Per-Thread Compositional Compilation for Confidentiality-Preserving Concurrent Programs

Robert Sison
13 Jan 2018

www.data61.csiro.au
A confidentiality-preserving program

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

Data61/DSTG project for de-duplicating user-facing hardware.
A confidentiality-preserving program

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

Challenge #1: value-dependent security classifications
A confidentiality-preserving program

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

Challenge #1: value-dependent security classifications
A confidentiality-preserving program

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

Challenge #1: value-dependent security classifications
A confidentiality-preserving concurrent program

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

Challenge #2: shared-variable concurrency
A confidentiality-preserving concurrent program

CDDC seL4-based software architecture:
A confidentiality-preserving concurrent program

CDDC seL4-based software architecture (simplified model):
Per-thread compositional verification

Challenge #3: per-thread compositionality of proofs
Per-thread compositional verification

Challenge #3: per-thread compositionality of proofs

Mechanized in Isabelle/HOL. (More to appear: EuroS&P’18.)
Per-thread compositional verification

Challenge #3: per-thread compositionality of proofs

A, B, C etc. obey 🗝️

shared mem

A B C

shared mem

⋯
Per-thread compositional compilation

Challenge #3: per-thread compositionality of proofs

A, B, C etc. obey

Source
Per-thread compositional compilation

Challenge #3: per-thread compositionality of proofs

Source

Target

A, B, C etc. obey

A', B', C' etc. obey
Challenge #3: per-thread compositionality of proofs

Source

Target
Per-thread compositional compilation

Challenge #3: per-thread compositionality of proofs

Source

Target
Per-thread compositional compilation

Challenge #3: per-thread compositionality of proofs

Source

Target
Per-thread compositional compilation

Challenge #3: per-thread compositionality of proofs
Per-thread compositional compilation

Challenge #3: per-thread compositionality of proofs

Source

Target
This talk

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

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

Source

Target
This talk: a preview

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

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

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

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

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

Source

Target
The confidentiality property

Concurrent value-dependent noninterference.
The confidentiality property

Concurrent value-dependent noninterference.

Simplest policy: High $\not\rightarrow$ Low
Low part of state must remain indistinguishable.
The confidentiality property

Concurrent value-dependent noninterference.

Simplest policy: High $\not\rightarrow$ Low
Low part of state must remain indistinguishable.
The confidentiality property

Concurrent value-dependent noninterference.

Simplest policy: High $\not\rightarrow$ Low
Low part of state must remain indistinguishable.

Reflects the attacker model.
The confidentiality property

*Concurrent value-dependent* noninterference.

Simplest policy: **High** $\not\rightarrow$ **Low**

Low part of state must remain indistinguishable.

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

Concurrent value-dependent noninterference.

Simplest policy: \textbf{High} $\nRightarrow$ \textbf{Low}

Low part of state must remain indistinguishable.
The confidentiality property

Concurrent value-dependent noninterference.

Simplest policy: \( \text{High} \not\rightarrow \text{Low} \)
Low part of state must remain indistinguishable.

\[ \forall n . \]
The confidentiality property

*Concurrent value-dependent* noninterference.

Simplest policy: \( \text{High} \not\rightarrow \text{Low} \)
Low part of state must remain indistinguishable.

e.g.

\[
\begin{array}{c}
\text{Initial State} \\
\text{Transition} \\
\text{Final State}
\end{array}
\]
The confidentiality property

Concurrent value-dependent noninterference.

Simplest policy: **High** $\rightarrow$ **Low**
Low part of state must remain indistinguishable.

e.g.
The confidentiality property

Concurrent value-dependent *noninterference*.

Simplest policy: \( \text{High} \not\rightarrow \text{Low} \)
Low part of state must remain indistinguishable.

e.g.
The confidentiality property

*Concurrent value-dependent* noninterference.

Simplest policy: **High → Low**
Low part of state must remain indistinguishable.

e.g.
The confidentiality property

Concurrent value-dependent noninterference.

Simplest policy: $\text{High} \not\rightarrow \text{Low}$
Low part of state must remain indistinguishable.

- A 2-safety hyperproperty.

\[\begin{figure}
\centering
\includegraphics[width=\textwidth]{image.png}
\end{figure}\]
The confidentiality property

Concurrent value-dependent \textit{noninterference}.

Simplest policy: $\text{High} \notightarrow \text{Low}$
Low part of state must remain indistinguishable.

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

*Concurrent value-dependent noninterference.*

Simplest policy: $\text{High} \notightarrow \text{Low}$

Low part of state must remain indistinguishable.

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

Concurrent value-dependent noninterference.

Simplest policy: $\text{High} \not\rightarrow \text{Low}$
Low part of state must remain indistinguishable.
Classification of state as H or L can vary over time.
The confidentiality property

*Concurrent value-dependent noninterference.*

Simplest policy: **High** ⇆ **Low**

Low part of state must remain indistinguishable.

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

*Concurrent* value-dependent noninterference.

Simplest policy: **High** → **Low**
Low part of state must remain indistinguishable. Classification of state as **H** or **L** can vary over time.
The confidentiality property

Concurrent *value-dependent* noninterference.

Simplest policy: High $\not\leftrightarrow$ Low

Low part of state must remain indistinguishable.

Classification of state as H or L *can vary over time.*
The confidentiality property

*Concurrent* value-dependent noninterference.

Simplest policy: \( \text{High} \not\rightarrow \text{Low} \)

Low part of state must remain indistinguishable.

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

*Concurrent* value-dependent noninterference.

Simplest policy: **High** $\not\rightarrow$ **Low**

Low part of state must remain indistinguishable.

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

\[
\begin{array}{ccc}
A & B & C \\
\ldots & \ldots & \ldots \\
\end{array}
\]

shared mem
The confidentiality property

*Concurrent* value-dependent noninterference.

Simplest policy: **High ↛ Low**

Low part of state must remain indistinguishable.
Classification of state as H or L can vary over time.

Per-thread, subject to havoc.
The confidentiality property

*Concurrent* value-dependent noninterference.

Simplest policy: \( \text{High} \not\rightarrow \text{Low} \)

Low, *unlocked* part of state must remain indistinguishable.

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

Per-thread, subject to havoc *that obeys locking discipline*. 
The confidentiality property

**Concurrent** value-dependent noninterference.

Simplest policy: $\text{High} \not\rightarrow \text{Low}$

Low, unlocked part of state must remain indistinguishable. Classification of state as H or L can vary over time.

Per-thread compositional property:

$$\forall n.$$
The confidentiality property

*Concurrent* value-dependent noninterference.

Simplest policy: \( \text{High} \not\rightarrow \text{Low} \)

Low, *unlocked* part of state must remain indistinguishable.
Classification of state as H or L can vary over time.

Per-thread compositional property:

\[ \forall n . \]
The confidentiality property

*Concurrent* value-dependent noninterference.

Simplest policy:  High $\nrightarrow$ Low

Low, *unlocked* part of state must remain indistinguishable.
Classification of state as H or L can vary over time.

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

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

*Concurrent* value-dependent noninterference.

Simplest policy: $\text{High} \not\rightarrow \text{Low}$

Low, *unlocked* part of state must remain indistinguishable.

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

Per-thread compositionality theorem:

[Murray+, CSF’16] instantiated with locking primitives.
The confidentiality property

*Concurrent* value-dependent noninterference.

Simplest policy: **High $\not\rightarrow$ Low**

Low, *unlocked* part of state must remain indistinguishable. Classification of state as H or L can vary over time.

Whole-system property:

[Murray+, CSF’16] instantiated with locking primitives.
The confidentiality property

*Concurrent* value-dependent noninterference.

Simplest policy: $\text{High} \not\rightarrow \text{Low}$

Low, *unlocked* part of state must remain indistinguishable.

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

Whole-system property:

$\forall \text{sched}$.
The confidentiality property

*Concurrent* value-dependent noninterference.

Simplest policy: \( \text{High} \notightarrow \text{Low} \)

Low, *unlocked* part of state must remain indistinguishable. Classification of state as H or L can vary over time.

Whole-system property:

\[ \forall \text{sched}. \]

![Diagram](image)
The confidentiality property

*Concurrent* value-dependent noninterference.

Simplest policy: **High** $\not\leftrightarrow$ **Low**

Low, *unlocked* part of state must remain indistinguishable.

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

Whole-system property:

$$\forall \text{sched} .$$

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

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

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

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

A, B, C etc. obey

A', B', C' etc. obey

Source

Target
Proof technique for compilation

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

\[ \forall n . \]

Source

\[ \forall n . \]

Target
Proof technique for compilation

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

![Diagram showing-per-thread-compositional-refinement]

$\forall n.$
Proof technique for compilation

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

∀ $n$.

Source

Target
Proof technique for compilation

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

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

$$\exists n .$$
Proof technique for compilation

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

\[ \exists n. \]
Proof technique for compilation

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

$$\exists n .$$
Proof technique for compilation

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

Then $B'$ ($= B_T$ of $B \mathcal{R} I$) establishes the target-level property:

![Diagram showing the proof technique for compilation]

**Source**

**Target**
Proof technique for compilation

Simpler proof technique than this!

\[ \exists n . \]

\[ B \rightarrow R \rightarrow I \]

\[ n \]

\[ R \rightarrow R \rightarrow I \]

\[ 1 \]
Proof technique for compilation

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

$$\exists n . \quad n = \text{abs_steps } A_2 A'_2$$

$$n = \text{abs_steps } A_1 A'_1$$

(See: https://www.isa-afp.org/entries/Dependent_SIFUM_Refinement.html)
Proof technique for compilation

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

$$\exists n .$$

$n = \text{abs\_steps } A_2 A'_2$

$n = \text{abs\_steps } A_1 A'_1$

Easy to prove if no H-branching in $A$

Proof technique for compilation

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

$$\exists n . n = \text{abs_steps } A_2 A'_2$$

$$n = \text{abs_steps } A_1 A'_1$$

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

Easy to prove if no H-branching in $A$, and no new H-branching.

(See: https://www.isa-afp.org(entries/Dependent_SIFUM_Refinement.html)
Proof technique for compilation

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

$$\exists n . n = \text{abs}_\text{steps} \ A \ A'$$

i.e. $\mathcal{R}$ a simulation of $A'$ by $A$. 
Proof technique for compilation

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

$$\exists n . \ n = \text{abs_steps} \ A \ A'$$

i.e. $\mathcal{R}$ a simulation of $A'$ by $A$, with provisos...
Proof technique for compilation

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

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

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

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

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

- $\mathcal{R}$ must preserve shared memory contents and locking state.
  - Under the hood: preserve assumptions and guarantees.
  - + any new locations permanently locked.

i.e. No new shared state.
Proof technique for compilation

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

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

Provisos for $R$, $I$:

- $R$ must preserve shared memory contents and locking state.
  - Under the hood: preserve assumptions and guarantees.
- $R$ must be closed under lock-permitted shared memory havoc.
Proof technique for compilation

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

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

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

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

Provisos for $R$, $I$:

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

Similar for $I$. 
This talk

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

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

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

$$\exists n . \ n = \text{abs\_steps } A A'$$
Compiler verification

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

Proof of concept: a While-to-RISC compiler

While (Imperative)
- Seq (i.e. $c_1 ; c_2$)
- Assign (i.e. $v \leftarrow e$)
- If $e \ c_1 \ c_2$
- While $e \ c$
- Skip
- ...

RISC (Assembly)
- Load $r \ v$
- Store $v \ r$
- Jmp $l$
- Jz $l \ r$
- Nop
- ...

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

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

Proof of concept: a While-to-RISC compiler

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

- Assign (i.e. $v \leftarrow e$)

- If $e \quad c_1 \quad c_2$

- While $e \quad c$

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

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

Proof of concept: a While-to-RISC compiler

- Seq (i.e. $c_1 ; c_2$)
  - $c_1$
  - $c_2$
- Assign (i.e. $v \leftarrow e$) Fixed!
  - $e$
  - $\text{Store}\ v\ \text{r}$
- If $e\ c_1\ c_2$
  - $e$
  - $\text{Jz}\ r$
  - $c_1$
  - $\text{Jmp}$
  - $c_2$
- While $e\ c$
  - $e$
  - $\text{Jz}\ r$
  - $c$
  - $\text{Jmp}$

New!

- LockAcq $l$
  - LockAcq $l$
- LockRel $l$
  - LockRel $l$
- Skip
  - Nop

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

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

Proof of concept: a While-to-RISC compiler
e.g. $\mathcal{R}$ cases for If construct

\[
\text{If } e \ c_1 \ c_2
\]
Compiler verification

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

Proof of concept: a While-to-RISC compiler
e.g. $\mathcal{R}$ cases for If construct

$$\text{If } e \quad \begin{array}{c} \text{e} \\ \text{Jz r} \end{array} \text{Jmp} \quad \begin{array}{c} c_1 \\ c_2 \end{array}$$
Compiler verification

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

Proof of concept: a While-to-RISC compiler
e.g. $\mathcal{R}$ cases for If construct, $c_1$ case:

```
If e c_1 c_2
```

Diagram:

```
e  Jz r  c_1  Jmp  c_2
```
Compiler verification

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

Proof of concept: a While-to-RISC compiler

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

$$\frac{\text{If } e \ c_1 \ c_2 \sim c_1}{\text{e Jz r c}_1 \text{ Jmp c}_2}$$

Relation is inductive for smaller program pairs $c_1$, $c_2$
Compiler verification

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

Proof of concept: a While-to-RISC compiler
e.g. $\mathcal{R}$ cases for If construct, $c_1$ case:

```
If e $c_1$ $c_2$ $\leadsto$ $c_1$ $\leadsto$ ...
```

Relation is inductive for smaller program pairs $c_1$, $c_2$
Compiler verification

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

Proof of concept: a While-to-RISC compiler
e.g. \(\mathcal{R}\) cases for If construct, \(c_1\) case:

\[
\text{If } e \quad c_1 \quad c_2 \quad \leadsto \quad c_1 \quad \leadsto \quad \ldots \quad \leadsto \quad \text{Stop}
\]

Relation is inductive for smaller program pairs \(c_1, c_2\)
Compiler verification

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

Proof of concept: a While-to-RISC compiler

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

\[
\text{If } e \quad c_1 \quad c_2 \quad \sim \quad c_1 \quad \sim \quad \ldots \quad \sim \quad \text{Stop}
\]

Relation is inductive for smaller program pairs $c_1$, $c_2$
Compiler verification

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

Proof of concept: a While-to-RISC compiler

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

$\exists n \cdot n = \text{abs\_steps } A_2 A'_2$

$\exists n \cdot n = \text{abs\_steps } A_1 A'_1$

$\exists n \cdot n = \text{abs\_steps } A A'$

$\mathcal{R}$
Compiler verification

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

Proof of concept: a While-to-RISC compiler

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

- **Theorem:** Compiler input is related to its output by $\mathcal{R}$
  - Started with same observable initial state.
  - No branching on H values. (Same as for type system.)
Compiler verification

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

Proof of concept: a While-to-RISC compiler
Compiler verification

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

Proof of concept: a While-to-RISC compiler

**Source**

**Target**
Compiler verification

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

Proof of concept: a While-to-RISC compiler

Exercised on verified Cross Domain Desktop Compositor model.
Limitations and future work ideas

- Optimisations to non-observable shared memory?
Limitations and future work ideas

- Optimisations to non-observable shared memory? Possibly too strict.
Limitations and future work ideas

- Optimisations to non-observable shared memory? Possibly too strict.
Limitations and future work ideas

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

Relax for shared memory out of reach of attacker model?
Limitations and future work ideas

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

Relax for shared memory out of reach of attacker model?
Limitations and future work ideas

- Optimisations to non-observable shared memory?
- Can existing compilers be proven to satisfy it?
Limitations and future work ideas

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

$$\exists n . n = \text{abs\_steps } A \ A'$$
Limitations and future work ideas

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

$$\exists n . n = \text{abs\_steps} A A'$$
Limitations and future work ideas

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

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

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

\[ \exists n . n = \text{abs}_{-}\text{steps} A A' \]

\[ \exists n . n = \text{abs}_{-}\text{steps} A_1 A_2 \]
Limitations and future work ideas

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

Thank you! Q & A
Appendix: Co-habiting attacker?

CDDC case study, again.
Appendix: Co-habiting attacker?

CDDC case study, again.

Untrusted sink: input device event stream out to Low machine.
Appendix: Co-habiting attacker?

CDDC case study, again.

Untrusted sink: input device event stream out to Low machine. What else can we afford to distrust?
Appendix: Co-habiting attacker?

CDDC case study, again.

Hypothetically, a co-habiting “attacker” …?
Appendix: Co-habiting attacker?

CDDC case study, again.

Hypothetically, a co-habiting “attacker” ...

... if it in fact cannot see/touch High nor locked part of state.
Appendix: Co-habiting attacker?

CDDC case study, again.

Hypothetically, a co-habiting “attacker”...

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

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

```
definition
simpler_refinement_safe R A R P abs_steps ⇔
∀ c₁₄ mds₁₄ mem₁₄ c₂₄ mds₂₄ mem₂₄. ((c₁₄, mds₁₄, mem₁₄) A, (c₂₄, mds₂₄, mem₂₄) A) ∈ R A ∧
((c₁₄, mds₁₄, mem₁₄) A, (c₁₄, mds₂₄, mem₂₄) C) ∈ R C ∧
((c₁₄, mds₁₄, mem₁₄) C, (c₂₄, mds₂₄, mem₂₄) C) ∈ P →
(abs_steps (c₁₄, mds₁₄, mem₁₄) A, (c₁₄, mds₂₄, mem₂₄) C) = stops₃ (c₂₄, mds₂₄, mem₂₄) C ∧
(abs_steps (c₁₄, mds₁₄, mem₁₄) A, (c₁₄, mds₂₄, mem₂₄) C) = stops₃ (c₂₄, mds₂₄, mem₂₄) C ∧
(abs_steps (c₁₄, mds₁₄, mem₁₄) A, (c₁₄, mds₂₄, mem₂₄) C) = abs_steps (c₂₄, mds₁₄, mem₂₄) A, (c₂₄, mds₂₄, mem₂₄) C) ∧
(abs_steps (c₁₄, mds₁₄, mem₁₄) A, (c₁₄, mds₂₄, mem₂₄) C) = abs_steps (c₂₄, mds₁₄, mem₂₄) A, (c₂₄, mds₂₄, mem₂₄) C) ∧
(abs_steps (c₁₄, mds₁₄, mem₁₄) A, (c₁₄, mds₂₄, mem₂₄) C) = abs_steps (c₂₄, mds₁₄, mem₂₄) A, (c₂₄, mds₂₄, mem₂₄) C) ∧
(abs_steps (c₁₄, mds₁₄, mem₁₄) A, (c₁₄, mds₂₄, mem₂₄) C) = abs_steps (c₂₄, mds₁₄, mem₂₄) A, (c₂₄, mds₂₄, mem₂₄) C) ∧
(abs_steps (c₁₄, mds₁₄, mem₁₄) A, (c₁₄, mds₂₄, mem₂₄) C) = abs_steps (c₂₄, mds₁₄, mem₂₄) A, (c₂₄, mds₂₄, mem₂₄) C) ∧
(abs_steps (c₁₄, mds₁₄, mem₁₄) A, (c₁₄, mds₂₄, mem₂₄) C) = abs_steps (c₂₄, mds₁₄, mem₂₄) A, (c₂₄, mds₂₄, mem₂₄) C) ∧
(abs_steps (c₁₄, mds₁₄, mem₁₄) A, (c₁₄, mds₂₄, mem₂₄) C) = abs_steps (c₂₄, mds₁₄, mem₂₄) A, (c₂₄, mds₂₄, mem₂₄) C) ∧
(abs_steps (c₁₄, mds₁₄, mem₁₄) A, (c₁₄, mds₂₄, mem₂₄) C) = abs_steps (c₂₄, mds₁₄, mem₂₄) A, (c₂₄, mds₂₄, mem₂₄) C) ∧
(abs_steps (c₁₄, mds₁₄, mem₁₄) A, (c₁₄, mds₂₄, mem₂₄) C) = abs_steps (c₂₄, mds₁₄, mem₂₄) A, (c₂₄, mds₂₄, mem₂₄) C) ∧
(abs_steps (c₁₄, mds₁₄, mem₁₄) A, (c₁₄, mds₂₄, mem₂₄) C) = abs_steps (c₂₄, mds₁₄, mem₂₄) A, (c₂₄, mds₂₄, mem₂₄) C)
```

Provisos and simulation relation:

```
definition
secure_refinement_simpler R A R P abs_steps ⇔
closed_others R A ∧
preserves_modes_mem R A ∧
new_vars_private R A ∧
simpler_refinement_safe R A R P abs_steps ∧
conclosed_glob_consistent P A ∧
∀ c₁₄ mds₁₄ mem₁₄ c₂₄ mds₂₄ mem₂₄. ((c₁₄, mds₁₄, mem₁₄) A, (c₂₄, mds₂₄, mem₂₄) C) ∈ R A →
∀ c₁₄ c₁₄ mds₁₄ mem₁₄ c₁₄ mds₂₄ mem₂₄. ((c₁₄, mds₁₄, mem₁₄) C, (c₁₄, mds₂₄, mem₂₄) C) ∈ R C →
∃ c₁₄ mds₁₄ mem₁₄. abs.neval (c₁₄, mds₁₄, mem₁₄) A, labs_steps (c₁₄, mds₁₄, mem₁₄) A, (c₁₄, mds₂₄, mem₂₄) C) (c₁₄, mds₂₄, mem₂₄) A ∧
[[c₁₄, mds₁₄, mem₁₄] A, (c₁₄, mds₂₄, mem₂₄) C) ∈ R))
```

(See: https://www.isa-afp.org/entries/Dependent_SIFUM_Refinement.html)
Invariant on integrity of Switch’s internal state w.r.t. indicator. To appear: EuroS&P’18.