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

Robert Sison (UNSW Sydney, Data61) and Toby Murray (University of Melbourne)
September 2019
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*?
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

2. Verified compiler
While-language to RISC-style assembly

(Formalisation: 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*
2. Verified compiler While-language to RISC-style assembly

(Formalisation: [https://covern.org/itp19.html](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

2. Verified compiler

Impact

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

(Proof-of-concept for technique)

(Formalisation: 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

2. Verified compiler While-language to RISC-style assembly

Impact

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

Motivation

(Why all this?)

(Proof-of-concept for technique)

(Technique)

(Formalisation: 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
   ![Diagram of decomposition principle]

2. Verified compiler
   While-language to RISC-style assembly
   ![Diagram of verified compiler]

Impact

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

(Why it’s hard!)

(Why all this?)

Motivation

(Proof-of-concept for technique)

(Reason why)

(Reason why)

(FORMALISATION: https://covern.org/itpl9.html)
Our contributions

Goal

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

Results

1. Decomposition principle for confidentiality-preserving refinement

2. Verified compiler

Impact

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

Motivation

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

(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

Verifying that a compiler preserves concurrent value-dependent infowflow security | Robert Sison and Toby Murray
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)
   - Concurrent value-dependent information-flow security
   - Doesn't leak secrets (storage channels)

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

Confidentiality

Verifying that a compiler preserves concurrent value-dependent infoflow security | Robert Sison and Toby Murray
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)
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.)

3. Compositionally! (per-thread effort)
   - Doesn't leak secrets (storage channels)

Example

(DSTG + Data61 collaboration)

Beaumont et al. (ACSAC’16)
Motivation
Confidentiality for modern software (CSF’16)

1. Multiple moving parts (well-synchronised)
   Doesn’t leak secrets (storage channels)

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

Concurrent value-dependent information-flow security

Verifying that a compiler preserves concurrent value-dependent infoflow security | Robert Sison and Toby Murray
Motivation
Confidentiality for modern software (CSF’16)

TOP SECRET

PROTECTED

Unclassified

Verifying that a compiler preserves concurrent value-dependent infoflow security | Robert Sison and Toby Murray
Motivation

Confidentiality for modern software (CSF'16)

1. Multiple moving parts (well-synchronized)
   - Doesn't leak secrets (storage channels)

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

Concurrent value-dependent information-flow security

Cross Domain Desktop Compositor (CDDC)

Verifying that a compiler preserves concurrent value-dependent infoflow security | Robert Sison and Toby Murray
Motivation

Confidentiality for modern software (CSF’16)

1. Multiple moving parts (well-synchronised)
   - Doesn’t leak secrets (storage channels)

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

Concurrent value-dependent information-flow security

Cross Domain Desktop Compositor (CDDC)

Verifying that a compiler preserves concurrent value-dependent infoflow security | Robert Sison and Toby Murray
Motivation

Confidentiality for modern software (CSF’16)

1. Multiple moving parts (well-synchronized)
   - Doesn’t leak secrets
   - Storage channels
2. Mixed-sensitivity reuse (of devices, space, etc.)

Concurrent value-dependent information-flow security

Cross Domain Desktop Compositor (CDDC)

Verifying that a compiler preserves concurrent value-dependent infoflow security | Robert Sison and Toby Murray
Motivation

Confidentiality for modern software (CSF\'16)

1. Multiple moving parts (well-synchronised)
   - Doesn't leak secrets (storage channels)

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

Concurrent value-dependent information-flow security

Cross Domain Desktop Compositor (CDDC)

Verifying that a compiler preserves concurrent value-dependent infoflow security  |  Robert Sison and Toby Murray
Motivation

Confidentiality for modern software (CSF’16)

1. Multiple moving parts (well-synchronised)
   - Doesn’t leak secrets (storage channels)

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

Concurrent value-dependent information-flow security

Cross Domain Desktop Compositor (CDDC)

Verifying that a compiler preserves concurrent value-dependent infoflow security | Robert Sison and Toby Murray
Motivation

Confidentiality for modern software (CSF'16)

1. Multiple moving parts (well-synchronised)
   - Doesn’t leak secrets
   - Storage channels

2. Mixed-sensitivity reuse (of devices, space, etc.)
   - Concurrent value-dependent information-flow security

Cross Domain Desktop Compositor (CDDC)

Verifying that a compiler preserves concurrent value-dependent infoflow security  |  Robert Sison and Toby Murray
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)

Cross Domain Desktop Compositor (CDDC)

SECRET, PROTECTED, or Unclassified?
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)

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)

Cross Domain Desktop Compositor (CDDC)

seL4-based software architecture

Verifying that a compiler preserves concurrent value-dependent infoflow security | Robert Sison and Toby Murray
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)

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

5 Verifying that a compiler preserves concurrent value-dependent infoflow security | Robert Sison and Toby Murray
Motivation

Confidentiality for modern software (CSF’16)

1. Multiple moving parts (well-synchronised)
   \[\text{Concurrent value-dependent information-flow security}\]

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

3. Compositionally! (per-thread effort)

Doesn't leak secrets (storage channels)

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

Verifying that a compiler preserves concurrent value-dependent infoflow security | Robert Sison and Toby Murray
Motivation
Confidentiality for modern software (CSF’16)

1. Multiple moving parts (well-synchronised)
   \[\Rightarrow\]
   \textbf{Concurrent value-dependent information-flow security}

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

3. \textbf{Compositionaly}! (per-thread effort)

\textit{Doesn't leak secrets (storage channels)}

\textit{Configuration}

\textit{seL4-based software architecture (Case study: simplified model)}

5 Verifying that a compiler preserves concurrent value-dependent infoflow security | Robert Sison and Toby Murray
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.)

3. Compositionally! (per-thread effort)

Doesn't leak secrets (storage channels)

Can a compiler preserve it?

Verifying that a compiler preserves concurrent value-dependent infoflow security | Robert Sison and Toby Murray
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)

Concurrent value-dependent information-flow security

Murray et al. CSF’16

control variable contents (sensitivity-switching)

Some extra stuff to preserve (not that hard)
Motivation

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

Concurrent value-dependent information-flow security

control variable contents (sensitivity-switching)

Murray et al. CSF’16

Some extra stuff to preserve
(not that hard)

This particularly makes it harder!
Motivation

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

Concurrent value-dependent information-flow security

Murray et al. CSF’16

control variable contents (sensitivity-switching)

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)

Concurrent value-dependent information-flow security

Motivation

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

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)

Some extra stuff to preserve
(not that hard)

Murray et al. CSF’16

Some extra stuff to preserve
(sensitivity-switching)

Control variable contents
Mantel et al. CSF’11

Relies/guarantees
(synchronisation)

Concurrent value-dependent information-flow security

This particularly makes it harder!

Interference-resilience (tricky)
Motivation

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

Concurrent value-dependent information-flow security

Mantel et al. CSF’11
relies/guarantees (synchronisation)

Murray et al. CSF’16
control variable contents (sensitivity-switching)

No storage leaks

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)

Concurrent value-dependent information-flow security

Mantel et al. CSF’11
relies/guarantees (synchronisation)

Murray et al. CSF’16
control variable contents (sensitivity-switching)

Some extra stuff to preserve (not that hard)

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)

Concurrent value-dependent information-flow security

Mantel et al. CSF’11

relies/guarantees (synchronisation)

Murray et al. CSF’16

control variable contents (sensitivity-switching)

No storage leaks

Some extra stuff to preserve (not that hard)

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)

Concurrent value-dependent information-flow security

Mantel et al. CSF’11
relied/guarantees (synchronisation)

Murray et al. CSF’16
control variable contents (sensitivity-switching)

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:

Program A

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

Program B

l := v

h isn’t assigned to anything

h isn’t even here!
Motivation

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

Concurrent value-dependent information-flow security

Relies/guarantees (synchronisation) | Control variable contents (sensitivity-switching)
Mantel et al. CSF’11 | Murray et al. CSF’16

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

Thread B
l := v
Motivation

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

Concurrent value-dependent information-flow security

Mantel et al. CSF’11
Murray et al. CSF’16

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

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

```
Thread B
l := v
```

No storage leaks

Verifying that a compiler preserves concurrent value-dependent infoflow security | Robert Sison and Toby Murray
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:

h = 0
Thread A
// Initially, v = 0
if (h) then
  skip
else
  skip; skip
fi
v = 0
Timing leak of h

Thread B
l := v

Mantel et al. CSF’11
relies/guarantees (synchronisation)

Murray et al. CSF’16
control variable contents (sensitivity-switching)

Concurrent value-dependent information-flow security

No storage leaks

Some extra stuff to preserve (not that hard)
Concurrent value-dependent information-flow security

Reasoning:

- Mantel et al. CSF’11: Relies/guarantees (synchronisation)
- Murray et al. CSF’16: Control variable contents (sensitivity-switching)

This particularly makes it harder!

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

Volpano & Smith, CSFW’98

Minimal example:

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

```
Thread B
l := v
```

Schedule $A, A, A, B, \ldots$

No storage leaks

Some extra stuff to preserve (not that hard)
Motivation
Why wouldn’t a compiler preserve it? (CSF’16)

Concurrent value-dependent information-flow security

This particularly makes it harder!
Interference-resilience (tricky) +
Each thread must prevent (scheduler-relative) timing leaks!
Volpano & Smith, CSFW’98

Minimal example:

```plaintext
h = 0
v = 0
Thread A
// Initially, v = 0
if (h) then
  skip
else
  \( \Rightarrow \)skip; skip
fi
v := 1
Timing leak of h

Thread B
l := v
```

No storage leaks ✅

Some extra stuff to preserve (not that hard)

- Mantel et al. CSF’11: relies/guarantees (synchronisation)
- Murray et al. CSF’16: control variable contents (sensitivity-switching)

Verifying that a compiler preserves concurrent value-dependent infoflow security  |  Robert Sison and Toby Murray
Motivation

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

Concurrent value-dependent information-flow security

This particularly makes it harder!
Interference-resilience (tricky) + Each thread must prevent (scheduler-relative) timing leaks!
Volpano & Smith, CSFW’98

Minimal example:

\[
\begin{align*}
  h &= 0 \\
  v &= 0 \\
  \text{Thread A} &
  \begin{cases}
    \text{if } (h) \text{ then} \\
    \quad \text{skip} \\
    \text{else} \\
    \quad \text{skip} \\
    \text{fi} \\
  \end{cases} \\
  \text{Thread B} &
  \begin{cases}
    \text{Schedule } A, A, A, B, \ldots \\
    l &= v \\
  \end{cases}
\end{align*}
\]

No storage leaks

Some extra stuff to preserve (not that hard)
Concurrent value-dependent information-flow security

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

Mantel et al. CSF’11
relies/guarantees (synchronisation)

Murray et al. CSF’16
control variable contents (sensitivity-switching)

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:

<table>
<thead>
<tr>
<th>h = 0</th>
<th>Thread A</th>
</tr>
</thead>
<tbody>
<tr>
<td>// Initially, v = 0</td>
<td></td>
</tr>
<tr>
<td>if (h) then</td>
<td></td>
</tr>
<tr>
<td>skip</td>
<td></td>
</tr>
<tr>
<td>else</td>
<td></td>
</tr>
<tr>
<td>skip</td>
<td></td>
</tr>
<tr>
<td>fi</td>
<td></td>
</tr>
<tr>
<td>v := 1</td>
<td></td>
</tr>
</tbody>
</table>

Timing leak of h

No storage leaks

Volpano & Smith, CSFW’98

Interference-resilience (tricky)
Motivation

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

Concurrent value-dependent information-flow security

This particularly makes it harder!

Interference-resilience (tricky) +

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

Volpano & Smith, CSFW’98

Minimal example:

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

```
Thread B
l = 0
l := v
```

Schedule A, A, A, B, ...

No storage leaks

Some extra stuff to preserve (not that hard)

relied/guarantees (synchronisation)

control variable contents (sensitivity-switching)

Mantel et al. CSF’11

Murray et al. CSF’16
Motivation
Why wouldn’t a compiler preserve it? (CSF’16)

Concurrent value-dependent information-flow security

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:

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

v := 1

Thread A

Schedule A, A, A, B, ...

Thread B

l := v

No storage leaks

Mantel et al. CSF’11
Murray et al. CSF’16

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

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

Concurrent value-dependent information-flow security

- Mantel et al. CSF’11
  - relies/guarantees (synchronisation)
- Murray et al. CSF’16
  - control variable contents (sensitivity-switching)

Some extra stuff to preserve (not that hard)

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:

```
\begin{align*}
\text{Thread A} & \quad h = 0 \\
& \quad h = 1 \quad \text{// Initially, } v = 0 \\
& \quad \text{if } (h) \text{ then} \\
& \quad \quad \text{skip} \\
& \quad \text{else} \\
& \quad \quad \text{skip; skip} \\
& \quad \text{fi} \\
& \quad v := 1 \\
\end{align*}
```

Timing leak of h

```
\text{Thread B} \\
\text{Schedule } A, A, A, B, \ldots
\begin{align*}
& \quad l = 0 \\
& \quad l := v
\end{align*}
```

Volpano & Smith, CSFW’98

Verifying that a compiler preserves concurrent value-dependent infoflow security | Robert Sison and Toby Murray
Motivation

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

Concurrent value-dependent information-flow security

Some extra stuff to preserve (not that hard)

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:

<table>
<thead>
<tr>
<th>h = 0</th>
<th>Thread A</th>
<th>Schedule A, A, A, B, …</th>
</tr>
</thead>
<tbody>
<tr>
<td>h = 1</td>
<td>// Initially, v = 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>if (h) then</td>
<td></td>
</tr>
<tr>
<td></td>
<td>skip</td>
<td></td>
</tr>
<tr>
<td></td>
<td>else</td>
<td></td>
</tr>
<tr>
<td></td>
<td>skip; skip</td>
<td></td>
</tr>
<tr>
<td>v = 0</td>
<td>fi</td>
<td></td>
</tr>
<tr>
<td>v := 1</td>
<td>Timing leak of h</td>
<td></td>
</tr>
</tbody>
</table>

Thread B

l = 0

l := v
Motivation

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

Concurrent value-dependent information-flow security

- Mantel et al. CSF’11
  - relies/guarantees (synchronisation)
- Murray et al. CSF’16
  - control variable contents (sensitivity-switching)

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

\[
\begin{align*}
  &h = 0 \\
  &h = 1 \quad // \text{Initially, } v = 0 \\
  &\text{if } (h) \text{ then} \\
  &\quad \text{skip} \\
  &\text{else} \\
  &\quad \text{skip; skip} \\
  &v = 1 \\
  &\text{fi}
\end{align*}
\]

Timing leak of h

Thread B

\[
\begin{align*}
  \text{Schedule } \mathcal{A}, \mathcal{A}, \mathcal{A}, \mathcal{B}, \ldots \\
  &l = 0 \\
  &l := v
\end{align*}
\]
Motivation

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

Concurrent value-dependent information-flow security

This particularly makes it harder!
Interference-resilience (tricky)

Each thread must prevent (scheduler-relative) timing leaks!
Volpano & Smith, CSFW’98

Minimal example:

<table>
<thead>
<tr>
<th>Schedule</th>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>A, A, A, B, ...</td>
<td>if (h) then skip else skip; skip</td>
<td>(l := v)</td>
</tr>
<tr>
<td>(l = 0)</td>
<td>(v = 1)</td>
<td>(l = 1)</td>
</tr>
</tbody>
</table>

Timing leak of \(h\)
Motivation

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

<table>
<thead>
<tr>
<th>Mantel et al. CSF’11</th>
<th>Murray et al. CSF’16</th>
</tr>
</thead>
<tbody>
<tr>
<td>relies/guarantees (synchronisation)</td>
<td>control variable contents (sensitivity-switching)</td>
</tr>
</tbody>
</table>

Concurrent value-dependent information-flow security

This particularly makes it harder!

Interference-resilience (tricky)

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

Volpano & Smith, CSFW’98

Minimal example:

Thread A

\[
\begin{align*}
  h &= 0 \\
  h &= 1 \\
  \text{// Initially, } v &= 0 \\
  \text{if } (h) \text{ then} \\
  \text{skip} \\
  \text{else} \\
  \text{skip; skip} \\
  v &= 1 \\
  \text{fi} \\
  v &= 1
\end{align*}
\]

Thread B

\[
\begin{align*}
  l &= 0 \\
  l &= 1 \\
  l &:= v
\end{align*}
\]

Schedule \(A, A, A, B, \ldots\)

Timing leak of \(h\)!

Storage leak of \(h\)!

No storage leaks

Some extra stuff to preserve (not that hard)
**Motivation**

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

- Mantel et al. CSF’11
  - relies/guarantees (synchronisation)
- Murray et al. CSF’16
  - 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:

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

v := 1
```

Schedule A, A, A, B, ...

**Thread A**

**Timing fix**

**Thread B**

**Timing leak** of h!

No storage leaks

Storage leak of h!
Concurrent value-dependent information-flow security

Motivation

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

Mantel et al. CSF’11

relies/guarantees (synchronisation)

Murray et al. CSF’16

control variable contents (sensitivity-switching)

This particularly makes it harder!

Interference-resilience (tricky) +

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

Volpano & Smith, CSFW’98

Minimal example:

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

v := 1
```

No storage leaks

But: Compiler may eliminate it!

Thread A

Schedule A, A, A, B, ...

Timed fix

Storage leak of h!

Thread B

l := v
Motivation

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

Mantel et al. CSF’11
relies/guarantees (synchronisation)

Murray et al. CSF’16
control variable contents (sensitivity-switching)

Concurrent value-dependent information-flow security

No storage leaks

But: Compiler may eliminate it!
(or, introduce new “if (h)”!)

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

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

v := 1

Schedule A, A, A, B, ...

Timing fix

Thread B

l := v

Storage leak
of h!

Timing leak
of h
Background

Why is it hard to prove? (CSF’16)

Concurrent value-dependent information-flow security

*preserving* refinement

\[
\begin{align*}
2A & \xrightarrow{B} 2A' \\
1A & \xrightarrow{R} 1A' \\
2C & \xrightarrow{I} 2C' \\
1C & \xrightarrow{R} 1C'
\end{align*}
\]
Background

Why is it hard to prove? (CSF’16)

Confidentiality-preserving refinement

\[
\begin{align*}
2A & \longrightarrow 2A' \\
1A & \longrightarrow 1A' \\
\mathcal{B} & \quad \mathcal{B} \\
\mathcal{R} & \quad \mathcal{R} \\
1C & \longrightarrow 1C' \\
\mathcal{I} & \quad \mathcal{I}
\end{align*}
\]
Background
Why is it hard to prove? (CSF’16)

Confidentiality-preserving refinement

Abstract

Concrete

Direction of compilation

AFP entry:
Dependent_SIFUM_Refinement
Background
Why is it hard to prove? (CSF’16)

Confidentiality-preserving refinement

Abstract

Concrete

Program configurations

Direction of compilation

Verifying that a compiler preserves concurrent value-dependent infoflow security | Robert Sison and Toby Murray
Background
Why is it hard to prove? (CSF’16)

Confidentiality-preserving refinement

Relations
(program configurations)

Abstract
Direction of compilation

Concrete

AFP entry:
Dependent_SIFUM_Refinement

Verifying that a compiler preserves concurrent value-dependent infoflow security  |  Robert Sison and Toby Murray
Background
Why is it hard to prove? (CSF’16)

Confidentiality-preserving refinement

Relations
Execution steps
(between)

Program configurations

Abstract
Direction of compilation
Concrete

AFF entry: Dependent_SIFUM_Refinement
Background

Why is it hard to prove? (CSF’16)

“Usual” refinement:

Abstract

Concrete

Direction of compilation

Verifying that a compiler preserves concurrent value-dependent infoflow security | Robert Sison and Toby Murray
Background
Why is it hard to prove? (CSF’16)

“Usual” refinement:

\[ A \text{ simulates } C \Rightarrow C \text{ refines } A \]

\[
\begin{array}{c}
B \\
1A \rightarrow \cdots \rightarrow \rightarrow 1A' \\
\mid \mid \mid \\
R \\
2C \rightarrow \cdots \rightarrow \rightarrow 2C' \\
\mid \mid \mid \\
I \\
1C \rightarrow \rightarrow 1C'
\end{array}
\]

Abstract
Direction of compilation
Concrete

Verifying that a compiler preserves concurrent value-dependent infoflow security | Robert Sison and Toby Murray
Background
Why is it hard to prove? (CSF’16)

“Usual” refinement:

\[ A \text{ simulates } C \Rightarrow C \text{ refines } A \]

Abstract
Direction of compilation
Concrete

For-all

7 Verifying that a compiler preserves concurrent value-dependent infoflow security | Robert Sison and Toby Murray
Background
Why is it hard to prove? (CSF’16)

“Usual” refinement:

A simulates $C \Rightarrow C$ refines $A$

For-all

Exists

Direction of compilation

Concrete

Abstract

Verifying that a compiler preserves concurrent value-dependent infoflow security  |  Robert Sison and Toby Murray
Background

Why is it hard to prove? (CSF’16)

Confidentiality-preserving refinement

From compiler front-end

Security proof (Bisimulation $B$)

Abstract

Concrete

Direction of compilation

AFP entry: Dependent_SIFUM_Refinement

7 Verifying that a compiler preserves concurrent value-dependent infoflow security | Robert Sison and Toby Murray
Background

Why is it hard to prove? (CSF’16)

Confidentiality-preserving refinement

From compiler front-end

Security proof (Bisimulation $B$)

+ ?

$\Rightarrow$

Security proof (Bisimulation $B_C$)

Compiler correctness proof (Refinement $R$)


depends on $B$, $B_C$

Abstract

$2A \xrightarrow{B} \quad 2A'$

$1A \xrightarrow{	ext{?}} \quad 1A'$

Concrete

$2C \xrightarrow{B_C} \quad 2C'$

$1C \xrightarrow{B_C} \quad 1C'$

AFP entry:

Dependent_SIFUM_Refinement

Verifying that a compiler preserves concurrent value-dependent infoflow security  |  Robert Sison and Toby Murray
Background
Why is it hard to prove? (CSF’16)

From compiler front-end

Security proof (Bisimulation $B$) + Compiler correctness proof (Refinement $R$) $\Rightarrow$ “For free”* Security proof (Bisimulation $B_c$ of $B R I$)

*Confidentiality-preserving refinement

Abstract

Direction of compilation

Concrete

AFP entry: Dependent_SIFUM_Refinement

7 Verifying that a compiler preserves concurrent value-dependent infoflow security | Robert Sison and Toby Murray
Background
Why is it hard to prove? (CSF’16)

*Confidentiality-preserving refinement
(Two-sided!)

From compiler front-end

Security proof (Bisimulation \(B\))

+ Compiler correctness proof (Refinement \(R\))

⇒ “For free”*

Security proof (Bisimulation \(B_c\) of \(B R I\))

\[
\begin{align*}
2A & \xrightarrow{B} 2A' \\
1A & \xrightarrow{R} 1A' \\
2C & \xrightarrow{R} 2C' \\
1C & \xrightarrow{I} 1C'
\end{align*}
\]

Abstract
Direction of compilation
Concrete

AFP entry: Dependent_SIFUM_Refinement

Verifying that a compiler preserves concurrent value-dependent infoflow security | Robert Sison and Toby Murray
Background
Why is it hard to prove? (CSF’16)

Security proof (Bisimulation $B$) + Compiler correctness proof (Refinement $R$) ⇒ “For free”*

*Confidentiality-preserving refinement (Two-sided!)

Compiler correctness proof (Refinement $R$)

Security proof (Bisimulation $B_c$ of $B R I$)

For-all

Exists

Abstract

Direction of compilation

Concrete

AFP entry: Dependent_SIFUM_Refinement

7 Verifying that a compiler preserves concurrent value-dependent infoflow security | Robert Sison and Toby Murray
Background

Why is it hard to prove? (CSF’16)

From compiler front-end

*Confidentiality-preserving refinement

(Two-sided!)

Security proof (Bisimulation $B$)

+ Compiler correctness proof (Refinement $R$)

$\Rightarrow$ “For free”*

Security proof (Bisimulation $B_c$ of $B R I$)

Abstract

Concrete

Direction of compilation

For-all

Exists

For-all

AFP entry: Dependent_SIFUM_Refinement

Verifying that a compiler preserves concurrent value-dependent infoflow security | Robert Sison and Toby Murray
Background

Why is it hard to prove? (CSF’16)

*Confidentiality-preserving refinement

(Two-sided!)

From compiler front-end

Security proof (Bisimulation $B$)

+ Compiler correctness proof (Refinement $R$)

⇒ “For free”*

Security proof (Bisimulation $B_c$ of $B R I$)

Abstract

Direction of compilation

Concrete

Verifying that a compiler preserves concurrent value-dependent infoflow security | Robert Sison and Toby Murray
Background
Why is it hard to prove? (CSF’16)

From compiler front-end

*Confidentiality-preserving refinement

(Two-sided!)

Security proof (Bisimulation $B$)

+ Compiler correctness proof (Refinement $R$)

⇒ “For free”*

Security proof (Bisimulation $B_c$ of $BRI$)

Abstract

Direction of compilation

Concrete

For-all

Exists

For-all

Exists

Compiler: correctness proof (Refinement $R$)

Security proof (Bisimulation $B$)

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

Verifying that a compiler preserves concurrent value-dependent infoflow security | Robert Sison and Toby Murray
Background
Why is it hard to prove? (CSF’16)

(Compare: Barthe et al. CSF’18)

*Confidentiality-preserving refinement
(Two-sided!)

From compiler front-end

Security proof (Bisimulation $B$)

+ Compiler correctness proof (Refinement $R$)

⇒ “For free”*

Security proof (Bisimulation $B_c$ of $B R I$)

FP entry:
Dependent_SIFUM_Refinement

Compiler correctness proof (Refinement $R$)

Direction of compilation

Abstract

Concrete

Verifying that a compiler preserves concurrent value-dependent infoflow security | Robert Sison and Toby Murray
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

(goal = (Technique))
Our contributions

Goal

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

Results

1. Decomposition principle for confidentiality-preserving refinement

2. Verified compiler While-language to RISC-style assembly

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

2. Verified compiler
While-language to RISC-style assembly

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

(Proof-of-concept for technique)
Our contributions

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

Results
1. Decomposition principle for confidentiality-preserving refinement
2. Verified compiler While-language to RISC-style assembly

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

Proof effort almost halved! (Technique)
(Proof-of-concept for technique)
The “cube”, decomposed
Simpler confidentiality-preserving refinement
The “cube”, decomposed
Simpler confidentiality-preserving refinement

\[
\begin{align*}
A & \xrightarrow{\text{abs-steps } A C} A' \\
\mathcal{R} & \\
C & \xrightarrow{} C'
\end{align*}
\]

\[
\begin{align*}
2A & \xrightarrow{\text{abs-steps } 2A 2C} \\
\mathcal{B} & = \\
1A & \xrightarrow{\text{abs-steps } 1A 1C} \\
\mathcal{R} & \\
2C & \xrightarrow{\text{stops } 2C} \\
\mathcal{I} & = \\
1C & \xrightarrow{\text{stops } 1C} \\
\mathcal{R} &
\end{align*}
\]

\[
\begin{align*}
2A & \xrightarrow{} 2A' \\
\mathcal{B} & \\
1A & \xrightarrow{\text{steps } 1A} \\
\mathcal{R} & \\
2C & \xrightarrow{} 2C' \\
\mathcal{I} & = \\
1C & \xrightarrow{} 1C'
\end{align*}
\]

\[
\text{implies}
\]

9 Verifying that a compiler preserves concurrent value-dependent infoflow security | Robert Sison and Toby Murray
The “cube”, decomposed
Simpler confidentiality-preserving refinement

1. “Usual” proof of refinement

\[
2A \rightarrow \rightarrow \text{abs-steps} \rightarrow 2A 2C \\
\]

\[
1A \rightarrow \rightarrow \text{abs-steps} \rightarrow 1A 1C \\
\]

\[
2C \rightarrow \rightarrow \text{stops} \rightarrow 2C \\
\]

\[
1C \rightarrow \rightarrow \text{stops} \rightarrow 1C \\
\]

\[
2A \\
\]

\[
1A \\
\]

\[
2C \\
\]

\[
1C \\
\]
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” \( \text{abs-steps} \)
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” \( \text{abs-steps} \)
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” \textit{abs-steps} for (refinement) relation \( R \)

Security witness (bisimulation) relation \( B \)

1. “Usual” proof of refinement

Standard compiler correctness!
(+ “extra stuff” for conc, val-dep)

2. Consistent pacing

Verifying that a compiler preserves concurrent value-dependent infoflow security | Robert Sison and Toby Murray
The “cube”, decomposed
Simpler confidentiality-preserving refinement

“Pacing function” $\text{abs-steps}$ 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” \( \text{abs-steps} \) for (refinement) relation \( R \)

\[
\begin{align*}
A & \xrightarrow{\text{abs-steps}} A' \\
1A & \xrightarrow{\text{abs-steps}} 1A 1C \\
B & = \\
2A & \xrightarrow{\text{abs-steps}} 2A 2C
\end{align*}
\]

Security witness (bisimulation) relation \( B \)

\[
\begin{align*}
2A & \xrightarrow{\text{abs-steps}} 2A 2C \\
1A & \xrightarrow{\text{abs-steps}} 1A 1C \\
B & = \\
2A & \xrightarrow{\text{abs-steps}} 2A 2C
\end{align*}
\]

1. “Usual” proof of refinement

Standard compiler correctness!

(+ “extra stuff” for conc, val-dep)

2. Consistent pacing

Verifying that a compiler preserves concurrent value-dependent infoflow security | Robert Sison and Toby Murray
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

Standard compiler correctness!
(+ “extra stuff” for conc, val-dep)
The “cube”, decomposed
Simpler confidentiality-preserving refinement

“Pacing function” $\text{abs-steps}$ for (refinement) relation $R$

1. “Usual” proof of refinement

Security witness (bisimulation) relation $B$

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” \( \text{abs-steps} \)
for (refinement) relation \( R \)

Security witness (bisimulation) relation \( B \)

1. “Usual” proof of refinement

Standard compiler correctness!
(+ “extra stuff” for conc, val-dep)

2. Consistent pacing and
3. Consistent stopping
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” \(\text{abs-steps}\) for (refinement) relation \(R\)

Security witness (bisimulation) relation \(B\)

1. “Usual” proof of refinement

Standard compiler correctness!

(“extra stuff” for conc, val-dep)

Verifying that a compiler preserves concurrent value-dependent infoflow security | Robert Sison and Toby Murray
The “cube”, decomposed
Simpler confidentiality-preserving refinement

“Pacing function” $abs\text{-}steps$
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
Standard compiler correctness!
(+ “extra stuff” for conc, val-dep)

2. Consistent pacing and
3. Consistent stopping

4. Closedness of “Concrete coupling invariant” relation $I$
The “cube”, decomposed
Simpler confidentiality-preserving refinement

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

$$A \xrightarrow{\text{abs-steps}} AC \xrightarrow{\text{abs-steps}} A'$$

$$R \xrightarrow{\text{abs-steps}} C \xrightarrow{\text{abs-steps}} C'$$

1. “Usual” proof of refinement

Standard compiler correctness!
(+ “extra stuff” for conc, val-dep)

Security witness (bisimulation) relation $B$

$$2A \xrightarrow{\text{abs-steps}} 2A 2C$$

$$1A \xrightarrow{\text{abs-steps}} 1A 1C$$

$$R \xrightarrow{\text{abs-steps}} 2A \xrightarrow{\text{abs-steps}} R$$

$$R \xrightarrow{\text{abs-steps}} 1A \xrightarrow{\text{abs-steps}} R$$

$$C \xrightarrow{\text{abs-steps}} 2C \xrightarrow{\text{abs-steps}} C'$$

$$I \xrightarrow{\text{abs-steps}} 2C \xrightarrow{\text{abs-steps}} I'$$

2. Consistent pacing and stopping

No new timing and termination leaks!

3. Consistent stopping

4. Closedness of “Concrete coupling invariant” relation $I$
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
   Standard compiler correctness!
   (+ “extra stuff” for conc, val-dep)

2. Consistent pacing and stopping

3. Consistent stopping

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

No new timing and termination leaks!
Proof effort comparison

Refinement example (excerpt) from CSF’16

if \( h \neq 0 \) then
    \( x := y \)
else
    \( x := y + z \)
fi

Abstract program

reg3 := h;
if reg3 \neq 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

\[
\begin{align*}
\text{if } h \neq 0 \text{ then} & \quad x := y \\
\text{else} & \quad x := y + z \\
\text{fi}
\end{align*}
\]

Concrete program

\[
\begin{align*}
\text{reg3 := } h; \\
\text{if } \text{reg3} \neq 0 \text{ then} & \quad \text{skip; skip; reg0 := y; x := reg0} \\
\text{else} & \quad \text{reg1 := y; reg2 := z; reg0 := reg1 + reg2; reg0 := reg0} \\
\text{fi}
\end{align*}
\]

formalisation artifact:
https://covern.org/itp19.html
Proof effort comparison

Refinement example (excerpt) from CSF’16

Abstract program

if $h \neq 0$ then
  $x := y$
else
  $x := y + z$
fi

Concrete program

reg3 := $h$
if reg3 $\neq 0$ then
  skip;
  skip;
  reg0 := $y$
  $x := reg0$
else
  reg1 := $y$
  reg2 := $z$
  reg0 := reg1 + reg2;
  reg0 := reg0
fi

branch on secret

padding to prevent timing leak

formalisation artifact:
https://covern.org/itp19.html
Proof effort comparison

Refinement example (excerpt) from CSF’16

Abstract program

\[
\begin{align*}
\textbf{if } & h \neq 0 \textbf{ then} \\
& x := y \\
\textbf{else} \\
& x := y + z \\
\textbf{fi}
\end{align*}
\]

Concrete program

\[
\begin{align*}
\textbf{if } & \text{reg3} \neq 0 \textbf{ then} \\
& \text{skip; skip; reg0 := y; x := reg0} \\
\textbf{else} \\
& \text{reg1 := y; reg2 := z; reg0 := reg1 + reg2; reg0 := reg0} \\
\textbf{fi}
\end{align*}
\]

• 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

   Goal = Results

   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

2. Verified compiler

While-language to RISC-style assembly

Impact

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

Proof effort almost halved!

(Proof-of-concept for technique)
An Isabelle/HOL \texttt{primrec} function

\begin{itemize}
  \item \texttt{While} (Imperative)
    \begin{itemize}
      \item \texttt{Seq} (i.e. \texttt{c\_1 ;; c\_2})
      \item \texttt{Assign} (i.e. \texttt{v \leftarrow e})
      \item \texttt{If} \texttt{e c\_1 c\_2}
      \item \texttt{While} \texttt{e c}
      \item \texttt{Skip}
    \end{itemize}
  \item \texttt{RISC} (Assembly)
    \begin{itemize}
      \item \texttt{Load r v}
      \item \texttt{Store v r}
      \item \texttt{Jmp l}
      \item \texttt{Jz l r}
      \item \texttt{Nop}
    \end{itemize}
\end{itemize}

(Formalisation: \url{https://covern.org/itp19.html})
An Isabelle/HOL `primrec` function

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
- ...

(Based on: Tedesco et al. CSF’16)

(Formalisation: [https://covern.org/itp19.html](https://covern.org/itp19.html))
**Verified compiler**

**Overview**

An Isabelle/HOL *primrec* function

```
While (Imperative)
  Seq (i.e. c1 ;; c2)
  Assign (i.e. v ← e)
  If e c1 c2
  While e c
  Skip
  ...
```

```
RISC (Assembly)
  Load r v
  Store v r
  Jmp l
  Jz l r
  Nop
  ...
```

- 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](https://covern.org/itp19.html))

(Based on: Tedesco et al. CSF’16)
Verified compiler

Overview

An Isabelle/HOL primrec function

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
- ...

- 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/itpl9.html)

(Based on: Tedesco et al. CSF’16)
Verified compiler

Overview

An Isabelle/HOL *primrec* function

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
...

- 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](https://covern.org/itp19.html))

(Based on: Tedesco et al. CSF’16)
Verified compiler
Overview

An Isabelle/HOL primrec function

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
  ...

• 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

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

While

RISC

(Formalisation: [https://covern.org/itp19.html](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 \((\text{Inductive})\)

(Formalisation: [https://covern.org/itp19.html](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 \textit{(Inductive)}

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

(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 \((Inductive)\)

(Formalisation: [https://covern.org/itp19.html](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 \textit{(Inductive)}

(Formalisation: \url{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 \textit{(Inductive)}

\begin{align*}
\text{If} & \quad e \quad \text{Jz} \quad r \\
& \quad e \\
& \quad c_1 \\
& \quad Jmp \\
& \quad c_2
\end{align*}

- \textbf{Theorem}: $R$ (for $B$, with $I$) is a secure refinement

(Formalisation: \url{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 \textit{(Inductive)}

\textbf{Theorem}: $R$ (for $B$, with $I$) is a secure refinement (via \textit{decomposition principle})

(Formalisation: \url{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 \textit{(Inductive)}

• \textbf{Theorem}: $R$ (for $B$, with $I$) is a secure refinement (via \textit{decomposition principle})

• \textbf{Theorem}: Compiler input related to output by $R$

(Formalisation: \url{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 \textit{(Inductive)}

- \textbf{Theorem:} $R$ (for $B$, with $I$) is a secure refinement (via decomposition principle)

- \textbf{Theorem:} Compiler input related to output by $R$

(Formalisation: \url{https://covern.org/itp19.html})

13 Verifying that a compiler preserves concurrent value-dependent infoflow security | Robert Sison and Toby Murray
**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

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

**Theorem:** Compiler input related to output by $R$

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

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

• Theorem: Compiler input related to output by $R$

(Formalisation: https://covern.org/itp19.html)
An Isabelle/HOL `primrec` function

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
- ...

• Proof approach: \( \sim 7K \) 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](https://covern.org/itp19.html))
Flushed

Verifying that a compiler preserves concurrent value-dependent infoflow security  |  Robert Sison and Toby Murray

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

(Based on: Tedesco et al. CSF’16)

(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. \( v \leftarrow e \))
If \( e c_1 c_2 \)
While \( e c \)
Skip
...

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

(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. \( v \leftarrow e \))
If \( e \; c_1 \; c_2 \)
While \( e \; c \)
Skip
...

Concurrent input-handling architecture
(*extremely simplified*)

(Formalisation: [https://covern.org/itp19.html](https://covern.org/itp19.html))
Verified compiler

Application: CDDC input-handling model

-~150 lines While-

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
...

(Formalisation: https://covern.org/itp19.html)
Verified compiler
Application: CDDC input-handling model

~150 lines While

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
...

(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. \( v \leftarrow e \))
If \( e \) \( c_1 \) \( c_2 \)
While \( e \) \( c \)
Skip
...

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

(Formalisation: https://covern.org/itp19.html)
Verified compiler
Application: CDDC input-handling model

~150 lines While

Proof effort preserved!

~250 RISC instructions

While (Imperative)
Seq (i.e. c₁ ;; c₂)
Assign (i.e. v ← e)
If e c₁ c₂
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

Proof effort preserved!

~250 RISC instructions

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
...

(Formalisation: https://covern.org/itp19.html)
Verified compiler
Application: CDDC input-handling model

~150 lines **While**

**Proof effort preserved!**

~250 **RISC** instructions

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

**RISC** (Assembly)
Load \( r \quad v \)
Store \( v \quad r \)
Jmp \( l \)
Jz \( l \quad r \)
Nop
...

(Formalisation: [https://covern.org/itp19.html](https://covern.org/itp19.html))
Verified compiler
Application: CDDC input-handling model

~150 lines While

Proof effort preserved!

~250 RISC instructions

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
...

(Formalisation: https://covern.org/itp19.html)
**Verified compiler**

**Overview**

An Isabelle/HOL *primrec* function

- **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
  - ...

- **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](https://covern.org/itp19.html))

(Based on: Tedesco et al. CSF’16)
Verified compiler

Overview

An Isabelle/HOL *primrec* function

\begin{align*}
\text{While (Imperative)} & : \text{Seq (i.e. } c_1 ;; c_2) \\
& \text{Assign (i.e. } v \leftarrow e) \\
& \text{If } e \text{ c}_1 \text{ c}_2 \\
& \text{While } e \text{ c} \\
& \text{Skip} \\
& \ldots
\end{align*}

\begin{align*}
\text{RISC (Assembly)} & : \text{Load } r \text{ v} \\
& \text{Store } v \text{ r} \\
& \text{Jmp } l \\
& \text{Jz } l \text{ r} \\
& \text{Nop} \\
& \ldots
\end{align*}

- 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](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

   ![Diagram showing decomposition principle]

   Proof effort almost halved!

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

2. Verified compiler While-language to RISC-style assembly

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

\[ A_{abs-steps}^1 C = A_{abs-steps}^2 \]

\[ B_{stops}^1 C_{stops} = C_{stops}^2 \]

2. Verified compiler

While-language to RISC-style assembly

Proof effort almost halved!

(Technique)

(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. \(c_1 ; c_2\))
  - \(c_1\)
  - \(c_2\)

- Assign (i.e. \(v \leftarrow e\))
  - \(e\)
  - Store \(v\)
  - \(r\)

- If \(e\) \(c_1\) \(c_2\)
  - \(e\)
  - \(Jz\)
  - \(r\)
  - \(c_1\)
  - \(Jmp\)
  - \(c_2\)

- While \(e\) \(c\)
  - \(e\)
  - \(Jz\)
  - \(r\)
  - \(c\)
  - \(e\)
  - \(Jmp\)

- Skip
  - \(Nop\)

Tedesco et al. CSF'16

Verifying that a compiler preserves concurrent value-dependent infoflow security  |  Robert Sison and Toby Murray
Appendix

Differences from Tedesco et al. CSF'16 compilation scheme

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

- Assign (i.e. \( v \leftarrow e \))
  - \( e \)
  - \( \text{Store } v \)

- 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

Our While-to-RISC compiler