



# Programmable Hardware Support for Ubiquitous Micro-Policy Enforcement

André DeHon, Benjamin C. Pierce

University of Pennsylvania

(joint work with Harvard and BAE)

HCSS, May 2014

# Where are we?

(wrt. software security)

Nowhere good

# How did we get here?

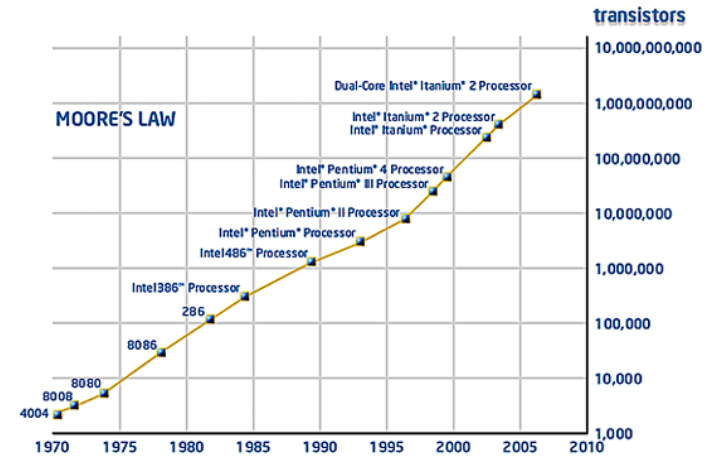
Lots of reasons!

Among them...

- Legacy of technology of the 1960s - 80s
  - Expensive hardware
  - Limited verification capabilities
  - Few computers, protecting a little, not networked
- Poor HW abstractions, high performance cost to isolation

# What's Changed?

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



But also...

- 4+ decades of Moore's Law
  - Hardware is cheap
- Huge progress in formalizing / verifying software

# Our Goals

**Idea:** Make hardware enforce more invariants

- Must first communicate invariants to the hardware!

**Win:**

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

# HARDWARE ARCHITECTURE

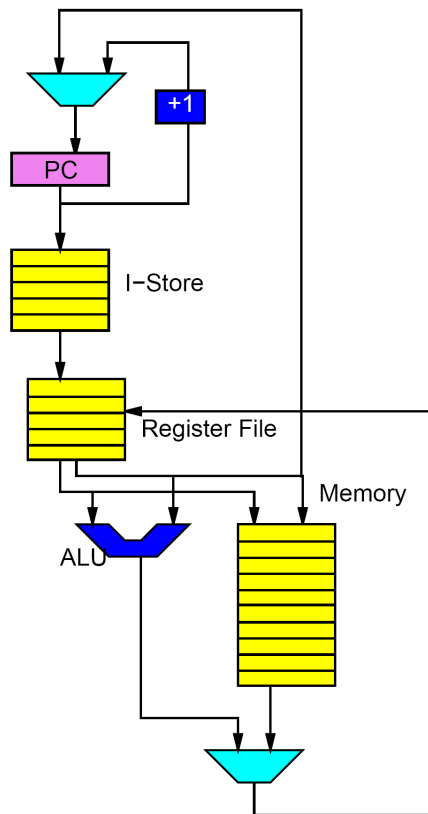
# PUMP Architecture

(Programmable Unit for Metadata Processing)

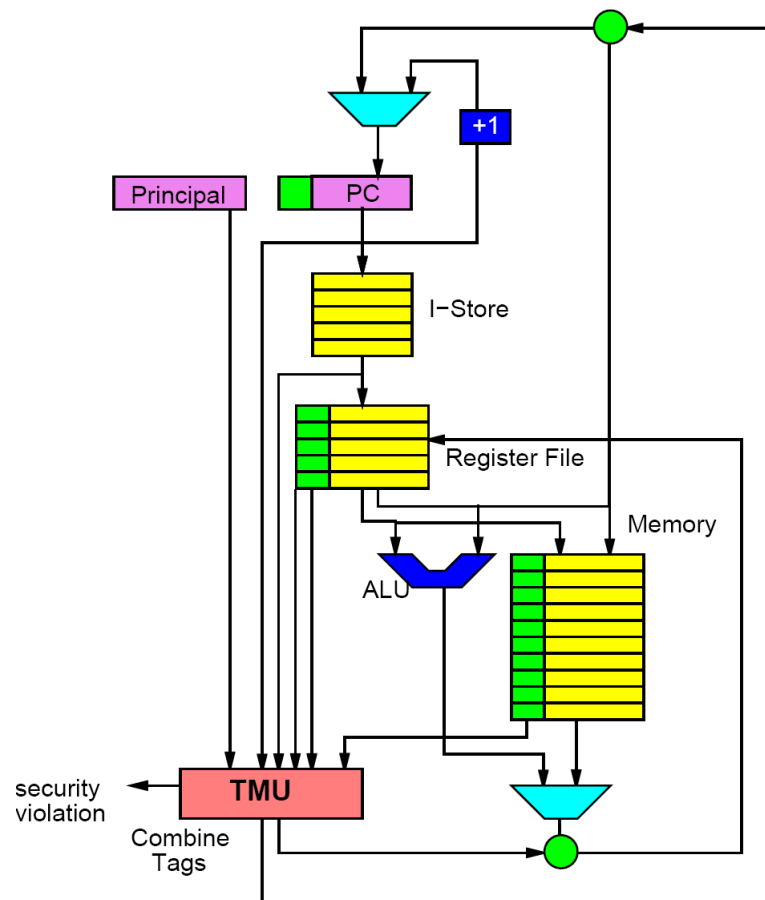
- Add full word-sized tag to every word
  - In memory, cache, register file...
  - (Conceptual model: efficient implementations may compress!)
- Tagged word is indivisible atom in machine
- Process tags in parallel with ALU operations
  - **Hardware** *rule cache*
  - **Software** *policy system* that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

# Process Tags in Parallel

Conventional processor

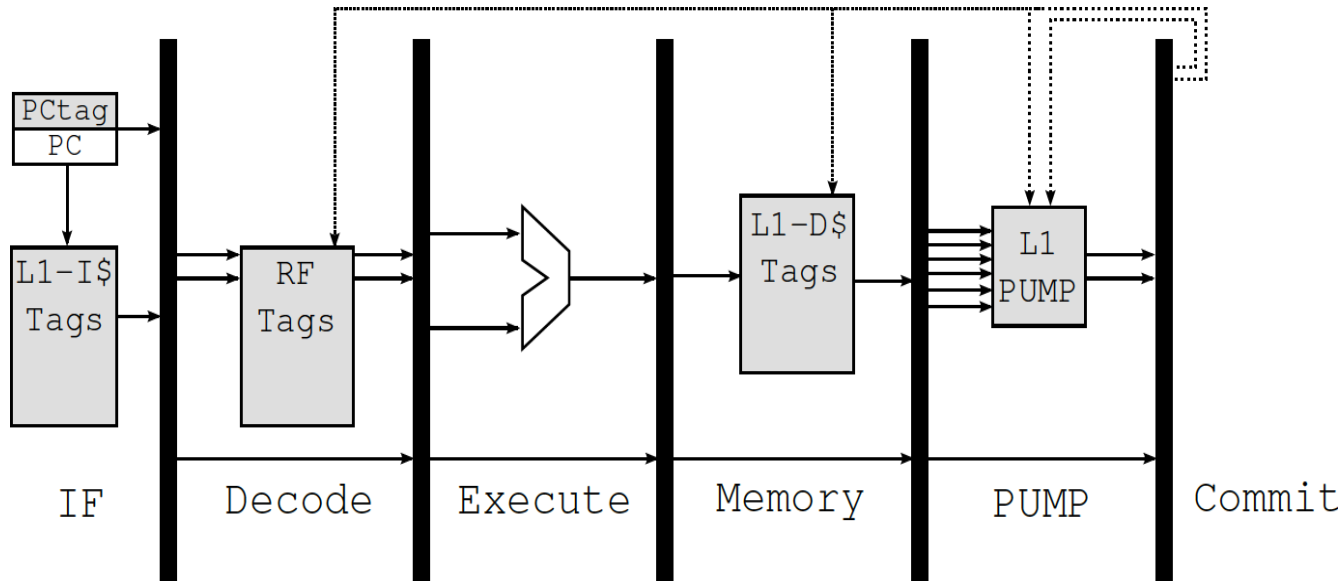


Processor + PUMP





# Integrate PUMP into Conventional RISC Processor Pipeline



# **EXAMPLE: INFORMATION-FLOW CONTROL**

*user code*

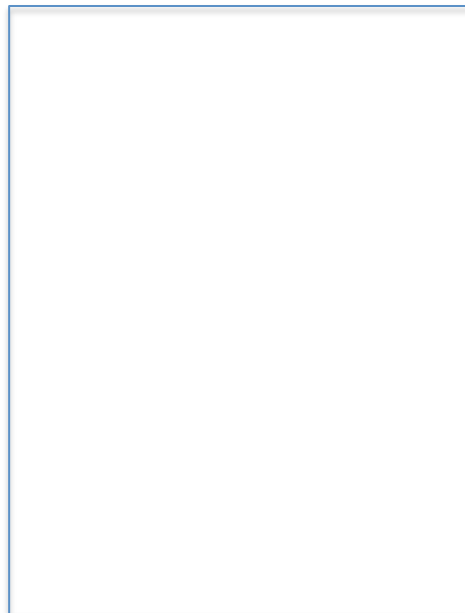
```
...  
add r1 r2 r3  
add r1 r4 r5  
...
```

*rule cache manager*

```
symbolic rules  
add(L1,L2) → max(L1,L2)  
...
```

*software*

*hardware*



*ALU*

`add(secret,secret)`



`secret`



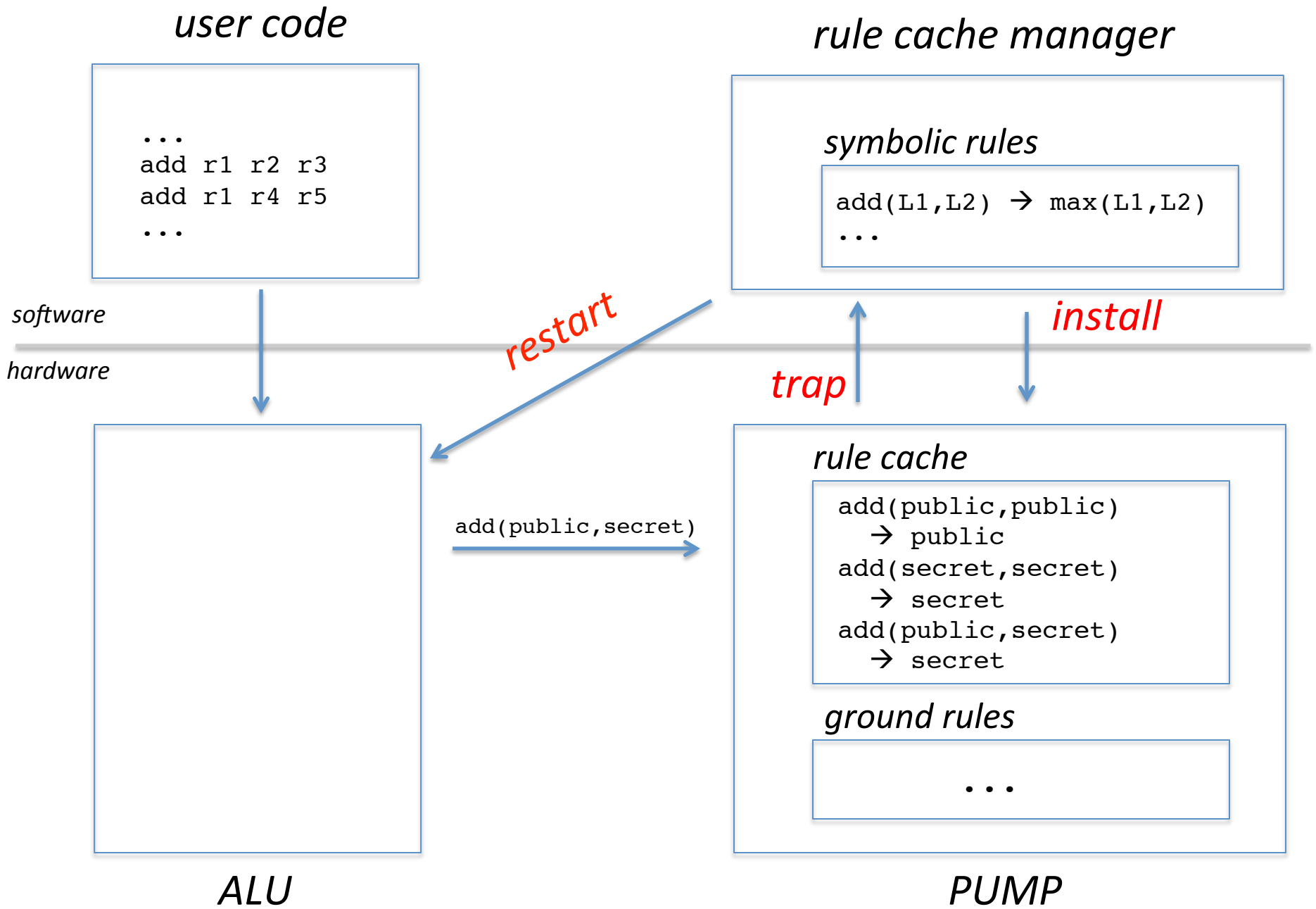
*rule cache*

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

*ground rules*

```
...
```

*PUMP*



*user code*

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

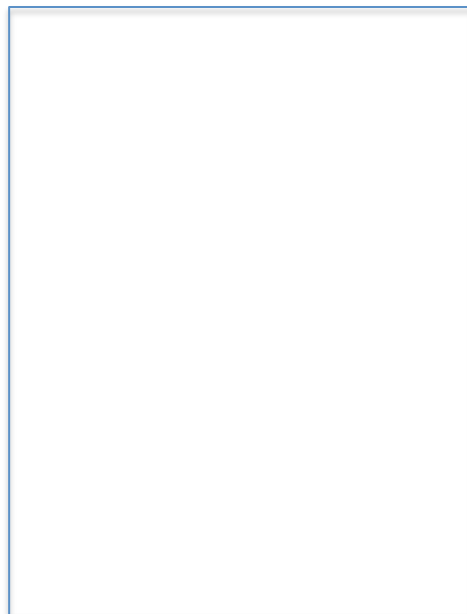
*rule cache manager*

*symbolic rules*

```
add(L1,L2) → max(L1,L2)  
...
```

*software*

*hardware*



*ALU*

add(public,secret)



secret



*rule cache*

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

*ground rules*

```
...
```

*PUMP*

# Scaling up to Full IFC

- Tag on PC tracks implicit flows
- Word-sized tags can hold pointers to arbitrary data structures
  - labels can represent, for example, *sets* of principals

# Protecting the Protector

**Q:** How do we prevent the attacker gaining control of the PUMP itself?

**A:** *Ground rules*

- Installed at boot time (by trusted boot sequence)
- Allow tag-manipulating instructions only in carefully controlled contexts

# The Role of Formal Methods

**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?
- How do we know that the symbolic policy is what the user intends?

**A:** Though complex, this is a small enough artifact that we can hope to *prove* these properties



# Formal Methods: Status

[POPL14, S&P13]

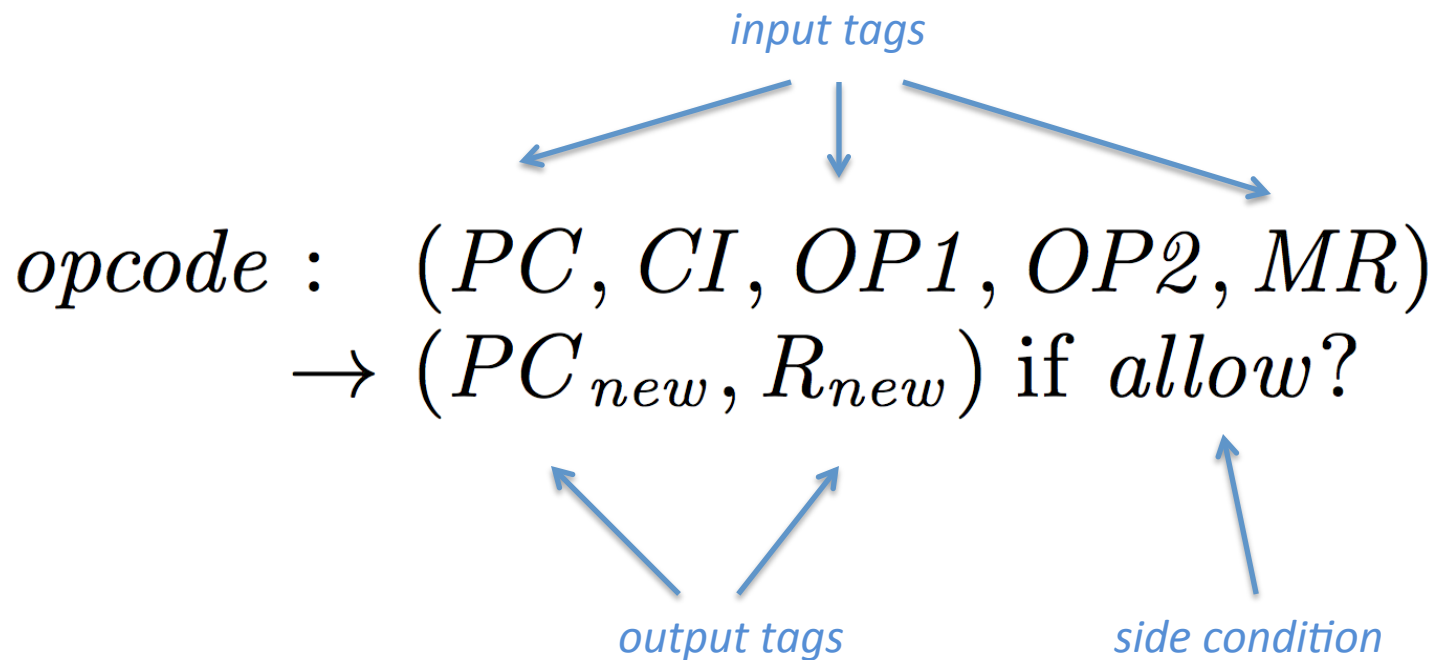
- Formal, machine-checked proofs (in Coq) of
  - noninterference for a simple symbolic IFC policy
  - correct implementation of this policy by a rule-table compiler and rule cache handler routine (on a simplified hardware architecture)
- Currently extending both methodologies to more realistic models, including
  - protection and compartmentalization of kernel code
  - additional policies beyond IFC...

# MICRO-POLICIES

# 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

# Symbolic Rules



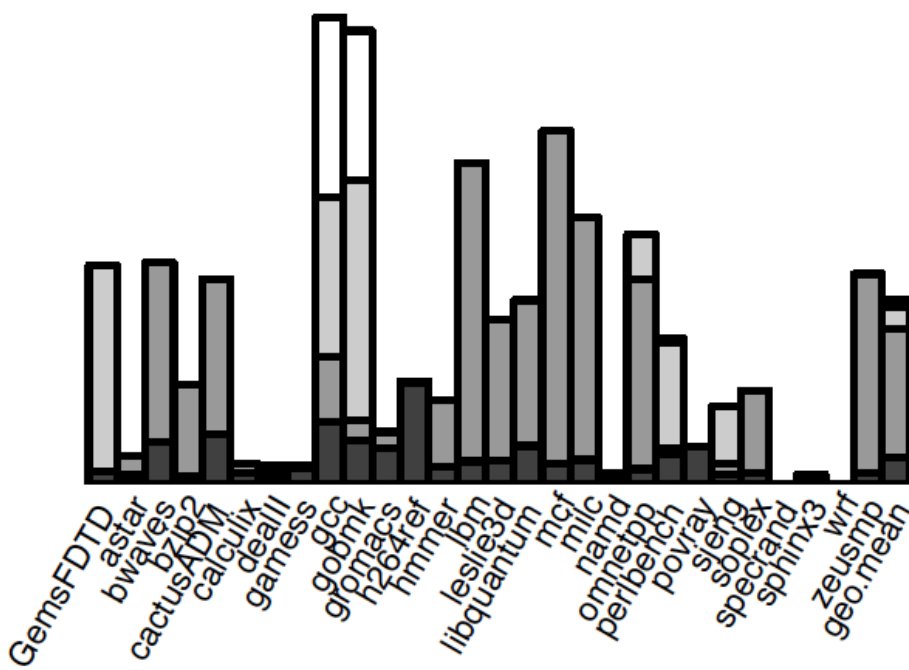
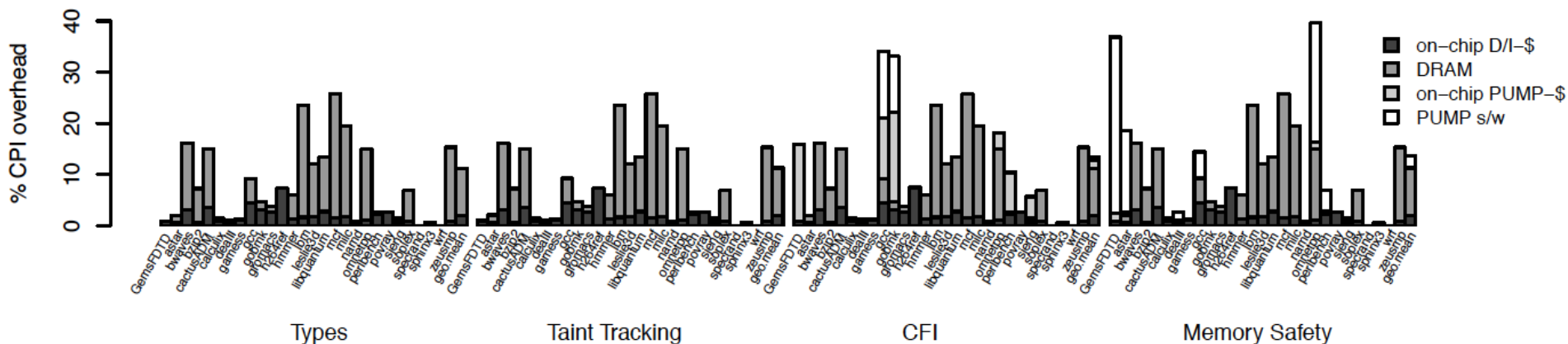
# Control-Flow Integrity

- **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

# Performance Overhead (SPEC2006)



[HASP 2014]

# FINISHING UP...



# Future Work

- Micro-architectural optimization
  - Reduce energy, area, delay overhead
- Define more  $\mu$ Policies, characterize security properties, implement, formally validate
- Understand policy composition
- Use to compartmentalize, shrink trusted computing base

# 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...
  - ...with the performance of hardware!
  - Reduce or eliminate security/performance tradeoff