FOSAD 2015

15th International School on Foundations of Security Analysis and Design

Secure Compilation Using Micro-Policies

Motivation, Key Points and Ongoing Activities

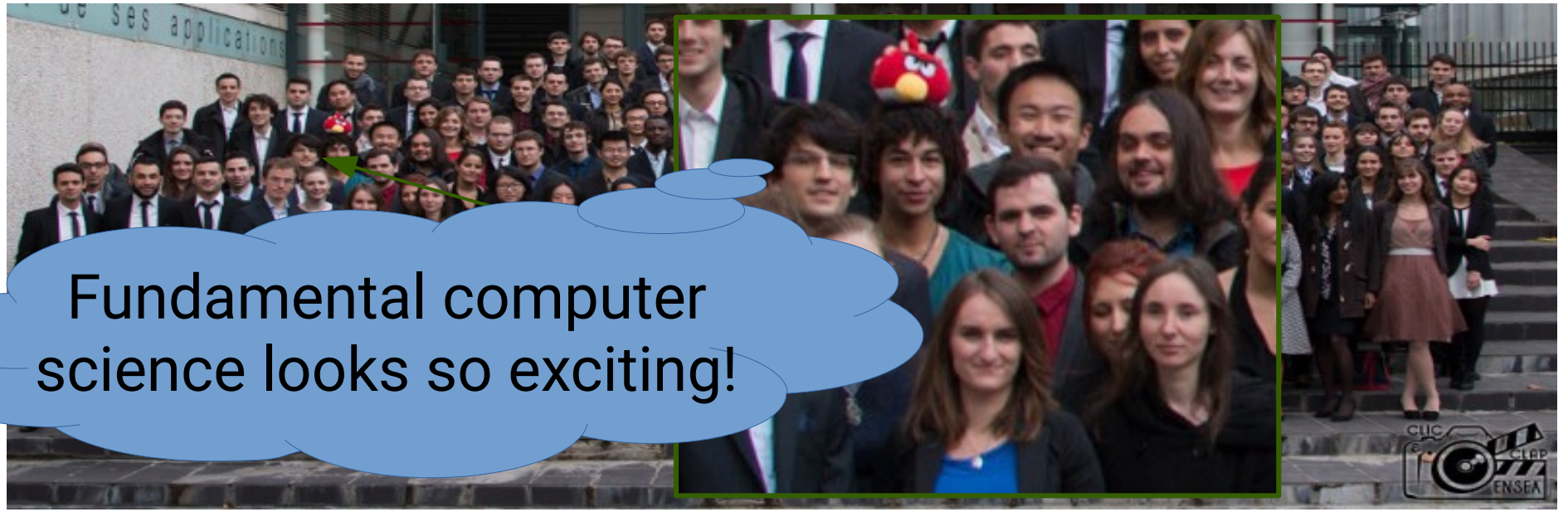
Yannis Juglaret

Inria Paris-Rocquencourt

Université Paris Diderot

Self-Introduction

- Master's degree in **computer engineering** and **digital systems** from ENSEA (France)
- **Research-oriented** Master's degree in **computer science** from Université Paris Diderot
- Inria PhD student with **C. Hriţcu** and **B. Blanchet**



Micro-Policies Project

Formal methods & **hardware architecture**

Current team:

- **UPenn**

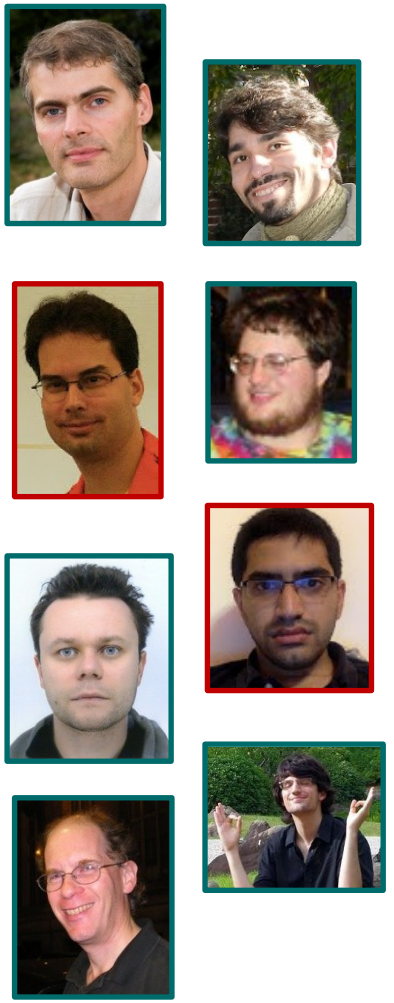
  Arthur Azevedo de Amorim, **André DeHon**, **Benjamin Pierce**, **Antal Spector-Zabusky**, **Udit Dhawan**

- **Inria**

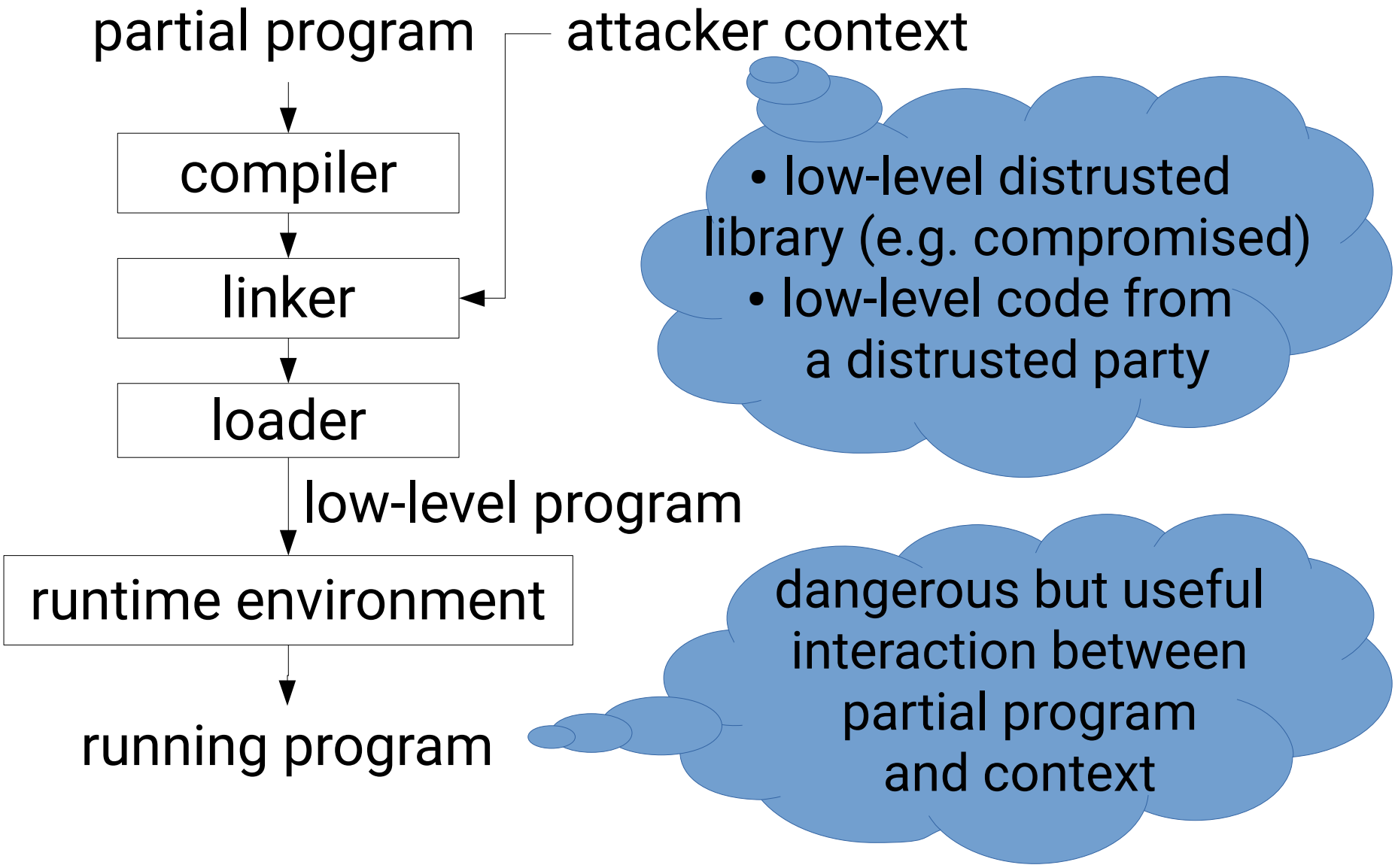
  Cătălin Hriţcu, **Yannis Juglaret**

- **Portland State**

  Andrew Tolmach

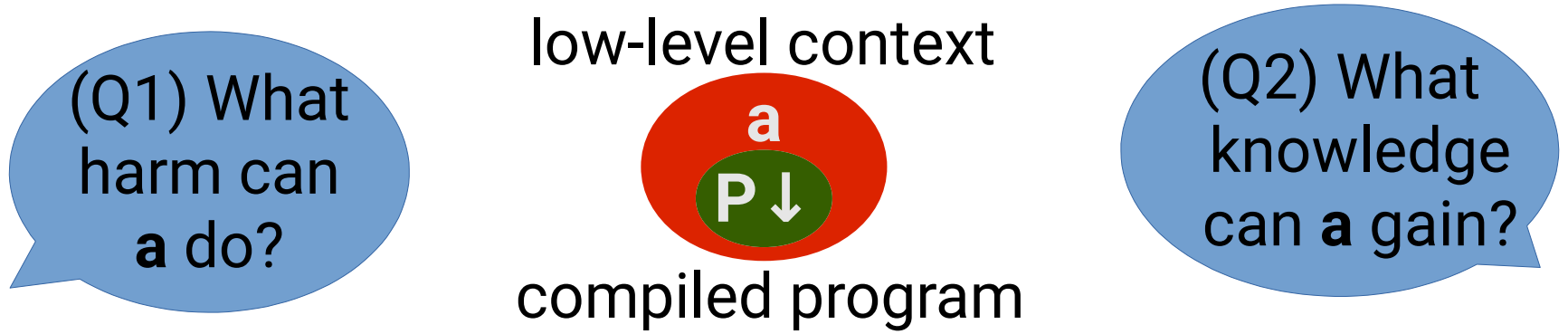


Typical Secure Compilation Layout

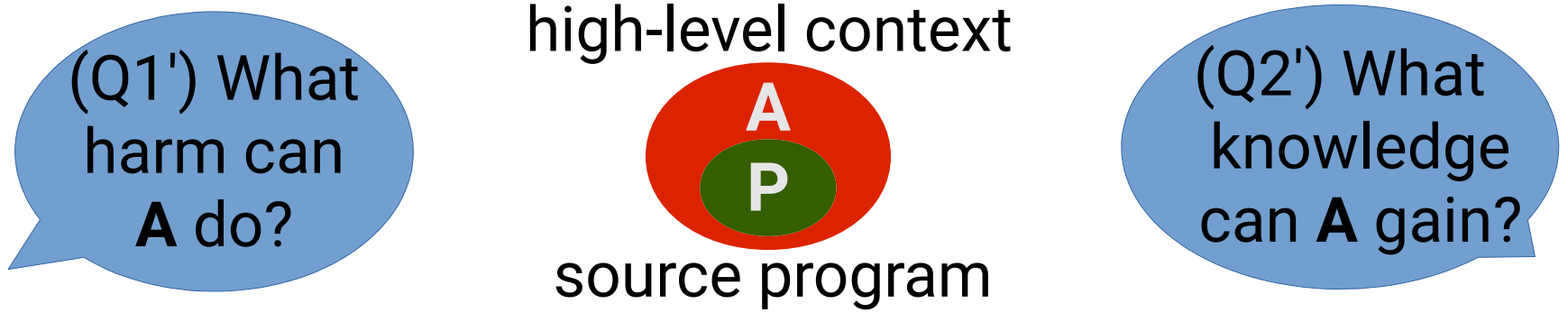


Partial Programs Security Reasoning

- Study **interaction** with attacker contexts

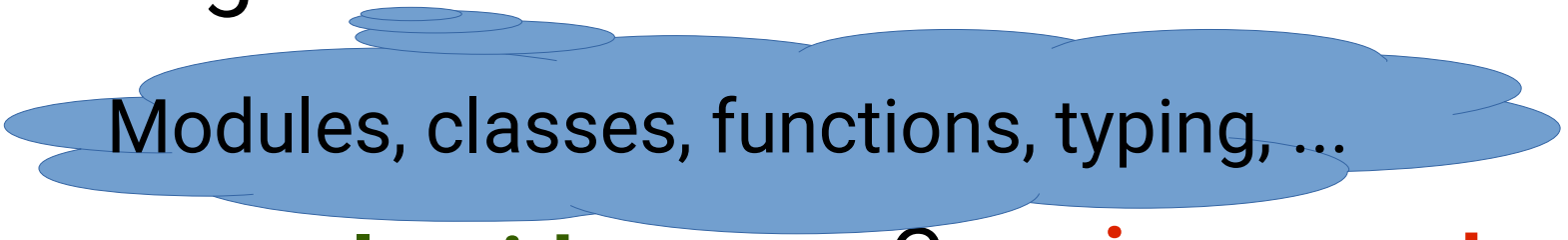


- Secure compilation **reduces** Q1,Q2 to Q1',Q2'



- SC enables **easier, source-level reasoning**

Comparing Attacker Power



- Must **comply with abstractions**:
  - More **structure**
  - Restricted **interaction** and **communication**
- Hence **restricted attacker power**

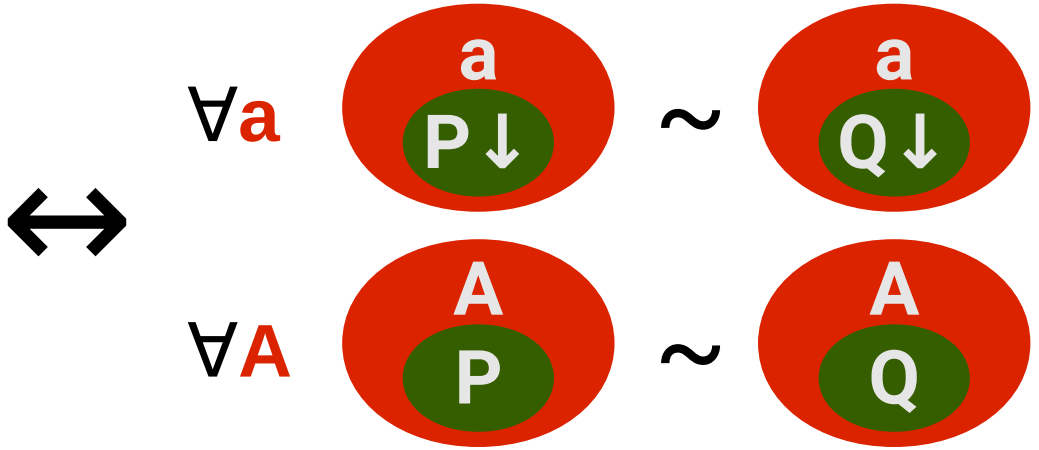
- Can **circumvent** abstractions:
  - Lacks structure
  - Typically unrestricted
    - Reads, writes, or jumps targeting any location
    - Untyped calls

Reasoning with high-level attackers is much easier!

Full Abstraction

- A **formal characterization** of secure compilation

  “no low-level attacker can distinguish **P↓** from **Q↓**”



  “no high-level attacker can distinguish **P** from **Q**”

- Low- and high-level attackers **equally powerful**

  low-level ones **can't do more harm**

- **Very strong** property, hard to achieve

Enforcing Full Abstraction

- Apply standard security techniques
  - **randomization**: e.g. of address space layout
  - **code rewriting**: add extra software checks
- Use enhanced hardware
  - **protected module architectures**: e.g. Intel SGX
  - **generic protection mechanisms**?
- Our approach: use **micro-policies**
  - **efficient hardware implementation**: the PUMP
  - **mechanized metatheory**

ASPLOS '15 | Oakland S&P '15

Challenges in Secure Compilation

- **Efficiency**
  - main challenge in the domain
  - in our case, we want **low-overhead monitoring**
  - good hopes: **4 complex policies <10% overhead**
- **Transparency**
  - mustn't reject **benign low-level contexts**, neither **statically** nor **dynamically**
  - in our case, we want **permissive monitoring**
  - need to enforce **exactly what is required**

A Finer-Grained View on Programs

- FA is stated for **contexts** and **partial programs**
- Let us view programs as **linked components**

- The previous setting is recoverable

Starting Simple: Our Source Language

- Simple **class-based object-oriented language**

a component = **a class** + **objects of that class**

public methods, private fields static object definitions static typing	no primitive types no inheritance no dynamic allocation

```
e ::= this | arg | o          reference
    | e.f | e.f := e          selection, update
    | e.m(e)                  call
    | e == e ? e : e          object identity test
    | exit e                  early termination
    | e; e                    sequence
```

- **Many more abstractions** than you would expect

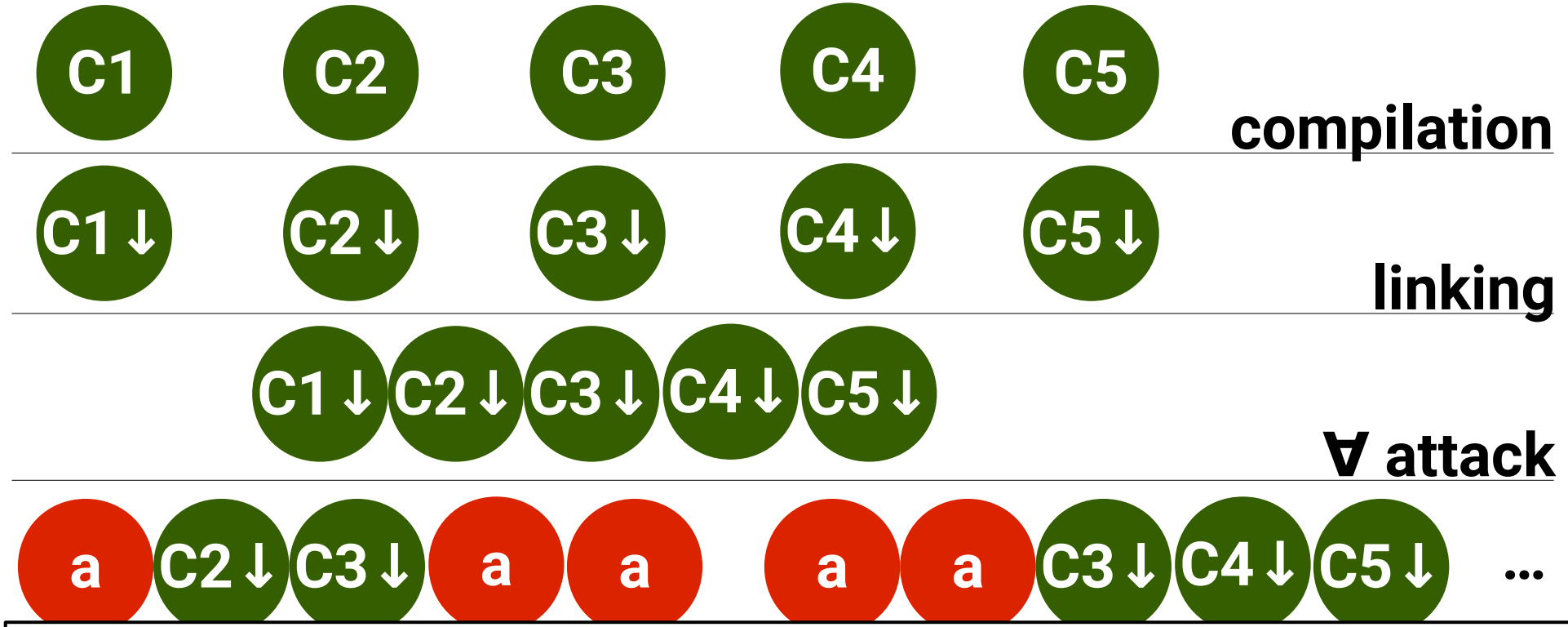
High-Level Abstractions

- Class isolation
  - **fields** are **private**
  - classes **can't read/write** each other's **code/data**
- Method call discipline
  - **method calls/returns** are the **only way to interact**
  - callees return **where callers expect them to**
  - callees give **no information** to callers except result
- Type safety

  method arguments/results are **well-typed**

What We Provide

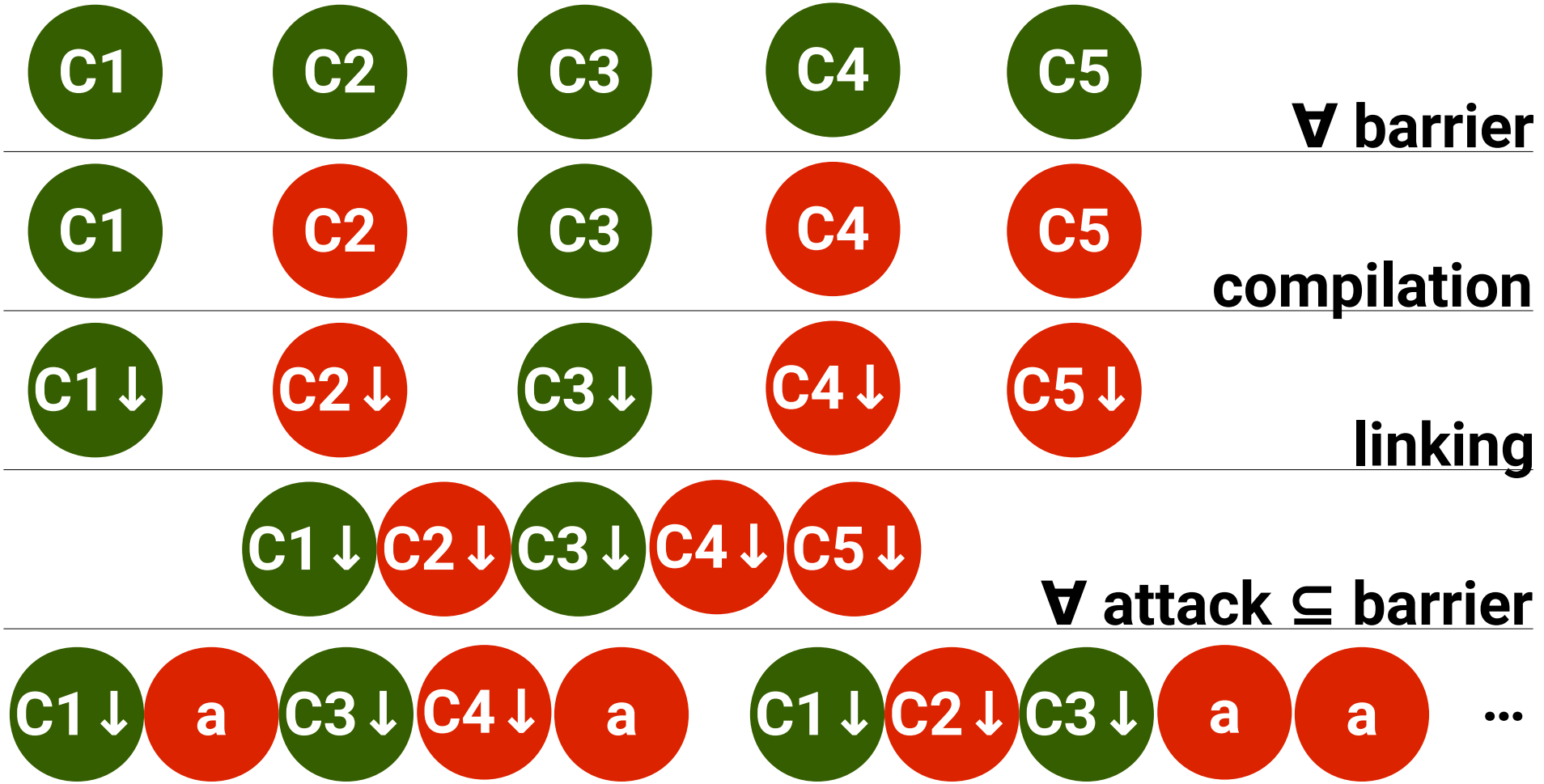
- **Mutual distrust at compile-time**
- **SC** for **non-compromised components**



→ compiler has **no knowledge** about **where** attacks happen
→ protection in **every dynamic compromise scenario**,
for every component that **behaves well**

What Full Abstraction Provides

- **Trusted/distrusted known at compile-time**
- Static protection: for **trusted components only**



Beyond Full Abstraction

- We achieve **more** than necessary

  need for a **full abstraction-like characterization**

- Two extended abstracts presented at FCS'15:
  - ***Multi-Module Full Abstraction*** by Patrignani et al.
  - ***Secure Compilation Using Micro-Policies***
- Ongoing research activities:
  - How to **properly formalize** this?
  - Do **proof techniques** for FA generalize? How?
  - Do **previous works** provide such guarantees?

Ongoing Activities and Future Work

- **Formalize our secure compilation**
- **Prove** full abstraction

- **Measure** efficiency
- **Test** transparency, **mitigate** with **wrappers**
- (**Extend** current language)
- Move to **functional languages**
- Study **micro-policy composition**

Bibliography

M. Abadi. Protection in programming-language translations, 1998.

Ú. Erlingsson. Low-level software security: Attacks and defenses, FOSAD 2007.

L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal war in memory. Oakland S&P 2013.

A. Azevedo de Amorim, N. Collins, A. DeHon, D. Demange, C. Hriţcu, D. Pichardie, B. C. Pierce, R. Pollack, and A. Tolmach. A verified information-flow architecture. POPL 2014.

U. Dhawan, C. Hriţcu, R. Rubin, N. Vasilakis, S. Chiricescu, J. M. Smith, T. F. Knight, Jr., B. C. Pierce, and A. DeHon. Architectural support for software-defined metadata processing, ASPLOS 2015.

A. Azevedo de Amorim, M. Dénès, N. Giannarakis, C. Hriţcu, B. C. Pierce, A. Spector-Zabusky, and A. Tolmach. Micro-policies: Formally verified, tag-based security monitors. Oakland S&P 2015.

M. Abadi and G. D. Plotkin. On protection by layout randomization, TISSEC 2012.

C. Fournet, N. Swamy, J. Chen, P. Dagand, P. Strub, and B. Livshits. Fully abstract compilation to Javascript, POPL 2013.

M. Patrignani, P. Agten, R. Strackx, B. Jacobs, D. Clarke, and F. Piessens. Secure compilation to protected module architectures. PLAS 2015.

M. Patrignani, D. Devriese, and F. Piessens. Multi-module fully abstract compilation (extended abstract). FCS 2015.

Y. Juglaret and C. Hriţcu. Secure compilation using micro-policies (extended abstract). FCS 2015.

M. Patrignani. The Tome of Secure Compilation: Fully Abstract Compilation to Protected Modules Architectures. PhD thesis, KU Leuven, 2015.

FIN

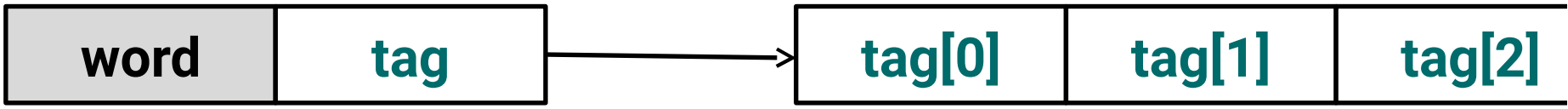
ASPLOS '15

Micro-Policies

Oakland S&P '15

- Add **large tag** to each machine word

unbounded metadata



- Words in memory and registers are all tagged

pc	tag
r0	tag
r1	tag
r2	tag

mem[0]	tag
mem[1]	tag
mem[2]	tag
mem[3]	tag

pc

* conceptual model, the hardware implements this efficiently

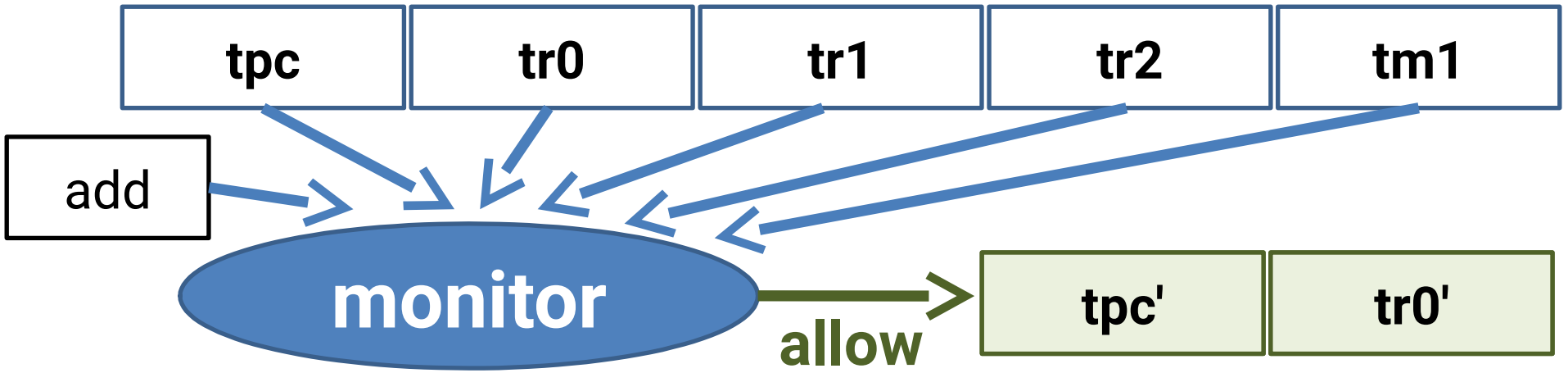
Tag-Based Instruction-Level Monitoring

pc	tpc
r0	tr0
r1	tr1
r2	tr2

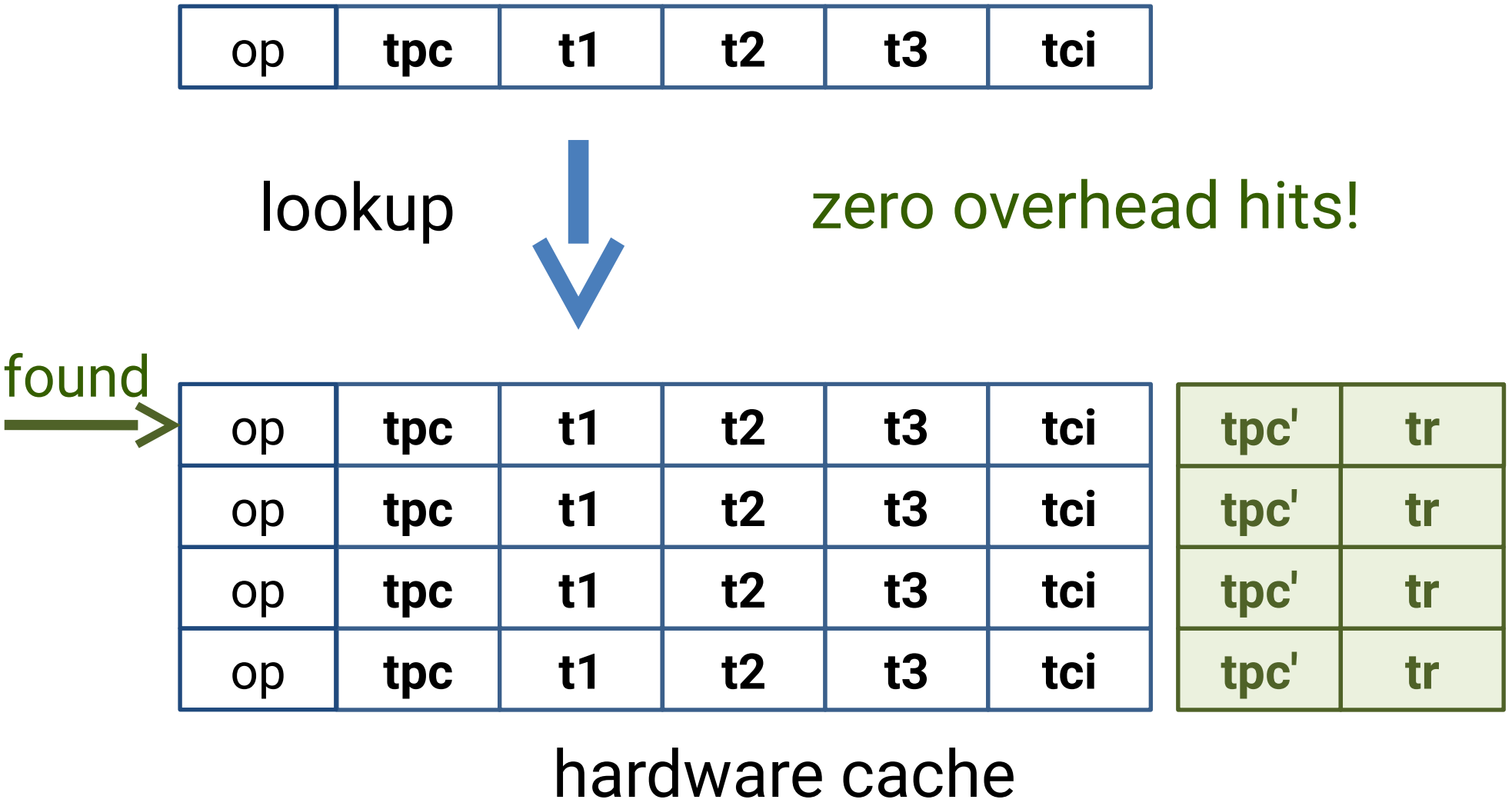
mem[0]	tm0
mem[1]	tm1
mem[2]	tm2
mem[3]	tm3

pc

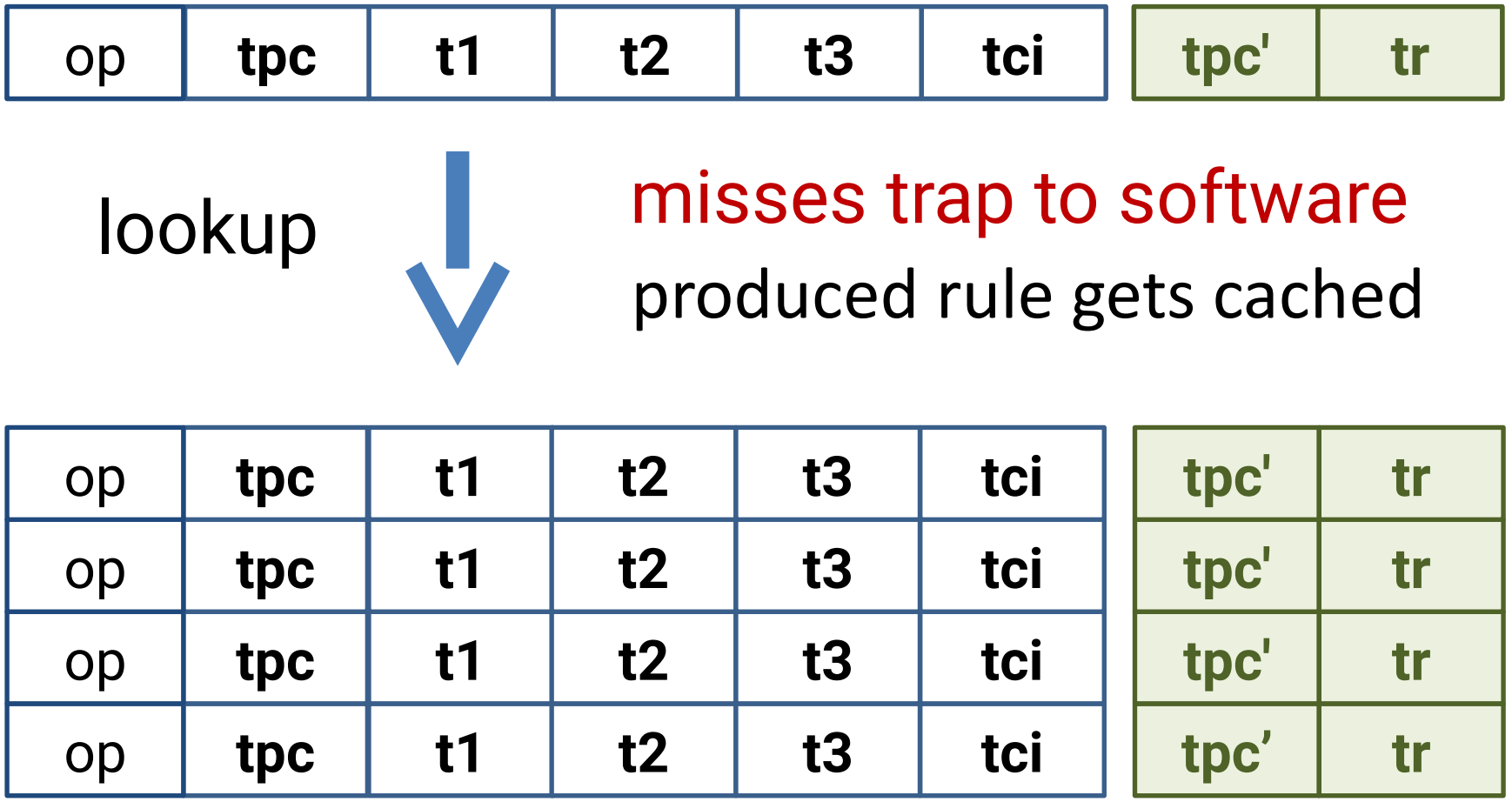
decode(mem[1]) = add r0 r1 r2



Efficiently Executing Micro-Policies



Efficiently Executing Micro-Policies

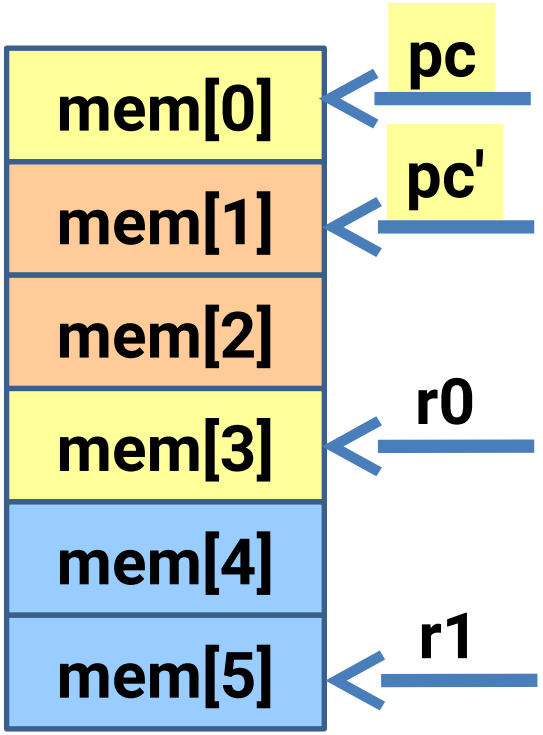


Micro-Policy Enforcing Abstractions

- Class isolation: by **compartmentalization**
- Method call discipline
  - distinguish **entry points**
  - use **linear return capabilities**
  - **clear registers** upon call and return
- Type safety: via **dynamic type-checking**
  - tag values with their **type**
  - tag entry points with **method signatures**
  - tag return capability with **expected return type**

Isolation Micro-Policy

Memory+PC tags embed a **class name** (a color)



decode(mem[0]) = store r0 r2

store: **tpc = tm0 = tm3 … → ✓ tpc tm3**

decode(mem[1]) = nop

_: tpc' ≠ tm1 … → ✗ failstop

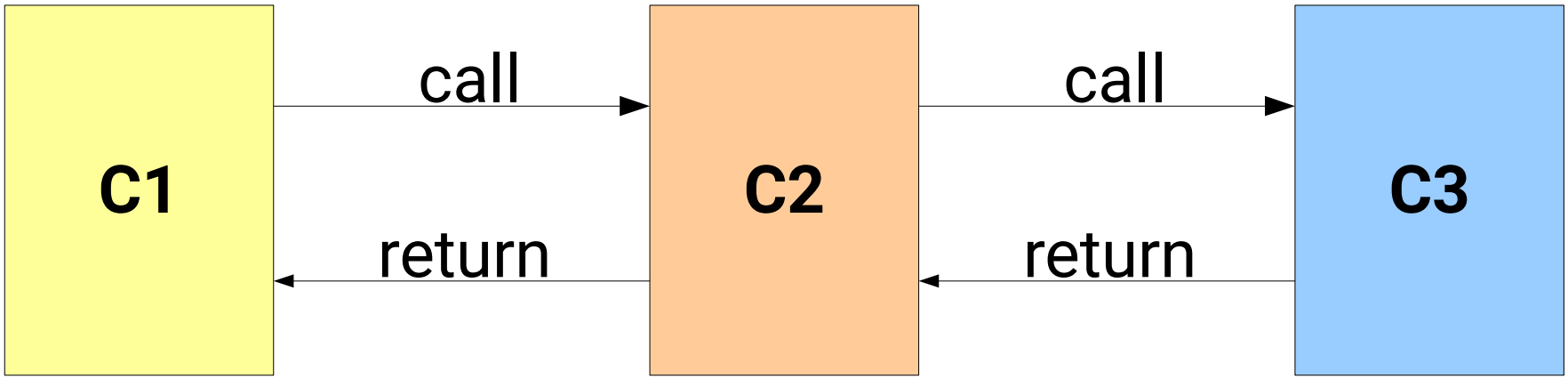
decode(mem[0]) = load r1 r2

load: tpc = tm0 ≠ tm5 … → ✗ failstop

Compilation of Method Calls

Low-level **call instruction**: Jal, jump and link

callee gets a **return address** in register ra



Matching sequence of low-level instructions:

Jal	Store ra	Jal	Jump ra	Load ra	Jump ra

Method Call Discipline Micro-Policy

- Use a different tag for **method entry points**
- Track **call depth** on PC tag
- Use **linear return capabilities**

```
Jal:    pc@d    m@Entry → pc@d+1  ra@d+1
Jump:   pc@d+1  r@d+1   → pc@d    r@⊥
Mov:    r@d             → r'@d    r@⊥
Store:  r@d             → m@d     r@⊥
Load:   m@d             → r@d     m@⊥
```

Jal | Store ra | Jal | Jump ra | Load ra | Jump ra