SECOMP
Efficient Formally Secure Compilers to a Tagged Architecture

Cătălin Hriţcu
INRIA Paris
Principal investigator: Cătălin Hrițcu

[2005-2011] MSc & PhD @ Saarland University, Saarbrücken, Germany
[2011-2013] Research Associate @ University of Pennsylvania, USA
[2013-now] Research Scientist @ INRIA Paris, France

• **Publications** (20+ papers, 500+ citations)
  
  **Best venues in security** (2×Oakland S&P, CCS, 3×CSF, 2×JCS)
  **and programming languages** (2×POPL, 2×ICFP, 2×JFP, ASPLOS, LMCS)

  **Software Foundations** teaching programming languages & logic with Coq

• **Currently supervising 2 PhD and 3 MSc students**

• **General chair** of IEEE European Symposium on Security & Privacy 2017

• **PC member** for POPL 2017, CSF 2016, ITP 2016, CPP 2016, POST 2017
Devising formal methods
- programming languages
- type systems, logics
- verification systems
- proof assistants
- property-based testing

Solving security problems
- formal attacker models
- provably secure systems
- stopping low-level attacks
- reference monitors
- security protocols

Resulted in many innovative tools
- Micro-Policies, F*, QuickChick, Luck, ...
The problem: devastating low-level attacks

- 1. inherently insecure low-level languages (C, C++)
  - memory unsafe: any buffer overflow can be catastrophic allowing remote attackers to gain complete control

- 2. unsafe interoperability with lower-level code
  - even code written in safer high-level languages (Java, C#, OCaml) has to interoperate with insecure low-level libraries (C, C++, ASM)
  - unsafe interoperability: all high-level safety guarantees lost

- Today’s languages & compilers plagued by low-level attacks
  - main culprit: hardware provides no appropriate security mechanisms
  - fixing this purely in software would be way too inefficient
Key enabler: Micro-Policies

software-defined, hardware-accelerated, tag-based monitoring

store

software monitor’s decision is hardware cached
Key enabler: Micro-Policies

[Oakland ’13 & ’15, POPL ’14, ASPLOS ’15]

software-defined, hardware-accelerated, tag-based monitoring

\[
\begin{array}{|c|c|}
\hline
pc & tpc \\
\hline
r0 & tr0 \\
\hline
r1 & tr1 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
\text{mem}[0] & tm0 \\
\hline
\text{"store r0 r1"} & tm1 \\
\hline
\text{mem}[2] & tm2 \\
\hline
\text{mem}[3] & tm3 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
tpc & tr0 & \text{tr1} \neq tm3 & \text{tm3} & \text{tm1} \\
\hline
\end{array}
\]

store

monitor

policy violation stopped!
(e.g. out of bounds write)
SECOMP grand challenge

Build the first efficient formally secure compilers for realistic programming languages

1. Provide secure semantics for low-level languages
   - C with protected components and memory safety

2. Enforce secure interoperability with lower-level code
   - ASM, C, and F* [F* = ML + verification, POPL ’16]
Formally verify: **full abstraction**

holy grail of secure compilation, enforcing abstractions all the way down
Formally verify: **full abstraction**

holly grail of secure compilation, enforcing abstractions all the way down

Benefit: **sound security reasoning in the source language**

- forget about compiler chain (linker, loader, runtime system)
- forget that libraries are written in a lower-level language
SECOMP: achieving full abstraction at scale

**F* language**  
(ML + verification)

**C language**  
+ memory safety  
+ components

miTLS*
SECOMP: achieving full abstraction at scale

F* language
(ML + verification)

C language
+ memory safety
+ components
SECOMP: achieving full abstraction at scale

**F* language**
(ML + verification)

**C language**
+ memory safety
+ components

- miTLS*
  - SecF* + SecML
  - memory safe C component
SECOMP: achieving full abstraction at scale

**F* language**
(ML + verification)

**C language**
+ memory safety
+ components

miTLS*

SecF* + SecML

memory safe C component
SECOMP: achieving full abstraction at scale

- **F* language** (ML + verification)
- **C language** + memory safety + components
- **ASM language** (RISC-V + micro-policies)
SECOMP: achieving full abstraction at scale

F* language
(ML + verification)

C language
+ memory safety
+ components

ASM language
(RISC-V + micro-policies)
**SECOMP: achieving full abstraction at scale**

**F* language**  
(ML + verification)

**C language**  
+ memory safety  
+ components

**ASM language**  
(RISC-V + micro-policies)

---

Diagram:

Diagram showing relationships between different components and languages, with labels for `miTLS*`, `SecF* + SecML`, `CompSec+`, `memory safe C component`, `legacy C component`, and `ASM component`. The diagram illustrates how various components and languages interact, with a focus on protecting component boundaries.
SECOMP: achieving full abstraction at scale

F* language
(ML + verification)

C language
+ memory safety
+ components

ASM language
(RISC-V + micro-policies)

protecting component boundaries
SECOMP: achieving full abstraction at scale

F* language
(ML + verification)

C language
+ memory safety
+ components

ASM language
(RISC-V + micro-policies)

miTLS*

SecF* + SecML

CompSec+

CompSec

memory safe C component

legacy C component

AS M component

protecting component boundaries

protecting higher-level abstractions
Protecting component boundaries

• Add mutually distrustful components to C
  – interacting only via strictly enforced interfaces

• **CompSec compiler chain** (based on CompCert)
  – propagate interface information to produced binary

• **Micro-policy simultaneously enforcing**
  – component separation
  – type-safe procedure call and return discipline

• **Fundamental challenge:** Proper attacker model
  – extending full abstraction to mutual distrust + unsafe source
Protecting higher-level abstractions

• **Enforcing more interesting abstractions** with micro-policies
  – ML: stronger types, value immutability, GC vs malloc/free, ...
  – F*: strong specifications (via dynamic boundary checks)

• **Fundamental challenge:** Micro-policies for C and ML
  – consequence: put micro-policies in the hands of programmers

• **Fundamental challenge:** Secure micro-policy composition
  – one micro-policy’s behavior can break another’s guarantees
SECOMP research team

- **Cătălin Hrițcu** (principal investigator, 75%)
- **ERC: 1 Junior Researcher, 2 PostDocs, 3 PhD students**
- 1 already funded PhD student: Yannis Juglaret

<table>
<thead>
<tr>
<th>WP</th>
<th>Year 1</th>
<th>Year 2</th>
<th>Year 3</th>
<th>Year 4</th>
<th>Year 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. CompSec</td>
<td>Yannis + JR</td>
<td></td>
<td>JR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. CompSafe</td>
<td></td>
<td>JR + PhD 2</td>
<td></td>
<td>PhD 2</td>
<td></td>
</tr>
<tr>
<td>3. CompSec+</td>
<td></td>
<td>JR + PhD 2</td>
<td></td>
<td>PhD 2 + PostDoc 2</td>
<td></td>
</tr>
<tr>
<td>4. Compose μP</td>
<td>PhD 1 + JR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5. C/ML + μP</td>
<td>PhD 1</td>
<td>PhD 1 + PostDoc 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6. SecML</td>
<td></td>
<td></td>
<td>PhD 3</td>
<td>PhD 3 + PostDoc 2</td>
<td></td>
</tr>
<tr>
<td>7. SecF*</td>
<td></td>
<td></td>
<td>PostDoc 1</td>
<td>PostDoc 2</td>
<td></td>
</tr>
<tr>
<td>8. miTLS*</td>
<td></td>
<td></td>
<td>PostDoc 1</td>
<td></td>
<td>PostDoc 2</td>
</tr>
</tbody>
</table>
Collaborators & Community

• Ongoing projects
  – **Micro-Policies**: INRIA, UPenn, MIT, Portland State, Draper Labs
  – **F* and miTLS**: INRIA, Microsoft Research
  – **CompCert**: INRIA, Princeton

• New potential collaborators
  – Several other researchers working on secure compilation
    • Deepak Garg (MPI-SWS), Frank Piessens (KU Leuven),
      Martin Abadi (Google), Amal Ahmed (Northeastern)

• **Secure compilation workshop** @ INRIA Paris, August 2016
  – build larger research community, identify open problems,
    bring together communities (hardware, systems, security,
    languages, verification, ...)


SECOMP in a nutshell

- We need more secure languages, compilers, hardware
- Key enabler: micro-policies (software-hardware protection)
- Grand challenge: the first efficient formally secure compilers for realistic programming languages (C, ML, F*)
- Answering challenging fundamental questions
  - attacker models, composition, micro-policies for C
- Achieving, testing, and proving full abstraction
- Very ambitious and risky milestone project, but ...
  - experience, preliminary results, team, collaborations, community
- Impact: unprecedented security, could become mainstream