# When Good Components Go Bad
# Formally Secure Compilation Despite Dynamic Compromise

**Cătălin Hrițcu**

Inria Paris

https://secure-compilation.github.io

# Collaborators


**Carmine Abate**

**Arthur Azevedo de Amorim**

**Rob Blanco**

**Ana Nora Evans**


**Guglielmo Fachini**

**Cătălin Hrițcu**

**Yannis Juglaret**

**Théo Laurent**

**Benjamin Pierce**

**Marco Stronati**

**Andrew Tolmach**

Inria Paris CMU U. Virginia U. Trento Paris 7 ENS Paris Portland State UPenn

# Devastating low-level vulnerabilities

# Devastating low-level vulnerabilities

- **Inherently insecure C-like languages**
  - **type and memory unsafe**:
    e.g. any buffer overflow is catastrophic
  - ~100 different **undefined behavior** reasons in the usual C compiler

# Devastating low-level vulnerabilities

- **Inherently insecure C-like languages**
  - **type and memory unsafe**:
    e.g. any buffer overflow is catastrophic
  - ~100 different **undefined behavior** reasons in the usual C compiler
  - **root cause**, but challenging to fix:
    - efficiency
    - precision
    - scalability
    - backwards compatibility
    - deployment

# Practical mitigation: compartmentalization

# Practical mitigation: compartmentalization

- **Main idea:**
  - break up security-critical C applications into **mutually distrustful components** with clearly specified privileges & interacting via strictly enforced interfaces

# Practical mitigation: compartmentalization

- **Main idea:**
  - break up security-critical C applications into **mutually distrustful components** with clearly specified privileges & interacting via strictly enforced interfaces
- **Strong security guarantees & interesting attacker model**
  - "a vulnerability in one component does not immediately destroy the security of the whole application"

# Practical mitigation: compartmentalization

- **Main idea:**
  - break up security-critical C applications into **mutually distrustful components** with clearly specified privileges & interacting via strictly enforced interfaces
- **Strong security guarantees & interesting attacker model**
  - "a vulnerability in one component does not immediately destroy the security of the whole application"
  - "each component is protected from all the others"

# Practical mitigation: compartmentalization

- **Main idea:**
  - break up security-critical C applications into **mutually distrustful components** with clearly specified privileges & interacting via strictly enforced interfaces
- **Strong security guarantees & interesting attacker model**
  - "a vulnerability in one component does not immediately destroy the security of the whole application"
  - "each component is protected from all the others"
  - "each components receives guarantees as long as it has not encountered undefined beehavior"

# Practical mitigation: compartmentalization

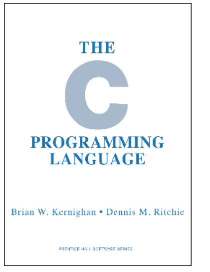
- **Main idea:**
  - break up security-critical C applications into **mutually distrustful components** with clearly specified privileges & interacting via strictly enforced interfaces
- **Strong security guarantees & interesting attacker model**
  - "a vulnerability in one component does not immediately destroy the security of the whole application"
  - "each component is protected from all the others"
  - "each components receives guarantees as long as it has not encountered undefined beehavior"

**Goal 1: Formalize this**

# Goal 2: Build secure compilation chains

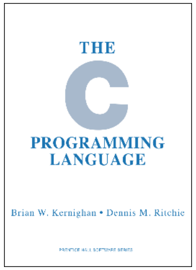
# Goal 2: Build secure compilation chains

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



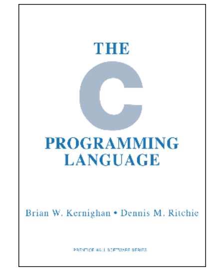

# Goal 2: Build secure compilation chains

- **Add components to C**
  - interacting only via **strictly enforced interfaces**
- **Enforce "component C" abstractions:**
  - component separation, call-return discipline, ...



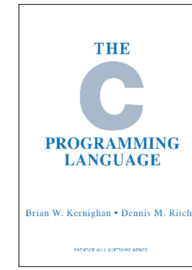

# Goal 2: Build secure compilation chains

- **Add components to C**
  - interacting only via **strictly enforced interfaces**
- **Enforce "component C" abstractions:**
  - component separation, call-return discipline, ...
- **Secure compilation chain:**
  - compiler, linker, loader, runtime, system, hardware




# Goal 2: Build secure compilation chains

- **Add components to C**
  - interacting only via **strictly enforced interfaces**
- **Enforce "component C" abstractions:**
  - component separation, call-return discipline, ...
- **Secure compilation chain:**
  - compiler, linker, loader, runtime, system, hardware
- **Use efficient enforcement mechanisms:**
  - OS processes (all web browsers)
  - software fault isolation (SFI)
  - hardware enclaves (SGX)
  - WebAssembly (web browsers)
  - capability machines
  - tagged architectures




# Goal 1: Formalizing the security of compartmentalizing compilation

# Restricting undefined behavior

- **Mutually-distrustful components**
  - restrict **spatial** scope of undefined behavior

# Restricting undefined behavior

- **Mutually-distrustful components**
  - restrict **spatial** scope of undefined behavior
- **Dynamic compromise**
  - restrict **temporal** scope of undefined behavior

# Restricting undefined behavior

- **Mutually-distrustful components**
  - restrict **spatial** scope of undefined behavior
- **Dynamic compromise**
  - restrict **temporal** scope of undefined behavior
  - undefined behavior = **observable trace event**
  - **effects of undefined behavior**
    shouldn't percolate before earlier observable events
    - careful with code motion, backwards static analysis, ...

# Restricting undefined behavior

- **Mutually-distrustful components**
  - restrict **spatial** scope of undefined behavior
- **Dynamic compromise**
  - restrict **temporal** scope of undefined behavior
  - undefined behavior = **observable trace event**
  - **effects of undefined behavior**
    shouldn't percolate before earlier observable events
    - careful with code motion, backwards static analysis, ...
  - CompCert **already offers** this saner temporal model

# Restricting undefined behavior

- **Mutually-distrustful components**
  - restrict **spatial** scope of undefined behavior
- **Dynamic compromise**
  - restrict **temporal** scope of undefined behavior
  - undefined behavior = **observable trace event**
  - **effects of undefined behavior**
    shouldn't percolate before earlier observable events
    - careful with code motion, backwards static analysis, ...
  - CompCert **already offers** this saner temporal model
  - GCC and LLVM **currently violate** this model

# Dynamic compromise

- each component gets guarantees as long as it has not encountered undefined behavior

# Dynamic compromise

- each component gets guarantees as long as it has not encountered undefined behavior

- a component only loses guarantees after an attacker discovers and exploits a vulnerability

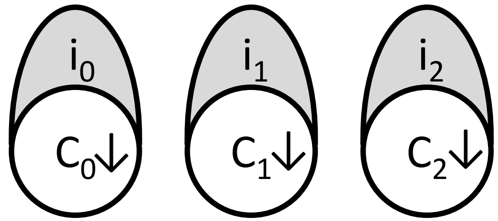
# Dynamic compromise

- each component gets guarantees as long as it has not encountered undefined behavior

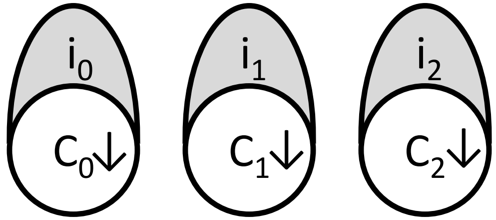
- a component only loses guarantees after an attacker discovers and exploits a vulnerability

- the mere existence of vulnerabilities doesn't immediately make a component compromised

If



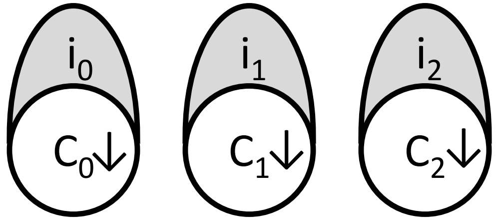

$\rightsquigarrow t$ then

If  ⇝ *t* then



∃ a **dynamic compromise scenario** explaining *t* in source language

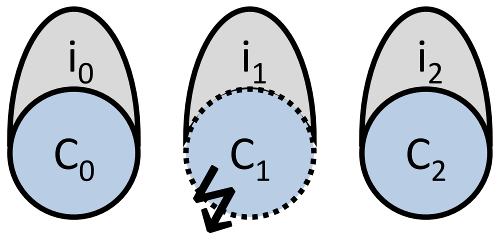
If



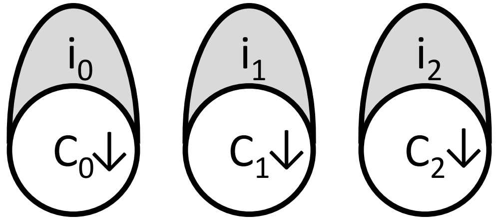

$\rightsquigarrow$ *t* then

∃ a **dynamic compromise scenario** explaining *t* in source language for instance leading to the following compromise sequence:

(0)

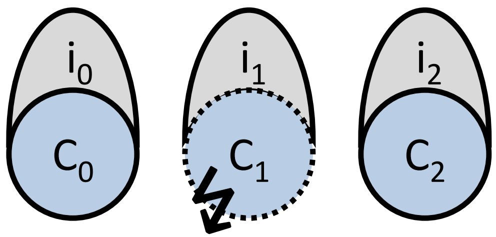



$\rightsquigarrow^*$ $m_1$;Undef($C_1$)

If  $\rightsquigarrow t$ then

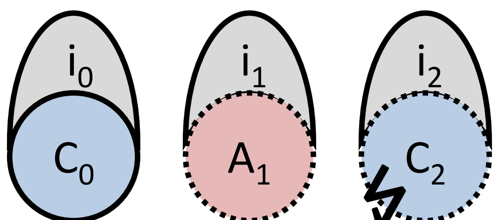


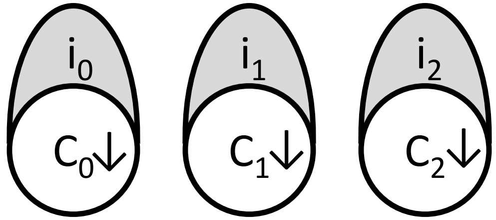
∃ a **dynamic compromise scenario** explaining *t* in source language for instance leading to the following compromise sequence:

(0)  $\rightsquigarrow^*$ $m_1$;Undef($C_1$)



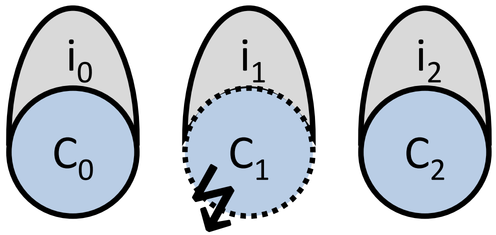
IΛ

(1) $\exists A_1$.  $\rightsquigarrow^*$ $m_2$;Undef($C_2$)

If 

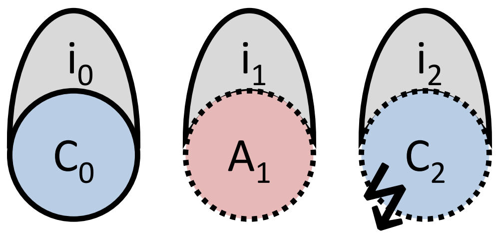
⇝ $t$ then

∃ a **dynamic compromise scenario** explaining $t$ in source language for instance leading to the following compromise sequence:

(0) 

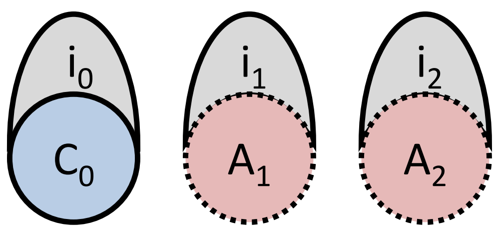
⇝* $m_1$;Undef($C_1$)

$\leq$

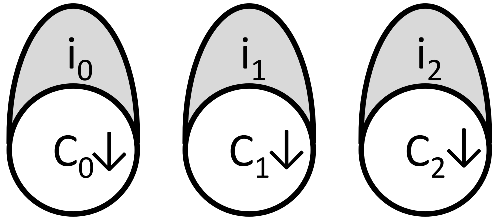
(1) $\exists A_1$. 

⇝* $m_2$;Undef($C_2$)

$\leq$

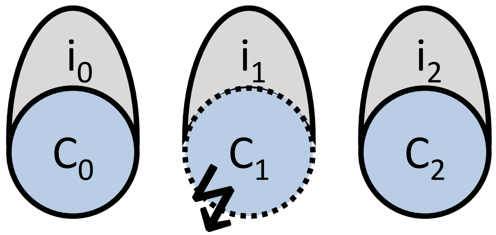
(2) $\exists A_2$. 

⇝ $t$

If 

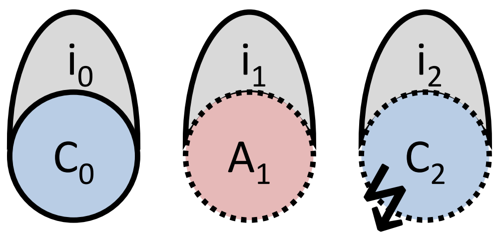
$\rightsquigarrow t$ then

∃ a **dynamic compromise scenario** explaining *t* in source language for instance leading to the following compromise sequence:

(0) 

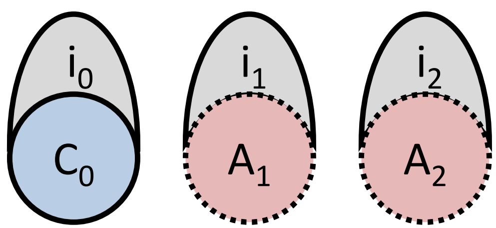
$\rightsquigarrow^* m_1;\mathrm{Undef}(C_1)$

$\leq$

(1) $\exists A_1.$ 

$\rightsquigarrow^* m_2;\mathrm{Undef}(C_2)$

$\leq$

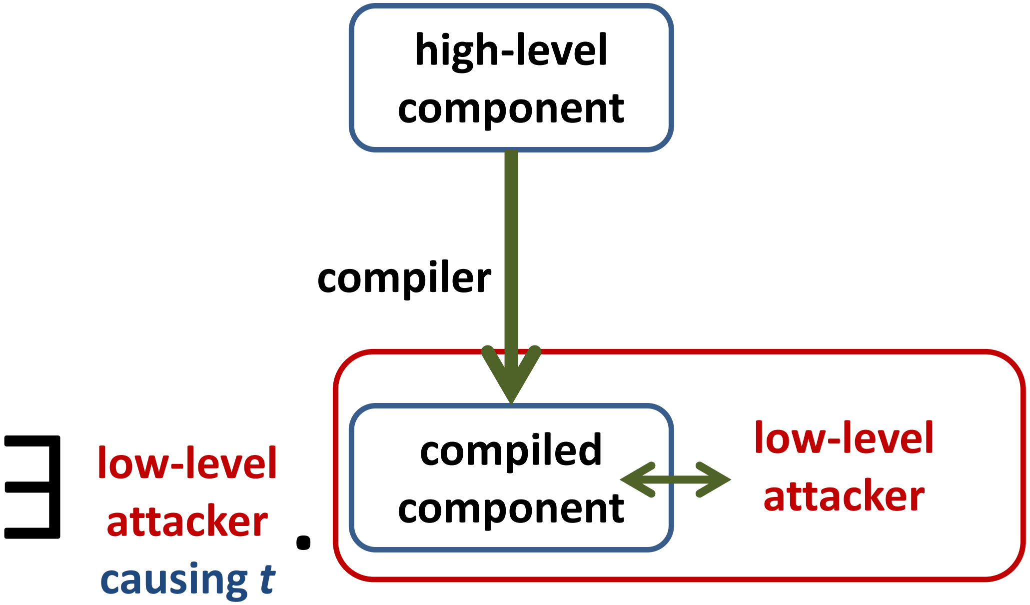
(2) $\exists A_2.$ 

$\rightsquigarrow t$

**Trace is very helpful**
- detect undefined behavior
- rewind execution

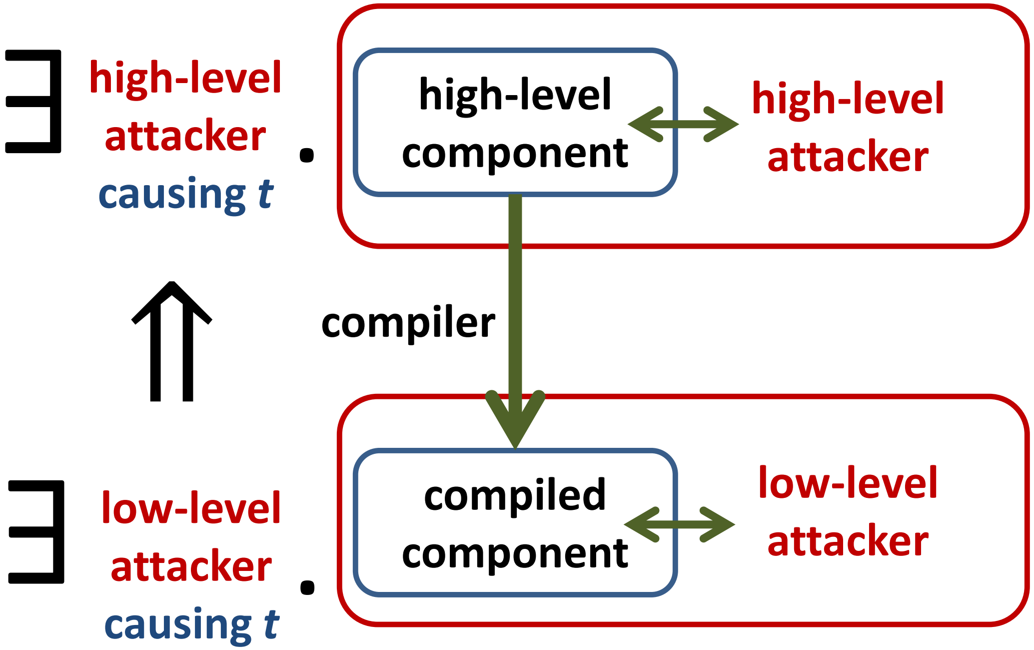
# We build this on Robust Compilation

∀(bad attack) trace *t*



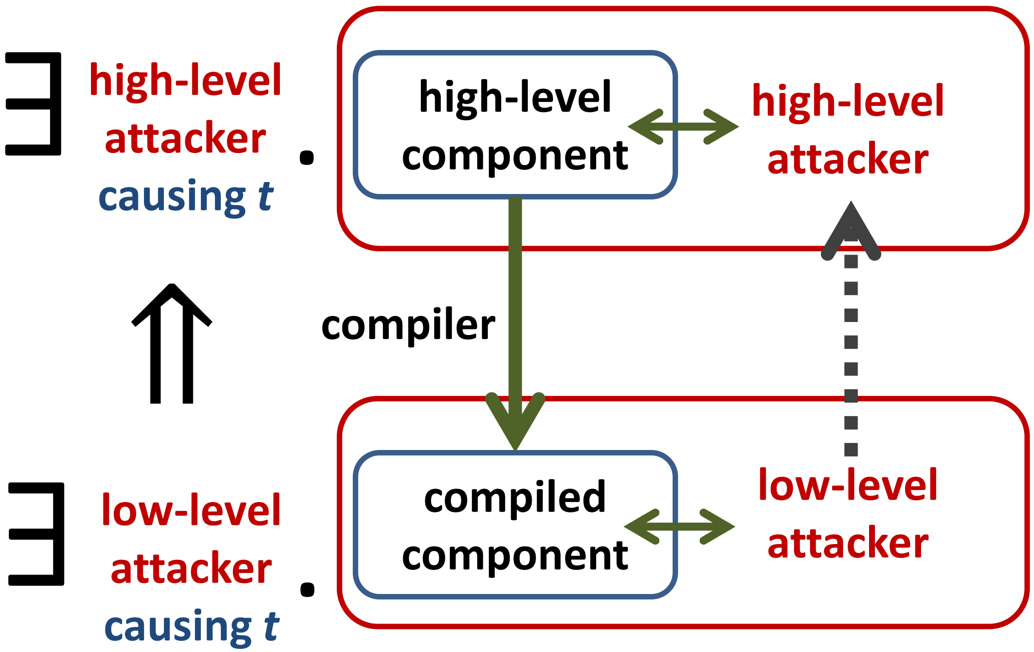
# We build this on Robust Compilation

∀(bad attack) trace *t*



# We build this on Robust Compilation

∀(bad attack) trace *t*



# We build this on Robust Compilation

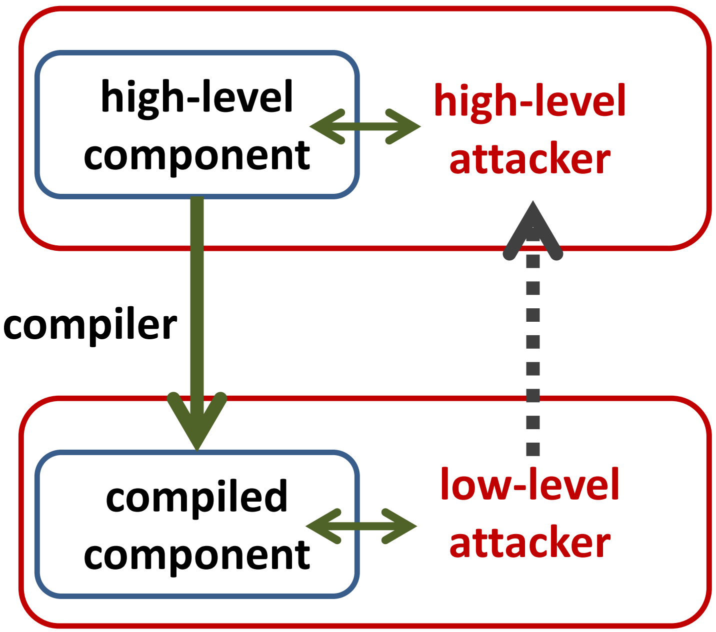
## ∀(bad attack) trace *t*

**robust trace property preservation**
(robust = in adversarial context)

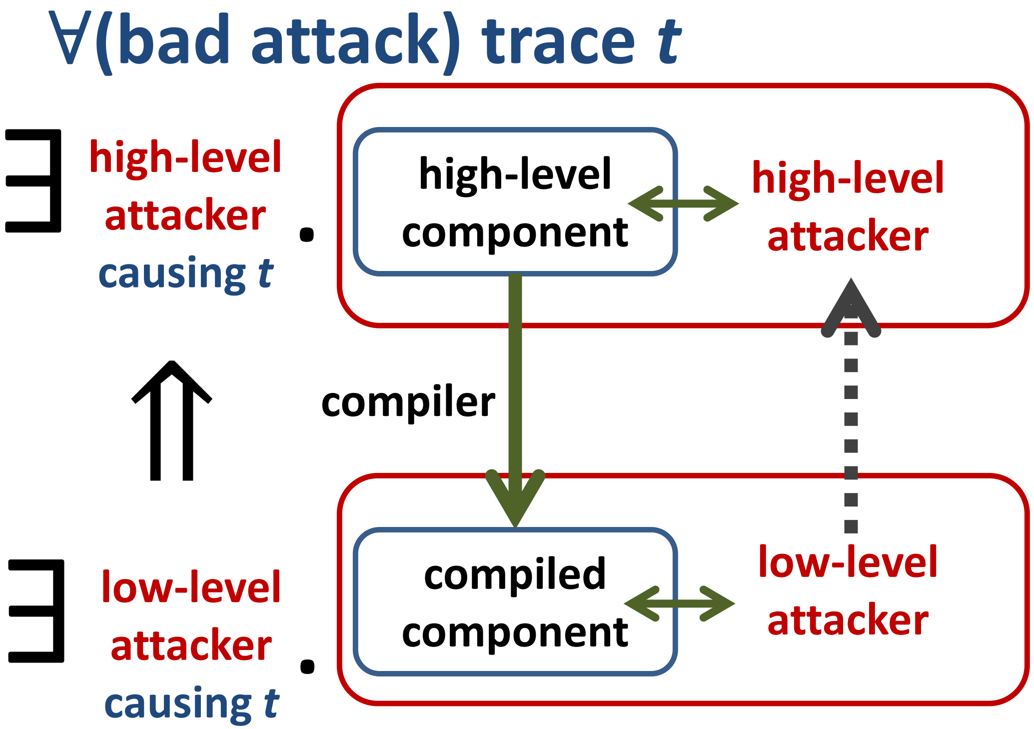
∃ high-level attacker causing *t* .

⇑

∃ low-level attacker causing *t* .



# We build this on Robust Compilation



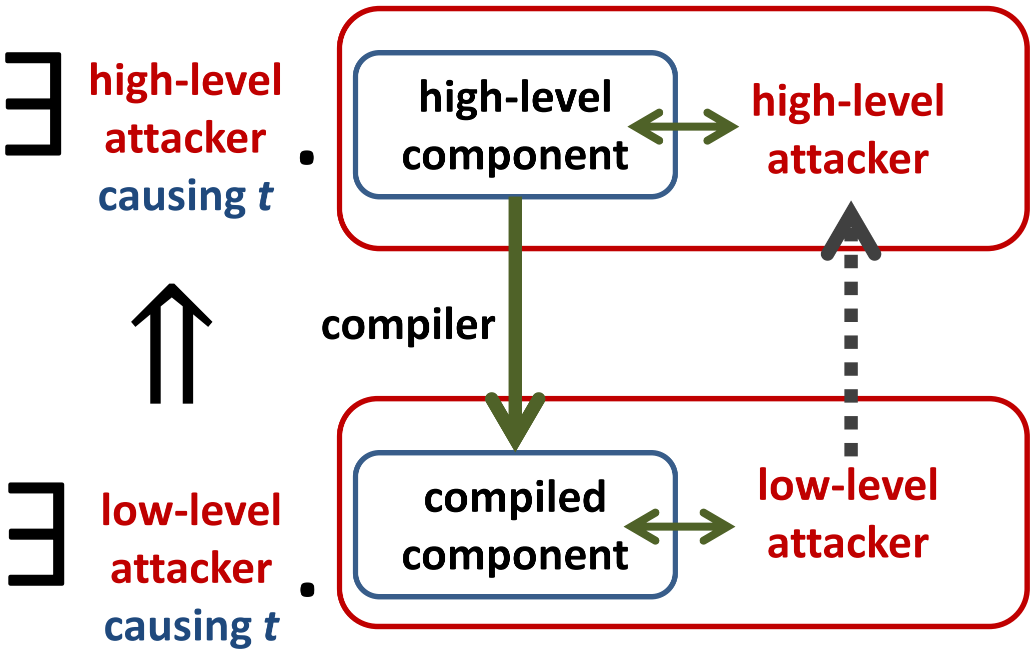

**robust trace property preservation**
(robust = in adversarial context)

**intuition:**

- **stronger** than compiler correctness (i.e. trace property preservation)

# We build this on Robust Compilation

**∀(bad attack) trace *t***



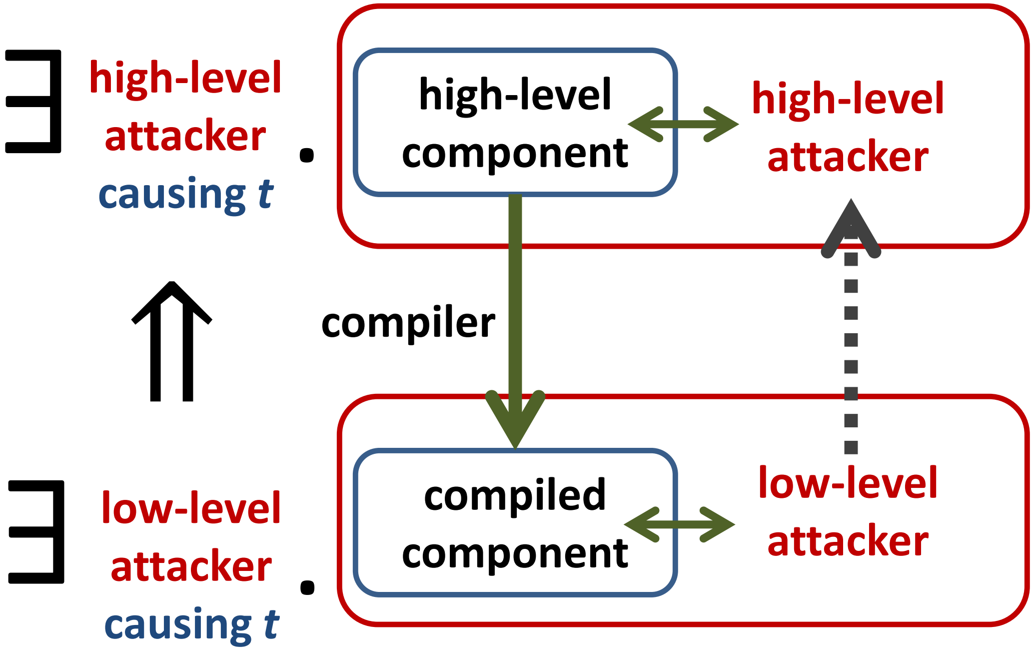

**robust trace property preservation**
(robust = in adversarial context)

**intuition:**

- **stronger** than compiler correctness (i.e. trace property preservation)
- **confidentiality not preserved** (i.e. no hyperproperties)

# We build this on Robust Compilation

∀(bad attack) trace *t*




**robust trace property preservation**
(robust = in adversarial context)

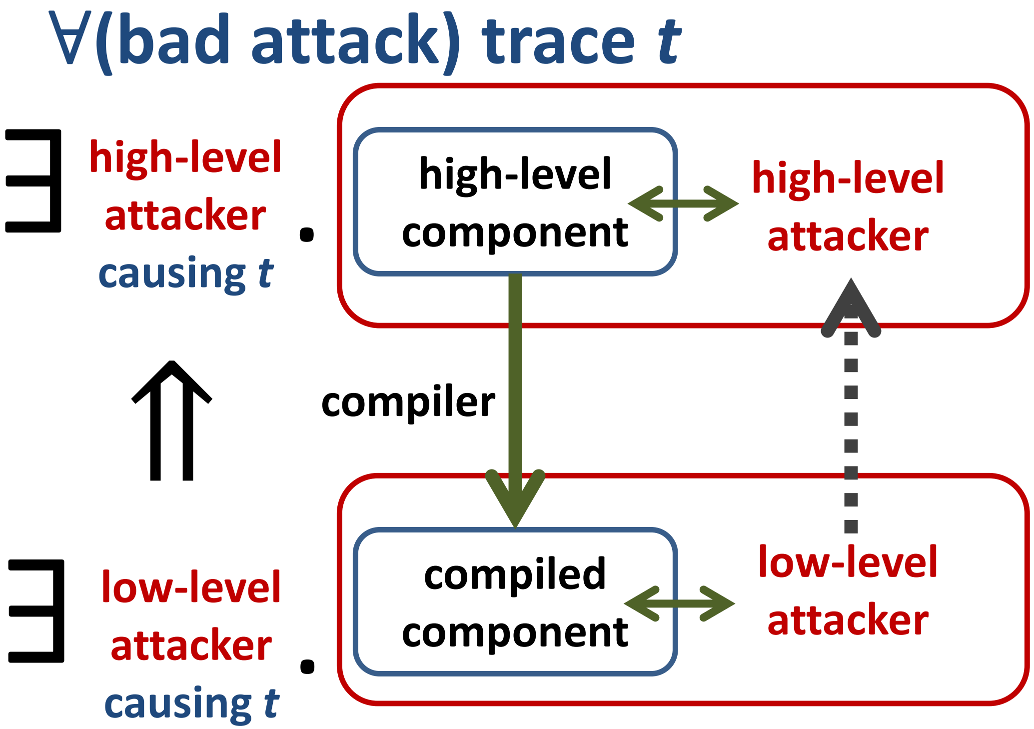
**intuition:**

– **stronger** than compiler correctness (i.e. trace property preservation)

– **confidentiality not preserved** (i.e. no hyperproperties)

– **less extensional** than fully abstract compilation

# We build this on Robust Compilation




**robust trace property preservation**
(robust = in adversarial context)

**intuition:**

- **stronger** than compiler correctness (i.e. trace property preservation)
- **confidentiality not preserved** (i.e. no hyperproperties)
- **less extensional** than fully abstract compilation

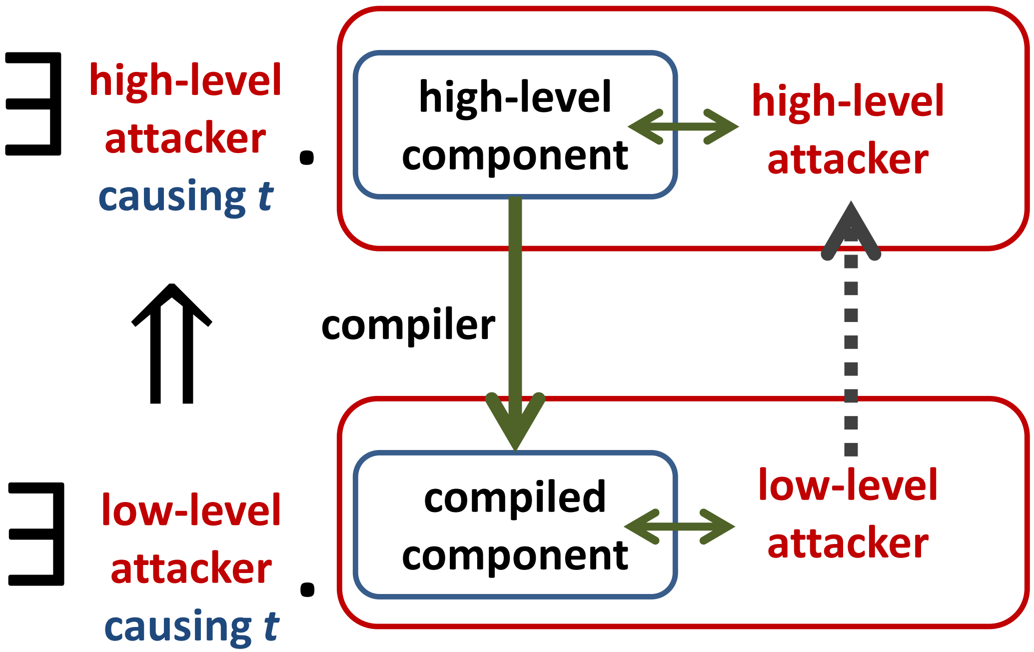
**Advantages**: **easier to realistically achieve and prove at scale**

**useful**: preservation of **invariants** and other **integrity properties**

generalizes to preserving [relational] hyperproperties!

# We build this on Robust Compilation

∀(bad attack) trace *t*




**robust trace property preservation**
(robust = in adversarial context)

**intuition:**

- **stronger** than compiler correctness (i.e. trace property preservation)
- **confidentiality not preserved** (i.e. no hyperproperties)
- **less extensional** than fully abstract compilation

**Advantages**: **easier to realistically achieve and prove at scale**

**useful**: preservation of **invariants** and other **integrity properties**

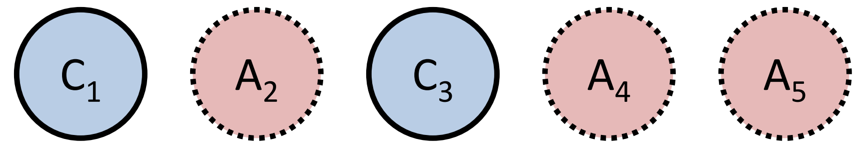
generalizes to preserving [relational] hyperproperties!

**extends to unsafe languages, supporting dynamic compromise**

# Now we know what these words mean!

(at least in the setting of compartmentalization for unsafe low-level languages)

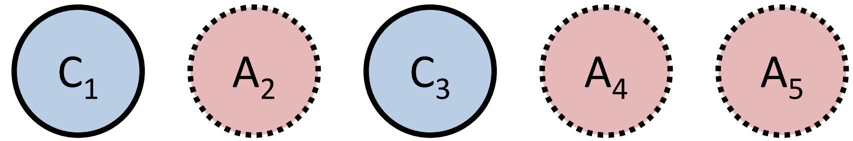
Mutual distrust



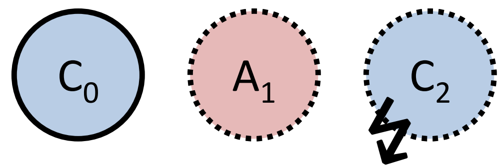
# Now we know what these words mean!

(at least in the setting of compartmentalization for unsafe low-level languages)

Mutual distrust




Dynamic compromise

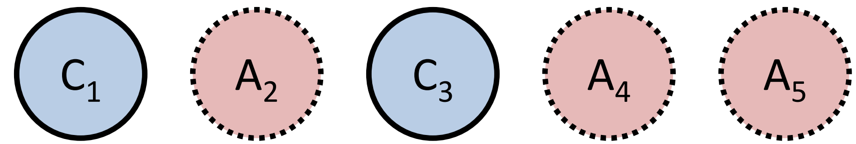



$\Downarrow m_2$; Undef($C_2$)

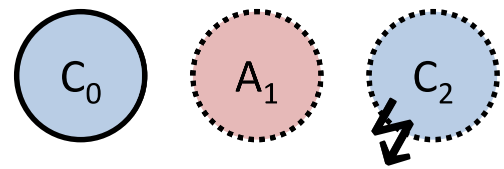
# Now we know what these words mean!

(at least in the setting of compartmentalization for unsafe low-level languages)

Mutual distrust

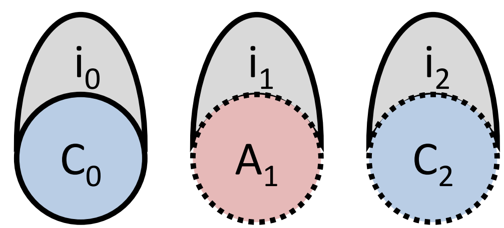



Dynamic compromise

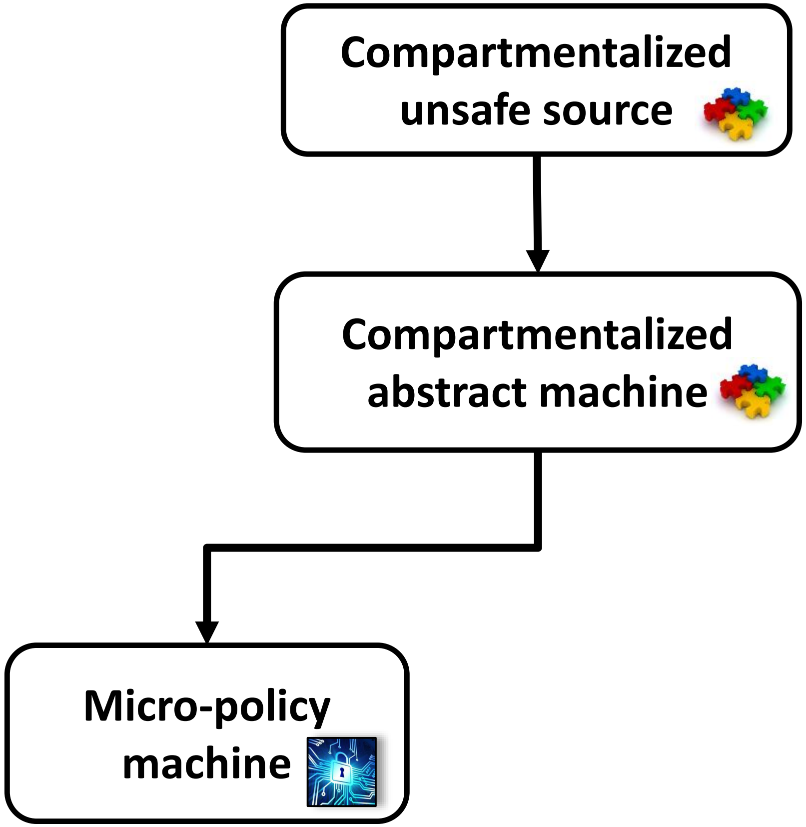


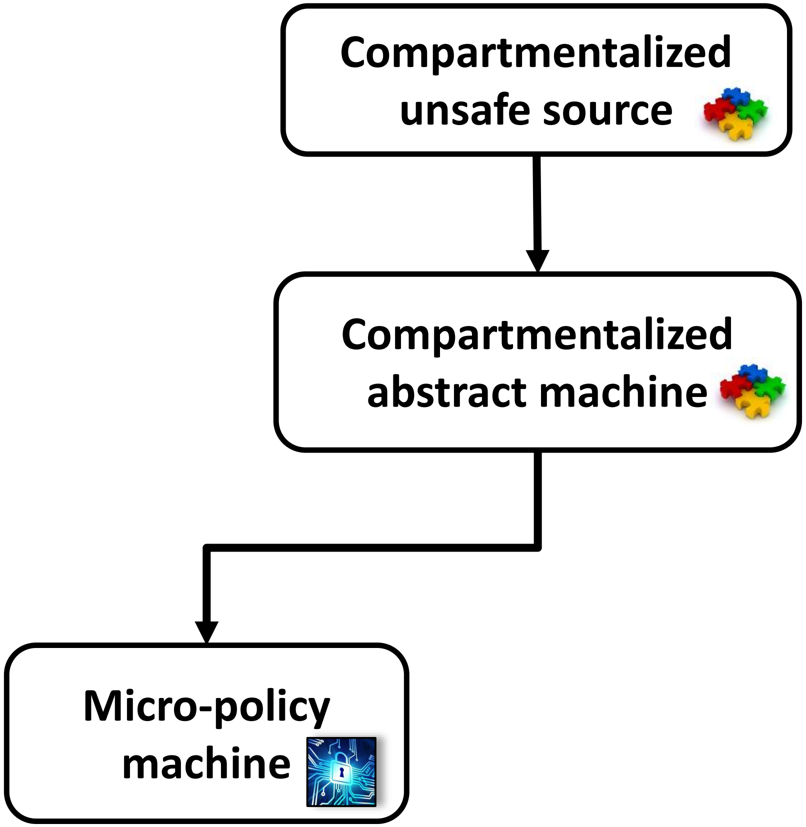

$\Downarrow m_2; \text{Undef}(C_2)$

Static privilege

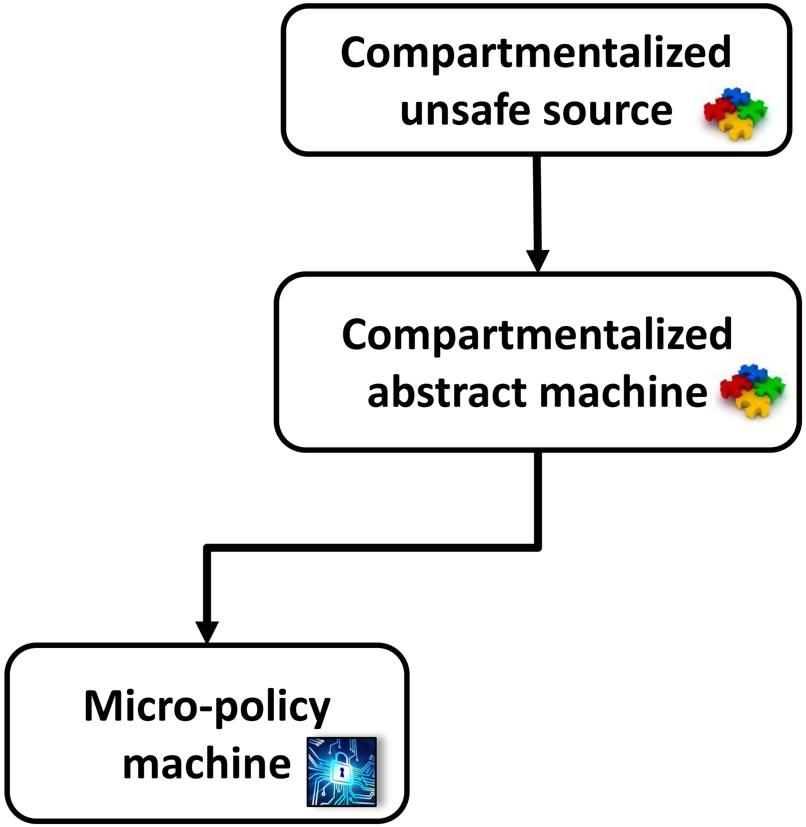


# Goal 2: Towards building secure compilation chains


Compartmentalized unsafe source
Compartmentalized abstract machine
Micro-policy machine

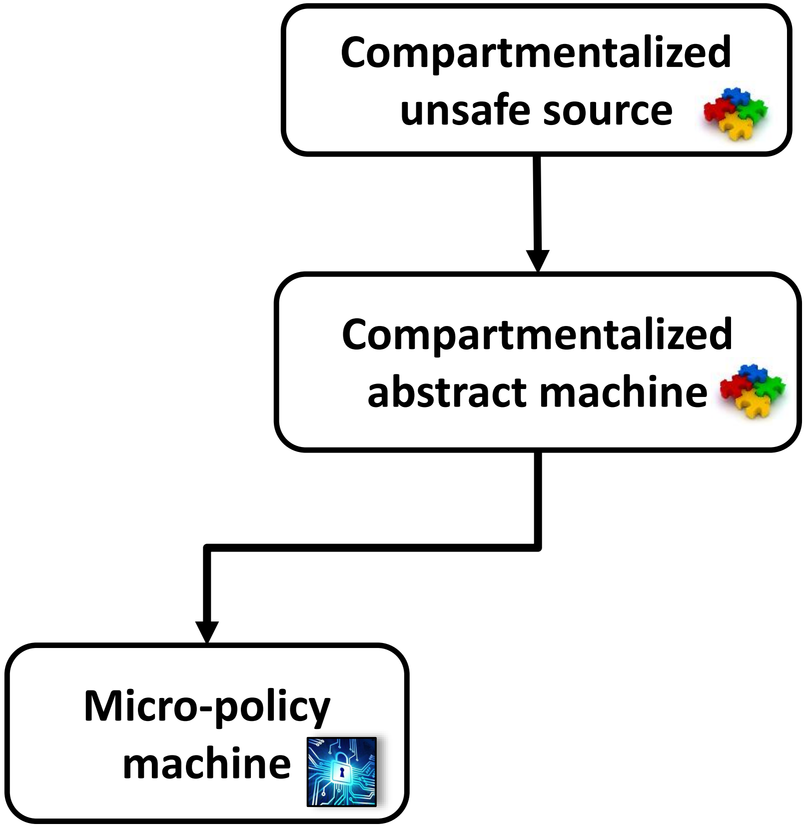



Buffers, procedures, components interacting via **strictly enforced interfaces**




Buffers, procedures, components interacting via **strictly enforced interfaces**

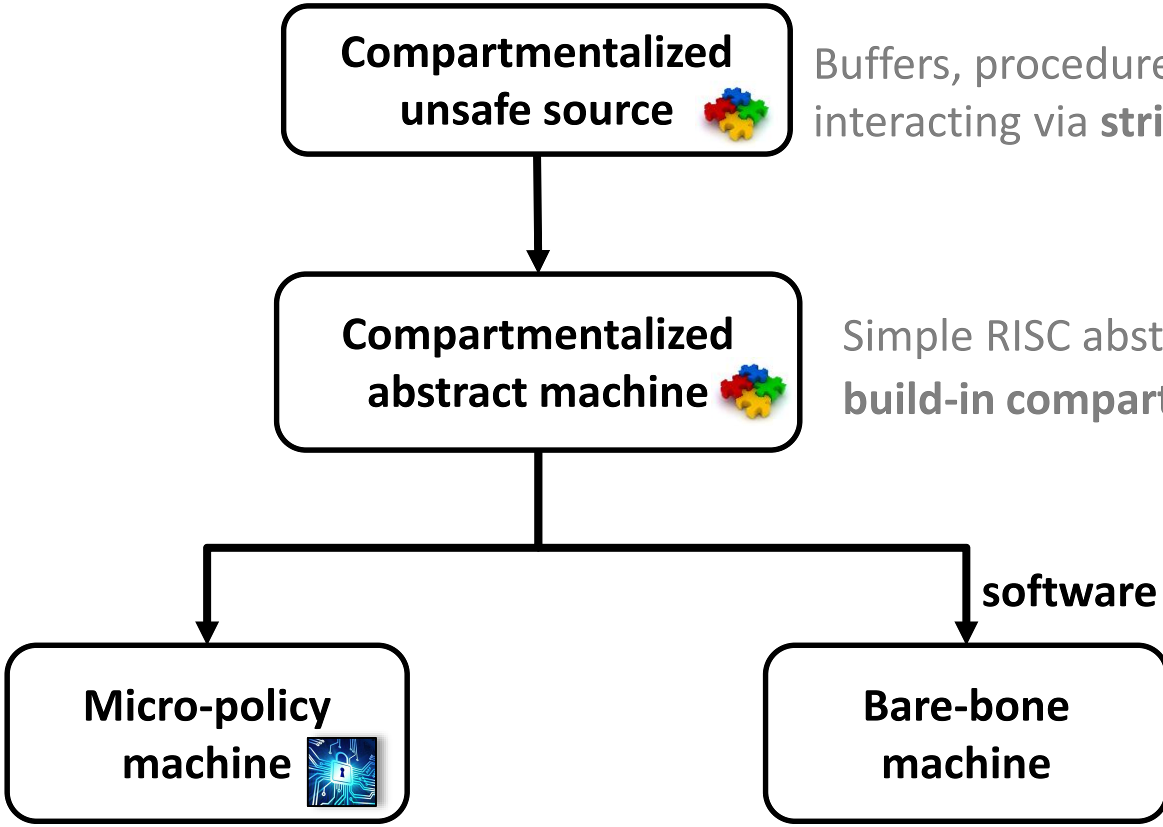
Simple RISC abstract machine with **build-in compartmentalization**




Buffers, procedures, components interacting via **strictly enforced interfaces**

Simple RISC abstract machine with **build-in compartmentalization**

**Tag-based reference monitor enforcing:**
- component separation
- procedure call and return discipline
(linear capabilities / linear entry points)



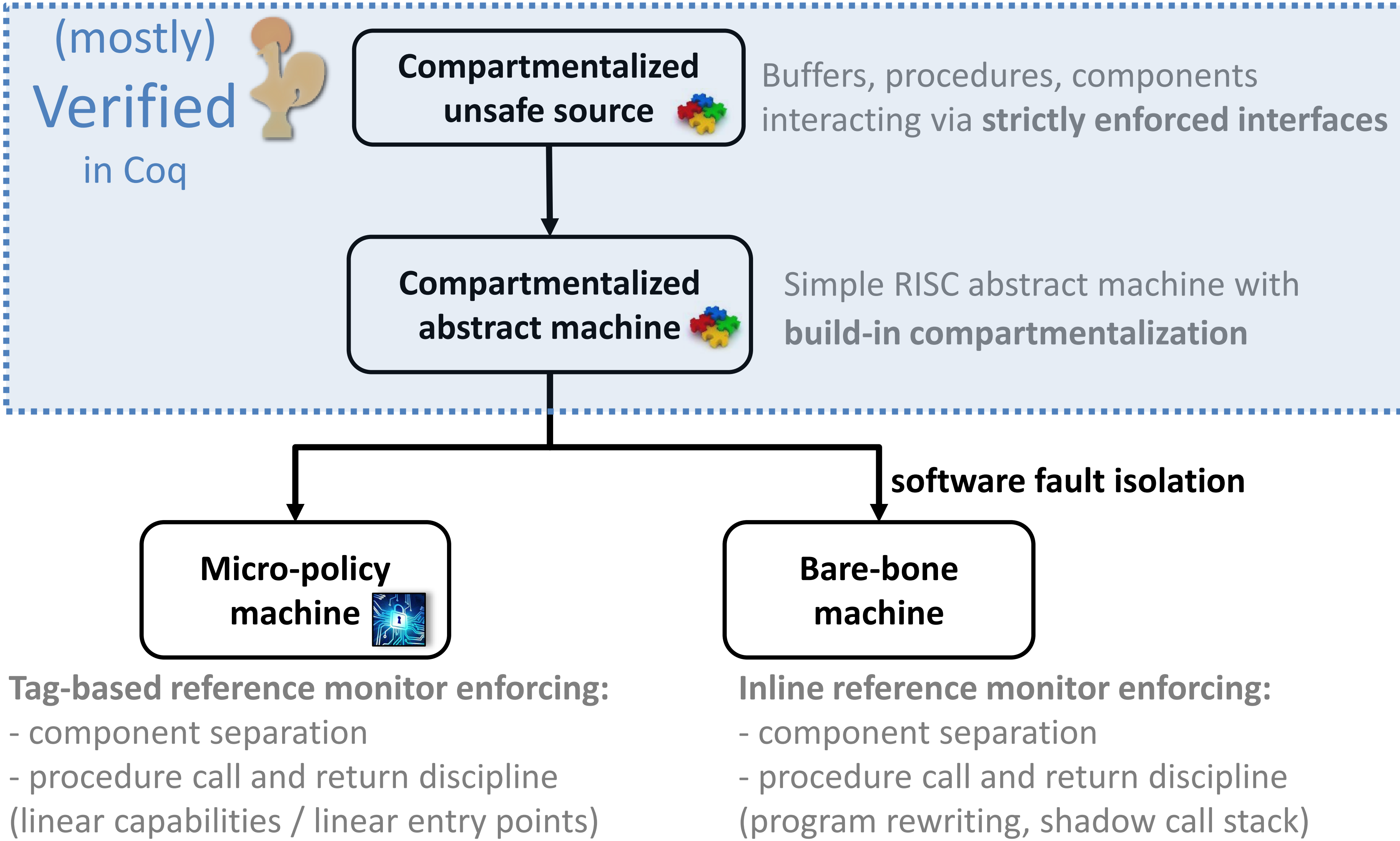

Buffers, procedures, components interacting via **strictly enforced interfaces**

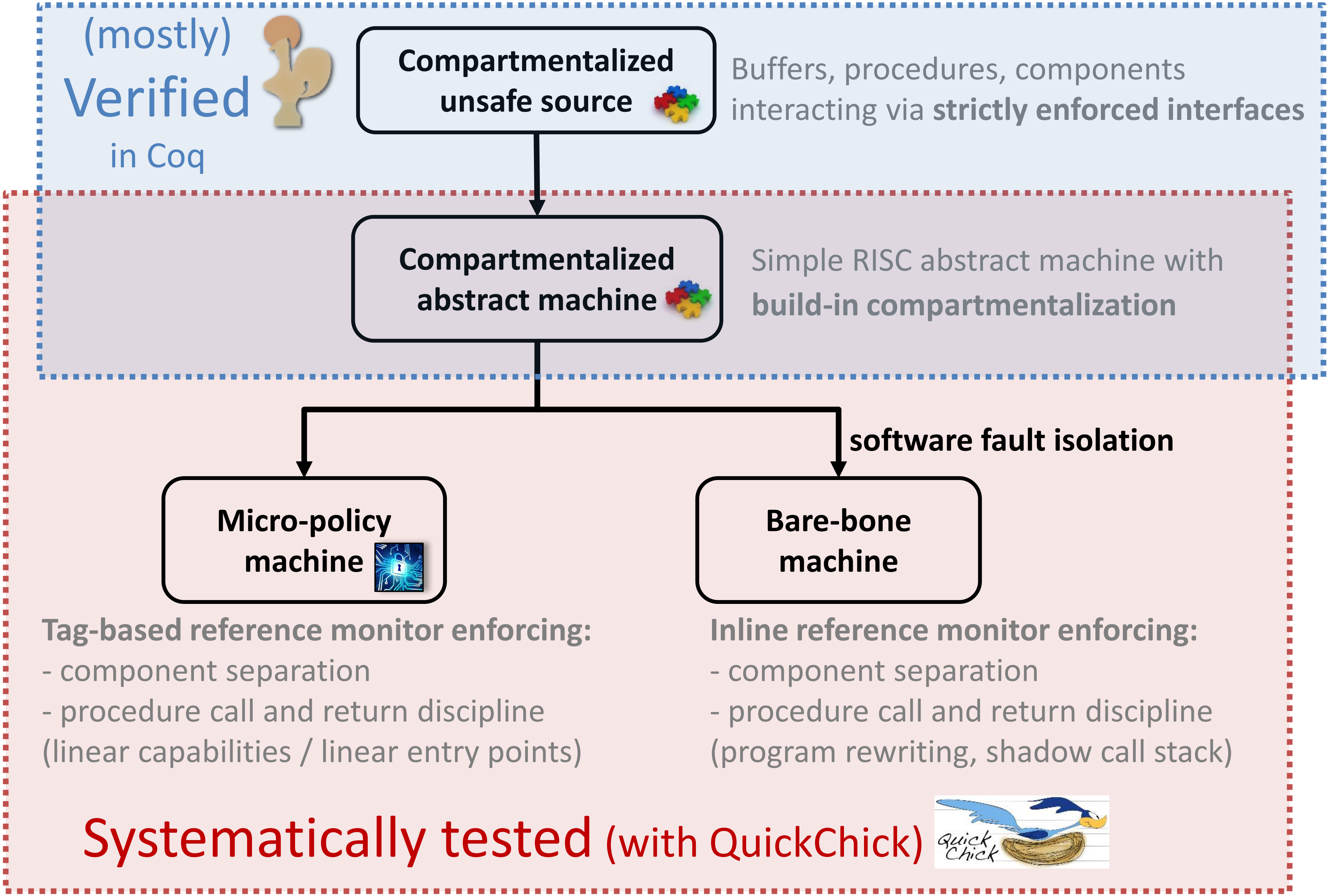
Simple RISC abstract machine with **build-in compartmentalization**

**software fault isolation**

**Tag-based reference monitor enforcing:**
- component separation
- procedure call and return discipline
(linear capabilities / linear entry points)

**Inline reference monitor enforcing:**
- component separation
- procedure call and return discipline
(program rewriting, shadow call stack)


(mostly)
Verified
in Coq
Compartmentalized unsafe source
Buffers, procedures, components interacting via **strictly enforced interfaces**
Compartmentalized abstract machine
Simple RISC abstract machine with **build-in compartmentalization**
software fault isolation
Micro-policy machine
Bare-bone machine
**Tag-based reference monitor enforcing:**
- component separation
- procedure call and return discipline
(linear capabilities / linear entry points)
**Inline reference monitor enforcing:**
- component separation
- procedure call and return discipline
(program rewriting, shadow call stack)

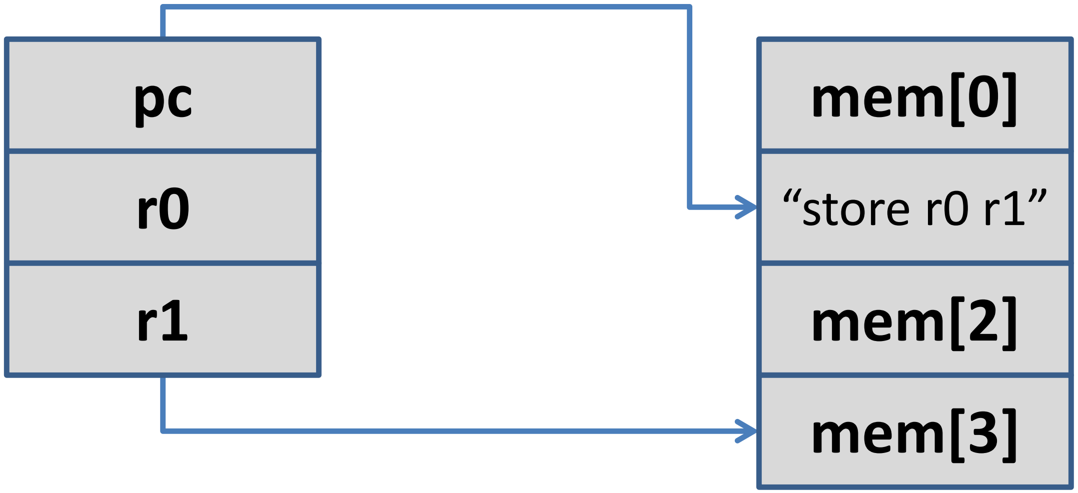

(mostly)
Verified
in Coq
Compartmentalized
unsafe source
Buffers, procedures, components
interacting via strictly enforced interfaces
Compartmentalized
abstract machine
Simple RISC abstract machine with
build-in compartmentalization
software fault isolation
Micro-policy
machine
Bare-bone
machine
Tag-based reference monitor enforcing:
- component separation
- procedure call and return discipline
(linear capabilities / linear entry points)
Inline reference monitor enforcing:
- component separation
- procedure call and return discipline
(program rewriting, shadow call stack)
Systematically tested (with QuickChick)
Quick
Chick

# Micro-Policies

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

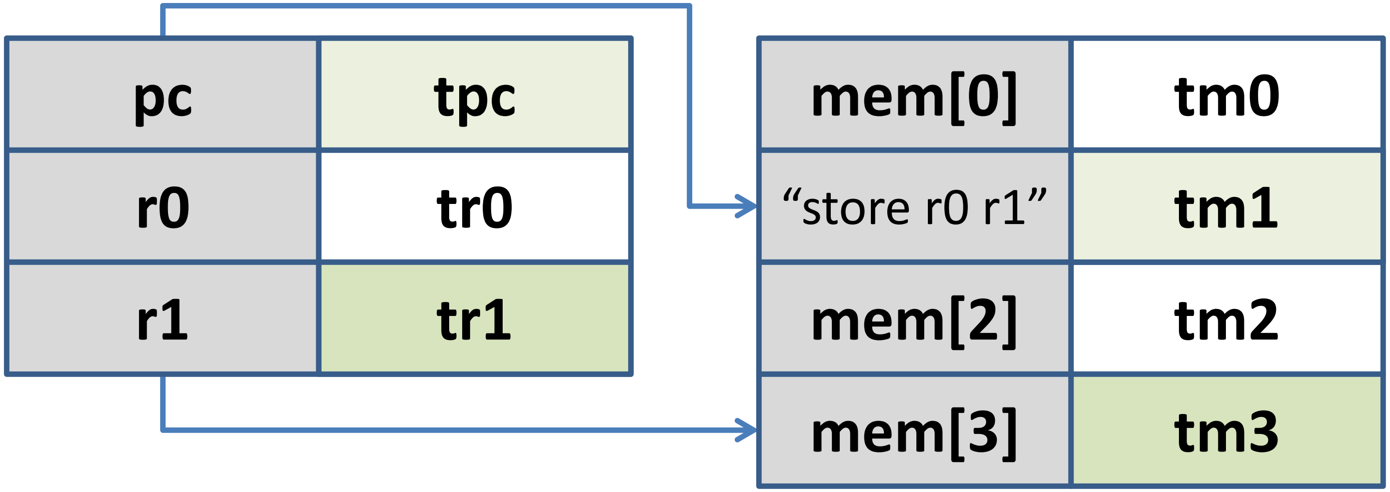
# Micro-Policies

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



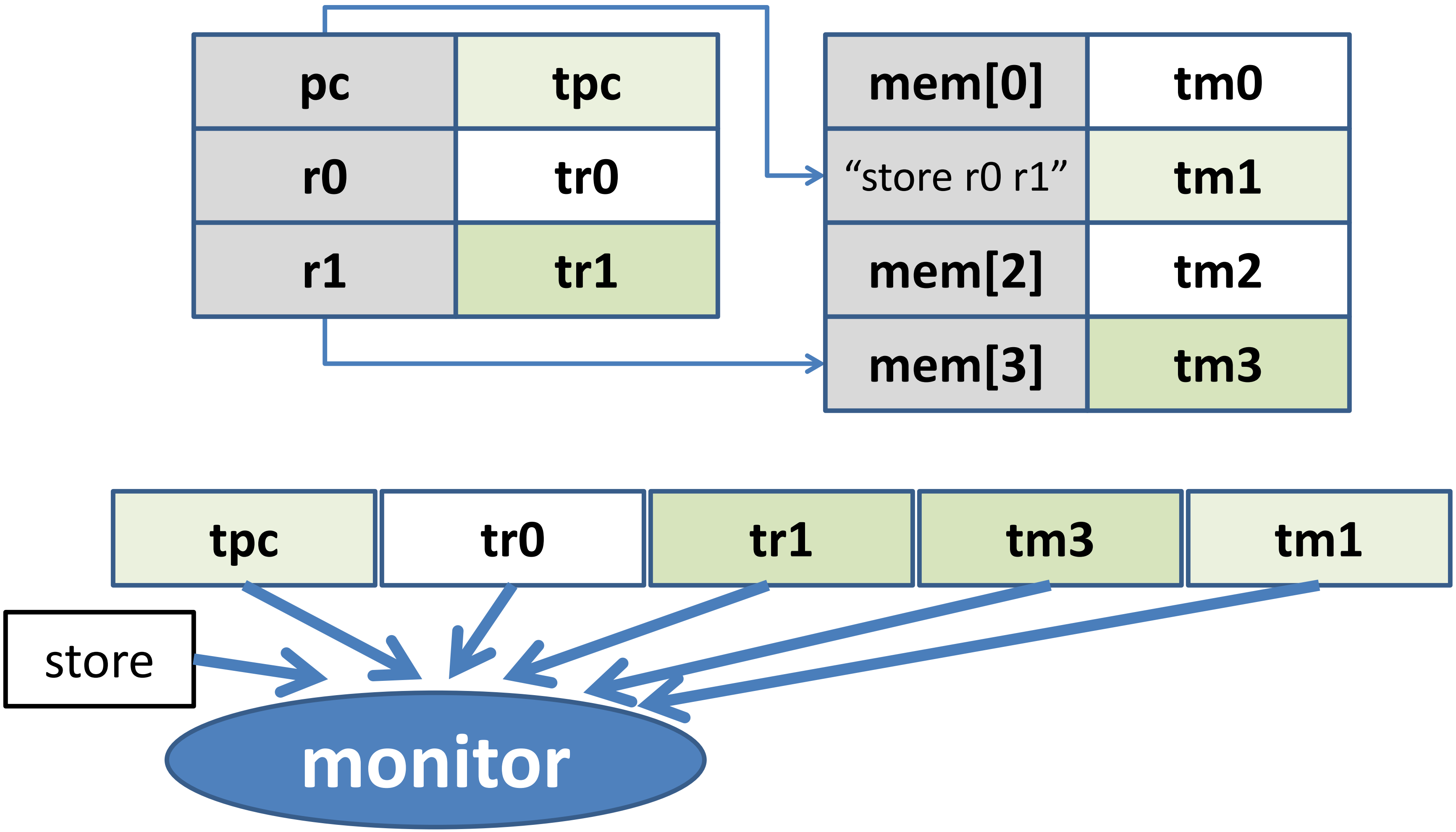
# Micro-Policies

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



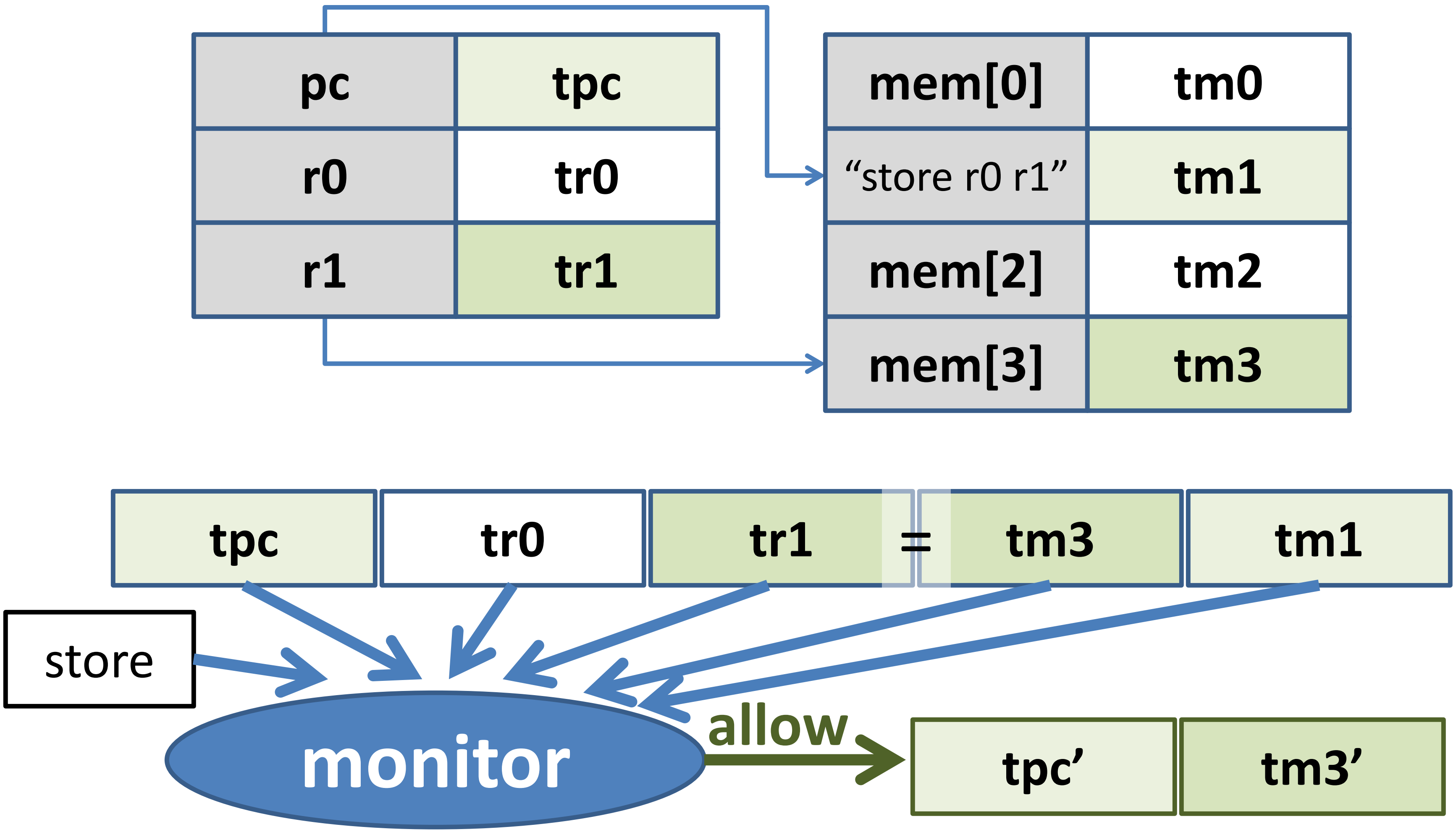
# Micro-Policies

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



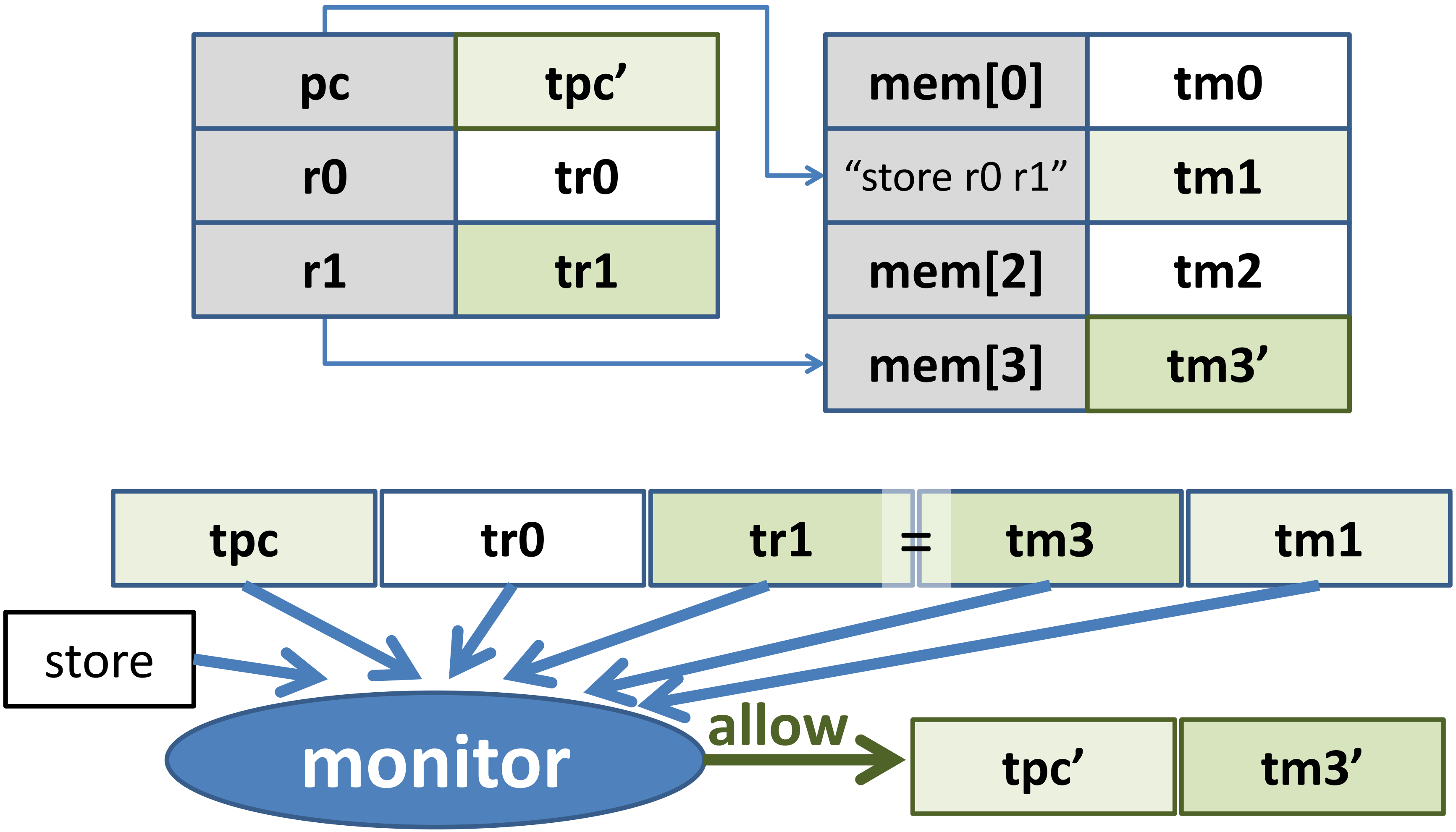
# Micro-Policies

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



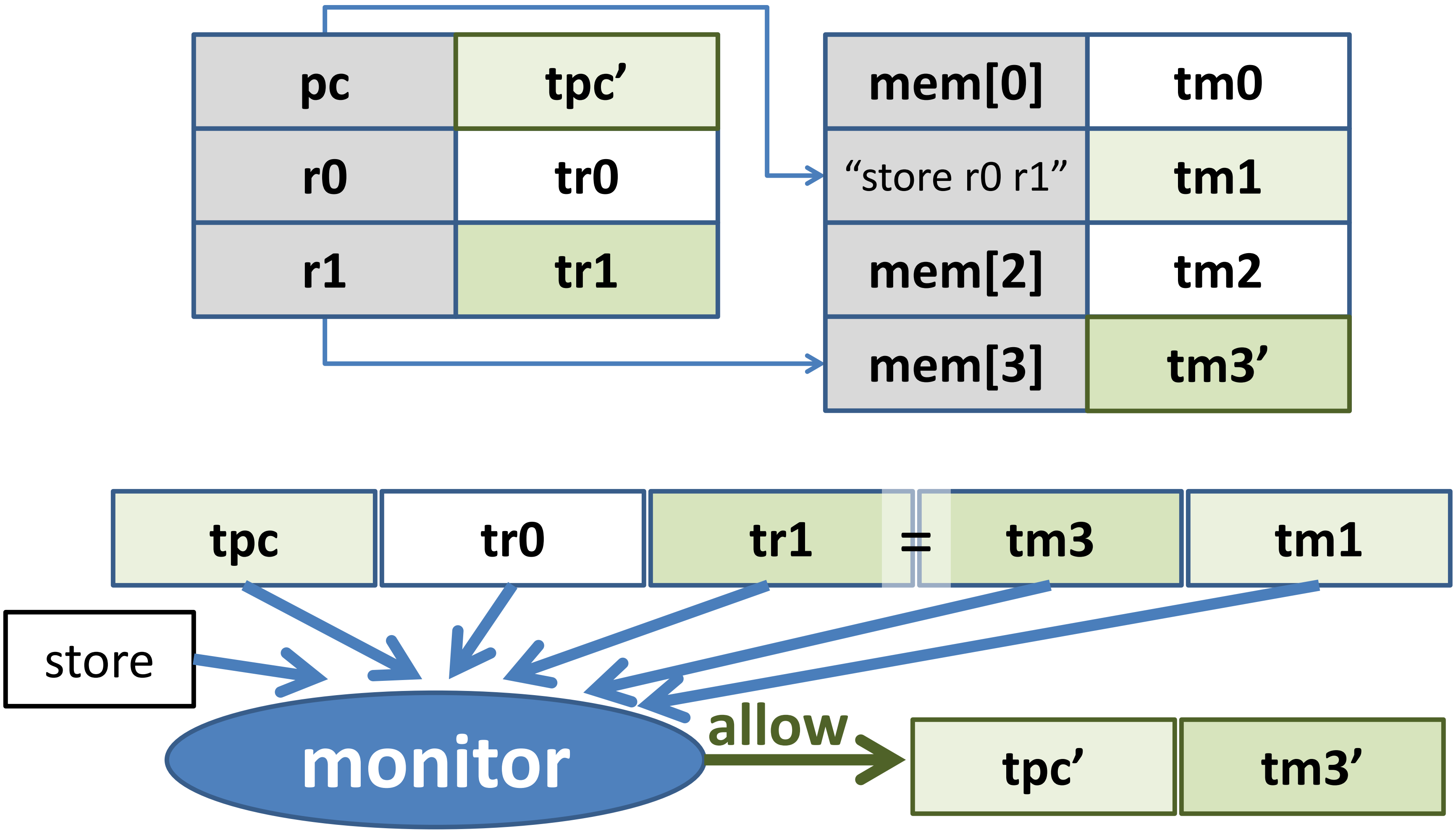
# Micro-Policies

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



# Micro-Policies

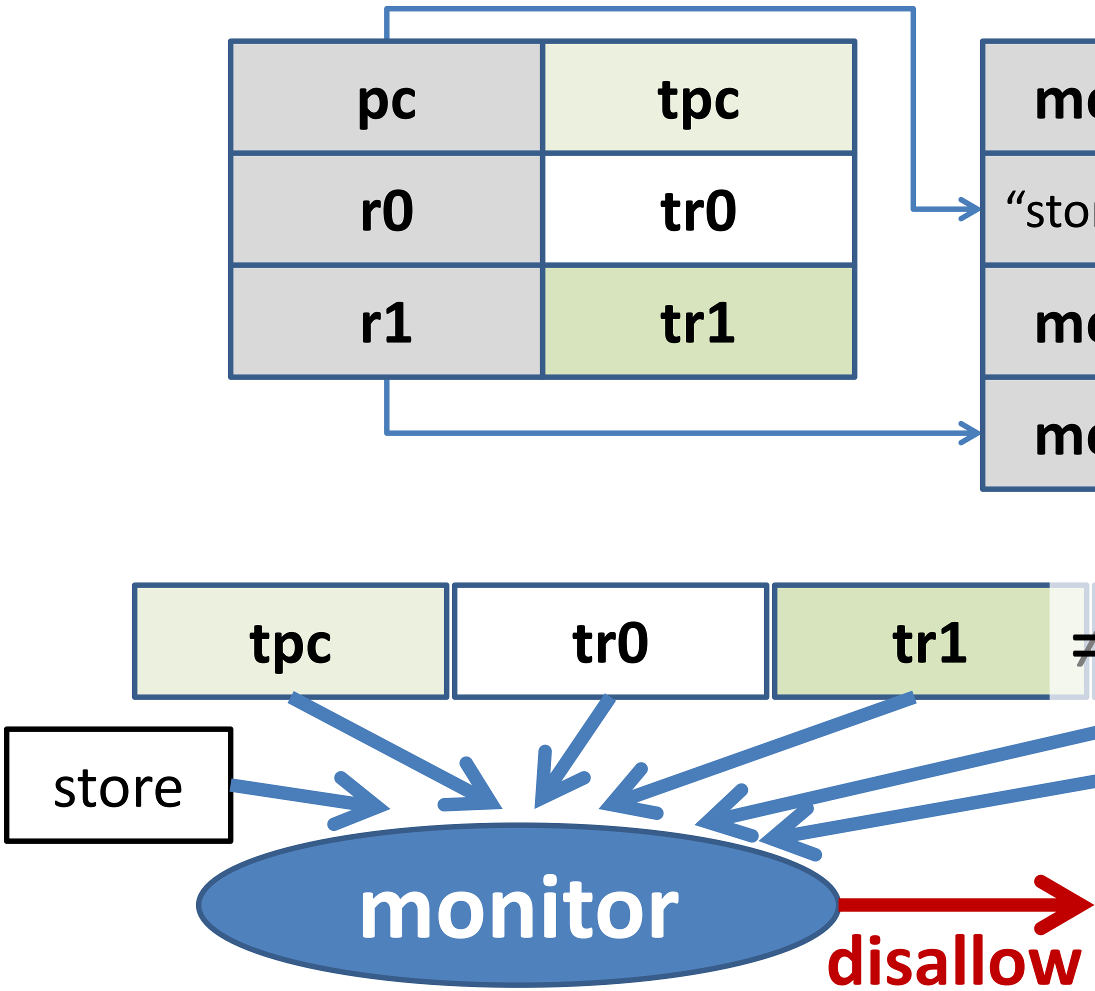
software-defined, hardware-accelerated, tag-based monitoring




**software monitor’s decision is hardware cached**

# Micro-Policies

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




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
- **expressive**: complex policies for secure compilation
- **secure** and **simple** enough to verify security in Coq
- **real**: FPGA implementation on top of RISC-V

DRAPER DOVER MICROSYSTEMS

# 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 DOVER MICROSYSTEMS

# 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]

spec

# Next steps towards making our secure compilation chain more practical

# Next steps towards making our secure compilation chain more practical

- **Scale up secure compilation to more of C**
  - first step: allow pointer passing (capabilities)

# Next steps towards making our secure compilation chain more practical

- **Scale up secure compilation to more of C**
  - first step: allow pointer passing (capabilities)
- **Verify compartmentalized applications**
  - put the source-level reasoning principles to work

# Next steps towards making our secure compilation chain more practical

- **Scale up secure compilation to more of C**
  - first step: allow pointer passing (capabilities)
- **Verify compartmentalized applications**
  - put the source-level reasoning principles to work
- **Extend all this to dynamic component creation**

# Next steps towards making our secure compilation chain more practical

- **Scale up secure compilation to more of C**
  - first step: allow pointer passing (capabilities)
- **Verify compartmentalized applications**
  - put the source-level reasoning principles to work
- **Extend all this to dynamic component creation**
- **... and dynamic privileges:**
  - capabilities, dynamic interfaces, HBAC, ...

# Next steps towards making our secure compilation chain more practical

- **Scale up secure compilation to more of C**
  - first step: allow pointer passing (capabilities)
- **Verify compartmentalized applications**
  - put the source-level reasoning principles to work
- **Extend all this to dynamic component creation**
- **... and dynamic privileges:**
  - capabilities, dynamic interfaces, HBAC, ...
- **Achieve confidentiality (hypersafety) preservation**
  - in a realistic attacker model **with side-channels,**
    but for this we probably need to clearly identify secrets

# Next steps towards making our secure compilation chain more practical

- **Scale up secure compilation to more of C**
  - first step: allow pointer passing (capabilities)
- **Verify compartmentalized applications**
  - put the source-level reasoning principles to work
- **Extend all this to dynamic component creation**
- **... and dynamic privileges:**
  - capabilities, dynamic interfaces, HBAC, ...
- **Achieve confidentiality (hypersafety) preservation**
  - in a realistic attacker model **with side-channels,**
    but for this we probably need to clearly identify secrets
- **Support other enforcement mechanisms (back ends)**

# Next steps towards making our secure compilation chain more practical

- **Scale up secure compilation to more of C**
  - first step: allow pointer passing (capabilities)
- **Verify compartmentalized applications**
  - put the source-level reasoning principles to work
- **Extend all this to dynamic component creation**
- **... and dynamic privileges:**
  - capabilities, dynamic interfaces, HBAC, ...
- **Achieve confidentiality (hypersafety) preservation**
  - in a realistic attacker model **with side-channels,** but for this we probably need to clearly identify secrets
- **Support other enforcement mechanisms (back ends)**
- **Measure & lower overhead**

# Formally Secure Compilation Despite Dynamic Compromise

- **restrict scope of undefined behavior**
  - **spatially** to the component that caused it
  - **temporally** by treating UB as an observable trace event

# Formally Secure Compilation Despite Dynamic Compromise

- **restrict scope of undefined behavior**
  - **spatially** to the component that caused it
  - **temporally** by treating UB as an observable trace event

- **We're hiring!**
  - Interns, PhD students, PostDocs, Young Researchers

# Formally Secure Compilation Despite Dynamic Compromise

- **restrict scope of undefined behavior**
  - **spatially** to the component that caused it
  - **temporally** by treating UB as an observable trace event

- **We're hiring!**
  - Interns, PhD students, PostDocs, Young Researchers


- **Another interesting event**
  - Workshop on Principles of Secure Compilation (PriSC) @ POPL