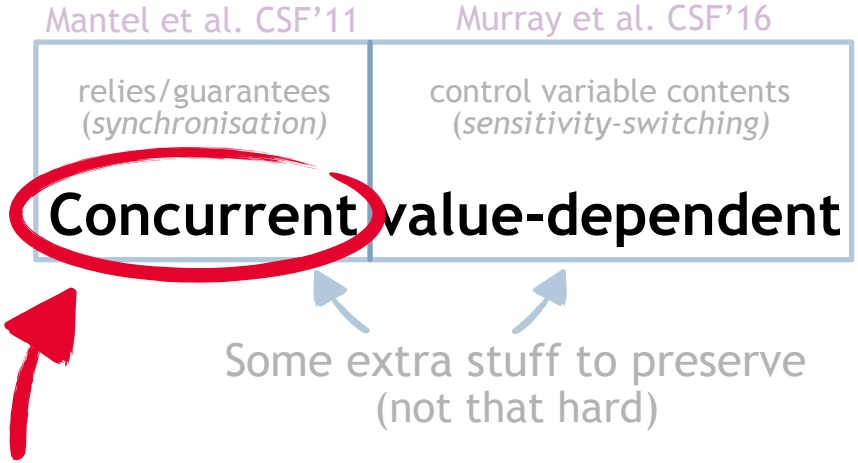
This particularly
makes it harder!

Interference-resilience (tricky)

+

Motivation

Why wouldn't a compiler preserve it? (CSF'16)



No storage leaks

This particularly
makes it harder!

Interference-resilience (tricky)

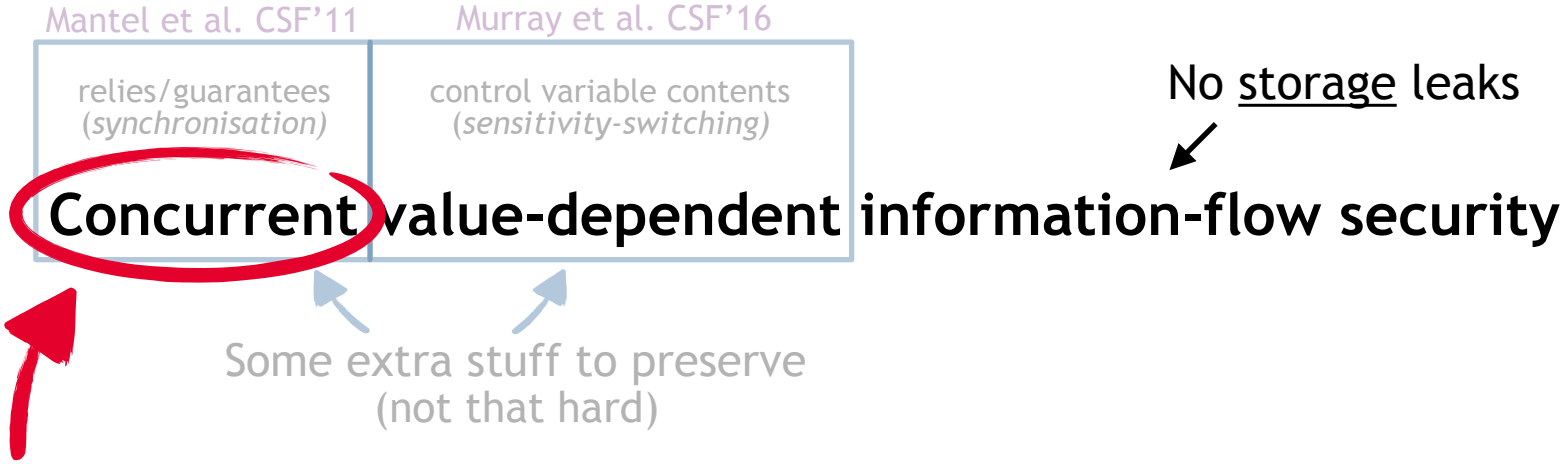
+

Each thread must prevent
(scheduler-relative)
timing leaks!

Volpano & Smith, CSFW'98

Motivation

Why wouldn't a compiler preserve it? (CSF'16)



This particularly makes it harder!

Interference-resilience (tricky)

+

Each thread must prevent (scheduler-relative) timing leaks!

Volpano & Smith, CSFW'98

Minimal example:

Program A

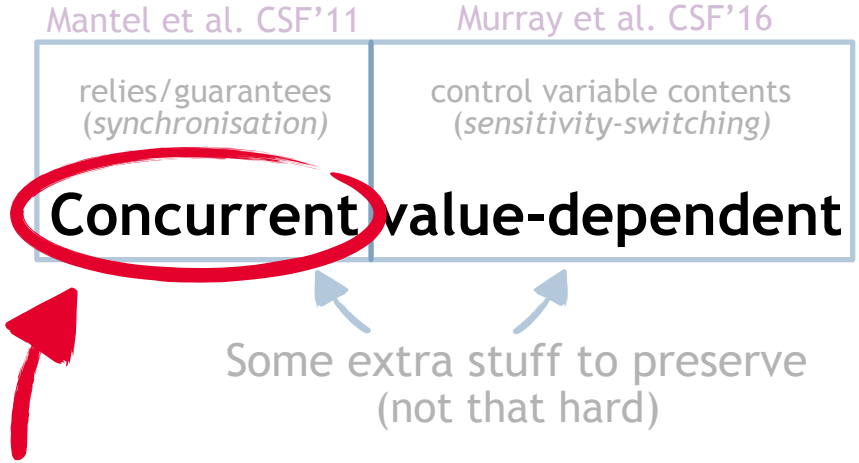
```
// Initially, v = 0
if (h) then
  skip
else
  skip; skip
fi
v := 1
```

Program B

```
l := v
```

Motivation

Why wouldn't a compiler preserve it? (CSF'16)



No storage leaks ✓

This particularly makes it harder!

Interference-resilience (tricky)

+

Each thread must prevent (scheduler-relative) timing leaks!

Volpano & Smith, CSFW'98

Minimal example:

Program A

```
// Initially, v = 0
if (h) then
  skip
else
  skip; skip
fi
v := 1
```

✓ h isn't assigned to anything

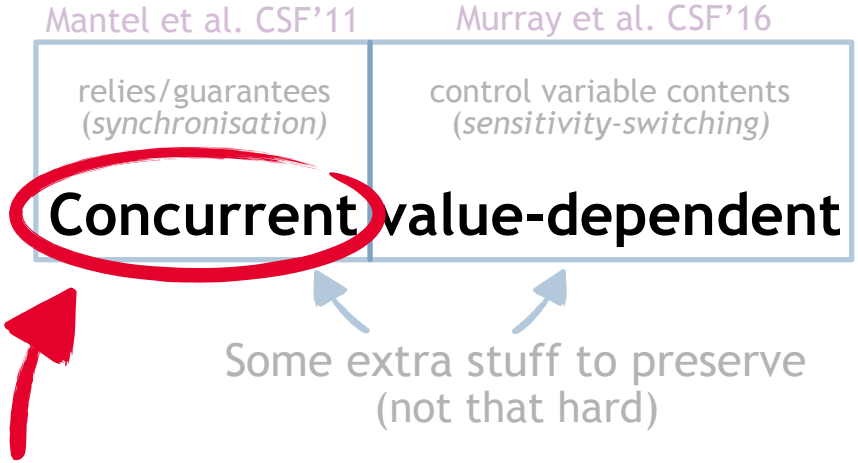
Program B

```
l := v
```

✓ h isn't even here!

Motivation

Why wouldn't a compiler preserve it? (CSF'16)



No storage leaks ✓

This particularly makes it harder!

Interference-resilience (tricky)

+

Each thread must prevent (scheduler-relative) timing leaks!

Volpano & Smith, CSFW'98

Minimal example:

Thread A		Thread B
// Initially, v = 0		
if (h) **then**		
skip		
else	‖	l := v
skip; skip		
fi		
v := 1		

Motivation

Why wouldn't a compiler preserve it? (CSF'16)

Mantel et al. CSF'11 | Murray et al. CSF'16

relies/guarantees (*synchronisation*) | control variable contents (*sensitivity-switching*)

Concurrent value-dependent information-flow security

No storage leaks ✓

Some extra stuff to preserve (not that hard)

This particularly makes it harder!

Interference-resilience (tricky)

+

Each thread must prevent (scheduler-relative) timing leaks!

Volpano & Smith, CSFW'98

Minimal example:

Thread A		Thread B

// Initially, v = 0
if (h) **then**
skip
else
skip; skip
fi
v := 1
	Timing leak of h	l := v

Motivation

Why wouldn't a compiler preserve it? (CSF'16)

Mantel et al. CSF'11 | Murray et al. CSF'16

relies/guarantees (*synchronisation*) | control variable contents (*sensitivity-switching*)

Concurrent value-dependent information-flow security

No storage leaks ✓

Some extra stuff to preserve (not that hard)

This particularly makes it harder!

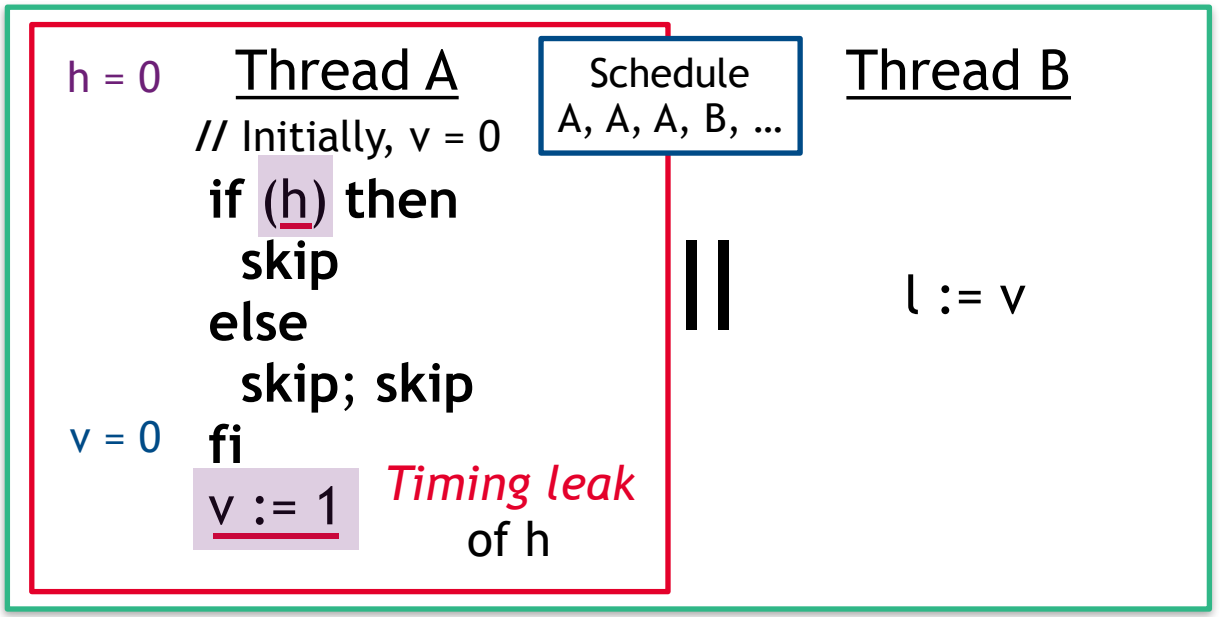
Interference-resilience (tricky)

+

Each thread must prevent (scheduler-relative) timing leaks!

Volpano & Smith, CSFW'98

Minimal example:



Motivation

Why wouldn't a compiler preserve it? (CSF'16)

Mantel et al. CSF'11 | Murray et al. CSF'16

relies/guarantees (*synchronisation*) | control variable contents (*sensitivity-switching*)

Concurrent value-dependent information-flow security

No storage leaks ✓

Some extra stuff to preserve (not that hard)

This particularly makes it harder!

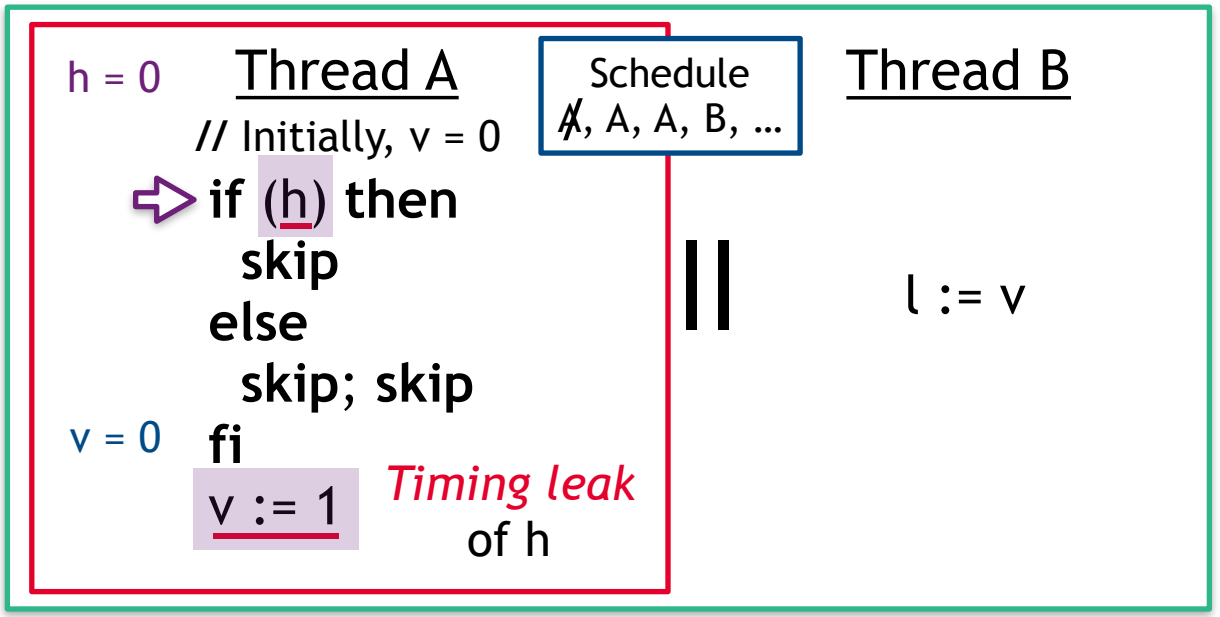
Interference-resilience (tricky)

+

Each thread must prevent (scheduler-relative) timing leaks!

Volpano & Smith, CSFW'98

Minimal example:



Motivation

Why wouldn't a compiler preserve it? (CSF'16)

Mantel et al. CSF'11 | Murray et al. CSF'16

relies/guarantees (*synchronisation*) | control variable contents (*sensitivity-switching*)

Concurrent value-dependent information-flow security

No storage leaks ✓

Some extra stuff to preserve (not that hard)

This particularly makes it harder!

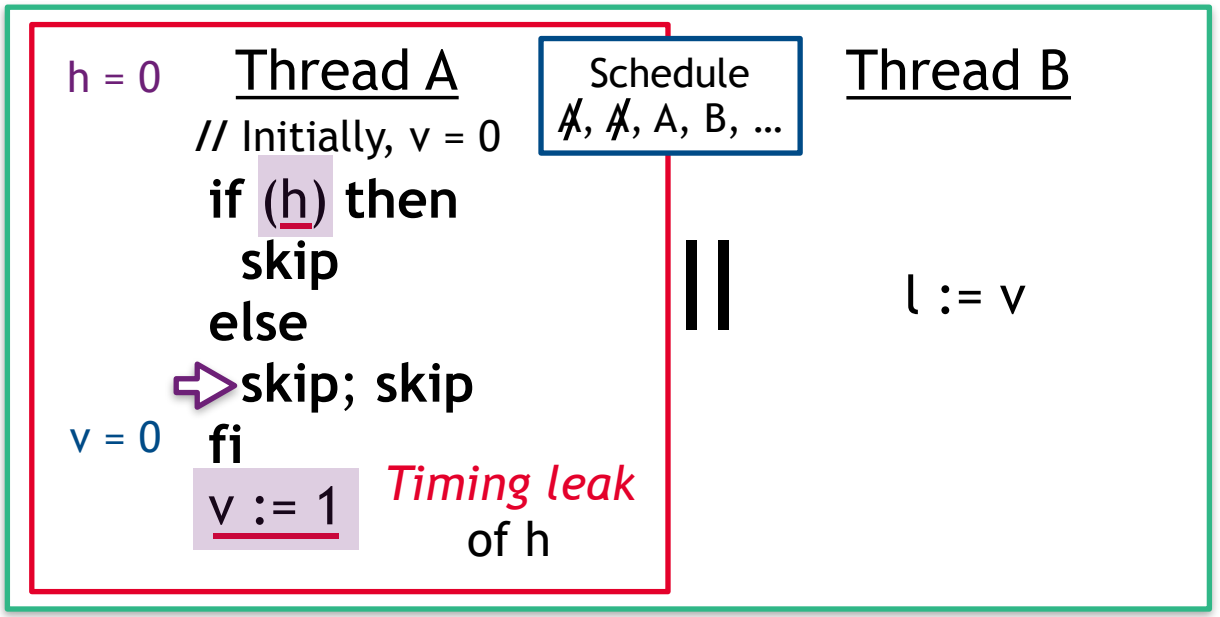
Interference-resilience (tricky)

+

Each thread must prevent (scheduler-relative) timing leaks!

Volpano & Smith, CSFW'98

Minimal example:



Motivation

Why wouldn't a compiler preserve it? (CSF'16)

Mantel et al. CSF'11 | Murray et al. CSF'16

relies/guarantees (*synchronisation*) | control variable contents (*sensitivity-switching*)

Concurrent value-dependent information-flow security

No storage leaks ✓

Some extra stuff to preserve (not that hard)

This particularly makes it harder!

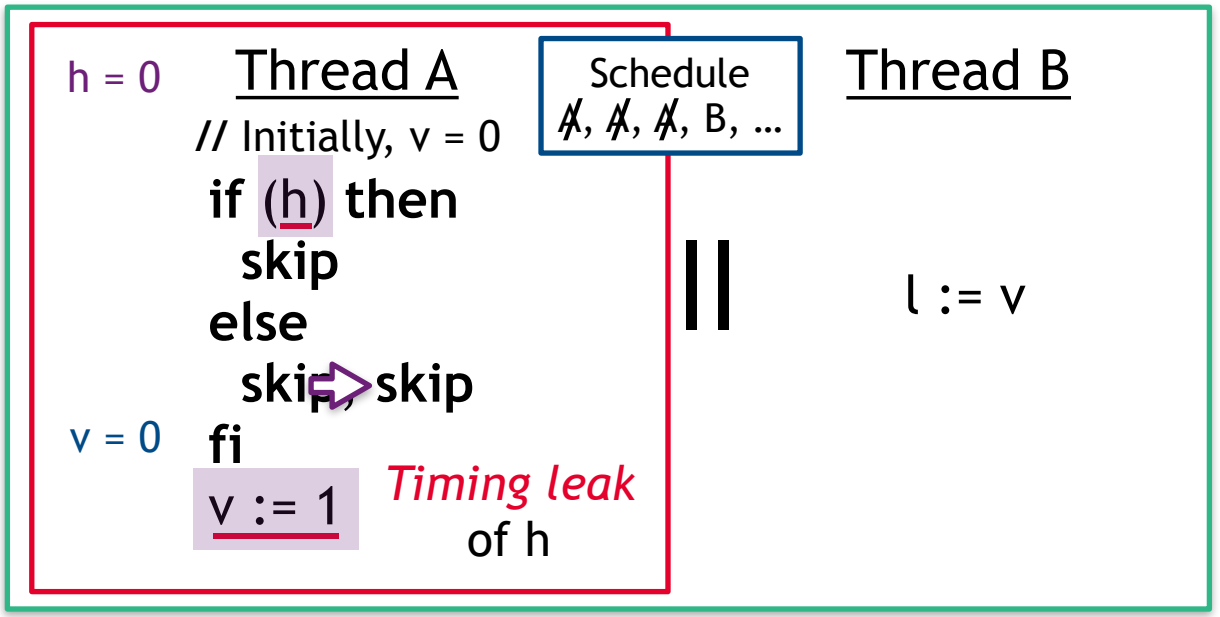
Interference-resilience (tricky)

+

Each thread must prevent (scheduler-relative) timing leaks!

Volpano & Smith, CSFW'98

Minimal example:



Motivation

Why wouldn't a compiler preserve it? (CSF'16)

Mantel et al. CSF'11 | Murray et al. CSF'16

relies/guarantees (*synchronisation*) | control variable contents (*sensitivity-switching*)

Concurrent value-dependent information-flow security

No storage leaks ✓

Some extra stuff to preserve (not that hard)

This particularly makes it harder!

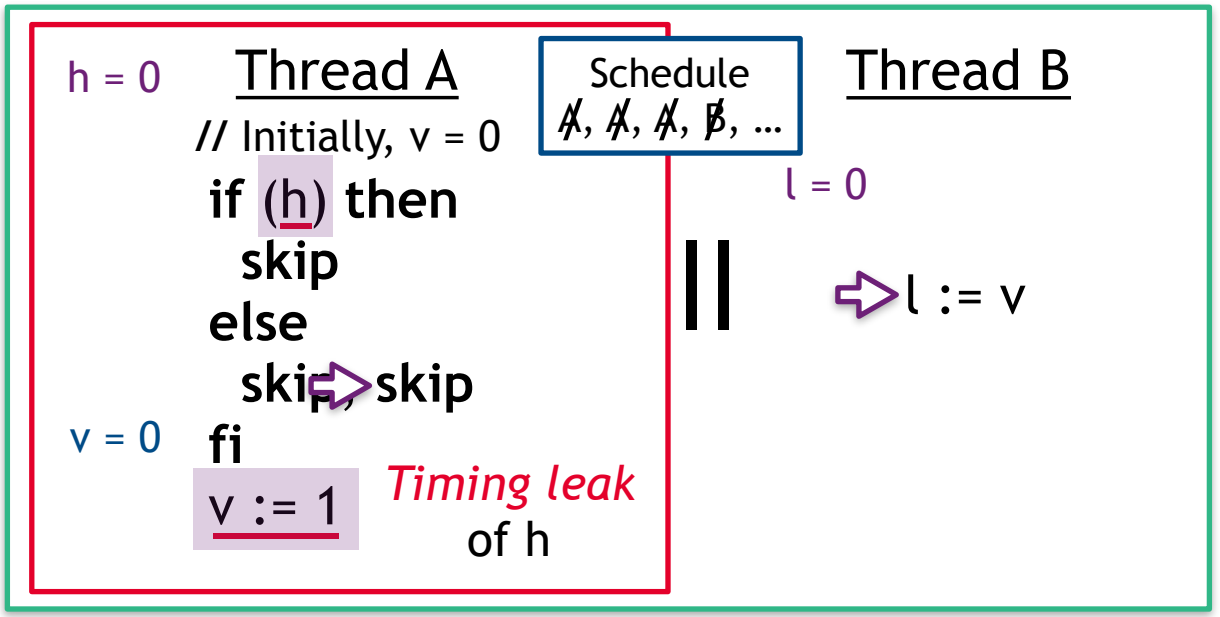
Interference-resilience (tricky)

+

Each thread must prevent (scheduler-relative) timing leaks!

Volpano & Smith, CSFW'98

Minimal example:



Motivation

Why wouldn't a compiler preserve it? (CSF'16)

Mantel et al. CSF'11 | Murray et al. CSF'16

relies/guarantees (*synchronisation*) | control variable contents (*sensitivity-switching*)

Concurrent value-dependent information-flow security

No storage leaks ✓

Some extra stuff to preserve (not that hard)

This particularly makes it harder!

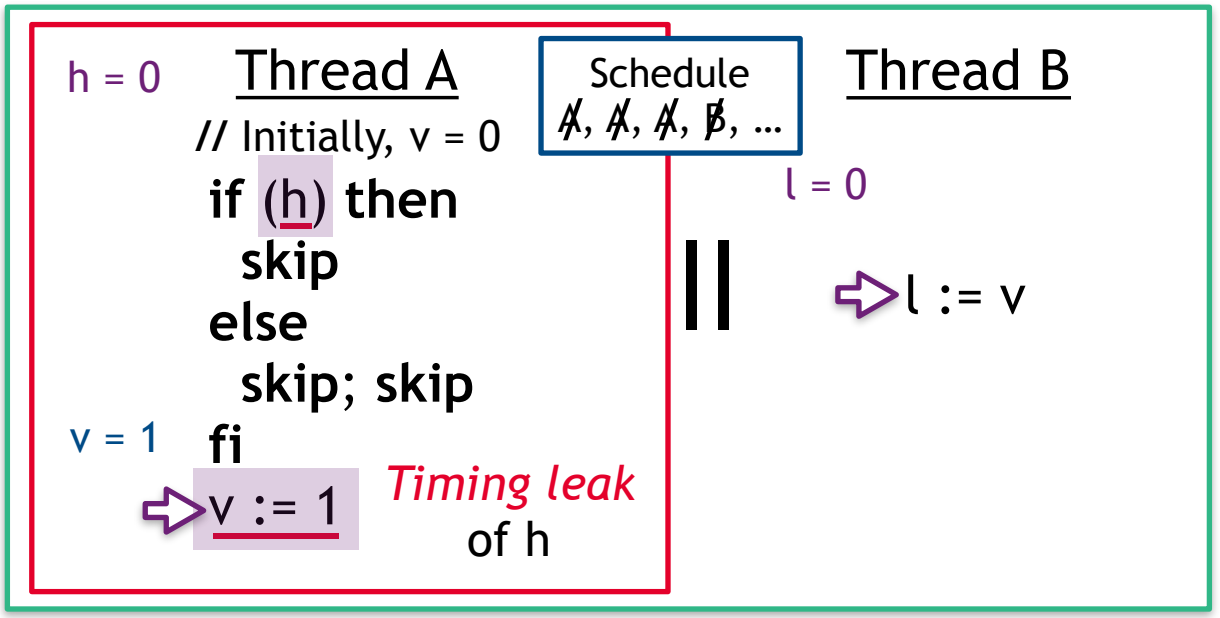
Interference-resilience (tricky)

+

Each thread must prevent (scheduler-relative) timing leaks!

Volpano & Smith, CSFW'98

Minimal example:



Motivation

Why wouldn't a compiler preserve it? (CSF'16)

Mantel et al. CSF'11 | Murray et al. CSF'16

relies/guarantees (*synchronisation*) | control variable contents (*sensitivity-switching*)

Concurrent value-dependent information-flow security

No storage leaks ✓

Some extra stuff to preserve (not that hard)

This particularly makes it harder!

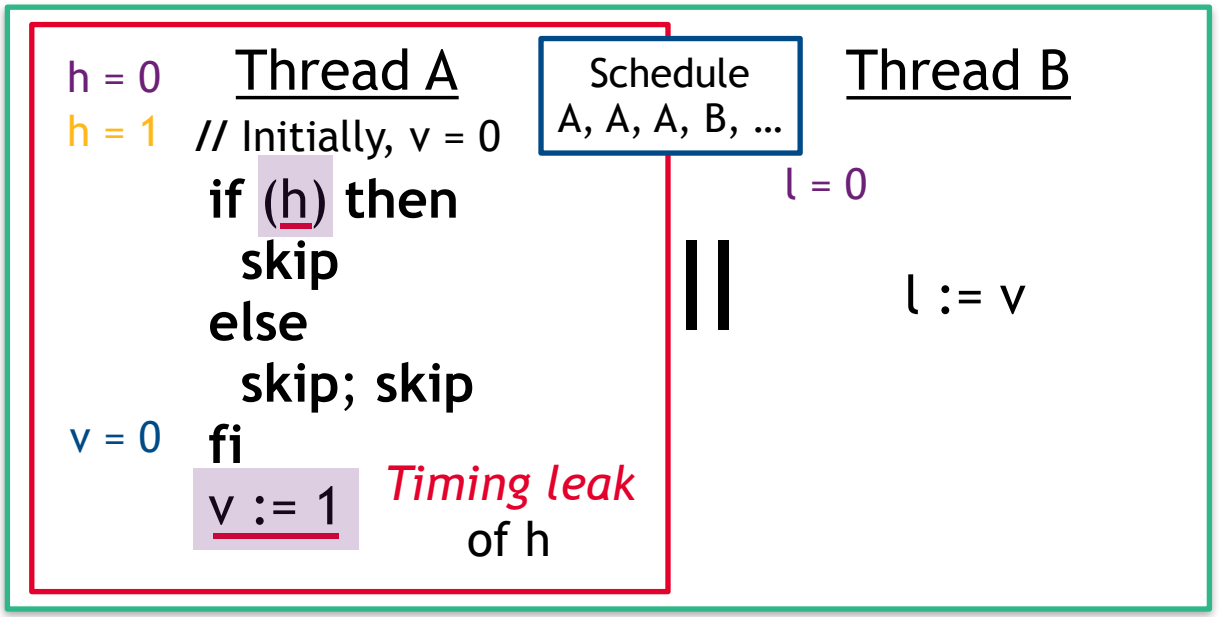
Interference-resilience (tricky)

+

Each thread must prevent (scheduler-relative) timing leaks!

Volpano & Smith, CSFW'98

Minimal example:



Motivation

Why wouldn't a compiler preserve it? (CSF'16)

Mantel et al. CSF'11 | Murray et al. CSF'16

relies/guarantees (*synchronisation*) | control variable contents (*sensitivity-switching*)

Concurrent value-dependent information-flow security

No storage leaks ✓

Some extra stuff to preserve (not that hard)

This particularly makes it harder!

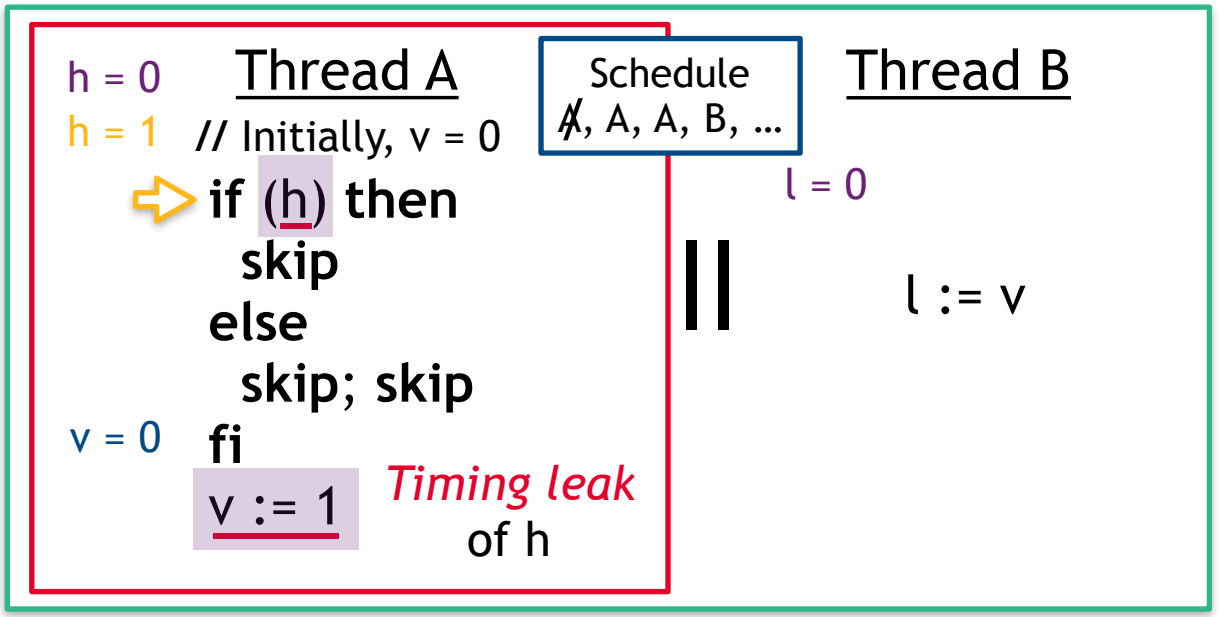
Interference-resilience (tricky)

+

Each thread must prevent (scheduler-relative) timing leaks!

Volpano & Smith, CSFW'98

Minimal example:



Motivation

Why wouldn't a compiler preserve it? (CSF'16)

Mantel et al. CSF'11 | Murray et al. CSF'16

relies/guarantees (*synchronisation*) | control variable contents (*sensitivity-switching*)

Concurrent value-dependent information-flow security

No storage leaks ✓

Some extra stuff to preserve (not that hard)

This particularly makes it harder!

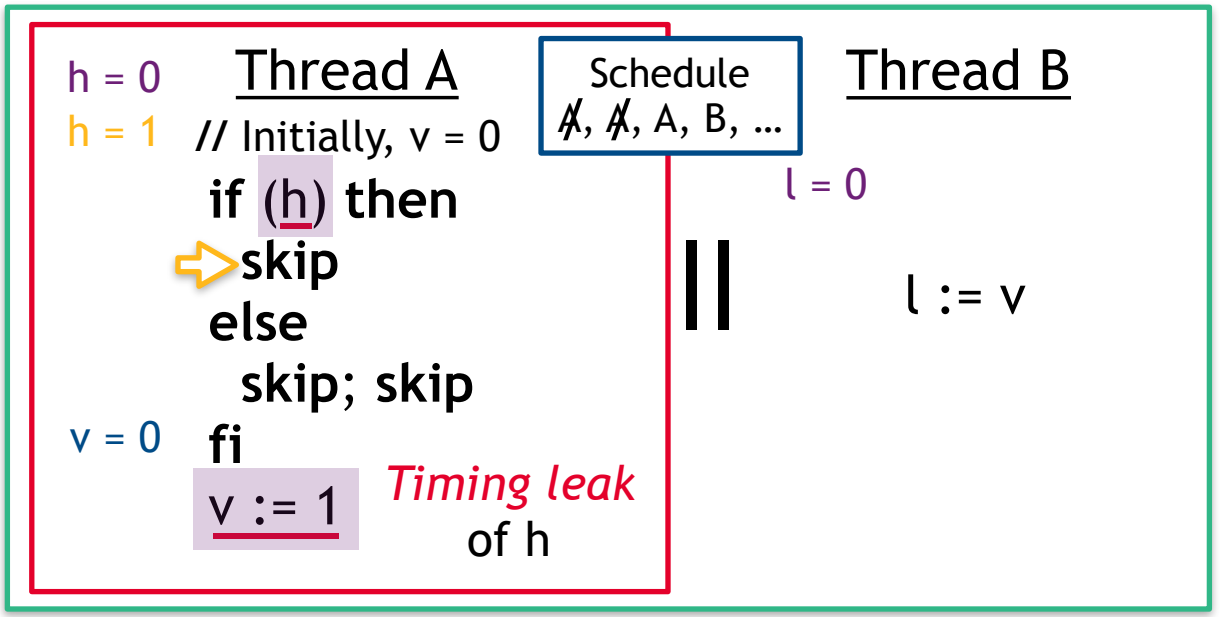
Interference-resilience (tricky)

+

Each thread must prevent (scheduler-relative) timing leaks!

Volpano & Smith, CSFW'98

Minimal example:



Motivation

Why wouldn't a compiler preserve it? (CSF'16)

Mantel et al. CSF'11 | Murray et al. CSF'16

relies/guarantees (*synchronisation*) | control variable contents (*sensitivity-switching*)

Concurrent value-dependent information-flow security

No storage leaks ✓

Some extra stuff to preserve (not that hard)

This particularly makes it harder!

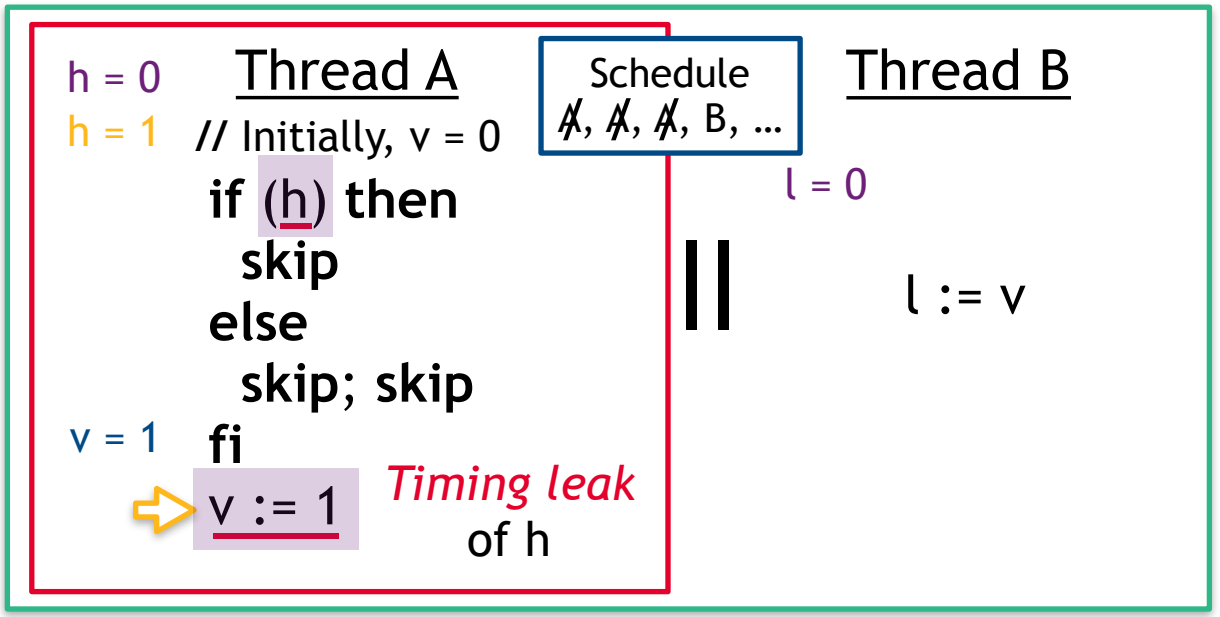
Interference-resilience (tricky)

+

Each thread must prevent (scheduler-relative) timing leaks!

Volpano & Smith, CSFW'98

Minimal example:



Motivation

Why wouldn't a compiler preserve it? (CSF'16)

Mantel et al. CSF'11 | Murray et al. CSF'16

relies/guarantees (*synchronisation*) | control variable contents (*sensitivity-switching*)

Concurrent value-dependent information-flow security

No storage leaks ✓

Some extra stuff to preserve (not that hard)

This particularly makes it harder!

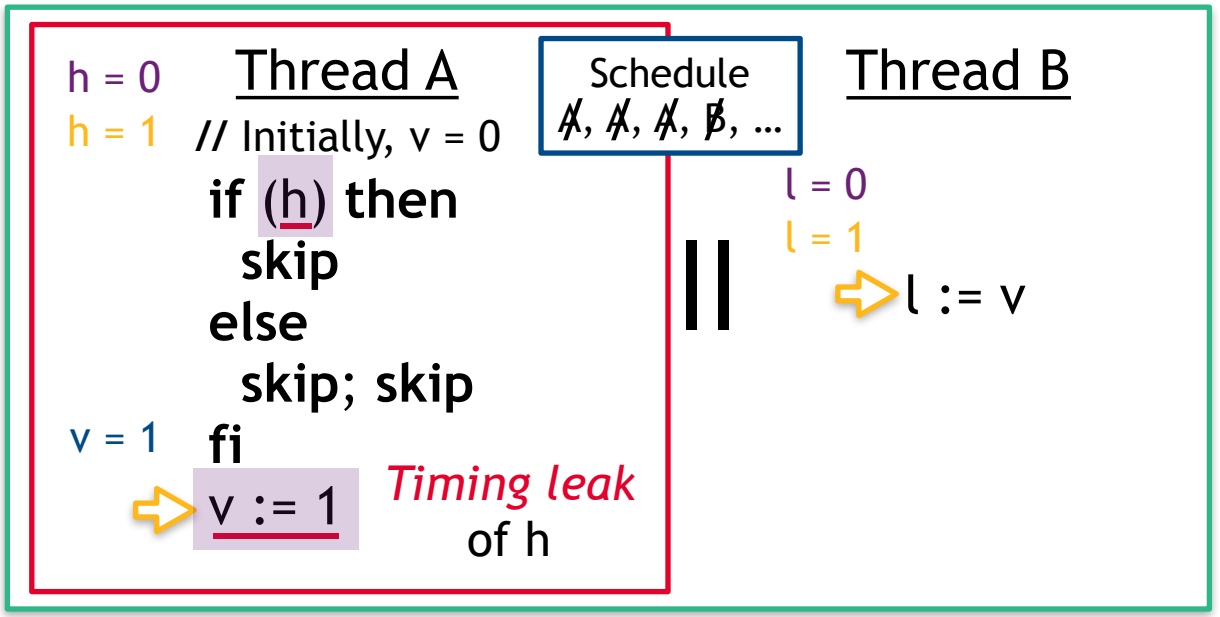
Interference-resilience (tricky)

+

Each thread must prevent (scheduler-relative) timing leaks!

Volpano & Smith, CSFW'98

Minimal example:



Motivation

Why wouldn't a compiler preserve it? (CSF'16)

Mantel et al. CSF'11 | Murray et al. CSF'16

relies/guarantees (*synchronisation*) | control variable contents (*sensitivity-switching*)

Concurrent value-dependent information-flow security

No storage leaks ✓

Some extra stuff to preserve (not that hard)

This particularly makes it harder!

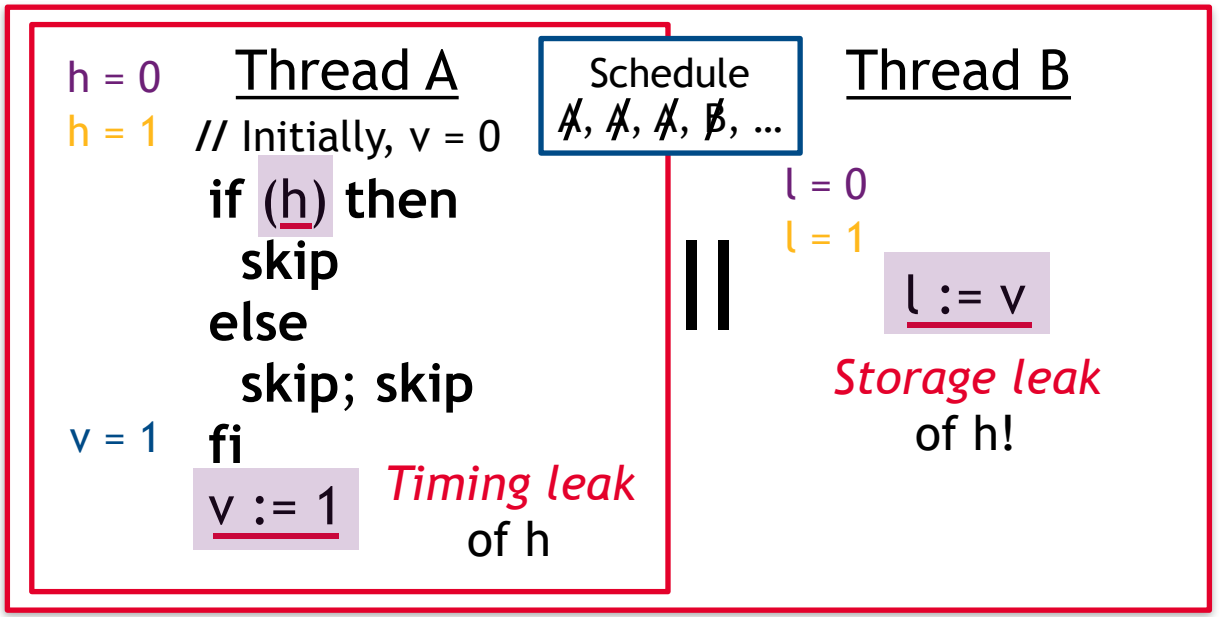
Interference-resilience (tricky)

+

Each thread must prevent (scheduler-relative) timing leaks!

Volpano & Smith, CSFW'98

Minimal example:



Motivation

Why wouldn't a compiler preserve it? (CSF'16)

Mantel et al. CSF'11 | Murray et al. CSF'16

relies/guarantees (*synchronisation*) | control variable contents (*sensitivity-switching*)

Concurrent value-dependent information-flow security

No storage leaks ✓

Some extra stuff to preserve (not that hard)

This particularly makes it harder!

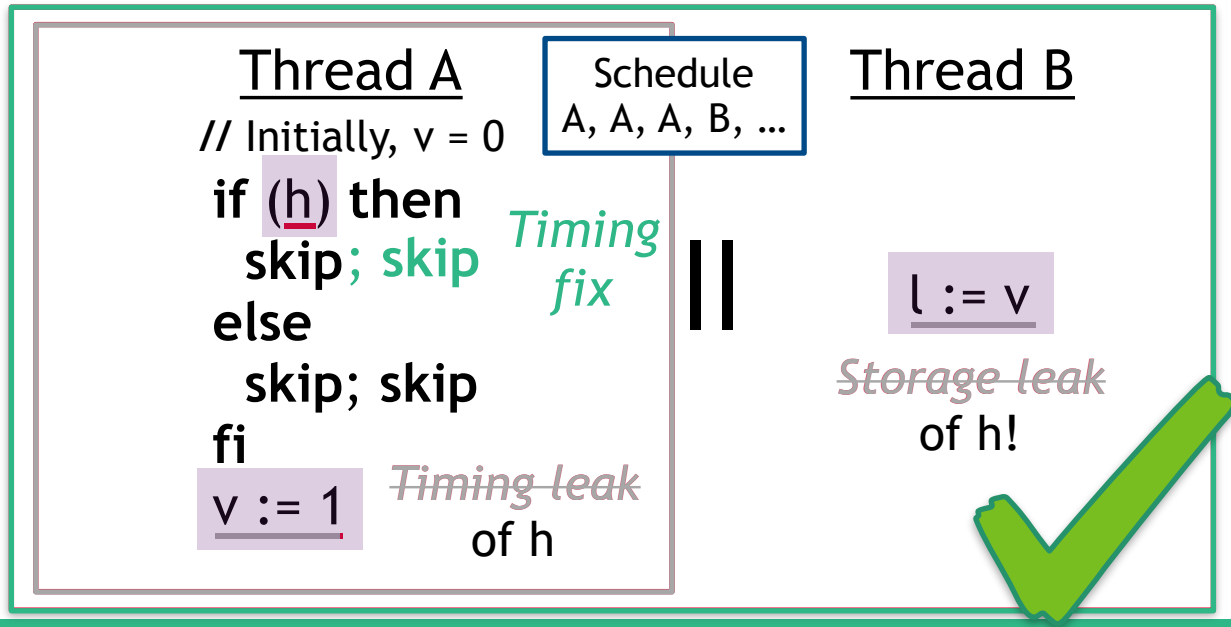
Interference-resilience (tricky)

+

Each thread must prevent (scheduler-relative) timing leaks!

Volpano & Smith, CSFW'98

Minimal example:



Motivation

Why wouldn't a compiler preserve it? (CSF'16)

Mantel et al. CSF'11 | Murray et al. CSF'16

relies/guarantees (*synchronisation*) | control variable contents (*sensitivity-switching*)

Concurrent value-dependent information-flow security

No storage leaks ✓

Some extra stuff to preserve (not that hard)

But: Compiler may eliminate it!

This particularly makes it harder!

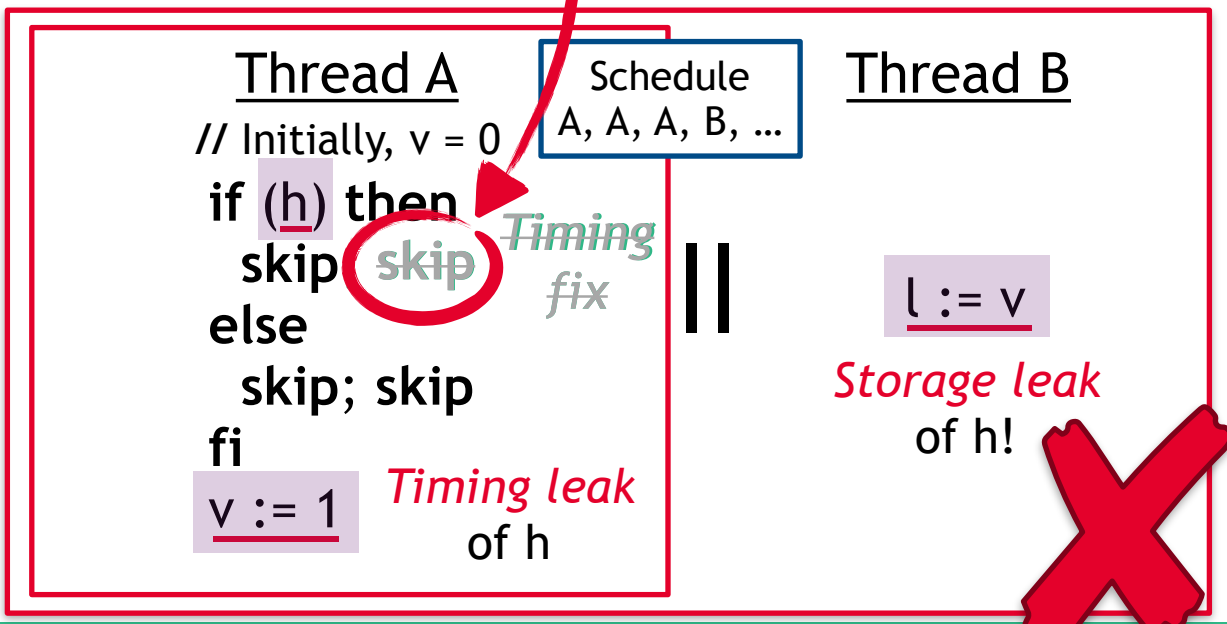
Interference-resilience (tricky)

+

Each thread must prevent (scheduler-relative) timing leaks!

Volpano & Smith, CSFW'98

Minimal example:



Motivation

Why wouldn't a compiler preserve it? (CSF'16)

Mantel et al. CSF'11 | Murray et al. CSF'16

relies/guarantees (*synchronisation*) | control variable contents (*sensitivity-switching*)

Concurrent value-dependent information-flow security

No storage leaks ✓

Some extra stuff to preserve (not that hard)

But: Compiler may eliminate it!

(or, introduce *new* "**if** (h)"!)

This particularly makes it harder!

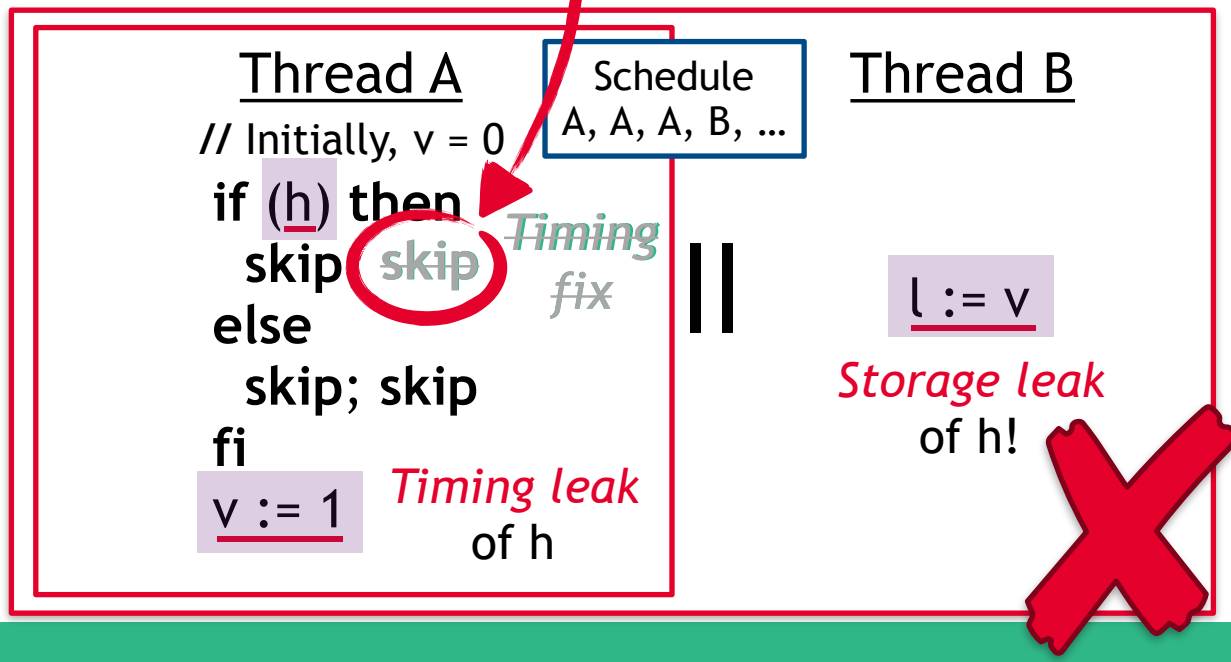
Interference-resilience (tricky)

+

Each thread must prevent (scheduler-relative) timing leaks!

Volpano & Smith, CSFW'98

Minimal example:

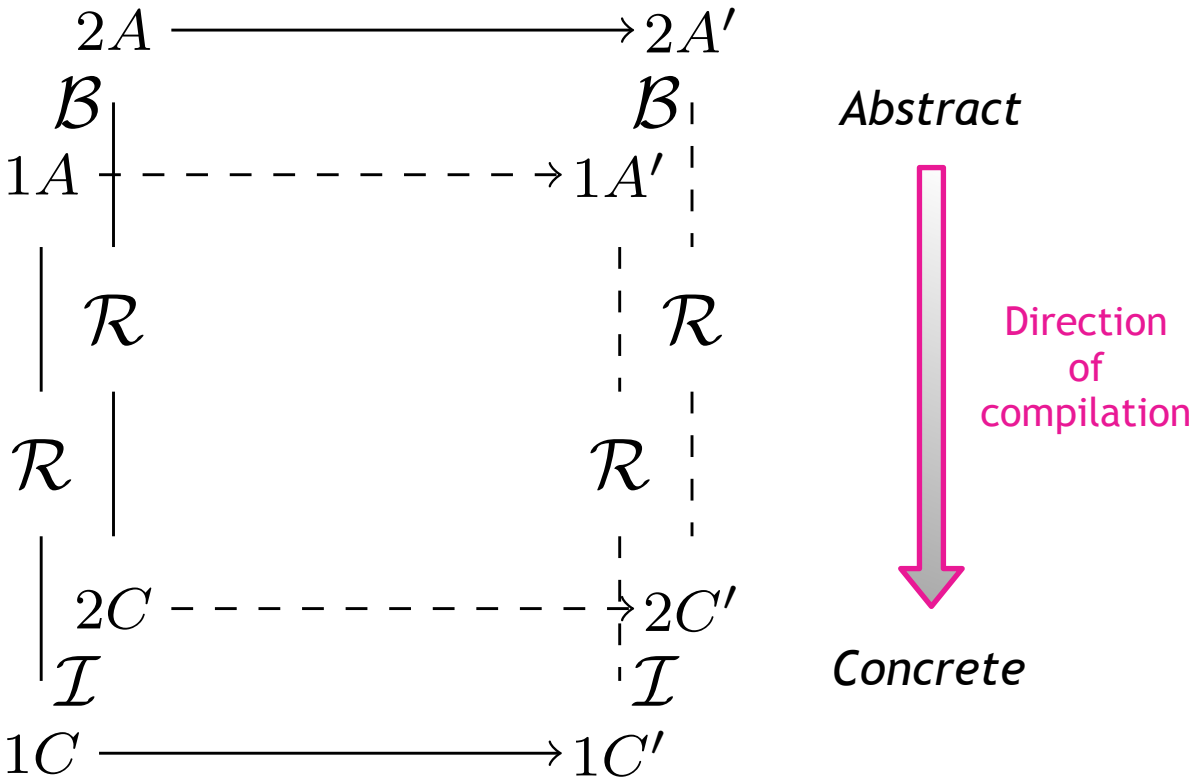


Background

Why is it hard to prove? (CSF'16)

Concurrent value-dependent information-flow security

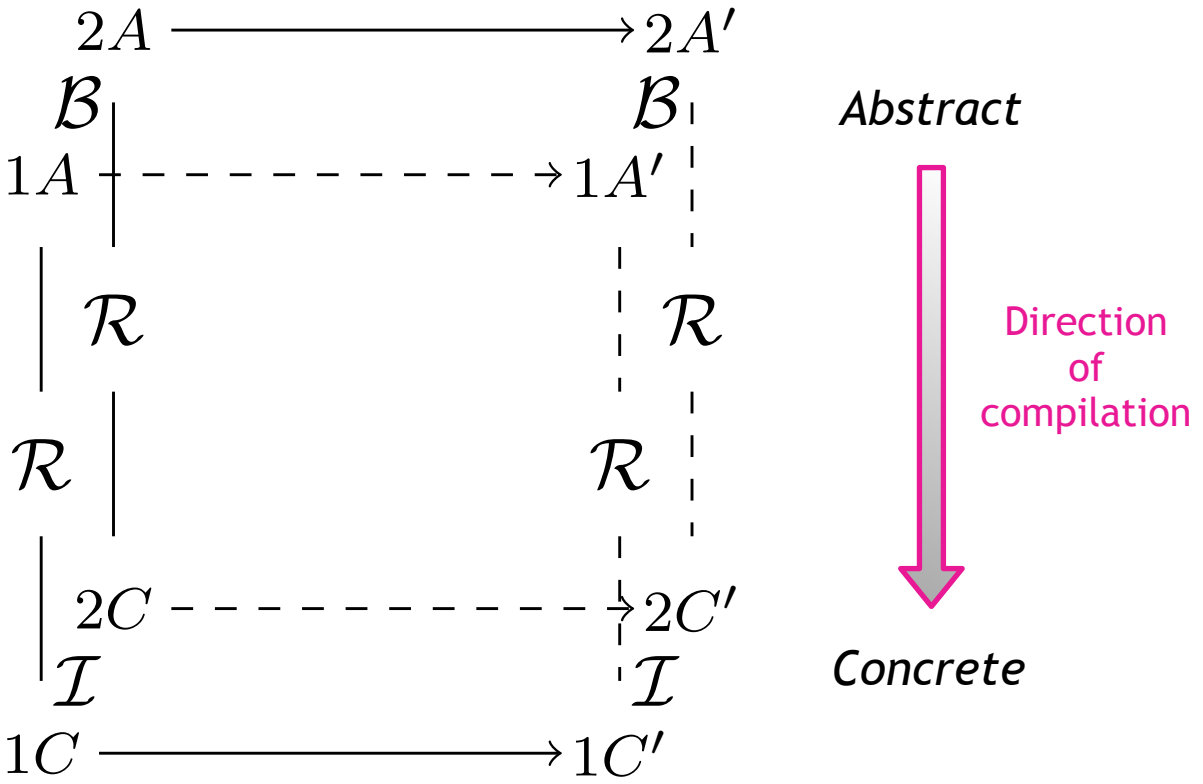
-preserving refinement



Background

Why is it hard to prove? (CSF'16)

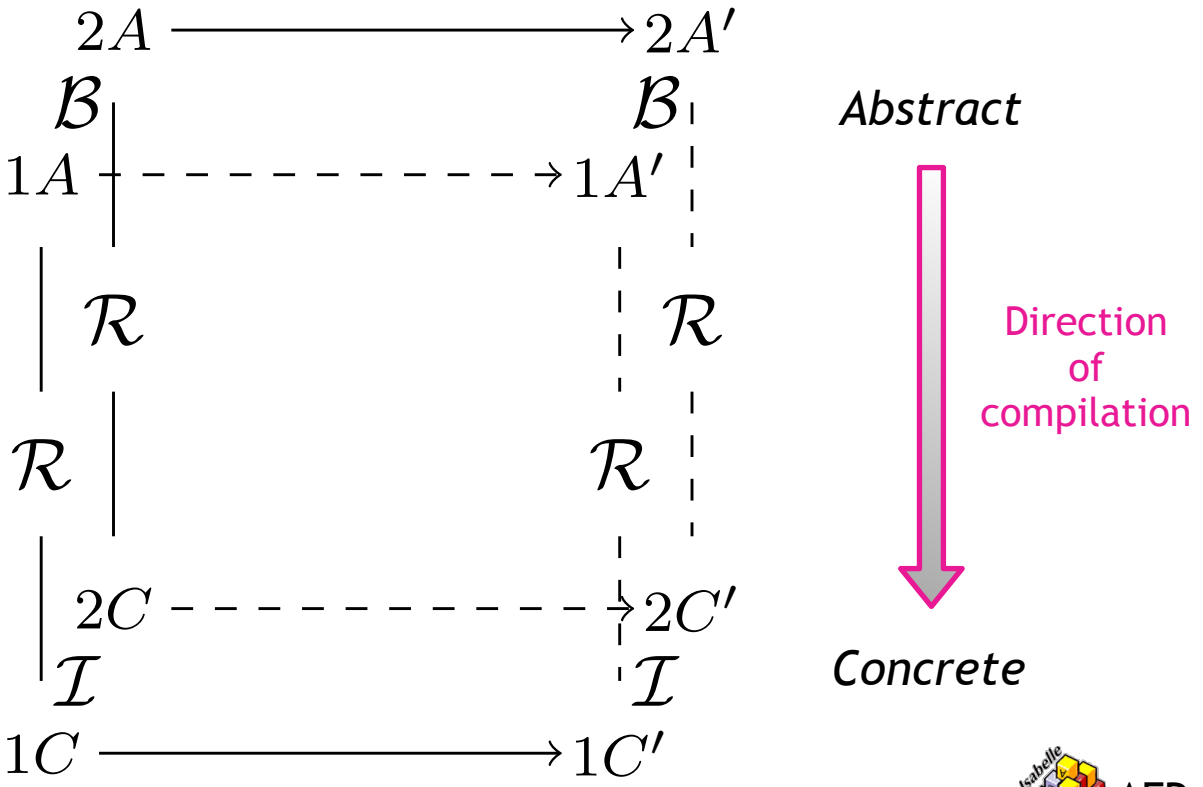
Confidentiality-preserving refinement



Background

Why is it hard to prove? (CSF'16)

Confidentiality-preserving refinement

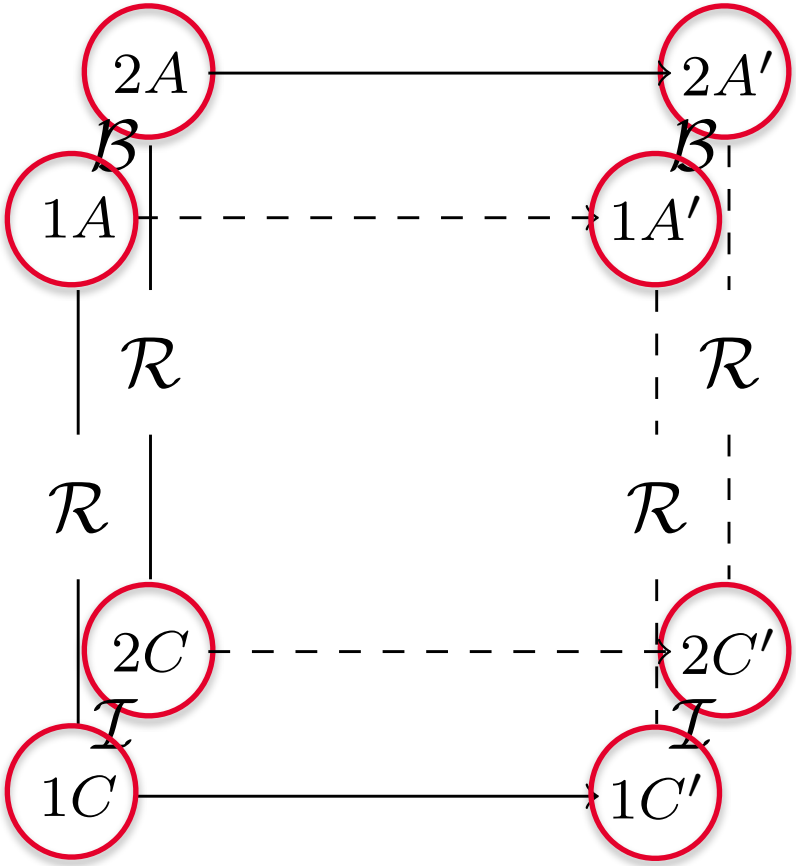


AFP entry:
`Dependent_SIFUM_Refinement`

Background

Why is it hard to prove? (CSF'16)

Confidentiality-preserving refinement



Program configurations

Abstract

Direction of compilation

Concrete

AFP entry:
`Dependent_SIFUM_Refinement`

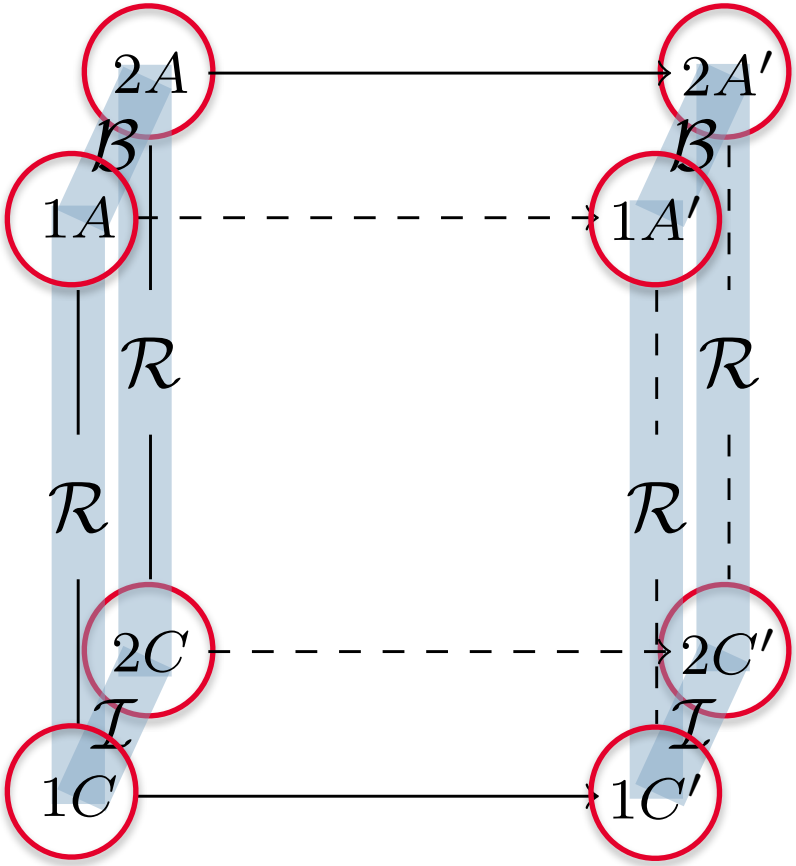
Background
Why is it hard to prove? (CSF'16)

Confidentiality-preserving refinement

Relations

(between)

Program configurations



Abstract

Direction of compilation

Concrete

AFP entry:
`Dependent_SIFUM_Refinement`

Background

Why is it hard to prove? (CSF'16)

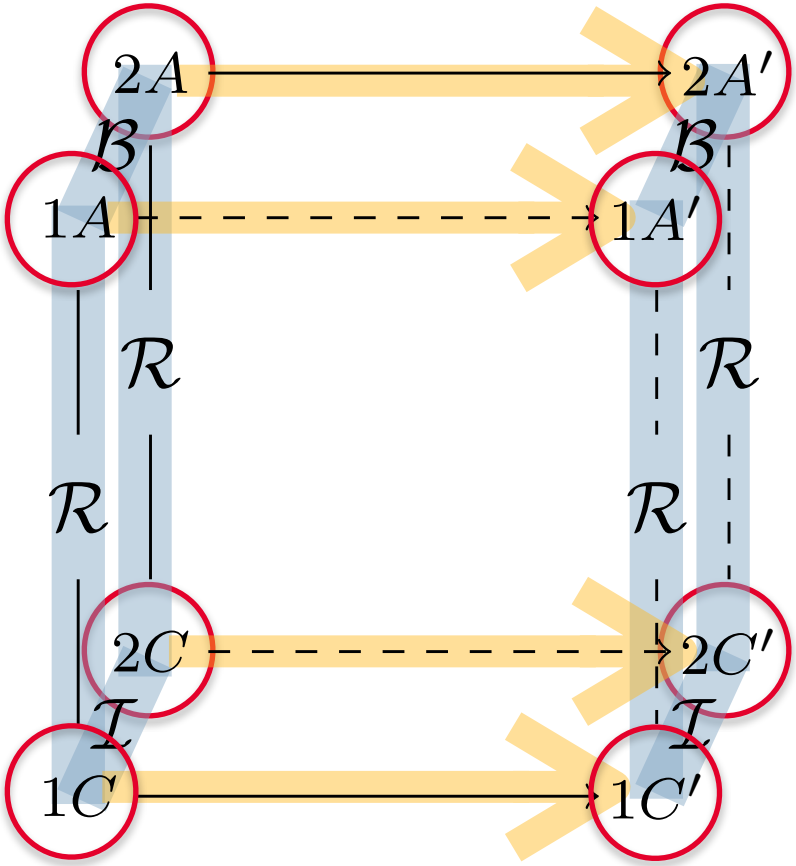
Confidentiality-preserving refinement

Relations

Execution steps

(between)

Program configurations



Abstract

Direction of compilation

Concrete

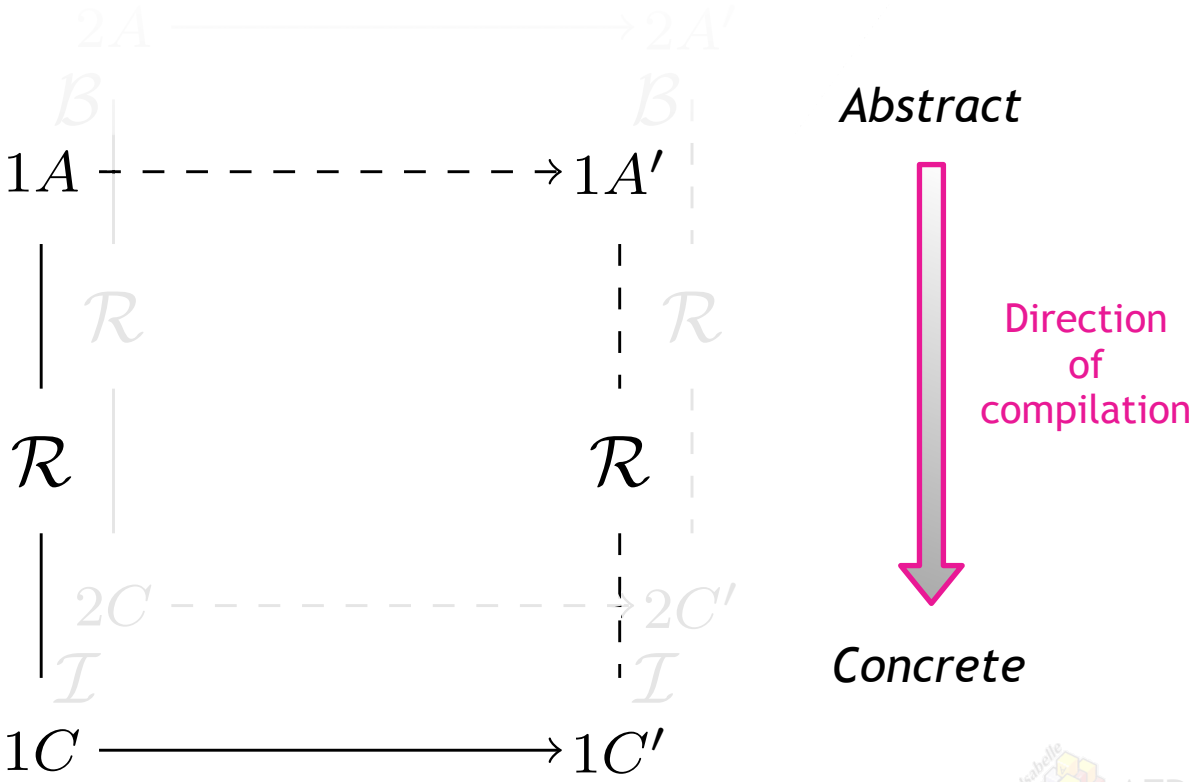
AFP entry:

`Dependent_SIFUM_Refinement`

Background

Why is it hard to prove? (CSF'16)

"Usual" refinement:

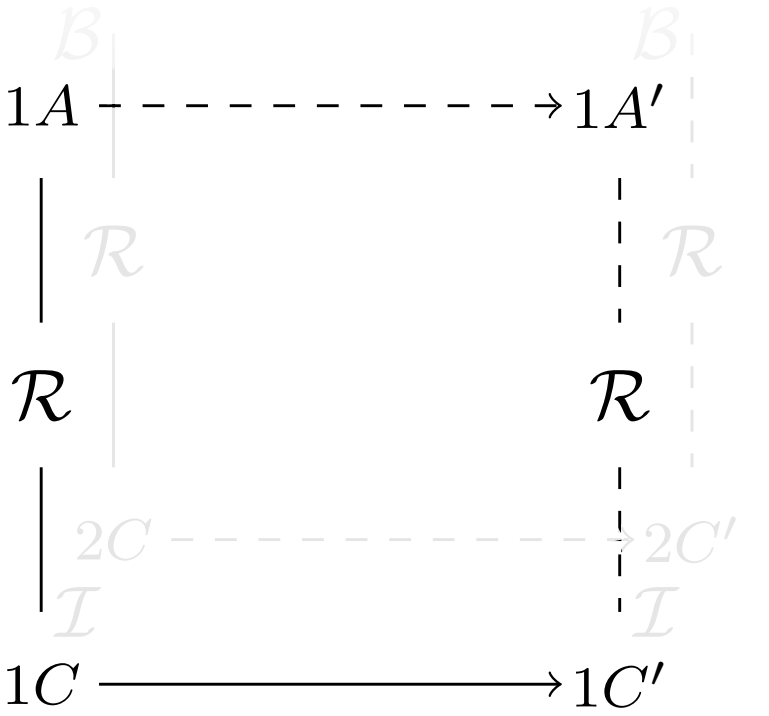


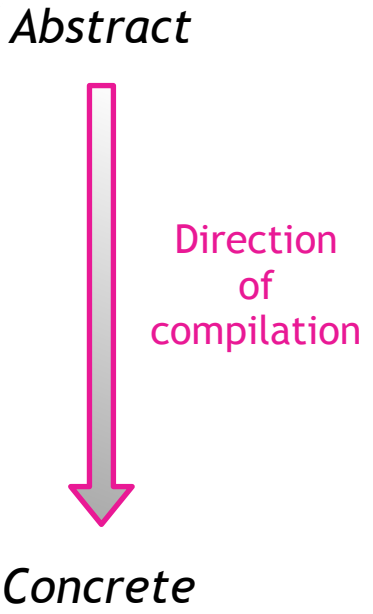
AFP entry: Dependent_SIFUM_Refinement

Background
Why is it hard to prove? (CSF'16)

"Usual" refinement:

A simulates C ⇒ C refines A





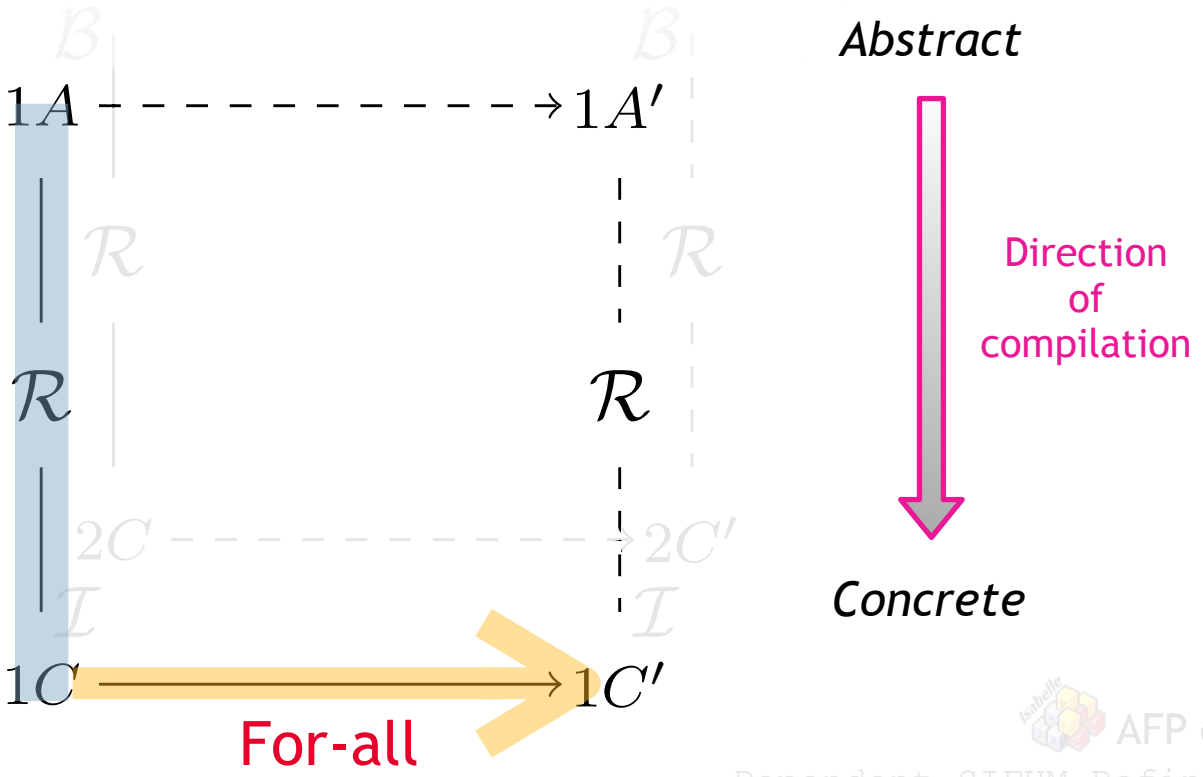
AFP entry: Dependent_SIFUM_Refinement

Background

Why is it hard to prove? (CSF'16)

"Usual" refinement:

A simulates C ⇒ C refines A



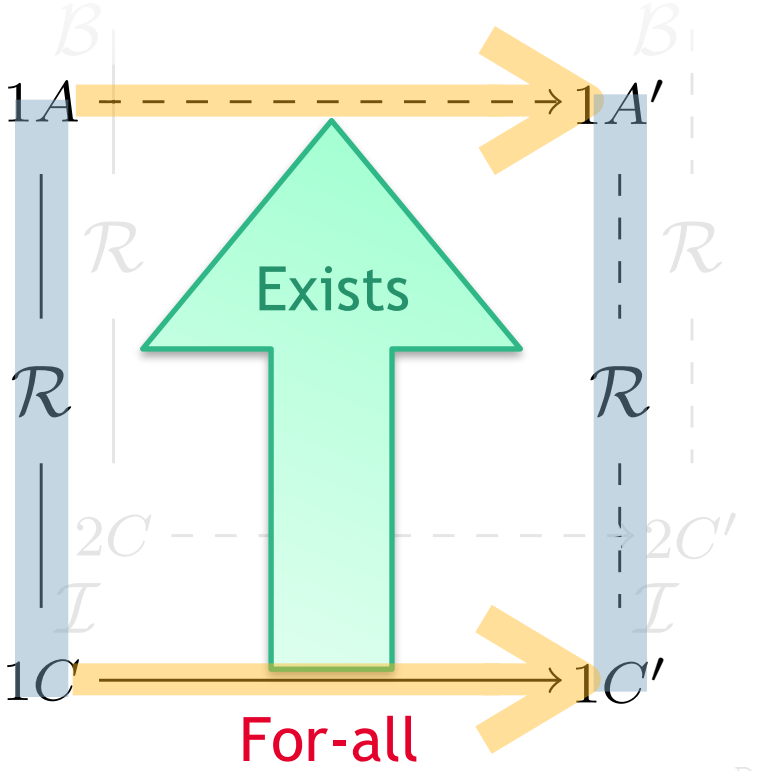
AFP entry: Dependent_SIFUM_Refinement

Background

Why is it hard to prove? (CSF'16)

"Usual" refinement:

A simulates C ⇒ C refines A



Abstract

Direction of compilation

Concrete

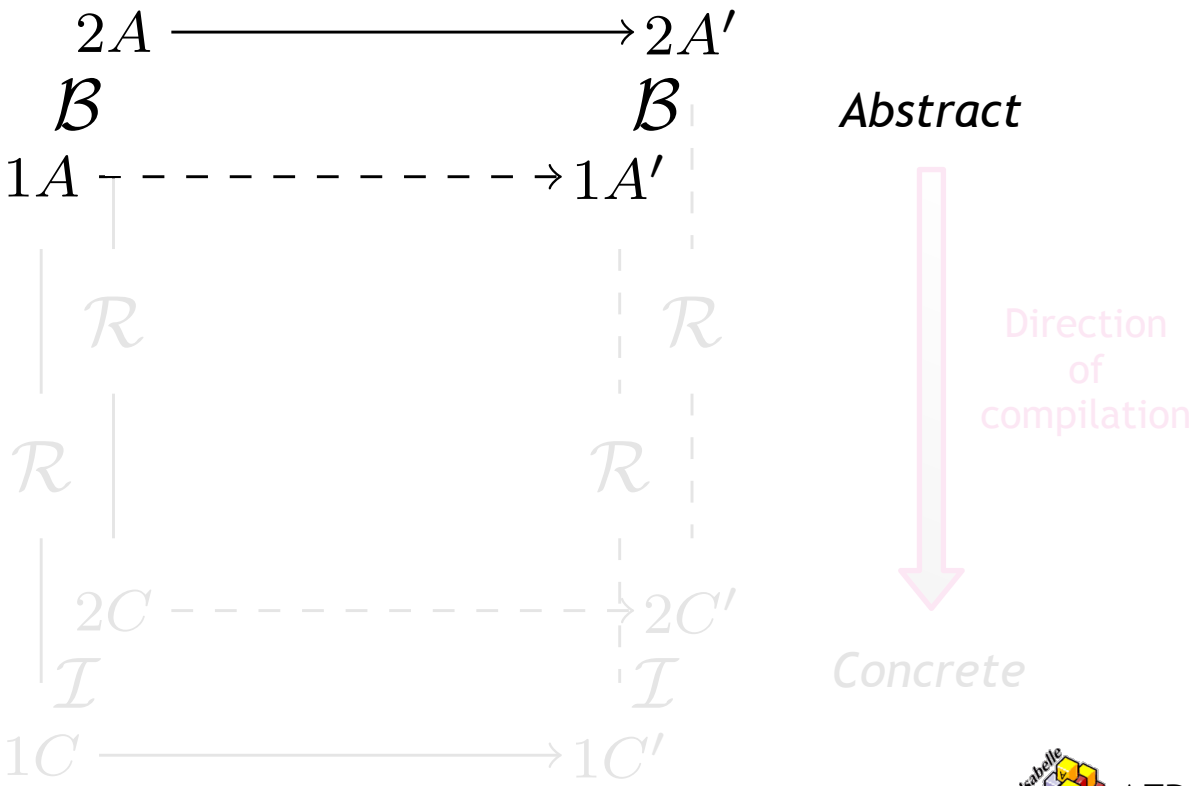
AFP entry: Dependent_SIFUM_Refinement

Background
Why is it hard to prove? (CSF'16)

From compiler front-end

Security proof (Bisimulation B)

Confidentiality-preserving refinement



AFP entry:
`Dependent_SIFUM_Refinement`

Background

Why is it hard to prove? (CSF'16)

From compiler front-end

Security proof (Bisimulation B)

+

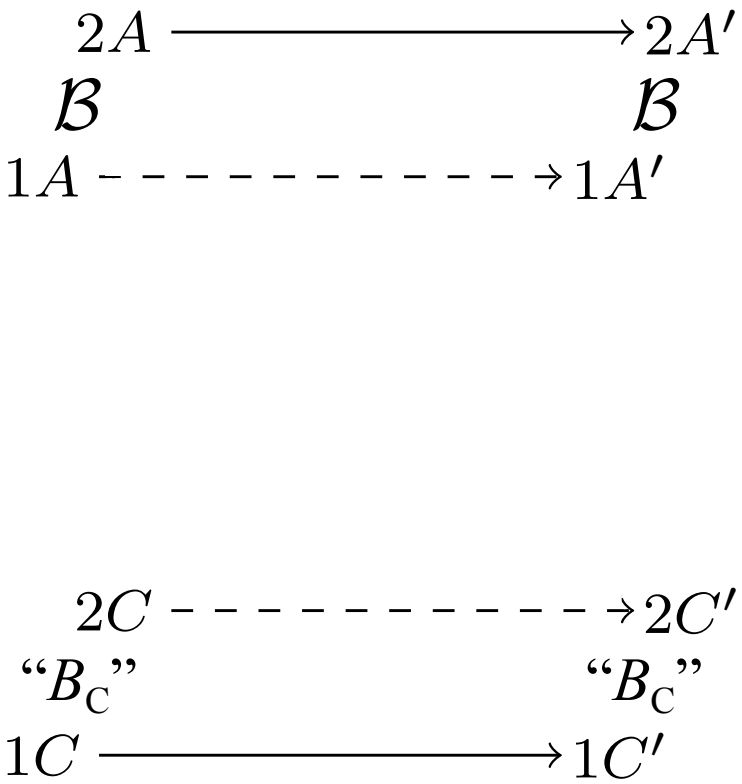
?

⇒

For *free* ?

Security proof (Bisimulation "B_C")

Confidentiality-preserving refinement



Abstract

Direction of compilation

Concrete

AFP entry:

`Dependent_SIFUM_Refinement`

Background

Why is it hard to prove? (CSF'16)

From compiler front-end

Security proof (Bisimulation B)

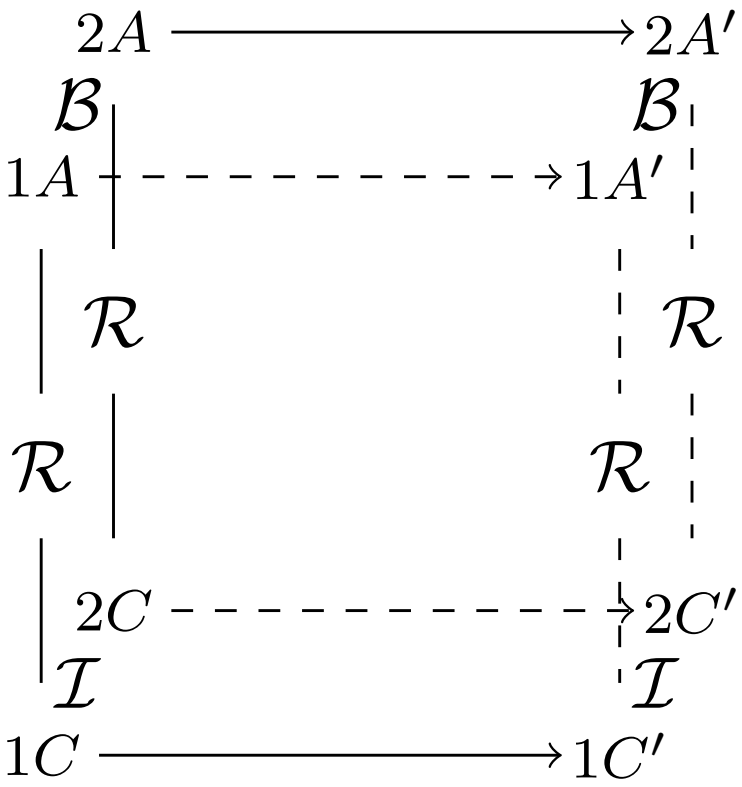
+

Compiler correctness proof (Refinement R)

⇒ "For free"*

Security proof (Bisimulation $\mathrm{B_c of}\ B\ R\ I$)

*Confidentiality-preserving refinement



Abstract

Direction of compilation

Concrete

AFP entry:
`Dependent_SIFUM_Refinement`

Background

Why is it hard to prove? (CSF'16)

From compiler front-end

Security proof (Bisimulation B)

+

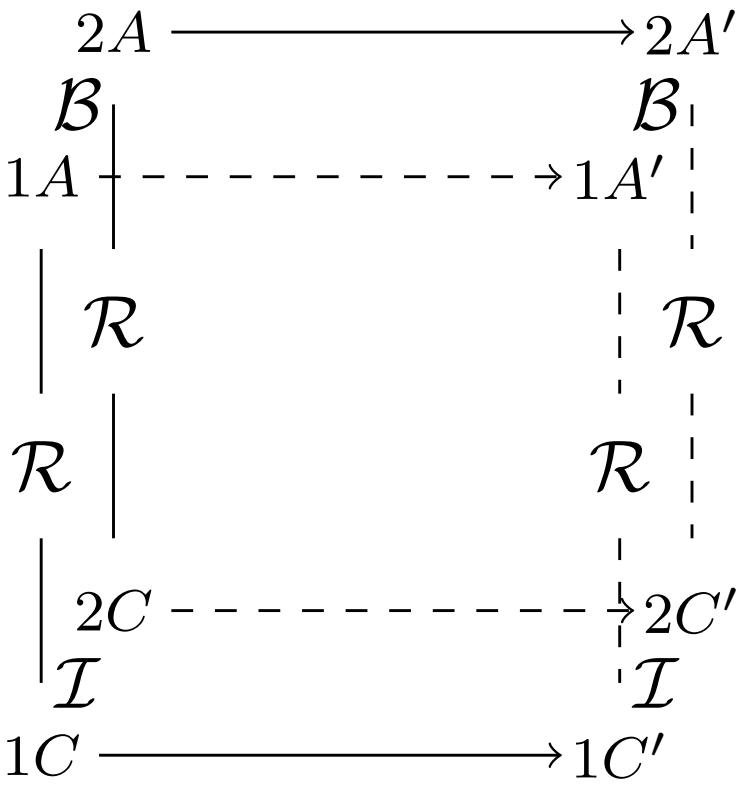
Compiler correctness proof (Refinement R)

⇒ "For free"*

Security proof (Bisimulation $\mathrm{B_c of}\ B\ R\ I$)

*Confidentiality-preserving refinement

(Two-sided!)



Abstract

Direction of compilation

Concrete

AFP entry:

`Dependent_SIFUM_Refinement`

Background

Why is it hard to prove? (CSF'16)

From compiler front-end

Security proof (Bisimulation B)

+

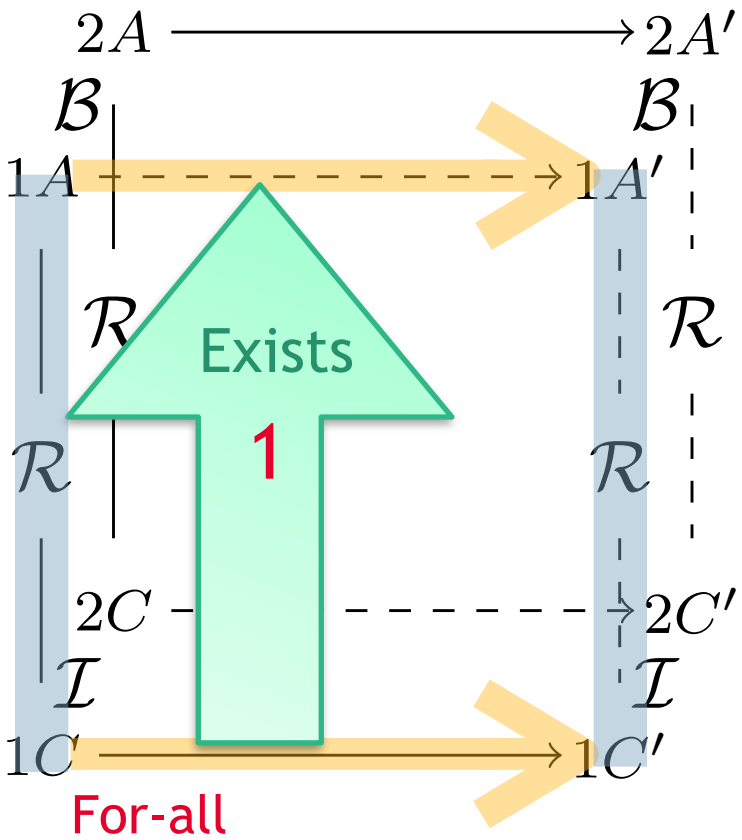
Compiler correctness proof (Refinement R)

⇒ "For free"*

Security proof (Bisimulation $\mathrm{B_c of}\ B\ R\ I$)

*Confidentiality-preserving refinement

(Two-sided!)



Abstract

Direction of compilation

Concrete

AFP entry:
`Dependent_SIFUM_Refinement`

Background

Why is it hard to prove? (CSF'16)

From compiler front-end

Security proof (Bisimulation B)

+

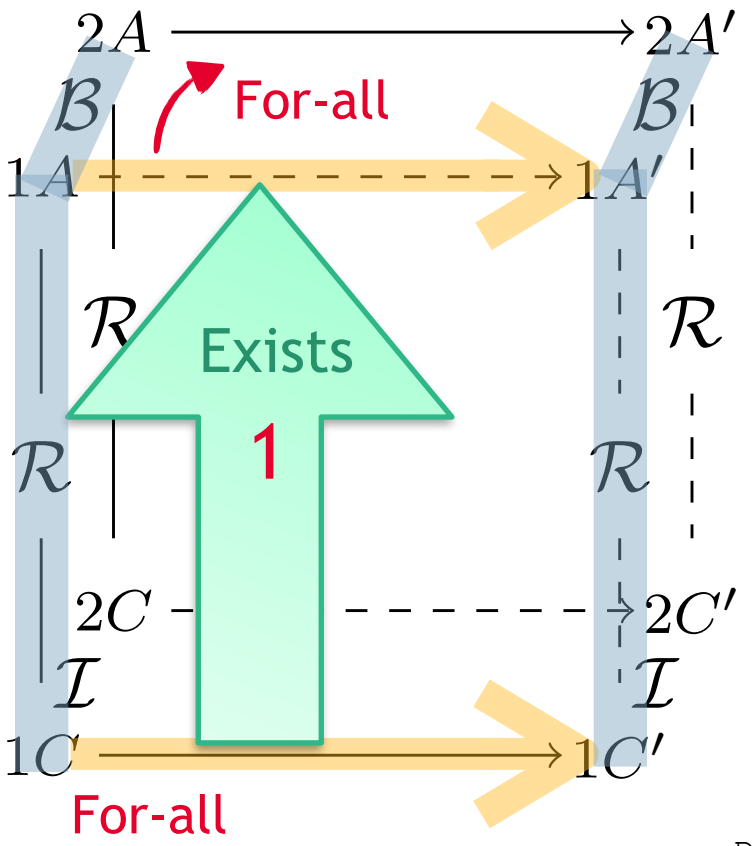
Compiler correctness proof (Refinement R)

⇒ "For free"*

Security proof (Bisimulation $\mathrm{B_c of}\ B\ R\ I$)

*Confidentiality-preserving refinement

(Two-sided!)



Abstract

Direction of compilation

Concrete

AFP entry:
`Dependent_SIFUM_Refinement`

Background

Why is it hard to prove? (CSF'16)

From compiler front-end

Security proof (Bisimulation B)

+

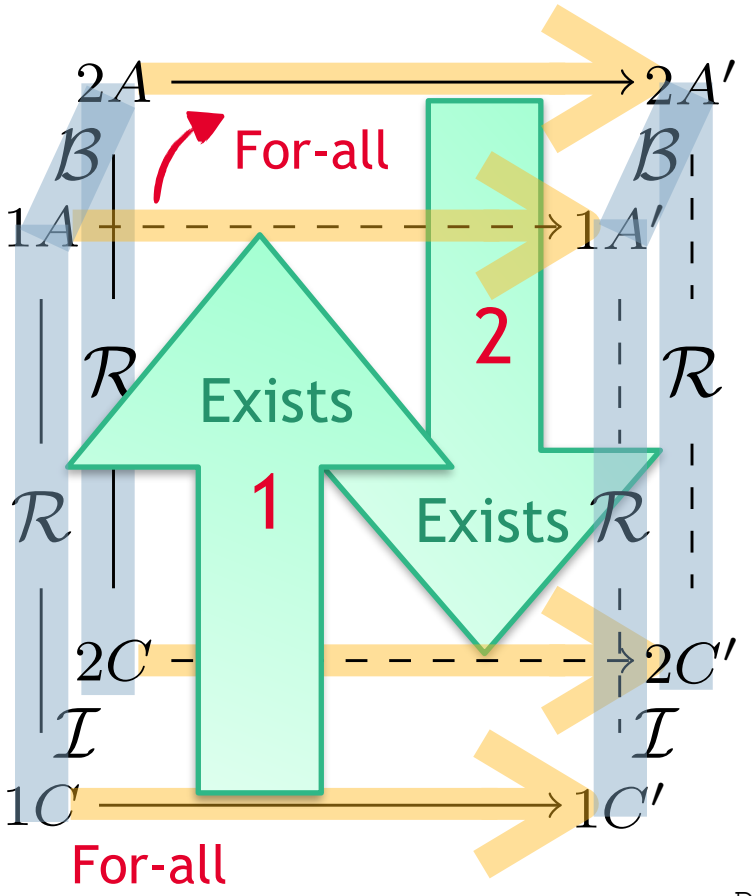
Compiler correctness proof (Refinement R)

⇒ "For free"*

Security proof (Bisimulation $\mathrm{B_c of}\ B\ R\ I$)

*Confidentiality-preserving refinement

(Two-sided!)



Abstract

Direction of compilation

Concrete

AFP entry:
`Dependent_SIFUM_Refinement`

Background

Why is it hard to prove? (CSF'16)

From compiler front-end

Security proof (Bisimulation B)

+

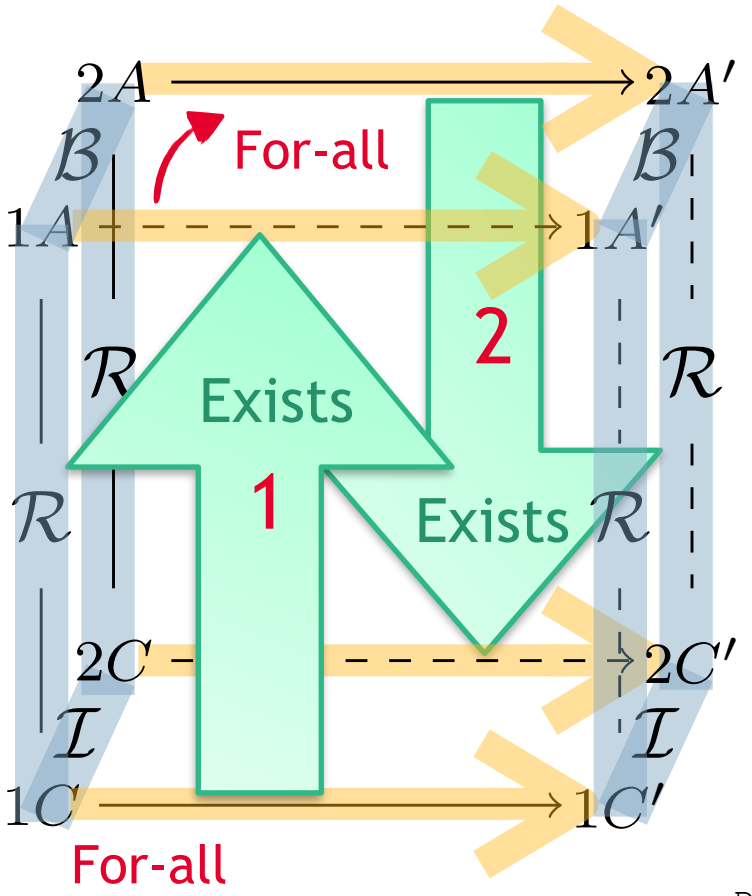
Compiler correctness proof (Refinement R)

⇒ "For free"*

Security proof (Bisimulation $\mathrm{B_c of}\ B\ R\ I$)

*Confidentiality-preserving refinement

(Two-sided!)



Abstract

Direction of compilation

Concrete

AFP entry:
`Dependent_SIFUM_Refinement`

Background

Why is it hard to prove? (CSF'16)

(**Compare:** Barthe et al. CSF'18)

From compiler front-end

Security proof (Bisimulation B)

+

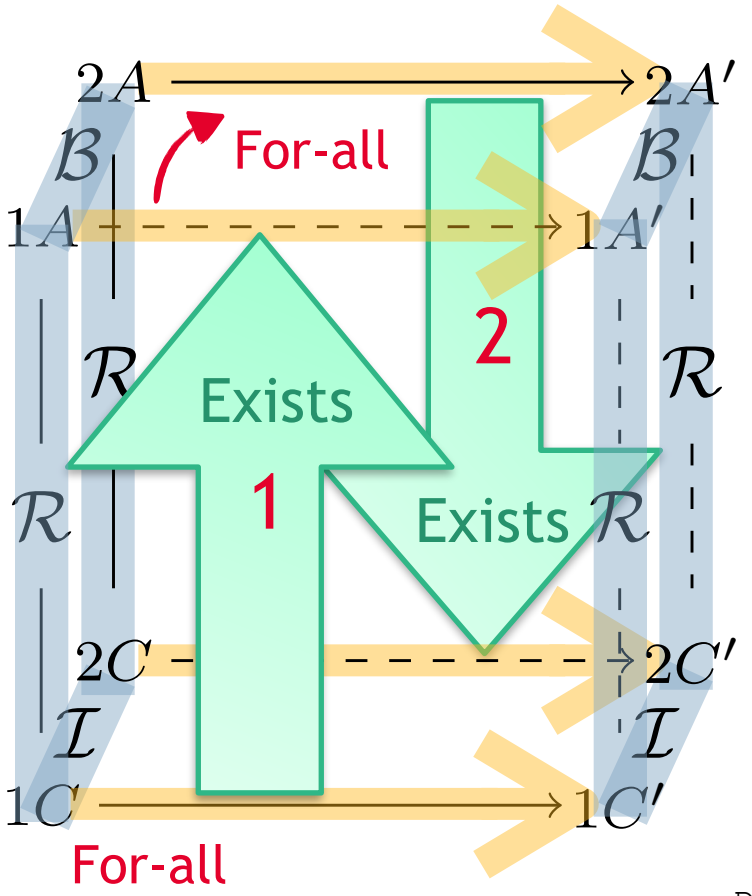
Compiler correctness proof (Refinement R)

⇒ "For free"*

Security proof (Bisimulation $\mathrm{B_c of}\ B\ R\ I$)

*_Confidentiality-preserving_ refinement

(Two-sided!)



Abstract

Direction of compilation

Concrete

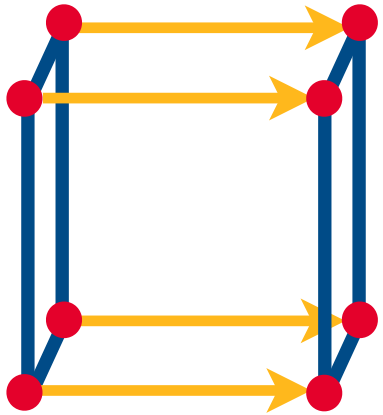
AFP entry:
`Dependent_SIFUM_Refinement`

Our contributions

Goal

Prove a compiler *preserves proofs* of concurrent value-dependent information-flow security

Plan: Use *confidentiality-preserving refinement*



Our contributions

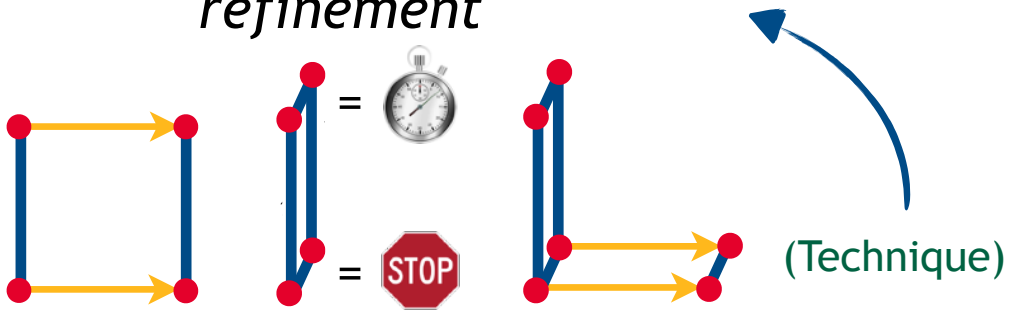
Goal

Prove a compiler *preserves proofs* of concurrent value-dependent information-flow security

Results

1. Decomposition principle

for *confidentiality-preserving refinement*



Our contributions

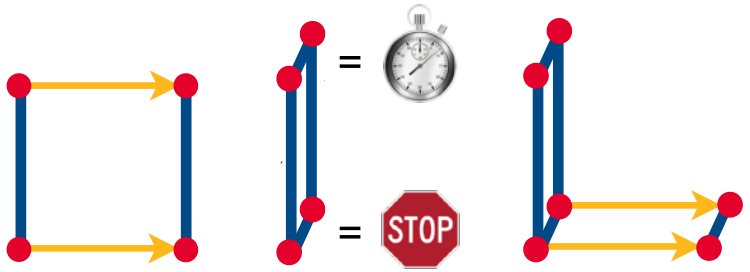
Goal

Prove a compiler *preserves proofs* of concurrent value-dependent information-flow security

Results

1. Decomposition principle

for *confidentiality-preserving refinement*



(Technique)

2. Verified compiler

While-language to RISC-style assembly

(Proof-of-concept for technique)

Impact

1st such proofs carried to assembly-level model by compiler

Our contributions

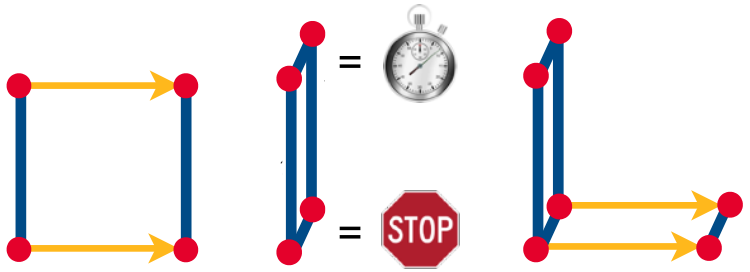
Goal

Prove a compiler *preserves proofs* of concurrent value-dependent information-flow security

Results

1. Decomposition principle

for *confidentiality-preserving refinement*



(Technique)

2. Verified compiler

While-language to RISC-style assembly

(Proof-of-concept for technique)

Impact

1st such proofs carried to assembly-level model by compiler

Our contributions

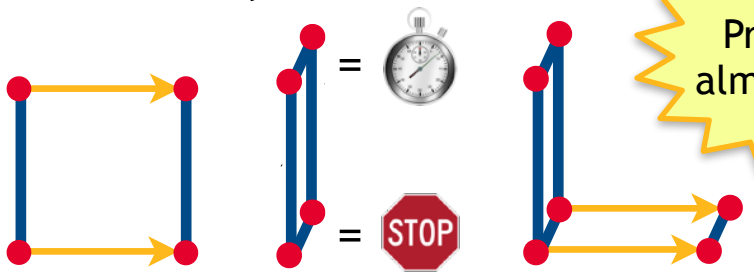
Goal

Prove a compiler *preserves proofs* of concurrent value-dependent information-flow security

Results

1. Decomposition principle

for *confidentiality-preserving refinement*



Proof effort almost *halved*!

(Technique)

2. Verified compiler

While-language to RISC-style assembly

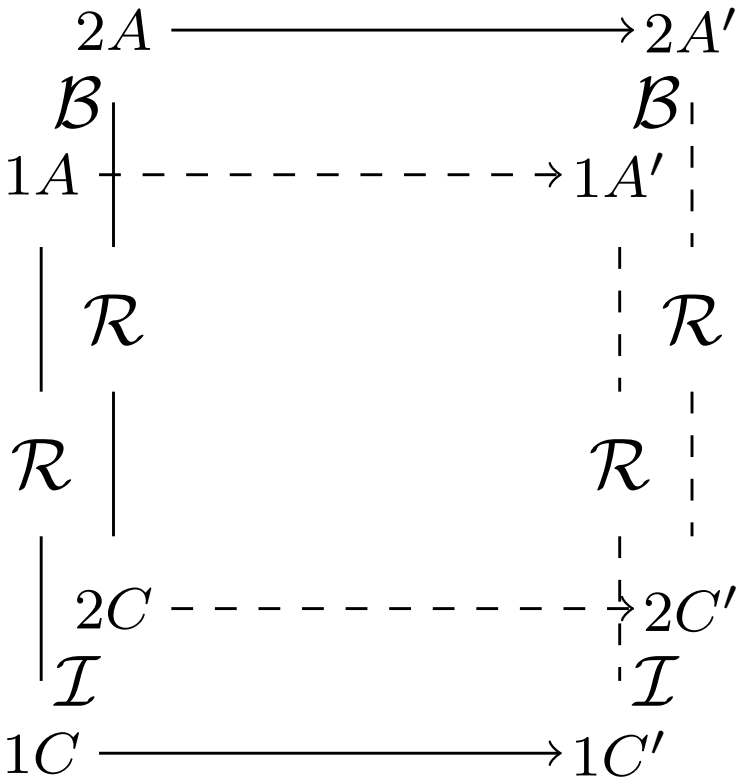
(Proof-of-concept for technique)

Impact

1st such proofs carried to assembly-level model by compiler

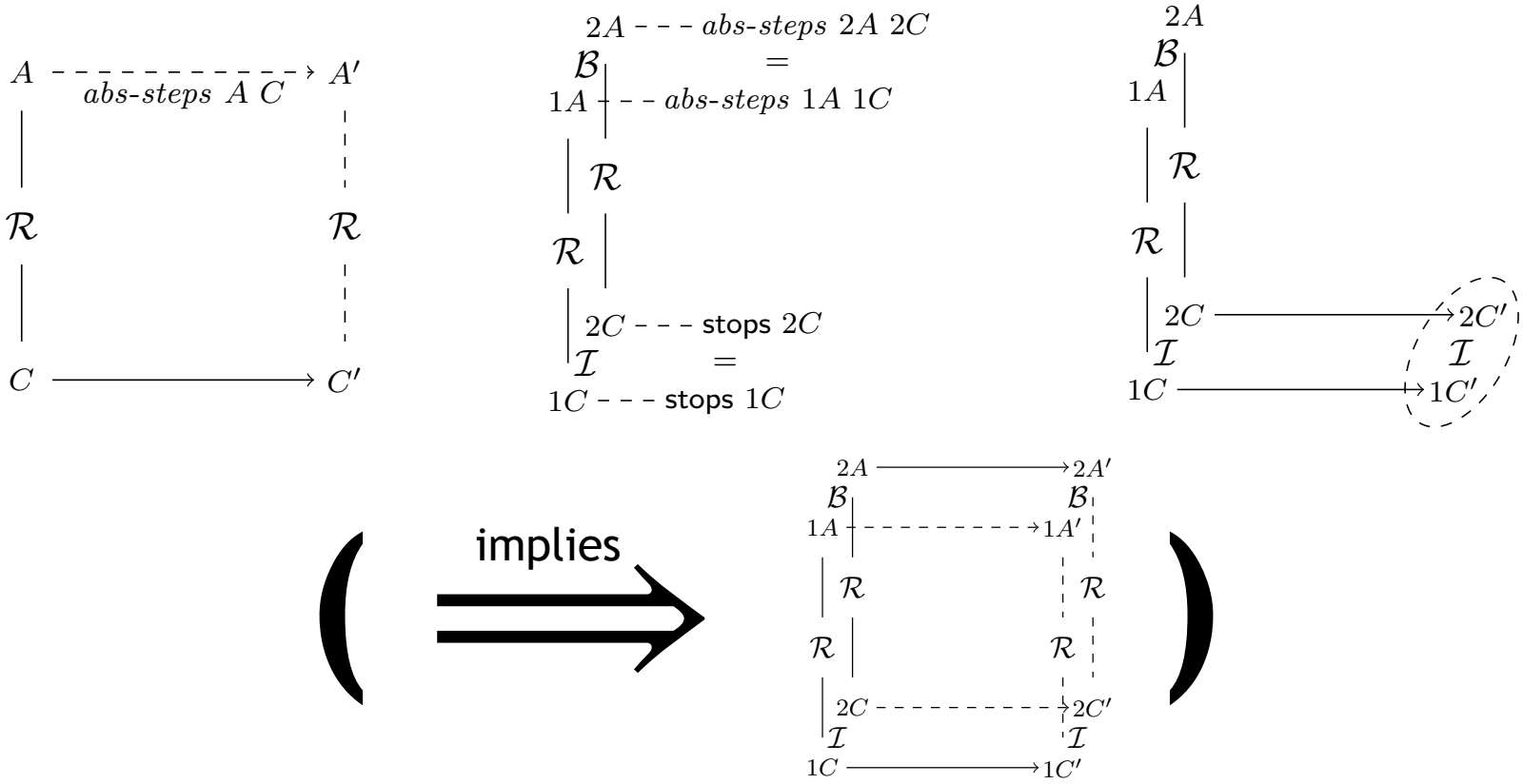
The “cube”, decomposed

Simpler confidentiality-preserving refinement



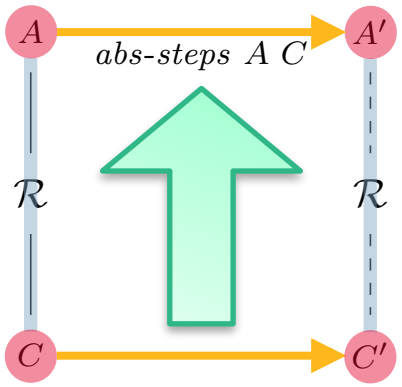
The “cube”, decomposed

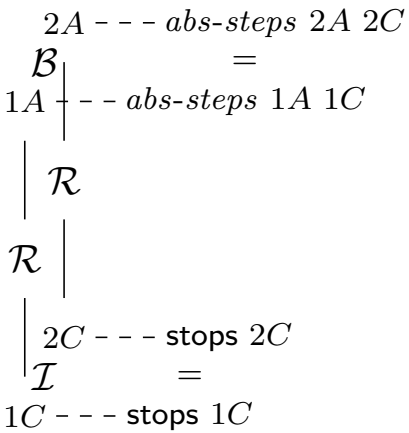
Simpler confidentiality-preserving refinement

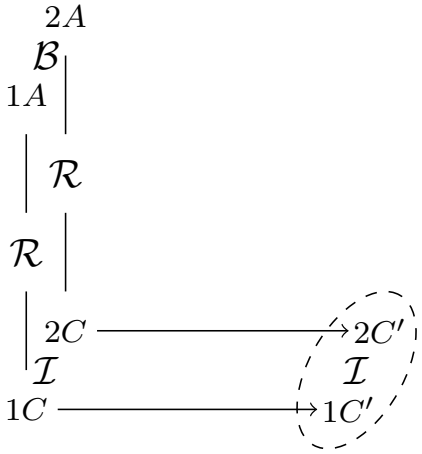


The “cube”, decomposed

Simpler confidentiality-preserving refinement



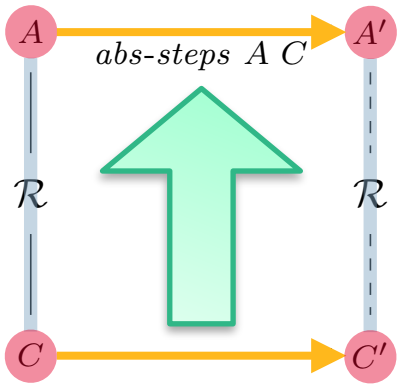


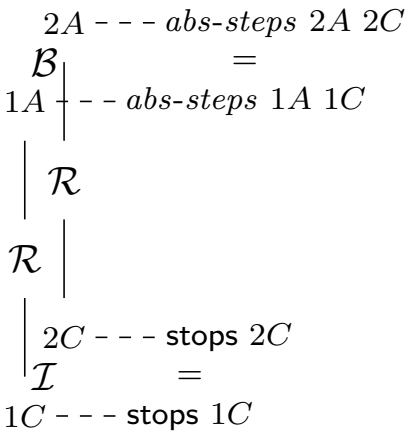


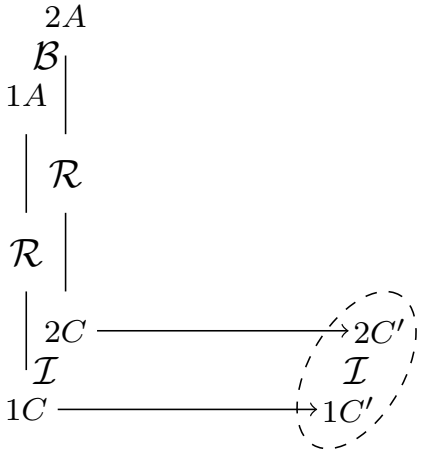
1. “Usual” proof of refinement

The “cube”, decomposed

Simpler confidentiality-preserving refinement







1. “Usual” proof of refinement

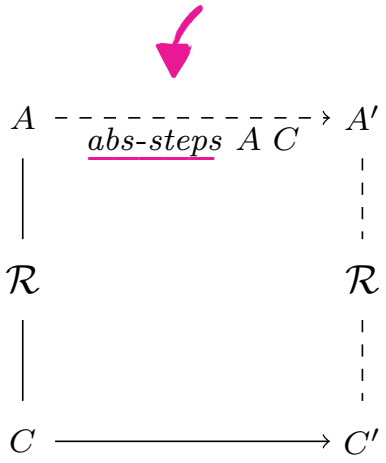
Standard compiler correctness!

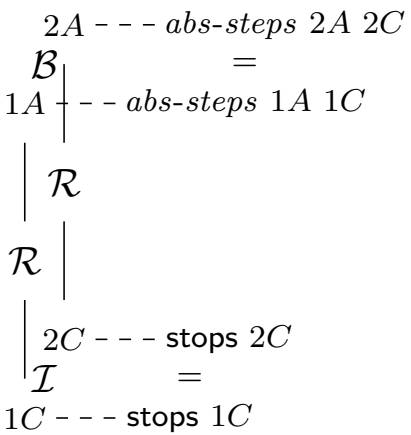
(+ “extra stuff” for conc, val-dep)

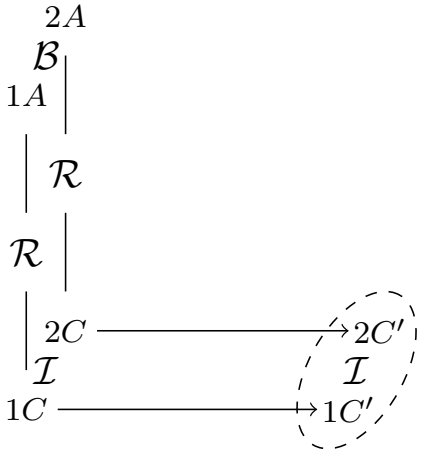
The “cube”, decomposed

Simpler confidentiality-preserving refinement

“Pacing function” *abs-steps*
for (refinement) relation R







1. “Usual” proof of refinement

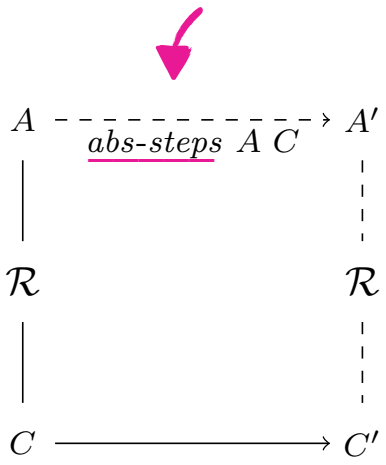
Standard compiler correctness!

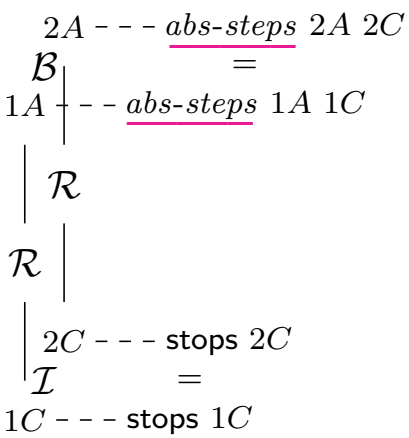
(+ “extra stuff” for conc, val-dep)

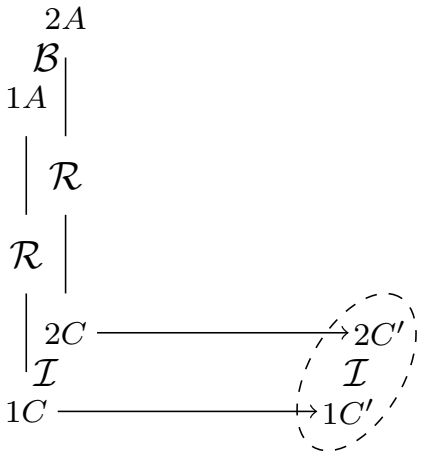
The “cube”, decomposed

Simpler confidentiality-preserving refinement

“Pacing function” *abs-steps*
for (refinement) relation R







1. “Usual” proof of refinement

2. Consistent pacing

Standard compiler correctness!

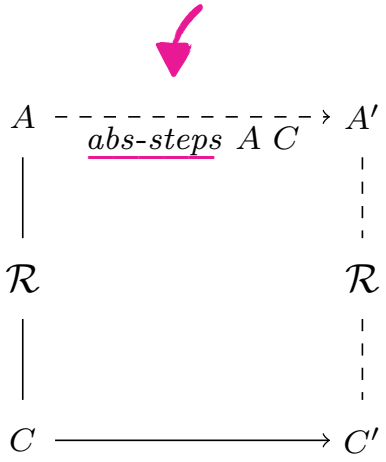
(+ “extra stuff” for conc, val-dep)

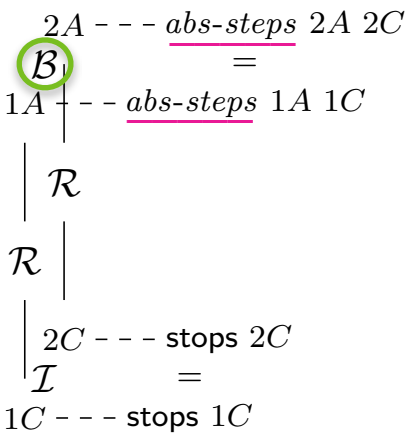
The “cube”, decomposed

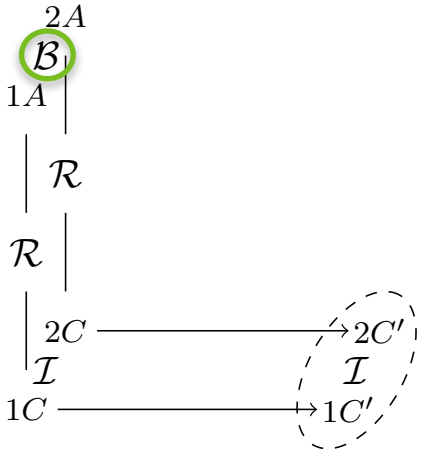
Simpler confidentiality-preserving refinement

“Pacing function” *abs-steps* for (refinement) relation R

Security witness (bisimulation) relation B







1. “Usual” proof of refinement

2. Consistent pacing

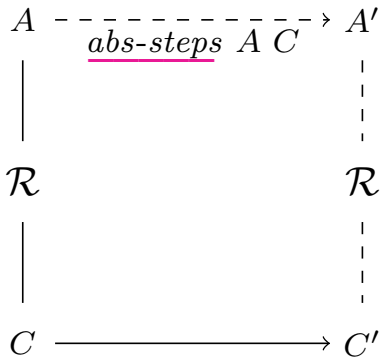
Standard compiler correctness!

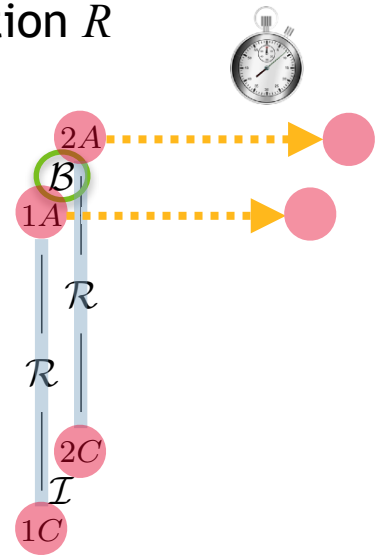
(+ “extra stuff” for conc, val-dep)

The “cube”, decomposed

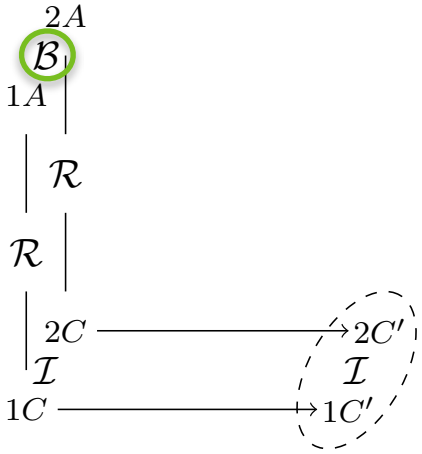
Simpler confidentiality-preserving refinement

“Pacing function” *abs-steps*
for (refinement) relation R





Security witness
(bisimulation) relation B



1. “Usual” proof of refinement

2. Consistent pacing

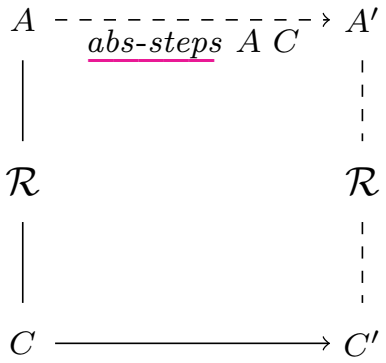
Standard compiler correctness!

(+ “extra stuff” for conc, val-dep)

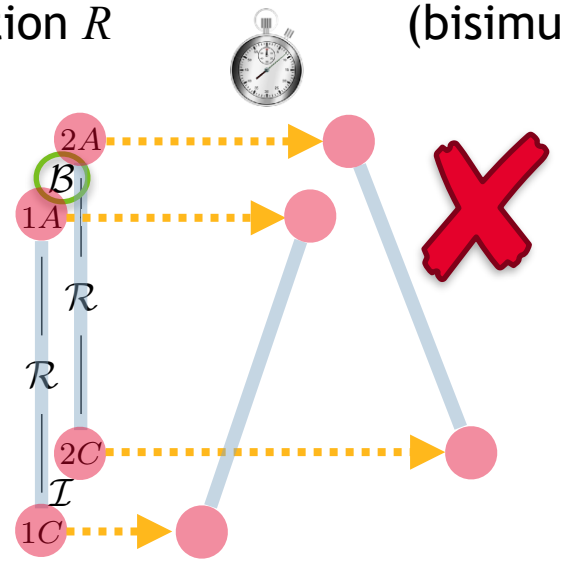
The “cube”, decomposed

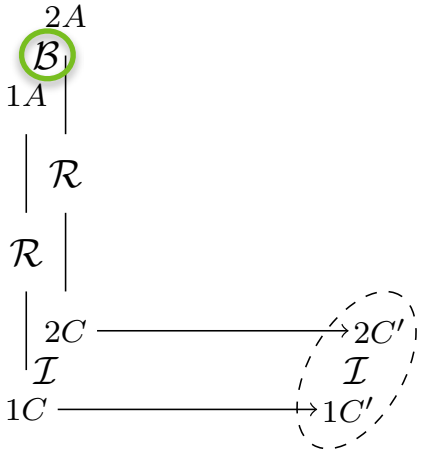
Simpler confidentiality-preserving refinement

“Pacing function” *abs-steps*
for (refinement) relation R



Security witness
(bisimulation) relation B





1. “Usual” proof of refinement

2. Consistent pacing

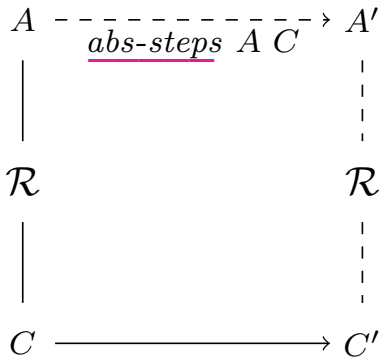
Standard compiler correctness!

(+ “extra stuff” for conc, val-dep)

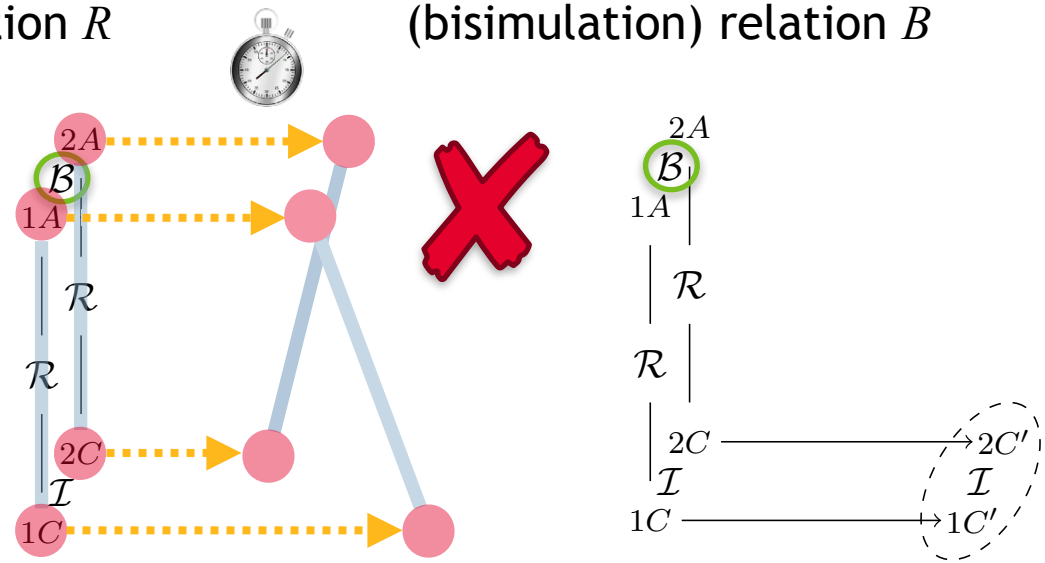
The “cube”, decomposed

Simpler confidentiality-preserving refinement

“Pacing function” *abs-steps* for (refinement) relation R



Security witness (bisimulation) relation B



1. “Usual” proof of refinement

2. Consistent pacing

Standard compiler correctness!

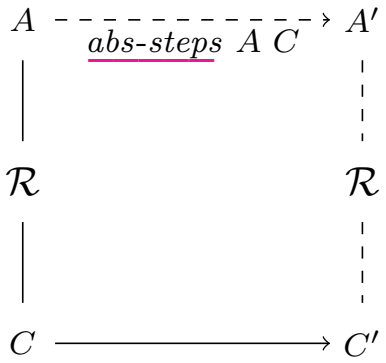
(+ “extra stuff” for conc, val-dep)

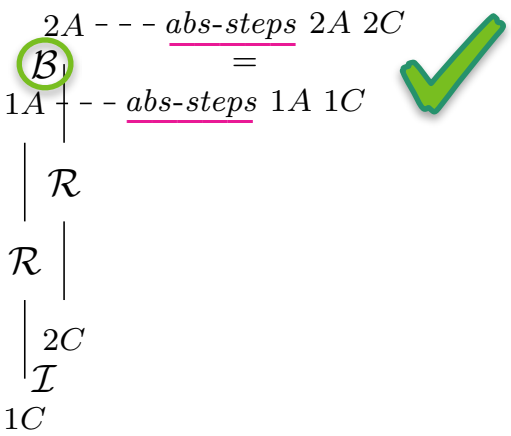
The “cube”, decomposed

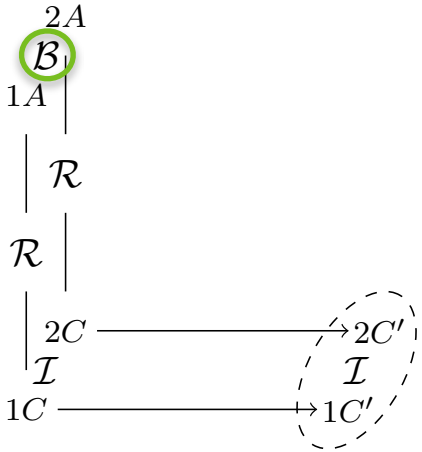
Simpler confidentiality-preserving refinement

“Pacing function” *abs-steps* for (refinement) relation R

Security witness (bisimulation) relation B







1. “Usual” proof of refinement

2. Consistent pacing

Standard compiler correctness!

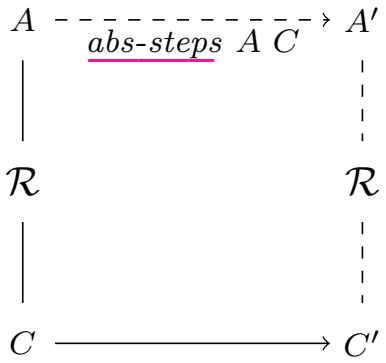
(+ “extra stuff” for conc, val-dep)

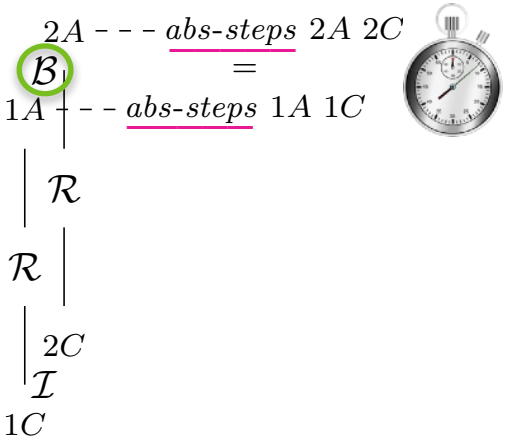
The “cube”, decomposed

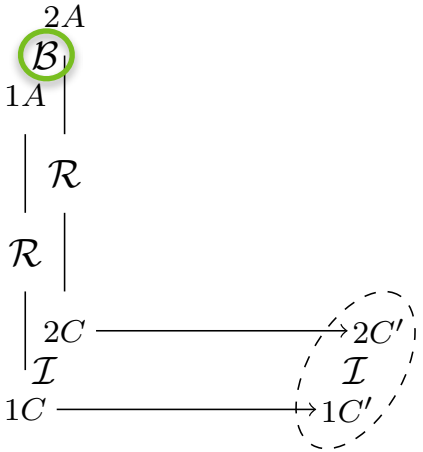
Simpler confidentiality-preserving refinement

“Pacing function” *abs-steps* for (refinement) relation R

Security witness (bisimulation) relation B







1. “Usual” proof of refinement

2. Consistent pacing

Standard compiler correctness!

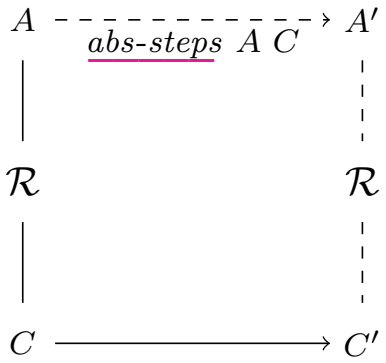
(+ “extra stuff” for conc, val-dep)

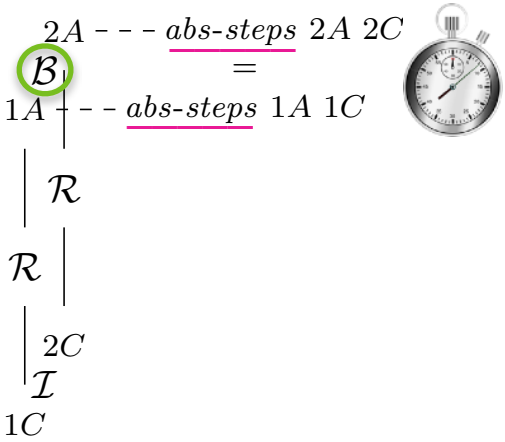
The “cube”, decomposed

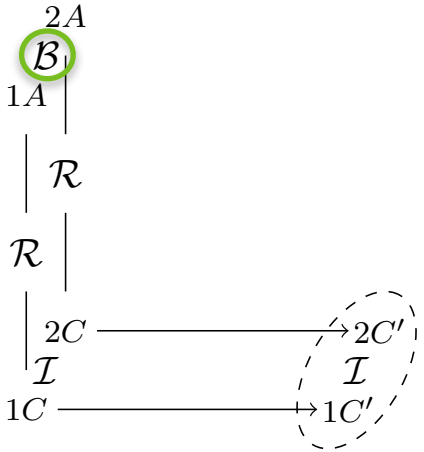
Simpler confidentiality-preserving refinement

“Pacing function” *abs-steps* for (refinement) relation R

Security witness (bisimulation) relation B







1. “Usual” proof of refinement

2. Consistent pacing
and
3. Consistent stopping

Standard compiler correctness!

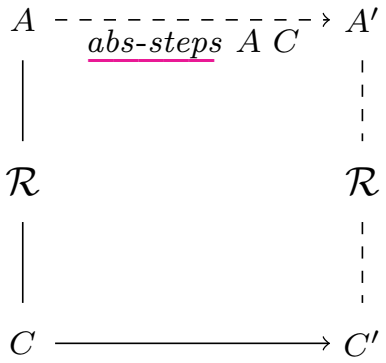
(+ “extra stuff” for conc, val-dep)

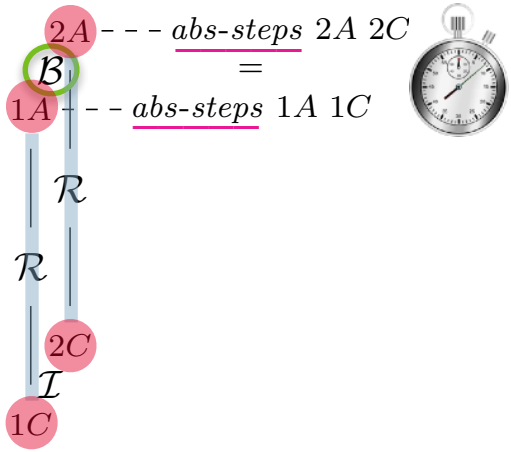
The “cube”, decomposed

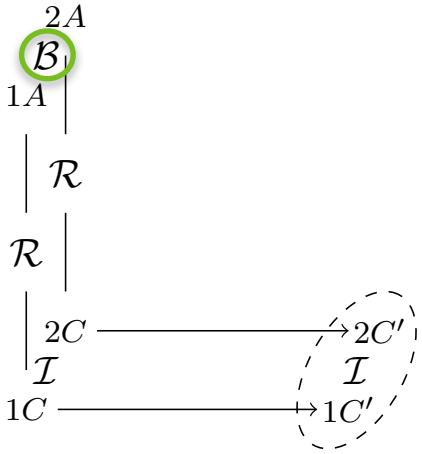
Simpler confidentiality-preserving refinement

“Pacing function” *abs-steps* for (refinement) relation R

Security witness (bisimulation) relation B







1. “Usual” proof of refinement

2. Consistent pacing
and
3. Consistent stopping

Standard compiler correctness!

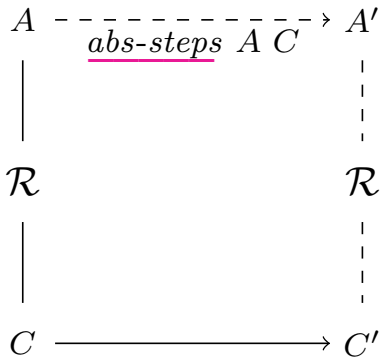
(+ “extra stuff” for conc, val-dep)

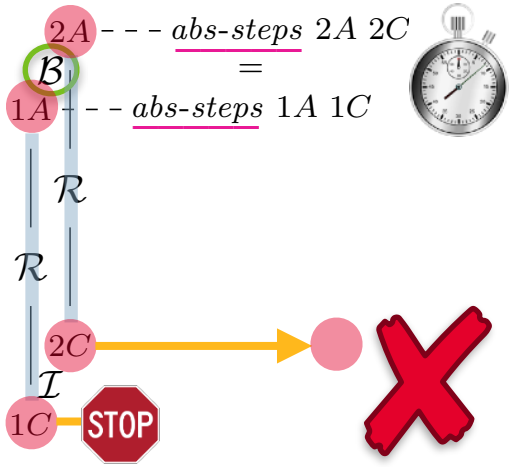
The “cube”, decomposed

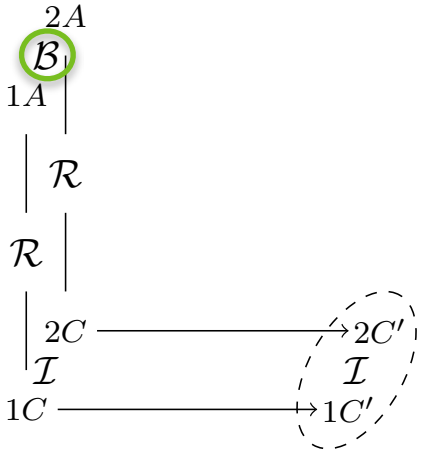
Simpler confidentiality-preserving refinement

“Pacing function” *abs-steps* for (refinement) relation R

Security witness (bisimulation) relation B







1. “Usual” proof of refinement

2. Consistent pacing
and
3. Consistent stopping

Standard compiler correctness!

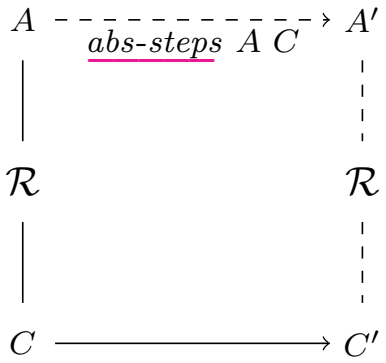
(+ “extra stuff” for conc, val-dep)

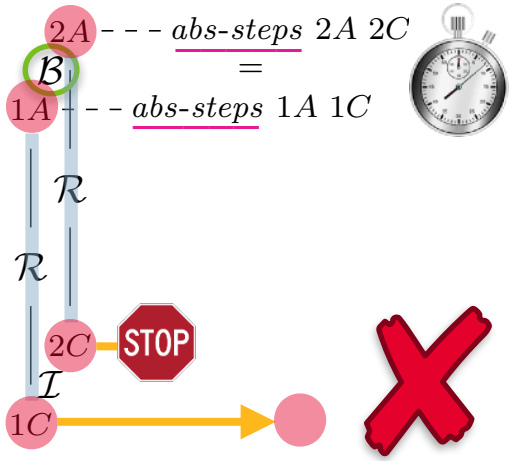
The “cube”, decomposed

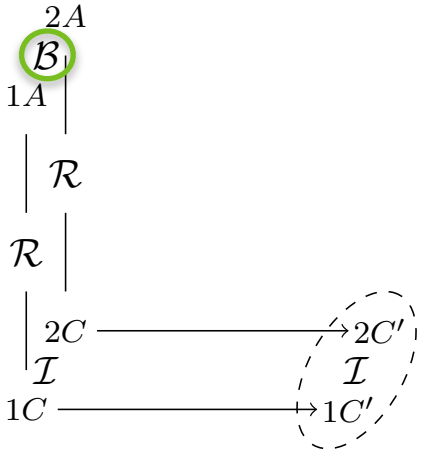
Simpler confidentiality-preserving refinement

“Pacing function” *abs-steps* for (refinement) relation R

Security witness (bisimulation) relation B







1. “Usual” proof of refinement

2. Consistent pacing
and
3. Consistent stopping

Standard compiler correctness!

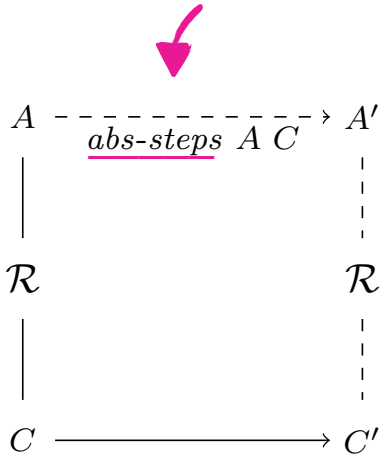
(+ “extra stuff” for conc, val-dep)

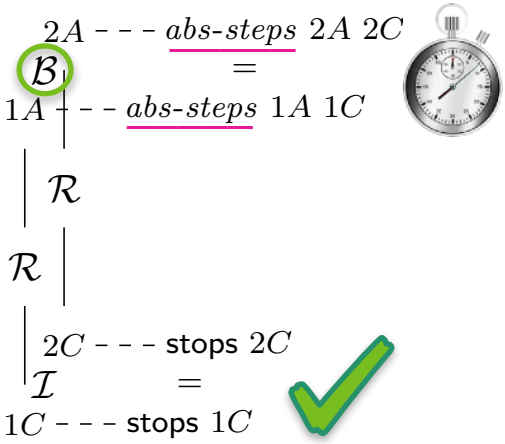
The “cube”, decomposed

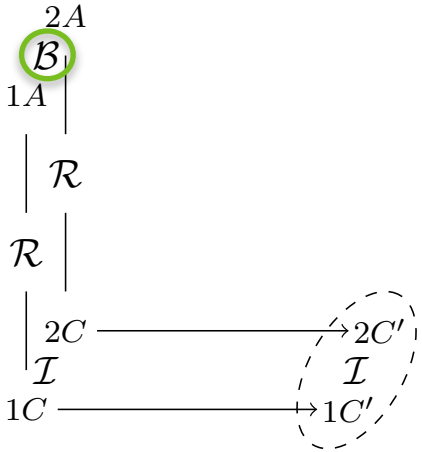
Simpler confidentiality-preserving refinement

“Pacing function” *abs-steps* for (refinement) relation R

Security witness (bisimulation) relation B







1. “Usual” proof of refinement

2. Consistent pacing
and
3. Consistent stopping

Standard compiler correctness!

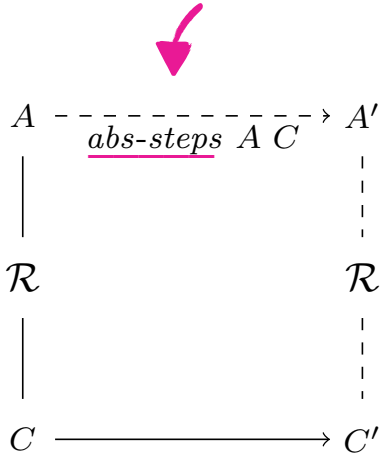
(+ “extra stuff” for conc, val-dep)

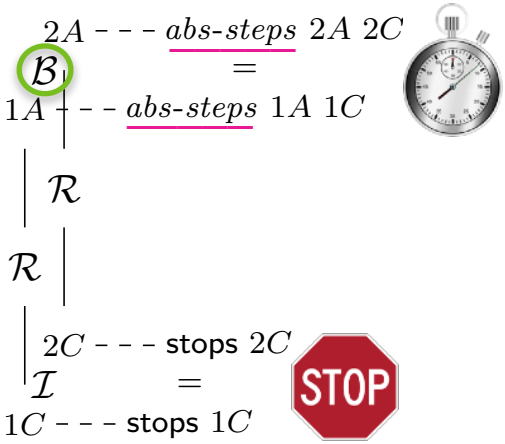
The “cube”, decomposed

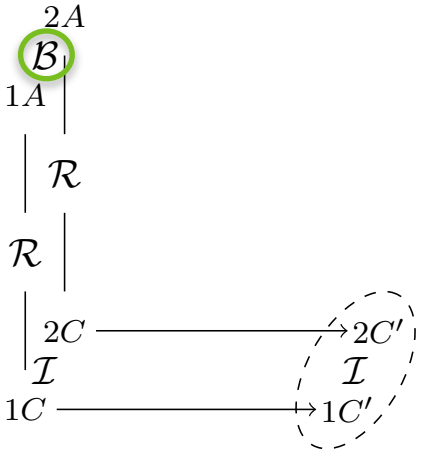
Simpler confidentiality-preserving refinement

“Pacing function” *abs-steps* for (refinement) relation R

Security witness (bisimulation) relation B







1. “Usual” proof of refinement

2. Consistent pacing
and
3. Consistent stopping

Standard compiler correctness!

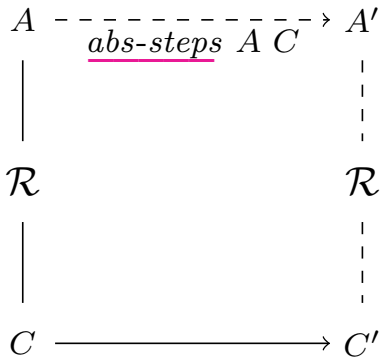
(+ “extra stuff” for conc, val-dep)

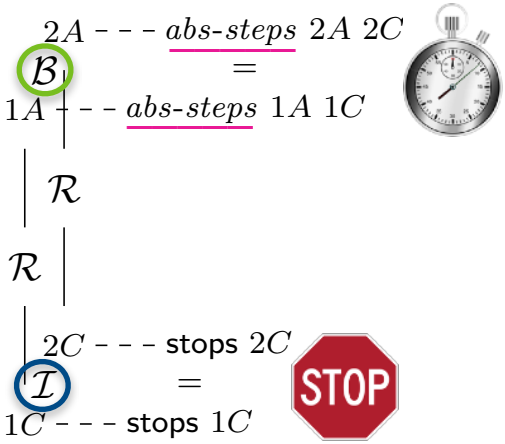
The “cube”, decomposed

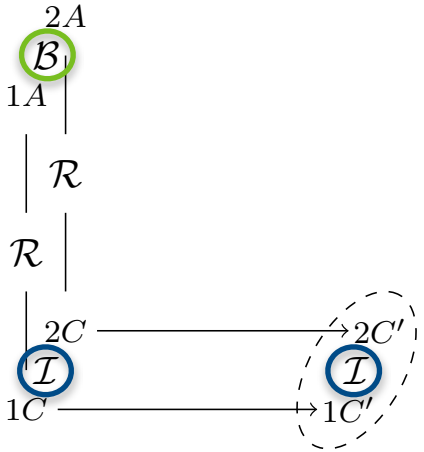
Simpler confidentiality-preserving refinement

“Pacing function” *abs-steps* for (refinement) relation R

Security witness (bisimulation) relation B







1. “Usual” proof of refinement

2. Consistent pacing
and
3. Consistent stopping

4. *Closedness* of “Concrete coupling invariant” relation I

Standard compiler correctness!

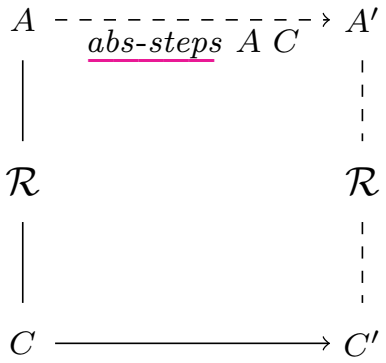
(+ “extra stuff” for conc, val-dep)

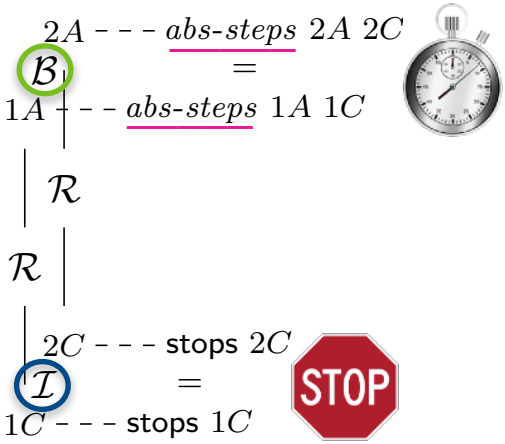
The “cube”, decomposed

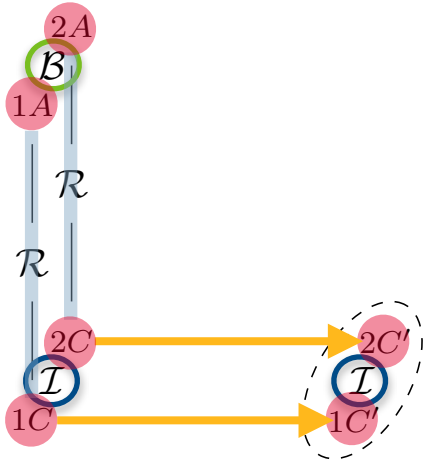
Simpler confidentiality-preserving refinement

“Pacing function” *abs-steps* for (refinement) relation R

Security witness (bisimulation) relation B







1. “Usual” proof of refinement

2. Consistent pacing
and
3. Consistent stopping

4. *Closedness* of “Concrete coupling invariant” relation I

Standard compiler correctness!

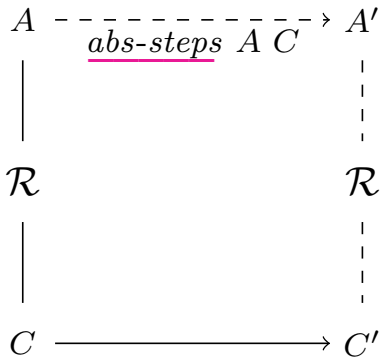
(+ “extra stuff” for conc, val-dep)

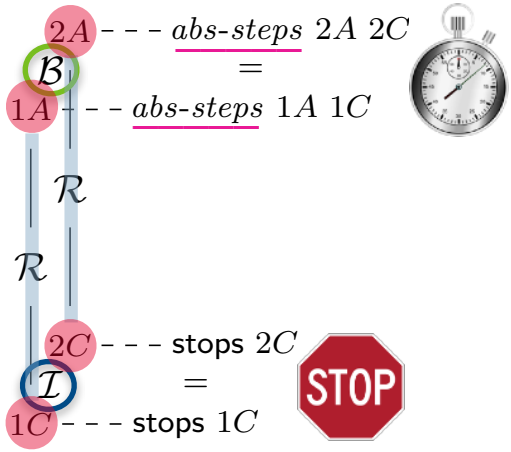
The “cube”, decomposed

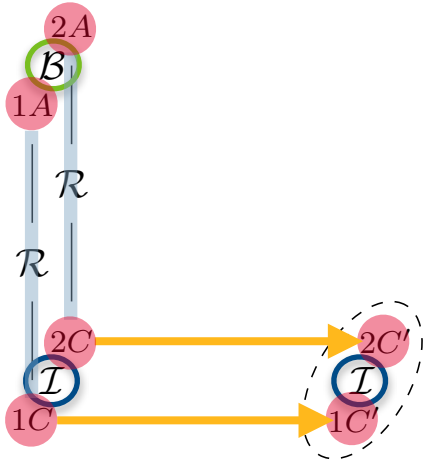
Simpler confidentiality-preserving refinement

“Pacing function” *abs-steps* for (refinement) relation R

Security witness (bisimulation) relation B







1. “Usual” proof of refinement

2. Consistent pacing and 3. Consistent stopping

4. *Closedness* of “Concrete coupling invariant” relation I

Standard compiler correctness!

(+ “extra stuff” for conc, val-dep)

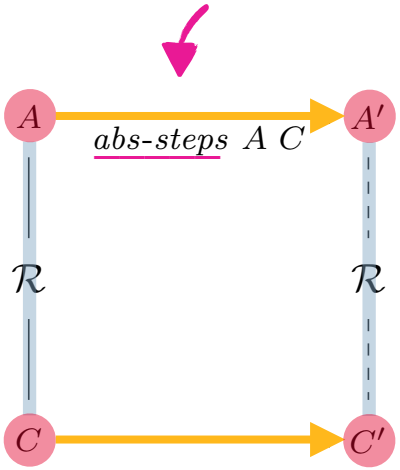
No new timing and termination leaks!

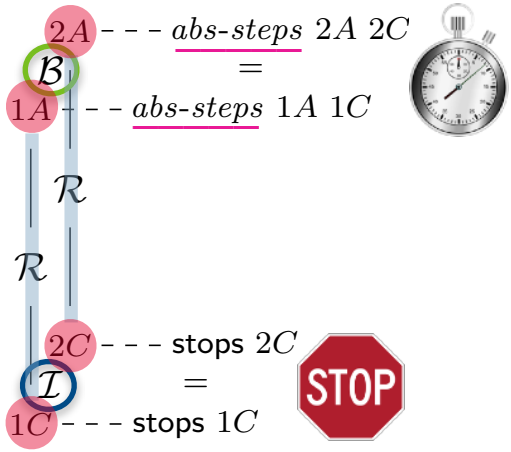
The “cube”, decomposed

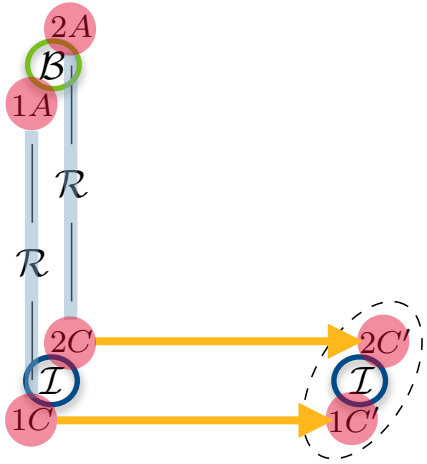
Simpler confidentiality-preserving refinement

“Pacing function” *abs-steps* for (refinement) relation R

Security witness (bisimulation) relation B







1. “Usual” proof of refinement

2. Consistent pacing
and
3. Consistent stopping

4. *Closedness* of “Concrete coupling invariant” relation I

Standard compiler correctness!

(+ “extra stuff” for conc, val-dep)

No new timing and termination leaks!

Proof effort comparison

Refinement example (excerpt) from CSF'16

```
if h ≠ 0 then
    x := y
else
    x := y + z
fi
```

Abstract program

```
reg3 := h;
if reg3 ≠ 0 then
    skip;
    skip;
    reg0 := y;
    x := reg0
else
    reg1 := y;
    reg2 := z;
    reg0 := reg1 + reg2;
    x := reg0
fi
```

Concrete program

formalisation artifact:
https://covern.org/itp19.html

Proof effort comparison

Refinement example (excerpt) from CSF'16

branch on secret

```
if h ≠ 0 then
    x := y
else
    x := y + z
fi
```

Abstract program

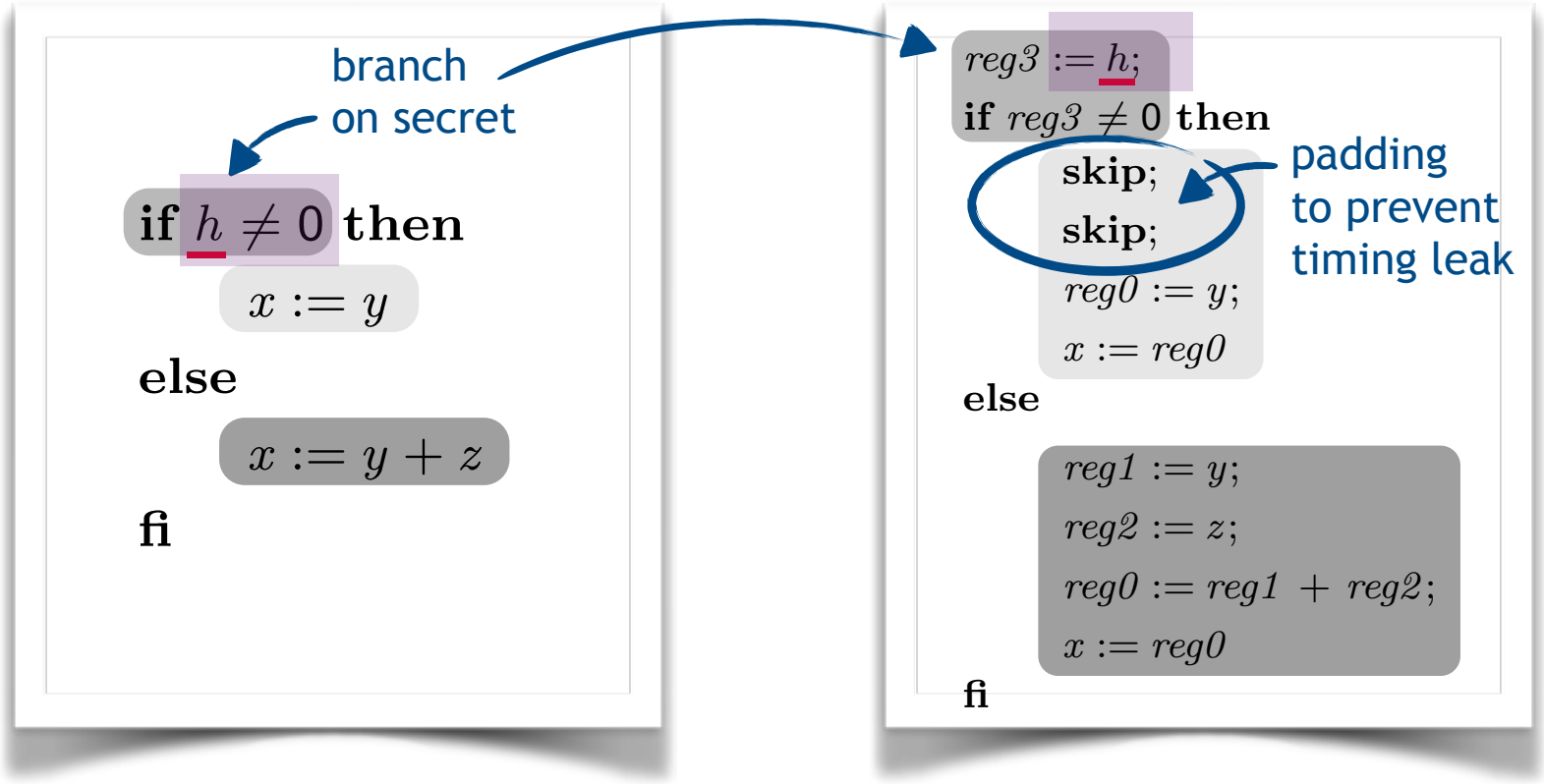
```
reg3 := h;
if reg3 ≠ 0 then
    skip;
    skip;
    reg0 := y;
    x := reg0
else
    reg1 := y;
    reg2 := z;
    reg0 := reg1 + reg2;
    x := reg0
fi
```

Concrete program

formalisation artifact:
https://covern.org/itp19.html

Proof effort comparison

Refinement example (excerpt) from CSF'16



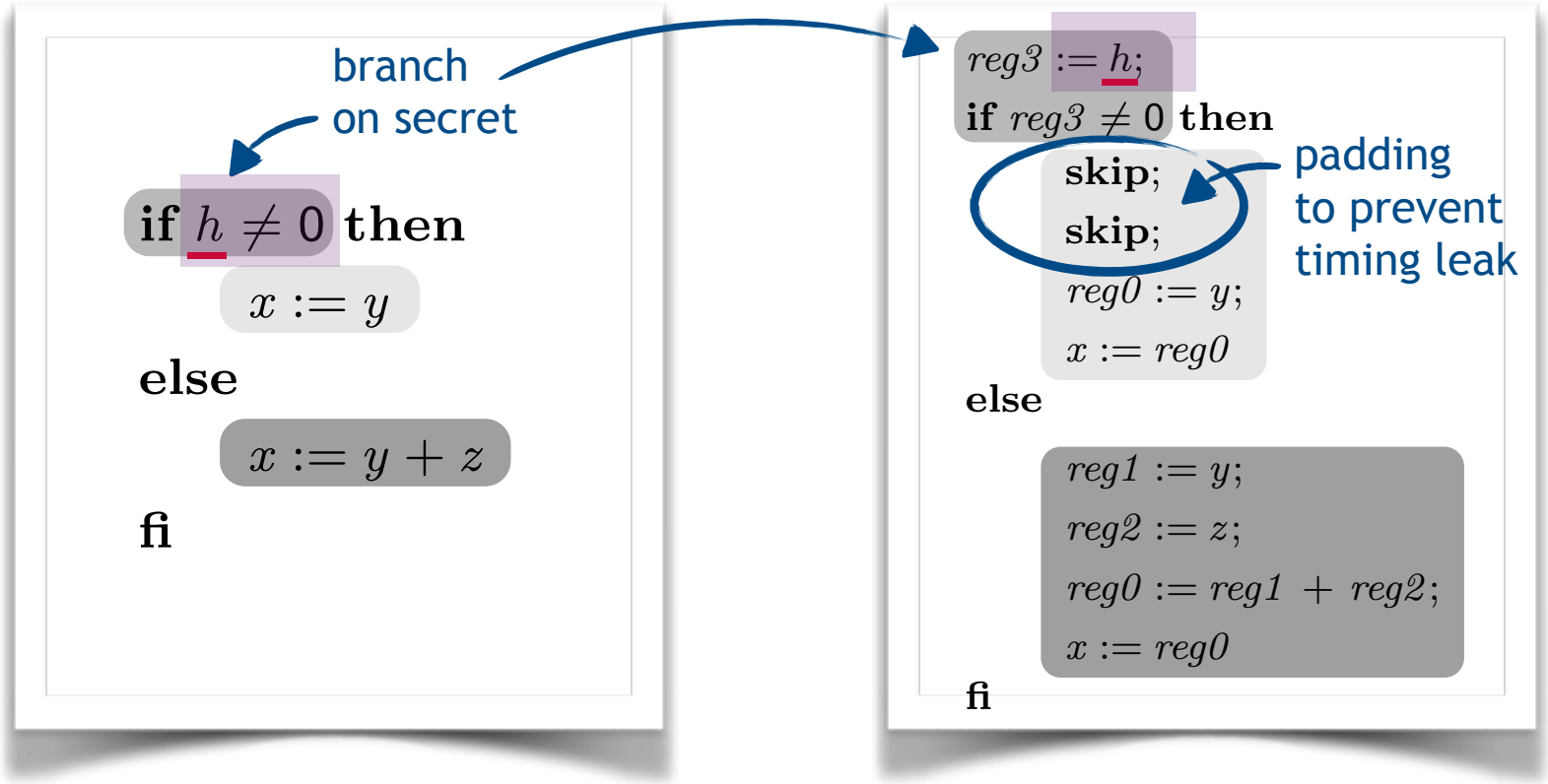
Abstract program

Concrete program

formalisation artifact:
https://covern.org/itp19.html

Proof effort comparison

Refinement example (excerpt) from CSF'16



Abstract program

Concrete program

- 44% shorter proof of secure refinement
  (~3.6K to ~2K lines of Isabelle/HOL proofs)

formalisation artifact:
https://covern.org/itp19.html

Our contributions

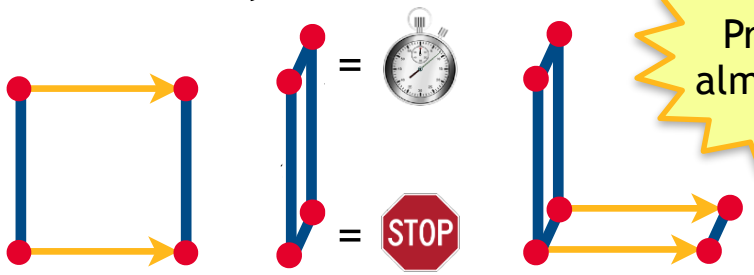
Goal

Prove a compiler *preserves proofs* of concurrent value-dependent information-flow security

Results

1. Decomposition principle

for *confidentiality-preserving refinement*



Proof effort almost *halved*!

(Technique)

2. Verified compiler

While-language to RISC-style assembly

(Proof-of-concept for technique)

Impact

1st such proofs carried to assembly-level model by compiler

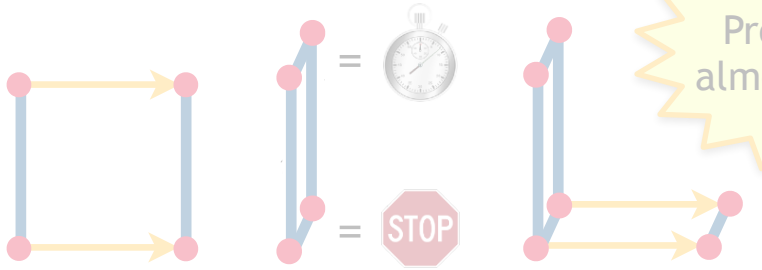
Our contributions

Goal

Prove a compiler *preserves proofs* of concurrent value-dependent information-flow security

Results

1. Decomposition principle for *confidentiality-preserving refinement*



Proof effort almost *halved*!

(Technique)

2. Verified compiler

While-language to RISC-style assembly

(Proof-of-concept for technique)

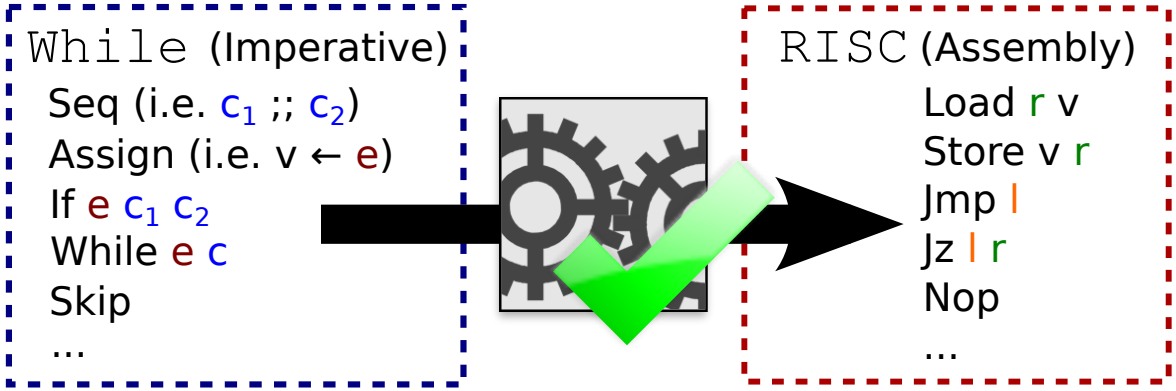
Impact

1st such proofs carried to assembly-level model by compiler

Verified compiler

Overview

An Isabelle/HOL *primrec* function



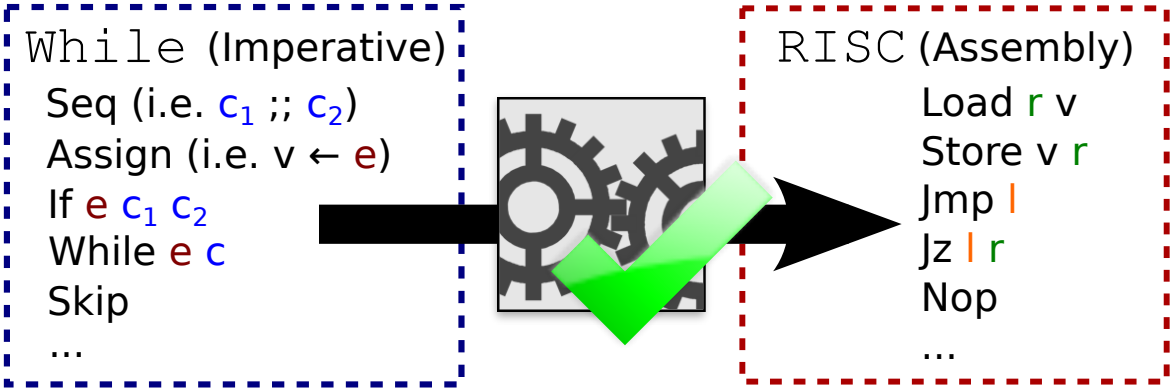
(Formalisation: https://covern.org/itp19.html)

Verified compiler

Overview

(Based on: Tedesco et al. CSF'16)

An Isabelle/HOL *primrec* function



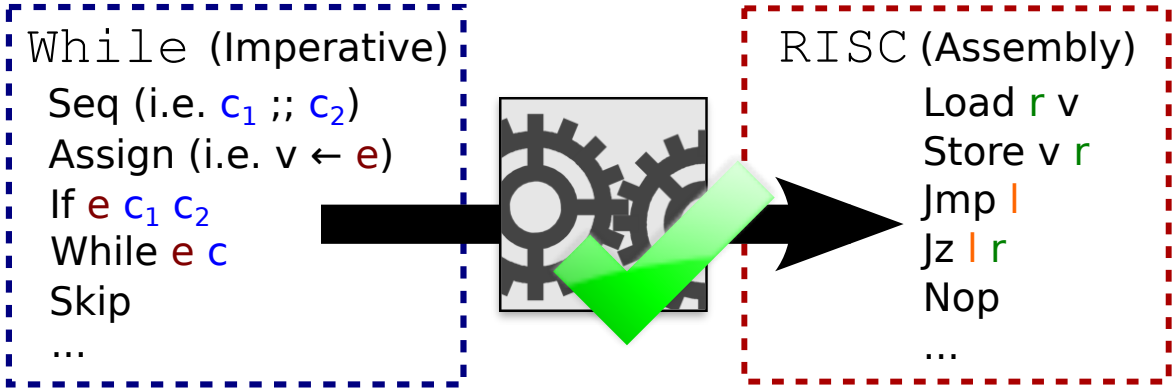
(Formalisation: https://covern.org/itp19.html)

Verified compiler

Overview

(Based on:
Tedesco et al. CSF'16)

An Isabelle/HOL *primrec* function



- Proof approach: ~7K lines of Isabelle/HOL script
  - Prevents data races on shared memory
  - Knows when safe to optimise reads

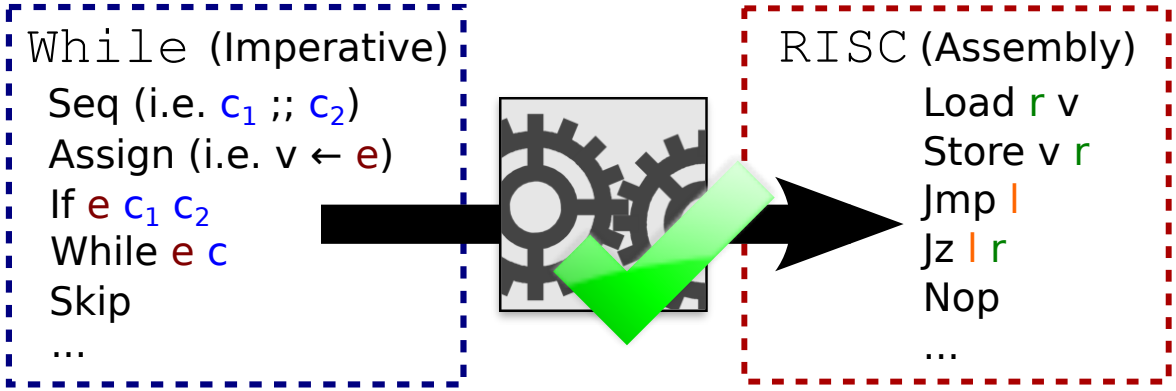
(Formalisation: https://covern.org/itp19.html)

Verified compiler

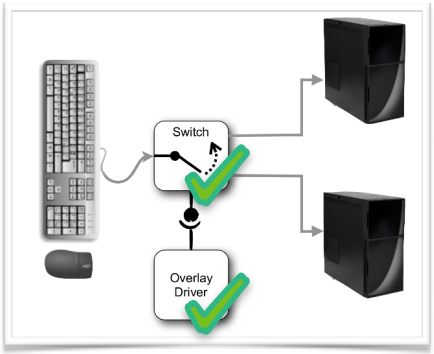
Overview

(Based on: Tedesco et al. CSF'16)

An Isabelle/HOL *primrec* function



- Proof approach: ~7K lines of Isabelle/HOL script
  - Prevents data races on shared memory
  - Knows when safe to optimise reads
- Application: 2-thread input-handling model of *Cross Domain Desktop Compositor*



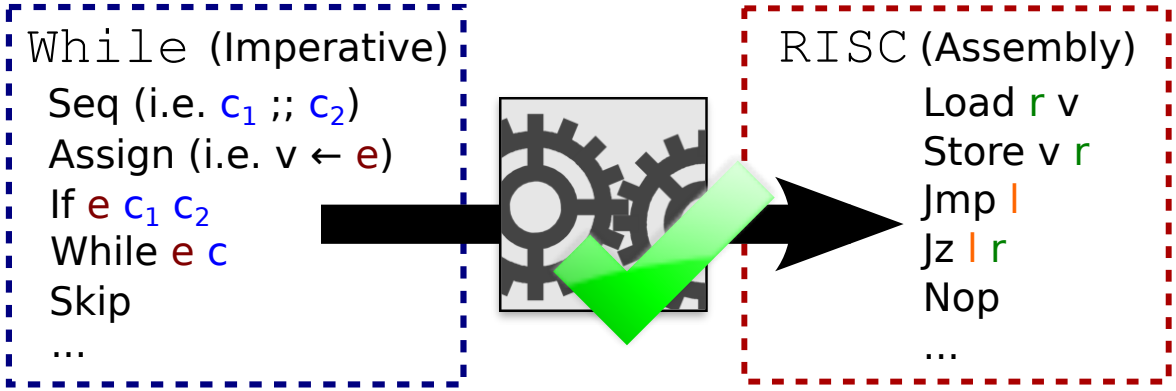
(Formalisation: https://covern.org/itp19.html)

Verified compiler

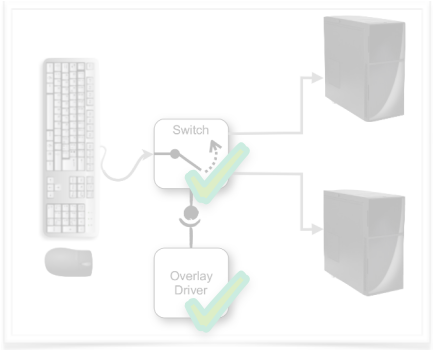
Overview

(Based on: Tedesco et al. CSF'16)

An Isabelle/HOL *primrec* function



- Proof approach: ~7K lines of Isabelle/HOL script
  - Prevents data races on shared memory
  - Knows when safe to optimise reads
- Application: 2-thread input-handling model of *Cross Domain Desktop Compositor*



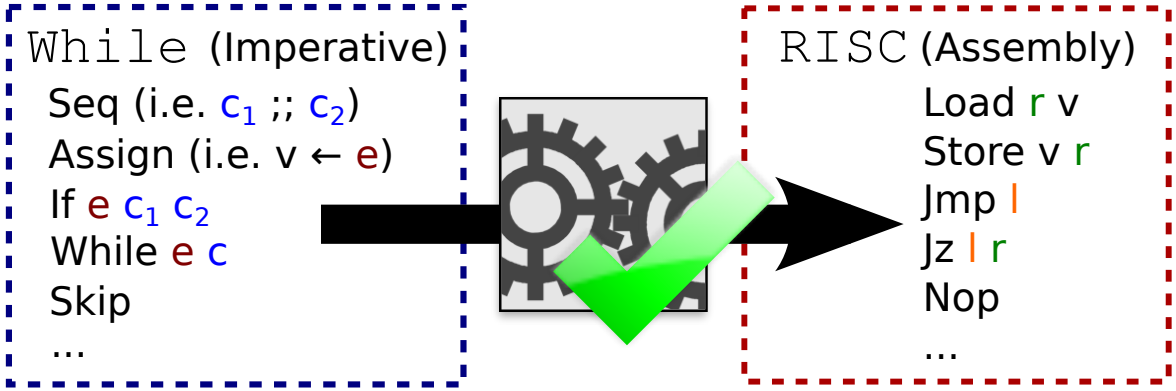
(Formalisation: https://covern.org/itp19.html)

Verified compiler

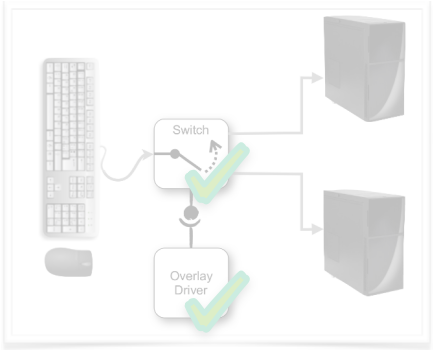
Overview

(Based on: Tedesco et al. CSF'16)

An Isabelle/HOL *primrec* function



- Proof approach: ~7K lines of Isabelle/HOL script
  - Prevents data races on shared memory
  - Knows when safe to optimise reads
- Application: 2-thread input-handling model of *Cross Domain Desktop Compositor*



(Formalisation: https://covern.org/itp19.html)

Verified compiler

Proof approach

(Formalisation: `https://covern.org/itp19.html`)

Verified compiler

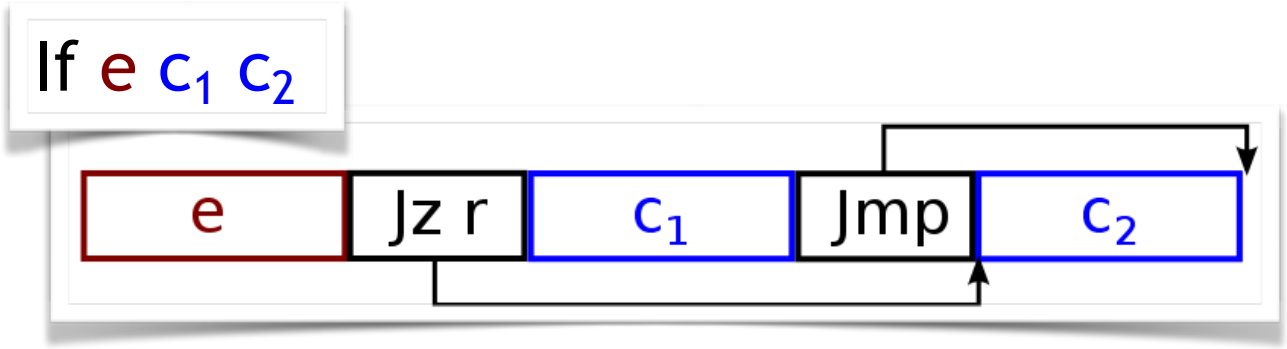
Proof approach

- Nominate R (and I) to characterise compilation (for proofs B produced by our type system)

(Formalisation: https://covern.org/itp19.html)

Verified compiler
Proof approach

- Nominate R (and I) to characterise compilation (for proofs B produced by our type system)
  - e.g. R cases for if-conditional



While

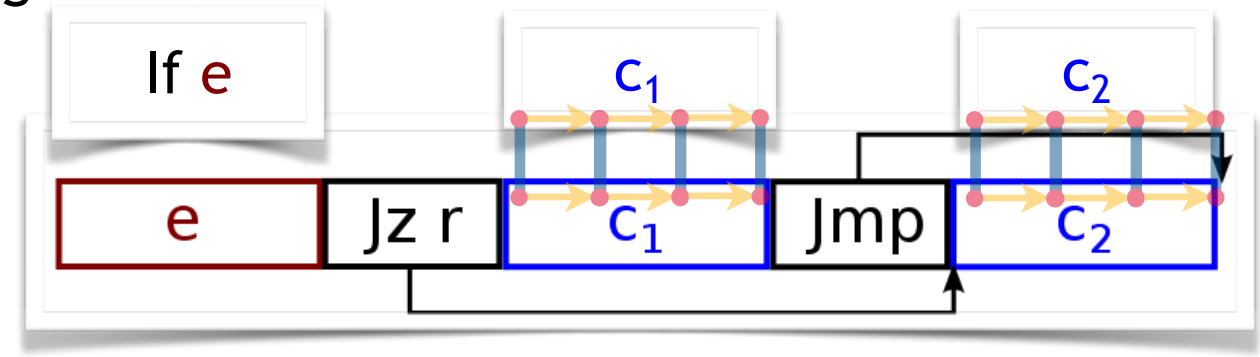
RISC

(Formalisation: https://covern.org/itp19.html)

Verified compiler

Proof approach

- Nominate R (and I) to characterise compilation (for proofs B produced by our type system)
  - e.g. R cases for if-conditional *(Inductive)*



While

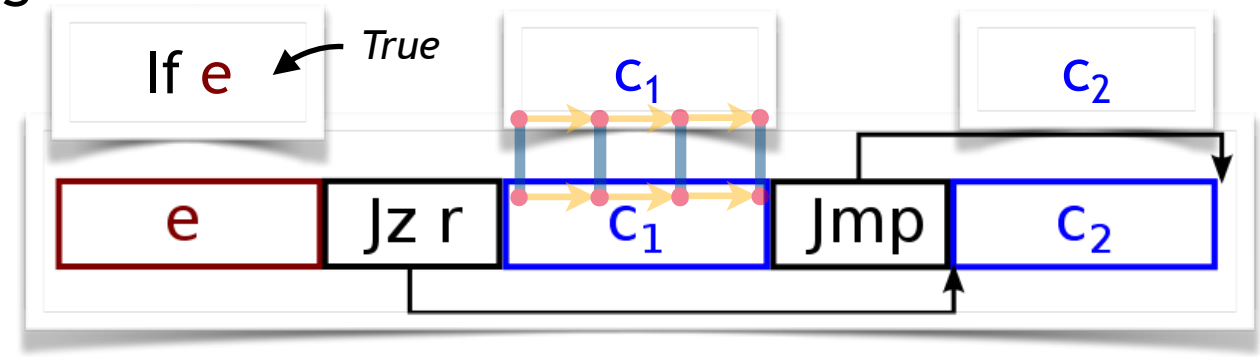
RISC

(Formalisation: https://covern.org/itp19.html)

Verified compiler

Proof approach

- Nominate R (and I) to characterise compilation (for proofs B produced by our type system)
  - e.g. R cases for if-conditional *(Inductive)*



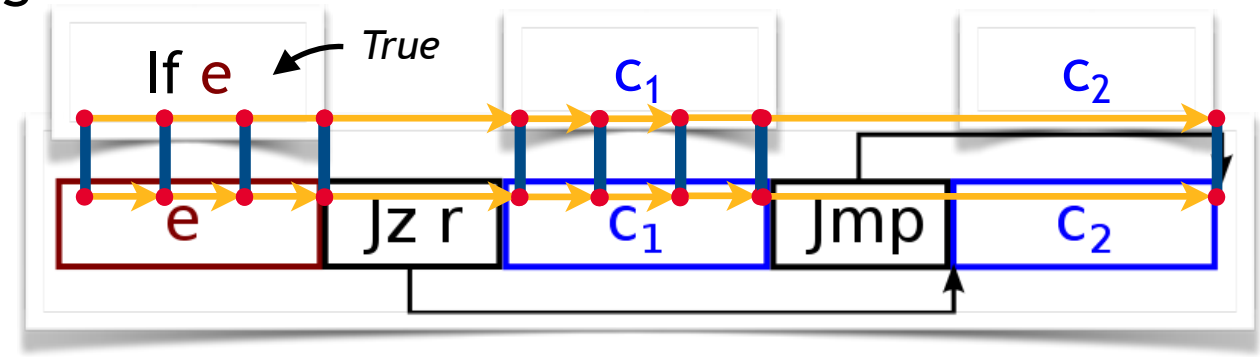
While

RISC

(Formalisation: https://covern.org/itp19.html)

Verified compiler
Proof approach

- Nominate R (and I) to characterise compilation (for proofs B produced by our type system)
  - e.g. R cases for if-conditional *(Inductive)*



While

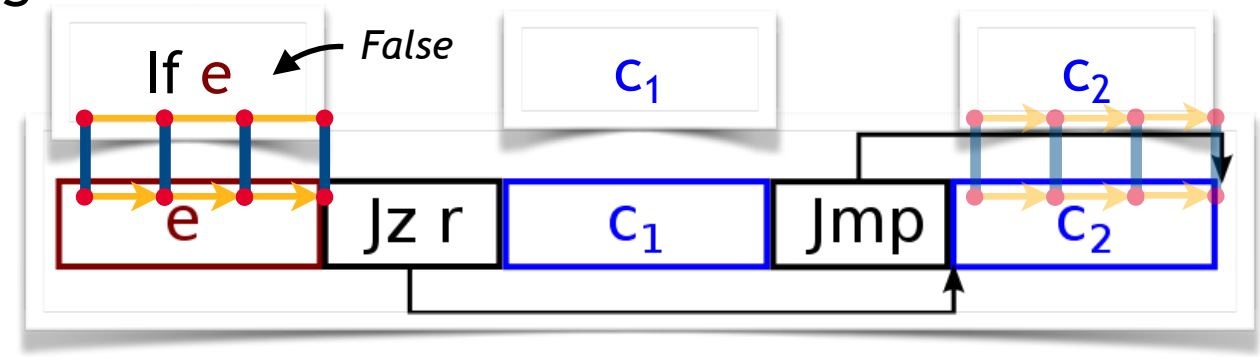
RISC

(Formalisation: https://covern.org/itp19.html)

Verified compiler

Proof approach

- Nominate R (and I) to characterise compilation (for proofs B produced by our type system)
  - e.g. R cases for if-conditional *(Inductive)*



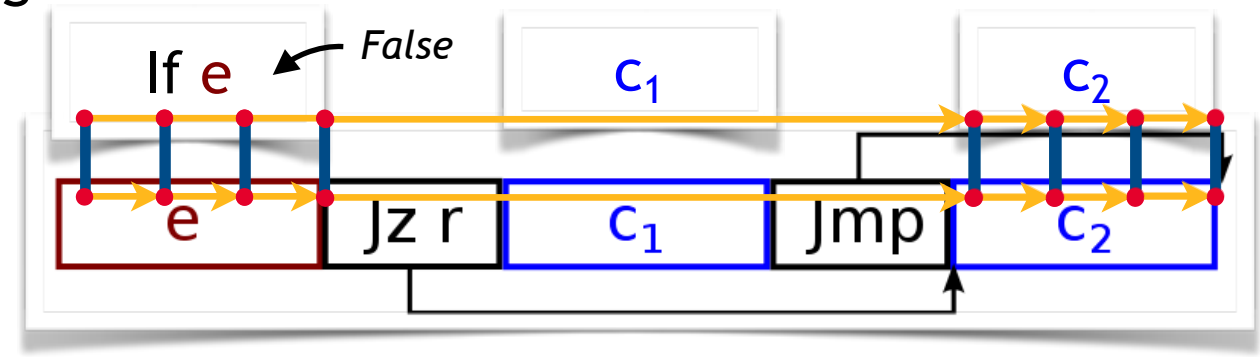
While

RISC

(Formalisation: https://covern.org/itp19.html)

Verified compiler
Proof approach

- Nominate R (and I) to characterise compilation (for proofs B produced by our type system)
  - e.g. R cases for if-conditional *(Inductive)*

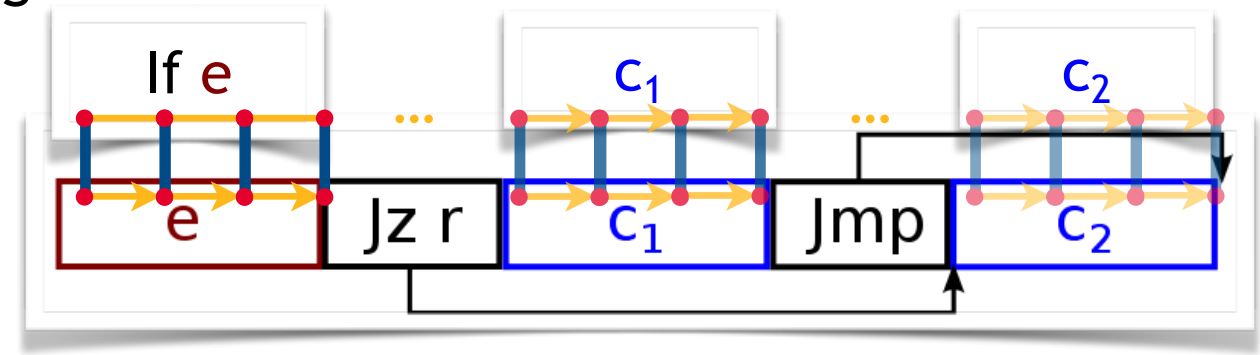


(Formalisation: https://covern.org/itp19.html)

Verified compiler

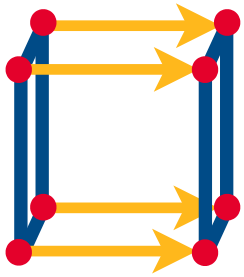
Proof approach

- Nominate R (and I) to characterise compilation (for proofs B produced by our type system)
  - e.g. R cases for if-conditional *(Inductive)*



While
↓
RISC

- Theorem: R (for B, with I) is a secure refinement



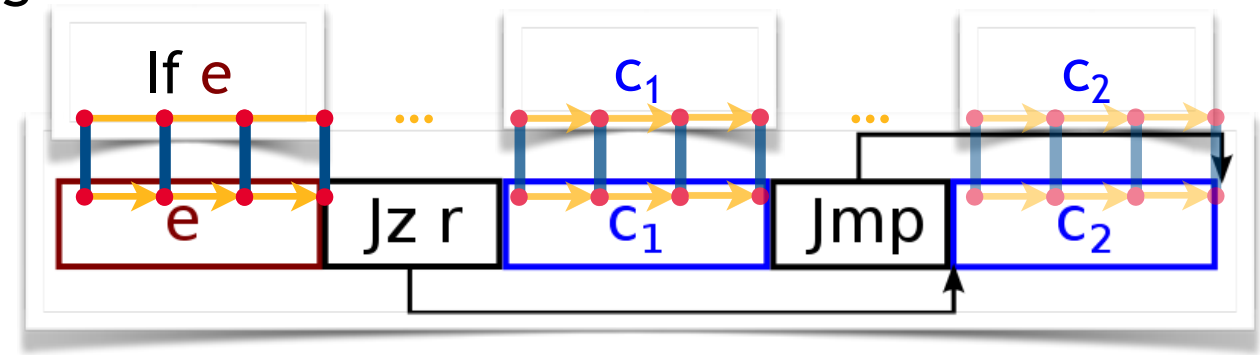
While
↓
RISC

(Formalisation: https://covern.org/itp19.html)

Verified compiler

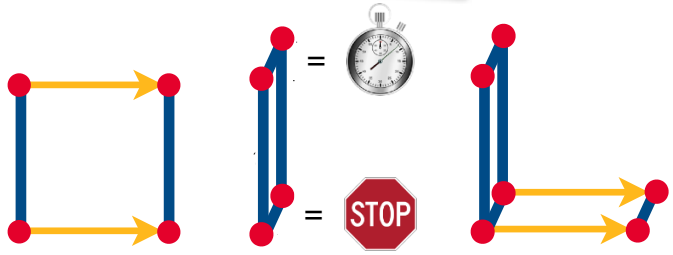
Proof approach

- Nominate R (and I) to characterise compilation (for proofs B produced by our type system)
  - e.g. R cases for if-conditional *(Inductive)*



While
RISC

- Theorem: R (for B, with I) is a secure refinement
  (via *decomposition principle*)



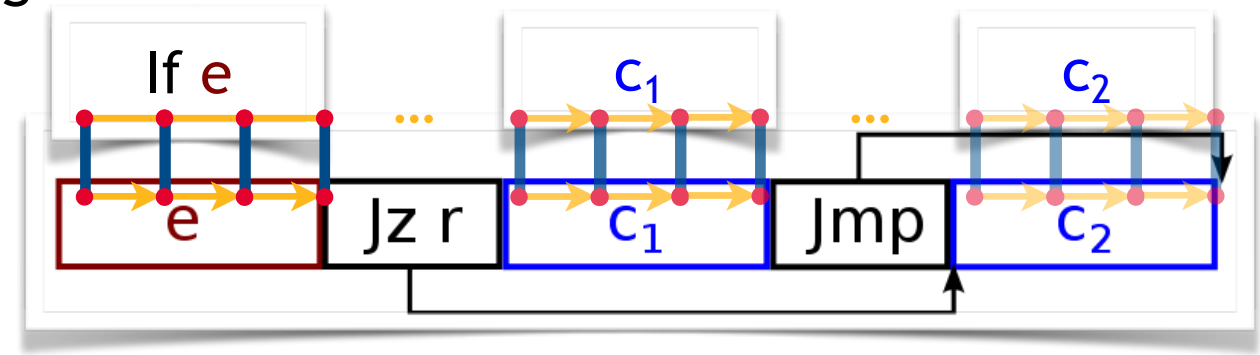
While
RISC

(Formalisation: https://covern.org/itp19.html)

Verified compiler

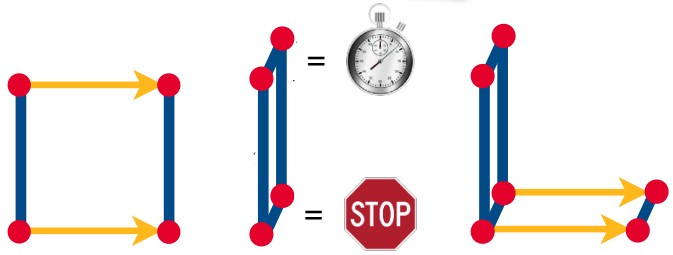
Proof approach

- Nominate R (and I) to characterise compilation (for proofs B produced by our type system)
  - e.g. R cases for if-conditional *(Inductive)*



While
RISC

- Theorem: R (for B, with I) is a secure refinement
  (via *decomposition principle*)



While
RISC

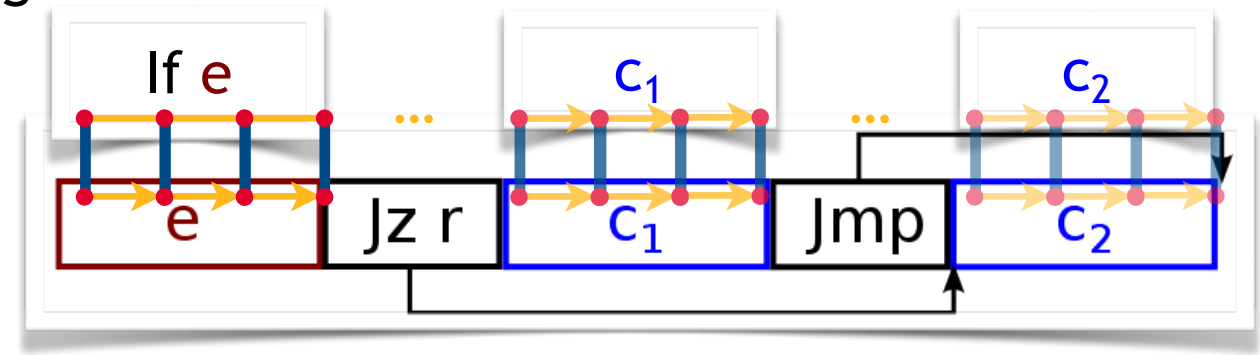
- Theorem: Compiler input related to output by R

(Formalisation: https://covern.org/itp19.html)

Verified compiler

Proof approach

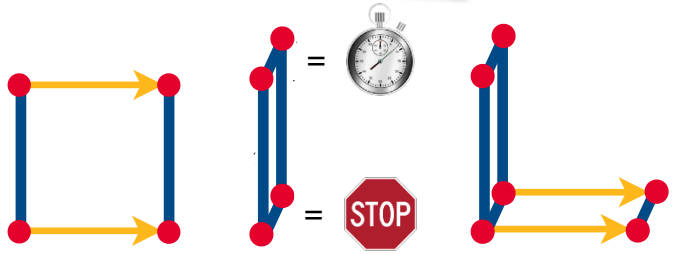
- Nominate R (and I) to characterise compilation (for proofs B produced by our type system)
  - e.g. R cases for if-conditional *(Inductive)*



While

RISC

- Theorem: R (for B, with I) is a secure refinement (via *decomposition principle*)



While

RISC

- Theorem: Compiler input related to output by R

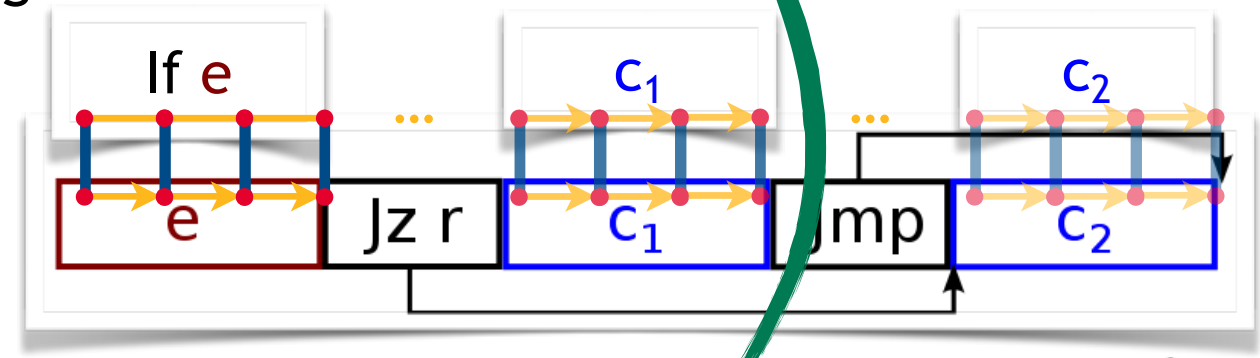
(Formalisation: https://covern.org/itp19.html)

Verified compiler

Proof approach

- Nominate R (and I) to characterise compilation (for proofs B produced by our type system)

  e.g. R cases for if-conditional *(Inductive)*

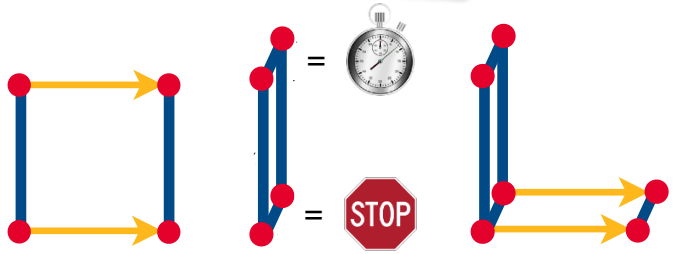


While

RISC

- Theorem: R (for B, with I) is a secure refinement

  (via *decomposition principle*)



While

RISC

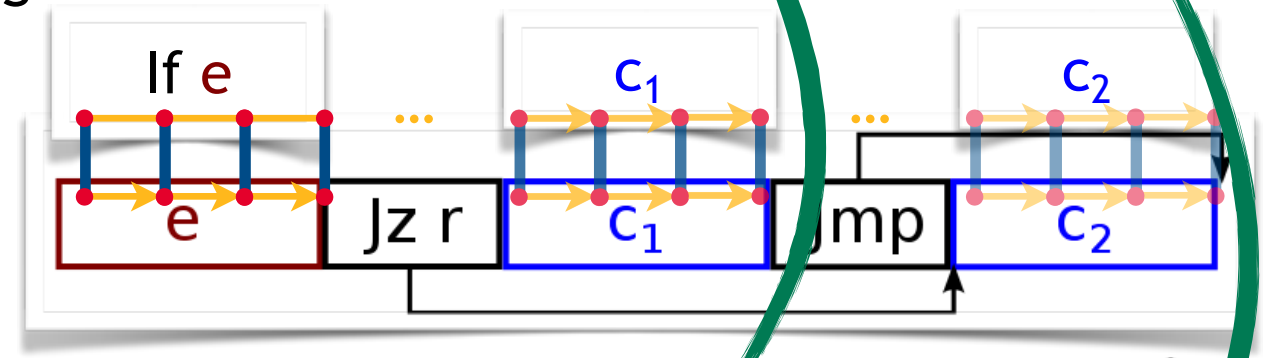
- Theorem: Compiler input related to output by R

(Formalisation: https://covern.org/itp19.html)

Verified compiler

Proof approach

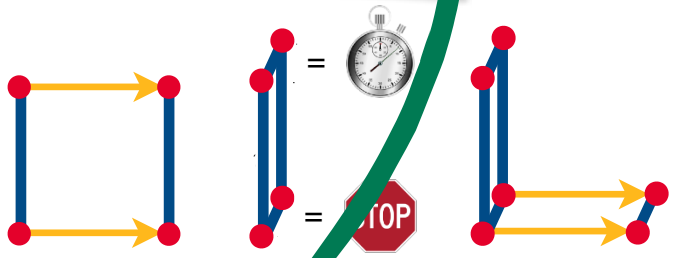
- Nominate R (and I) to characterise compilation (for proofs B produced by our type system)
  - e.g. R cases for if-conditional *(Inductive)*



While

RISC

- Theorem: R (for B, with I) is a secure refinement (via *decomposition principle*)



While

RISC

- Theorem: Compiler input related to output by R

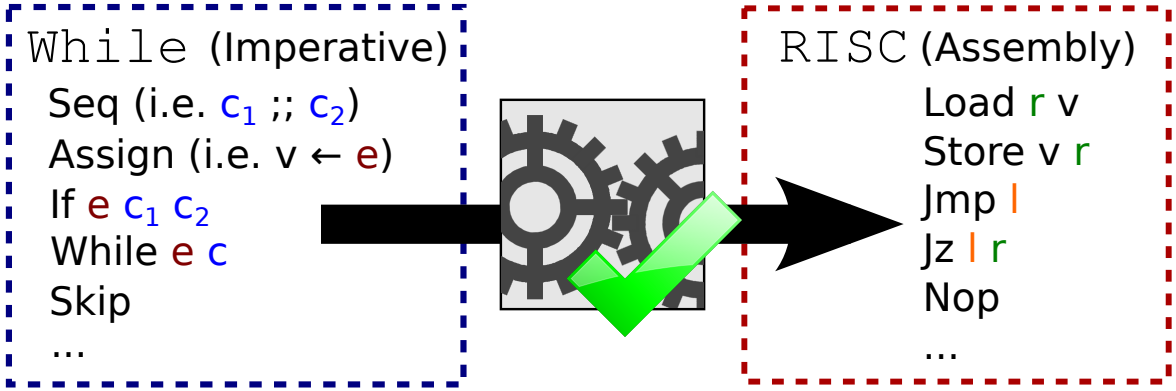
(Formalisation: https://covern.org/itp19.html)

Verified compiler

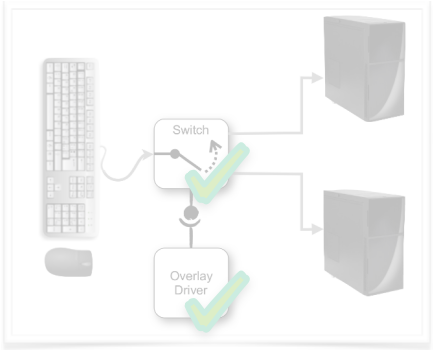
Overview

(Based on: Tedesco et al. CSF'16)

An Isabelle/HOL *primrec* function



- Proof approach: ~7K lines of Isabelle/HOL script
  - Prevents data races on shared memory
  - Knows when safe to optimise reads
- Application: 2-thread input-handling model of *Cross Domain Desktop Compositor*



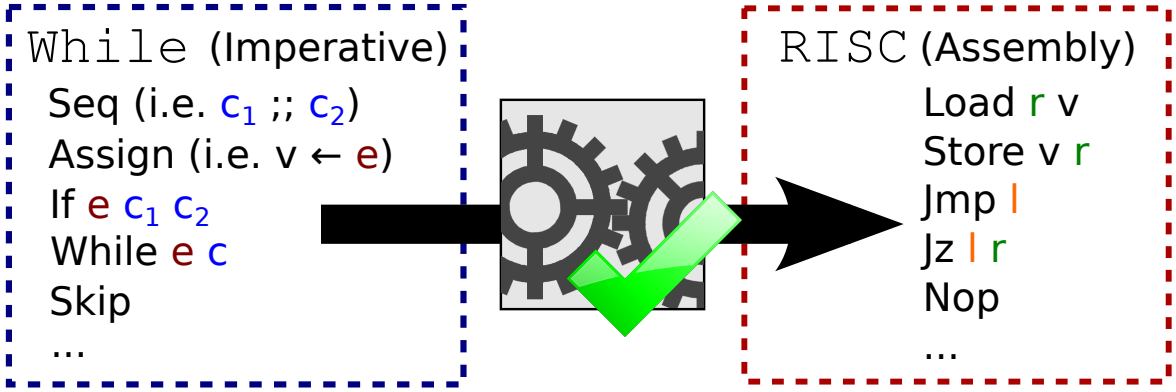
(Formalisation: https://covern.org/itp19.html)

Verified compiler

Overview

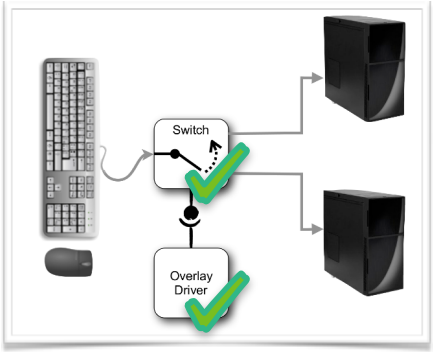
(Based on:
Tedesco et al. CSF'16)

An Isabelle/HOL *primrec* function



- Proof approach: ~7K lines of Isabelle/HOL script
  - Prevents data races on shared memory
  - Knows when safe to optimise reads

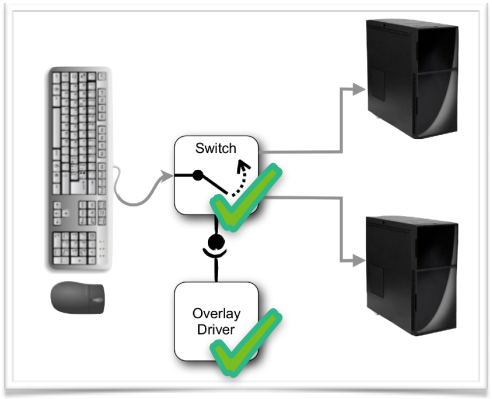
- Application: 2-thread input-handling model of *Cross Domain Desktop Compositor*

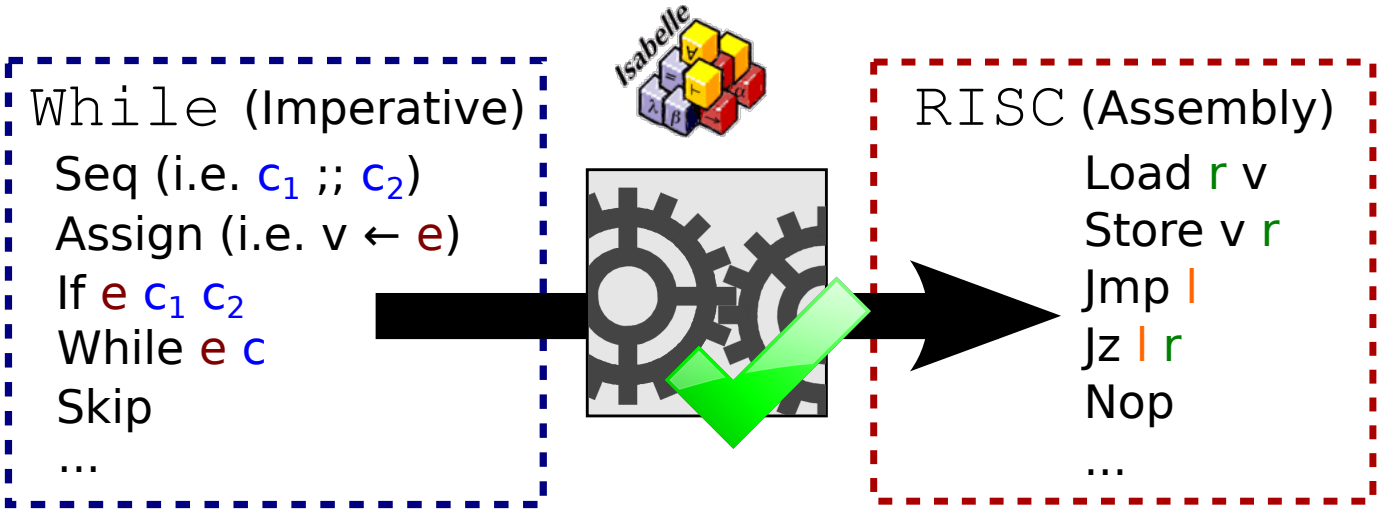


(Formalisation: https://covern.org/itp19.html)

Verified compiler

Application: CDDC input-handling model

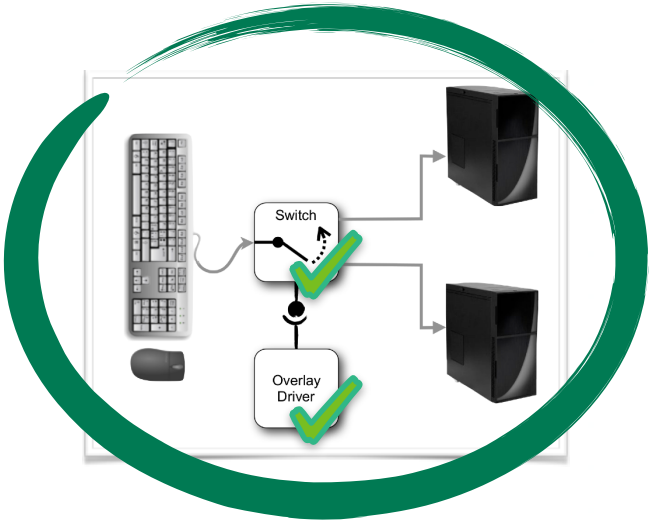




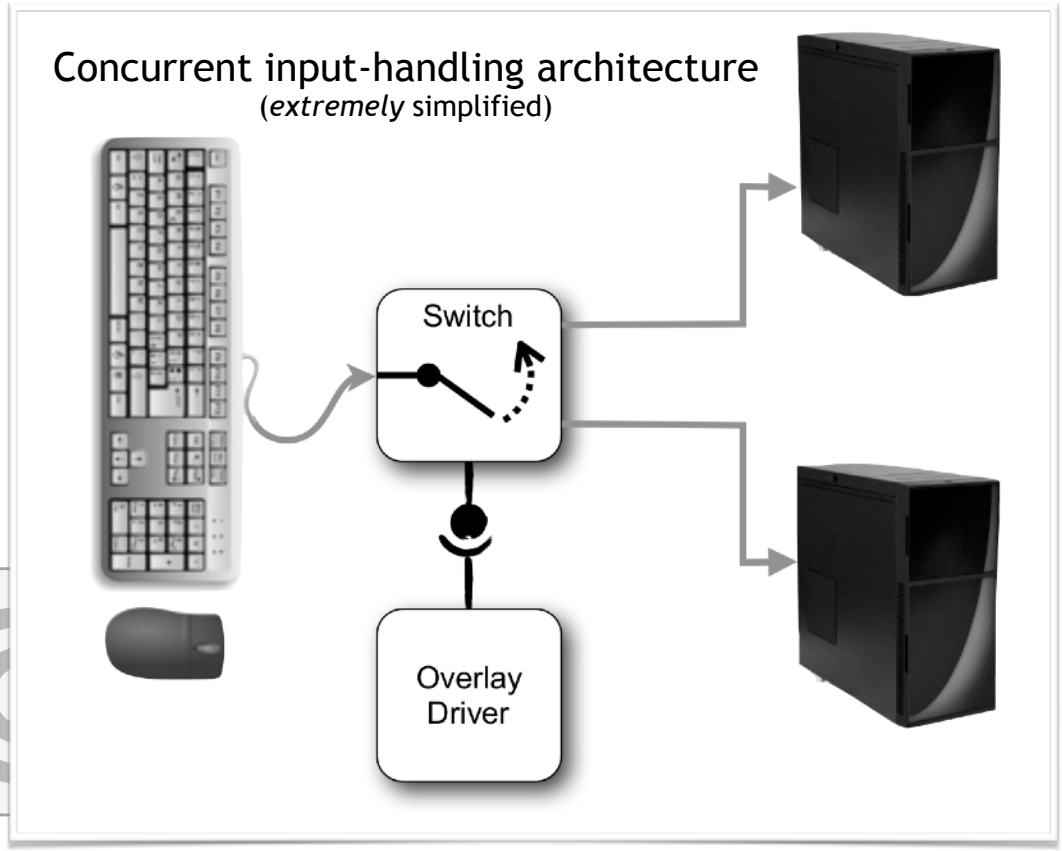
(Formalisation: https://covern.org/itp19.html)

Verified compiler

Application: CDDC input-handling model



`While` (Imperative)
- Seq (i.e. c_1 ;; c_2)
- Assign (i.e. v ← e)
- If e c_1 c_2
- While e c
- Skip
- ...

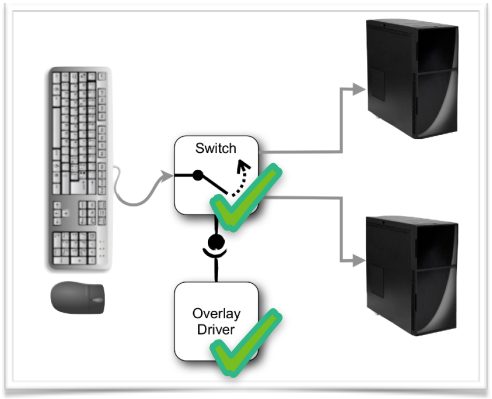


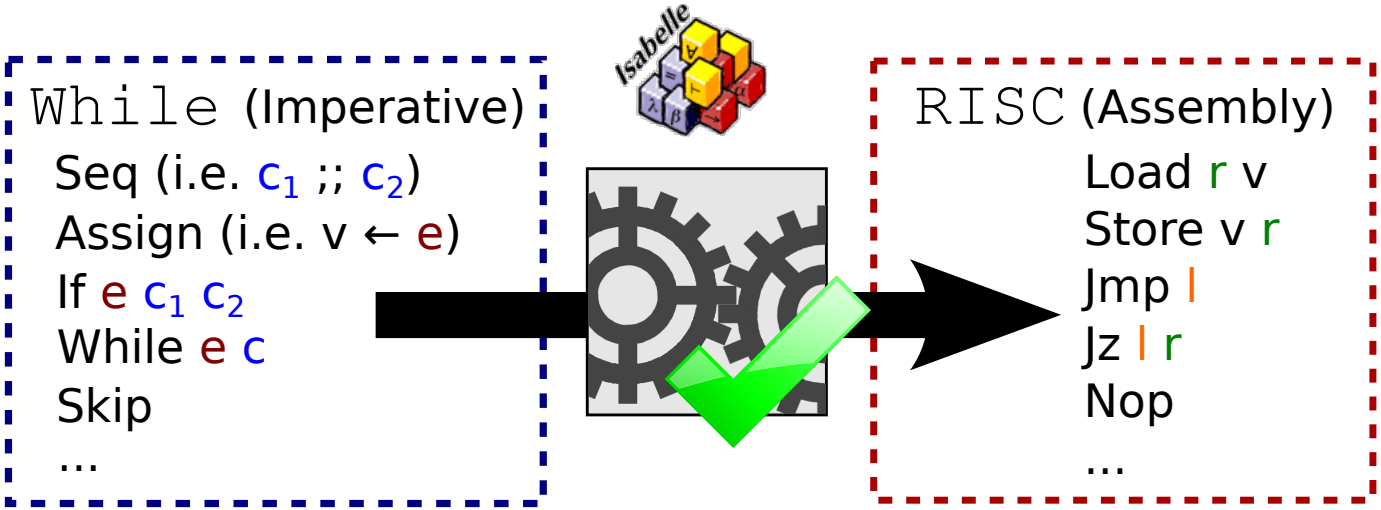
(Formalisation: https://covern.org/itp19.html)

Verified compiler

Application: CDDC input-handling model

~150 lines `While`



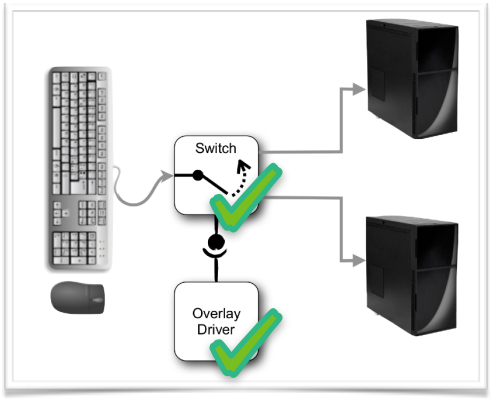


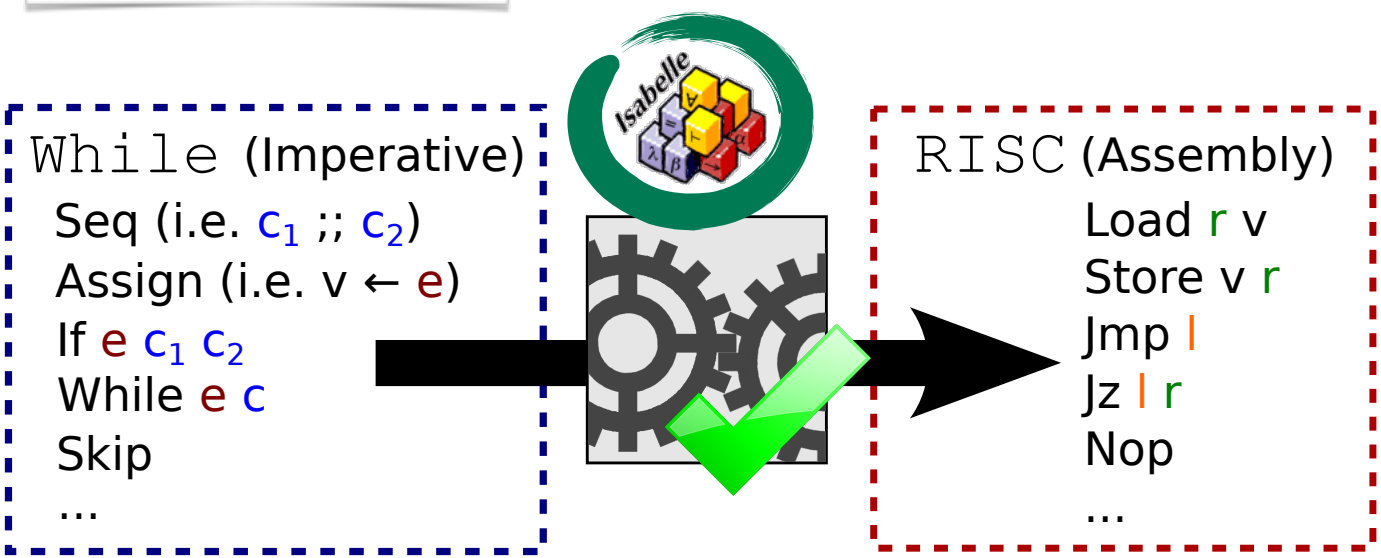
(Formalisation: https://covern.org/itp19.html)

Verified compiler

Application: CDDC input-handling model

~150 lines `While`





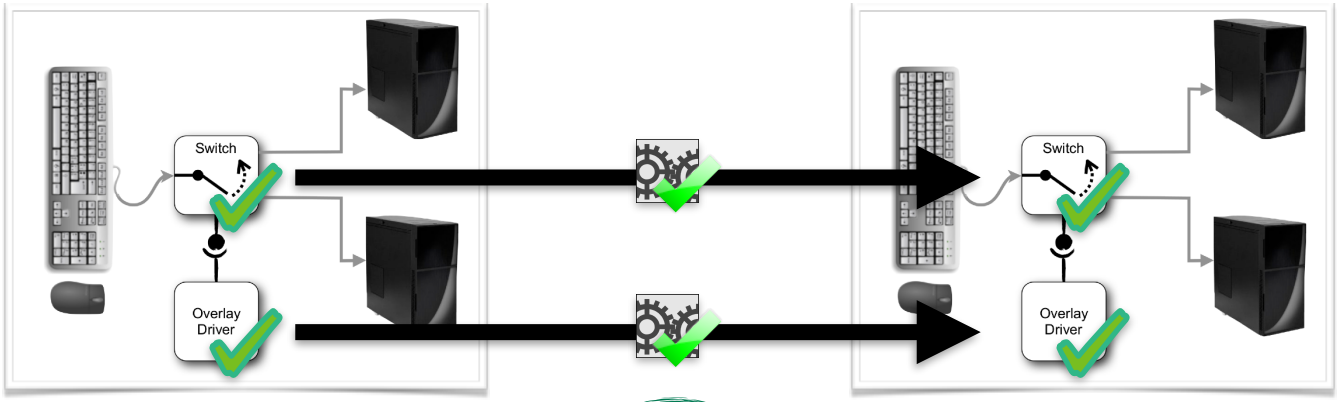
(Formalisation: https://covern.org/itp19.html)

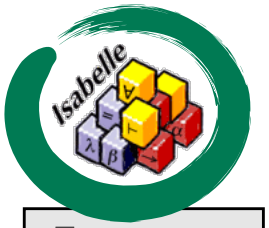
Verified compiler

Application: CDDC input-handling model

~150 lines `While`

~250 `RISC` instructions





`While` (Imperative)

- Seq (i.e. c_1 ;; c_2)
- Assign (i.e. v ← e)
- If e c_1 c_2
- While e c
- Skip
- ...

`RISC` (Assembly)

- Load r v
- Store v r
- Jmp l
- Jz l r
- Nop
- ...

(Formalisation: https://covern.org/itp19.html)

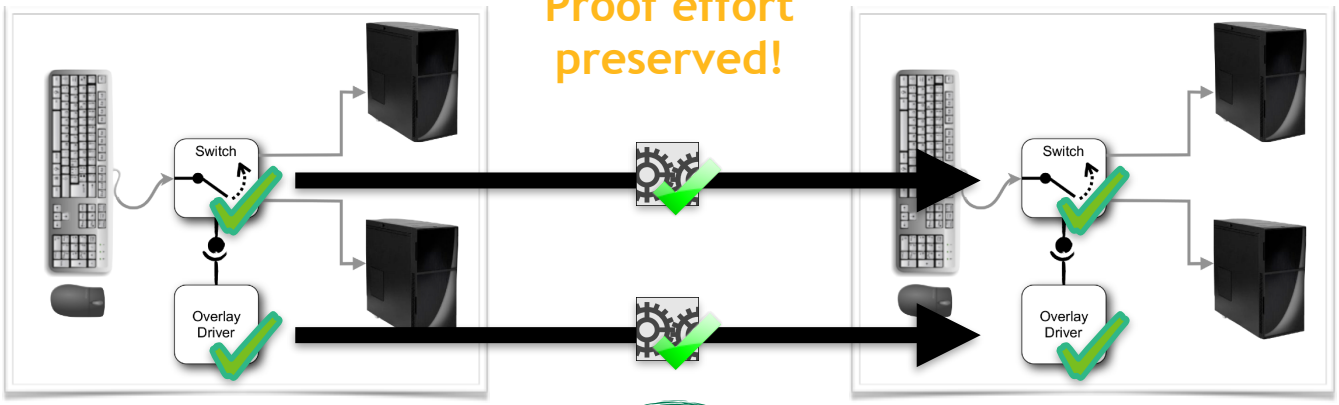
Verified compiler

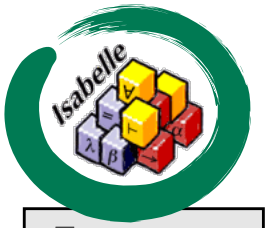
Application: CDDC input-handling model

~150 lines `While`

~250 `RISC` instructions

Proof effort preserved!





`While` (Imperative)

- Seq (i.e. c_1 ;; c_2)
- Assign (i.e. v ← e)
- If e c_1 c_2
- While e c
- Skip
- ...

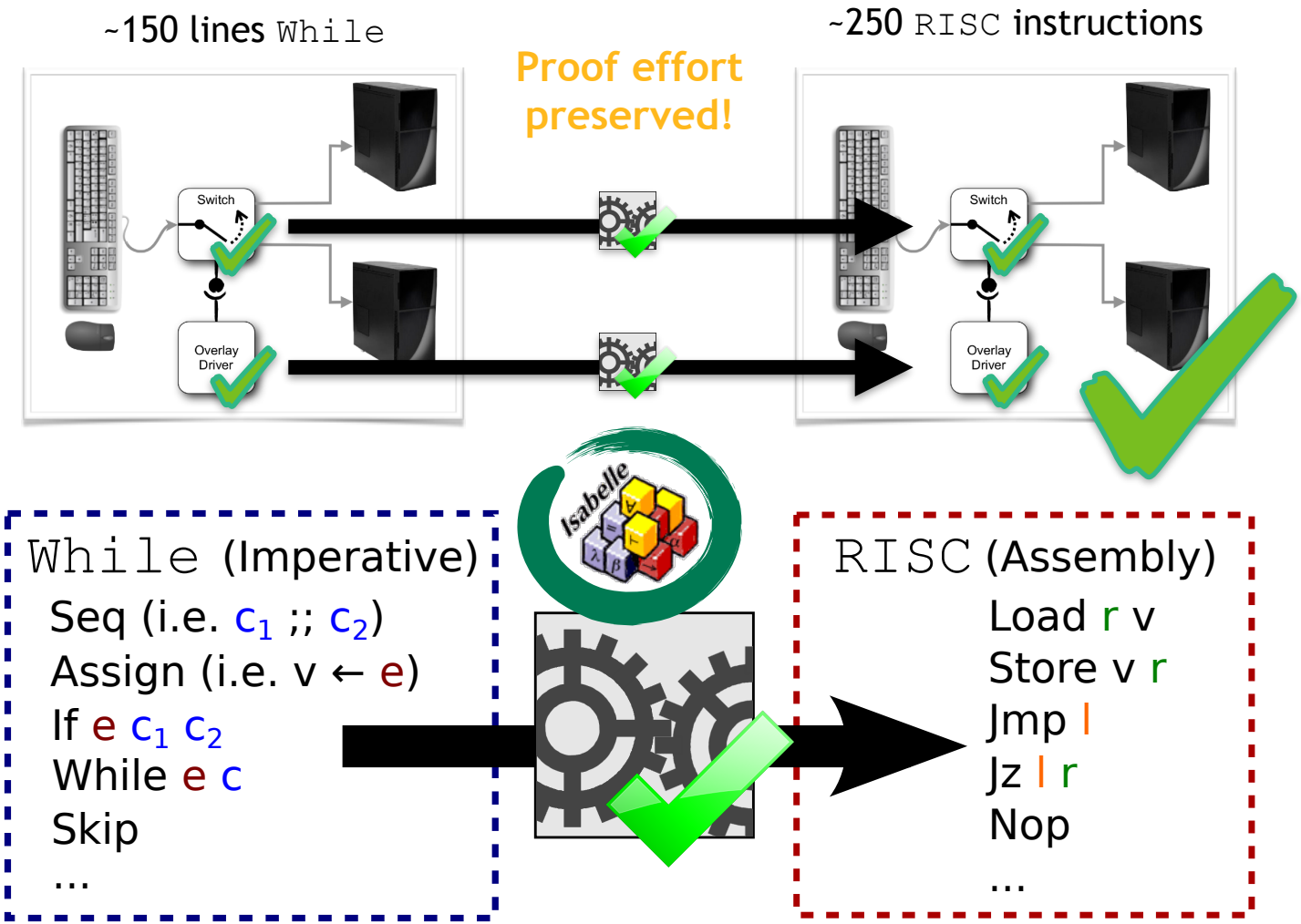
`RISC` (Assembly)

- Load r v
- Store v r
- Jmp l
- Jz l r
- Nop
- ...

(Formalisation: https://covern.org/itp19.html)

Verified compiler

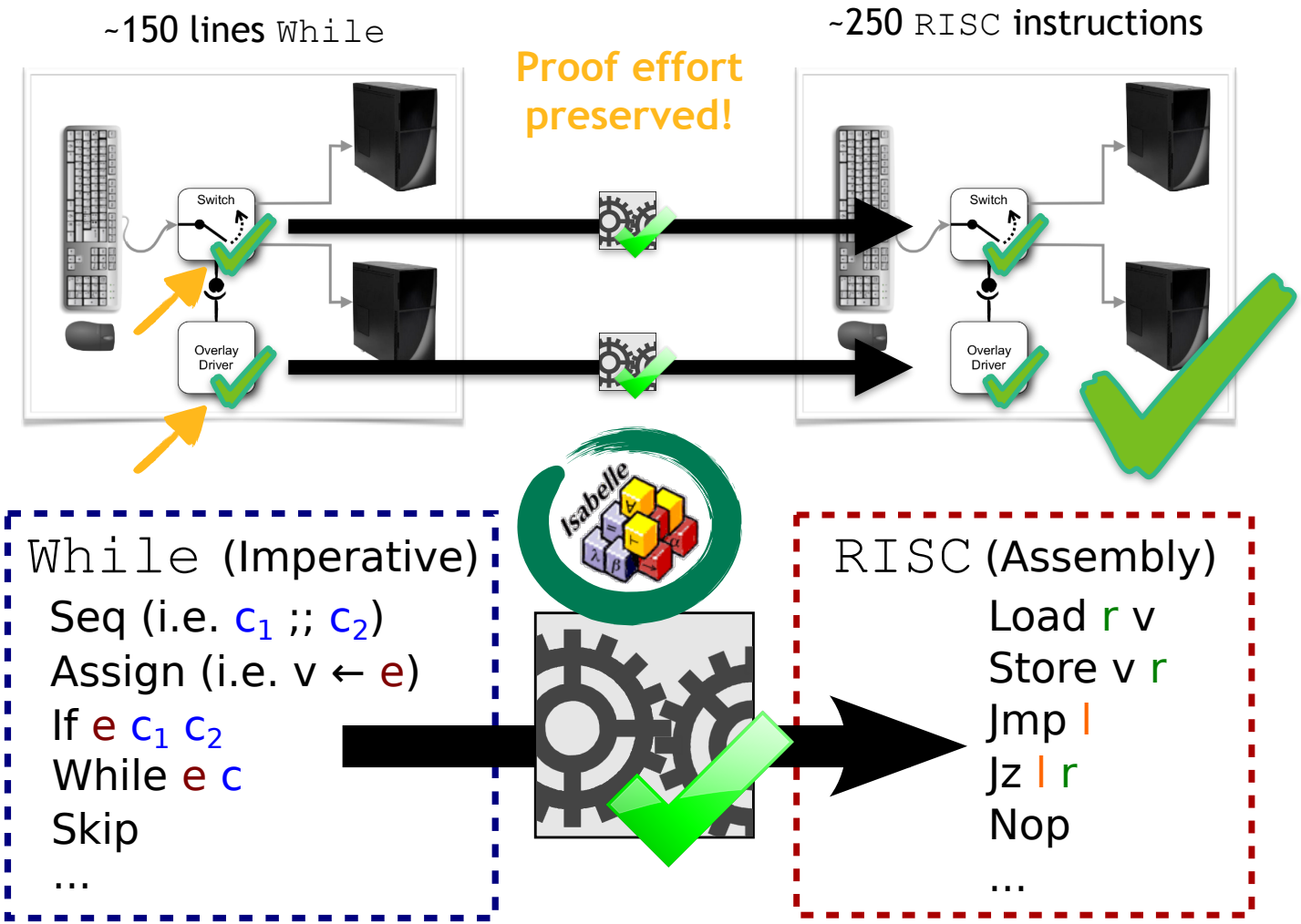
Application: CDDC input-handling model



(Formalisation: https://covern.org/itp19.html)

Verified compiler

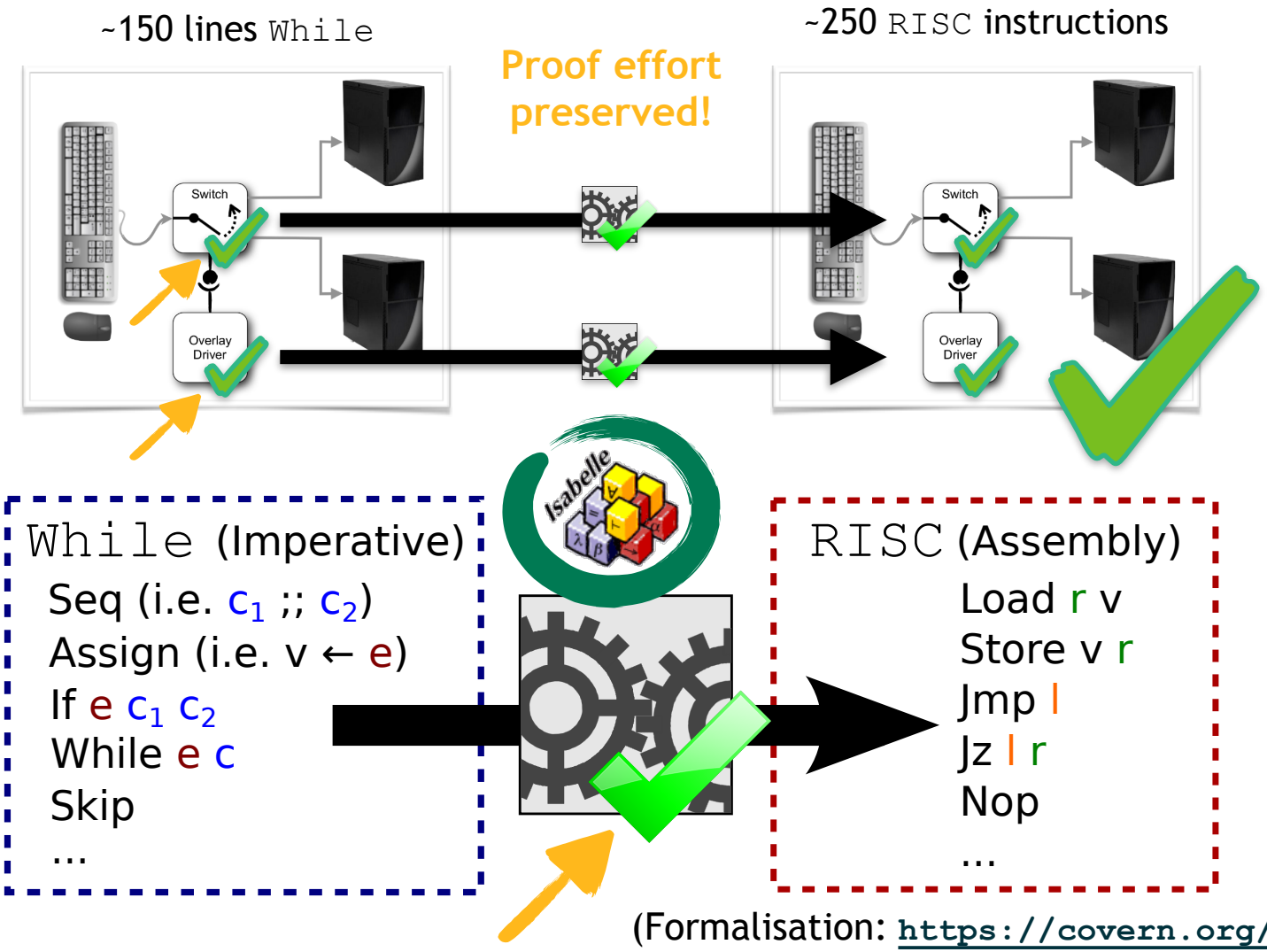
Application: CDDC input-handling model



(Formalisation: https://covern.org/itp19.html)

Verified compiler

Application: CDDC input-handling model



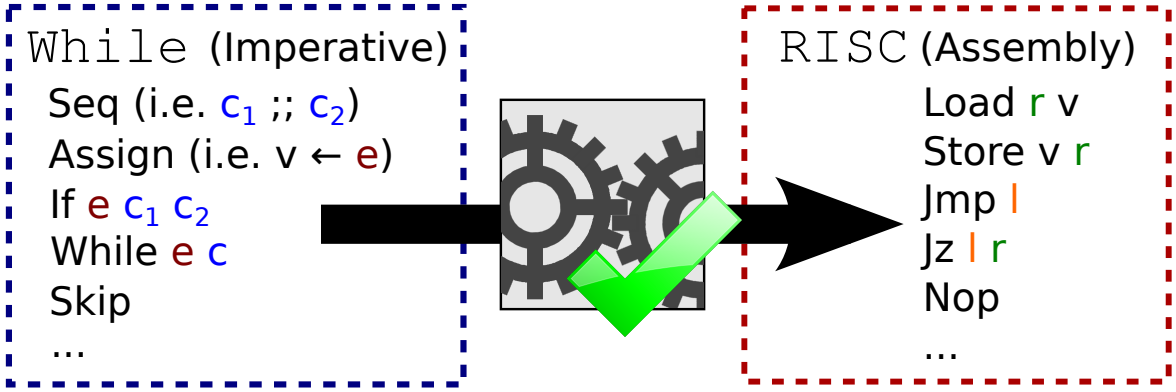
(Formalisation: https://covern.org/itp19.html)

Verified compiler

Overview

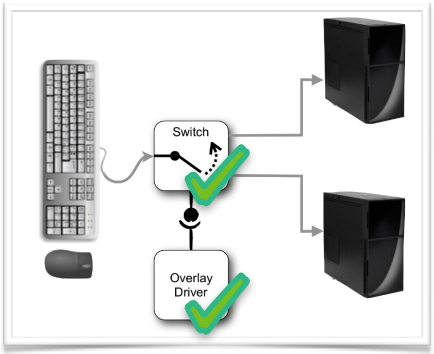
(Based on: Tedesco et al. CSF'16)

An Isabelle/HOL *primrec* function



- Proof approach: ~7K lines of Isabelle/HOL script
  - Prevents data races on shared memory
  - Knows when safe to optimise reads

- Application: 2-thread input-handling model of *Cross Domain Desktop Compositor*



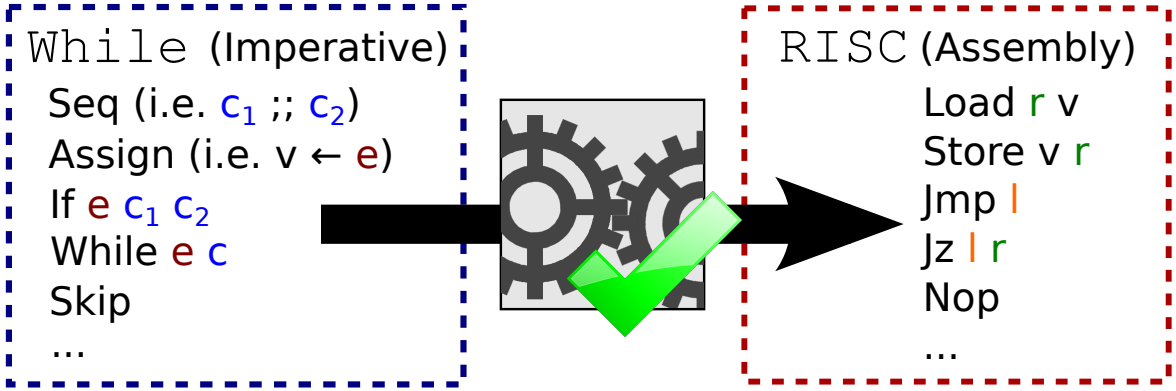
(Formalisation: https://covern.org/itp19.html)

Verified compiler

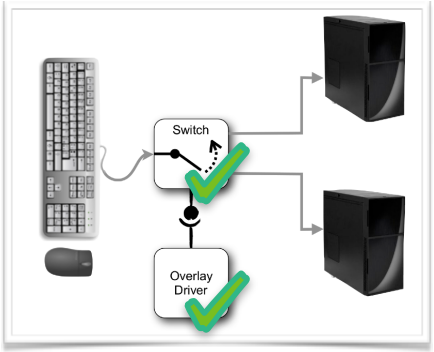
Overview

(Based on: Tedesco et al. CSF'16)

An Isabelle/HOL *primrec* function



- Proof approach: ~7K lines of Isabelle/HOL script
  - Prevents data races on shared memory
  - Knows when safe to optimise reads
- Application: 2-thread input-handling model of *Cross Domain Desktop Compositor*



(Formalisation: https://covern.org/itp19.html)

Our contributions (Conclusion)

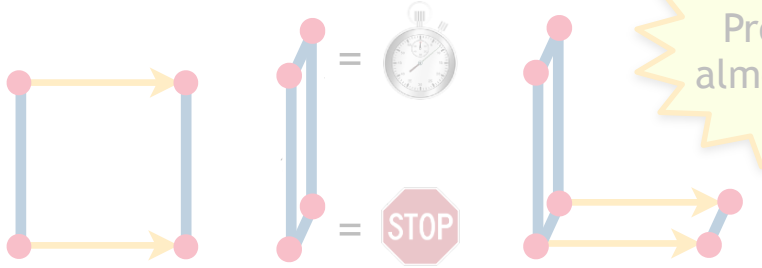
Goal

Prove a compiler *preserves proofs* of concurrent value-dependent information-flow security

Results

1. Decomposition principle

for *confidentiality-preserving refinement*



Proof effort almost *halved*!

(Technique)

2. Verified compiler

While-language to RISC-style assembly

(Proof-of-concept for technique)

Impact

1st such proofs carried to assembly-level model by compiler

(Formalisation: `https://covern.org/itp19.html`)

Our contributions (Conclusion)

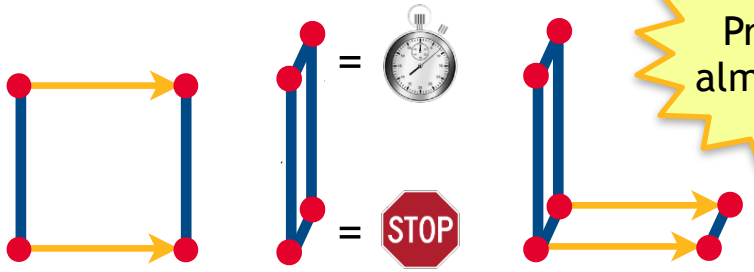
Goal

Prove a compiler *preserves proofs* of concurrent value-dependent information-flow security

Results

1. Decomposition principle

for *confidentiality-preserving refinement*



Proof effort almost *halved*!

(Technique)

2. Verified compiler

While-language to RISC-style assembly

(Proof-of-concept for technique)

Impact

1st such proofs carried to assembly-level model by compiler

(Formalisation: `https://covern.org/itp19.html`)

Our contributions (Conclusion) + *Q & A*

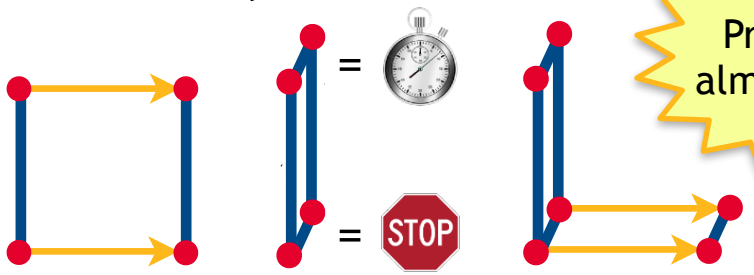
Goal

Prove a compiler *preserves proofs* of concurrent value-dependent information-flow security

Results

1. Decomposition principle

for *confidentiality-preserving refinement*



Proof effort almost *halved*!

(Technique)

2. Verified compiler

While-language to RISC-style assembly

(Proof-of-concept for technique)

Impact

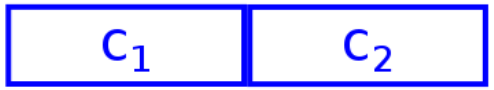
1st such proofs carried to assembly-level model by compiler

Thank you! Please see → (Formalisation: `https://covern.org/itp19.html`)

Appendix

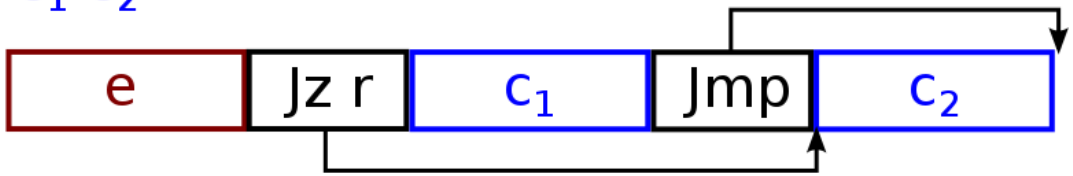
Differences from Tedesco et al. CSF'16 compilation scheme

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

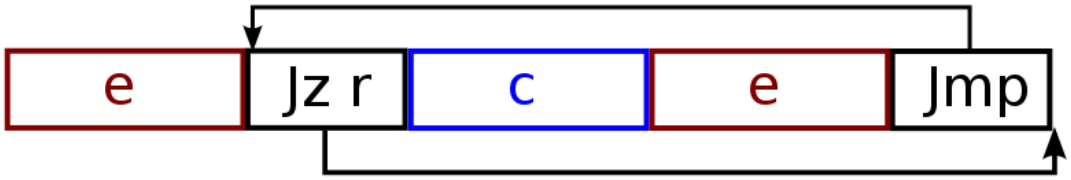


- Assign (i.e. v ← e)

- If e c_1 c_2

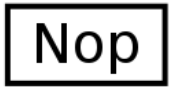


- While e c



- ~~LockAcq l~~
- ~~LockRel l~~

- Skip

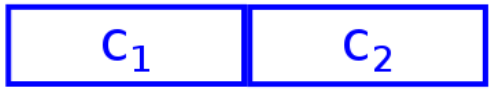


Tedesco et al. CSF'16

Appendix

Differences from Tedesco et al. CSF'16 compilation scheme

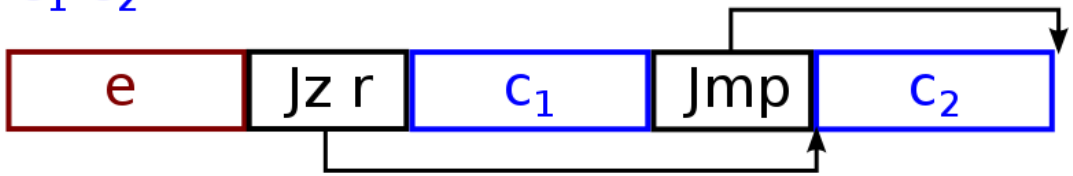
- Seq (i.e. c_1 ;; c_2)



- Assign (i.e. v ← e)

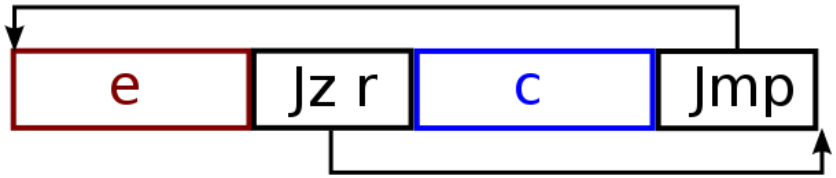
Fixed!

- If e c_1 c_2



- While e c

Simplified!

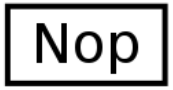


New!

- LockAcq l

- LockRel l

- Skip



Our While-to-RISC compiler