Per-Thread Compositional Compilation for Confidentiality-Preserving Concurrent Programs

Robert Sison
13 Jan 2018
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

shared mem

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
Challenge #3: per-thread compositionality of proofs
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

Source

Target
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
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

Source

Target
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: \textbf{High} $\nrightarrow$ \textbf{Low}

Low part of state must remain indistinguishable.
The confidentiality property

Concurrent value-dependent *noninterference.*

Simplest policy: \( \text{High} \notightarrow \text{Low} \)
Low part of state must remain indistinguishable.
The confidentiality property

Concurrent value-dependent noninterference.

Simplest policy: High $\nrightarrow$ 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.
The confidentiality property

*Concurrent value-dependent noninterference.*

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

Low part of state must remain indistinguishable.
The confidentiality property

Concurrent value-dependent noninterference.

Simplest policy: \textbf{High} \nrightarrow \textbf{Low}

Low part of state must remain indistinguishable.

$\forall n$ .
The confidentiality property

*Concurrent value-dependent noninterference.*

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

e.g.

![Diagram](image)
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 $\nrightarrow$ Low
Low part of state must remain indistinguishable.

e.g.
The confidentiality property

Concurrent value-dependent noninterference.

Simplest policy: $\text{High} \rightarrow \text{Low}$
Low part of state must remain indistinguishable.

e.g.

\begin{center}
\begin{tikzpicture}
\node[round, text centered] (A) at (0,0) {5};
\node[round, text centered] (B) at (2,0) {5};
\node[round, text centered] (C) at (0,2) {4};
\node[round, text centered] (D) at (2,2) {4};
\draw[->] (A) -- (B);
\draw[->] (B) -- (D);
\draw[->] (C) -- (D);
\end{tikzpicture}
\end{center}
The confidentiality property

Concurrent value-dependent noninterference.

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

e.g.

- A 2-safety hyperproperty.
The confidentiality property

Concurrent value-dependent \textit{noninterference}.

Simplest policy: \textbf{High} $\not\leftrightarrow$ \textbf{Low}

Low part of state must remain indistinguishable.

\begin{itemize}
  \item A 2-safety hyperproperty.
  \item Timing-sensitive. (Want this for concurrency reasons.)
\end{itemize}
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.
The confidentiality property

Concurrent \textit{value-dependent} noninterference.

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

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: H → L
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: \( \text{High} \not\leftrightarrow \text{Low} \)

Low part of state must remain indistinguishable. Classification of state as \( \text{H} \) or \( \text{L} \) 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.
The confidentiality property

*Concurrent* value-dependent noninterference.

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

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.

A  B  C

\[ \cdots \]

shared mem
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.

Per-thread, subject to havoc.
The confidentiality property

*Concurrent* value-dependent noninterference.

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

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: High $\not\rightarrow$ Low

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

Per-thread compositional property:

\[
\forall n . \quad \text{steps with havoc obeying locks} \quad \text{steps with same havoc as above}
\]
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:
The confidentiality property

*Concurrent* value-dependent noninterference.

Simplest policy: **High ↛ 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:

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:

\[ \text{[Murray+, CSF'16] instantiated with locking primitives.} \]
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: \( \text{High} \not\rightarrow \text{Low} \)

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

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

Whole-system property:

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

\[ \text{sched} \]
The confidentiality property

*Concurrent* value-dependent noninterference.

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

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 of state transition]

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

Source

Target
This talk

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

Per-thread compositional refinement [Murray+, CSF’16]
Proof technique for compilation

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

∀n.

Source

Target
Proof technique for compilation

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

Given bisimulation $B$ establishing the property, 

$$\forall n .$$
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.:

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

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

Given bisimulation $B$ establishing the property, nominate $\mathcal{R}$, $\mathcal{I}$ s.t.
Proof technique for compilation

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

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

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

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

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

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

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

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

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

Simpler proof technique than this!

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

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

$$\exists n .$$

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

$\exists n . n = \text{abs_steps } A_1 A_1'$

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

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

$$\exists n .$$

$\mathcal{B}$

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

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

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

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

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

$$\exists n .$$

$\mathcal{R}$

$\mathcal{B}$

$\mathcal{A}_2$

$n = \text{abs_steps } \mathcal{A}_2 \mathcal{A}'_2$

$\mathcal{A}_1$

$\mathcal{R}$

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

$\mathcal{I}$

$\mathcal{A}'_2$

1

$\mathcal{I}$

$\mathcal{A}'_1$

1

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

Easy to prove if no H-branching in $\mathcal{A}$, 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:

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

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

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

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

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

Provisos for $R$, $I$:

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

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

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

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

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

  + any new locations permanently locked.

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

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

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

Provisos for \( \mathcal{R}, \mathcal{I} \):

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

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

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

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

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

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

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

Similar for $\mathcal{I}$. 

Equal only if locked!
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 $R$ characterising a compiler.

Proof of concept: a While-to-RISC compiler

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

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

Proof of concept: a While-to-RISC compiler

- **Seq** (i.e. $c_1 ;; c_2$)
  
  \[
  \begin{array}{c}
  c_1 \\
  c_2
  \end{array}
  \]

- **Assign** (i.e. $v \leftarrow e$)
  
  \[
  \begin{array}{c}
  e \\
  \text{Store } v \text{ r}
  \end{array}
  \]

- **If** $e c_1 c_2$
  
  \[
  \begin{array}{c}
  e \\
  \text{Jz } r \\
  c_1 \\
  \text{Jmp} \\
  c_2
  \end{array}
  \]

- **While** $e c$
  
  \[
  \begin{array}{c}
  e \\
  \text{Jz } r \\
  c \\
  e \\
  \text{Jmp}
  \end{array}
  \]

- **Skip**
  
  \[
  \begin{array}{c}
  \text{Nop}
  \end{array}
  \]

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

Per-thread *simpler compositional refinement* [Murray+, AFP], instantiated with \( R \) 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. \( v \leftarrow e \))
  - \( e \) \( \text{Store} v \) \( r \) \text{Fixed!}
- If \( e \) \( c_1 \) \( c_2 \)
  - \( e \) \( \text{Jz} \) \( r \) \( c_1 \) \( \text{Jmp} \) \( c_2 \) \text{Simplified!}
- While \( e \) \( c \)
  - \( e \) \( \text{Jz} \) \( r \) \( c \) \( \text{Jmp} \) \text{Simplified!}
- LockAcq \( l \)
  - LockAcq \( l \) \text{New!}
- LockRel \( l \)
  - LockRel \( l \)
- Skip
  - \( \text{Nop} \)

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

```
If e c₁ c₂
```

```latex
\begin{center}
\begin{tabular}{c|c|c|c|c}
 & e & Jz & r & c₁ & Jmp & c₂ \\
\end{tabular}
\end{center}
```
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:

\[
\text{If } e \quad c_1 \quad c_2
\]
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:

$$\text{If } e \ c_1 \ c_2 \sim c_1$$

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

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

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

$$\text{If } e \ c_1 \ c_2 \sim c_1 \sim \ldots$$

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 $e$ $c_1$ $c_2$ $\sim$ $c_1$ $\sim$ ... $\sim$ Stop

e  Jz  r  c_1  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:

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

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

- 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.

∃$n$. $n = \text{abs_steps} A_1 A'_{1}$

∃$n$. $n = \text{abs_steps} A A'$
Compiler verification

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

Proof of concept: a While-to-RISC compiler

- 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* \([\text{Murray}+, \text{AFP}]\), 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 $B$, $I$:

\[ \exists n. \quad n = \text{abs_steps } A \quad A' \]

\[ \exists n. \quad n = \text{abs_steps } A \quad A' \]

\[ \exists n. \quad n = \text{abs_steps } A \quad A' \]

\[ \exists n. \quad n = \text{abs_steps } A \quad A' \]
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? AVR, wasm?
- Branching on H values? Exercise with richer $\mathcal{B}$, $\mathcal{I}$:

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

\[
\exists n . n = \text{abs_steps } A_1 A'_1
\]

\[
\exists n . n = \text{abs_steps } A A'
\]

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:

Provisos and simulation relation:
Invariant on integrity of Switch’s internal state w.r.t. indicator. To appear: EuroS&P’18.