Efficient Formally Secure Compilers to a Tagged Architecture

Cătălin Hrițcu

Inria Paris

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

Hardware architectures

- **Today's processors are mindless bureaucrats**
  - "write past the end of this buffer"
  - "jump to this untrusted integer"
  - "return into the middle of this instruction"

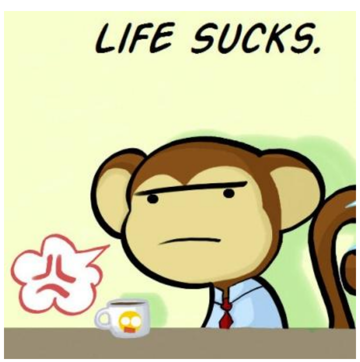
- **Software bears most of the burden for security**
- **Manufacturers have started looking for solutions**
  - 2015: Intel Memory Protection Extensions (MPX) and Intel Software Guard Extensions (SGX)
  - 2016: Oracle Silicon Secured Memory (SSM)

"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
- **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>

- *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)
- **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**

Efficient Secure Compilation to Micro-Policies

2nd part of this talk (more speculative)

1. **Secure semantics for low-level languages**
2. **Secure interoperability with lower-level code**

- Formally: **fully abstract compilation**
  – holy grail, enforcing abstractions all the way down
  – **currently this would be way too expensive**

- **Key enabling technology: micro-policies**
  – **hardware-accelerated tag-based monitoring**

1st part of this talk

MICRO-POLICIES

Micro-Policies team

- **Formal methods** & **architecture** & **systems**
- **Current team**:
  - *Inria*: **Cătălin Hrițcu**, **Yannis Juglaret**
  - *UPenn*: **Arthur Azevedo de Amorim**, **André DeHon**, **Benjamin Pierce**, **Nick Roessler**, **Antal Spector-Zabusky**
  - *Portland State*: **Andrew Tolmach**
  - *MIT:* **Howard E. Shrobe, Stelios Sidiroglou-Douskos**
  - *Industry*: **Draper Labs, Bluespec Inc**
- **Spinoff of past project: DARPA CRASH/SAFE (2011-2014)**

Micro-policies

- add **large tag** to each machine word **unbounded metadata**

word	tag

(not addressable)

→

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

(protected meta-data space)

- 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

*Conceptual model, our hardware implements this efficiently

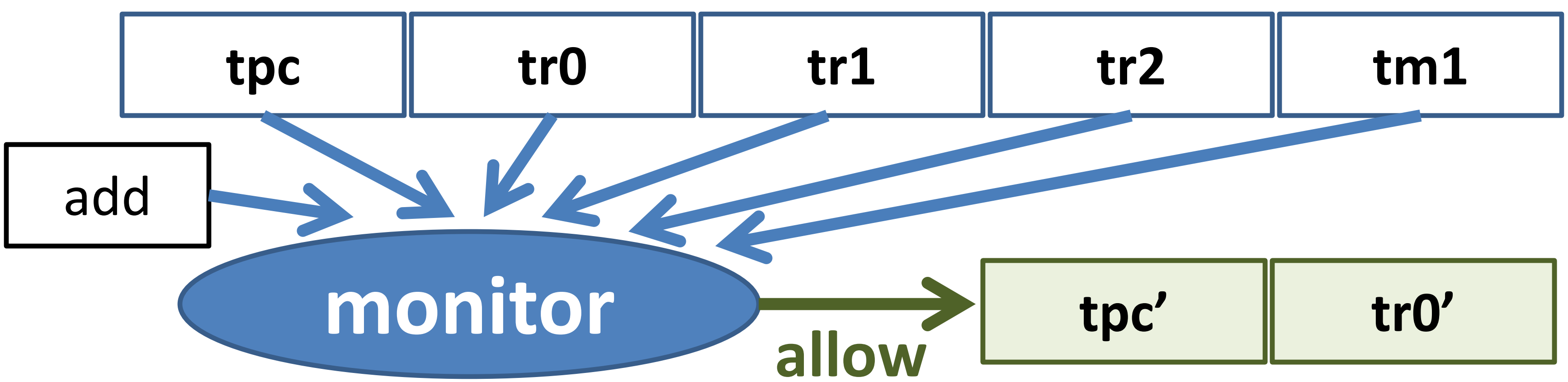
Tag-based instruction-level monitoring

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

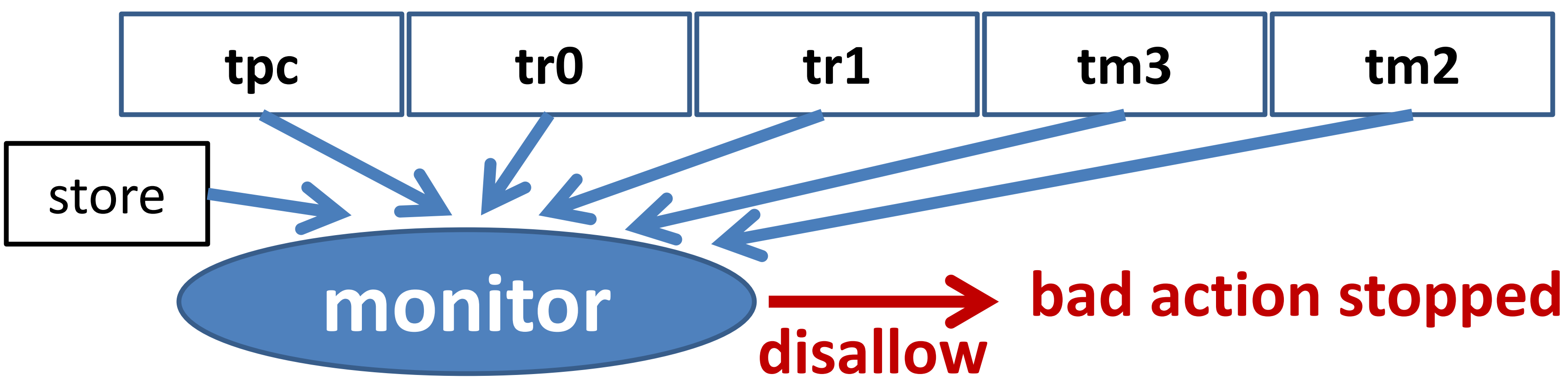


Tag-based instruction-level monitoring

pc	tpc
r0	tr0
r1	tr1
r2	tr2

mem[0]	tm0	
mem[1]	tm1	
mem[2]	tm2	pc
mem[3]	tm3	r0

decode(mem[1]) = store r0 r1



Efficiently executing micro-policies

op	**tpc**	**t1**	**t2**	**t3**	**tci**

lookup

zero overhead hits!

found

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

Efficiently executing micro-policies

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

lookup

misses trap to software

produced “rule” 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

Micro-policies are cool!

- **low level + fine grained**: unbounded per-word metadata, checked & propagated on each instruction
- **efficient**: accelerated using hardware caching
- **expressive**: can enforce large number of policies
- **flexible**: tags and monitor defined by software
- **secure**: simple enough to formally verify security
- **real**: FPGA implementation on top of RISC-V CPU

DRAPER bluespec

Expressiveness

- information flow control (IFC) [Oakland'13, POPL'14]
- monitor self-protection
- compartmentalization
- dynamic sealing
- heap memory safety
- code-data separation
- control-flow integrity (CFI)
- taint tracking
- ...

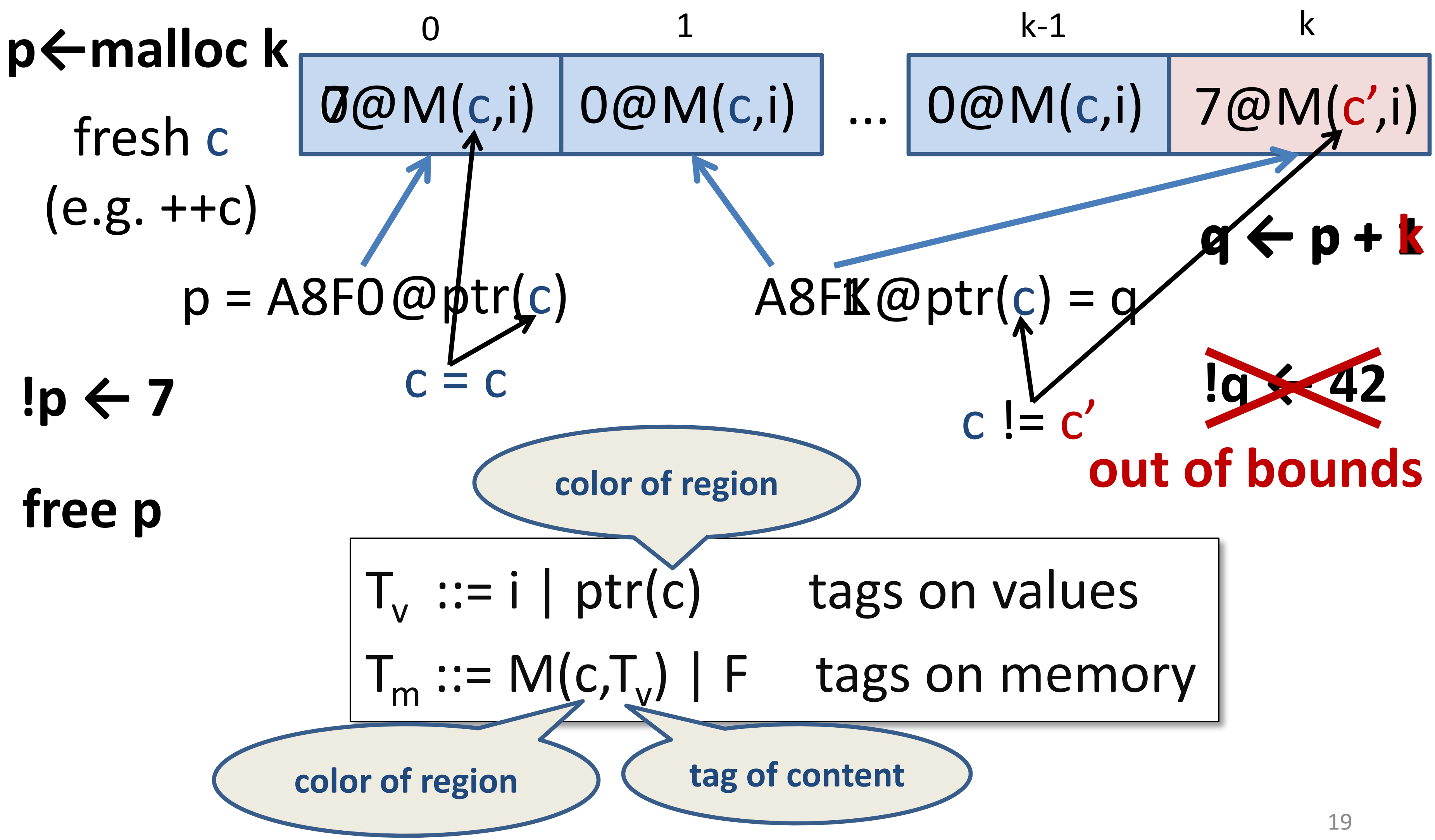
Verified
(in Coq)
[Oakland'15]

Evaluated
(<10% runtime overhead)
[ASPLOS'15]

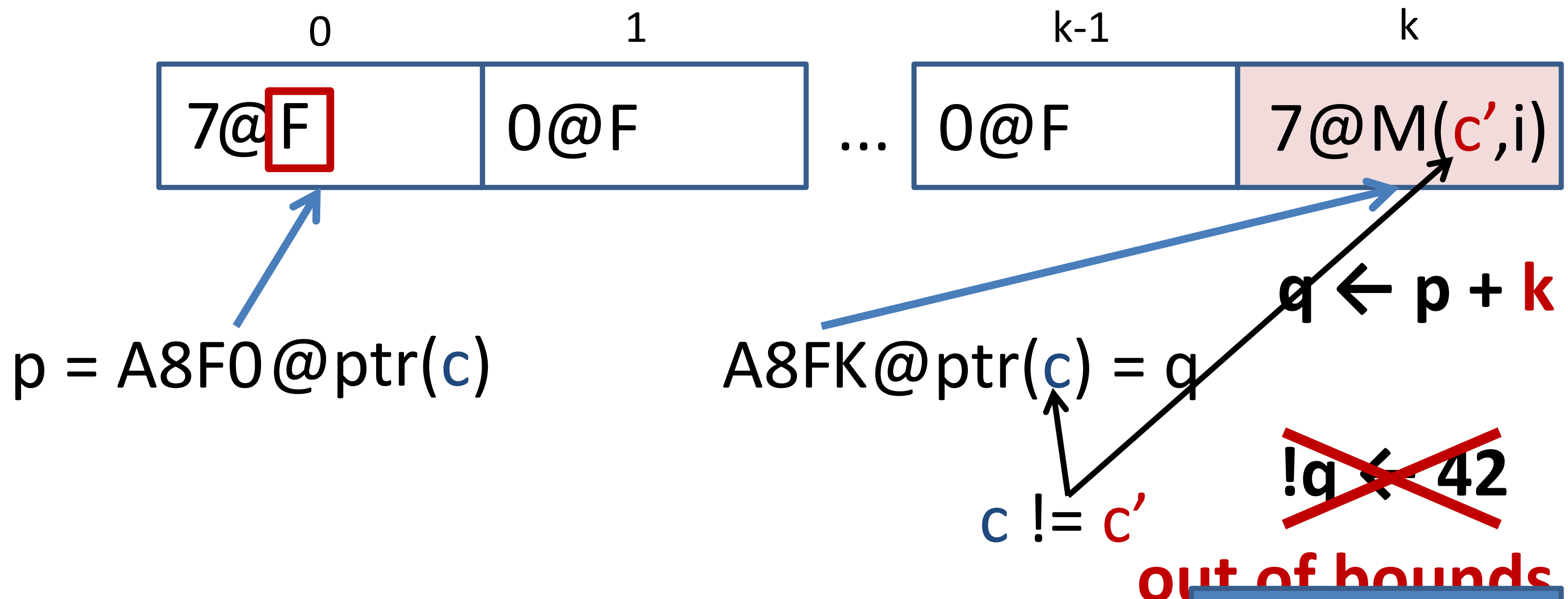
Flexibility (by example)

- **Heap memory safety** micro-policy prevents
  - **spatial violations**: reading/writing out of bounds
  - **temporal violations**: use after free, invalid free
  - for **heap-allocated data**
- Pointers become **unforgeable capabilities** 
  - can only obtain a valid pointer to a heap region
    - by allocating that region or
    - by copying/offsetting an existing pointer to that region

Memory safety micro-policy



Memory safety micro-policy



free p

~~**x ← !p**~~

use after free

Intel MPX cannot detect this

T_v ::= i | ptr(c)	tags on values	
T_m ::= M(c,T_v) | F	tags on memory	

Oracle Silicon Secured Memory (2016) similar, but with only 16 colors

Micro-Policies
+ can adapt to new threats
+ expressive enough for efficient secure compilation

SECURE COMPILATION

Joint work with Yannis Juglaret

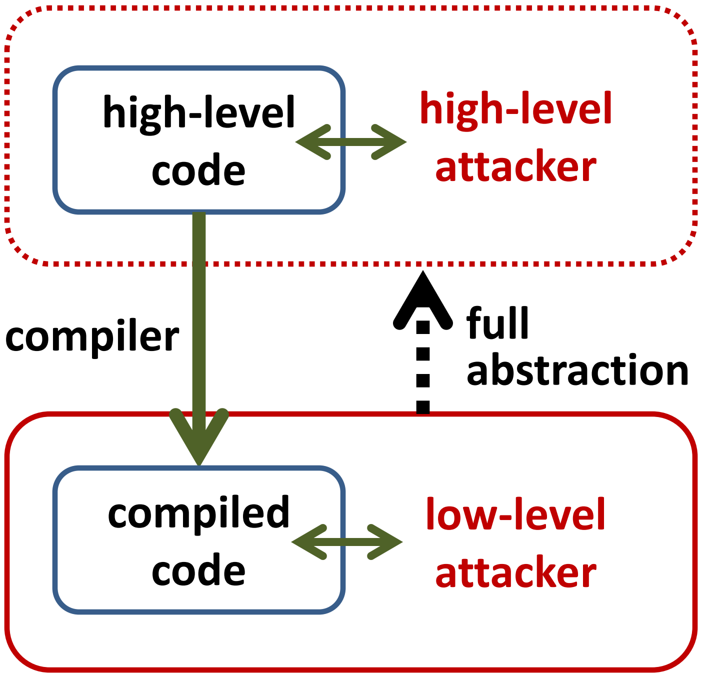
Secure compilation

- **Goal:** to build the **first efficient secure compilers** for **realistic programming languages**

1. **Secure semantics for low-level languages**
   – C with memory safety and compartmentalization
2. **Secure interoperability with lower-level code**
   – ASM, C, ML, and F* (verification system for ML)
   – problems are quite different at different levels

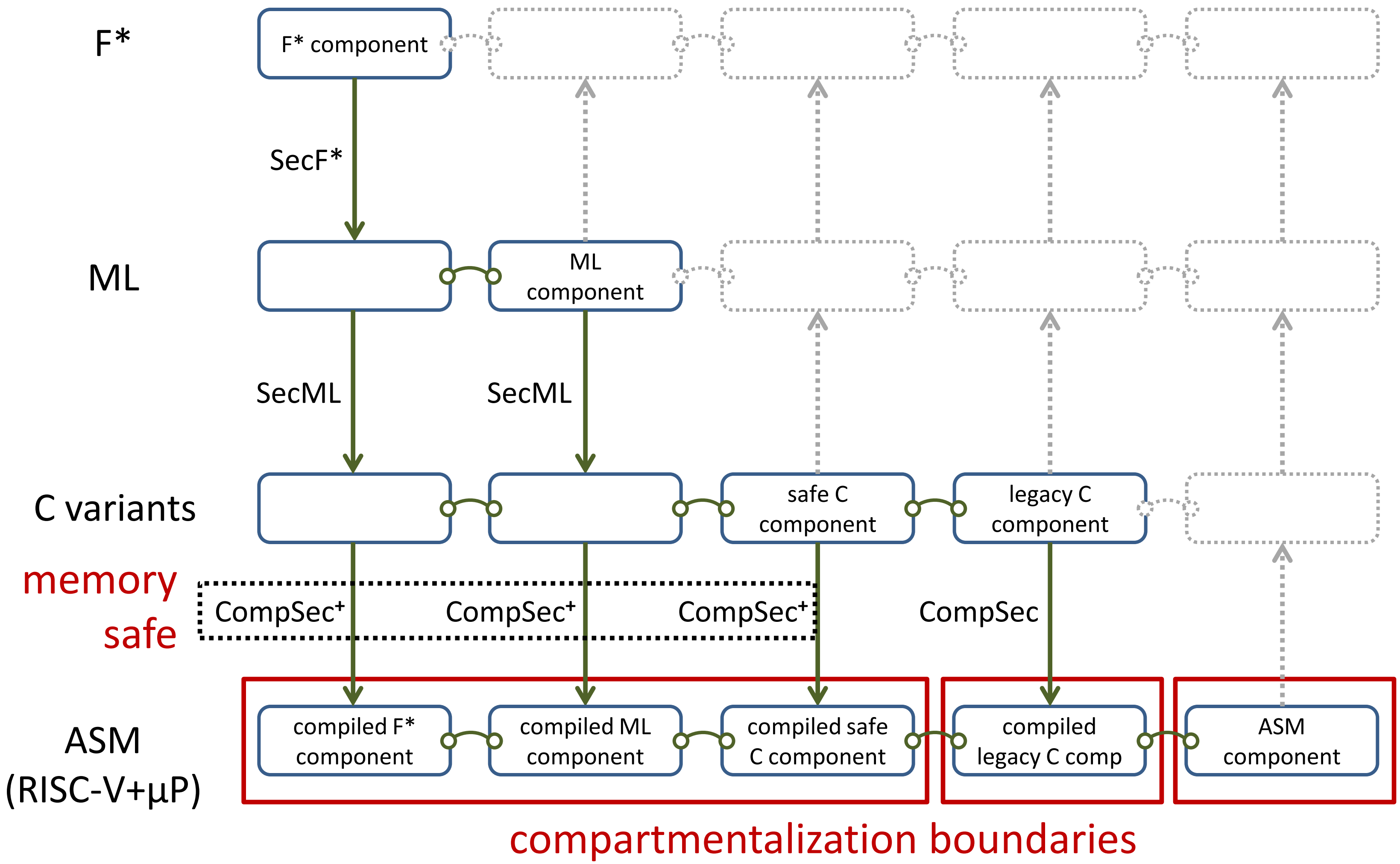
- Formally: **fully abstract compilation**
  – enforcing abstractions all the way down

Fully abstract compilation, intuition



Benefits: can reason about security in the source language; forget about compiler, linker, loader, runtime system, and (to some extent) low-level libraries

Very long term vision



Low-level compartmentalization

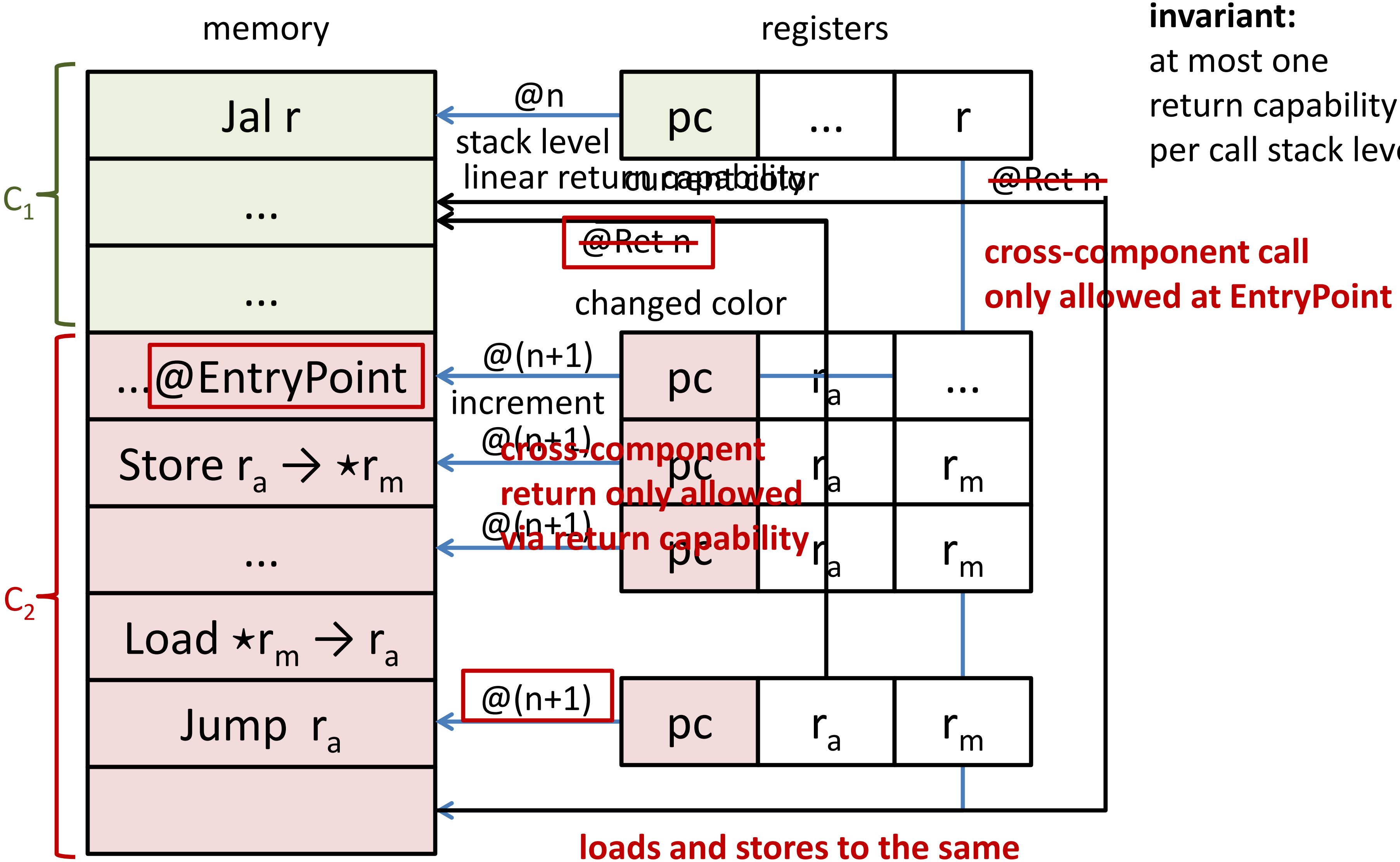
- Break up software into **mutually distrustful components** running with **minimal privileges** & interacting only via **well-defined interfaces**
- **Limit the damage** of control hijacking attacks to just the C or ASM components where they occur
- Not a new idea, already deployed in practice:
  - process-level privilege separation
  - software-fault isolation
- Micro-policies can give us **better interaction model**
- We also aim to **show security formally**

Compartmentalized C

- Want to **add components with typed interfaces to C**
- Compiler (e.g. CompCert), linker, loader propagate interface information to low-level memory tags
  - each component's memory tagged with unique color
  - procedure entry points tagged with procedure's type
- Micro-policy enforcing:
  - **component isolation**
  - **procedure call discipline** (entry points)
  - **stack discipline for returns** (linear return capabilities)
  - **type safety** on cross-component interaction

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

Compartmentalization micro-policy

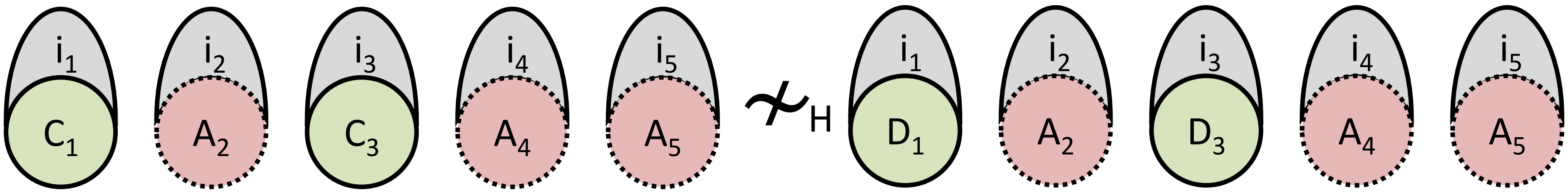


invariant:
at most one
return capability
per call stack level

loads and stores to the same component always allowed

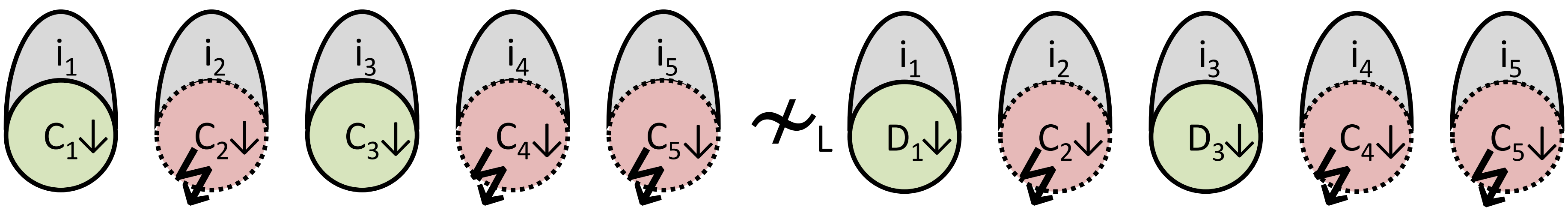
Secure compartmentalization property

∀compromise scenarios.



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

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



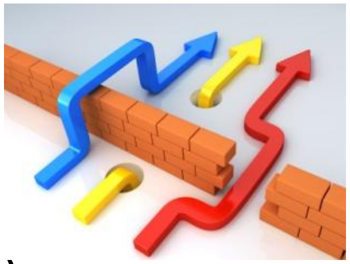
follows from “structured full abstraction for unsafe languages” + “separate compilation”

[Beyond full abstraction, Juglaret, Hritcu, et al, draft’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, ...
- **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**

Secure compilation



- Solving conceptual challenges
  - **Secure micro-policy composition**
  - **Higher-level micro-policies** (for C and ML)
  - **Formalizing security properties** (i.e. attacker models)
- Building the first **efficient secure compilers** for **realistic programming languages**
  - C (CompCert): memory safety & compartmentalization
  - ML and F*: protecting higher-level abstractions
- **Measuring & lowering the cost of secure compilation**
  - better hardware, hybrid enforcement (static + dynamic), weaker properties (robust compilation)

- **Showing formally that these compilers are indeed secure**

Conclusion

- **There is a pressing practical need for ...**
  - **more secure languages** providing strong abstractions
  - **more secure compiler chains** protecting these abstractions
  - **more secure hardware** making the cost of all this acceptable
  - **clear attacker models** & **strong formal security guarantees**
- Building the first **efficient secure compilers** for **realistic programming languages** (C, ML, F*)

- **Targeting micro-policies** = new mechanism for hardware-accelerated tag-based monitors

Thank you!