SECOMP

Efficient Formally Secure Compilers to a Tagged Architecture

Cătălin Hrițcu

Inria Paris

Prosecco team

5 year vision

European Research Council

new grant

https://secure-compilation.github.io/

Computers are insecure

- **devastating low-level vulnerabilities**
- **programming languages, compilers, and hardware architectures**
  – designed in an era of scarce hardware resources
  – too often trade off security for efficiency
- **the world has changed** (2016 vs 1972*)
  – security matters, hardware resources abundant
  – time to revisit some tradeoffs

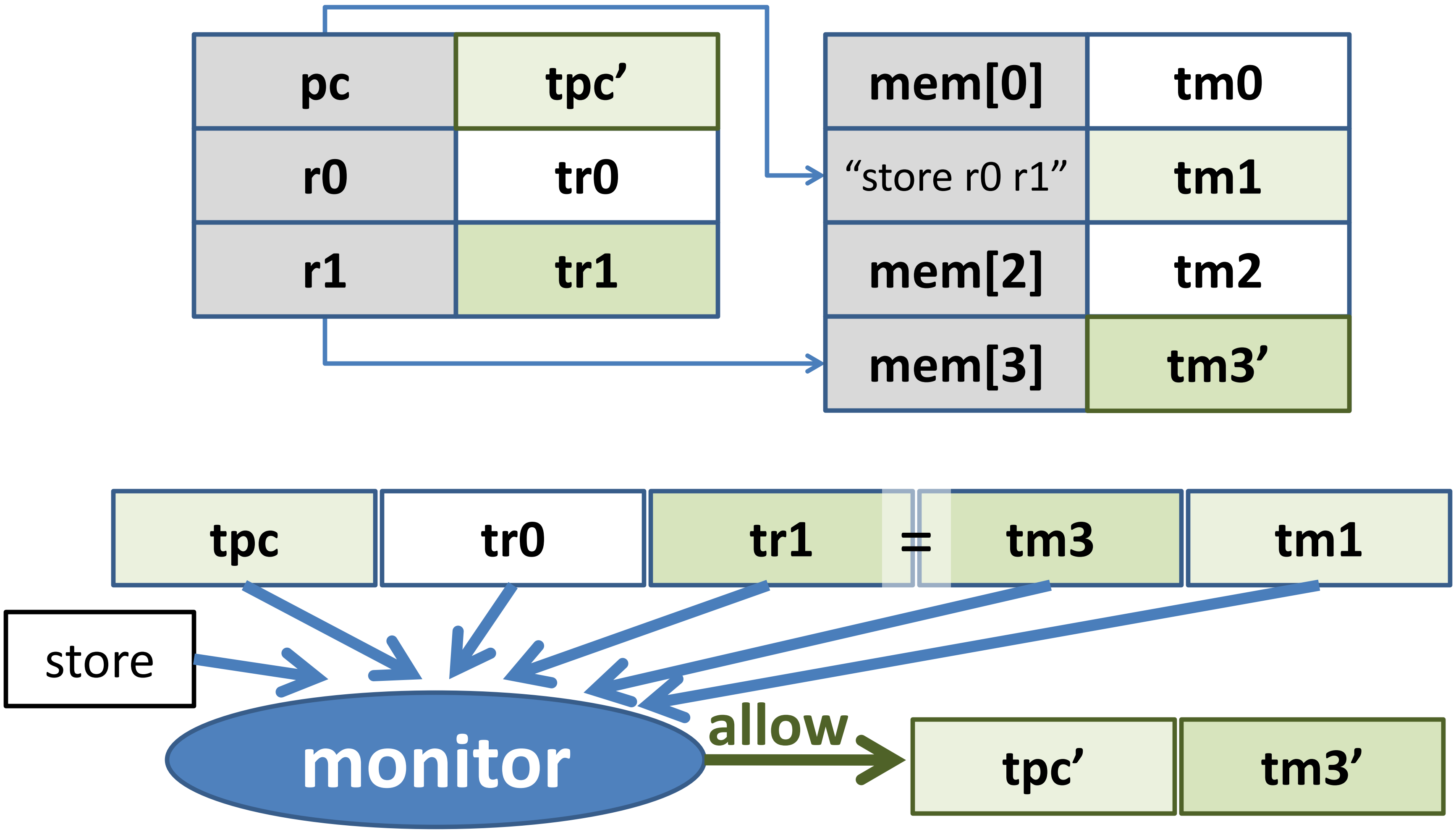
* "...the number of UNIX installations has grown to 10, with more expected..."
-- *Dennis Ritchie and Ken Thompson, June 1972*

Teasing out 2 important problems

- **1. inherently insecure low-level languages**
  - **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** has to interoperate with **insecure low-level libraries**
  - **unsafe interoperability:** all high-level safety guarantees lost

Key enabler: Micro-Policies

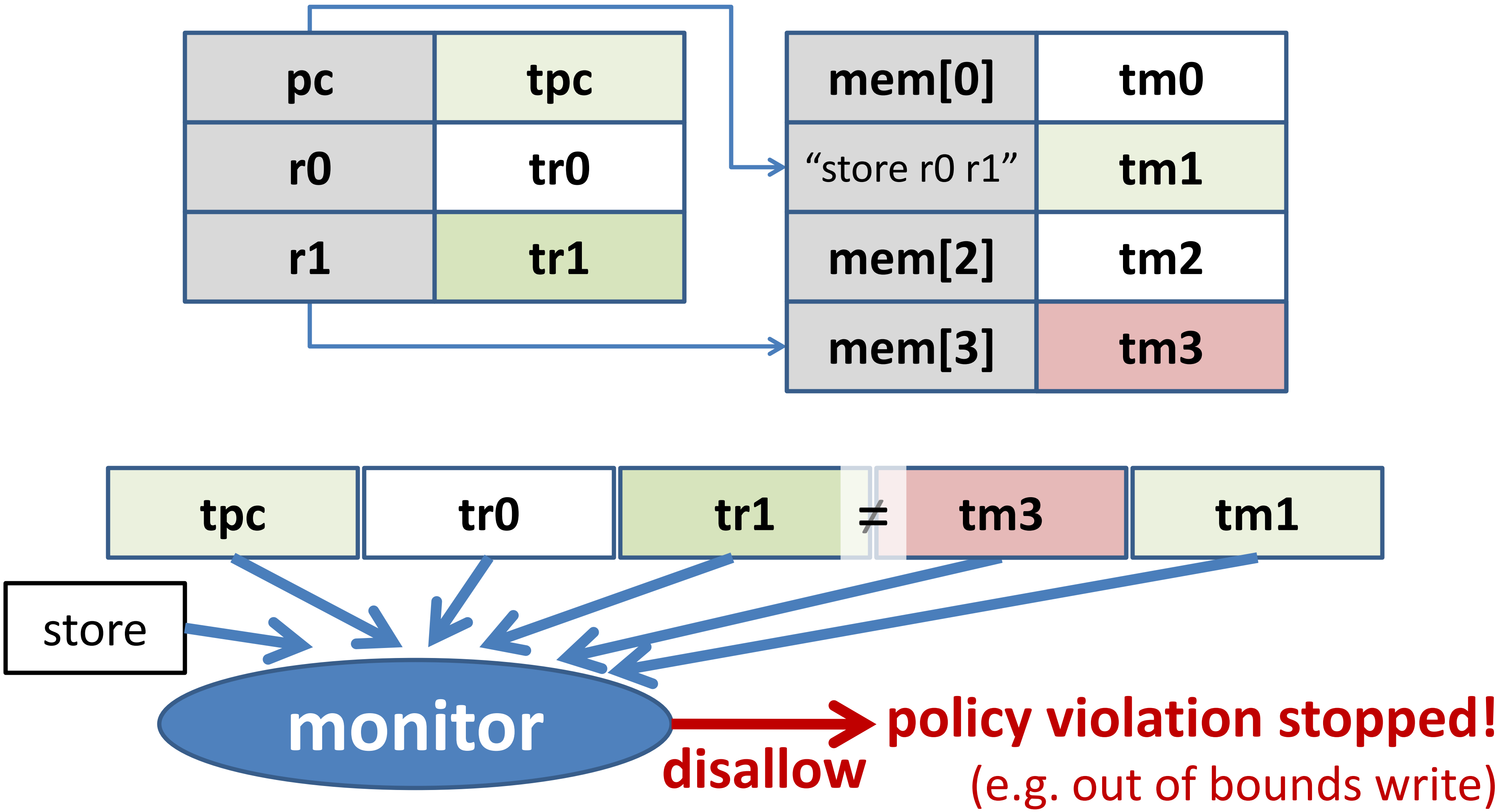
software-defined, hardware-accelerated, tag-based monitoring



software monitor’s decision is hardware cached

Key enabler: Micro-Policies

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



Micro-policies are cool!

- **low level + fine grained**: unbounded per-word metadata, checked & propagated on each instruction
- **flexible**: tags and monitor defined by software
- **efficient**: software decisions hardware cached
- **expressive**: complex policies for secure compilation
- **secure** and **simple** enough to verify security in Coq
- **real**: FPGA implementation on top of RISC-V

Expressiveness

Way beyond MPX, SGX, SSM, etc

- information flow control (IFC) [POPL'14]
- monitor self-protection
- protected compartments
- dynamic sealing
- heap memory safety
- code-data separation
- control-flow integrity (CFI)
- taint tracking
- ...

Verified
(in Coq)
[Oakland'15]

Evaluated
(<10% runtime overhead)
[ASPLOS'15]

Micro-Policies team

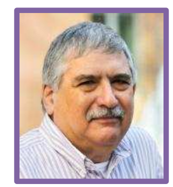
- **Formal methods** & **architecture** & **systems**
- **Current team**:
  - *Inria Paris*: **Cătălin Hrițcu, Guglielmo Fachini, Marco Stronati,** (**Yannis Juglaret**)
  - *UPenn*: **André DeHon**, **Benjamin Pierce, Arthur Azevedo de Amorim**, **Nick Roessler**
  - *Portland State*: **Andrew Tolmach**
  - *MIT:* **Howie Shrobe, Stelios Sidiroglou-Douskos**
  - *Industry*: **Draper Labs**
- **Spinoff of past project: DARPA CRASH/SAFE (2011-2014)**









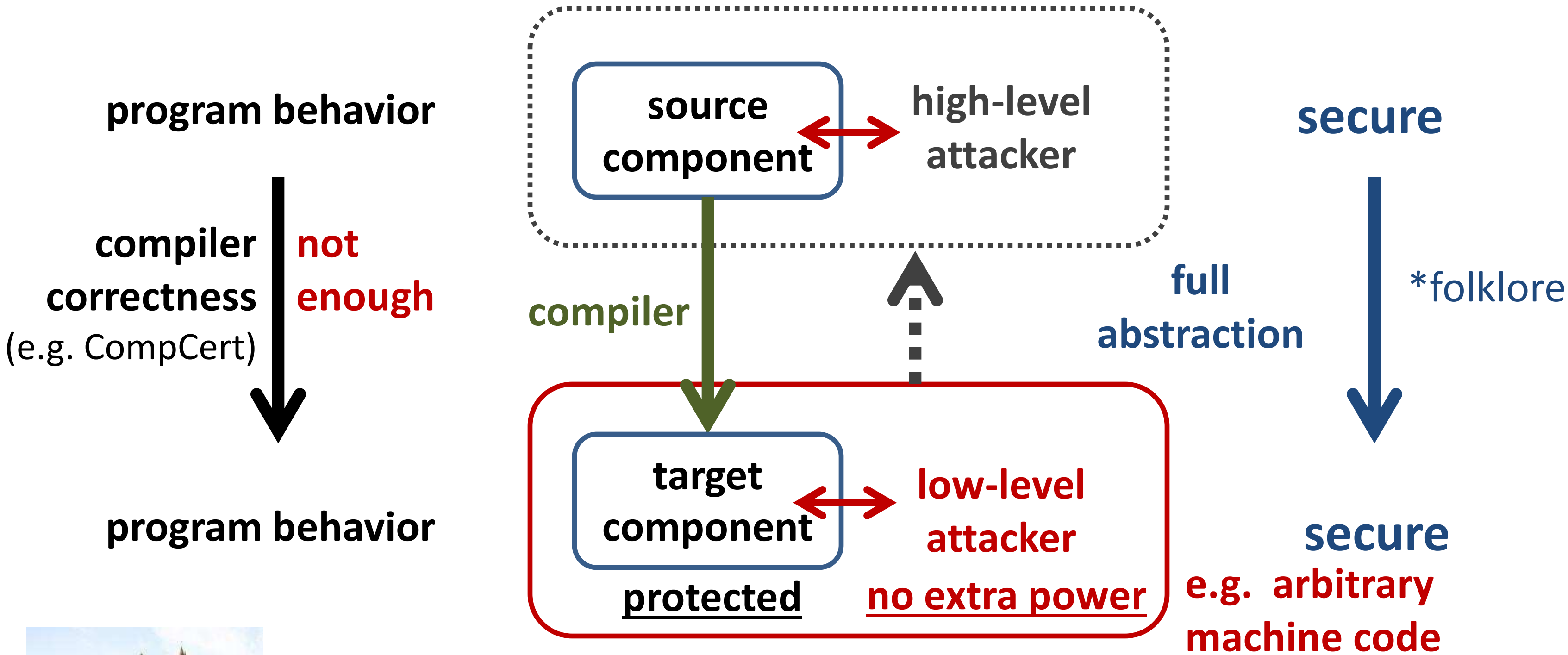

SECOMP grand challenge

Use micro-policies to 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 Low* [= C subset embedded in F* for verification]

Formally verify: full abstraction

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

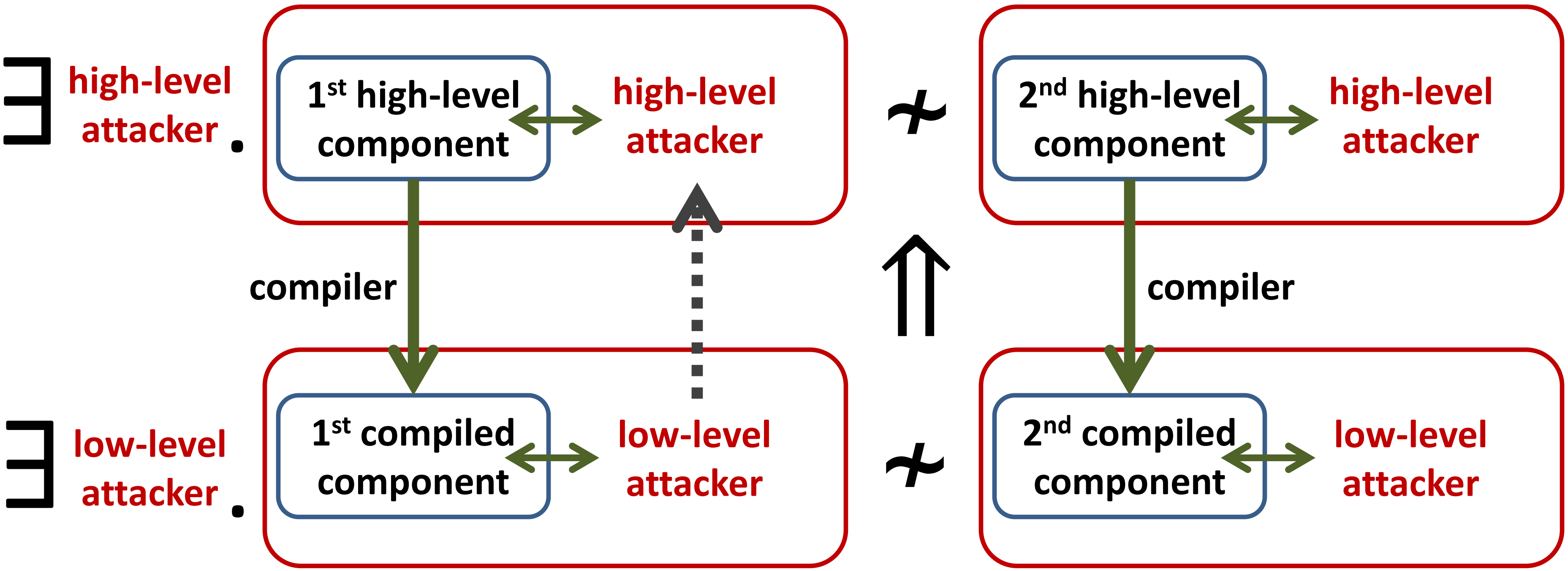


Benefit: sound security reasoning in the source language

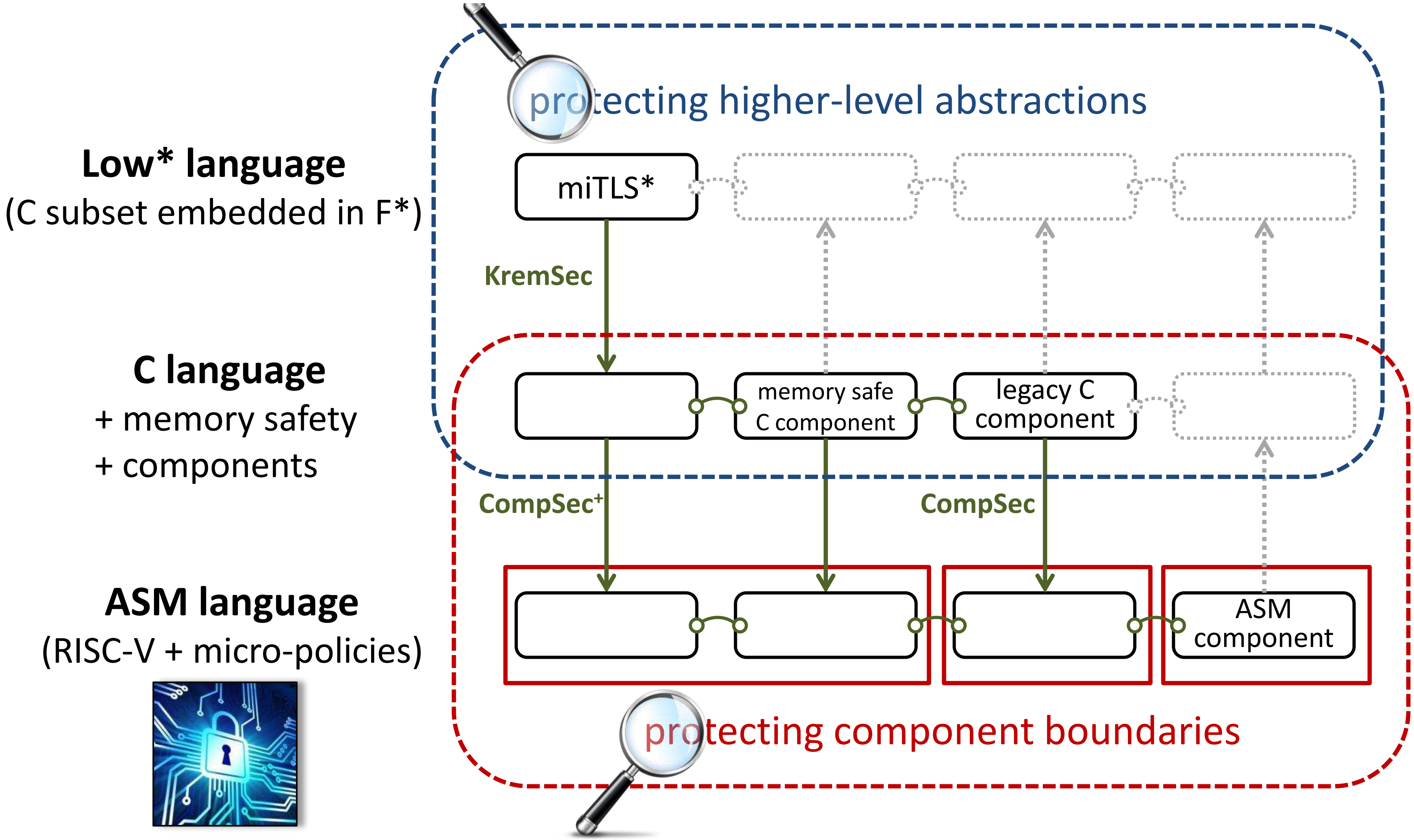
forget about compiler chain (linker, loader, runtime system)

forget that libraries are written in a lower-level language

Fully abstract compilation, definition



SECOMP: achieving full abstraction at scale



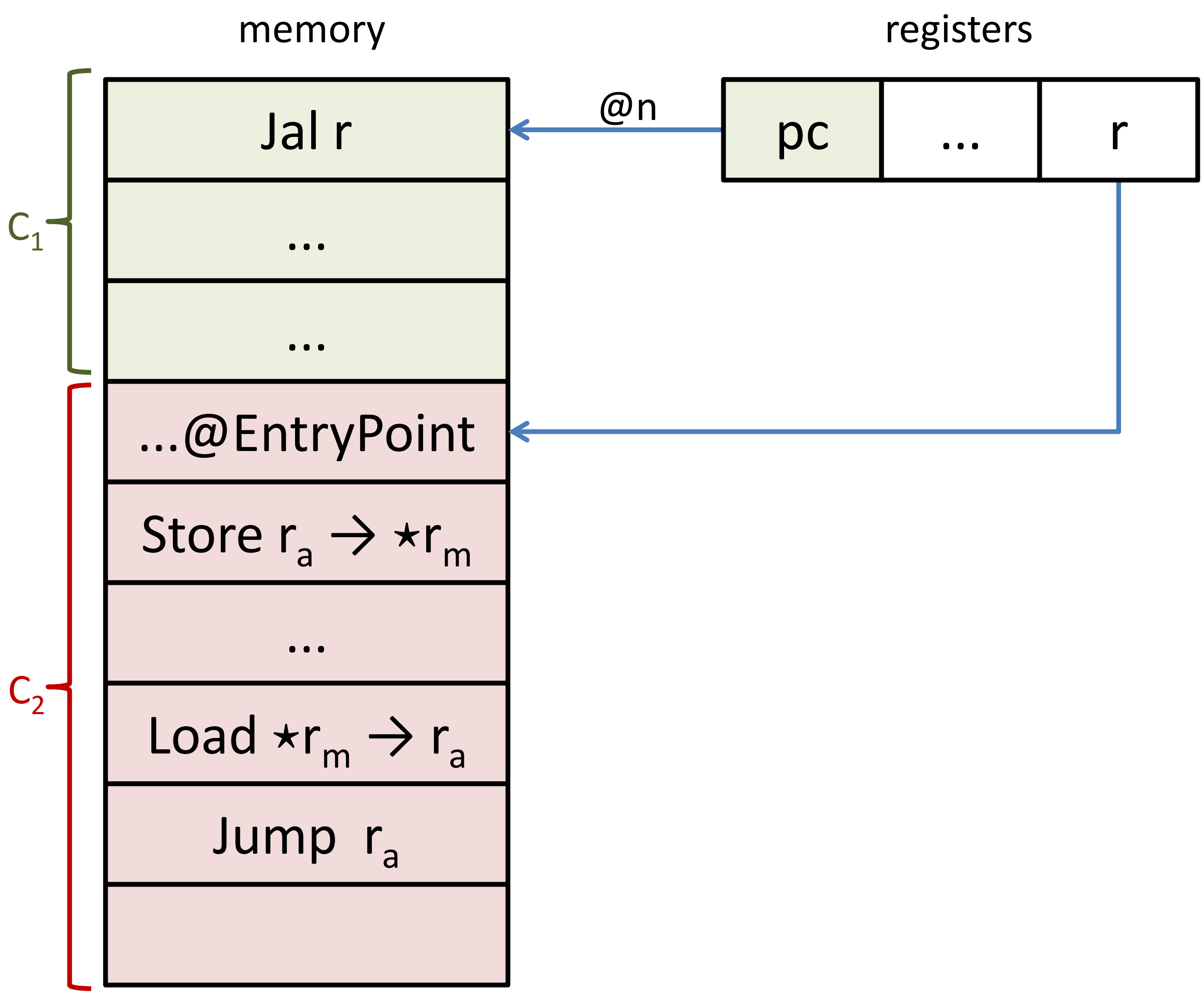
Protecting component boundaries

- **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
- **Interesting attacker model**
  - extending full abs. to mutual distrust + unsafe source

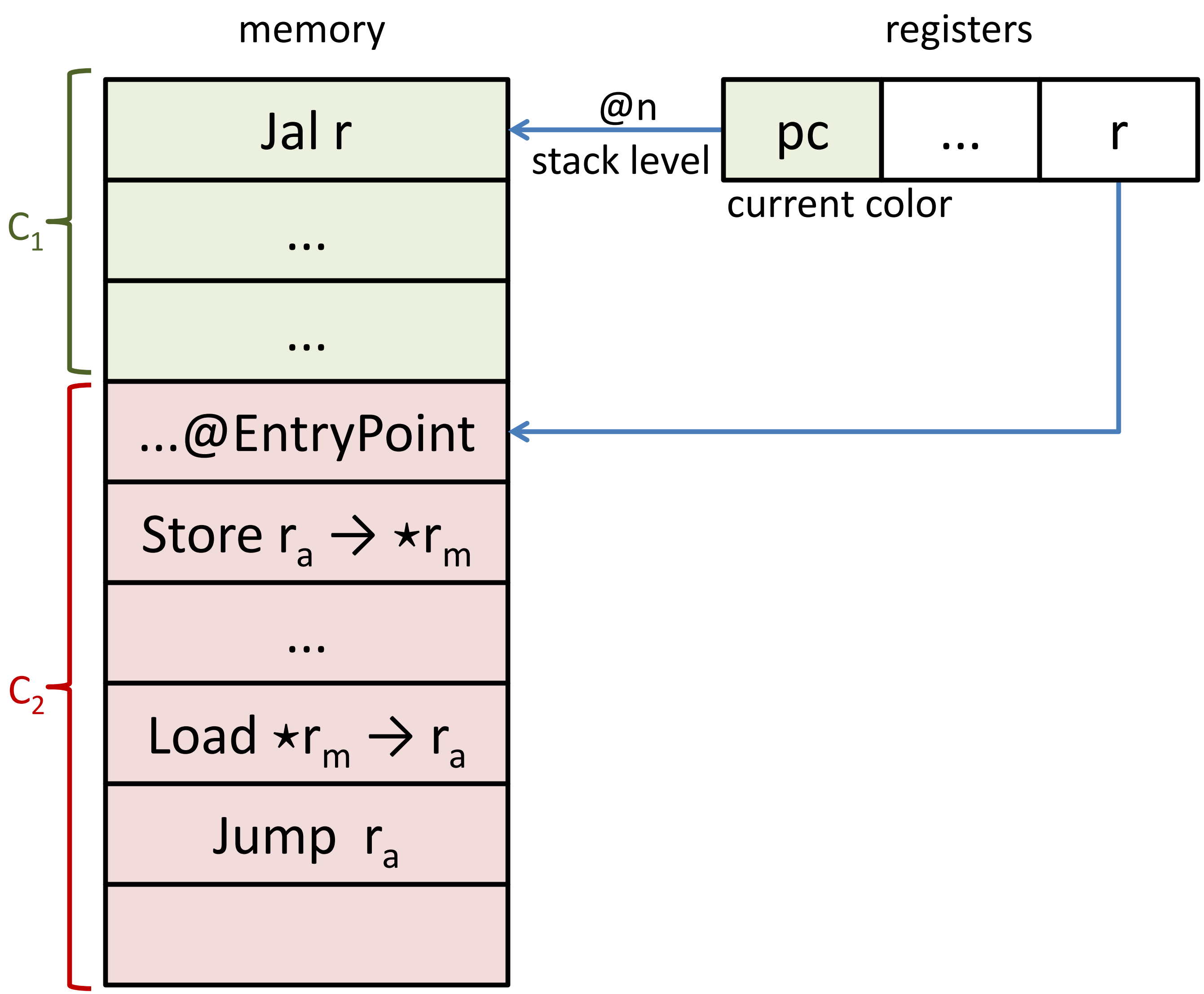
Recent work, joint with Yannis Juglaret et al

Protected components micro-policy



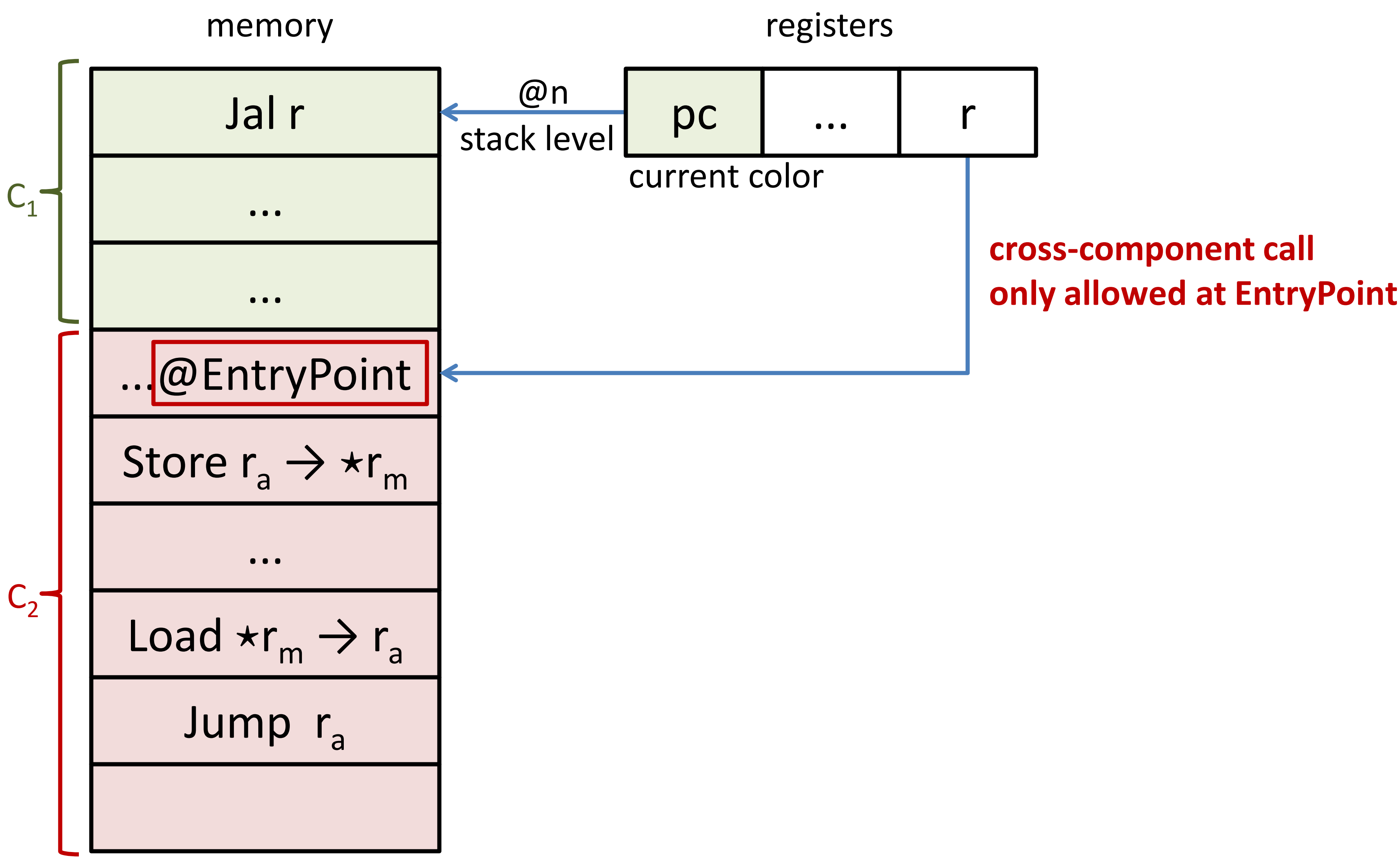
[Towards a Fully Abstract Compiler Using Micro-Policies, Juglaret et al, TR 2015]

Protected components micro-policy



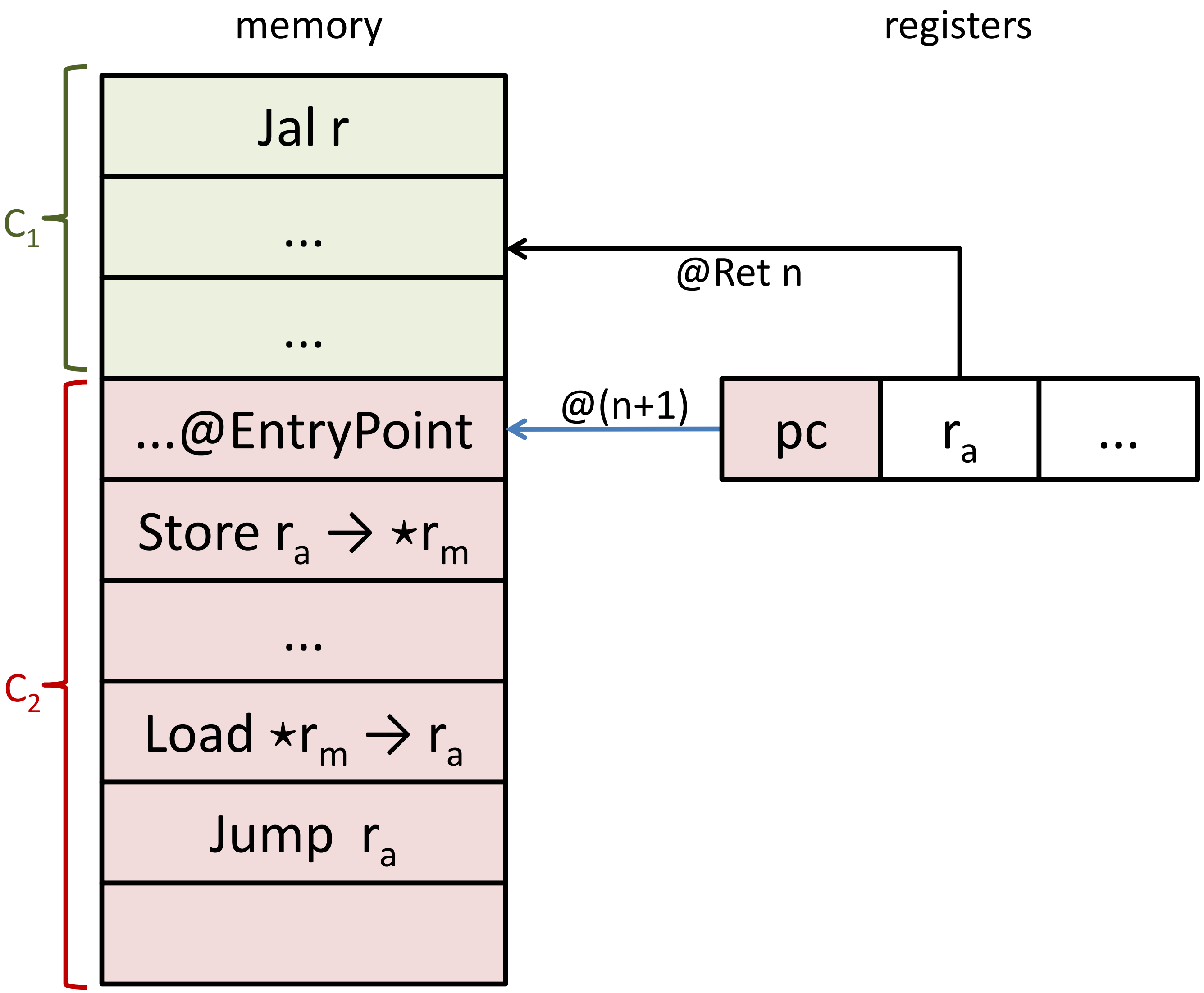
[Towards a Fully Abstract Compiler Using Micro-Policies, Juglaret et al, TR 2015]

Protected components micro-policy



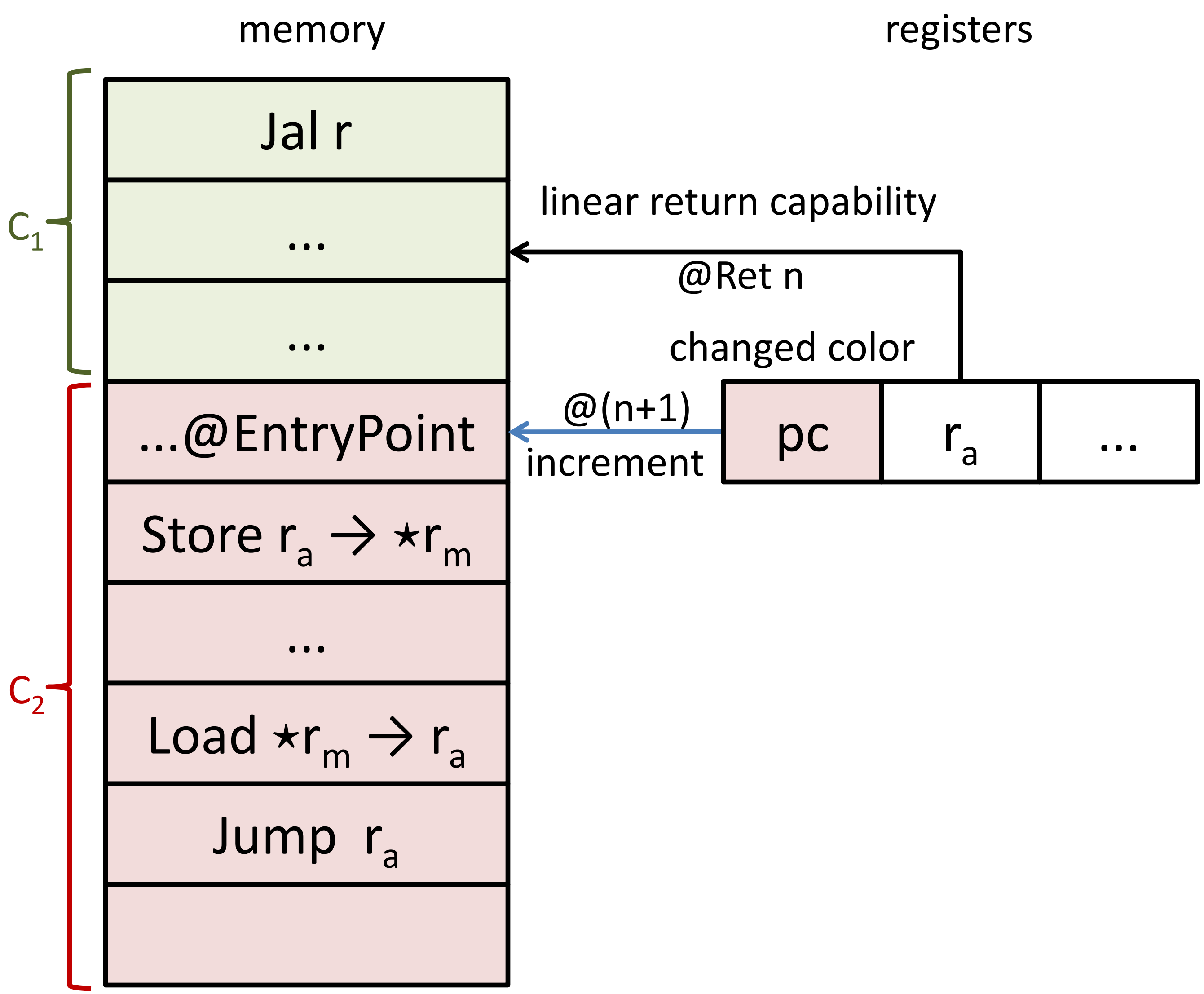
[Towards a Fully Abstract Compiler Using Micro-Policies, Juglaret et al, TR 2015]

Protected components micro-policy



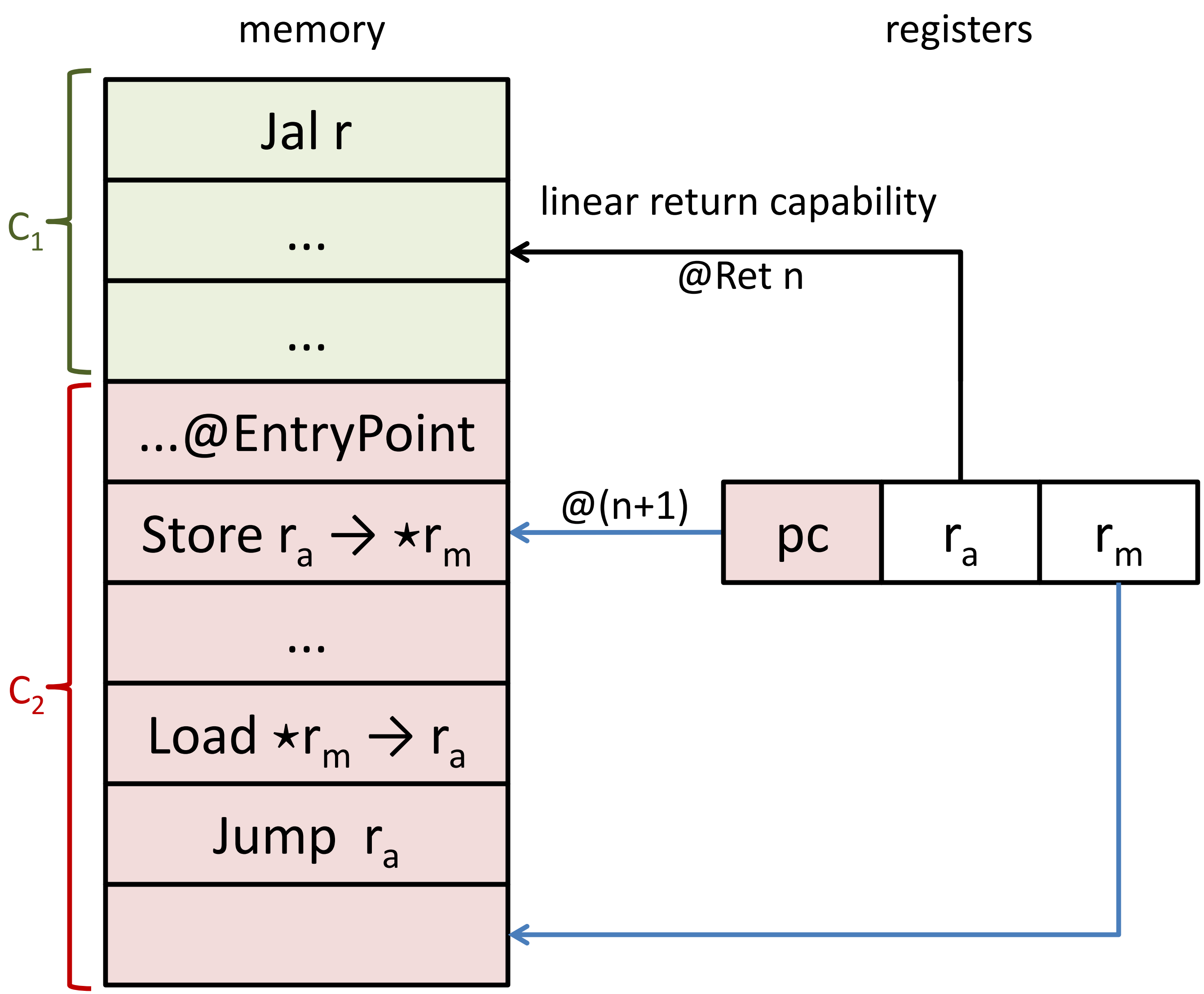
[Towards a Fully Abstract Compiler Using Micro-Policies, Juglaret et al, TR 2015]

Protected components micro-policy



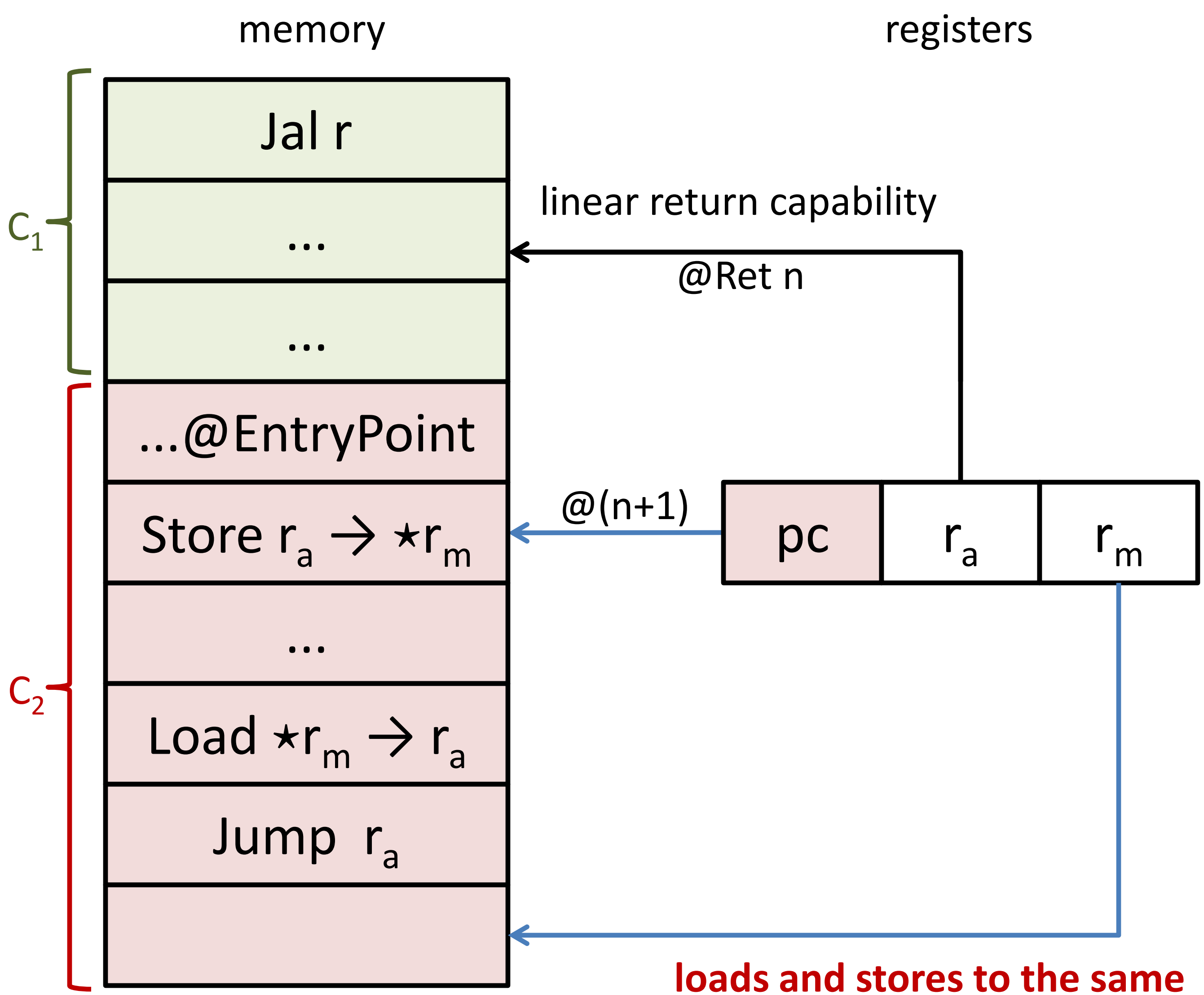
[Towards a Fully Abstract Compiler Using Micro-Policies, Juglaret et al, TR 2015]

Protected components micro-policy



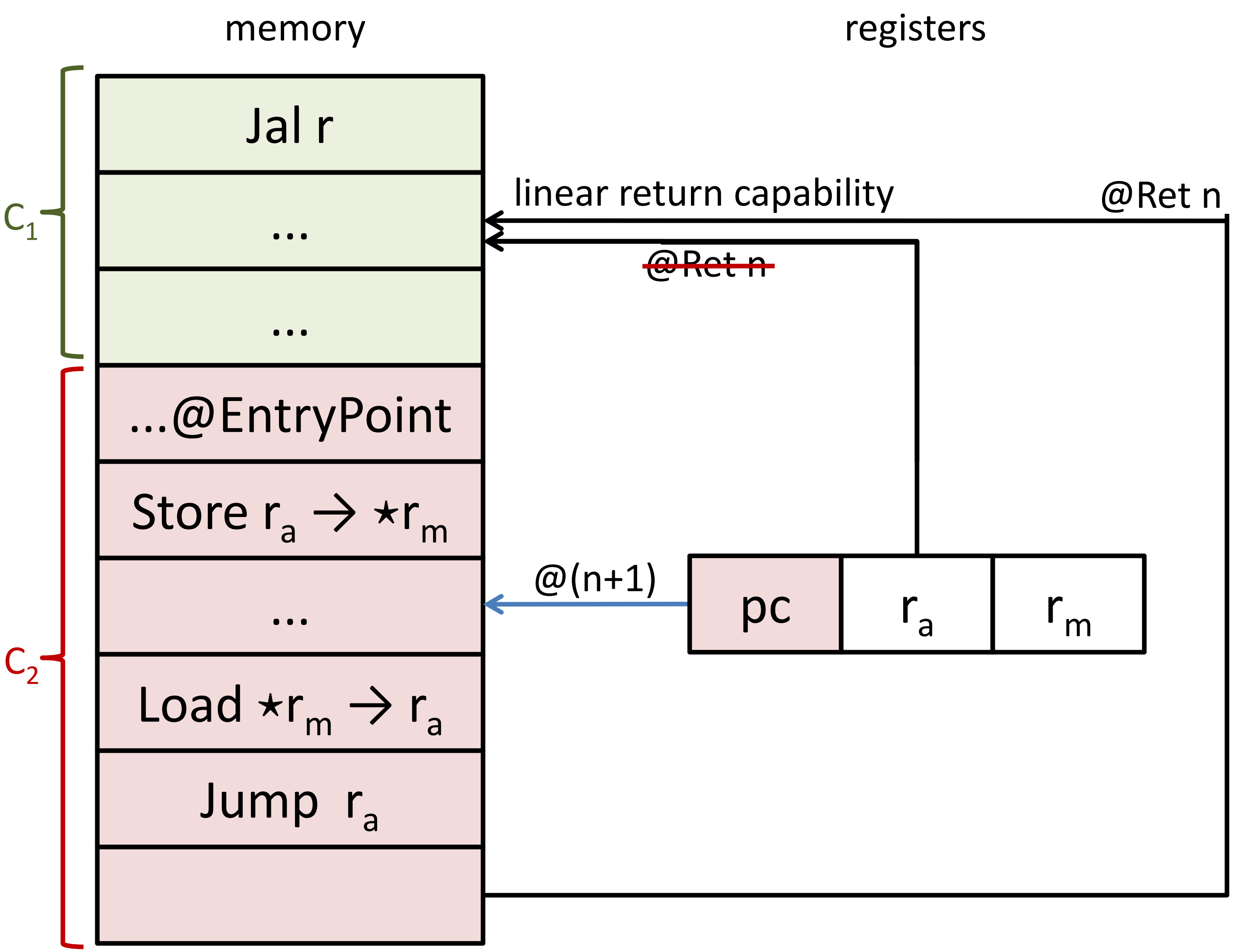
[Towards a Fully Abstract Compiler Using Micro-Policies, Juglaret et al, TR 2015]

Protected components micro-policy

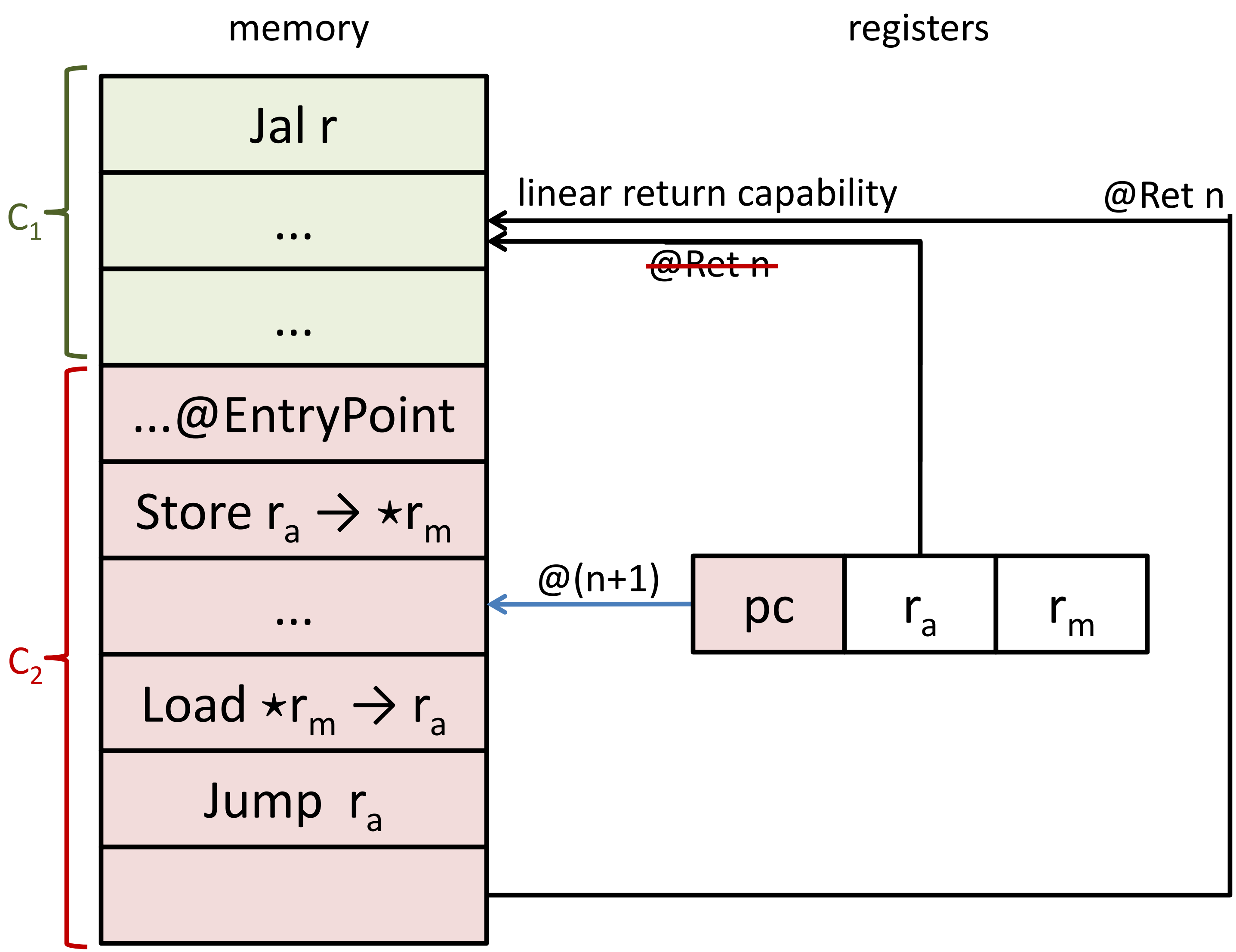


loads and stores to the same component always allowed

Protected components micro-policy

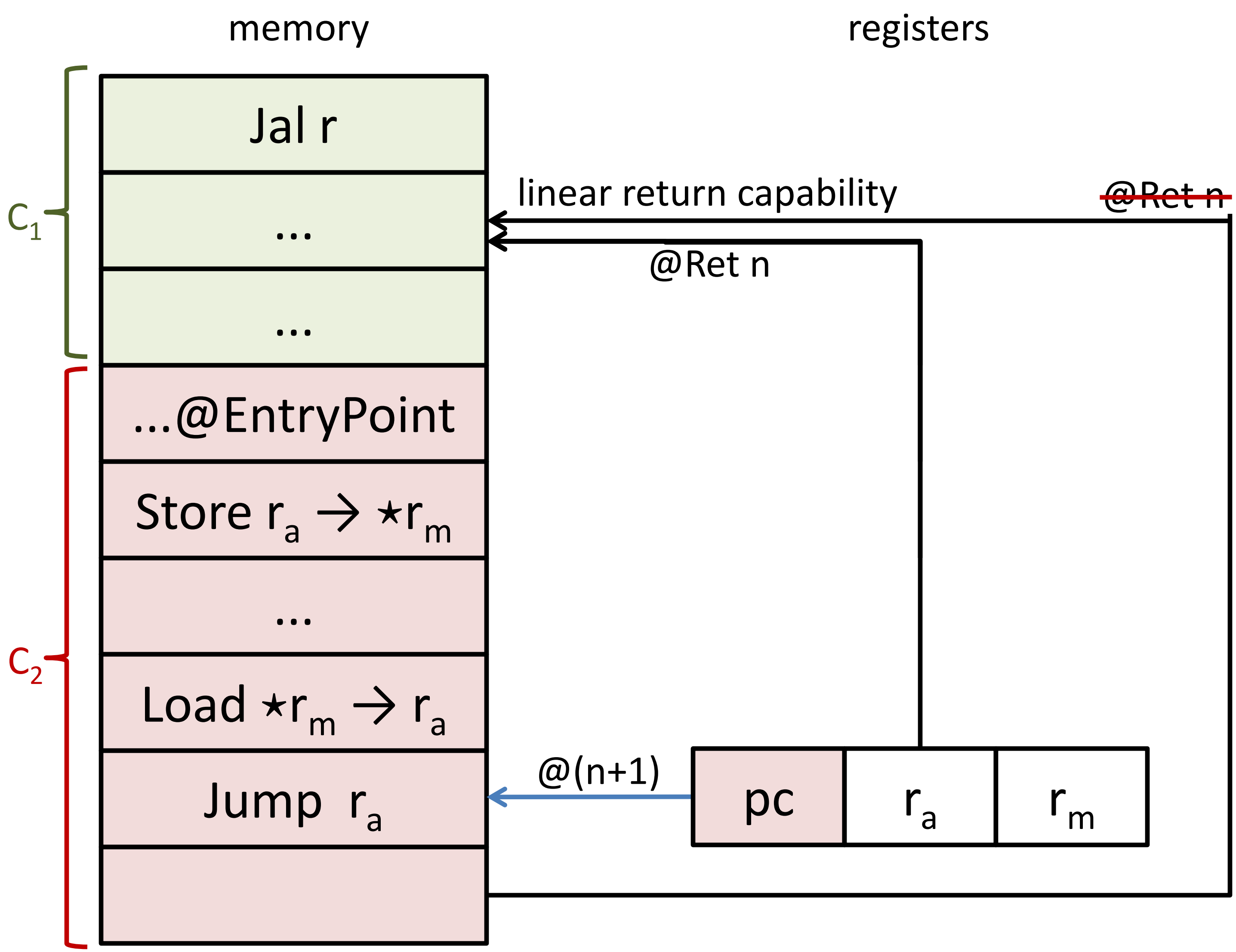


Protected components micro-policy



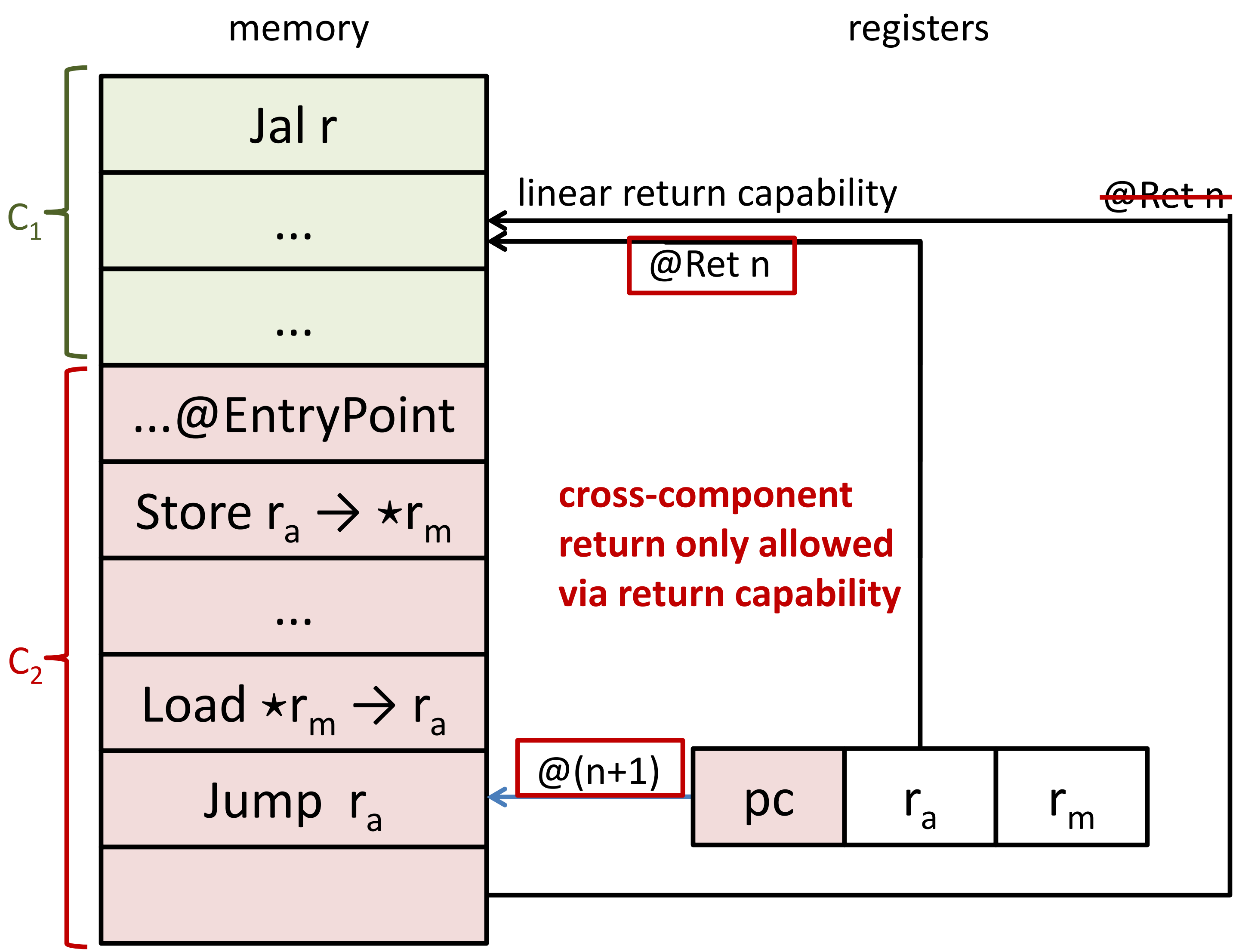
invariant:
at most one
return capability
per call stack level

Protected components micro-policy



invariant:
at most one return capability per call stack level

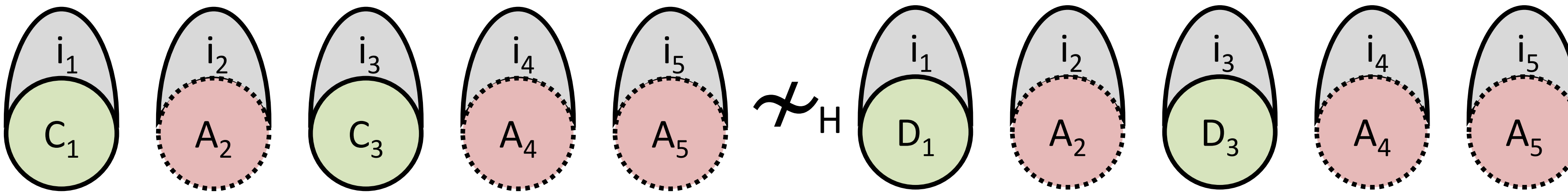
Protected components micro-policy



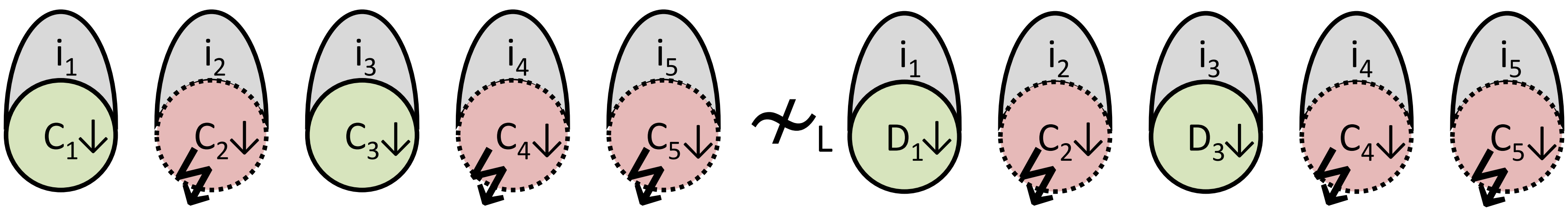
invariant:
at most one return capability per call stack level

Secure compartmentalizing compilation (SCC)

∀compromise scenarios.



∀ low-level attack from compromised $C_2\downarrow$, $C_4\downarrow$, $C_5\downarrow$
∃ high-level attack from some fully defined A_2, A_4, A_5

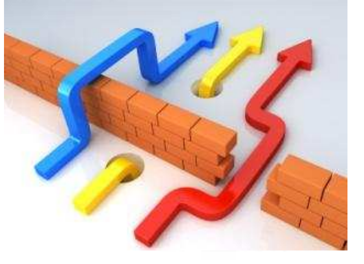


follows from "structured full abstraction
for unsafe languages" + "separate compilation"

[Beyond Good and Evil, Juglaret, Hritcu, et al, CSF'16]

Protecting higher-level abstractions

- **Low*: enforcing specifications using micro-policies**
  - some can be turned into **contracts,** checked dynamically
  - fully abstract Low* to C compiler **trivial for C interfaces** (because F* allows and tracks effects, as opposed to Coq)
- **Limits of purely-dynamic enforcement**
  - functional purity, termination, relational reasoning
  - **push these limits further and combine with static analysis**



SECOMP focused on dynamic enforcement **but combining with static analysis can ...**

- **improve efficiency**
  - **removing spurious checks**
  - e.g. turn off pointer checking for a statically memory safe component that never sends or receives pointers
- **improve transparency**
  - **allowing more safe behaviors**
  - e.g. statically detect which copy of linear return capability the code will use to return
  - in this case **unsound static analysis is fine**

Beyond full abstraction

- Is full abstraction the right notion of secure compilation? Is full abstraction the right attacker model?
- **Variants / similar properties**
  – secure compartmentalizing compilation (SCC)
  – preservation of all hyper-safety properties [Garg et al.]
- **Strictly weaker properties** (easier to enforce!):
  – preservation of particular hyper-safety properties
  – robust compilation (some integrity but no confidentiality)
- **Orthogonal properties**:
  – memory safety (e.g. enforcing CompCert memory model)

What secure compilation adds over compositional compiler correctness

- **mapping back arbitrary low-level contexts**
- **preserving integrity properties**
  - robust compilation achieves some of this
- **preserving confidentiality properties**
  - full abstraction and preservation of hyper-safety phrased in terms of this
- **stronger notion of components and interfaces**
  - secure compartmentalizing compilation adds this

Verification and testing

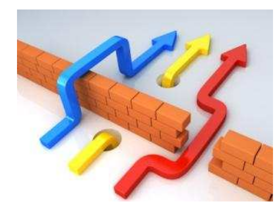
- So far all secure compilation work **on paper**
  - but one can't verify an interesting compiler on paper
- SECOMP will use **proof assistants**: Coq and F*
- **Reduce effort**
  - better automation (e.g. based on SMT, like in F*)
  - integrate testing and proving (QuickChick and Luck)
- **Problems not just with effort/scale**
  - devising good **proof techniques** for full abstraction is a hot research topic of it's own

Micro-policies:
remaining fundamental challenges

- **Micro-policies for C**
  - needed for vertical compiler composition
  - will put micro-policies in the hands of programmers
- **Secure micro-policy composition**
  - micro-policies are **interferent** reference monitors
  - one micro-policy's behavior can break another's guarantees
    - e.g. composing anything with IFC can leak

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 and Low*)
- **Answering challenging fundamental questions**



  - attacker models, proof techniques
  - secure composition, micro-policies for C
- **Achieving strong security properties like full abstraction**
  + testing and proving formally that this is the case
- **Measuring & lowering the cost of secure compilation**

- Most of this is **vaporware** at this point but ...

  - building a community, looking for collaborators, and hiring
    ... **in order to try to make some of this real**

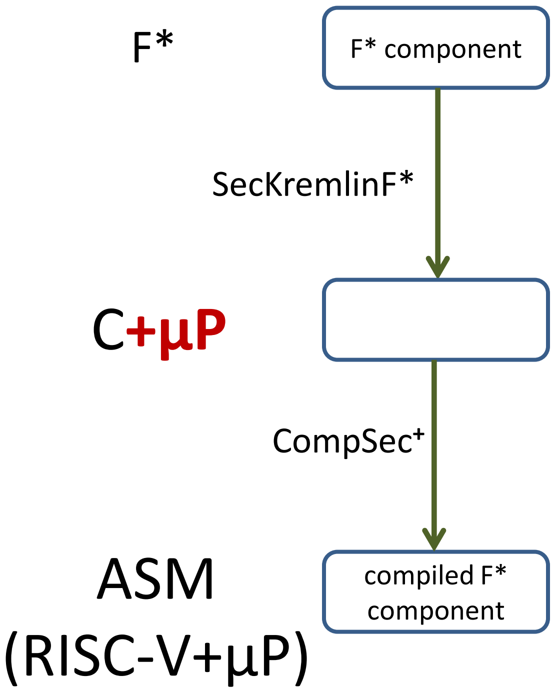
- Looking for excellent **interns**, **PhD students**, **PostDocs**, **starting researchers**, and **engineers**
- We can also support outstanding candidates in the **Inria permanent researcher competition**

Collaborators & Community

- **Traditional collaborators from Micro-Policies project**
  - UPenn, MIT, Portland State, Draper Labs
- **Several other researchers working on secure compilation**
  - Deepak Garg (MPI-SWS), Frank Piessens (KU Leuven), Amal Ahmed (Northeastern), Cedric Fournet & Nik Swamy (MSR)
- **Secure compilation meetings (informal)**
  - 1st at Inria Paris in August 2016
  - 2nd in Paris on 15 January 2017 before POPL at UPMC
  - Proposal for Dagstuhl seminar for 2018
  - **build larger research community, identify open problems, bring together communities** (hardware, systems, security, languages, verification, ...)

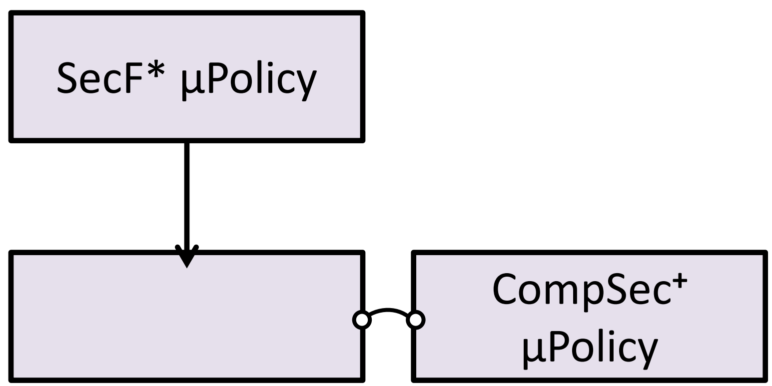
BACKUP SLIDES

Composing compilers and higher-level micro-policies



To compose compilers need

1. **higher-level micro-policies**
2. **composing micro-policies**



User-specified higher-level policies

- By composing more micro-policies we can allow **user-specified micro-policies for C**
- Good news:
  micro-policy composition is easy since tags can be tuples
- But how do we ensure programmers won't break security?
- Bad news: **secure micro-policy composition is hard!**