POLICY IMPLEMENTATION AND ENGINEERING FOR TAGGED ARCHITECTURES

Nick Roessler

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2021

ACKNOWLEDGEMENTS

First, I'd like to thank my advisor André DeHon for his support and guidance on my journey; he demonstrated the courage to think big and challenge assumptions. I'm also grateful to the other members of my committee, Jonathan Smith, Benjamin Pierce, Greg Sullivan, and Joseph Devietti, for their valuable feedback on this work.

Luke Valenta has been a close friend and I'm thankful for the adventures and inspiring conversations we shared while going through this phase of our lives together. I can't thank Rafi Rubin enough for his wise words, calming perspective, and teaching me sysadmining skills; he always made lab a fun place to be. Marcella Hastings, Stephen Phillips and Nikos Vasilakis have been wonderful friends that made my grad school experience such a pleasure. Alex Pezzati, Justin Goodrich, William Marshall and Jinesh Desai have enriched my life and I feel incredibly lucky to know them.

Lastly, I'd like to thank Ariella Mansfield for her love and support. Unexpectedly, we survived a pandemic quarantined together and I cherish the time we had.

This work was supported by National Science Foundation grant TWC-1513854 and DARPA contract HR0011-18-C-0011 under the System Security Integrated Through Hardware And Firmware (SSITH) program.

ABSTRACT

POLICY IMPLEMENTATION AND ENGINEERING FOR TAGGED ARCHITECTURES

Nick Roessler

André DeHon

Tagged architectures have seen renewed interest as a means to improve the security and reliability of computing systems. Rich, programmable tag-based hardware security monitors like the PUMP [43] allow software-defined security policies to benefit from hardware acceleration. The thesis of this work is that policies for programmable tagged architectures (1) can be engineered to enforce critical security properties at low cost, (2) can protect real programs running on real ISAs, and (3) can be applied automatically to programs—that is with compilation passes or automatic analysis—so that the benefits of such an architecture can be brought to existing and new software with minimal human intervention.

To support this claim, I have constructed a range of security policies that run on real workloads automatically, modeled their overheads using architectural simulations, explored tradeoffs in policy design and engineering to reduce their costs, and finally characterized them by their security properties. As examplar policies, I have created stack and heap memory protection policies that can thwart traditional memory corruption vulnerabilities. Additionally, I have built a compartmentalization framework that allows a security engineer to automatically generate and evaluate a wide range of tag-based compartmentalization strategies. To generate compartments automatically, the framework includes algorithms for quantitatively minimizing overprivilege and packing the rules required for those policies into manageable sets that can be cached favorably for high performance. Across these three categories of policies, I present the following policy engineering contributions: (1) lazy tagging, an optimization that reduces the cost of tagging memory objects, (2) rule packing, a technique for relaxing policies in key ways to improve their performance, and (3) rule prefetching, a technique that can exploit predictable rule sequences by preemptively fetching and installing rules before they are needed.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iii

ABSTRACT iv

LIST OF ILLUSTRATIONS viii

CHAPTER 1 : Introduction 1

1.1 Motivation 1

1.2 Outline 3

1.3 Contributions From Others 6

CHAPTER 2 : Background 7

2.1 The PUMP Architecture 7

2.2 Security and Threat Model 9

CHAPTER 3 : Stack Protection Policies 11

3.1 Introduction 11

3.2 Threat Model and Assumptions 14

3.3 Stack Protection Policies 15

3.4 Evaluation 29

3.5 Optimizations: Lazy Tagging 33

3.6 Security Characterization 38

3.7 Policy Compatibility 40

3.8 Discussion and Conclusion 43

CHAPTER 4 : Heap Protection Policies 48

4.1 Introduction 48

4.2 Background 49

4.3 Threat Model and Assumptions 54
4.4 Policy Formulation and Implementation 54
4.5 Evaluation 60
4.6 Security Characterization 66
4.7 Policy Compatibility 71
4.8 Limitations and Future Work 74
4.9 Conclusion 75

CHAPTER 5 : Compartmentalization Policies 77
5.1 Introduction 77
5.2 Background and Related Work 81
5.3 Threat Model 84
5.4 Privilege Reduction and Compartmentalization 85
5.5 Compartmentalization Tag Policy Formulation 86
5.6 The Tracing Policy 89
5.7 Privilege Quantification Model 91
5.8 Policy Exploration 92
5.9 Performance Model 100
5.10 Evaluation 101
5.11 Prefetching 108
5.12 Security, Overprivilege and Work-factor 114
5.13 Future Work 117
5.14 Comparisons with Related Embedded System Security Work 119
5.15 Runtime Modes and Dynamic Analysis 120
5.16 Conclusion 121

CHAPTER 6 : Conclusion 122

BIBLIOGRAPHY 123

LIST OF ILLUSTRATIONS

FIGURE 1 : Typical Alpha stack maintenance code 15

FIGURE 2 : Return Address Protection rules 19

FIGURE 3 : Static Authorities example 22

FIGURE 4 : Static Authorities rules 26

FIGURE 5 : Depth Isolation rules 28

FIGURE 6 : Evaluation Framework 28

FIGURE 7 : Return Address Protection overhead 32

FIGURE 8 : Static Authorities overhead 32

FIGURE 9 : Depth Isolation overhead 32

FIGURE 10 : Optimizations applied to Static Authorities 37

FIGURE 11 : Optimizations applied to Depth Isolation 37

FIGURE 12 : Stack threat taxonomy 39

FIGURE 13 : The layout of the `malloc_chunk` structure 50

FIGURE 14 : An example heap overflow 54

FIGURE 15 : Rules vs number of heap colors 61

FIGURE 16 : Overhead of N-Color heap policy 63

FIGURE 17 : Overhead of the Allocation-Site heap policy 64

FIGURE 18 : Relaxing the heap policy on `omnetpp`. 65

FIGURE 19 : Security characterization of heap policy variations 71

FIGURE 20 : Pointer tag propagation ambiguities for `and` opcode 72

FIGURE 21 : Sub-word data manipulation in a Quicksort implementation. . . . 74

FIGURE 22 : The PIPE Architecture 81

FIGURE 23 : The output of the tracing policy 90

FIGURE 24 : Automated compartmentalization algorithms and the resulting number of compartments, rules, overprivilege and miss rates. 96
FIGURE 25 : Rule savings calculation example for the Working-Set algorithm . 98
FIGURE 26 : The impact of the WS_{max} parameter on the rule cache miss rate . 103
FIGURE 27 : Overhead vs Overprivilege Ratio from SCALPEL's algorithms . . 106
FIGURE 28 : Syntactic Domains and Syntactic Constraints 107
FIGURE 29 : The Rule-Successor Graph 110
FIGURE 30 : The results of the SCALPEL's rule prefetching system. 113
FIGURE 31 : A buffer overflow example 115

CHAPTER 1 : Introduction

1.1. Motivation

Conventional computing systems are highly insecure. Substantial portions of their software stacks, including operating systems, device drivers, runtime environments, and web browsers, are written in languages such as C, C++, and assembly. These languages have seen widespread adoption in the systems development world due to the precise low-level control that they expose to programmers and their fast execution speeds, but they are also notoriously insecure. Notably, they lack array bounds checking, they rely on manual memory management which can lead to use-after-free and double-free errors, they permit undefined behavior which compilers may exploit to unexpectedly remove security checks or do other surprising code transformations [130], they employ complicated implicit type casting rules and expose low-level architectural behavior (*e.g.,* integer overflows) that may lead to hard-to-detect bugs in application logic—and the list goes on. Repeated studies have shown that essentially all software contains bugs [15, 60, 88], and bugs in programs written in unsafe languages produce executable artifacts that may violate language-level abstractions (such as isolating memory objects from each other), thus paving the path for attackers to manipulate computing systems to malicious and devastating ends.

Many security properties of programs that could be enforced to prevent these kinds of errors (or at least mitigate their malicious effects) have been identified in the literature. For example, memory pointers created to reference objects should not be permitted to access memory outside of the object's bounds; when a machine transfers control-flow from one instruction to another, the edge taken should be in the expected control-flow graph of the program; when an operation takes place on or between objects, the operation should be valid in terms of the types of the objects; errors in one component of a system should not be able to affect other unrelated components—many other such properties could be described. Despite the security community both knowing these properties and how to enforce them—or at least relaxed versions of them—on critical software, they have not been widely deployed to protect computing systems; one of the key reasons is that the overheads imposed by enforcing them are deemed to be too high to justify their benefits. For a defense to be used in practice, it must sit favorably in the performance cost versus protection tradeoff space. The few mitigations that have seen widespread adoption, such as stack canaries [33], address space layout randomization (ASLR), and Write-XOR-Execute memory permissions (W⊕X), not surprisingly, are all mitigations that do not impose substantial overheads.

As the cost of transistors goes down, the opportunity to invest hardware resources to accelerate security policies becomes increasingly viable. With hardware support, the acceptable points in the protection-versus-overhead tradeoff space can shift favorably towards stronger mitigations. In this dissertation I focus specifically on a software/hardware co-designed programmable tag-based hardware security monitor, the Programmable Unit for Metadata Processing (PUMP) [43]. In this architecture, the CPU maintains a metadata tag on each word of memory in the system, on each register in the register file, and on the program counter. As each instruction executes, a software security monitor is consulted. It inspects the tags relevant to the instruction, determines if the operation should be permitted according to a specified policy (or policies), and if so, supplies tags for the results of the operation. To accelerate the behavior of the security monitor, the CPU maintains a cache of the recently encountered rules—properly designed [42], such a cache can be consulted in a single cycle without stalling the CPU. This means that policies can be enforced at low costs as long the rule locality is high *i.e.,* misses in the rule cache are rare.

To be useful, that is to identify and halt invalid or malicious executions, the PUMP must be provided with a software-defined policy (or set of policies) expressed in a fine-grained and low-level fashion suitable for the PUMP: we call such policies *micropolicies. This dissertation is about the construction and engineering of useful micropolicies.*

While previous research has shown that micropolicies can be formulated and proven to be correct for abstract symbolic machines [11], this dissertation takes a pragmatic approach.

Concretely, (1) it models policies on a full and real RISC ISA and deals with the nuances of real architectures and compilers, (2) it pays heed to the overhead imposed by the enforcement of policies by modeling a concrete PUMP with a finite rule cache, a limitation that influences policy design, and (3) it is concerned with policies that can be applied *automatically*—that is, with compilation passes or automatic analysis, so that minimal human intervention is required to bring the benefits of the PUMP to existing and new software.

1.2. Outline

Chapter 2 covers the background material for this dissertation. It introduces the PUMP architecture and the C-based threat model that our policies are built to protect against.

Chapter 3 is about policies for protecting the program call stack; its contents are drawn from a published paper on the topic [106]. The runtime stack is a critical system component with a long history of being exploited by attackers [5]. The call stack serves as a storage repository for a range of uses to support the abstraction of a function call. In memory unsafe languages, attackers can tamper with data items stored in the stack to hijack the control-flow of the machine or otherwise maliciously manipulate or read program data. The chapter presents three stack protection policies that secure the stack abstraction at various levels of protection and costs. The goal of the policies is to carry forward information available to the compiler about correct program behavior (such as which instruction should access which fields of a frame) and to enforce them at runtime with tags and rules. The first policy, Return Address Protection, is designed as a lightweight and simple policy; it uses tags to protect return addresses stored in stack memory by limiting access to them to just the compiler-generated instructions explicitly emitted for stack management. The policy has an overhead of only 1.2% and provides protection comparable to stack canaries. The chapter then presents two richer policies that protect all stack objects—to do so, these policies (1) insert instrumentation to tag all stack elements as stack frames are pushed and popped from the stack, and (2) tag program code to communicate the expected behavior of those instructions such that they can be validated at runtime. Most of the overhead of these policies comes from the cost of tagging and then clearing stack memory—consequently, we investigate optimizations for lazily tagging and clearing stack memory to reduce these costs while still providing object-level protection of all stack elements. With these optimizations, the stack protection policies impose an overhead of only 3-4%. The chapter concludes with an attack taxonomy to characterize the security properties of the policies and their optimizations.

Chapter 4 presents policies for protecting the heap, a source of dynamic memory. In manually managed languages such as C/C++, programmers must explicitly allocate and deallocate memory objects by invoking a software component called an allocator. Manual memory management of this form is notoriously error prone, introducing new error classes such as use-after-frees and double-frees that can also lead to memory corruption. Heap-based exploits have become very popular among attacks against real systems [17], even overtaking other kinds of memory errors in recent years [127]. To protect the heap, we introduce policies based upon the dynamic tainting and checking technique introduced in [30]. In these policies, the allocator is modified to tag the memory chunks that it allocates with a *color* identifier, and also to tag the pointer it returns to the program with the same color. On each memory access performed by the program, the color of the pointer is compared against the color of the memory word, and if they do not match then the access can be determined to be illegal and a violation is raised. Unlike the stack policies, programs do not require instrumentation to benefit from the protection of the heap policies. The range of heap policies we present vary only in their coloring schemes *i.e.,* how many identifiers are used to differentiate allocations and how they are assigned to those allocations. We introduce a simple One-Color policy that can protect against a range of common vulnerability types with an overhead of only around 1%, as well as an Infinite-Color policy that provides complete spatial and temporal memory safety for heap memory at a higher cost 37% (although for many workloads the overhead is less than 10%). Between these two extremes, we explore several other variations and additionally how the Infinite-Color policy can be relaxed in key places to reduce its costs while maintaining as much protection as possible.

Chapter 5 is about compartmentalization policies. Compartmentalized systems are more robust to attacks than monolithic systems, because when a breach occurs the attacker is constrained to just the data and privileges available in the compromised compartment. In this chapter we present SCALPEL, a tool for automatically compartmentalizing systems and enforcing those compartments with tags and rules. In the first phase, SCALPEL uses a tracing policy to record the set of fine-grained privileges required by a program to perform its tasks, including both control-flow transitions and memory access patterns. The tracing policy is implemented as a drop-in policy replacement, allowing to run on the same software and hardware without any other system changes. After running the tracing policy, SCALPEL then uses the tracing data to systematically generate a wide range of possible system decompositions. To generate its decompositions, SCALPEL treats compartment generation as an optimization problem over the privilege-performance space. To produce high-performance compartmentalizations, SCALPEL targets the number of rules that are needed by a program during any of its phases and packs those rules down into manageable sets that can be favorably cached. To decide how to relax the decompositions to achieve its rule targets, it introduces a quantitative privilege representation and uses overprivilege as an objective function to minimize. At the extreme, SCALPEL can target packing an entire compartmentalization policy into a set of rules small enough to fit into the rule cache, thus achieving no runtime rule misses and the predictable performance profile required for real-time systems. Lastly, a final optimization used by SCALPEL is a rule prefetching system. When a program enters a compartment, many of the rules that will be required by that program can be predicted in advance, which means they can be installed preemptively to avert future misses. We show that prefetching can reduce the overhead costs of fine-grained separations by almost 4X, allowing tighter separations to run at lower overheads.

The common thread throughout the chapters is *policy engineering*, *i.e.,* the design of security policies that achieve useful security properties while managing the number of tags, rules, and other runtime costs for enforcement. In the stack policies, the dominant source of overhead arose from tagging and clearing stack memory, which we solved with the lazy tagging and lazy clearing clearing policy designs. In both the stack and the heap policies, we found that generating unique identifiers for each dynamic stack frame or each heap object could challenge the rule cache, and a possible policy design strategy to reduce costs is to map identifiers to static entities, *i.e.,* Static Authorities for the stack and the Allocation-Site policy for the heap policies. For the compartmentalization policies, we introduce *rule packing*, a technique in which the set of rules required for each program phase is packed into a set that can be cached favorably. Lastly, we design and evaluate a rule prefetching system that exploits predictable rule sequences to reduce the number of runtime rule resolutions.

1.3. Contributions From Others

The stack work in Chapter 3, including both the stack policies and the lazy tagging optimization, was done by me under the guidance of André DeHon.

The initial N-Color heap policy implementation in Chapter 4 was developed by Udit Dhawan, which I then refined substantially. The Allocation-Site variations and the security characterization were done by me.

The compartmentalization work in Chapter 5 began as a joint effort between myself, André DeHon, and Nathan Dautenhahn, in which the privilege representation model, the privilege quantification model, and compartmentalization algorithms were developed. The translation of the compartmentalization framework from Linux to FreeRTOS and the development of a tag-based back-end, as well as tracing policy, rule packing, syntactic constraints and prefetching were done by me and advised by André. The tag-based framework was supported by Arun Thomas and Chris Casinghino at Draper Laboratory and was built on top of the PIPE framework that was developed there.

CHAPTER 2 : Background

2.1. The PUMP Architecture

Classic Von Neumann computing architectures do not differentiate between code and data, nor do they track any other kind of metadata or typing information about internal machine state. Tagged architectures, broadly speaking, aim to increase the robustness of the machine by binding tags to pieces of internal state such that those tags can be taken into consideration to validate the machine's operations, *e.g.,* to assure that data is not interpreted as code and executed. Early uses of tags trace their histories back to the 1960s; in 1973 Feustel [50] proposed using tags for typing data elements stored by a machine, with types such as *int*, *bool*, *vector* or *linked list*. With increasing hardware resources as a result of Moore's Law, the number of bits that can be allocated for tags as well as the complexity of the tagging hardware has increased over time [38, 128, 55].

The PUMP [43] architecture, the focus of this dissertation, generalizes prior tagged architectures by providing *software-programmable*, but *hardware-accelerated* metadata processing over tags of unbounded size and complexity. The programmable nature of the PUMP is important for several reasons: (1) the complete set of security policies that one might want to enforce is both unknown and evolving in response to new threats, and (2) it permits deployments to configure tradeoffs by updating the software—this allows a single hardware mechanism to accommodate diverse end-user requirements, which may change depending on the security requirements, overhead tolerance, attack-surface exposure, policy compatibility, and even the individual application.

The core idea is that a full word-sized tag is indivisibly associated with each word of data in the system, including on each word of data in memory, on each register, and also on the program counter. As each instruction executes, the tags on the inputs to that instruction are used to validate that instruction against a software *policy.* The policy interprets the tags, decides if the operation is legal, and if so, it provides new tags for the output of that instruction. The amount of metadata held in a tag is unbounded: the tag can be treated as a pointer to an arbitrary data structure, which can compose data from multiple different policies to permit the enforcement of an arbitrary set of security policies (*e.g.,* CFI, memory safety).

There are up to six inputs to each instruction: (1) the instruction's opcode (*e.g.,* add, load, jump), (2) the tag associated with the program counter (PC), (3) the tag on the instruction itself (CI), (4-5) the tags on the register inputs ($R1$, $R2$), and (6) the tag on the word of memory (M), if the instruction is a load or store. Each instruction may produce up to two outputs: (1) an new tag on the program counter (PC'), and (2) a tag for the result of the output of the instruction (MR), whether it is a memory word or register. For compact short hand, a policy may be written as a collection of rules of the form:

$$Op : (PC, CI, R1, R2, M) \rightarrow (PC', MR)$$

While this function of six inputs to two outputs is restrictive compared to the entire set of tags in the full machine state that a security monitor might want to consider and modify in the general case, it is still highly expressive and importantly lends itself well to high-performance implementations: it (1) restricts where input tags can come from to simplify the hardware and (2) means that *rules*, that is, mappings of inputs to outputs, are manageable in size. This is important, as the core accelerator of policies is the *PUMP rule cache*, an additional, on-chip hardware store of the recently encountered rules. Properly designed [42], such a cache can perform a rule match within a single cycle so that the CPU does not stall if there is a hit in the cache. This means that the policy software runs only as a *miss handler* when a rule does not match a rule that is in the cache. When such a miss occurs, control flow is transferred to the miss handler software, which either computes a new rule or raises a security violation. If a violation is raised, the OS handles the violation, which will typically include terminating the offending program. If no violation is raised, then the miss handler installs a new rule into the rule cache and returns control back to the program. A consequence of the rule cache design is that tags are *immutable*, *i.e.,* tags cannot change their meaning over time which could invalidate the semantics of a cached rule.

For good performance, this means policies should keep their working sets of rules manageable to avoid frequent cache misses. A key thread throughout this dissertation is *policy engineering*, which deals with the interaction between policy design and the resulting number of tags, rules, and thus enforcement costs that arise as a result.

While a naive implementation of the PUMP architecture would double the size of all the registers and the on-chip caches, architectural optimizations can reduce much of this overhead [43]. For example, one key optimization is that tags in the L1 subsystem can be represented with a smaller number of bits, say 10, and then translated automatically back to longer tags as needed. This allows the L1 data and rule caches to remain much smaller, reducing the area and energy overheads associated with a PUMP implementation. This optimization allows the architecture to be closer in costs to shorter tag designs while still maintaining the full expressive power of word-sized tags.

The PUMP architecture has a close relative, the PIPE architecture, which is used in Chapter 5. The PIPE differs from the PUMP in that it uses a dedicated coprocessor for policy execution (the PEX core) in addition to the primary application core (the AP core). Furthermore, the PIPE architecture does not change the number of bits used by the host architecture in order to accomodate tag bits, but instead maintains a "tag map table" that maps host memory addresses to tag addresses (and maintains a shadow register file).

2.2. Security and Threat Model

While the PUMP is capable of expressing a wide range of security policies, in this dissertation we narrow our focus down to a standard C-based, system-security threat model for protection evaluation. Software written in languages such as C and C++ comprises a substantial portion of the trusted code underlying modern computing systems (operating systems, device drivers, runtime environments), as well as many widely-used applications (e.g., web browsers, document viewers, etc). These languages produce executable artifacts that do not enforce the abstractions present at the language level, such as isolating memory objects from one another. As a result, programming errors allow clever attackers to corrupt or manipulate the state of a machine to take control of its operations or steal data from it. These kinds of errors have been responsible for a large fraction of attacks against computing systems over the last several decades [1] and continue to be extremely problematic to this day [47].

Consequently, the threat model for this work is that untrusted input, such as from files or over a network, may be processed by programs written in languages such as C and C++. Bugs in these programs may allow an attacker to violate computing abstractions, such as through buffer overflows, use-after-free errors or double-free errors. The policies presented in this work, given the programmable nature of the hardware and the rich range of policy designs, aim to either (1) prevent these kinds of violations entirely, (2) turn a fraction of bugs from exploitable to unexploitable as other successful mitigations have done [33], or (3) contain the effects of such bugs to reduce the harm that a bug may cause or increase the attacker effort/cost required to weaponize such a bug.

These kinds of vulnerabilities are not the only ways in which computing systems can be compromised. For example, side channel attacks [71] or hardware attacks such as Rowhammer [69] can also be devastating; however, these kinds of attack vectors are outside of the threat model under consideration in this work.

CHAPTER 3 : Stack Protection Policies

3.1. Introduction

Low-level, memory-unsafe languages such as C/C++ are widely used in systems code and high-performance applications. Unfortunately, they are also responsible for many of the classes of problems that expose applications to attacks. Even today, C/C++ remain among the most popular programming languages [125], and code written in these languages exists within the Trusted Computing Base (TCB) of essentially all modern software stacks. In memory-unsafe languages the burden of security assurance is left to the application developer, inevitably leading to human error and a long history of bugs in critical software.

The program call stack is a common target for attacks that exploit memory safety vulnerabilities. Stack memory exhibits high spatial and temporal predictability, it is readable and writeable by an executing program, and it serves as a storage mechanism for a diverse set of uses related to the function call abstraction: the stack holds, in contiguous memory, local function variables, return addresses, passed arguments, and spilled registers, among other data. The particular concrete layout of stack memory, chosen by the compiler and calling convention, is exposed. An attacker can wield a simple memory safety vulnerability to overwrite a return address, corrupt stack data, or hijack the exposed function call mechanism in a host of other malicious ways.

Consequently, protecting the stack abstraction is critical for application security. Currently deployed defenses such as W$\oplus$X and stack canaries [56] make attacks more difficult to conduct, but do not protect against more sophisticated attack techniques. Full memory safety can be retrofitted onto existing C/C++ code through added software checks, but at a high cost of 100% or more in runtime overhead [90]. These expensive solutions are unused in practice due to their unacceptably high overheads [123].

Recent work has shown that programmable, hardware-accelerated rich metadata tag-based security monitors are capable of expressing and enforcing a large range of low-level security policies [43]. In this model, the processor core is enriched with expressive metadata tags attached to every word of data in the system, including on registers and on memory. The hardware propagates metadata tags and checks each instruction against a software-defined security policy. The same hardware mechanism accelerates any policy (or composition of policies) expressed in a unified programming model by caching a subset of the security monitor's behavior in hardware. Policies can be updated in-field or configured on a per-application basis.

In this work we develop tag-based stack protection policies for the Software-Defined Metadata Processing model (SDMP) that are efficiently accelerated by an architecture that caches metadata tag rules [43]. We propose a simple policy that utilizes only a few tags, as well as richer policies that generate thousands of tags for fine-grained, object-level stack protection. Our policies leverage the compiler as a rich source of information for protecting the stack abstraction. The compiler is responsible for the low-level arrangement of the stack, including how arguments are passed, how registers are spilled, and where program variables are stored; consequently, the compiler is aware of which parts of a program should be reading and writing each item on the stack. In conventional runtime implementations this information is simply discarded after compilation—by instead carrying it alongside the data and instruction words in a computation with metadata tags, we can enforce the compiler's intent and prevent the machine from violating the stack abstraction at runtime.

Stack protection SDMP policies face two major sources of overhead. The first is the slowdown incurred by software policy evaluation that must run to resolve security monitor requests when they miss in the hardware security monitor cache. The rate at which these misses occur is driven by the locality of metadata security rules, which in turn is driven by the diversity and use of metadata tags by the policy being enforced. We design our policies specifically to exploit the regular call structure found in typical programs by reusing identifiers for the same static function (Sec. 3.3.4) or by the stack depth (Sec. 3.3.4) to achieve cacheability of the required metadata rules.

The second significant source of overhead for stack protection policies is the cost of keeping stack memory tagged, which is a requirement faced by our richer policies. In conventional runtime implementations on standard architectures, stack memory is allocated and reclaimed with fast single instruction updates to the stack pointer. To tag this memory naively, we would need to insert code into the prologue and epilogue of every function to tag and then clear the allocated stack memory, effectively replacing an $\mathcal{O}(1)$ allocation operation with an $\mathcal{O}(N)$ one. This change is particularly costly for stack memory; heap allocations, in contrast, spend hundreds to thousands of cycles in allocator routines, which makes the relative overhead of tagging the allocated memory less severe.

To alleviate the cost of tagging stack memory, we consider several optimizations. One is an architectural change, Cache Line Tagging (Sec. 3.5.2), that gives the machine the capability of tagging an entire cache line at a time. Alternatively, we propose two variations to our policies that avoid adding additional instructions to tag memory, Lazy Tagging (Sec. 3.5.1) and Lazy Clearing (Sec. 3.5.3).

Lastly, to characterize our policies, we provide a taxonomy of stack threats (Sec. 3.6.1) and show how our policies as well as protection mechanisms from previous work protect against those threats.

The policies we derive in this work provide word-level memory protection of the stack abstraction, have low overhead (<6%), can compose with other SDMP policies to be accelerated with the same hardware (Sec. 3.8.1), interoperate with unmodified library code, do not require source code changes, and are compatible with existing code and idioms (run on the SPEC benchmarks).

Our contributions in this work are:

- The formulation of a range of stack protection policies within the SDMP model

- Three optimizations for our stack policies: Lazy Tagging, Lazy Clearing and Cache Line Tagging

- The performance modeling results of our policies on a standard benchmark set, including the impact of our proposed optimizations

- The protection characterization of our policies and comparison to prior work with a stack threat taxonomy

3.2. Threat Model and Assumptions

In developing our stack protection policies we assume the same powerful but realistic attacker capabilities of most related work, e.g., [72][35]. In this threat model an attacker provides arbitrary input to a program that contains a memory safety vulnerability, leading to adversarial reads or writes into the program address space. As a consequence, any attacks against stack data are in scope, including control flow hijacking and data corruption or data leaking attacks. We consider side channels and hardware attacks such as Rowhammer [69] to be out of scope. In Sec. 3.6.1 we provide a set of specific threats to demonstrate an attacker's capabilities within our threat model.

Our policies leverage compiler-level information such as the locations of objects on the stack and occasionally require adding instructions into programs. We thus consider the toolchain (the compiler, linker, and loader) to be in our TCB and assume we can recompile programs. Our policies do not, however, require code changes or programmer annotations.

We develop our policies specifically for the Alpha architecture, a RISC ISA, and use the *gcc* toolchain. These choices do impact the low-level stack details used in our policy descriptions and experiments. However, our policies should be easy to port to any RISC ISA; CISC ISAs would require some more care to handle the more complex memory operations such as *CALL*s that side effect both memory and register state. To illustrate typical stack maintenance operations under our ABI and architecture, we show a simple annotated function

```
main:
  lda    sp,-32(sp)     ; allocate frame
  stq    ra,8(sp)       ; store return address
  stq    fp,16(sp)      ; store old frame pointer
  mov    sp,fp          ; set new frame pointer
  stq    a0,0(fp)       ; write arg for foo()
  bsr    ra,<foo>       ; call foo()
  mov    fp,sp          ; reset sp before epilogue
  ldq    ra,8(sp)       ; restore return address
  ldq    fp,16(sp)      ; restore frame pointer
  lda    sp,32(sp)      ; release frame
  ret                   ; jump to return address
```

Figure 1: Typical Alpha stack maintenance code

disassembly in Fig. 1.

3.3. Stack Protection Policies

In this section we describe our stack protection policies. We begin with the motivation for our policy designs (3.3.1), proceed to connect our mechanism of tags and rules to the stack abstraction (3.3.2), enumerate the stack invariants that we would like to maintain (3.3.3), and finally give three concrete policies (3.3.4).

3.3.1. Motivation

Memory errors on stack-allocated objects can allow a program to perform invalid stack accesses, which attackers exploit to compromise the stack abstraction. To prevent these violations, our policies tag stack objects with both a *frame-id* (an identifier for a stack frame) and an *object-id* (an identifier for an object within a frame), and tag program code to allow the machine to validate accesses to these words using appropriate metadata rules. Formulating identifiers in this way allows us to express a range of policies; we are driven both by a desire for strong protection (precise notions of *object-id* and *frame-id*) and the performance of our policies (the cacheability of our metadata rules), making the choice of how we identify frames and differentiable objects inside them core to our designs. In general, cacheability concerns drive us to avoid creating a unique identifier for each dynamic procedure call to avoid the compulsory misses that would be required.

3.3.2. Tags and Rules

The building blocks of SDMP policies are tags and rules. Our policies use tags on (1) memory words, (2) registers, and (3) instructions. Tags on stack memory words encode a *frame-id* and an *object-id*, which together identify the frame that owns a word and which of the differentiable objects held by that frame is stored there. The tag on a register may be either $\perp$ (in the case that the register holds a value unrelated to the stack), or it may encode an allowed *frame-id* and *object-id* if it holds a pointer to a stack word. Lastly, instruction tags are used by the compiler to grant instructions specific capabilities, such as the right to set the tags on memory words, to set the tags on registers as pointers are crafted, to clear memory tags, or to perform other policy-specific functionality.

Rules allow us to define the set of permitted operations and describe how result tags are computed from input tags. For example, to validate a memory access, we can check that the *object-id* and *frame-id* fields on a pointer tag match those of the tag on the accessed memory word. Furthermore, during such a load, we could use additional fields on the memory word tag to describe how to tag the resulting value produced by the load. As another example, we can propagate a pointer tag along with a pointer value as the pointer is moved around the system (including between registers, to and from memory, and through operations such as pointer arithmetic) with appropriate rules, allowing us to use the dynamic tainting rules as in [30] to maintain pointer tags.

3.3.3. Stack Invariants

As a program executes, we would like to verify that objects on the stack are accessed in ways that the compiler expects with respect to our identifiers; i.e., the *object-id* and *frame-id* accessed by memory instructions match the compiler's intentions. Several kinds of accesses capture typical stack behavior (under our ABI and Alpha stack discipline), which we describe below.

Some stack objects, like return addresses, stored frame pointers and callee-saved values, are accessed strictly by code produced by the compiler specifically to maintain the stack abstraction. These objects are accessed in a highly restricted way: they are written to the stack once in the function prologue and are read only in the return sequence before returning control to the caller. Statically the compiler has emitted specific instructions for these purposes, and so, by the principle of least privilege, we would like to restrict access to these objects to just those predetermined instructions. For accesses of this variety, we place the *object-id* intention directly on the instruction performing the access.

Local stack variables are accessed in two ways. One way is through a fixed offset access from the frame pointer register. Accesses of this type, like above, allow us to encode the *object-id* intentions directly on the instructions that perform the accesses. In this case the *object-id* might be V_i, where V_i is an identifier for ith variable belonging to a particular frame. The second way that local stack variables can be accessed is through pointers held in general-purpose registers that are crafted by the program. This type of access occurs when accessing non-scalar types such as arrays, when the address of a local variable is taken and dereferenced, or when a piece of code obtains a pointer to stack data (e.g., was passed a pointer to stack local data as an argument). To validate this kind of access, we require that the accessing pointer was crafted specifically to access the object it is used to read or write; i.e., it was intentionally provided the capability to access a particular *object-id* inside a *frame-id*. This definition allows a pointer to a specific stack object to be passed as an argument to another function, but restricts the use of that pointer by the callee to just the intended *object-id* and *frame-id*.

A final class of memory operations used in the stack abstraction is the case of accessing function arguments themselves. This is a special case—function arguments are held in the caller's frame, but no pointer is passed to the callee to be treated as a capability for accessing them. Instead, the locations of arguments are implicitly dictated by the calling convention, and the callee will compute an offset beyond its own frame to access the arguments it has been passed. While we will still use compiler-level information to validate these accesses, we leave our discussion of how this is done to each of our concrete policies.

3.3.4. Policies

In this subsection we describe three concrete policies. In each case, we (1) give a high level description of the policy, (2) describe the implementation, and (3) detail the security properties of the policy. The rules for each policy written in SDMP notation are available in the appendix.

We focus on the the core policy behavior in this section—additional details pertaining to how our policies handle common low-level features and optimizations including *setjmp*, *longjmp* and Exceptions, tail calls, and dynamic stack memory allocations are discussed in Sec. 3.7.

Return Address Protection

Policy Description: The first stack protection policy we present, Return Address Protection, is a lightweight policy that is concerned only with control flow hijacking attacks that overwrite return addresses. It treats return addresses as special objects and restricts access to words containing return addresses to the specific instructions generated by the compiler for this purpose (i.e., Sec. 3.3.3). It is designed to have comparable protection characteristics to mechanisms such as stack canaries [33], shadow stacks [35], or the HDFI stack protection policy [116], namely the protection of return addresses stored on the stack. We abbreviate "return address" with RA in our tags and rules.

Because the policy is only concerned with differentiating return addresses stored on the stack from all other stack objects, it only needs two *object-id*s: *RA* and *OTHER*. As another simplification, we will not differentiate return addresses by any notion of their owner, thus choosing to use a single *frame-id* in all cases. Conceptually, this is equivalent to removing the *frame-id* field from the tags for this policy; we choose this interpretation for the rest of the section. The full rules for the policy are available in Fig 2.

$(1) \mathit{Store} : (\bot, \mathit{STORE–RA}, \bot, \bot, \mathit{OTHER}) \rightarrow (\bot, \mathit{RA})$
$(2) \mathit{Load} : (\bot, \mathit{READ–RA}, \bot, \bot, \mathit{RA}) \rightarrow (\bot, \bot)$
$(3) \mathit{Store} : (\bot, \mathit{REMOVE–RA}, \bot, \bot, \mathit{RA}) \rightarrow (\bot, \mathit{OTHER})$
$(4) \mathit{Store} : (\bot, \mathit{INSTR}, \bot, \bot, \mathit{OTHER}) \rightarrow (\bot, \mathit{OTHER})$
$(5) \mathit{Load} : (\bot, \mathit{INSTR}, \bot, \bot, \mathit{OTHER}) \rightarrow (\bot, \bot)$
$(6) \mathit{Other} : (\bot, \mathit{INSTR}, \bot, \bot, \bot) \rightarrow (\bot, \bot)$
$(6) \mathit{Store} : (\bot, \mathit{LONGJMP–CLR}, \bot, \bot, \_) \rightarrow (\bot, \bot)$

Figure 2: Return Address Protection rules

Policy Implementation: This policy requires support from the compiler only to appropriately tag the instructions that store and retrieve return addresses from the stack. Specifically, the compiler tags the instruction in the function prologue that stores the return address to the stack with a special tag *STORE–RA*, which, with an appropriate rule, causes the written memory word to become tagged *RA*. Similarly, the compiler tags the instruction in the function epilogue generated to retrieve the return address from the stack with a special tag *READ–RA*. With an appropriate rule, instructions with this tag are granted the unique permission to read words marked *RA* from the stack.

In this policy all other memory words are tagged *OTHER*, and all other instructions are tagged generically as *INSTR*. Instructions tagged *INSTR* are permitted to access memory words tagged *OTHER* but not those tagged *RA*.

One final detail wraps up the policy: in standard stack disciplines, the return address (which we will have tagged *RA*) is left on the stack after a function returns. We insert one additional instruction in the function epilogue that cleans up the *RA* tag left on the stack by performing a store to the word containing the return address. This cleanup instruction is tagged *REMOVE–RA* by the compiler, granting it the unique permission to overwrite words tagged *RA*, which it tags with the generic *OTHER*.

Security Properties: The Return Address Protection policy uses information from the compiler and appropriate rules to keep return addresses saved on the stack tagged RA and all other words tagged as $OTHER$. Only specific instructions generated by the compiler to manage the stack abstraction have permission to access words tagged RA, which prevents any other code from overwriting them to hijack control flow. Separately, instructions that load return addresses from the stack require valid RA targets; this prevents an attacker from tricking the machine into using an attacker-synthesized return addresses, such as in a typical ROP attack.[1]

This policy is complementary to CFI policies that restrict the control-flow edges taken by a program to match those of a control-flow graph. Return edges are imprecise in that they can potentially return to any of their call cites [2]; the additional protection for return addresses in memory could replace a shadow stack proposed by [2] for this purpose.

Static Authorities

Policy Description: The next policy we present, Static Authorities, greatly expands upon the set of *object-id*s and *frame-id*s that will be used to differentiate objects on the stack. The key design decision of the policy is to statically assign a unique identifier to each function in a program, and to reuse that same identifier as the *frame-id* for each dynamic function instance that is pushed onto the runtime call stack. Conceptually, each function will tag the stack memory that it allocates with its unique *frame-id*, and instructions belonging to that function are the only instructions tagged in the appropriate way to access (or create pointers to) that allocated memory. In this sense, each function in a program is the authority over the memory that it allocates.

In this policy we enrich our notion of *object-id*s for precise object protection internal to a frame. Within each frame we statically assign a unique *object-id* to each program-level variable used by that function, including each primitive, array and structure in the frame; i.e., for each variable V_i belonging to a function f we assign a new differentiable *object-id* i.

[1]We note, however, that this simple policy would not prevent sophisticated code reuse attacks, e.g., [27]. Our later policies provide protection for other code pointers on the stack as well.

Like Return Address Protection, we continue to use additional *object-id*s to manage the stack control data, but now we expand the set to include the return address, the saved frame pointer and callee-saved registers; these other objects can also be used to mount attacks, e.g., [70, 32]. Due to the restricted way in which these compiler-managed objects are accessed (Sec. 3.3.3), we reuse the same *object-id* for them all; we only need to isolate them from the other program-managed objects on the stack to secure them. Leveraging this piece of static analysis allows us to avoid unnecessary tag and rule diversity.

At a high level, the implementation is then concerned with (1) tagging stack memory according to the Static Authorities formulation above, and (2) tagging instructions and defining appropriate rules to validate accesses to these stack objects to enforce the invariants (Sec. 3.3.3). The full rules for the policy are available in Fig. 4 and are referenced throughout the next section. In Fig. 3 we show an example of how the stack memory would be tagged when our tagging scheme is applied to the code shown. For demonstrative purposes, we assume the first argument is passed on the stack.

Policy Implementation:

Initialization: To initialize this policy, we tag all stack memory words with a special tag, *EMPTY_STACK*, indicating that the cell is unclaimed.[2] Instructions are tagged with both their corresponding *frame-id* (authority identifier) and an *instruction-type* field that is set generically as *INSTR* unless otherwise indicated below. We initialize non-stack memory to $\perp$.

Tagging Stack Memory: In each function prologue, a function first tags the stack pointer with its *frame-id* using the instruction that decrements the stack pointer (rule 1). Next, the function uses instructions added by the compiler to tag the freshly allocated stack words with their appropriate *frame-id* and *object-id*. These instructions are tagged with both the *instruction-type* *SET_MEM* and the *object-id* that they are initializing; with rule 2,

[2]For simplicity, we assume a fixed, maximum stack size, although with additional OS and loader support stack pages could be allocated lazily and tagged on demand as they are faulted in.

```
long square(long i ){
    long r = i * i ;
    return r ;
}
int main(){
    long x = 3;
    long r ;
    r = square(x) ;
}
```

(a) A simple function to illustrate the Static Authorities tagging scheme. The **`main`** function has been assigned one *frame-id* (green), and **`square`** has been assigned another *frame-id* (blue).

Data	Tag
-	EMPTY-STACK
-	EMPTY-STACK
-	EMPTY-STACK
`main`'s arg for square	obj-2, argfor=square
`main`'s return address	obj-1
`main`'s frame pointer	obj-1
`main`.x	obj-3
`main`.r	obj-4

(b) The state of stack memory at the time when **`square`** is called by **`main`**. In **`square`**'s prologue, it will first tag the stack pointer with its blue identifier (rule 1), and then tag the stack elements in the new frame (rule 2). After this tagging is complete, the stack will look like (c).

Data	Tag
`square`'s return address	obj-1
`square`'s frame pointer	obj-1
`square`.r	obj-2
`main`'s arg for square	obj-2, argfor=square
`main`'s return address	obj-1
`main`'s frame pointer	obj-1
`main`.x	obj-3
`main.r`	obj-4

(c) **`square`**'s frame is now tagged. When **`square`** accesses a local variable, the tag on the instruction will be checked against the tag on the memory word (rules 3 and 4) to validate the access. The **`square`** function will be permitted to read its argument out of **`main`**'s frame using rule 16.

Data	Tag
-	EMPTY-STACK
-	EMPTY-STACK
-	EMPTY-STACK
`main`'s arg for square	obj-2, argfor=square
`main`'s return address	obj-1
`main`'s frame pointer	obj-1
`main`.x	obj-3
`main`.r	obj-4

(d) Before **`square`** returns, it will first release its stack memory by clearing the tags back to the *EMPTY-STACK* state (rule 3). The stack is now in the same state as it began (b).

Figure 3: An example illustrating the Static Authorities tagging scheme. In this example we show how **`square`**'s memory would be tagged when it is called from **`main`**, which has already tagged its memory. The referenced rules are show in Fig. 4

SET_MEM instructions become the only type of instructions that can claim empty stack memory, which they convert from *EMPTY_STACK* to the appropriate *frame-id* and *object-id* of the allocated word. Functions that do not allocate stack memory (e.g., handwritten assembly code in *libc*) tag no memory—they require no stack protection.

Tagging and Propagating Pointers: The compiler places the *MAKE-PTR instruction-type* along with the *frame-id* and appropriate *object-id* on instructions that create pointers to stack objects (rule 1). We use the same dynamic tainting rules as in [30] to propagate pointer tags between registers (rules 6-10), as well as to and from memory (rules 13 and 14).

Accessing Objects: The way in which accesses to stack objects are validated depends on the access type. For direct frame pointer offset accesses, instructions are tagged with the *instruction-type* ACCESS_LOCAL and the specific *object-id* that they access; these accesses use the *frame-id* from the frame pointer (rules 4 and 5). For the general pointer case, a an access is allowed when the *frame-id* and *object-id* of the accessing pointer matches the *frame-id* and *object-id* of the stack word (rules 13 and 14).

Retagging the Stack Pointer: After each function call, the compiler inserts one instruction to tag the stack pointer back to the authority identifier of the caller (rule 1). The frame pointer gets the correct tag by retrieving the stored frame pointer from the stack memory in the function epilogue (rule 14).

Passing Arguments: To handle the special case of argument passing, the Static Authorities policy sets aside a special *object-id* for arguments (*ARG*) and tags stack words that contain passed arguments with this special *object-id* using rule 17. These argument words are extended with another field, *argument_for*, containing the authority (*frame-id*) of the intended consumer. Access to words marked *ARG* are permitted with rules 15 and 16 if the accessor's *frame-id* matches the argument's indicated *argument_for* field. The way in which we tag *ARG*s with the appropriate authority identifier of the expected callee depends on the type of function call. For direct calls, the needed information is trivially available to the compiler, and these words can be set up by appropriately tagging the instructions that prepare the arguments before the call instruction. For indirect calls (in which the callee authority identifier is not known statically), we add additional fields to keep function pointers tagged with their appropriate *frame-id*, so that at runtime we can setup the argument words with correct *frame-id* based on the dynamic function pointer being used. We describe these details in Sec. 3.7.

Clearing Memory: To clear a function's allocated memory, the compiler adds additional instructions into the function epilogue tagged *CLEAR_MEM* that, with rule 3, allow the program to release the stack memory allocated by the function by retagging the words currently owned by the function's *frame-id* with the tag *EMPTY_STACK*. We choose epilogue clearing over prologue clearing to limit the writing privilege of each function to just the memory that it has allocated itself.

For readers interested in additional low-level policy details, the uses of the other rules are discussed in Sec. 3.7. A reader considered with only the high-level policy behavior may continue on here.

Security Properties: The Static Authorities policy tags each object on the stack with a *frame-id*, indicating which function owns the object, as well as an *object-id*, indicating which object held by that frame is stored there. Accesses to stack objects are validated with compiler assistance, using tags on instructions and pointers. Accesses are permitted only if the correct *frame-id* and *object-id* are used, preventing the out-of-bounds accesses that give rise to stack attacks; both *inter-frame* and *intra-frame* violations are prevented with the Static Authorities tagging scheme. However, in order to achieve cacheability of the metadata rules, the policy does reuse the same *frame-id* for each dynamic instance of a function. This reuse constrains the number of tags and rules that are generated to remain modest, i.e., remain proportional to the number of active functions in an application. It also means that the policy does not differentiate between dynamic instances of a stack object; it shares this limitation with systems built on static points-to analysis like WIT [4] and others [26]. The Static Authorities policy provides both spatial and temporal security properties—a dangling pointer is still bound to its specific *frame-id* and *object-id*.

Non-stack pointers are tagged $\perp$, which prevents them from accessing stack memory. Stack pointers are prevented from accessing other memory regions, which are tagged $\perp$. These rules prevent gross cross-region violations, including “stack clashes” [103]. Additionally, by combining these rules with strict epilogue rules that require the stack pointer tag to not be $\perp$, the policy protects against stack pivots similar to [101].

Depth Isolation

Policy Description: The last policy we present, Depth Isolation, is constructed in almost the same way as Static Authorities. However, instead of using a unique function identifier to serve as the *frame-id*, the Depth Isolation policy uses the current stack depth, d, as the *frame-id* for each function instance—this allows the policy to discriminate between dynamic instances of a particular stack object. The policy uses the same set of differentiable objects within a frame as in Static Authorities: that is, a unique *object-id* for each program variable, an *object-id* for stack control data, and an *object-id* for argument passing.

Conceptually, the system will maintain the current stack depth, d, and all functions will use it to tag the dynamic instances that they allocate. The full rules for the policy are shown in Fig. 5.

Policy Implementation: Our Depth Isolation implementation differs from Static Authorities in only a few aspects, so we present the differences here. The other implementation details are the same.

Maintaining Stack Depth: This policy requires tracking the current stack depth to serve as the *frame-id*, which we choose to place in the tag on the stack pointer register. In the

$(1) \mathit{Arith} : (\bot, (\mathit{MAKE–PTR}, f, o), \bot, \_, \bot) \rightarrow (\bot, (f, o, \bot))$

$(2) \mathit{Store} : (\bot, (\mathit{SET–MEM}, f, o), \bot, (f, \_, \bot), \mathit{EMPTY–STACK}) \rightarrow (\bot, (f, o, \bot, \bot, \bot))$

$(3) \mathit{Store} : (\bot, (\mathit{CLEAR–MEM}, f, \bot), \bot, (f, \bot, \bot), \_) \rightarrow (\bot, \mathit{EMPTY–STACK})$

$(4) \mathit{Store} : (\bot, (\mathit{ACCESS–LOCAL}, f, o), (f2, o2, p), (f, \bot, \bot), (f, o, \_, \_, \_)) \rightarrow (\bot, (f, o, f2, o2, p))$

$(5) \mathit{Load} : (\bot, (\mathit{ACCESS–LOCAL}, f, o), \_, (f, \bot, \bot), (f, o, f2, o2, p)) \rightarrow (\bot, (f2, o2, p))$

$(6) \mathit{Arith\_prop} : (\bot, (\mathit{INSTR}, \_), \bot, \bot, \bot) \rightarrow (\bot, \bot)$

$(7) \mathit{Arith\_prop} : (\bot, (\mathit{INSTR}, \_), \bot, (f, o, p), \bot) \rightarrow (\bot, (f, o, p))$

$(8) \mathit{Arith\_prop} : (\bot, (\mathit{INSTR}, \_), (f, o, p), \bot, \bot) \rightarrow (\bot, (f, o, p))$

$(9) \mathit{Arith\_prop} : (\bot, (\mathit{INSTR}, \_), (f1, o1, p1), (f2, o2, p2), \bot) \rightarrow (\bot, \bot)$

$(10) \mathit{Arith\_no\_prop} : (\bot, (\mathit{INSTR}, \_), (f1, o1, p1), (f2, o2, p2), \bot) \rightarrow (\bot, \bot)$

$(11) \mathit{Store} : (\bot, (\mathit{INSTR}, \_), \bot, \bot, \bot) \rightarrow (\bot, \bot)$

$(12) \mathit{Load} : (\bot, (\mathit{INSTR}, \_), \bot, \bot, \bot) \rightarrow (\bot, \bot)$

$(13) \mathit{Store} : (\bot, (\mathit{INSTR}, \_), (f2, o2, p), (f1, o1, \bot), (f1, o1, \_, \_, \_)) \rightarrow (\bot, (f1, o1, f2, o2, p))$

$(14) \mathit{Load} : (\bot, (\mathit{INSTR}, \_), \_, (f1, o1, \bot), (f1, o1, f2, o2, p)) \rightarrow (\bot, (f2, o2, p))$

$(15) \mathit{Store} : (\bot, (\mathit{INSTR}, f), (f2, o2, p2), \_, (\_, \mathit{ARG}, \_, \_, \_, \mathit{ARGFOR} = f)) \rightarrow$
$\quad (\bot, (f, \mathit{ARG}, f2, o2, p2, \mathit{ARGFOR} = f)$

$(16) \mathit{Load} : (\bot, (\mathit{INSTR}, f), \_, \_, (\_, \mathit{ARG}, f2, o2, p, \mathit{ARGFOR} = f)) \rightarrow (\bot, (f2, o2, p))$

$(17) \mathit{Store} : (\bot, (\mathit{SET–ARG}, f1, f2), (f3, o3, p), (f1, \bot, \bot), (f1, \_, \_, \_, \_)) \rightarrow$
$\quad (\bot, (f1, \mathit{ARG}, f3, o3, \mathit{ARGFOR} = f2))$

$(18) \mathit{Arith} : (\bot, (\mathit{CREATE–FP}, f, p), \bot, \_, \bot) \rightarrow (\bot, p)$

$(19) \mathit{Store} : (\bot, (\mathit{LONGJMP–CLEAR}, \_), \bot, \_, (\_, \_, \_, \_, \_)) \rightarrow (\bot, \mathit{EMPTY–STACK})$

$(20) \mathit{Other} : (\bot, (\mathit{INSTR}, \bot, \bot), \bot, \bot, \bot) \rightarrow (\bot, \bot)$

$(21) \mathit{Arith\_prop} : (\bot, (\mathit{BEGIN–INDIRECT–CALL}, f), \bot, (\bot, \bot, p), \bot) \rightarrow (p, \bot)$

$(22) \mathit{Store} : (pc, (\mathit{SET–ARG–FROM–PC}, f), (f2, o2, p), (f, \_, \bot), (f, \_, \_, \_, \_)) \rightarrow$
$\quad (pc, (f, \mathit{ARG}, f2, o2, p, \mathit{ARGFOR} = pc))$

$(23) \mathit{Jump} : (\_, (\_, \mathit{INSTR}, \bot, \bot), \bot, \bot, \bot) \rightarrow (\bot, \bot)$

Figure 4: Static Authorities rules

function prologue, the compiler tags the instruction that allocates the stack frame with *INCR–DEPTH*; (Fig. 5 rule 1), this causes the value held in the tag, *d*, to be updated to *d*+1. Similarly, in the function epilogue, the compiler tags the instruction that releases the stack frame with *DECR–DEPTH*, which, with rule 2, replaces the current depth, *d*, with *d*-1.

Argument Passing: Argument passing in the Depth Isolation policy is simpler than in the Static Authorities policy. We tag stack words that contain arguments with the *object-id* *ARG* and the current depth of caller *d* (rule 17), but we do not need to extend them with *argument_for* as was done in Static Authorities. Instead, in the Depth Isolation policy, we require that the depth of the accessor to argument words is either *d*, the depth of the owner, or *d*+1, the depth that will be used by the callee (rules 18-21); no other depths are permitted to access arguments.

Other: The Depth Isolation policy does not need to retag the stack pointer after returning from a call because there is no authority identifier kept on the stack pointer; the depth decrement by the caller sufficiently resets the stack pointer. In Depth Isolation instructions have no authority identifier and so are only tagged with their *instruction-type* on initialization.

Security Properties: The Depth Isolation policy, like Static Authorities, prevents out-of-bounds accesses to objects on the stack by requiring that the *frame-id* and *object-id* tags of the instruction or pointer match those of the accessed memory word—and so it has similar security properties to Static Authorities. However, the Depth Isolation policy provides better spatial memory safety properties than Static Authorities, as each live function instance (even of the same static function) has a unique *frame-id*. The Depth Isolation policy has weaker temporal guarantees; a dangling pointer tagged for a particular *frame-id* and *object-id* may be able to be used for unintended instances.

$(1) Arith : (\bot, INCR\text{–}DEPTH, \bot, (d, \bot), \bot) \rightarrow (\bot, (d + 1, \bot))$
$(2) Arith : (\bot, DECR\text{–}DEPTH, \bot, (d, \bot), \bot) \rightarrow (\bot, (d - 1, \bot))$
$(3) Arith : (\bot, (MAKE\text{–}PTR, o), \bot, (d, \bot), \bot) \rightarrow (\bot, (d, o))$
$(4) Store : (\bot, (SET\text{–}MEM, o), \bot, (d, \bot), EMPTY\text{–}STACK) \rightarrow (\bot, (d, o, \bot, \bot))$
$(5) Store : (\bot, CLEAR\text{–}MEM, \bot, (d, \bot), \_) \rightarrow (\bot, EMPTY\text{–}STACK)$
$(6) Store : (\bot, (ACCESS\text{–}LOCAL, o), (d2, o2), (d, \bot), (d, o, \_, \_)) \rightarrow (\bot, (d, o, d2, o2))$
$(7) Load : (\bot, (ACCESS\text{–}LOCAL, o), \_, (d, \bot), (d, o, d2, o2)) \rightarrow (\bot, (d2, o2))$
$(8) Arith\_prop : (\bot, INSTR, \bot, \bot, \bot) \rightarrow (\bot, \bot)$
$(9) Arith\_prop : (\bot, INSTR, \bot, (d, o), \bot) \rightarrow (\bot, (d, o))$
$(10) Arith\_prop : (\bot, INSTR, (d, o), \bot, \bot) \rightarrow (\bot, (d, o))$
$(11) Arith\_prop : (\bot, INSTR, (d, o), (d, o), \bot) \rightarrow (\bot, \bot)$
$(12) Arith\_no\_prop : (\bot, INSTR, (d1, o1), (d2, o2), \bot) \rightarrow (\bot, \bot)$
$(13) Store : (\bot, INSTR, \bot, \bot, \bot) \rightarrow (\bot, \bot)$
$(14) Load : (\bot, INSTR, \bot, \bot, \bot) \rightarrow (\bot, \bot)$
$(15) Store : (\bot, INSTR, (d2, o2), (d1, o1, \bot), (d1, o1, \_, \_, \_)) \rightarrow (\bot, (d1, o1, d2, o2))$
$(16) Load : (\bot, INSTR, \_, (d1, o1), (d1, o1, d2, o2)) \rightarrow (\bot, (d2, o2))$
$(17) Store : (\bot, SET\text{–}ARG, (d2, o2), (d1, \_), (d1, \bot, \bot, \bot,)) \rightarrow (\bot, (d1, ARG, d2, o2)$
$(18) Store : (\bot, INSTR, (d2, o2), (d1, \_), (d1, ARG, \_, \_,)) \rightarrow (\bot, (d1, ARG, d2, o2)$
$(19) Store : (\bot, INSTR, (d2, o2), (d1, \_), (d1 + 1, ARG, \_, \_,)) \rightarrow (\bot, (d1 + 1, ARG, d2, o2)$
$(20) Load : (\bot, INSTR, \_, (d1, \_), (d1, ARG, d2, o2)) \rightarrow (\bot, (d2, o2)$
$(21) Load : (\bot, INSTR, \_, (d1, \_), (d1 + 1, ARG, d2, o2)) \rightarrow (\bot, (d2, o2)$
$(22) Store : (\bot, LONGJMP\text{–}CLEAR, \bot, \_, \_) \rightarrow (\bot, EMPTY\text{–}STACK)$
$(23) Other : (\bot, (INSTR, \bot, \bot), \bot, \bot, \bot) \rightarrow (\bot, \bot)$

Figure 5: Depth Isolation rules

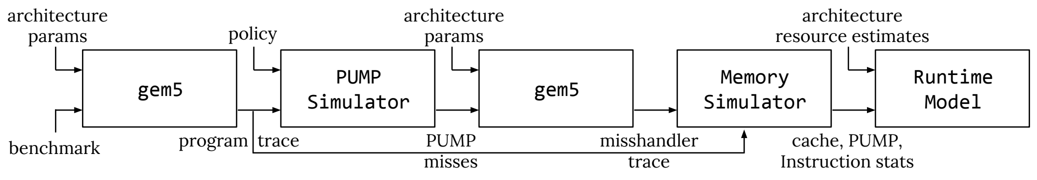


Figure 6: Evaluation Framework

3.4. Evaluation

3.4.1. Methodology

We model the runtime overheads for our stack protection policies on the SPEC CPU2006 [121] benchmark set running on a simulated metadata-enhanced Alpha microarchitecture. We compile the benchmarks using `gcc` with the `-O2` optimization level. We allow each benchmark to complete any benchmark-specific initialization, such as parsing input files or setting up data structures, and then run it for an additional one billion warm up instructions. After completing initialization and warm up, we then collect statistics from the system for a 500M instruction measurement period.

Microarchitecture

For concrete evaluation, we target a single-issue, in-order Alpha microarchitecture with a unified 512KB L2 cache, a 64KB L1 instruction cache and a 64KB L1 data cache. We use a wide-word, coupled metadata implementation for tags, so tags are moved atomically with their associated data words. We simulate a 1024 entry L1 PUMP cache and a 4096 entry L2 PUMP cache. We use the same basic architecture optimizations as in [43]. Shortened metadata tags in our L1 cache system are 11 bits, and shortened metadata tags in our L2 system are 14 bits, with full 64-bit tags in DRAM. At these sizes, running with a 1 GHz clock in a 32 nm process, the L1 and L2 cache access cycles are 1 and 5 cycles for both the baseline and tagged cases based on CACTI [89] estimates. Cache lines are 8 words and require 100 cycles to fetch from DRAM in the no-tag case and up to 130 cycles in the tagged case; since tags live on the same DRAM page with the data, they cost additional cycles for bandwidth but do not require additional latency for page access or writeback. The main memory cache compression from [43] means most cache line accesses can fetch compressed tag descriptions for the cache line and consequently require fewer than 130 cycles to fetch the data and tags from DRAM.

Tagging Instructions

Our stack protection policies require tagging individual instructions in policy-specific ways. Ideally, all instruction tags would be provided by a modified policy-aware compiler. For our prototyping purposes, we use a custom instruction tagger. The instruction tagger takes as input the DWARF [54] debug information generated by `gcc`, which we extract from the benchmark binaries and process using `libdwarf` [7]. This debug information gives the instruction tagger the layout of the stack memory, which it uses to tag instructions as described by the policies.

Simulation

Our evaluation framework is shown in Figure 6. We use `gem5` [18] for architectural statistics and generating instruction traces, a custom PUMP simulator for simulating the metadata tag subsystems of the simulated processor, and CACTI [89] for estimating memory access latencies for the final runtime calculations. After running an initial `gem5` simulation of the application, we process the instruction trace in the PUMP simulator that models the metadata tags on the registers, memory and program counter, as well as computes the SDMP policy rules for creating new tags. We then run a separate, second pass of `gem5` on the SDMP software to generate the instruction trace of the misshandler code itself. Finally, we run a memory simulator to model the memory and rule cache system performance with a composite trace assembled from the benchmark instruction trace, the misshandler trace, and the instructions added by the stack protection policies.

3.4.2. Results

Return Address Protection

The Return Address Protection policy has a mean runtime overhead of 1.2% (Figure 7). The policy needs only 6 static tags and 8 total rules. The small set of rules fits into the

L1 PUMP rule cache; after the misshandler evaluates and installs each of them into the cache, no more cycles are spent on policy evaluation. The misshandler took an average of 21 instructions to evaluate a miss. The runtime overhead comes from the one instruction added to every function epilogue to clear the RA (0.4%) and the additional DRAM cycles to transfer tag-extended memory words (0.8%).

Static Authorities

The Static Authorities policy has a mean runtime overhead of 11.9% (Figure 8). It generates an average of 5,213 tags and 12,412 unique rules. The average L1 rule cache hit rate is 99.76%. 13 out of 24 benchmarks experienced no rule misses in the measurement period at all, and most others experienced very few; only two benchmarks experienced enough misses to incur a $> 1\%$ overhead for resolving security monitor requests. The misshandler took an average of 46 instructions to evaluate a miss. The high degree of locality of rules results from a high degree of locality of tags, which the policy achieves by using a single *frame-id* for all dynamic instances of a function. This causes the number of tags and rules needed by the policy to be driven by the size of the working set of active functions (authorities) in the benchmark. The SPEC benchmarks have an average of 2,507 static functions (including libraries), but we found that only an average of 399 were called at least once, and only an average of 93 were active during the core benchmark behavior. A further reduction in the number of tags comes from a reduction in the number of *object-id*s provided by the compiler's optimizations. Many program-level variables either get allocated strictly in registers or optimized away entirely, meaning that the actual number of stack-allocated variables is much lower than would appear from the program source code. The benchmarks that challenged the rule caches (*gobmk*, *perlbench*, *gcc*) were the ones with large working sets of functions.

Most of the overhead of the policy (60% of the 11.9%, or individually 7.1%) comes from the instructions that are added in the prologues and epilogues to maintain the tags on stack

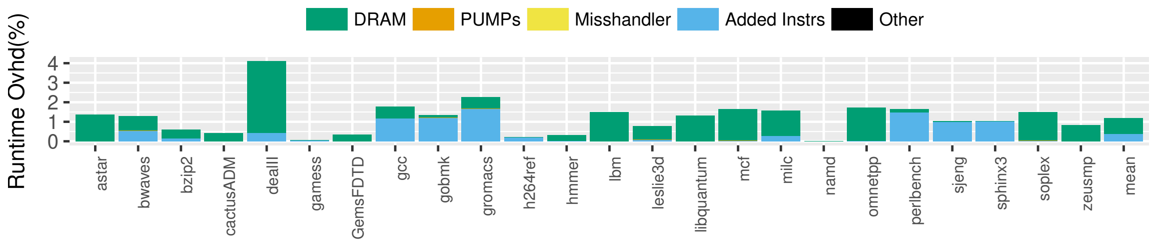


Figure 7: Return Address Protection overhead

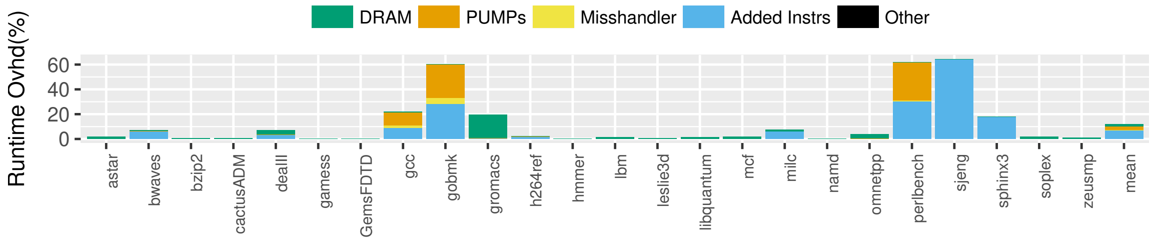


Figure 8: Static Authorities overhead

memory. As can be seen in Figure 8, this alone accounts for an overhead of more than 60% for *sjeng*. *sjeng* is a chess-playing benchmark that rapidly allocates large 16KB stack frames that are defensively sized to hold a worst-case number of chess moves, but in the common case a much smaller number of moves is found and most of the memory goes unused. This causes our policy to spend many cycles setting up and clearing memory tags unnecessarily. Most benchmarks that have a high added instruction overhead have a similar root cause. Some functions in *libc* exhibit this behavior to a lesser degree, such as *_IO_vfprintf* that contains *char work_buffer[1000]*, which is larger than needed in the common case, for example. We attribute this pattern to the programmer's understanding that stack memory is typically cheap (i.e., $\mathcal{O}(1)$) to allocate.

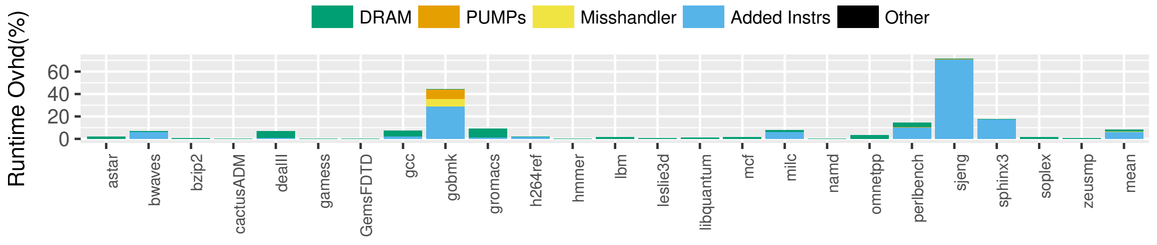


Figure 9: Depth Isolation overhead

Depth Isolation

The Depth Isolation policy has a mean runtime overhead of 8.5% (Figure 9). It generates an average of 1,127 tags and 3,603 unique rules. It has an average L1 rule cache hit rate of 99.98%. 14 of the 24 benchmarks experienced no rule misses in the measurement period, and only one benchmark experienced enough misses to incur a >1% overhead for policy evaluation. The miss handler took an average of 53 instructions to evaluate a miss. The high degree of locality of rules comes from a high degree of locality of tags, which this policy achieves by reusing the *frame-id*s for each dynamic function instance that occurs at the same depth. This locality emerges from the call graph of common applications; rarely do the benchmarks traverse a large range of stack depths, allowing the rules for the depths encountered to remain cached. The benchmarks had an average max stack depth of 60 (median 18) in the full trace, and an average of 32 (median 8) unique depths in the measurement period. The benchmark that most challenged the rule caches for this policy was *gobmk*, a Go playing program that performs some recursive game state operations. This policy, like Static Authorities, also exploits the compiler optimization passes that reduce the number of actual *object-id*s allocated on the stack. The main source of overhead for the policy was also the instructions added to tag and clear stack memory (73% of the 8.5% overhead, or individually 6.2%).

3.5. Optimizations: Lazy Tagging

In the preceding evaluation section, we show that the dominant source of overhead for the stack protection policies arises from instructions added to tag the stack. Consequently, to reduce the overhead we focus on techniques that allow us to reduce or remove the need to add these instructions. Two of the optimizations we present, Lazy Tagging and Cache Line Tagging, allow us to speed up the policies without changing their security properties. The last optimization we present, Lazy Clearing, explores recasting the policies from memory safety policies to data-flow integrity [26] policies in order to remove the instructions that clean up stack memory in the function epilogue. When using this optimization, we consider the policies to be fundamentally different and categorize them separately in our taxonomy (Sec. 3.6.1).

3.5.1. Lazy Tagging

Asymptotically, an unfortunate overhead of the current policy design is the cost of tagging stack elements that are allocated but never used. The ratio of used stack frame words to allocated stack frame words can be arbitrarily small (see discussion about *sjeng* in Sec. 3.4.2). For the stack elements that are used, the need to tag each with their appropriate *frame-id* and *object-id* means the policies are doubling the stack write traffic for stack elements that are only written once. Ideally, we'd like to combine the stack tagging operation with the first program write to the same word to avoid this overhead and simultaneously avoid tagging unused stack elements.

We can address both of these issues for stack writes with the Lazy Tagging optimization, in which we allow all stack pointers to write over *EMPTY_STACK* memory and update the tag on the memory cell to that of the stack pointer or instruction when a write occurs. This eliminates the need to tag stack memory in the function prologue, and so we eliminate those added instructions. From a security perspective, we are still assured that stack pointers and instructions are never used to access claimed (non *EMPTY_STACK*) stack memory that does not match the *frame-id* and *object-id* of the current instruction and stack pointer. We keep the full cleanup loop in function epilogues to maintain the invariant that unused stack frames are marked with *EMPTY_STACK* to allow future function calls to succeed.

A write to the stack beyond the frame's intended allocation will not be prevented nor cleaned up, but it will be caught by a *frame-id* and *object-id* mismatch when a later function attempts to use the memory cell. By removing this initialization, we cut the added instructions roughly in half. When applying Lazy Tagging, the average overhead for Static Authorities goes from 11.9% to 8.9% and the average overhead for Depth Isolation goes from 8.5% to 6.3% (see Figs. 10 and 11).

3.5.2. Cache Line Tagging

Next, and independently from Lazy Tagging, we explore the impact of adding a cache line wide write operation to the Alpha ISA to perform rapid tagging of memory blocks. We model a new instruction for this purpose—this is lightweight to add both for the base datapath and for the metadata rule cache. Typical cache lines are wider than a single word, and the cache memory can read or write the entire line in a memory cycle, so we are exploiting capabilities that the cache already possesses.

To avoid complicating the SDMP rule checking, we demand all words in the cache line have identical tags for this instruction to succeed; this assures the same metadata rule is applicable to every word in the cache line. The SDMP processor applies the single metadata rule and writes the result tag to all of the words in the cache line. If any of the tags on words in the cache line differ, then the instruction instead fails and the machine falls back by jumping to a displacement encoded in the instruction that contains the logic for handling a failure—we model this exception handling code as a series of store instructions that write a value with the same tag as the faulting cache line-wide store instruction would have written.

For this optimization, we align all stack frames to cache lines and model the compiler using the new instruction for the tagging and clearing of stack memory. While this approach does not asymptotically remove the burden of stack frame tagging, it provides an $8\times$ speedup in the best case for the 64-byte cache lines and 8-byte words we assume in our experiments. This significantly reduces the tagging overhead costs for large stack frames such as those used in `sjeng` (See Figs. 10 and 11). We show the impact of both using Cache Line Tagging alone (for both setup and cleanup) and when it is combined with Lazy Tagging (used just for cleanup). When used alone, the average overhead for Static Authorities goes from 11.9% to 7.9% and the average overhead for Depth Isolation goes from 8.5% to 5.5%. When combined with Lazy Tagging, the average overhead for Static Authorities goes from 8.9% to 5.7% and the average overhead for Depth Isolation goes from 6.3% to 4.5%.

3.5.3. Lazy Clearing

Lazy Tagging removes the need for adding instructions in the function prologue to claim memory, but it does not remove the need to clear every allocated word in the epilogue when a function returns. As a result, the policies are still faced with an asymptotic overhead when the allocated stack frame size does not match the actual stack frame usage. Removing the tags from released stack frames is required by the policies so that the subsequent functions, which use the same stack memory, can claim clean cells tagged *EMPTY_STACK*.

In the Lazy Clearing optimization, we remove the tag cleanup loop in the function epilogue and allow all stack writes to succeed. This way, future function calls do not experience violations when they attempt to write over already-claimed memory. When a write occurs, the memory cell gets the authority and object (*frame-id* and *object-id*) for which the write is intended. When using this optimization, we only validate stack *reads*, which assure that the *frame-id* and *object-id* of the stack word being read matches the intent of the compiler as encoded in the instructions and pointers used in the access. Erroneous code can overflow buffers and write indiscriminantly over the stack memory, but the code tagging rules assure that any violations to the stack abstraction will be detected by the reading instruction before the corrupted or unintended data is actually used. Violations that overwrite data that is never read will not be detected, but that's precisely because those violations do not impact the result of the computation since they are not observed. In essence, with this optimization, our policies provide a data-flow integrity property instead of a memory safety property.

This change does mean that the tag on a memory cell during a write can now be uncorrelated to the instruction and stack pointer performing the write. If we needed to supply rules for all combinations of instruction tags, stack pointer tags, and old memory tags, we could end up needing a greater number of rules than in the eager stack clearing case. However, if we

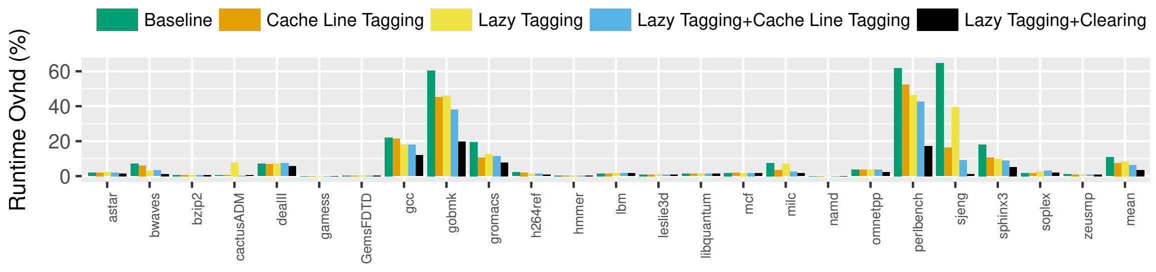


Figure 10: Optimizations applied to Static Authorities

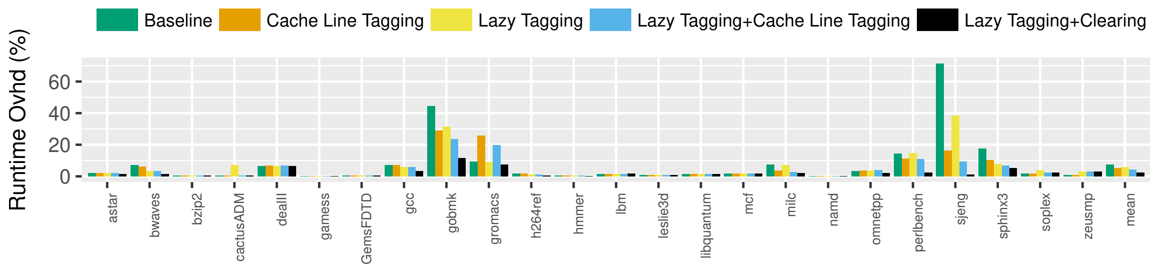


Figure 11: Optimizations applied to Depth Isolation

exploit the ability to indicate that the memory tag is irrelevant to the rule computation (is a don't-care), this will not result in an increase in the number of necessary rules. The don't-care feature exists in [43], and it turns out to be quite important to extracting the benefits of Lazy Clearing for some applications.

While running with the Lazy Clearing optimization, we discovered several cases in the SPEC2006 benchmarks where the original C code does use uninitialized data from the stack. These are errors, and our policy rules correctly flag these errors as violations. They allow data to flow from an unintended *frame-id* and *object-id* and to be used to effect the computation. We believe the correct response is to fix these errors in the original code. To generate a complete and consistent set of data, we selectively disabled lazy optimizations on just the functions that were flagged as using uninitialized data.

The impact of Lazy Clearing, which we always combine with Lazy Tagging, is shown in Figs. 10 and 11. When applied in addition to Lazy Tagging, the average overhead for Static Authorities goes from 8.9% to 3.6% and the average overhead for Depth Isolation goes from 6.3% to 2.4%.

3.6. Security Characterization

3.6.1. Taxonomy

To demonstrate the security properties of our stack protection policies and relate them to other stack protection work, we provide a taxonomy of stack threats in Figure 12. We select threats that decompose stack protection mechanisms along the main dimensions in which they differ and show which protection mechanisms provide protection against each threat.

First, we show whether the protection mechanism prevents the reading of unused stack memory, where previous functions may have left critical data (security keys, etc). Next, we show whether the protection mechanism prevents return addresses from being overwritten, which is the most common vehicle for control flow hijacking attacks. We differentiate between two kinds of memory safety attacks as in [35], the contiguous case and the arbitrary case. In the contiguous case, an attacker must access memory contiguously from an existing pointer (e.g., the attacker controls the source of an unchecked `strcpy`); in the arbitrary case, an attacker can access memory arbitrarily (e.g., the attacker controls the source of an unchecked `strcpy` and the index into the destination buffer).

Many stack protection mechanisms only protect return addresses. However, many of the other items stored on the stack are security-critical as well—these include code pointers such as function pointers, permissions bits, security keys and private information among many other possibilities, so the last threats in the taxonomy concern accesses to other stack data. We differentiate read accesses (R) from read/write accesses (✓) to discriminate where violations are detected and enforced in different policies. Finally, we show the overhead for each of the protection mechanisms.

3.6.2. Microbenchmarks

Due to the difficulty of porting an existing security benchmarking suite such as RIPE [132] to Alpha, we instead constructed a set of security microbenchmarks for testing and

		Read freed stack memory	Contiguous access return address	Arbitrary access return address	Contiguous access wrong stack object	Arbitrary access wrong stack object	Overhead
Existing Work	StackGuard [33] [35]	X	✓	X	X	X	2.8%
	Parallel Shadow Stack [35]	X	✓	X	X	X	3.5%
	SmashGuard [96]	X	✓	✓	X	X	$\sim$ 0%
	Intel's Control-flow Enforcement Technology [65]	X	✓	✓	X	X	
	AddressSanitizer [111]	X	✓	X	✓	X	73%
	CPI/CPS [72]	X	✓	✓	X	X	8.5%/1.9%
	Hardware-Assisted Dataflow-Isolation [116]	X	✓	✓	X	X	$<$ 2 %
	SoftBound [91]	✓	✓	✓	✓	✓	67%
	HardBound [41]	✓	✓	✓	✓	✓	5-9%
Our Policies	Return Address Protection (Sec. 3.3.4)	X	✓	✓	X	X	1.2%
	Static Authorities (Sec. 3.3.4)						
	Memory Safety	✓	✓	✓	✓	✓	5.7%
	Data-flow Integrity	✓	R	R	R	R	3.6%
	Depth Isolation (Sec. 3.3.4)						
	Memory Safety	✓	✓	✓	✓	✓	4.5%
	Data-flow Integrity	✓	R	R	R	R	2.4%

Figure 12: Stack threat taxonomy

characterizing our policies. We use a simple vulnerable C program for each of the threats in taxonomy and craft payloads that allow an attacker to execute the threat shown. Our system halts the offending program at the expected instruction when we display a ✓in the taxonomy and does not halt the program when we display X. Note that for the rest of the security mechanisms in the taxonomy, the ✓or X comes from our understanding of the work and not an empirical evaluation.

3.7. Policy Compatibility

Supporting setjmp/longjmp and Exceptions: System code written in C, as well as the SPEC benchmarks, occasionally use *setjmp()* and *longjmp()*, in which key program state (including the PC and frame pointer) is stored to a memory buffer and later restored. The *longjmp()* operation causes the machine to pop many stack frames with no unwinding operations; as a result, all of the discarded memory would remain tagged, which would later cause our eager policies to encounter violations. To handle this functionality, we add additional code into the *longjmp()* routine that includes a store instruction with a special *LONGJMP–CLEAR* instruction tag; this tag allows it to overwrite the discarded memory, which it tags with *EMPTY_STACK* (Fig. 4 rule 19). These stores are violations of the stack invariants as discussed in Sec. 3.3.3; we are granting additional power to the *longjmp()* routine through this special *instruction-type*. Similarly, C++ exceptions could be handled by providing additional power to the exception handling code with special instruction tags. In the Depth Isolation policy, the stack depth d is stored on the frame pointer and retrieved appropriately by the standard policy rules, so after *longjmp()* the system again has the correct depth that was active at the time of the *setjmp()*.

Tail Call Recursion: Tail call and sibling call elimination optimizations allow a program to reuse a caller's stack frame for its callee in the special case of tail calls. These optimizations are activated with *gcc*'s *-foptimize-sibling-calls* optimization pass which is included in the -O2 optimization level. Our policies retag stack frames, as the authority identifer may have changed. Additionally, arguments prepared for one authority (in the *argument_for* field) may be stale for the new authority identifier after a sibling call. To handle this case, we insert instructions with a special DELEGATE_ARG tag that allows an authority to permanently forgo its access rights and grant them to the sibling authority before making a sibling call.

Dynamic Stack Allocations: Programs can perform dynamic memory allocations on the stack using *alloca()* or by using dynamically sized arrays. We insert additional instructions to tag this memory at the time of the allocation, and similarly insert additional instructions to clear the allocated memory when the stack pointer is again incremented. Note that these setup and cleanup operations are not in the function prologue or epilogue, in contrast to the tagging operations discussed in the policy descriptions. A current limitation of our implementation is that we assign the same *object-id* to all dynamically allocated stack objects. Dynamic stack memory allocations are very rare in the SPEC benchmarks.

Variadic Functions: The C language includes support for *variadic functions*, that is functions that take a variable number of arguments. Another limitation of our current framework is that we assign the same argument identifier to all such arguments. In future work, it would be possible to assign identifiers to those arguments based on their argument number and replace the macros that access those arguments with tag-aware variations for finer protection.

Supporting Indirect Calls: In this section we explain how we tag arguments for indirect function calls (as referenced in Sec. 3.3.4). To handle indirect function calls, we track all function pointers in the system with their corresponding *frame-id* by extending the tags on registers and memory words with another field for this purpose. In effect, these tags identify *code locators* [84] that may be used for indirect function calls. When a function pointer is then used for an indirect call, we use its tag to identify the dynamic function that is being called in order to set up its arguments. Before such a call takes place, we insert a special NOP instruction tagged *BEGIN–INDIRECT–CALL*, which, with Fig. 4 rule 21, takes the *frame-id* of the register being used by the indirect call (e.g., *jsr*) and tags the

Program Counter tag with the *frame-id* of the dynamic authority identifier. Instructions that prepare arguments for indirect calls are tagged with a special *SET–ARG–FROM–PC* tag and use the authority identifier held in the program counter tag to set the appropriate *argument_for* field (rule 22). Finally, the indirect call instruction clears the tag on the PC when it executes (rule 23). In other words, we simply use the function pointer tag to intialize the argument tags for the correct dynamic function being called, and then clear the tag state afterwards.

This strategy requires having all function pointers tagged with their appropriate *frame-id*. To achieve this, we tag entries held in structures such as Global Offset Table (GOT) at initialization with their appropriate *frame-id*; these tags then propagate when they are read using rule 14. Function pointers can also be crafted dynamically by a program using arithmetic instructions that compute at offset from the global register. We extend our instruction tagger to cover these cases, and tag these instructions with the *instruction-type CREATE–FP* along with appropriate *frame-id* for the function pointer that they are creating (rule 18). All function pointers used by the SPEC benchmarks are covered by these cases, although there may be other originating sources of function pointers on other systems; for a complete discussion on identifying code locators, see [84].

In the rules we show in Fig. 4, we display tags on instructions as pairs of the form (*instruction-type*, *frame-id*), tags on registers as triples of the form (*frame-id*, *object-id*, *func_ptr*) and tags on memory as 5-tuples of the form (*frame-id*, *object-id*, *frame-id-ptr*, *object-id-ptr*, *func_ptr*). Tags on memory words require these field so that when a stack pointer is stored into stack memory, a future load can produce an appropriately tagged pointer or function pointer identifier (e.g., rule (5)). In some cases we extend fields for particular *instruction-types* as required by the policy.

3.8. Discussion and Conclusion

3.8.1. Related Work

Stack Protection Due to the prevalence of stack memory safety exploits, stacks have been the subject of many defensive efforts [123]. Traditional protection mechanisms such as Data Execution Prevention (DEP) and Address Space Layout Randomization (ASLR) increase the difficulty of conducting attacks, but do not prevent them entirely. For example, DEP does not protect against code reuse attacks such as ROP [113, 24, 92, 115], and ASLR can be subverted with information leaks [86].

Low-overhead, software-only stack protection solutions such as StackGuard [33] and shadow stacks [35] protect return addresses, but do not protect other stack data and can be defeated by attack techniques such as direct writes and information leaks. Recent work found that shadow stacks have a performance overhead of about 10% [35]; we include the optimized Parallel Shadow Stack variant in our taxonomy. Hardware support for shadow stacks has been proposed (SmashGuard [96]); recently Intel has announced upcoming hardware support for the feature in their Control-flow Enforcement Technology [65].

AddressSanitizer [111] instruments all memory accesses with checks against "red zones" in a shadow memory that pads all objects. It protects stack and heap objects, but only against the contiguous write case. It bears a high runtime overhead of 73% and a high memory usage overhead of 3.3×.

A recent research direction has proposed providing full memory safety just for code pointers (Code Pointer Integrity [72]). While this technique provides an effective level of protection for the incurred overhead on commodity hardware, it does not protect all stack data. Recent work has shown that even non-control data attacks can be Turing complete [63]. The SafeStack component of this work explores splitting the stack into a "safe stack" and a "regular stack". Objects that are accessed in a statically, provably-safe way, such as return addresses and spilled registers, are placed onto the safe stack. Other objects, like arrays and structs, are placed on the regular stack. This spatial separation is useful for protecting items on the safe stack and additionally has almost no performance overhead; however, it is opportunistic, protecting the items that can be cheaply protected and, without CPI, provides no protection for items on the unsafe stack. The safe region itself is protected only with information hiding on 64-bit systems, and implementations have been attacked [49].

Hardware-Assisted Data-flow Isolation (HDFI) [116] uses a single metadata tag bit for efficient security checks. This enables it to achieve a low overhead, but with only a single metadata bit it can only provide coarse protection (e.g., just return addresses or just code pointers, similar to our Return Address Protection). It can distinguish two classes of data and make sure that data from one class is not mistaken for data in the other, but cannot provide fine-grained frame and object separation. Recent work shows that single-bit tags, such as needed for HDFI, can be added without changing the physical memory word width by using a separate tag table with low overhead [66]. LowRISC provides two bits of tagging in its memory system that could be used to implement HDFI with its ltag/stag operations [82, 119].

Some commercial products are beginning to provide features that can approximate HDFI. ARM's v8.3 pointer authentication feature could be used on the return address, or other code pointers, to detect tampering [80] without the need for separate tag bits. Using a unique encoding per return point, this can be extended to provide some CFI protection as well. Oracle's Application Data Integrity (ADI) could be used to assign one of its 16 colors to spilled stack frames at a cache-line granularity to serve a similar function to the single tag bit in HDFI [75]. These offerings are available on commercially available chips, but only provide protection similar to our Return Address Protection policy.

Like other data-flow integrity models [26], the DFI variants of our policies keep track of writers to memory words. Instead of using static instructions as writers, our policies use identifiers for stack objects. In this case of Depth Isolation, we differentiate dynamic instances of the same variable. However, in this work we restrict the policies to just stack objects.

Bounds checking approaches such as SoftBound + CETS [91, 90] can provide complete memory safety using software checks, but are expensive (116% overhead). Hardware support for bounds checking, such as HardBoud [41], Intel's MPX [64] and CHERI [134, 29] can reduce these overheads drastically. Metadata tags are an alternative mechanism that can provide memory protection, and so this work can be seen as exploring the space of tag-based policies for memory safety.

SDMP Policies

The stack protection policies we present in this work are complementary to, and can be composed with, other SDMP policies. Prior work has detailed policies for Control-Flow Integrity (CFI) [43, 13], Information-Flow Control (IFC) [12, 11], Instruction and Data Tainting [43], Minimal Typing [43], Compartmentalization [13], Dynamic Sealing [13], Self Protection [13], and Heap Memory Safety [43, 13]. These previous policies did not address protecting the program stack. The previous memory safety work [43] [13] only addressed heap allocated data, where simply instrumenting the allocator was sufficient to build the policies. As we have seen, object-level stack memory protection is significantly more involved. Interesting future work would be to apply some of the optimizations we describe in this work, such as the DFI variants of the policies, to previous heap safety policies.

Policy Applicability

Several systems provide programmable, multi-bit metadata tags that could exploit the policies we derive here [23, 38, 44, 37]. Aries [23] would need to be extended to include tags on memory. Harmoni [38] lacks instruction tags, but does decode control from instructions; most of our uses of instruction tags could be replaced with augmented instructions. Here, Depth Isolation, where ownership comes from depth on pointers, would make more sense than Static Authorities, which would require authority to be embedded in the instructions. The original Harmoni design has only two inputs to its tag update table (UTBL), while some of our rules need 3 inputs, beyond the instruction tag, to track tags on both register arguments and the memory. The SAFE Processor [44] has a hardware isolated control stack, so does not need to use a metadata policy for protecting procedure call control data. The policies in this work can be seen as an option to unify stack protection under the single mechanism of tagged metadata, rather than adding a separate mechanism for just protecting stack control data. DOVER [37] follows SDMP closely and would be a direct match for our policies.

Emerging flexible, decoupled monitoring architectures support parallel checking of events with metadata maintained in a parallel monitor [28, 55, 129]. LBA and FADE [28, 55] add hardware support to filter and accelerate events with structures similar to the SDMP rule cache. The accelerators in reported designs do not include accelerated handling for metadata on the program counter and instructions, but such extensions appear feasible. As with Harmoni, instruction tags could be handled as augmented instructions. ARMHEx exploits the ARM CoreSight debug port, added instrumentation code, and programmable logic to perform tagged information tracking on existing ARM SoCs such as a Xilinx Zynq [129]. Combining the instrumentation to pass necessary data and programmable logic to implement tracking and checking, it should be able to implement the stack policies described here. The Depth Isolation and Static Authorities policies we describe have richer metadata and are more sophisticated than any of the policies assessed in these monitoring architecture papers.

3.8.2. Limitations and Future Work

Other variations of policies we present could be constructed. With additional compiler support, subfield sensitive policies (i.e., *object-id*s for individual fields of structs) could be derived for stronger protection. Variants of the policies that combine the notions of static owner and depth could overcome the limitations of the Static Authorities and Depth Isolation policies. Our policies do not differentiate between arguments, which would also be a straightforward addition. Policies designed against a stronger threat model (e.g., untrusted code) would also be an interesting extension to this work.

3.8.3. Conclusion

In this work we demonstrate how a general-purpose tagged architecture can accelerate stack protection security policies expressed in the Software-Defined Metadata Processing model. We propose a simple policy that only protects return addresses, as well as two richer policies that provide object-level protection of all stack data. Our policies carry forward information available to the compiler about the arrangement of stack memory and the intent of the various accesses to the stack and validate them at runtime with metadata tags and rules. Our policies exploit the locality properties of typical programs to achieve effective hardware acceleration via a metadata tag rule cache. The main source of overhead incurred by the policies is the instructions added to tag and clear stack memory. We explore optimizations for reducing this overhead, bringing the overheads for our policies below 6% for memory safety and 4% for data-flow integrity. Although we derive our policies in the SDMP model, our designs and optimizations are likely applicable to other tagged architectures.

CHAPTER 4 : Heap Protection Policies

4.1. Introduction

In this chapter, we turn to constructing micropolicies for protecting the heap, a source of *dynamic memory.* In contrast to stack memory, which is *implicitly* allocated and deallocated via function call and return, heap memory is managed *explicitly* by invoking a software component called an allocator. The allocator obtains large blocks of memory from the underlying OS (*e.g.,* with `sbrk`), and in turn it services dynamic memory requests and releases from the application from its pool of memory. Modern allocators (such as the one provided in the GNU C library [124]) are sophisticated and highly-tuned to balance performance and memory fragmentation, among other objectives.

Vulnerabilities related to heap memory and the management thereof can arise from several bug classes. Simple spatial memory errors can also occur on the heap, *i.e., heap overflows* are an analog to *stack overflows* as discussed in the previous chapter. Additionally, manual memory management is notoriously error-prone and introduces several new bug classes that are unique to the heap. In manually managed languages, programmers must strictly obey implicit (and unenforced) allocation rules (do not access freed memory, do not pass a pointer to free more than once, do not pass a pointer to free that did not originate from the allocator, and so on [25]). A failure to uphold these rules on any program execution is undefined behavior and can lead to exploitable vulnerabilities. Heap-based exploits have become very popular among attacks against real systems [17], even overtaking other kinds of memory errors in recent years [127]. Our goal in this section is to enumerate invariants of the heap abstraction, construct micropolicies that can enforce them, and explore the performance and protection tradeoffs of various policy designs.

4.2. Background

4.2.1. The Heap

Dynamic memory in C/C++ provides programmers with a long-lived data store for constructing their applications; stack memory, in contrast, is automatically released when a function returns to its caller. Dynamic memory is managed by a software component called an *allocator*, which is typically included in the C Standard Library, such as `glibc`. In C, programmers directly interact with the allocator through functions such as `malloc` and `free`, which allocate new memory blocks of at least a specified size and release them back to the allocator, respectively. In C++ it is more common to invoke `new` and `delete` to manage object lifecycles. In either case, it is up to the programmer to allocate new objects and release them when they are no longer needed. From the allocator's perspective, a running program performs a series of allocation and deallocation requests, and the allocator's goal is to quickly return memory blocks of the desired size to the program while minimizing fragmentation to reduce the overall memory requirements of that application.

To serve these memory requests, the allocator obtains large blocks of memory from the underlying OS with the `brk` system call which extends the program's data segment. From this large region the allocator divides out smaller, variable-sized *chunks* of memory for the program. Most modern allocators choose to use an *in-band* metadata design [135], where the allocator's metadata (such as the size of the block) is stored inside the allocated chunk at the beginning before the memory that is intended for the program's use; we will use the terminology *payload* to refer to the portion of the chunk that is intended for the program's use, and *control-data* to refer to the allocator's own metadata words inside each chunk. We use *control-data* instead of the more standard *metadata* terminology to avoid conflation with tags, which are also a form of metadata. To create room for the control-data, allocated chunks are several words larger then the requested size of the payload.

When a program calls `free` on a pointer, its corresponding memory chunk can be reclaimed;

allocators will typically maintain lists of freed chunks, and a freshly freed chunk is placed onto an appropriate free list. Future allocations can be satisfied with chunks from a free list, effectively recycling memory such that new calls to `brk` can be avoided and the memory footprint of the application does not grow. Most modern allocators will maintain a number of distinct free lists, usually demarcated by their size, allowing the allocator to more quickly locate available memory blocks of a particular size from the available free lists; this design strategy is known as *binning* [135]. When a chunk is freed, it may be merged with neighboring free memory in a process known as *coalescing*.

The experiments in this section use the `glibc` allocator which is closely based on the `ptmalloc2` design. The layout of a chunk as depicted in `malloc.c` from `glibc` is shown below:

```
    chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of previous chunk, if allocated            | |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of chunk, in bytes                         |P|
      mem-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             User data starts here...                          .
            .                                                               .
            .             (malloc_usable_space() bytes)                     .
            .                                                               |
nextchunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of chunk                                     |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 13: The layout of the `malloc_chunk` structure from the `glibc` allocator.

Note that freed chunks are placed into a doubly-linked list, and the `next` and `prev` pointers are placed following the chunk size when interpretted as a freed chunk.

4.2.2. Heap Vulnerabilities and Exploitation

Manual memory management is notoriously error-prone; programmers must follow a strict set of implicit heap rules (Tab. 15). A violation to any one of these rules is undefined behavior, and the exploitation of this undefined behavior has become a popular attack vector of modern exploits [127]. Heap exploitation is the process of wielding a program

Rule	Bug Classification If Violated
Always check allocation result for NULL	Null Dereference
Only free pointers that originated from allocator	Invalid free
Do not access bytes outside of allocation's payload	Heap overflow / heap underflow
Do not use a freed pointer	Use-after-free
Do not free the same memory more than once	Double-free

Table 1: Rules for dynamic memory management in C and the resulting bug classes if violated; rules shown are abridged, see the SEI CERT C Coding Standard [25].

bug (rule violation) to achieve an attacker's goal, such as hijacking the control-flow of a target process, manipulating its program data, or leaking other information. There are a range of attack techniques that have been developed for targeting the various components of the allocator's attack surface. Many of the techniques depend on low-level features of particular allocators (the memory layouts of chunks, bins and searching algorithms, coalescing strategies, etc); as a result, writing heap exploits typically involves gaining a deep understanding of a particular allocator's internals, and exploits are typically customized to the allocator the attacker assumes will be used by the victim program.

Heap overflows/underflows: Heap overflow and underflow bugs are bugs in which a pointer to a heap-allocated object can be driven outside the bounds of the payload of that allocated chunk, and then dereferenced for illegal reads or writes. It is common for programs to keep many kinds of data on the heap, such as structs, arrays, and many of the program's internal data structures and state. With a heap overflow, an attacker can exploit the bug to write to nearby data elements on the heap, corrupting any data that may be there [17]. Common targets for attackers include data and code pointers; code pointers are of particular interest, in that they can be corrupted to hijack control-flow, *e.g.,* [105]. Heap overflows and underflows are considered violations to *spatial* memory safety, in that a pointer is driven outside of the bounds of the object it targets and then dereferenced.

Use-after-frees and Double-frees: After a pointer has been passed to `free`, that pointer and all other existing pointers to the same object are said to be *dangling.* Dereferencing a dangling pointer is undefined behavior, and doing so is referred to as a *use-after-free.*

Use-after-free bugs can be exploited by attackers to achieve a range of memory corruption goals: for example, if a new object has been allocated in the same memory that has been freed (*i.e.,* the block was placed onto a free-list and then subsequently used to satisfy an allocation), then accesses from the dangling pointer can be used to perform violating reads or writes to the new object.

Additionally, it is undefined behavior to call `free` on the same object more than once, and doing so is called a *double-free.* Double-free bugs can also be exploited by attackers to achieve memory corruption: for example, after the first free, the allocator will place the memory chunk onto a free list and may use it to serve future allocations. If the dangling pointer is freed a second time after the chunk has been recycled for a different allocation, it will inadvertently free the new object's chunk, thus leading to future use-after-frees. Use-after-free and double-free bugs are both examples of *temporal* memory safety violations, in that pointers are used after their targeted objects have logically expired.

Control-Data Corruption: A common set of techniques for achieving reliable exploitation of heap vulnerabilities involve *control-data corruption or manipulation.* These techniques take advantage of the allocator's in-band metadata design, where the allocator's own control-data is stored adjacent to the user data and is exposed for corruption without protection or sanitization. The goal of these attacks is to trick the allocator into performing illegal or invalid heap management operations of the attacker's control by poisoning the allocator's control-data; this corruption can be achieved through any of the above types of heap memory errors. The first known attack of this category is a famous exploit by Solar Designer [39] that uses a heap overflow to manipulate the next and previous chunk pointers in a freed chunk; the `unlink` macro that operates on the linked list data structure of freed chunks can be tricked into nearly arbitrary writes, which he uses to corrupt a code pointer to compromise the Netscape web browser from a JPEG rendering component. In the following years, new techniques for heap exploitation of this variety were developed and refined [9, 85, 67] and continued to defeat hardening efforts of allocators [99, 20, 21].

More recently, Tavis Ormandy of Google Project Zero showed that a single NULL byte overflow into `glibc`'s control-data was sufficient to compromise the Chrome web browser [95], and new techniques for heap exploitation continue to be discovered [135]. Lastly, note real heap exploits on complex software are typically complex, multistage engineering efforts that combine together multiple bugs or use a single bug to trigger additional vulnerabilities; for example, a recent in-the-wild exploit against iOS was reverse engineered and was discovered to first trigger a heap overflow that caused a second, synthetic use-after-free error that was significantly more reliable and controllable than the initial vulnerability [17], a technique known as *vulnerability conversion*.

Heap Grooming: Unlike stack memory, where there is a high degree of spatial and temporal predictability of the memory layout both within a stack frame and in the broader calling context (effectively fixed at compile time), heap memory is dynamic: the spatial and temporal relationships of chunks to one another depend on the (likely data-dependent) sequence of allocation and deallocation events performed by the program, as well as the specific binning, searching and coelescing choices taken by the allocator. In order to reliably control heap vulnerabilities, exploit developers have engineered a range of techniques for achieving predictable (or at least probabilistically likely) chunk arrangements by driving the allocator through a particular sequence of heap operations, *i.e.,* consuming all available freed chunks to trigger a fresh `brk` call (and thus predictable spatial arrangements of subsequent allocations that are carved from a single contiguous fresh data block), or `free` ing a placeholder object of a particular size to cause a victim object to be allocated in its place, and so on. This category of technique includes heap spraying, heap grooming, and heap manipulation [120]. Additionally, it is even possible to fingerprint an allocator version to determine the exact build, data offsets, and functional behavior required for this kind of exploitation [67].

4.3. Threat Model and Assumptions

In developing our micropolicies to protect the heap, we take on a powerful and realistic threat model for systems software. A program written in C or C++ is linked with the glibc allocator (the current default on most Linux systems) with its in-band metadata design. The program may contain vulnerabilities related to the heap, including heap overflow or underflow vulnerabilities, as well as use-after-free or double-free vulnerabilities. Attackers may wield these vulnerabilities by manipulating the program data under their control to trigger bugs and use them to corrupt either heap-allocated program data or allocator control-data. For example, if there is a scanf call without a bounds check, an attacker may supply a large string and use it to corrupt allocator control-data or heap-allocated data objects (Fig. 14). We assume the allocator implementation may be modified to communicate information about its operations to the tagging system.

```
// Vulnerable code: auth overwrite
struct secret { int auth; };
int main(){
   char * name = malloc(32);
   secret * my_secret =
      malloc(sizeof(struct secret));
   init(my_secret);
   scanf("%s", name);
   printf("User %s has auth %d.\n",
        name, my_secret -> auth);
}
```

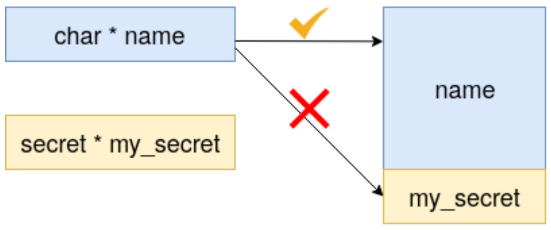


Figure 14: An example heap overflow error that exposes auth to memory corruption from scanf (left) and the heap protection micropolicy halting an invalid access (right).

4.4. Policy Formulation and Implementation

Our core policy design for heap protection micropolicies is based on the dynamic tainting and checking technique introduced in [30], which we translate to an SDMP policy and extend in several ways. The scheme works as follows: when an allocation occurs, the allocated block is painted with a color *c*. When a pointer is created that targets that allocation, the pointer itself is marked with the same color. When a pointer is dereferenced, the color of the pointer and the color of the memory word are compared; if the colors are not the same, then the access is determined to be illegal and it can be halted by the policy. In Fig. 14, the `name` allocation is colored blue and the `my_secret` allocation is colored yellow; when the the `scanf` overflow occurs, the program can be halted when a blue pointer would write to a yellow object.

Color tags on pointers propagate as they move between registers as well as to and from memory. Because a colored pointer can be stored in a colored memory region, colors on memory words contain two separate colors: the cell color (the color of the memory location itself) and the pointer color (the color on the value stored in that location). The variety of policies we consider in this section differ only in their coloring schemes, *i.e.,* how we choose c for each allocation. Unlike the stack policies, which require code instrumentation in addition to tagging, the heap policies are much less intrusive; previous work [30] has shown this taint-tracking scheme to be compatible with x86 and MIPS binaries. In the rest of this section we show how this coloring scheme can be translated into a tag policy for the SDMP used to detect and eliminate invalid heap memory accesses.

4.4.1. Tag Structure

Tags on memory locations are pairs (`CellColor`, `PointerColor`), where each color is an integer value indicating the corresponding taint identifier. Intuitively, the `CellColor` can be thought of as a tag on the memory location (which allocation the word belongs to), while the `PointerColor` is a tag on the data stored there (either the color of the heap pointer, or $\perp$ if the word does not hold a heap pointer). Additionally, tags have two other bits to indicate the presence of two special values: the `Free` special value indicates that the cell holds unallocated heap memory, and the `Allocator` special value is used to mark the allocator's control-data to protect it from the program. If either of these two special tag bits is set, then the color values are ignored by the policy. If neither bit is set and the `CellColor` value is 0, then the memory is interpretted as non-heap memory. To reduce the number of tags and rules required by the policy, `Free` and `Allocator` tags are always supplied in a canonical form where the `CellColor` and `PointerColor` tags are 0.

Tags on registers contain a single field representing a numeric value, `PointerColor`. If the value is 0, then the tag is interpretted as $\perp$, indicating that it is not a heap pointer. Otherwise, the value of `PointerColor` is interpretted as the taint identifier for that pointer.

4.4.2. Tagging Allocated and Freed Memory

The heap policies are built on a foundation of labeling both memory cells and pointer values with taint marks (colors) such that their operations can be checked. Heap memory that has not yet been allocated is marked with the special `Free` tag bit, which indicates that the memory is unallocated heap memory. This tag is placed on all words in new memory blocks that are acquired by the allocator, either through `sbrk` or `mmap`.

The program invokes the allocator by calling one of its interface functions, which in `glibc` includes `malloc`, `calloc`, `realloc`, `valloc` and `pvalloc`. When such a function is called, the allocator partitions out a new chunk, then chooses a `CellColor` value and applies it to each program-usable word in the allocated chunk's payload. These setup instructions are the only instructions in the program permitted to claim `Free` memory, which they convert to allocated memory and label with the appropriate `CellColor` tag. Next, the allocator tags the control-data values inside that chunk with the `Allocator` tag. Both kinds of chunk setup instructions require all words to be previously-marked as `Free`, otherwise the allocator produces a policy violation.

After allocation is complete, the allocator tags the resulting pointer with a `PointerColor` matching the `CellColor` chosen by the allocator. The rules used to govern the propagation of pointer tags are discussed in the following subsection.

When the program has finished using an allocated block, it releases it by calling `free` with an appropriate pointer. The allocator inspects the tag on the incoming pointer; if that pointer is $\perp$, then an invalid free error is detected and the policy raises a violation. Otherwise, the allocator tags each freed word with the canonical `Free` tag. All such freed words must contain a `CellColor` that matches the color on `PointerColor` that was freed, otherwise a policy violation is raised.

The most complex function to manage for proper tagging of heap memory was `realloc`. The `realloc` function takes as input a pointer p and a size s. If the new size s is larger than the existing allocated chunk, then the chunk may grow; depending on the state of the allocator, the chunk may grow in-place or move (including copying bytes from the old location to the new one). If the new size s is smaller than the allocated chunk, then the allocated chunk may shrink. If the incoming pointer p is NULL, then the C standard dictates that the `realloc` be treated as a fresh `malloc`. If the new size s is 0, then the `realloc` is to be treated as a `free`. As such, `realloc` is a complete interface to the allocator and we treat each such case seperately.

If the allocator exhausts its supply of free memory, then the allocation fails and it returns a NULL pointer to the program. On this code path and this code path only, the resulting pointer is tagged with a special `INVALID-PTR` tag, thus granting the program no access to heap memory. Returning an `INVALID-PTR` in is not considered a policy violation, but a future *dereference* of that pointer by the program *is* a violation. This policy design means that a tag violation is thrown only if a program fails to properly handle its out-of-memory error cases.

4.4.3. Propagating Pointer Tags

The allocator sets the *PointerColor* tag on pointers returned to the program, giving it access rights to the freshly allocated block by dereferencing that pointer. The pointer may subsequently be moved between registers or stored-to and loaded-from memory as the program continues to perform its operations. To handle these cases, we apply analogous rules to [30] that transfer the tag along with the data word to which it is associated, such that the `PointerColor` is maintained on pointers to the allocated block.

For example, consider the case where a pointer of color c_{ptr} is stored into a heap region of color c_{cell}. To perform this access, the program *must* have a pointer of color c_{cell} thus granting it access to the memory word. If the value stored there is colored c_{ptr}, then after the store, the tag on that word will be (c_{mem}, c_{ptr}). This tag means that the word is in an allocation of color c_{mem} but holds a pointer to an allocation of color c_{ptr}.

Subsequently, if the program uses a pointer of color c_{mem} it can perform a legal load to that word. If it does, the resulting *register* value would be tagged with c_{ptr}, thus providing the program with an appropriately tagged pointer value for accessing those allocations. In this way, the color tags on pointers are maintained as the program stores and retrieves them from memory.

Note that a pointer may undergo mathematical operations (*e.g.,* pointer arithmetic) as well as `mov` operations between registers, in which case the pointer color is similarly preserved. Other types of instructions such as `xor` remove the pointer color from the result tag. We discuss practical issues that were encountered related to the propagation of pointer tags in Sec. 4.7.

4.4.4. Policy Variations

The policy variations we present in this section are constructed by varying the ways in which new colors are assigned by the allocator to fresh allocations. We include a simple policy that provides only coarse-grained protection using a small number of tags and rules; other policies variations use more colors for stronger protection. In Sec. 4.5 we evaluate the policies on their number of used tags, rules and dynamic rule cache hit rate. In Sec. 4.6 we characterize them in terms of the types of errors and violations they are able to protect against.

One-Color: The One-Color policy uses a single color c for all allocations. It is designed as a lightweight policy that uses a modest number of tags to separate memory words into several classes (allocated heap memory, freed heap memory and non-heap memory) and pointers into two classes (heap pointers and all other values).

N-Color: Rather than assigning a single color to all allocations, the N-Color policy cycles through a pool of N such total colors. This approach is taken by other existing tagging architectures such as SPARC CPUs with ADI [94] which can supply a modest number (8 or 16) of differentiable tags. This allows for more reliable detection and prevention of memory errors at the cost of increased tags and rules, but still limits the total number of identifiers to reduce the rule cache pressure.

Infinite-Color: At the limit, we assign a unique color to each fresh allocation without any tag reuse at all. The Infinite-Color policy generates the most tags and rules, but also provides the strongest security guarantee: complete spatial and temporal memory safety for heap allocations.

Note that an integer `CellColor` of any fixed bit width will eventually exhaust its supply of unique identifiers. This issue is a practical concern for 32-bit integers or smaller, but not likely a concern for 64-bit integers or larger on the time scale of human lives and typical program executions. More importantly, recent work has shown that a program can asynchronously scan its memory and reclaim freed identifiers at a modest cost [76]; in the limit, such an approach would drive the number of required unique identifiers to only match the number of live, extant allocations by a program, a much more manageable task. This approach would allow a policy to achieve the same security properties of the infinite case without excessive tag sizes. Indeed, any real memory system has a finite number of words of memory, which means the number of live tags has a finite bound.

Allocation-Site: Lastly, we consider a final policy variation that takes a slightly different approach from the other variations. Rather than cycling through a pool of colors that is shared by the entire program like the N-Color policy, the Allocation-Site policy cycles through a pool of colors *per allocation-site* in the program. At a color limit of 1, then there would be a unique color per allocation site, but the objects allocated from each of those sites would be undifferentiable from each other. At higher color limits, additional colors per allocation-site are dispensed to separate objects from those sites. This policy variation also reduces the number of tags and rules compared to the Infinite-Color case, but provides the additional guarantee that there will be no color reuse between data types or allocator calls from different program points, a useful security improvement we discuss further in Sec. 4.6.

4.5. Evaluation

Evaluation Framework: We evaluate our heap protection micropolicies using the same basic methodology and architectural parameters as the stack chapter (Chapter 3 Sec. 3.4.1). A custom PUMP simulator is used to simulate metadata tags and policy evaluation logic. We run our experiments on the SPEC CPU2006 benchmarks; each benchmark is run until it finishes its initial setup logic, then it runs for 1 billion warmup instructions followed by a measurement period of 500 million instructions. In this chapter, we exclude the Fortran benchmarks which do not use dynamic memory[1] and perform evaluation only on the C/C++ benchmarks.

Tags, Rules and Colors: The PUMP accelerates security policies by caching metadata rules, and so the diversity and locality of those rules is the major driving force of policy overhead. As a result, one of the first questions to investigate is the relationship between the number of unique identifiers that are assigned to heap allocations and the resulting number of tags and rules that are generated by the policy logic. In Fig. 15 we show the average number of unique rules processed by the PUMP per million instructions as a function of the number of colors used in the N-color policy. If the number of rules per million instructions is smaller than the rule cache size (1024), then the cost of enforcement will be low. When more rules than can fit in the cache are required by the policy, there will be more runtime costs for policy enforcement. At the limit, if the number of rules is so high that the rule cache is thrashed, policy costs can become very high.

The One-Color policy (a maximum colors of 1) uses only ten unique tags and an average of

[1]Dynamic memory was introduced in Fortran 90, but these workloads largely predate those language changes and/or do not use the `ALLOCATE` command. The Fortran benchmarks still use the `libc` runtime which yields runtime calls to `malloc`, but they do not result from direct programmer invocations.

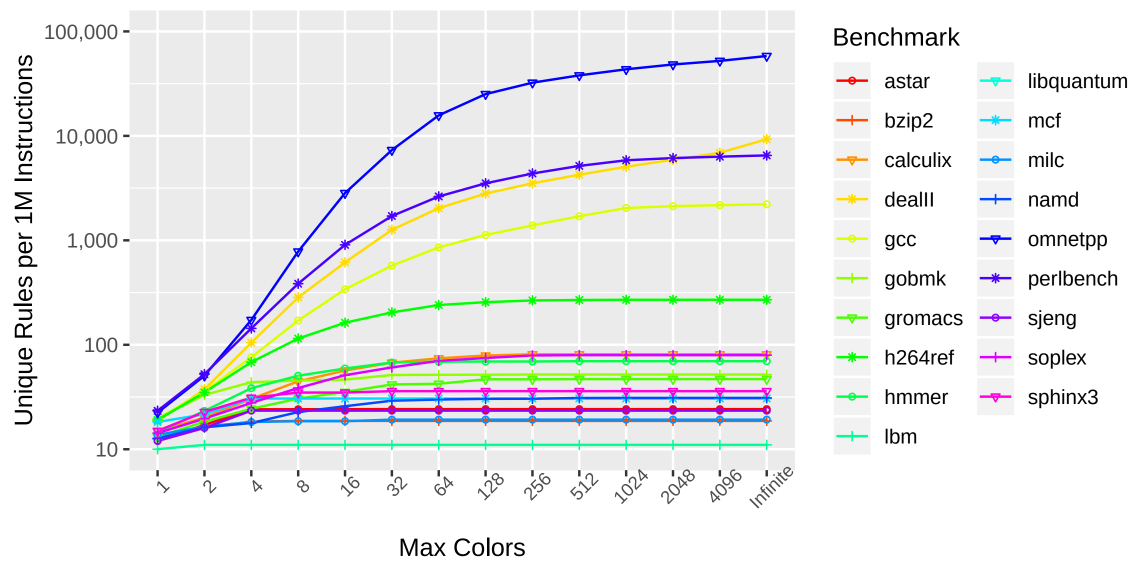


Figure 15: The number of unique rules processed by the PUMP per million instructions during each benchmark's measurement period as a function of the number of colors used to differentiate allocations.

only twenty-two total rules, making it a very lightweight policy that can quite comfortably fit in the rule cache. As more colors are used to differentiate allocations, the total number of tags and rules begins to increase. Most benchmarks see only a modest increase in rules as more colors are used; these are benchmarks that perform only moderate rates of dynamic memory allocations, which means assigning them fresh allocations does not impose much rule pressure. The total number of rules required per million instructions is typically less than one thousand, and many of them require less than a thousand total during their entire measurement period. This means that at the infinite color limit these policies still have manageable rule cache behavior.

In contrast, we also see several benchmarks—`dealII`, `gcc`, `perlbench` and `omnetpp`—that perform rapid memory allocations and generate frequent color pairs throughout their executions. These workloads generate significant numbers of tags and rules, and at high color limits can require thousands to tens of thousands of rules per million instructions. The worst offending benchmark, `omnetpp`, is a network traffic simulator that simulates packets and routers containing packet buffers, each of which are heap-allocated objects. Pointers to packets are frequently stored into packet buffers, generating many combinations of colors. In fact, at a maximum colors of 128, `omnetpp` generates nearly the full 128 × 128 product of all color combinations over its full measurement period. Additionally, `omnetpp` simulates network traffic in small timesteps and updates the entire graph each timestep, producing very little rule locality, a near worst-case scenario for the PUMP and its heap policy.

Overheads and Colors:

Next, we investigate how the number of colors impacts the dynamic rule cache hit rate and thus final runtime overhead of the heap policy variations. In Fig. 16 we show the runtime overhead as a function of the number of unique colors in the N-Color policy, including both the One-Color and Infinite-Color cases. The One-Color policy imposes only a 1% overhead due to the small number of rules required to represent the policy. Even at the Infinite-Color limit, fifteen out of nineteen policies have overheads less than 10%; the four benchmarks with high dynamic rule counts see higher overheads, with `omnetpp` imposing more than 400% overhead. This shows that while most workloads with modest dynamic-memory usage produce rule patterns that can be cached favorably, some workloads challenge the rule cache and require frequent dynamic rule resolutions.

Lastly, in Fig. 17 we show the overheads results for the Allocation-Site policy which dispenses pools of colors per allocation site. Note that in this plot, the number of colors indicates the number of colors *per allocation site*. With only a single color per allocation site, the policy imposes only a 1.2% overhead. The results for this policy trend similarly to the N-Color policy but have favorable security properties, which we explore in the Sec. 4.6.

Case study: Omnetpp The `omnetpp` benchmark was the most challenging workload for the PUMP to cache due to the rapid rate of heap allocations and the frequent generation of color pairs. Rather than limiting the total number of colors across all objects in the system, an alternative approach to reducing overhead costs is to focus specifically on the object types that are responsible for the largest number of tags and rules. To accomplish

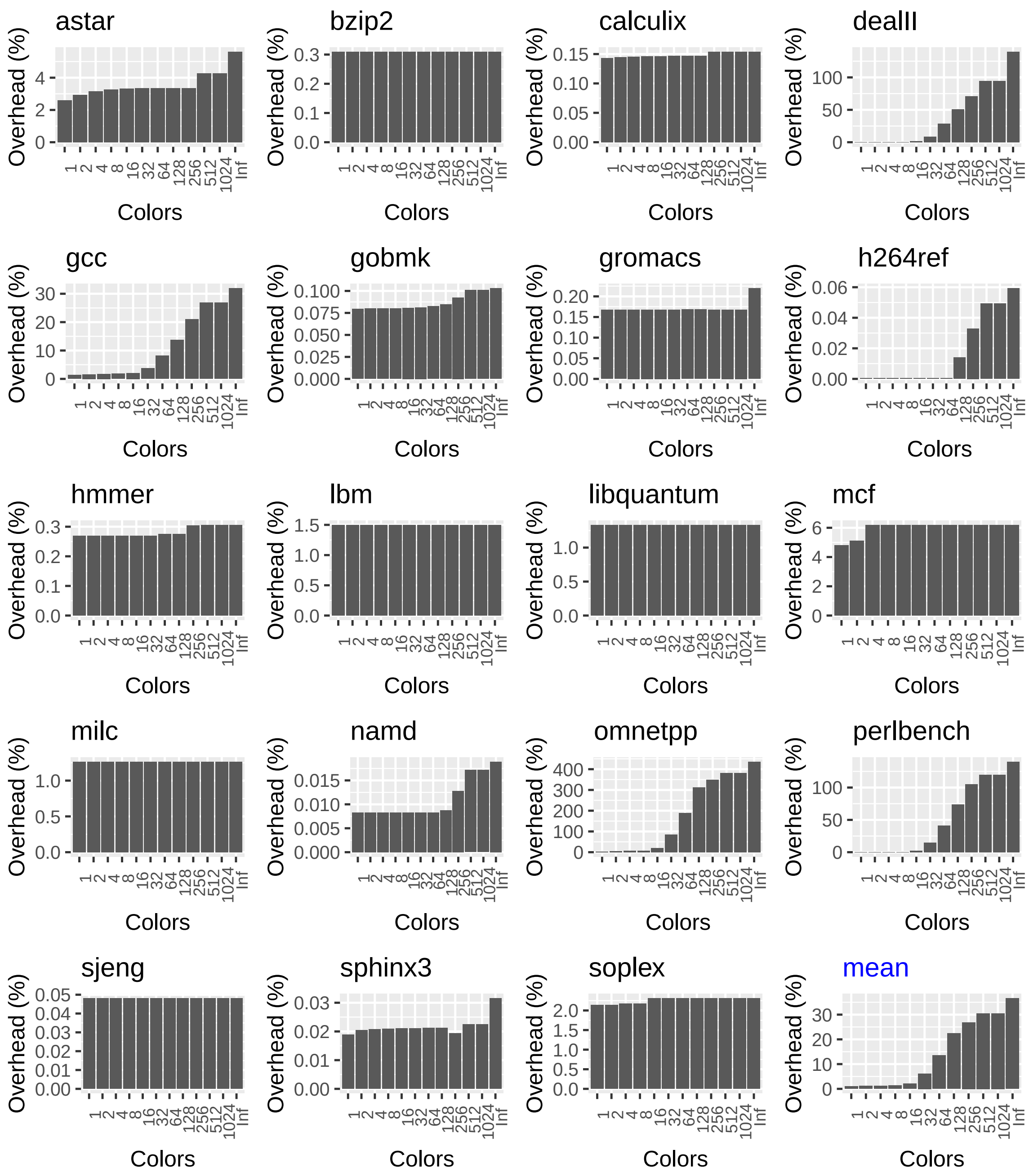


Figure 16: The overhead imposed by the N-color heap micropolicy as a function of the maximum number of unique color identifiers. In this policy, the colors are drawn from a single, global pool of size N.

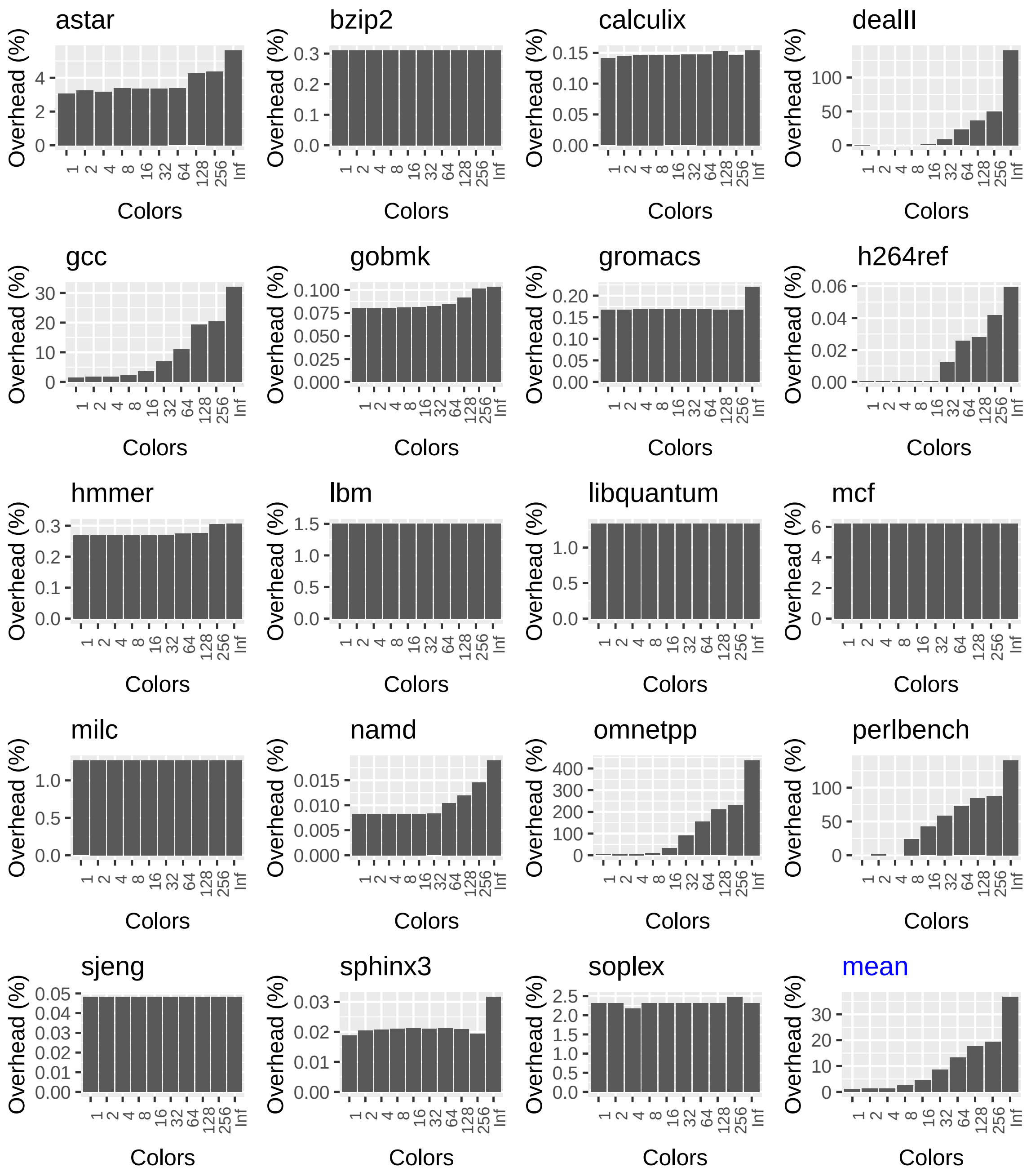
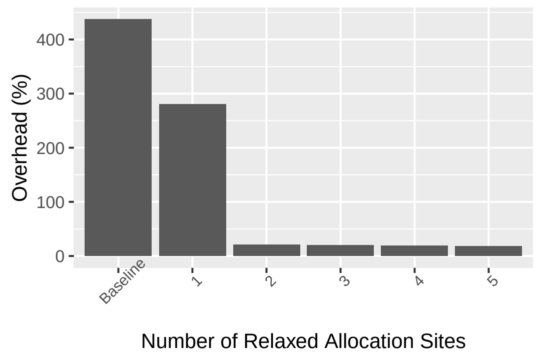
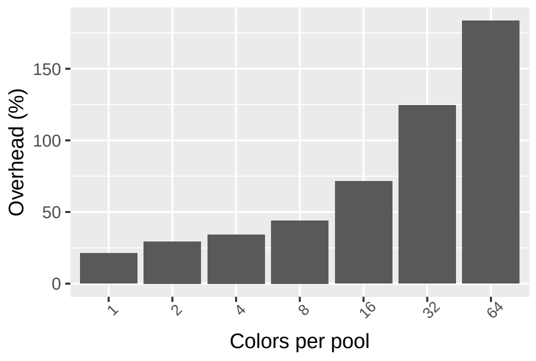


Figure 17: The overhead imposed by the Allocation-Site heap micropolicy. In this policy, a pool of N colors is assigned to each allocation site.



(a) The impact of selectively relaxing the allocation sites most responsible for rule generation in `omnetpp`. Relaxing only two allocation sites is sufficient to reduce the overhead from over 400% to under 20%.



(b) The impact of increasing the color pool sizes for the two allocation sites most responsible for tag and rule diversity. Small pool sizes lead to only marginal increases in enforcement costs.

Figure 18: A case study on relaxing the heap policy on `omnetpp`, the worst-performing benchmark.

this, we add additional tracking into our framework to attribute each tag and rule that is generated back to the allocation site from which it originated. This allows us to calculate the allocation-sites that are responsible for most of the tag and rule diversity; from this data, we can run a hybrid policy in which only a select few allocation sites are treated with the Allocation-Site policy while maintaining the Infinite Color policy for all the rest of the object types in the system.

We conduct two experiments. First, we identify which allocation sites must be relaxed to remove the majority of the rule diversity. For this experiment, we consider relaxing up to the five allocation sites most responsible for the most rules to a single color per allocation site. We show these results in Fig. 18a. As can be seen, by relaxing just two of the allocation sites in this way, the overhead drops from over 400% to less than 20%. The two allocation sites responsible for the high costs in the policy were those for constructing and duplicating packet objects. Second, we show the impact of increasing the pool size per allocation site for these two allocation sites. These results are shown in Fig. 18b. More colors can be used to differentiate these allocations for increasing cost, with only marginal increases in cost for small pool sizes.

Next, we consider the protection offered by the various policy designs. A mitigation is useful when it is able to disarm bug classes by rendering them non-exploitable, typically by providing fail-stop behavior *i.e.,* halting the machine before it would perform dangerous undefined behavior. In this section we consider a range of bug classes inspired by the safe dynamic memory management rules from Tab. 15 and discuss the protection offered by each of the various policy variations.

Null Dereference (1):

```c
// Bug class 1: Null Dereference
int main(){
  char * p = malloc(LARGE_SIZE);
  sprintf(p, "Test");
}
```

In this bug class, a program fails to check an allocator invocation's result for NULL, then proceeds to access memory with the resulting pointer. On some platforms Null Dereference bugs may only lead to system crashes / denial of service attacks, while on other platforms including Windows 7 and prior (as well as compatibility modes for older software on modern Windows systems) they can be exploited for devastating memory corruption [108]. The Null Dereference bug class is prevented from exploitability by the One-Color policy as well as all subsequent policies; when the allocator returns a NULL pointer, it has the `INVALID-PTR` tag, which produces a tag violation if the program attempts to dereference it.

Invalid Free (2):

```c
// Bug class 2: Invalid Free
int main(int argc, char ** argv){
  char * message = malloc(32);
  snprintf(message, 32, "%s", argv[1]);
  ...
  if (error){
    message = "error.";
  }
  printf("%s\n", message);
  free(message);
}
```

In the invalid free bug class, a program calls `free` on a pointer that did not originate from the allocator. If an attacker controls the data the pointer points to (not shown), it can be used to forge synthetic allocator control-data to hijack the allocator's behavior, leading to further memory errors and system compromise. This bug class is also prevented by the One-Color policy and all subsequent policies: legal heap pointers are differentiated from all other pointers by their `PointerColor` tag, and when `free` is passed a pointer with the $\perp$ tag, the PUMP halts program execution. Additionally, valid allocator control-data has the `Allocator` tag, and any forged control-data would not, which would also produce a tag violation when the allocator accesses those words.

Contiguous Heap Overflow (3):

```
// Bug class 3: Contiguous Heap Overflow
struct secret { int auth; };
int main(){
   char * name = malloc(32);
   secret * my_secret =
      malloc(sizeof(struct secret));
   init(my_secret);
   scanf("%s", name);
   printf("User %s has auth %d.\n",
       name, my_secret -> auth);
}
```

In this common and high-severity bug class, a program contains a contiguous memory error in a heap object. A contiguous error is one in which the bug permits only contiguous accesses from an existing buffer, such as this unsafe call to `scanf`. Contiguous errors are less powerful exploit primitives that non-contiguous (arbitrary) errors in which an attacker may also control the offset of the overflow to perform more sophisticated memory manipulation. Contiguous errors may be used to exploit either other heap-allocated objects or allocator control-data: in this case the `auth` field may be overwritten.

This bug class is prevented by the One-Color policy as well as all subsequent policies; the allocated payload of each chunk contains an `Allocator` tag on either side, which prevents contiguous errors from writing beyond their payloads.

Access Free Memory (4):

```c
// Bug class 4: Access Free Memory
int main(int argc, char ** argv){
  char * p = malloc(32);
  free(p);
  sprintf(p, "%s", argv[1]);
}
```

In this bug class, a program reads or writes to memory that is currently free, either via a dangling pointer or double-free. Free memory should never be accessed by a program: for example, while free, a chunk's contents are interpretted as additional allocator control-data which can be written to poison the allocator's behavior the next time it processes that freed chunk.

The integrity of free memory is protected by the One-Color policy and all subsequent policies: free memory is tagged with the special `Free` tag and program code is always forbidden to access it.

Noncontiguous Heap Overflow To Different Object (5):

```c
 1 // Bug class 5: Noncontiguous Heap Overflow To Different Object
 2 struct secret { int auth; };
 3 void init (struct secret * s) { s -> auth = 42; }
 4 int main(int argc, char ** argv){
 5   char * first_name = argv[1];
 6   char * last_name = argv[2];
 7   char * full_name = malloc(32 + 1);
 8   struct secret * my_secret =
 9     malloc(sizeof(struct secret));
10   init(my_secret);
11   snprintf(full_name, 16, "%s", first_name);
12   snprintf(full_name + strlen(first_name), 16, " %s", last_name);
13   printf("User %s has auth %d.\n",
14          full_name, my_secret -> auth);
15 }
```

In this bug class, the program provides the attacker with a powerful non-contiguous memory error on a heap object; we add the additional constraint that exploitation requires corrupting a different object (allocation site). This may occur either because the source object itself does not provide useful footing for an attacker (such as a simple string), or because only one object from that allocation site is live at the time that the bug may be triggered. In this example, a maximum of 16 bytes from each of `first_name` and `last_name` plus a space character are copied into `full_name`, which is 33 bytes. However, on line 12 the length of `first_name` is not verified, which allows that `snprintf` call to become out-of-bounds when a name longer than 16 bytes is provided. As a result, this program allows the corruption of the `auth` field of the `secret` variable from the program's arguments *without* writing to the intermediate `Allocator` tag like a contiguous error would. This bug class typically results from indexing errors or buffer sizing errors on heap-allocated arrays.

This bug is prevented when the color of `full_name` is not the same as other objects that an attacker may want to target, such as `my_secret`. The One-Color policy never satisfies this requirement, and so it fails to prevent this bug class. The N-Color policy *partially* prevents this bug class: whether or not another object that matches the color of `full_name` is on the heap depends on both the program and the number of colors. The allocation-site policy always prevents this bug class, as the color of `full_name` is guaranteed to match no other heap objects, rendering the bug disarmed. This benefit shows us the advantage of binding colors to allocation sites instead of recycling through a global pool.

Use-After-Free to Corrupt Different Object (6):

```c
// Bug class 6: Use-After-Free to Corrupt Different Object
struct secret { int auth; char * message};
struct linked_list { int val; struct linked_list * next };

int main(){

  // Allocate and free an object
  struct linked_list * l  = malloc(sizeof(struct linked_list);
  free(l);

  // Receive same chunk from allocator
  struct secret * my_secret = malloc(sizeof(struct secret));

  // Use-after-free from l to corrupt secret -> auth
  l -> val = 1;

}
```

In this bug class, a dangling pointer from one object type (allocation-site) is used to access the memory of a different type of object. In the example shown above, a dangling pointer to a `linked_list` object is used to corrupt a `secret` object. The One-Color policy does not prevent this bug class, as all objects are marked with the same color. The N-Color policy *partially* protects against this bug class, depending on the number of colors and the capability of an attacker to groom heap operations. The Allocation-Site policy always protects against this bug class by never reusing colors between allocation sites, which guarantees that a dangling pointer can never access another type of object. Like bug class 5, this shows the advantage of not reusing colors between allocation sites.

Arbitrary Spatial or Temporal Error (7): Lastly, we consider an arbitrary spatial or temporal error. This time we make no assumptions about the allocation-sites or error type. Arbitrary logic may take place between allocator events, and the attacker may influence the program's behavior to achieve arbitrary heap grooming operations. The only policy that can protect against this powerful bug class is the Infinite-Color policy, which never allocates a color that is currently in use by any pointer or memory chunk.

Summary: The heap policy variations and the bug classes they defend against are summarized in Fig. 19. We find that the One-Color policy is sufficient to defend against four of these threats (including the common and high severity contiguous heap overflow) with only a handful of tags and rules. The Allocation-Site policy with a single color per site is similarly modest at 1% overhead but additionally defeats cross-object type corruptions compared to just the One-Color policy. The N-Color policy is effective at detecting bugs, but with both high costs associated with larger number of colors and the possibility of a powerful adversary (1) using grooming techniques to match colors even on an invalid access or (2) repeatedly attempting exploitation to brute force a successful intrusion, the policy provides limited additional security comparatively against a strong threat model. As a result, it is wiser to allocate pools of colors per allocation site to spend additional tag diversity for hardening systems, or simply using the Infinite-Color policy. The Infinite-Color policy has a small overhead (<10%) for fifteen of the nineteen applications with lower dynamic memory usage and provides the strongest guarantees; for many workloads, this policy is clearly the best choice. For workloads that perform poorly with the Infinite-Color policy, one can either relax just a couple allocation site that are most responsible for rule generation, or one can apply the general Allocation-Site policy.

Policy	Prevent Null Dereference (1)	Prevent Invalid Free (2)	Prevent Contiguous Heap Overflow (3)	Prevent Access to Free Memory (4)	Prevent Noncontiguous Heap Overflow To Different Object (5)	Prevent Use-After-Free on Different Object (6)	Prevent All Spatial and Temporal Violations (7)	Overhead
One-Color	✓	✓	✓	✓	X	X	X	1.01%
N-Color	✓	✓	✓	✓	—	—	X	1.01% - 37%
Allocation-Site	✓	✓	✓	✓	✓	✓	X	1.2% - 37%
Infinite Color	✓	✓	✓	✓	✓	✓	✓	37%

Figure 19: Security characterization of heap policy variations

4.7. Policy Compatibility

The heap protection policies introduced in this section are built on a scheme of tracking taint colors on both pointers and memory locations. While this scheme is compatible with most software ([30] argue for binary compatibility on x86 and MIPS), the expressive power provided in C/C++ to manipulate the bits and bytes of pointers or inline raw assembly code means that there is no *guarantee* that all existing functional behavior will pass the heap protection micropolicies' taint checks. In this section we explore some of the compatibility issues and porting efforts that were necessary to run the SPEC benchmarks with the heap micropolicies. Sixteen out of nineteen benchmarks did not require any manual porting, whereas three benchmarks did: `gcc`, `perlbench` and `soplex`.

Pointer Color Propagation: The most common issue encountered while constructing, refining, and porting the heap micropolicy to the SPEC benchmarks was determining the pointer-propagation rules for the various types of machine instructions. The vast majority of instruction types can be unambiguously assigned to the always-propagate (*e.g.,* `mov`) or the never-propagate (*e.g.,* `shr`) classes as one might expect. However, for some operation types, the classification is not so simple.

To illustrate this issue, consider this code snippet from gcc in ggc-page.c:

```
1       page = (char *) (((size_t) allocation + G.pagesize - 1) & -G.pagesize);
2       head_slop = page - allocation;
3       tail_slop = ((size_t) allocation + alloc_size) & (G.pagesize - 1);
4       enda = allocation + alloc_size - tail_slop;
```

Figure 20: Pointer tag propagation ambiguity for the `and` opcode in the `gcc` benchmark.

In this example, `G.pagesize` is known to be a power of two. On line 1, `-G.pagesize` is calculated, which will contain `1` values in the most-significant bits and `0` values in the least-significant bits, thus making it a bitmask for `and` that preserves the high bits and only zeroes-out the low bits of the operand. This `and` instruction with such a bitmask *should* propagate the pointer color of `allocation` to the `page` variable, as it will be dereferenced by the program as a pointer to the same chunk.

However, on line 3, `G.pagesize -1` is calculated, which produces a result with `0` values in the most-significant bits and `1` values in the least-significant bits. This time, the bitmask for `and` zeroes-out the high bits and keeps only the low bits, with an integer result that should *not* constitute a pointer. In other words, the correct propagation rule for `and` is *data-dependent* on the value of the operand (bitmask).

It should be noted that in all such examples we encountered, the proper propagation policy could be determined by the expected type of the result: `page` is a pointer and `tail_slop` an integer, which means a tag-aware compiler could supply the required differentiation. As a result, we find that the heap policies are quite compatible with most software, but may occasionally require either (1) additional compiler assistance, or (2) some manual propagation assignments to support programs that manipulate pointers in this way. Our framework takes the later approach: `gcc` has ten such instructions that are manually tagged to be pointer preserving and `soplex` has three, whereas all others receive a default classification of not-preserving.

Sub-word Object Manipulation: Another type of issue that arises while doing memory protection with tags is *sub-word* data manipulation. In the PUMP architecture, tags are associated with each *word* of data, *i.e.,* a 4-byte pointer is associated with a 4-byte tag on that pointer. Heap allocations are always aligned at the word boundary and their size is always a multiple of the machine's word size, which means that heap objects themselves lend well to being tagged. However, in C it is fairly common for programmers to cast data objects to the `char *` type and operate on them as bytes, which causes `gcc` to emit machine instructions such as `stb` that operate on single bytes. To accommodate for cases in which *pointers themselves* may be copied in sub-word increments, we loosen the color propagation rules on pointers to include sub-word size transfers. A consequence of this distinction is that it may be possible for a word to become tainted with a pointer color while itself not constituting a full pointer. We believe this is generally acceptable in that either (1) the full pointer value will be copied eventually, in which case the pointer will point to the appropriate memory region with the correct pointer tag, or (2) a value with a pointer tag will *not* point to the appropriate memory chunk, in which case any dereference will be correctly determined to be invalid.

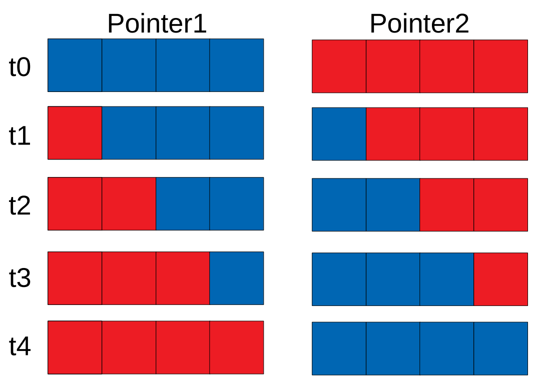
The most egregious code pattern of this variety was observed in the `gcc` benchmark in a Quicksort implementation. The code casts all data elements to byte arrays, and then swaps elements byte-by-byte as the algorithm dictates. In this case *two pointers are swapped in-place*, as illustrated below:

Each time a pair of bytes is swapped, the colors of each pointer word also swap due to the loosened propagation rules. After any *even* number of such swaps, the colors on the pointers will end where they started. After four such swaps, the actual pointer values will have traded places, but the pointer tags will end where they started, thus yielding an incorrect state of the pointer tags. To resolve this issue, we supply a new sorting algorithm that operates on full words at a time. This was the only change to source code required to run the SPEC benchmarks, but does highlight the difficulties in supporting the range of behavior found in C code.

Custom Allocators: Lastly, some applications choose to deploy their own memory allo-

```
void qst(char * base, int n,
         int size, int(*compar)){
  ...
 do {
   ...
   if (j != i) {
     ii = qsz;
     do {
      c = *i;
      *i++ = *j;
      *j++ = c;
     } while (--ii);
   }
   ...
 }
  ...
}
```

(a) A Quicksort implementation that swaps elements byte-by-byte.



(b) A red pointer and a blue pointer are swapped in-place a single byte a time by a sorting algorithm. At timesteps t1 - t3, each word contains a portion of each pointer.

Figure 21: Sub-word data manipulation in a Quicksort implementation.

cators, such as the `gcc` benchmark. To handle these cases for fine-grained protection, it would be necessary to supply the proper tagging hooks on the memory events of the custom allocator. We do not hook `gcc`'s allocator in this way, which means the actual color tracking is more coarse-grained than the program's true allocations; it should be noted that `gcc` uses the default allocator in many cases as well, so the color tracking is actually at a mixed granularity. We note that many of `gcc`'s pointer compatibility issues discussed above (Fig. 20) are related to the custom allocator implementation which requires the shown bit-level data manipulation. Several of the benchmarks also use wrappers to allocators (such as `xmalloc`), and so we treat those as primary allocation sites as well.

4.8. Limitations and Future Work

The heap micropolicies presented in this section are *field insensitive*, meaning that subobjects inside the same heap allocation are not differentiated. As a result, even the Infinite-Color policy does not protect against a buffer overflow internal to a single heap allocation that holds a composite structure or object with multiple fields. With additional compiler support, it would be possible to assign separate colors to subobjects for additional protection at the cost of more tags and rules. Investigating this cost would be interesting future work.

Another bug class that was not addressed by the policies presented in this section are *uninitialized data* errors. With the application of the lazy tagging policy variation, allocations could start at an uninitialized state and the tagging system could be used to track initializations with the first write by the program, much the same as the lazy tagging stack strategy (Sec. 3.5.1). With this approach, it would be possible to detect and halt the access to uninitialized data, which the current policy designs do not detect or protect against. In the stack work we found that programs including the SPEC benchmarks did use uninitialized stack data relatively commonly (such as initializing a subset of the fields of a struct and then copying it), which is likely to be observed for heap-allocated objects as well. As a result, typical C code may not pass this policy; the policy would, however, indicate to programmers exactly where the uninitialized data accesses are occurring such that the code could be further hardened from these kinds of bugs. Additionally, there are some bug classes specific to the heap that are not best addressed with tags, such as memory leaks.

Another attack surface of allocators is their global data structures such as arena metadata that can be corrupted; a final production heap policy would need to take care to protect these objects as well. Lastly, another interesting direction for future work would be a formal treatment of the policies' security properties or the correctness of the policy rules.

4.9. Conclusion

The manual management of dynamic memory in C/C++ is notoriously error-prone and heap-related memory errors have become the most popular attack vector for modern systems exploits [17, 127]. Micropolicies on tagged architectures can be used to protect against a range of common bug classes related to heap memory. We find that a simple One-Color policy is sufficient to defend against a useful range of heap-related errors with a very modest number of unique tags and rules. The Allocation-Site policy is also quite lightweight and protects against an additional bug class compared to the One-Color policy. Both of these policies impose overheads of around 1%. The N-Color can be used to differentiate allocations with increasing numbers of identifiers at higher costs. At the limit, the Infinite Color policy provides complete spatial and temporal memory safety. While this policy can become expensive for some workloads, we show how it can be relaxed in key ways to maintain more protection. We find that the heap policies are quite compatible with most software and required only a handful of manual refactorings to run on the SPEC benchmarks.

CHAPTER 5 : Compartmentalization Policies

5.1. Introduction

Modern software stacks are notoriously vulnerable. Operating systems, device drivers, and countless applications, including most embedded applications, are written in unsafe languages and run in large, monolithic protection domains where any single vulnerability may be sufficient to compromise an entire machine. Privilege separation is a defensive approach in which a system is separated into components, and each is limited to (ideally) just the privileges it requires to operate. In a such a separated system, a vulnerability in one component (*e.g.,* the networking stack) is isolated from other system components (*e.g.,* sensitive process credentials), making the system substantially more robust to attackers, or at least increasing the effort of exploitation in cases where it is still possible.

While the principle of least privilege [107] is a powerful guiding force in secure system design, in practice it is often at odds with system performance. Given the limited hardware resources that have been allocated for security, privilege separation has typically relied on coarse-grained, process-level separation in which the virtual memory system is used to provide isolation. For example, some security-critical and network-facing software such as OpenSSH [19] and Dovecot [46] have been manually decomposed into multiple intercommunicating-but-isolated processes to contain the effects of vulnerabilities, should they be found.

While these select cases are a win for privilege reduction, the prevailing wisdom has been that only coarse-grained privilege separation is feasible in practice given the high cost of virtual memory context switching. Indeed, all modern OSs run on insecure but performant monolithic kernels, with more functionality frequently moving into the highly-privileged kernel to reduce such costs; privilege separated microkernels, in contrast, remain plagued with the perception of high overheads and have seen little adoption. IoT and embedded systems—which we now find ourselves surrounded by in our every-day lives—have fallen even farther behind in security than their general-purpose counterparts. They are also written in memory unsafe languages, typically C, often lack basic modern exploit mitigations [3], and many run directly on bare metal with no separation between any parts of the system at all.

However, there has recently been a surge of interest—both academic and in industry—in architectural and hardware support for new security primitives. For example, ARM recently announced that it will integrate hardware capability support (CHERI [134]) into its chip designs, Oracle has released SPARC processors with coarse-grained memory tagging support (ADI [93]), and NXP has announced it will use Dover's CoreGuard [110], among many others [10]. One interesting and practical use case for these primitives is privilege separation enforcement. In this chapter, we build privilege separation policies for a fine-grained, hardware-accelerated security monitor design (the PIPE architecture [122, 43], Chapter 2.1)).

A flexible, tag-based hardware security monitor, like the PIPE, provides an exciting opportunity to enforce fine-grained, hardware-accelerated privilege separation. At a bird's-eye view, one can imagine using metadata tags on code and data to encode logical protection domains, with rules dictating which memory operations and control-flow transitions are permitted. The PIPE leaves tag semantics to software interpretation, meaning one can express policies ranging from coarse-grained decompositions, such as a simple separation between "trusted" and "untrusted" components, to hundreds or thousands of isolated compartments depending on the privilege reduction and performance characteristics that are desired.

To explore this space, we present SCALPEL (Secure Compartments Automatically Learned and Protected by Execution using Lightweight metadata), a tool that enables the rapid self-learning of low-level privileges and the automatic creation and implementation of compartmentalization security policies for a tagged architecture. At its back-end, SCALPEL contains a policy compiler that decouples logical compartmentalization policies from their underlying concrete enforcement with the PIPE architecture. The back-end takes as input a particular compartmentalization strategy, formulated in terms of C-level constructs and their allowed privileges, and then automatically tags a program image to instantiate the desired policy. To ease policy creation and exploration, the SCALPEL front-end provides a tracing mode, compartment-generation algorithms, and analysis tools, to help an engineer quickly create, compare and then instantiate strategies using the back-end. These tools build on a range of similar recent efforts that treat privilege assessment quantitatively and compartment generation algorithmically [31, 81, 98], allowing SCALPEL's automation to greatly assist in the construction of good policies, a task that would otherwise be costly in engineering time. In cases where human expertise is available for additional fine-tuning, SCALPEL easily integrates human supplied knowledge in its policy exploration; for example, a human can add additional constraints to the algorithms, such as predefining a set of boundaries, adjusting the weight on an object, or specifying that a particular object is security-critical and should not be exposed to additional, unnecessary code.

Additionally, SCALPEL presents two novel techniques for optimizing security policies to a tagged architecture. The first is a policy-construction algorithm that directly targets the rule cache characteristics of an application: the technique is based on packing rules needed for different program phases into sets that can be cached favorably. While we apply this technique on SCALPEL's compartmentalization policies, the core idea could be used to improve the performance of other policies on tagged architectures. Additionally, we show that this same technique can be used to pack an *entire policy* into a fixed-size set such that no runtime rule cache misses will be taken—this makes it possible to achieve real-time guarantees while using a hardware security monitor like the PIPE, which may be of particular value to embedded, real-time devices and applications. Secondly, we design and evaluate a rule prefetching system that exploits the highly-predictable nature of compartmentalization rules; by intelligently prefetching rules before they are needed, we show that the majority of stalled cycles spent waiting for policy evaluation can be avoided.

We evaluate SCALPEL and its optimizations on a typical embedded, IoT environment consisting of a FreeRTOS stack targeting a PIPE-extended RISC-V core. We implement our policies on several applications, including an HTTP web server, the bzip (de)compressor, an H264 video encoder, the GNU Go engine, and the libXML parsing library. Using SCALPEL, we show how to automatically derive compartmentalization strategies for off-the-shelf software that balance privilege reduction with performance, and that hundreds of isolated compartments can be simultaneously enforced with acceptable overheads on a tagged architecture.

To summarize, SCALPEL combines (1) hardware support for fine-grained metadata tagging and policy enforcement with (2) compartmentalization and privilege analysis tools, which together allow a thorough exploration of the level of privilege separation that can be achieved with hardware tagging support. Our primary contributions are:

- A tool that automatically creates and instantiates tag-based compartmentalization policies on real software without manual refactorings.
- Compartment-generation algorithms and analysis tools that quantify the privilege exposure and performance of a wide range of automatically-generated compartmentalization alternatives, providing the security engineer with a variety of privilege-performance design points to explore and evaluate.
- New techniques for using a tagged architecture for the rapid self-learning of privileges on unmodified software.
- A new technique for optimizing a security policy to a tagged architecture by directly targeting its rule cache characteristics.
- A new rule prefetching technique for tagged architectures in which additional rules are identified and prefetched on a cache miss.

5.2. Background and Related Work

5.2.1. The PIPE Architecture

The PIPE (Processor Interlocks for Policy Enforcement) [122] is a software/hardware co-designed processor extension for hardware-accelerated security monitor enforcement that is a close relative of the PUMP (Chapter 2.1). Like the PUMP, the core idea is that word-sized metadata tags are associated with data words in the system, including on register values, words stored in memory, and also on the program counter. However, unlike the PUMP, the PIPE uses a coprocessor design: the core application processor (referred to as the *AP core*) is supported by a dedicated coprocessor, the policy execution (*PEX*) core (depicted below).

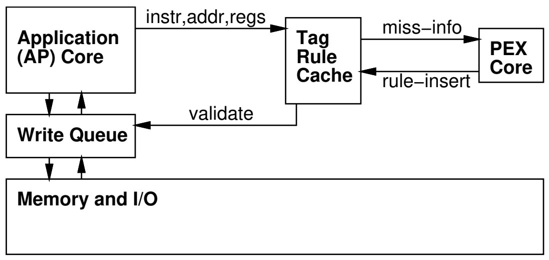


Figure 22: The PIPE Architecture

The AP core can run ahead of rule resolution; a Write Queue holds unvalidated writes until the write instruction and its predecessors can be validated by the rule cache.

5.2.2. The protection-performance tradeoff

While the PIPE can express memory safety policies [43, 106], fine-grained enforcement of all memory accesses can become expensive for some workloads. Compartmentalization policies represent an alternative design point that can very flexibly tune performance-protection tradeoffs through changing compartment sizes and intelligently drawing boundaries for high-performance. With a small number of tags, one can separate out trusted from untrusted components such as ARM TrustZone [10] or OS from the application as in [118, 117], but ultimately we are interested in exploring finer-grained separations. For example, one question we may pose is how tightly can we compartmentalize a software system with tag support while maintaining a certain rule cache hit rate, say 99.9%.

Walking the line between protection and overhead costs is a well-known problem space. Dong et al. [45] observed that different decomposition strategies for web browser components produced wildly different overhead costs, which they manually balanced against domain code size or prior bug rates. Mutable Protection Domains [98] proposes dynamically adjusting separation boundaries in response to overhead with a custom operating system and manually engineered boundaries. Several recent works have proposed more quantitative approaches to privilege separation. Program-Mandering [81] uses optimization techniques to find good separation strategies that balance reducing sensitive information flows with overhead costs, but requires manual identification of sensitive data, and ACES [31] similarly measures the average reduction in write exposure to global variables as a property of compartmentalizations. While these systems begin to automate portions of the compartment formation problem that SCALPEL builds upon, they all still rely on manual input. SCALPEL takes a policy derivation approach with a much stronger emphasis on automation: it uses analysis tools and performance experiments to explore the space of compartmentalizations, then automatically optimizes and lowers them to its hardware backend, a tag-extended RISC-V architecture, for enforcement.

5.2.3. Automatic Privilege Separation

The vast majority of compartmentalization work to date has been manual, demanding a security expert manually identify and refactor the code into separate compartments. This includes the aforementioned projects like OpenSSH [19] and Dovecot [46], and even MicroKernel design [77, 48, 104] using standard OS process isolation, and run-time protection for embedded systems with metadata tags [118, 117]. Academic compartmentalization work has also relied on manual or semi-manual techniques for labeling and partitioning [19, 58, 62, 133].

In contrast, one goal for SCALPEL is *automation*; that is, to apply tag-based privilege-separation defenses to applications without expensive refactorings or manual-tagging; automated efforts relieve the labor-intensive costs of prior manual compartmentalization frameworks. Additionally, automation is important to ease the task of protecting existing software with the PIPE—SCALPEL decouples policy creation from enforcement by automatically lowering an engineer's C-level compartmentalization strategies to the instruction-level enforcement provided by the PIPE.

ACES [31] is an automated compartmentalization tool for embedded systems and shares similarities with SCALPEL. It begins with a program dependence graph (PDG) representation of an application and a security policy (such as *Filename*, or one of several other choices), which indicates a potential set of security boundaries. It then lowers the enforcement of the policy to a target microcontroller device to meet performance and hardware constraints. The microcontroller it targets supports configurable permissions for eight regions of physical memory using a lightweight Memory Protection Unit (MPU); protection domains in the desired policy are merged together until they can be enforced with these eight regions. Unlike ACES, SCALPEL targets a tagged architecture to explore many possible policies, some of which involve hundreds of protection domains, for fine-grained separation, far beyond what can be achieved with the handful of segments supported by conventional MPUs [79].

Towards Automatic Compartmentalization of C Programs on Capability Machines [126] is also similar to SCALPEL. In this work, the compiler translates each compilation unit of an application into a protection domain for enforcement with the CHERI capability machine [134]. This allows finer-grained separation than can be afforded with a handful of memory segments, but provides no flexibility in policy exploration to tune performance and security characteristics like SCALPEL does.

To summarize, *SCALPEL is a complete tool for automatically compartmentalizing unmodified software for hardware acceleration, including automatically self-learning the required* *privileges, systematically exposing the range privilege-performance design-points through algorithmic exploration, and optimizing policies for good rule cache performance.* It complements and extends prior work along four axes: (1) quantitatively scoring the overprivilege in compartmentalization strategies, (2) providing complete automatic generation of compartments without manual input, (3) offering decomposition into much larger numbers of compartments (hundreds), and (4) automatically identifying the privilege-performance tradeoff curves for a wide-range of compartmentalization options.

5.3. Threat Model

We assume a standard [31, 100] but powerful threat model for conventional C-based systems, in which an attacker may exploit bugs in either FreeRTOS or the application to gain read/write primitives on the system, which they may use to hijack control-flow, corrupt data, or leak sensitive information. Attackers supply inputs to the system, which, depending on the application, may include through a network connection or through files to be parsed or encoded. We assume both FreeRTOS and the application are compiled statically into a single program image with no separation before our compartmentalization; as such, a vulnerability in any component of the system may lead to full compromise. We assume that the test cases for the tracing mode are trusted, *i.e.,* an attacker may not supply malicious tests.

The protection supplied by SCALPEL isolates memory read and write instructions to the limited subset of objects dictated by the policy, and also limits program control-flow operations to valid entry points within domains as dictated by the policy. Additionally, SCALPEL is composed with a write-xor-execute [8] tag memory permissions policy that guarantees that instructions (executable code) are not writeable, and any writeable value is not executable. This means attackers cannot inject new executable code into the system. These constraints prevent bugs from reaching the full system state and limit the impacts of attacks to their contained compartments.

5.4. Privilege Reduction and Compartmentalization

The goal of privilege separation is to *reduce the severity* of bugs. To illustrate these benefits in practice, consider the Chrome web browser, perhaps the most well-studied and battle-tested example of modern sandboxing. Since its inception, Chrome has used a *broker process model*, in which the core browser process spawns subprocesses for various tasks and interposes on their communication via IPC channels [14, 57]. Because the renderer contains a substantial amount of historically error-prone code (including a multitude of parsers and decoders for various file formats that are continually exposed to malicious inputs), it was the first subcomponent to be sandboxed. Each site thus runs its own renderer process, which means vulnerabilities inside the renderer are contained to their site-specific process in the case of compromise. To see the impact this design has had on practical exploitation against Chrome, one needs only to analyze modern exploit chains: in a recent writeup by Google Project Zero [22], CVE-2019-5782, a memory error inside the renderer, is paired with an *entirely separate vulnerability*, Issue 1755, inside the core browser process, to achieve a full exploit chain from the renderer to the rest of the browser. In other words, the severity of the renderer vulnerability is reduced to such a significant extent due to the process' reduced privileges, that even gaining arbitrary code execution inside that process was insufficent to compromise the target machine. Instead, that attackers needed to find a separate bug in the browser to escape the renderer's sandbox and achieve their goal, resulting in a substantially more complex attack that burned through two unique vulnerablities. What this shows is that even coarse-grained privilege separation *does* have pragmatic value: the attacker work factor is increased as they (1) require additional vulnerabilities to complete their attacks, and (2) face the increased complexity of reliably chaining together multiple vulnerabilities per successful attack. The further privileges are separated, the further these effects are exaggerated as the number of steps between the compromised components (*e.g.,* the renderer) and the target component (*e.g.,* the high-privileged browser) increases.

Indeed, although this separated design increases Chrome's complexity and thus costs of development and maintenance, the practioners involved believe the benefits are worth their cost. In fact, over the years Chrome engineers have continued to push further privilege separation measures: as of the time of writing, the GPU-acceleration component, auxillary services such as printing, and the networking services, have each been moved into their own isolated processes [57]. In a recent talk, the team has expressed their vision of further isolating the networking and storage services of *each individual site* into their own isolated processes, in which perhaps dozens of processes may be spawned per site to perform as much separation as possible [97]—explicitly, they list performance costs of separation mechanisms as limiting factor for deploying further privilege separation. What this shows is that *practioners value privilege separation, even when it is coarse-grained.* In this light, SCALPEL can be viewed as a tool for providing practioners with privilege separation that they desire by enabling the use of a tagged architecture for fine-grained, hardware-accelerated privilege separation. Notably, however, it approaches the problem from the opposite direction of incrementally sandboxing components: it starts with a least-privilege decomposition and relaxes boundaries as needed to reach a performance target.

5.5. Compartmentalization Tag Policy Formulation

In this section we sketch our general policy model for compartmentalizing software using a tagged architecture. The goal of the compartmentalization policies is to decompose a system into separate logical protection domains, with runtime enforcement applied to each to ensure that memory accesses and control-flow transfers are permitted according to the valid operations granted to that domain. How do we enforce policies like these with a tagged architecture?

The PIPE provides hardware support for imposing a security monitor on the execution of each instruction. Whether or not each instruction is permitted to execute can depend on the tags on the relevant pieces of architectural state (Sec. 5.2.1). For example, we may have a stored private key that should only be accessible to select cryptographic modules. We can mark the private key object with a `private_key` tag and the instructions in the signing function with a `crypto_sign` tag. Then, when the signing function runs and the PIPE sees a load operation with instruction tag `crypto_sign` and data tag `private_key`, it can allow the operation. However, if a video processing function whose instructions are labeled `video_encode` tries to access the private key, the PIPE will see a load operation with instruction tag `video_encode` and data tag `private_key` and disallow the invalid access.

In general, to enable compartmentalization policies, we place a *Domain-ID* label (an integer identifier) on each instruction in executable memory indicating the logical protection domain to which the instruction belongs; this enables rules to conditionally permit operations upon their tagged domain grouping, which serves as the foundation for dividing an application's code into isolated domains. Similarly, we tag each object with an *Object-ID* (also an integer identifier) to demarcate that object as a unique entity onto which privileges can be granted or revoked. For static objects, such as global variables and memory mapped devices, these object identifiers are simply placed onto the tags of the appropriate memory words at load time. Objects that are allocated dynamically (such as from the heap), require us to decide how we want to partition out and grant privileges to those objects. We choose to identify all dynamic objects that are allocated from a particular program point (*e.g.,* a call to `malloc`) as a single object class, which we will refer to simply as an object. For example, all strings allocated from a particular `char * name = malloc(16)` call are the same object from SCALPEL's perspective; this formulation is particularly well-suited to the PIPE because it enables rules in the rule cache to apply to all such dynamic instances. It also means that all dynamic objects allocated from the same allocation site must be treated the same way in terms of their privilege—dynamic objects could be differentiated further (such as by the calling context of the program point) to provide finer separation, but we leave such exploration to future work. As a result of these subject and object identification choices, the number of subjects and objects in a system is fixed at compile time.

Between pairs of subjects and objects (or in the case of a call or return, between two subjects), we would like to grant or deny *operations*. Accordingly, the tag on each instruction in executable memory also includes an *opgroup* field that indicates the operation type of that instruction. We define four *opgroups* and each instruction is tagged with exactly one opgroup: *read*, *write*, *call*, and *return*. For example, in the RISC-V ISA, the `sw`, `sh`, `sb`, etc. instructions would compose the *write* opgroup.

When an instruction is executed, the security monitor determines if the operation is legal based upon (1) the *Domain-ID* of the executing instruction, (2) the type of operation $op \in \{read, write, call, return\}$ being executed, and (3) the *Object-ID* of the accessed word of memory (for loads and stores), or the *Domain-ID* of the target instruction (for calls and returns). As a result, the set of permitted operations can be expressed as a set of triples $(subject, operation, object)$ with all other privileges revoked (default deny). In this way, the security monitor check can be viewed as a simple lookup into a privilege table or access-control matrix whose dimensions are set by the number of *Domain-IDs*, *Object-IDs* and the four operation types. Such a check can be efficiently implemented in the security monitor software as single hash table lookup; once validated in software, a privilege of this form is represented as a single rule that is cached in the PIPE rule cache for hardware-accelerated, single-cycle privilege validation. Additionally, we define a fifth *unprivileged* opgroup, which is placed on instructions that do not represent privileges in our model (*e.g.,* `add`); these instructions are always permitted to execute.

We define a *compartmentalization* as an assignment of each instruction to a *Domain-ID*, an assignment of each object to an *Object-ID*, and a corresponding set of permitted operation triples (*Domain-ID*, op, *Object-ID*). The SCALPEL backend takes a compartmentalization as an input and then automatically lowers it to a tag policy kernel suitable for enforcement with the PIPE. In this way, SCALPEL decouples policy construction from the underlying tag enforcement. The opgroup mapping is the same across all compartmentalizations.

In addition to these privilege checks, the SCALPEL backend also applies three additional defenses to support the enforcement of the compartmentalization. The first is a write-xor-execute policy that prevents the execution of writeable data and the overwrite of executable code so that an attacker cannot inject new executable code into the system. The second is that the words of memory inside individual heap objects that store allocator metadata (*e.g.,* the size of the block) are tagged with a special *ALLOCATOR* tag. The allocator itself is placed in a special *ALLOCATOR* compartment that is granted the sole permission to access such words; as a result, sharing heap objects between domains permits only access to the data fields of those objects and not the inlined allocator metadata. Lastly, SCALPEL is built with the Clang compiler and uses LLVM's [53] static analysis to compute the set of instructions that are valid call and return entry points. These are tagged with special *CALL-TARGET* and *RETURN-TARGET* tags, and we apply additional clauses to the rules to validate that each taken control-flow transfer is both to a permitted domain and to a legal target instruction; this means that when a call or return privilege is granted, it is only granted for valid entry and return points.

An advantage of this policy design is that privilege enforcement is conducted entirely in the tag plane and software does not require refactoring to be protected with SCALPEL. Lastly, we note that there are multiple ways to encode compartmentalization policies on a tagged architecture. For example, the current compartment context could be stored on the program counter tag and updated during domain transitions, rather than from being inferred from the currently executing code. Some of these alternate formulations may work better with different concrete tagging architectures. However, for the PIPE, these formulations are largely equivalent to the above static formulation combined with localizing code into compartments (and making some decisions about object ownership), and we choose the static variant for a slight reduction in policy complexity; the choice is not particularly significant and SCALPEL could produce policies for many such formulations.

5.6. The Tracing Policy

While a motivated developer or security engineer could manually construct a *compartmentalization* for a particular software artifact and provide it to the SCALPEL back-end,

```
vIPNetworkUpCalls:          prvIPTimerReload:             prvIPTimerStart:
   Call prvIPTimerReload       Call prvIPTimerStart          Call vTaskSetTimeOutState
   Call vDNSInitialise         Return vIPNetworkUpCalls      Return prvIPTimerReload
   Return vDHCPProcess         Return prvIPTask              Return prvIPTimerCheck
   Write global_xNetworkUp     Return vIPReloadDHCPTimer     Return prvCheckNetworkTimers
                               Write global_xARPTimer        Read global_xARPTimer
                               Write global_xDHCPTimer       Read global_xDHCPTimer
                               Write global_xTCPTimer        Read global_xTCPTimer
                                                             Write global_xARPTimer
                                                             Write global_xDHCPTimer
                                                             Write global_xTCPTimer
```

Figure 23: The set of privileged operations recorded by the tracing policy for three functions in the FreeRTOS TCP/IP networking stack. SCALPEL uses a tag-based hardware security monitor to automatically self-learn and then enforce fine-grained privileges like these at runtime.

SCALPEL seeks to assist in policy derivation by providing a tracing mode (similar to *e.g.,* AppArmor [83]) as well as a set of analysis tools for understanding the tradeoffs in different decomposition strategies. To this end, we build a compartmentalization *tracing policy*, which collects a lower-bound on the privileges exercised by a program as well as rule cache statistics we use later for policy optimization. While the PIPE architecture was designed for *enforcing* security policies, in this case we repurpose the same mechanism for fine-grained, programmable dynamic analysis. SCALPEL's tracing policy provides several significant practical advantages over other approaches by (1) greatly simplifying tracing by running as a policy replacement on the same hardware and software, (2) directly using the PIPE for hardware-accelerated self-learning of low-level privileges (3), and making it possible to run in real environments and on unmodified software.

For the tracing policy, code and objects should be labeled at the finest granularity at which a security engineer may later want to decompose them into separate domains. On the code side, we find that function-level tracing provides a good balance of performance and precision, and so in this work SCALPEL tags each function with a unique *Domain-ID* during tracing. As a result, our SCALPEL implementation considers functions to be the smallest unit of code that can be assigned to a protection domain. Note that this is a design choice, and the PIPE could collect finer-grained (instruction-level) privileges at a higher cost to the tracing overhead.

On the object side, the tracing policy also assigns an *Object-ID* to each primitive object in the system. For software written in C, this includes a unique *Object-ID* for each global variable, a unique *Object-ID* for each memory-mapped device/peripheral in the memory map (*e.g.,* Ethernet, UART), and a unique *Object-ID* associated with each allocation call site to an allocator as discussed in Sec. 5.5. All data memory words in a running system receive an *Object-ID* from one of these static or dynamic cases.

With these identifiers in place, the tracing policy is then used to record the observed dynamic behavior of the program. The PIPE invokes a software miss handler when it encounters a rule that is not in the rule cache. When configured as the tracing policy, the miss handler simply records the new privileges it encounters—expressed as interactions of *Domain-IDs*, operation types, and *Object-IDs*—as valid privileges that the program should be granted to perform; it then installs a rule so the program can use that privilege repeatedly without invoking the miss handler again. Unlike other policies, the tracing policy never returns a policy violation. We show an example output from the tracing policy in Fig. 23 with identifiers printed as strings from their source program objects. From this tracing information, we form a fined-grained function and object graph, where each function and each object is a node and each privilege is an edge between nodes.

In addition to collecting privileges, the tracing policy also periodically records the rules that were encountered every N_{epoch} instructions, which we set to one million. As we'll see in later sections (Sec. 5.8), this provides the SCALPEL analysis tools with valuable information about rule co-locality which it uses to construct low-overhead policies.

5.7. Privilege Quantification Model

In practice, one likely wants to deploy compartmentalizations that are coarser than the tracing policy granularity (*i.e.,* individual functions and C-level objects) to reduce the number of tags, rules and thus runtime costs associated with policy enforcement. Importantly, the tracing policy leads to a natural privilege quantification model we can use to compare these relaxed decompositions against the finest-grained function/object granularity. We can think of each rule in the tracing policy (*Domain-ID*, op, *Object-ID*) as a *privilege*, to which we can assign a *weight*. The *least privilege* of an application is the sum of the lower-bound privileges that it requires to run; without any of these, the program could not perform its task. For any coarser-grained compartmentalization, we can compute its privilege by counting up the number of fine-grained privileges it permits, which will include additional privileges beyond those in the least-privilege set. This enables us to compute the *overprivilege ratio (OR)*, which we define as the ratio of the privileges allowed by a particular compartmentalization compared to the least-privilege minimum; *i.e.,* an OR of 2.0 means that twice as many privileges are permitted as strictly required. While crude, the OR provides a useful measure of how much privilege decomposition has been achieved, both to help understand where various compartmentalization strategies lie in the privilege-performance space and as an objective function for SCALPEL automatic policy derivation. For our weighting function, we choose to weight each object and function by its size in bytes; this helps account for composite data structures such as a struct that may have multiple fields and should count for additional privilege. Optionally, a developer can manually adjust the weights of functions or objects relative to other components in the system and interactively rerun the algorithms to easily tune the produced compartmentalizations.

5.8. Policy Exploration

To assist in creating and exploring compartment policies, SCALPEL provides three compartment generation algorithms. The first and simplest such approach, presented in Sec. 5.8.1, generates compartment boundaries based upon the static source code structure, such as taking each compilation unit or source code file as a compartment. The second algorithm, presented in Sec. 5.8.2, instead takes an algorithmic optimization approach that uses the tracing data to group together collections of mutually interacting functions. This algorithm is parameterized by a target domain size, allowing it to expose many design points, ranging from hundreds of small compartments to several large compartments. This is an architecture-independent approach that broadly has the property that larger compartments need fewer rules that will compete for space in the rule cache. Lastly, in Sec. 5.8.3 we present a second algorithmic approach that specifically targets producing efficient policies for the PIPE architecture; it targets packing policies into working sets of rules for improved cache characteristics. This algorithm uses both the tracing data and the cache co-locality data (Sec. 5.6) to produce optimized compartmentalization definitions, and is the capstone algorithm proposed in SCALPEL.

5.8.1. Syntactic Compartments

A simple and commonly-used approach [31, 126] for defining compartment boundaries is to mirror the static source code structure into corresponding security domains—we call these the *syntactic domains*. We define the *OS* syntactic domain by placing all of the FreeRTOS code into one compartment (*Domain-ID* 1) and all of the application code into a second compartment (*Domain-ID* 2). This decomposition effectively implements a kernel/userspace separation for an embedded application that does not otherwise have one. Similarly, the *directory* syntactic domains are constructed by placing the code that originates from each directory of source code into a separate domain, *i.e., Domain-ID i* is assigned to the code generated from the *ith* directory of code. Programmers typically decompose large applications into separate, logically-isolated-but-interacting modules, and the directory domains implement these boundaries for such systems. Lastly, the *file* and *function* syntactic domains are constructed by assigning a protection domain to each individual source code file or function that composes the program. Note that each syntactic domain is a strict decomposition from the one before it; for example, compilation units are a sub-decomposition of the OS/application boundary.

For the syntactic compartments, objects are labeled at the fine, individual object granularity (a fresh *Object-ID* for each global variable and heap allocation site); afterwards, all objects with identical permission classes based upon the tracing data are merged together. For example, if two global variables are accessed only by *Domain-ID* 1, then they can be joined into a single *Object-ID* with no loss in privilege; however, if one is shared and one is not, then they must be assigned separate *Object-IDs* to encode their differing security policies.

A second use we find for the syntactic code domains is applying syntactic constraints to other algorithms: for example, we can generate compartments algorithmically but disallow merging code across compilation units to maintain interfaces and preserve the semantic module separation introduced by the programmer. These results are presented in Sec. 5.10.4.

5.8.2. Domain-Size Compartments

While the syntactic domains allow us to quickly translate source code structure into security domains, we are ultimately interested in exploring privilege-performance tradeoffs in a more systematic and data-driven manner than can be provided by the source code itself. We observe that the output of the tracing policy (Sec. 5.6) is a rich trove of information—a complete record of the code and object interactions including their dynamic runtime counts—on top of which we can run optimization algorithms to produce compartments.

Because optimal clustering is known to be NP-Hard [6], we start with the fine-grained function and object graph and employ a straightforward greedy clustering algorithm that groups together sets of mutually-interacting functions into domains while reducing unnecessary overprivilege. The algorithm is parameterized by C_{max}, the maximum number of bytes of code that are permitted per cluster. The algorithm works as follows: upon initialization, each function is placed into a separate compartment C_i with size $C_{i_{size}}$ taken to be the size (in bytes) of that function. At each step of the algorithm, two compartments C_A and C_B are chosen to merge together; the size of the resulting compartment is simply the sum of the sizes of the compartments being merged: $C_{AB_{size}} = C_{A_{size}} + C_{B_{size}}$. To determine which two compartments to merge at each merge step, we compute the ratio of a utility function to that of a cost function for all pairs and select the pair with the highest ratio. The utility function we use is the number of cross-compartment calls and returns found by the tracing policy between those two compartments (*i.e.,* their call affinity). The cost function we use is the increase in privilege (as given by Sec. 5.7) that would result from the merge: that is, we would like to identify mutually interacting functions with high affinity and group them together, while reducing unnecessary overprivilege. The algorithm terminates when no legal merges remain; that is, no candidates A and B maintain $C_{A_{size}} + C_{B_{size}} \leq C_{max}$.

After completion, each cluster C_i is translated into security domain *Domain-ID i* and objects are processed in the same manner as described in 5.8.1. We show results of the Domain-Size algorithm on our HTTP web server application in Fig. 24. First, Fig. 24 (a) shows how the number of compartments trends with the C_{max} parameter on the HTTP web server application. As can be seen, the Domain-Size algorithm coupled with a target size parameter exposes a wide range of compartmentalization strategies, from just a few compartments up to hundreds. Fig. 24 (b) shows the total number of rules that are required to represent the compartmentalization policy at that granularity. With fewer compartments there are fewer unique subject domains, object domains and interaction edges, which means fewer total rules are required for the design. Fig. 24 (c) shows the dynamic runtime rule cache miss rate of the compartmentalization produced from that C_{max} value. Lastly, Fig. 24 (d) shows how the Overprivilege Ratio trends with target domain size.

5.8.3. Working-Set Compartments

The Domain-Size compartment algorithm allows us to explore a wide range of compartmentalization strategies independent of the security architecture, but it is not particularly well-suited to the PIPE. The utility function that drives cluster merge operation is the number of dynamic calls and returns between those clusters. For enforcement mechanisms that impose a cost per domain transition (such as changing capabilities [131] or changing page tables between processes when using virtual memory process isolation), such a utility function would be a reasonable choice, as it does lead to minimizing the number of cross-compartment interactions. Grouping together code and data in this way does reduce the number of tags, rules, and thus cache characteristics of enforcing the compartmentalization on the PIPE, but there is only a broad correlation (Fig. 24(c)).

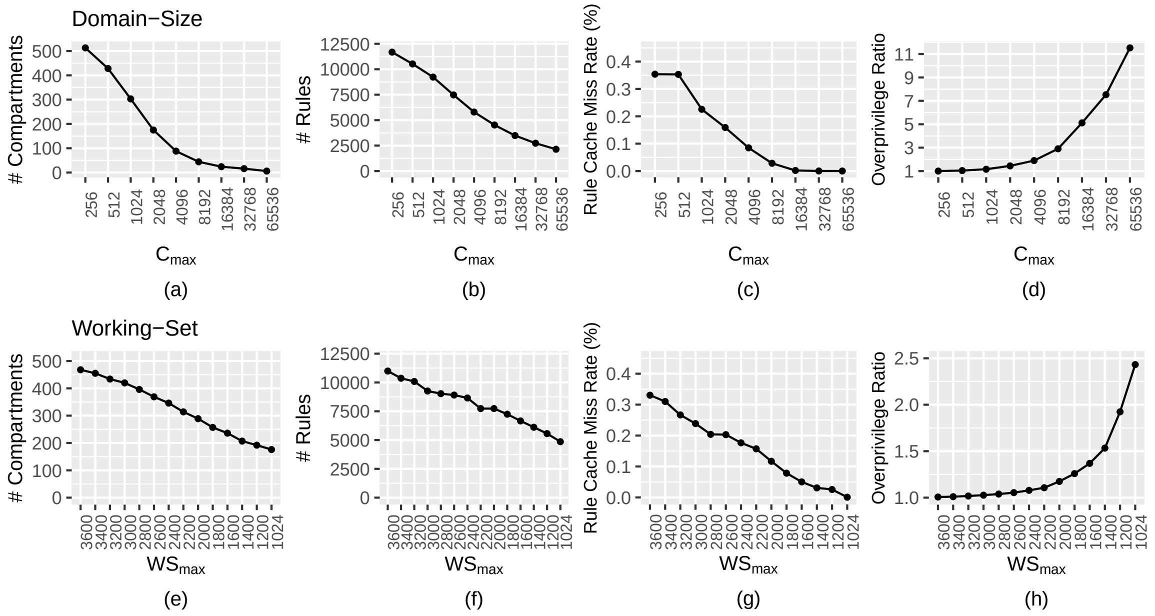


Figure 24: Impact of clustering algorithms for the HTTP web server (Sec. 5.10.1) on 1024-entry rule cache. Top Row: The Domain-Size algorithm. Fig. 24 (a) shows how the number of compartments in final web server design trends with the C_{max} parameter. Fig. 24 (b) shows how many rules are required to represent the compartmentalization at a given C_{max} value. Fig. 24 (c) shows the runtime rule cache miss rate of the compartmentalization produced from that C_{max} value. Lastly, Fig. 24(d) shows the overprivilege ratio of the design. Bottom Row: The Working-Set algorithm. Similar to the top row, Fig. 24 (e), (f), (g), (h) show how the number of compartments, rules, the rule cache miss rate, and the overprivilege ratio all trend with WS_{max} parameter in the Working-Set algorithm. Note how the Working-Set algorithm outperforms the Domain-Size algorithm in terms of the number of compartments, rules, and overprivilege ratio for the same rule cache miss rate.

For the PIPE, there is no cost to change domains, provided the required rules are already cached; instead, what matters is *rule locality*. As a result, to produce performant policies for the PIPE, we instead would like to optimize the runtime rule cache characteristics rather than minimizing the number of domain transitions. To this end, we construct an algorithm based on reducing the set of rules required by each of a program's phases so that each set will fit comfortably into the rule cache for favorable cache characteristics.

How do we identify program phases such that we can consider their cache characteristics? Recall that the tracing policy records the rules that it encounters during each epoch of 1M instructions (Sec. 5.6). We consider the set of rules encountered during each epoch to compose a *working set*. As an intuitive, first-order analysis, if we can keep the rules in each working set below the cache size and the product of those rules and the miss handling time small compared to the epoch length, the overhead for misses in the epoch will be small. As we will see, since not all rules are used with high frequency in an epoch, it isn't strictly necessary to reduce the rules in the epoch below the cache size. While there is prior work on program phase detection [114, 109], SCALPEL takes a simple epoch-based approach that we see is adequate to generate useful working sets; integrating more sophisticated phase detection into SCALPEL would be interesting future work and would only improve the benefits of the PIPE protection policy.

The Working-Set algorithm targets a maximum number of rules allowed per working set, WS_{max}. We construct the Working-Set algorithm in a similar fashion to the Domain-Size algorithm (Sec. 5.8.2) starting with the fine-grained function and object graph, except that we consider clustering of both subjects and objects simultaneously under a unified cost function. The algorithm works as follows: upon initialization, each function is placed into a subject domain S_i and each primitive object is placed into a separate object domain O_i. We then initialize the rules in each working set to those found by the tracing policy during that epoch. At each step of the algorithm, either a pair of subjects or a pair of objects are chosen for merging together. The pair that is chosen is the pair with the highest ratio of a

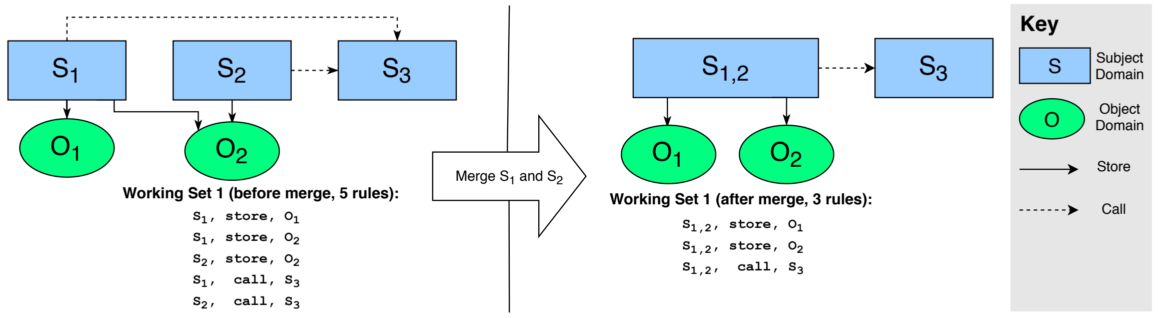


Figure 25: An example of how the rule savings is calculated for merging the S_1 and S_2 domains together. In this example, there are five rules (privilege triples) in Working Set 1 before the merge, and three rules afterwards, for a total of two rules saved. However, S_2 did not have write access to O_1 before the merge, so overprivilege is also introduced by the merge. Assuming all components of the system have a uniform weight of one, then the utility for this merge would be two (two rules saved) and the cost would be one (one additional privilege exposed), for a ratio of $2/1 = 2$. The Working-Set algorithm is driven by the ratio of rules saved in working sets to the increase in privilege, allowing it to enforce as much of the fine-grained access control privileges as possible for a given rule cache miss rate. Note that following the depicted subject merge, merging objects O_1 and O_2 would be chosen next by the algorithm, as it would save an additional rule at no further increase in privilege; in this way, the Working-Set algorithm simultaneously constructs both subject and object domains.

utility function to that of a cost function across all pairs. In contrast to the Domain-Size algorithm, the utility function we use is the sum of the rules that would be saved across all working sets that are currently over the target rule limit WS_{max}. We show an example of how the rule delta calculation is performed in Fig. 25. After performing a merge, the new, smaller set of rules that would be required for each affected working set is calculated, and then the process repeats. The Working-Set algorithm uses the same cost function as the Domain-Size algorithm, *i.e.,* the increase in privilege that would result from combining the two subjects or objects into a single domain. As a result, the Working-Set algorithm attempts to reduce the number of rules required by the program during each of its program phases down to a cache-friendly number while minimizing the overprivilege. The algorithm is run until the number of rules in all the working sets is driven below the target value of WS_{max}.

Like the Domain-Size algorithm, we can vary the value of WS_{max} to produce a range of

Model	**Cost (cycles)**
Cyc_{L1} (64KB, 4-way)	1
Cyc_{L2} (512KB, 8-way)	3
Cyc_{DRAM} (2GB)	100
Cyc_{policy_eval}	300
Cyc_{PIPE} (DMHC [42], 1024)	1

Table 2: Architectural modeling parameters

compartmentalizations at various privilege-performance tradeoffs. If we set our WS_{max} target to match the actual rule cache size, we will pack the policy down to fit comfortably in the cache and produce a highly performant policy; on the other hand, we find that this tight restriction isn't strictly necessary—Fig 24 (g) shows how the rule cache miss rate trends with the target WS_{max} value, achieving an almost linear reduction in miss rate with WS_{max}.

The core advantage of the Working-Set algorithm is that it is able to coarsen a compartmentalization in only the key places where it actually improves the runtime cache characteristics of the application, while maintaining the majority of fine-grained rules that don't actually contribute meaningfully to the rule cache pressure. In Fig 24 (e), (f), (g), (h) we show how the number of compartments, the number of rules, the rule cache miss rate, and the overprivilege ratio trend with the WS_{max} parameter. In contrast to the Domain-Size algorithm, we can see that many more compartments and rules are maintained as the algorithms are driven to smaller and smaller rule cache miss rates, demonstrating the advantages of the Working-Set algorithm in intelligently producing compartmentalization policies at much lower levels of overprivilege. For example, at the WS_{max} of 1024 the Working-Set algorithm achieves the same rule cache miss rate as the Domain-Size algorithm does at a C_{max} of 16,364, but it has more than twice as many total rules and an Overprivilege Ratio that is twice as small, a much more privilege-restricted design for the same overhead. We illustrate the differences between the algorithms more directly in Sec. 5.10 (Evaluation).

5.9. Performance Model

Our SCALPEL evaluation targets a single-core, in-order RISC-V CPU that is extended with the PIPE tag-based hardware security monitor. To match a typical, lightweight embedded processor [78], we assume 64KB L1 data and instruction caches and a unified 512KB L2 cache.

To this we add a 1,024 entry DMHC [42] PIPE rule cache. The application is a single, statically-linked program image that includes both the FreeRTOS operating system as well as the program code. The image is run on a modified QEMU that simulates data and rule caches inline with the program execution to collect event statistics. SCALPEL is built ontop of an open-source PIPE framework that includes tooling for creating and running tag policies [74]. The architectural modeling parameters we use are given in Tab. 2. We use the following model for baseline execution time:

$$T_{baseline} = N_{inst} + N_{L1I_{miss}} \times Cyc_{L2} + N_{L1D_{miss}} \times Cyc_{L2} + N_{L2_{miss}} \times Cyc_{DRAM}$$

Beyond the baseline, SCALPEL policies add overhead time to process misses:

$$T_{SCALPEL} = T_{baseline} + N_{PIPE_{miss}} \times Cyc_{policy\_eval}$$

We take Cyc_{policy_eval} to be 300 cycles based on calibration measurements from our hash lookup implementation.

Lastly, we calculate overhead as:

$$Overhead = \frac{T_{SCALPEL} - T_{baseline}}{T_{baseline}} \times 100\% \tag{5.1}$$

5.10. Evaluation

In this section we present the results of our SCALPEL evaluation. Sec. 5.10.1 details the applications we use to conduct our experiments. Sec. 5.10.2 shows statistics about the applications and the results of the tracing policy. Sec. 5.10.3 shows the privilege-performance results of SCALPEL's Domain-Size and Working-Set algorithms. Sec. 5.10.4 shows the Syntactic Domains and the results of applying the syntactic constraints to the Working-Set algorithm. Lastly, Sec. 5.10.5 shows how SCALPEL's Working-Set rule clustering technique can be used to pack entire policies for real-time systems.

5.10.1. Applications

Web Server: One application we use to demonstrate SCALPEL is an HTTP web server built around the `FreeRTOS + FAT + TCP` demo application [112]. Web servers are common portals for interacting with embedded/IoT devices, such as viewing baby monitors or configuring routers. Our final system includes a TCP networking stack, a FAT file system, an HTTP web server implementation, and a set of CGI programs that compose a simple hospital management system. The management system allows users to login as patients or doctors, view their dashboard, update their user information, and perform various operations such as searches, prescribing medications, and checking prescription statuses. All parts of the system are written in C and are compiled together into a single program image. To drive the web server in our experiments, we use `curl` [34] to generate web requests. The driver program logs in as a privileged or unprivileged user, performs random actions available from the system as described above, and then logs out. For the tracing policy, we run the web server for 500 web requests with a 0.25s delay between requests, which we observe is sufficient to cover the web server's behavior. For performance evaluation, we run 5 trials of 100 requests each and take the average.

libXML Parsing Library: Additionally, we port the libXML2 [102] parsing library to our FreeRTOS stack. To drive the library, we construct a simple wrapper around the `xmlTextReader` SAX interface which parses plain XML files into internal XML data structures. For our evaluation experiments, we run it on the MONDIAL XML database [52], which contains aggregated demographic and geographic information. It is 1MB in size and contains 22k elements and 47k attributes. Parsing structured data is common in many applications and is also known to be error-prone and a common source of security vulnerabilities: libXML2 has had 65 CVEs including 36 memory errors between 2003 and 2018 [40]. Our libXML2 is based on version 2.6.30. Timing-dependent context switches causes nondeterministic behavior; we run the workload 5 times and take the average.

H264 Video Encoder, bzip2, GNU Go: Additionally, we port three applications from the SPEC benchmarks that have minimal POSIX dependencies (*e.g.,* processes or filesystems) to our FreeRTOS stack. Porting the benchmarks involved translating the `main` function to a FreeRTOS task, removing their reliance on configuration files by hardcoding their execution settings, and integrating them with the FreeRTOS dynamic memory allocator. The H264 Video Encoder is based on `464.h264ref`, the bzip2 compression workload is based on `401.bzip2`, and the GNU Go implementation is based on `445.gobmk`. Video encoders are typical for any systems with cameras (baby monitors, smart doorbells), compression and decompression are common for data transmission, and search implementations may be found in simple games or navigation devices. We run the H264 encoder on the reference `SSS.yuv`, a video with 171 frames with a resolution of 512x320 pixels. We run `bzip2` on the reference HTML input and the reference blocking factors. We run GNU Go in benchmarking mode, where it plays both black and white, on an 11x11 board with 4 random initial stones. Timing-dependent context switches causes nondeterministic behavior; we run each workload 5 times and take the average.

5.10.2. Application Statistics, The Tracing Policy, and Rule Cache Miss Rates

In Tab. 3 we show the application statistics of our workloads and the results of the tracing policy. First, to give a broad sense for the application sizes, we show the total lines of code; this column includes only the application, on top of which there is an additional 12k lines of

Application	Lines of Code	Live Functions	Live Objects	Total Rules	Monolithic OR
bzip2	8k	128	109	2,880	39
Web Server	49k	1,231	218	12,025	96
H264	53k	363	692	19,641	244
GNU Go	198k	3,288	10,532	30,077	187
libXML	290k	260	384	10,221	538

Table 3: Application statistics and results of the tracing policy

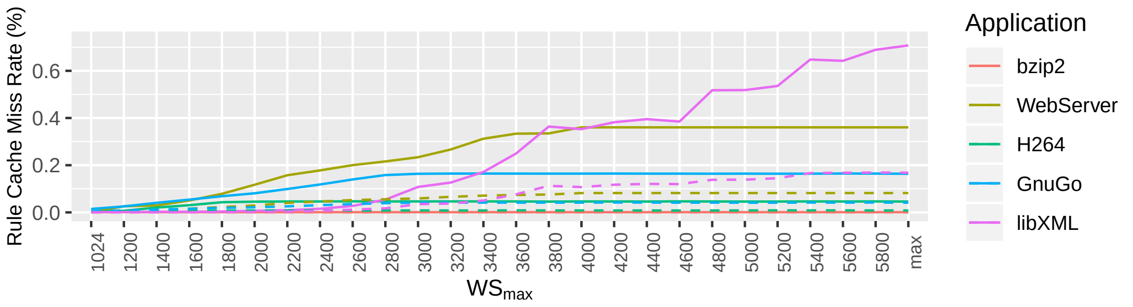


Figure 26: The impact of the WS_{max} parameter on the rule cache miss rate for a 1024-entry rule cache. The `max` value corresponds to the tracing-level granularity (Tab. 3) and the solid lines show how the rule cache miss rate trends with WS_{max}. As can be seen, the SCALPEL algorithms allow a designer to generate compartmentalization designs that target any desired rule cache miss rate. The dashed lines show the even lower rule cache miss rate that is achieved by prefetching rules, which we describe in Sec. 5.11.

core FreeRTOS code. Next, we show the total number of live functions and objects logged by the tracing policy during the program's execution. These subjects and objects compose the fine-grained privileges that SCALPEL enforces. In the Total Rules column we show the total number of unique rules generated during the entire execution of the program under the tracing policy granularity (Sec. 5.6). While this number indicates the complexity of the program's data and control graph, it is not necessarily predictive of the cache miss rate, which depends on the dynamic rule locality. We show the rule cache hit in Fig. 26: the rightmost point (max) corresponds to the miss rate at the tracing policy granularity. As can be seen, the web server has fewer rules than libXML, but also has a lower cache hit rate due to the larger amount of logic that runs at its steady-state web serving phases (such as receiving network requests, parsing them, running CGI programs and sending output). In contrast, `H264` has more total rules, but exhibits more locality as it spends long program phases on a small subset of code and data (*e.g.,* doing motion estimation), a much more rule-cache friendly workload. Very simple workloads, such as `bzip2`, require only a couple thousand rules rules and have effectively no rule cache misses even at the tracing-level granularity.

5.10.3. Privilege-performance Tradeoffs

A key question we would like to answer is how we can trade-off privilege for performance on a per-application basis using the range of SCALPEL compartment generation algorithms (Sec. 5.8). First, Fig. 26 shows how the rule cache miss rate trends with the WS_{max} parameter to the Working-Set algorithm: as can be seen, it allows a designer to target any desired rule cache miss rate for an application.

To explore these compartmentalization options, in Fig. 27 we show the privilege-performance curves (where each compartmentalization design is scored by its overhead and Overprivilege Ratio) generated from the both the Domain-Size algorithm and the Working-Set algorithms. The top-left point in the Domain-Size algorithm corresponds to the tracing-level granularity; this point enforces the full, fine-grained access control matrix, but also imposes large overheads; for example, on the web server application, the cost of enforcement is >100%. The other points in this line correspond to larger values of the C_{max} parameter, which produces fewer, larger compartments for more favorable runtime overheads; however, as can be seen, these coarser compartmentalizations also introduce additional overprivilege. The Working-Set lines in the plots correspond to the range of compartmentalizations produced from the Working-Set algorithm and its WS_{max} parameter. The top-left point corresponds to the maximum value of WS_{max} where no clustering is performed, and the bottom-right point corresponds to packing the rules in each working set to the rule cache size (1,024), producing designs that have very favorable performance characteristics but more overprivilege.

Note that in both cases the curves have a very steep downward slope, meaning large improvements in runtime performance can be attained with little increases in privilege; the curves eventually flatten out, at which point additional decreases in overhead come at the expense of larger amounts of overprivilege. Note that the Working-Set compartments strictly dominate the Domain-Size compartments, producing more privilege reduction at lower costs than the Domain-Size counterparts. As can be seen, SCALPEL allows designers to easily explore the tradeoffs in compartmentalization tag policies. These runs represent the default, fully-automatic toolflow. A designer can then easily inspect the produced compartmentalization files, tune the privilege weights, and rerun the tools interactively as time and expertise allow.

5.10.4. Syntactic Compartments and Syntactic Constraints

In Fig. 28a we show the syntactic compartments (Sec. 5.8.1) on the HTTP web server application. Unlike the Domain-Size and Working-Set algorithms, which are parameterized to produce a wide range of compartmentalization options, the syntactic compartments only provide a handful of decomposition choices. And, as can be seen, none of the options are competitive compared to the Domain-Size of Working-Set decompositions, which suggests that it is indeed useful to approach the compartmentalization problem with more sophisti-

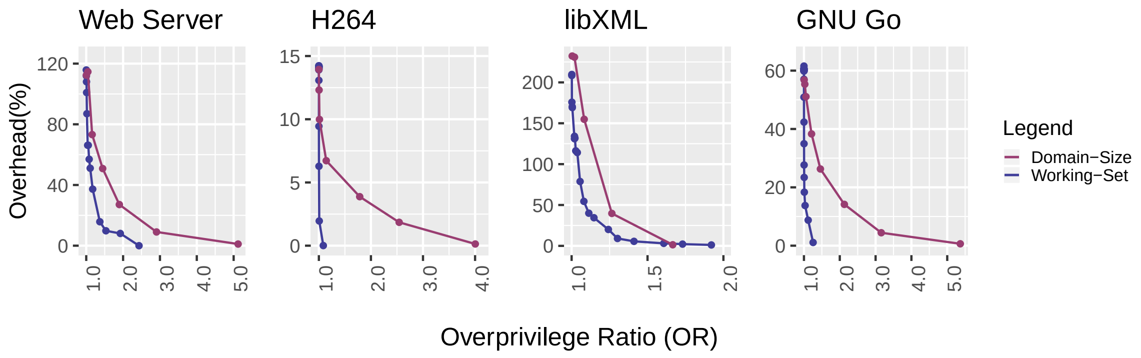
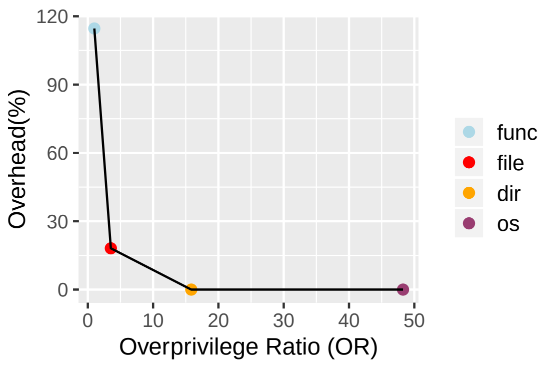


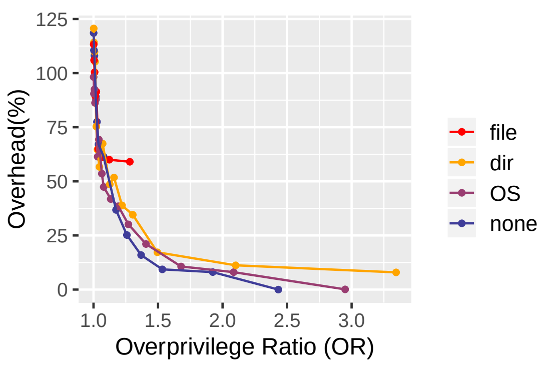
Figure 27: The range of compartmentalizations produced from SCALPEL's algorithms on a 1024-entry rule cache. Each point corresponds to a single specific concrete compartmentalization that is run for performance evaluation and characterized by its runtime overhead (Y axis) and aggregate Overprivilege Ratio (X axis). The Domain-Size line shows compartments that are generated from the various values of C_{max} to the Domain-Size algorithm, and the Working-Set line shows compartments that are generated from the various values of WS_{max} to the Working-Set algorithm. As can be seen, SCALPEL allows a security engineer to rapidly create and evaluate many compartmentalization strategies to explore design tradeoffs without the excessive labor required for manual separations. Note that the Working-Set algorithm dominates the Domain-Size algorithm, with a particularly strong advantage at the low-end of the overhead spectrum.

cated techniques.

However, it is also true that software engineers often decompose their own projects into modules, and those modules boundaries bear semantic information about code interfaces and relationships. For example, the web server application has the core FreeRTOS code in one directory, the TCP/IP networking stack in another directory, the web server application (CGI app) in another directory, and the FAT filesystem implementation in another separate directory. When the algorithmic compartment generation algorithms (Secs. 5.8.2, 5.8.3) optimize for privilege-performance design points, they have the full freedom to reconstruct boundaries in whatever way they find produces better privilege-performance tradeoffs. However, if we would like to preserve the original syntactic boundaries during the algorithmic optimization process, we can add additional constraints, such as a *syntactic constraint*, which limits the set of legal merges allowed by the algorithms. For example, under the *file* syntactic constraint, two global variables can only be merged if they originate from the same source file. This allows SCALPEL to optimize privilege separation internal to a mod-



(a) The privilege-performance points generated by the Syntactic Domains (Sec. 5.8.1). Unlike the Domain-Size and Working-Set algorithms, the Syntactic Domains only provide a handful of choices, none of which are competitive with the alternate algorithms.



(b) Impact of syntactic constraints. The algorithmic compartment algorithms (Sec. 5.8.2, 5.8.3) can optionally take an additional constraint, such as a syntactic boundary, that must be respected while creating the compartmentalization. In this case, we show the application of three syntactic domains to the Working-Set algorithm.

Figure 28: Privilege-Performance impact of Syntactic Domains and Constraints on the HTTP web server running on a 1024-entry rule cache.

ule while respecting the interfaces to that module. We note that a compartmentalization that is a strict sub-decomposition of another compartmentalization is never less secure.

In Fig. 28b we show the application of the syntactic domains as constraints to the Working-Set algorithm. The OS restriction adds little additional overhead to the produced design points but guarantees a cleaner separation of the OS and application than may be found by the algorithms naturally. On the other hand, the *file* constraint is very restrictive, reducing the number of moves available to the algorithms to such a large extent that many of the WS_{max} targets fail to generate.[1] These examples illustrate the benefits of the rapid exploration enabled by SCALPEL, and we note that a manually-constructed constraint can be a very convenient method for interacting with SCALPEL's automation.

[1]Note that the Working-Set algorithm can only perform merge operations between code or object domains that are found to co-reside in the same epoch, which means that they may not reach the same minimum overhead as the corresponding syntactic domain.

5.10.5. Packing Policies for Real-Time Systems

The ideas presented in the Working-Set algorithm (Sec. 5.8.3) can be used to pack an entire security policy (*i.e.,* the complete set of rules that compose the policy) into a single, fixed-size set of rules. For this construction, we simply take the union of all rules required to represent the policy and present it to the Working-Set algorithm as a single working set—the entire policy will then be packed down to a number of rules equal to WS_{max}. Importantly, this means that the policy can be loaded in a constant amount of time, and assuming the WS_{max} matches the rule cache size, then no additional runtime rule resolutions will occur, giving the system predictable runtime characteristics suitable for real-time systems. We show the results of this technique in Tab. 4 when applied to a range of rule targets. The overprivilege points generated from this technique could be used to decide on a particular rule cache size for a specific embedded application to achieve target protection and performance characterstics. Note that the working-set cached case achieves lower OR at a the same 1024-entry rule capacity since it only needs to load one working-set at a time. It will take a larger rule cache to achieve comparably low OR. However, it is worth noting that, if the rule memory does not need to act as a cache, it can be constructed more cheaply than a dynamically managed cache, meaning the actual area cost is lower than the ratio of rules, and might even favor the fixed-size rule memory. Furthermore, if one is targeting a particular application, the tag bits can also be reduced to match the final compartment and object count (e.g., can be 8b instead of a full word width), which will further decrease the per rule area cost.

5.11. Prefetching

Finally, we consider one last performance optimization to reduce the overhead costs of SCALPEL's policies: rule prefetching. During the normal operation of the PIPE, rules are evaluated and installed into the rule cache only when an application misses on that rule. When such a miss occurs, the PEX core awakens, evaluates the rule, and finally installs it into the rule cache. Much like prefetching instructions or data blocks for processor

	Real-Time Rule Target				
Application	512	1024	2048	4096	8192
bzip2	2.82	1.34	1.01	1.00	1.000
Web Server	8.92	5.76	2.71	1.35	1.005
H264	11.9	2.81	1.46	1.05	1.002
libXML	12.3	7.46	2.61	1.18	1.000
Gnu Go	29.4	12.7	3.47	1.43	1.033

Table 4: The Overprivilege Ratio (OR) of the applications when they are packed for real-time performance to the given total rule count. When packed in this way, they can be (1) loaded in constant time and (2) experience no additional runtime rule resolutions, making them suitable for real-time systems.

caches [87], there is an opportunity for the PEX core to preemptively predict and install rules into the cache. Such a technique can greatly reduce the number of runtime misses that occur, provided that the required rules can reliably be predicted and prefetched before they are needed. In this section we explore the design and results of a rule prefetching system.

5.11.1. The Rule-Successor Graph

The core data structure of our prefetching design is the *Rule-Successor Graph.* The Rule-Successor Graph is a directed, weighted graph that records the immediate temporal relationships of rule evaluations. A rule is represented as a node in the graph, and a weighted edge between two nodes indicates the relative frequency of the miss handler evaluating the source rule followed immediately by evaluating the destination rule.

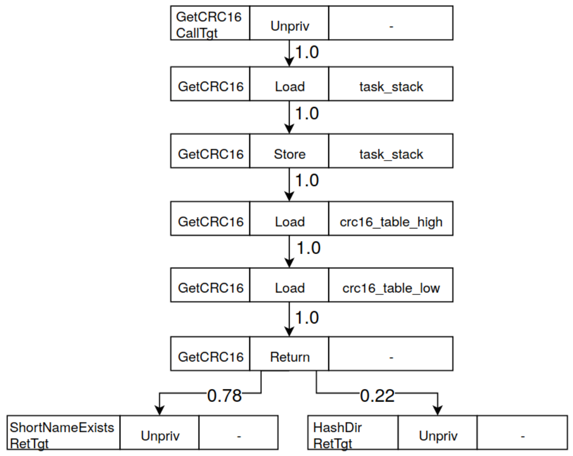
Fig. 29(a) shows an example function from the FreeRTOS FAT filesystem, and Fig. 29(b) shows the Rule-Successor Graph for its function-entry rule. When this function is called, it issues loads and stores to the task's stack, and then it issues loads to the `crc16_table_high` and `crc16_table_low` global variables in exactly that order; this deterministic sequence is learned and encoded in the Rule-Successor Graph. Many kinds of rule relationships are highly predictable, such as rules that are required for sequential instructions in the same basic block.

On the other hand, data or control-flow dependent program behavior can produce less

```
short GetCRC16(char *pbyData, int stLength){
  short bTableValue;
  short wCRC = 0;

  while(stLength-- != 0 ){
    bTableValue = ((wCRC & 0x00FF) ^ *pbyData++);
    wCRC = (((crc16_table_high[bTableValue]) << 8) +
            (crc16_table_low[bTableValue] ^ ((wCRC >> 8)
                                             & 0x00FF)));
  }
  return wCRC;
}
```

(a) The `GetCRC16` function in the FreeRTOS FAT Filesystem.



GetCRC16 CallTgt | Unpriv | -
1.0
GetCRC16 | Load | task_stack
1.0
GetCRC16 | Store | task_stack
1.0
GetCRC16 | Load | crc16_table_high
1.0
GetCRC16 | Load | crc16_table_low
1.0
GetCRC16 | Return | -
0.78
0.22
ShortNameExists RetTgt | Unpriv | -
HashDir RetTgt | Unpriv | -

(b) The Rule-Successor Graph for the `GetCRC16` function. Each rule (privilege triple) is shown as a rectangle with three fields corresponding to the subject, operation and object tags.

Figure 29: The Rule-Successor Graph, a data structure used by SCALPEL's prefetching system

predictable rule sequences—for example, a return instruction can have many, low-weighted rule successors if that function is called from many locations within a program. In this example, `GetCRC16` has two callers and may return to either, although one is much more common than the other; similarly, `GetCRC16` also accepts a data pointer `pbyData` which could produce data-dependent rule sequences depending on the object it points to, although in this program it always points to the task's stack, which does not require another rule. Lastly, if `stLength` were 0, then the program would take an alternate control-flow path and several of the rules would be skipped. Like other architectural optimizations such as caches and branch predictors [51], optimistic prefetching accelerates common-case behavior, but may have a negative impact on performance when the prediction is wrong.

A program's Rule-Successor Graph can be generated from the miss handler software with no other changes to the system. To do so, the miss handler software simply maintains an account of the last rule that it handled. When a new miss occurs, the miss handler software updates the Rule-Successor Graph by updating the weight from the last rule to the current rule (and adding any missing nodes). Finally, the record of which rule was the last rule is updated to the current rule, and the process continues.

5.11.2. Generating Prefetching Policies

A *prefetching policy* is a mapping from each individual rule (called the *source rule*) to a list of rules (the *prefetch rules*) that are to be prefetched by the miss handler when a miss occurs on that source rule. Prefetching policies are generated offline using a program's Rule-Successor Graph; the goal is to determine which rules (if any) should be prefetched from each source rule on future runs of that program.

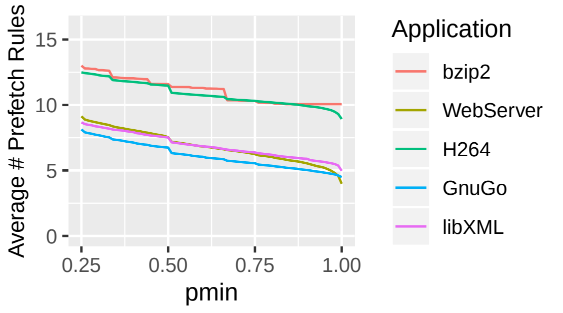
To find good candidate prefetch rules for each source rule, we deploy a Breadth-First Search algorithm on the Rule-Successor Graph to discover high likelihood, subsequent rules. Each such search begins on a source rule with an initial probability $p = 1.0$. When a new node (rule) is explored by the search algorithm, its relative probability is calculated by multiplying the current probability by the weight of the edge taken. When a new, unexplored rule is discovered, it is added to a table of explored nodes, and its depth and probability are recorded with it. If a rule is already in the table when it is explored from a different path, then the running probability is added to the value in the table to reflect the sum of the probabilities of the various paths on which the rule may be found.

The algorithm terminates searching on any path in which the probability falls below a minimum threshold value. We set this value to 0.1%, which we observe sufficiently captures the important rules across our benchmarks. After search is complete, the table of explored nodes is populated and ready to be used for deriving prefetching policies. To test the impact of various degrees of prefetching, we add a pruning pass in which any rules below a target probability p_{min} are discarded from the table. For example, if p_{min} is set to the maximum of 1.0, then rules are only included in the prefetching set if they are always observed to occur in the Rule-Successor Graph following the source rule. On the other hand, Lower p_{min} run a higher risk of both not averting future misses, and in the worst-case may pollute the rule cache by evicting a potentially more-important rule. In Fig. 29(b), the bottom left rule has a probability of 0.78. If no rules remain after pruning, then no prefetch rules are found. Otherwise, the remaining rules are sorted to compose the final list of prefetch rules. They are sorted by depth (smallest first), then within the same depth by probability (highest first) to order the rules in a sequence most likely to be useful. We vary the values of p_{min} from 1.0 to 0.25 to explore the impact of various levels of prefetching on final performance.

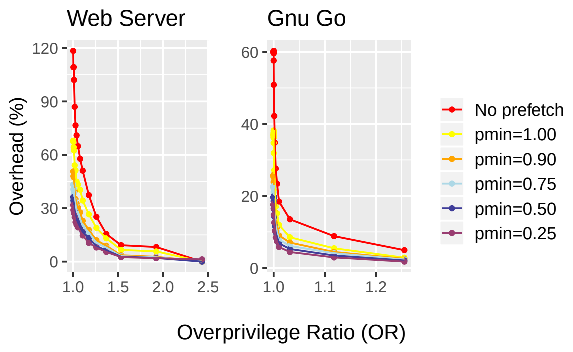
5.11.3. Prefetching Cost Model

When the PIPE misses on a rule, it traps and alerts the PEX core for rule evaluation. In SCALPEL, a rule evaluation is a hash table lookup that checks the current operation against a privilege table (Sec. 5.5). When prefetching is enabled, we choose to store the prefetch rules in the privilege hash table along with the source rule to which they belong. When a miss occurs, the miss handler performs the initial privilege lookup on the source rule and installs it into the PIPE cache, allowing the AP core to resume processing. Afterwards, the PEX core continues to asynchronously load and install the rest of the prefetch rules in that hash table entry. Assuming a cache line size of 64B and a rule size of 28B (five 4B input tags and two 4B output tags), then two rules fit in a single cache line. As such, the first prefetch rule can be prepared for insertion immediately following the resolution of the source rule. We assume a 10 cycle install time into the PIPE cache for each rule installation. For each subsequent cache line (which can hold up to two rules), we add an additional cost of 20 cycles for a DRAM CAS operation, in addition to the 10 cycle insertion time for each rule. We set the maximum number of prefetch rules to seven so that all eight rules (including the source rule) may fit onto a single same DRAM page, assuming a 2048b page size. Our data shows that this number is sufficient to capture a majority of the benefits of prefetching on our workloads.

We begin by looking at the structure of the Rule-Successor Graph (Fig. 30a) from our various applications to get a sense for the number and likelihood of prefetch rules per source rule that might be prefetched; the more high-likelihood rules there are, the more benefits we expect to see from prefetching. In this graph, the X axis shows the p_{min} cutoff



(a) The structure of the Rule-Successor Graph.



(b) The impacts of prefetching on the privilege-performance curves generated from the Working-Set algorithm. The red line shows the baseline case (no prefetching), and the other lines correspond to the prefetching policies generated from the shown p_{min} values.

Figure 30: The results of the SCALPEL's rule prefetching system.

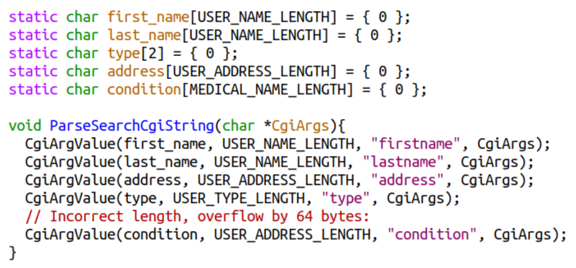
probability in the range of [0.25,1], and the Y axis shows the average number of rules per source rule that have at least the given cutoff probability. The data shows that there around five rules per source rule that can be prefetched even at the maximum p_{min} value of 1.0 (*i.e.*, they always follow the source rule during tracing) in our benchmarks; `H264` and `bzip2` are more predictabile than the other three benchmarks with values closer to ten. At lower values of p_{min}, more rules make the cutoff, although the slope is low (less than one rule on average per 10% decrease in likelihood) meaning there are significantly diminishing returns on prefetching larger numbers of rules.

Next, to see the the results of prefetching on rule cache miss rate, we show the prefetching cases as dashed lines in Fig. 26. To see the final impact on program overhead, Fig. 30b shows the resulting privilege-performance curves generated from the various prefetching policies. The red line shows the baseline (no prefetching) case, and the other lines show the prefetching policies generated from the various values of p_{min}. All of the prefetching cases strictly dominate the baseline case on privilege and performance; the yellow line ($p_{min} = 1.0$) captures a majority of the benefits, but each additional relaxation to p_{min} continues to lower enforcement costs for the same privilege reduction level in diminishing amounts. On the high end of the overhead spectrum (*i.e.,* the tracing-level granularity), the prefetching system reduces the overhead from an average of 105% to only 27%, a 3.9X reduction in overhead costs. This shows the predictable nature of the PIPE rules in a compartmentalization policy and the large benefit of prefetching rules to reduce costs. On the lower end of the overhead spectrum, the benefits of prefetching are less pronounced but do enable the system to achieve even finer privilege separation at the same costs: at overhead of 10%, the prefetching cases allow for 20% more rules and an OR that is 12% smaller.

5.12. Security, Overprivilege and Work-factor

Vulnerabilities such as memory safety errors permit a program to perform behaviors that violate its original language-level abstractions, *e.g.,* they allow a program to perform an access to a memory location that is either outside the object from which the pointer is derived or has since been freed and is therefore temporally expired. An exploit developer has the task of using such a vulnerability to corrupt the state of the machine and to redirect the operations of the new, emergent program such as to reach new states that violate underlying security boundaries or assumptions, such as changing an authorization level, leaking private data, or performing system calls with attacker-controlled inputs. In practice, bugs come in a wide range of expressive power, and even memory corruption vulnerabilities are often constrained in one or more dimensions, *e.g.,* a typical contiguous overflow error may only write past the end of an existing buffer [105], or an off-by-one-error allows an attacker to write a pointer value past the end of an array but gives the attacker no control of the written data [17]. Modern exploits are typically built from *exploit chains* in which a series of bugs are assembled together to achieve arbitrary code execution [17], and complex exploits take many man-months of effort to engineer [16] even in the monolithic environments in which they run.

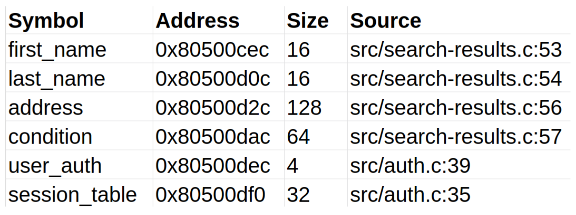
The privilege separation defenses imposed by SCALPEL limit the reachability of memory accesses and control-flow instructions to a small subset of the full machine's state. These



```
static char first_name[USER_NAME_LENGTH] = { 0 };
static char last_name[USER_NAME_LENGTH] = { 0 };
static char type[2] = { 0 };
static char address[USER_ADDRESS_LENGTH] = { 0 };
static char condition[MEDICAL_NAME_LENGTH] = { 0 };

void ParseSearchCgiString(char *CgiArgs){
  CgiArgValue(first_name, USER_NAME_LENGTH, "firstname", CgiArgs);
  CgiArgValue(last_name, USER_NAME_LENGTH, "lastname", CgiArgs);
  CgiArgValue(address, USER_ADDRESS_LENGTH, "address", CgiArgs);
  CgiArgValue(type, USER_TYPE_LENGTH, "type", CgiArgs);
  // Incorrect length, overflow by 64 bytes:
  CgiArgValue(condition, USER_ADDRESS_LENGTH, "condition", CgiArgs);
}
```

(a) A buffer overflow vulnerability in the Web Server's search functionality. The vulnerable CGI program can be reached and triggered by any user who browsers to the `search.html` page. This vulnerability allows an attacker to corrupt the `user_auth` and `session_table` variables.



Symbol	Address	Size	Source
first_name	0x80500cec	16	src/search-results.c:53
last_name	0x80500d0c	16	src/search-results.c:54
address	0x80500d2c	128	src/search-results.c:56
condition	0x80500dac	64	src/search-results.c:57
user_auth	0x80500dec	4	src/auth.c:39
session_table	0x80500df0	32	src/auth.c:35

(b) The symbol table showing the addresses and sizes of several of the symbols in the Web Server's data section.

Figure 31

restrictions affect the attacker's calculus in two ways: first, they may lower the impact of bugs sufficiently to disarm them entirely, *i.e.*, rendering them unable to impart meaningful divergence from the original program. Second, they may vastly increase the number of exploitation steps and bugs required to reach a particular target from a given vulnerability: an attacker must now perform repeated confused deputy attacks [59] at each stage to incrementally reach a target; when privilege reduction is high, these available operations become substantially limited, thus driving up attacker costs and defeating attack paths for which no exploit can be generated due to the imposed privilege separation limitations.

We illustrate these ideas with a vulnerability example from the Web Server application in Fig. 31. A buffer overflow in the `search` CGI code is reachable from the server's web interface and can cause the program to write beyond the end of the `condition` buffer onto objects located at higher addresses, which are the `user_auth` and `session_table` variables. Corrupting `user_auth` can allow an unprivileged user to escalate their privileges. However, the fault is entirely contained if `user_auth` is tagged with an *Object-ID* for which `CgiArgValue` does not have write permission, because any out-of-bounds write will incur a policy violation. In Tab. 5 we show the range of compartmentalizations generated from the Working-Set algorithm. Row 1 shows the compartmentalization's Overprivilege Ratio, and row 2 shows whether the `user_auth` overwrite is prevented (which we verify against our policy implementation by triggering the buffer overflow to classify as ✓or X in the table). If that write is not prevented, then an attacker can (1) escalate their privileges, and (2) there is also an possibility to corrupt the subsequent `session_table` as well if *that object* is also writable from `CgiArgValue`. The `session_table` is a structure that contains a hash table root node, which includes a code pointer `session_table->compare`. Like the `user_auth` object, this object is protected if the `CgiArgValue` code does not have permission to write to it. We show this relationship in row 3. If it can be corrupted, it could provide additional footing to compromise the contained compartment, such as through hijacking the program's control flow by overwriting the `session_table->compare` field.

While we have illustrated that these specific vulnerabilities are eliminated at specific higher compartmentalization levels and lower ORs, we expect this trend to generally hold for other vulnerabilities—as OR lowers, at some point each specific vulnerability based on a privilege violation is eliminated. Each vulnerability may, in general, be eliminated at a different OR. Consequently, we expect lower OR to generally correlate well with lower vulnerability. Lastly, in row 4 we show the total number of legal call targets that are permitted by the domain containing `HashTableEqual` (the only function in the program that performs indirect calls using `session_table->compare`) to show the reachability of such a control-flow hijack. What this shows is that even if the code pointer is corrupted, the attacker is limited to only a handful of options to continue their attack, which for many of our domains is around ten or less; furthermore, even *those targets* are all functions related to the hash table operations, which would require further steps still to reach other parts of the system. In other words, both examples show there is a relationship between the overprivilege permitted to each component of a system and the effort expended by exploit developers to weaponize their bugs to reach their targets.

WS_{max}	max	3200	2800	2400	2000	1800	1600	1400	1200	1024	512
Overprivilege Ratio (OR)	1.00	1.02	1.04	1.08	1.17	1.26	1.36	1.53	1.92	2.43	8.42
Protect `user_auth`	✓	✓	✓	✓	✓	✓	✓	✓	X	X	X
Protect `session_table`	✓	✓	✓	✓	✓	✓	✓	✓	✓	X	X
`HashTableEqual` call targets	1	1	1	1	7	7	11	12	17	67	241

Table 5: The relationship between WS_{max}, the Overprivilege Ratio, and the exploitability of a vulnerability in the Web Server. We validate the ✓and X vulnerability assignments by running the Web Server with the policy shown and triggering the buffer overflow vulnerability. Lastly, we show the number of call targets reachable from the `HashTableEqual` function under the given compartmentalization if its code pointer is corrupted.

5.13. Future Work

5.13.1. Privilege Metric Extensions

The Overprivilege Ratio (OR) is presented here in its simplest form, in which all objects and functions are weighted by their size in bytes. This weighting scheme is appropriate under the *a priori* assumption that all privileges are equal, *i.e.,* larger data structures contain more fields and thus should account for more privilege than smaller ones, and also that larger functions tend to have more privileges than smaller ones and should likewise be weighted higher in their privilege. However, in reality, all objects and operation types are not of equal security importance to attackers under a concrete threat model. To better represent object privilege, one should consider the attack paths that they would like to prevent: a defense can only be analyzed under a particular threat model, and so it is from that threat model that one should turn to derive privilege weights. For example, in classic memory safety exploits, code pointers are of high priority to attackers, and thus preserving their integrity is of great security value [73] compared to other data in the system. Data pointers, in turn, typically have a security impact that is dependent on the objects and operation types to which they grant the program access. For example, a data pointer with write permission might be usable for a memory corruption, while a read-only data pointer may only have value in what it can leak to an attacker. We can also consider which particular objects are more important than others: some data objects may be of little significance (such as the bytes composing a large image), whereas other objects (such as control or authorization data) may be of great practical significance. Furthermore, objects may be more or less significant under different operation types: security keys, for example, may be highly sensitive to reads (*i.e.,* secrecy is highly valued), whereas control data may be only impactful to attackers when it can be overwritten (*i.e.,* integrity is highly valued). The privilege model is well-equipped to handle these cases by providing per-object and per-operation type weights to capture the relative importance of these privileges as one adapts their weights to match their threat model.

To better model code privilege, we note that there is a great variance in the number and significance of privileges that are granted to each function (or finer-grained piece of code), and that this property could also be captured with a privilege weight. Code with high privilege, both in terms of what objects it may access and what other code it may call or return to, should be weighted higher than other, less-privileged code. For example, one could weight the privilege of a function call in proportion to *the callee's privileges.* As such, the ability to call a function with access to an important, critical object could be treated (perhaps correctly) as significant as access to the object itself. As one can see, there is a rich space available to capture domain-specific security knowledge in our privilege model to tune our automation in such a way as to better match the threat models, object importance, and security assumptions under which they will operate. In whatever way one deems most appropriate to tune their weights, the algorithms and approach presented here apply just the same. Exploring these extensions to SCALPEL's OR would be interesting future work.

5.13.2. On Correlating OR with Security

In Sec. 5.12 we presented an attack example to illustrate the benefits of privilege reduction and in Sec. 5.13.1 we illustrated some of the ways that the privilege metric can be tuned to match a concrete threat model. Nonetheless, a key missing piece in the quantitative approach to privilege separation is a deeper study on the relationship between the privilege metric, its resulting security impact in practice under a concrete threat model, and the most effective weighting schemes under that threat model to make best use of the available privileges separation resources.

5.13.3. Applying Prefetching to Other Policies

The prefetching technique introduced in Sec. 5.11 was shown to be highly effective at accelerating SCALPEL's compartmentalization policies due to the highly-predictable rule sequences that they tend to produce. It is likely that rule prefetching could be applied successfully to other kinds of micropolicies as well. For example, the stack protection policies (Chapter 3) would could be a good candidate for prefetching: many stack-related operations (storing stack control data, initializing frame tags, accessing data elements, *etc.*) occur each time a function is called which means those rules will likely be highly predictable as well. As a result, prefetching has promise to reduce enforcement costs for policies beyond compartmentalization.

5.14. Comparisons with Related Embedded System Security Work

Hex-Five's MultiZone Security [61] is a state-of-the-art compartmentalization framework for RISC-V. However, it requires a developer to manually decompose the application into separated binaries called "zones", each of which are very coarse grained—the recommended decomposition is one zone for FreeRTOS, one for the networking stack, and one or several for the application. MultiZone Security requires hundreds of cycles to switch contexts, which is negligible when only employed at millisecond intervals, but the overprivilege is very high, as large software components still have no separation; as a result, MultiZone Security achieves a privilege reduction that falls in between the OS and dir syntactic points shown in Fig. 28a. SCALPEL imposes significantly finer grained separation and provides substantially easier policy development and exploration. MINION [68] is another compartmentalization tool for embedded systems. However, it also enforces only very coarse-grained separation by switching between a small number of memory views and provides no method for exploring policies to tune protection and performance characterstics.

ACES [31] is closer to SCALPEL in terms of providing automatic separation for applications, however it targets enforcement using the handful of segments provided by the MPU. ACES has negligible overhead for some applications, but 20-30% overhead is more typical, with some applications requiring over 100% overhead. As a close comparison point, we ran the Domain-Size algorithm with a few modifications to target four code and four object domains; the resulting design for the HTTP web server application has an OR of 28.7 compared to SCALPEL's OR of 1.26 (at a WS_{max} of 1800 for a comparable overhead), which is more than 20$\times$ more separation at that level; including prefetching, that same level of protection can be enforced at an overhead that is 3X lower. As a result, SCALPEL shows that a hardware tag-based security monitor can be used to provide unprecedented levels of privilege separation for embedded systems.

5.15. Runtime Modes and Dynamic Analysis

5.15.1. Runtime Modes

SCALPEL has two primary runtime modes: `alert mode` and `enforcment mode`. In `alert mode`, SCALPEL does not terminate a program if a policy violation is encountered; instead, it produces a detailed log of the privilege violations that have been observed; this mode could provide near real-time data for intrusion detection and forensics in the spirit of Transparent Computing [36]. Alternatively, in `strict mode`, any policy violation produces fail-stop behavior.

5.15.2. Dynamic Analysis Limitations

SCALPEL uses dynamic analysis to capture the observed low-level operations performed by a program. Observing dynamic behavior is important for SCALPEL to capture performance statistics to build performant policies (Sec. 5.8). However, this also means that our captured traces represent a lower bound of the true privileges that might be exercised by a program, which could produce false positives in `enforcement mode`. There are a number of ways to handle this issue, and SCALPEL is agnostic to that choice. In cases where extensive test suites are available or can be constructed, one might use precise SCALPEL;

that is, the traced program behavior serves as a ground truth for well-behaved programs and any violations produce fail-stop behavior; some simpler embedded systems applications may fit into this category. For higher usability on more complex software, SCALPEL could be combined with static analysis techniques for a hybrid policy design. In that case, the policy construction proceeds exactly as described in this paper for capturing important performance effects, but the allowed interactions between *Domain-IDs* and *Object-IDs* would be relaxed to the allowed sets as found by static analysis. The best choice among these options will depend on security requirements, the quality and availability of test suites, and the tolerable failure rate of the protected application. We consider these issues orthogonal to SCALPEL's primary contributions.

5.16. Conclusion

We presented SCALPEL, a tool for producing highly-performant compartmentalization policies for the PIPE architecture. The SCALPEL back-end is a policy compiler that automatically lowers compartmentalization policies to the PIPE for hardware-accelerated enforcement. The SCALPEL front-end provides a set of compartment generation algorithms to help a security engineer explore the privilege-performance tradeoff space that can be achieved with the PIPE. The capstone algorithm presented in SCALPEL constructs policies by targetting a limit on the number of rules during each of a program's phases to achieve highly favorable cache characteristics. We show that the same technique can be used to produce designs with predictable runtime characteristics suitable for real-time systems. All together, SCALPEL shows that the PIPE can use fine-grained privilege separation with hundreds of compartments to achieve a very low overprivilege ratio with very low overheads.

CHAPTER 6 : Conclusion

Programmable, tag-based hardware security monitors like the PUMP can express and enforce instruction-level security policies. The focus of this dissertation is *policy engineering*, *i.e.,* the construction of policies that can (1) provide useful security properties, (2) impose low costs of enforcement, and (3) be applied to real software automatically to minimize the human involvement in their deployment to protect systems.

Chapter 3 introduced stack protection policies that carry forward information from the compiler about correct program behavior to enforce at runtime with tags and rules. The dominant source of overhead for these policies was the cost of tagging and clearing stack frames, which led us to design the *lazy tagging* policy engineering technique to reduce those costs; as a result, we find that object-level stack protection can be enforced at a cost of less than 5%.

In Chapter 4 we explore policies to protect the heap. We show that many common classes of heap errors can be defeated with the One-Color policy or the Allocation-Site policy with a single color at only 1% overhead. Full temporal and spatial heap safety can typically be provided at less than 10%, but some workloads can challenge the rule cache; we show how to relax the policies to reduce this overhead while maintaining useful security properties by cycling through pools of colors by allocation site.

Lastly, in Chapter 5 we show how high-performance compartmentalization policies can be constructed. For good performance on the PUMP, the number of rules required for each program phase should cache favorably; we introduce *rule packing*, a method to reduce the set of rules required by compartmentalization policies into working sets that achieve the targeted cache performance. To decide how to relax the compartmentalizations to achieve these goals, we introduce a quantitative privilege metric and a set of analysis tools that treat compartment generation as an optimization problem to automate the construction of compartmentalization policies. These allow a security engineer to quickly instantiate and evaluate tag-based compartmentalization policies. Lastly, we design and evaluate a rule prefetching system that can reduce the enforcement costs of compartmentalization policies by up to 4X by preemptively installing rules into the rule cache before they are required by the program.

BIBLIOGRAPHY

[1] CWE/SANS Top 25 Most Dangerous Software Errors. `https://cwe.mitre.org/top25/`, 2011.

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

[3] Ali Abbasi, Jos Wetzels, Thorsten Holz, and Sandro Etalle. Challenges in designing exploit mitigations for deeply embedded systems. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 31–46. IEEE, 2019.

[4] Periklis Akritidis, Cristian Cadar, Costin Raicui, Manuel Costa, and Miguel Castro. Preventing memory error exploits with WIT. In *Proceedings of the 29th IEEE Symposium on Security and Privacy*. IEEE, 2008.

[5] Aleph One. Smashing The Stack For Fun and Profit, November 1996.

[6] Daniel Aloise, Amit Deshpande, Pierre Hansen, and Preyas Popat. Np-hardness of euclidean sum-of-squares clustering. *Machine Learning*, 75(2):245–248, May 2009.

[7] David Anderson. David A's DWARF Page. `https://www.prevanders.net/dwarf.html`, 2017. 2017-8-12.

[8] Anmibe. Cpu features: Non-executable memory. `https://wiki.ubuntu.com/Security/CPUFeatures`. Accessed: 2021-2-13.

[9] anonymous. Once upon a free(). *Phrack Magazine*, (Vol. 0x0b Issue 0x39), 2001. `http://phrack.org/issues/57/9.html`.

[10] ARM. TrustZone technology for ARM v8-M Architeture. `https://developer.arm.com/documentation/100690/latest/`, 2016.

[11] Arthur Azevedo de Amorim. *A methodology for micro-policies*. PhD thesis, University of Pennsylvania, 2017.

[12] Arthur Azevedo de Amorim, Nathan Collins, André DeHon, Delphine Demange, Cătălin Hriţcu, David Pichardie, Benjamin C. Pierce, Randy Pollack, and Andrew Tolmach. A verified information-flow architecture. In *Proceedings of the 41st Symposium on Principles of Programming Languages*, POPL, pages 165–178. ACM, January 2014.

[13] Arthur Azevedo de Amorim, Maxime Dénès, Nick Giannarakis, Cătălin Hriţcu, Benjamin C. Pierce, Antal Spector-Zabusky, and Andrew Tolmach. Micro-policies: Formally verified, tag-based security monitors. In *36th IEEE Symposium on Security and Privacy (Oakland S&P)*, pages 813–830. IEEE Computer Society, May 2015.

[14] Adam Barth, Collin Jackson, Charles Reis, TGC Team, et al. The security architecture of the chromium browser. In *Technical report*. Stanford University, 2008.

[15] V.R. Basili and B.T. Perricone. Software errors and complexity: An empirical investigation. *Communications of the ACM*, pages 42–52, 1984.

[16] Ian Beer. An ios zero-click radio proximity exploit odyssey.

[17] Ian Beer. In-the-wild ios exploit chain 1. `https://googleprojectzero.blogspot.com/2019/08/in-wild-ios-exploit-chain-1.html`, Aug 2019.

[18] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. In *ACM SIGARCH Computer Architecture News*, 2014.

[19] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting Applications into Reduced-privilege Compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 309–322, Berkeley, CA, USA, 2008. USENIX Association.

[20] blackngel. Malloc Des-Maleficarum. *Phrack Magazine*, (Vol. 0x0d Issue 0x42), 2009. `http://phrack.org/issues/66/10.html`.

[21] blackngel. The House of Lore: Reloaded ptmalloc v2 and v3: Analysis and Corruption. *Phrack Magazine*, (Vol. 0x0e Issue 0x43), 2010. `http://phrack.org/issues/67/8.html`.

[22] Mark Brand. Virtually unlimited memory: Escaping the chrome sandbox. `https://googleprojectzero.blogspot.com/2019/04/virtually-unlimited-memory-escaping.html`, April 2019.

[23] J. Brown and Thomas 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.

[24] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. In *Proc. ACM CCS*, pages 27–38, Oct. 2008.

[25] Carnegie Mellon University Software Engineering Institute. *SEI CERT C Coding Standard*, 2016.

[26] Miguel Castro, Manual Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *USENIX Symposium on Operating System Design and Implementation*, 2006.

[27] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-Oriented Programming without Returns. In *International Conference on Information Systems Security (CCS)*. ACM, 2010.

[28] Shimin Chen, Michael Kozuch, Theodoros Strigkos, Babak Falsafi, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran, Olatunji Ruwase, Michael P. Ryan, and Evangelos Vlachos. Flexible Hardware Acceleration for Instruction-Grain Program Monitoring. In *35th International Symposium on Computer Architecture (ISCA)*, pages 377–388. IEEE, 2008.

[29] David Chisnall, Colin Rothwell, Robert N.M. Watson, Jonathan Woodruff, Munraj Vadera, Simon W. Moore, Michael Roe, Brooks Davis, and Peter G. Neumann. Beyond the PDP-11: Architectural support for a memory-safe C abstract machine. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 117–130. ACM, 2015.

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

[31] Abraham A. Clements, Naif Saleh Almakhdhub, Saurabh Bagchi, and Mathias Payer. ACES: Automatic compartments for embedded systems. In *27th USENIX Security Symposium (USENIX Security 2018)*, pages 65–82. USENIX Association, 2018.

[32] Mauro Conti, Stephen Crane, Lucas David, Michael Franz, Per Larsen, Christopher Liebchen, Marco Negro, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. Losing Control: On the Effectiveness of Control-Flow Integrity under Stack Attacks. In *ACM Conference on Computer and Communications Security*, pages 952–963. ACM, 2015.

[33] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaraon Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Symposium*, January 1998.

[34] The curl project. curl: command line tool and library. `https://curl.se/`. Accessed: 2021-2-13.

[35] Thurston Dang, Petros Maniatis, and David Wagner. The Performance Cost of Shadow Stacks and Stack Canaries. In *Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIACCS)*, April 2015.

[36] DARPA. Transparent computing. `https://www.darpa.mil/program/transparent-computing`. Accessed: 2020-9-30.

[37] André DeHon, Eli Boling, Rishiyur Nikhil, Darius Rad, Julie Schwarz, Niraj Sharma, Joseph Stoy, Greg Sullivan, and Andrew Sutherland. DOVER: A Metadata-Extended

RISC-V. In *RISC-V Workshop*, January 2016. Accompanying talk at `http://youtu.be/r5dIS1kDars`.

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

[39] Solar Designer. Jpeg com marker processing vulnerability, Jul 2000.

[40] CVE Details. Cve details: Libxml2 vulnerability statistics. `https://www.cvedetails.com/product/3311/Xmlsoft-Libxml2.html?vendor_id=1962`. Accessed: 2020-10-25.

[41] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. HardBound: Architectural support for spatial safety of the C programming language. pages 103–114, 2008.

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

[43] Udit Dhawan, Cătălin Hriţcu, Rafi Rubin, Nikos Vasilakis, Silviu Chiricescu, Jonathan M. Smith, Thomas F. Knight, Jr., Benjamin C. Pierce, and André DeHon. Architectural support for software-defined metadata processing. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 487–502, 2015.

[44] Udit Dhawan, Albert Kwon, Edin Kadric, Cătălin Hriţcu, Benjamin C. Pierce, Jonathan M. Smith, André DeHon, Gregory Malecha, Greg Morrisett, Thomas F. Knight, Jr., Andrew Sutherland, Tom Hawkins, Amanda Zyxnfryx, David Wittenberg, Peter Trei, Sumit Ray, and Greg Sullivan. Hardware support for safety interlocks and introspection. In *SASO Workshop on Adaptive Host and Network Security*, September 2012.

[45] Xinshu Dong, Hong Hu, Prateek Saxena, and Zhenkai Liang. A quantitative evaluation of privilege separation in web browser designs. In *European Symposium on Research in Computer Security*, pages 75–93. Springer, 2013.

[46] Dovecot. Dovecot mail server. `https://github.com/dovecot/core`. Accessed: 2020-10-12.

[47] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman. The Matter of Heartbleed. In *Proceedings of the Conference on Internet Measurement Conference*, 2014.

[48] Kevin Elphinstone and Gernot Heiser. From L3 to seL4 what have we learnt in 20 years of L4 microkernels? In *Proceedings of the ACM Symposium on Operating Systems Principles*, SOSP '13, pages 133–150, New York, NY, USA, 2013. ACM.

[49] Isaac Evans, Sam Fingeret, Julian Gonzalez, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. Missing the Point(er): On the Effectiveness of Code Pointer Integrity. In *IEEE Symposium on Security and Privacy (Oakland S&P)*. IEEE Computer Society, May 2015.

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

[51] Joseph A Fisher and Stefan M Freudenberger. Predicting conditional branch directions from previous runs of a program. *ACM SIGPLAN Notices*, 27(9):85–95, 1992.

[52] Institute for Informatics Georg-August-Universitat Gottingen. The mondial database. `https://www.dbis.informatik.uni-goettingen.de/Mondial`. 2020.

[53] LLVM Foundation. The llvm compiler infrastructure. `https://llvm.org/`. Accessed: 2021-2-13.

[54] Free Standards Group. *DWARF Debugging Information Format.*

[55] Sotiria Fytraki, Evangelos Vlachos, Yusuf Onur Koçberber, Babak Falsafi, and Boris Grot. FADE: A programmable filtering accelerator for instruction-grain monitoring. In *20th IEEE International Symposium on High Performance Computer Architecture, HPCA 2014, Orlando, FL, USA, February 15-19, 2014*, pages 108–119, 2014.

[56] GNU Project. GCC 4.1 Release Series Changes, New Features, and Fixes. `https://gcc.gnu.org/gcc-4.1/changes.html`, 2006. 2017-05-05.

[57] Google. Chrome security architecture. `https://docs.google.com/drawings/d/1TuECFL9K7J5q5UePJLC-YH3satvb1RrjLRH-tW_VKeE/edit`. [Online; accessed April 14, 2021].

[58] Khilan Gudka, Robert N.M. Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G. Neumann, and Alex Richardson. Clean Application Compartmentalization with SOAAP. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1016–1031, New York, NY, USA, 2015. ACM.

[59] Norm Hardy. The Confused Deputy (or why capabilities might have been invented). *SIGOPS Operating Systems Review*, 22(4):36–38, October 1988.

[60] L. Hatton. Reexamining the fault density component size connection. *IEEE Software*, 14(2):89–97, Mar 1997.

[61] HEX-Five. *MultiZone Security Reference Manual.* 2020.

[62] Terry Ching-Hsiang Hsu, Kevin Hoffman, Patrick Eugster, and Mathias Payer. Enforcing Least Privilege Memory Views for Multithreaded Applications. In *ACM Conf on Computer and Communication Security*, 2016.

[63] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks. In *IEEE Symposium on Security and Privacy (Oakland S&P)*. IEEE Computer Society, May 2016.

[64] Intel Corporation. Introduction to Intel Memory Protection Extensions. `https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions`, 2013. 2017-05-12.

[65] Intel Corporation. Control-flow Enforcement Technology Preview. `https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf`, 2016. 2017-05-17.

[66] A. Joannou, J. Woodruff, R. Kovacsics, S. W. Moore, A. Bradbury, H. Xia, R. N. Watson, D. Chisnall, M. Roe, B. Davis, E. Napierala, J. Baldwin, K. Gudka, P. G. Neumann, A. Mazzinghi, A. Richardson, S. Son, and A. T. Markettos. Efficient tagged memory. In *Proceedings of the International Conference on Computer Design (ICCD)*, pages 641–648, 2017.

[67] jp. Advanced Doug Lea's malloc exploits. *Phrack Magazine*, (Vol. 0x0b Issue 0x3d), 2001. `http://phrack.org/issues/61/6.html`.

[68] Chung Hwan Kim, Taegyu Kim, Hongjun Choi, Zhongshu Gu, Byoungyoung Lee, Xiangyu Zhang, and Dongyan Xu. Securing real-time microcontroller systems through customized memory view switching. In *NDSS*, 2018.

[69] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In *41st International Symposium on Computer Architecture (ISCA)*. IEEE, 2014.

[70] klog. The Frame Pointer Overwrite. *Phrack Magazine*, (Vol. 9 Issue 55), 1999. `http://phrack.org/issues/55/8.html`.

[71] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS and Other Systems. In *Proceedings of Crypto*, 1996.

[72] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-Pointer Integrity. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[73] Volodymyr Kuznetzov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. Code-pointer integrity. In *The Continuing Arms Race: Code-Reuse Attacks and Defenses*, pages 81–116. 2018.

[74] Draper Laboratory. Hope-tools github repository. `https://github.com/draperlaboratory/hope-src`. Accessed: 2020-10-05.

[75] lazytyped. ADI vs. ROP. `https://lazytyped.blogspot.com/2017/09/adi-vs-rop.html`, September 2017.

[76] Jiahao Li. Color reclamation for heap memory coloring scheme in pipe tagged-memory architecture. Master's thesis, Massachusetts Institute of Technology, 2019.

[77] Jochen Liedtke. On micro-Kernel Construction. In *15th ACM Symposium on Operating Systems Principles*, pages 237–250, 1995.

[78] Arm Limited. Arm cortex-a53 specification. `https://developer.arm.com/ip-products/processors/cortex-a/cortex-a53`. Accessed: 2020-10-05.

[79] Arm Limited. Armv8-m architecture reference manual. `https://developer.arm.com/documentation/ddi0553/ab/`. 2016.

[80] ARM Limited. ARM architecture reference manual: ARMv8, for ARMv8-A architecture profile, December 2017.

[81] Shen Liu, Dongrui Zeng, Yongzhe Huang, Frank Capobianco, Stephen McCamant, Trent Jaeger, and Gang Tan. Program-mandering: Quantitative privilege separation. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, New York, NY, USA, 2019. ACM.

[82] lowRISC project team. Tagged memory and minion cores in the lowRISC SoC. lowRISC-MEMO 2014-001, Computer Laboratory, University of Cambridge, December 2014. `http://www.lowrisc.org/docs/memo-2014-001-tagged-memory-and-minion-cores/`.

[83] Canonical Ltd. Apparmor. `https://wiki.ubuntu.com/AppArmor`. Accessed: 2020-9-11.

[84] Kangjie Lu, Chengyu Song, Byoungyoung Lee, Simon P Chung, Taesoo Kim, and Wenke Lee. Aslr-guard: Stopping address space leakage for code reuse attacks. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*, pages 280–291, 2015.

[85] MaXX. Vudo malloc tricks. *Phrack Magazine*, (Vol. 0x0b Issue 0x39), 2001. `http://phrack.org/issues/57/8.html`.

[86] MITRE. CVE-2012-0769. Available from MITRE, CVE-2012-0769 CVE-2012-0769., 2012.

[87] Sparsh Mittal. A survey of recent prefetching techniques for processor caches. *ACM Computing Surveys (CSUR)*, 49(2):1–35, 2016.

[88] P. Mohagheghi, R. Conradi, O. M. Killi, and H. Schwarz. An empirical study of software reuse vs. defect-density and stability. In *Proceedings. 26th International Conference on Software Engineering*, pages 282–291, May 2004.

[89] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman 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.

[90] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. CETS: Compiler enforced temporal safety for C. In *International Symposium on Memory Management*, June 2010.

[91] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdencewic. SoftBound: Highly compatible and complete spatial memory safety for c. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[92] Tim Newsham. Bugtraq: Re: Smashing the Stack: prevention?, Apr. 1997.

[93] Oracle. Introduction to SPARC M7 and Application Data Integrity (ADI). `https://swisdev.oracle.com/_files/What-Is-ADI.html`. Accessed: 2019-12-09.

[94] Oracle. Adi manual pages. `https://docs.oracle.com/cd/E86824_01/html/E54765/adi-2.html`, 2017.

[95] Tavis Ormandy and Chris Evans. The Poisoned Nul Byte 2014 Edition. `https://googleprojectzero.blogspot.com/2014/08/the-poisoned-nul-byte-2014-edition.html`, Aug 2014.

[96] Hilmi Ozdoganoglu, T.N. Vijaykumar, Carla E. Brodley, Benjamin A. Kuperman, and Ankit Jalote. SmashGuard: A Hardware Solution to Prevent Security Attacks on the Function Return Address. *IEEE Transactions on Computers*, 55:1271–1284, October 2006.

[97] Chris Palmer. The limits of sandboxing and next steps. USENIX Association, February 2021.

[98] Gabriel Parmer and Richard West. Mutable protection domains: Adapting system fault isolation for reliability and efficiency. *IEEE Transactions on Software Engineering*, 38(4):875–888, 2011.

[99] Phantasmal Phantasmagoria. The Malloc Maleficarum: Glibc Malloc Exploitation Techniques. `https://dl.packetstormsecurity.net/papers/attack/MallocMaleficarum.txt`, 2005.

[100] Marios Pomonis, Theofilos Petsios, Angelos D. Keromytis, Michalis Polychronakis, and Vasileios P. Kemerlis. kRˆX: Comprehensive Kernel Protection against Just-In-Time Code Reuse. In *Proc. of EuroSys*, pages 420–436, 2017.

[101] Aravind Prakash and Heng Yin. Defeating ROP Through Denial of Stack Pivot. In *Annual Computer Security Applications Conference*. ACM, 2015.

[102] The GNOME Project. The xml c parser and toolkit of gnome. `http://www.xmlsoft.org/`. Accessed: 2020-10-4.

[103] Qualsys, Inc. Qualys Security Advisory—The Stack Clash. `https://www.qualys.com/2017/06/19/stack-clash/stack-clash.txt`, 2017. 2018-03-29.

[104] Richard F. Rashid and George G. Robertson. Accent: A communication oriented network operating system kernel. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, SOSP '81, pages 64–75, New York, NY, USA, 1981. ACM.

[105] Nick Roessler. Exploiting LaTeX with CVE-2018-17407. `https://nickroessler.com/latex-cve-2018-17407/`, 2018.

[106] Nick Roessler and Andre DeHon. Protecting the Stack with Metadata Policies and Tagged Hardware. In *IEEE Symposium on Security and Privacy (S&P)*, 2018.

[107] Jerry H. Saltzer and Mike D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[108] Fuzzy Security. Part 12: Kernel Exploitation: Null Pointer Dereference. `https://www.fuzzysecurity.com/tutorials/expDev/16.html`.

[109] Andreas Sembrant. *Efficient techniques for detecting and exploiting runtime phases.* PhD thesis, Uppsala University, 2012.

[110] NXP Semiconductors. NXP selects dover microsystems' state-of-the-art coreguard cybersecurity technology for future embedded platforms. `https://media.nxp.com/news-releases/news-release-details/nxp-selects-dover-microsystems-state-art-coreguard-cybersecurity`, October 2018.

[111] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX Annual Technical Conference*, 2012.

[112] Amazon Web Services. Http web server example. `https://freertos.org/FreeRTOS-Plus/FreeRTOS_Plus_TCP/HTTP_web_Server.html`. Accessed: 2020-9-30.

[113] Hovav Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *ACM Conference on Computer and Communications Security*, pages 552–561. ACM, 2007.

[114] Timothy Sherwood, Erez Perelman, Greg Hamerly, Suleyman Sair, and Brad Calder. Discovering and exploiting program phases. *IEEE micro*, 23(6):84–93, 2003.

[115] Solar Designer. Bugtraq: Getting around non-executable stack (and fix), Aug. 1997.

[116] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. HDFI: Hardware-assisted data-flow isolation. In *IEEE Symposium on Security and Privacy (Oakland S&P)*. IEEE Computer Society, May 2016.

[117] Jia Song. *Security Tagging for a Real-time Zero-kernel Operating System: Implementation and Verification.* PhD thesis, University of Idaho, 2014.

[118] Jia Song and Jim Alves-Foss. Security tagging for a zero-kernel operating system. In *System Sciences (HICSS), 2013 46th Hawaii International Conference on*, pages 5049–5058. IEEE, 2013.

[119] Wei Song, Alex Bradbury, and Robert Mullins. Towards general purpose tagged memory. In *Proceedings of the RISC-V Workshop*, June 2015. `https://riscv.org/wp-content/uploads/2015/06/riscv-tagged-mem-workshop-june2015.pdf`.

[120] Alexander Sotirov. Heap feng shui in javascript. *Black Hat Europe*, 2007:11–20, 2007.

[121] Standard Performance Evaluation Corporation. SPEC CPU 2006. `https://www.spec.org/cpu2006/`, 2006.

[122] Gregory T Sullivan, André DeHon, Steven Milburn, Eli Boling, Marco Ciaffi, Jothy Rosenberg, and Andrew Sutherland. The dover inherently secure processor. In *2017 IEEE International Symposium on Technologies for Homeland Security (HST)*, pages 1–5. IEEE, 2017.

[123] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal war in memory. In *IEEE Symposium on Security and Privacy*, pages 48–62. IEEE Computer Society, 2013.

[124] The GNU Project. The GNU C Library (glibc). `https://www.gnu.org/software/libc/`.

[125] TIOBE. TIOBE Index for October 2017. `https://www.tiobe.com/tiobe-index/`, 2017. 2017-10-14.

[126] Stylianos Tsampas, Akram El-Korashy, Marco Patrignani, Dominique Devriese, Deepak Garg, and Frank Piessens. Towards automatic compartmentalization of c

programs on capability machines. In *Workshop on Foundations of Computer Security 2017*, pages 1–14, 2017.

[127] Victor Van der Veen, Lorenzo Cavallaro, Herbert Bos, et al. Memory errors: The past, the present, and the future. In *International Workshop on Recent Advances in Intrusion Detection*, pages 86–106. Springer, 2012.

[128] Guru Venkataramani, Ioannis Doudalis, Yan Solihin, and Milos Prvulovic. FlexiTaint: A programmable accelerator for dynamic taint propagation. pages 173–184, February 2008.

[129] M. A. Wahab, P. Cotret, M. N. Allah, G. Hiet, V. Laptre, and G. Gogniat. ARMHEx: A hardware extension for DIFT on ARM-based SoCs. In *Proceedings of the International Conference on Field-Programmable Logic and Applications*, pages 1–7, Sept 2017.

[130] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M Frans Kaashoek. Undefined Behavior: What Happened to My Code? In *Proceedings of the Asia-Pacific Workshop on Systems*, July 2012.

[131] R. N. M. Watson, R. M. Norton, J. Woodruff, S. W. Moore, P. G. Neumann, J. Anderson, D. Chisnall, B. Davis, B. Laurie, M. Roe, N. H. Dave, K. Gudka, A. Joannou, A. T. Markettos, E. Maste, S. J. Murdoch, C. Rothwell, S. D. Son, and M. Vadera. Fast protection-domain crossing in the CHERI capability-system architecture. *IEEE Micro*, 36(5):38–49, Sept 2016.

[132] John Wilander, Nick Nikiforakis, Yves Younan, Mariam Kamkar, and Wouer Joosen. RIPE: Runtime Intrusion Prevention Evaluator. In *27th Annual Computer Security Applications Conference (ACSAC)*. ACM, 2011.

[133] Emmett Witchel, Junghwan Rhee, and Krste Asanović. Mondrix: Memory Isolation for Linux Using Mondriaan Memory Protection. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 31–44, New York, NY, USA, 2005. ACM.

[134] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The CHERI capability model: Revisiting RISC in an age of risk. In *Proc. of the International Symposium on Computer Architecture (ISCA)*, pages 457–468, June 2014.

[135] Insu Yun, Dhaval Kapil, and Taesoo Kim. Automatic techniques to systematically discover new heap exploitation primitives. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.