SECOMP

Efficient Formally Secure Compilers to a Tagged Architecture

Cătălin Hrițcu

Inria Paris

(currently on leave at MSR Redmond)

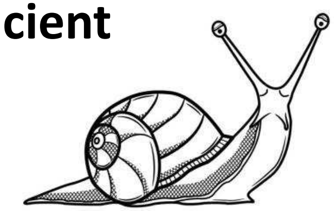
SECOMP

- **grant recently funded by European Research Council (ERC)**
  - "most prestigious" individual research grants in Europe
- **5 year research project at Inria Paris** (2017-2021)
  - hiring: 3 PhD students, 2 PostDocs, 1 Starting Researcher

- **new people starting officially in January 2017:**
  - Marco Stronati (PostDoc, working on privacy at Cornell Tech NY)
  - Guglielmo Fachini (Research Engineer, “pre-PhD intern”)
- **more collaborators and community building**
  - visits, sabbaticals, new secure compilation workshop, etc.
- **project builds mainly on Micro-Policies and Yannis' work**
  - Yannis left Inria end of September, very unfortunate for us

The problem: devastating low-level attacks

- **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
- **Today's languages & compilers plagued by low-level attacks**
  - **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

- 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

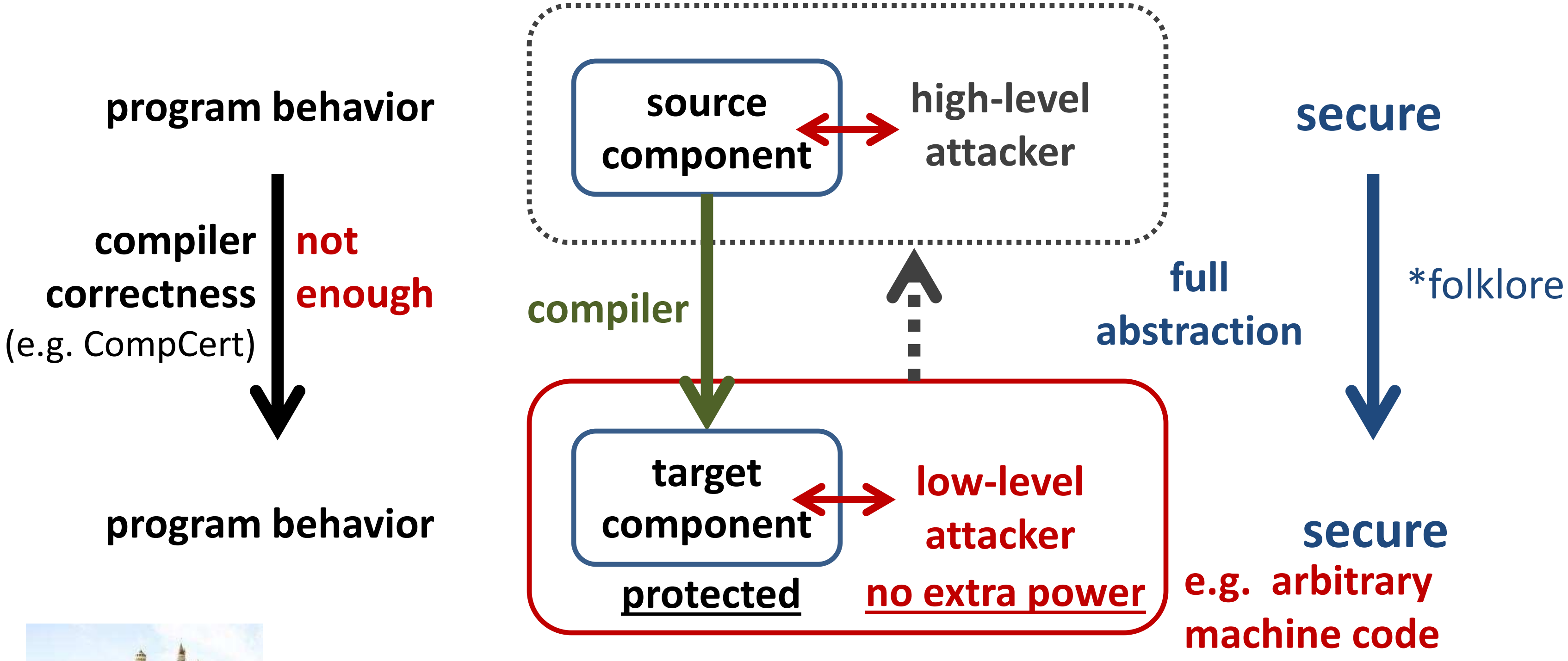
SECOMP grand challenge

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 F* [= ML + 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)

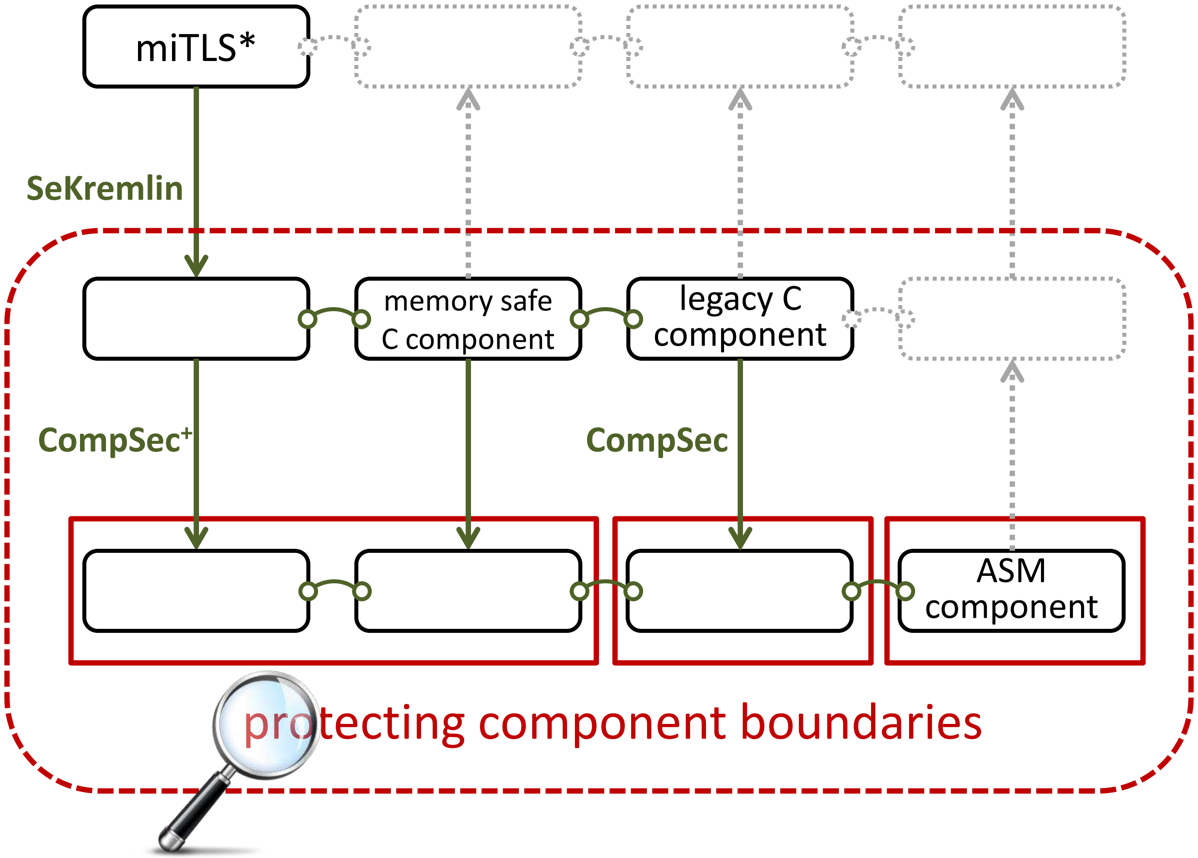
forget that libraries are written in a lower-level language

SECOMP: achieving full abstraction at scale

F* language
(ML + verification)

C language
+ memory safety
+ components

ASM language
(RISC-V + micro-policies)



miTLS*

SeKremlin

memory safe C component

legacy C component

CompSec$^+$

CompSec

ASM component

protecting component boundaries

Protecting component boundaries

- Break up software into **mutually distrustful components** running with **minimal privileges** & interacting only via **well-defined interfaces**
- **Limit the damage** of control hijacking attacks to just the C or ASM components where they occur
- Not a new idea, already deployed in practice:
  - process-level privilege separation
  - software-fault isolation

chrome

SSH... OpenSSH KEEPING YOUR COMMUNIQUÉS SECRET

TOP SECRET

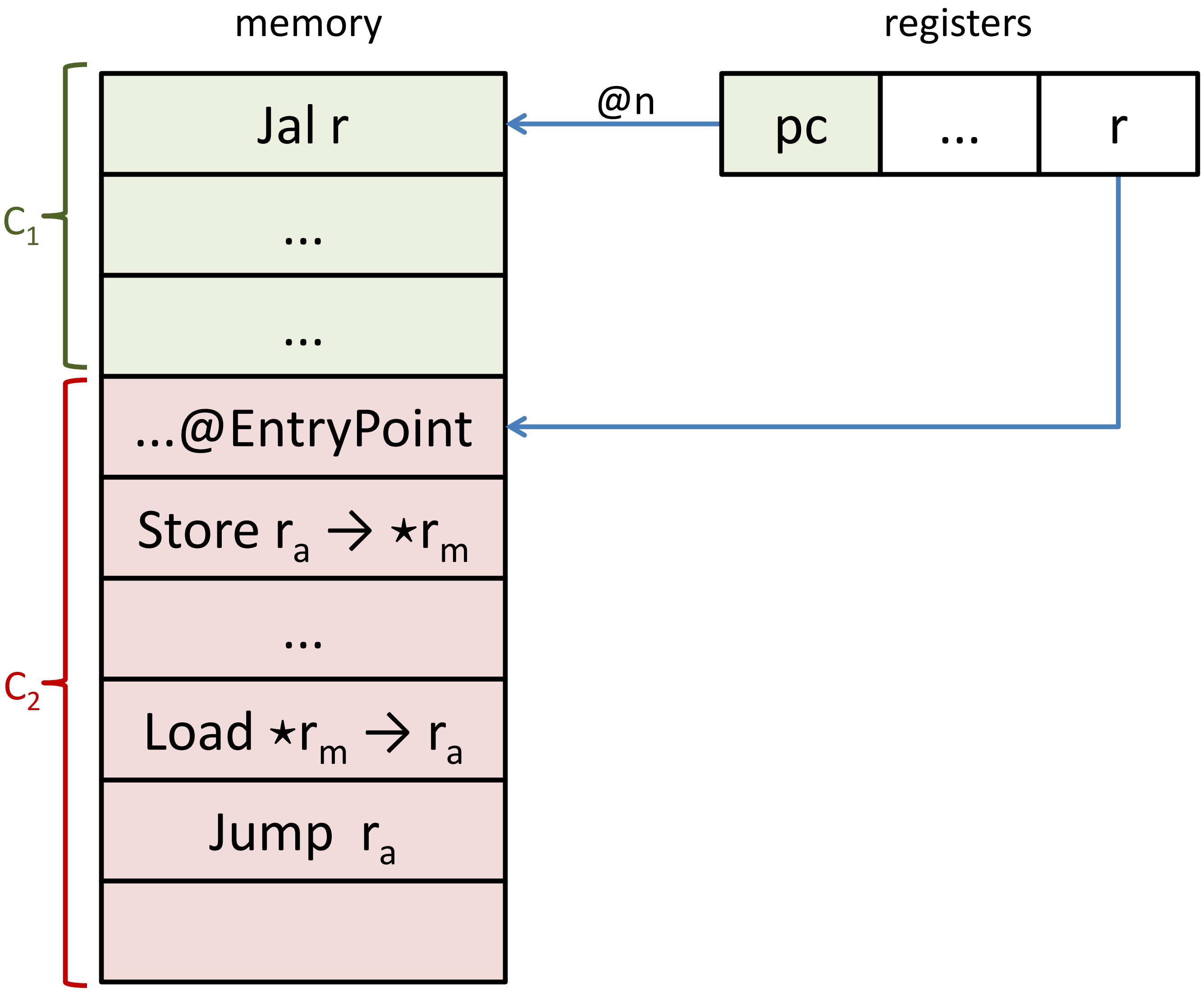
- Micro-policies can give us **better interaction model**
- We also aim to **show security formally**

Towards compartmentalized C

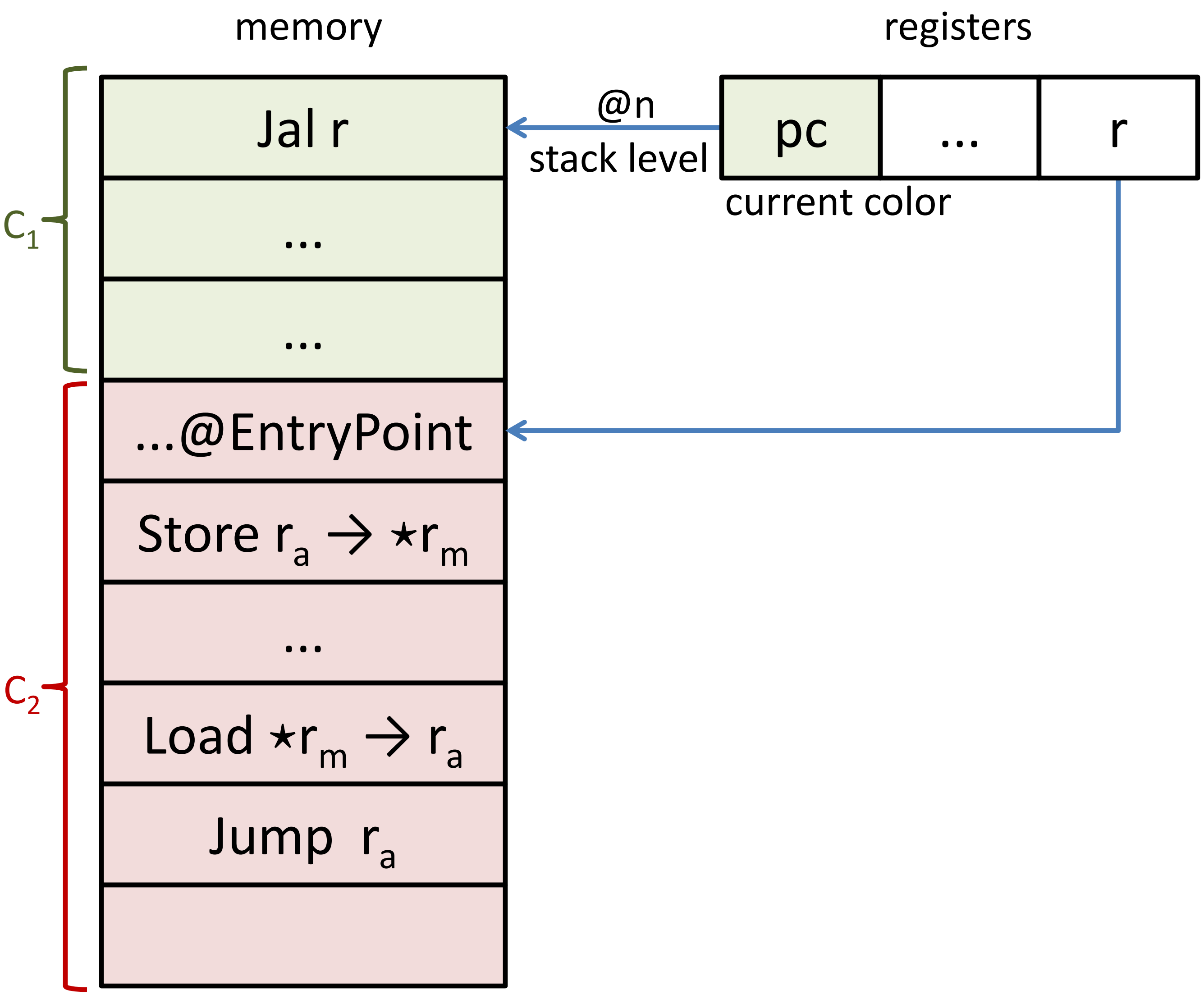
- Want to **add components with typed interfaces to C**
- Compiler (e.g. CompCert), linker, loader propagate interface information to low-level memory tags
  - each component's memory tagged with unique color
  - procedure entry points tagged with procedure's type
- Micro-policy enforcing:
  - **component isolation**
  - **procedure call discipline** (entry points)
  - **stack discipline for returns** (linear return capabilities)
  - **type safety** on cross-component interaction

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

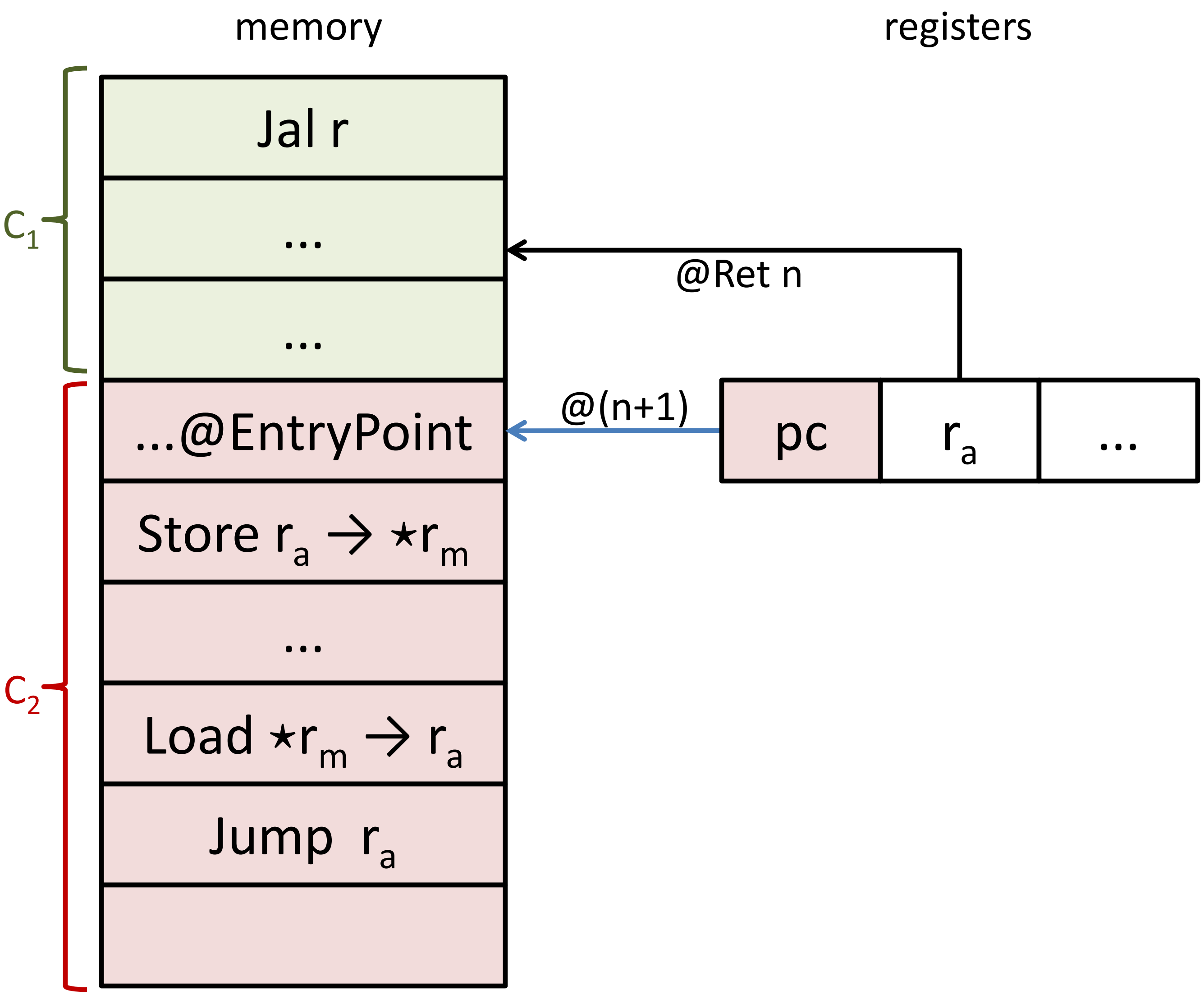
Compartmentalization micro-policy



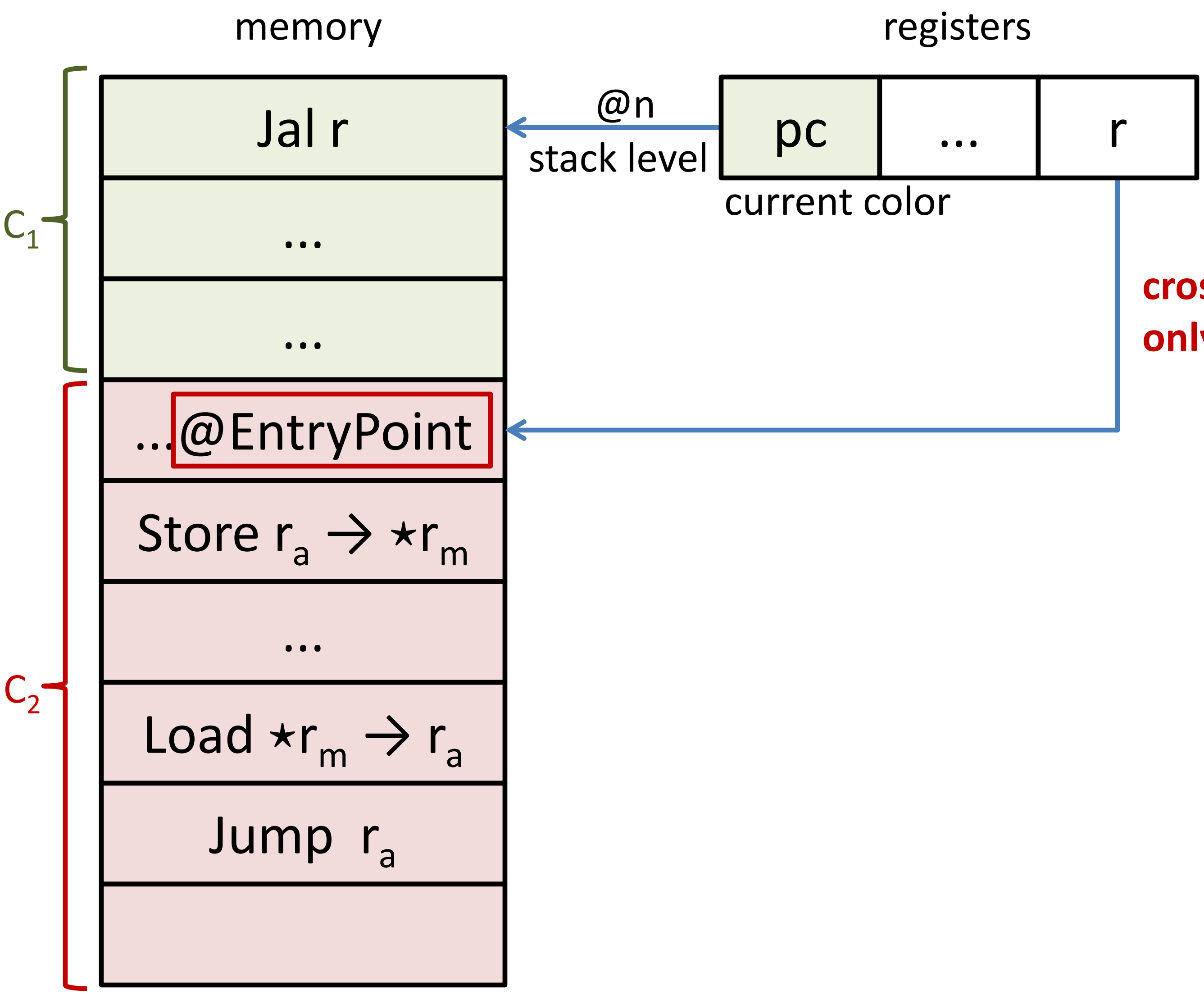
Compartmentalization micro-policy



Compartmentalization micro-policy

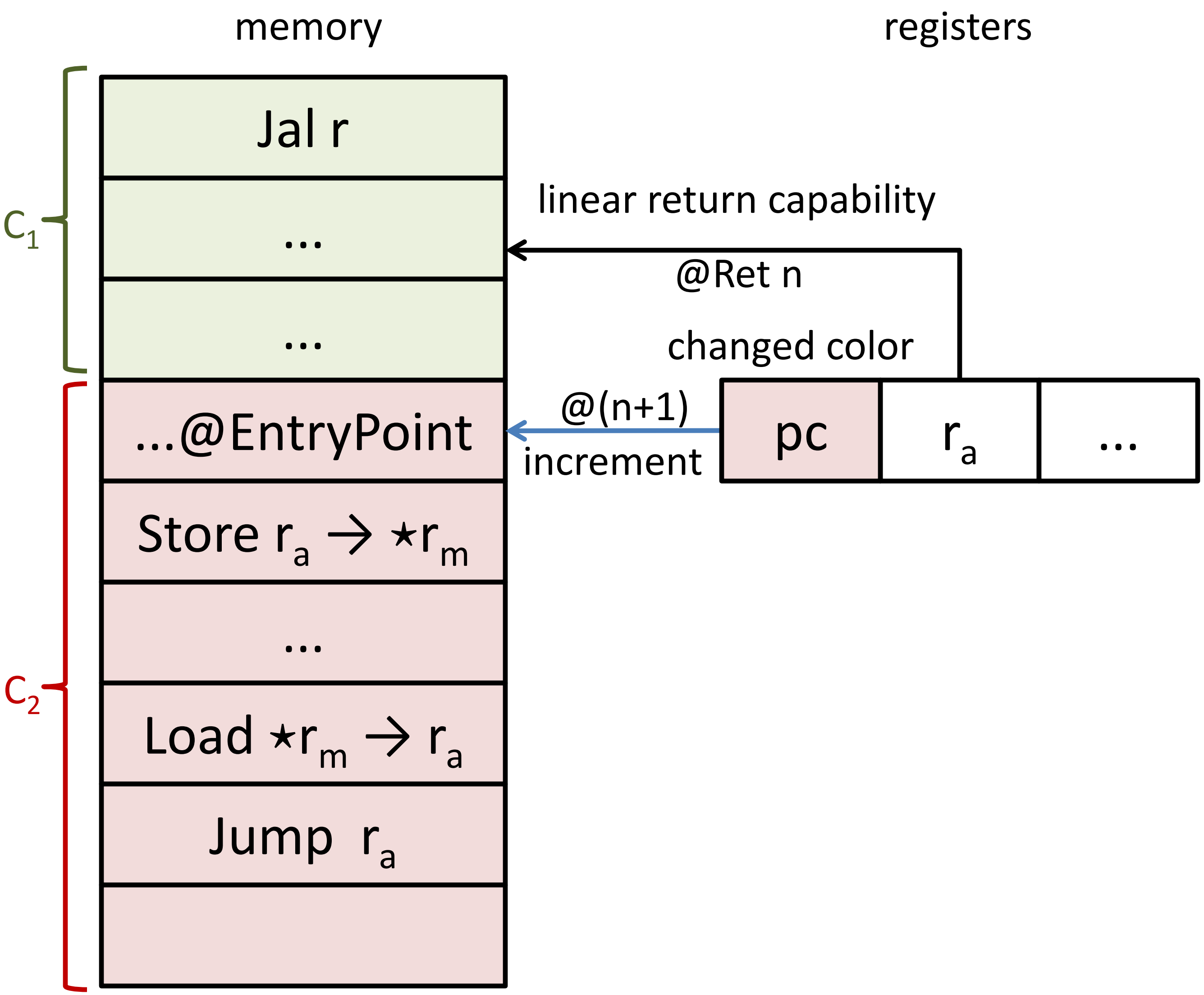


Compartmentalization micro-policy

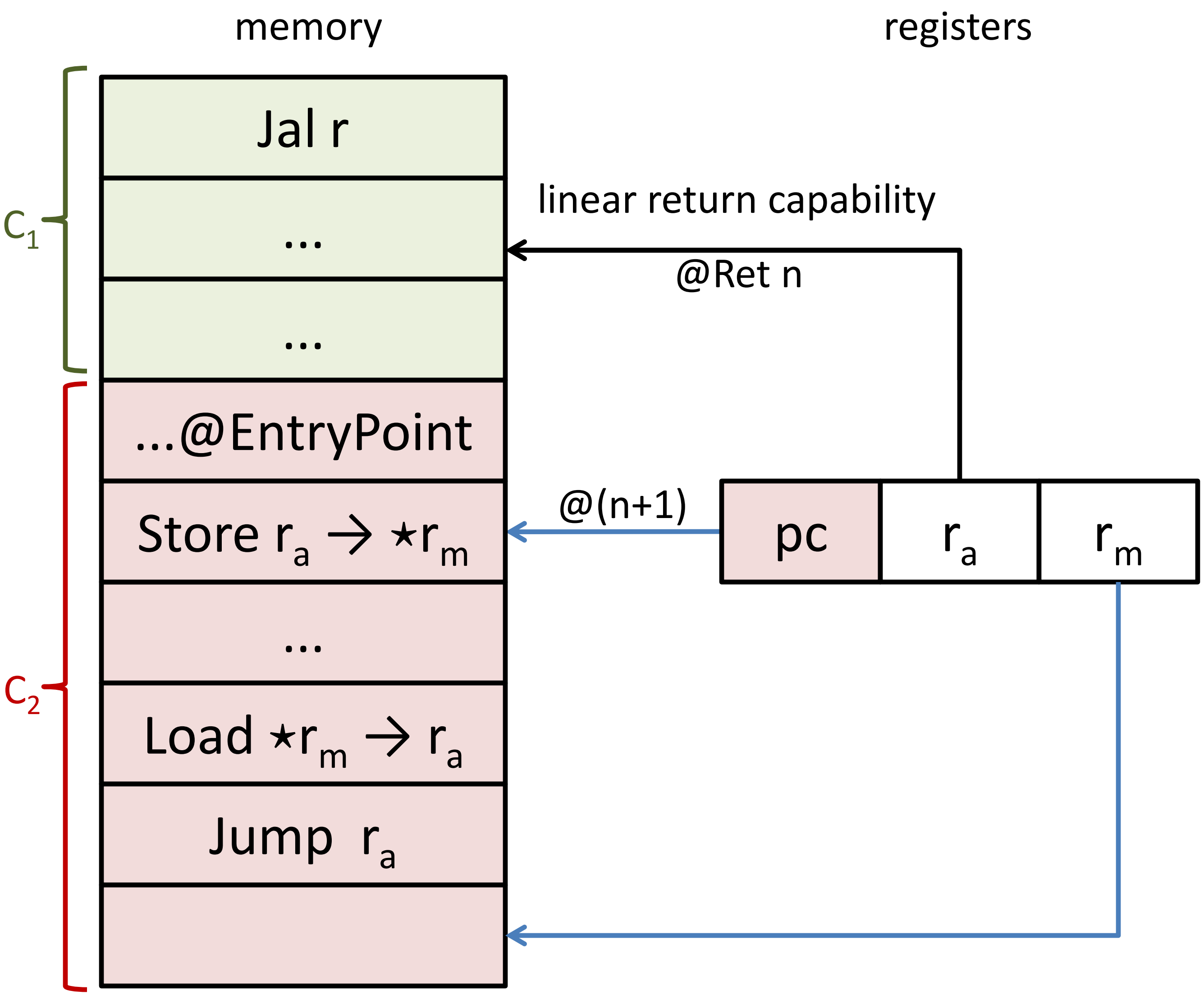


cross-component call only allowed at EntryPoint

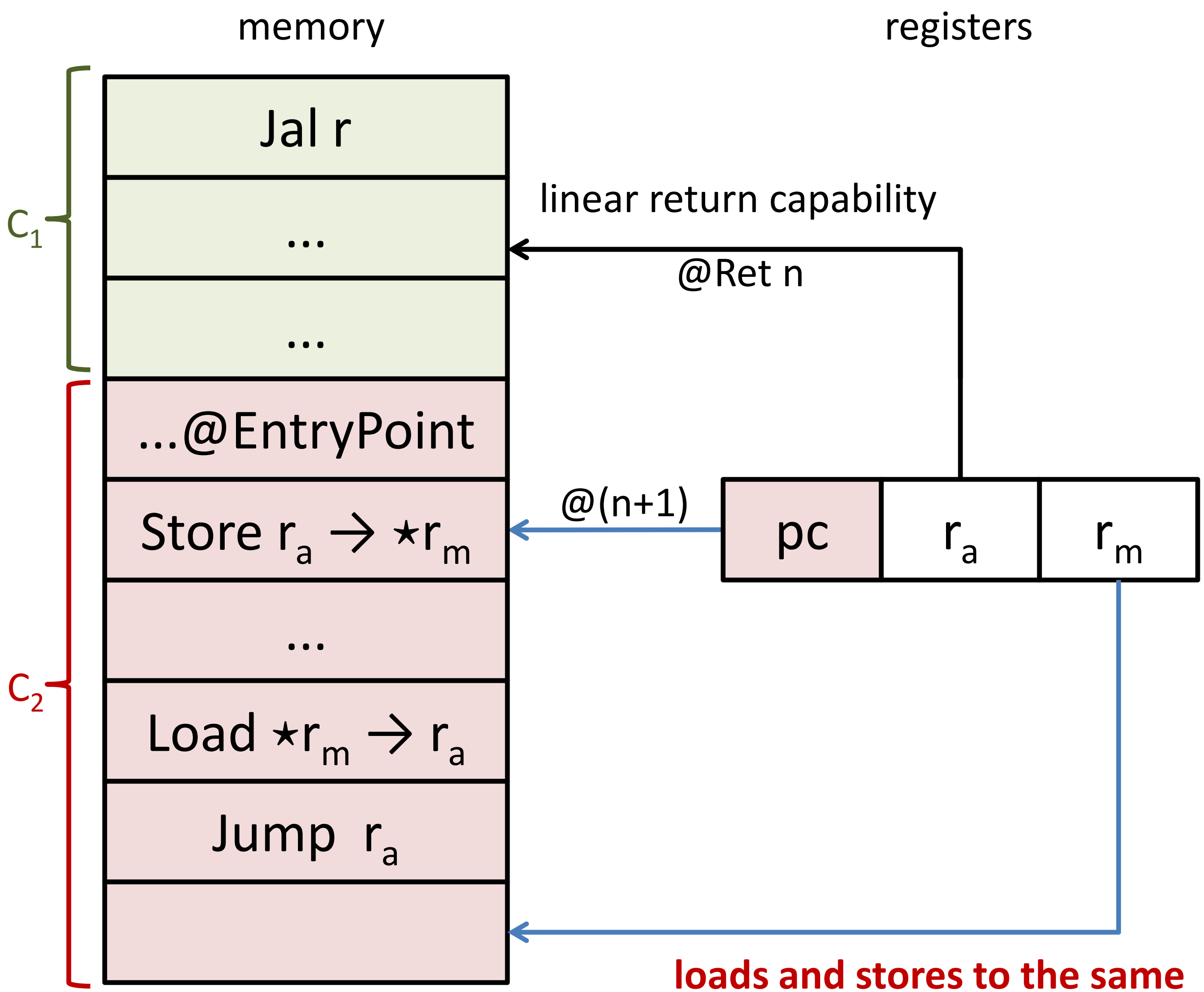
Compartmentalization micro-policy



Compartmentalization micro-policy

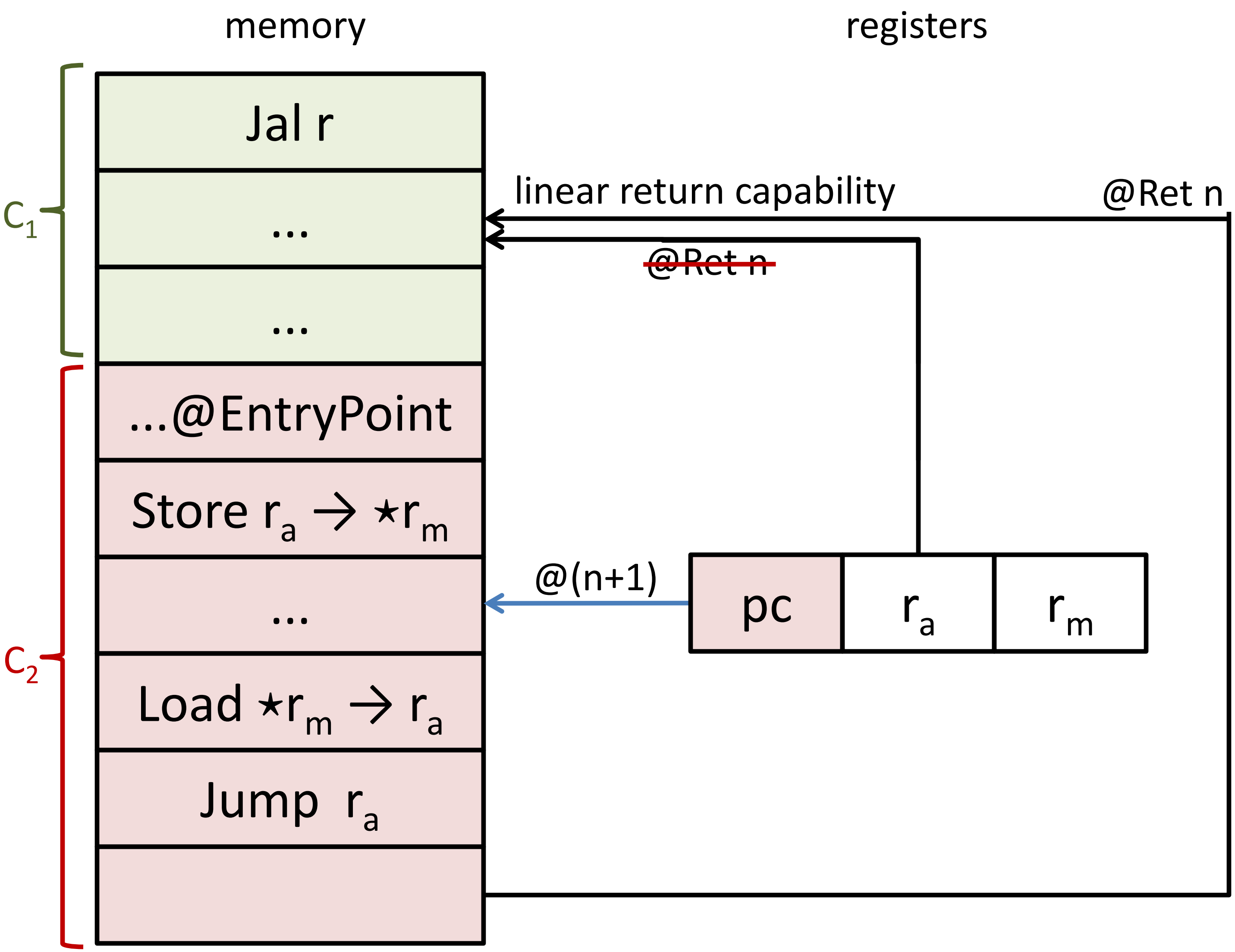


Compartmentalization micro-policy

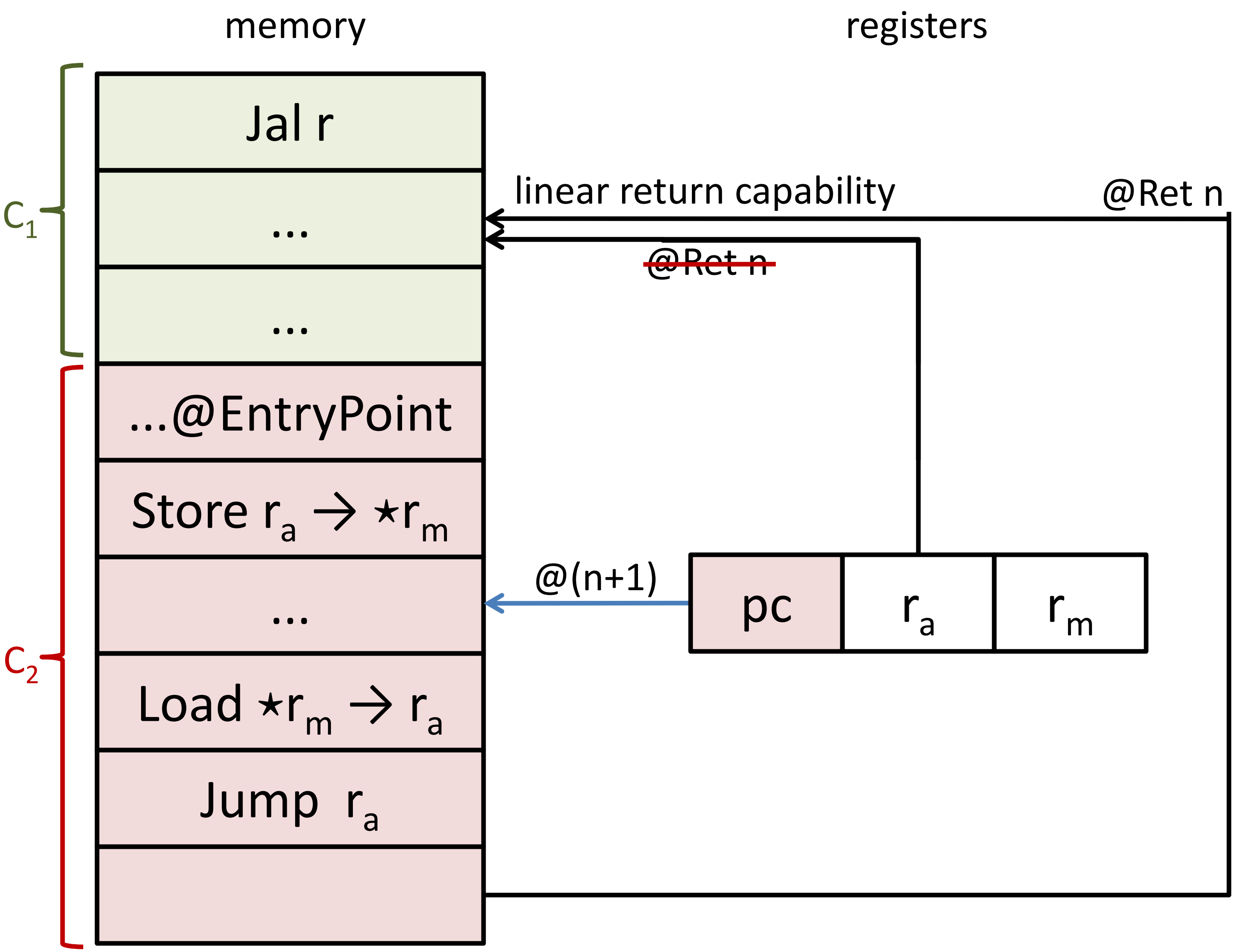


loads and stores to the same component always allowed

Compartmentalization micro-policy

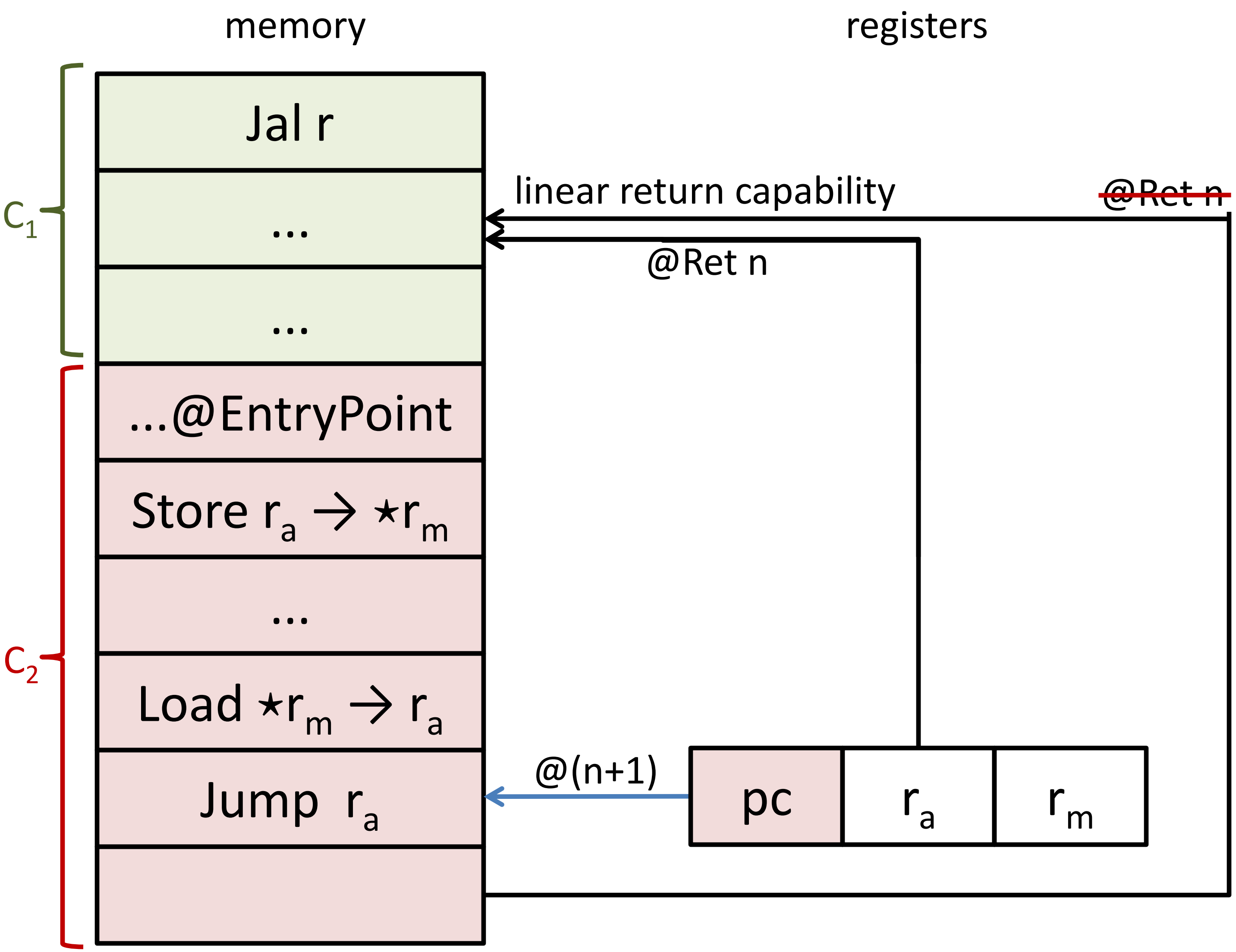


Compartmentalization micro-policy



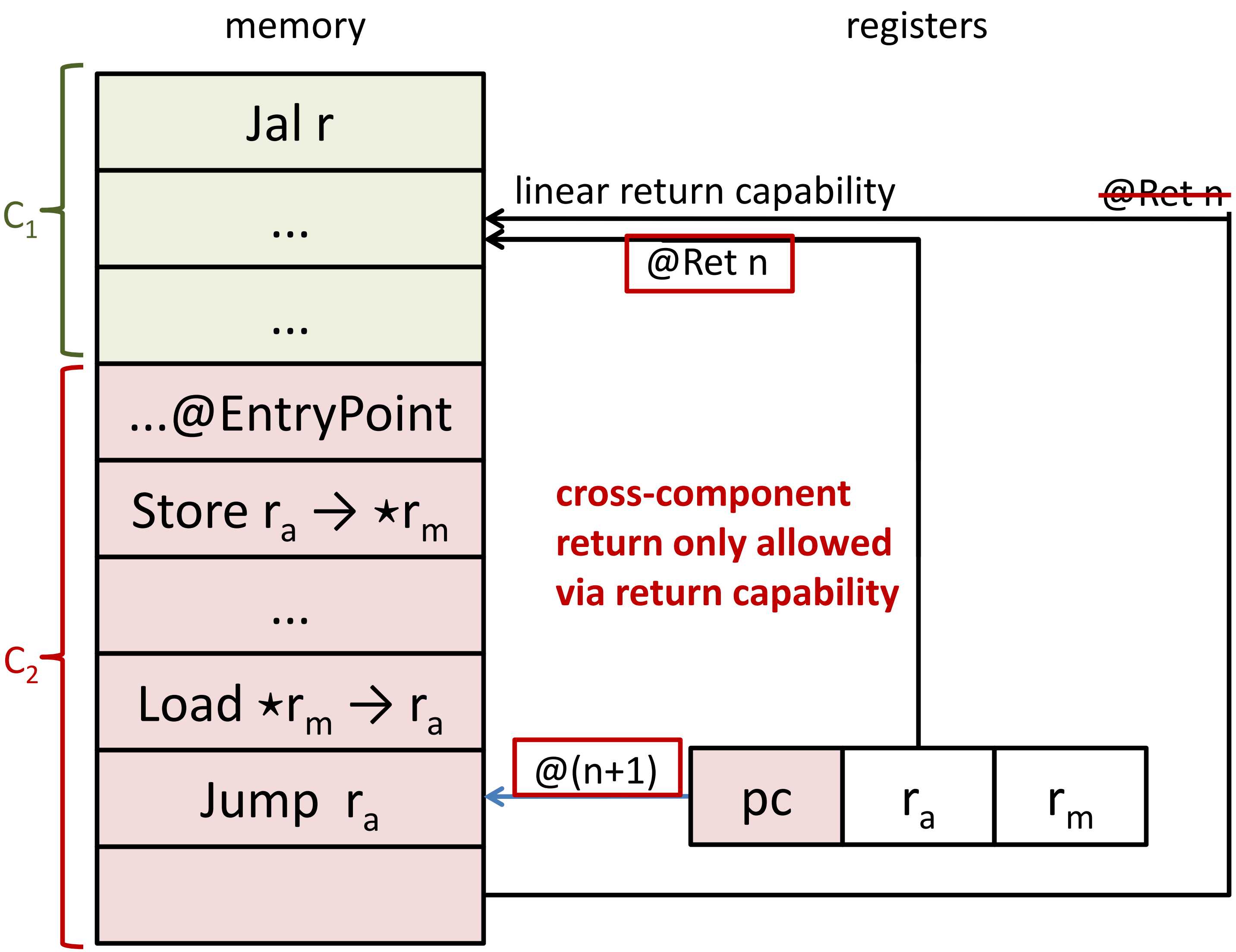
invariant:
at most one
return capability
per call stack level

Compartmentalization micro-policy



invariant:
at most one
return capability
per call stack level

Compartmentalization micro-policy



cross-component return only allowed via return capability

invariant:
at most one return capability per call stack level

Compartmentalization micro-policy?

	Yannis et al, TR 2015 and later	Antal et al, Oakland 2014 (SFI inspired)
abstraction level (source)	**core C (and core Java)**	machine code
compilation target	**simple RISC machine code**	
compartment lifetime	static	**dynamic compartment creation**
target property / attacker model	**full abstraction variant [CSF 2016]**	correct isolation (could be extended though)
mutual distrust justified by	**unsafe source (C) + linking with unsafe / malicious machine code**	interacting with unsafe / malicious machine code
enforced interaction model	**valid calls and returns (µP); register cleaning and restoring; typed arguments and results (µP)**	cross-compartment jumps only to designated entry points
memory protection	no cross-compartments writes or reads	**sets of allowed cross-compartments reads and writes**

Open problems on compartmentalization (1)

- **Dealing with more of C ... towards setjmp/longjmp ☺**
  - Yannis had **compartment-local stacks**
  - vs C: shared stack for all compartments, trickier
- **More hardware support could help**
  - **shared stack (+memory safety) require** setting tags on large regions of memory; in software (slow) or hardware (discussed)
  - **linear return capabilities** require PUMP inputs also be outputs
  - **cleaning registers** assumes compiler introduced instructions (restore) or some hardware support (all registers both inputs and outputs seems crazy, right?)

Open problems on compartmentalization (2)

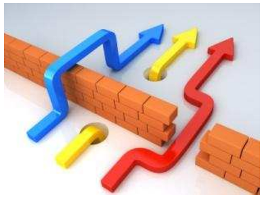
- **Passing pointers between compartments**
  - currently can only allow immutable capabilities
    - e.g. code pointers as call capabilities
    - e.g. read/write capabilities to individual memory cells
  - capability is lost if pointer is changed
  - combining compartmentalization with memory safety allows richer object capability model
- **Linear return capabilities not transparent**
  - use wrapper or static analysis to gain transparency

Micro-policies:
remaining fundamental challenges

- **Micro-policies for C**
  - needed for vertical compiler composition
  - rule-based DSL for monitoring C programs
  - 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 F*)
- **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**

BACKUP SLIDES

SECOMP focused on dynamic enforcement **but static analysis could help too**

- **Improving efficiency**
  - removing spurious checks
  - just that by using micro-policies our compilers add few explicit checks
  - e.g. turn off memory safety checking for a statically memory safe component that never sends or receives pointers
- **Improving transparency**
  - allowing more safe behaviors
  - e.g. we could statically detect which copy of the linear return capability the code will use to return (in this case static analysis untrusted)

Beyond full abstraction

- Is full abstraction always the right notion of secure compilation? The right attacker model?
- **Similar properties**
  - secure compartmentalizing compilation (SCC)
  - preservation of hyper-safety properties [Garg et al.]
- **Strictly weaker properties** (easier to enforce!):
  - robust compilation (integrity but no confidentiality)
- **Orthogonal properties**:
  - memory safety (enforcing CompCert memory model)

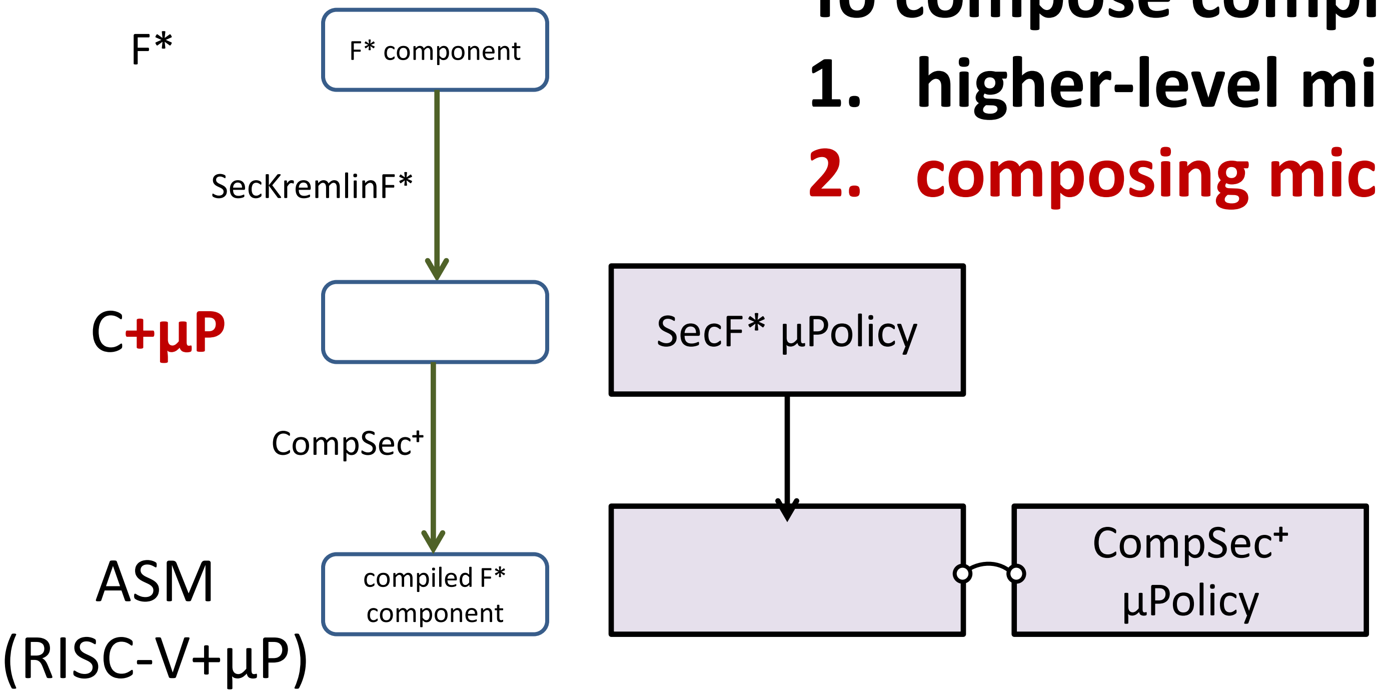
Collaborators & Community

- **Current collaborators from Micro-Policies project**
  - UPenn, MIT, Portland State, Draper Labs
- **Looking for additional collaborators**
  - 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 (very informal)**
  - 1st at INRIA Paris in August 2016
  - 2nd in Paris on 15 January 2017 before POPL at UPMC
  - **build larger research community, identify open problems, bring together communities** (hardware, systems, security, languages, verification, ...)

Composing compilers and higher-level micro-policies

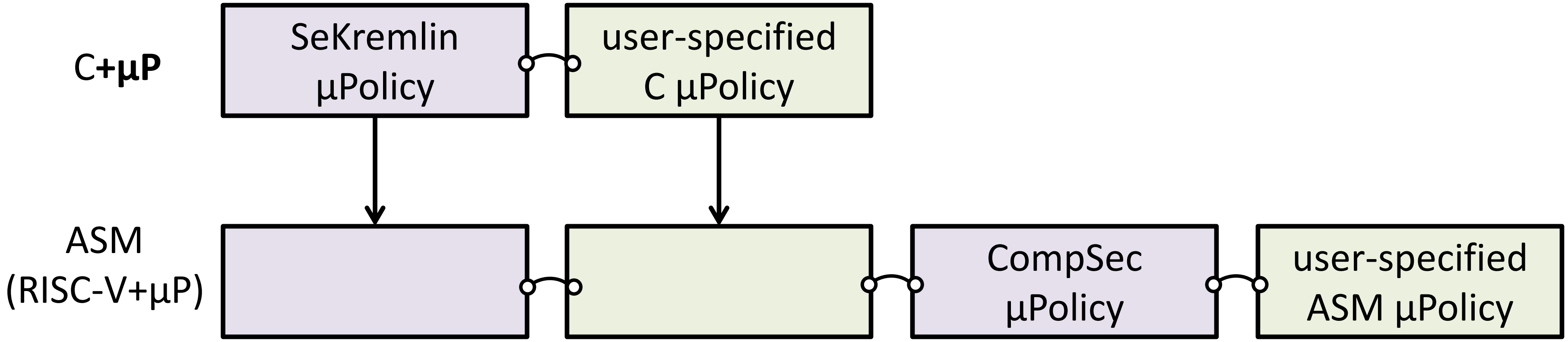
To compose compilers need

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!**


C+μP
SeKremlin μPolicy
user-specified C μPolicy
ASM (RISC-V+μP)
CompSec μPolicy
user-specified ASM μPolicy