CSC Report – Foundations of Secure Compilation

Marco Patrignani\textsuperscript{1,2}

23\textsuperscript{rd} June 2021
Talk Outline

My Stanford Experience

Foundations of Secure Compilation

Future Outlook
My Stanford Experience
5. Lucile Packard Children's Hospital Stanford

Stanford Hospital at 300 Pasteur Drive

STOCK FARM GARAGE

L-22

Arrillaga Outdoor Education and...
terrific experience

- mentoring (John & lunches)
terrific experience

- mentoring (John & lunches)
- teaching (courses & lunches)
terrific experience

- mentoring (John & lunches)
- teaching (courses & lunches)
- research (+ talks)
terrific experience

- mentoring (John & lunches)
- teaching (courses & lunches)
- research (+ talks)
- new perspective
terrific experience

- mentoring (John & lunches)
- teaching (courses & lunches)
- research (+ talks)
- new perspective
- skiing (who’d have thought?)
1. Motivation behind SC
2. History of SC
3. Our contributions to the foundations of SC
4. Current and future applications
Special Thanks to:

(wrt the contents of this talk)

Carmine Abate  Amal Ahmed  Roberto Blanco  Stefan Ciobaca  Dave Clarke  Dominique Devriese

Akram El-Korashy  Deepak Garg  Marco Guarnieri  Catalin Hritcu  Robert Künnemann  Frank Piessens

Eric Tanter  Jeremy Thibault  Stelios Tsampas  Marco Vassena  Riad Wahby
Special Thanks to:

(wrt the contents of this talk)

Carmine Abate Amal Ahmed Roberto Blanco Stefan Ciobaca Dave Clarke Dominique Devriese Akram El-Korashy Deepak Garg Marco Guarnieri Catalin Hritcu Robert Künnemann Frank Piessens

please interrupt & ask questions
Good PLs (R, TypeScript, Go, ... ) provide:

• helpful abstractions to write secure code
Programming Languages: Pros and Problems

Good PLs (R, TypeScript, ..., ...) provide:

• helpful abstractions to write secure code

but

• when compiled (J⋅K) and linked with adversarial target code
Good PLs (R, TypeScript, , , …) provide:

- helpful abstractions to write secure code

but

- when compiled (lenmiş) and linked with adversarial target code
- these abstractions are NOT enforced
Secure Compilation: Example

F* HACL*: ... CCS’17

Asm

[ChaCha20] [Poly1305] […]
Secure Compilation: Example

ChaCha20, Poly1305, ...

HACL*: ... CCS’17

Asm

[ChaCha20], [Poly1305], [...]

160x C/C++ code (unsafe)
Secure Compilation: Example

Preserve the security of

ChaCha20 → Poly1305 → ...

Asm

[ChaCha20] → [Poly1305] → [...]
Secure Compilation: Example

Preserve the security of

ChaCha20 → Poly1305 → ...

\( F^* \) \( \text{HACL}^*: \ldots \text{CCS}'17 \)

Asm

[ChaCha20] → [Poly1305] → [...]

when interoperating with
Secure Compilation: Example

**Correct compilation**

F*  
HACL*: … CCS’17

Asm

[ChaCha20]  [Poly1305]  […]
Secure Compilation: Example

Secure compilation

ChaCha20  Poly1305  ...

F*  HACL*: ... CCS’17

Asm

[ChaCha20]  [Poly1305]  [...]

F*  HACL*: ... CCS’17

Asm
Secure Compilation: Example

Enable source-level security reasoning

F*\text{HACL*; … CCS’17}

Asm

\begin{align*}
\text{ChaCha20} & \quad \text{Poly1305} & \quad \ldots \\
\text{[ChaCha20]} & \quad \text{[Poly1305]} & \quad \text{[…]} 
\end{align*}
What does it mean for a compiler to be secure?
What does it mean for a compiler to be secure?

Known for type systems, CC but not for SC
Secure Implementation of Channel Abstractions

Martín Abadi  
ma@pa.dec.com  
Digital Equipment Corporation  
Systems Research Center

Cédric Fournet  
Cedric.Fournet@inria.fr  
INRIA Rocquencourt

Georges Gonthier  
Georges.Gonthier@inria.fr  
INRIA Rocquencourt

Abstract

Communication in distributed systems often relies on useful abstractions such as channels, remote procedure calls, and remote method invocations. The implementations of these abstractions sometimes provide security properties, in particular through encryption. In this spaces are on the same machine, and that a centralized operating system provides security for them. In reality, these address spaces could be spread across a network, and security could depend on several local operating systems and on cryptographic protocols across machines.

For example, when an application requires secure

Theorem 1 The compositional translation is fully-abstract, up to observational equivalence: for all join-calculus processes $P$ and $Q$,

$$ P \approx Q \text{ if and only if } \Env[[P]] \approx \Env[[Q]] $$

From the join-calculus to the sjoin-calculus
they needed a definition that their implementation of secure channels via cryptography was secure
Fully Abstract Compilation (FAC)

Theorem 1 The compositional translation is fully-abstract, up to observational equivalence: for all join-calculus processes $P$ and $Q$,

$$ P \approx Q \text{ if and only if } \text{Env}[[P]] \approx \text{Env}[[Q]] $$
How does Fully Abstract Compilation entail security?

FAC ensures that a target-level attacker has the same power of a source-level one as captured by the semantics.
How does Fully Abstract Compilation entail security?
How does Fully Abstract Compilation entail security?

FAC ensures that a target-level attacker has the same power of a source-level one as captured by the semantics.
Fully Abstract Compilation: Definition

\[ P_1 \sim_{ctx} P_2 \]

\[ \llbracket P_1 \rrbracket \sim_{ctx} \llbracket P_2 \rrbracket \]
Fully Abstract Compilation: Definition

\[ P_1 \simeq_{ctx} P_2 \]

\[ [P_1] \simeq_{ctx} [P_2] \]
Fully Abstract Compilation: Definition

\[ P_1 \approx_{ctx} P_2 \]

\[ \llbracket P_1 \rrbracket \approx_{ctx} \llbracket P_2 \rrbracket \]
Fully Abstract Compilation: Definition

\[ P_1 \sim_{ctx} P_2 \]

\[ \forall A. A \llbracket [P_1] \rrbracket \downarrow \iff A \llbracket [P_2] \rrbracket \downarrow \]
Are there Alternatives to FAC?

• **FAC is not precise** about security
Are there Alternatives to FAC?

- FAC is not precise about security
- this affects efficiency and proof complexity
Are there Alternatives to FAC?

- FAC is not precise about security
- this affects efficiency and proof complexity
- in certain cases we want easier/more efficient alternatives
Are there Alternatives to FAC?

- FAC is not precise about security
- this affects efficiency and proof complexity
- in certain cases we want easier/more efficient alternatives

preserve classes of security
(hyper)properties
Robust Compilation Criteria

Robust Relational Hyperproperty Preservation
Robust $K$-Relational Hyperproperty Preservation
Robust 2-Relational Hyperproperty Preservation

Robust Relational Property Preservation
Robust $K$-Relational Property Preservation
Robust 2-Relational Property Preservation

Robust Hyperproperty Preservation
Robust Subset-Closed Hyperproperty Preservation
Robust $K$-Subset-Closed Hyperproperty Preservation
Robust 2-Subset-Closed Hyperproperty Preservation

Robust Trace Property Preservation
Robust Dense Property Preservation

Robust Relational Safety Preservation
Robust Finite-Relational Safety Preservation
Robust $K$-Relational Safety Preservation
Robust 2-Relational Safety Preservation

Robust Trace Equivalence Preservation

Robust Trace Determinacy

Robust Hypersafety Preservation

Robust $K$-Hypersafety Preservation
Robust 2-Hypersafety Preservation

Robust Termination-Insensitive Noninterference Preservation

Tradeoffs for code efficiency, security guarantees, proof complexity
Robust Compilation Criteria

Tradeoffs for code efficiency, security guarantees, proof complexity
Robust Compilation Criteria

Tradeoffs for code efficiency, security guarantees, proof complexity
Robust Criteria: Intuition

Each point has two equivalent criteria:

• Property – ful:
  + clearly tells what class it preserves
Each point has two equivalent criteria:

- Property – ful:
  - clearly tells what class it preserves
  - harder to prove
Robust Criteria: Intuition

Each point has two **equivalent** criteria:

- **Property – ful**:
  + clearly tells what **class** it preserves
  - harder to prove
- **Property – free**:
  + **easier** to prove
Robust Criteria: Intuition

Each point has two equivalent criteria:

• Property – ful:
  + clearly tells what class it preserves
  - harder to prove

• Property – free:
  + easier to prove
  - unclear what security classes are preserved
Robust Criteria: Intuition

Each point has two equivalent criteria:

• Property – ful:
  + clearly tells what class it preserves
  - harder to prove

• Property – free:
  + easier to prove
  - unclear what security classes are preserved
  = akin to some crypto statements (UC)
In Depth Example: RSC

\[ \cdot \] = compiler \hspace{1cm} \boxed{\cdot} : \text{RSP} \overset{\text{def}}{=} \]
In Depth Example: RSC

\[ \lfloor \cdot \rfloor : \text{RSP} \overset{\text{def}}{=} \forall \pi \approx \pi \in \text{Safety}. \]

\[ \lfloor \cdot \rfloor = \text{compiler} \]

\[ \pi / \pi = \text{set of traces} \]
In Depth Example: RSC

[\cdot] = compiler
\pi / \pi = set of traces
P = partial program

\[ [\cdot] : \text{RSP} \overset{\text{def}}{=} \forall \pi \approx \pi \in \text{Safety. \forall P}. \]
In Depth Example: RSC

\[ \cdot \] : RSP \quad \text{def} \quad \forall \pi \sim \pi \in Safety. \forall P.

\text{if} (\forall A, t.

[\cdot] = \text{compiler}
\pi / \pi = \text{set of traces}
P = \text{partial program}
A / A = \text{attacker}
t / t = \text{trace of events}
In Depth Example: RSC

\[
[\cdot] : \text{RSP} \overset{\text{def}}{=} \forall \pi \sim \pi \in \text{Safety}. \forall P. \\
\text{if } (\forall A, t. A[P] \leadsto t)
\]

- \[\cdot\] = compiler
- \(.\) = set of traces
- \(P\) = partial program
- \(A/\cdot\) = attacker
- \(t/t\) = trace of events
- \([\cdot]\) = linking
- \(\leadsto/\leadsto\) = trace semantics
In Depth Example: RSC

\[
\begin{align*}
\llbracket \cdot \rrbracket : \text{RSP} & \overset{\text{def}}{=} \forall \pi \approx \pi \in \text{Safety}. \forall P. \\
& \text{if } (\forall A, t. A[P] \leadsto t \Rightarrow t \in \pi) \\
\end{align*}
\]

\[\llbracket \cdot \rrbracket = \text{compiler} \]
\[\pi / \pi = \text{set of traces} \]
\[P = \text{partial program} \]
\[A / A = \text{attacker} \]
\[t / t = \text{trace of events} \]
\[\llbracket \cdot \rrbracket = \text{linking} \]
\[\leadsto / \leadsto = \text{trace semantics} \]
In Depth Example: RSC

\[ \llbracket \cdot \rrbracket : \text{RSP} \overset{\text{def}}{=} \forall \pi \approx \pi \in \text{Safety}. \forall P. \]

\[ \text{if } (\forall A, t. A[P] \sim t \implies t \in \pi) \]

then \( (\forall A, t. \).

\[ \] = compiler
\( \pi / \pi = \) set of traces
\( P = \) partial program
\( A / A = \) attacker
\( t / t = \) trace of events
\( \llbracket \cdot \rrbracket = \) linking
\( \sim / \sim = \) trace semantics
In Depth Example: RSC

\[
\left[ \cdot \right]: \text{RSP} \overset{\text{def}}{=} \forall \pi \approx \pi \in \text{Safety}. \forall P. \\
\text{if } (\forall A, t. A[P] \sim t \Rightarrow t \in \pi) \\
\text{then } (\forall A, t. A[\lbrack P \rbrack] \sim t \Rightarrow
\]

- \left[ \cdot \right] = \text{compiler}
- \pi / \pi = \text{set of traces}
- P = \text{partial program}
- A / A = \text{attacker}
- t / t = \text{trace of events}
- \lbrack \cdot \rbrack = \text{linking}
- \sim / \sim = \text{trace semantics}
In Depth Example: RSC

$\boxed{\cdot} : \text{RSP} \overset{\text{def}}{=} \forall \pi \approx \pi \in \text{Safety}. \forall P.$

If $(\forall A, t. A[P] \leadsto t \Rightarrow t \in \pi)$

then $(\forall A, t. A[\boxed{P}] \leadsto t \Rightarrow t \in \pi)$

- $\boxed{\cdot}$ = compiler
- $\pi/\pi$ = set of traces
- $P$ = partial program
- $A/A$ = attacker
- $t/t$ = trace of events
- $\boxed{\cdot}$ = linking
- $\leadsto/\leadsto$ = trace semantics
In Depth Example: RSC

\[ [\cdot] : \text{RSP} \overset{\text{def}}{=} \forall \pi \approx \pi \in Safety. \forall P. \]
\[
\text{if } (\forall A, t. A[P] \leadsto t \Rightarrow t \in \pi)
\]
\[
\text{then } (\forall A, t. A[\llbracket P \rrbracket] \leadsto t \Rightarrow t \in \pi)
\]

\[ [\cdot] : \text{RSC} \overset{\text{def}}{=} \]

- [:] = compiler
- \(\pi / \pi\) = set of traces
- \(P\) = partial program
- \(A / A\) = attacker
- \(t / t\) = trace of events
- [:] = linking
- \(\leadsto / \leadsto\) = trace semantics
In Depth Example: RSC

\[
\begin{align*}
\llbracket \cdot \rrbracket : \text{RSP} & \overset{\text{def}}{=} \forall \pi \approx \pi \in \text{Safety}. \forall P. \\
& \text{if } (\forall A, t. A[P] \sim t \Rightarrow t \in \pi) \\
& \text{then } (\forall A, t. A[\llbracket P \rrbracket] \sim t \Rightarrow t \in \pi)
\end{align*}
\]

\[
\begin{align*}
\llbracket \cdot \rrbracket : \text{RSC} & \overset{\text{def}}{=} \forall P, A, m.
\end{align*}
\]
In Depth Example: RSC

\([\cdot] : \text{RSP} \overset{\text{def}}{=} \forall \pi \approx \pi \in \text{Safety}. \ \forall P. \)

\[
\begin{align*}
\text{if} \ (\forall A, t. A [P] \leadsto t \Rightarrow t \in \pi) \\
\text{then} \ (\forall A, t. A [\llbracket P \rrbracket] \leadsto t \Rightarrow t \in \pi)
\end{align*}
\]

\([\cdot] : \text{RSC} \overset{\text{def}}{=} \forall P, A, m. \\
\text{if} \ A [\llbracket P \rrbracket] \leadsto m
In Depth Example: RSC

\[
\text{[\cdot]} : \text{RSP} \overset{\text{def}}{=} \forall \pi \approx \pi \in \text{Safety.} \forall P.
\text{if } (\forall A, t. A[P] \leadsto t \Rightarrow t \in \pi)
\text{then } (\forall A, t. A[[P]] \leadsto t \Rightarrow t \in \pi)
\]

\[
\text{[\cdot]} : \text{RSC} \overset{\text{def}}{=} \forall P, A, m.
\text{if } A[[P]] \leadsto m
\text{then } \exists A, m.
\]
### In Depth Example: RSC

\[\llbracket \cdot \rrbracket : \text{RSP} \overset{\text{def}}{=} \forall \pi \simeq \pi \in \text{Safety}. \forall P. \]

\[
\quad \text{if } (\forall A, t. A[P] \rightsquigarrow t \Rightarrow t \in \pi) \]

\[
\quad \text{then } (\forall A, t. A[\llbracket P \rrbracket] \rightsquigarrow t \Rightarrow t \in \pi)
\]

\[\llbracket \cdot \rrbracket : \text{RSC} \overset{\text{def}}{=} \forall P, A, m. \]

\[
\quad \text{if } A[\llbracket P \rrbracket] \rightsquigarrow m \]

\[
\quad \text{then } \exists A, m. A[P] \rightsquigarrow m
\]
In Depth Example: RSC

\[ \text{[\cdot]} : \text{RSP} \overset{\text{def}}{=} \forall \pi \sim \pi \in \text{Safety}. \forall P. \]

\[ \text{if } (\forall A, t. A[P] \sim t \Rightarrow t \in \pi) \]

\[ \text{then } (\forall A, t. A[[P]] \sim t \Rightarrow t \in \pi) \]

\[ \text{[\cdot]} : \text{RSC} \overset{\text{def}}{=} \forall P, A, m. \]

\[ \text{if } A[[P]] \sim m \]

\[ \text{then } \exists A, m. A[P] \sim m \text{ and } m \sim m \]
Understanding RSC

RSP/RSC:

• adaptable to reason about complex features: concurrency, undefined behaviour
Understanding RSC

RSP/RSC:

• adaptable to reason about complex features: concurrency, undefined behaviour

RSP:

• provable if source is robustly-safe
Understanding RSC

RSP/RSC:

• adaptable to reason about complex features: *concurrency, undefined behaviour*

RSP:

• provable *if source is robustly-safe*

RSC:

• easiest *backtranslation proof*
Both:

- **robust** (\(\forall A\))
- rely on program semantics

**FAC**
- yields a language result

**RSC**/
- extends the semantics (\(\text{uni219D}\)) to focus on security
Both:

• robust \( (\forall A) \)
Both:

- robust ($\forall A$)
- rely on program semantics
Both:

- robust ($\forall A$)
- rely on program semantics

FAC:
Both:

- robust ($\forall A$)
- rely on program semantics

FAC:

- yields a language result
Both:

- robust (\(\forall A\))
- rely on program semantics

FAC:

- yields a language result

RSC/RSP:

- extends the semantics (\(\sim\)) to focus on security
Is There More?

Some still unknown foundations include:

• optimisation
• composition (multipass & linking)
Is There More?

Some **still unknown** foundations include:
Is There More?

Some **still unknown** foundations include:

- optimisation
Is There More?

Some **still unknown** foundations include:

- optimisation
- composition (multipass & linking)
Robust(ly Safe) Compilation at Work

Instantiate RSC to specific properties

- absence of *speculation leaks*
Robust(ly Safe) Compilation at Work

Instantiate RSC to specific properties

- absence of speculation leaks
- memory safety preservation (spatial, temporal)

CCS’21
Robust(ly Safe) Compilation at Work

Instantiate RSC to specific properties

- absence of speculation leaks
- memory safety preservation (spatial, temporal)
- constant-time preservation
Robust(ly Safe) Compilation at Work

Instantiate RSC to specific properties

- absence of speculation leaks
- memory safety preservation (spatial, temporal)
- constant-time preservation
- ...

CCS'21
Future Outlook
More Secure Compilation

• secure compilation for Spectre V/two.osf/plus.osf (w. Imdea, Cispa)
• secure compilation to webassembly (w. UCSD, Harvard, Cispa)
• secure compilation & universal composability (w. Stanford, Cispa)
• secure compilation for linear languages (w. Novi / FB)
• . . . (some PL too, w. Stanford, KU Leuven)
More Secure Compilation

- secure compilation for Spectre V2+ (w. Imdea, Cispa)
More Secure Compilation

- secure compilation for Spectre V2+ (w. Imdea, Cispa)
- secure compilation to webassembly (w. UCSD, Harvard, Cispa)
More Secure Compilation

• secure compilation for Spectre V2+ (w. Imdea, Cispa)
• secure compilation to webassembly (w. UCSD, Harvard, Cispa)
• secure compilation & universal composability (w. Stanford, Cispa)
More Secure Compilation

• secure compilation for Spectre V2+ (w. Imdea, Cispa)
• secure compilation to webassembly (w. UCSD, Harvard, Cispa)
• secure compilation & universal composability (w. Stanford, Cispa)
• secure compilation for linear languages (w. Novi / FB)
More Secure Compilation

- secure compilation for Spectre V2+ (w. Imdea, Cispa)
- secure compilation to webassembly (w. UCSD, Harvard, Cispa)
- secure compilation & universal composability (w. Stanford, Cispa)
- secure compilation for linear languages (w. Novi / FB)
- ...(some PL too, w. Stanford, KU Leuven)
Questions?