FCS 2015
Workshop on Foundations of Computer Security

Secure Compilation Using Micro-Policies

Yannis Juglaret [1,2]
Cătălin Hriţcu [1]

[1] Inria Paris-Rocquencourt
[2] Université Paris Diderot

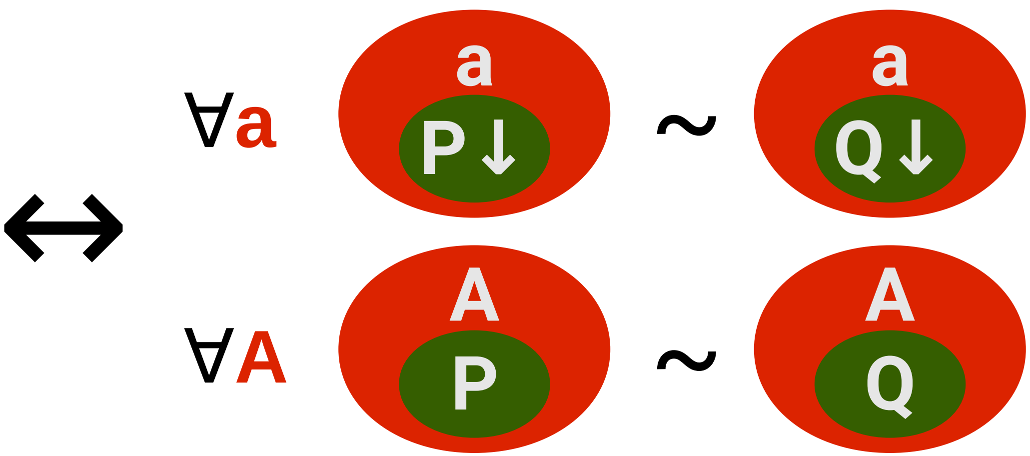
Motivating Secure Compilation

- **Abstractions** help **reasoning** by giving **structure**
  modules, **classes**, **functions**, etc.
- Compiled programs **run in the low-level**
  surrounding environment seen as **an attacker**
- **Secure compilation** preserves **abstractions**
  - low-level attackers **can't bypass** abstractions
  - **reasoning in the high-level** becomes sufficient
- **Challenging problem** with **inefficient solutions**
  too expensive, usual compilers are **not secure**

Secure Compilation by Full Abstraction

- About **partial programs** in an **attacker context**

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



compiled programs in a low-level context

source programs in a high-level context

“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

Micro-Policies Project

- **Formal methods** & **hardware architecture**
- Current team
  - **UPenn**

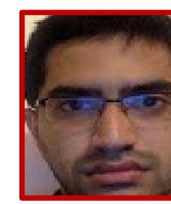
    Arthur Azevedo de Amorim,
    André DeHon, **Benjamin Pierce**,
    Antal Spector-Zabusky,
    Udit Dhawan
  - **Inria**

    Cătălin Hrițcu, **Yannis Juglaret**
  - **Portland State**

    Andrew Tolmach








Micro-Policies

Oakland S&P '15

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

unbounded metadata

word	tag

tag[0]	tag[1]	tag[2]

- 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

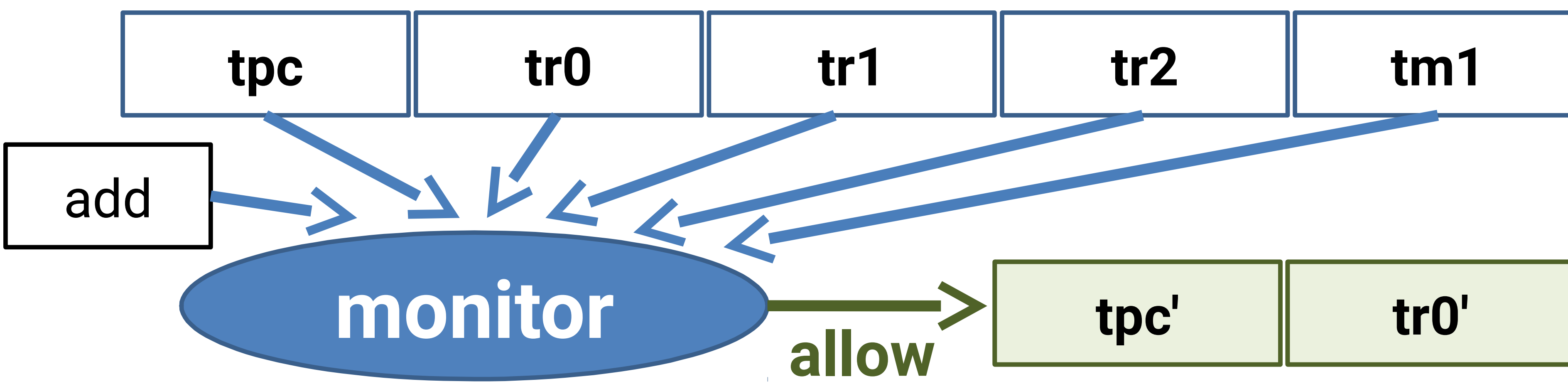
Tag-Based Instruction-Level Monitoring

pc	tpc
r0	tr0
r1	tr1
r2	tr2

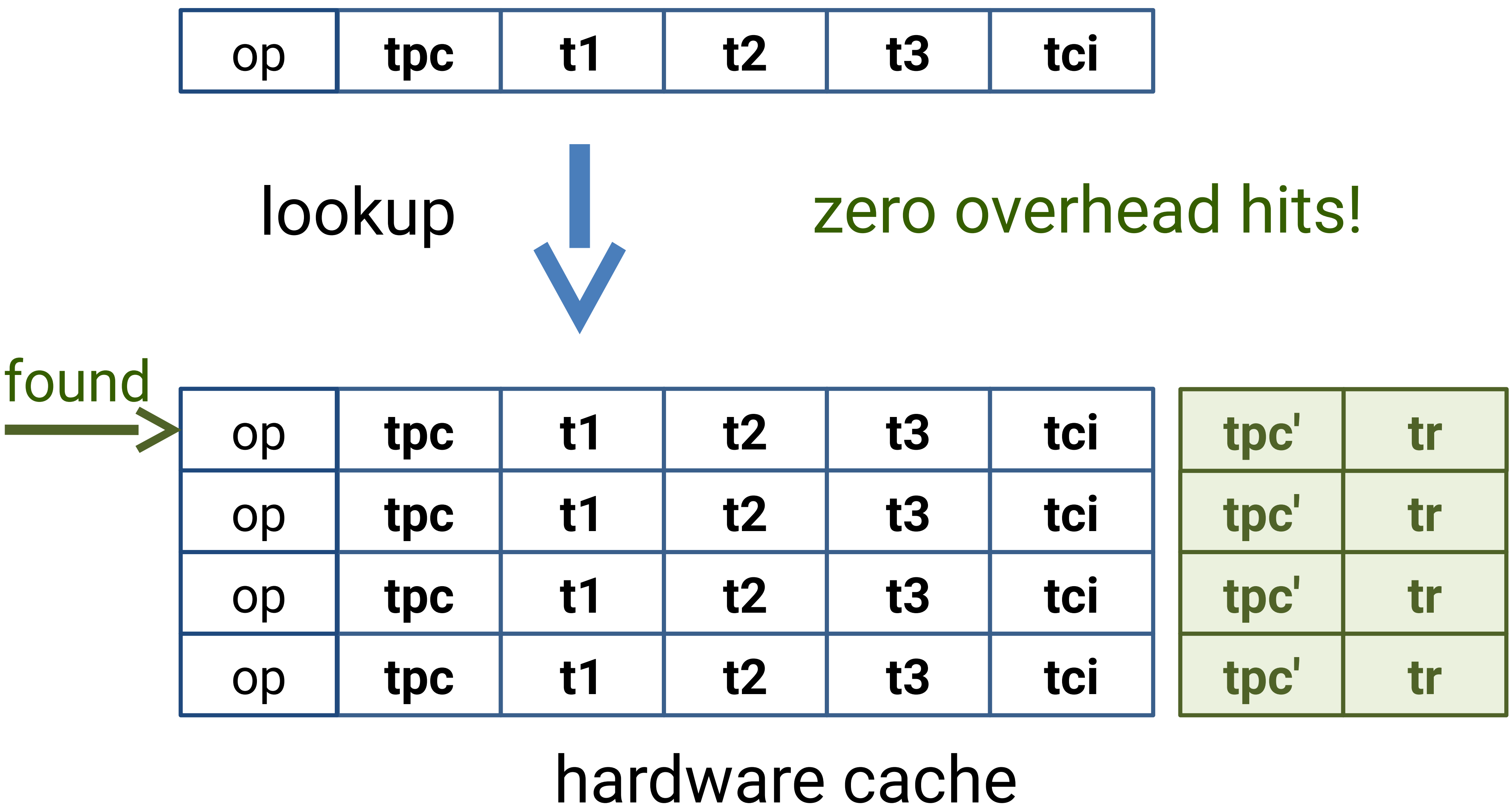
mem[0]	tm0
mem[1]	tm1
mem[2]	tm2
mem[3]	tm3

pc

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



Efficiently Executing Micro-Policies



Efficiently Executing Micro-Policies

op	**tpc**	**t1**	**t2**	**t3**	**tci**	**tpc'**	**tr**

lookup

misses trap to software

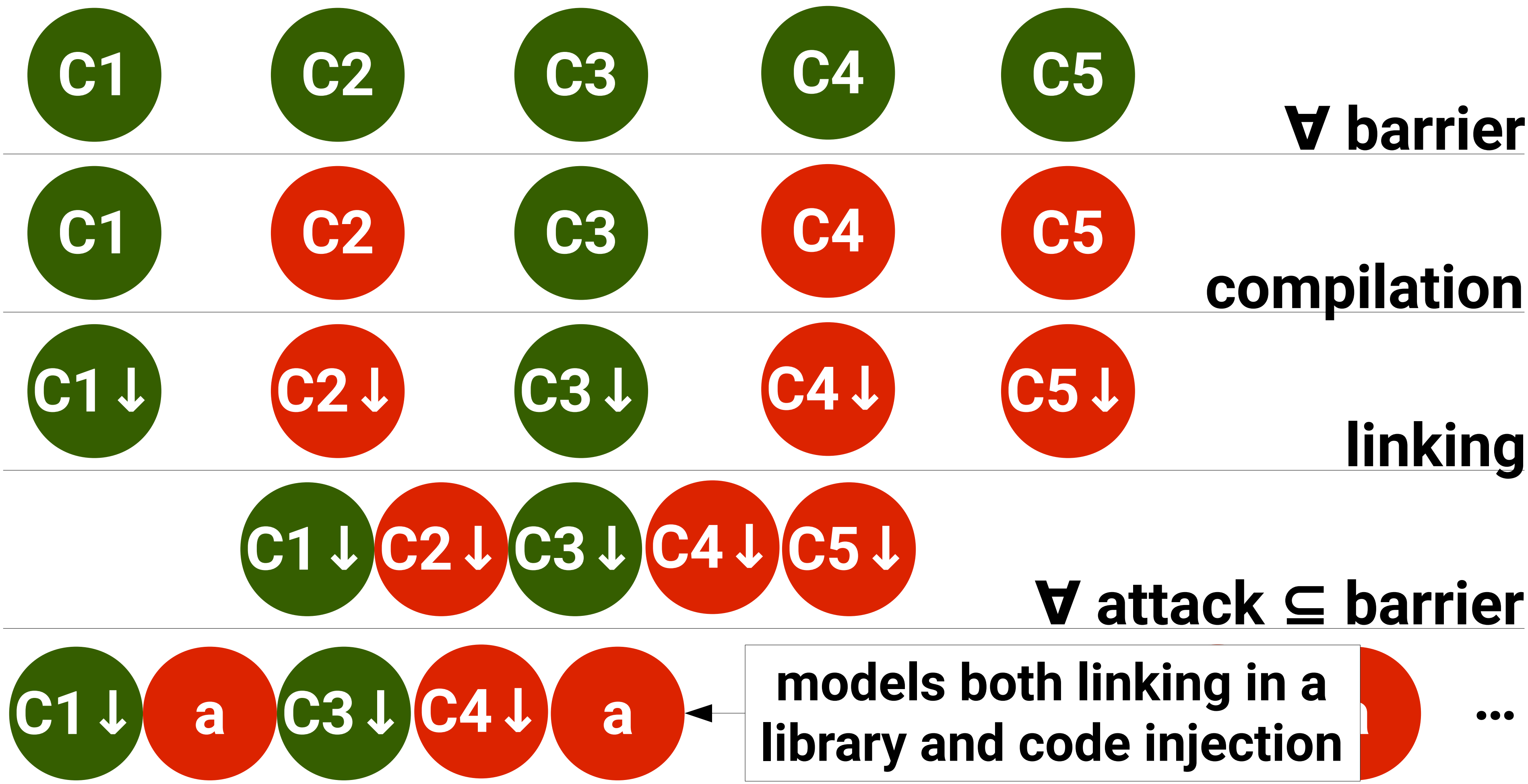
produced rule gets cached

op	**tpc**	**t1**	**t2**	**t3**	**tci**	**tpc'**	**tr**
op	**tpc**	**t1**	**t2**	**t3**	**tci**	**tpc'**	**tr**
op	**tpc**	**t1**	**t2**	**t3**	**tci**	**tpc'**	**tr**
op	**tpc**	**t1**	**t2**	**t3**	**tci**	**tpc’**	**tr**

hardware cache

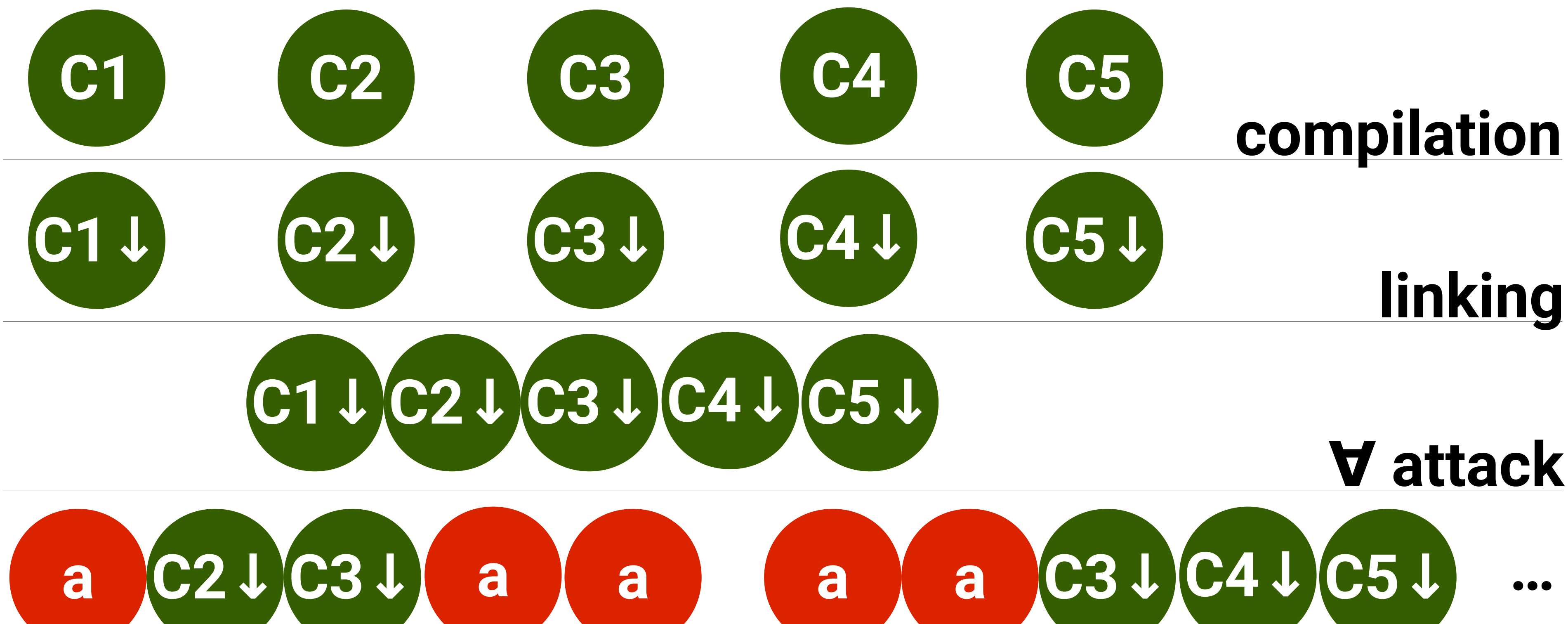
A First Attacker Model

- **Trusted**/**distrusted known at compile-time**
- **SC** for **trusted components** only



A Stronger Attacker Model

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



compiler has **no knowledge** about **where** attacks happen
→ protection in **every compromise scenario**

Goals and Challenges

- **Protection** using **monitoring**

  against our attacker model for **mutual distrust**
- **Confidence** thanks to **simplicity**, **formalism**

  including **correctness proofs**
- **Efficiency** tackled with **hardware acceleration**

  for **compiled programs** and **low-level contexts**
- **Transparency** addressed with **flexibility**

  not rejecting **benign low-level contexts**,
  neither **statically** nor **dynamically**

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

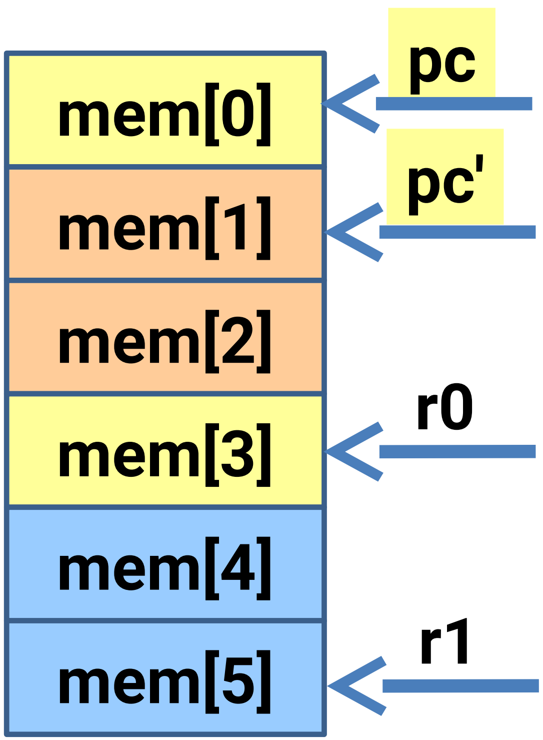
High-Level Abstractions

- 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**

Isolation Micro-Policy

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



decode(mem[0]) = store r0 r2

store: tpc = tm0 = tm3 … $\xrightarrow{✓}$ tpc tm3

decode(mem[1]) = nop

_: tpc' ≠ tm1 … $\xrightarrow{✗}$ failstop

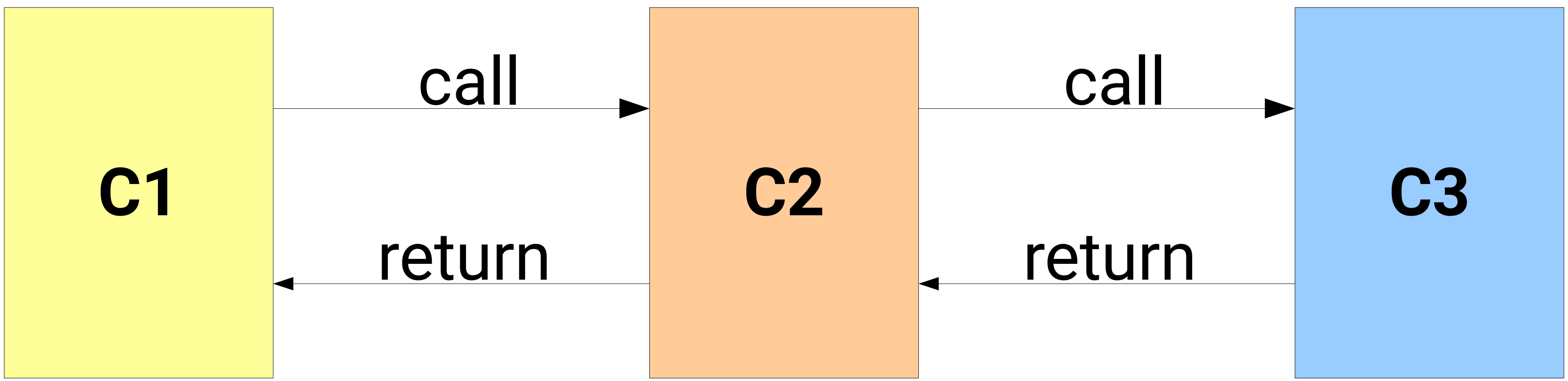
decode(mem[0]) = load r1 r2

load: tpc = tm0 ≠ tm5 … $\xrightarrow{✗}$ 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

Extra Hardware Support Study

✓ Update **input tags** as well as output tags

to **transfer** a linear capability

✓ Check and update **tags on some fixed registers**

for **dynamic type-checking** and **register cleaning**

✗ **Revoke** tags?

- would allow **revocable capabilities**
- very powerful, **no mechanism** at the moment

Towards More Realistic Languages

- Extend with **common features of OO languages**

  add dynamic allocation, inheritance, packages…

- Turn to **functional languages**
  - implicit dynamic allocation
  - closures as values

- Study **clean subsets** of **real-world languages**

  no **undefined behaviors**, **Obj.magic**, etc.

Dealing With Transparency

- Mustn't reject **benign contexts**

  e.g. low-level libraries, code from other compilers
- Need to enforce **exactly what is required**
  - no checks on **internal calls and returns**
  - **wrappers** when **capability not used as expected**
- Communication **driven by the language**
  - have **wrappers** allowing for communication
  - **no fancy types** in **interfaces**

Towards Measuring Efficiency

- We expect **very good efficiency**

  ASPLOS '15

  4 complex micro-policies, <10% overhead
- Impact on **arbitrary low-level contexts**
  - use **standard benchmarking suites**, e.g. SPEC2006
  - **transparency** required for these programs to run
  - **aim for ~0% overhead** when running **in isolation**
  - use **wrappers** to measure **communication overhead**
- Impact on programs **from our compiler**

  synthetic benchmarks until target = real language

Take-Away

- **Secure compilation** is **interesting**, **challenging**
- **Micro-policies** are **well-suited** for this problem

  with some **hardware extensions**
- **Strong**, **realistic attacker model**

  for **mutually distrustful components**
- **Good hopes** for **efficiency** and **transparency**
- Raises a lot of **research directions**

  … work **in progress**!

Thank you!

END

An Example: Private Fields

- Private fields become **secret-holding boxes**
  - high-level contexts **can't read** private fields
  - so **neither can** low-level contexts!

```
P ::= class E { Bool b }
      object o : E { true }

Q ::= class E { Bool b }
      object o : E { false }
```

  - from **high-level semantics**: $\forall A,\ A[P] \sim A[Q]$
  - hence, **applying FA**: $\forall a,\ a[P\downarrow] \sim a[Q\downarrow]$
- Will be the **easiest to enforce** abstraction in this talk