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

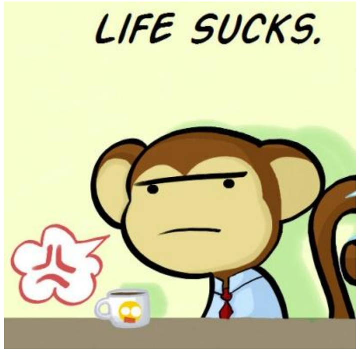
- **Today's processors are mindless bureaucrats**
  - "write past the end of this buffer"
  - "jump to this untrusted integer"
  - "return into the middle of this instruction"
- **Software bears most of the burden for security**
- **Manufacturers have started looking for solutions**
  - 2015: Intel Memory 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

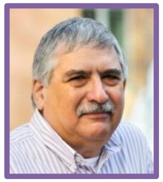
- **Formal methods** & **architecture** & **systems**
- **Current team**:
  - *Inria*: **Cătălin Hrițcu**, **Yannis Juglaret**
  - *UPenn*: **Arthur Azevedo de Amorim**, **André DeHon**, **Benjamin Pierce**, **Nick Roessler**, **Antal Spector-Zabusky**
  - *Portland State*: **Andrew Tolmach**
  - *MIT:*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









DRAPER bluespec

Micro-policies

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

unbounded metadata

word	tag	→	tag[0]	tag[1]	tag[2]

(not addressable) (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

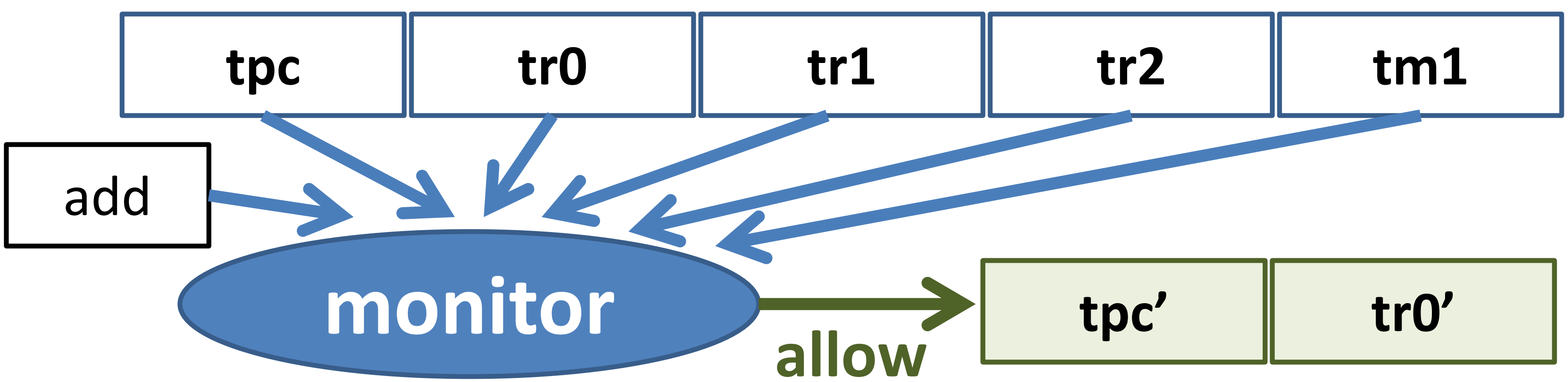
Tag-based instruction-level monitoring

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

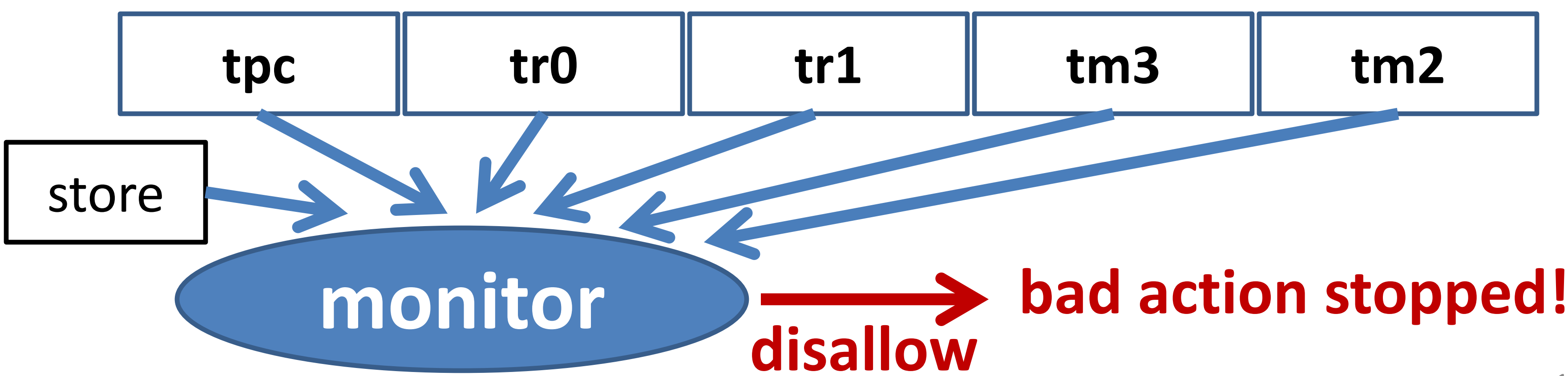


Tag-based instruction-level monitoring

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



Micro-policies are cool!

- **low level + fine grained**: unbounded per-word metadata, checked & propagated on each instruction
- **expressive**: can enforce large number of policies
- **flexible**: tags and monitor defined by software
- **efficient**: accelerated using hardware caching
- **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

p←malloc k

fresh c

(e.g. ++c)

0	1		k-1	k
7@M(c,i)	0@M(c,i)	...	0@M(c,i)	7@M(c',i)

p = A8F0@ptr(c)

c = c

A8FK@ptr(c) = q

q ← p + k

c != c'

!p ← 7

~~**!q ← 42**~~

out of bounds

free p

color of region

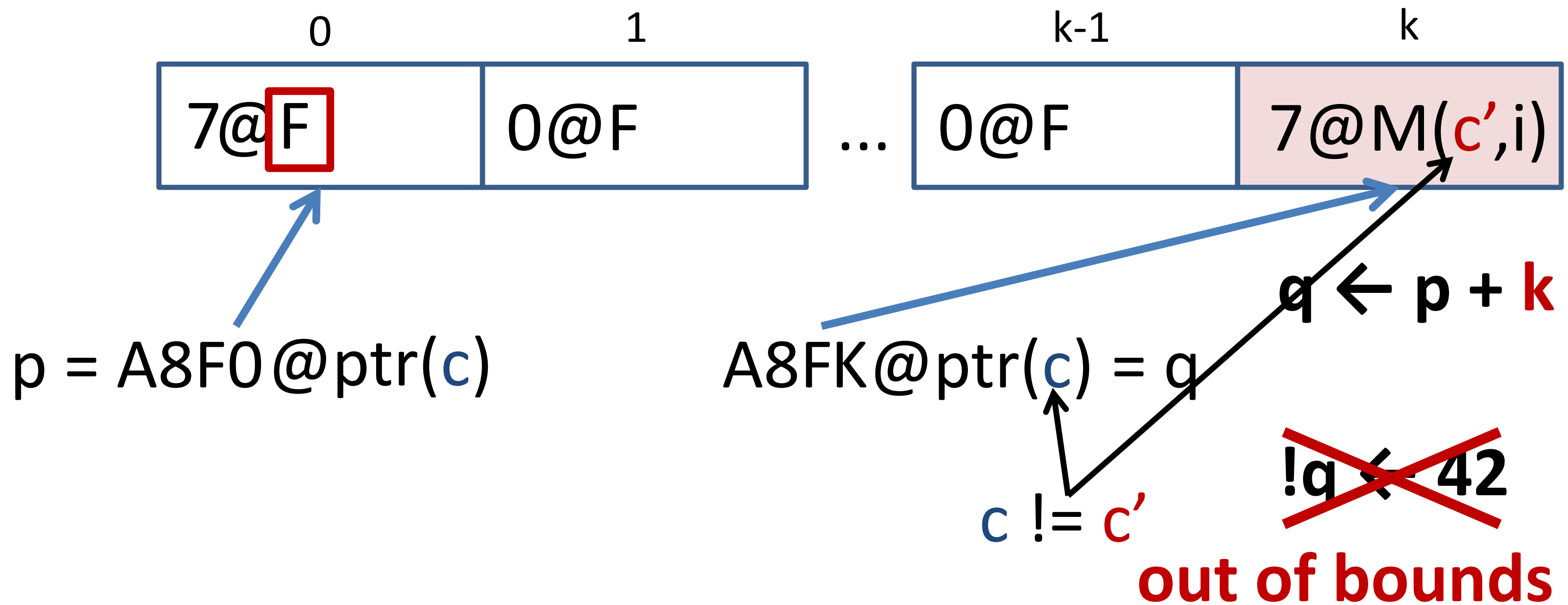
T_v ::= i | ptr(c) tags on values

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

color of region

tag of content

Memory safety micro-policy



free p

~~**x ← !p**~~

use after free

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

Intel MPX cannot detect this

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

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

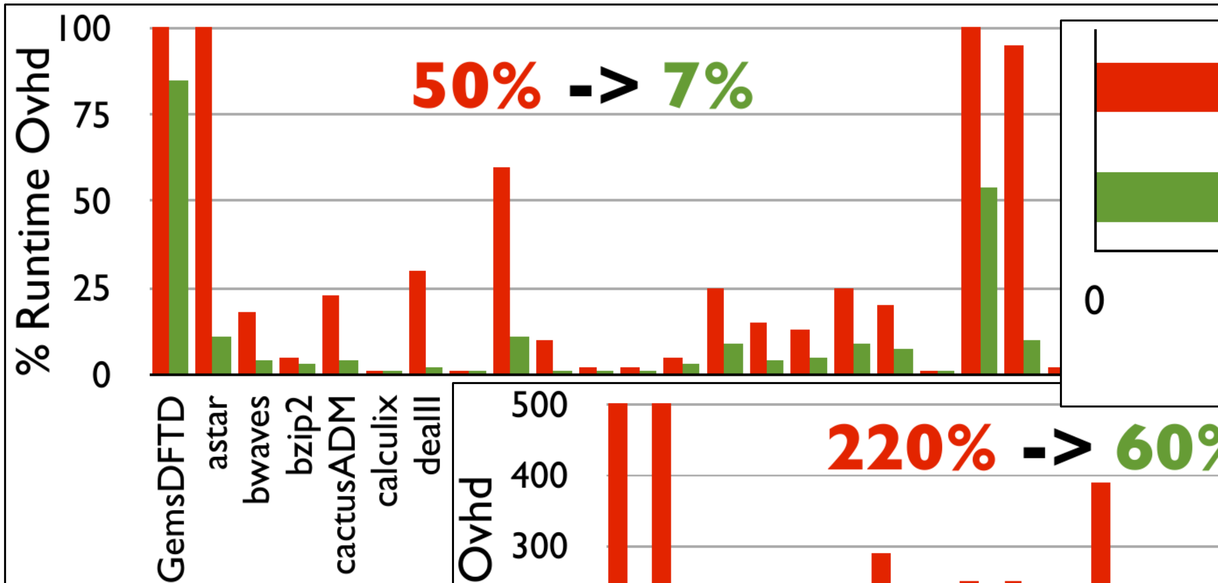
produced “rule” cached

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**

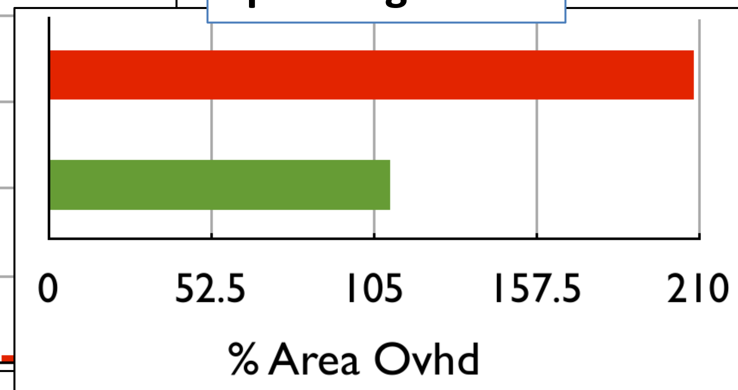
hardware cache

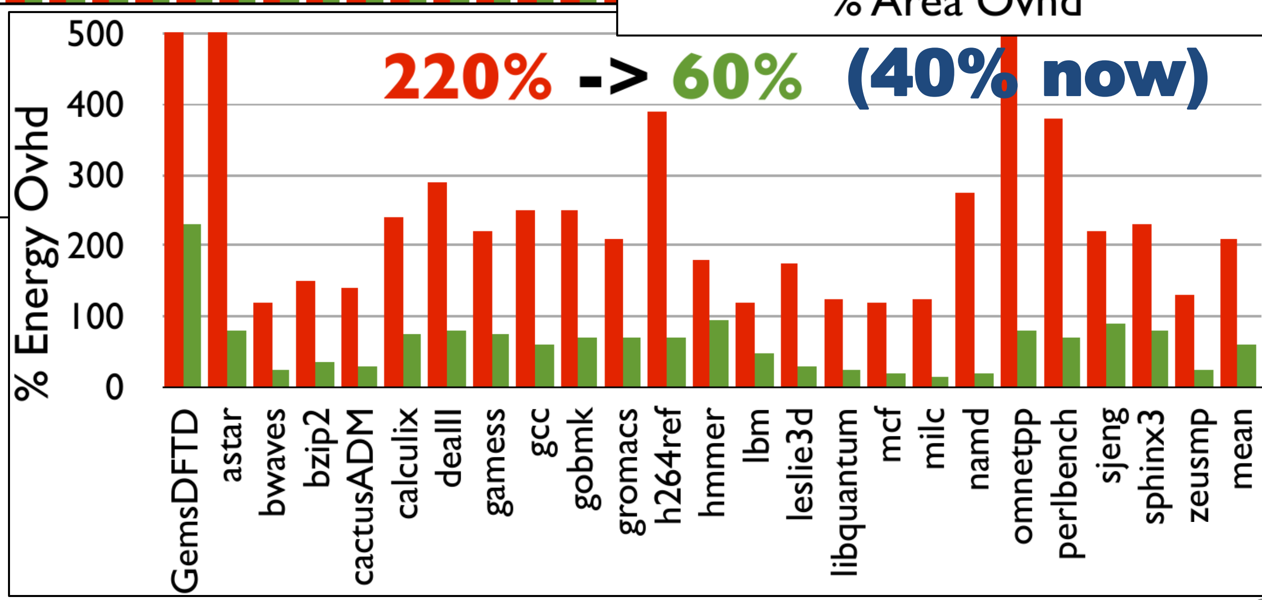
Experimental evaluation (simulations)

heap memory safety + code-data separation + taint tracking + control-flow integrity
simple RISC processor: single-core 5-stage in-order Alpha (pre RISC-V transition)



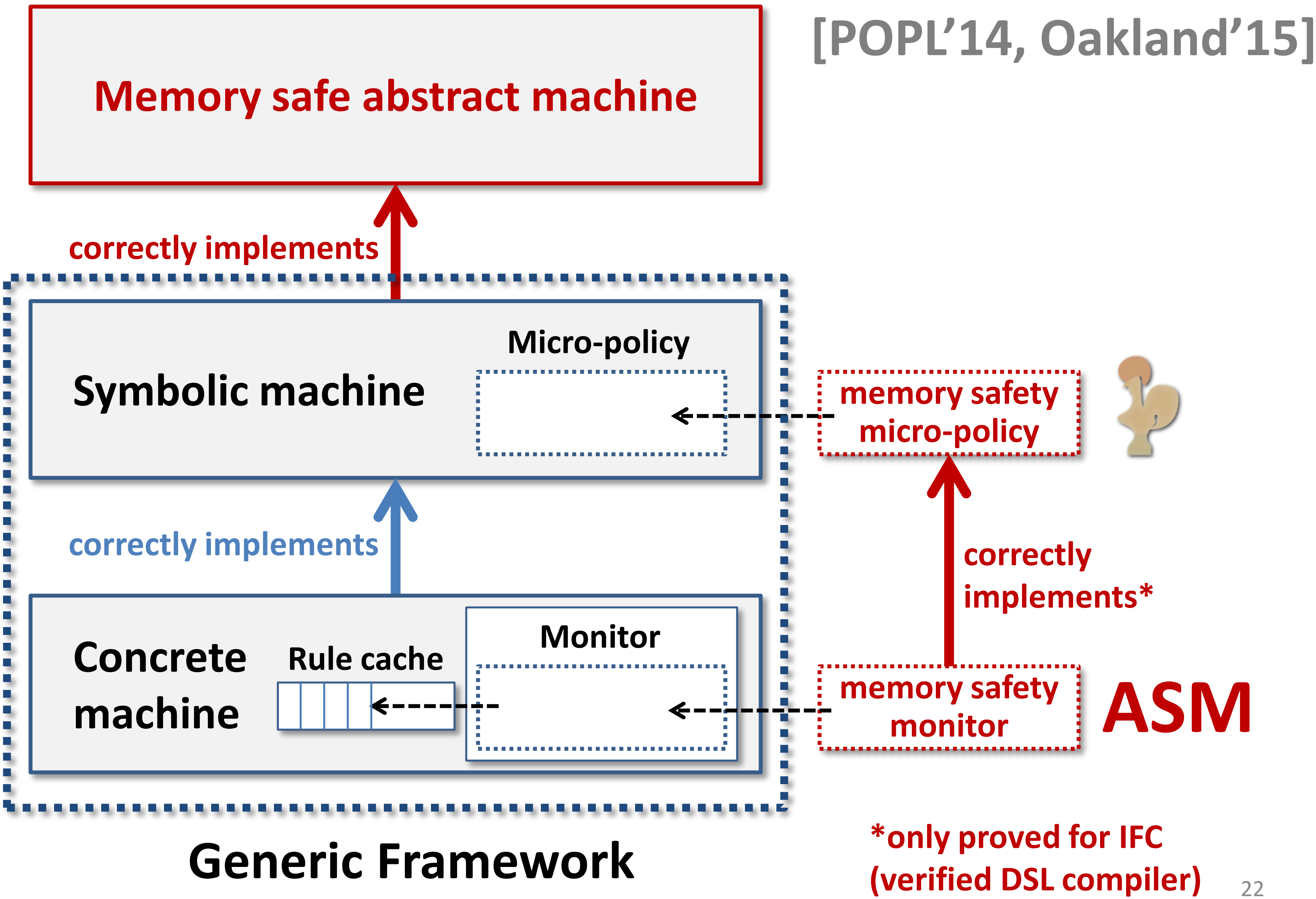
spending silicon



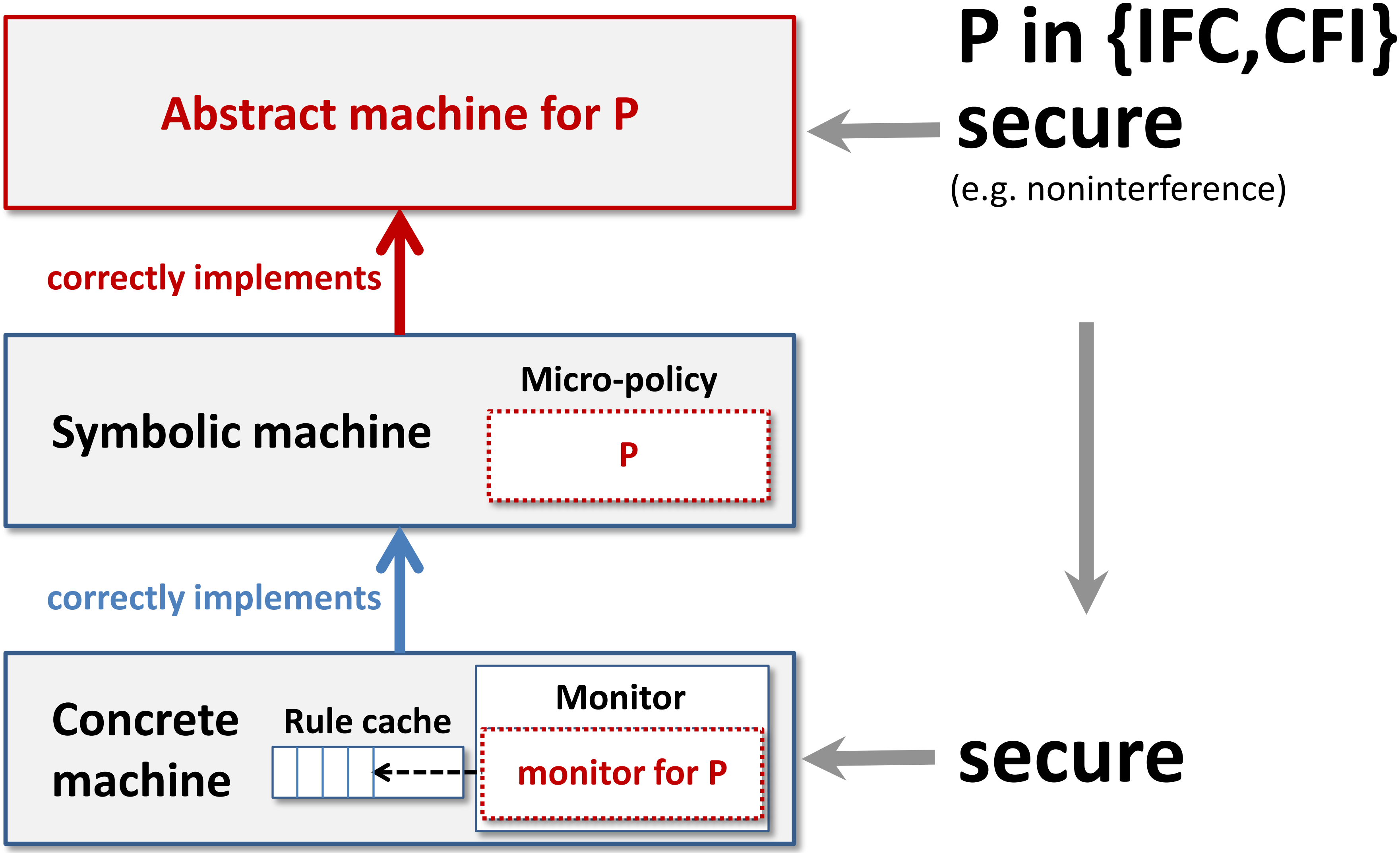


More details
[ASPLOS'15]

Formal verification in Coq



Is this secure?



* Working on **extrinsic definition of memory safety**
[Alpha is for address, Azevedo de Amorim et al, draft 2015]

SECURE COMPILATION

Joint work with Yannis Juglaret

INVENTORS FOR THE DIGITAL WORLD

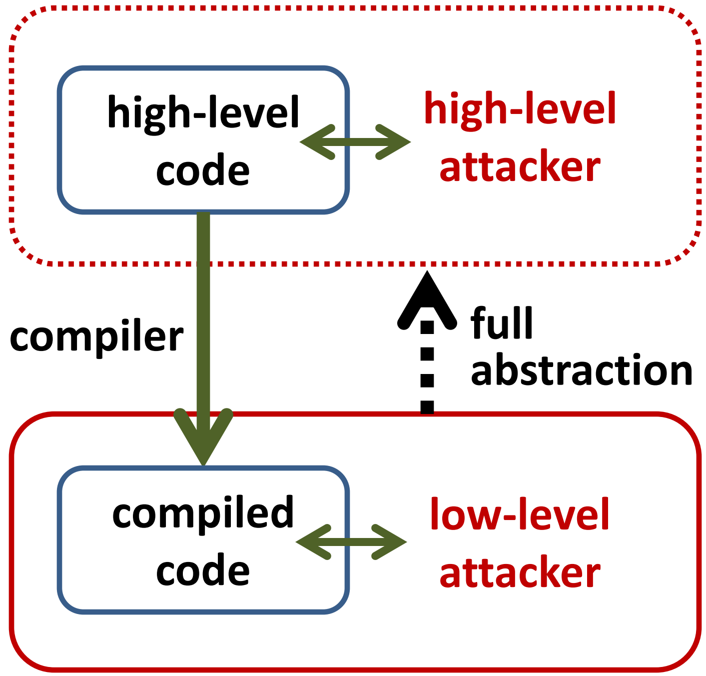
Secure compilation

- **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

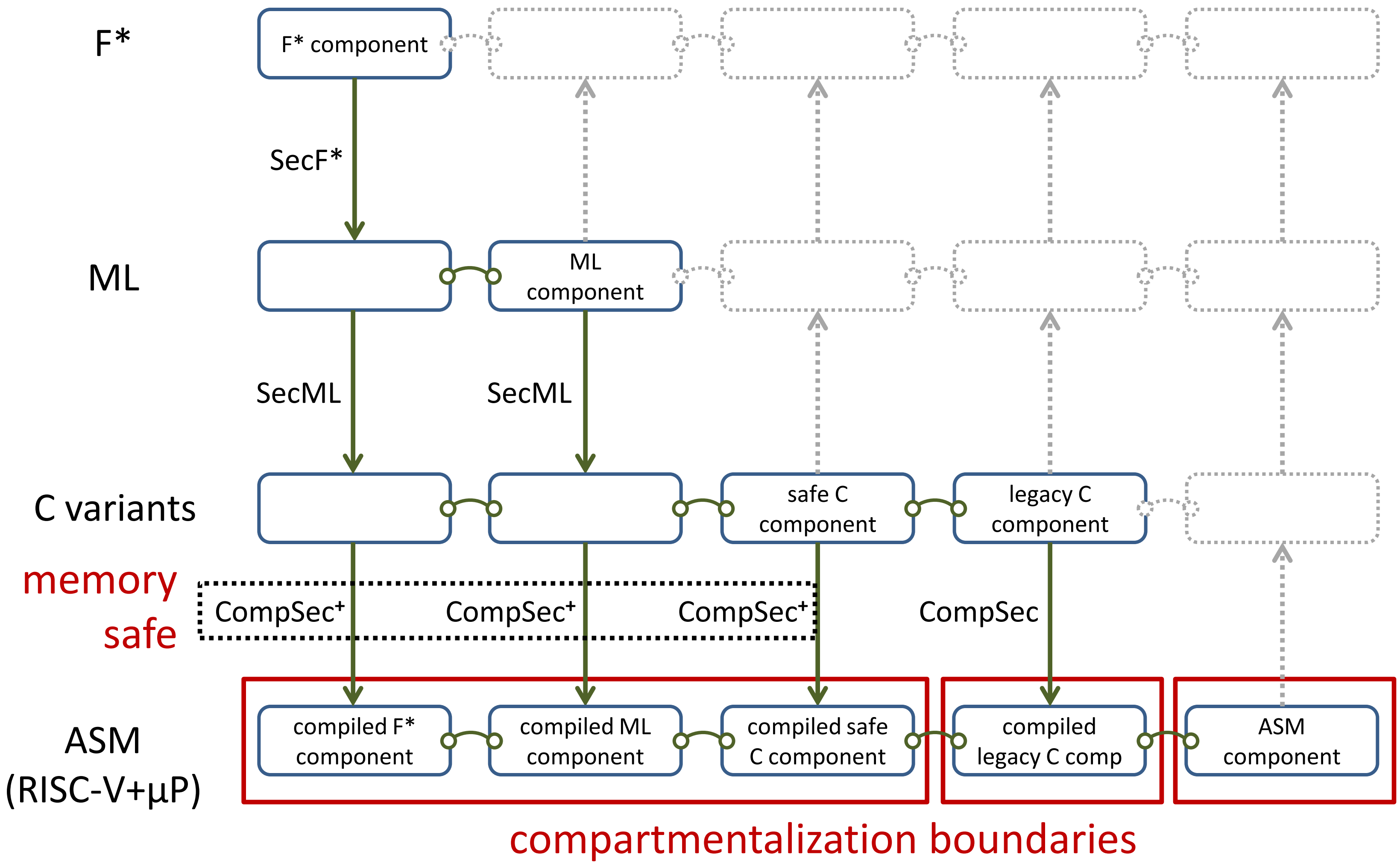
- Formally: **fully abstract compilation**
  - enforcing abstractions all the way down

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

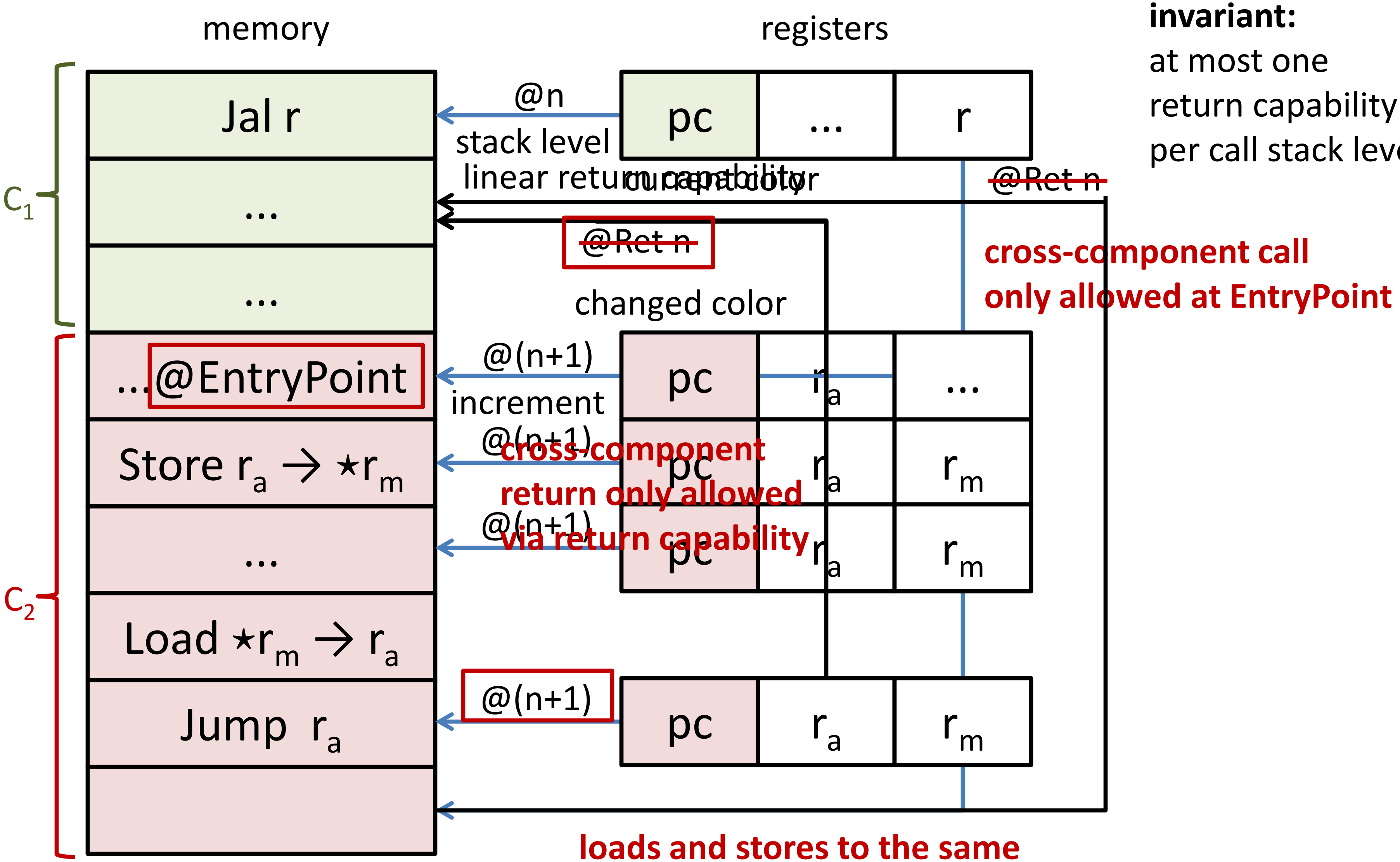
- Break up software into **mutually distrustful components** running with **minimal privileges** & interacting only via **well-defined interfaces**
- **Limit the damage** of control hijacking attacks to just the C or ASM components where they occur
- Not a new idea, already deployed in practice:
  - 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

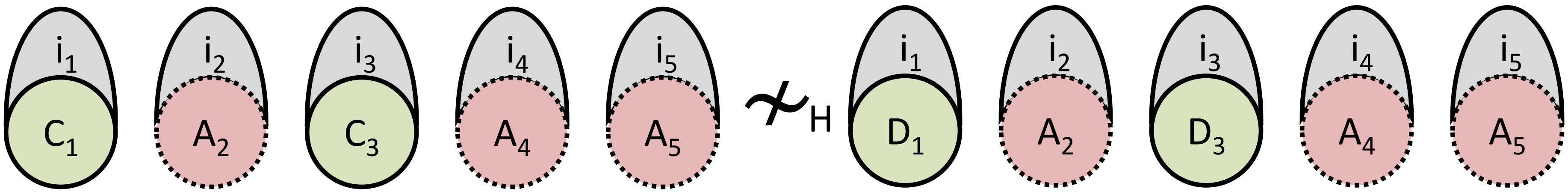


invariant:
at most one
return capability
per call stack level

loads and stores to the same component always allowed

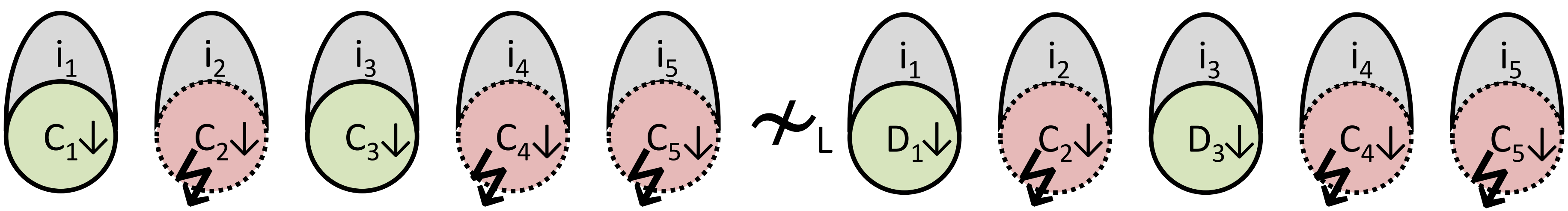
Secure compartmentalization property

∀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

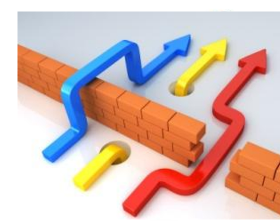


follows from "structured full abstraction
for unsafe languages" + "separate compilation"

[Beyond full abstraction, Juglaret, Hritcu, et al, draft'16]

Protecting higher-level abstractions

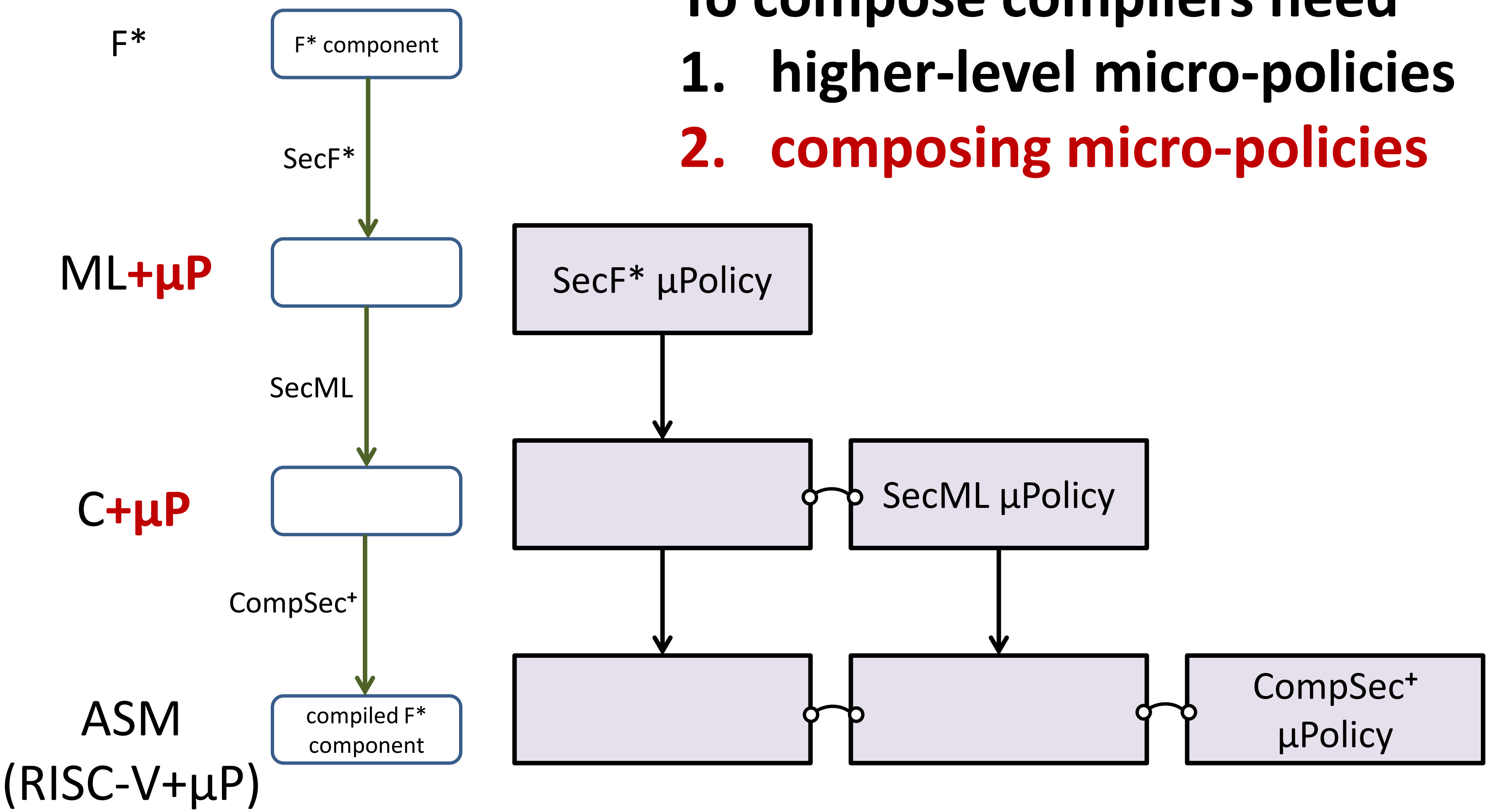
- **ML abstractions we want to enforce with micro-policies**
  - types, value immutability, opaqueness of closures, parametricity (dynamic sealing), GC vs malloc/free, ...
- **F*: enforcing full specifications using micro-policies**
  - some can be turned into **contracts,** checked dynamically
  - fully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



Composing compilers and higher-level micro-policies

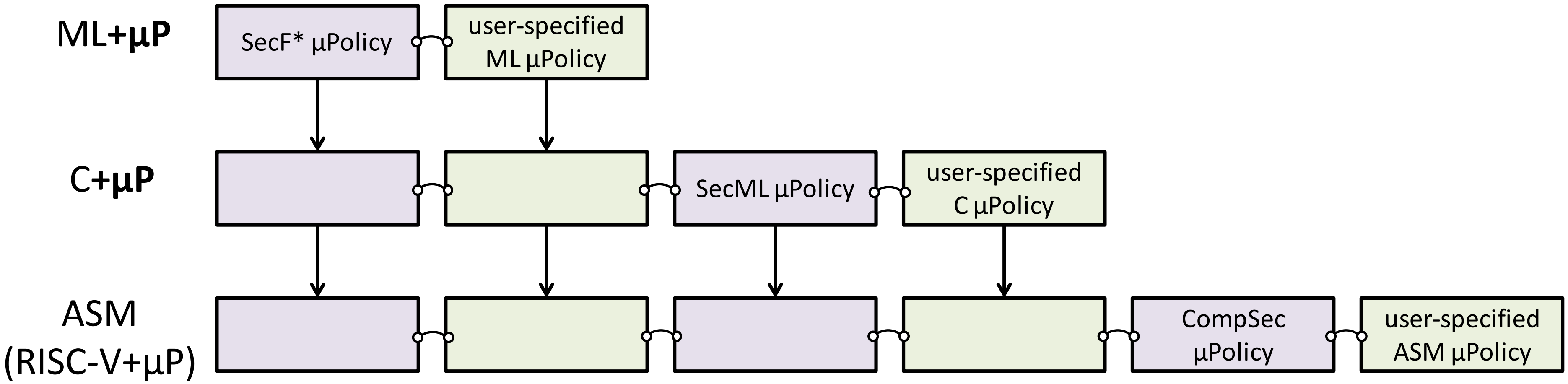
To compose compilers need

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



User-specified higher-level policies

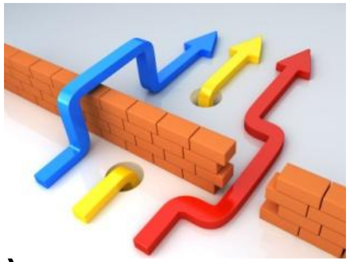
- By composing more micro-policies we can allow **user-specified micro-policies for ML and C**
- Good news: **micro-policy composition is easy** since tags can be tuples
- But how do we ensure programmers won't break security?
- Bad news: **secure micro-policy composition is hard!**



Secure micro-policy composition

- **securely composing reference monitors is easy**
  - ... as long as they **can only stop execution**
- micro-policies have **richer interaction** model:
  - **monitor services:** malloc, free, classify, declassify, ...
  - **recoverable errors** are similar
- **composing micro-policies can break them**
  - e.g. composing anything with IFC can leak
  - memory safety + compartmentalization

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

- Showing that these compilers are indeed secure
  - **Better verification and testing techniques**

- Redesigned ML verification system [POPL'16]
  1. **functional programming language with effects**
     (like OCaml, F#, Standard ML, Haskell)
  2. **deductive verification system based on SMT solvers**
     (like FramaC, Why3, Dafny, Boogie, VCC, ESC/Java2)
  3. **interactive proof assistant based on dependent types**
     (like Coq, Lean, Agda)
- Working on **language design, formal foundations, logical aspects, proof assistant, self-certification**
- Main practical application:
  - **verified reference implementation of upcoming TLS 1.3**

Dependable property-based testing

- QuickCheck effective at finding bugs
- **reducing the testing effort**
  - language for property-based generators
- **obtaining stronger confidence**
  - polarized mutation testing
- **providing stronger formal foundations**
  - verified testing, generator synthesis(?)
- **integrating testing in proof assistants**
  - reducing the cost of interactive verification

Inria INVENTORS FOR THE DIGITAL WORLD

Penn UNIVERSITY of PENNSYLVANIA

CHALMERS UNIVERSITY OF TECHNOLOGY

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!

BACKUP SLIDES

About my research

Formal methods, broadly

- programming languages
- type systems
- deductive verification
- proof assistants
- formal metatheory
- certified tools
- property-based testing

Solving security problems

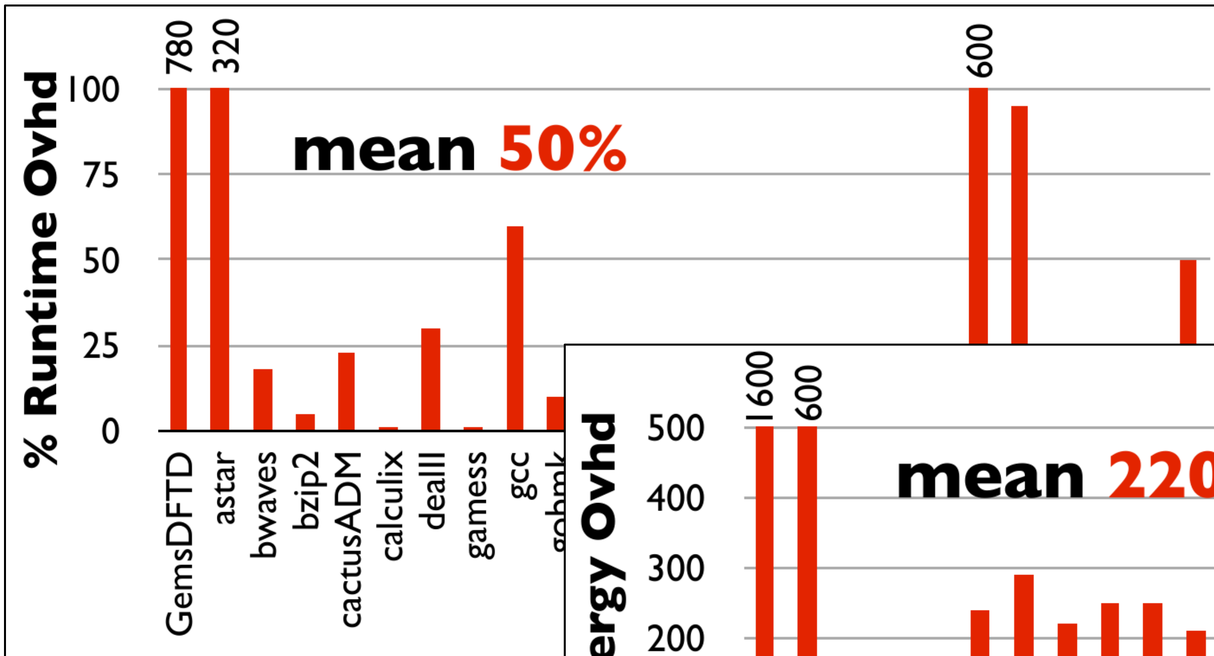
- formal attacker models
- designing and building more secure systems
- stopping low-level attacks
- dynamic monitoring
- integrity, information flow
- security protocols

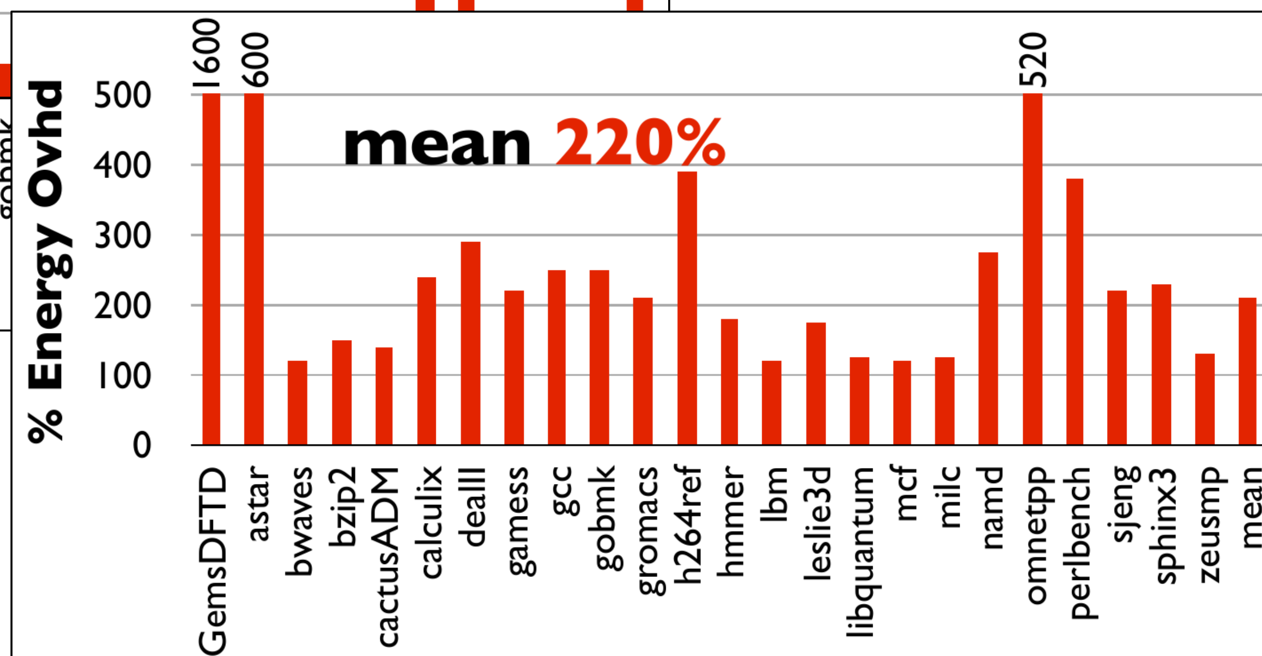
Useful tools, choose one that's well-suited for the problem

Build and release open source software based on research

Simulations for **naive** implementation

memory safety + code-data separation + taint tracking + control-flow integrity
simple RISC processor: single-core 5-stage in-order Alpha





simulation numbers;
but this naive version also implemented on FPGA
(part of SAFE machine)
[FPGA ’13, TRETS ‘15]

Targeted [micro-]architectural optimizations

[ASPLOS'15]

- grouping opcodes and ignoring unused tags
  - **increases effective rule cache capacity**
- transferring only unique tags to/from DRAM
  - **reduces runtime and energy overhead**
- using much shorter tags for on-chip data caches
  - **reduces runtime, energy, and area overhead**
- caching composite policies separately
  - **makes rule cache misses much cheaper**

Expressiveness

- Micro-policy mechanism can efficiently enforce:
  - heap memory safety
  - code-data separation
  - control-flow integrity
  - compartment isolation
  - taint tracking
  - information flow control
  - monitor self-protection
  - dynamic sealing

... and a lot more!

Memory safety micro-policy

1. Sets of tags

T_v ::= i | ptr(c)

T_m ::= M(c,T_v) | F

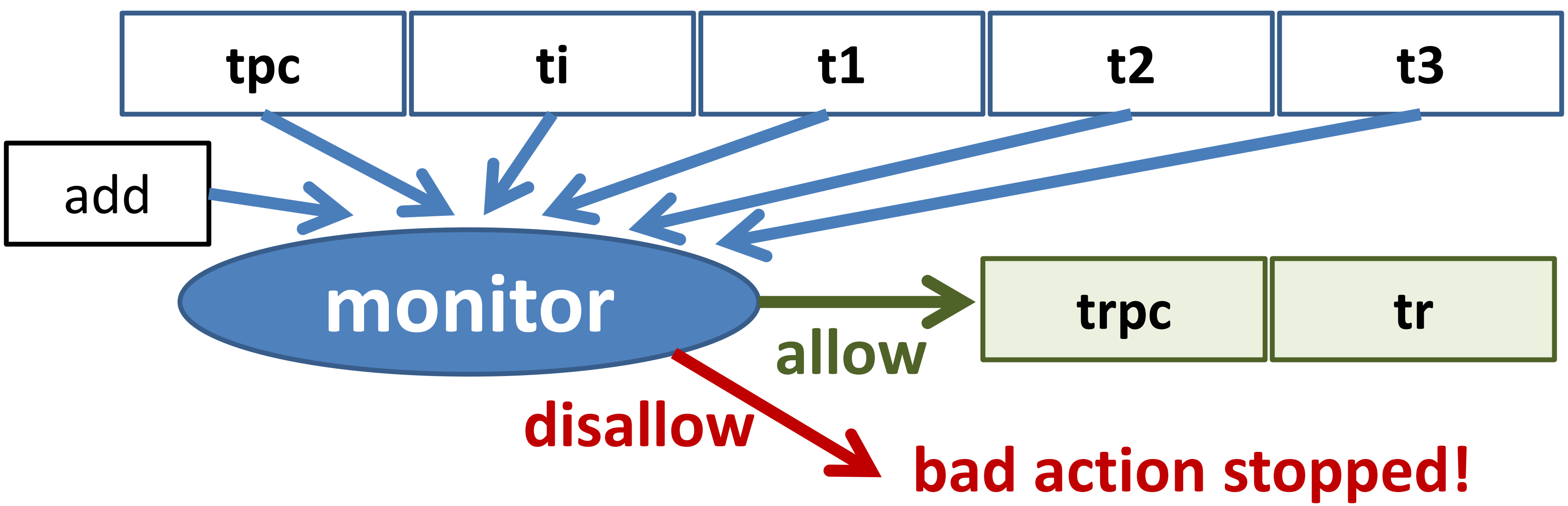
T_{pc} ::= T_v

2. Transfer function

Record **IVec** := { op:opcode ; t_{pc}:T_{pc} ; t_i:T_m ; ts: ... }

Record **OVec** (op:opcode) := { t_{rpc} : T_{pc} ; t_r : ... }

transfer : (iv:IVec) -> option (OVec (op iv))



Memory safety micro-policy

1. Sets of tags

$\mathbf{T_v}$ ::= i | ptr(c)

$\mathbf{T_m}$ ::= M(c,T_v) | F

$\mathbf{T_{pc}}$ ::= T_v

2. Transfer function

Record **IVec** := { op:opcode ; t_{pc}:T_{pc} ; t_i:T_m ; ts: ... }

Record **OVec** (op:opcode) := { t_{rpc} : T_{pc} ; t_r : ... }

transfer : (iv:IVec) -> option (OVec (op iv))

Definition **transfer** iv :=

match iv with

| {op=Load; t_{pc}=ptr(c_{pc}); t_i=M(c_{pc},i); ts=[ptr(**c**); M(**c**,$\mathbf{T_v}$)]}

=> {t_{rpc}=ptr(c_{pc}); t_r=$\mathbf{T_v}$}

| {op=Store; t_{pc}=ptr(c_{pc}); t_i=M(c_{pc},i); ts=[ptr(**c**); $\mathbf{T_v}$; M(**c**,T_v')]}

=> {t_{rpc}=ptr(c_{pc}); t_r=M(**c**,$\mathbf{T_v}$)}

...

Memory safety micro-policy

1. Sets of tags

T_v ::= i | ptr(c)

T_m ::= M(c,T_v) | F

T_{pc} ::= T_v

2. Transfer function

Record **IVec** := { op:opcode ; t_{pc}:T_{pc} ; t_i:T_m ; ts: ... }

Record **OVec** (op:opcode) := { t_{rpc} : T_{pc} ; t_r : ... }

transfer : (iv:IVec) -> option (OVec (op iv))

3. Monitor services

Record **service** := { addr : word; sem : state -> option state; ... }

Definition **mem_safety_services** : list service :=

[**malloc; free; base; size; eq**].

*This takes us beyond “noninterferent” reference monitors (more soon)

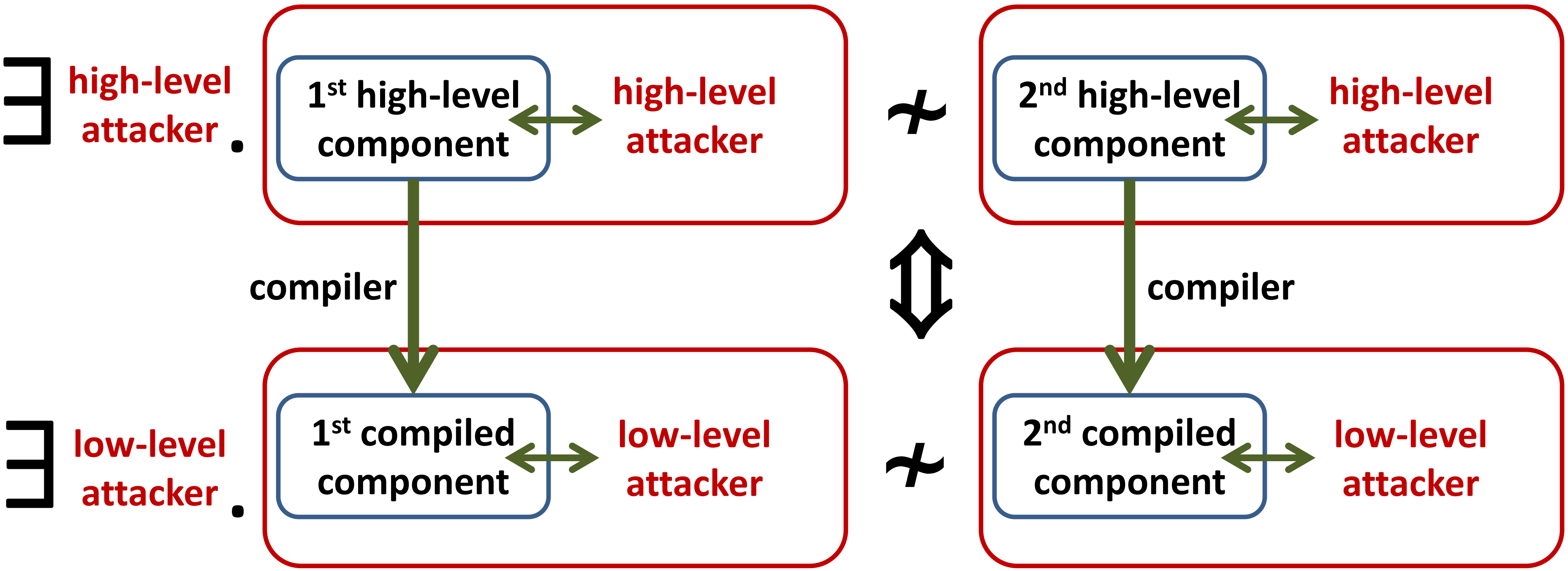
Open problems

- **Interaction with PL, compiler, loader, linker, OS**
- **Secure micro-policy composition**
- **Verified optimizing compiler for micro-policies**
- **Reduced/more adaptive energy usage**
- **More realistic processor**
  (our-of-order execution, even multi-core)
- **Cache side channels**

Take away

- **Micro-policies**, novel security mechanism
  - **low level, fine grained, expressive, flexible, efficient, formally secure, real**
- **cool research direction with many interesting open problems for us and others to solve**
- other projects:
  - **F***: formal verification of ML programs
  - **QuickChick**: property-based testing for Coq
- **Thank you!**

Fully abstract compilation, definition



Memory safety for C

- **starting point: heap memory safety policy**
- **additional complications**:
  - unboxed structs, stack allocation, byte addressing, unaligned memory accesses, custom allocators, ...
- **different attacker model / security property** (not full abstraction)
  - absence of (spatial&temporal) memory safety violations
  - high-level reasoning principles enabled by memory safety [Alpha is for Address, draft'15]