Programming the PUMP

Hardware-Assisted Micro-Policies for Security

ABSTRACT

A wide range of security policies can be formulated as rules on metadata at the ISA-level and enforced efficiently in programmable hardware. We elaborate a programming model for such policies based on the *Programmable Unit for Metadata Processing* (PUMP) architecture, which supports flexible rule evaluation on uninterpreted metadata alongside the main computation. We illustrate the model's generality by implementing a diverse set of safety and security policies of varying complexity, in four specific domains—spatial and temporal memory safety, taint tracking, control-flow integrity, and primitive typing. We characterize the performance of these policies for a simple RISC ISA, both singly and in combination. The average runtime overhead for most policies is only 8%. This shows that the PUMP model can achieve the flexibility and adaptability of software enforcement with the performance of dedicated hardware.

1. INTRODUCTION

It is far too easy for attackers to subvert a program's intent. Modern processors, designed to be agnostic to the intended high-level semantics of the operations they perform, are complicit in this state of affairs—a legacy of the technology era when transistors were expensive and the primary design goal was runtime performance. With computer systems increasingly entrusted with critical tasks, system security has finally become a key design goal. At the same time, processors are now small compared to even modest system-on-a-chip dies, making it feasible and inexpensive to augment them with security-enhancing hardware. For tomorrow's computers to adequately protect the privacy and integrity of the data they manage, we must re-architect the entire computing stack with security mechanisms consistent with modern threats and hardware costs.

The security literature offers a vast range of runtime policies that can reduce vulnerabilities due to malicious and erroneous code. These policies often encode high-level language abstractions (this is a numeric array, this is a code pointer, ...) or user-level security invariants (this string came from the network) into *metadata* annotations on the program's data and code. High-level semantics or policies are enforced by propagating this metadata as computation proceeds and dynamically checking for violations at appropriate points. We call these low-level, fine-grained enforcement mechanisms *micro-policies* (or informally just "policies").

Software realizations of micro-policies can define arbitrary metadata and arbitrarily powerful computations over them. Software implementation facilitates fast deployment of new policies, but it can be prohibitively expensive in terms of runtime and energy costs (1.5×–10×) [43], leading to unfavorable security-performance trade-offs. Simple micro-policies can be supported in hardware with low overhead [42, 1]; However, hardware customized to support a single policy can take years to deploy and is slow to adapt. Today's dynamic cyber-attack landscape calls for mechanisms that support rapid in-field responses to evolving threats.

The desire for greater flexibility has prompted a number of recent efforts to make policy-enforcement hardware more programmable [19, 46, 20, 14] (see §5). Here, we consider a design called the PUMP [8], a "Programmable Unit for Metadata Processing" that allows a wide range of low-level runtime policies to be defined in terms of instruction-grained computation on arbitrary metadata. At the hardware level, every word of data is associated with a word-sized *metadata tag*. These tags are not interpreted by the hardware; in software, they can be mapped to representations of information such as the type, provenance, classification level, or trustworthiness of the data to which they are attached. Since tags are large enough to represent pointers, they can refer to data structures of arbitrary size and complexity, including tuples of metadata, allowing multiple orthogonal policies to be enforced in parallel. The program counter is tagged to support tracking the history of the program's control state; program code is tagged to support policies on code provenance, control flow, and compartmentalization. The processor core is augmented with a *rule cache* that allows for high-performance rule resolution synchronously with instruction execution and a special operational mode for fast context switch to the policy handling code when lookups miss in this cache. This allows the PUMP to facilitate enforcement of a wide range of low-level policies with the expressiveness and adaptability of software and the performance of hardware.

Our goal in this paper is to show both that PUMP-like tagging and rule processing is *useful* against real threats and that writing policies in the form of rules is *tractable*. We do this by elaborating how the PUMP can be programmed to support a diverse collection of low-level security and safety policies. We present detailed implementations and evaluations of four families of policies (all familiar in the literature): (i) *primitive types*, enforcing a weak form of type safety; (ii) *spatial and temporal memory safety*, catching bounds and use-after-free errors for heap-allocated data, (iii) *control-flow integrity (CFI)* [3], preventing code-reuse attacks; (iv) *taint tracking*, where taints can represent data sources or components that may have contributed to a given piece of data. Most of these policies go beyond what current systems can efficiently support in software. Finally, we show how these policies can be applied *simultaneously*. Since these policies have been well-studied in the existing literature, our main focus is not on the security guarantees they provide, but rather on exploring how they can be expressed as rules and enforced with the PUMP. We use instruction trace simulations to estimate the runtime impact of these policies across the SPEC CPU2006 Benchmark Suite when the PUMP is attached to a simple, in-order RISC processor (an Alpha [2]). We show that the PUMP can support policies with a wide range of

complexities and quantify the performance impacts. This range illustrates the ability to refine the policies as threats evolve and how this evolution may impact performance.

This paper is an extended, enriched, and refocused version of [8], a short paper to be presented at a workshop later this summer. The previous paper focuses on a straightforward hardware integration of the PUMP into a RISC processor, establishes reasonable performance on most benchmarks, and identifies areas for improvement. In the present work we eschew microarchitectural considerations, which are well explained in [8], focusing instead on the programming model and on a much more detailed explanation and evaluation of the policies themselves. We also explain how the PUMP software services protect themselves from abuse. The performance we report improves on [8] due to: (i) the use of opgroups (§2), (ii) a more accurate estimation of miss costs (§3) and, (iii) the reduction of DRAM accesses by using pointer tags only where needed (§4).

In summary, the main contributions of this work are (i) a programming model and supporting interface model for compactly and precisely describing policies supported by this architecture (§2 and §3); (ii) detailed examples of policy encoding and composition using four diverse classes of well-studied policies; and (iii) quantification of the requirements, complexity, and performance for these policies (§4). In §5 and §6, we discuss related and future work. Several additional materials are available in anonymized form at `http://git.io/8K7lKA`. These include: an appendix with complete definitions for the studied policies, the source code of our experiments, and an anonymized version of [8].

2. POLICY PROGRAMMING MODEL

A *PUMP policy* consists of a set of *tag values* together with a collection of *rules* that manipulate these tags to implement some desired tracking and enforcement mechanism. 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.

Example. To illustrate the operation of the PUMP, let's consider a simple example policy for restricting return points during program execution. The motivation for this policy comes from a class of attacks known as *return-oriented programming (ROP)* [40], where the attacker identifies a set of "gadgets" in the binary executable of the program under attack and uses these to assemble complex malicious behaviors by constructing appropriate sequences of stack frames, each containing a return address pointing to some gadget; a buffer overflow or other vulnerability is then exploited to overwrite the top of the stack with the desired sequence, causing the snippets to be executed in order.

One simple way of limiting ROP attacks is to constrain the targets of `return` instructions to well-defined return points. We can do this using the PUMP by tagging instructions that are valid return points with a metadata tag `target`. 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, we notice that the PC tag is `check`, verify that the tag on the current instruction is `target`, and signal a security violation if not. We will see later in this section that, by making the metadata richer, we can precisely control which `return` instructions can return to which return points. By making it yet richer, we can implement full-blown CFI checking [3] (see §4.3).

Symbolic Rules. From the point of view of the policy designer and the software parts of the PUMP, policies are compactly described using *symbolic rules* written in a tiny domain-specific language. Each symbolic rule has the form:

$$\mathit{opgroup}: (PC, CI, OP_1, OP_2, MR) \to (PC', R') \text{ if } \mathit{guard}?$$

which says that the rule matches on a set of instruction opcodes (*opgroup*) together with the metadata tags on the program counter (PC), the current instruction (CI), up to two operands from the register file (OP_1, OP_2), and the memory location referenced by the instruction (MR), if any. The rule applies if all relevant tag expressions match and the *guard*? predicate holds. In this case, the right-hand side determines how to update the tags on the PC (PC') and on the result of the operation (R'). We use op*groups* instead of opcodes since, in most policies, there will be many opcodes with identical rules. We write "$-$" to indicate input or output fields that are ignored ("wildcard"). When the *guard*? condition is just `true`, we elide it.

For the simple ROP policy just sketched, we split the opcodes into two opgroups—`return` (containing just a single opcode) and $\overline{\texttt{return}}$ (all the rest); the possible tag values are `check`, `target`, and $\bot$. The PC will always be tagged either `check` or $\bot$, and each instruction will be tagged either `target` or $\bot$. (Instruction tags are supplied by a trusted loader; see §3.) The symbolic rules are:

$$\texttt{return}: (\bot, -, -, -, -) \to (\texttt{check}, -) \quad (1)$$

$$\overline{\texttt{return}}: (\texttt{check}, \texttt{target}, -, -, -) \to (\bot, -) \quad (2)$$

$$\overline{\texttt{return}}: (\bot, -, -, -, -) \to (\bot, -) \quad (3)$$

$$\texttt{return}: (\texttt{check}, \texttt{target}, -, -, -) \to (\texttt{check}, -) \quad (4)$$

Rule 1 says that, when the current operation is a `return` (and the PC is not already tagged `check`), we change the tag on the PC to `check`. When we run an instruction with the PC tagged `check` (Rule 2), we check that the instruction tag, CI, is `target`; if so, we allow the operation and clear the tag on the PC. If the current operation is not a `return` and the PC tag is $\bot$, we simply proceed (Rule 3). Rule 4 handles the special case where a valid target of a `return` is itself a `return`. If no rule applies, the operation is not allowed (e.g., the configuration $PC = \texttt{check}$ and $CI = \bot$ is not allowed). We assume that the symbolic rules do not overlap.

Next, let's consider a more precise variant of this policy, where we make sure not only that every return reaches *some* valid return target, but that it targets a code point from which it could actually have been called. This policy assumes that the compiler has full knowledge of return points and can analyze, for each one, which call sites it could potentially return to. Using this information, we can attach a unique tag to each `return` and to each potential return target. Upon encountering a `return`, the PUMP copies the tag on the instruction (rather than the generic tag `check`) onto the PC (Rules 1' and 4'). On the next step, it checks that the actual return point is among the expected ones—i.e., that a return from PC to CI is allowed (Rules 2' and 4').

$$\texttt{return}: (\bot, ci, -, -, -) \to (ci, -) \quad (1')$$

$$\overline{\texttt{return}}: (pc, ci, -, -, -) \to (\bot, -) \text{ if } (pc, ci) \in \chi \quad (2')$$

$$\overline{\texttt{return}}: (\bot, -, -, -, -) \to (\bot, -) \quad (3')$$

$$\texttt{return}: (pc, ci, -, -, -) \to (ci, -) \text{ if } (pc, ci) \in \chi \quad (4')$$

In these rules we use χ (a set of pairs of code location identifiers provided by the compiler) to denote the allowed indirect control flows via `return` in the code. As shown here, the expressions describing tags in symbolic rules are not limited to constant values: we can write more general expressions that compactly describe large sets of tags.

Concrete Rules. Symbolic rules can compactly encode a great variety of metadata tracking mechanisms. At the hardware level, however, we need a rule representation that is tuned for efficient interpretation to avoid slowing down the primary computation. To this end, we introduce a lower-level rule format called *concrete rules*. Intuitively, each symbolic rule for a given policy can be expanded into an equivalent set of concrete rules. However, since a single symbolic rule might in general generate an unbounded number of concrete rules, we perform this elaboration *lazily*, generating concrete rules as needed while the system executes.

The PUMP hardware includes a *cache* of concrete rules that can be consulted in parallel with the processor's ALU operations. When an instruction is issued, the rule cache performs an associative match of the tags from the current machine state (the current PC tag, tags on the operands of the current instruction, etc.) against all the concrete rules in the cache. If a match is found, the cache returns the new tag for the PC and a tag for the instruction's result. Otherwise, the processor faults to a *rule miss handler*—a software routine that consults the symbolic rules of the policy and determines whether the faulting machine state should be allowed to proceed; if so, it generates an appropriate concrete rule, installs it in the cache, and restarts the faulting instruction. Otherwise, it invokes a suitable security fault handler. The general format for concrete rules is:

$$\mathit{opgroup}: (PC, CI, OP_1, OP_2, MR) \Rightarrow (PC', R')$$

where the input and output fields are fixed tags. Note that the "*guard*?" field in the symbolic rule format is not needed, since the miss handler checks the corresponding condition before adding any concrete rules into the cache.

One handy encoding trick greatly reduces the number of concrete rules. We observe that it is very common for all the symbolic rules for a given opgroup to mark a particular input or output as "wildcard." For example, in our ROP policy, the rules for the `return` and $\overline{\texttt{return}}$ opgroups do not need to match on the OP_1, OP_2 and MR inputs and do not need to produce an R' result. To avoid generating concrete rules for all possible values of the unused input fields, we define a bit vector containing a *don't-care bit* for each opgroup and input field, which determines whether the corresponding tag is actually used in the rule cache lookup. Similarly, the don't-care vector marks unused outputs, for which a default tag is returned (below we use $\bot$ for this).

For example, since for the ROP policy the $\overline{\texttt{return}}$ opgroup has don't-care bits set for OP_1, OP_2, MR, and R', Rule 2' results in just two concrete rules

$$\overline{\texttt{return}}: (t_1, t_2, \bot, \bot, \bot) \Rightarrow (\bot, \bot)$$
$$\overline{\texttt{return}}: (t_1, t_3, \bot, \bot, \bot) \Rightarrow (\bot, \bot)$$

if the compiler knows that the `return` instruction tagged t_1 is the only `return` in the code and it can only return to the return targets tagged t_2 and t_3. The "don't-care" positions were masked to $\bot$. On the other hand, symbolic rule 3' corresponds to four concrete rules:

$$\overline{\texttt{return}}: (\bot, \bot, \bot, \bot, \bot) \Rightarrow (\bot, \bot)$$
$$\overline{\texttt{return}}: (\bot, t_1, \bot, \bot, \bot) \Rightarrow (\bot, \bot)$$
$$\overline{\texttt{return}}: (\bot, t_2, \bot, \bot, \bot) \Rightarrow (\bot, \bot)$$
$$\overline{\texttt{return}}: (\bot, t_3, \bot, \bot, \bot) \Rightarrow (\bot, \bot)$$

Since CI is not a "don't-care" position for $\overline{\texttt{return}}$ (while Rule 3' does mark CI as a wildcard, Rule 2' does not, and both rules are about the same opcode), we get a different concrete rule for each of the possible values it can take—$\bot$ plus all identifiers (in this example, just t_1, t_2 and t_3).

The mapping from opcodes to opgroups and don't-care vectors is programmable. The ROP policy uses only two opgroups (`return` and $\overline{\texttt{return}}$), but other policies may need more; for example, the primitive types policy (§4.1) uses ten.

Structured Tags. For policies with richer metadata tags than ROP, the translation from symbolic to concrete rules follows the same general lines, but the details become a bit more intricate. For example, the taint-tracking policy (§4.4) takes tags to be *pointers* to memory data structures, each describing an arbitrarily sized set of taints (representing data sources or system components that may have contributed to a given piece of data). The symbolic rule for the `load` opgroup says that the taint on the loaded value should be the union of the taints on the instruction itself, the target address for the load, and the memory at that address:

$$\texttt{load}: (-, ci, op_1, -, mr) \rightarrow (-, ci \cup op_1 \cup mr)$$

Suppose that, at some moment, (i) the next instruction to be executed is `ld r0 r1` and its tag is t_{ci}, register `r0` contains a pointer p tagged t_p, and the memory at address p contains a value tagged $t_\varnothing$; (ii) t_{ci} points to a data structure (an array of taint ids, say) representing the set $\{T_A, T_B\}$; (iii) t_p points to a representation of $\{T_C, T_D\}$; and (iv) $t_\varnothing$ points to the empty set. Furthermore, suppose that we have never before encountered the taint $\{T_A, T_B, T_C, T_D\}$— *i.e.*, there is currently no data structure in memory that represents the set that we should use to taint the result of the load. In this case, the rule cache lookup will miss and execution will fault into the rule miss handler, which will generate an appropriate concrete rule and install it in the cache, perhaps evicting another rule to make space. This will require allocating new memory (say, at address t_{new}) and initializing it to represent $\{T_A, T_B, T_C, T_D\}$. The generated concrete rule will then be:

$$\texttt{load}: (\bot, t_{ci}, t_p, \bot, t_\varnothing) \Rightarrow (\bot, t_{new})$$

After the instruction is restarted, the next cache lookup will succeed, and the loaded value in `r1` will be tagged t_{new}.

To reduce the number of distinct tags (and, hence, pressure on the rule cache), metadata structures are internally stored in canonical form and since tags are immutable sharing is fully exploited (e.g., set elements are given a canonical order so that sets can be compactly represented sharing common prefix subsets). When no longer needed, these structures can be reclaimed (e.g., by garbage collection).

Composite Policies. Going one step further, we can simultaneously enforce multiple orthogonal policies by letting tags be *pointers to tuples* of tags from several component policies. (In general, multiple policies may not be orthogonal; we return to this point in §6.) For example, to compose the first ROP policy with the taint-tracking policy we've just

sketched, we would let each tag be a pointer to a representation of a tuple (r, t), where r is an ROP-tag (a code location identifier or $\perp$) and t is a taint tag (a pointer to a set of taints). The cache lookup process is exactly the same, but when a miss occurs the miss handler extracts the components of the tuple and dispatches to routines that evaluate both sets of symbolic rules. The operation is allowed only if both policies have a rule that applies; in this case the resulting tag is a pointer to a pair containing the results from the two sub-policies.

Instruction Modifiers and Ephemeral Rules. Some policies (e.g., memory safety) require fresh tags to be generated dynamically. One way to achieve this effect is to use the tag on an instruction such as `move` as a modifier to communicate a request for a fresh tag to the policy management system.

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

This says that a `move` instruction tagged with $t_{policygen}$ is interpreted as a request to generate a fresh tag. The result, t_{newtag}, is a unique tag associated with the specified policy. The tag on the instruction, $t_{policygen}$, also serves as an authorization or capability for this service request; without that tag, it is not possible to make the call; the trusted loader ensures that only specially designated code regions (e.g., the `malloc` routine, in the memory safety policy in §4.2) are annotated with this tag. The "1" indicates an *ephemeral* rule, whose result is not persistently stored in the hardware rule cache (since it changes on every invocation).

Code for initializing tags may also need to override the "steady-state" rules. For example, in the memory safety policy, `malloc` will need to initialize the tags on the newly allocated memory region. The standard rule is that a pointer can only write into a memory region that is suitably tagged to match the pointer. But `malloc` must be allowed to override this rule while writing the newly minted tag onto each word in the new region. We do this by giving the `store` operation a special modifier tag (used only in `malloc`):

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

3. POLICY SYSTEM AND PROTECTION

The policy system exists as a separate region of memory within each user process. It includes the code for the miss handler, the policy rules, and the data structures representing the policy's metadata tags. Placing the policy system in the process is minimally invasive with the existing Unix process model and facilitates lightweight switching between the policy system and the user code. The policy system is isolated from user code using mechanisms described next.

Metadata Threat Model. Clearly, the protection offered by the PUMP would be useless if the attacker could rewrite metadata tags or change their interpretation. Our system is designed to prevent such attacks. We trust the kernel, loader, and (for some policies) compiler. In particular, we depend on the compiler to assign initial tags to words and, where needed, communicate rules to the policy system. We assume the loader will preserve the tags provided by the compiler, and that the path from the compiler to the loader is protected from tampering, e.g., using cryptographic signatures. We assume a standard Unix-style kernel, which sets up the initial memory image for each process. (It may be possible to use micro-policies to eliminate some of these assumptions, further reducing the size of the TCB—see §6.) We further assume that the rule-cache-miss-handling software is correctly implemented. This is small, hence a good target for formal verification; recent work [9] demonstrates feasibility for a programming model similar to the PUMP.

Our primary concern is to prevent user code running in a process from undermining the protection provided by the process's policy. User code should not be able to (i) manipulate tags directly—all tag changes should be performed in accordance with the policy rules currently in effect; (ii) manipulate the data structures and code used by the miss handler; (iii) directly insert rules in the hardware rule cache.

Addressing. To prevent direct manipulation of tags by user code, the tags attached to every 64b word are not, themselves, separately addressable. In particular, it is not possible to specify an address that corresponds only to a tag or a portion of a tag in order to read or write it. All user-accessible instructions operate on (data,tag) pairs as atomic units—the standard ALU operating on the value portion and the PUMP operating on the tag portion.

Miss-Handler Architecture. The policy system is only activated on misses to the PUMP cache. To provide isolation between the policy system and user code, we add a miss-handler operational mode to the processor; we also expand the integer register file with 16 additional registers that are available only to the miss handler, to avoid saving and restoring registers. The PC of the faulting instruction, the rule inputs (opgroup and tags), and the rule outputs appear as registers while in miss handler mode. We also add a `miss-handler-return` instruction, which finishes installing a concrete rule into the cache and returns to user code.

The normal behavior of the PUMP is disengaged while the processor is in miss-handler mode. Instead, a single hardwired rule is applied: all instructions and data touched by the miss handler must be tagged with a predefined `miss-handler` tag that is distinct from the tags used by any policy. This ensures isolation between miss handler code and data and the user code in the same address space. User code cannot touch or execute policy system data or code, and the miss handler cannot accidentally touch user data and code. The `miss-handler-return` instruction can only be issued in miss-handler mode, preventing user code from inserting any rules into the PUMP.

4. POLICIES AND EXPERIMENTS

In this section, we show how to use the PUMP to implement four families of policies enforcing a diverse set of security invariants. For each family, we first sketch a threat model. We then describe policies and corresponding rules that mitigate it. Using examples from a public vulnerability suite [11], we show how each policy would catch a typical exploit. Most importantly, we describe the loads that each policy puts on the system. We close by comparing with similar policies from the literature.

To evaluate policy loads, we use 28 C, C++, and Fortran applications from the SPEC CPU2006 [26] benchmark suite and simulate them for a 64-bit Alpha ISA [2] with the `gem5` simulation environment [10] (we exclude the `tonto` and `xalancbmk` benchmarks, on which `gem5` fails). The `gem5` simulation does not directly model the PUMP; rather, it produces instruction traces that we run through a separate

	none		Types		Memory Safety					CFI						Taint Tracking						∪	
	Ⓐ no miss handler	Ⓑ trivial handler	Ⓒ `insn/addr/other`	Ⓓ `i/a/o + retaddr`	Ⓔ sandboxing	Ⓕ 1 color	Ⓖ 8 colors	Ⓗ 32 colors	Ⓘ full protection (2^{64} colors)	Ⓙ 1ID [3]	Ⓚ 2ID [3]	Ⓛ CCFIR [52]	Ⓜ PUMP JOP	Ⓝ PUMP ROP	Ⓞ PUMP CFI (Ⓜ,Ⓝ)	Ⓟ 1 taint	Ⓠ per stream	Ⓡ per library	Ⓢ per file	Ⓣ per function	Ⓤ per stream & lib (Ⓠ,Ⓡ)	Ⓥ minimal comp. (Ⓒ,Ⓕ,Ⓛ,Ⓟ)	Ⓦ complete comp. (Ⓓ,Ⓘ,Ⓞ,Ⓤ)
tag usage	-	-	~~pc~~		~~pc~~	~~ci~~	op1	op2	mr	~~pc~~	ci	~~op1~~	~~op2~~	~~mr~~	pc	~~r~~	~~pc~~	ci	op1	op2	mr	~~pc~~	r	all	
opgroups	-	-	10	11	7					2	3	3	2				7						13	
symbolic rules	-	-	15	16	13					2	3	4	4				8						49	53
initial tags	1	-	3	4	34	6	6	6	6	2	4	6	351	1.7K	2.1K	2	3	34	371	1.2K	36	16	2.2K
dyn. alloc. tags	0	-	0	0	2	2	18	241	102K	0	0	0	0	0	0	0	1	76	381	2.3K	77	4	252K
final tags	1	-	3	4	36	8	26	247	102K	2	4	6	351	1.7K	2.1K	2	4	110	752	2.5K	113	20	255K
concrete rules	1	-	17	19	14	18	93	578	386K	5	11	21	426	1.8K	2.3K	10	14	1.1K	6.5K	47K	1.2K	79	1.1M
metadata struct	-	-	-	-	2	2	2	2	2	-	-	-	-	-	-	1	2	5	13	50	5	4	5
metadata space	-	-	0	0	2	2	24	418	51K	0	0	0	2K	26K	28K	3	9	510	13K	573K	517	76	652K
policy-depend instrs	0	10	116	136	64					24	44	64	682			810						1240	1800
(dynamic)	0	10	25	26	20	20	32	36	38	16	26	31	117	120	120	56	174	430	1076	4110	430	211	710
runtime overhead (%)	8	8	8	8	8	8	8	8	13	8	8	8	8	8	8	8	8	8	9	314	8	8	38

Figure 1: Policies and Characteristics (average across 28 SPEC CPU2006 benchmarks)

PUMP simulator. This phased simulation is sufficient for the policies described here, since their only impact on the computation is to abort execution when a policy violation occurs. We simulate a 4096-entry pre-miss-handler rule cache.

The abstract programming model described in §2 places no limits on the number of unique tags, the number of concrete rules, or the size of the data structures used to represent metadata at the software level. To understand how the PUMP performs in practice, a number of questions must be considered. How many unique metadata tags do a given policy, application, and dataset actually generate? With O opgroups and T tags, in theory a program could require $O \cdot T^5$ concrete rules, but what is the typical case? How do the total number of metadata tags and the size of metadata representations impact performance? How much locality is there to tag and rule usage? How costly is concrete rule resolution, and how do rule cache misses impact performance? Does performance degrade gracefully as tags, rules, metadata size, or rule resolution time increase?

To begin to understand these effects, we measure, for each of the policies, a number of characteristics besides runtime overhead—see Fig. 1. *Tag usage* shows which tags are not used by any of the rules in the policy. *Opgroups* is the minimum number of opgroups needed to capture the policy; the fewer opgroups we use, the greater compression we get for concrete rules and hence the greater is the effective PUMP capacity. *Symbolic rules* is the number of symbolic rules we wrote to express the policy. *Initial tags* is the number of tags in the initial memory image before execution begins. During execution more tags will be dynamically allocated (*dyn. alloc. tags*). Furthermore, policies like taint tracking will create tags to represent unions of sets of taints, and composite policies will form tuples of individual policy tags. *Final tags* identify the number of tags that exist at the end of the one billion instruction simulation period; this gives some sense of policy complexity and can be used to infer the rate of tag creation. *Concrete rules*, the number of unique concrete rules generated during the simulation period, characterizes the number of compulsory misses needed to resolve symbolic rules to concrete rules and, effectively, the compulsory miss rate. *Metadata struct*, the average size in words of the data structure pointed to by each tag, illustrates the value of having unbounded metadata. *Metadata space*, the number of words required for all of the data structures holding policy-related information to which the metadata tags point, characterizes the memory overhead beyond the tags themselves. *Policy-depend instrs* is the total number of instructions required for the code that resolves symbolic rules to concrete ones; this is useful in understanding the complexity of the policy. *Policy-depend instrs (dynamic)* is the average number of policy-dependent instructions executed to resolve from a symbolic rule to a concrete rule; this is indicative of the runtime complexity of the miss handler for each of the policies. The impact of the policy-dependent portion depends on the complexity of the rules, the metadata data structures, the locality of the metadata data structures, and the need to allocate new result tags. The policy-independent part of the miss handler requires only a few tens of instructions (see column Ⓑ in Fig. 1). *Runtime overhead* is the ratio of wall-clock runtime for the application running the policy compared to a baseline Alpha with no PUMP. There is some runtime overhead just for adding hardware structures for tags and PUMP even if no policy is used. In particular, the L1 caches on the tag-augmented processor are half the effective capacity of the PUMP-less baseline Alpha in order to achieve the same cycle time while accommodating the larger tagged word width. This results in a higher L1 miss rate for the tag-augmented processor. This overhead is captured in the first column (Ⓐ) where all tags are default, there is a single rule, and the miss handler is effectively never invoked.

The average numbers in Fig. 1 are necessarily simplifications for compactness. The benchmarks exhibit a range of effects. These are shown in Fig. 3–6 where we use boxplots to show the distribution of characteristics across the applications in the SPEC CPU2006 benchmark set. Fig. 6 plots runtime overhead in excess of Ⓐ.

We measure only *runtime* performance, leaving aside some other nontrivial costs. In particular, in a naive implementation, adding a word-sized tag to every word of cache and memory imposes a minimum of 2× area overhead. Adding the impact of PUMP caches and larger memories, this can translate into 4× energy overhead. We are optimistic that

careful optimizations can reduce these numbers to around 30% area and 50% energy, or perhaps even lower; we are working to demonstrate this claim.

4.1 Primitive Types

Threat Model. Data misinterpretation is a common way to trick processors into performing unintended operations. Here we are concerned with a form of low-level type confusion where code running on behalf of an adversary can try to use any data value as a pointer or execute a word as an instruction. we enforce that data cannot be executed and code cannot be created or modified at run time (see also §4.3).

Policy and Rules. In policy Ⓒ we use tags to separate instructions (tagged `insn`), addresses (`addr`), and all other data (`other`). Instructions cannot be created or modified, and only instructions can be executed. Only addresses can be used with memory access instructions. The `other` type tag is used as a catch-all for words that are not instructions or addresses. The following rule validates that a `nop` (for example) is indeed tagged `insn` before it is executed:

$$\texttt{nop}\ :\quad (-, \texttt{insn}, -, -, -) \rightarrow (-, -) \tag{5}$$

Address arithmetic is allowed—for instance, when one of the arguments to `add` is an address the result is an address:

$$\texttt{add}\ :\quad (-, \texttt{insn}, \texttt{addr}, \texttt{other}, -) \rightarrow (-, \texttt{addr}) \tag{6}$$

We also enforce that load and store instructions dereference only pointers, and do not read or write instructions:

$$\texttt{load}\ :\quad (-, \texttt{insn}, \texttt{addr}, -, t) \rightarrow (-, t) \text{ if } t \neq \texttt{insn} \tag{7}$$

$$\texttt{store}\ :\quad (-, \texttt{insn}, t, \texttt{addr}, -) \rightarrow (-, t) \text{ if } t \neq \texttt{insn} \tag{8}$$

To help prevent attacks where a return address is overwritten (e.g., through stack smashing), we consider an extended policy (Ⓓ) that adds a forth tag for return addresses (`retaddr`). We use this to tag the return address of calls (Rule 9). Calls in the Alpha ISA put the return address in `reg26`, while a return transfers control to the address in this register (the register is spilled to the stack on further calls). Rule 10 checks that the value in `reg26` is typed `retaddr` when the `return` instruction is executed.

$$\texttt{call}\ :\quad (-, \texttt{insn}, \texttt{addr}, -, -) \rightarrow (-, \texttt{retaddr}) \tag{9}$$

$$\texttt{return}:\quad (-, \texttt{insn}, \texttt{retaddr}, -, -) \rightarrow (-, -) \tag{10}$$

An instrumented compiler could infer these type tags and apply them to the initial memory image of a binary—all the generated instructions get tagged `insn`, pointers to stack-allocated memory get tagged `addr`, and everything else gets tagged `other`; new `addr`-typed words come into existence through dynamic memory allocation. However, since we currently do not have such a compiler, we use a different method to deduce these tags for our simulations and analyses. First, we tag all the instructions in the binary executable `insn`. To deduce words that should tagged `addr`, we use an after-the-fact analysis of the execution trace, keeping track of when and from where each register is loaded and whether it is later used as the pointer operand to a load or store. Everything else is tagged `other`. This method of obtaining the inital tags allows us to measure the runtime impact of the typing policies on the SPEC benchmarks. However, this setup does not allow us to make any claim about whether our typing policy would be premissive enough to accept all the benchmarks without raising unnecessary alarms. This is caused by the tight compiler integration needed for typing, and does not occur for the other policies we present below.

Protection Demonstration. We use an instance of CWE-843 (Type Confusion) [31] in which the programmer typecasts an integer to a function pointer and later invokes this function. This translates into loading an immediate value tagged as `other` into a register, and, at a later point, jumping to the address pointed to by that register. Using policy Ⓒ we are able to catch the faulting instruction since the policy allows indirect jumps only to values tagged `addr`.

Characteristics. Policies Ⓒ and Ⓓ do not create new tags. Ⓒ can be encoded with 15 symbolic rules that generate only 17 concrete ones, while Ⓓ requires 16 symbolic rules and 19 concrete ones. Since the total number of rules is small, we only see a negligible runtime overhead (less than 0.01% compared to the no-miss-handler policy Ⓐ). Thus, the PUMP provides the performance of simple, hard-wired type tags, without baking the policy into hardware.

Related Work. One of the first uses of tags in computer architectures was to distinguish the types of the words in the machine [35, 24]. The Symbolics LISP Machines [32] allocated 2–8b for tagging out of their 36b primitive word to distinguish a set of primitive types including instructions, several flavors of pointers, integers, floats, and uninitialized values; the Berkeley SPUR [44] used a 6b object-type tag.

4.2 Spatial and Temporal Memory Safety

Threat Model. The next group of policies target the memory safety of heap-allocated data, preventing attackers from exploiting programming errors such as referencing beyond an object's bounds (spatial violation), referencing through a pointer after the region has been freed, or freeing an invalid pointer (temporal violation). This includes typical heap-based attacks such as heap smashing and pointer forging. The policies we study here only guard heap-allocated data, for which calls to `malloc` and `free` tell us how to set up and tear down memory regions; we do not deal with stack allocation or unboxed structs. These could in principle also be handled, assuming some compiler support (see [33]).

Policy and Rules. Intuitively, for each new allocation we make up a fresh *block id*, say c (for "color"), 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.

We use an additional tag $\perp$ for non-pointers, and write t for a tag that is either a color c or $\perp$. We take care of one additional detail—memory cells may contain pointers. So a 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. 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):

$$\texttt{load}\ :\quad (-, -, c_1, -, (c_2, t_2)) \rightarrow (-, t_2) \text{ if } c_1 = c_2 \tag{11}$$

$$\texttt{store}: \quad (-,-,t_1,c_2,(c_3,t_3)) \rightarrow (-,(c_3,t_1)) \text{ if } c_2 = c_3 \tag{12}$$

Address arithmetic operations preserve the pointer tag:

$$\texttt{add}: \quad (-,-,c,\bot,-) \rightarrow (-,c) \tag{13}$$

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 $\bot$.

We augment `malloc` and `free` to tag memory regions using the instruction modifiers and ephemeral rules described at the end of §2. In `malloc` we generate a fresh tag for the pointer to the new region via 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

$$\texttt{store}: \quad (-,t_{mallocinit},t_1,c_2,F) \rightarrow (-,(c_2,t_1)) \tag{14}$$

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

$$\texttt{store}: \quad (-,t_{freeinit},t_1,c_2,(c_3,t_4)) \rightarrow (-,F) \tag{15}$$

before returning the memory region to the free list.

We implemented several variants of this policy, illustrating different performance/security tradeoffs. In the first (Ⓔ), we assign a single color to all memory regions allocated by a given source module. This sandboxing policy provides per-module isolation within a process, similar to software-based fault isolation [47]. In the next variants we use different numbers of colors to tag regions returned by successive calls to `malloc`—from just a single color (Ⓕ)—this provides the weakest form of spatial and temporal memory safety, only distinguishing allocated from unallocated memory—to 8 (Ⓖ) and 32 (Ⓗ) colors. Increasing the number of colors reduces the *aliasing* effect that arises due to re-use of colors. Finally, we implement a precise full memory safety policy (Ⓘ), using the entire 64-bit tag space for colors.

Protection Demonstration. We use two attacks from the Juliet suite [11]. The first is a case of CWE-416 (Use After Free) [29] where the application is caught using policy Ⓘ trying to load from a memory location tagged F. The second is a case of CWE-122 (Heap-Based Buffer Overflow) [28] in which a buffer is allocated and later written beyond its bounds (using `strcpy`), overwriting a valid region. Using Ⓘ, the PUMP halts the instruction that tries to put a character in a memory location tagged F.

Characteristics. Sandboxing (Ⓔ) and the policies with a small number of colors (Ⓕ, Ⓖ, and Ⓗ) only allocate a few tags and create a small number of rules (less than 600 for the 32-color case). These do not add runtime overhead—the rules all fit in the cache. Full memory safety (Ⓘ) is more expensive: it allocates one tag per memory allocation, for which new concrete rules must be added to the cache. This requires more trips through the miss handler and means that, in some of the benchmarks, the set of concrete rules is bigger than the cache. Nonetheless, rule locality is high (See Fig. 7), and the average runtime overhead is only 13%. We see the largest overhead of about 130% for `GemsFDTD`.

Related Work. Clause et al. [17] first demonstrated spatial and temporal memory protection using metadata tainting. Deng et al. [20, 21] supported this tainting with hardware tag management. HardBound [22] is an approach to spatial memory safety that places the bounds information in a shadow space to maintain data structure layout compatibility between monitored and unmonitored code. HardBound's runtime overheads are 10–20%. Watchdog [33] is a follow-up of HardBound that additionally prevents temporal violations by generating a unique identifier for each allocation; it has 24% average runtime overhead. SoftBound [34] is a software approach that, like HardBound, provides spatial memory safety for C, but at a cost of increased runtime overhead (67% on SPEC and Olden benchmarks). Baggy Bounds [4] also targets only spatial violations and achieves 60% runtime overhead on SPEC2000.

4.3 Control-Flow Integrity

Threat Model. This group of policies targets code-reuse attacks. We make the standard assumption [3] that the attacker can neither execute data nor inject or modify code. (We can use the primitive types policy from §4.1 to enforce this assumption, as we do in §4.5 with our composed policies.) Instead, the attacker tries to chain together existing code snippets (gadgets) to induce malicious behavior.

Policy and Rules. A common element of all code-reuse attacks is to introduce control flows that do not exist in the original binary. We implement a family of CFI policies that validate each indirect control flow (computed jumps against the program's control-flow graph. Since the code is fixed, direct jumps do not need to be checked dynamically [3]. First we implement the coarse-grained CFI policies of [3, 52] (Ⓙ, Ⓚ and Ⓛ). Ⓙ tags all indirect call, indirect jump, and return instructions and their potential targets with a *single* tag $\{f\}$. Upon executing an instruction that is the source of an indirect control flow, we transfer this tag to the PC:

$$\texttt{indir}: \quad (-,\{f\},-,-,-) \rightarrow (\{f\},-) \tag{16}$$

All other instructions are tagged $\varnothing$. Whenever the PC is tagged $\{f\}$, the current instruction must have the same tag:

$$\overline{\texttt{indir}}: \quad (pc,ci,-,-,-) \rightarrow (\varnothing,-) \text{ if } pc \subseteq ci \tag{17}$$

Policy Ⓚ uses more tags ($\varnothing$, $\{r\}$, $\{c\}$, and $\{r,c\}$) to separately track the control flows originating from returns (whose tag contains r) from the ones originating from indirect calls and jumps (whose tag contains c). Policy Ⓛ extends Ⓚ with two additional tags ($\{p\}$ and $\{p,c\}$) for returns into privileged code (whose tag contains p), allowing additional protection for critical code snippets [52].

As the attack of Göktaş et al. [23] shows, these loose CFI policies are not a sufficient protection against sophisticated code-reuse attacks. We also implemented a set of fine-grained CFI policies, which Göktaş et al. described as "ideal CFI." We first introduce two orthogonal policies: `PUMP JOP` (Ⓜ), which precisely tracks the association between indirect jumps and calls and their targets; and `PUMP ROP` (Ⓝ), which does the same for returns, as presented in §2 (Rules 1'–4'). We finally merge these two policies into `PUMP CFI` (Ⓞ)—a single policy precisely tracking and validating all indirect control flows. In all these policies, the compiler or linker is assumed to compute a sound overapproximation of indirect control flows and tag instructions accordingly.

Protection Demonstration. We tested these policies against a specially crafted program consisting of a single call to an "innocuous" function. The code also includes a "bad" function that is never called, mimicking dormant gadgets not

part of the execution path but that can be exploited to cause unintended behavior. To simulate a return-oriented attack, inlined assembly in the innocuous function overwrites the stack pointer with the address of the bad function, tricking the execution into returning into the bad function. Policy Ⓝ detects this simulated attack by noticing that the bad return is not in the set of valid control flows.

Characteristics. Each of the CFI policies above can be encoded very compactly with only 2–4 symbolic rules. The simpler policies (Ⓙ, Ⓚ and Ⓛ) also require a very small number of tags and concrete rules. As shown in Fig. 1, the largest of these (Ⓛ), uses 6 constant tags and requires no more than 21 concrete rules. With such small working set sizes these policies do not incur observable runtime overhead over the empty policy. Applying the stronger CFI policies (Ⓜ, Ⓝ, Ⓞ) to the SPEC benchmarks produces up to a few thousand concrete rules for these policies, which fit completely into the 4096 entry, pre-miss-handler PUMP cache. Consequently, we gain the added protection of ideal CFI with no additional runtime overhead. The complete CFI (Ⓞ) policy requires an average of 28K words to store the control-flow graph for the application (for this simulation, we extract it from the instruction trace generated by `gem5`; in practice, it will take more space than shown including allowed control flow paths that are never exercised in our simulations).

Related Work. CFI [3] offers an attractive defense against common code reuse attacks, but it has often been considered too expensive. Recent work [52] has demonstrated a low-overhead CFI scheme that uses "springboards" both to provide branch target checking and to randomize as a further defense against successfully constructing gadgets. However, this work only locks down allowed call and return targets, similar to the single-`target` example in Rules 1-4 from §2, not specific return points with specific targets as policies Ⓝ, Ⓜ, and Ⓞ do, leaving it vulnerable to attacks [23]; nor does it address intra-procedural CFI as our Ⓜ and Ⓞ do.

4.4 Taint Tracking

Threat Model. This policy addresses cases where an attacker inputs malformed data to a program that does no input sanitization, invoking unintended or malicious behavior (*e.g.*, SQL or OS command injection).

Policy and Rules. Taint tracking mitigates these threats by detecting when untrusted data may flow into sensitive operations. The PUMP facilitates fine-grained taint tracking with an unlimited number of sources, a separate taint per source, and multiple taints on each piece of data, allowing each tag to be a pointer to a set of source ids. The taint on a value is the union of taints on the values used to compute it. Typical taint propagation rules include:

$$\texttt{add}: \quad (-, ci, op_1, op_2, -) \rightarrow (-, ci \cup op_1 \cup op_2) \tag{18}$$

$$\texttt{load}: \quad (-, ci, op_1, -, mr) \rightarrow (-, ci \cup op_1 \cup mr) \tag{19}$$

$$\texttt{store}: \quad (-, ci, op_1, op_2, -) \rightarrow (-, ci \cup op_1 \cup op_2) \tag{20}$$

All the policies we study use the same set of symbolic rules, differing only in the number and sources of initial taints. We introduce taints in two different ways: by input sources (Ⓟ and Ⓠ) and by code regions (Ⓡ, Ⓢ and Ⓣ). In Ⓟ we use a single taint for all input sources (i.e., standard I/O streams and input files, for the SPEC programs). This is similar to most previous work [42], where a single-bit taint t simply indicates whether or not any data from an untrusted source has been used in computing a value tagged t. Policy Ⓠ extends Ⓟ by assigning each input stream a unique taint id; there is no limit on the number of streams.

Tainting by program code protects against untrusted libraries and buggy components. We vary the granularity by using an unique taint for (i) each library (Ⓡ), (ii) each included header file (Ⓢ), or (iii) each function in the code (Ⓣ). These policies require the compiler to tag the instructions with relevant taint identifiers. Finally, we combine Ⓠ and Ⓡ to form policy Ⓤ.

Protection Demonstration. We consider a case of CWE-78 (OS Command Injection) [30] where a user is only allowed to parametrize the arguments of an `ls` command passed to the `system` system call. The malicious user adds a parameter string that starts with the command terminator character, along with an arbitrary command. This translates into data that are post-sanitization tagged as "untrusted" to be passed as arguments to the `execve` system call. Using policy Ⓤ, the PUMP stops execution when it sees it is about to combine untrusted with system-call taints.

Characteristics. All these policies use the same set of 8 symbolic rules, defined in terms of 7 opgroups. The first two (Ⓟ and Ⓠ) use the input streams as taint sources. For Ⓠ, across all the SPEC programs, we only see 2 sources on average, and we need just 10 and 14 concrete rules. Consequently, these policies incur no noticeable runtime overhead. For policies Ⓡ–Ⓤ, we see larger working sets.

The taint by function experiment (Ⓣ) deliberately pushes the mechanism to an extreme, providing finer-grained tagging than is probably useful in practice. Its large number of taints result in an order of magnitude more rules than the PUMP cache can hold at once. Furthermore, the tag-handling overhead becomes large (4110 instructions). These factors result in an average runtime overhead of 314%. This shows the PUMP mechanism does strain under complex policies but can still support them. Taint per file (Ⓢ) is also finer-grained than is likely useful and it achieves low runtime overhead at 9% due to the smaller rule set and tighter miss handler resolution.

Policies Ⓡ and Ⓤ, where we assign taints to whole libraries, represent a more reasonable usage. Here, the average runtime overhead remains indistinguishable from the no-miss-handler case. This shows that the PUMP is able to represent and support much richer models (compared to prior work using 1b- or 4b-taints) with essentially no additional runtime overhead. Furthermore, across these various taint cases, the final tags are only 2–3× the initial and dynamically allocated tags; this shows that, while we do create non-singleton tag sets, we see nothing close to the theoretical worst-case power set effects.

Related Work. Vulnerabilities that have been addressed using taint tracking include format string attacks [49, 18, 42, 19, 13], cross-site scripting [49, 19, 13], memory exploits [49, 18, 42, 15, 19, 37, 13], code injection [49, 18, 19, 13] and others [50, 19]. Most existing work focuses on software techniques, where programs are instrumented. Typically, these introduce significant runtime overheads (often over 2×, some up to 20×), apart from other obstacles (*e.g.*, handling race conditions in dynamic binary translations [16]).

Hardware approaches like DIFT [42], Minos [18], and SIFT [36] use a single taint bit. Raksha—both on-core [19] and dedicated-coprocessor [27] variants—supports up to four concurrent policies using 4-bit tags. In contrast, we allow arbitrary sets of taints, corresponding to multiple untrusted sources, perhaps with different levels of trustworthiness. More flexible, tagging schemes are discussed in §5.

4.5 Composite Policies

Each of these policies is potentially useful, but it would be a shame if one had to pick only a single policy to enforce at a time—e.g., make a choice between protecting against buffer overflow or command injection vulnerabilities. Instead, one typically wants the protection that comes from composing multiple policies. In fact, some of our individual policies require mutual protection to guard against full threats (CFI depends on types protection for ensuring that data cannot be executed and code cannot be created or modified).

Composition can potentially increase the number of tags as well as the number of rules created, thereby considerably degrading performance. In order to characterize the combinatorial effect, we implement two composite policies. First we implement a fairly minimal one based on the simplest instances of each of the four protection classes: 3 primitive types (Ⓒ), a simple memory safety (Ⓕ), CCFIR (Ⓛ), and, single-bit input-taint (Ⓟ). Second, we implement a more complete and powerful protection that is the composition of 4 primitive types (Ⓓ), full spatial and temporal memory safety (Ⓘ), PUMP CFI (Ⓞ), and the composition of per stream input-taint and per library code-taint (Ⓤ).

Characteristics. The simple composite policy Ⓥ fits in the cache and has the same performance as the constituent policies. For the larger composite policy Ⓦ, the need to resolve all the policies increases the number of instructions required for rule resolution in the miss handler substantially, raising it from 38 to 710. The increase in final tags is only 2.5× suggesting there are some product set effects from composite tags, but it is nowhere near the worst-case scenarios. Furthermore, the concrete rules only grow about 3×, both due to the larger set of tags and the additional opgroups. The combination of the larger concrete rule set (now *much* larger than the PUMP cache capacity) and increased miss handler cost, results in an average overhead of 38% with the worst-case overhead going as high as 280% (`GemsFDTD`). This shows that the PUMP can handle the large set of rules resulting from the composite at the expense of an impact on performance. For many applications, the overhead remains modest, but for some it becomes unreasonably large. This, along with the taint by function experiment (Ⓣ), points to the need for additional software and microarchitectural optimizations to reduce miss handler service times in order to achieve reasonable performance on rich composites such as this, which is a focus of our ongoing work.

4.6 Discussion

The total number of rules doesn't completely capture the locality of rules and consequently the effective working set sizes. Fig. 7 shows a cumulative distribution function (CDF) of the number of unique rules used within each one million instruction sequence within the one billion instruction simulation for the `gcc` benchmark. This shows that full memory safety (Ⓘ) has a very tight working set (mostly less than 3000); this is significant since it has the largest number of concrete rules of any non-composite policy. This locality helps explain why the performance overhead remains low despite the much larger set of rules. Complete CFI (Ⓞ) needs more rules, but still fits well in the 4096 entry cache.

While previous work has used clever schemes to compactly represent or approximate safety and security policies (*e.g.* [43]), this is often a compromise on the intended policy, and it may trade complexity for compactness. We show that it is possible to include richer metadata that captures the needs of the security policies both more completely and more naturally with little or no additional runtime overhead. Rather than imposing a fixed bound on the metadata representation and policy complexity, the PUMP provides a graceful degradation in performance. This allows policies to use more data where needed without impacting the common case performance and size. It further allows the incremental refinement and performance tuning of policies, since even complex policies can easily be represented and executed.

4.7 Other Micro-Policies

We believe our programming model can encode a host of other policies. Information-flow control (e.g., [7, 38, 41, 25, 9]) is richer than the simple taint tracking models here, but tracking implicit flows can be supported either with RIFLE-style binary translation [45] or by using the *PC* tag with some support from the compiler. Micro-policies can support lightweight access control and compartmentalization [48]. Tags can be used to distinguish unforgeable resources [51]. Unique, generated tokens can act as keys for sealing and endorsing data, which in turn can be used for strong abstraction—guaranteeing that data is only created and destructured by authorized code components. Micro-policy rules can enforce data invariants such as immutability and linearity. Micro-policies can support parallelism as out-of-band metadata for synchronization primitives such as full/empty bits for data or futures (*e.g.* [6]) or as state to detect race conditions on locks (*e.g.*, [39, 53]). A system architect can apply specific micro-policies to existing code without auditing or rewriting every line.

5. RELATED WORK

Work related to our example policies has been covered in §4. Here, we discuss work related more generally 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 Fig. 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 (*e.g.*, [19]), multiple bit tags (*e.g.*, [46]), and more flexible policies (*e.g.*, [20]). The only design to support more than one policy at a time, Raksha, supported at most four taint tracking policies [19].

The prior systems closest to ours are Aries [12], FlexiTaint [46], Log-Based Architecture (LBA) [14], and Harmoni [21], 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: Fig. 1 summarizes

Ref.	HW/ SW	Tag Size	Tag IO	Security Policy
[15, 18, 36]	H	1b	2/1	hardwired taint
[42]	H+S	1b	2/1	hardwired taint
[13]	H+S	1b	2/1	limited prog.
[19, 27]	H+S	4b	2/1	four 1b policies; limited SW prog.
[46]	H+S	4b	2/1	limited prog.
[5]	H+S	1–8b	1/–	no propagation, mostly memory
[20]	H	var	2/1	FPGA reconfigurable
[21]	H+S	var	2/1	limited program. plus table
[14]	H+S	64b	5/-	SW on separate, augmented core
PUMP	H+S	64b	5/2	fine-grained, SW-defined policies

Figure 2: Hardware Tagging Approaches

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 LBA (e.g., the 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 overheads of 8% for most single policies. 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.

6. FUTURE WORK

The PUMP design offers an attractive combination of flexibility and performance, supporting a diverse collection of low-level, fine-grained security policies with single policy performance comparable to dedicated mechanisms in many cases while supporting richer and composite policies with mostly graceful performance degradation as rule complexity grows. To more thoroughly understand this design space, a number of issues will require further investigation. First, once we have a running hardware implementation, we will need to integrate the PUMP hardware and low-level software with a host operating system and software toolchain (e.g., compilers, linkers, and loaders). Second, we wonder whether the mechanisms provided by the PUMP can be used to protect its own software structures. We believe we can replace the special miss-handler operational mode by implementing a "compartmentalization" micro-policy using the PUMP and using this to protect the miss-handler code. Finally, we have seen here that it is easy to combine orthogonal sets of policies, where the protections provided by each one are completely independent of the others. But policies often interact: for example, an information-flow policy may need to place tags on fresh regions being allocated by a memory safety policy. Policy composition requires more study both in expression and in efficient hardware support.

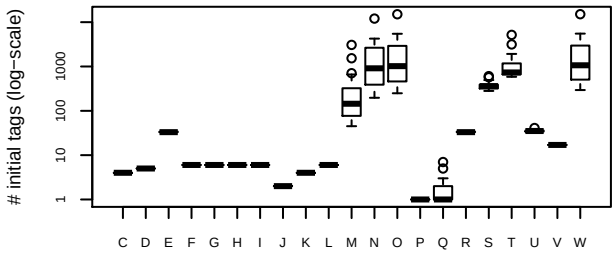

Figure 3: Distribution of Initial Tags

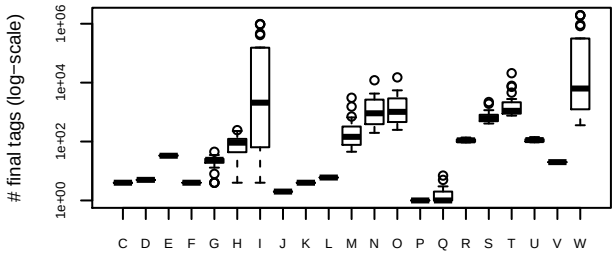

Figure 4: Distribution of Final Tags

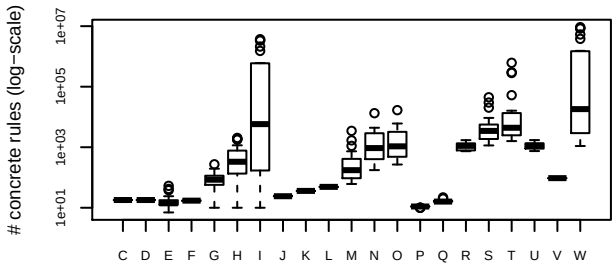

Figure 5: Distribution of Concrete Rules

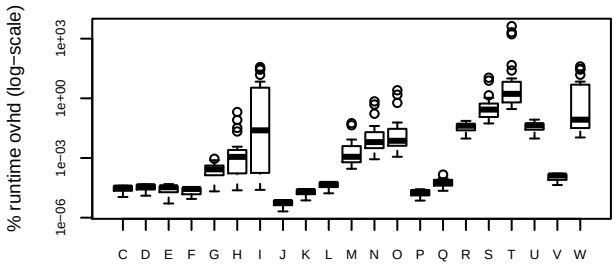

Figure 6: Distribution of Runtime Overhead (above policy A)

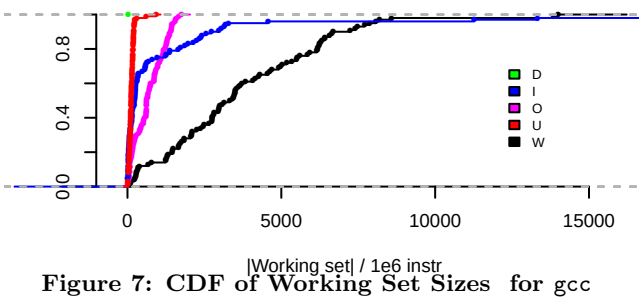

Figure 7: CDF of Working Set Sizes for gcc

7. REFERENCES

[1] Introduction to Intel Memory Protection extensions. `http://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions`. Accessed: 2013-08-01.

[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] P. Akritidis, M. Costa, M. Castro, and S. Hand. Baggy bounds checking: an efficient and backwards-compatible defense against out-of-bounds errors. In *Proc. USENIX Security*, pages 51–66, 2009.

[5] 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.

[6] 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.

[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] (authors removed for anonymity). PUMP – A Programmable Unit for Metadata Processing, 2014. To appear.

[9] A. Azevedo de Amorim, N. Collins, A. DeHon, D. Demange, C. Hriţcu, D. Pichardie, B. C. Pierce, R. Pollack, and A. Tolmach. A verified information-flow architecture. In *POPL*, pages 165–178. ACM, Jan. 2014.

[10] 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.

[11] T. Boland and P. E. Black. Juliet 1.1 C/C++ and Java test suite. *Computer*, pages 88–90, 2012.

[12] 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.

[13] 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.

[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 *Proc. ISCA*, pages 377–388, 2008.

[15] 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.

[16] J. Chung, M. Dalton, H. Kannan, and C. Kozyrakis. Thread-safe dynamic binary translation using transactional memory. In *HPCA*, pages 279–289. IEEE, 2008.

[17] J. A. Clause, I. Doudalis, A. Orso, and M. Prvulovic. Effective memory protection using dynamic tainting. In *Proc. ASE*, pages 284–292. ACM, 2007.

[18] J. R. Crandall and F. T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *Proc. IEEE MICRO*, pages 221–232, 2004.

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

[20] 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.

[21] 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.

[22] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic. HardBound: Architectural support for spatial safety of the C programming language. In S. J. Eggers and J. R. Larus, editors, *ASPLOS*, pages 103–114. ACM, 2008.

[23] E. Göktaş, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out Of Control: Overcoming Control-Flow Integrity. In *Proc. IEEE S&P*, 2014.

[24] C. J. Haley, S. M. Luera, M. D. Schanken, and W. B. Geer. Final evaluation report unisys a series mcp/as release 3.7. Technical Report CSC-EPL-871003, Library No. S-228,515, National Computer Security Center, Fort Meade, MD, August 5 1987.

[25] 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.

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

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

[28] MITRE Corp. CWE-122: Heap-based buffer overflow.

[29] MITRE Corp. CWE-416: Use after free.

[30] MITRE Corp. CWE-78: Improper neutralization of special elements used in an OS command (OS command injection).

[31] MITRE Corp. CWE-843: Access of resource using incompatible type (type confusion).

[32] D. A. Moon. Architecture of the Symbolics 3600. In *Proc. ISCA*, pages 76–83, Los Alamitos, CA, USA, 1985. IEEE Computer Society.

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

[34] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic. SoftBound: highly compatible and complete spatial memory safety for C. In *Proc. PLDI*, pages 245–258. ACM, 2009.

[35] E. I. Organick. *Computer System Organization: The*

B5700/B6700 Series. Academic Press, 1973.

[36] 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.

[37] 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 *Proc. IEEE MICRO*, pages 135–148, 2006.

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

[39] 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.

[40] 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.

[41] 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.

[42] 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.

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

[44] G. S. Taylor, P. N. Hilfinger, J. R. Larus, D. A. Patterson, and B. G. Zorn. Evaluation of the SPUR lisp architecture. In *Proc. ISCA*, pages 444–452, 1986.

[45] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. RIFLE: An architectural framework for user-centric information-flow security. In *Proc. IEEE MICRO*, 2004.

[46] 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.

[47] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *SOSP*, pages 203–216, 1993.

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

[49] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks. In *Proc. USENIX Security*, Berkeley, CA, USA, 2006.

[50] H. Yin, D. X. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proc. CCS*, pages 116–127. ACM, 2007.

[51] 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.

[52] 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.

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