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

Value-dependent classification

Shared-memory concurrency

Overlay Driver
Per-thread compositional verification

Challenge #3: per-thread compositionality of proofs

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

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

Challenge #3: per-thread compositionality of proofs

Source
Per-thread compositional compilation

Challenge #3: per-thread compositionality of proofs

Source

A, B, C etc. obey

Target

A', B', C' etc. obey
Per-thread compositional compilation

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
Challenge #3: per-thread compositionality of proofs

Source

Target

Per-thread compositional compilation

A, B, C etc. obey

A', B', C' etc. obey
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

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

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: \textbf{High} \not\rightarrow \textbf{Low}
Low part of state must remain indistinguishable.
The confidentiality property

Concurrent value-dependent noninterference.

Simplest policy: High $\not\leftrightarrow$ Low

Low part of state must remain indistinguishable.

Reflects the attacker model.
The confidentiality property

Concurrent value-dependent noninterference.

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

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

Low part of state must remain indistinguishable.

$\forall n$. [Diagram of states and transitions]
The confidentiality property

Concurrent value-dependent noninterference.

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

\hspace{1cm}

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

\hspace{1cm}
e.g.
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: High $\not\rightarrow$ Low
Low part of state must remain indistinguishable.

e.g.
The confidentiality property

*Concurrent value-dependent noninterference.*

Simplest policy: High \(\not\rightarrow\) Low

Low part of state must remain indistinguishable.

e.g.
The confidentiality property

Concurrent value-dependent noninterference.

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

Low part of state must remain indistinguishable.

\begin{itemize}
  \item A 2-safety hyperproperty.
\end{itemize}
The confidentiality property

Concurrent value-dependent noninterference.

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

e.g.

- A 2-safety hyperproperty.
- Timing-sensitive. (Want this for concurrency reasons.)
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: **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** $\not\rightarrow$ **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: \( \text{High} \not\rightarrow \text{Low} \)

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.
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: **High** $\not\rightarrow$ **Low**

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

A  B  C

\[\ldots\]

shared mem
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.

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

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]:

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

[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:

$[\text{Murray}+, \text{CSF'16}]$ 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:

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

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

*Concurrency* 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:

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

\[ \text{sched} \]

\[ \text{sched} \]

\[ \text{sched} \]

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

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.

\[ \text{Source} \]

\[ \text{Target} \]
Proof technique for compilation

Per-thread *compositional refinement* [Murray+, CSF’16]
Given bisimulation \( B \) establishing the property,

\[
\forall n. \quad A' \rightarrow^n B \rightarrow^n A
\]
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.:

$\forall n$. 

Source

Target
Proof technique for compilation

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

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

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

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

Source

Target
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.

$$\exists n .$$

$R$ $A_2$ $A_1$

$B$

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

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

$I$ $A'_2$

$A'_1$

(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 \( A \)

(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.

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

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

($\mathcal{I}$ 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:

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

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

Simpler proof technique! Nominate \( \mathcal{R}, \mathcal{I}, \text{abs\_steps} \). 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.

\[ \mathcal{R} \]
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 \( R, I \):

- \( R \) 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

Source

Target
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.

$$\exists n \cdot n = \text{abs\_steps } A A'$$
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. $c_1 ;; c_2$)
  
  \[ \begin{array}{c}
  c_1 \\
  c_2 
  \end{array} \]

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

- 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 $\mathcal{R}$ 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!
  - $e$
  - Store $v \rightarrow r$

- If $e$ $c_1$ $c_2$
  - $e$
  - Jz $r$
  - $c_1$
  - Jmp
  - $c_2$

- While $e$ $c$
  - $e$
  - Jz $r$
  - $c$
  - Jmp

- New!
  - LockAcq $l$
  - LockRel $l$

- New!
  - LockAcq $l$
  - LockRel $l$

- New!
  - 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

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

```
If e c_1 c_2
```

![Diagram of compiler verification process](image)
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 of If construct]
Compiler verification

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

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

```
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 $\mathcal{R}$ characterising a compiler.

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

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

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

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

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

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
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 $\mathcal{B}$ produced by our type system (no H-branching).
  - for $\mathcal{I}$ asserting equal pc and program text.
Compiler verification

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

Proof of concept: a While-to-RISC compiler

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

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

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

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

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

Proof of concept: a While-to-RISC compiler

Source

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>shared mem</td>
<td>shared mem</td>
<td>shared mem</td>
<td>...</td>
</tr>
</tbody>
</table>

Target

<table>
<thead>
<tr>
<th>A'</th>
<th>B'</th>
<th>C'</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>shared mem</td>
<td>shared mem</td>
<td>shared mem</td>
<td>...</td>
</tr>
</tbody>
</table>

A, B, C etc. obey

A', B', C' etc. obey
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.

Diagram showing shared memory regions with A, A', em, and lock symbols.
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

\[ \exists n. n = \text{abs}\_\text{steps} A A' \]
Limitations and future work ideas

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

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

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

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

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:

Provisos and simulation relation:

(See: https://www.isa-afp.org/entries/Dependent_SIFUM_Refinement.html)
Appendix: CDDC 3-component architecture verification

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