Micro-Policies
A Framework for Tag-Based Security Monitors

Benjamin C. Pierce
University of Pennsylvania

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

December, 2014
Where are we?
(wrt. software security)
How did we get here?

Lots of reasons!

Among them...

– Legacy technology of the 1960s - 80s
  • Few computers, protecting a little, not networked
  • Expensive hardware

⇒ Poor hardware abstractions
What’s Changed?

(In)security more urgent...

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

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
   – *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 $\rightarrow$ extend conventional ISA
  – Low-level information-flow-control $\rightarrow$ 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
Current Status

• Prototype implementations of several micro-policies...
  – dynamic sealing
  – memory safety
  – control-flow integrity
  – compartmentalization
  – information-flow control (IFC)

• Formalization of (simplified) hardware and proofs of correctness for these micro-policies

• Experiments with simulated Alpha processor + tag-propagation hardware + low-level support software
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 composite policy) show...

- modest impact on runtime (typically <10%) and power ceiling (<10%)
- more 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)
...

rule cache

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

ground rules

...
user code

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

rule cache manager

symbolic rules

add(L1,L2) -> max(L1,L2)
...

rule cache

add(public,public) -> public
add(secret,secret) -> secret
add(public,secret) -> secret

ground rules

...
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

**Monitor 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 Monitor, 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 Monitor
- Allow tag-manipulating instructions only when PC is tagged Monitor
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

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

\[
\text{Store} : \ (Data, Data, Data, t_{src}, -) \rightarrow (Data, t_{src}) \\
\text{Jal} : \ (Data, Data, Data, -, -) \rightarrow (Data, Data)
\]
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
Compartimentalization

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

• Challenge: 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?
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:** [under submission]
  – 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 Monitor 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

![Bar chart showing absolute power comparison between Baseline and Tagged for various benchmarks. The x-axis represents different benchmarks including GemsFDTD, astar, bzip2, cacti, gcc, etc., while the y-axis shows power in pJ/cycle.]
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

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

• Additional benefits...
  – Ubiquitous policy enforcement at all system levels
  – Safety interlocks: tolerate errors in operation (bugs in trusted code, transient errors)