SECOMP

Efficient Formally Secure Compilers to a Tagged Architecture

Cătălin Hrițcu

Prosecco team

SECOMP

Efficient Formally Secure Compilers to a Tagged Architecture

Cătălin Hrițcu

Prosecco team

5 year vision

SECOMP

Efficient Formally Secure Compilers to a Tagged Architecture

Cătălin Hrițcu

Prosecco team

5 year vision

European Research Council

new grant

Computers are insecure

- **devastating low-level vulnerabilities**

Computers are insecure

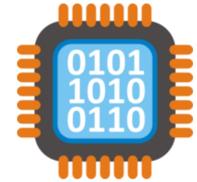
- **devastating low-level vulnerabilities**
- **programming languages, compilers, and hardware architectures**
  - designed in an era of scarce hardware resources
  - too often trade off security for efficiency

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

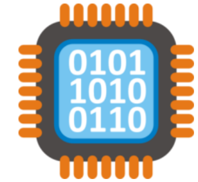
* "...the number of UNIX installations has grown to 10, with more expected..."
-- *Dennis Ritchie and Ken Thompson, June 1972*

Hardware architectures



- **Today’s processors are mindless bureaucrats**
  - “write past the end of this buffer” ... *yes boss!*
  - “jump to this untrusted integer” ... *right boss!*
  - “return into the middle of this instruction” ... *sure boss!*

Hardware architectures



- **Today's processors are mindless bureaucrats**
  - "write past the end of this buffer" ... *yes boss!*
  - "jump to this untrusted integer" ... *right boss!*
  - "return into the middle of this instruction" ... *sure boss!*
- **Software bears most of the burden for security**

Hardware architectures

- **Today's processors are mindless bureaucrats**
  - "write past the end of this buffer" ... *yes boss!*
  - "jump to this untrusted integer" ... *right boss!*
  - "return into the middle of this instruction" ... *sure boss!*
- **Software bears most of the burden for security**
- **Manufacturers have started looking for solutions**
  - 2015: Intel Memory Protection Extensions (MPX) and Intel Software Guard Extensions (SGX)
  - 2016: Oracle Silicon Secured Memory (SSM)

Hardware architectures

- **Today's processors are mindless bureaucrats**
  - "write past the end of this buffer" ... *yes boss!*
  - "jump to this untrusted integer" ... *right boss!*
  - "return into the middle of this instruction" ... *sure boss!*
- **Software bears most of the burden for security**
- **Manufacturers have started looking for solutions**
  - 2015: Intel Memory Protection Extensions (MPX) and Intel Software Guard Extensions (SGX)
  - 2016: Oracle Silicon Secured Memory (SSM)

"Spending silicon to improve security"

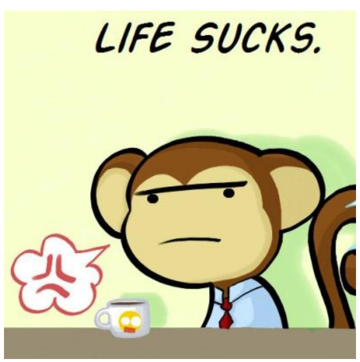
Unsafe low-level languages

- C (1972) and C++ **undefined behavior**
  - including buffer overflows, checks too expensive
  - compilers optimize aggressively assuming undefined behavior will simply not happen

Unsafe low-level languages

- C (1972) and C++ **undefined behavior**
  - including buffer overflows, checks too expensive
  - compilers optimize aggressively assuming undefined behavior will simply not happen
- **Programmers bear the burden for security**
  - just write secure code ... all of it

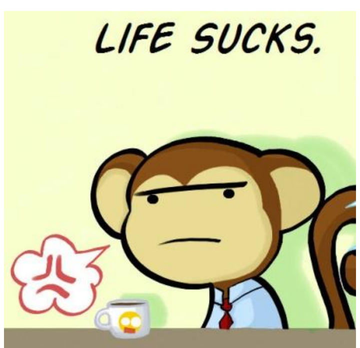
THE C PROGRAMMING LANGUAGE



Unsafe low-level languages

- C (1972) and C++ **undefined behavior**
  - including buffer overflows, checks too expensive
  - compilers optimize aggressively assuming undefined behavior will simply not happen
- **Programmers bear the burden for security**
  - just write secure code ... all of it

THE C PROGRAMMING LANGUAGE



[PATCH] CVE-2015-7547 --- glibc getaddrinfo() stack-based buffer overflow

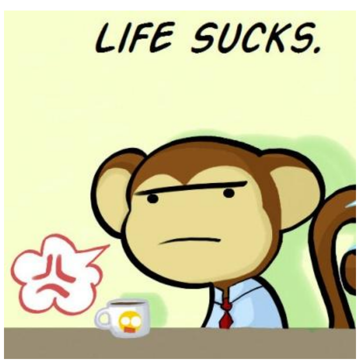
- *From*: "Carlos O'Donell" <carlos at redhat dot com>
- *To*: GNU C Library <libc-alpha at sourceware dot org>
- *Date*: Tue, 16 Feb 2016 09:09:52 -0500
- *Subject*: [PATCH] CVE-2015-7547 --- glibc getaddrinfo() stack-based buffer overflow
- *Authentication-results*: sourceware.org; auth=none
- *References*: <56C32C20 dot 1070006 at redhat dot com>

The glibc project thanks the Google Security Team and Red Hat for reporting the security impact of this issue, and Robert Holiday of Ciena for reporting the related bug 18665.

Unsafe low-level languages

- C (1972) and C++ **undefined behavior**
  - including buffer overflows, checks too expensive
  - compilers optimize aggressively assuming undefined behavior will simply not happen
- **Programmers bear the burden for security**
  - just write secure code ... all of it

THE C PROGRAMMING LANGUAGE



[PATCH] CVE-2015-7547 --- glibc getaddrinfo() stack-based buffer overflow

DNS queries

...ell" <carlos at redhat dot com>

vulnerable since May 2008

- *Date*: Tue, 16 Feb 2016
- *Subject*: [PATCH] CVE ... r overflow
- *Authentication-results*: sourceware.org; auth=none
- *References*: <56C32C20 dot 1070006 at redhat dot com>

```
The glibc project thanks the Google Security Team and Red Hat for
reporting the security impact of this issue, and Robert Holiday of Ciena
for reporting the related bug 18665.
```

Safer high-level languages

- **memory safe** (at a cost)

Safer high-level languages

- **memory safe** (at a cost)
- **useful abstractions** for writing secure code:
  - GC, type abstraction, modules, immutability, ...

Safer high-level languages

- **memory safe** (at a cost)
- **useful abstractions** for writing secure code:
  - GC, type abstraction, modules, immutability, ...
- **not immune to low-level attacks**
  - large runtime systems, in C++ for efficiency
  - **unsafe interoperability with low-level code**
    - libraries often have large parts written in C/C++
    - **enforcing abstractions all the way down too expensive**

Summary of the problem

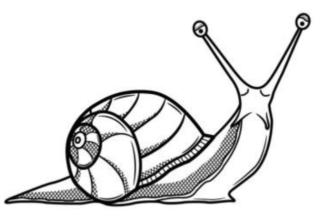
- **1. inherently insecure low-level languages**
  - **memory unsafe**: any buffer overflow can be catastrophic allowing remote attackers to gain complete control

Summary of the problem

- **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

Summary of the problem

- **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**
  - **main culprit: hardware provides no appropriate security mechanisms**
  - fixing this purely in software would be way **too inefficient**

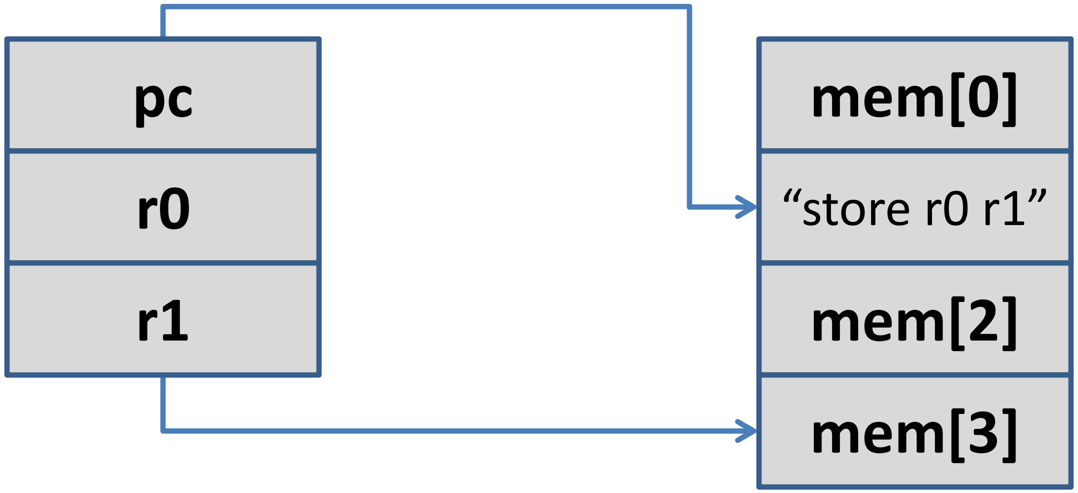


Key enabler: Micro-Policies

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

Key enabler: Micro-Policies

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



Key enabler: Micro-Policies

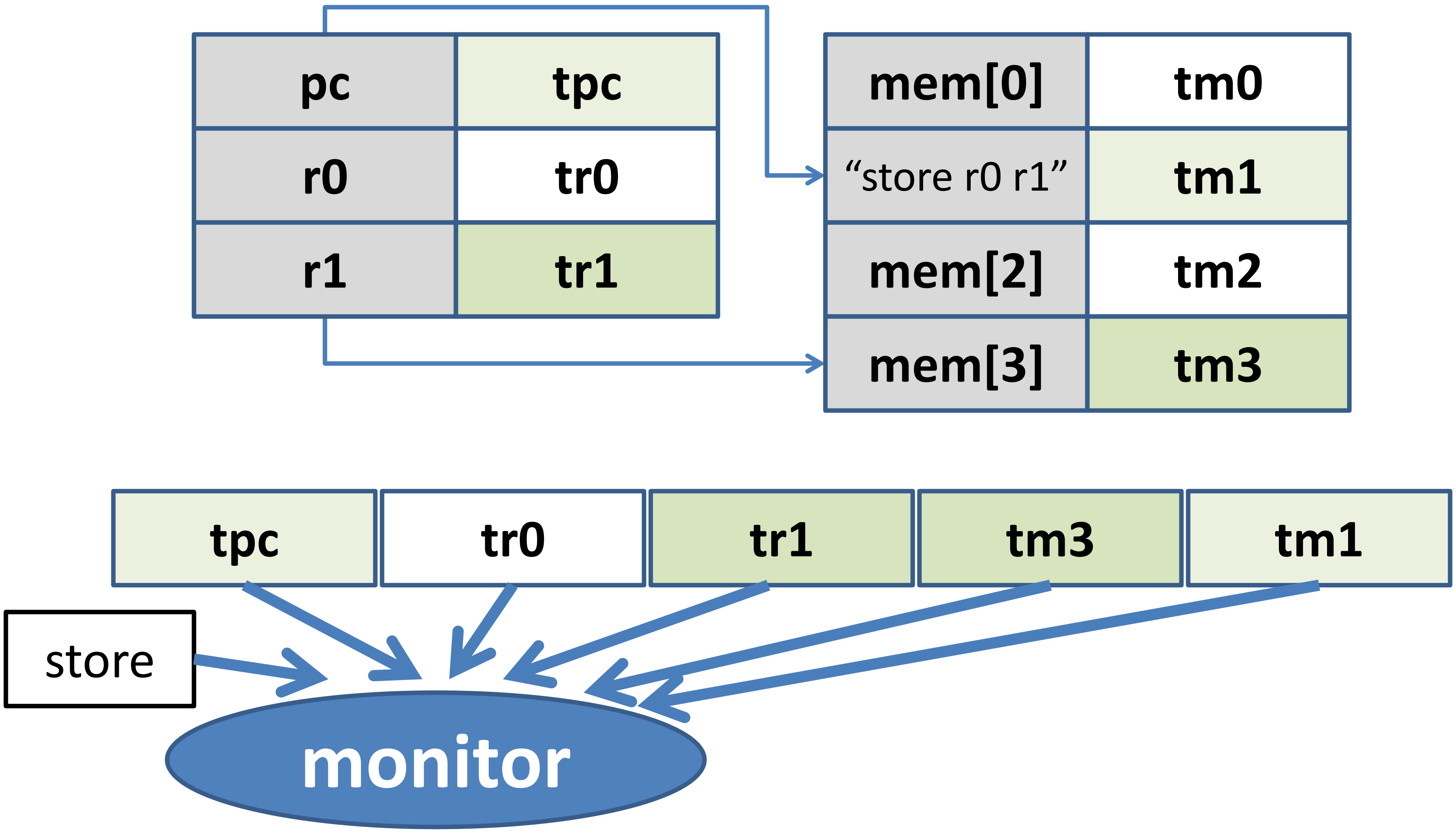
software-defined, hardware-accelerated, tag-based monitoring

pc	tpc
r0	tr0
r1	tr1

mem[0]	tm0
“store r0 r1”	tm1
mem[2]	tm2
mem[3]	tm3

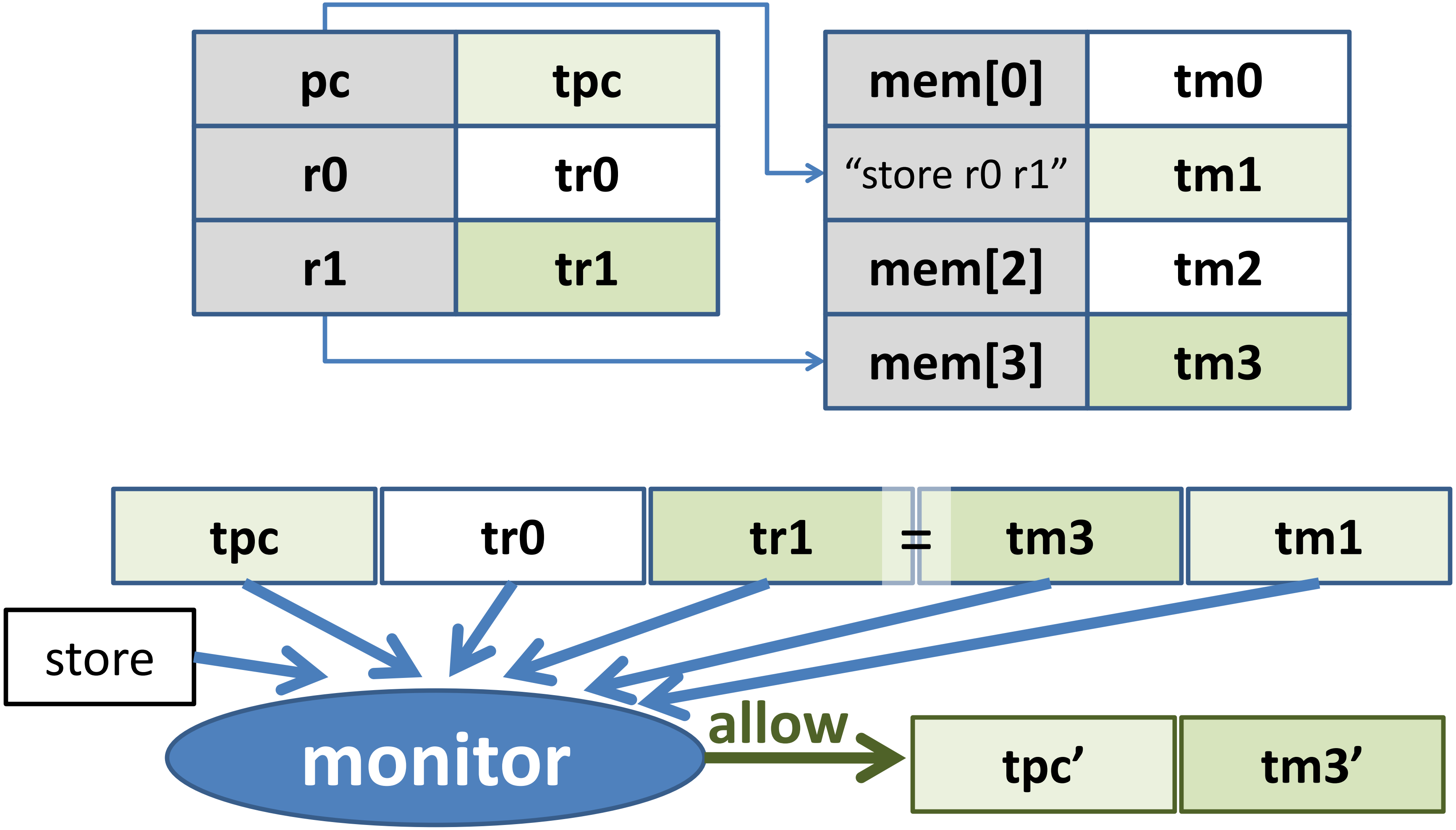
Key enabler: Micro-Policies

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



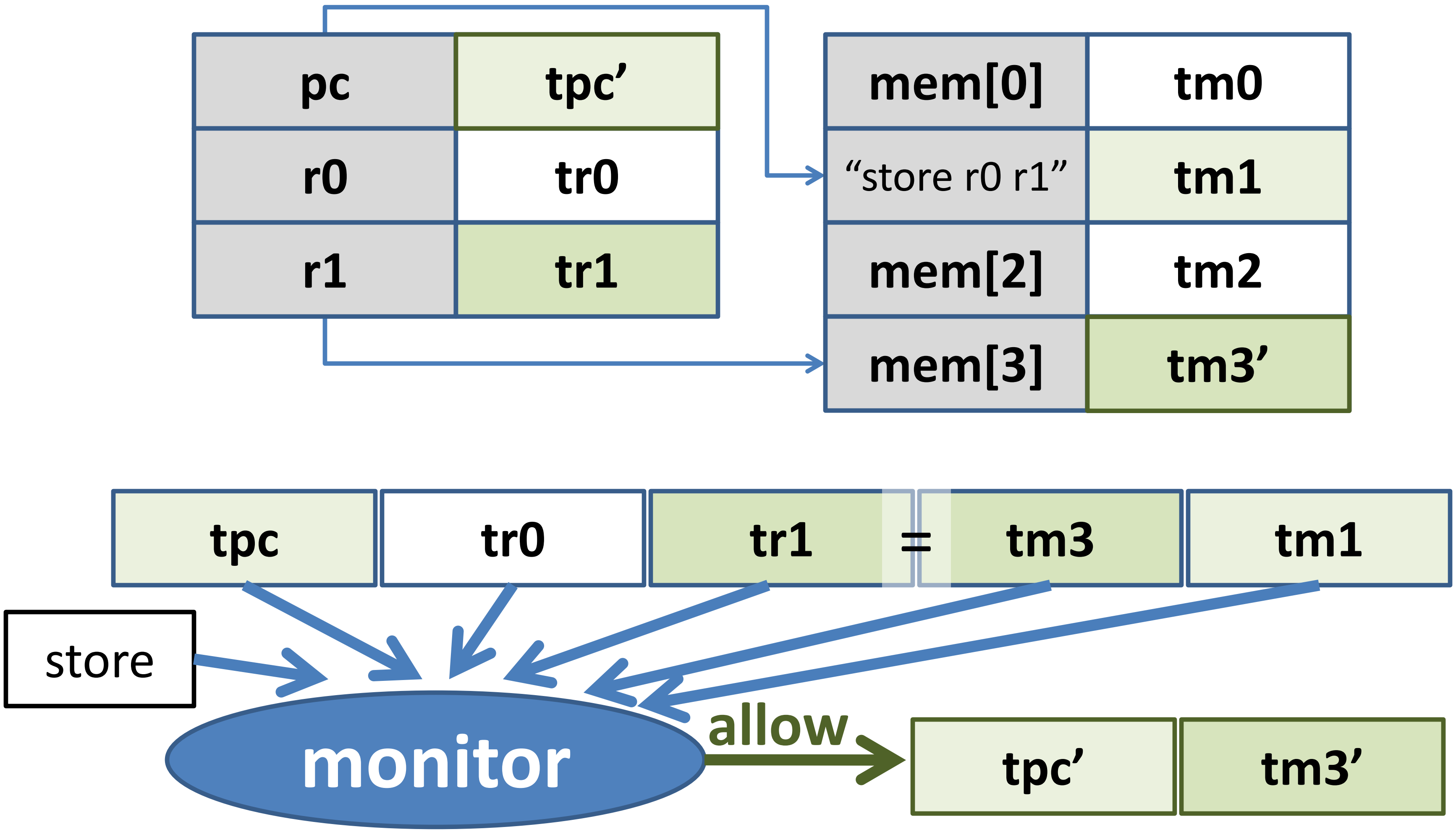
Key enabler: Micro-Policies

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



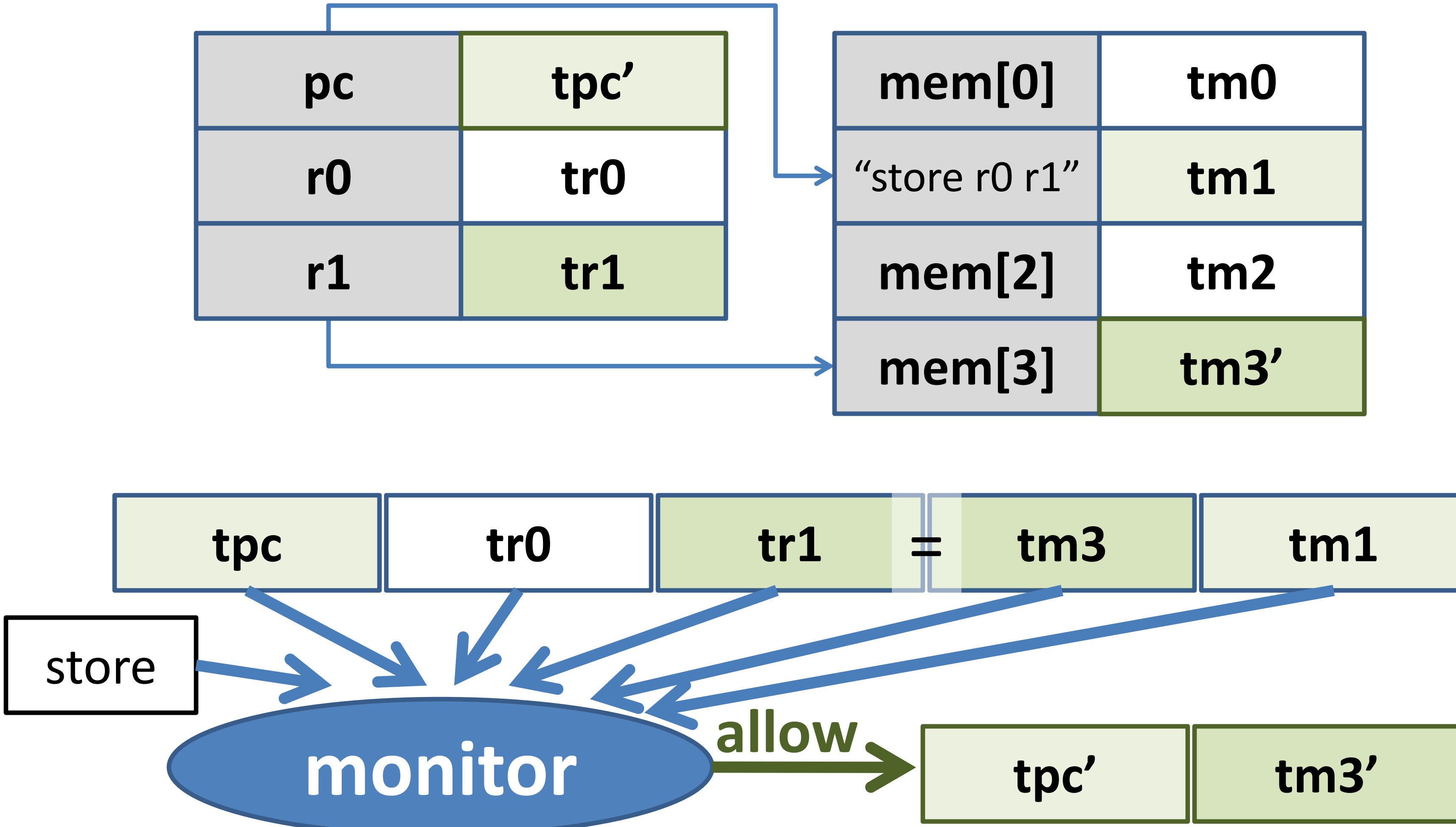
Key enabler: Micro-Policies

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



Key enabler: Micro-Policies

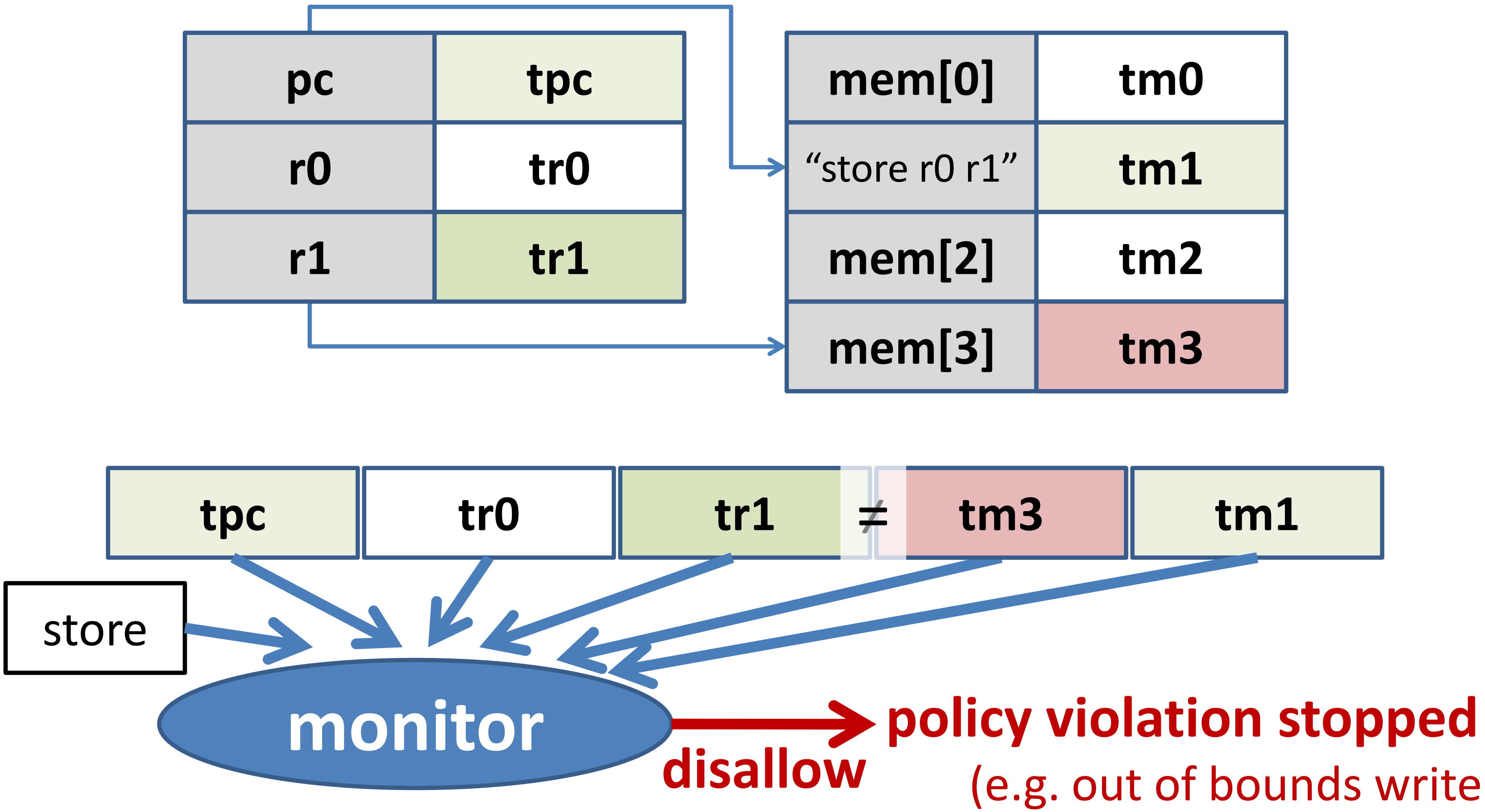
software-defined, hardware-accelerated, tag-based monitoring



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

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

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

DRAPER bluespec

Expressiveness

- information flow control (IFC) [POPL'14]

Expressiveness

- 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
- ...

Expressiveness

- 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]

Expressiveness

- 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]

SECOMP grand challenge

Use micro-policies to build **the first efficient formally secure compilers** for **realistic programming languages**

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

SECOMP grand challenge

Use micro-policies to build **the first efficient formally secure compilers** for **realistic programming languages**

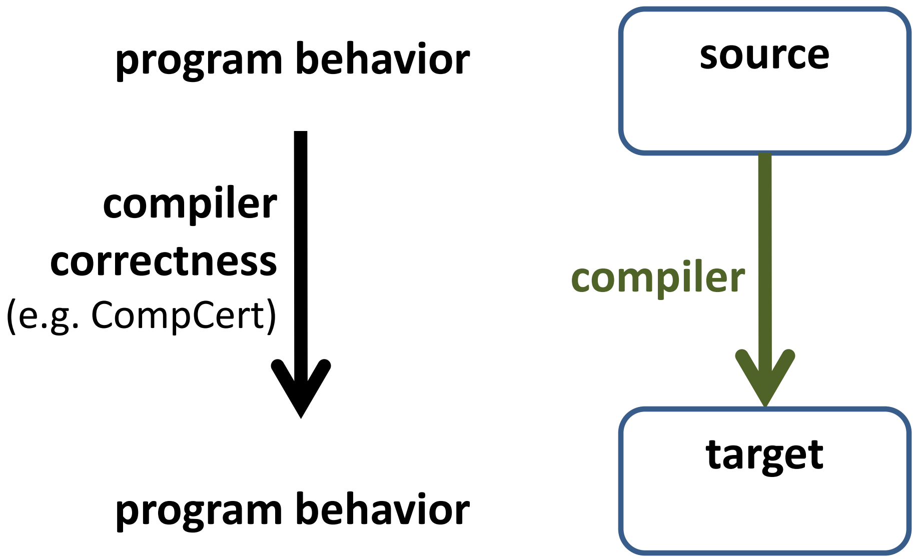
1. **Provide secure semantics for low-level languages**
   - C with protected components and memory safety
2. **Enforce secure interoperability with lower-level code**
   - ASM, C, and F* [F* = ML + verification]

Formally verify: full abstraction

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

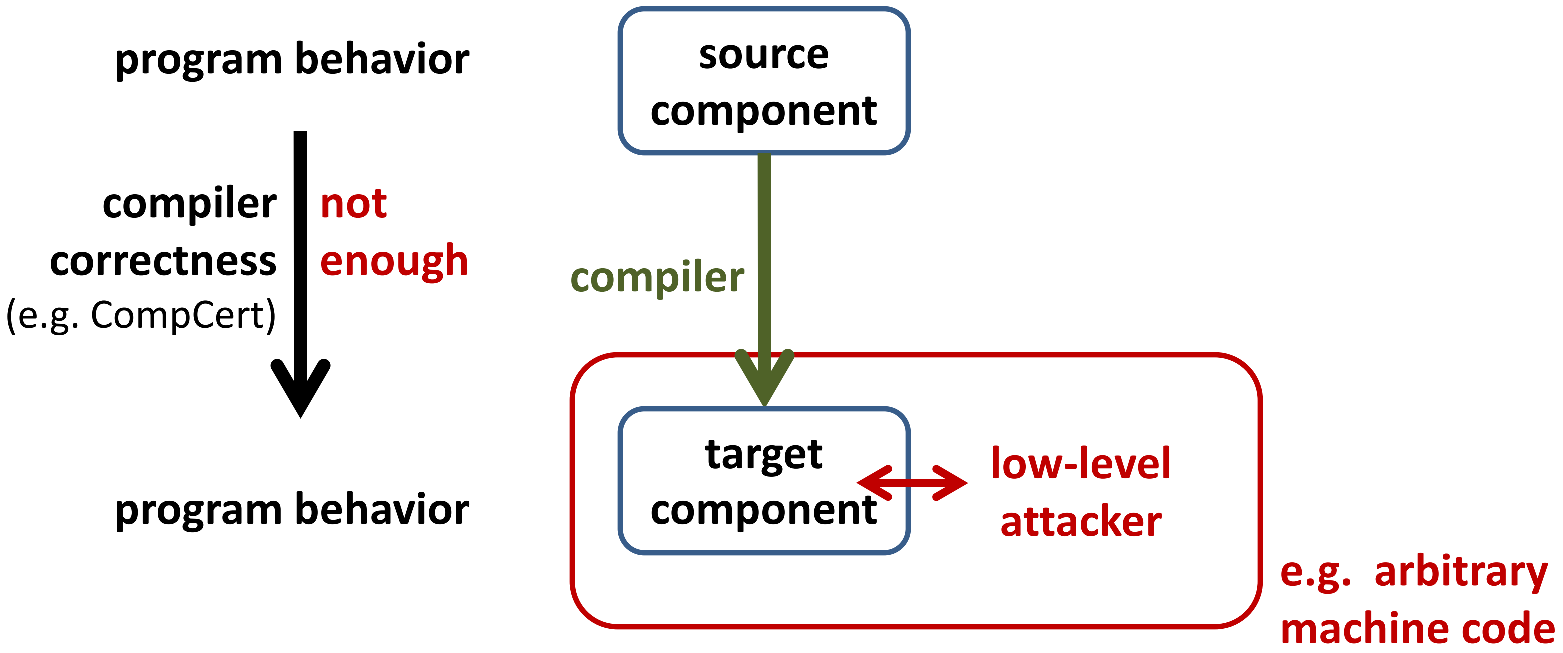
Formally verify: full abstraction

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



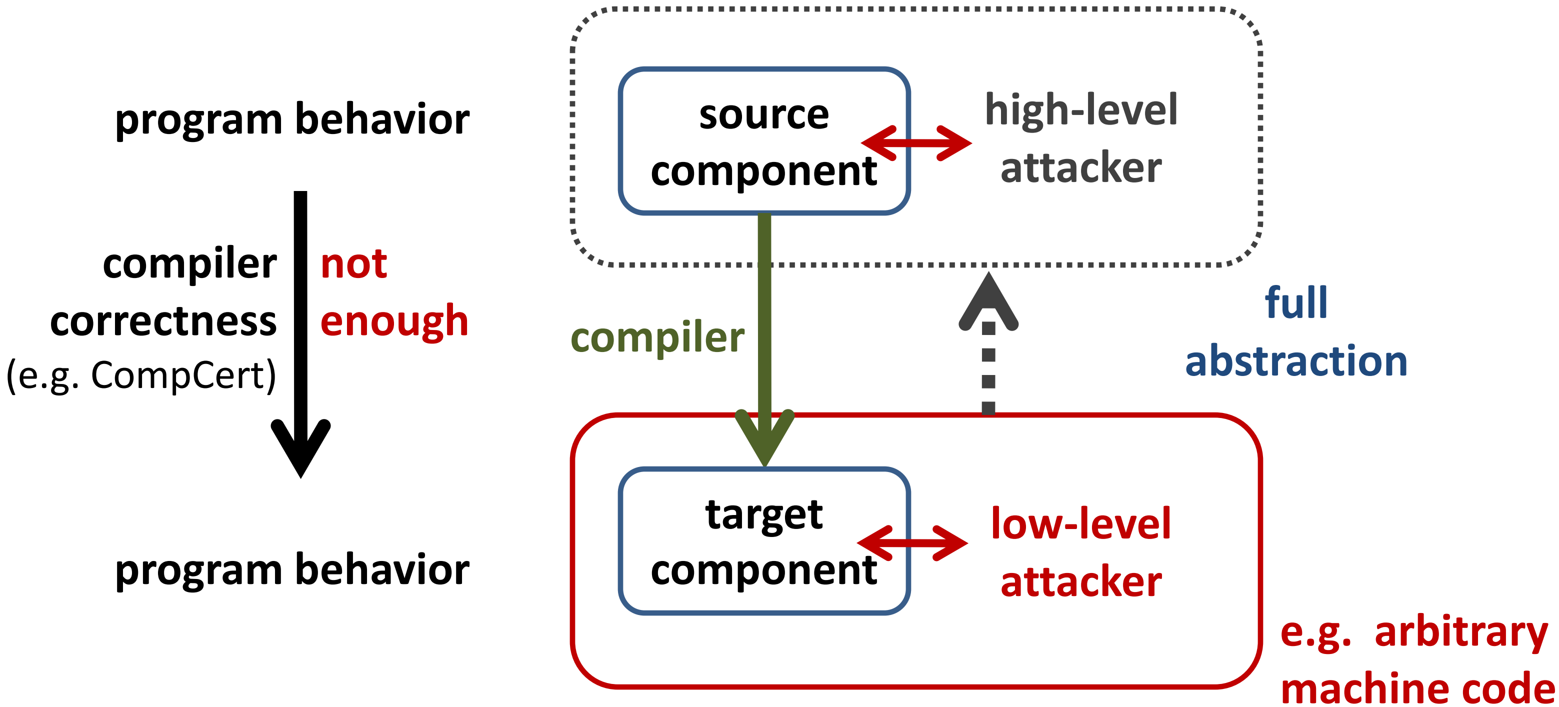
Formally verify: full abstraction

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



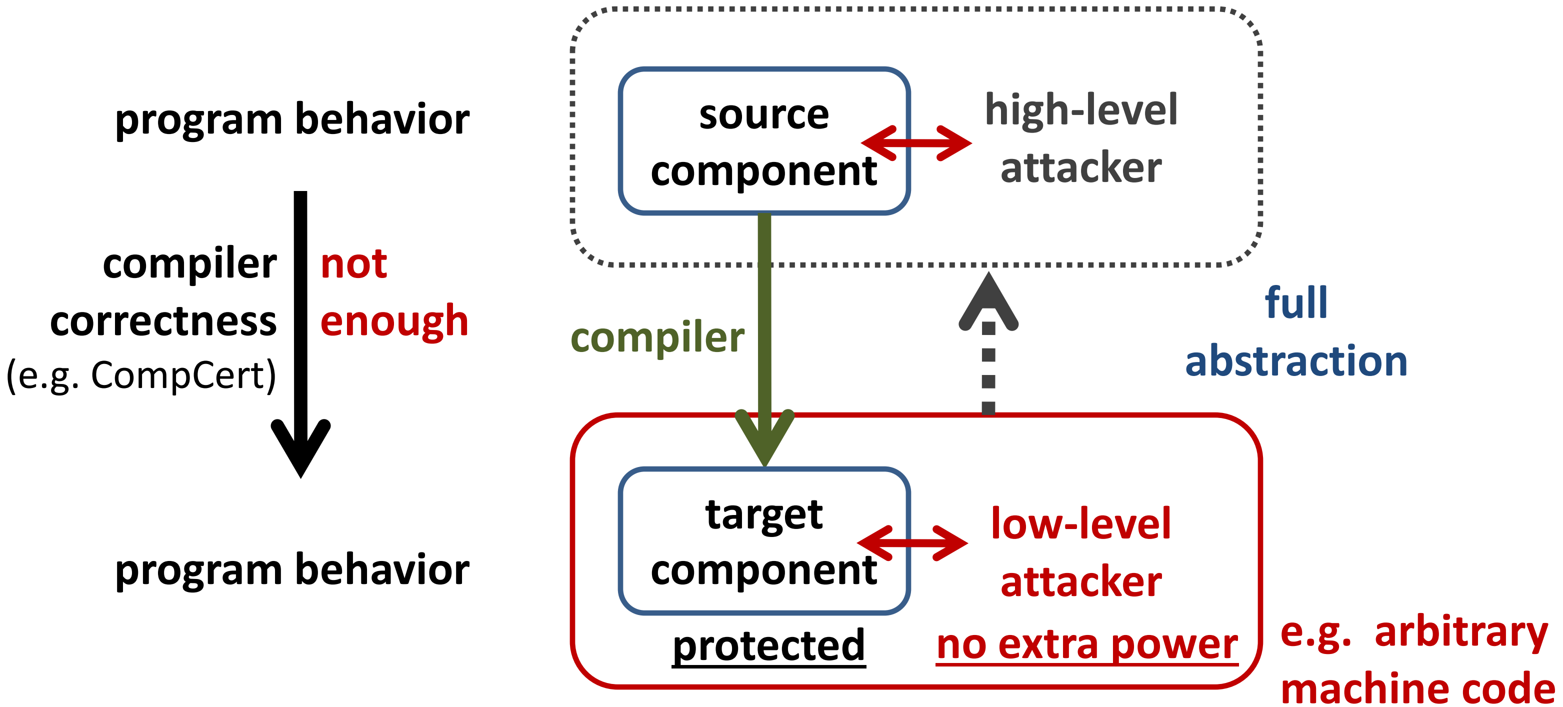
Formally verify: full abstraction

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



Formally verify: full abstraction

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



Formally verify: full abstraction

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

source component ⟷ high-level attacker

program behavior

compiler correctness (e.g. CompCert) ↓ **not enough**

program behavior

secure

full abstraction ↓ *folklore

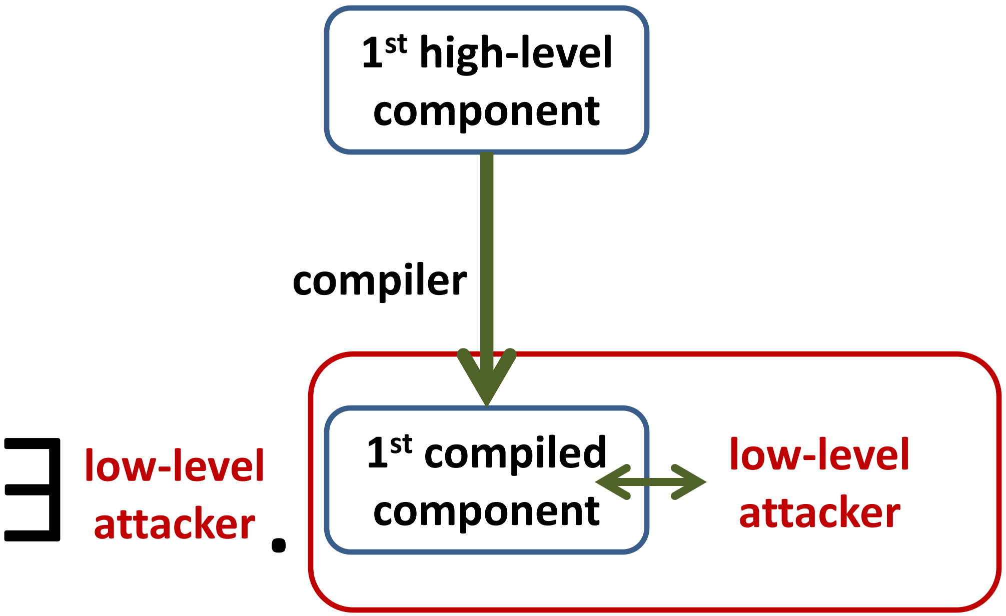
secure

Benefit: sound security reasoning in the source language

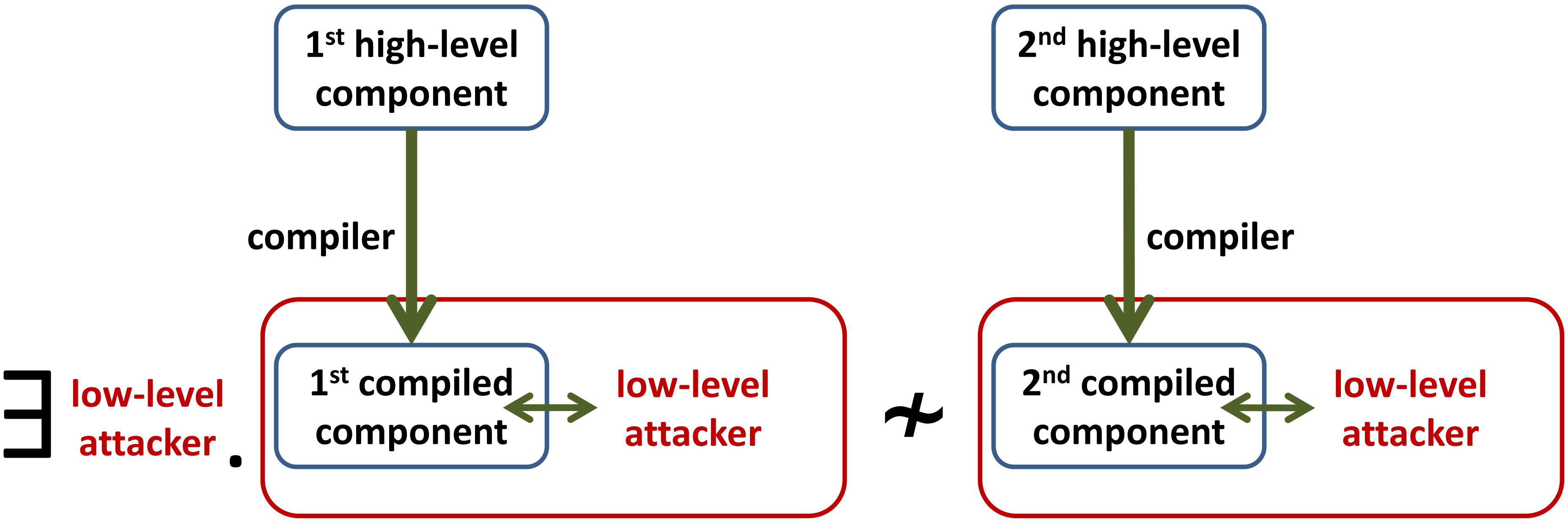
forget about compiler chain (linker, loader, runtime system)

forget that libraries are written in a lower-level language

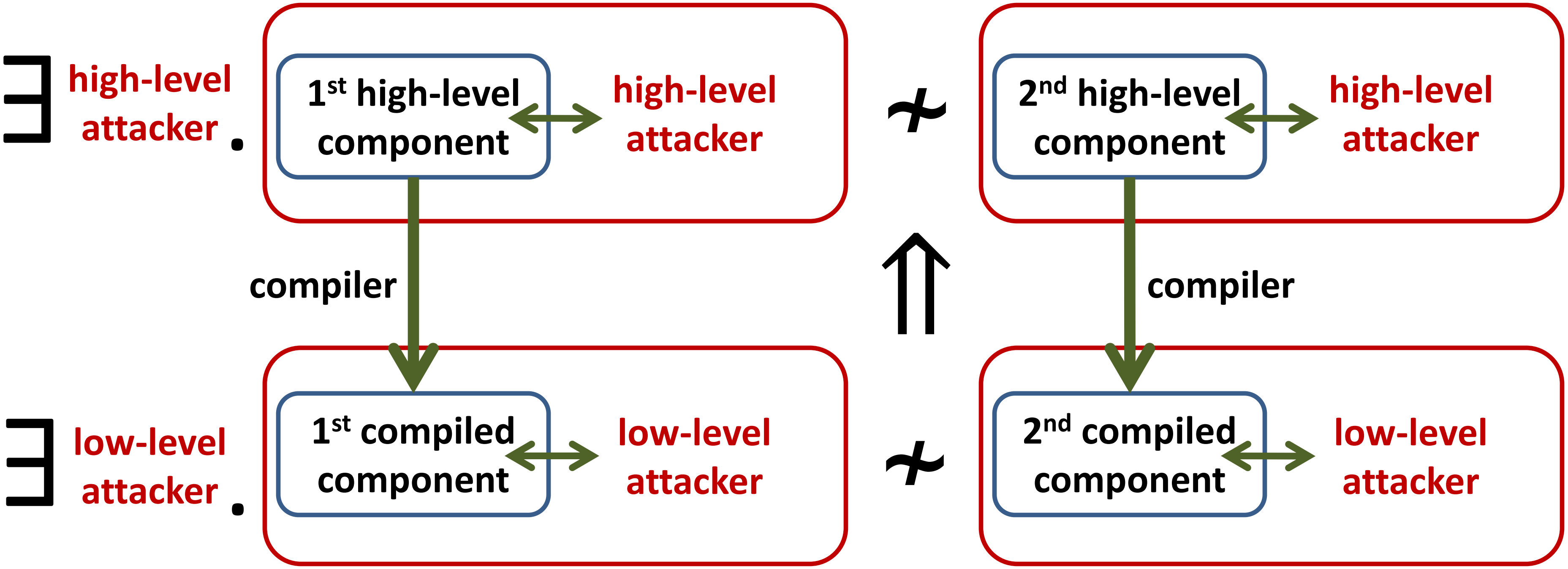
Fully abstract compilation, definition



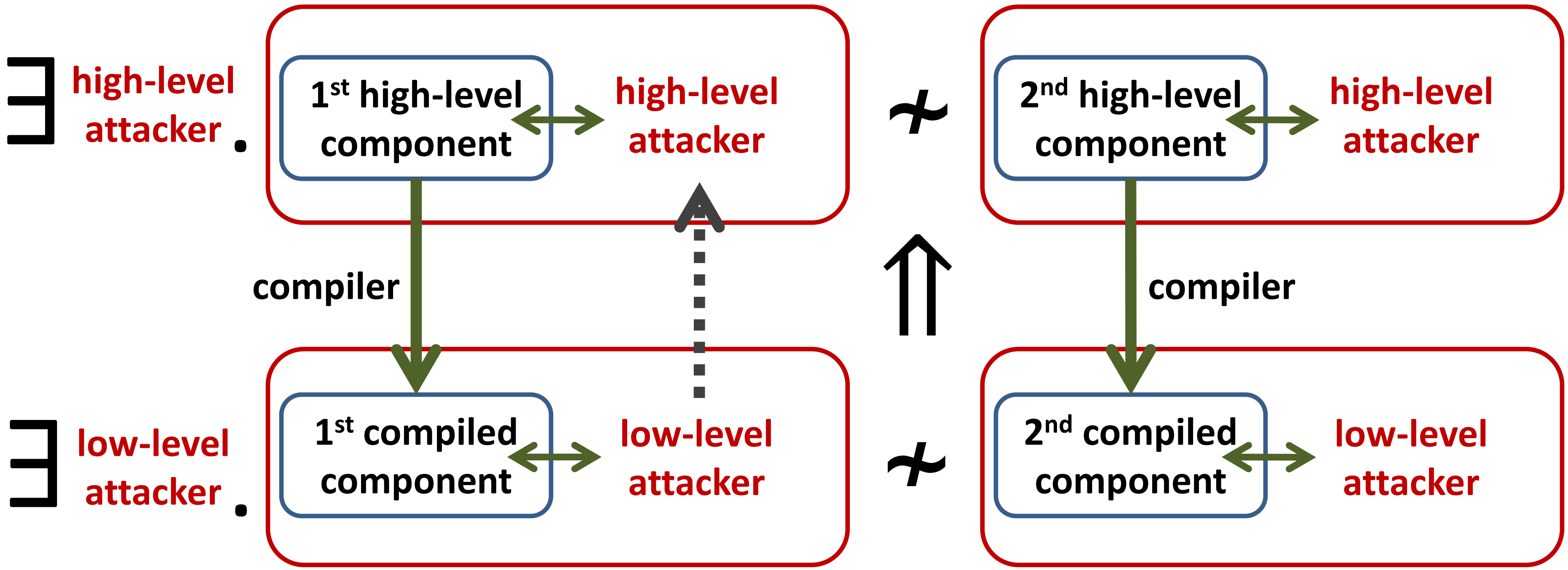
Fully abstract compilation, definition



Fully abstract compilation, definition



Fully abstract compilation, definition



SECOMP: achieving full abstraction at scale

F* language
(ML + verification)

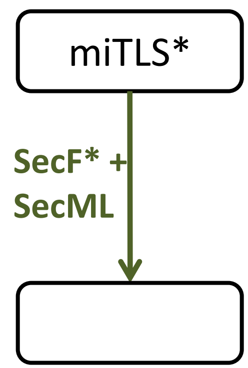
miTLS*

C language
+ memory safety
+ components

SECOMP: achieving full abstraction at scale

F* language
(ML + verification)

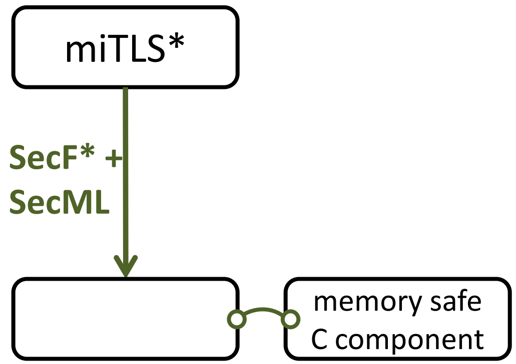
C language
+ memory safety
+ components



SECOMP: achieving full abstraction at scale

F* language
(ML + verification)

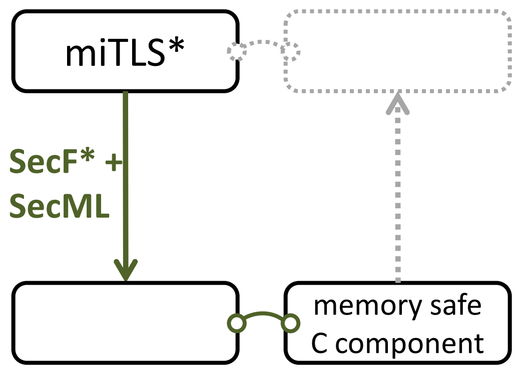
C language
+ memory safety
+ components



SECOMP: achieving full abstraction at scale

F* language
(ML + verification)

C language
+ memory safety
+ components

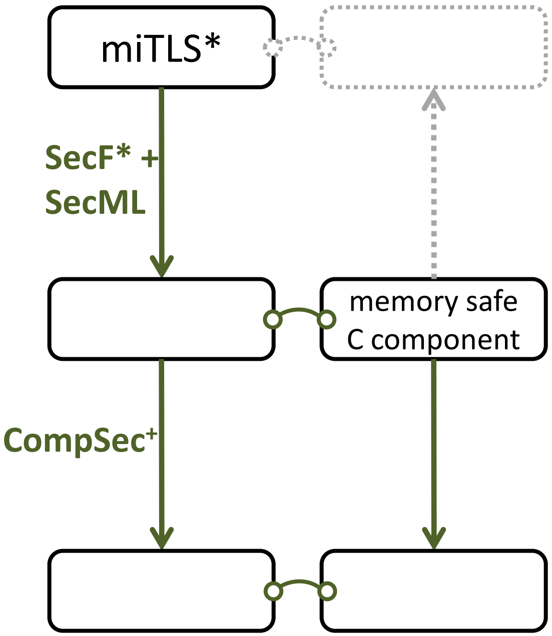


SECOMP: achieving full abstraction at scale

F* language
(ML + verification)

C language
+ memory safety
+ components

ASM language
(RISC-V + micro-policies)

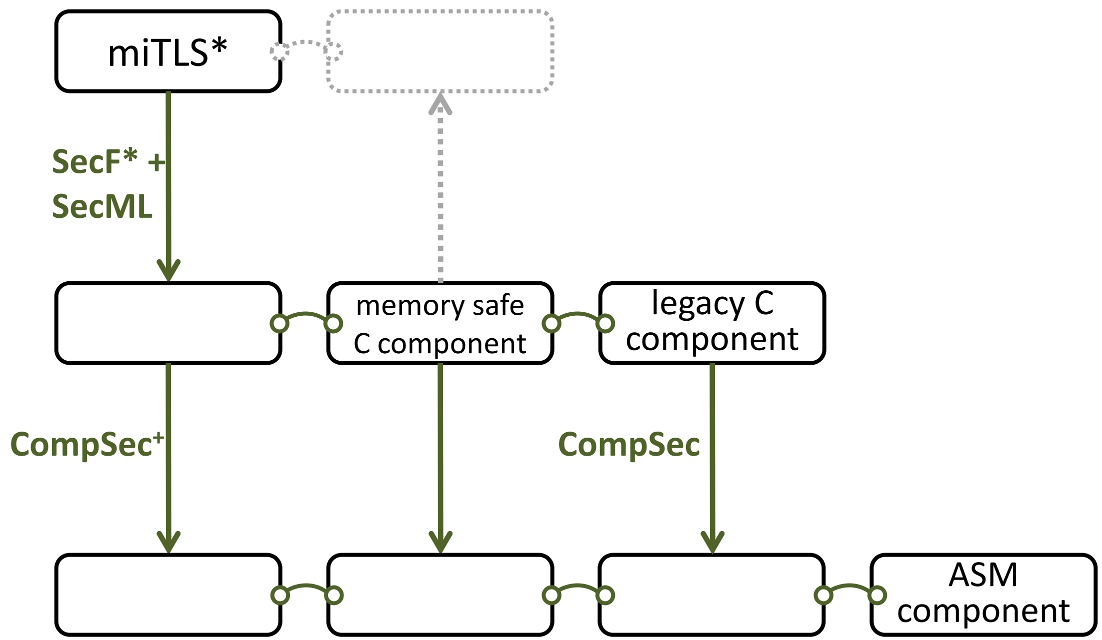


SECOMP: achieving full abstraction at scale

F* language
(ML + verification)

C language
+ memory safety
+ components

ASM language
(RISC-V + micro-policies)

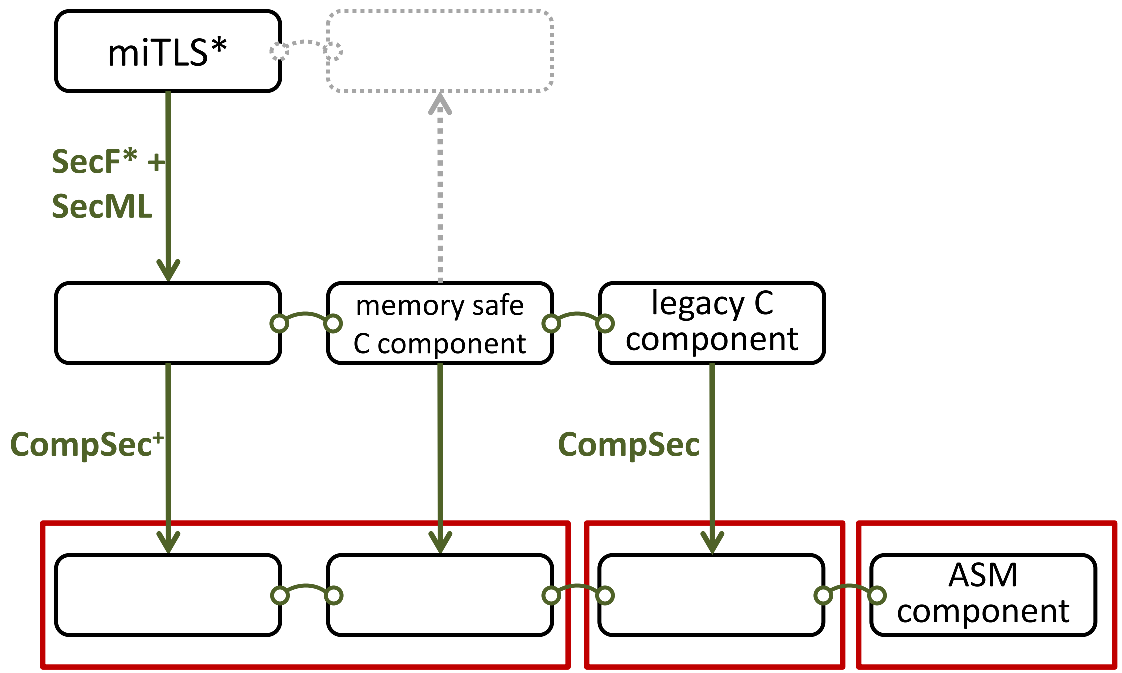


SECOMP: achieving full abstraction at scale

F* language
(ML + verification)

C language
+ memory safety
+ components

ASM language
(RISC-V + micro-policies)



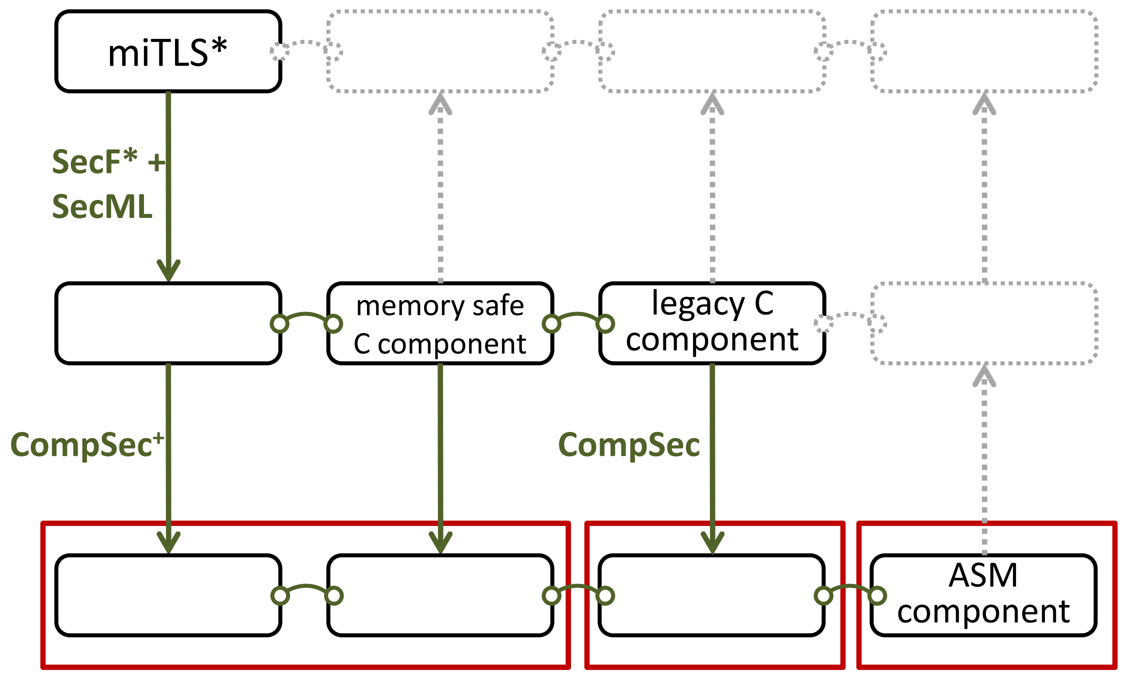
protecting component boundaries

SECOMP: achieving full abstraction at scale

F* language
(ML + verification)

C language
+ memory safety
+ components

ASM language
(RISC-V + micro-policies)



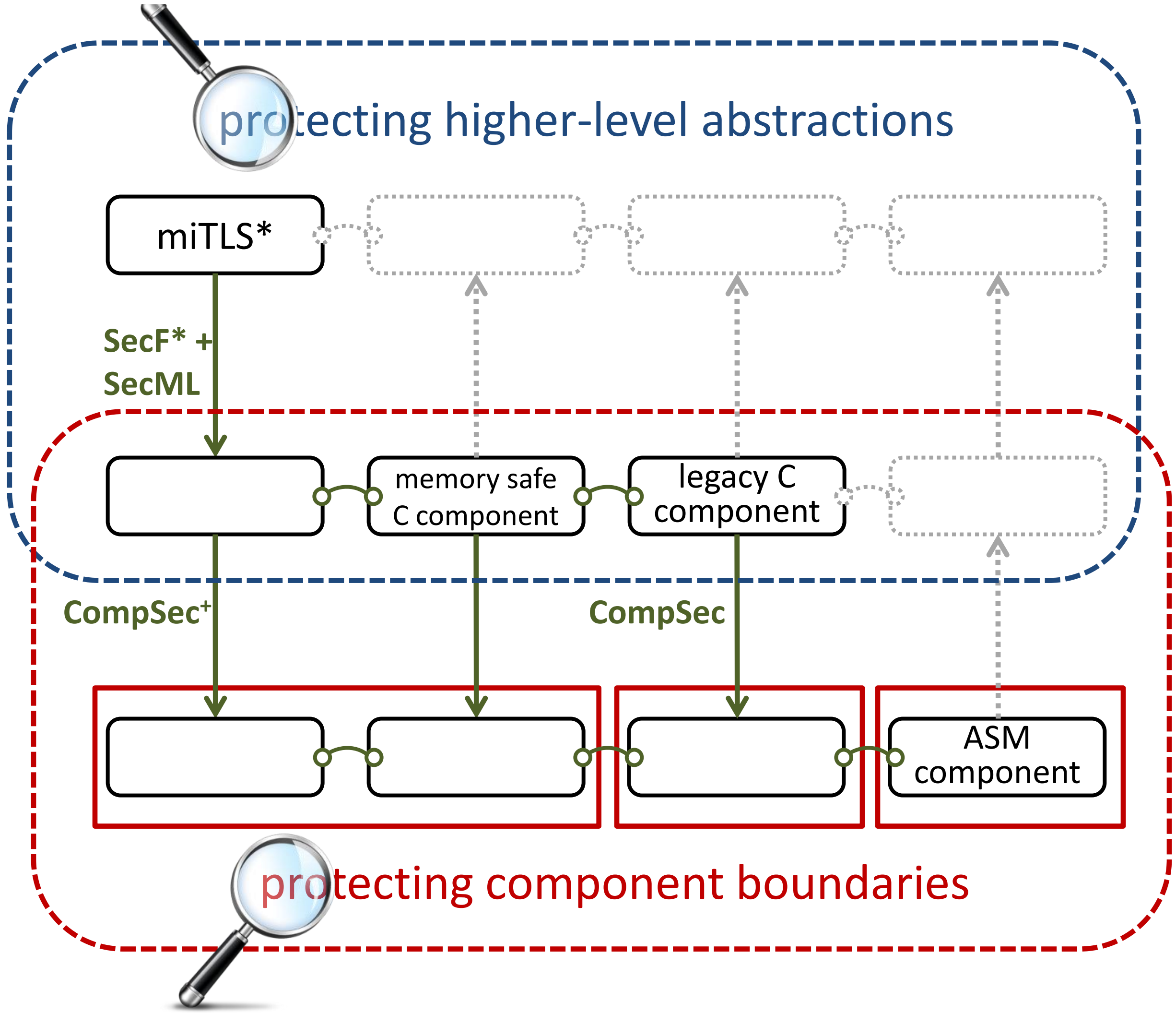
protecting component boundaries

SECOMP: achieving full abstraction at scale

F* language
(ML + verification)

C language
+ memory safety
+ components

ASM language
(RISC-V + micro-policies)



Protecting component boundaries

- **Add mutually distrustful components to C**
  - interacting only via **strictly enforced interfaces**

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

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

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

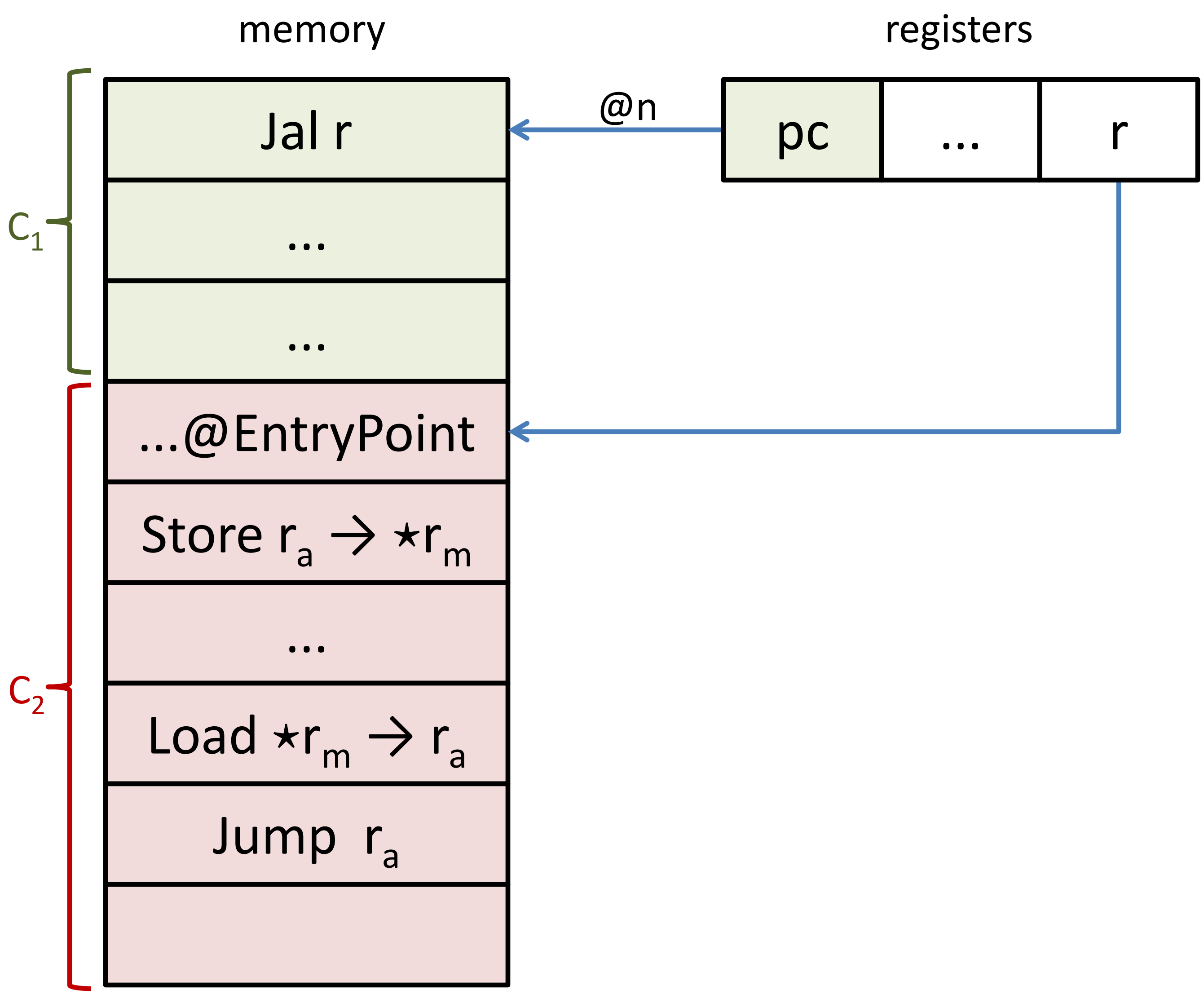
Protecting component boundaries

- **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

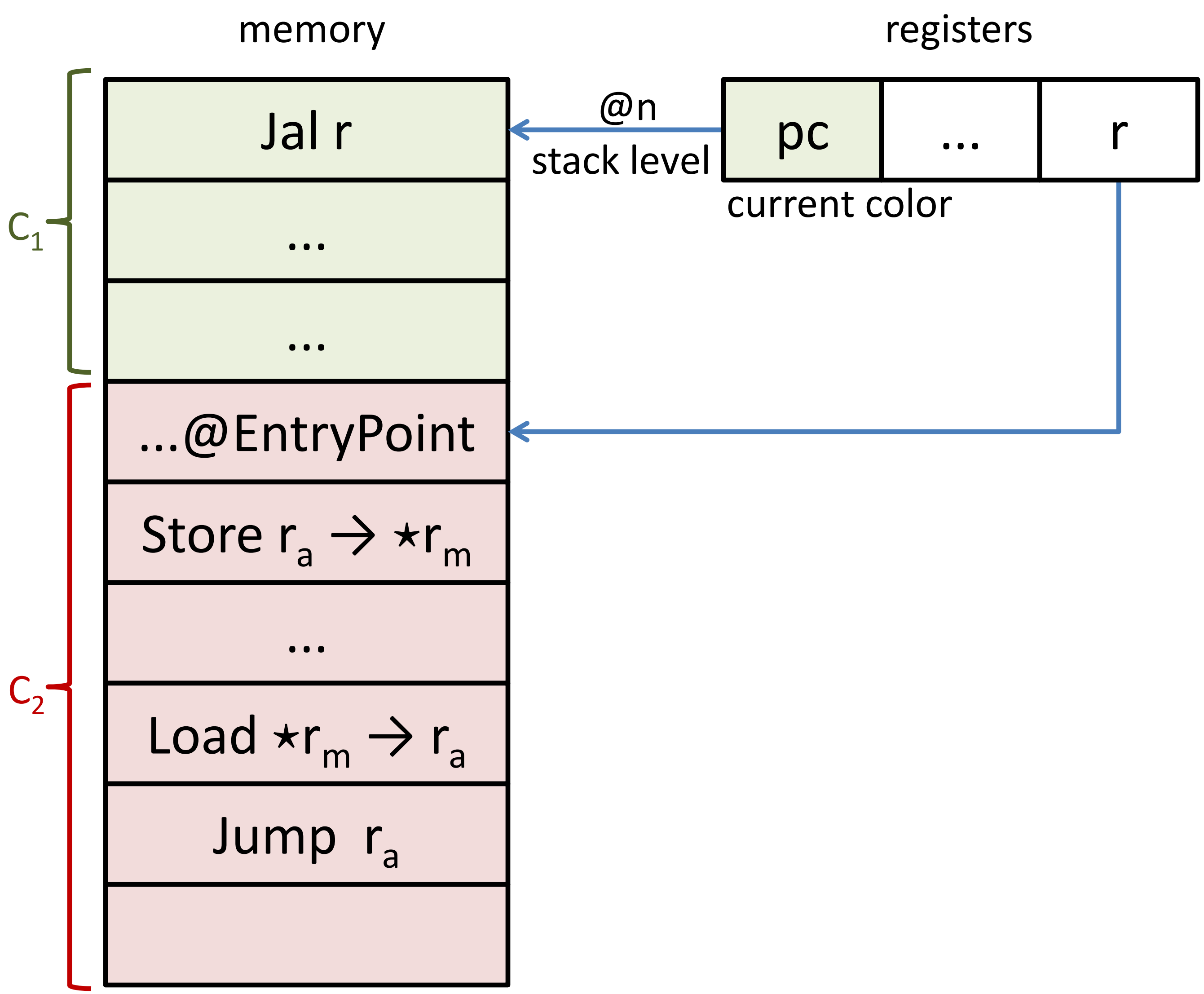
Recent preliminary work, joint with Yannis Juglaret et al

Compartmentalization micro-policy



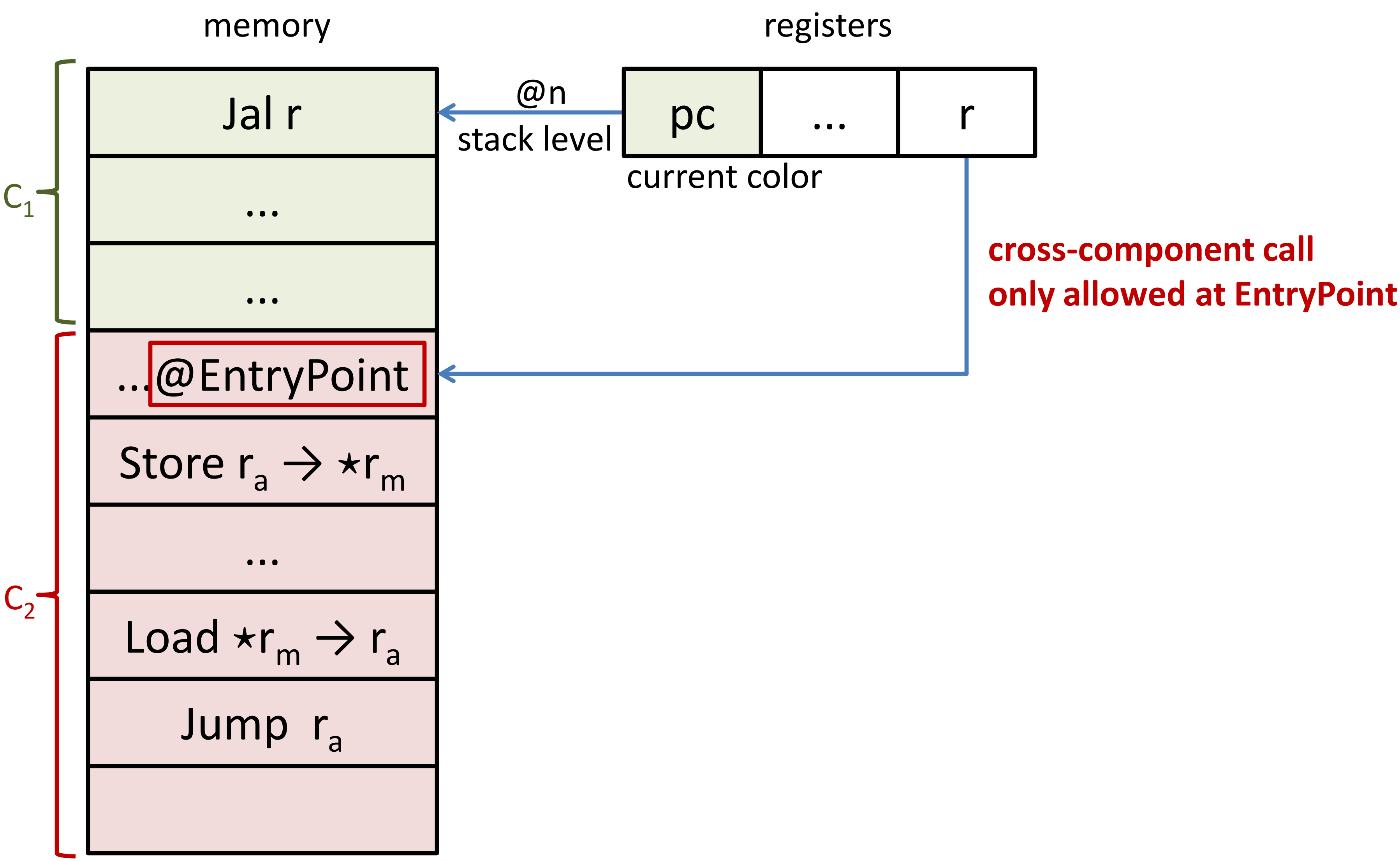
[Towards a Fully Abstract Compiler Using Micro-Policies, Juglaret et al, TR 2015]

Compartmentalization micro-policy



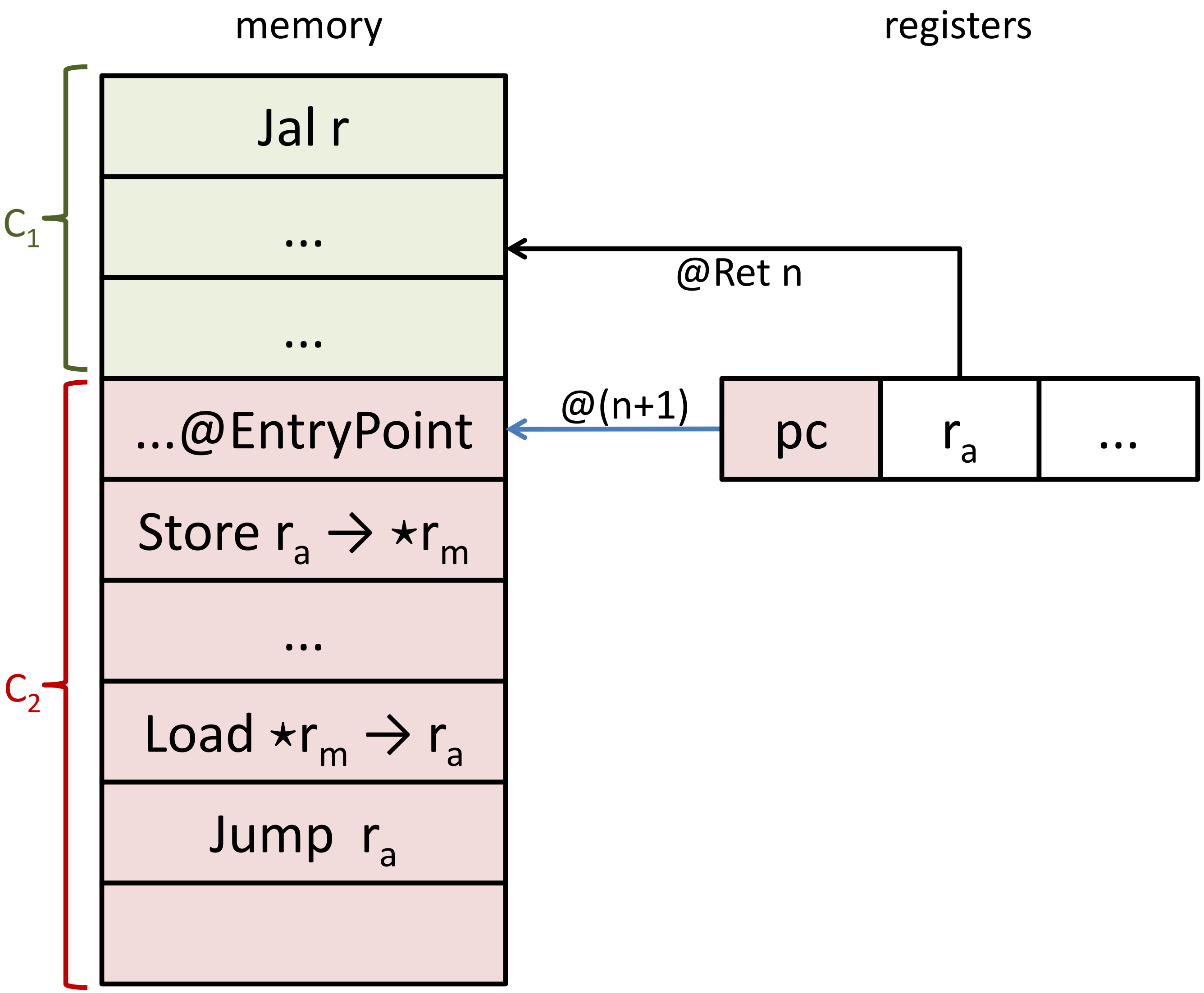
[Towards a Fully Abstract Compiler Using Micro-Policies, Juglaret et al, TR 2015]

Compartmentalization micro-policy



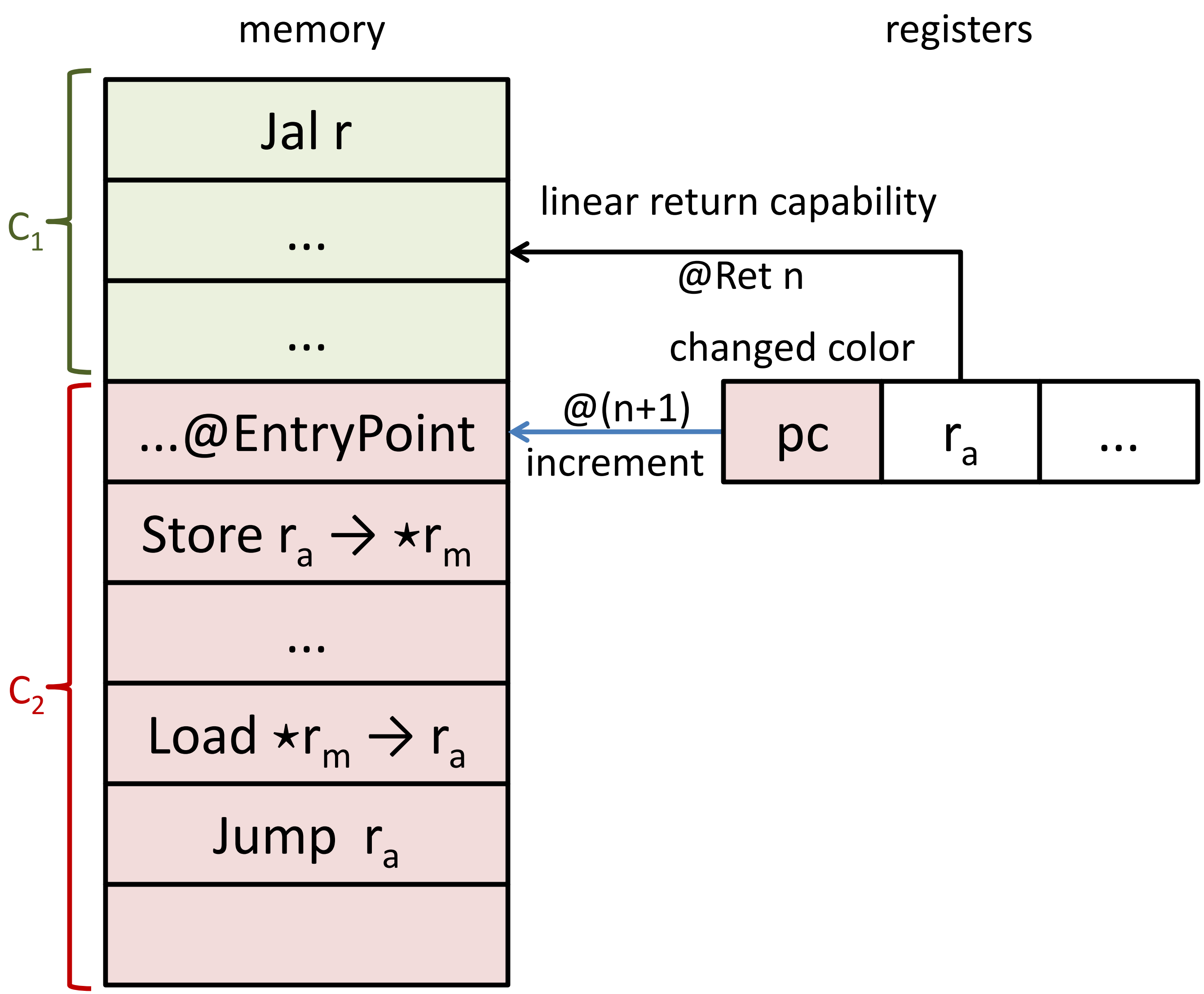
[Towards a Fully Abstract Compiler Using Micro-Policies, Juglaret et al, TR 2015]

Compartmentalization micro-policy



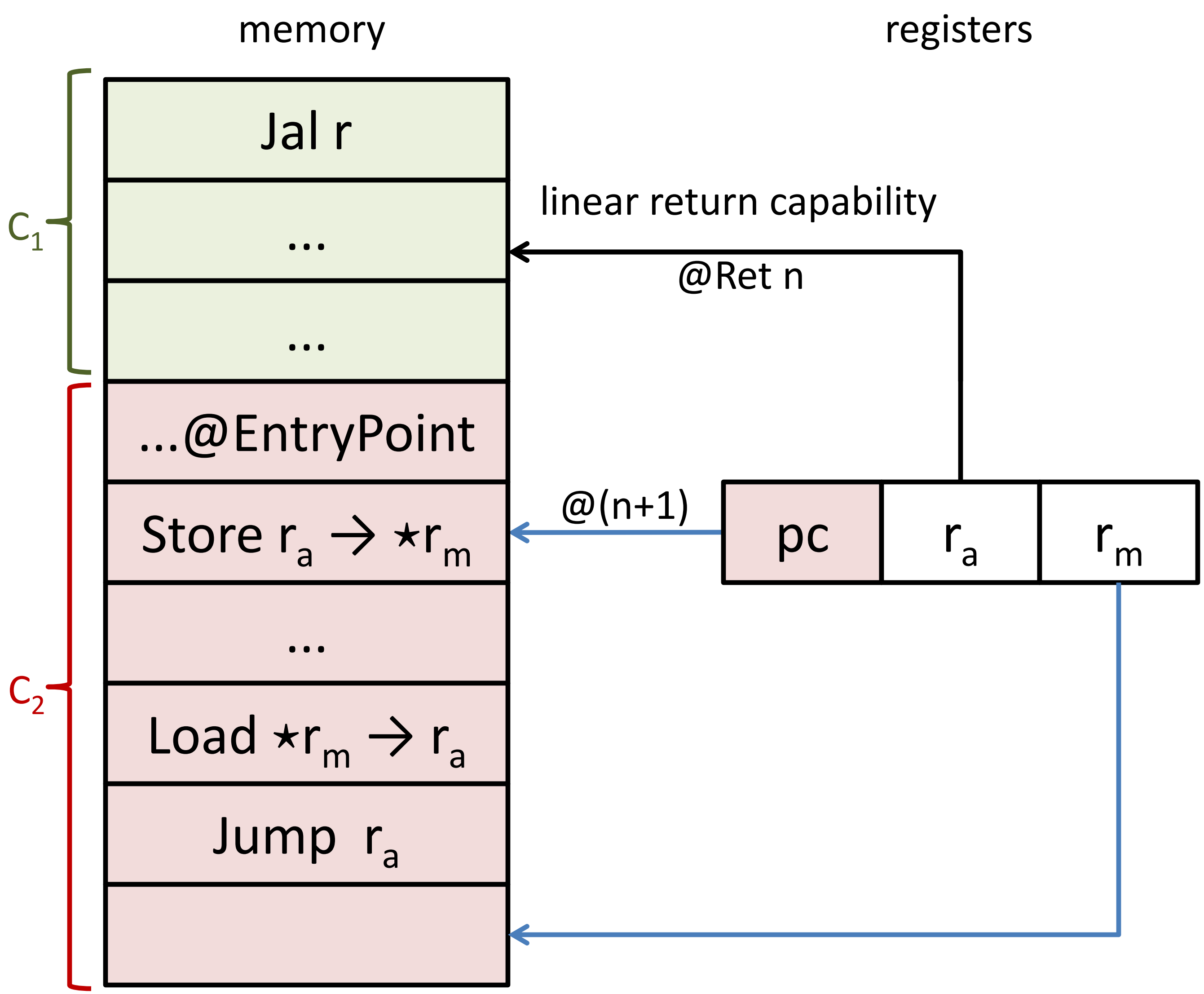
[Towards a Fully Abstract Compiler Using Micro-Policies, Juglaret et al, TR 2015]

Compartmentalization micro-policy



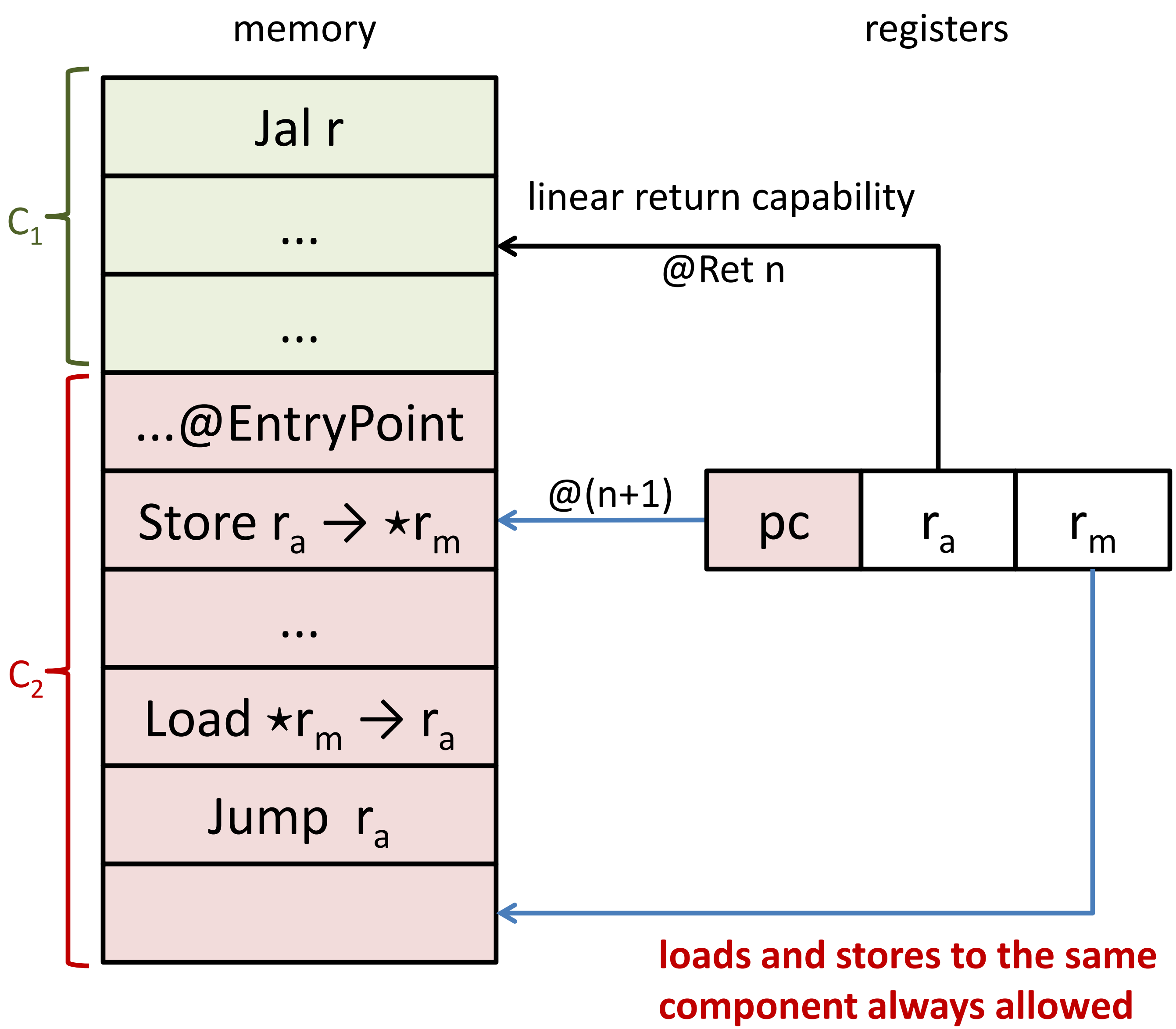
[Towards a Fully Abstract Compiler Using Micro-Policies, Juglaret et al, TR 2015]

Compartmentalization micro-policy

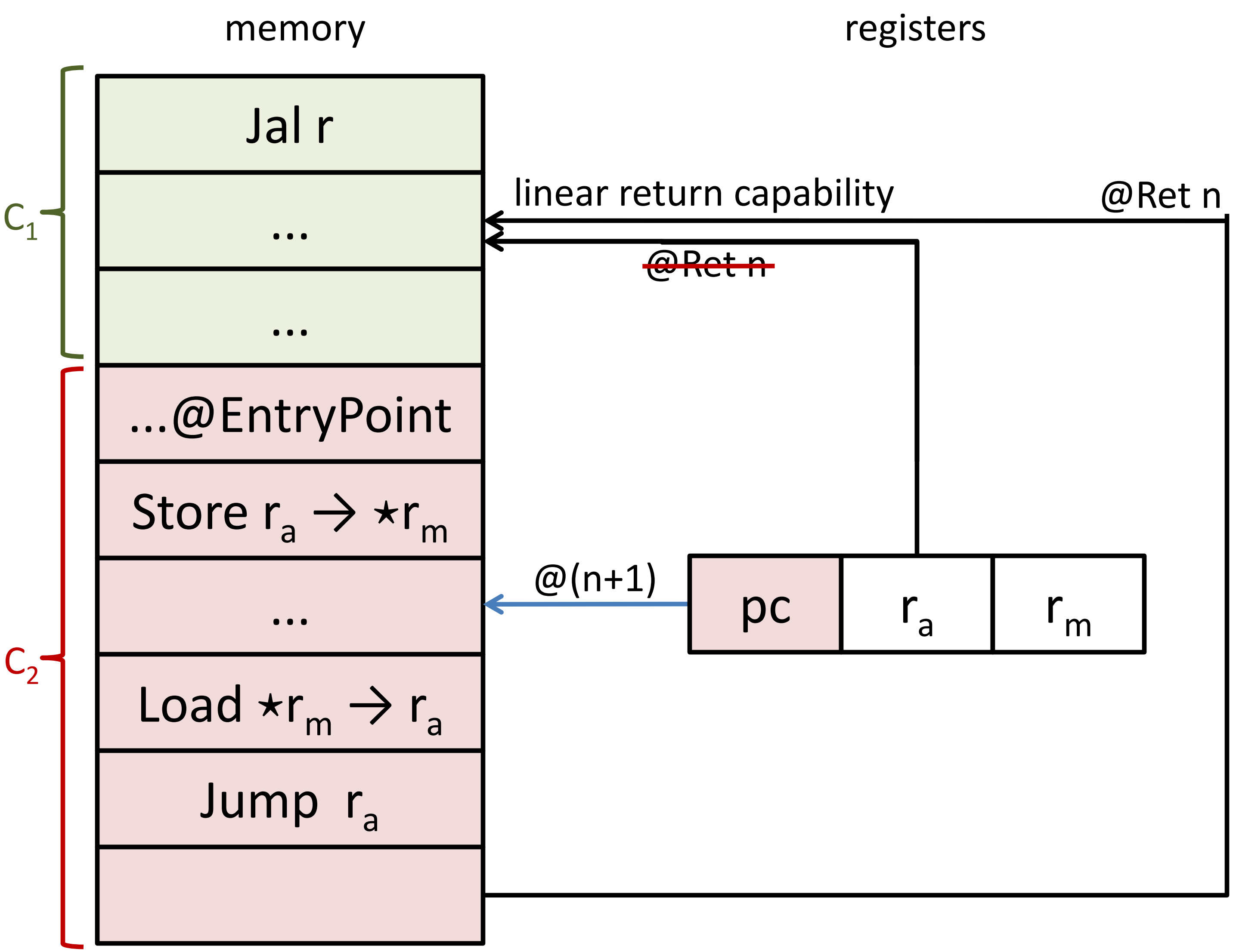


[Towards a Fully Abstract Compiler Using Micro-Policies, Juglaret et al, TR 2015]

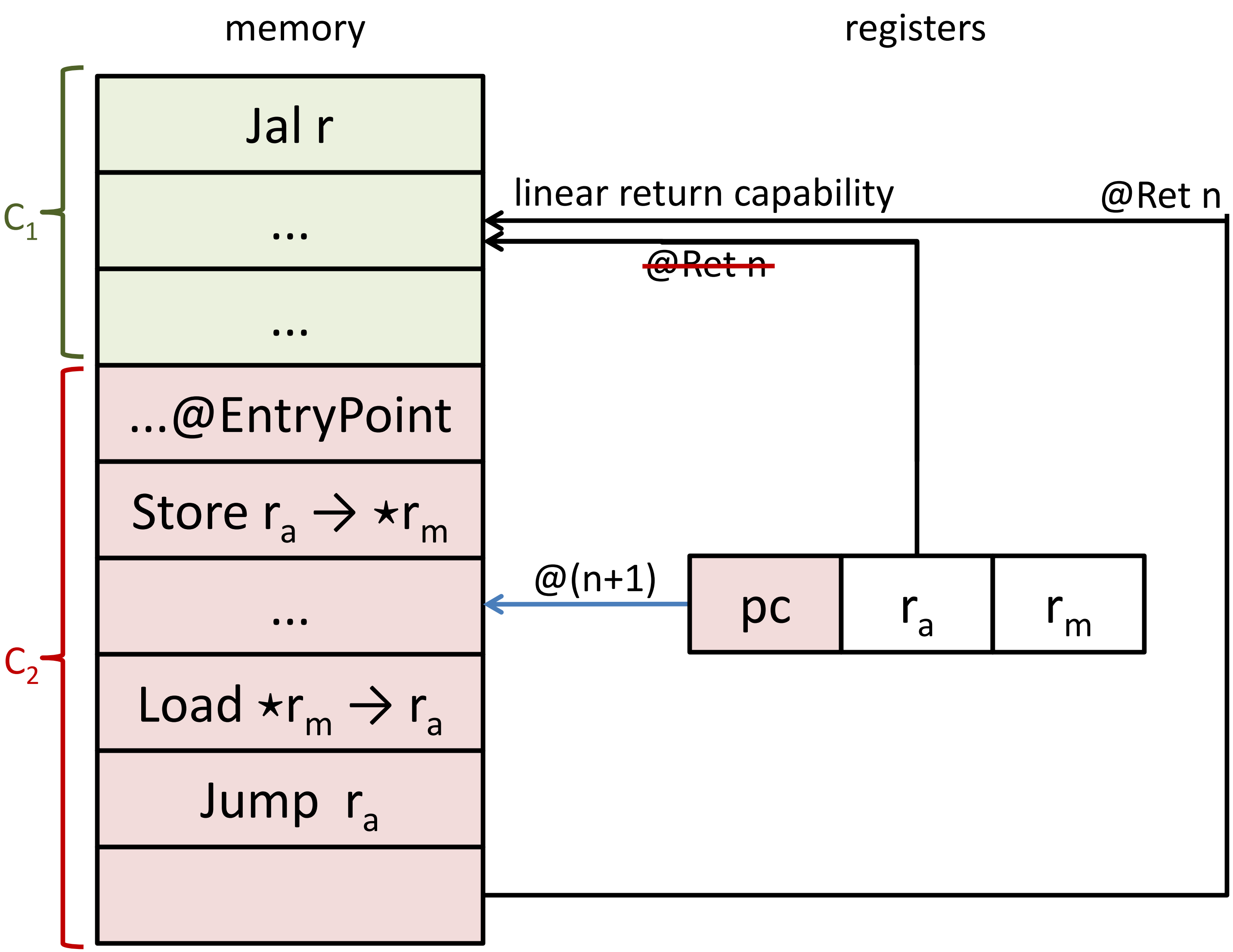
Compartmentalization micro-policy



Compartmentalization micro-policy

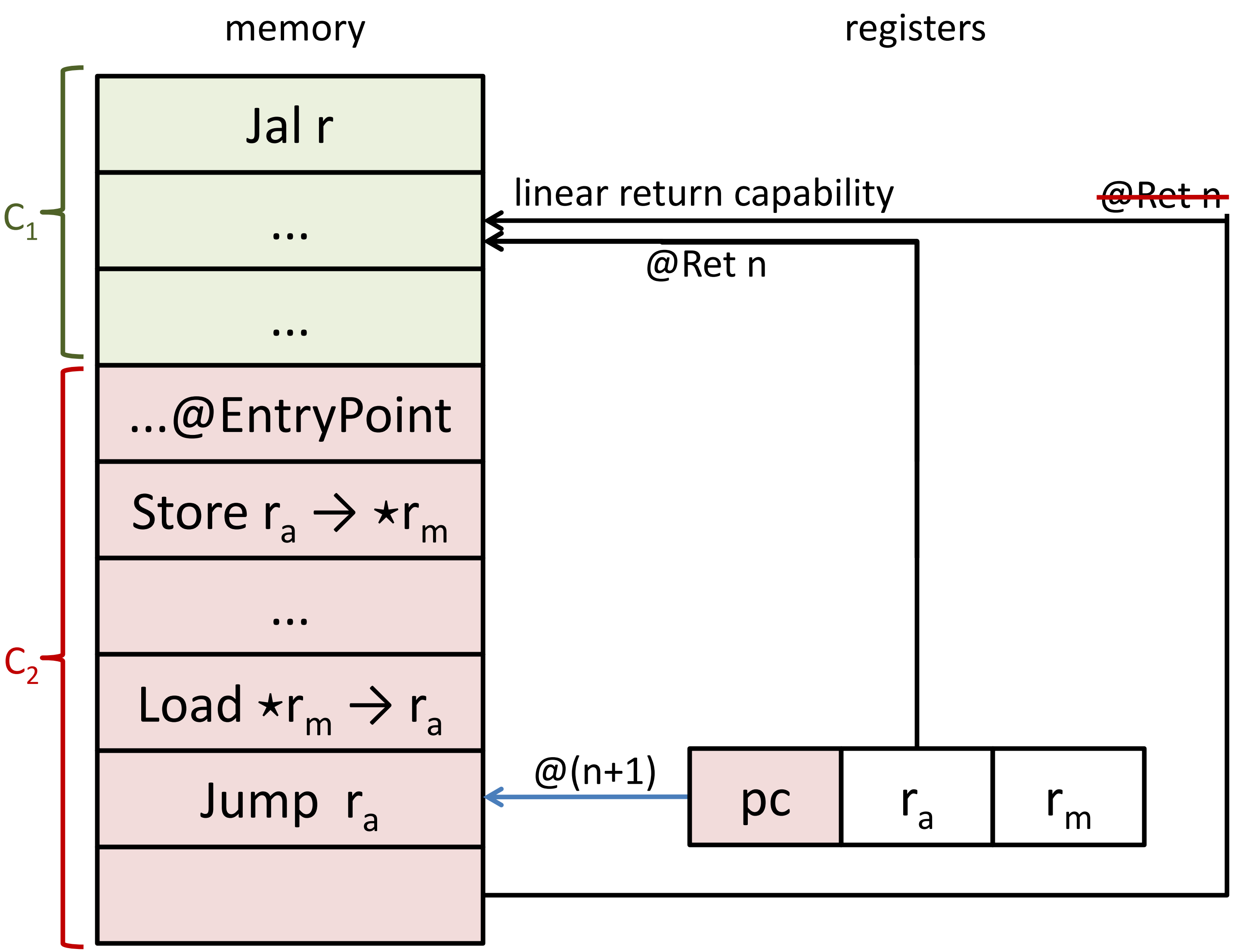


Compartmentalization micro-policy



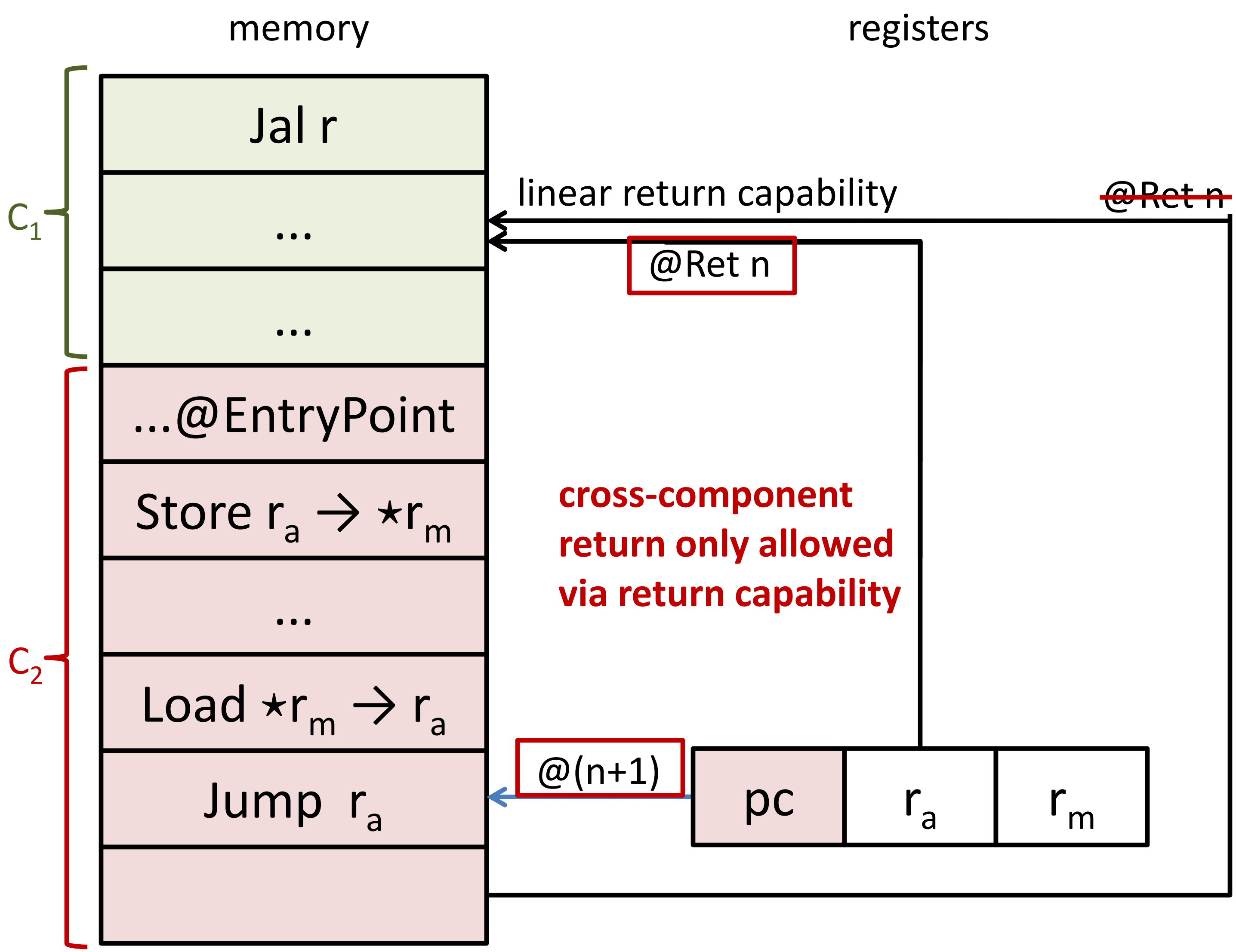
invariant:
at most one
return capability
per call stack level

Compartmentalization micro-policy



invariant:
at most one return capability per call stack level

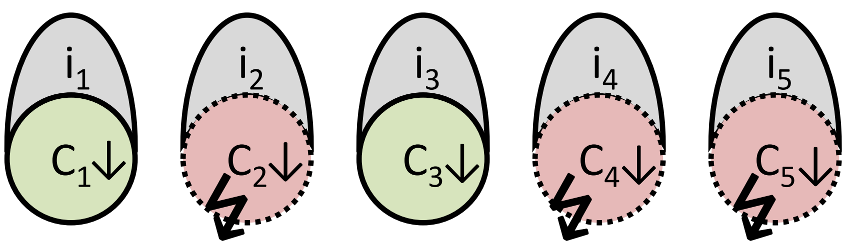
Compartmentalization micro-policy



invariant:
at most one return capability per call stack level

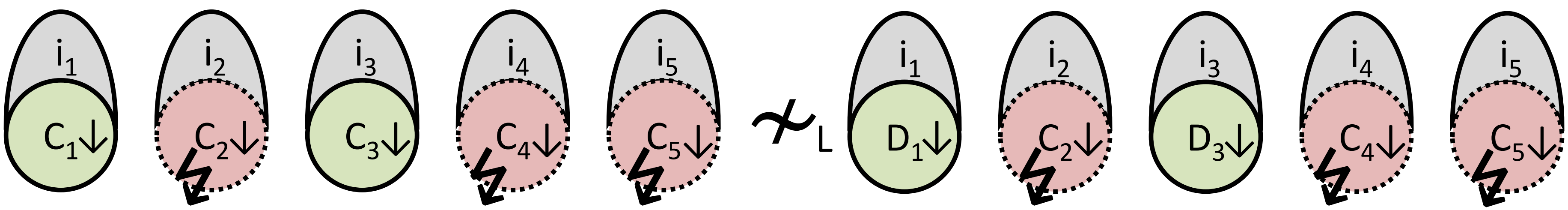
Secure compartmentalizing compilation (SCC)

∀compromise scenarios.



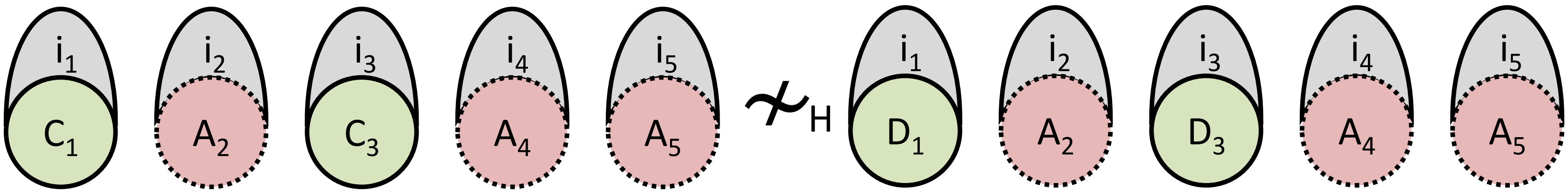
Secure compartmentalizing compilation (SCC)

∀compromise scenarios.



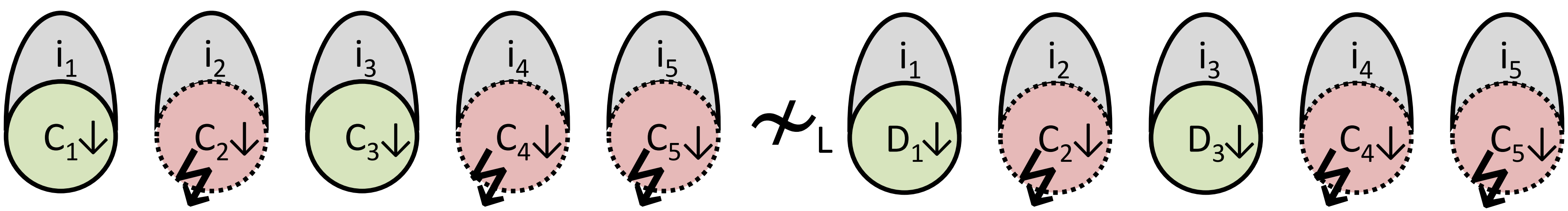
Secure compartmentalizing compilation (SCC)

∀compromise scenarios.



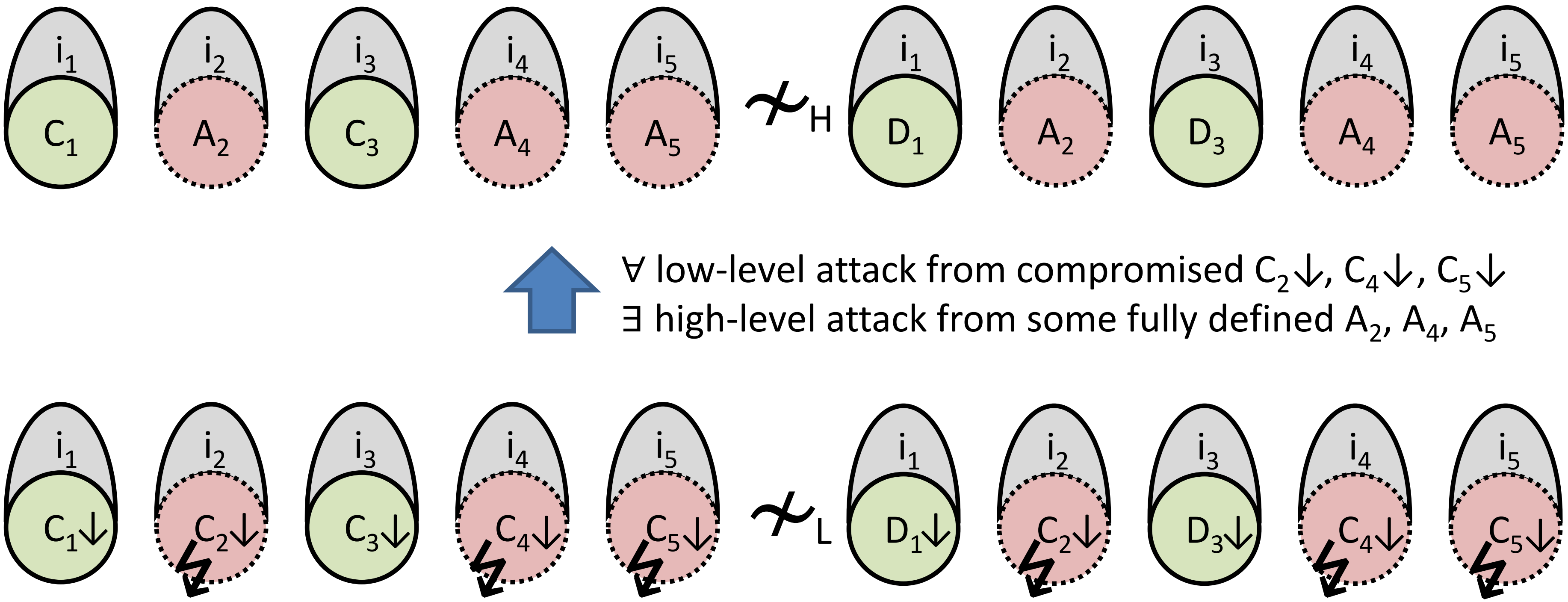
∀ low-level attack from compromised $C_2\downarrow$, $C_4\downarrow$, $C_5\downarrow$

∃ high-level attack from some fully defined A_2, A_4, A_5



Secure compartmentalizing compilation (SCC)

∀compromise scenarios.



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

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

Protecting higher-level abstractions

- **ML abstractions we want to enforce with micro-policies**
  - types, value immutability, opaqueness of closures, parametricity (dynamic sealing), GC vs malloc/free, ...

Protecting higher-level abstractions

- **ML abstractions we want to enforce with micro-policies**
  - types, value immutability, opaqueness of closures, parametricity (dynamic sealing), GC vs malloc/free, ...
- **F*: enforcing full specifications using micro-policies**
  - some can be turned into **contracts,** checked dynamically
  - fully abstract compilation of F* to ML **trivial for ML interfaces** (because F* allows and tracks effects, as opposed to Coq)

Protecting higher-level abstractions

- **ML abstractions we want to enforce with micro-policies**
  - types, value immutability, opaqueness of closures, parametricity (dynamic sealing), GC vs malloc/free, ...
- **F*: enforcing full specifications using micro-policies**
  - some can be turned into **contracts,** checked dynamically
  - fully abstract compilation of F* to ML **trivial for ML interfaces** (because F* allows and tracks effects, as opposed to Coq)
- **Limits of purely-dynamic enforcement**
  - functional purity, termination, relational reasoning

Protecting higher-level abstractions

- **ML abstractions we want to enforce with micro-policies**
  - types, value immutability, opaqueness of closures, parametricity (dynamic sealing), GC vs malloc/free, ...
- **F*: enforcing full specifications using micro-policies**
  - some can be turned into **contracts,** checked dynamically
  - fully abstract compilation of F* to ML **trivial for ML 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 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

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)

Micro-policies:
remaining fundamental challenges

Micro-policies:
remaining fundamental challenges

- **Micro-policies for C and ML**
  - needed for vertical compiler composition
  - will put micro-policies in the hands of programmers

Micro-policies:
remaining fundamental challenges

- **Micro-policies for C and ML**
  - 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

Beyond full abstraction

- Is full abstraction always the right notion of secure compilation? The right attacker model?

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.]

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)

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)

What secure compilation adds over compositional compiler correctness

- **mapping back arbitrary low-level contexts**
- **preserving integrity properties**
  - robust compilation phrased in terms 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

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*

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)

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

SECOMP in a nutshell

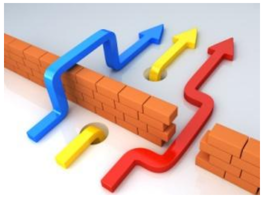
- **We need more secure languages, compilers, hardware**

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*)

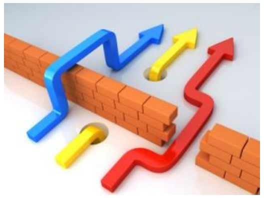
SECOMP in a nutshell

- **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, proof techniques
  - secure composition, micro-policies for C and ML



SECOMP in a nutshell

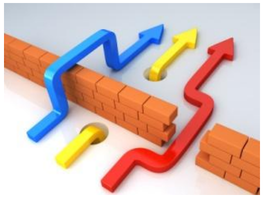
- **We need more secure languages, compilers, hardware**
- **Key enabler: micro-policies** (software-hardware protection)
- **Grand challenge: the first efficient formally secure compilers** for **realistic programming languages** (C, ML, F*)
- **Answering challenging fundamental questions**
  - attacker models, proof techniques
  - secure composition, micro-policies for C and ML



- **Achieving strong security properties like full abstraction**
  + testing and proving formally that this is the case

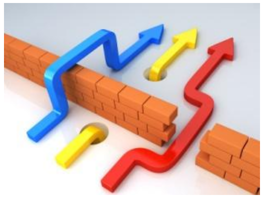
SECOMP in a nutshell

- **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, proof techniques
  - secure composition, micro-policies for C and ML



- **Achieving strong security properties like full abstraction**
  + testing and proving formally that this is the case
- **Measuring & lowering the cost of secure compilation**

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, proof techniques
  – secure composition, micro-policies for C and ML
- **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**
- Prosecco can also support outstanding candidates in the **CR2 competition**

Collaborators & Community

- **Current collaborators from Micro-Policies project**
  - UPenn, MIT, Portland State, Draper Labs

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)
  - **Amal Ahmed** coming to Paris for 1 year sabbatical (from 09/2017)

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)
  - **Amal Ahmed** coming to Paris for 1 year sabbatical (from 09/2017)
- **Secure compilation meetings (very informal)**
  - 1st at INRIA Paris on August 2016
  - 2nd in Paris on 15(?) January 2017 ... maybe at UPMC
  - **build larger research community, identify open problems,**
    bring together communities (hardware, systems, security,
    languages, verification, ...)

Questions for Gallium

- What do you think? Is this plan outrageous?
- Would CompCert be a good base for some of this?
- Is there any plan for a RISC-V backend for CompCert?
- Is anyone from Gallium interested in working on secure compilation?