Micro-Policies

A Framework for Tag-Based Security Monitors

Benjamin C. Pierce
University of Pennsylvania
and Chalmers University

with Arthur Azevedo de Amorim, Silviu Chiarescu, Andre Dehon, Maxime Dénès, Udit Dhawan, Nick Giannarakis, Catalin Hritçu, Antal Spector-Zabusky, Andrew Tolmach

October, 2014

Where are we?

(with software security)

:-(

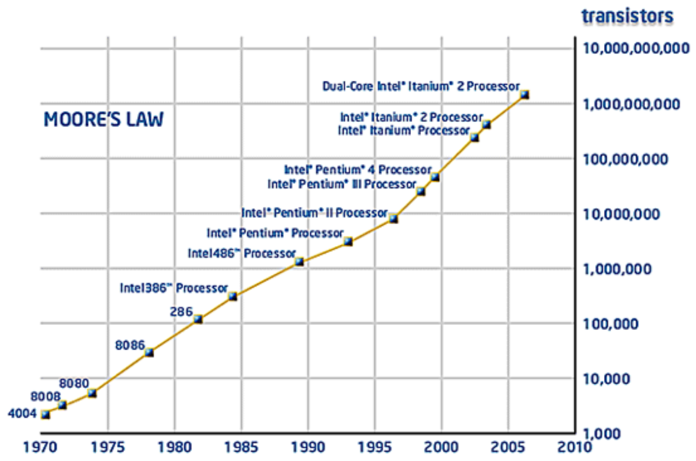
How did we get here?

Lots of reasons!

Among them…

- Legacy technology of the 1960s - 80s
  - Expensive hardware
  - Few computers, protecting a little, not networked
- ➔ Poor HW abstractions

What’s Changed?



(In)security more urgent…

- Bigger software
  - (harder to get right)
- Ubiquitous networking
- Protecting more valuable stuff

But also…

- 4+ decades of Moore’s Law
  - Hardware is cheap

Our Goals

Idea: Make hardware enforce more invariants

- (First, communicate invariants to the hardware!)

Approach: **Micro-Policies**

- Hardware-accelerated, instruction-level enforcement of security policies based on checking and propagating rich metadata

Win:

- Ubiquitous policy enforcement at all system levels
- Safety interlock: tolerate errors in operation (bugs in trusted code, transient errors)
- *Programmable* hardware supports a wide range of policies and allows rapid adaptation to threats

Origins

- This work is an outgrowth of the DARPA-funded CRASH/SAFE design
- CRASH/SAFE was a clean-slate, whole system redesign
  - ISA, hardware, OS, languages, compilers, applications…
- Recent focus:
  - Custom processor → extend conventional ISA
  - Low-level information-flow-control → enforcement of a range of *micro-policies* (including IFC among many others)

(Potential) Micro-Policies

- Information-Flow Control
- Signing
- Sealing
- Endorsement
- Taint
- Confidentiality
- Low-Level Type Safety
- Memory Safety
- Control-Flow Integrity
- Stack Safety
- Unforgeable Resource Identifiers
- Abstract Types
- Immutability
- Linearity
- Software Architecture Enforcement
- Numeric Units
- Mandatory Access Control
- Classification levels
- Lightweight compartmentalization
- Sandboxing
- Access control
- Capabilities
- Provenance
- Full/Empty Bits
- Concurrency: Race Detection
- Debugging
- Data tracing
- Introspection
- Audit
- Reference monitors
- GC support
- Bignum common cases

Status

- Prototype implementations of several micro-policies
  - dynamic sealing
  - memory safety
  - control-flow integrity
  - compartmentalization
  - information-flow control (IFC)
- Experiments with simulated Alpha processor + micro-policy hardware and low-level support software
- Formalization of simplified hardware and proofs of correctness for a few micro-policies

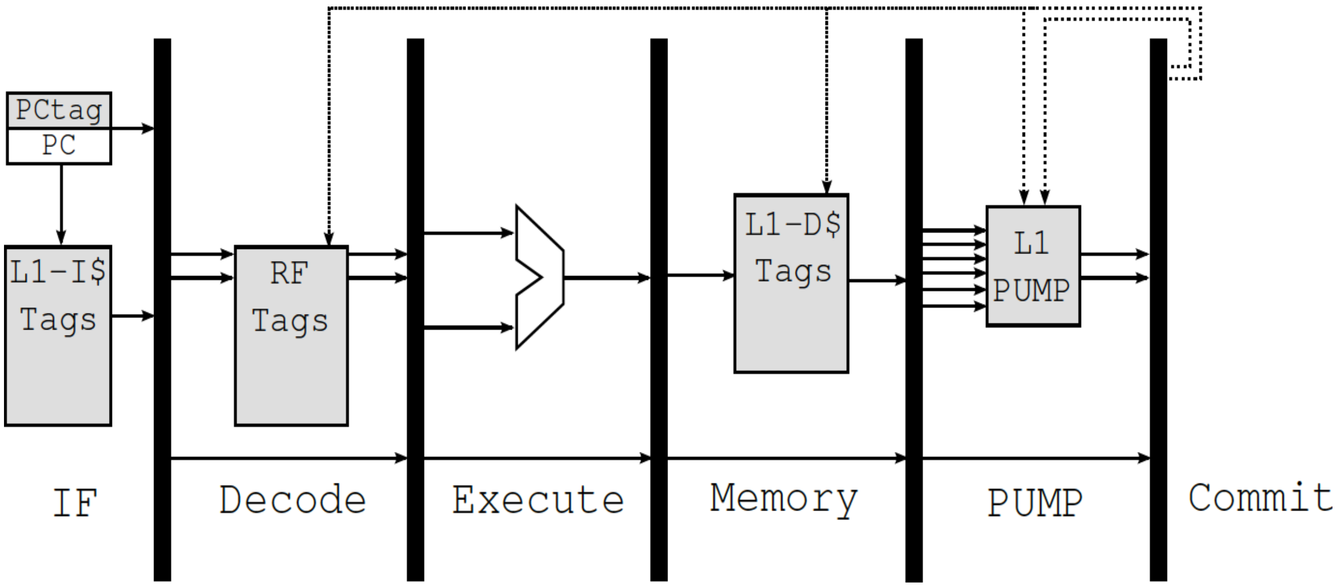
HARDWARE ARCHITECTURE

PUMP Architecture

(Programmable Unit for Metadata Processing)

- Start with conventional processor architecture (e.g. Alpha)
- Add full word-sized tag to every word
  - In memory, cache, register file…
  - (Conceptual model: efficient implementations may compress!)
- Tagged word is indivisible atom in machine
- Process tags in parallel with ALU operations
  - **Hardware** *rule cache*
  - **Software** *policy monitor* that fills hardware cache as needed

Integrate PUMP into Conventional RISC Processor Pipeline



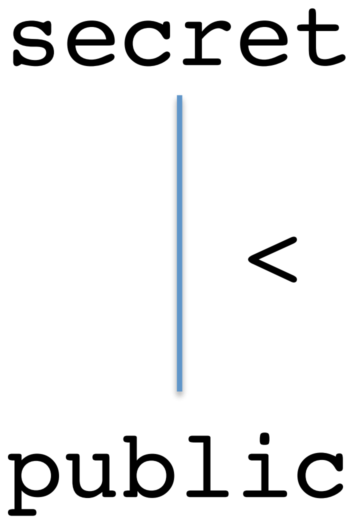
Overheads

Experiments (using SPEC2006 benchmarks, running on a simulated Alpha + PUMP, enforcing a fairly rich policy) show…

- modest impact on runtime (typically <10%) and power ceiling (<10%)
- significant (but bearable?) increase in energy (typically <60%) and area for on-chip memory structures (110%)

EXAMPLE:
TAINT TRACKING

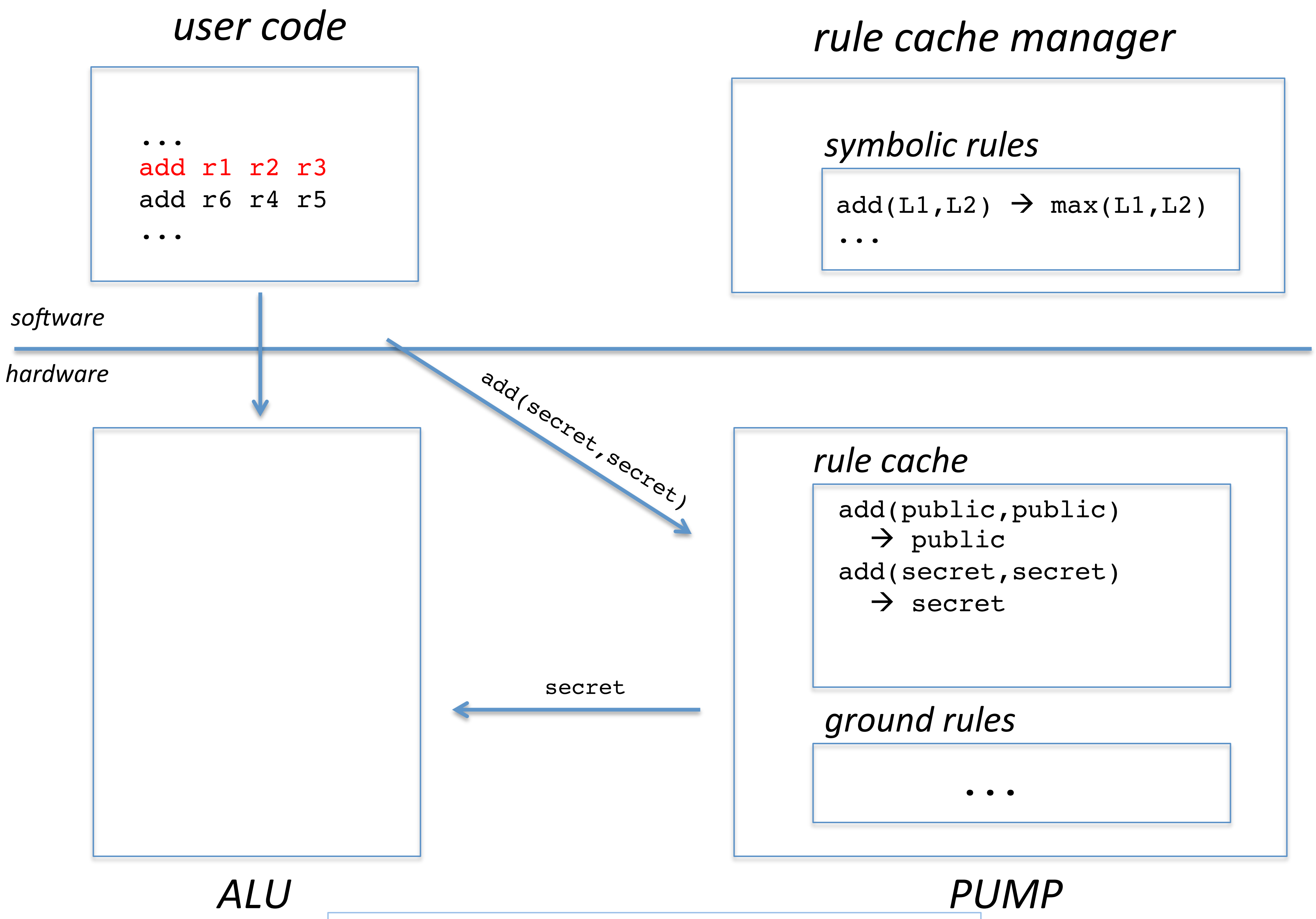
Tags for Taint Tracking



secret

<

public



user code

```
...
add r1 r2 r3
add r6 r4 r5
...
```

rule cache manager

symbolic rules

```
add(L1,L2) → max(L1,L2)
...
```

software

hardware

`add(secret,secret)`

rule cache

```
add(public,public)
  → public
add(secret,secret)
  → secret
```

`secret`

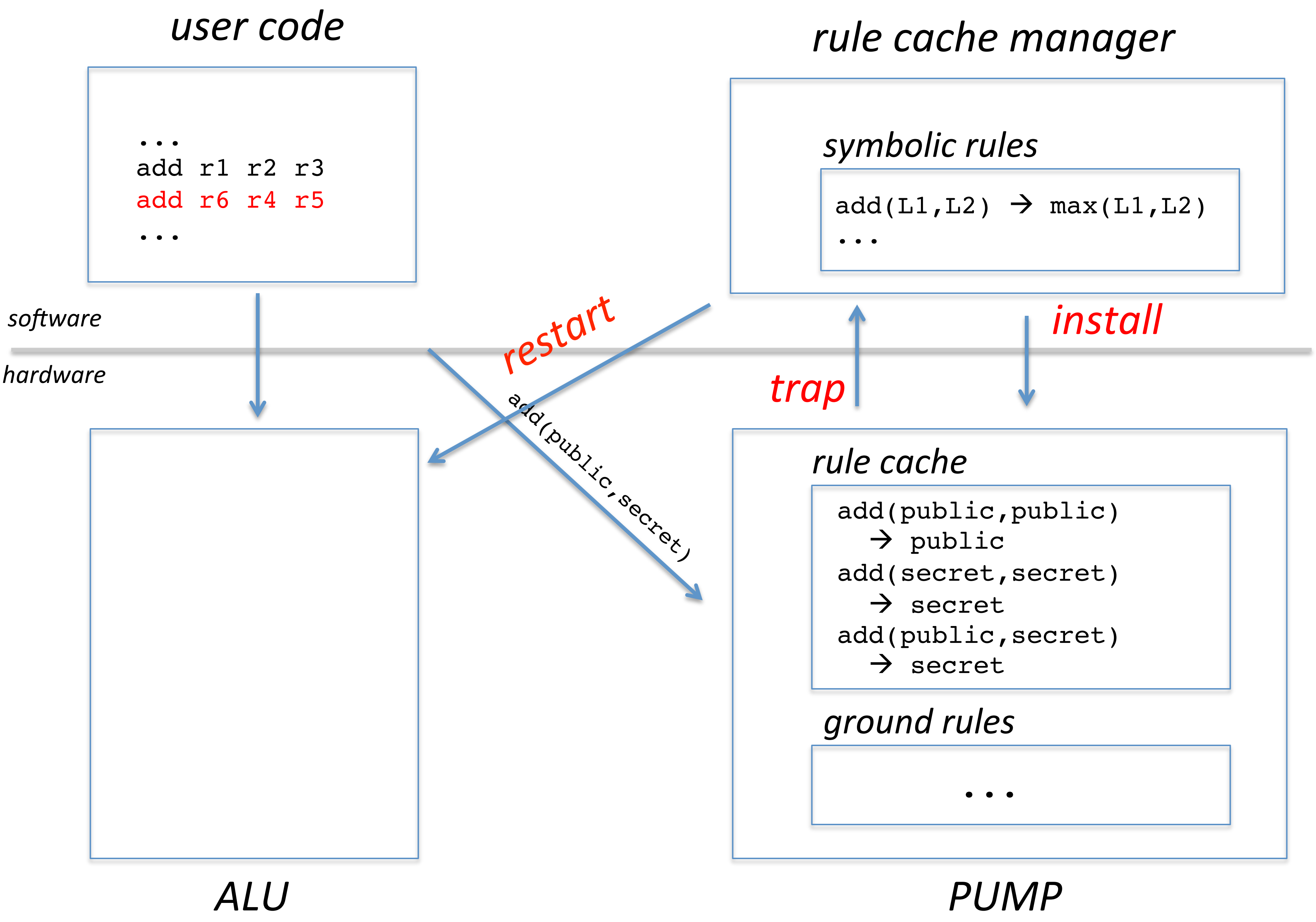
ground rules

```
...
```

ALU

PUMP

(simplified picture, showing only some tags!)



user code

```
...
add r1 r2 r3
add r6 r4 r5
...
```

rule cache manager

symbolic rules

```
add(L1,L2) → max(L1,L2)
...
```

software

hardware

restart

trap

install

add(public,secret)

rule cache

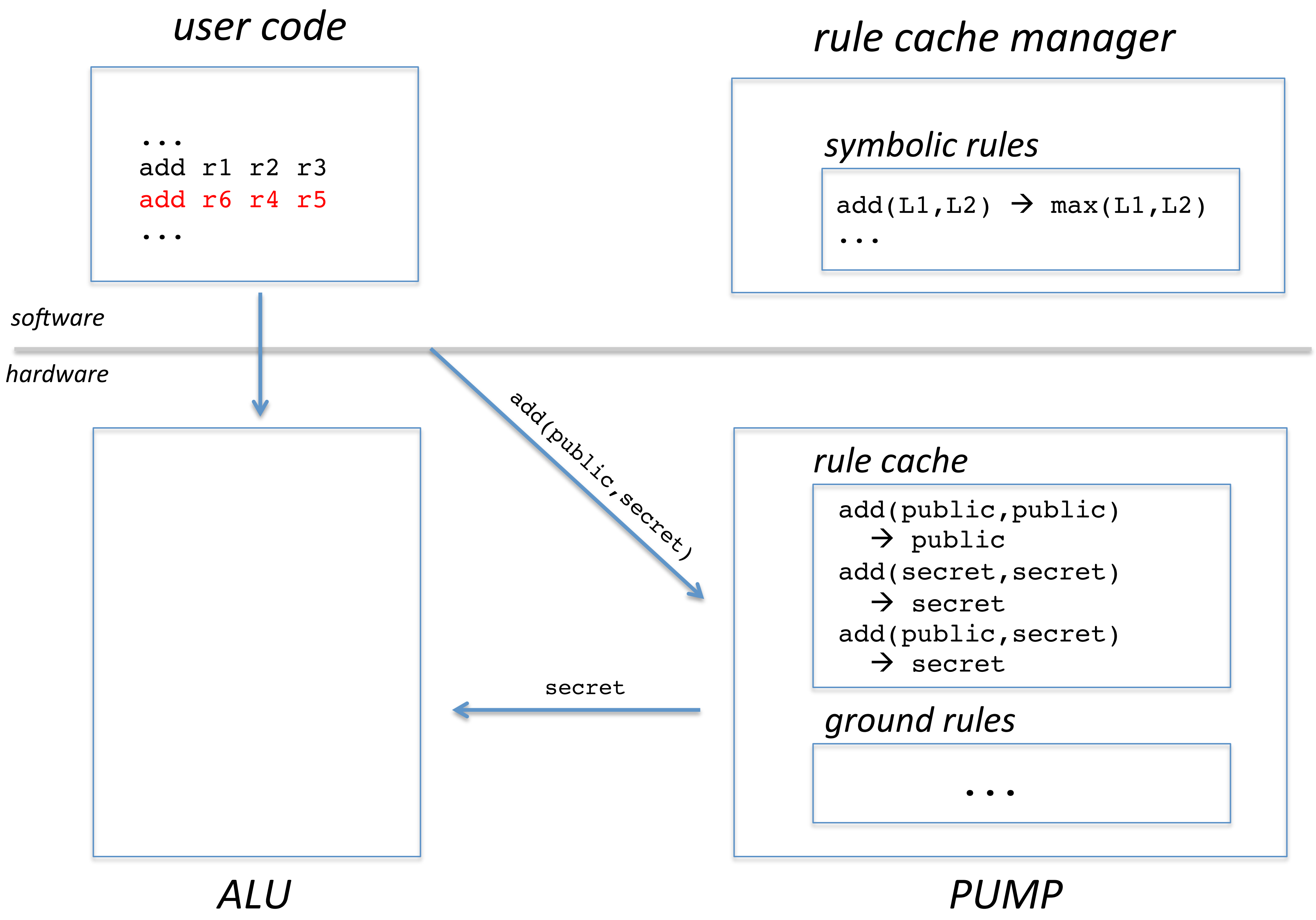
```
add(public,public)
  → public
add(secret,secret)
  → secret
add(public,secret)
  → secret
```

ground rules

```
...
```

ALU

PUMP



user code

```
...
add r1 r2 r3
add r6 r4 r5
...
```

rule cache manager

symbolic rules

```
add(L1,L2) → max(L1,L2)
...
```

software

hardware

`add(public,secret)`

rule cache

```
add(public,public)
  → public
add(secret,secret)
  → secret
add(public,secret)
  → secret
```

ground rules

```
...
```

`secret`

ALU

PUMP

Scaling up to Full Dynamic Information-Flow Control

- Use tag on PC to track implicit flows
- Word-sized tags can hold pointers to arbitrary data structures

  → labels can represent, for example, *sets* of principals

  – N.b.: tags are still just bit patterns as far as the hardware is concerned!

Protecting the Protector

Q: How does all this work when the code that’s running is the rule cache manager itself?

A: Very carefully!

Protecting the Protector

Kernel tag

- Predefined bit pattern used (only) to tag micro-policy code and private data structures
- On rule cache misses, store current machine state, set PC tag to Kernel, and start executing cache manager code at fixed location
- When cache manager finishes, return to user code (resetting PC *and its tag* to previous values)

Ground rules

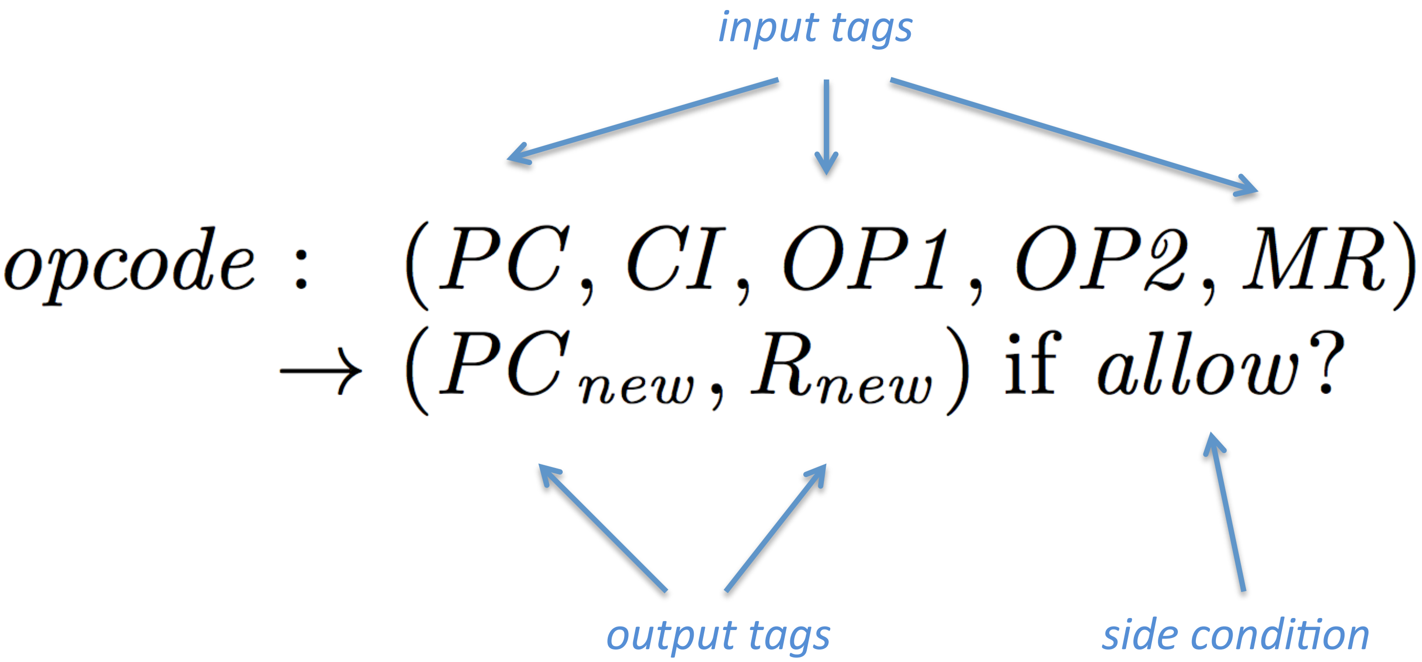
- Installed at boot time (by trusted boot sequence)
- Allow instructions to proceed only when *both* PC and current instruction are tagged Kernel
- Allow tag-manipulating instructions only when PC is tagged Kernel

MICRO-POLICIES

Anatomy of a Micro-Policy

- Set of **tags** for labeling registers, memory, PC
- **Rules** for propagating tags as the machine executes each instruction
- **Monitor services** for performing larger / more global operations involving tags

Symbolic Rules



input tags

$$opcode : \quad (PC, CI, OP1, OP2, MR) \\ \rightarrow (PC_{new}, R_{new}) \text{ if } allow?$$

output tags

side condition

Dynamic Sealing

- **Tags:** *Data* | *Key(k)* | *Sealed(k)*
- **Monitor services:**
  - *NewKey* generates a new key *k* and returns 0 tagged with *Key(k)*
  - *Seal* takes arguments *v@Data* and _*@Key(k)* and returns *v@Sealed(k)*
  - *Unseal* takes *v@Sealed(k)*and _*@Key(k)* and returns *v@Data*
- **Rules:**
  - Data movement instructions (Mov, Load, Store) preserve tags.
  - Data manipulation instructions (indirect jumps, arithmetic, …) fault on tags other than *Data*

$$
\begin{aligned}
Store &: (Data, Data, Data, t_{src}, -) \rightarrow (Data, t_{src}) \\
Jal &: (Data, Data, Data, -, -) \rightarrow (Data, Data)
\end{aligned}
$$

Control-Flow Integrity

- **Tags:** Each instruction that can be the source or target of a control-flow edge is tagged (by compiler) with a unique tag
- **Rules:**
  - On a jump, call, or return, copy tag of current instruction onto tag of PC
  - Whenever PC tag is nonempty, compare it with current instruction tag (and abort on mismatch)

Memory Safety

- **Tags:**
  - Each call to malloc generates a fresh tag T
  - Newly allocated memory cells tagged with T
  - Pointer to new region tagged “pointer to T”
- **Rules:**
  - Load and store instructions check that their targets are tagged “pointer to T” and that the referenced memory cell is tagged T (for the same T)
  - Pointer arithmetic instructions preserve “pointer to T” tags

Compartmentalization

à la SFI

- **Idea:**
  - Divide memory into finite set of compartments
  - Each compartment can jump and write only to predetermined set of addresses in other compartments
- **Tags:**
  - PC tagged with current compartment
  - Each memory location is tagged with the set of compartments that are allowed to affect it
- **Rules:**
  - On each write and after each branch, compare PC tag with tag of memory location being written or executed
- **Monitor services:**
  - *NewCompartment* splits the current compartment into two subcompartments (legal jump and write targets are provided as parameters—must be a subset of parent compartment's)

Composition

- Current topic: How do we *compose* micro-policies??
- Some policies are essentially orthogonal:
  - E.g., memory safety and CFI or sealing
  - Compose by tupling
  - Just need to designate a *default tag* for each policy
- But some are not…
  - E.g. memory safety and compartmentalization
    - (because newly allocated regions need their compartment tags reset)
- Possible approaches:
  - Identify a small set of primitive operations like memory allocation that need special treatment
  - And/or compose policies "in series" rather than "in parallel" (in the style of Haskell monad transformers or "algebraic effects")

PROOF ARCHITECTURE

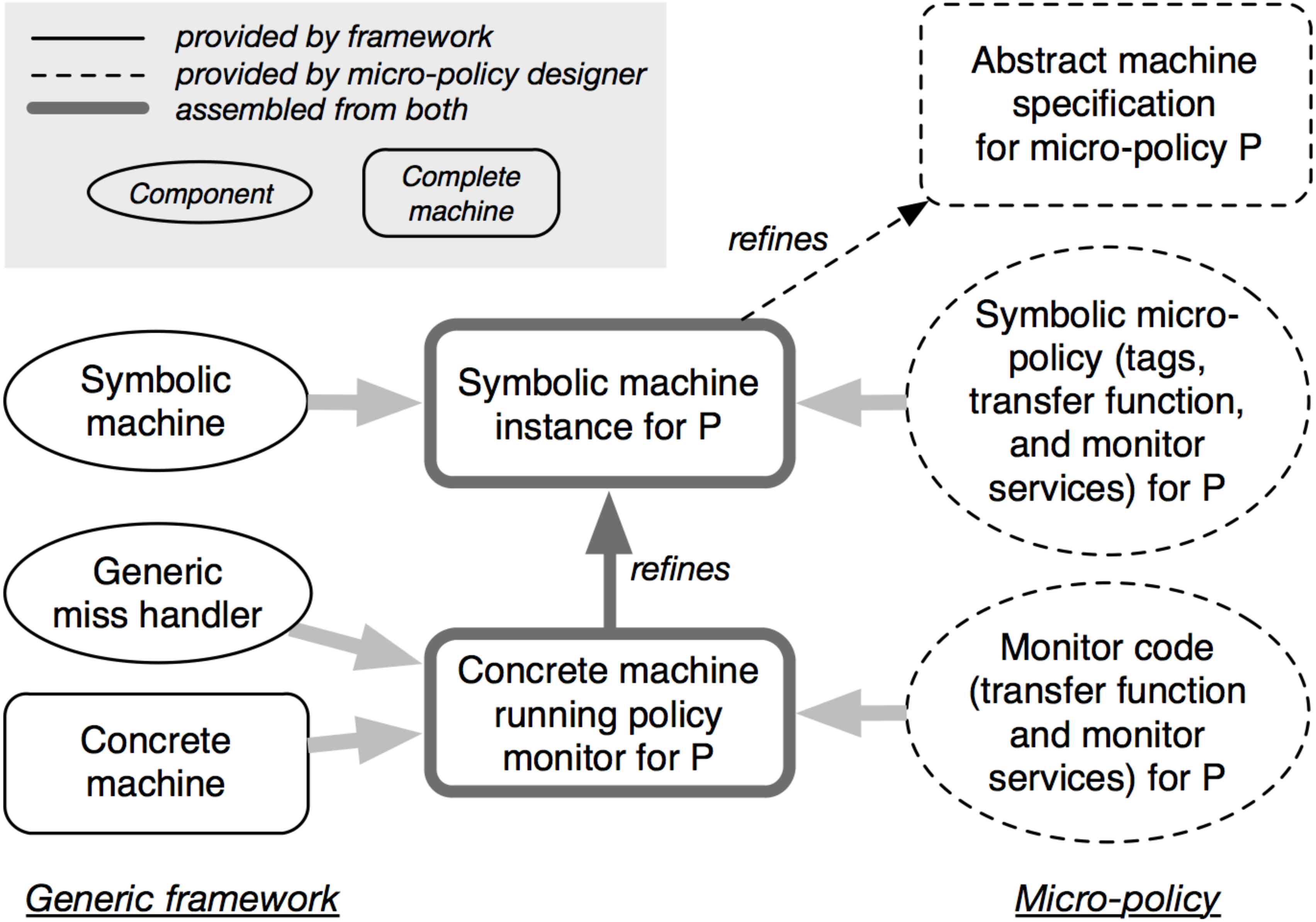
Some things to prove…

Q: The interplay between the hardware rule cache, the software rule cache manager, the ground rules, and the symbolic policy is somewhat intricate…

- How do we know that it works correctly in all cases?

Q: For each micro-policy, how do we know that its realization in terms of tags and rules corresponds to some intended high-level constraint on program behavior?

- I.e., how do we know that the symbolic policy is what the user intends?

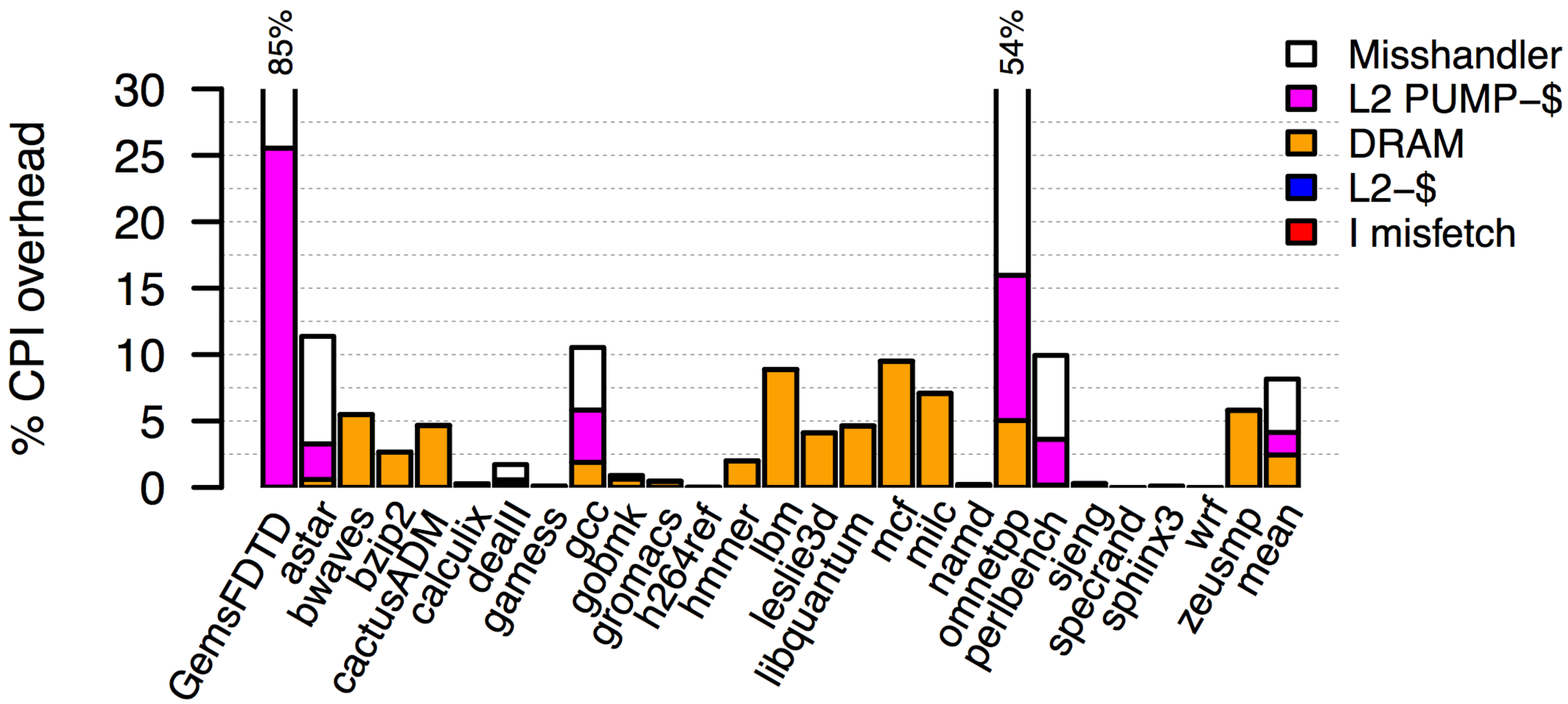

provided by framework
provided by micro-policy designer
assembled from both
Component
Complete machine
Abstract machine specification for micro-policy P
refines
Symbolic machine
Symbolic machine instance for P
Symbolic micro-policy (tags, transfer function, and monitor services) for P
Generic miss handler
refines
Concrete machine
Concrete machine running policy monitor for P
Monitor code (transfer function and monitor services) for P
Generic framework
Micro-policy

Results

- Last year: [POPL14]
  - noninterference for a simple symbolic IFC policy
  - correct implementation of this policy by a rule-table compiler and rule cache handler routine
  - on a simplified SAFE architecture
- This year:
  - four diverse micro-policies (sealing, compartmentalization, memory safety, CFI)
  - proofs of correctness (refinement) of symbolic policies wrt. high-level abstract machines
  - protection and compartmentalization of kernel code
  - …on a simple RISC + PUMP

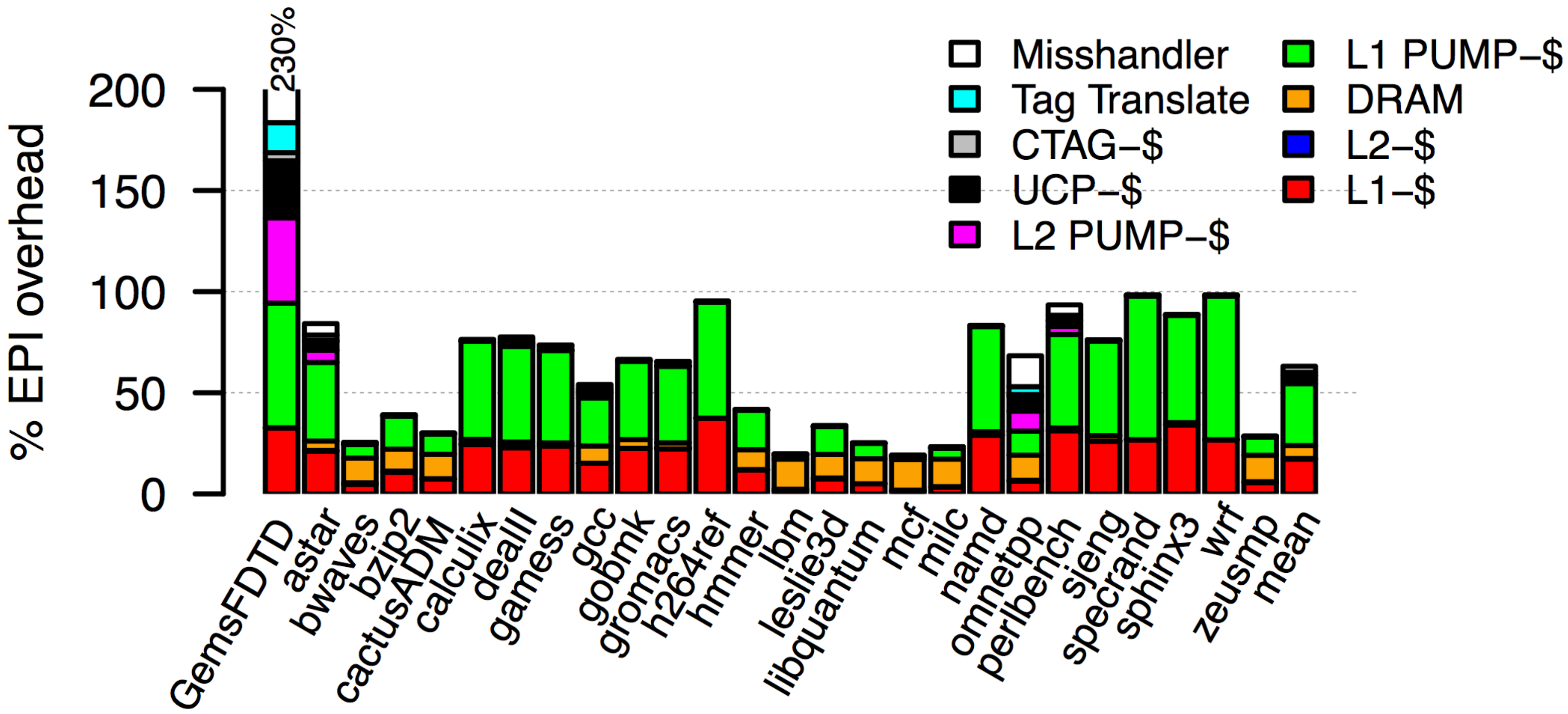
EMPIRICAL EVALUATION

Runtime Overhead

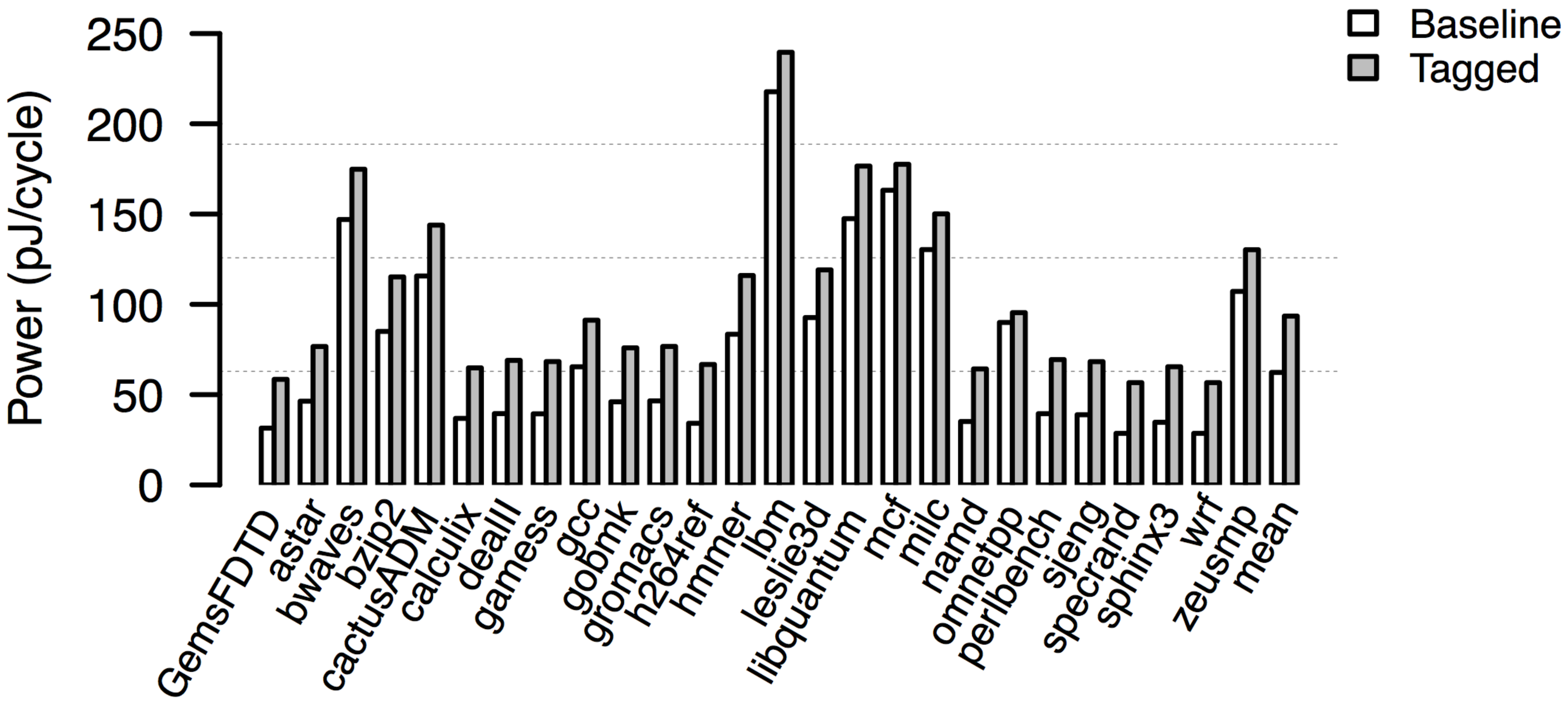


Simulated Alpha+PUMP running SPEC2006 benchmark suite with composite micro-policy (memory safety + CFI + taint tracking)

Energy Overhead



Absolute Power

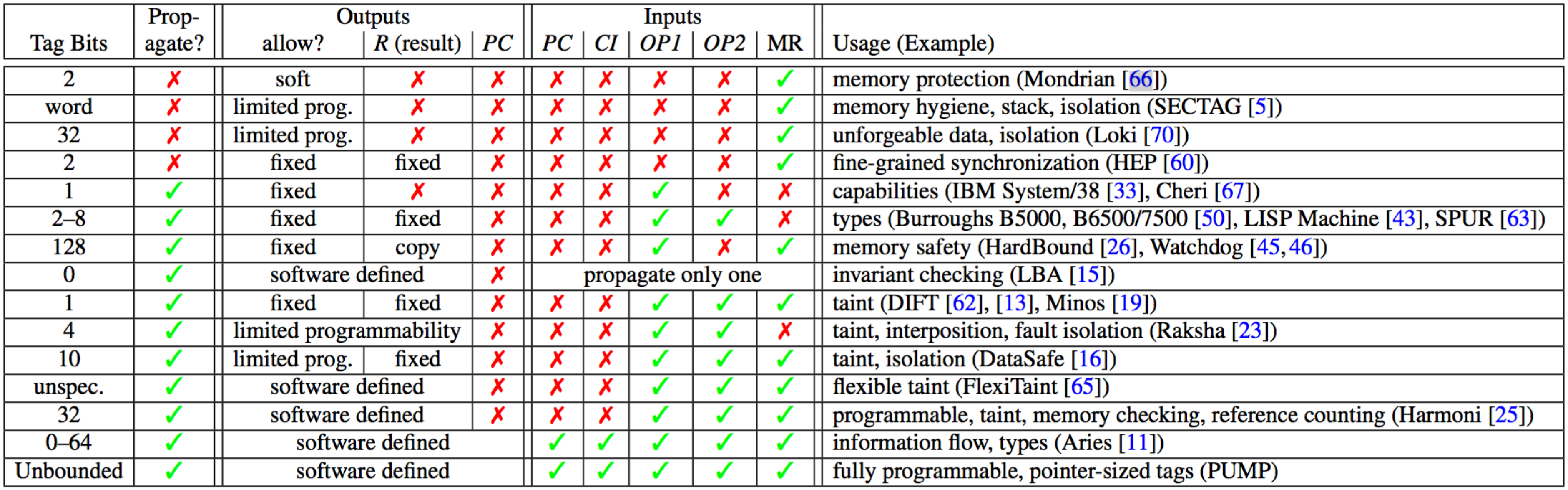


Area

- Significant on-chip area overhead (mostly for memory structures)
  - around 110%
- Existing optimization techniques (Mondriaan Memory, etc.) should help for off-chip memory

FINISHING UP…

Related Work



	Prop-	Outputs			Inputs					
Tag Bits	agate?	allow?	*R* (result)	*PC*	*PC*	*CI*	*OP1*	*OP2*	MR	Usage (Example)
2	✗	soft	✗	✗	✗	✗	✗	✗	✓	memory protection (Mondrian [66])
word	✗	limited prog.	✗	✗	✗	✗	✗	✗	✓	memory hygiene, stack, isolation (SECTAG [5])
32	✗	limited prog.	✗	✗	✗	✗	✗	✗	✓	unforgeable data, isolation (Loki [70])
2	✗	fixed	fixed	✗	✗	✗	✗	✗	✓	fine-grained synchronization (HEP [60])
1	✓	fixed	✗	✗	✗	✗	✓	✗	✗	capabilities (IBM System/38 [33], Cheri [67])
2–8	✓	fixed	fixed	✗	✗	✗	✓	✓	✗	types (Burroughs B5000, B6500/7500 [50], LISP Machine [43], SPUR [63])
128	✓	fixed	copy	✗	✗	✗	✓	✗	✓	memory safety (HardBound [26], Watchdog [45, 46])
0	✓	software defined		✗	propagate only one					invariant checking (LBA [15])
1	✓	fixed	fixed	✗	✗	✗	✓	✓	✓	taint (DIFT [62], [13], Minos [19])
4	✓	limited programmability		✗	✗	✗	✓	✓	✗	taint, interposition, fault isolation (Raksha [23])
10	✓	limited prog.	fixed	✗	✗	✗	✓	✓	✓	taint, isolation (DataSafe [16])
unspec.	✓	software defined		✗	✗	✗	✓	✓	✓	flexible taint (FlexiTaint [65])
32	✓	software defined		✗	✗	✗	✓	✓	✓	programmable, taint, memory checking, reference counting (Harmoni [25])
0–64	✓	software defined			✓	✓	✓	✓	✓	information flow, types (Aries [11])
Unbounded	✓	software defined			✓	✓	✓	✓	✓	fully programmable, pointer-sized tags (PUMP)

Future Work

- More μPolicies!
- Policy composition?
- User-defined policies?
- Pure-software or hybrid implementation?
- Zero-kernel OS?

Conclusion

- Host of security problems arise from violation of well-understood low-level invariants
- Spend modest hardware to check
  - – Ubiquitously enforce in parallel with execution
- Programmable PUMP Model
  - – Richness and flexibility of software enforcement…
  - – …with the performance of hardware!
  - – Reduce or eliminate security/performance tradeoff