Robustly Safe Compilation

10th April 2019

1. Stanford University
2. CISPA
3. MAX PLANCK INSTITUTE FOR SOFTWARE SYSTEMS
What is Secure Compilation?

Rust

Asm

P

[P_1]

[P_2]

... 

[P_n]

P′
What is Secure Compilation?

Rust

Asm

y = &mut

\[ P_1 \]

\[ P_2 \]

\[ \ldots \]

\[ P_n \]

P

\[ [P_1] \]

\[ [P_2] \]

\[ \ldots \]

\[ [P_n] \]

P'
What is Secure Compilation?

Rust

Asm

\( y = \&\text{mut} P_1 \)

\( P_2 \)

\( \ldots \)

\( P_n \)

used linearly

\( \text{P} \)

\([P_1]\)

\([P_2]\)

\( \ldots \)

\([P_n]\)

\( P' \)
What is Secure Compilation?

Rust

Asm

\[ y = \& \text{mut} \]

\[ P_1 \]

\[ P_2 \]

\[ \ldots \]

\[ P_n \]

used linearly

\[ [y = \& \text{mut}] \]

\[ [P_1] \]

\[ [P_2] \]

\[ \ldots \]

\[ [P_n] \]

\[ P' \]
What is Secure Compilation?

`y = &mut P_1`

Rust

Asm

`P_1`  `P_2`  ...  `P_n`

violate linearity
What is Secure Compilation?

Preserve the security properties of

\[ y = \&\text{mut} \]

\[ P_1 \]

\[ P_2 \]

\[ \ldots \]

\[ P_n \]

Rust

Asm

\[ [y = \&\text{mut}] \]

\[ [P_1] \]

\[ [P_2] \]

\[ \ldots \]

\[ [P_n] \]

\[ P' \]
What is Secure Compilation?

Preserve the security properties of

\[
y = \&\text{mut } P_1 \rightarrow P_2 \rightarrow \ldots \rightarrow P_n
\]

Rust

Asm

when interoperating with
What is Secure Compilation?

Preserve the security properties of $P_1, P_2, \ldots, P_n$ when interoperating with PL sec (e.g., no side channels) when interoperating with Rust and Asm.

$y = \&\text{mut } P_1$

Rust

Asm

$P$

$[P_1]$

$[P_2]$

$[P_n]$

$P'$
What is Secure Compilation?

Correct compilation

\[
y = \&\text{mut } P_1 \\
P_2 \\
\ldots \\
P_n
\]

Rust

\[
[y = \&\text{mut}]

\begin{bmatrix}
P_1 \\
P_2 \\
\ldots \\
P_n
\end{bmatrix}

P'

Asm
What is Secure Compilation?

Correct compilation

Correct compilation respects linearity

Rust

Asm

Respect linearity
What is Secure Compilation?

**Secure compilation**

\[ y = \&\text{mut} \ P_1 \]
\[ P_2 \]
\[ \ldots \]
\[ P_n \]

---

**Rust**

\[ [y = \&\text{mut}] \]
\[ [P_1] \]
\[ [P_2] \]
\[ \ldots \]
\[ [P_n] \]

**Asm**

\[ P \]
\[ [P_1] \]
\[ [P_2] \]
\[ \ldots \]
\[ [P_n] \]

\[ P' \]
What is Secure Compilation?

Enable source-level security reasoning

Rust

Asm

y = &mut P_1

P_2

\ldots

P_n

y = &mut [P_1]

[P_2]

\ldots

[P_n]

P'

\text{Enable source-level security reasoning}
Do Secure Compilers Exist?

Yes!

They rely on security mechanisms:

• enclaves
• capabilities
• types
• tagged memory
• ASLR
• CFI, SFI
• processes
• ...
Do Secure Compilers Exist?

Yes!

- enclaves
- capabilities
- types
- tagged memory
- ASLR
- CFI, SFI
- processes
- ...
Do Secure Compilers Exist?

Yes!

They rely on **security mechanisms:**

- enclaves
- capabilities
- types
- tagged memory
- ASLR
- CFI, SFI
- processes
- ...
Some secure compilers:

- P1: lack formal proof of their security guarantees
But...

Some secure compilers:

- **P1**: lack formal proof of their security guarantees
- **P2**: prove preservation of ad-hoc security properties
But...

Some secure compilers:

• P1: lack formal proof of their security guarantees
• P2: prove preservation of ad-hoc security properties
• P3: inefficient
Some secure compilers:

- P1: lack formal proof of their security guarantees
- P2: prove preservation of *ad-hoc* security properties
- P3: inefficient

complex proofs

dictated by existing definitions

unclear how to generalise
Goal:

Define a formal criterion for secure compilation:
Goal:

Define a **formal criterion** for secure compilation:

- attainable
- **efficient** (wrt existing ones)
- easy not too hard to prove
Goal:

Define a formal criterion for secure compilation:

- attainable
- efficient (wrt existing ones)
- easy not too hard to prove
- with clear security guarantees
Contributions

• $\text{RSC}$: known criterion, meets our goals

• Three compilers $J \cdot K$ that attain $\text{RSC}$

• Relying on memory isolation (via capabilities or enclaves)

• No runtime checks!

• Two proof techniques for $\text{RSC}$

• Simplifications on existing ones

• A comparison between $\text{RSC}$ and $\text{FAC}$
Contributions

- $RSC$: known criterion, meets our goals
  - a compiler preserves all safety properties
Contributions

- $RSC$: known criterion, meets our goals
  - a compiler preserves all safety properties
- three compilers $[\cdot\cdot]$ that attain $RSC$
Contributions

- **$RSC$:** known criterion, meets our goals
  - a compiler preserves all safety properties
- three compilers $\ldots$ that attain $RSC$
  - relying on memory isolation (via capabilities or enclaves)
Contributions

• \( \textit{RSC} \): known criterion, meets our goals
  • a compiler preserves all safety properties
• three compilers \([\cdot\cdot]\) that attain \( \textit{RSC} \)
  • relying on memory isolation (via capabilities or enclaves)
  no runtime checks!
Contributions

• \textit{RSC}: known criterion, meets our goals
  • a compiler preserves all \textit{safety} properties
• three compilers \([\cdot\cdot]\) that attain \textit{RSC}
  • relying on \textit{memory isolation} (via capabilities or enclaves)
    no runtime checks!
• two \textit{proof techniques} for \textit{RSC}
Contributions

- \( RSC \): known criterion, meets our goals
  - a compiler preserves all safety properties
- three compilers \([\cdot]\) that attain \( RSC \)
  - relying on memory isolation (via capabilities or enclaves)
    no runtime checks!
- two proof techniques for \( RSC \)
  - simplifications on existing ones
Contributions

• \( RSC \): known criterion, meets our goals
  • a compiler preserves all safety properties
• three compilers \([\cdot]\) that attain \( RSC \)
  • relying on memory isolation (via capabilities or enclaves)
    no runtime checks!
• two proof techniques for \( RSC \)
  • simplifications on existing ones
• a comparison between \( RSC \) and \( FAC \)
Contributions

- **RSC**: known criterion, meets our goals
  - a compiler preserves all safety properties
- three compilers \([\cdot]\) that attain \(RSC\)
  - relying on memory isolation (via capabilities or enclaves)
    no runtime checks!
- two proof techniques for \(RSC\)
  - simplifications on existing ones
- a comparison between \(RSC\) and \(FAC\)
Contributions

- **RSC**: known criterion, meets our goals
  - a compiler preserves all safety properties
- three compilers $[\cdot\cdot\cdot]$ that attain $RSC$
  - relying on memory isolation (via capabilities or enclaves)
  - no runtime checks!

- two **proof techniques** for $RSC$
  - simplifications on existing ones
- a comparison between $RSC$ and $FAC$
Talk Roadmap

Robust Safety

Robustly Safe Compilation

Backtranslation Proof Technique
Robust Safety

- Robustness
- Behaviour
- Safety
Robustness

Our code

newBlk(c)
adBlk(b)
verifyCh()
Robustness

Our code

newBlk(c)
addBlk(b)
verifyCh()

Our imports

lib₁
abs(c)

lib₂
printf(s)

lib₃
main(args)
Robustness

Our code

newBlk(c)
addBlk(b)
verifyCh()

Malicious code (arbitrary)

A

lib1
abs(c)

lib2
printf(s)

lib3
main(args)

Our imports
Robustness

Our code

newBlk(c)
addBlk(b)
verifyCh()

Malicious code (arbitrary)

∀A. A [P]

Our imports

lib₁
abs(c)

lib₂
printf(s)

lib₃
main(args)

code-attacker
interoperation
formally:
Program Behaviour

Our code

newBlk(c)
addBlk(b)
verifyCh()

Malicious code (arbitrary)

lib₁  A  lib₂  lib₃

/nine.osf//two.osf/one.osf
Program Behaviour

Our code

- newBlk(c)
- addBlk(b)
- verifyCh()

verifyCh()? + ret true!

Malicious code (arbitrary)

/lib1 /lib2 /lib3

Malicious code (arbitrary)

verifyCh()?

ret true!
Program Behaviour

Our code

newBlk(c)
addBlk(b)
verifyCh()

verifyCh()?

Malicious code (arbitrary)

lib_1
A
lib_2
lib_3

ret true!

Observable actions $\alpha_?$, $\alpha_!$
Program Behaviour

Our code

newBlk(c)
addBlk(b)
verifyCh()

verifyCh()?

ret true!

Observable actions $\alpha\,?,\,\alpha\!$

Malicious code (arbitrary)

Code behaviour = sequence of actions

$\overline{\alpha} \overset{\text{def}}{=} \alpha_1\,?,\,\alpha_2\!,\,...$
Program Behaviour

Our code

newBlk(c)
addBlk(b)
verifyCh()

Malicious code (arbitrary)

lib₁ A lib₂ lib₃

verifyCh()? ret true!

Code behaviour = sequence of actions

\[ \overline{\alpha} \overset{\text{def}}{=} \alpha₁?, \alpha₂!, \ldots \]
Program Behaviour

Our code

\(\text{newBlk}(c)\) \(\text{addBlk}(b)\) \(\text{verifyCh()}\)

Malicious code (arbitrary)

\(\text{verifyCh()}? \text{ret true!} \quad \text{addBlk}(b)? \text{ret ok!} \quad \text{verifyCh()}? \text{ret true!}

\(\alpha \overset{\text{def}}{=} \alpha_1?, \alpha_2!, \ldots\)

Code behaviour = sequence of actions
Program Behaviour

Our code:
- newBlk(c)
- addBlk(b)
- verifyCh()

Malicious code (arbitrary):
- verifyCh()? ret true!
- addBlk(b)? ret ok!
- verifyCh()? ret true!

Code behaviour formally:
\[
A[P] \xrightarrow{\overline{\alpha}} -
\]

Code behaviour = sequence of actions
\[
\overline{\alpha} \overset{\text{def}}{=} \alpha_1 ?, \alpha_2 !, \ldots
\]
Safety Properties

no bad thing happens (finitely)

newBlk(c)
addBlk(b)
verifyCh()

verifyCh()? ret true!
addBlk(b)? ret ok!
verifyCh()? ret true!

lib₁ + A + lib₂ + lib₃

No: addBlk(b)? ret ok!
verifyCh()? ret false!

/one.osf/zero.osf/two.osf/one.osf
Safety Properties

no bad thing happens (finitely)

newBlk(c)
addBlk(b)
verifyCh()

verifyCh()? ret true!
addBlk(b)? ret ok!
verifyCh()? ret true!

lib1  A  lib2  lib3

safety = integrity, functional correctness, weak secrecy, ...

/eone.osf/zero.osf/two.osf/one.osf
Safety Properties

no bad thing happens (finitely)

verifyCh()? ret true!
addBlk(b)? ret ok!
verifyCh()? ret true!

newBlk(c)
addBlk(b)
verifyCh()

e.g.,
chain is always valid

NO: addBlk(b)? ret ok!
verifyCh()? ret false!

Safety - integrity, functional correctness, weak secrecy, ...
Safety Properties

- A newBlk(c)
- addBlk(b)
- verifyCh()

no bad thing happens (finitely)

- verifyCh() ? ret true
- addBlk(b) ? ret ok
- verifyCh() ? ret true

M = safety property encoding

e.g.,
- chain is always valid

NO: addBlk(b) ? ret ok!
- verifyCh() ? ret false!

Safety + integrity, functional correctness, weak secrecy, ...
• for a safety property
Robust Safety

• for a safety property
• no matter what we link against
Robust Safety

• for a safety property
• no matter what we link against
• our program behaves in a way
Robust Safety

- for a safety property
- no matter what we link against
- our program behaves in a way
- that respects that safety property
Robust Safety

• for a safety property ($M$)
• no matter what we link against ($\forall A, \overline{\alpha}$)
• our program behaves in a way (if $A[P] \xrightarrow{\alpha}$)
• that respects that safety property (then $M \vdash \overline{\alpha}$)
Robust Safety

- for a safety property \( (M) \)
- no matter what we link against \( (\forall A, \alpha) \)
- our program behaves in a way (if \( A[P] \xrightarrow{\alpha} \))
- that respects that safety property (then \( M \vdash \alpha \))

robust safety formally \( M \vdash P \)
Robustly Safe Compilation
Robust Safety Across Compilation

Our code + Malicious code (arbitrary)

newBlk(c) addBlk(b) verifyCh()

lib₁ A lib₂ lib₃
Robust Safety Across Compilation

Our code

\[ \text{newBlk}(c) \]
\[ \text{addBlk}(b) \]
\[ \text{verifyCh}() \]

+ 

Malicious code (arbitrary)

\[ \text{lib}_1 \quad \text{A} \quad \text{lib}_2 \quad \text{lib}_3 \]

Compiled code

\[ \begin{aligned}
&\text{newBlk}(c) \\
&\text{addBlk}(b) \\
&\text{verifyCh}() 
\end{aligned} \]

+ 

Target Attacker!

\[ \begin{aligned}
&\text{code}_1 \\
&\text{breakCh}() \\
&\text{code}_2 \\
&\text{printf}(s) \\
&\text{code}_3 \\
&\text{forgeBlk}() 
\end{aligned} \]
Robust Safety Across Compilation

Our code:
- `newBlk(c)`
- `addBlk(b)`
- `verifyCh()`

Malicious code (arbitrary):
- `newBlk(c)`
- `addBlk(b)`
- `verifyCh()`

Compiled code:
- `code_1`
- `A`
- `code_2`
- `code_3`

Target Attacker!

Assume safety (push verification here)

verifyCh()? ret true!
Robust Safety Across Compilation

Our code

\[
\begin{align*}
\text{newBlk}(c) \\
\text{addBlk}(b) \\
\text{verifyCh}() 
\end{align*}
\]

Malicious code (arbitrary)

\[
\begin{align*}
\text{assume safety} \\
(\text{push verification here}) \\
\text{verifyCh}() \text{? ret true!} \\
\text{verifyCh}() \text{? ret 0!} \\
\text{target Attacker!}
\end{align*}
\]

Compiled code

\[
\begin{align*}
\text{code}_1 \quad A \\
\text{code}_2 \\
\text{code}_3
\end{align*}
\]
Robust Safety Across Compilation

Our code

\(\text{newBlk}(c)\)
\(\text{addBlk}(b)\)
\(\text{verifyCh}()\)

Malicious code (arbitrary)

\(\text{newBlk}(c)\)
\(\text{addBlk}(b)\)
\(\text{verifyCh}()\)

Verbally:

- Our code
  - \(\text{newBlk}(c)\)
  - \(\text{addBlk}(b)\)
  - \(\text{verifyCh}()\)

- Malicious code
  - \(\text{newBlk}(c)\)
  - \(\text{addBlk}(b)\)
  - \(\text{verifyCh}()\)

- Compiled code

- Target Attacker!
  - no direct access to chain

\[
\begin{align*}
\text{verifyCh}() & \text{? ret true!} \\
\text{verifyCh}() & \text{? ret 0!} \\
\end{align*}
\]
Robust Safety Across Compilation

Our code

newBlk(c)
addBlk(b)
verifyCh()

Malicious code (arbitrary)

lib1
A
lib2
lib3

Compiled code

newBlk(c)
addBlk(b)
verifyCh()

compiled code

Target Attacker!

verifyCh()? ret true!

verifyCh()? ret 0!

no block forging
Robust Safety Across Compilation

Our code:
- `newBlk(c)`
- `addBlk(b)`
- `verifyCh()`

verifyCh()? ret true!

Malicious code (arbitrary):
- lib1
- lib2
- lib3

RSC formally:
given $M \approx M$
if $M \vdash P$ then $M \vdash [P]$

Compiled code:
- [newBlk(c)]
- [addBlk(b)]
- [verifyCh()]

Target Attacker!

verifyCh()? ret 0!
Concerns

\(RSC\) so far:

- attainable
- efficient
Concerns

\( \text{RSC} \) so far:

- attainable
- efficient
- possibly tricky to prove
Concerns

\( RSC \) so far:

- attainable
- efficient
- possibly tricky to prove

\( PF-RSC \): equivalent definition
Concerns

$RSC$ so far:

- attainable
- efficient
- possibly tricky to prove

$PF-RSC$: equivalent definition easier to prove than $RSC$
Concerns

$RSC$ so far:

- attainable
- efficient
- possibly tricky to prove

$PF-RSC$: equivalent definition easier to prove than $RSC$

(equivalence to be proven, generally true)
Backtranslation Proof Technique
Backtranslation: Build $A$ From $A$ or $\overline{a}$

HP: $P$ is RS

newBlk(c)  
addBlk(b)  
verifyCh()

+  

lib$_1$  
lib$_2$  
lib$_3$

[  
newBlk(c)  
addBlk(b)  
verifyCh()  
]

+  

code$_1$  
code$_2$  
code$_3$

Compiled code
Backtranslation: Build $\mathbf{A}$ From $\mathbf{A}$ or $\overline{\alpha}$

HP: $P$ is RS

newBlk(c) 
addBlk(b) 
verifyCh()

+ 

\begin{align*}
\text{lib}_1 & \\
\text{lib}_2 & \\
\text{lib}_3 & 
\end{align*}

\begin{align*}
\text{newBlk(c)} & \\
\text{addBlk(b)} & \\
\text{verifyCh()} & 
\end{align*}

+ 

\begin{align*}
\forall & \\
\text{code}_{4} & \\
\text{code}_{2} & \\
\text{code}_{3} & 
\end{align*}

\begin{align*}
\text{verifyCh()} & \text{? ret 0!} \\
\text{addBlk(b)} & \text{? ret ok!} \\
\text{verifyCh()} & \text{? ret 0!} 
\end{align*}

Compiled code

For Any Attacker!
Backtranslation: Build A From A or $\overline{A}$

**HP: P is RS**
- `newBlk(c)`
- `addBlk(b)`
- `verifyCh()`

**Exists Attacker!**
- `lib_1`
- `lib_2`
- `lib_3`

**For Any Attacker!**
- `code_1`
- `code_2`
- `code_3`

For Any Attacker:
- `verifyCh()`? ret true!
- `addBlk(b)`? ret ok!
- `verifyCh()`? ret true!

Compiled code:
- `newBlk(c)`
- `addBlk(b)`
- `verifyCh()`

- `verifyCh()`? ret 0!
- `addBlk(b)`? ret ok!
- `verifyCh()`? ret 0!
Backtranslation: Build $A$ From $A$ or $\overline{A}$

**HP:** $P$ is RS

$$\text{newBlk}(c) \quad \text{addBlk}(b) \quad \text{verifyCh}()$$

**Exists Attacker!**

$$\exists A \quad \text{lib}_1 \quad \text{lib}_2 \quad \text{lib}_3$$

```
HP: P is RS
newBlk(c) addBlk(b) verifyCh()
```

```
exists Attacker!
```

```
 "same" trace
```

```
Compiled code
```

```
For Any Attacker!
```

```
∀ A
```

```
ret 0
```

```
ret ok
```

```
ret true
```

```
ret 0
```

```
ret ok
```

```
ret true
```

```
exists Attacker!
```

```
"same" trace
```

```
forall A
```

```
ret 0
```

```
ret ok
```

```
ret true
```

```
exists Attacker!
```

```
forall A
```

```
ret 0
```

```
ret ok
```

```
ret true
```
**Backtranslation: Build $A$ From $A$ or $\overline{\alpha}$**

- **HP:** $P$ is RS
  - `newBlk(c)`
  - `addBlk(b)`
  - `verifyCh()`

- **Exists Attacker!**
  - `verifyCh()`? ret true!
  - `addBlk(b)`? ret ok!
  - `verifyCh()`? ret true!

- **“same” trace**
  - $\overline{\alpha}$ must be safe by HP (all $\overline{\alpha}$ are safe)

- **Compiled code**

- **For Any Attacker!**

- `lib1` $\exists A$ `lib2` `lib3`

- Compiled code

- `∀` For Any

- `∃` Exists
Safety as a Dual and $PF-RSC$

- Safety = nothing bad happens
Safety as a Dual and \( PF-RSC \)

- Safety = nothing bad happens
  so if it were to happen it would finitely
Safety as a Dual and $PF-RSC$

- Safety = nothing bad happens so if it were to happen it would finitely
- given any behaviour ($A [[P]] \xrightarrow{\alpha}$)
Safety as a Dual and $PF-RSC$

- Safety = nothing bad happens
  so if it were to happen it would finitely

- given any behaviour $(A [[P]] \xrightarrow{\alpha})$

- if we can replicate that $(\exists A. A [P] \xrightarrow{\alpha})$
Safety as a Dual and $PF-RSC$

- Safety = nothing bad happens so if it were to happen it would finitely
  - given any behaviour $(A [[P]] \xrightarrow{\alpha})$
  - if we can replicate that $(\exists A.A[P] \xrightarrow{\alpha})$
  - then $\alpha$ is not bad
Safety as a Dual and \( PF - RSC \)

- Safety = nothing bad happens so if it were to happen it would finitely
- given any behaviour \( (A [[P]] \xrightarrow{\alpha} ) \)
- if we can replicate that \( (\exists A. A [P] \xrightarrow{\alpha} ) \)
- then \( \overline{\alpha} \) is not bad because \( \overline{\alpha} \) does not violate safety (by RS of P) (for \( \overline{\alpha} \approx \overline{\alpha} \))
Safety as a Dual and \( PF-RSC \)

- Safety = nothing bad happens so if it were to happen it would finitely
- given any behaviour \( \alpha \)
- if we can replicate \( \alpha \)
- then \( \overline{\alpha} \) is not bad because \( \overline{\alpha} \) does not violate safety (by RS of \( P \)) (for \( \overline{\alpha} \approx \overline{\alpha} \))

\[ PF-RSC \] formally:

if \( \forall A. A [P] \xrightarrow{\overline{\alpha}} \) then \( \exists A. A [P] \xrightarrow{\overline{\alpha}} \) and \( \overline{\alpha} \approx \overline{\alpha} \)
**RSC** and **PF-RSC**

**RSC**: given $M \approx \text{M}$
if $M \vdash P$ then $M \vdash [P]$

**PF-RSC**: if $\forall A. A [\,[P]\,]\xrightarrow{\alpha}$
then $\exists A. A [P] \xrightarrow{\overline{\alpha}}$ and $\overline{\alpha} \approx \overline{\alpha}$

\[•\text{ must be proven (when needed)}\]
\[•\text{ proof is (generally) trivial}\]
\[•\text{ sanity-check for cross-language safety}\]
\[\text{encoding (}M \approx \text{M)}\]
**RSC and PF-RSC**

**RSC**: given $M \approx M$

if $M \vdash P$ then $M \vdash [P]$

**PF-RSC**: if $\forall A.A [[P]] \xrightarrow{\alpha}$

then $\exists A.A[P] \xrightarrow{\bar{\alpha}}$ and $\bar{\alpha} \approx \bar{\alpha}$

- $\iff$ must be proven (when needed)
RSC and PF-RSC

**RSC**: given $M \approx M$
if $M \vdash P$ then $M \vdash \llbracket P \rrbracket$

**PF-RSC**: if $\forall A. A \llbracket P \rrbracket \xrightarrow{\alpha}$
then $\exists A. A \llbracket P \rrbracket \xrightarrow{\bar{\alpha}}$ and $\bar{\alpha} \approx \bar{\alpha}$

- $\iff$ must be proven (when needed)
- proof is (generally) trivial
**RSC and PF-RSC**

**RSC**: given $M \approx M$
if $M \vdash P$ then $M \vdash [P]$

**PF-RSC**: if $\forall A. A [[P]] \overset{\alpha}{\rightarrow}$
then $\exists A. A [P] \overset{\alpha}{\rightarrow}$ and $\overline{\alpha} \approx \overline{\alpha}$

- $\iff$ **must** be proven (when needed)
- proof is (generally) **trivial**
- **sanity-check** for cross-language safety encoding ($M \approx M$)
Take Home Message

What to make of this result?
What to make of this result?

• **encode** safety properties in the systems
Take Home Message

What to make of this result?

- **encode** safety properties in the systems
- ensure the desired property **follows** from the encoding
Take Home Message

What to make of this result?

- **encode** safety properties in the systems
- ensure the desired property **follows** from the encoding
- **use our** proof techniques to prove safety is preserved
The paper (or the techreport) contains more:

- one $RSC \llbracket \cdot \rrbracket_{LP}^{LU}$ from untyped while to capabilities
- one $RSC \llbracket \cdot \rrbracket_{L^\tau\pi}^{LT}$ from typed, concurrent while to capabilities
- one $RSC \llbracket \cdot \rrbracket_{LI}^{LT}$ from typed, concurrent while to enclaves
- a backtranslation-based $RSC$ proof (for $\llbracket \cdot \rrbracket_{LP}^{LU}$)
- two simulation-based $RSC$ proofs (for $\llbracket \cdot \rrbracket_{L^\pi}^{LT}$ and $\llbracket \cdot \rrbracket_{LI}^{LT}$)
- a $FAC \llbracket \cdot \rrbracket_{LP}^{LU}$ from untyped while to capabilities
- a backtranslation-based $FAC$ proof sketch (for $\llbracket \cdot \rrbracket_{LP}^{LU}$)
- a comparison of efficiency and proof complexity between $\llbracket \cdot \rrbracket_{LP}^{LU}$ and $\llbracket \cdot \rrbracket_{LP}^{LU}$
Questions?
Backtranslation Example

main(z) ⇔
- let x = new 4 in L :: ⟨x, 1⟩;
- let x = new 3 in L :: ⟨x, 2⟩;
- call f 0;
- let x =!L(2) in L :: ⟨x, 3⟩;
- let x = new L(1) in x := 55;
- let x = new L(3) in x := 15;
- call f 2;

(1) call f 0 (1 ⇔ 4 : ⊥, 2 ⇔ 3 : ⊥)?
(2) ret (1 ⇔ 4 : ⊥, 2 ⇔ ⟨3, k⟩ : ⊥, 3 ⇔ 11 : k)!
(3) call f 2 (1 ⇔ 55 : ⊥, 2 ⇔ ⟨3, k⟩ : ⊥, 3 ⇔ 15 : k)?
Simulation-Based Proof

Set up cross-language relation $\approx_\beta$ that:

- knows trusted locations: $\tau \not\models o$.
- splits heaps (source and target) into trusted and untrusted;
- constitutes trusted heap by trusted locations ($\tau \not\models o$);
- relates trusted heap to trusted heap
- protects every trusted location by a capability;
- capability protecting a trusted location is not in attacker code, nor in the untrusted heap