Compositional Secure Compilation against Spectre

Marco Patrignani\textsuperscript{1,2}
Compositional Secure Compilation against Spectre

Special thanks to
1. **Formal framework** for assessing security of Spectre v1 compiler countermeasures
Contributions & Outline

1. **Formal framework** for assessing security of Spectre v1 compiler countermeasures

2. **Proofs of security** and insecurity of existing Spectre v1 compiler countermeasures
1. Formal framework for assessing security of Spectre v1 compiler countermeasures
   - Based on recent secure compilation theory

2. Proofs of security and insecurity of existing Spectre v1 compiler countermeasures
Contributions & Outline

1. **Formal framework** for assessing security of Spectre v1 compiler countermeasures
   - Based on recent **secure compilation** theory
   - Preservation of SNI: **semantic** notion of security

2. **Proofs of security** and insecurity of existing Spectre v1 compiler countermeasures

---

Guarnieri et al. S&P'19
Contributions & Outline

1. **Formal framework** for assessing security of Spectre v1 compiler countermeasures
   - Based on recent *secure compilation* theory
   - Preservation of SNI: *semantic* notion of security
     - *Source* semantics is “standard”  
     
2. **Proofs of security** and insecurity of existing Spectre v1 compiler countermeasures
Contributions & Outline

1. **Formal framework** for assessing security of Spectre v1 compiler countermeasures
   - Based on recent *secure compilation* theory
   - Preservation of SNI: *semantic* notion of security
     - *Source* semantics is “standard”
     - *Target* semantics is speculative

2. **Proofs of security** and insecurity of existing Spectre v1 compiler countermeasures
1. **Formal framework** for assessing security of Spectre v1 compiler countermeasures
   - Based on recent *secure compilation* theory
   - Preservation of SNI: *semantic* notion of security
   - **Source** semantics is “standard”
   - **Target** semantics is speculative

2. **Proofs of security** and insecurity of existing Spectre v1 compiler countermeasures
   - **Secure**: Intel ICC, SLH(ish), SSLH
   - **Insecure**: MSVC, Interprocedural SLH
1. **Formal framework** for assessing security of Spectre v1 compiler countermeasures
   - Based on recent **secure compilation** theory
   - Preservation of SNI: **semantic** notion of security
     - **Source** semantics is “standard”
     - **Target** semantics is speculative

2. **Proofs of security** and insecurity of existing Spectre v1 compiler countermeasures
   - **Secure**: Intel ICC, SLH(ish), SSLH
   - **Insecure**: MSVC, Interprocedural SLH
1. **Formal framework** for assessing security of Spectre v1 compiler countermeasures
   - Based on recent secure compilation theory
   - Preservation of SNI: semantic notion of security (Guarnieri et al. S&P’19)
   - Source semantics is “standard”
   - Target semantics is speculative

2. **Proofs of security** and insecurity of existing Spectre v1 compiler countermeasures
   - **Secure:** Intel ICC, SLH(ish), SSLH
   - **Insecure:** MSVC, Interprocedural SLH
Contributions & Outline

(3) Formal framework for assessing security of Spectre v1 compiler countermeasures
- Based on recent secure compilation theory
- Preservation of SNI: semantic notion of security
  Guarnieri et al. S&P'19

(2) Source semantics is "standard"
- Target semantics is speculative

(1)

2. Proofs of security and insecurity of existing Spectre v1 compiler countermeasures
- Secure: Intel ICC, SLH(ish), SSLH
- Insecure: MSVC, Interprocedural SLH
Contributions & Outline

1. Formal framework for assessing security of Spectre v1 compiler countermeasures
   - Based on recent secure compilation theory
   - Preservation of SNI: semantic notion of security
     - Source semantics is “standard”
     - Target semantics is speculative

2. Proofs of security and insecurity of existing Spectre v1 compiler countermeasures
   - Secure: Intel ICC, SLH(ish), SSLH
   - Insecure: MSVC, Interprocedural SLH
Contributions & Outline

(3) Formal framework for assessing security of Spectre v1 compiler countermeasures

(5) What about preserving multiple variants?

(1) Source semantics is “standard”
(2) Target semantics is speculative

2. Proofs of security and insecurity of existing Spectre v1 compiler countermeasures

(4) Secure: Intel ICC, SLH(ish), SSLH

• Insecure: MSVC, Interprocedural SLH
Contributions & Outline

1. Formal framework for assessing security of Spectre v1 compiler countermeasures
   - Based on recent secure compilation theory
   - Preservation of SNI: semantic notion of security
     - Guarnieri et al.
   - Source semantics is “standard”
   - Target semantics is speculative

2. Proofs of security and insecurity of existing Spectre v1 compiler countermeasures
   - Secure: Intel ICC, SLH(ish), SSLH
   - Insecure: MSVC, Interprocedural SLH

(5) What about preserving multiple variants?
Composition (wip)
Speculative Semantics & SNI

```c
void f (int x) { if (x < A.size) { y = B[A[x]]; }
```
Speculative Semantics & SNI

```c
void f (int x) ↦ if(x < A.size) \{ y = B[A[x]] \}
```

call f 128

**Speculative Semantics & SNI**

```c
void f (int x) => if (x < A.size) { y = B[A[x]] }
```

void f (int x) \rightarrow \text{if}(x < A\text{.}size) \{ y = B[A[x]] \}
run 1: A\text{.}size = 16, A[128] = 3
Speculative Semantics & SNI

```c
void f (int x) { if (x < A.size) { y = B[A[x]] } }
```


Speculative Semantics & SNI

```c
void f (int x) { 
    if (x < A.size) { 
        y = B[A[x]]
    }
}
```


call f 128

if (128 < 16) { y = B[A[128]] } 

skip

y = B[A[128]]

rd A[128]

y = B[3]

rd B[3]

y = _

A program is SNI (\(P : SNI\)) if, given two runs from low-equivalent states:

- if the non-speculative traces are low-equivalent
- then the speculative traces are also low-equivalent

A program attains SNI robustly (\(P : RSNI\)) if it is SNI no matter what attacker it links against.

\(\forall A. A[P] : SNI\)
Speculative Semantics & SNI

```c
void f (int x) { if (x < A.size) { y = B[A[x]] } }
```


call f 128

if (128 < 16) { y = B[A[128]] }

skip

y = B[A[128]]


y = B[3]

y = _

different traces ⇒ SNI violation

A program is SNI (\(P : \text{SNI}\)) if, given two runs from low-equivalent states:

- if the non-speculative traces are low-equivalent
- then the speculative traces are also low-equivalent

A program attains SNI robustly (\(P : \text{RSNI}\)) if it is SNI no matter what attacker it links against.

\(\forall A. \ A[P] : \text{SNI}\)
Speculative Semantics & SNI

void f (int x) ↦ if(x < A.size) { y = B[A[x]]}


A program is SNI (\(\overset{\text{P}}{\overset{\text{SNI}}{\Rightarrow}}\)) if, given two runs from low-equivalent states:
• - if the non-speculative traces are low-equivalent
• - then the speculative traces are also low-equivalent

A program attains SNI robustly (\(\overset{\text{P}}{\overset{\text{RSNI}}{\Rightarrow}}\)) if it is SNI no matter what attacker it links against.
\(\forall A.\overset{\text{A}}{\overset{\text{P}}{\overset{\text{SNI}}{\Rightarrow}}}\)

\(\text{two.osf//two.osf//zero.osf}\)
Speculative Semantics & SNI

```c
void f (int x) { 
  if (x < A.size) { y = B[A[x]] } 
}
```

run 2: A[128] = 7 different H values

call f 128


A program is SNI (∷ P :: SNI) if, given two runs from low-equivalent states:
• if the non-speculative traces are low-equivalent
• then the speculative traces are also low-equivalent

A program attains SNI robustly (∷ P :: RSNI) if it is SNI no matter what attacker $A$ it links against.

∀ A. $\forall A[P::SNI] :: SNI$ /two.osf//two.osf/zero.osf
void f (int x) ↦ if (x < A.size) { y = B[A[x]] }
run 2: A[128] = 7 different H values

call f 128
if (128 < 16) { y = B[A[128]] }

rd A[128]
void f (int x) ↔ if(x < A.size) {y = B[A[x]]}


run 2: A[128] = 7 different H values

call f 128

void f (int x) \implies \text{if}(x < A.\text{size}) \{ y = B[A[x]] \}

run 1: A.\text{size} = 16, A[128] = 3
run 2: A[128] = 7 \quad \text{different H values}

call f 128

if (128 < 16) \{ y = B[A[128]] \}
	race 1: \quad \text{rd} A[128] \quad \text{rd} B[3] \quad \text{different traces}
trace 2: \quad \text{rd} A[128] \quad \text{rd} B[7] \quad \Rightarrow \text{SNI violation}
A program is **SNI** (\(\vdash P : \text{SNI}\)) if, given two runs from low-equivalent states:

- if the non-speculative traces are low-equivalent
- then the speculative traces are also low-equivalent

trace 1: \(\text{rd } A[128]\) \(\text{rd } B[3]\) different traces \(\Rightarrow\) SNI violation

trace 2: \(\text{rd } A[128]\) \(\text{rd } B[7]\)
Speculative Semantics & SNI

```c
void f (int x) { if (x < A.size) { y = B[A[x]] } }

run 2: A[128] = 7 different H values
```

call f 128

```c
if (128 < 16) { y = B[A[128]] }
```

trace 1: 
rd A[128]  
rd B[3] different traces

trace 2: 
rd A[128]  
rd B[7] ⇒ SNI violation
A program attains \textbf{SNI robustly} (\(\vdash P : \text{RSNI}\)) if it is \textbf{SNI} no matter what attacker \(A\) it links against.

\[\forall A. \vdash A[P] : \text{SNI}\]

Trace 1: \(\text{rd } A[128]\) \(\text{rd } B[3]\) different traces \(\Rightarrow\) SNI violation

Trace 2: \(\text{rd } A[128]\) \(\text{rd } B[7]\)
Problem: Proving compiler preserves RSNI is hard
Problem: Proving compiler preserves RSNI is hard

Solution: overapproximate RSNI with a novel property: robust speculative safety (RSS)
Speculative Safety ($RSS$)

Semantic-Irrelevant Taint Tracking
Speculative Safety ($RSS$)

**Semantic-Irrelevant Taint Tracking**

```c
void f (int x) { if (x < A.size) { y = B[A[x]] } }
```

only 1 run needed: A.size=16, A[128]=3

**integrity lattice:** $S \subset U \quad S \cap U = S \quad U$ does not flow to $S$

---

call f 128

call 128

call 128

pc : S

```c
if (128 < 16) { y = B[A[128]] }
```

pc : S

---

A program is SS ($P : SS$) if its traces do not contain $U$ actions

A program is SS robustly ($P : RSS$) if it is SS no matter what attacker $A$ it links against.
Speculative Safety (RSS)

Semantic-Irrelevant Taint Tracking

void f (int x) \iff (x < A.size) \{ y = B[A[x]] \}

only 1 run needed: A.size=16, A[128]=3

integrity lattice: \( S \subset U \quad S \cap U = S \quad U \) does not flow to \( S \)

call f 128 pc : S
Speculative Safety (RSS)

Semantic-Irrelevant Taint Tracking

```c
void f (int x) \iff (x < A.size) \{ y = B[A[x]] \}
```

Only 1 run needed: A.size=16, A[128]=3

Integrity lattice: $S \subset U \quad S \cap U = S \quad U$ does not flow to $S$

---

call f 128

$pc : S$

if (128 < 16) \{ y = B[A[128]] \}

$pc : S$

$128 : S$

$y = B[A[128]]$

$pc : U$
Speculative Safety (RSS)

Semantic-Irrelevant Taint Tracking

void f (int x) ↔ if (x < A.size) { y = B[A[x]] } 
only 1 run needed: A.size=16, A[128]=3 
integrity lattice: $S \subset U$, $S \cap U = S$, $U$ does not flow to $S$

call f 128 
   pc : S

call f 128 
   pc : S

   pc : S

rd A[128] : S

128 : S

   pc : U


y = B[ 3 ] 
   pc : U
Speculative Safety ($\textit{RSS}$)

Semantic-Irrelevant Taint Tracking

```c
void f (int x) { 
  if (x < A.size) { 
    y = B[A[x]]
  }
}
```

only 1 run needed: A.size=16, A[128]=3

integrity lattice: $S \subset U$, $S \cap U = S$, $U$ does not flow to $S$
Speculative Safety \((RSS)\)

Semantic-Irrelevant Taint Tracking

```c
void f (int x) { if (x < A.size) { y = B[A[x]]; } }
```

only 1 run needed: A.size=16, A[128]=3

integrity lattice: \(S \subset U\) \(S \cap U = S\) \(U\) does not flow to \(S\)

```plaintext
call f 128
pc : S

if (128 < 16) { y = B[A[128]] } pc : S

skip
pc : S

rd A[128] : S

```
Speculative Safety ($RSS$)

A program is SS (\(\perp P : SS\)) if its traces do not contain U actions.

A program is SS robustly (\(\perp P : RSS\)) if it is SS no matter what attacker A it links against.
Speculative Safety (RSS)

Semantic-Irrelevant Taint Tracking

```c
void f (int x) { 
  if (x < A.size) { 
    y = B[A[x]];
  }
}
```

only 1 run needed: A.size=16, A[128]=3

integrity lattice: \( S \subseteq U \) \( S \cap U = S \) \( U \) does not flow to \( S \)

call f 128

\( pc : S \)

if (128 < 16) { 
  y = B[ A[ 128 ] ];
}

\( pc : S \)

rd A[128] : S

Robustness & Preserving NI-like Props

1. preserve for whole programs (e.g., CCT), cube-like proofs
Robustness & Preserving NI-like Props

1. preserve for whole programs (e.g., CCT), cube-like proofs
2. preserve safety overapproximation, for partial programs and robustly
Robustness & Preserving NI-like Props

1. preserve for whole programs (e.g., CCT), cube-like proofs
2. preserve safety overapproximation, for partial programs and robustly

Robustness pros and cons:

✓ realistic, (not) lossy, precise attacker + actions awareness
X coqability, precision, sometimes inefficient
RSS and RSNI

RSS overapproximates RSNI, so:

- in the **target**: \( \forall P. \vdash P : \text{RSS} \Rightarrow \vdash P : \text{RSNI} \)
RSS and RSNI

RSS overapproximates RSNI, so:

• in the **target**: $\forall P. \vdash P : \text{RSS} \Rightarrow \vdash P : \text{RSNI}$

• in the **source**: $\forall P. \vdash P : \text{RSS} \iff \vdash P : \text{RSNI}$

(recall, no speculative execution in **source**)
RSS-Preserving Compiler: \textit{RSSC} \& \textit{RSSP}

\[
\llbracket \cdot \rrbracket : \text{RSSP} \overset{\text{def}}{=} \text{if } \forall A. \vdash A [P] : \text{RSS} \text{ and } \text{RSS} \sim \text{RSS} \text{ then } \forall A. \vdash A [\llbracket P \rrbracket] : \text{RSS}
\]
RSS-Preserving Compiler: \( \text{RSSC} \& \text{RSSP} \)

\[
\llbracket \cdot \rrbracket : \text{RSSP} \overset{\text{def}}{=} \text{if } \forall A. \leftarrow A [P] : \text{RSS} \text{ and } \text{RSS} \sim \text{RSS} \text{ then } \forall A. \leftarrow A \llbracket [P] \rrbracket : \text{RSS}
\]

\[
\llbracket \cdot \rrbracket : \text{RSSC} \overset{\text{def}}{=} \text{if } \forall A.A \llbracket [P] \rrbracket \sim m \text{ then } \exists A.A [P] \sim m \sim m
\]

\( \sim = \) same traces, plus \( S \) actions in \( m \)
RSS-Preserving Compiler: RSSC & RSSP

\[ \text{def} \quad [\cdot] : \text{RSSP} \overset{\text{def}}{=} \text{if } \forall A. \vdash A [P] : \text{RSS} \text{ and } \text{RSS} \sim \text{RSS} \text{ then } \forall A. \vdash A [[P]] : \text{RSS} \]

\[ \text{def} \quad [\cdot] : \text{RSSC} \overset{\text{def}}{=} \text{if } \forall A.A [[P]] \sim m \text{ then } \exists A.A [P] \sim m \approx m \]

\[ \approx= \text{ same traces, plus } S \text{ actions in } m \]

- \forall A: explicit attacker model (robustness)
RSS-Preserving Compiler: RSSC & RSSP

\[ \llbracket . \rrbracket : \text{RSSP def } if \ \forall A. \vdash A [P] : \text{RSS and RSS } \sim \text{RSS} \]

then \ \forall A. \vdash A [[P]] : \text{RSS} \]

\[ \llbracket . \rrbracket : \text{RSSC def } if \ \forall A. A [[P]] \sim m \text{ then } \exists A.A [P] \sim m \sim m \]

\( \sim = \text{ same traces, plus } S \text{ actions in } m \)

- \( \forall A: \text{ explicit attacker model (robustness)} \)
- Proof: RSSC & RSSP are equivalent

RSSC : clear security guarantees
RSSP : simpler proofs
RSS-Preserving Compiler: **RSSC & RSSP**

$\llbracket \cdot \rrbracket : \text{RSSP} \overset{\text{def}}{=} \text{if } \forall A. A[P] : \text{RSS and } \text{RSS} \sim \text{RSS} \\\n\text{then } \forall A. A [[P]] : \text{RSS}$

$\llbracket \cdot \rrbracket : \text{RSSC} \overset{\text{def}}{=} \text{if } \forall A. A [[P]] \rightarrow m \text{ then } \exists A. A[P] \rightarrow m \approx m$\n
$\overset{\text{def}}{=} \text{same traces, plus } S \text{ actions in } m$

- $\forall A$: explicit attacker model (**robustness**)  

**Danger:** RSSC & RSSP are **equivalent**

**Proof:** RSSC : clear security guarantees  
RSSP : simpler proofs
Secure Compilation Framework for Spectre

∀P ∈ source
⊢ P : RSS

⊢ [·] : RSSC

criteria equality

⊢ [·] : RSSP

⊢ [P] : RSS

overapproximation

⊢ [P] : RSNI
Secure Compilation Framework for Spectre

\[ \forall P \in \text{source} \]

\[ \vdash P : \text{RSS} \]

\[ \vdash [\cdot] : \text{RSSC} \]

\[ \vdash [\cdot] : \text{RSSP} \]

\[ \vdash [P] : \text{RSS} \]

\[ \vdash [P] : \text{RSNI} \]

- all source programs are trivially RSS
∀ P ∈ source
\[ P : RSS \]

\[ [\cdot] : RSSC \]

criteria
equality

\[ [\cdot] : RSSP \]

\[ [P] : RSS \]

overapproximation

\[ [P] : RSNI \]

- all source programs are trivially RSS
- to show security: simply prove RSSC
Preservation or Enforcement?

Is it still preservation if the property is trivial in source?
Preservation or Enforcement?

Is it still preservation if the property is trivial in source?

Preservation can generalise the proof
Preservation or Enforcement?

Is it still preservation if the property is trivial in source?

Preservation can generalise the proof

Enforcement cannot work for classes (more on this later)
2 notions of RSS and RSNI (thus 2 targets):

- **strong(+):** no speculative leaks
• 2 notions of RSS and RSNI (thus 2 targets):
  • strong(+): no speculative leaks
  • weak(-): allows speculative leaks of data retrieved non-speculatively
Security Spectrum

• 2 notions of RSS and RSNI (thus 2 targets):
  • strong(+) : no speculative leaks
  • weak(-) : allows speculative leaks of data retrieved non-speculatively

```c
void get (int y)
    if (y < size) then
        temp = B[A[y]*512]
```

Violates + and -

```c
void get (int y)
    x = A[y];
    if (y < size) then
        temp = B[x];
```

Violates +, Satisfies -
void f(int x) \rightarrow \text{if}(x < A.size) \{y = B[A[x]]\} \quad \text{// A.size=16, A[128]=3}

\lceil \cdot \rceil = \text{void f(int x)} \rightarrow \text{if}(x < A.size)\{\text{lfence}; y = B[A[x]]\}
void f(int x) \mapsto \text{if}(x < \text{A.size}) \{ y = B[A[x]] \} \quad \text{// A.size}=16, A[128]=3

\text{[\cdot]} = \text{void f(int x) \mapsto if}(x < \text{A.size}) \{ \text{lfence}; y = B[A[x]] \}
void f(int x) \rightarrow \text{if}(x < A.\text{size}) \{ y = B[A[x]] \}  \\
\text{// A.size}=16, A[128]=3

\text{\[\cdot\]= void f(int x) \rightarrow \text{if}(x < A.\text{size})\{lfence; y = B[A[x]]\}\]}

call f 128
\text{pc : S}

\text{pc : S}

lfence; y = B[ A[ 128 ] ]
\text{pc : U}
void f(int x) ↔ if(x < A.size) { y = B[A[x]] }  // A.size=16, A[128]=3

[.] = void f(int x) ↔ if(x < A.size){ lfence; y = B[A[x]] }

call f 128

if (128 < 16) { lfence; y = B[A[128]] }

pc : S
void f(int x) $\leftrightarrow$ if ($x < A\.size$) $\{ y = B[A[x]] \}$  
$\text{[]} = \text{void f}(\text{int } x) \leftrightarrow \text{if}(x < A\.size)\{ \text{lfence}; y = B[A[x]] \}$

- call f 128
- pc : S
- pc : S
- skip
- pc : S
 void f(int x) ↔ if(x < A.size){y = B[A[x]]}  // A.size=16, A[128]=3
[] = void f(int x) ↔ if(x < A.size){y = B[mask(A[x])]}
void f(int x) { if (x < A.size) { y = B[A[x]] } }  // A.size=16, A[128]=3

\[ \] = void f(int x) { if (x < A.size) { y = B[\text{mask}(A[x])] } }
void f(int x) ↔ if (x < A.size) { y = B[A[x]] }  // A.size=16, A[128]=3
[::] = void f(int x) ↔ if (x < A.size) { y = B[mask(A[x])] }

call f 128
pc : S

if (128 < 16) { y = B[ mask(A[ 128 ])] }  pc : S

SLH preserves RSS- (and thus RSNI-) but not RSS/+ (and thus not RSNI/+)
Framework benefits: fine-grained analyses of countermeasures security
void f(int x) -> if(x < A.size) { y = B[A[x]] } // A.size=16, A[128]=3

[.] = void f(int x) -> if(x < A.size) { y = B[mask(A[x])]}
RSSC for SLH

```c
void f(int x) {
    if (x < A.size) { y = B[A[x]] }
} // A.size=16, A[128]=3
```

```
void f(int x) {
    if (x < A.size) { y = B[mask(A[x])] }
}
```

```
call f 128
pc : S
```

```
if (128 < 16) { y = B[mask(A[128])] }
```

```
z = cmv 128 \neq 16 ? 0 : A[128]
y = B[z]
```

```
z = cmv 128 \neq 16 ? 0 : A[128]
y = B[z]
```

```
128 : S
```

```
pc : U
```

SLH preserves RSS- (and thus RSNI-) but not RSS/plus (and thus not RSNI/plus)
```

Framework benefits: fine-grained analysis of countermeasures security
void f(int x) ↔ if (x < A.size) { y = B[A[x]] } // A.size=16, A[128]=3

[·] = void f(int x) ↔ if (x < A.size) { y = B[mask(A[x])] }

SLH preserves RSS- (and thus RSNI-) but not RSS/plus.osf (and thus not RSNI/plus.osf)

Framework benefits: fine-grained analysis of countermeasures security
SLH preserves RSS- (and thus RSNI-) but not RSS/plus (and thus not RSNI/plus).

Framework benefits: fine-grained analysis of countermeasures security.
void f(int x) ↔ if(x < A.size){y = B[A[x]]}  

[·] = void f(int x) ↔ if(x < A.size){y = B[mask(A[x])]}

RSSC for SLH

SLH preserves RSS- (and thus RSNI-)
but not RSS/+ (and thus not RSNI/+)
Framework benefits: fine-grained
analysis of countermeasures security
SLH preserves RSS- (and thus RSNI-) but **not** RSS+ (and thus not RSNI+)

Framework benefits: **fine-grained analysis** of countermeasures security
Insecurity Results

- MSVC is Insecure
- Non-interprocedural SLH is insecure

Both omit speculation barriers
Proofs Insight

\begin{align*}
\langle A \rangle / A &\quad \text{executes} \\
P / [P] &\quad \text{executes} \\
\langle A \rangle / A &\quad \text{executes}
\end{align*}

either $A$ or $[P]$ executes

either $A$ or $[P]$ executes
Beyond V1 Protection

- $RSSP$ with V1 trace model = $RSSP_1$
Beyond V1 Protection

- \(RSSP\) with V1 trace model = \(RSSP_1\)
- \(wh: \succeq [\cdot]_T^S: RSSP_1\) (produces V1-secure code)
Beyond V1 Protection

- \( RSSP \) with \( V1 \) trace model = \( RSSP_1 \)
- \( \text{wh: } \vdash [\cdot]^S_T : RSSP_1 \) (produces \( V1 \)-secure code)
- take \( [\cdot]^T_T \) that produces \( V4 \)-secure code
Beyond V1 Protection

- \textit{RSSP} with \textit{V1 trace model} = \textit{RSSP}_1
- wh: \( \left[ \cdot \right]^S_T : \textit{RSSP}_1 \) (produces V1-secure code)
- take \( \left[ \cdot \right]^T_T \) that produces V4-secure code
- if \( \left[ \cdot \right]^S_T : \textit{RSSP}_1 \)
- and \( \left[ \cdot \right]^T_T : \textit{RSSP}_4 \)
- what do we know about \( \left[ \left[ \cdot \right]^S_T \right]^T_T \)?
• “Unknown” (but expected(?)):

\[
\text{if } \vdash [\cdot]_S^L : X \quad (RSSP_1) \\
\text{and } \vdash [\cdot]_T^I : Y \quad (RSSP_4) \\
\text{then } \vdash [\cdot]_T^I : X \cap Y
\]
“Unknown” (but expected(?)):

\[
\begin{align*}
\text{if } & (\cdot)_{\mathcal{S}}^i : X \\
\text{and } & (\cdot)_{\mathcal{T}}^i : Y \\
\text{then } & (\cdot)_{\mathcal{T}}^{i}^{\mathcal{S}} : X \cap Y
\end{align*}
\]

problem:

\[RSSP_1 \cap RSSP_4 \neq RSSP_1 \cup RSSP_4\]
Proposed Solution

Instrumentations:

- **preserve** some [class of] (hyper)property \( X \)
- **enforce** a specific (hyper)property \( Y \)

\[ \vdash [\cdot] \succcurlyeq_x Y \]
Proposed Solution

Instrumentations:

- **preserve** some [class of] (hyper)property $X$
- **enforce** a specific (hyper)property $Y$

$\vdash [\cdot] >_X Y$
Proposed Solution (wip)

Instrumentations:

- preserve some \( [\text{class of}] \) (hyper)property \( X \)
- enforce a specific (hyper)property \( Y \)

\[
\text{cannot enforce classes}
\]

\[\vdash [\cdot] >_X Y\]
if $\begin{bmatrix} \cdot \end{bmatrix}^S : RSSP_1$

and $\begin{bmatrix} \cdot \end{bmatrix}^T >_{RSSP_1} RSSP_4$

then $\begin{bmatrix} \begin{bmatrix} \cdot \end{bmatrix}^S \end{bmatrix}^T : RSSP_1 \cup RSSP_4$
More Generally

• some optimisation passes may not preserve some property $X$ (specific, not class)
More Generally

• some optimisation passes may not preserve some property $X$ (specific, not class)
• we need later passes to enforce $X$
• some optimisation passes may not preserve some property $X$ (specific, not class)
• we need later passes to enforce $X$
• interesting (unknown(?)) metatheory, very interesting application
Questions?