PUMP: A Programmable Unit for Metadata Processing

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

[1] University of Pennsylvania
[2] BAE Systems
[3] Ginkgo Bioworks

ABSTRACT

We introduce the Programmable Unit for Metadata Processing (PUMP), a novel software-hardware element that allows flexible computation with uninterpreted metadata alongside the main computation with modest impact on runtime performance (typically 10–40% for single policies, compared to metadata-free computation on 28 SPEC CPU2006 C, C++, and Fortran programs). While a host of prior work has illustrated the value of ad hoc metadata processing for specific policies, we introduce an architectural model for extensible, programmable metadata processing that can handle arbitrary metadata and arbitrary sets of software-defined rules in the spirit of the time-honored 0-1-∞ rule. Our results show that we can match or exceed the performance of dedicated hardware solutions that use metadata to enforce a single policy, while adding the ability to enforce multiple policies simultaneously and achieving flexibility comparable to software solutions for metadata processing. We demonstrate the PUMP by using it to support four diverse safety and security policies—spatial and temporal memory safety, code and data taint tracking, control-flow integrity including return-oriented-programming protection, and instruction / data separation—and quantify the performance they achieve, both singly and in combination.

Categories and Subject Descriptors

C.1 [**Processor Architecture**]: Miscellaneous—*security*

Keywords

security, metadata, tagged architecture, control-flow integrity, taint tracking, memory safety

1. INTRODUCTION

Present-day processors are mindless bureaucrats, performing whatever operations are asked of them even when these do not make sense (*e.g.*, "run this pointer as an instruction", "return into the middle of this string"). A consequence of dumb hardware is that software bears the major burden for security, leading to unfortunate security-performance tradeoffs [35]. Software can insert barriers between components, introspect on computation, and maintain safety and security invariants, but only at a cost (high runtime)—one that is often deemed unacceptable, leading to systems built to minimize any such protections. Furthermore, when protection is optional, programmers are left to decide for themselves how to protect their own system components, leading to myriad weak points in every large computer system.

There is an actively growing body of knowledge about runtime policies that can reduce the vulnerability of computers and software to malicious subversion (see, *e.g.*, [17, 7] and the references in §6). These policies provide sanity checks that current hardware lacks (*e.g.*, this piece of code should never return to that point, data from the network should not be treated as code without careful scrutiny, etc.). Many of these invariants simply enforce language abstractions that current software and hardware omit as part of the aforementioned security-performance trade-off (*e.g.*, bounds checking on objects, control-flow integrity); these policies can be automatically applied to all code with no additional programmer burden. Others are policies a system architect might like to apply to code without auditing or rewriting every line of code in a larger system (*e.g.*, private data should be encrypted before flowing to I/O devices). A large class of such policies can be supported by adding metadata to the data being processed (*e.g.,* this is an instruction, this is from the network, this is private), propagating the metadata through the computation, and checking that rules on the metadata are enforced throughout the computation.

With transistors now cheap, it is reasonable to invest some hardware in reducing the traditional security-performance trade-off. Over the past few years, hardware has been proposed that handles specific cases of limited metadata processing and rule enforcement (surveyed in §6), establishing that these policies can be enforced with little performance penalty by data tagging and hardware checking. However, these proposals typically support only a single policy.

In the spirit of the 0-1-∞ rule[1] we introduce an architectural model for extensible, programmable metadata processing that can handle arbitrary metadata and arbitrary sets of software-defined rules. A hardware rule cache allows these rules to be executed entirely by the hardware in the common case, in parallel with instruction execution. Our preliminary results show that we can match or exceed the performance

HASP '14, June 15 2014, Minneapolis, MN, USA

ACM 978-1-4503-2777-0/14/06&#0133;$15.00.
http://dx.doi.org/10.1145/2611765.2611773.

[1] `http://www.catb.org/jargon/html/Z/Zero-One-Infinity-Rule.html`

of dedicated hardware that uses metadata to enforce a single policy while enforcing multiple policies simultaneously and achieving flexibility comparable to software solutions for metadata processing. We illustrate the flexibility and performance of our architecture by applying four security and safety policies to the C, C++, and Fortran programs in the SPEC CPU2006 benchmark suite: (i) instruction and data separation, (ii) spatial and temporal memory safety on the heap, (iii) control-flow integrity, and (iv) data and code taint tracking.

We introduce our software/hardware architecture for supporting generic, unbounded metadata rule processing in §2. §3 describes the microarchitecture implementation including estimates of key performance parameters. We present the application of four diverse metadata propagation and checking policies in §4 and evaluate their impact on runtime for the SPEC CPU2006 benchmarks in §5. We discuss related work in §6 and conclude in §7.

2. PUMP ARCHITECTURE

Our generalization over prior work on tagged hardware processing (§6) starts by using a *pointer-sized tag* for metadata, allowing a rich range of arbitrarily structured metadata. Our basic addressable memory word is indivisibly extended with this metadata tag, making all value slots, including memory, caches, and registers, suitably wider. The metadata tag is not addressable by the user program. Instead, it is carried along with all values in the computation and updated (if needed) on each instruction in parallel with the associated values.

The core architectural feature proposed in this paper is the PUMP—an architectural mechanism designed to enforce runtime policies. The policies manifest themselves in terms of rules that define an atomic operation for the PUMP. Since the rules apply to each instruction, the PUMP must provide a result per instruction. To avoid slowing down the computation, the PUMP should operate in parallel with the normal ALU computation. To the hardware, the metadata tags are uninterpreted. Accordingly, the hardware does not compute rules. Rather, since rules are purely functional, hardware can *cache* the mapping between the inputs and outputs of rules. Therefore, we add PUMP caches to the processor. To allow software interpretation and support software-hardware trade-offs in the PUMP design, when a rule is not found in the PUMP cache, it traps to a software handler that can resolve the rule and insert it into the hardware cache.

The PUMP allows a programmer or a system designer to create policies. A *policy* is defined as a functional mapping of a set of *tags* to another set of *tags* resulting in a collection of *rules* that implement some desired tracking and enforcement mechanism and also manipulate the tags. Rules come in two forms, depending on whether we are talking about the software layer (*symbolic rules*) or hardware layer (*concrete rules*) of the system.

Symbolic Rules.

From the point of view of the policy programmer and the software parts of the PUMP, policy rules are compactly described using *symbolic tags* and are written in a tiny domain-specific language. These make up the *symbolic rules*. Each symbolic rule has the form

$$\begin{aligned} \textit{opcode}: \quad & (PC, CI, \textit{OP1}, \textit{OP2}, MR) \\ & \rightarrow (PC_{new}, R, \textit{allow}?) \end{aligned}$$

which says that the rule matches on the given *opcode* together with the metadata tags on the program counter (PC), on the current instruction (CI), on the two operands from the register file ($OP1$, $OP2$), if any, and on the value read from memory (MR), if any; on a match, the right-hand side of the rule determines if the operation is allowed (*allow*?) and how to update the metadata tag on the PC (PC_{new}) and on the result of the operation (R, which may be a destination register ($OP3$) or a value to be written to memory (MW), depending on the instruction). We write "−" to indicate ignored input fields and unused output fields. The input set represents the subset of the architecturally visible state that can affect an instruction, while the output set is simply the state that can be affected as a result of the instruction. This extends the prior work where mostly a single output is computed from two inputs (see §6).

An example problem that can be addressed by the PUMP is *return-oriented programming (ROP)* [33, 9]. ROP attacks work by identifying a set of "gadgets" in the binary executable of the program under attack and using these to assemble complex malicious behaviors by constructing an appropriate sequence of stack frames, each containing a return address pointing to some gadget, and then using a buffer overflow or other vulnerability to overwrite the top of the stack with this sequence, causing the snippets to be executed in order. We can limit return targets only to well-defined program points using the PUMP by tagging instructions that are valid return points with a metadata tag **tgt** and creating policy rules that behave as follows – each time we execute a `return` instruction, we set the metadata tag on the PC to **check** to indicate that a `return` has just occurred. On the next instruction (*i.e.*, whenever the PC tag is **check**), we check that the tag on this instruction is **tgt**; if it is not, we terminate the currently executing process.

For the simple ROP policy just sketched, the possible tag values are **empty**, **check**, and **tgt**; by convention, the PC will always be tagged either **empty** or **check** and each instruction will be tagged either **empty** or **tgt**. The symbolic rules for this policy are:

$$\begin{aligned} \texttt{return}: \quad & (\textbf{empty}, -, -, -, -) \rightarrow (\textbf{check}, -, \textbf{true}) & (1) \\ \overline{\texttt{return}}: \quad & (\textbf{check}, \textbf{tgt}, -, -, -) \rightarrow (\textbf{empty}, -, \textbf{true}) & (2) \\ \overline{\texttt{return}}: \quad & (\textbf{empty}, -, -, -, -) \rightarrow (\textbf{empty}, -, \textbf{true}) & (3) \\ \texttt{return}: \quad & (\textbf{check}, \textbf{tgt}, -, -, -) \rightarrow (\textbf{check}, -, \textbf{true}) & (4) \end{aligned}$$

Rule 1 says that, when we encounter a return instruction (and the PC is not tagged **check**), we change the metadata tag on the PC to **check**. When we run an instruction with the PC metadata marked as **check** (Rule 2), we check if the instruction metadata, *CI*, is **tgt**; if so, we clear the metadata on the PC ("$\overline{\texttt{return}}$" means "any opcode except `return`"). If the operation is not a `return` and the PC metadata is **empty**, we do nothing (Rule 3). Rule 4 handles the special case where the target of a `return` instruction is itself a `return`. If no rule applies, we disallow the operation.

The expressions describing tags and opcodes in these symbolic rules are not limited to constant values: we can write more general expressions that compactly describe large sets of opcodes or tags. So far, we have used this flexibility only in a simple way (the two instances of overbars denoting set complement); for a more interesting example, let's consider a more precise variant of the return-matching policy. This refinement makes sure not only that every return reaches *some* valid return target, but also that each return targets a code point from which it could *actually* have been called. This policy works on compiled code, where the compiler has full knowledge of return points and can analyze, for each one, which call sites it might validly return to. Using this information, we can attach unique metadata tags to each **`return`** instruction and to each valid return target. Upon encountering a **`return`** instruction, the PUMP copies the specific tag on the instruction (rather than the generic tag **check**) onto the PC. On the next step, it can check that the actual return point was among the expected ones by checking if a rule with that particular *PC* / *CI* tag combination exists. For instance, if the compiler knows that the `return` instruction tagged t_1 can only return to the return sites tagged t_2 and t_3, the policy will contain the following rules for the `return` instruction:

$$\texttt{return}: \quad (\mathbf{empty}, t_1, -, -, -) \rightarrow (t_1, -, \mathbf{true}) \tag{5}$$

$$\overline{\texttt{return}}: \quad (t_1, t_2, -, -, -) \rightarrow (\mathbf{empty}, -, \mathbf{true}) \tag{6}$$

$$\overline{\texttt{return}}: \quad (t_1, t_3, -, -, -) \rightarrow (\mathbf{empty}, -, \mathbf{true}) \tag{7}$$

Concrete Rules. Symbolic rules allow a programmer to express the policy rules by using abstract symbolic tags. At the hardware level, however, we need a rule representation that is tuned for efficient implementation to avoid slowing down the primary computation. To this end, we introduce a lower-level rule format, called *concrete rules* which makes use of *concrete tags* (specific instances of symbolic tags). Intuitively, the symbolic rules for a given policy get converted into concrete ones. However, since a single symbolic rule might in general result in several concrete rules, we perform this conversion *lazily*, generating only those concrete rules that are actually needed as the system executes—likely a much smaller number.

When an instruction is issued, the PUMP needs to perform the metadata computation as defined by the policy, which essentially means that the PUMP needs to check if there is a *rule* that validates the current instruction. Conceptually if all the policy rules are elaborated into a tabular form, the PUMP operation boils down to aggregating the tags from the current architectural state and performing an associative match against all the rules in that table. If a match is found, the PUMP cache returns the new tag for the PC and, if needed, a tag for the instruction's result. If there is no match in the PUMP cache, the processor faults to a PUMP *miss handler*. This consults the symbolic rules for the policy and determines whether the faulting machine state is actually allowed; if so, it generates an appropriate concrete rule, installs it in the cache, and restarts the faulting instruction. Or, if the 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 rule policy; typically, it would shut down the offending process, but in some cases it might return a suitable "safe value" instead [29, 22].

Composite Policies. As a security policy programmer, it is easier to create multiple, mostly orthogonal policies that maintain different invariants and perform distinct analyses. As such, for full protection, we would like to run all of these policies simultaneously and be able to add additional policies as they are developed. One of the features of our general, uninterpreted tag approach 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 taken from several component policies. For example, to combine the first ROP policy with a taint policy, we would let each tag be a pointer to a representation of a tuple (r, t), where r is an ROP-tag (**empty**, **check**, etc.) and t is a taint tag (another pointer to a set of taints). The rule cache lookup is exactly the same, but when a miss occurs, both sets of symbolic rules are evaluated separately, the operation is allowed only if both *allow?* expressions evaluate to true, and the resulting tags are pairs of results from the two sub-policies.

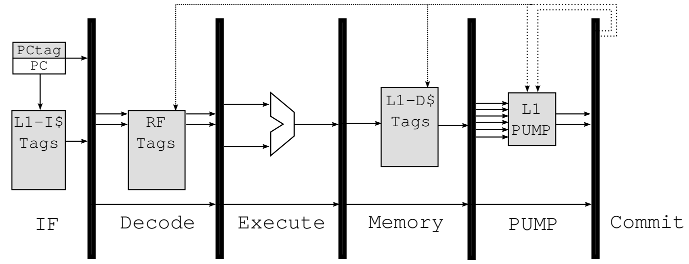

Figure 1: L1 PUMP cache in a processor pipeline

3. MICROARCHITECTURE

The PUMP hardware cache is essentially a fixed-capacity associative map. It is closer to a modern translation lookaside buffer than a normal memory cache. The wide match key makes it more challenging to implement lookups than for a simple address-to-word translation. As with any caching scheme, we can use multiple levels of caching to implement the memory hierarchy for the PUMP. In this paper we specifically consider a 2-level PUMP. L2-PUMP misses invoke the software miss handler. In this section we take a deeper look at how this hardware can be implemented.

In this paper, we are concerned primarily with establishing achievable runtime performance. The simple implementation described here has fairly high area and energy costs, but we believe these can be tamed with suitable microarchitectural optimizations; these are the focus of our ongoing work. For concreteness, we use a simple in-order, single-issue, 5-stage pipelined processor core with the 64-bit ALPHA ISA [2] as the baseline—this is representative of the energy-efficient cores used in modern embedded devices [1].

Pipeline Integration. First we show how to add the PUMP support to the baseline processor. For simplicity we consider a *coupled tagging scheme*—any data that flows in to the processor pipeline flows with its tag. Within the pipeline, we split off the metadata from the payload and dispatch them to their respective operation units.

A rule may take as input the value read from memory (such as in the case of a `load`) that is not available until a later pipe stage. To avoid introducing unnecessary stalls, we place the PUMP lookups in a pipe stage after the result from the memory read is available. We call this the `PUMP` stage. To avoid stalls, we require that this stage return the output tags in a single cycle; to identify PUMP cache misses, the input tags can be matched in the next stage before the instruction is committed. The resulting pipeline has 6 stages, as shown in Fig. 1. All the PUMP inputs are available at the end of the `Memory` stage, so the additional pipeline stage does not introduce any new stalls. Fig. 1 also shows that we bypass the tags from the `Commit` stage back to the `PUMP` stage. Since we allow rules that depend on the tag of the memory location that is being overwritten in memory, write operations become read-modify-write operations. The tag of the old value of the memory word 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.

PUMP Service. For UNIX-style [30] 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, symbolic rules, and miss handling support into the address space of the process, avoiding the need for an OS-level context switch. This does mean that rules in the PUMP caches will be tagged by process id.

When an L2-PUMP-miss occurs we need to: (i) transfer to a suitable miss handler, (ii) obtain the opcode and tags for the instruction that missed in the PUMP, (iii) consult the symbolic rules of the policy and generate an appropriate concrete rule, (iv) install this rule into the PUMP, and (v) restart the faulting instruction. To provide isolation between the (highly privileged!) miss handler and the rest of the system software, we add a miss-handler mode to the processor. To avoid the need to save and restore registers, we expand the integer register file with 16 additional registers that are available only to the miss handler. Additionally, the PC of the faulting instruction, the rule inputs (opcode and tags), and the rule outputs appear as registers while in miss handler mode (*cf.* register windows [28]). We also add a new **`miss-handler-return`** instruction to finish installing the concrete rule into the PUMP and return to user code.

While the processor is in miss-handler mode, the PUMP applies a single, hardwired rule: all instructions and data touched by the miss handler must be tagged with a predefined **miss-handler** tag. This tagging provides isolation between the miss handler code, data, and user code in the same address space. This prevents user code from directly calling miss-handler code or manipulating miss-handler data structures. Conversely, it prevents the miss handler from accidentally touching user data or code.

PUMP Cache Implementation. The PUMP requires a long match key (5 address-sized tags plus an instruction opcode) compared to a traditional memory address key (less than the address width). Using a fully associative cache would lead to a high access latency (several cycles). Instead, we use the `dMHC` scheme from [18]. This cache provides performance close to that of a fully associative memory, at much lower cost. We use a `FIFO` replacement policy when the cache reaches capacity. The L1-PUMP cache is designed to produce the result in a single cycle while checking for a false hit in the second cycle using the `Fast-Value dMHC`. For the L2-PUMP cache, we use the space-efficient `Two-level dMHC`, taking multiple cycles to return a result.

Resource Estimates. For concreteness, we consider a 32 nm Low Operating Power (LOP) process. We use CACTI 6.5 [25] for evaluating our processor components. The baseline implementation has a 64KB L1 D$ with a latency around 820 ps. We assume the L1 D$ in the baseline processor 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. Extending each 64b word with a 64b tag increases this latency to 1300 ps if we keep the effective cache capacity the same, adding a stall cycle for each L1 access. We avoid these stalls by implementing a 64KB L1 cache with only 32KB effective capacity; tags occupy the remaining 32KB. Since the L2 cache access is already greater than a single cycle, we implement a bigger L2 cache to give us the same effective capacity. Consequently, the unified L2 cache accesses take 4 cycles instead of 3 cycles in the baseline. The L1-PUMP cache, which we need to operate in a single cycle to prevent it from pacing operation, can return the result in 695 ps, while the L2-PUMP cache has a hit latency of 3 cycles. Since we now fetch twice as many bits from the DRAM, each DRAM access is 30% slower than in the baseline case. The memory organization and cycle costs for the baseline and the tagged architecture are summarized in Tab. 1.

Unit	Design	Organization	Cyc
RF	Baseline	64b, 2R1W, {32 Integer, 32 Floating}	1
	tagged	128b, 2R1W, {48 Integer, 32 Floating}	1
L1 Cache (I or D)	Baseline	64KB, 4-way, 64B/line	1
	tagged	64KB, 4-way, 128B/line (eff. 32KB, 64B/line)	1
L2 Cache	Baseline	512KB, 8-way, 64B/line	3
	tagged	1MB, 8-way, 128B/line (eff. 512KB, 64B/line)	4
DRAM (main memory)	Baseline	1GB, access 64B line	100
	tagged	1GB, access 128B line (eff. 64B line)	130
L1 PUMP	tagged	1024-entry FV† dMHC(4,2) 328b match, 128b out	1
L2 PUMP	tagged	4096-entry, TL‡ dMHC(4,2) 328b match, 128b out	3

Table 1: Cycle estimates for baseline and PUMP-extended Processor at 32nm LP (CACTI 6.5) (†=Fast-Value, ‡=Two-Level [18])

4. RUNTIME POLICY CASE STUDIES

In this section we sketch four different policies addressing a range of useful program invariants can be implemented using the PUMP. This exercise illustrates the flexibility of the mechanism we are proposing and provides a diverse set of examples for evaluating its performance.

Primitive Types. As a simple illustration of a policy, we consider a set of primitive type tags for `C`—instruction (`insn`), address (`address`), and data (`data`). Only instructions can be executed, and they cannot be created at runtime. Only addresses can be used for memory addressing. The `data` type tag is used as a catch-all for words that are not instructions or data. This level of safety prevents accidental misinterpretation of data as addresses or instructions, and provides some protection against code injection.

Spatial and Temporal Memory Safety. We next use the PUMP to implement a scheme due to Nagarakatte *et al.* [26] that identifies all temporal and spatial violations in memory. Intuitively, for each new allocation we make up a fresh *block id* t and write t as the tag on each memory location in the newly created memory block (*á la* `memset`). The pointer to the new block is also tagged t. Later, when we dereference through 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, tags on all its cells are changed to a default value representing non-referenceable memory. Our implementation also deals with address arithmetic and the fact that an address in memory must be separately tagged with both the block it is in and the block to which it points. Our current implementation only guards heap data (where calls to `malloc` and `free` tell us how to set up memory regions), not stack frames.

Control-Flow Integrity. As discussed in §2, hijacking control flow is a common element of many attacks. Control-Flow Integrity (CFI) [3] is a powerful technique that analyzes programs and limits run-time control flows to those found in the source code. To address this set of attacks using the PUMP, we divide the notion of CFI into three parts: `CFI-ROP`, which restricts returns; `CFI-Call`, which restricts where procedures can be called from; and `CFI-IntraProc`, which also restricts branch source-target pairs within a procedure. We combine the three components together into a Complete Control-Flow Integrity (`CCFI`) policy. In comparison to the low-overhead but limited ROP protection in [39] and the ROP protection without CFI provided by DISE [13], this solution provides complete and precise call-return alignment with low performance overhead. In fact, the PUMP avoids the vulnerabilities in the weaker CCFI policy by allowing us to implement the ideal CFI as described in [19].

Taint Tracking. Taint tracking techniques track the provenance of data values, detecting situations where untrusted data or data produced by untrusted code flows into sensitive operations. Previous work has typically used a binary taint model, where a single-bit taint t simply indicates whether or not any data from an external source has been used in computing a value tagged t. Using the PUMP we can do finer-grained taint tracking, with an unlimited number of sources and a separate taint element per source. The tags for this policy are pointers to sets of source identifiers. We introduce taint in two different ways, namely tainting program inputs and tainting program code. Tainted inputs might include data received from the network or a file. We taint program code to protect against untrusted libraries and possibly buggy components. For the benchmarks that follow we use a `taint-by-library` policy that gives each library and input source a unique taint identifier.

Combinations. The PUMP architecture allows us to simultaneously enforce multiple policies. In order to demonstrate this and measure its effect on working set sizes, we implemented a `Composite` policy, where each composite tag is a tuple of tags from the individual policies discussed above.

5. PERFORMANCE EVALUATION

PUMP performance is a function of the tag and rule workloads created by policies such as those of §4. Our evaluation methodology is described in §5.1 and its results in §5.2.

5.1 Methodology

Processor Parameters. The parameters for the baseline and the experimental tagged processor are shown in Tab. 1. For the tag-extended processor, we implement 1024-entry L1 and 4096-entry L2 PUMP caches. We account for the longer wait cycles required for the tag-extended unified L2 cache and the main memory as well as L2 PUMP cache while calculating the runtime on the tag-extended processor.

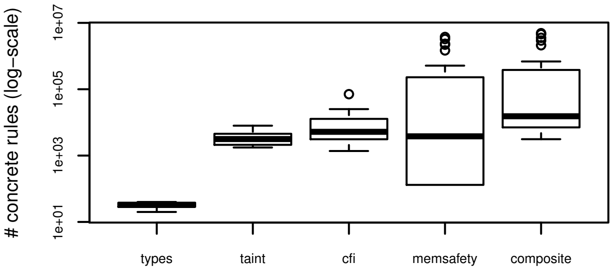

Figure 2: Concrete rules for SPEC CPU2006

Benchmarks. We estimate the performance impact of the PUMP architecture using 28 benchmarks from the SPEC CPU2006 Suite [20] (omitting `xalancbmk` and `tonto`, on which the `gem5` simulator fails) with reference inputs. We use the `gem5` simulation framework [6] to generate instruction traces for these programs for the baseline ALPHA ISA. Each benchmark is first run on an unmodified processor, with no tags or policies. Then we run the resulting trace through a PUMP simulator, which performs metadata computation for each instruction.[2] This approach is in the spirit of simulating various cache memory organizations against address traces generated by simulating programs [21]. The PUMP simulator also takes as input the policy to simulate on the application, and we assign initial metadata tags in the memory depending on the policy. We simulate each benchmark for a warm-up period of 1B instructions and then evaluate the next 500M instructions.

PUMP Miss-Handler Cost. As part of our evaluation, we account for the time required to service L2 PUMP misses in software. This time varies with the complexity of the policy being enforced, with the simplest taking a few hundred cycles and the most complex taking over a thousand cycles. We developed a model for the miss handler time based on parameters we collect from our trace simulator (*e.g.* number of used rule inputs, size of tag data structure, memory references needed to canonicalize a tag, whether or not a new tag must be allocated). The most complex single policy is taint tracking with large numbers of taint sources. The case where we taint each library in the source code requires on average 900 cycles to handle a miss. The composite case for all four policies takes close to 1100 cycles on average.

5.2 Runtime Performance

We evaluate the impact of our PUMP architecture on the runtime over a baseline execution to identify the architectural and microarchitectural impact due to our mechanism. At the architectural level, we want to identify the working set sizes and time spent in metadata computations in software and hardware, while at the microarchitectural level we want to observe the impact of smaller L1 caches, a wider L2 cache and the PUMP caches.

[2] In our experiments we simulate tag propagation throughout the execution, but do not enforce checking of rule violations.

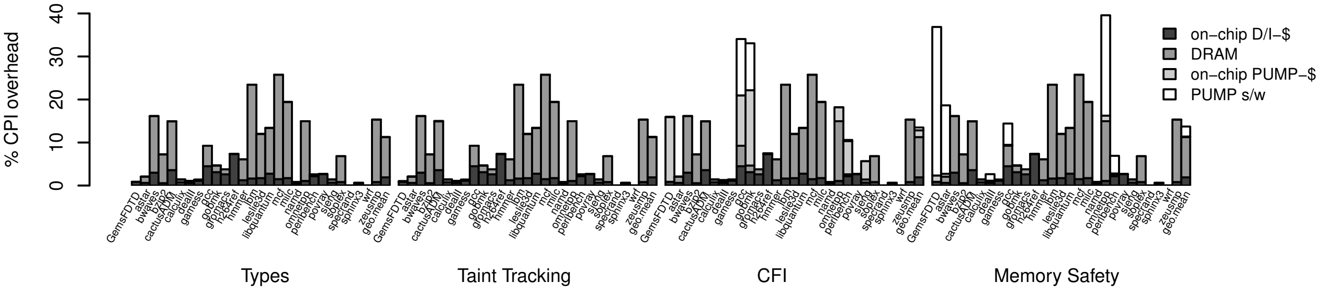

Figure 3: Runtime overhead for the policies in §4

Microarchitectural Impact. To keep the L1 D/I cache accesses within a single cycle in our tag-extended processor, we implement these with half the capacity of the baseline untagged design (leaving the other half for tag bits), potentially increasing the number of misses. The unified L2 cache has the same capacity but is twice as wide due to the tag bits, adding an extra cycle of latency. However, this has a small effect on the runtime—only 1.9% on average. In the tagged design, we also fetch twice as many bits from the main memory. This adds another 9.3% on average to the runtime over the baseline execution. Consequently, there is an average slowdown of about 11.2% due to our tag-extended, on-chip caches and main memory that impacts all programs before any policy is applied.

Policy Impact. Fig. 2 shows the range of concrete rules generated for the SPEC CPU2006 benchmark programs for all the policies from §4. There are two key observations to be made here: (1) different policies induce widely varying concrete rule sets based on the metadata computations, giving us enough variation across benchmarks to evaluate our mechanism, and (2) the number of concrete rules needed by the composite policy is close to the worst-case component policy. The size of rulesets, along with the *temporal locality* of these rules, determine the impact on runtime. We show the overall runtime overhead for the four individual policies in Fig. 3.

Primitive Types. For this simple policy there is no runtime overhead due to the PUMPs—the working set for this policy is small enough to fill the caches during the warm-up period and incur no cost in the evaluation period. The entire runtime overhead comes from the microarchitectural changes to the existing on-chip caches and the main memory.

Taint Tracking. For these experiments, we tainted each library in `glibc` and each input file descriptor with a unique taint label. The number of initial code taint sources varies from 33 to 35 across all benchmarks. While code executes, existing taints get aggregated, creating new taint values. We only see 159 unique taint tags in the worst case, `omnettpp`; `cactusADM` requires the largest number of concrete rules, at 7960, while `libquantum` uses the least, at 1752. On average (geometric mean) the runtime overhead from on-chip PUMP caches and the PUMP software handler is less than 1%.

CFI. With this policy we begin to see the effects of larger working sets for the PUMP. The `gobmk` and `gcc` benchmarks incur the highest overheads, 28% and 25%. For `gcc`, we generate 71003 concrete rules, and for `gobmk`, 16333 concrete rules, while the average number of concrete rules across all the benchmarks is 10371; `gobmk` spends relatively more time in the L1 PUMP cache misses that hit in the L2 PUMP cache, compared to `gcc`. The smallest number of rules is 1376 (for `specrand`, which incurs no runtime overhead due to the PUMP after the warmup period). On average there is an overhead of 1.6% and 1% from PUMP caches and the software miss handler, respectively, across all the benchmarks. Overall slowdown for the CFI policy due to the PUMP is close to 2.6% on average.

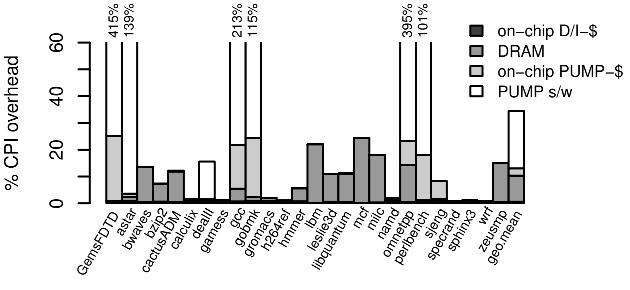

Figure 4: Composite Policy Runtime Overhead

Memory Safety. With this policy, we see a large range for the number of concrete rules (Fig. 2). In general, each new allocation generates a fresh tag resulting in a new set of related concrete rules for the instructions that use the newly tagged data. There are some benchmarks that require no dynamic memory allocation (such as `specrand`); Our worst-case applications, `GemsFDTD`, performs more than 550K allocations. Nonetheless, for each new tag the set of rules is usually a small number and exhibits a high degree of temporal locality. Consequently, there is a small overhead due to the on-chip PUMP caches—about 0.1% on average, while the PUMP software contributes close to 2.3% on average. In the worst case for `GemsFDTD` we see roughly 1.5% overhead in the on-chip PUMP caches and close to 35% overhead from the PUMP software.

Composite. Fig. 4 shows the impact on runtime due to the `Composite` policy described in §4. We see that in the worst case the overhead is close to 415% for `GemsFDTD` where almost 390% comes from the PUMP software. On average the overhead from on-chip PUMP caches is 2.7% and PUMP software is 21%, bringing the overall runtime overhead to about 35% on average. The maximum PUMP cache overhead is 25%, again for `GemsFDTD`.

Two effects make composite policies slower than individual policies: (1) they require more rules, creating greater cache pressure (*i.e.*, more misses), and (2) the miss handler takes more time to service them. As Fig. 2 shows, the total number of rules is not much greater than for the memory safety policy. For our worst-behaved composite policy (`GemsFDTD`), the maximum number of rules generated for the

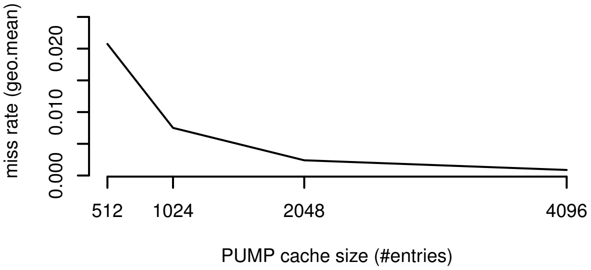

Figure 5: Impact of cache size on miss rate for the composite policy

composite policy is 4.9M, while the same benchmark requires about 3.75M rules when using memory-safety policy alone; so, even here, the total ruleset growth is modest. The big effect comes from (2), since the miss handler for the composite policy requires 1100 cycles, whereas the memory-safety miss handler requires only 150 cycles. As a result, the 11× overhead increase for the PUMP software (from 35% to 390%) for `GemsFDTD` is due to the 1100/150≈8× increase in miss handler service time and a 1.4× increase in the number of PUMP misses. Given the rate of compulsory misses due to new memory tags, the 8× increase due to miss-handler service time becomes the clear performance-limiting effect. We expect this cost can be reduced significantly with additional software tuning and microarchitectural support.

Fig. 5 shows how the miss-rate varies in relation to the capacity in a PUMP cache for the composite policy. We see that miss-rate falls with cache capacity, providing evidence that there is a high degree of locality in the concrete rules.

6. PRIOR HARDWARE-METADATA WORK

Due to space limitations, we cannot do proper justice to all work related to the specific policies we support in §4. Here, we discuss work specifically related to hardware tag checking and propagation. With a few exceptions noted below, most of the prior work uses a small set of tag bits with hardwired or highly restricted policies (See Tab. 2). The first wave of taint hardware supported a single taint bit attached to each word, with hardwired taint propagation logic. Later systems added the ability to handle multiple, independent taint tags [15], multiple bit tags [36], and more flexible policies [16]. None of these schemes attempt to address the flow of taint data through the PC (implicit flows [24]). The only design to support more than one policy at a time, Raksha, supported at most four policies [15], not an unbounded number as suggested by the 0-1-∞ design principle.

The prior systems closest to ours are Aries [8], Flexitaint [36], Log-Based Architecture (LBA) [11], and Harmoni [17], all of which propose programmable rule caches backed by software handlers. Only Flexitaint and LBA detail specific example security policies that use the programmable rule cache. In all cases except LBA, the rule cache is based on two inputs for the two operands of an operation and produce a single output, while the PUMP potentially takes up to five inputs and produces two outputs: Tab. 3 summarizes how these tag sources and destinations are used in our security policies. LBA potentially takes multiple inputs, but it does not handle production of metadata in hardware. Some of the innovations in [11] (restriction of general propagation tracking to unary inheritance tracking including giving up on taint combining) that made it fast specifically give up generality that our solution provides. Even with these restricted policies, LBA has 50% runtime overhead compared to our average single-policy runtime overheads of 10–20%. 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. The policies we have shown are also richer than those illustrated for LBA, while achieving lower runtime and energy overheads.

	HW/	Tag		Security
Ref.	SW	Size	IO	Policy
[12, 14, 27]	H	1b	2/1	hardwired taint
[34]	H+S	1b	2/1	hardwired taint
[10]	H+S	1b	2/1	limited prog.
[15, 23]	H+S	4b	2/1	four 1b policies; limited SW prog.
[36]	H+S	4b	2/1	limited prog.
[4]	H+S	1–8b	1/–	no propagation, mostly memory
[16]	H	var	2/1	FPGA reconfigurable
[17]	H+S	var	2/1	limited program. plus table
[11]	H+S	64b	5/-	SW on separate, augmented core
PUMP	H+S	64b	5/2	fine-grained, SW-defined policies

Table 2: Overview of Hardware Tagging Approaches

Policy	pc	ci	op1	op2	mr	pc	op3mw
Taint Tainting	✓	✓	✓	✓	✓	✓	✓
Memory Safety			✓	✓	✓		✓
Control Flow	✓	✓				✓	
Primitive Types			✓	✓	✓		✓

Table 3: Tag Usage by Policy

7. CONCLUSIONS AND FUTURE WORK

We have introduced a flexible and extensible policy model and a programmable unit for metadata processing (PUMP) and illustrated the new hardware's uses on a series of increasingly complex safety and security policies. Our benchmark simulations show that a PUMP-augmented conventional RISC processor architecture can support these policies with low typical runtime overhead.

However, while the initial evidence is promising, it also raises many questions that merit further attention. We believe the metadata policy model will be applicable to a large range of policies beyond those illustrated here, including sound information-flow control (*e.g.*, [31]), fine-grained access control (*e.g.*, [37]), integrity, synchronization (*e.g.* [5]), race detection (*e.g.*, [32]), debugging, application-specific policies (*e.g.*, [38]), and controlled dynamic code generation. We need to properly model area and energy costs and optimize the implementation to keep their overheads down to acceptable levels. Optimization of PUMP structures should also further reduce runtime overheads. In particular, with additional hardware support, we believe we can reduce the miss-handler overheads that play a significant role in the performance of composite policies for some benchmarks.

8. 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.

9. REFERENCES

[1] ARM Cortex-A5 Processor, http://www.arm.com/products/processors/cortex-a/cortex-a5.php.

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

[3] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proc. ACM CCS*, pages 340–353, 2005.

[4] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha. Architectural support for run-time validation of program data properties. *IEEE Trans. VLSI Sys.*, 15(5):546–559, May 2007.

[5] Arvind, R. S. Nikhil, and K. K. Pingali. I-structures: Data structures for parallel computing. In *Proc. Wkshp on Graph Reduction (Springer-Verlag LNCS 279)*, Sept. 1986.

[6] 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.

[7] E. Bosman, A. Slowinska, and H. Bos. Minemu: The World's Fastest Taint Tracker. In *Proc. RAID*, volume 6961 of *LNCS*, pages 1–20. Springer, 2011.

[8] 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.

[9] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. In *Proc. ACM CCS*, pages 27–38, Oct. 2008.

[10] H. Chen, X. Wu, L. Yuan, B. Zang, P.-c. Yew, and F. T. Chong. From Speculation to Security: Practical and Efficient Information Flow Tracking Using Speculative Hardware. In *Proc. ISCA*, pages 401–412, 2008.

[11] 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 *Proc. ISCA*, pages 377–388, 2008.

[12] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and R. Iyer. Defeating memory corruption attacks via pointer taintedness detection. In *Proc. IEEE DSN*, pages 378–387, 2005.

[13] M. L. Corliss, E. C. Lewis, and A. Roth. Using DISE to protect return addresses from attack. *SIGARCH Comput. Archit. News*, 33(1):65–72, Mar. 2005.

[14] J. R. Crandall, F. T. Chong, and S. F. Wu. Minos: Architectural support for protecting control data. *ACM Trans. Archit. and Code Opt.*, 5:359–389, December 2006.

[15] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: a flexible information flow architecture for software security. In *Proc. ISCA*, pages 482–493, 2007.

[16] D. Y. Deng, D. Lo, G. Malysa, S. Schneider, and G. E. Suh. Flexible and Efficient Instruction-Grained Run-Time Monitoring Using On-Chip Reconfigurable Fabric. In *Proc. IEEE MICRO*, pages 137–148, 2010.

[17] D. Y. Deng and G. E. Suh. High-performance parallel accelerator for flexible and efficient run-time monitoring. In *Proc. IEEE DSN*, pages 1–12, 2012.

[18] U. Dhawan and A. DeHon. Area-efficient near-associative memories on FPGAs. In *Proc. ACM TRETS*, 2014.

[19] E. Goktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *Proc. IEEE S&P*, 2014.

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

[21] M. A. Holliday. Techniques for cache and memory simulation using address reference traces. *Int. J. Comput. Simul*, 1:129–151, 1990.

[22] C. Hriţcu, M. Greenberg, B. Karel, B. C. Pierce, and G. Morrisett. All your IFCException are belong to us. In *Proc. IEEE S&P*, 2013.

[23] H. Kannan, M. Dalton, and C. Kozyrakis. Decoupling Dynamic Information Flow Tracking with a Dedicated Coprocessor. In *Proc. IEEE DSN*, pages 105–114, 2009.

[24] D. King, B. Hicks, M. Hicks, and T. Jaeger. Implicit flows: Can't live with 'em, can't live without 'em. In *Proc. ICISS*, pages 56–70, 2008.

[25] 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.

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

[27] M. Ozsoy, D. Ponomarev, N. B. Abu-Ghazaleh, and T. Suri. SIFT: a low-overhead dynamic information flow tracking architecture for SMT processors. In *Conf. Computing Frontiers*, page 37, 2011.

[28] D. A. Patterson and C. H. Sequin. RISC I: A Reduced Instruction Set VLSI Computer. In *Proc. ISCA*, pages 443–457, 1981.

[29] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and J. William S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *Proc. OSDI*, December 2004.

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

[31] A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proc. CSF*, pages 186–199, 2010.

[32] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic race detector for multi-threaded programs. *ACM Trans. Comp. Sys.*, 15(4), 1997.

[33] 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.

[34] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *Proc. ASPLOS*, pages 85–96, 2004.

[35] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal war in memory. In *Proc. IEEE S&P*, pages 48–62, 2013.

[36] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic. FlexiTaint: A programmable accelerator for dynamic taint propagation. In *Proc. HPCA*, pages 173–184, Feb. 2008.

[37] E. Witchel, J. Cates, and K. Asanović. Mondrian memory protection. In *Proc. ASPLOS*, pages 304–316, New York, NY, USA, 2002. ACM.

[38] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertions. In *Proc. SOSP*, October 2009.

[39] 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 *Proc. IEEE S&P*, 2013.