# Architectural Support for Software-Defined Metadata Processing

Udit Dhawan[1] Cătălin Hriţcu[2] Raphael Rubin[1] Nikos Vasilakis[1] Silviu Chiricescu[3]
Jonathan M. Smith[1] Thomas F. Knight Jr.[4] Benjamin C. Pierce[1] André DeHon[1]

[1]University of Pennsylvania [2]INRIA, Paris [3]BAE Systems [4]Ginkgo Bioworks

Contact author: udit@seas.upenn.edu

## Abstract

Optimized hardware for propagating and checking software-programmable metadata tags can achieve low runtime overhead. We generalize prior work on hardware tagging by considering a generic architecture that supports software-defined policies over metadata of arbitrary size and complexity; we introduce several novel microarchitectural optimizations that keep the overhead of this rich processing low. Our model thus achieves the efficiency of previous hardware-based approaches with the flexibility of the software-based ones. We demonstrate this by using it to enforce four diverse safety and security policies—spatial and temporal memory safety, taint tracking, control-flow integrity, and code and data separation—plus a composite policy that enforces all of them simultaneously. Experiments on SPEC CPU2006 benchmarks with a PUMP-enhanced RISC processor show modest impact on runtime (typically under 10%) and power ceiling (less than 10%), in return for some increase in energy usage (typically under 60%) and area for on-chip memory structures (110%).

***Categories and Subject Descriptors*** C.1 [*Processor Architecture*]: Miscellaneous—security

***Keywords*** security, metadata, tagged architecture, CFI, Taint Tracking, Memory Safety

## 1. Introduction

Today's computer systems are notoriously hard to secure, and conventional processor architectures are partly to blame, admitting behaviors (pointer forging, buffer overflows, ...) that blatantly violate higher-level abstractions. The burden of closing the gap between programming language and hardware is left to software, where the cost of enforcing airtight abstractions is often deemed too high.

Several recent efforts [2, 37, 53, 57] have demonstrated the value of propagating metadata during execution to enforce policies that catch safety violations and malicious attacks as they occur. These policies can be enforced in software [3, 8, 28, 48, 69], but typically with high overheads that discourage their deployment or motivate coarse approximations providing less protection [23, 31]. Hardware support for fixed policies can often reduce the overhead to acceptable levels and prevent a large fraction of today's attacks [19, 25, 40, 63]. Following this trend, Intel recently announced hardware for bounds checking [36] and isolation [42]. While these mitigate many of today's attacks, fully securing systems will require more than memory safety and isolation. Some needs we can already identify (like control-flow integrity and information flow control) but the complete set remains unknown. Attacks rapidly evolve to exploit any remaining forms of vulnerability. What is needed is a flexible security architecture that can be quickly adapted to this ever-changing landscape. Some recent designs have made the hardware metadata computation configurable [24, 66] but have limited bits to represent metadata and only support a limited class of policies. A natural question, then, is: *Is it possible to provide hardware to support extensible, software-defined metadata processing with low overhead?* In particular, in the spirit of the 0-1-$\infty$ rule, can we efficiently support fine-grained, software-defined metadata propagation without placing a visible, hard bound on the number of bits allocated to metadata or a bound on the number of policies simultaneously enforced?

To achieve this goal, we introduce a rich architectural model, the Programmable Unit for Metadata Processing (PUMP), that indivisibly associates a metadata tag with every word in the system's main memory, caches, and registers. To support unbounded metadata, the tag is large enough to indirect to a data structure in memory. On every instruction, the tags of the inputs are used to determine if the operation is allowed, and if so to determine the tags for the results. The tag checking and propagation rules are defined in software; however, to minimize performance impact, these rules are cached in a hardware structure, the *PUMP rule cache*, that



*ASPLOS 2015*, March 14–18, 2015, Istanbul, Turkey.

ACM 978-1-4503-2835-7/15/03...$15.00.
http://dx.doi.org/10.1145/694344.2694383

operates in parallel with the ALU. A software *miss handler* services cache misses based on the policy rule set currently in effect.

We measure the performance impact of the PUMP using a composition of four different policies (Tab. 1) that stress the PUMP in different ways and illustrate a range of security properties: (1) a *Non-Executable Data and Non-Writable Code (NXD+NWC)* policy that uses tags to distinguish code from data in memory and provides protection against simple code injection attacks; (2) a *Memory Safety* policy that detects all spatial and temporal violations in heap-allocated memory, extending [17] with an effectively unlimited ($2^{60}$) number of colors ("taint marks" in [17]); (3) a *Control-Flow Integrity* (CFI) [3] policy that restricts indirect control transfers to only the allowed edges in a program's control flow graph, preventing return-oriented-programming-style attacks [59] (we enforce fine-grained CFI [3, 50], not coarse-grained approximations [20, 72] that are potentially vulnerable to attack [23, 31]); and (4) a fine-grained *Taint Tracking* policy (generalizing [49]) where each word can potentially be tainted by multiple sources (libraries and IO streams) simultaneously. Since these are well-known policies whose protection capabilities have been established in the literature, we focus only on measuring and reducing the performance impact of enforcing them using the PUMP. Except for NXD+NWC, each of these policies needs to distinguish an essentially unlimited number of unique items; by contrast, solutions with a limited number of metadata bits can, at best, support only grossly simplified approximations.

As might be expected, a simple, direct implementation of the PUMP is rather expensive. Adding pointer-sized (64b) tags to 64b words at least doubles the size and energy usage of all the memories in the system; rule caches add area and energy on top of this. For this simple implementation, we measured area overhead of 190% and geomean energy overhead around 220%; moreover, runtime overhead is disappointing (over 300%) on some applications (see §3). Such high overheads would discourage adoption, if they were the best we could do.

However, we find that most policies exhibit spatial and temporal locality for both tags and the rules defined over them. The number of unique rules can be significantly reduced by defining them over a group of identical instructions, reducing compulsory misses and increasing the effective capacity of the rule caches. Off-chip memory traffic can be reduced by exploiting spatial locality in tags. On-chip area and energy overhead can be minimized by using a small number of bits to represent the subset of the pointer-sized tags in use at a time. Runtime costs of composite policy miss handlers can be decreased by providing hardware support for caching component policies. These optimizations allow the PUMP to achieve low overheads without compromising its rich policy model.

The main contributions of this work are (a) a metadata processing architecture supporting unbounded metadata and fully software-defined policies (§2); (b) an empirical evaluation (§3) of the runtime, energy, power ceiling, and area impacts of a simple implementation of the PUMP integrated with an in-order Alpha microarchitecture (§2) on the SPEC CPU2006 benchmark set under four diverse policies and their combination; (c) a set of microarchitectural optimizations (§4); and (d) an assessment of their impact (§5), showing typical runtime overhead under 10%, a power ceiling impact of 10%, and typically energy overhead under 60% by using 110% additional area for on-chip memory structures. Tab. 2 summarizes related work, which is discussed in §6; §7 concludes and highlights future work.

## 2. The PUMP

We illustrate the PUMP as an extension to a conventional RISC processor (we use an Alpha [1], but the mechanisms are not ISA-specific) and an in-order implementation with a 5-stage pipeline suitable for energy-conscious applications [10]. In this section, we describe the ISA-level extensions that constitute the PUMP's hardware interface layer, the basic microarchitectural changes, and the accompanying low-level software.

***Metadata Tags*** Every word in a PUMP system is associated with a pointer-sized tag. These tags are uninterpreted in the hardware. At the software level, a tag may represent metadata of unbounded size and complexity, as defined by the policy. Simple policies that need only a few bits of metadata may store it directly in the tag; if more bits are needed, indirection is used to store the metadata as a data structure in memory, with the address of this structure used as the tag. The basic addressable word is indivisibly extended with a tag, making all value slots, including memory, caches, and registers, suitably wider. The program counter is also tagged. This notion of software-defined metadata and its representation as a pointer-sized tag extends previous tagging approaches, where only a few bits are used for tags and/or they are hardwired to fixed interpretations (see Tab. 2).

All tag manipulation is defined and implemented with PUMP rules. Metadata tags are not addressable by user programs, only by software policy handlers invoked on rule cache misses as detailed below.

***Tag-propagation rules*** We define metadata computations in terms of *rules* of the form
$opcode : (PC, CI, OP1, OP2, MR) \Rightarrow (PC_{new}, R)$,
which should be read: "If the current opcode is *opcode*, the current tag on the program counter is $PC$, the tag on the current instruction is $CI$, the tags on its input operands (if any) are $OP1$ and $OP2$, and the tag on the memory location (in case of load/store) is $MR$, then the tag on the program counter in the next machine state should be $PC_{new}$ and the tag on the instruction's result (a destination register or a memory location, if any) should be $R$". Instead of $(PC_{new}, R)$, the function can also fail, indicating that the operation is not allowed. The PUMP can enforce any pol-

| Policy | Threat | Metadata | Max Unique Tags | Tag Check (allow?) | Tag Propagation Rules | Ref. |
|---|---|---|---|---|---|---|
| **NXD+NWC** | code injection | DATA or CODE on memory locations | 2 | $CI$=CODE; $MR \neq$ CODE on write | no | |
| **Memory Safety** | spatial/temporal memory safety violation on heap | color on pointers; region color + payload color on memory locations | (#mallocs)$^2$ | pointer color == referenced region color | $R \leftarrow OPi$ on mov/add/sub; $R \leftarrow payload(MR)$ on load | [17] |
| **CFI** | control-flow hijacking (JOP/ROP/code reuse) | unique id on each indirect control-flow source and target | #sources+#targets | $(PC, CI) \in$ program's control-flow graph | $PC \leftarrow CI$ on indirect jumps (including call, return) | [3] |
| **Taint Tracking** | untrusted code, low-integrity data from IO | source-taint-id on instructions and IO; set of input taint-ids on words | $2^{(\#code+\#IO\ ids)}$ | user-defined check | $R \leftarrow CI \cup OP1 \cup OP2 \cup MR$ | [49] |
| **Composite** | all of the above | structure with 4 elements | product of above | all of the above (Alg. 2 with $N = 4$) | | |

Table 1: Summary of Investigated Policies (simplified for brevity; details not critical to understand architecture and optimizations)

| Tag Bits | Propagate? | Outputs: allow? | Outputs: $R$ (result) | Outputs: $PC$ | Inputs: $PC$ | Inputs: $CI$ | Inputs: $OP1$ | Inputs: $OP2$ | Inputs: $MR$ | Usage (Example) |
|---|---|---|---|---|---|---|---|---|---|---|
| 2 | ✗ | soft | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | memory protection (Mondrian [67]) |
| word | ✗ | limited prog. | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | memory hygiene, stack, isolation (SECTAG [5]) |
| 32 | ✗ | limited prog. | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | unforgeable data, isolation (Loki [71]) |
| 2 | ✗ | fixed | fixed | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | fine-grained synchronization (HEP [61]) |
| 1 | ✓ | fixed | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | capabilities (IBM System/38 [34], Cheri [68]) |
| 2–8 | ✓ | fixed | fixed | ✗ | ✗ | ✗ | ✓ | ✓ | ✗ | types (Burroughs B5000, B6500/7500 [51], LISP Machine [44], SPUR [64]) |
| 128 | ✓ | fixed | copy | ✗ | ✗ | ✗ | ✓ | ✗ | ✓ | memory safety (HardBound [25], Watchdog [46, 47]) |
| 0 | ✓ | software defined | | ✗ | propagate only one | | | | | invariant checking (LBA [14]) |
| 1 | ✓ | fixed | fixed | ✗ | ✗ | ✗ | ✓ | ✓ | ✓ | taint (DIFT [63], [15], Minos [19]) |
| 4 | ✓ | limited programmability | | ✗ | ✗ | ✗ | ✓ | ✓ | ✗ | taint, interposition, fault isolation (Raksha [22]) |
| 10 | ✓ | limited prog. | fixed | ✗ | ✗ | ✗ | ✓ | ✓ | ✓ | taint, isolation (DataSafe [16]) |
| unspec. | ✓ | software defined | | ✗ | ✗ | ✗ | ✓ | ✓ | ✓ | flexible taint (FlexiTaint [66]) |
| 32 | ✓ | software defined | | ✗ | ✗ | ✗ | ✓ | ✓ | ✓ | programmable, taint, memory checking, reference counting (Harmoni [24]) |
| 0–64 | ✓ | software defined | | | ✓ | ✓ | ✓ | ✓ | ✓ | information flow, types (Aries [11]) |
| Unbounded | ✓ | software defined | | | ✓ | ✓ | ✓ | ✓ | ✓ | fully programmable, pointer-sized tags (PUMP) |

Table 2: Taxonomy of Tagging Schemes (Propagate = tag propagates with data (✓) or is a property of memory address (✗); allow? = logic for allowing / disallowing operations; $R$ = policy impacts tag of result; $PC$ = programmable tag on program counter; $CI$ = is current instruction tag checked or propagated; $OP1$, $OP2$, $MR$ = is tag on these inputs checked or propagated)

icy that can be expressed as a function of this form. The mapping should be "purely functional"; i.e., it should neither depend on nor modify any mutable state. The policy function is defined and resolved entirely in software (see App. A [27] for an example). This rule format, allowing two output tags to be computed from up to five input tags, is markedly more flexible than those considered in prior work (see Tab. 2), which typically compute one output from up to two inputs. It should also generalize easily across most RISC architectures. Beyond previous solutions that only track data tags ($OP1$, $OP2$, $MR$, $R$), we add a current instruction tag ($CI$) that can be used to track and enforce provenance, integrity, and usage of code blocks; as well as a $PC$ tag that can be used to record execution history [4], ambient authority [43], and "control state" including implicit information flows [39]. The CFI policy (Tab. 1) exploits the $PC$ tag for recording the sources of indirect jumps and the $CI$ tag for identifying jump targets, NXD+NWC leverages the $CI$ to enforce that data is not executable, and Taint Tracking uses the $CI$ to taint data based on the code that produced it.

***Rule Cache*** The other main feature of the PUMP is hardware support for single-cycle common-case computation on metadata. To resolve the rules in a single cycle in the common case, we provide a *hardware cache* of the most recently used rules; as long as we hit in this cache, we do not add any extra cycles. Fig. 1 shows the microarchitecture of our current rule cache implementation. Specifically, the rule cache does nothing more than perform an associative mapping between the instruction opcode and the 5 input tags and the two output tags. Since rules are purely functional, the rule cache can directly map between pointer tag inputs and pointer tag outputs without dereferencing the pointers or examining the metadata structures they point to. Failure cases are never inserted into the PUMP rule cache since they must transfer control to software cleanup.

While the PUMP can, in principle, implement any policy function, the rule cache can only effectively accelerate policy functions that generate small working sets and hence can be cached well, including: (i) policies that use only a limited number of distinct tags, and hence rules, and (ii) policies where the potential universe of tags is large but the set of tags actually encountered is still modest (e.g., library and file taint-tracking with up to 34 unique taints could see $2^{34}$ distinct sets in principle, but only about 5–10,000 are seen in practice). Good performance depends upon temporal locality, just as good performance in the instruction and data caches also depend on temporal locality. As with data caches and virtual memory, the PUMP allows the policy programmer to express the policy without being concerned with arbitrary hard limits on the number of tags or rules. If the policies do have (or mostly do have) small tag and rule sets, the policy will perform well. If the policies occasionally need to use more tags or rules, performance degrades only in proportion to the cases where many rules are required. If the policy

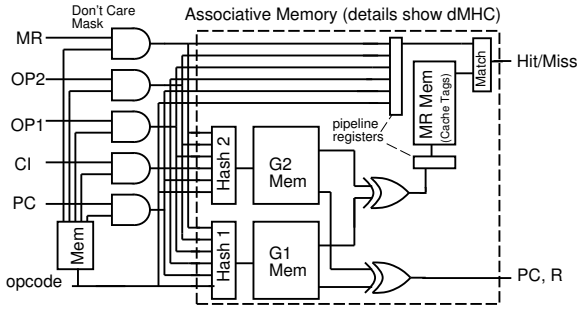



Figure 1: PUMP Rule Cache Dataflow and Microarchitecture

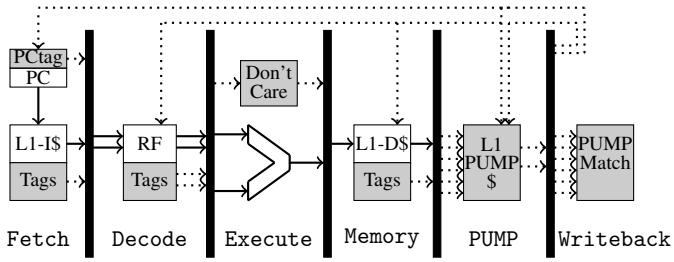



Figure 2: L1 PUMP Rule Cache in a Processor Pipeline

writer develops a policy with poor locality, the rule cache will thrash, just as a program with poor locality will thrash virtual memory and instruction and data caches. The experimental data in this paper shows that the PUMP rule caches can be engineered to efficiently support four quite different example policies, and their composition.

Depending on the instruction and policy, one or more of the input slots in a rule may be unused. For example, a register-to-register `add` will never depend on the value of the unused MR tag. To avoid polluting the cache with rules for all possible values of the unused slots, the rule-cache lookup logic refers to a bit vector containing a *don't-care bit* for each input slot–opcode pair (shown in the `Execute` stage in Fig. 2), which determines whether the corresponding tag is actually used in the rule cache lookup. To handle these "don't care" inputs efficiently, we mask them out before presenting the inputs to the PUMP rule cache (`AND` gates in Fig. 1). The don't-care bit vectors are set by a privileged instruction as part of the miss handler installation. The implementation details of the rule cache are presented in §3.

***Pipeline Integration*** Fig. 2 shows how we revise the 5-stage processor pipeline to incorporate the PUMP hardware. We add the rule cache lookup as an additional stage and bypass tag and data independently so that the `PUMP` stage does not create additional stalls in the processor pipeline.

Placing the PUMP as a separate stage is motivated by the need to provide the tag on the word read from memory (`load`), or to be overwritten in memory (`store`), as an input to the PUMP. Since we allow rules that depend on the existing tag of the memory location that is being written, write operations become `read-modify-write` operations. The existing tag is read during the `Memory` stage like a read, the rule is checked in the `PUMP` stage, and the write is performed during the `Commit` stage. As with any caching scheme, we can use multiple levels of caches for the PUMP; in this paper, we use two.

**Algorithm 1** Taint Tracking Miss Handler (Fragment)

1: **switch** $(op)$
2: **case** `add`, `sub`, `or`**:**
3: $PC_{new} \leftarrow PC$
4: $R \leftarrow$ canonicalize$(CI \cup OP1 \cup OP2)$
5: *(... cases for other instructions omitted ...)*
6: **default:** trap to error handler

***Miss Handler*** When a last-level miss occurs in the rule cache, it is handled as follows: (i) the current opcode and tags are saved in a (new) set of processor registers used only for this purpose and (ii) control is transferred to the policy miss handler (described in more detail below), which (iii) invokes the policy function to decide if the operation is allowed and, if so, generates an appropriate rule. When the miss handler returns, the hardware (iv) installs this rule into the rule caches, and (v) re-issues the faulting instruction. To provide isolation between the privileged miss handler and the rest of the system software and user code, we add a *miss-handler operational mode* to the processor, controlled by a bit in the processor state that is set on a last-level rule cache miss and reset when the miss handler returns. To avoid the need to save and restore registers on every miss handler invocation, we expand the integer register file with 16 additional registers that are available only to the miss handler. Additionally, the rule inputs and outputs appear as registers while in miss handler mode (*cf.* register windows [52]), allowing the miss handler (but nothing else) to manipulate the tags as ordinary values.

We add a new `miss-handler-return` instruction to finish installing the rule into the PUMP rule caches and return to user code; this instruction can only be issued when in miss-handler mode. While in miss-handler mode, the rule cache is ignored and the PUMP instead applies a single, hardwired rule: all instructions and data touched by the miss handler must be tagged with a predefined MISSHANDLER tag, and all instruction results are given the same tag. In this way, the PUMP architecture prevents user code from undermining the protection provided by the policy. User code cannot: (1) divide, address or replace tags; (2) touch metadata data structures and miss handler code; and (3) directly insert rules into the rule cache.

Alg. 1 illustrates the operation of the miss handler for our Taint-Tracking policy. To minimize the number of distinct tags (and hence rules), the miss handler takes care to use a single tag for logically equivalent metadata by "canonicalizing" any new data structures that it builds. While it would be functionally correct to create a new data structure, with a new metadata pointer, for the set resulting from the unions

**Algorithm 2** N-Policy Miss Handler

1: **for** i=1 to N **do**
2: $M_i \leftarrow \{op, PC[i], CI[i], OP1[i], OP2[i], MR[i]\}$
3: $\{pc_i, res_i\} \leftarrow \text{policy}_i\ (M_i)$
4: $PC_{new} \leftarrow \text{canonicalize}([pc_1, pc_2, ..., pc_N])$
5: $R \leftarrow \text{canonicalize}([res_1, res_2, ..., res_N])$

in Line 4, it is important for performance to make sure that equivalent sets are assigned the same tag—i.e., the same canonical name—to increase reuse and sharing of rules in the rule cache, thereby increasing its effectiveness.

Rather than forcing users to choose a single policy, we want to enforce multiple policies simultaneously and add new ones later. An advantage of our "unbounded" tags is that we can in principle enforce any number of policies at the same time. This can be achieved by letting tags be *pointers to tuples* of tags from several component policies. For example, to combine the NXD+NWC policy with the taint-tracking policy, we can let each tag be a pointer to a tuple $(s, t)$, where $s$ is a NXD+NWC tag (either DATA or CODE) and $t$ is a taint tag (a pointer to a set of taints). The rule cache lookup is exactly the same, but when a miss occurs, both component policies are evaluated separately: the operation is allowed only if both policies allow it, and the resulting tags are pairs of results from the two component policies. Alg. 2 shows the general behavior of the composite miss handler for any $N$ policies. Depending on how correlated the tags in the tuple are, this could result in a large increase in the number of tags and hence rules. In order to demonstrate the ability to support multiple policies simultaneously and measure its effect on working set sizes, we implement the composite policy ("Composite") comprising all four policies described in §1. The Composite policy represents the kind of policy workloads we would like to support; hence we use it for most of the experiments described later.

Most policies will dispatch on the opcode to select the appropriate logic. Some, like NXD+NWC, will just check whether the operation is allowed. Others may consult a data structure (e.g., the CFI policy consults the graph of allowed indirect call and return ids). Memory safety checks equality between address and memory region colors. Taint tracking computes fresh result tags by combining the input tags (Alg. 1). Policies that must access large data structures (CFI) or canonicalize across large aggregates (Taint Tracking, Composite) may make many memory accesses that could miss in the on-chip caches and go to DRAM. On average across all of our benchmarks, servicing misses for NXD+NWC required 30 cycles, Memory Safety 60, CFI 85, Taint Tracking 500, and Composite 800.

If the policy miss handler determines that the operation is *not* allowed, it invokes a suitable security fault handler. What this fault handler does is up to the runtime system and the policy; typically, it would shut down the offending process, but in some cases it might return a suitable "safe value" instead [35, 54]. For incremental deployment with UNIX-style [55] operating systems, we assume policies are applied per process, allowing each process to get a different set of policies. It also allows us to place the tags, rules, and miss handling support into the address space of the process; together with the *miss-handler operational-mode*, this allows fast transitions to and from the miss-handler software without the software bookkeeping, data copying, or management of address translation tables that would be necessary for an OS-level context switch. Longer term, perhaps PUMP policies can be used to protect the OS as well.

## 3. Evaluation of a Simple Implementation

We next describe our evaluation methodology for measuring runtime, energy, area, and power overheads and apply it on a simple implementation of the PUMP hardware and software, using 128b words (64b payload and 64b "integrated" tag [38]) and the modified pipeline sketched in Fig. 2. Although the optimized implementation of §4 is the one whose overheads (relative to the baseline processor) we really care about, it is useful to describe and measure the simple PUMP implementation first, both because it gives us a chance to show basic versions of the key mechanisms before getting to more sophisticated versions and because it allows us to confirm the natural intuition that the simple implementation does not perform very well.

***Resource Estimates*** To estimate the physical resource impact of the PUMP, we focus on memory costs, since the memories are the dominant area and energy consumers in a simple RISC processor [21] and in the PUMP hardware extensions. We consider a 32 nm Low Operating Power (LOP) process for the L1 memories and Low Standby Power (LSTP) for the L2 memories and use CACTI 6.5 [45] for modeling the area, access time, energy per access, and static (leakage) power of the main memory and the processor on-chip memories.

*Baseline Processor:* Our (no-PUMP) baseline processor has separate 64KB L1 caches for data and instructions and a unified 512KB L2 cache. We use delay-optimized L1 caches and an energy-optimized L2 cache. All caches use a write-back discipline. The baseline L1 cache has a latency around 880 ps; we assume that it can return a result in one cycle and set its clock to 1 ns, giving us a 1 GHz-cycle target—comparable to modern embedded and cell phone processors. Tab. 3 shows the parameters for this processor.

*PUMP Implementation:* The PUMP hardware implementation has two parts: extending all memories with integrated tags, and adding PUMP rule caches to the processor. Extending each 64b word in the on-chip memories with a 64b tag increases their area and energy per access and worsens their access latency. This is potentially tolerable for the L2 cache, which already has a multi-cycle access latency and is not used every cycle. But adding an extra cycle of latency to ac-

| Unit | Design | Organization | Area ($mm^2$) | Access Energy Read/Write (pJ) | Static Power (pJ/cyc) | Latency (ps) | Cyc |
|---|---|---|---|---|---|---|---|
| Register File (Int. and FP) | Baseline | 64b, 2R1W, {32 Integer, 32 Floating} | 0.002 | 0.3/0.5 | 0.080 | 264 | 1 |
| | 64b tag extended | 128b, 2R1W, {48 Integer, 32 Floating} | 0.007 | 1.0/1.4 | 0.23 | 295 | 1 |
| L1-$ (I and D) | Baseline | 64KB, 4-way, 64B/line | 0.236 | 17/11 | 14.4 | 880 | 1 |
| | 64b tag extended | 64KB, 4-way, 128B/line (eff. 32KB, 64B/line) | 0.244 | 19/14 | 14.5 | 880 | 1 |
| L2-$ (unified) | Baseline | 512KB, 8-way, 64B/line | 1.207 | 393/481 | 0.111 | 4000 | 5 |
| | 64b tag extended | 1MB, 8-way, 128B/line (eff. 512KB, 64B/line) | 2.350 | 758/1223 | 0.214 | 4930 | 5 |
| TLB (I and D) | Either | 1KB, 2-way set-assoc. | 0.040 | 3.6/4.5 | 2.0 | 800 | 1 |
| DRAM | Baseline | 1GB, access 64B line per transfer | - | 15,000 | - | - | 100 |
| | 64b tag extended | 1GB, access 128B line (eff. 64B line per transfer) | - | 31,000 | - | - | 130 |
| L1 PUMP-$ | 64b tag | fully associative 1024 entry, 328b match, 128b out (not used; shown only for reference) | 1.500 | 750/900 | | 3000 | 4 |
| | | 1024 entry, 328b match, 128b out `Fast-Value dMHC(4,2)` | 0.683 | 51/62 | 32.0 | 500 | 1 |
| L2 PUMP-$ | 64b tag | 4096 entry, 328b match, 128b out `2-level dMHC(4,2)` | 0.994 | 173/444 | 0.085 | 3300 | 4 |
| | | **Total Baseline Area** | **1.485mm$^2$** | | | | |
| | | **Total 64b-tagged Area** | **4.318mm$^2$ (+190% over baseline)** | | | | |

Table 3: Memory Resource Estimates for the Baseline and Simple PUMP-extended Processors at 32nm node

cess the L1 caches can lead to stalls in the pipeline. To avoid this, in this simple implementation we reduce the effective capacity of our L1 caches to half of those in the baseline design and then add tags; this gives the same single-cycle access to the L1 caches, but can degrade performance due to increased misses.

The PUMP rule caches require a long match key (5 pointer-sized tags plus an instruction opcode, or 328b) compared to a traditional cache address key (less than the address width), and they return a 128b result. Using a fully associative L1 rule cache would lead to high energy and delay (Tab. 3). Instead, we use a four-hash-function instance of a dynamic Multi-Hash Cache (dMHC) [26], a near-associative multi-hash cache scheme. The L1 rule cache, illustrated concretely with a two-hash function instance in Fig. 1, is designed to produce a result in a single cycle, checking for a false hit (MR Mem lookup and comparison) in the second cycle, while the L2 rule cache is designed for low energy, giving a multi-cycle access latency. Entries are added to the dMHC in such a way that the `XOR` of the values stored in the hashed memory locations (G1 and G2 in Fig. 1) produce the desired output. As long as one of the hashed table entries is empty, a new rule mapping can be added without evicting or displacing existing rule mappings. Tab. 3 shows the parameters for 1024-entry L1 and 4096-entry L2 rule caches used in the simple implementation. When these caches reach capacity, we use a simple FIFO replacement policy, which appears to work well in practice for our current workloads (FIFO is within 6% of LRU here).

***Evaluation Methodology*** To estimate the performance impact of the PUMP, we use a combination of ISA, PUMP, and address-trace simulators (Fig. 3). We use the `gem5` simulator [9] to generate instruction traces for the SPEC CPU2006 [33] programs on a 64-bit Alpha baseline ISA (omitting `xalancbmk` and `tonto`, on which `gem5` fails). We simulate each program for each of the four policies listed in §1 and the Composite policy for a warm-up period of 1B instructions and then evaluate the next 500M instructions. In `gem5`, each benchmark is run on the baseline processor

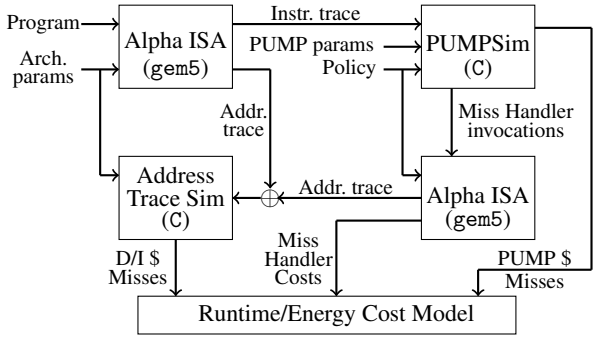



Figure 3: PUMP Evaluation Framework

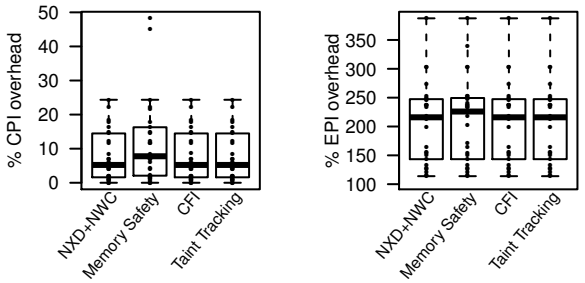
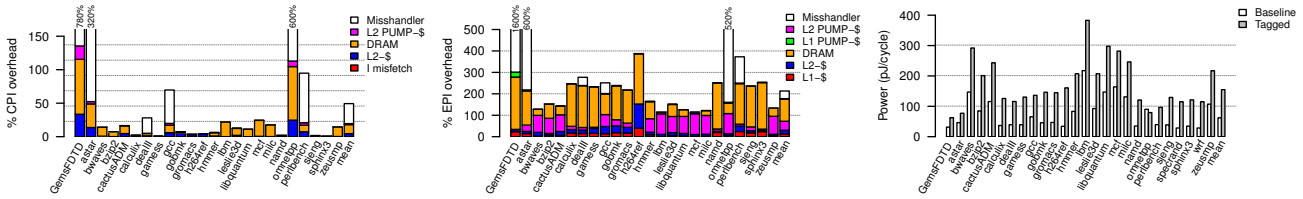



Figure 4: Single Policies with Simple Implementation

with no tags or policies. The resulting instruction trace is then run through a PUMP simulator that performs metadata computation for each instruction. This "phased" simulation strategy is accurate for fail-stop policies (the only ones we consider), where the PUMP's results cannot cause a program's control flow to diverge from its baseline execution. While address-trace simulations can be inaccurate for highly pipelined and out-of-order processors, they are quite accurate for our simple, in-order, 5- and 6-stage pipeline. On the baseline configuration, the `gem5` instruction simulation and address trace generation followed by our custom address-trace simulations and accounting were within 1.2% of `gem5`'s cycle-accurate simulations.

We provide the PUMP simulator with miss-handler code (written in C) to implement each policy, and we assign meta-




(a) Composite Policy Runtime (b) Composite Policy Energy (c) Absolute Power

Figure 5: Overhead of Simple Implementation with 64b Tags Compared to Baseline

data tags on the initial memory depending on the policy. Our PUMP simulator allows us to capture the access patterns in the PUMP rule caches and estimate the associated runtime and energy costs, accounting for the longer wait cycles required to access the L2 rule cache. Since the PUMP miss handler code also runs on the processor, we separately simulate our miss handler on `gem5` to capture its dynamic behavior. Since the miss-handler code potentially impacts the data and instruction caches, we create a merged address trace that includes properly interleaved memory accesses from both user and miss-handler code, which we use for the final address-trace simulation to estimate the performance impact of the memory system.

***Tag and Rule Usage*** One of the contributions of the PUMP model is support for unbounded metadata. The Composite policy applied to the benchmark set exercises this support. The Composite policy uses from 400 (`specrand`) to 2M (`GemsFDTD`) tags, requiring 1700 (`specrand`) to 14M (`GemsFDTD`) rules. (The large number of tags and rules in `GemsFDTD` are driven largerly by the memory safety policy, which itself uses 640K memory component tags.) The composite tags are pointers that point to metadata structures for the component policies. The Taint Tracking policy tag component is itself a set represented as an ordered list. The largest single metadata structure is 176B. The total metadata structure memory ranges from 1KB (`specrand`) to 56MB (`GemsFDTD`). As a fraction of the maximum heap allocated memory, this represents only 1–10%.

***Simple Implementation*** We evaluate the simple PUMP implementation, comparing it to the no-PUMP baseline.

*Area Overhead:* The overall area overhead of the PUMP on top of the baseline processor is 190% (Tab. 3). The dominant portion of this area overhead (110%) comes from the PUMP rule caches. The unified L2 cache contributes most of the remaining area overhead. The L1 D/I caches stay roughly the same, since we halved their effective capacity. This high memory area overhead roughly triples the static power, contributing to 24% of the energy overhead.

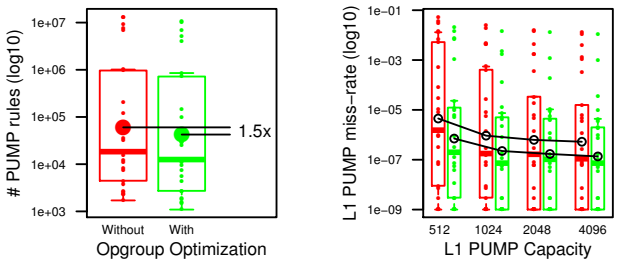
*Runtime Overhead:* For all single policies on most benchmarks, the average runtime overhead of even this simple implementation is only 10% (see Fig. 4; to read boxplots [12]: bar is the median, box covers one quartile above and below (middle 50% of cases), dots represent each individual data point, whiskers denote full range except for outliers (more than 1.5× respective quartile)), with the dominant overhead coming from the additional DRAM traffic required to transfer tag bits to and from the processor. For the Memory Safety policy, we see a few benchmarks that exhibit high miss handler overhead, pushing their total overhead up to 40-50% due to compulsory misses on newly allocated memory blocks. For the Composite policy, five of the benchmarks suffer from very high overheads in the miss handler (Fig. 5a), with the worst case close to 780% and the geomean reaching 50%. Two factors contribute to this overhead: (1) the large number of cycles required to resolve a last-level rule cache miss (since every component miss handler must be consulted), and (2) an explosion in the number of rules, which expands the working set size and increases the rule cache miss rate. In the worst case, the number of unique composite tags could be the product of the unique tags in each component policy. However, we do not see this worst case—rather, total rules increase by a factor of 3–5 over the largest single policy, Memory Safety.

*Energy Overhead:* Moving more bits, due to wider words, and executing more instructions, due to miss handler code, both contribute to energy overheads, impacting both the single and composite policies (Figs. 4 and 5b). The CFI and Memory Safety policies—and hence also the Composite policy—access large data structures that often require energy-expensive DRAM accesses. The worst-case energy overhead is close to 400% for single policies, and about 1600% for the Composite policy, with geomean overhead around 220%.

*Power Ceiling:* For many platform designs the worst-case power, or equivalently, energy per cycle, is the limiter. This power ceiling may be driven by the maximum current they can draw from a battery or the maximum sustained operating temperature either in a mobile or in a wired device with ambient cooling. Fig. 5c shows that the simple implementation raises the maximum power ceiling by 76% with `lbm` driving the maximum power in both the baseline and simple PUMP implementations. Note that this power ceiling increase is lower than the worst-case energy overhead in part because some benchmarks slow down more than the extra energy they consume and in part because the benchmarks with high energy overhead are the ones consuming the least absolute energy per cycle in the baseline design. Typically the data working set of these energy-efficient programs fits

| Unit | Parameter | Range | Final |
|---|---|---|---|
| L1 PUMP-$ | Capacity | 512–4096 | 1024 |
| | Tag Bits | 8–12 | 10 |
| L2 PUMP-$ | Capacity | 1024–8192 | 4096 |
| | Tag Bits | 13–16 | 14 |
| UCP-$ | Capacity | 512–4096 | 2048 |
| CTAG-$ | Capacity | 128–1024 | 512 |

Table 4: PUMP Parameter Range Table




(a) Number of PUMP Rules (b) L1 PUMP-$ Miss-Rate

Figure 6: Impact of Opgroup Optimization

into the on-chip caches, so they seldom pay the higher cost of DRAM accesses.

## 4. Optimizing the PUMP

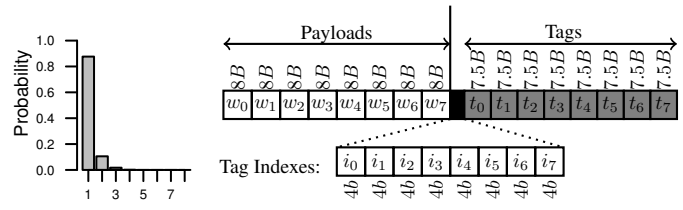
Though the simple implementation described above achieves reasonable performance on most benchmarks, the runtime overhead for the Composite policy on some of them and the energy and power overheads on all policies and benchmarks seem unacceptably high. To address these overheads, we introduce a series of targeted microarchitecture optimizations and examine the impact of the architectural parameters associated with the PUMP components (Tab. 4) on the overall costs. We use groupings of opcodes with identical rules to increase the effective capacity of the rule caches, tag compression to reduce the delay and energy of DRAM transfers, short tags to reduce the area and energy in on-chip memories, and Unified Component Policy and Composition Tag caches to decrease the overheads in the miss handlers.

***Opgroups*** In practical policies, it is common to define similar rules for several opcodes. For example, in the Taint Tracking policy, the rules for the `add` and `sub` instructions are identical (Alg. 1). However, in the simple implementation, these rules occupy separate entries in the rule caches. Consequently, we group instruction opcodes with the same rules into "opgroups", reducing the number of rules needed. Which opcodes can be grouped together depends on the policy; we therefore expand the don't-care SRAM in the `Execute` stage (Fig. 2, also shown as "Mem" in Fig. 1) to also translate opcodes to opgroups before the rule cache lookup. For the Composite policy, we can reduce 300+ Alpha opcodes to 14 opgroups and the total number of rules by a factor of 1.1×–6×, with an average of 1.5× (Fig. 6a measures this effect across all the benchmarks). This effectively increases the rule cache capacity for a given investment in silicon area. Opgroups also reduce the number of compulsory misses, since a miss on a single instruction in the group installs the rule that applies to every instruction opcode in the group. Fig. 6b summarizes the miss-rate across all the benchmarks for different L1 rule cache sizes for the Composite policy with and without opgrouping. This figure shows that both the range and the mean of the miss-rates are reduced by opgrouping. With opgroup optimization, a 1024-entry rule cache has a lower miss rate than a 4096-entry rule cache without it. A lower miss-rate naturally reduces the time and energy spent in miss handlers (Fig. 14), and smaller rule caches directly reduce area and energy.

***Main Memory Tag Compression*** Using 64b tags on 64b words doubles the off-chip memory traffic and hence roughly doubles the associated energy. Typically, though, tags exhibit spatial locality—many adjacent words have the same tag. For example, Fig. 7a plots the distribution of unique tags for each DRAM transfer for the `gcc` benchmark with the Composite policy, showing that most words have the same tag: on average we see only 1.14 unique tags per DRAM transfer of an 8-word cache line. We exploit this spatial locality to compress the tag bits that must be transferred to and from the off-chip memory. Since we are already transferring data in cache lines, we use cache lines as the basis for this compression. We allocate 128B per cache line in the main memory, to keep addressing simple. However, rather than storing 128b tagged words directly, we store eight 64b words (payloads) followed by eight 4b indexes and then up to eight 60b tags (see Fig. 7b). The index identifies which of the 60b tags goes with the associated word. We trim the tag to 60b to accommodate the indexes, but this does not compromise the use of tags as pointers: assuming byte addressing and 16B (two 64b words) aligned metadata structures, the low 4b of the 64b pointer can be filled in as zeros. As a result, after transferring the 4B of indexes, we only need to transfer the unique 7.5B tags in the cache line. For instance, if the same tag is used by all the words in the cache line, we transfer 64B+4B=68B in a first read, then 8B in a second read for a total of 76B instead of 128B. The second read is to the same DRAM page, needing only CAS cycles and not the full random-access cycle time. The 4b index can be either a direct index or a special value. We define a special index value to represent a default tag, so that there is no need to transfer any tag in this case. By compressing tags in this manner we are able to reduce the average energy overhead per DRAM transfer from 110% to 15%.

We chose this compression scheme for its combination of simplicity and effectiveness at reducing off-chip memory energy. Many clever schemes for fine-grained memory tagging exist—including multi-level tag page tables [60], variable-grained TLB-like structures [67, 71], and range caches [65]—and these could also be used to reduce the DRAM footprint.

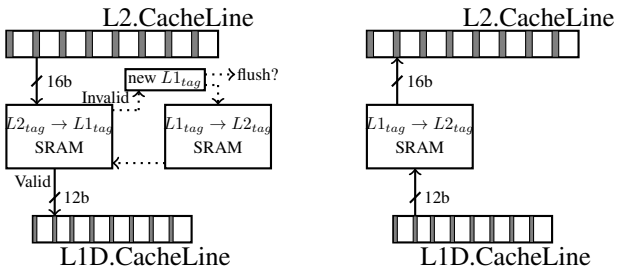
***Tag Translation*** The simple PUMP rule caches are large (adding 110% area) since each cached rule is 456b wide.




(a) Unique tags per DRAM transfer for `gcc`

(b) Cache line compression in DRAM

Figure 7: Main Memory Tag Compression




(a) L1 miss, L2 hit

(b) L1 to L2 writeback

Figure 8: Translation Between 12b L1 Tags and 16b L2 Tags

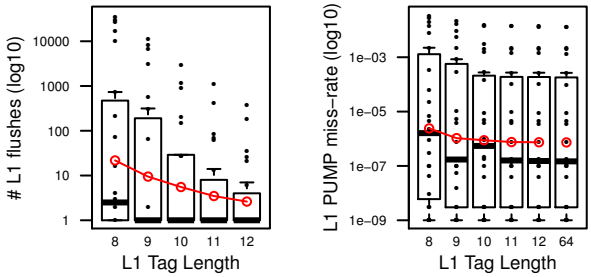



Figure 9: Impact of L1 Tag Length on L1 PUMP Miss-Rates

Supporting the PUMP also required extending the baseline on-chip memories (RFs and L1/L2 caches) with 64b tags. Using a full 64b (or 60b) tag for each 64b word here incurs heavy area and energy overheads, as we saw in §3. However, a 64KB L1 D-$ holds only 8192 words and hence at most 8192 unique tags. Along with a 64KB L1 I-$, we can have at most 16384 unique tags in the L1 memory subsystem; these can be represented with just 14b tags, reducing the delay, area, energy, and power in the system. Caches exist to exploit temporal locality, and this observation suggests we can leverage that locality to reduce area and energy. If we reduce the tag bits to 14b, the PUMP rule cache match key is reduced from 328b to 78b.

To get these savings without losing the flexibility of full, pointer-sized tags, we use different-width tags for different on-chip memory subsystems and translate between these as needed. For example, one might use 12b tags in the L1 memories and 16b tags in the L2 memories. Fig. 8 shows how we manage tag translation between L1 and L2 memory subsystems. Moving a word from L2 cache to L1 cache requires translating its 16b tag to the corresponding 12b tag, creating a new association if needed. We use a simple SRAM for the L2-tag-to-L1-tag translation, with an extra bit indicating whether or not there is an L1 mapping for the L2 tag. Translating an L1 tag to L2 tag (on a write-back or an L2 lookup) is again performed with an SRAM lookup using the L1 tag as the address. A similar translation occurs between the 60b main memory tags and 16b L2 tags.

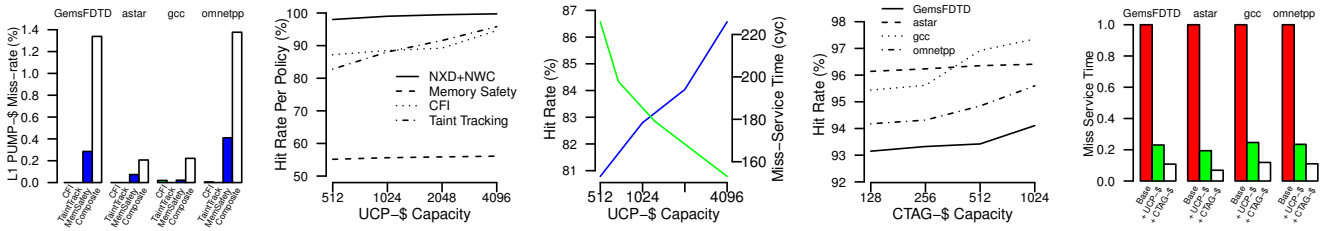
When a long tag is not in the long-to-short translation table, we must allocate a new short tag, potentially reclaiming a previously allocated short tag that is no longer in use. There is a rich design space to explore for determining when a short tag can be reclaimed, including garbage collection and tag-usage counting. For simplicity, we allocate short tags sequentially and flush all caches above a given level (instruction, data, PUMP) when the short tag space is exhausted, avoiding the need to track when a specific short tag is available for reclamation. We assume all caches are designed with a lightweight gang clear [41], making flushes inexpensive.

Compared to Tab. 3, where each L1 rule cache access costs 51pJ, we get down to 10pJ with 8b L1 tags or 18pJ with 16b L1 tags, with the energy scaling linearly with tag length between these points. The energy impact on the L1 instruction and data caches is small. Similarly, with 16b L2 tags, L2 rule cache access costs 120pJ, down from 173pJ with 64b tags.

Slimming L1 tags also allows us to restore the capacity of the L1 D/I caches. With 12b tags, the full-capacity (76KB, effective 64KB) L1 cache will meet single-cycle timing requirements, reducing the performance penalty the simple implementation incurred from the reduced cache capacity. As a result, we limit L1 tag length exploration to 12 bits or less. While even shorter tags reduce energy, they also increase the frequency of flushes. Fig. 9 shows how flushes decrease with increasing L1 tag length, as well as the impact on the L1 rule cache miss-rate.

***Miss-Handler Acceleration*** Enforcing large policies obtained by the orthogonal combination of smaller policies is, not surprisingly, expensive. We illustrated this by combining *four* policies into a single Composite policy. As shown in Alg. 2, each invocation of a $N$-policy miss handler must take apart a tuple of tags, compute result tags for each component policy, reassemble them into tuples, and canonicalize these tuples to prevent duplication. Moreover, the greater number of tags and rules needed for the Composite policy increases the rule cache miss rates—in Fig. 10a, even though the Taint Tracking and CFI policies individually have a low miss-rate, a higher miss-rate from the Memory Safety policy drives the miss-rate for the Composite policy high as well. The lower miss rates of the individual policies suggest that their results may be cacheable even when the composite rules are not.

*Microarchitecture:* We introduce two hardware structures to optimize composite policy miss handlers. First we add a Unified Component Policy (UCP) cache where we simply cache the most recent component policy results. The general miss-handler for composite policies is modified to perform lookups in this cache while resolving component policies




(a) Per policy miss-rates (b) UCP Cache Hit-rate (c) Miss Service Performance (d) Miss-Handler CTAG-$ (e) Miss Handler Cost

Figure 10: Optimizing Composite Policy Miss Handler

**Algorithm 3** N-Policy Miss Handler with HW support

```
1:  for i=1 to N do
2:      M_i ←{op, PC[i], CI[i], OP1[i], OP2[i], MR[i]}
3:      {hit, pc_i, res_i} ← UCP-$ (i, M_i)
4:      if !hit then
5:          {pc_i, res_i} ← policy_i (M_i)
6:          insert {i, pc_i, res_i} into UCP-$
7:  pc[1..N] ← {pc_1, pc_2, ..., pc_N}
8:  {hit, PC_new} ← CTAG-$ (pc[1..N])
9:  if !hit then
10:     PC_new ← canonicalize(pc[1..N])
11:     insert {pc[1..N], PC_new} into CTAG-$
12: res[1..N] ← {res_1, res_2, ..., res_N}
13: {hit, R} ← CTAG-$ (res[1..N])
14: if !hit then
15:     R ← canonicalize(res[1..N])
16:     insert {res[1..N], R} into CTAG-$
```

(Alg. 3, line 3). When this cache misses for a component policy we perform its policy computation in software (and insert the result in this cache). We implement the UCP cache with the same hardware organization as the regular PUMP rule cache, with an additional policy identifier field. For simplicity we use a FIFO replacement policy for this cache, but it may be possible to achieve better results by prioritizing space using a metric such as the re-computation cost for the component policies. With modest capacity, this cache filters out most policy recomputations (Fig. 10b; the low hit rate for memory safety is driven by compulsory misses associated with new memory allocations). As a result, we are able to reduce the average number of miss handler cycles by a factor of 5 for our most challenging benchmarks (Fig. 10e). It is possible for every policy to hit in the UCP cache when there is a miss in the L2 PUMP since the composite rules needed could be a product of a small number of component policy rules. For `GemsFDTD`, we hit in 3 or more component policies about 96% of the time.

Second, we add a cache that translates a tuple of result tags into its canonical composite result tag (Alg. 2, Line 4). This identifies the single pointer tag that points to the component result tag tuple, when such a pointer already exists; it is canonicalized so there is a single, unique tag for any distinct combination of components in order to maximize the effectiveness of the rule caches. This Composition Tag (CTAG) cache is implemented by hashing together all the tuple components to get an index into a small CTAG memory. Multiple hashes and memories are used to minimize conflicts as in the PUMP rule cache (Fig. 1). Once the CTAG cache returns a candidate composite tag, the hardware dereferences the pointer result to check that the hashed result is a valid hit. This CTAG cache is effective (Fig. 10d) because it is common for several component policy rules to return the same tuple of result tags. For example, in many cases the $PC_{tag}$ will be the same, even though the result tag is different. Furthermore, many different rule inputs can lead to the same output. For example, in Taint Tracking we perform set unions, and many different unions will have the same result; e.g., $(Blue, \{A, B, C\})$ is the composite answer for writing the result of both $\{A\} \cup \{B, C\}$ and $\{A, B\} \cup \{B, C\}$ (Taint Tracking) into a *Blue* slot (Memory Safety). We also use a FIFO replacement policy for this cache. The CTAG cache reduces the average miss handler cycles by another factor of 2 (Fig. 10e). Taken together, a 2048-entry UCP cache and a 512-entry CTAG cache reduce the average time spent on each L2 rule cache miss from 800 cycles to 80 cycles.

*Rule Prefetch:* Another way to reduce the compulsory miss rate is to pre-compute rules that might be needed in the near future. A full treatment of prefetching is beyond the scope of this paper. However, one easy case has high value for the Memory Safety rules. When we allocate a new memory tag, we will likely need the 7 rules (initialize (1), add offset to pointer and move (3), scalar load (1), scalar store (2)) for that tag in the near future. Consequently, we add all of these rules to the UCP cache at once. For the single-policy Memory Safety case, we add the rules directly into the rule caches. This reduces the number of Memory Safety miss-handler invocations by 2×.

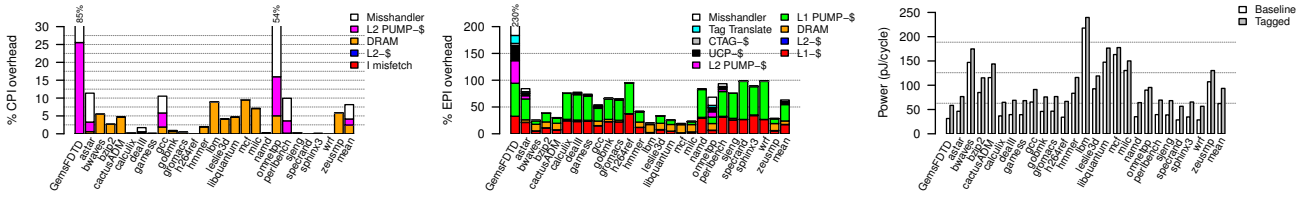
Fig. 11 shows how the optimizations discussed in this section extend the PUMP-microarchitecture.

## 5. Overall Evaluation

***Design-Point Selection*** For the most part, the architecture parameters (Tab. 4) monotonically impact a particular cost, providing tradeoffs among energy, delay, and area, but not defining a minimum within a single cost criteria. There is

| Unit | Design | Organization | Area ($mm^2$) | Access Energy Read/Write (pJ) | Static Power (pJ/cyc) | Latency (ps) | Cyc |
|---|---|---|---|---|---|---|---|
| Register File | Extended 10b | 74b, 2R1W, {48 Integer, 32 Floating} | 0.005 | 0.4/0.5 | 0.13 | 360 | 1 |
| L1 Cache | 10b-tag | 74KB, 4-way, 74B/line (eff. 64KB, 64B/line) | 0.272 | 19/13 | 16.4 | 975 | 1 |
| L2 Cache | 14b-tag | 592KB, 8-way, 78B/line (eff. 512KB, 64B/line) | 1.247 | 343/640 | 0.133 | 4600 | 5 |
| TLB | – | 1KB, 2-way set-assoc. | 0.040 | 3.6/4.5 | 2.0 | 800 | 1 |
| DRAM | 64b-tag | 1GB, access 128B line (move 76B) | | 17,500 | | | 112 |
| L1 PUMP Cache | 10b L1 tag | 1024-entry, 58b match, 20b out `Fast-Value dMHC(4,2)` | 0.095 | 15/43.2 | 2.22 | 520 | 1 |
| L2 PUMP Cache | 14b L2 tag | 4096-entry, 78b match, 28b out `2-level dMHC(4,2)` | 0.287 | 99.4/267 | 0.032 | 2800 | 3 |
| full→L2-tag | 64b→14b | 8196-entry, 64b match, 14b out `dMHC(4,2)` | 0.432 | 166/436 | 0.052 | 3400 | 4 |
| L2-tag→full | 14b→64b | 16K×64 SRAM | 0.216 | 55.5/31.5 | 0.027 | 1700 | 2 |
| L2-tag→L1-tag | 14b→10b | 16K×11 SRAM | 0.038 | 8.8/4.7 | 0.0040 | 1420 | 2 |
| L1-tag→L2-tag | 10b→14b | 1K×14 SRAM | 0.004 | 0.8/1 | 0.0004 | 780 | 1 |
| UCP Cache | 64b tags | 2048-entry, 328b match, 128b out `2-level dMHC(4,2)` | 0.377 | 196/479 | 0.035 | 2730 | 3 |
| CTAG Cache | 64b tags | 512-entry `2-level dMHC(4,2)` | 0.107 | 56/139 | 0.009 | 1700 | 2 |
| | | **Total area** | **3.120mm$^2$ (+110% over baseline)** | | | | |

Table 5: Memory Resource Estimates for the PUMP-optimized Processor at 32nm node




(a) Runtime Overhead (b) Energy Overhead (c) Absolute Power

Figure 12: Overhead of Optimized Implementation as Shown in Tab. 5 (Composite Policy)

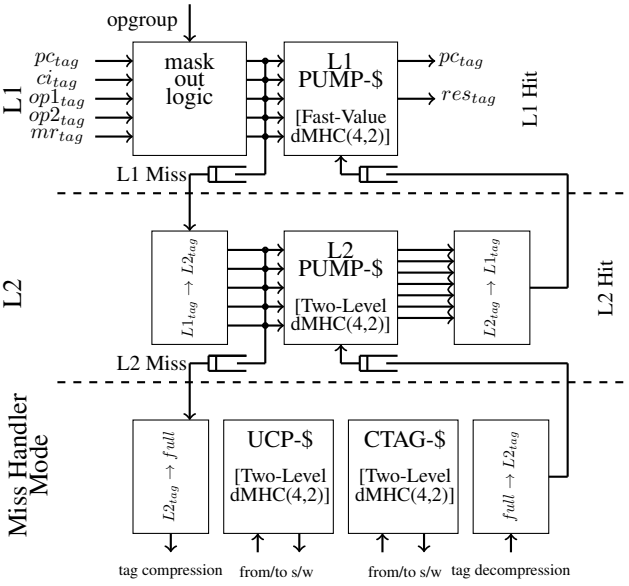
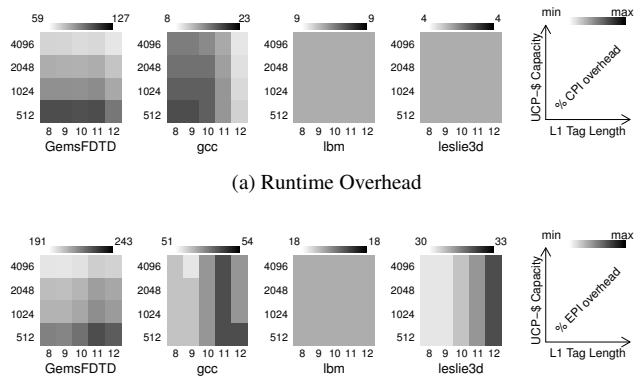



Figure 11: PUMP Microarchitecture




(a) Runtime Overhead

(b) Energy Overhead

Figure 13: Impact of Tag Bits and UCP-$ Capacity

the threshold effect that, once the tag bits are small enough, the L1 D/I caches can be restored to the capacity of the baseline, so we adopt that as our upper bound to explore for L1 tag length, but beyond that point, decreasing tag length reduces energy with small impact on performance. Fig. 13b shows that reducing tag length is the dominant energy effect for most benchmark programs (e.g., `leslie3d`, `mcf`), with a few programs showing equal or larger benefits from increasing UCP cache capacity (e.g., `GemsFDTD`, `gcc`). Ignoring other cost concerns, to reduce energy, we would select large miss handler caches and few tag bits. Runtime overhead (Fig. 13a) is also minimized with larger miss handler caches, but benefits from more rather than fewer tag bits (e.g., `GemsFDTD`, `gcc`). The magnitude of the benefits vary across benchmarks and policies. Across all benchmarks, the benefit beyond 10b L1 tags is small for the SPEC CPU2006 benchmarks, so we use 10b as the compromise between energy and delay and use a 2048-entry UCP cache and a 512-entry CTAG cache to reduce area overhead while coming close to the minimum energy level within the space of the architecture parameters explored.

***Runtime and Energy Impact of Optimizations*** Fig. 14 shows the overall impact on runtime and energy overheads of applying the optimizations from §4 in sequence. Every optimization is dominant for some benchmark (e.g., op-

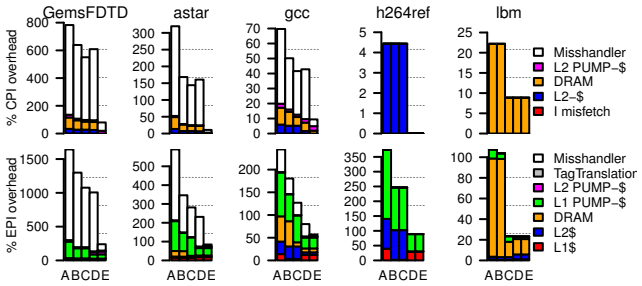



Figure 14: Runtime and Energy Impact of Optimizations on Representative Benchmarks (Composite Policy) (A: Simple; B: A+Opgrouping; C: B+DRAM Compression; D: C+(10b L1,14b L2) short tags; E: D+(2048-UCP, 512-CTAG))

groups for `astar`, DRAM tag compression for `lbm`, short tags for `h264ref`, miss handler acceleration for `GemsFDTD`), and some benchmarks see benefits from all optimizations (e.g., `gcc`), with each optimization successively removing one bottleneck and exposing the next. The different behavior from the benchmarks follows their baseline characteristics as detailed below.

Applications with low locality have baseline energy and performance driven by DRAM due to high main memory traffic. The overhead in such benchmarks (e.g., `lbm`) trends to the DRAM overhead, so reductions in DRAM overhead directly impact runtime and energy overhead. Applications with more locality are faster in the baseline configuration, consume less energy, and suffer less from DRAM overheads; as a result, these are more heavily impacted by the reduced L1 capacity and the tag energy in the L1 D/I and rule caches. DRAM optimization has less effect on these applications, but using short tags has a large effect on energy and removes the L1 D/I cache capacity penalty (e.g., `h264ref`).

The benchmarks with heavy dynamic memory allocation have higher L2 rule cache miss rates due to compulsory misses as newly created tags must be installed in the cache. This drove the high overheads for several benchmarks (`GemsFDTD, omnetpp`) in the simple implementation. The miss handler optimizations reduce the common-case cost of such misses, and the opgroup optimization reduces the capacity miss rate. For the simple implementation, `GemsFDTD` took an L2 rule cache miss every 200 instructions and took 800 cycles to service each miss driving a large part of its 780% runtime overhead. With the optimizations, it services an L2 rule cache miss every 400 instructions and takes only 140 cycles on average per miss, reducing its runtime overhead to 80%.

Overall, these optimizations bring runtime overhead below 10% for all benchmarks except `GemsFDTD` and `omnetpp` (Fig. 12a), which are high on memory allocation. The mean energy overhead is close to 60%, with only 4 benchmarks exceeding 80% (Fig. 12b).

***Power Ceiling*** The optimizations reduce the impact on power ceiling to 10% (Fig. 12c), suggesting the optimized PUMP will have little impact on the operating envelope of the platform. The programs that still have the worst energy overhead (`GemsFDTD`, `h264ref`) are well below the baseline absolute power ceiling set by `lbm`. DRAM compression reduces the energy overhead for `lbm` to 20%; since it also slows down by 9%, its power requirement increases by 10%.

***Area*** The area overhead of the optimized design is around 110% (Tab. 5), compared to the 190% of the simple design (Tab. 3). On the one hand, short tags significantly reduce the area of the L1 and L2 caches (now adding only 5% over the baseline) and of the rule caches (adding only 26%). On the other hand, the optimized design spends some area to reduce runtime and energy overhead. The UCP and CTAG caches add 33% area overhead, while the translation memories for short tags (both L1 and L2) add another 46%. While these additional hardware structures add area, they provide a net reduction in energy, since they are accessed infrequently and the UCP and CTAG caches also substantially reduce the miss-handler cycles.

***Policy Scaling*** A key goal of our model (and optimizations) is to make it relatively painless to add additional policies that are simultaneously enforced. The Composite policy on the simple PUMP design incurred more than incremental costs for several benchmarks due to the large increase in miss handler runtime, but we reduced these with the miss handler optimizations. For most benchmarks, one policy (typically Memory Safety) dominates the performance of the composition. Adding more policies to this dominant one incurs little further performance degradation—the runtime overhead is increased by less than 1% per policy. This is shown in Fig. 15 where we first show the overhead of each single policy, then show composites that incrementally add one policy at a time to Memory Safety, our most complex single policy. Between the Memory-Safety-only case and the full composite, the average runtime overhead grows from 5% to 8% and the average energy overhead grows from 50% to 60%. The progression makes it clearer what overhead comes simply from adding any policy as opposed to adding a higher-overhead policy. These results show that the caches are effective for these benchmarks: the working set does not grow substantially, and the time spent in miss-handlers resolving all the policies is effectively contained such that it does not degrade performance.

To provide a sense of scaling beyond the four policies measured here, we have decomposed the CFI policy into returns (CFI1) and indirect jumps (CFI2) and the Taint Tracking policy into code tainting and I/O tainting. This does not increase the main PUMP cache working set, as it requires the same number of distinct composite rules, but it does increase the miss-handler complexity and pressure in the UCP and CTAG caches. The fact that we do not see overhead jumps between code and I/O tainting and between CFI1 and CFI2 suggests the working set remains small enough that the additional miss-handler complexity has only incremental impact on overall overhead.

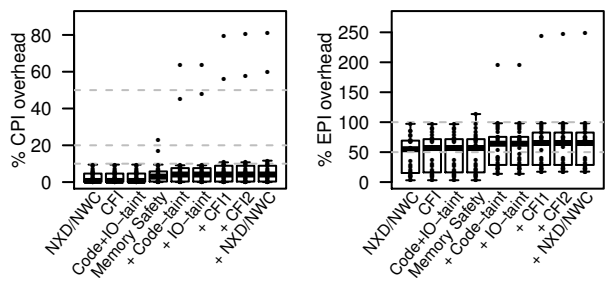


Figure 15: Impact Per Policy in Composition

Two outliers (`GemsFDTD` and `omnetpp`) have a larger increase in runtime overhead (20–40%), with only one (`GemsFDTD`) showing a larger increase in energy. The increases occur only when we add the first taint and first CFI policy, due mostly to higher memory safety compulsory miss rates for these benchmarks coupled with the increased miss-handler resolution complexity.

## 6. Related Work

The idea of using metadata for safety and security has been around since the 1960s [29]. During the past decade [63], it has been greatly refined and expanded (see Tab. 2). Our proposal extends previous work by offering a unified architecture for metadata tags and making it fully programmable.

The earliest uses of tags were for types, capabilities, and synchronization [34, 51, 61]. More recent uses include taint tracking [15, 19, 63] and bounds checking [17, 46]. Raksha [22] and DataSafe [16] generalize from the 1-bit taints and hardwired policies of earlier systems to larger taint fields (4b and 10b, respectively) and taint-specific configurable propagation rules. Our proposal further generalizes these systems by offering pointer-sized tags and fully programmable, software-defined rules for metadata validation and propagation.

Metadata has long been used for page-size units; later work has associated metadata at finer granularities, down to individual words or bytes [5, 67]. These systems can also trap to software for handling, though the tags do not propagate with the data and no tag results are computed.

DISE [18], Log-Based Architecture (LBA) [13, 14], and FADE [30] provide software-defined checking by inserting instructions in the processor's instruction stream (DISE) or running software on a processor core dedicated to checking (LBA, FADE). Aside from performance gains from code compactness, the performance and energy of DISE are comparable to software-only solutions. LBA adds hardware accelerators for checking and simple propagation (but not computation) of metadata, and FADE adds the the ability to perform programmable `AND` or `OR` computations. Some of the innovations in these accelerators (e.g., the restriction of taint propagation to unary inheritance tracking, excluding taint combining in LBA or the programmable `AND` or `OR` in the FADE Event Table) give up generality that our solution retains. LBA and FADE were demonstrated on simpler policies than our single policies, but have $\sim$50% (LBA) and 20–80% (FADE) runtime overhead compared to our 6% single-policy overhead (Fig. 15). The PUMP can cache any rule for single-cycle PUMP resolution, even rules involving composite tags, while FADE can only capture a subset of operations in a single cycle ("single-shot"), with others requiring multi-cycle resolution. Using a second processor for log processing incurs roughly 100% energy overhead—higher than our worst-case single-policy energy overhead of 60%. Conversely, FADE is designed to work with superscalar, out-of-order cores, while the PUMP design presented here is only for in-order cores.

Other work aims to support software-defined metadata processing with hardware acceleration, similar to our approach. Aries sketches a vision for cached metadata propagation rules [11], and more recently Harmoni shows where a rule cache would fit in a programmable tag processing architecture [24], though neither shows how to map policies into rule tables or evaluates the runtime characteristics of rule-cache-supported policies. FlexiTaint [66] is closest to our design; it demonstrates support for two taint propagation policies and their combination. The policies we show here are richer than the ones supported by FlexiTaint, due both to the extra tag inputs and outputs and to the richer tag metadata. FlexiTaint could in principle be scaled to wide metadata; it would then suffer performance and energy problems similar to our simple design (§3) and would likely benefit from our optimizations.

## 7. Conclusions and Future Work

With evidence mounting for the value of metadata-based policy enforcement, the time is ripe to define an architecture for software-defined metadata processing and identify accelerators to remove most of the runtime overhead. The PUMP architecture neither bounds the number of metadata bits nor the number of policies simultaneously supported; its microarchitectural optimizations (opgroups, tag compression, tag translation, and miss handler acceleration—see §4) achieve performance comparable to dedicated, hardware metadata propagation solutions (§5). We believe the software-defined metadata policy model and its acceleration will be applicable to a large range of policies beyond those illustrated here, including sound information-flow control [7, 8, 32, 56, 62], fine-grained access control [67, 71], integrity, synchronization [6, 61], race detection [58, 73], debugging, application-specific policies [70], and controlled generation and execution of dynamic code.

## Acknowledgments

This material is based upon work supported by the DARPA CRASH program through the United States Air Force Research Laboratory (AFRL) under Contract No. FA8650-10-C-7090. The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

## References


[1] *Alpha Architecture Handbook*. Digital Equipment Corporation, 1992.

[2] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity. In *12th ACM Conference on Computer and Communications Security*, pages 340–353. ACM, 2005.

[3] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information System Security*, 13(1), 2009.

[4] M. Abadi and C. Fournet. Access control based on execution history. In *Proceedings of the 10th Annual Network and Distributed System Security Symposium*, pages 107–121. The Internet Society, 2003.

[5] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha. Architectural support for run-time validation of program data properties. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 15(5):546–559, May 2007.

[6] Arvind, R. S. Nikhil, and K. K. Pingali. I-structures: Data structures for parallel computing. In *Proceedings of the Workshop on Graph Reduction (Springer-Verlag Lecture Notes in Computer Science 279)*, Sept. 1986.

[7] T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *Workshop on Programming Languages and Analysis for Security (PLAS)*, PLAS, pages 113–124. ACM, 2009.

[8] A. Bichhawat, V. Rajani, D. Garg, and C. Hammer. Information flow control in WebKit's JavaScript bytecode. In *3rd International Conference on Principles of Security and Trust*, volume 8414 of *Lecture Notes in Computer Science*, pages 159–178. Springer, 2014.

[9] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, Aug. 2011.

[10] E. Blem, J. Menon, and K. Sankaralingam. Power Struggles: Revisiting the RISC vs. CISC Debate on Contemporary ARM and x86 Architectures. In *Proc. HPCA*, pages 1–12, 2013.

[11] J. Brown and T. F. Knight, Jr. A minimally trusted computing base for dynamically ensuring secure information flow. Technical Report 5, MIT CSAIL, November 2001. Aries Memo No. 15.

[12] J. M. Chambers, W. S. Cleveland, B. Kleiner, and P. A. Tukey. *Graphical Methods for Data Analysis*. Wadsworth Statistics/Probability Series. Duxbury Press, 1983.

[13] S. Chen, B. Falsafi, P. B. Gibbons, M. Kozuch, T. C. Mowry, R. Teodorescu, A. Ailamaki, L. Fix, G. R. Ganger, B. Lin, and S. W. Schlosser. Log-based architectures for general-purpose monitoring of deployed code. In *1st Workshop on Architectural and System Support for Improving Software Dependability (ASID)*, pages 63–65. ACM, 2006.

[14] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. C. Mowry, V. Ramachandran, O. Ruwase, M. P. Ryan, and E. Vlachos. Flexible hardware acceleration for instruction-grain program monitoring. In *35th International Symposium on Computer Architecture (ISCA)*, pages 377–388. IEEE, 2008.

[15] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and R. Iyer. Defeating memory corruption attacks via pointer taintedness detection. In *International Conference on Dependable Systems and Networks (DSN)*, pages 378–387, 2005.

[16] Y.-Y. Chen, P. A. Jamkhedkar, and R. B. Lee. A software-hardware architecture for self-protecting data. In *ACM Conference on Computer and Communications Security*, pages 14–27. ACM, 2012.

[17] J. A. Clause, I. Doudalis, A. Orso, and M. Prvulovic. Effective memory protection using dynamic tainting. In *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 284–292. ACM, 2007.

[18] M. L. Corliss, E. C. Lewis, and A. Roth. DISE: a programmable macro engine for customizing applications. *SIGARCH Comput. Archit. News*, 31(2):362–373, May 2003.

[19] J. R. Crandall, F. T. Chong, and S. F. Wu. Minos: Architectural support for protecting control data. *ACM Transactions on Architecture and Code Optimization*, 5:359–389, December 2006.

[20] J. Criswell, N. Dautenhahn, and V. Adve. KCoFI: Complete control-flow integrity for commodity operating system kernels. In *IEEE Security and Privacy Symposium*, 2014.

[21] W. J. Dally, J. Balfour, D. Black-Shaffer, J. Chen, R. C. Harting, V. Parikh, J. Park, and D. Sheffield. Efficient embedded computing. *IEEE Computer*, 41(7):27–32, July 2008.

[22] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: a flexible information flow architecture for software security. In *International Symposium on Computer Architecture (ISCA)*, pages 482–493, 2007.

[23] L. Davi, A. Sadeghi, D. Lehmann, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *23rd USENIX Security Symposium*, pages 401–416, 2014.

[24] D. Y. Deng and G. E. Suh. High-performance parallel accelerator for flexible and efficient run-time monitoring. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12. IEEE Computer Society, 2012.

[25] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic. HardBound: Architectural support for spatial safety of the C programming language. In *13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 103–114, 2008.

[26] U. Dhawan and A. DeHon. Area-efficient near-associative memories on FPGAs. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, pages 191–200, 2013.

[27] U. Dhawan, C. Hriţcu, R. Rubin, N. Vasilakis, S. Chiricescu, J. M. Smith, T. F. Knight, Jr., B. C. Pierce, and A. DeHon. Online appendix to Architectural support for software-defined metadata processing. Available from `http://ic.ese.upenn.edu/abstracts/sdmp_asplos2015.html`, January 2015.

[28] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *7th Symposium on Operating Systems Design and Implementation*, pages 75–88. USENIX Association, 2006.

[29] E. A. Feustel. On the advantages of tagged architectures. *IEEE Transactions on Computers*, 22:644–652, July 1973.

[30] S. Fytraki, E. Vlachos, Y. O. Koçberber, B. Falsafi, and B. Grot. FADE: A programmable filtering accelerator for instruction-grain monitoring. In *20th IEEE International Symposium on High Performance Computer Architecture*, pages 108–119, 2014.

[31] E. Göktaş, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *IEEE Symposium on Security and Privacy*, 2014.

[32] D. Hedin and A. Sabelfeld. Information-flow security for a core of JavaScript. In *25th IEEE Computer Security Foundations Symposium (CSF)*, CSF, pages 3–18. IEEE, 2012.

[33] J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, Sept. 2006.

[34] M. E. Houdek, F. G. Soltis, and R. L. Hoffman. IBM System/38 Support for Capability-based Addressing. In *Proceedings of the Eighth Annual Symposium on Computer Architecture*, pages 341–348, 1981.

[35] C. Hriţcu, M. Greenberg, B. Karel, B. C. Pierce, and G. Morrisett. All your IFCException are belong to us. In *34th IEEE Symposium on Security and Privacy*, pages 3–17. IEEE Computer Society Press, May 2013.

[36] Introduction to Intel Memory Protection extensions. `http://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions`. Accessed: 2014-05-24.

[37] M. G. Kang, S. McCamant, P. Poosankam, and D. Song. DTA++: Dynamic taint analysis with targeted control-flow propagation. In *Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2011.

[38] H. Kannan. Ordering decoupled metadata accesses in multiprocessors. In *Proceedings of IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 381–390, 2009.

[39] D. King, B. Hicks, M. Hicks, and T. Jaeger. Implicit flows: Can't live with 'em, can't live without 'em. In *4th International Conference on Information Systems Security*, ICISS, pages 56–70, 2008.

[40] A. Kwon, U. Dhawan, J. M. Smith, T. F. Knight, Jr., and A. DeHon. Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 721–732. ACM, 2013.

[41] K. Mai, R. Ho, E. Alon, D. Liu, Y. Kim, D. Patil, and M. Horowitz. Architecture and Circuit Techniques for a 1.1GHz 16-kb Reconfigurable Memory in 0.18um-CMOS. *IEEE J. Solid-State Circuits*, 40(1):261–275, January 2005.

[42] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Workshop on Hardware and Architectural Support for Security and Privacy*, pages 10:1–10:1, 2013.

[43] M. S. Miller, K.-P. Yee, and J. Shapiro. Capability myths demolished. Version submitted to Usenix Security 2003., 2003.

[44] D. A. Moon. Architecture of the Symbolics 3600. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, ISCA, pages 76–83, Los Alamitos, CA, USA, 1985. IEEE Computer Society.

[45] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. CACTI 6.0: A tool to model large caches. HPL 2009-85, HP Labs, Palo Alto, CA, April 2009. Latest code release for CACTI 6 is 6.5.

[46] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Hardware-Enforced Comprehensive Memory Safety. *IEEE Micro*, 33(3):38–47, May-June 2013.

[47] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. WatchdogLite: Hardware-accelerated compiler-based pointer checking. In *12th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, page 175. ACM, 2014.

[48] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic. CETS: compiler enforced temporal safety for C. In *9th International Symposium on Memory Management*, pages 31–40. ACM, 2010.

[49] J. Newsome and D. X. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *the Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2005.

[50] B. Niu and G. Tan. Modular control-flow integrity. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 58. ACM, 2014.

[51] E. I. Organick. *Computer System Organization: The B5700/B6700 Series*. Academic Press, 1973.

[52] D. A. Patterson and C. H. Sequin. RISC I: A Reduced Instruction Set VLSI Computer. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, ISCA '81, pages 443–457, 1981.

[53] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu. LIFT: A low-overhead practical information flow tracking system for detecting security attacks. In *39th IEEE/ACM International Symposium on Microarchitecture (MICRO-39)*, pages 135–148, 2006.

[54] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee, Jr. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, December 2004.

[55] D. Ritchie and K. Thompson. The UNIX Time-Sharing System. *Bell System Technical Journal*, 57(6):1905–1930, 1978.

[56] A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *23rd Computer Security Foundations Symposium (CSF)*, CSF, pages 186–199. IEEE Computer Society, 2010.

[57] O. Ruwase and M. S. Lam. A Practical Dynamic Buffer Overflow Detector. In *Proc. NDSS*, pages 159–169, 2004.

[58] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4), 1997.

[59] H. Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *Proc. ACM CCS*, pages 552–561, Oct. 2007.

[60] R. Shioya, D. Kim, K. Horio, M. Goshima, and S. Sakai. Low-overhead architecture for security tag. In *Proceedings of the 2009 15th IEEE Pacific Rim International Symposium on Dependable Computing*, PRDC '09, pages 135–142, Washington, DC, USA, 2009. IEEE Computer Society.

[61] B. J. Smith. A pipelined, shared-resource MIMD computer. In *Proc. ICPP*, pages 6–8, 1978.

[62] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in Haskell. In *4th Symposium on Haskell*, pages 95–106. ACM, 2011.

[63] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–96, 2004.

[64] G. S. Taylor, P. N. Hilfinger, J. R. Larus, D. A. Patterson, and B. G. Zorn. Evaluation of the SPUR lisp architecture. In *Proceedings of the 13th annual International Symposium on Computer architecture*, ISCA, pages 444–452, 1986.

[65] M. Tiwari, B. Agrawal, S. Mysore, J. Valamehr, and T. Sherwood. A Small Cache of Large Ranges: Hardware Methods for Efficiently Searching, Storing, and Updating Big Dataflow Tags. In *Proc. MICRO*, pages 94–105, 2008.

[66] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic. FlexiTaint: A programmable accelerator for dynamic taint propagation. In *14th International Symposium on High Performance Computer Architecture (HPCA)*, pages 173–184, Feb. 2008.

[67] E. Witchel, J. Cates, and K. Asanović. Mondrian memory protection. In *10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, pages 304–316, New York, NY, USA, 2002. ACM.

[68] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe. The CHERI capability model: Revisiting RISC in an age of risk. In *Proc. of the International Symposium on Computer Architecture (ISCA)*, pages 457–468, June 2014.

[69] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: a sandbox for portable, untrusted x86 native code. *Communications of the ACM*, 53(1):91–99, 2010.

[70] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertions. In *Proceedings of the Symposium on Operating Systems Principles*, Big Sky, MT, USA, October 2009.

[71] N. Zeldovich, H. Kannan, M. Dalton, and C. Kozyrakis. Hardware enforcement of application security policies using tagged memory. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI, pages 225–240. USENIX Association, 2008.

[72] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical Control Flow Integrity & Randomization for Binary Executables. In *IEEE Symposium on Security and Privacy*, 2013.

[73] P. Zhou, R. Teodorescu, and Y. Zhou. HARD: Hardware-assisted lockset-based race recording. In *Proc. HPCA*, 2007.

## A. Policy Example: Memory Safety

***Description*** We implement a scheme (based on [17]) that identifies all temporal and spatial violations in heap-allocated memory. Intuitively, for each new allocation we make up a fresh *color-id*, $c$, and write $c$ as the tag on each memory location in the newly created memory block (*à la* `memset`). The pointer to the new block is also tagged $c$. Later, when we dereference a pointer, we check that its tag is the same as the tag on the memory cell to which it points. When a block is freed, the tags on all its cells are changed to a constant $F$ representing free memory. The heap is initially all tagged $F$. We use a special tag $\perp$ for non-pointers, and we write $t$ for a tag that is either a color $c$ or $\perp$.

Because memory cells may contain pointers, in general each word in memory has to be associated with *two* tags. We handle this by making the tag on each memory cell be a pointer to a pair $(c, t)$, where $c$ is the id of the memory block in which this cell was allocated and $t$ is the tag on the word stored in the cell. We use a domain-specific language based on the rule function described in §2 for specifying a policy in terms of symbolic rules. The rules for `load` and `store` take care of packing and unpacking these pairs, along with checking that each memory access is valid (*i.e.*, the accessed cell is within the block pointed to by this pointer):

$$\texttt{load}: \ (-,-,c_1,-,(c_2,t_2)) \rightarrow (-,t_2) \text{ if } c_1 = c_2 \quad (1)$$

$$\texttt{store}: \ (-,-,t_1,c_2,(c_3,t_3)) \rightarrow (-,(c_3,t_1)) \text{ if } c_2 = c_3 \quad (2)$$

The checking shows up as conditions under which the symbolic rule is valid (e.g., $c_2 = c_3$ above). The "–" symbol indicates the don't care fields in the rule.

Address arithmetic operations preserve the pointer tag:

$$\texttt{add}: \ (-,-,c,\perp,-) \rightarrow (-,c) \quad (3)$$

To maintain the invariant that tags on pointers can only originate from allocation, operations that create data from scratch (like loading constants) set its tag to $\perp$.

We augment `malloc` and `free` to tag memory regions using the tagged instructions and ephemeral rules (which are deleted from the rule cache once they are used). In `malloc` we generate a fresh tag for the pointer to the new region via an ephemeral rule.

$$\texttt{move}: \ (-,t_{malloc},t,-,-) \xrightarrow{1} (-,t_{newtag}) \quad (4)$$

The arrow annotated with the 1 denotes an ephemeral rule. We then use the newly tagged pointer to write a zero to every word in the allocated region using a special store rule

$$\texttt{store}: \ (-,t_{mallocinit},t_1,c_2,F) \rightarrow (-,(c_2,t_1)) \quad (5)$$

before returning the tagged pointer. Later, `free` uses a modified store instruction to retag the region as unallocated

$$\texttt{store}: \ (-,t_{freeinit},t_1,c_2,(c_3,t_4)) \rightarrow (-,F) \quad (6)$$

before returning the memory region to the free list.

***Complete Symbolic Rule Set*** We use opgroups to compactly describe the ruleset:

$$\texttt{nop}, \texttt{cbranch}, \texttt{ubranch}, \texttt{ijump}, \texttt{return}: \ (-,-,-,-,-) \rightarrow (-,-) \quad (1)$$

$$\texttt{ar2s1d}: \ (-,-,\perp,\perp,-) \rightarrow (-,\perp) \quad (2)$$

$$\texttt{ar2s1d}: \ (-,-,c,\perp,-) \rightarrow (-,c) \quad (3)$$

$$\texttt{ar2s1d}: \ (-,-,\perp,c,-) \rightarrow (-,c) \quad (4)$$

$$\texttt{ar2s1d}: \ (-,-,c,c,-) \rightarrow (-,\perp) \quad (5)$$

$$\texttt{ar1s1d}: \ (-,-,t,-,-) \rightarrow (-,t) \quad (6)$$

$$\texttt{ar1d}, \texttt{dcall}, \texttt{icall}, \texttt{flags}: \ (-,-,-,-,-) \rightarrow (-,\perp) \quad (7)$$

$$\texttt{load}: \ (-,-,c_1,-,(c_2,t_2)) \rightarrow (-,t_2) \text{ if } c_1 = c_2 \quad (8)$$

$$\texttt{store}: \ (-,ci,t_1,c_2,(c_3,t_3)) \rightarrow (-,(c_3,t_1)) \text{ if } c_2 = c_3 \wedge ci \notin \{t_{mallocinit}, t_{freeinit}\} \quad (9)$$

$$\texttt{store}: \ (-,t_{mallocinit},t_1,c_2,F) \rightarrow (-,(c_2,t_1)) \quad (10)$$

$$\texttt{store}: \ (-,t_{freeinit},t_1,c_2,(c_3,t_4)) \rightarrow (-,F) \quad (11)$$

$$\texttt{move}: \ (-,t_{malloc},t,-,-) \xrightarrow{1} (-,t_{newtag}) \quad (12)$$

$$\texttt{move}: \ (-,\overline{t_{malloc}},t,-,-) \rightarrow (-,t) \quad (13)$$

***Concrete Rules*** The symbolic rules used above for policy specification are written using *variables*, allowing a few symbolic rules to describe the policy over an unbounded universe of distinct values. The concrete rules stored in the rule cache, however, must refer to specific, concrete tag values. For example, if 23 and 24 are valid memory block colors, we will need concrete instances of Rule 3 in the PUMP rule cache for $c = 23$ and $c = 24$. Assuming we encode $\perp$ as 0 and mark don't care fields as 0, the concrete rules are:

$$\texttt{ar2s1d}: \ (0,0,23,0,0) \rightarrow (0,23)$$

$$\texttt{ar2s1d}: \ (0,0,24,0,0) \rightarrow (0,24)$$

The miss handler gets the concrete input tags and runs code compiled from the symbolic rules to produce the associated concrete output tags in order to insert rules into the PUMP rule cache. When the symbolic rule identifies a violation, control transfers to an error handler and no new concrete rules are inserted into the PUMP rule cache.

***Tag and Rule Statistics*** In the first 1.5B instructions, the `GemsFDTD` benchmark generates 580K memory block colors. 3.3M concrete rules are generated from the 13 symbolic rules above. Since not all combinations of memory block and pointer tags actually occur, symbolic rules like Rule 8 do not generate the theoretical maximum number of concrete rules ((580K)$^2$=330B$\gg$3.3M).