SECOMP

Efficient Formally Secure Compilers to a Tagged Architecture

Cătălin Hrițcu

Inria Paris

(visiting researcher at Microsoft until end of November)

(member of Everest expedition)

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

it’s all relative ☺

SECOMP

Efficient Formally Secure Compilers to a Tagged Architecture

Cătălin Hrițcu

Inria Paris

(visiting researcher at Microsoft until end of November)

(member of Everest expedition)

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

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"

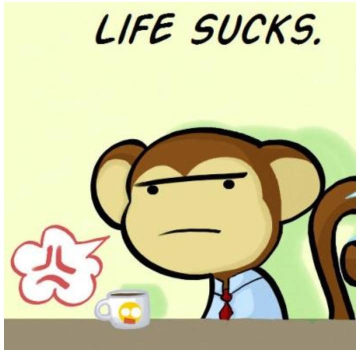
Unsafe low-level languages

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

THE
C
PROGRAMMING
LANGUAGE

Unsafe low-level languages

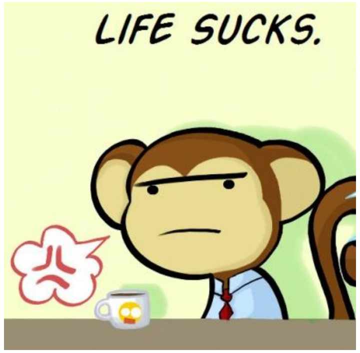
- C (1972) and C++ **undefined behavior**
  - including buffer overflows, checks too expensive
  - compilers optimize aggressively assuming undefined behavior will simply not happen
- **Programmers bear the burden for security**
  - just write secure code ... all of it

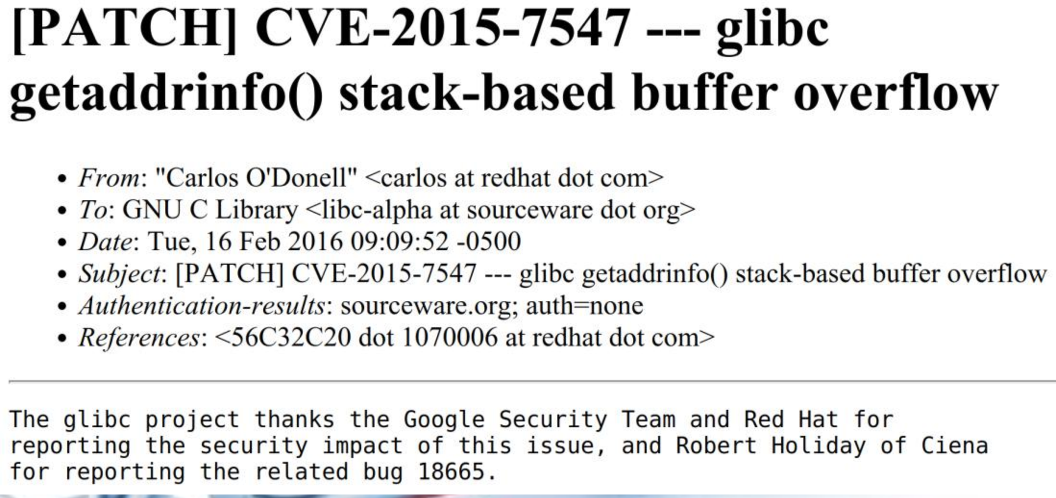


Unsafe low-level languages

- 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

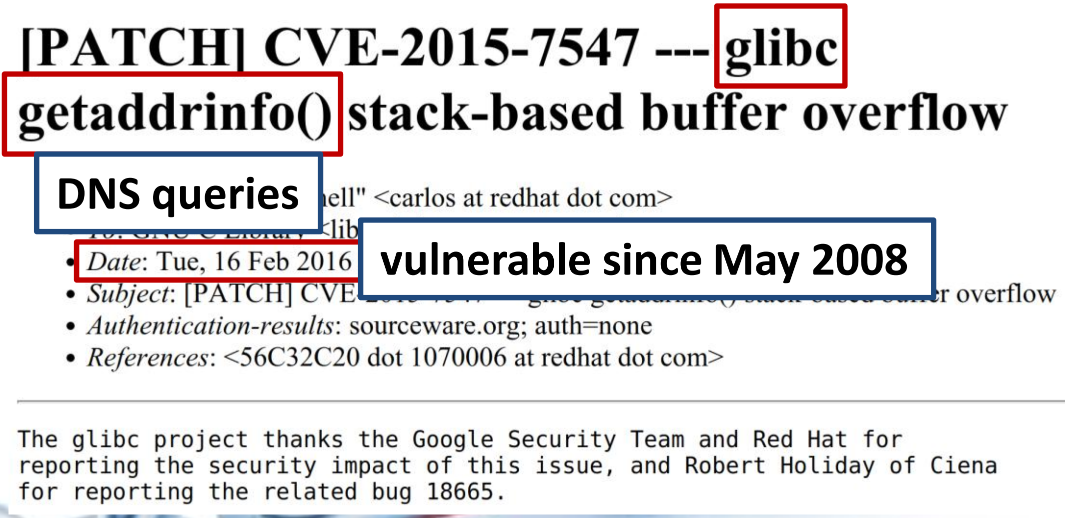
- *From*: "Carlos O'Donell" <carlos at redhat dot com>
- *To*: GNU C Library <libc-alpha at sourceware dot org>
- *Date*: Tue, 16 Feb 2016 09:09:52 -0500
- *Subject*: [PATCH] CVE-2015-7547 --- glibc getaddrinfo() stack-based buffer overflow
- *Authentication-results*: sourceware.org; auth=none
- *References*: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

To: GNU C Library <lib

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

vulnerable since May 2008

```
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)

OCaml
F#
Java
#
Haskell

Safer high-level languages?

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

Safer high-level languages?

OCaml F#

Java Haskell

- **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**

Teasing out 2 different problems

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

Teasing out 2 different problems

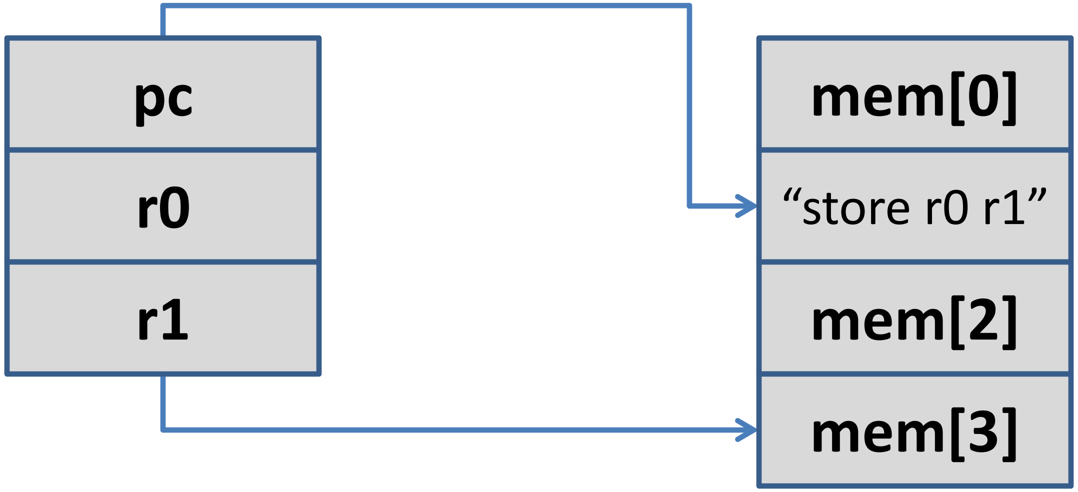
- **1. inherently insecure low-level languages**
  - **memory unsafe**: any buffer overflow can be catastrophic allowing remote attackers to gain complete control
- **2. unsafe interoperability with lower-level code**
  - even code written in **safer high-level languages** has to interoperate with **insecure low-level libraries**
  - **unsafe interoperability:** all high-level safety guarantees lost

Key enabler: Micro-Policies

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

Key enabler: Micro-Policies

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



pc

r0

r1

mem[0]

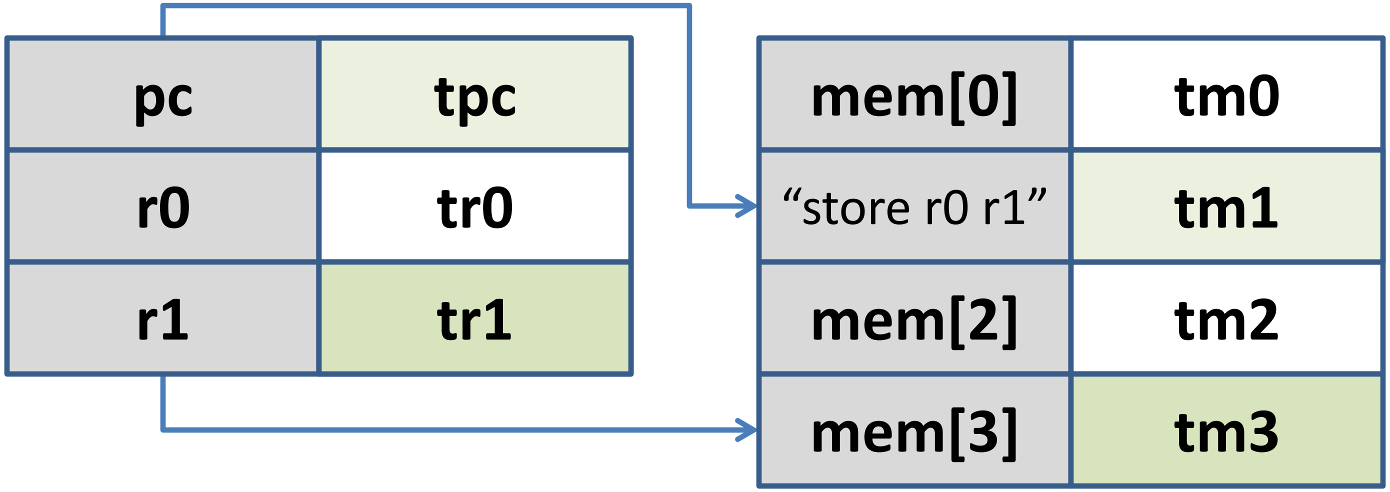
"store r0 r1"

mem[2]

mem[3]

Key enabler: Micro-Policies

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

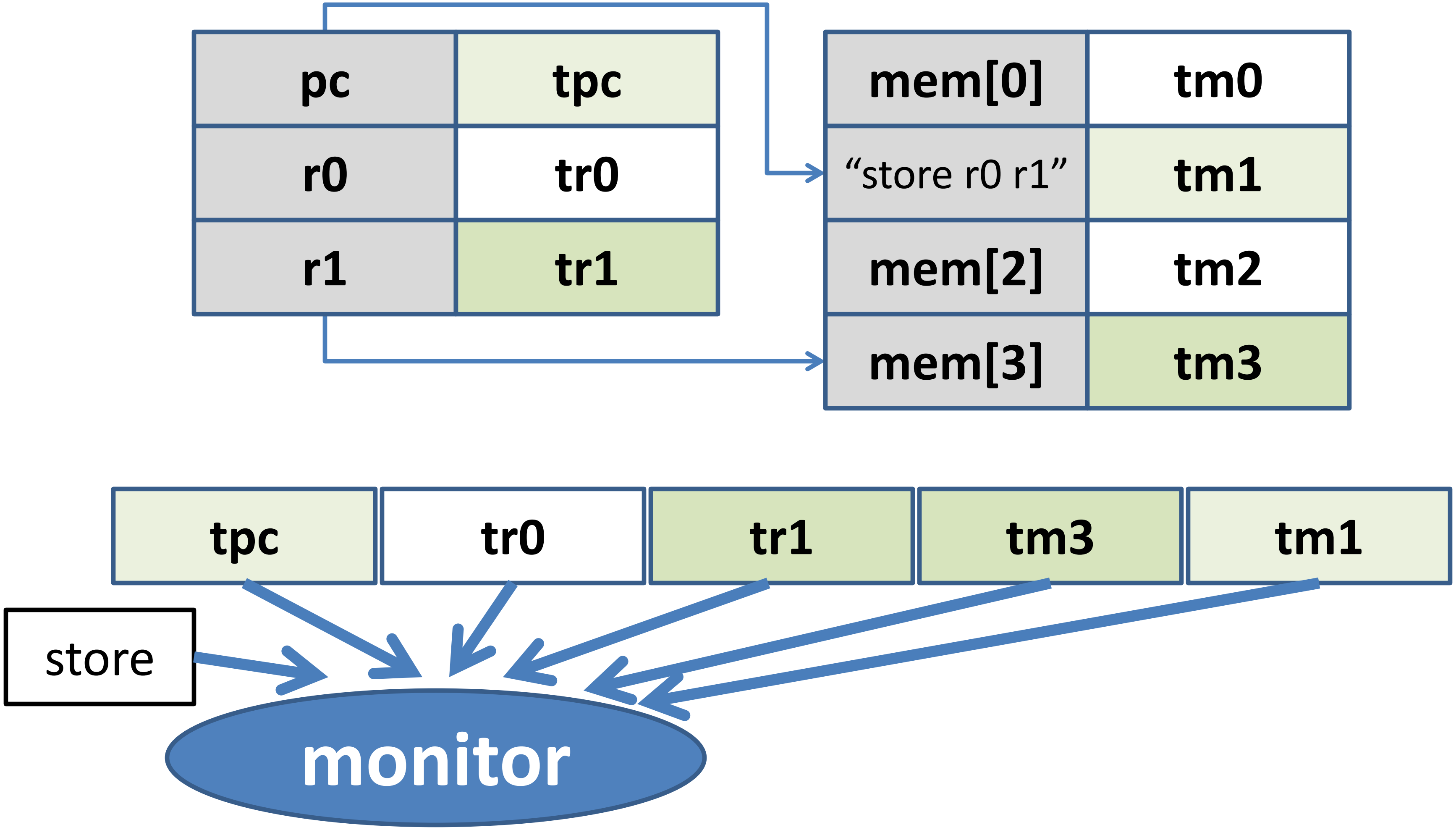


pc	tpc
r0	tr0
r1	tr1

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

Key enabler: Micro-Policies

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



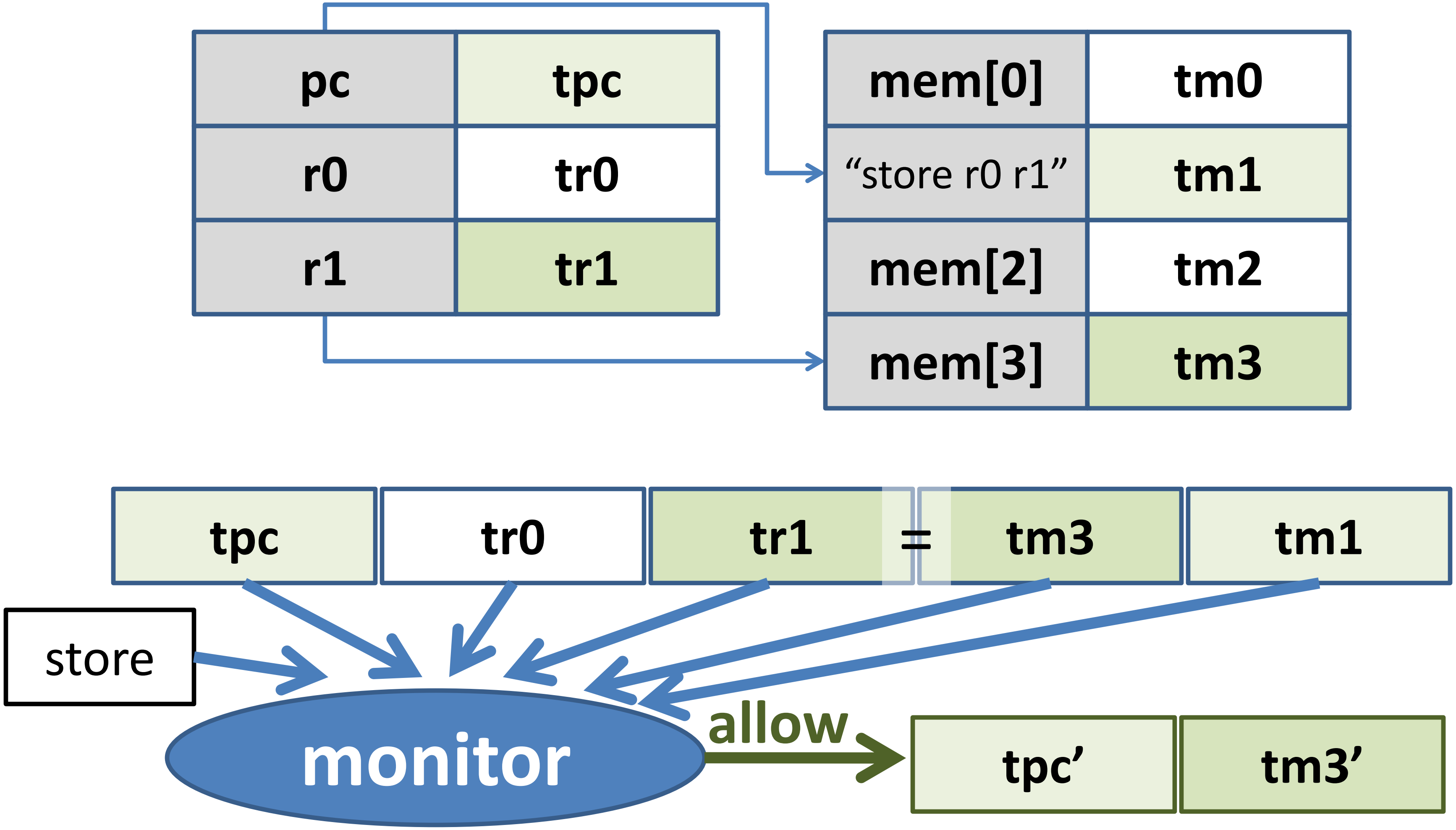
pc	tpc
r0	tr0
r1	tr1

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

tpc	tr0	tr1	tm3	tm1

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

tpc | tr0 | tr1 = tm3 | tm1

store

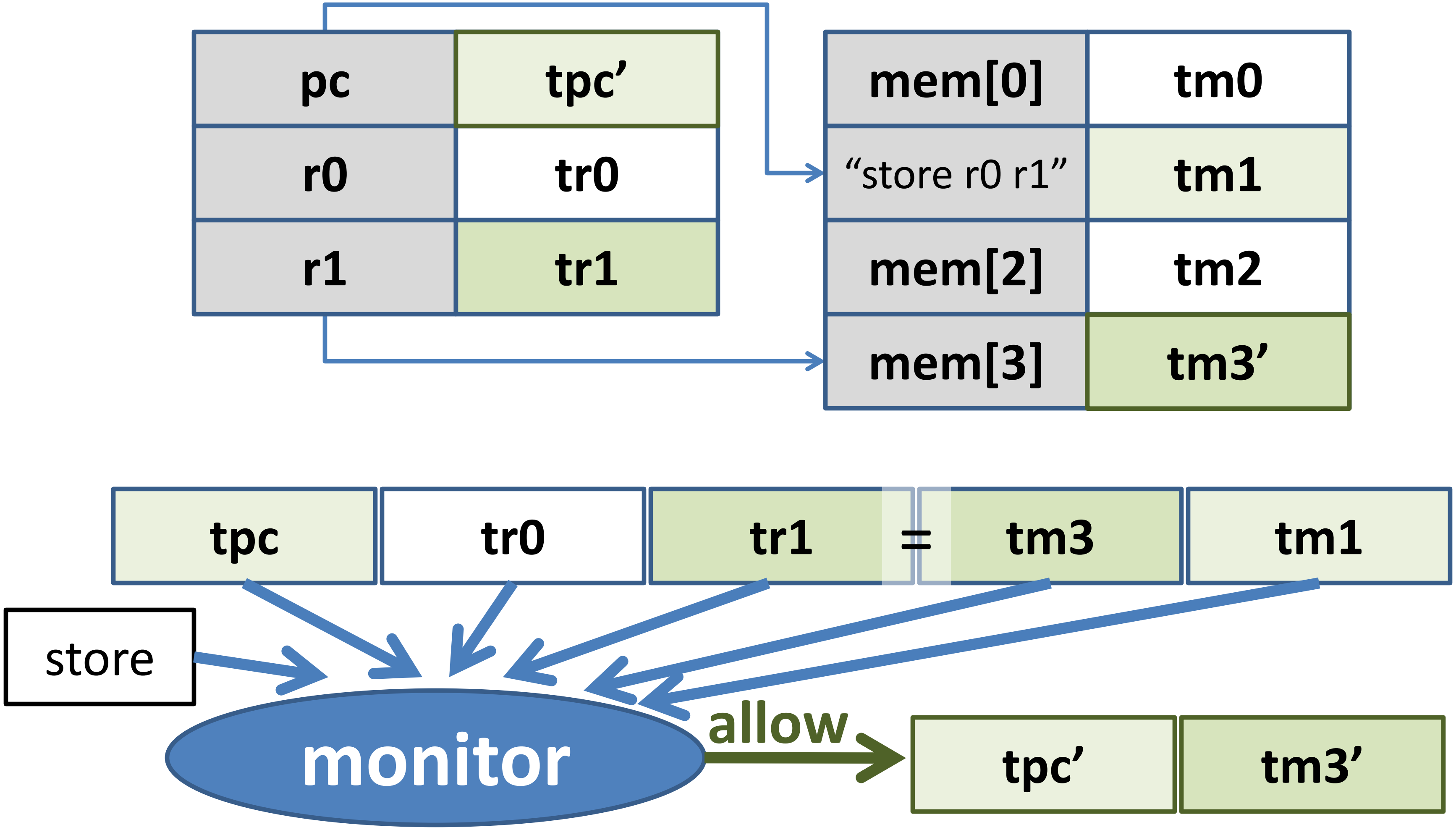
monitor

allow

tpc' | tm3'

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’

tpc

tr0

tr1 = tm3

tm1

store

monitor

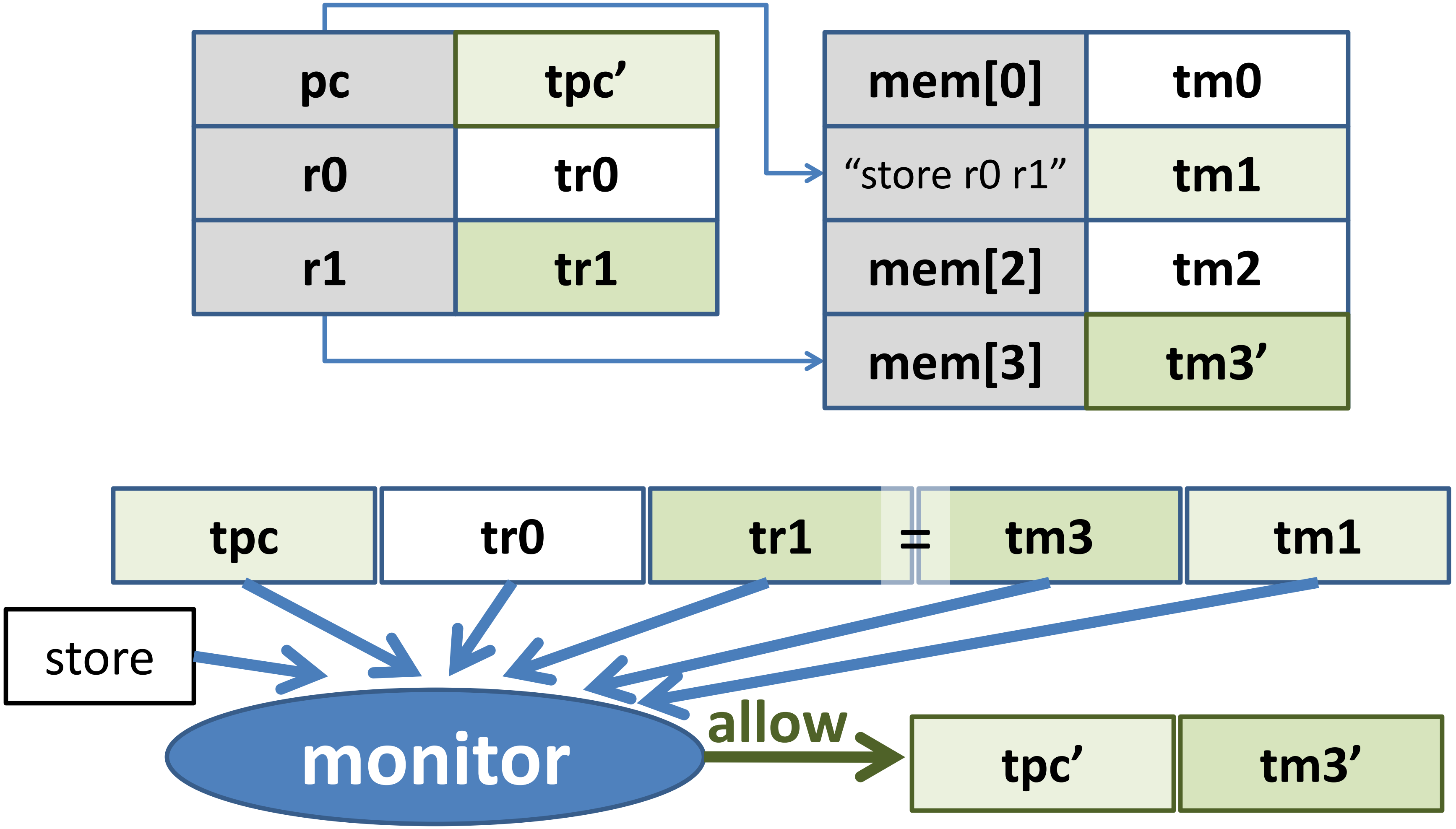
allow

tpc’

tm3’

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’

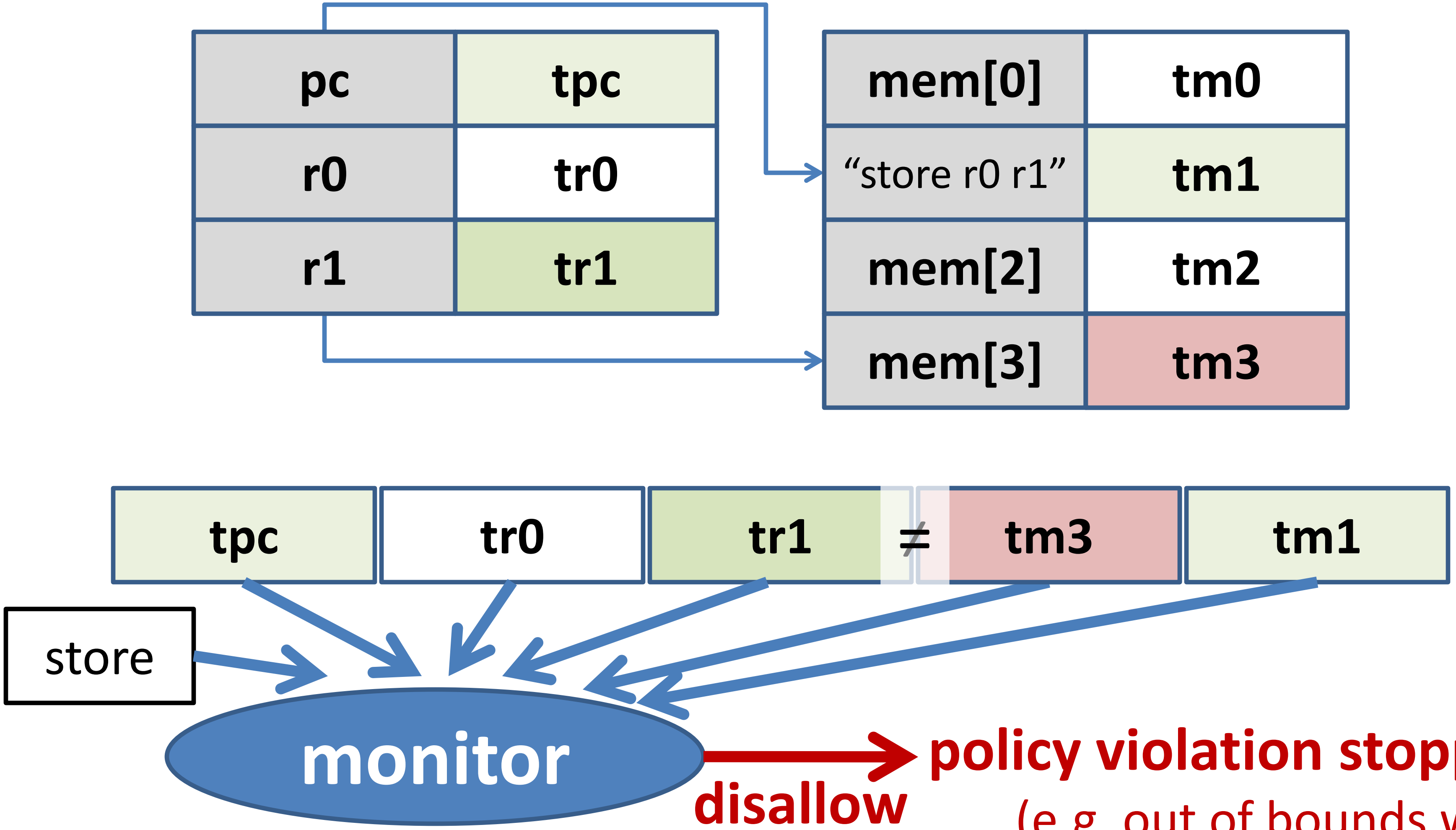
tpc	tr0	tr1 =	tm3	tm1

tpc’	tm3’

software monitor’s decision is hardware cached

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

tpc	tr0	tr1 ≠	tm3	tm1

disallow

policy violation stopped!

(e.g. out of bounds write)

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

spec

- **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

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

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
- ...

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]

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]

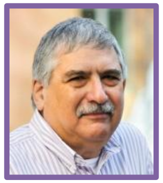
spec

Micro-Policies team

- **Formal methods** & **architecture** & **systems**
- **Current team**:
  - *Inria Paris*: **Cătălin Hrițcu, Marco Stronati** (until recently **Yannis Juglaret**, **Boris Eng**)
  - *UPenn*: **André DeHon**, **Benjamin Pierce, Arthur Azevedo de Amorim**, **Nick Roessler**
  - *Portland State*: **Andrew Tolmach**
  - *MIT:* **Howie Shrobe, Stelios Sidiroglou-Douskos**
  - *Industry*: **Draper Labs, Bluespec Inc**

Micro-Policies team

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



DRAPER

bluespec

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**

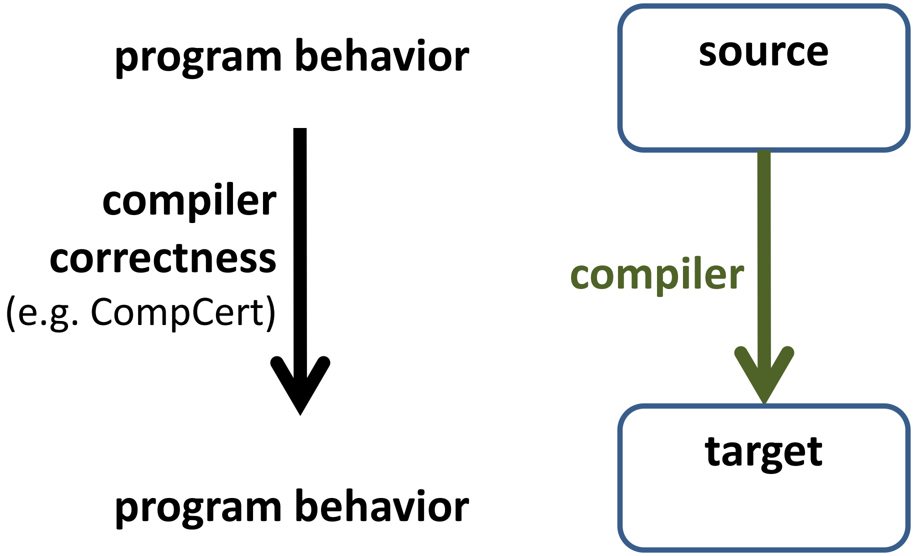
1. **Provide secure semantics for low-level languages**
   - C with protected components and memory safety
2. **Enforce secure interoperability with lower-level code**
   - ASM, C, and F* [= OCaml/F# + verification]

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



program behavior

compiler correctness (e.g. CompCert)

program behavior

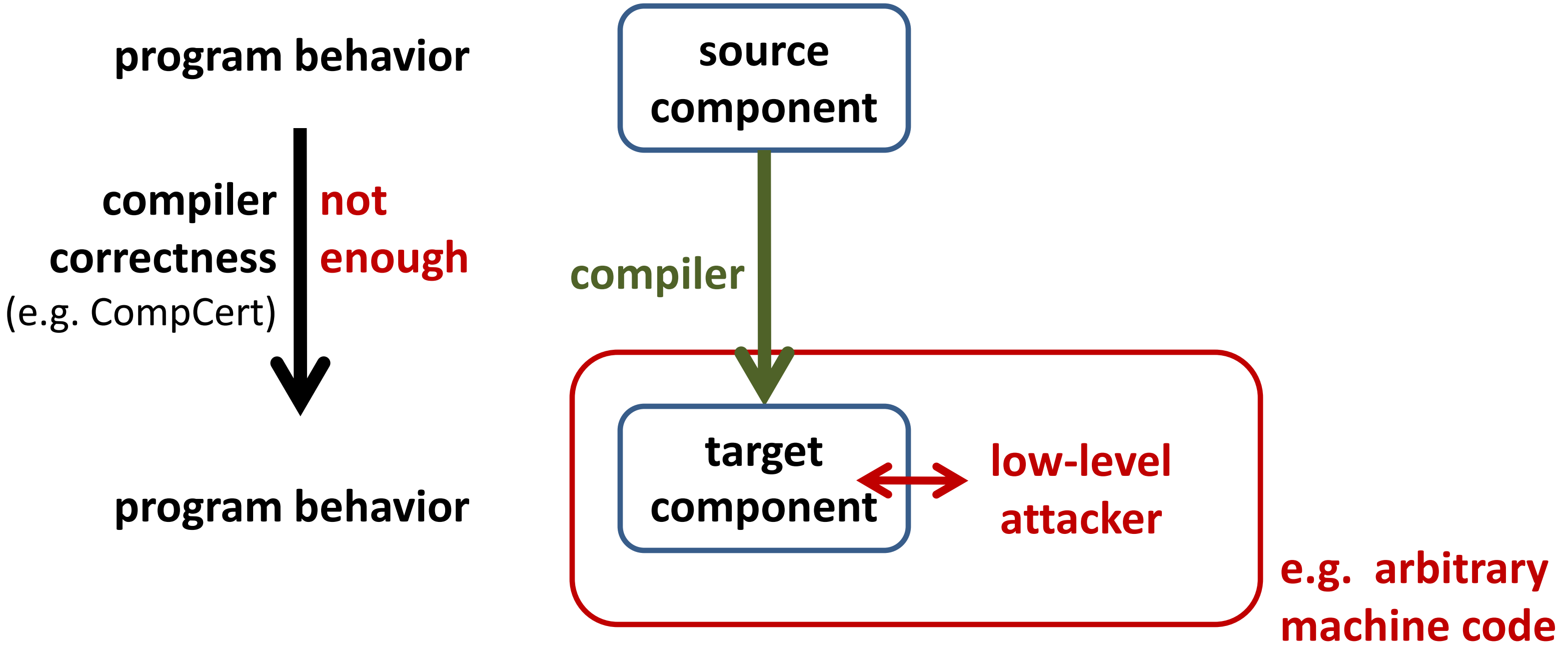
source

compiler

target

Formally verify: full abstraction

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



program behavior

compiler correctness (e.g. CompCert)

not enough

program behavior

source component

compiler

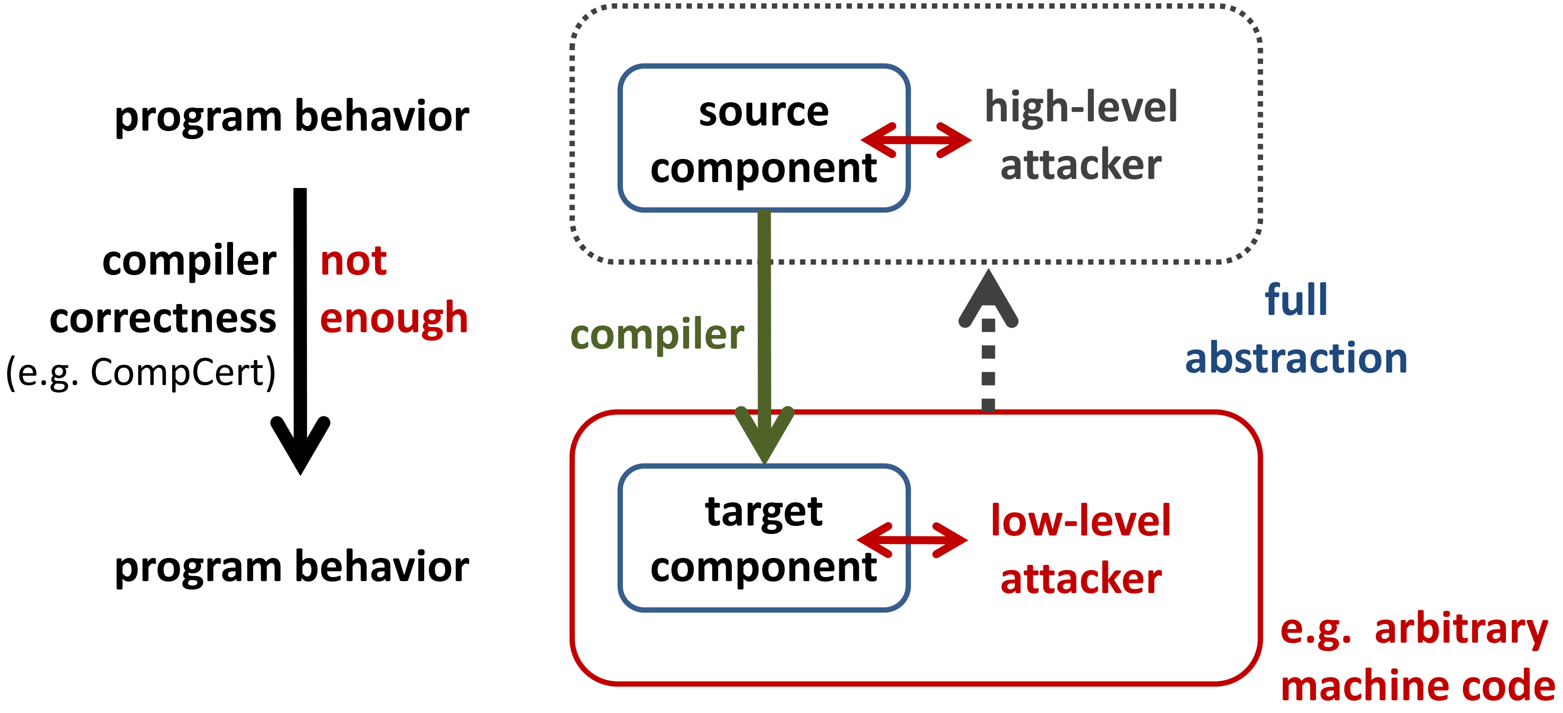
target component

low-level attacker

e.g. arbitrary machine code

Formally verify: full abstraction

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



program behavior

compiler correctness (e.g. CompCert)

not enough

program behavior

source component

high-level attacker

compiler

full abstraction

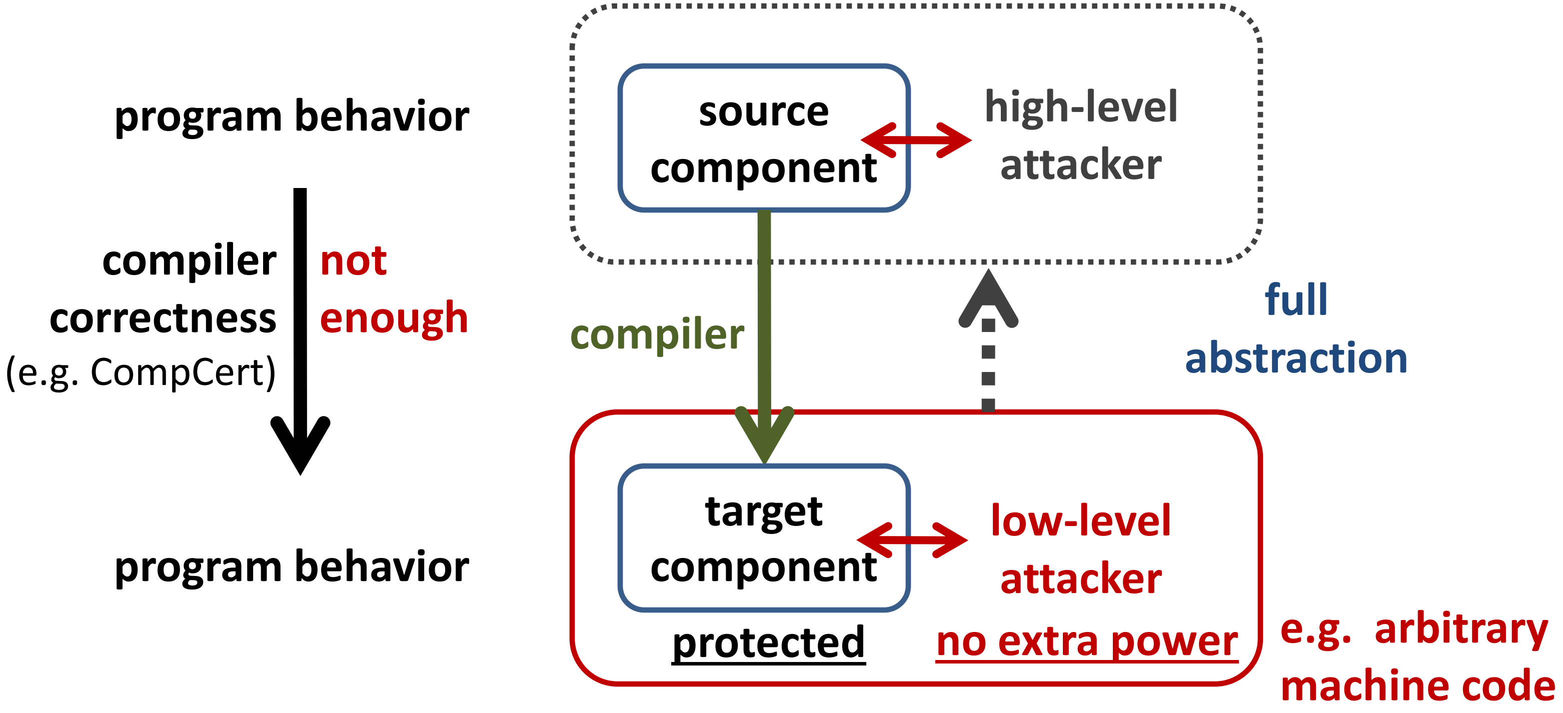
target component

low-level attacker

e.g. arbitrary machine code

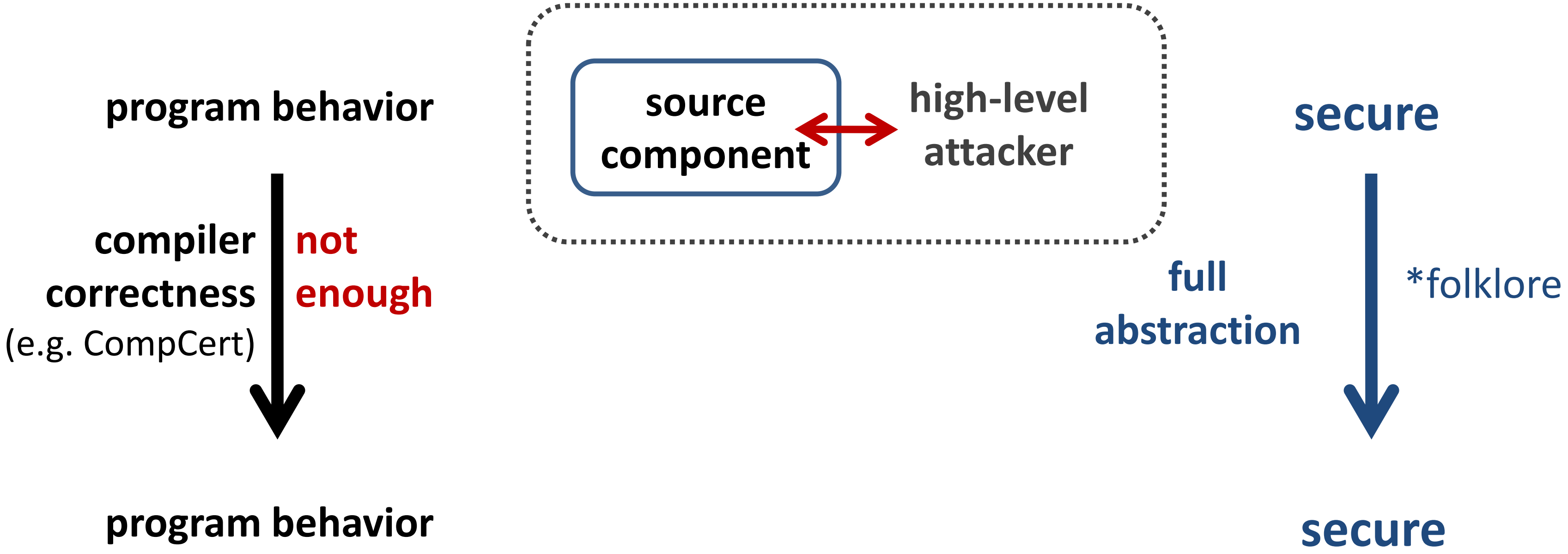
Formally verify: full abstraction

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



program behavior

compiler correctness (e.g. CompCert)

not enough

program behavior

source component

high-level attacker

full abstraction

secure

*folklore

secure

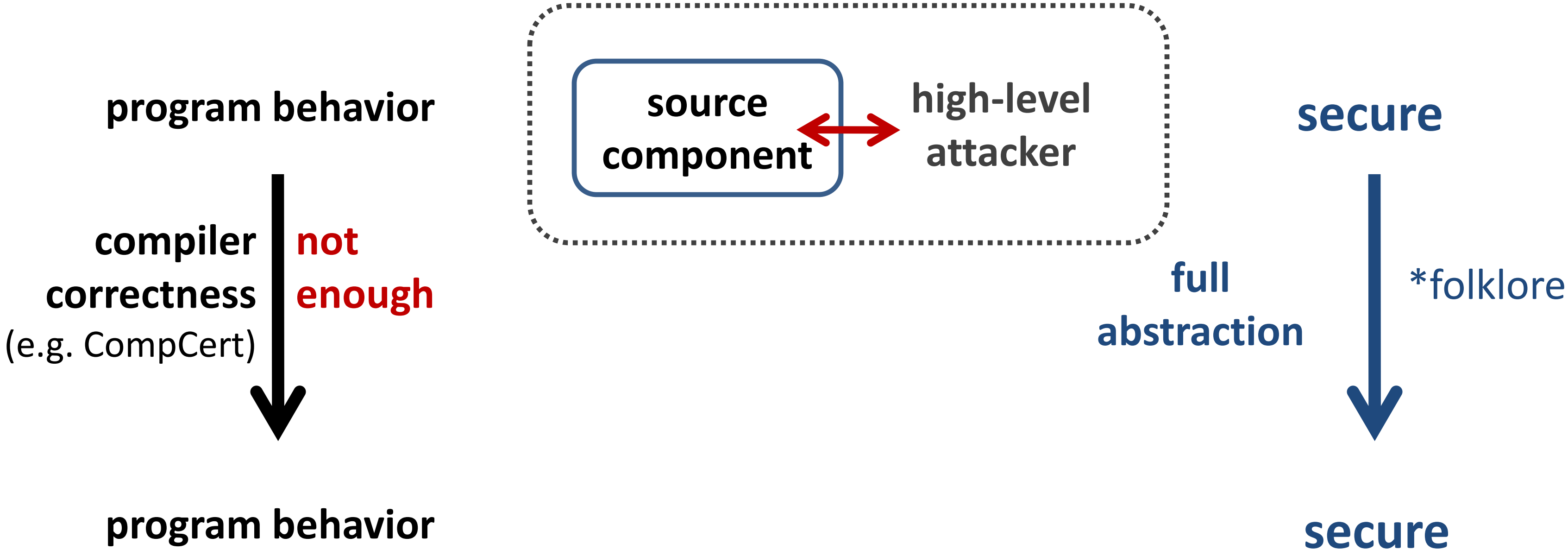
Benefit: sound security reasoning in the source language

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

forget that libraries are written in a lower-level language not efficiently achievable today

Formally verify: full abstraction

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

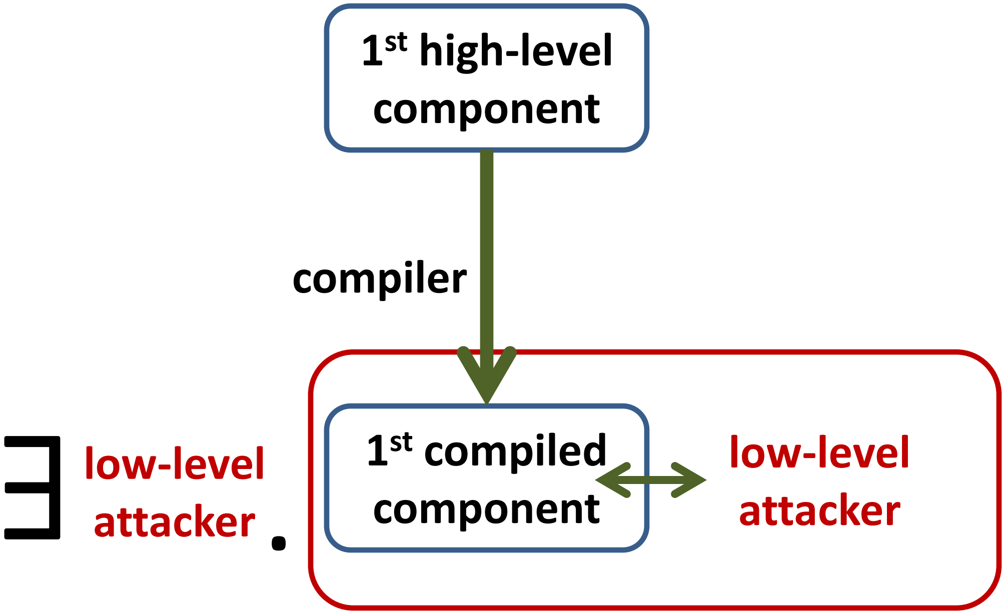


Benefit: sound security reasoning in the source language

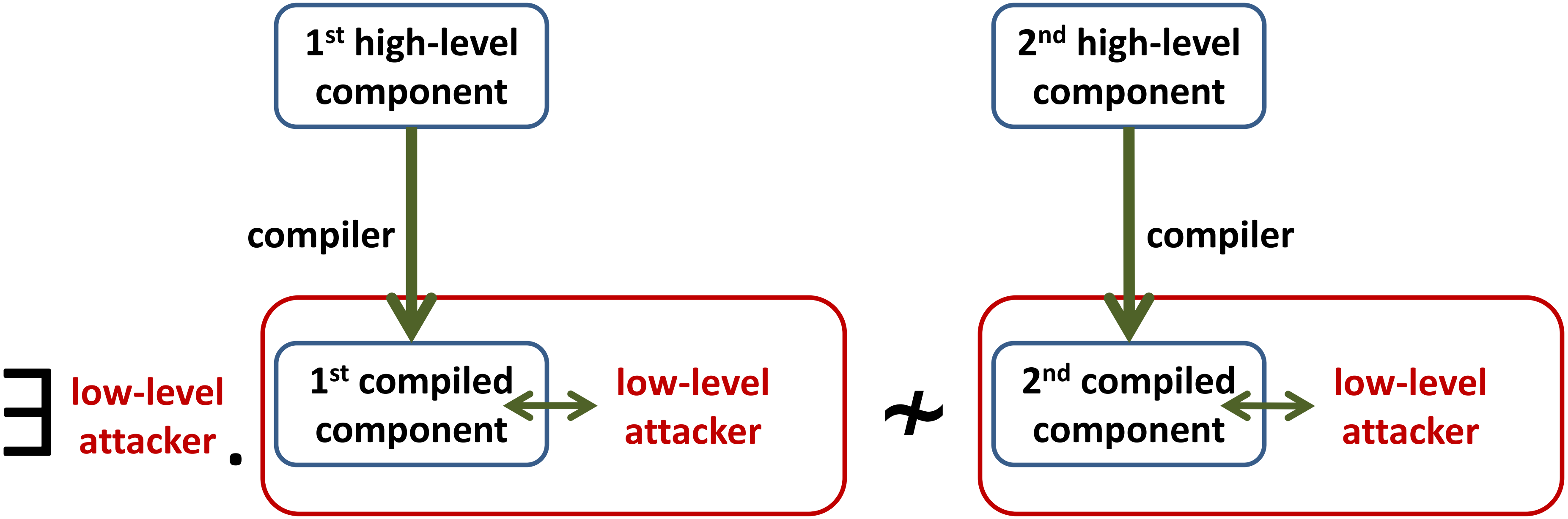
forget about compiler chain (linker, loader, runtime system)

forget that libraries are written in a lower-level language

Fully abstract compilation, definition



Fully abstract compilation, definition



1st high-level component

compiler

2nd high-level component

compiler

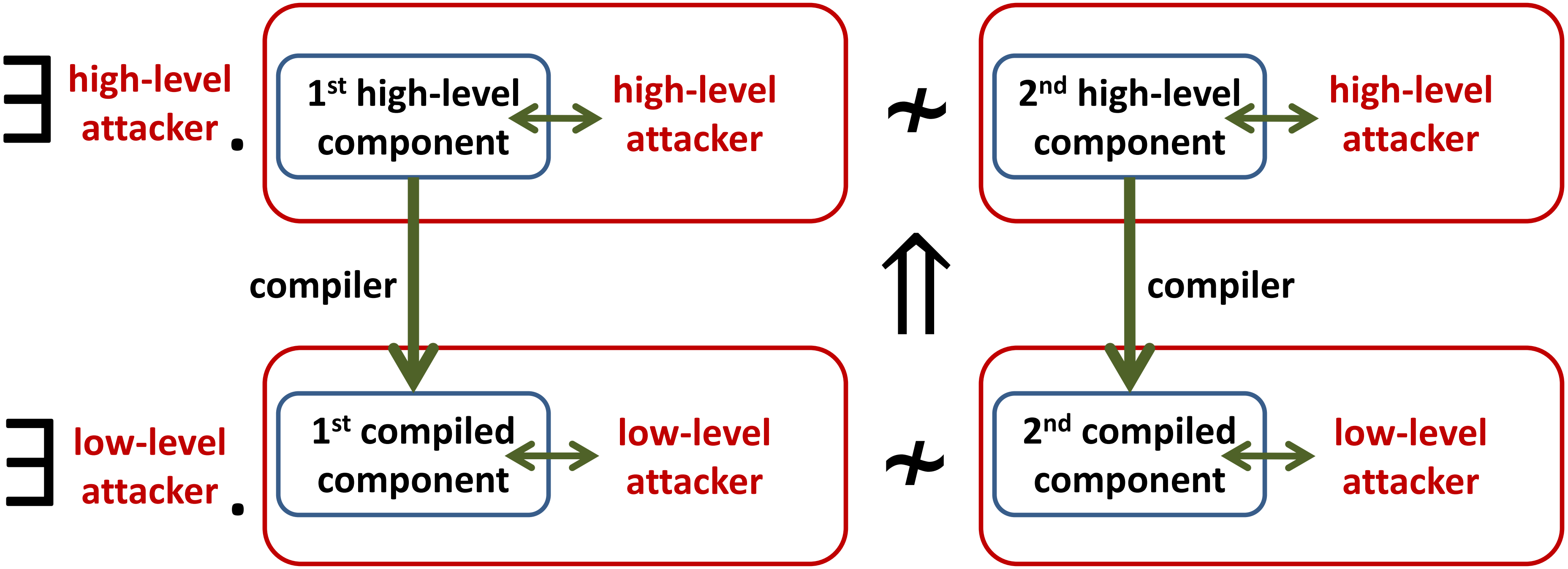
∃ low-level attacker .

1st compiled component ↔ low-level attacker

≁

2nd compiled component ↔ low-level attacker

Fully abstract compilation, definition



$\exists$ high-level attacker .

1st high-level component ↔ high-level attacker

$\nsim$

2nd high-level component ↔ high-level attacker

compiler

$\Uparrow$

compiler

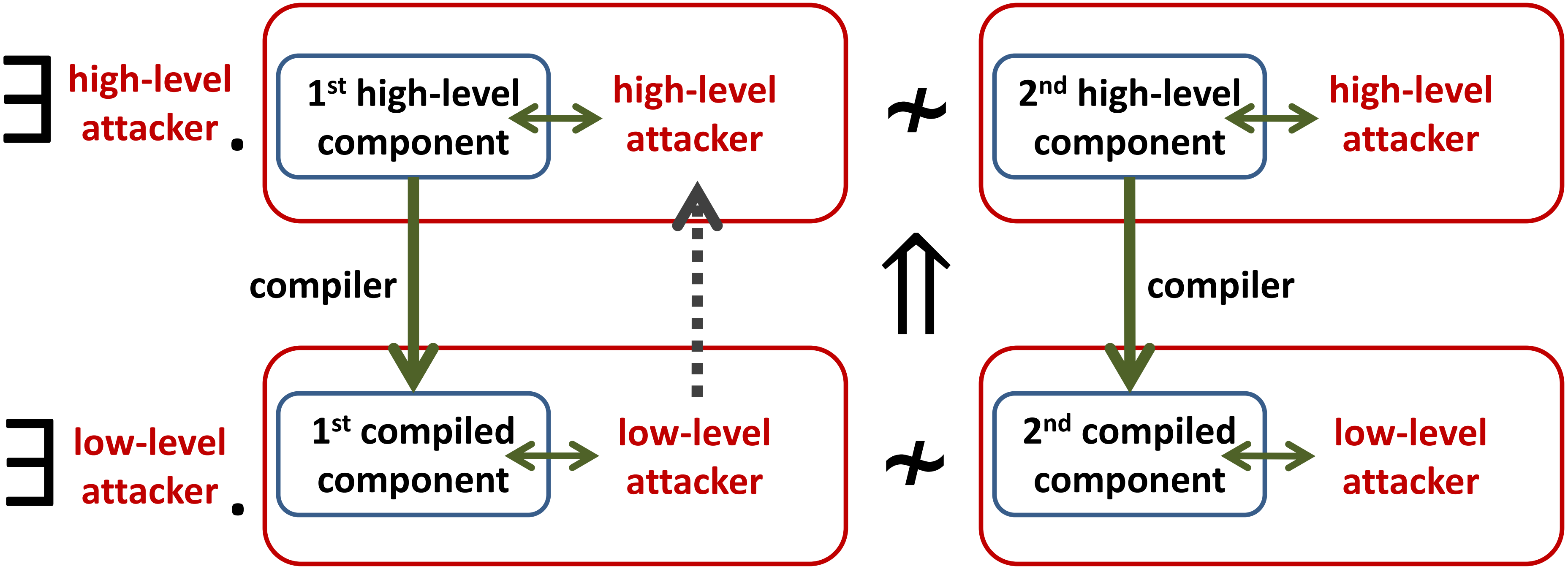
$\exists$ low-level attacker .

1st compiled component ↔ low-level attacker

$\nsim$

2nd compiled component ↔ low-level attacker

Fully abstract compilation, definition



SECOMP: achieving full abstraction at scale

F* language
(OCaml/F# + verification)

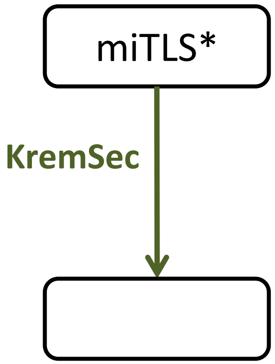
miTLS*

C language
+ memory safety
+ components

SECOMP: achieving full abstraction at scale

F* language
(OCaml/F# + verification)

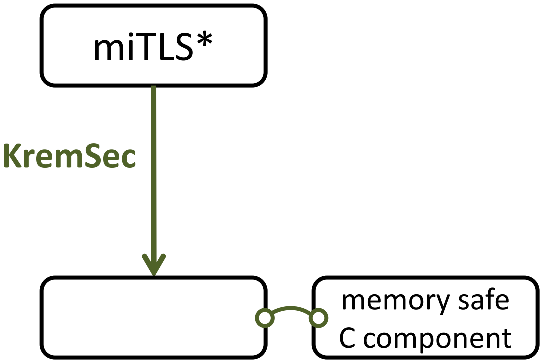
C language
+ memory safety
+ components



SECOMP: achieving full abstraction at scale

F* language
(OCaml/F# + verification)

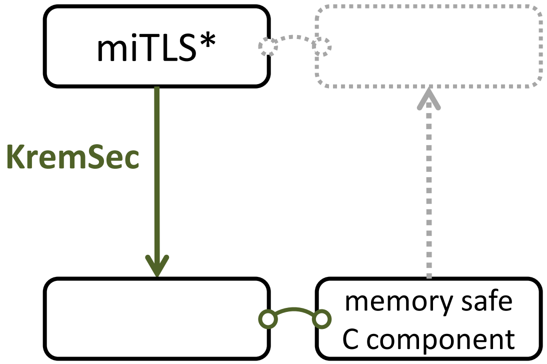
C language
+ memory safety
+ components



SECOMP: achieving full abstraction at scale

F* language
(OCaml/F# + verification)

C language
+ memory safety
+ components

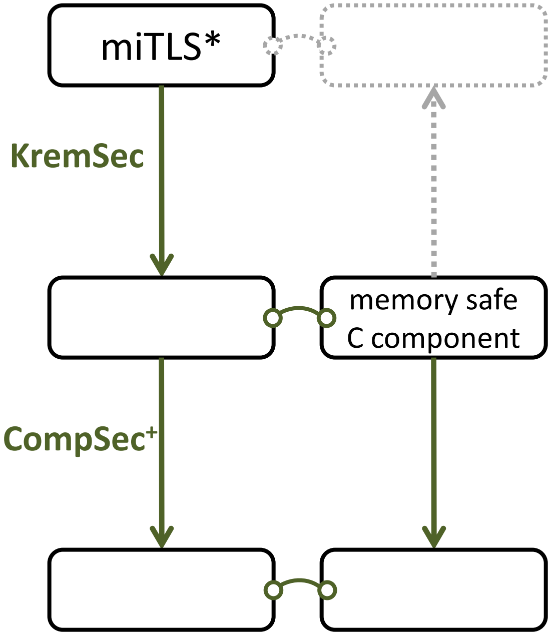


SECOMP: achieving full abstraction at scale

F* language
(OCaml/F# + verification)

C language
+ memory safety
+ components

ASM language
(RISC-V + micro-policies)

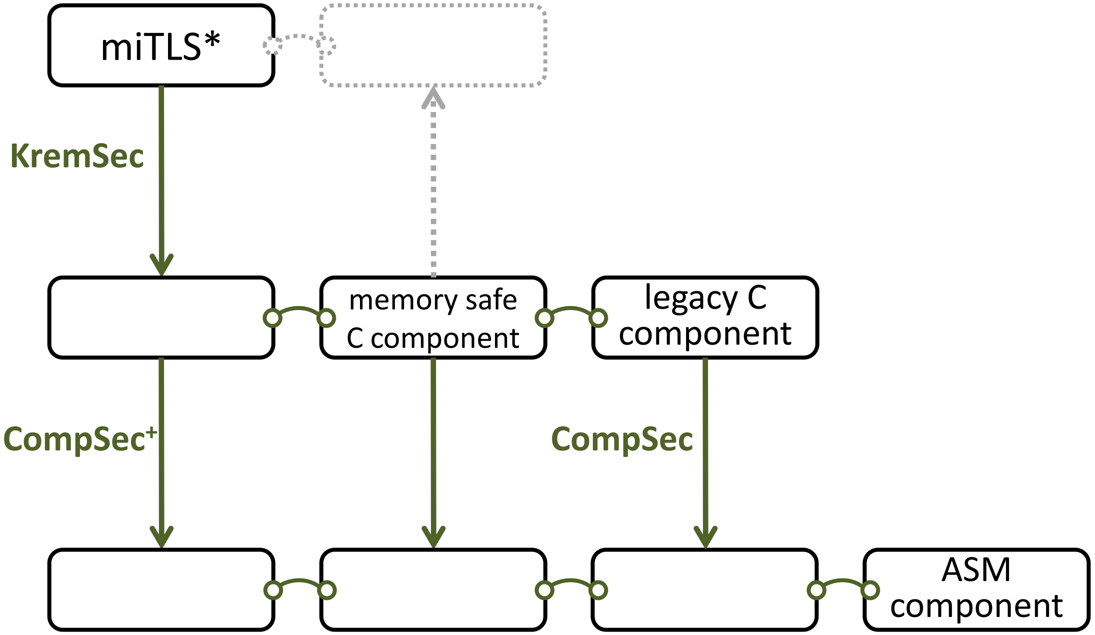


SECOMP: achieving full abstraction at scale

F* language
(OCaml/F# + verification)

C language
+ memory safety
+ components

ASM language
(RISC-V + micro-policies)

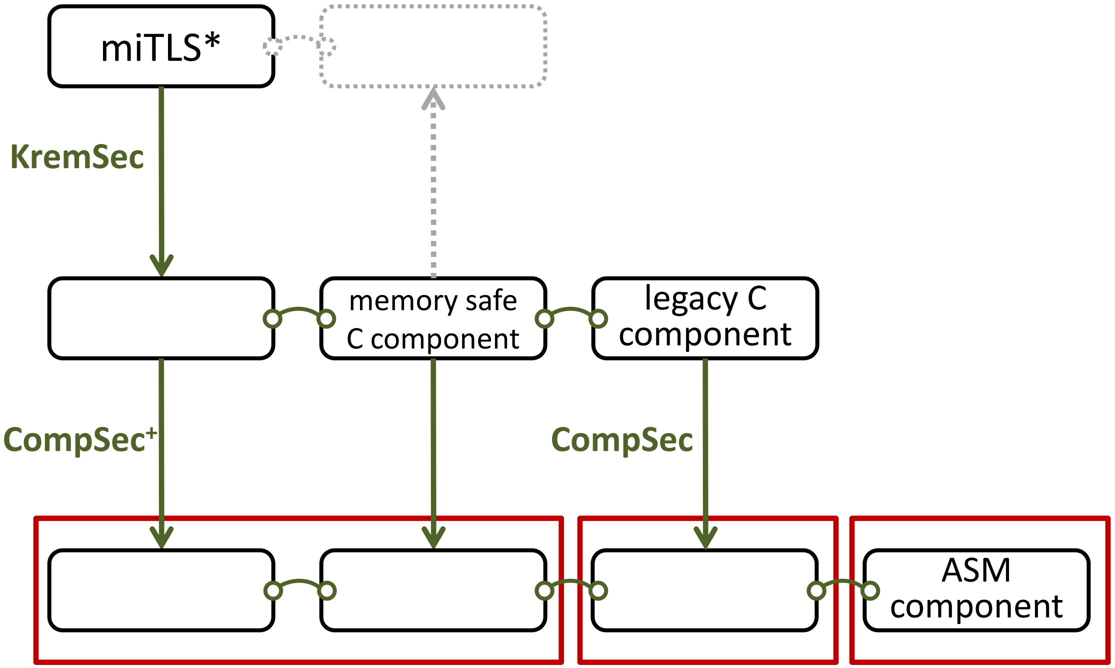


SECOMP: achieving full abstraction at scale

F* language
(OCaml/F# + verification)

C language
+ memory safety
+ components

ASM language
(RISC-V + micro-policies)



miTLS*

KremSec

memory safe C component

legacy C component

CompSec$^+$

CompSec

ASM component

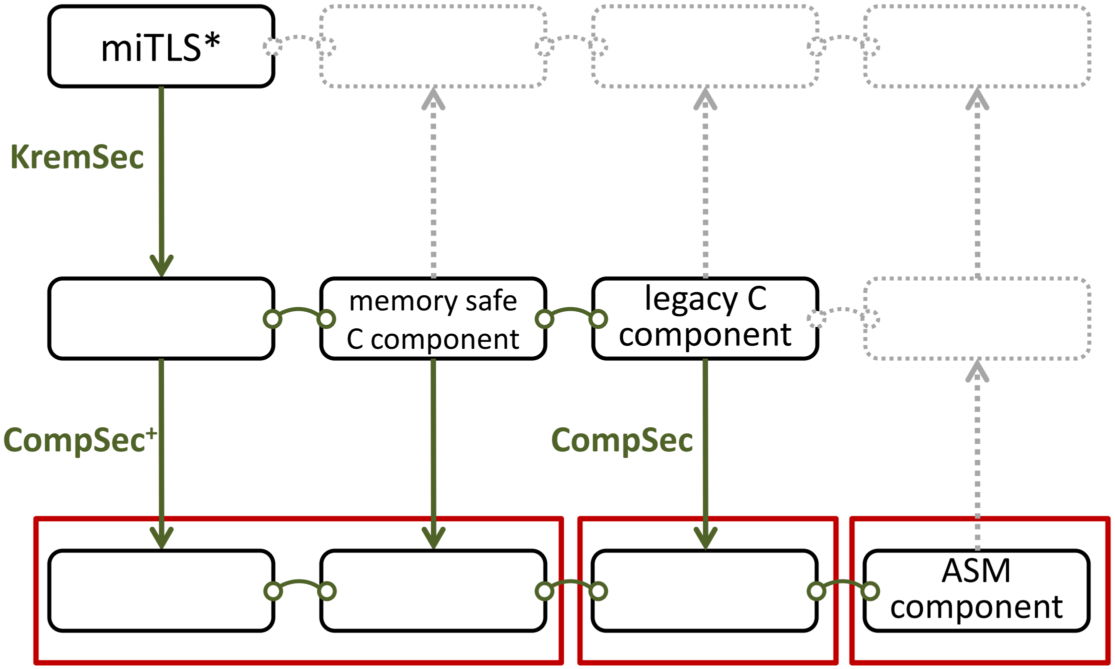
protecting component boundaries

SECOMP: achieving full abstraction at scale

F* language
(OCaml/F# + verification)

C language
+ memory safety
+ components

ASM language
(RISC-V + micro-policies)



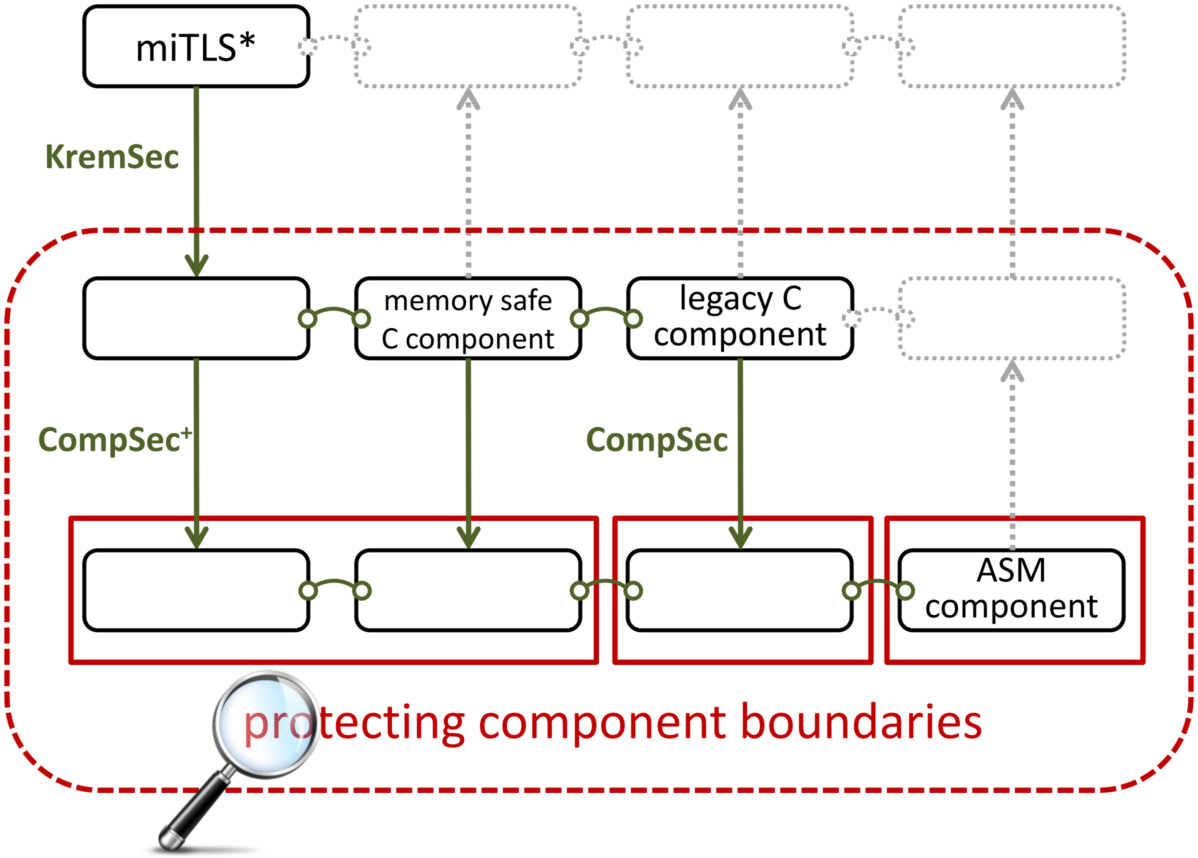
protecting component boundaries

SECOMP: achieving full abstraction at scale

F* language
(OCaml/F# + verification)

C language
+ memory safety
+ components

ASM language
(RISC-V + micro-policies)

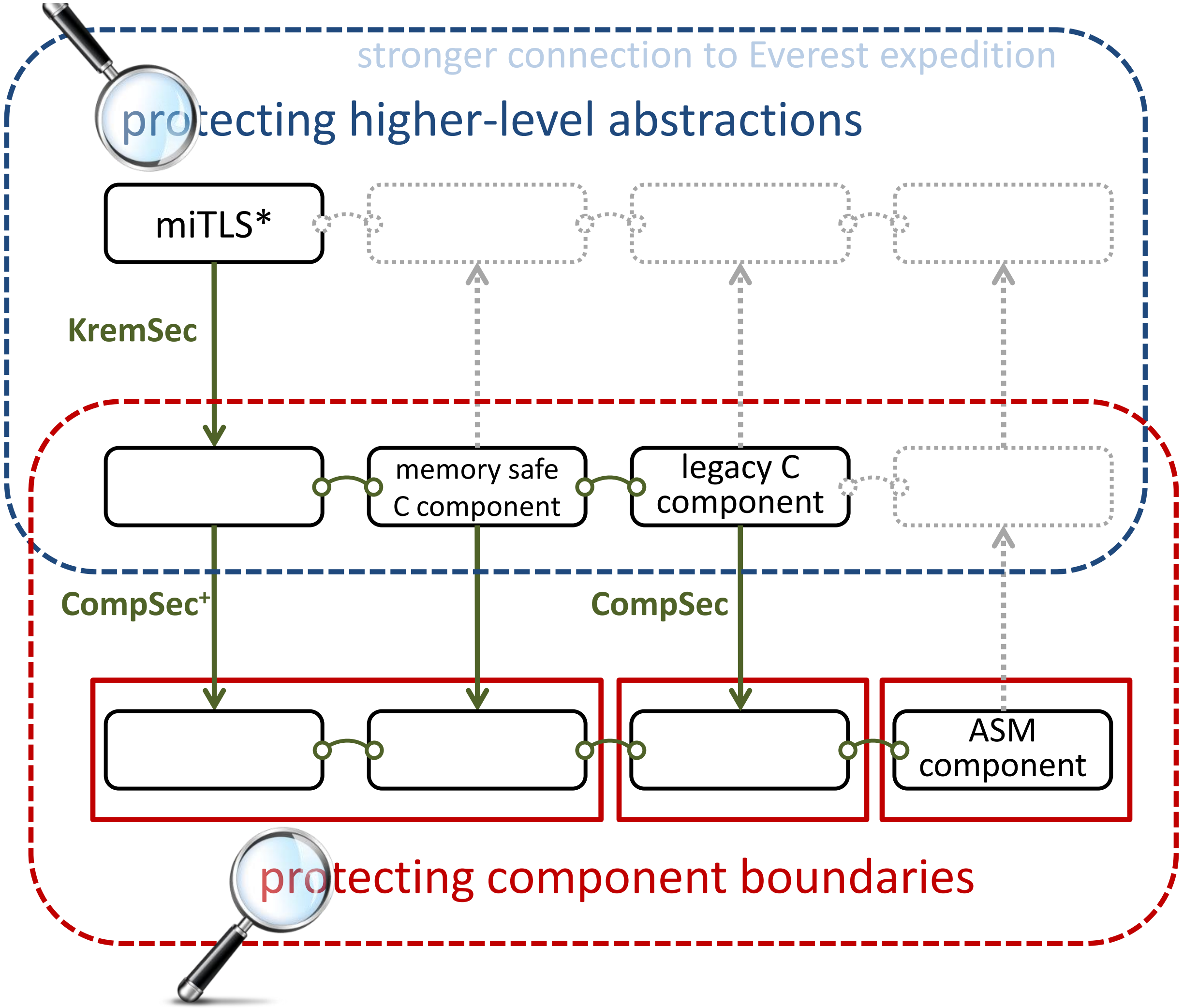


SECOMP: achieving full abstraction at scale

F* language
(OCaml/F# + verification)

C language
+ memory safety
+ components

ASM language
(RISC-V + micro-policies)


stronger connection to Everest expedition
protecting higher-level abstractions
miTLS*
KremSec
memory safe
C component
legacy C
component
CompSec$^+$
CompSec
ASM
component
protecting component boundaries

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

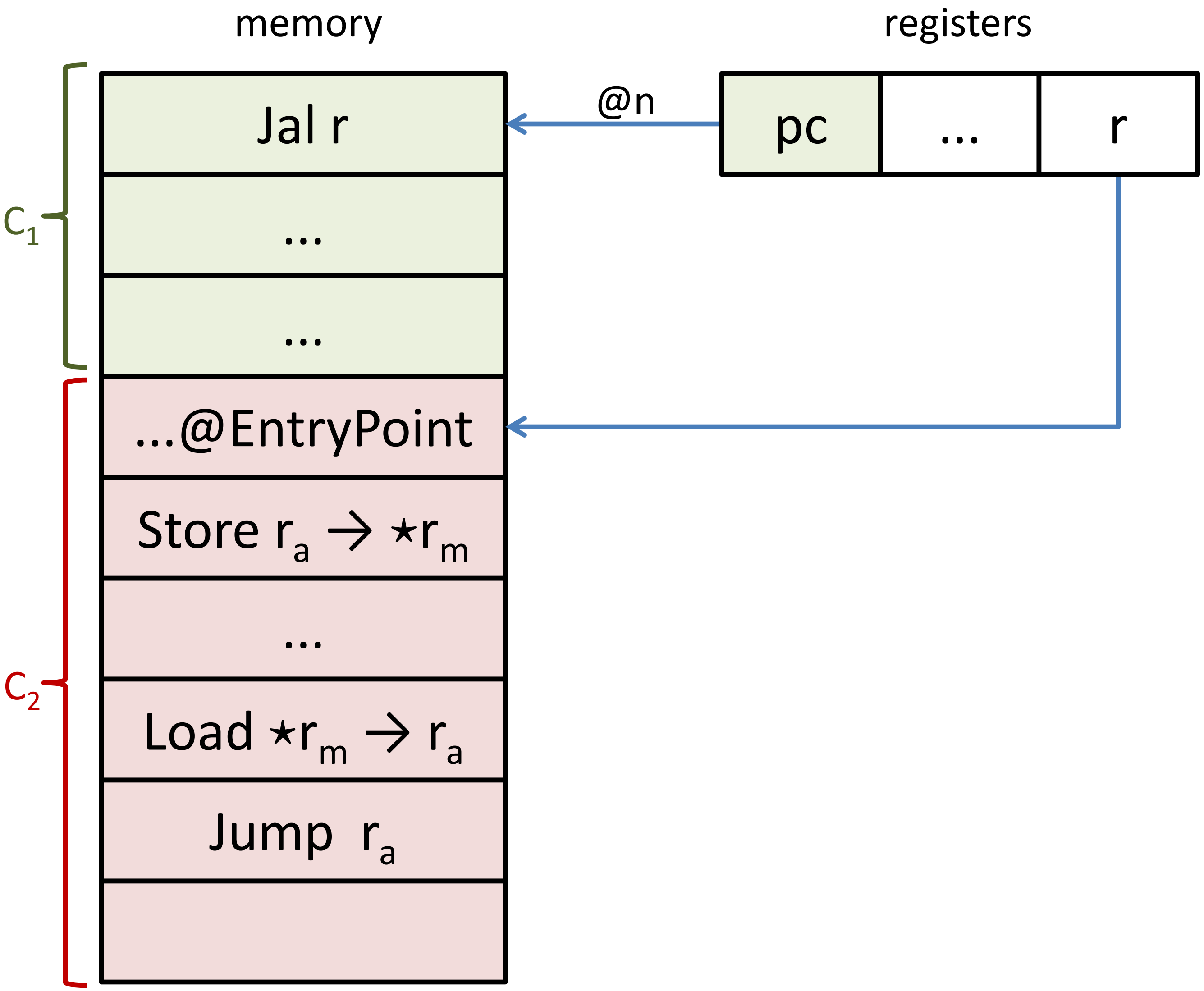
- **Add mutually distrustful components to C**
  - interacting only via **strictly enforced interfaces**

- **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

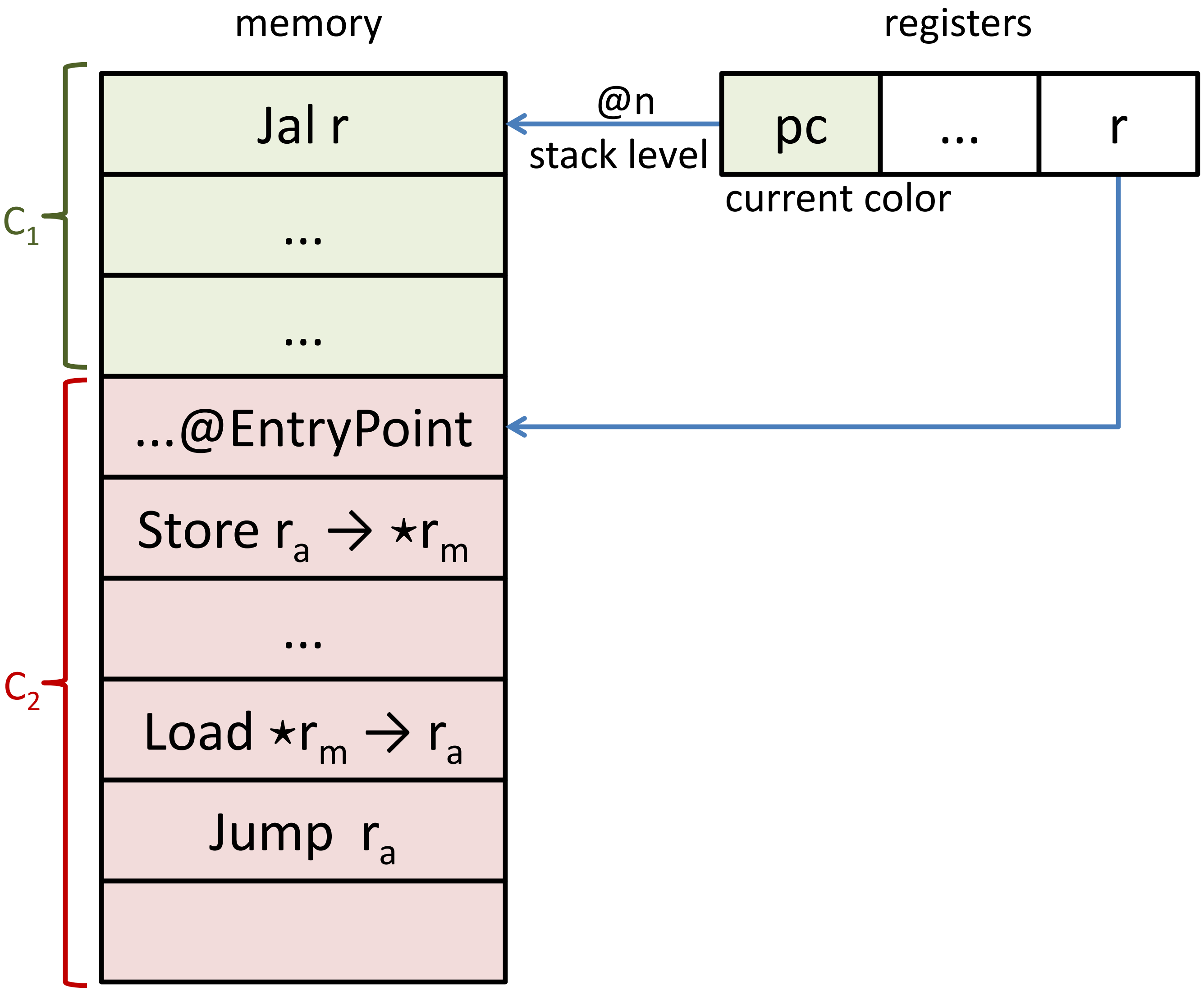
Recent work, joint with Yannis Juglaret et al

Compartmentalization micro-policy



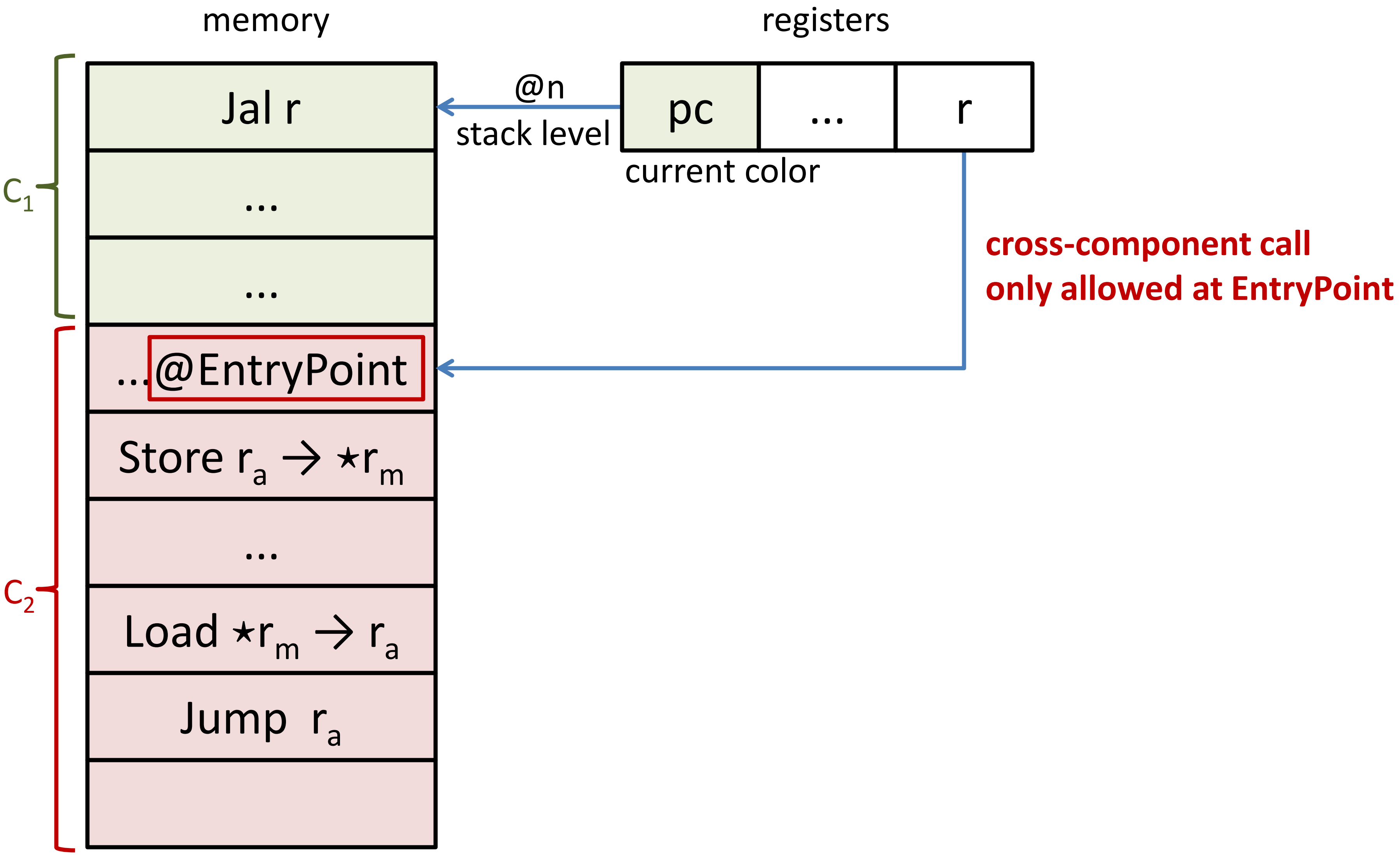
[Towards a Fully Abstract Compiler Using Micro-Policies, Juglaret et al, TR 2015]

Compartmentalization micro-policy



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

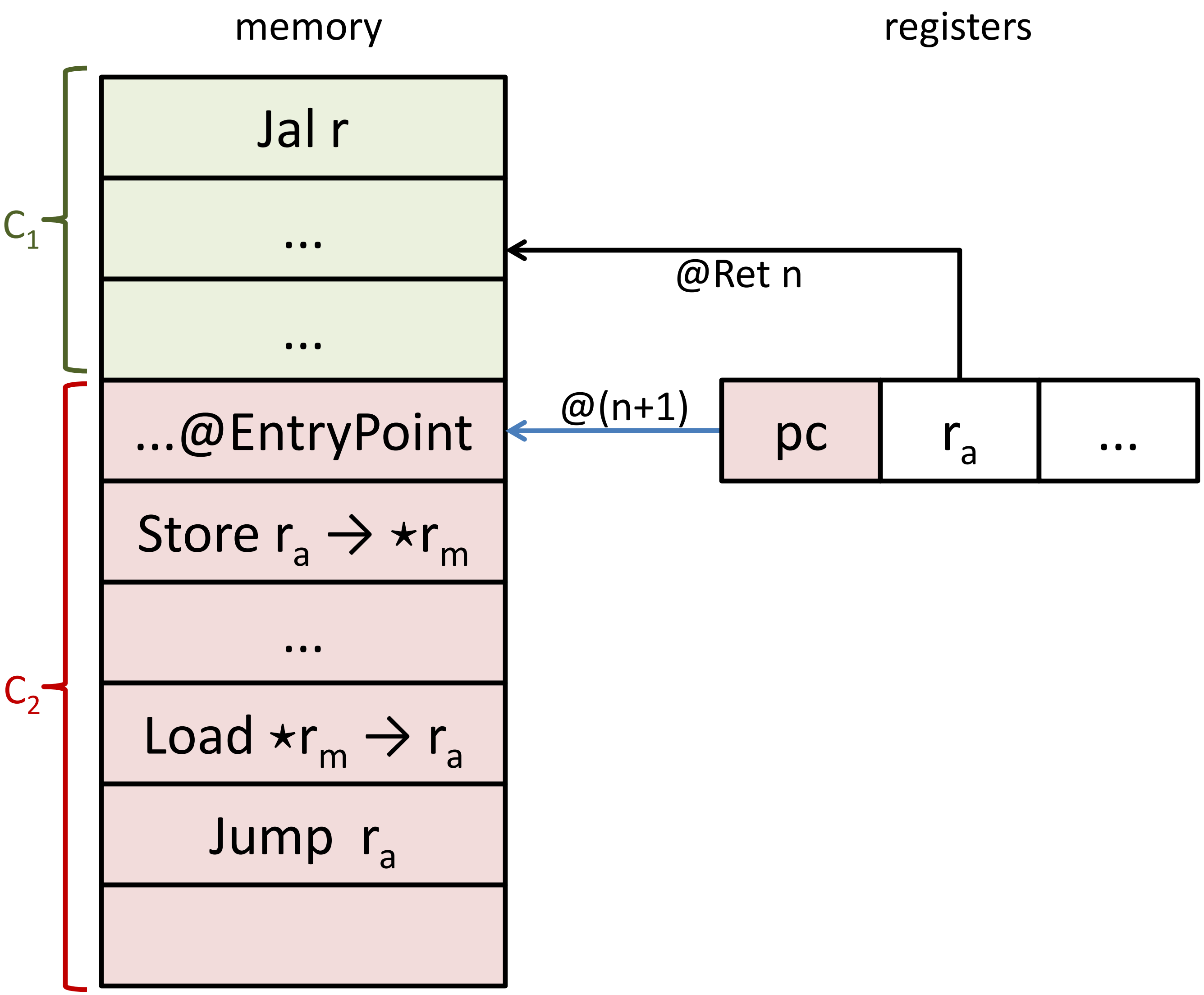
Compartmentalization micro-policy



cross-component call only allowed at EntryPoint

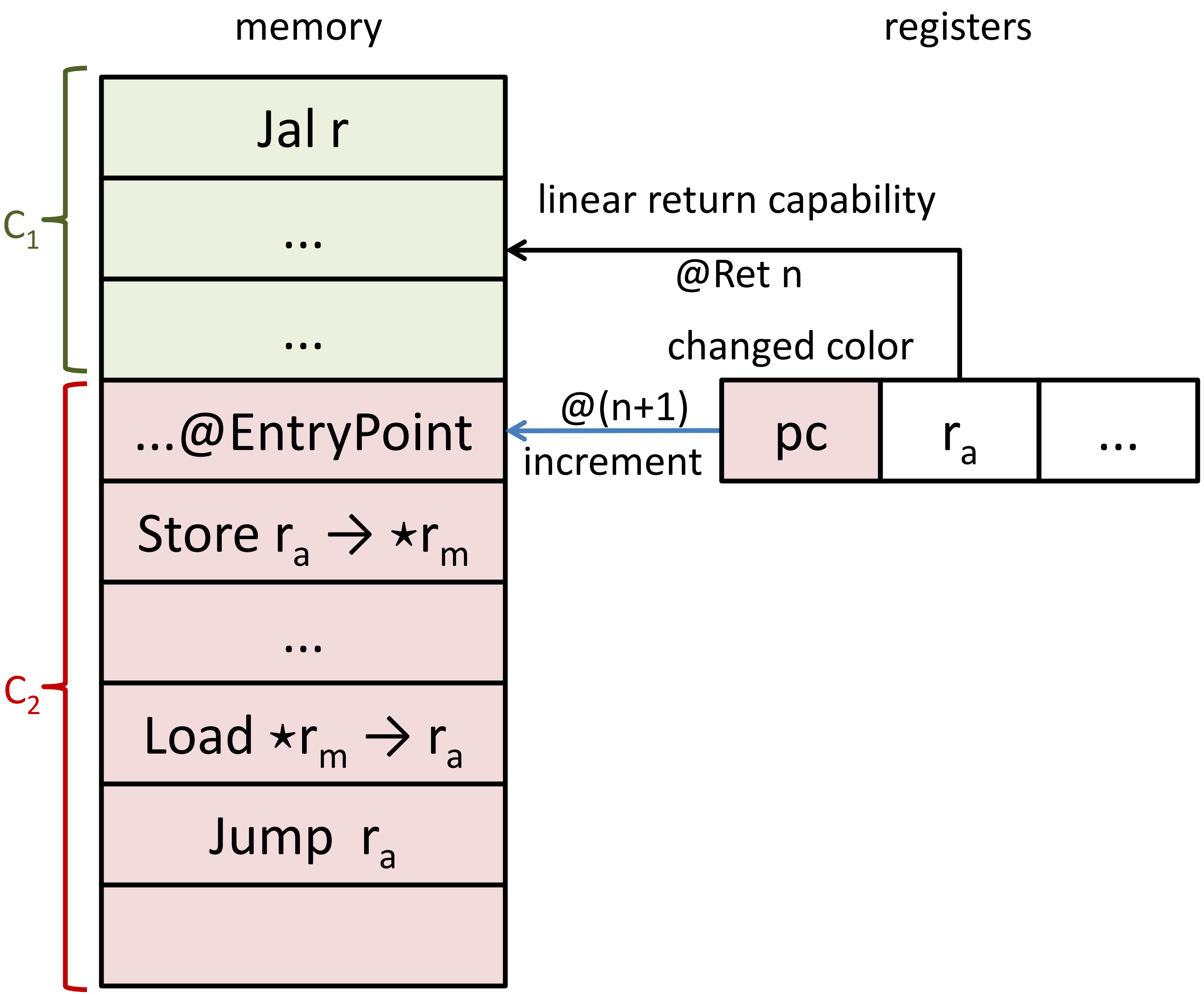
[Towards a Fully Abstract Compiler Using Micro-Policies, Juglaret et al, TR 2015]

Compartmentalization micro-policy



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

Compartmentalization micro-policy



memory

registers

C_1

Jal r

...

...

linear return capability

@Ret n

changed color

C_2

...@EntryPoint

@(n+1)

increment

pc

r_a

...

Store $r_a \rightarrow \star r_m$

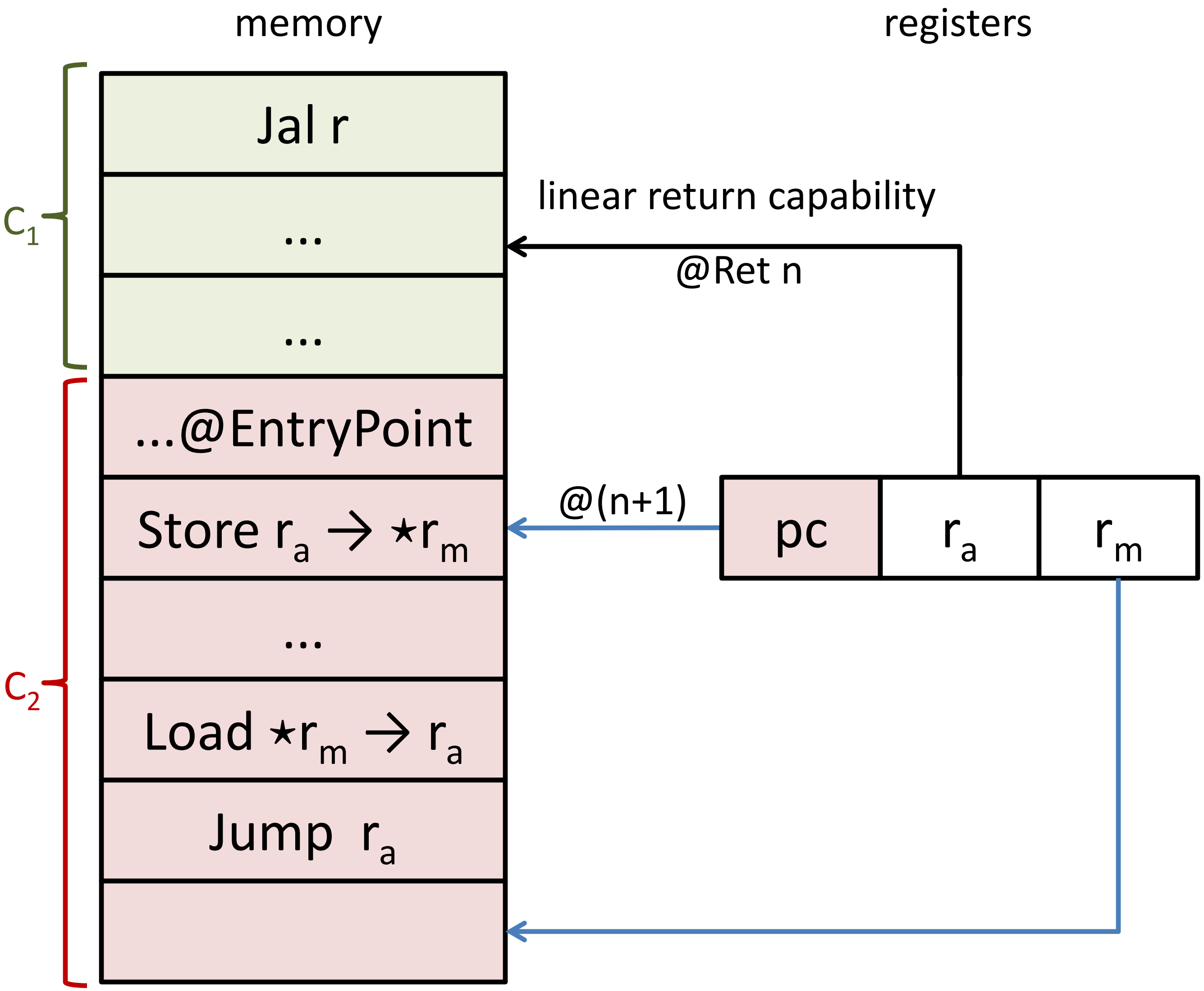
...

Load $\star r_m \rightarrow r_a$

Jump r_a

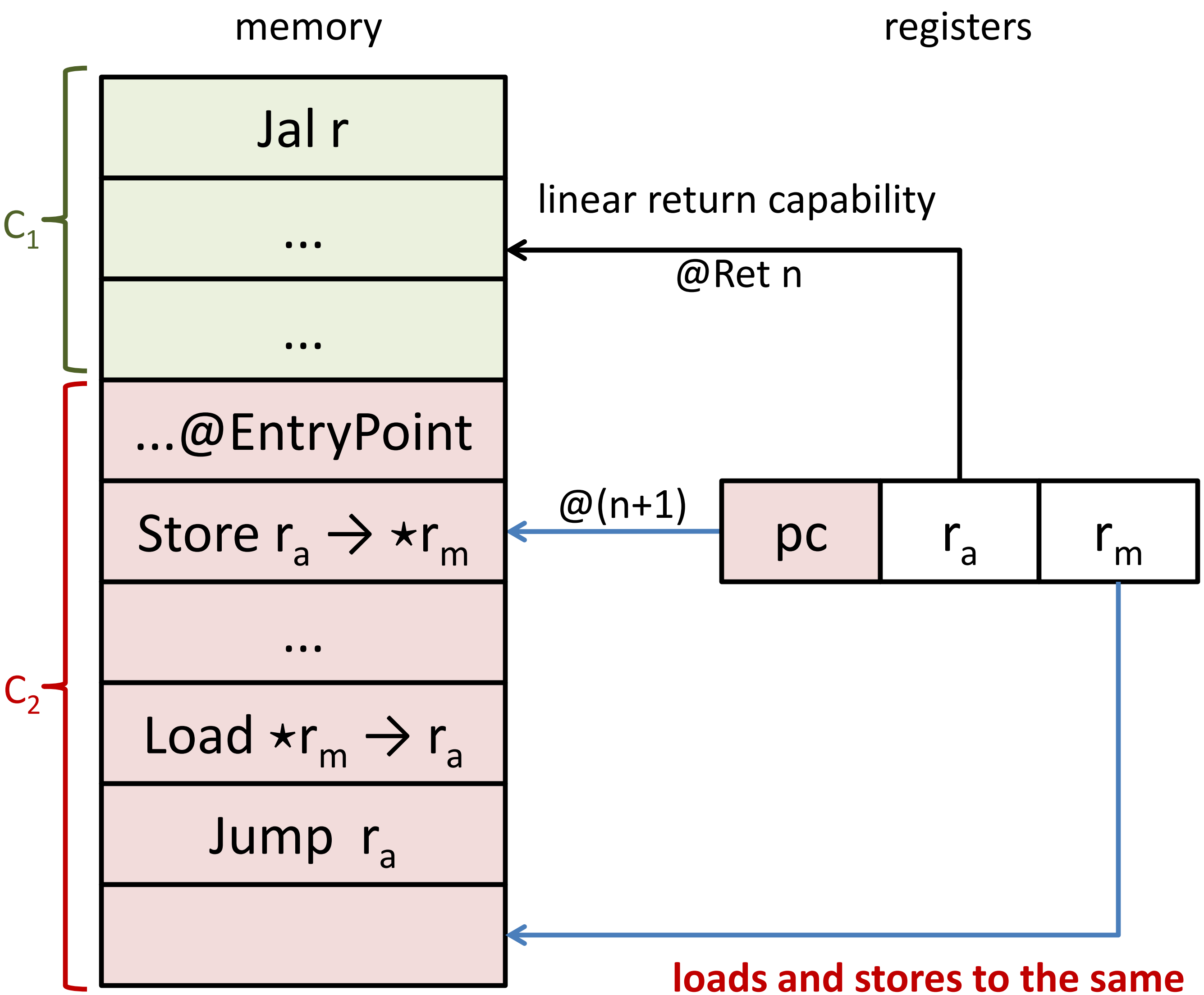
[Towards a Fully Abstract Compiler Using Micro-Policies, Juglaret et al, TR 2015]

Compartmentalization micro-policy



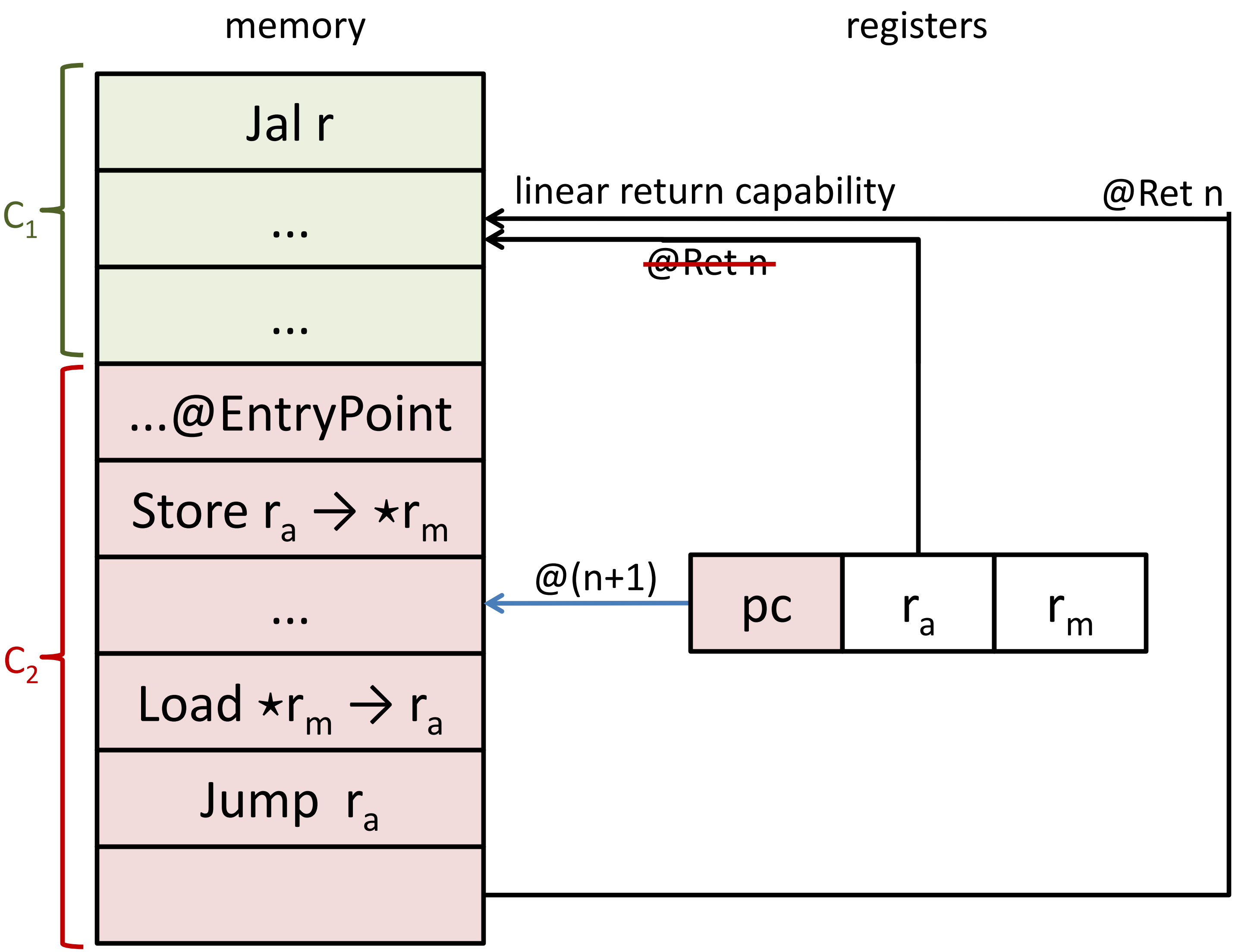
[Towards a Fully Abstract Compiler Using Micro-Policies, Juglaret et al, TR 2015]

Compartmentalization micro-policy

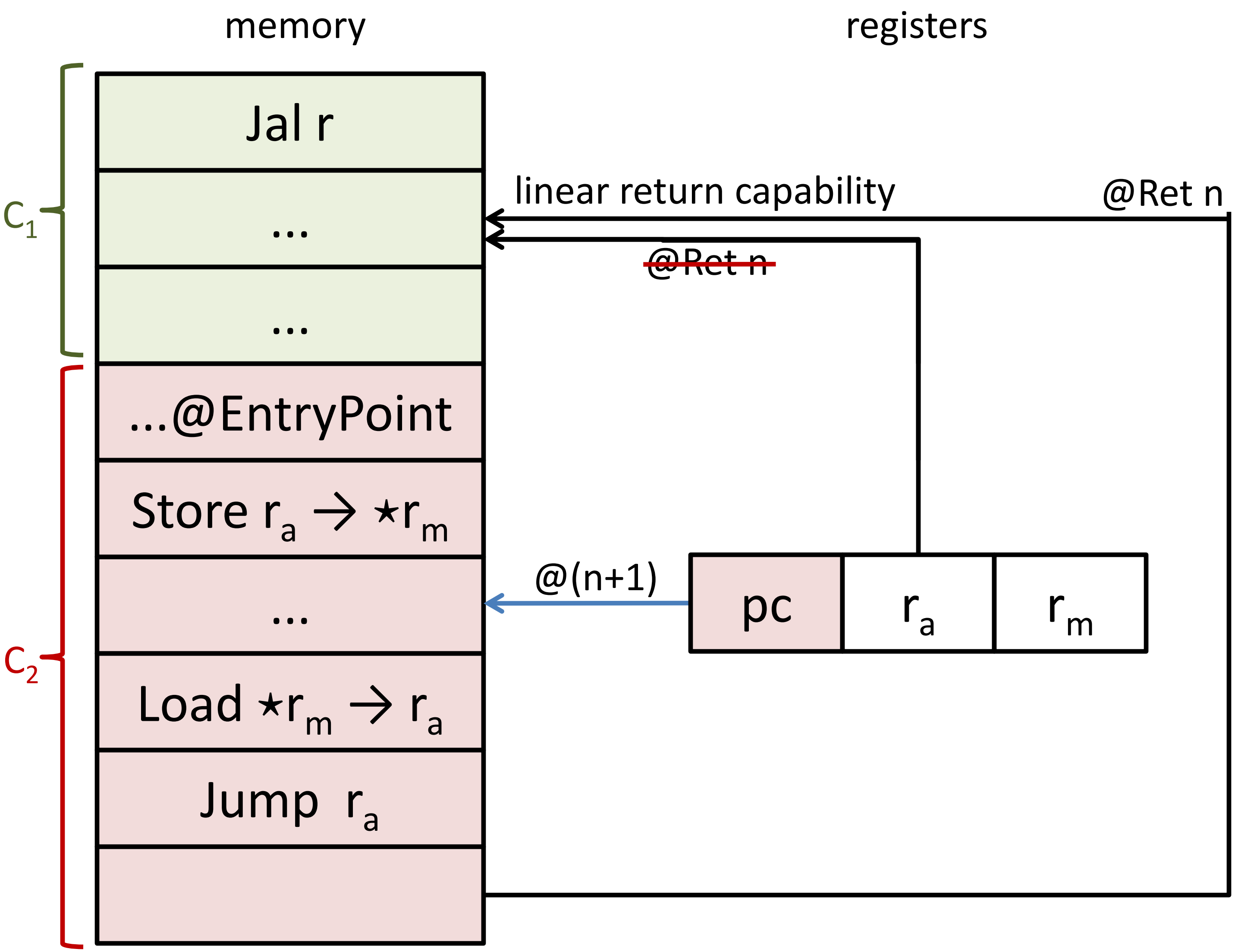


loads and stores to the same component always allowed

Compartmentalization micro-policy

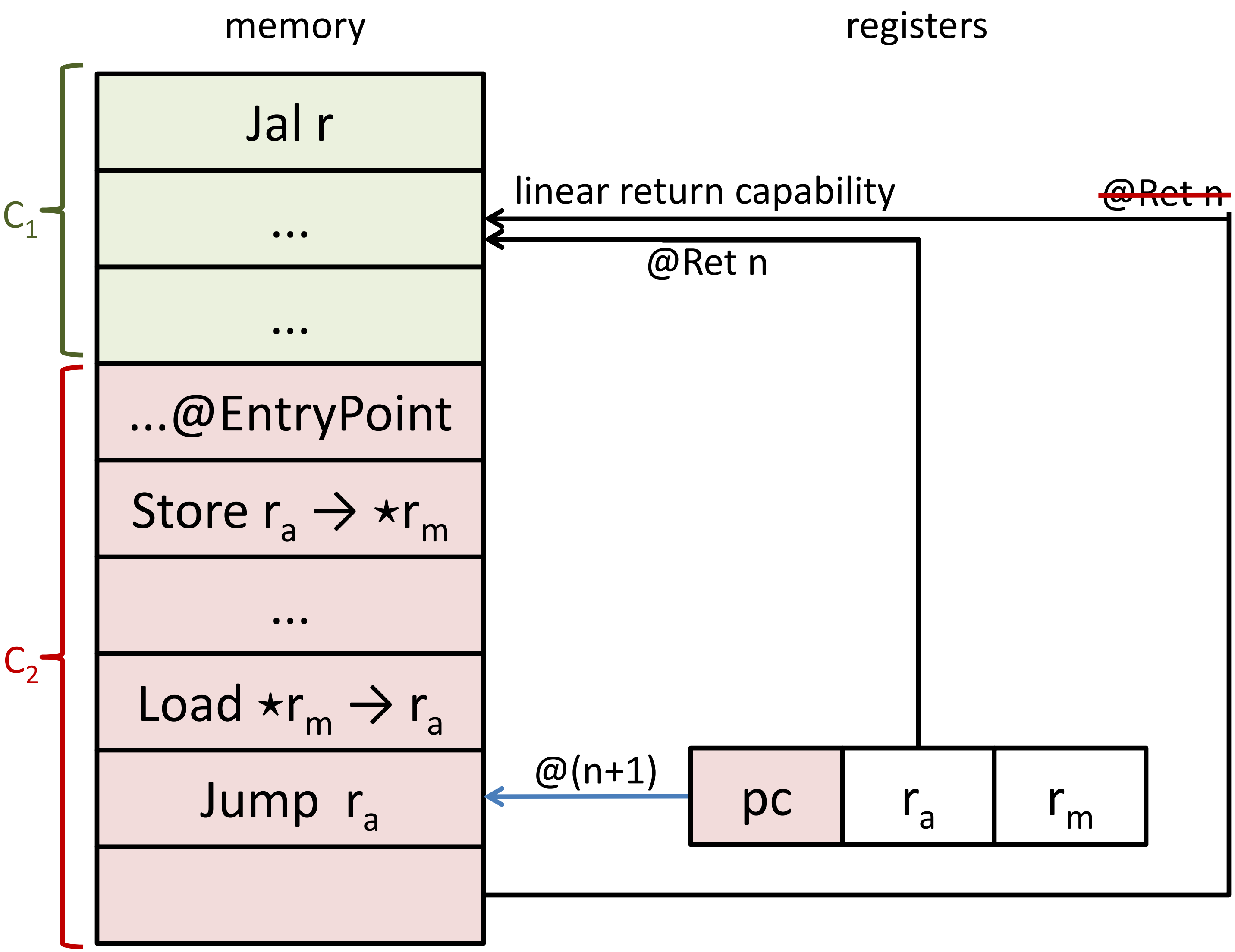


Compartmentalization micro-policy



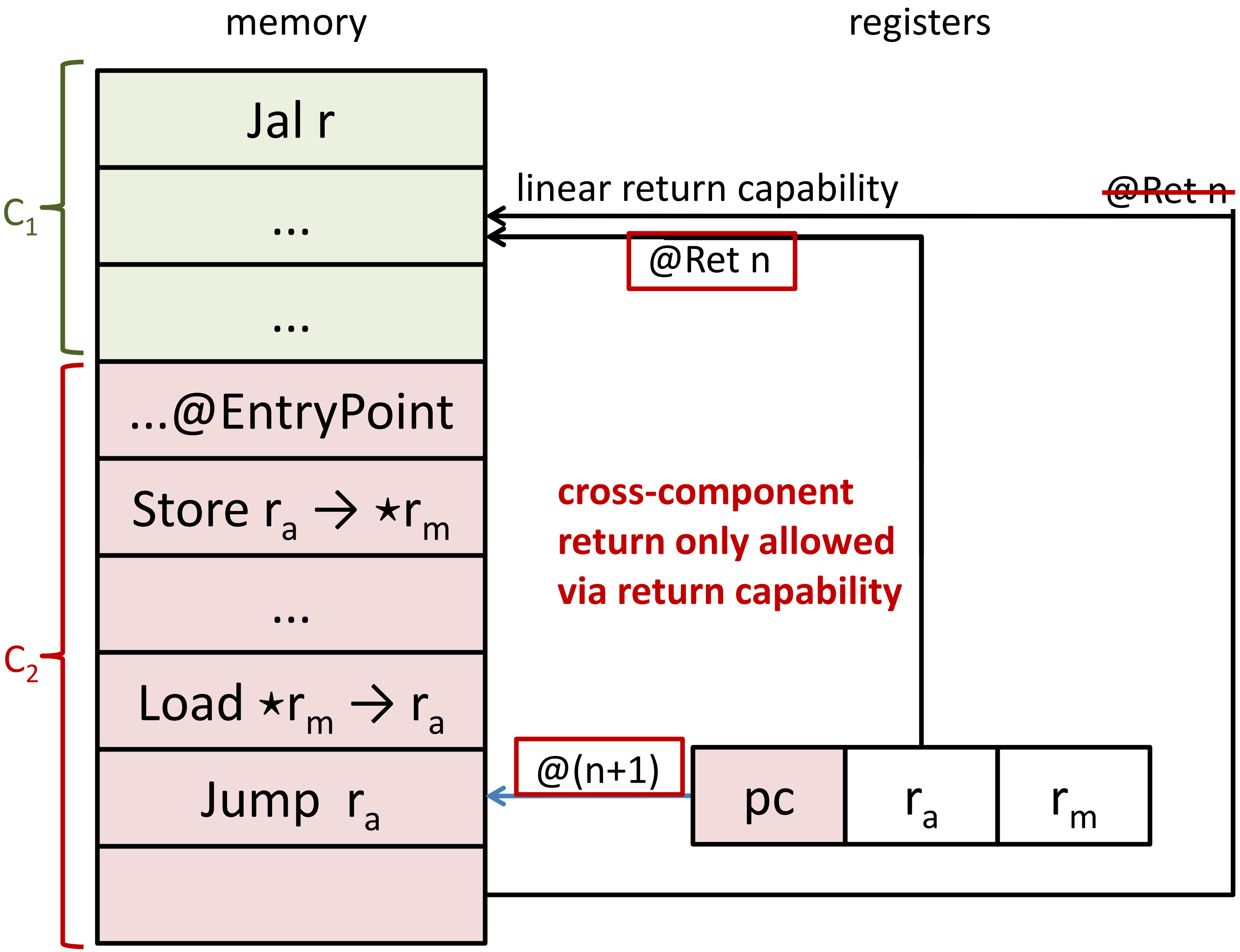
invariant:
at most one
return capability
per call stack level

Compartmentalization micro-policy



invariant:
at most one
return capability
per call stack level

Compartmentalization micro-policy

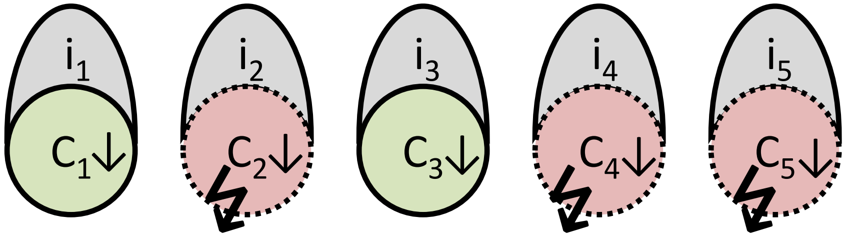


cross-component return only allowed via return capability

invariant:
at most one return capability per call stack level

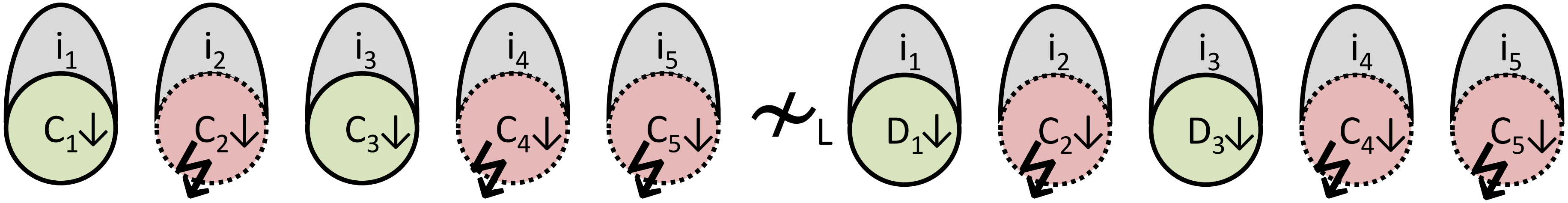
Secure compartmentalizing compilation (SCC)

∀compromise scenarios.



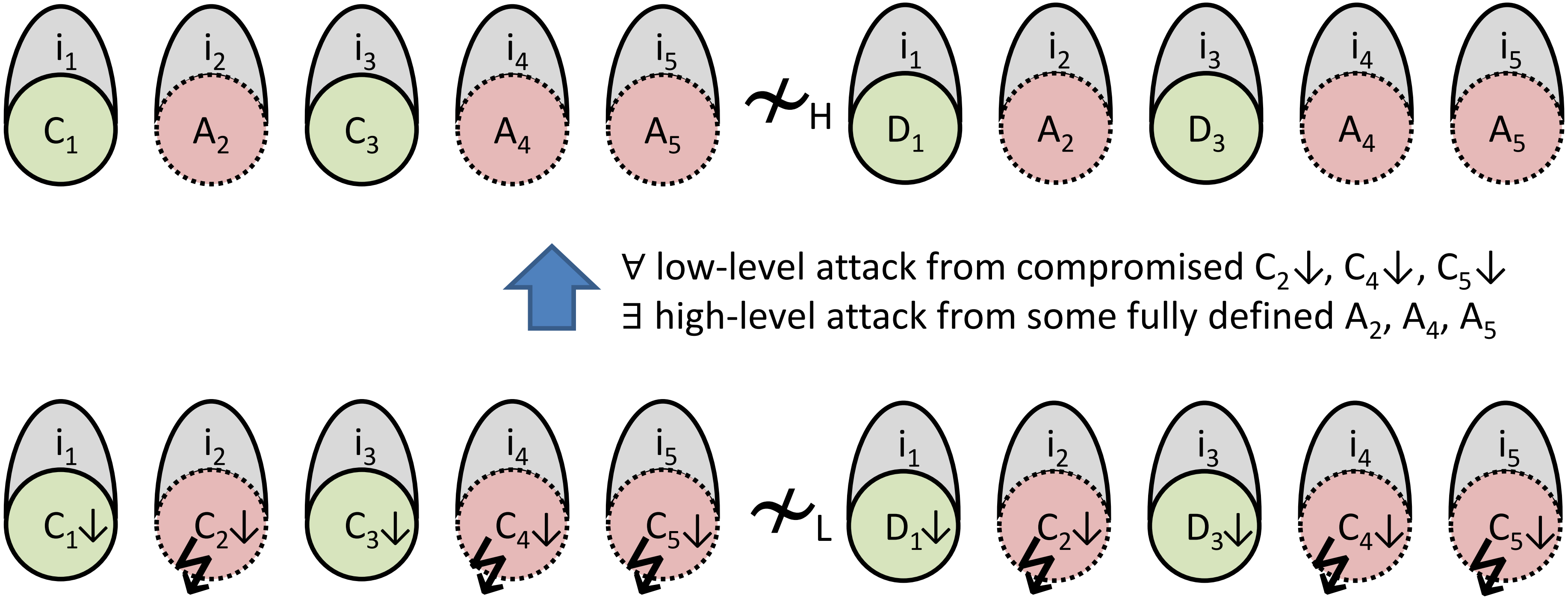
Secure compartmentalizing compilation (SCC)

∀compromise scenarios.



Secure compartmentalizing compilation (SCC)

∀compromise scenarios.

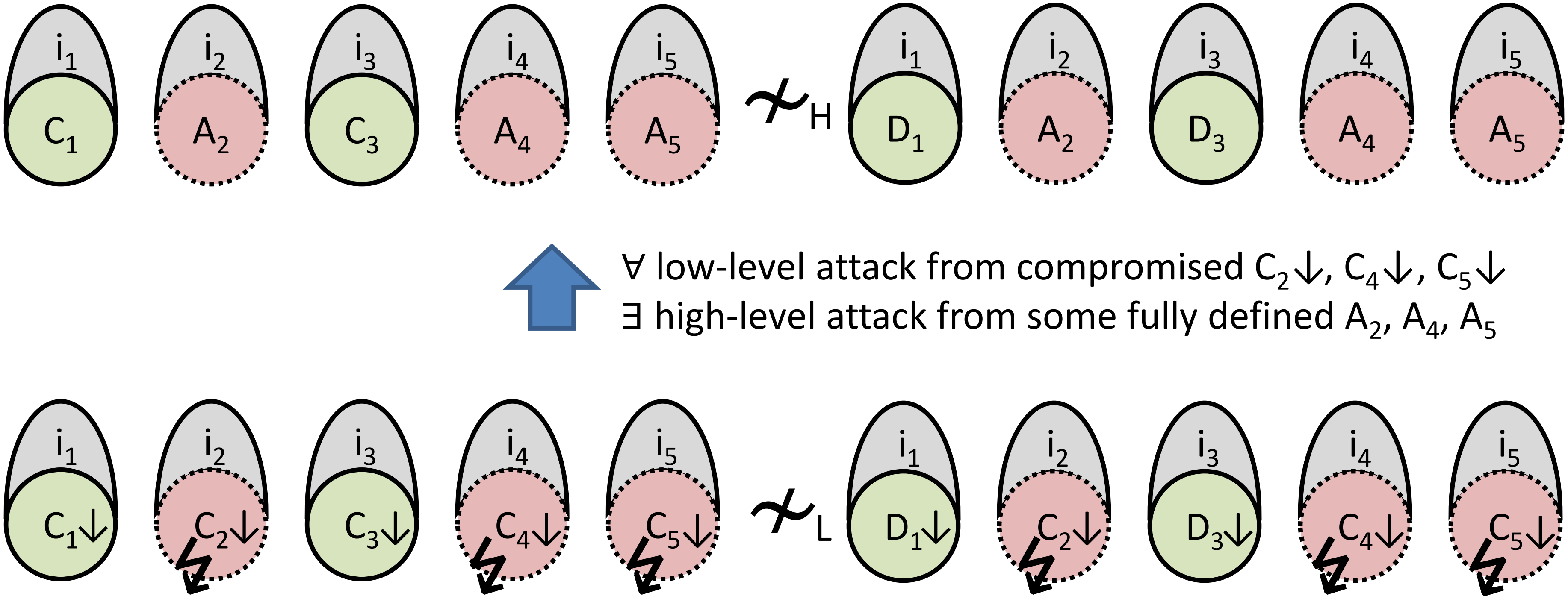

i_1 i_2 i_3 i_4 i_5
C_1 A_2 C_3 A_4 A_5
$\not\sim_H$
i_1 i_2 i_3 i_4 i_5
D_1 A_2 D_3 A_4 A_5
i_1 i_2 i_3 i_4 i_5
$C_1\downarrow$ $C_2\downarrow$ $C_3\downarrow$ $C_4\downarrow$ $C_5\downarrow$
$\not\sim_L$
i_1 i_2 i_3 i_4 i_5
$D_1\downarrow$ $C_2\downarrow$ $D_3\downarrow$ $C_4\downarrow$ $C_5\downarrow$

∀ 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.


i_1 i_2 i_3 i_4 i_5
C_1 A_2 C_3 A_4 A_5
$\nsim_H$
i_1 i_2 i_3 i_4 i_5
D_1 A_2 D_3 A_4 A_5
i_1 i_2 i_3 i_4 i_5
$C_1\downarrow$ $C_2\downarrow$ $C_3\downarrow$ $C_4\downarrow$ $C_5\downarrow$
$\nsim_L$
i_1 i_2 i_3 i_4 i_5
$D_1\downarrow$ $C_2\downarrow$ $D_3\downarrow$ $C_4\downarrow$ $C_5\downarrow$

∀ 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

- **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

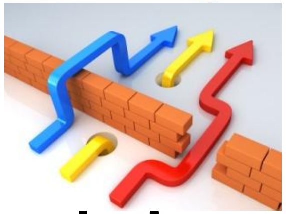
- **ML abstractions we want to enforce with micro-policies**
  - types, value immutability, opaqueness of closures, parametricity (dynamic sealing), GC vs malloc/free, ...
- **F*: enforcing full specifications using micro-policies**
  - some can be turned into **contracts,** checked dynamically
  - fully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

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

SECOMP in a nutshell

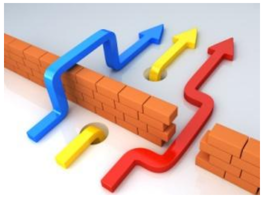
- **We need more secure languages, compilers, hardware**

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 and F*)

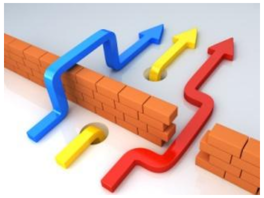
SECOMP in a nutshell

- **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 and F*)
- **Answering challenging fundamental questions**
  - attacker models, proof techniques
  - secure composition, micro-policies for C



SECOMP in a nutshell

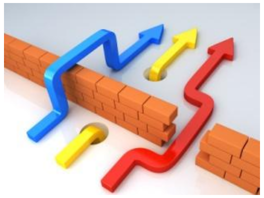
- **We need more secure languages, compilers, hardware**
- **Key enabler: micro-policies** (software-hardware protection)
- **Grand challenge: the first efficient formally secure compilers**
  for **realistic programming languages** (C and F*)
- **Answering challenging fundamental questions**
  - attacker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

SECOMP in a nutshell

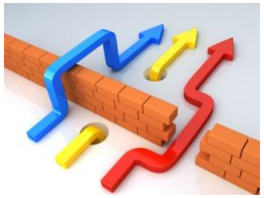
- **We need more secure languages, compilers, hardware**
- **Key enabler: micro-policies** (software-hardware protection)
- **Grand challenge: the first efficient formally secure compilers**
  for **realistic programming languages** (C and F*)
- **Answering challenging fundamental questions**
  - attacker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

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**

- Most of this is **vaporware** at this point but ...
  - building a community, looking for collaborators, and hiring

Collaborators & Community

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

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)

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 (very informal)**
  - 1st at Inria Paris in August 2016
  - 2nd in Paris on 15 January 2017 before POPL at UPMC
  - Work in progress proposal for Dagstuhl seminar in 2018
  - **build larger research community, identify open problems, bring together communities** (hardware, systems, security, languages, verification, ...)

BACKUP SLIDES

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

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

Micro-policies:

remaining fundamental challenges

Micro-policies:
remaining fundamental challenges

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

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

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.]

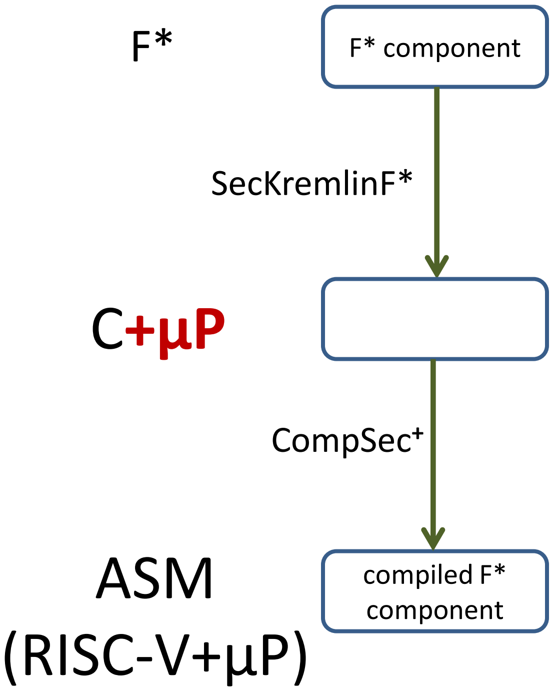
Beyond full abstraction

- 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)

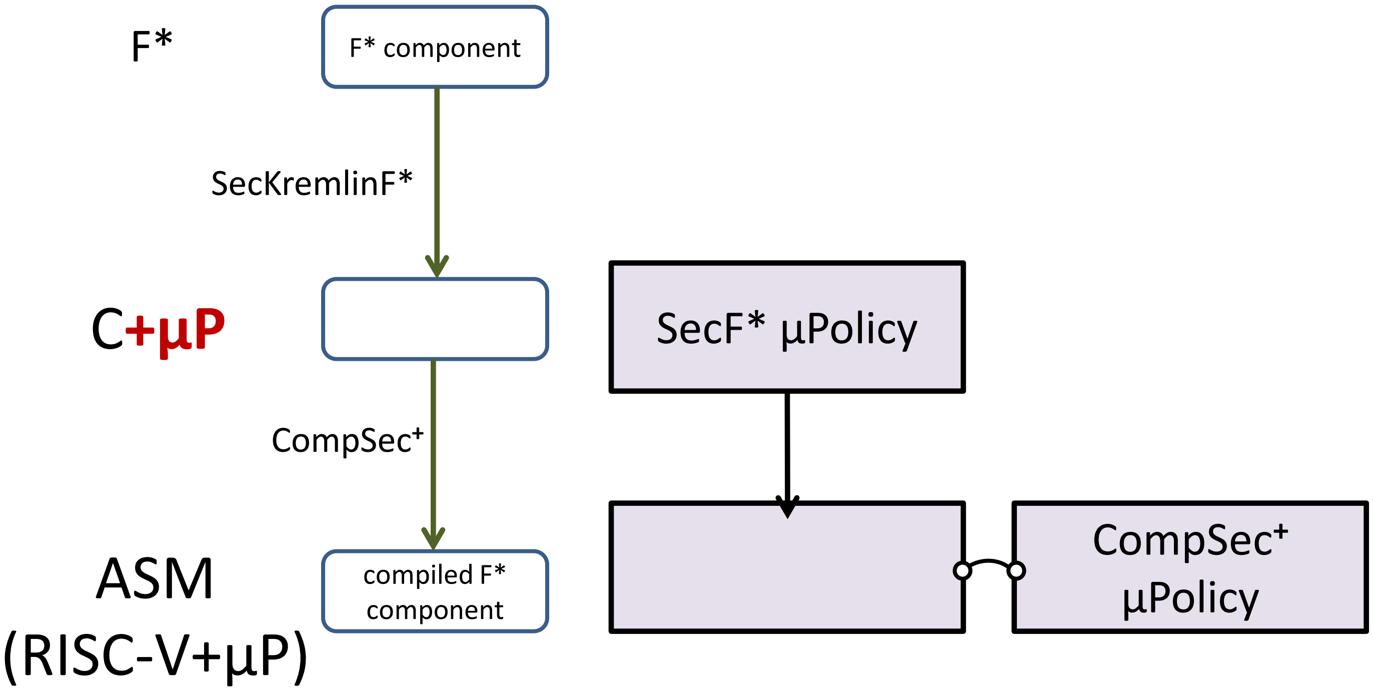
Beyond full abstraction

- 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)

Composing compilers and higher-level micro-policies



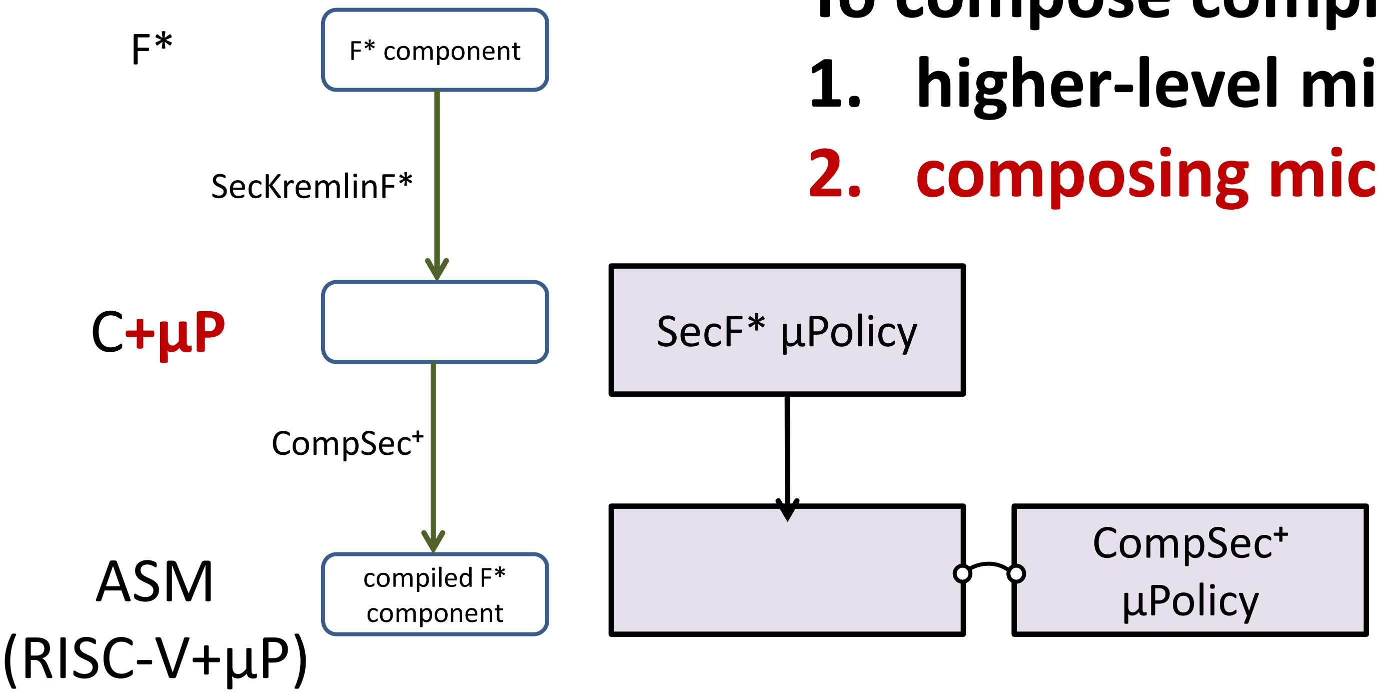
Composing compilers and higher-level micro-policies



Composing compilers and higher-level micro-policies

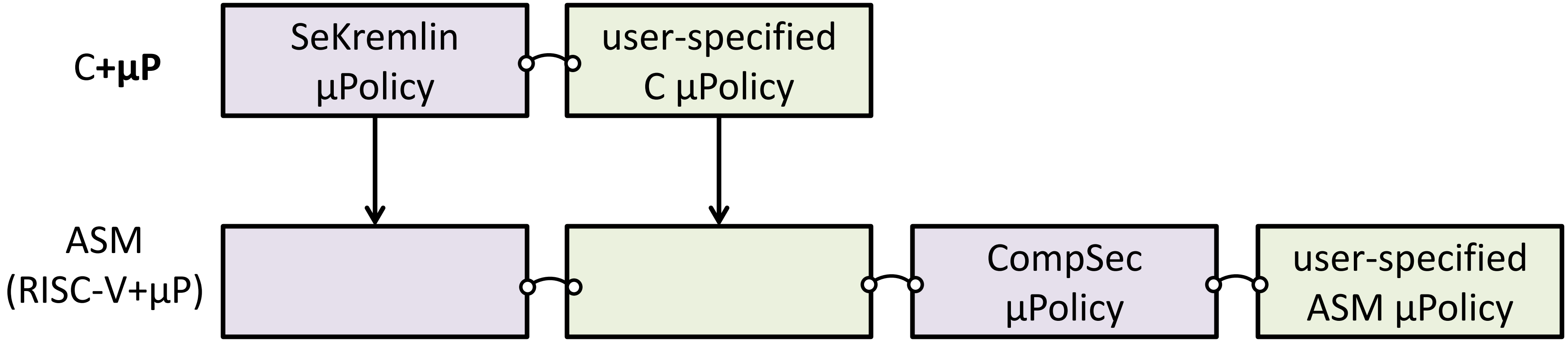
To compose compilers need

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



User-specified higher-level policies

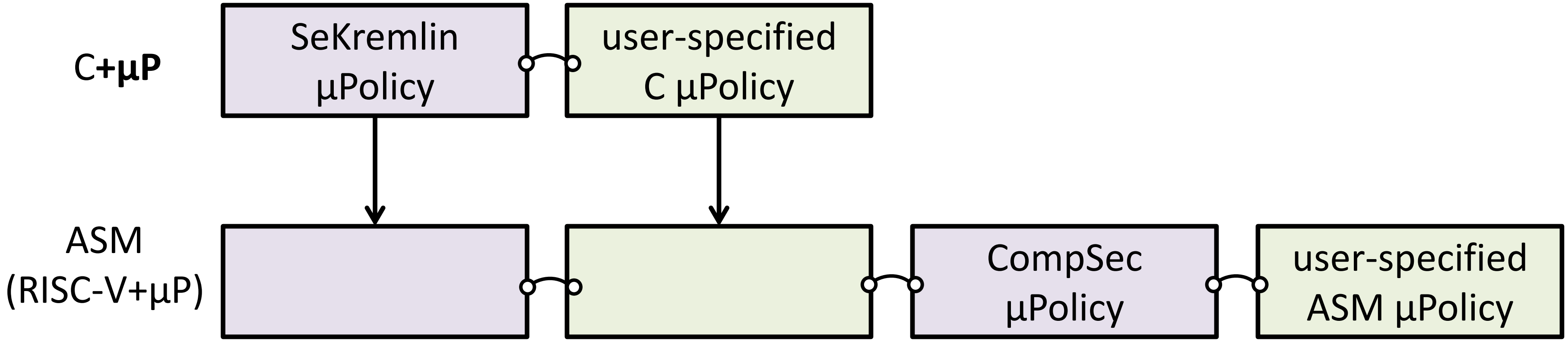
- By composing more micro-policies we can allow **user-specified micro-policies for C**



User-specified higher-level policies

- By composing more micro-policies we can allow **user-specified micro-policies for C**
- Good news:

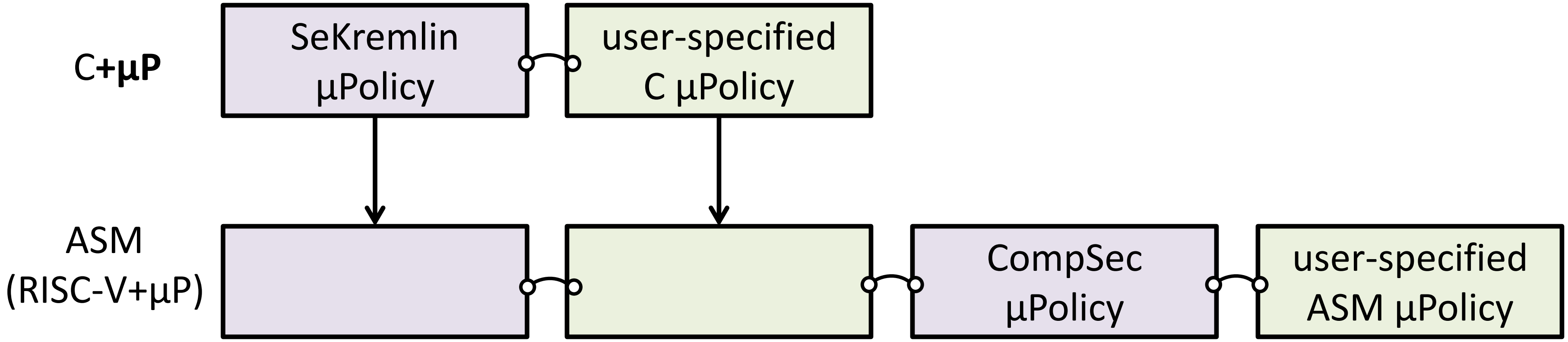
  micro-policy composition is easy since tags can be tuples



User-specified higher-level policies

- By composing more micro-policies we can allow **user-specified micro-policies for C**
- Good news:

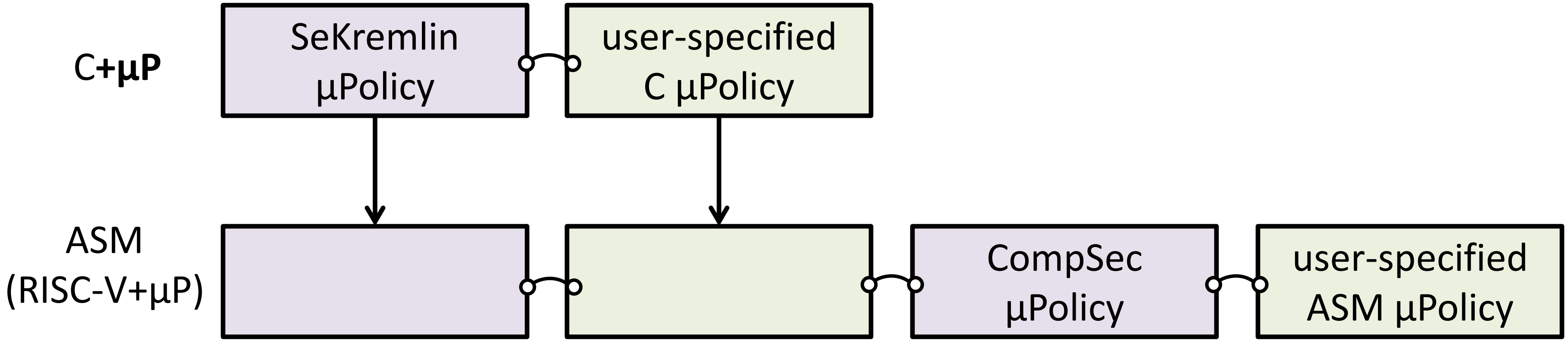
  micro-policy composition is easy since tags can be tuples
- But how do we ensure programmers won't break security?



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