2018 IEEE Symposium on Security and Privacy

# Protecting the Stack with Metadata Policies and Tagged Hardware

Nick Roessler
University of Pennsylvania
nroess@seas.upenn.edu

André DeHon
University of Pennsylvania
andre@acm.org

***Abstract*—The program call stack is a major source of exploitable security vulnerabilities in low-level, unsafe languages like C. In conventional runtime implementations, the underlying stack data is exposed and unprotected, allowing programming errors to turn into security violations. In this work, we design novel metadata-tag based, stack-protection security policies for a general-purpose tagged architecture. Our policies specifically exploit the natural locality of dynamic program call graphs to achieve cacheability of the metadata rules that they require. Our simple Return Address Protection policy has a performance overhead of 1.2% but just protects return addresses. The two richer policies we present, Static Authorities and Depth Isolation, provide object-level protection for all stack objects. When enforcing memory safety, our Static Authorities policy has a performance overhead of 5.7% and our Depth Isolation policy has a performance overhead of 4.5%. When enforcing data-flow integrity (DFI), in which we only detect a violation when a corrupted value is read, our Static Authorities policy has a performance overhead of 3.6% and our Depth Isolation policy has a performance overhead of 2.4%. To characterize our policies, we provide a stack threat taxonomy and show which threats are prevented by both prior work protection mechanisms and our policies.**

## I. Introduction

Low-level, memory-unsafe languages such as C/C++ are widely used in systems code and high-performance applications. However, 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 [1], 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, is readable and writeable by an executing program, and 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⊕X and stack canaries [2] 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 [3]. These expensive solutions are unused in practice due to their unacceptably high overheads [4].

There is a long history of accelerating security policies with hardware to bring their overheads to more bearable levels [5], [6], [7], [8], [9], [10], [11], [12], [13], [14]. However, introducing a dedicated hardware mechanism to address a single kind of vulnerability has disadvantages. Not only does a new hardware feature take many years to implement and deploy, but each mechanism can require independent changes to the entire hardware/software stack. For example, Intel's recent Memory Protection Extensions (MPX) [15], a hardware-accelerated mechanism for performing spatial memory safety checks on pointer accesses, added new hardware registers and new instructions in the instruction set, as well as required updated compilers, recompiled software and new operating system routines specific to MPX. Nonetheless, these additions did not fully address stack protection, demanding the later addition of separate hardware support and new instructions for stack protection in the form of CET, Control-flow Enforcement Technology [16]. Repeating this lengthy process for all desired security policies will result in bloated hardware (i.e., poor economy of mechanism) that cannot adapt to security threats at the rate at which they evolve.

Furthermore, a single, fixed policy will not be best suited for the range of applications and security requirements in practice. Protection mechanisms make tradeoffs between performance overhead, the protection provided, and compatibility, among other metrics. Different requirements and performance budgets likely lead to a range of solutions. A rigid, hardwired security mechanism, however, necessarily positions itself at a fixed point in the tradeoff space. CET, for example, provides hardware acceleration for coarse-grained Control-Flow Integrity (CFI) but cannot be used for fine-grained protection.

Recent work has shown that programmable, hardware-accelerated rich metadata tag-based security monitors are


DOI 10.1109/SP.2018.00066

capable of expressing and enforcing a large range of low-level security policies [17]. 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 [17]. 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, registers are spilled and where program variables are stored; similarly, 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 validate the compiler's intent and prevent the machine from violating the stack abstraction in unexpected ways 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. IV-D2) or by the stack depth (Sec. IV-D3) 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. VI-B), 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. VI-A) and Lazy Clearing (Sec. VI-C).

Lastly, to characterize our policies, we provide a taxonomy of stack threats (Sec. VII-A) and show how our policies as well as previous work protection mechanisms 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. VIII-B), 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

## II. Software-Defined Metadata Processing

The Software-Defined Metadata Processing (SDMP) model provides an abstraction for tag data processing that allows flexible, programmable policies to be enforced with hardware acceleration support. In the model, every word in the system, including memory, registers, and the program counter, is indivisibly extended with a metadata tag. As each instruction executes, the metadata on the inputs to the instruction are checked versus a software-defined policy. If the policy permits the operation, it supplies a metadata tag for the result, otherwise it raises a policy exception so the operating system can determine how to handle the security violation. Typically, the OS will terminate the offending program.

Abstractly, the metadata tag is unbounded. Concretely, the tag bits can be treated as a pointer to a rich data structure. These data structures can compose data from multiple different protection policies (e.g., CFI, heap memory, taint tracking) to allow simultaneous enforcement of an arbitrary number of different policies.

The inputs associated with an instruction include the OP-code of the current instruction (OP) (e.g., add, load, jump) and the tags associated with the Program Counter (PC), the Current Instruction itself (CI), the Register Source inputs (RS1, RS2), and the Memory input (M). In turn, the policy can provide result metadata for the Register or Memory Result (MR) and new metadata for the PC (PC′). The SDMP model allows software to define an arbitrary function from the operator and the 5 metadata inputs to an allow check and 2 metadata outputs. This function is pure in that no additional input state is

part of the functional computation. As we will see, all of these inputs and outputs are needed by stack protection policies.

For compact short hand, we typically write policies as a collection of rules of the form:

$$Op : (PC, CI, RS1, RS2, M) \rightarrow (PC', MR)$$

To accelerate computation, an SDMP implementation will typically include a hardware rule cache that maps from the rule inputs (operation and metadata tags) to results, such as the PUMP in [17]. Appropriately designed (e.g., [18]), a level-1 rule cache can perform this mapping in a single processor cycle so that policy rule checking does not slow execution. As with a normal data cache, the rule cache can have multiple levels to provide greater capacity without impacting common case cycle time. Misses to the final level of the rule cache trap to software handlers that compute the policy function and insert the missing rule into the rule cache. Rules in the cache are based on the concrete encoding of the metadata tags, including the pointer addresses. Because the metadata data structures are immutable and rule outputs depend only on the tag inputs, the rule cache does not need to dereference pointers to see their data or interpret the meaning of the tags.

Performance overhead is tightly related to rule locality. If a policy only needs a small number of distinct tags and rules for a program, the rules can fit into the level-1 rule cache with little overhead required to run software rule miss handlers to define the results. Similarly, if locality means the working set of tags and rules is small, there is little overhead. If the program must traverse a large number of distinct tags, it can exceed the level-1 rule cache capacity. If the program and policy create new tags rapidly, compulsory misses to create rules for the new tags can add to the overhead.

Tagged architectures have a long history [19] with early uses for typing [20]. Early work provided tags with hardwired semantics. Modern security interest was revived with single-bit information flow tracking [10] and has expanded in flexibility and bits [21], [22], [23], [24], with SDMP providing the most general and programmable metadata architecture. Prior flexible tagged architectures and monitoring architectures have not directly explored the strong stack protections we introduce here, and our policies can likely be adapted to many of these architectures (Sec. VIII-C).

## III. 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., [25][26]. 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 [27] to be out of scope. In Sec. VII-A we provide a set of specific threats to demonstrate an attacker's capabilities within our threat model.

```
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
```

Fig. 1: Typical Alpha stack maintenance code

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. In Fig. 1 we show typical Alpha assembly code for maintaining the stack.

## IV. Stack Protection Policies

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

### A. Motivation

Attacks on the stack arise from violations of high-level abstractions that are unchecked by the low-level code produced by compilers. Attackers exploit the machine's willingness to increment or decrement a pointer beyond the bounds of its intended object and to perform abstraction-violating reads and writes.

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.

### B. 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. Tags on registers encode the *frame-id* and *object-id* that a particular stack pointer is granted access, if the register contains a pointer to stack data. Lastly, instruction tags are used by the compiler to grant instructions capabilities beyond what generic instructions would have, 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 [28] to maintain pointer tags.

### C. 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 stack behavior, 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 $i$th 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.

### D. 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 and discuss how our policies handle common low-level features and optimizations in Appendix A, including *setjmp/longjmp*, tail calls, and dynamic stack memory allocations such as through *alloca*.

#### 1) 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 just the specific instructions generated by the compiler for this purpose (i.e., Sec. IV-C). It is designed to have comparable protection characteristics to mechanisms such as stack canaries [29], shadow stacks [26], or the HDFI stack protection policy [12], namely just 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 Appendix B.

**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 attacks that require attacker-synthesized return addresses, for which no corresponding call instructions were issued, from being loaded during return sequences (e.g., a standard 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 [31]; the additional protection for return addresses in memory is useful to assure a return flows to the correct instance. This policy could replace a shadow stack proposed by [31] for this purpose.

[1]We note, however, that this simple policy would not prevent sophisticated code reuse attacks, e.g., [30]. Our later policies provide protection for other code pointers on the stack as well.

*2) 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.

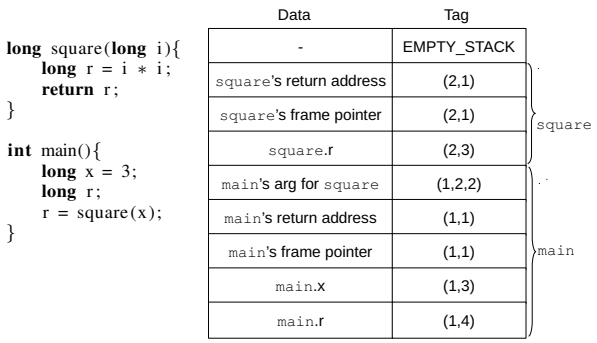
In this policy we enrich our notion of *object-id*s for precise object protection internal to a frame. Within each frame we statically assign a unique *object-id* to each program-level variable used by that function, including each primitive, array and structure in the frame; i.e., for each variable $V_i$ belonging to a function $f$ we assign a new differentiable *object-id* $i$. 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., [32], [33]. Due to the restricted way in which these compiler-managed objects are accessed (Sec. IV-C), 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. IV-C). The full rules for the policy are available in Appendix C. In Fig. 2 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, the compiler adds instructions that 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 an appro-

[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.




(a) Source Code (b) Stack Memory

Fig. 2: The Static Authorities tagging scheme. The tags we show are pairs (*frame-id*, *object-id*). In this example we assign *frame-id 1* to *main()* and *frame-id 2* to *square()*. We assign the *object-id 1* for stack control data, *object-id 2* for arguments, and use *3* and higher for program level variables. The word containing the passed argument is described in the text.

priate rule, *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 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. A special rule tags the resulting register with the corresponding *frame-id* and *object-id*. Additionally, in the function prologue, a *MAKE-PTR* is placed on the arithmetic instruction that subtracts from the stack pointer register to allocate the fresh frame. This transfers the *frame-id* from the static instruction to the active stack pointer (and subsequently the frame pointer). We use the same dynamic tainting rules as in [28] to propagate pointer tags between registers, to and from memory, and through pointer operations such as pointer arithmetic.

*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. For the general pointer case, a special rule allows the access when the *frame-id* and *object-id* of the accessing pointer matches the *frame-id* and *object-id* of the stack word.

*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. The frame pointer gets the correct tag by retrieving the stored frame pointer from the stack memory in the function epilogue.

*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*. 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 a special rule that allows the accesses 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 Appendix C.

*Clearing Memory*: To clear a function's allocated memory, the compiler adds additional instructions into the function epilogue tagged *CLEAR_MEM* that, with an appropriate rule, can 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.

**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 [34] and others [35]. 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" [36]. 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 [37].

*3) 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 available in Appendix D.

**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 function prologue, the compiler tags the instruction that allocates the stack frame with *INCR–DEPTH*; with an appropriate rule, 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 an appropriate rule, 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*, 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; 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 Depth Isolation, 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.

## V. Evaluation

### A. Methodology

We model the runtime overheads for our stack protection policies on the SPEC CPU2006 [38] 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.

*1) 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 [17]. 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 [39] 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 [17] 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.

*2) 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 [40] debug information generated by `gcc`, which we extract from the benchmark binaries and process using `libdwarf` [41]. This debug information gives the instruction tagger the layout of the stack memory, which it uses to tag instructions as described by the policies.

*3) Simulation:* Our evaluation framework is shown in Figure 3. We use `gem5` [42] for architectural statistics and generating instruction traces, a custom PUMP simulator for simulating the metadata tag subsystems of the simulated processor, and CACTI [39] for estimating memory access latencies for the final runtime calculations. After running an initial `gem5` simulation of the application, we process

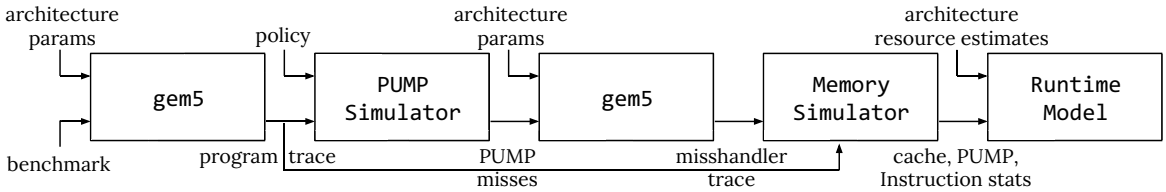



Fig. 3: Evaluation Framework

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.

### B. Results

*1) Return Address Protection:*

The Return Address Protection policy has a mean runtime overhead of 1.2% (Figure 4). 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%).

*2) Static Authorities:*

The Static Authorities policy has a mean runtime overhead of 11.9% (Figure 5). 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 memory. As can be seen in Figure 5, 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.

*3) Depth Isolation:*

The Depth Isolation policy has a mean runtime overhead of 8.5% (Figure 6). 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 misshandler 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. 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%).

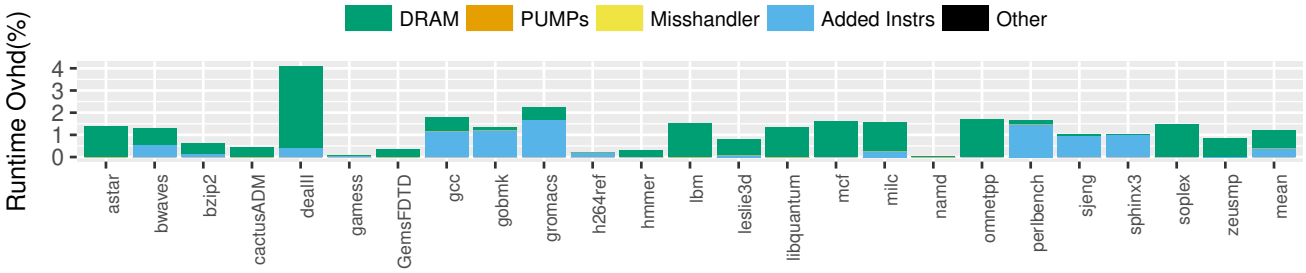



Fig. 4: Return Address Protection overhead

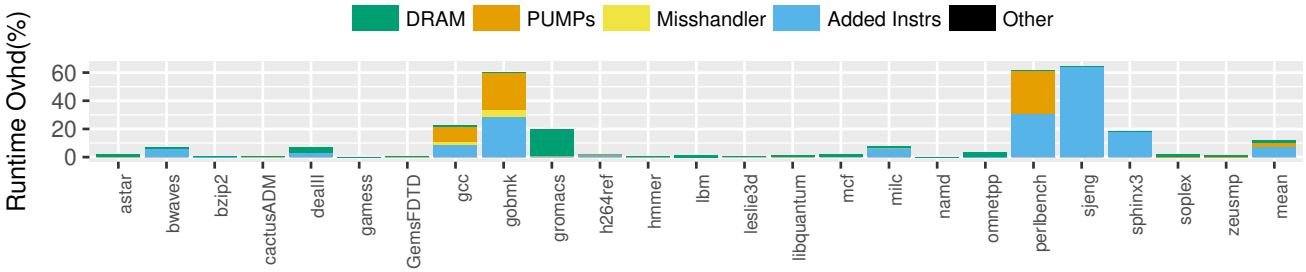



Fig. 5: Static Authorities overhead

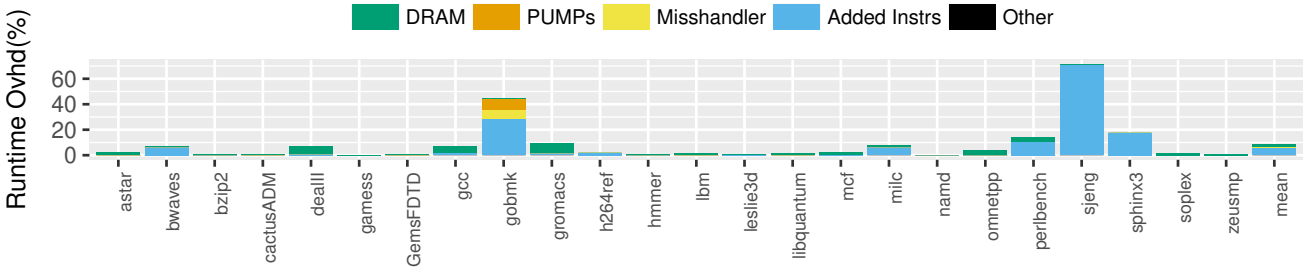



Fig. 6: Depth Isolation overhead

## VI. Optimizations

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 [35] 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. VII-A).

### A. 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. V-B2). 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. 7 and 8).

### B. 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× 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. 7 and 8). 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%.

### C. 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 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 [17], 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. 7 and 8. 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%.

## VII. Security Characterization

### A. 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 9. We select threats that decompose stack protection mechanisms along

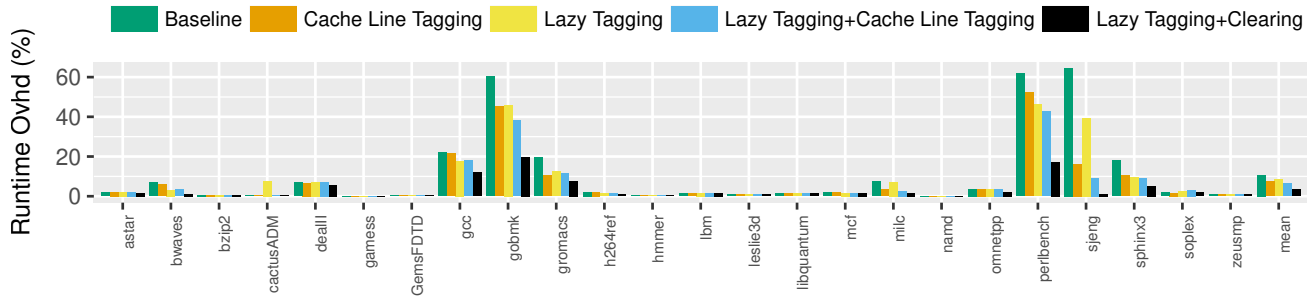



Fig. 7: Optimizations applied to Static Authorities

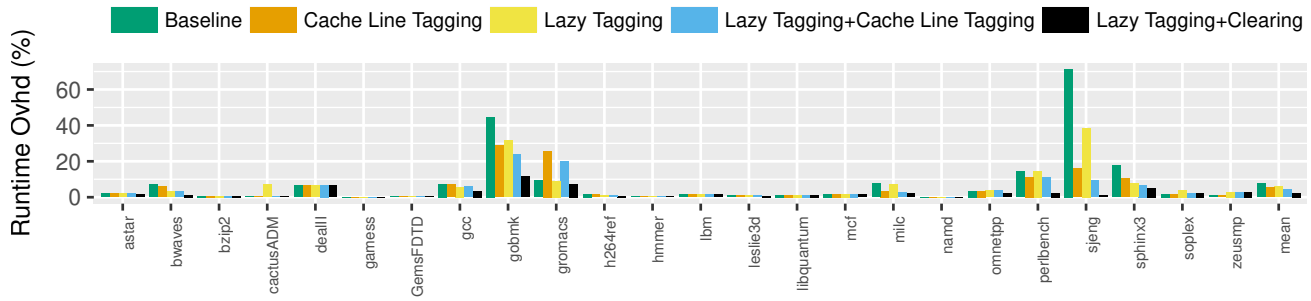



Fig. 8: Optimizations applied to Depth Isolation

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 [26], 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.

### B. Microbenchmarks

Due to the difficulty of porting an existing security benchmarking suite such as RIPE [46] to Alpha, we instead constructed a set of security microbenchmarks for testing and 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.

## VIII. Related Work

### A. Stack Protection

Due to the prevalence of stack memory safety exploits, stacks have been the subject of many defensive efforts [4]. 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 [47], [48], [49], [50], and ASLR can be subverted with information leaks [51].

Low-overhead, software-only stack protection solutions such as StackGuard [29] and shadow stacks [26] 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% [26]; we include the optimized Parallel Shadow Stack variant in our taxonomy. Hardware support for shadow stacks has been proposed (SmashGuard [43]); recently Intel has announced upcoming hardware support for the feature in their Control-flow Enforcement Technology [16].

AddressSanitizer [44] instruments all memory accesses with checks against “red zones” in a shadow memory that pads all

| | | 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 [29] [26] | X | ✓ | X | X | X | 2.8% |
| | Parallel Shadow Stack [26] | X | ✓ | X | X | X | 3.5% |
| | SmashGuard [43] | X | ✓ | ✓ | X | X | ∼ 0% |
| | Intel's Control-flow Enforcement Technology [16] | X | ✓ | ✓ | X | X | |
| | AddressSanitizer [44] | X | ✓ | X | ✓ | X | 73% |
| | CPI/CPS [25] | X | ✓ | ✓ | X | X | 8.5%/1.9% |
| | Hardware-Assisted Dataflow-Isolation [12] | X | ✓ | ✓ | X | X | $< 2$ % |
| | SoftBound [45] | ✓ | ✓ | ✓ | ✓ | ✓ | 67% |
| | HardBound [11] | ✓ | ✓ | ✓ | ✓ | ✓ | 5-9% |
| **Our Policies** | Return Address Protection (Sec. IV-D1) | X | ✓ | ✓ | X | X | 1.2% |
| | Static Authorities (Sec. IV-D2) | | | | | | |
| | Memory Safety | ✓ | ✓ | ✓ | ✓ | ✓ | 5.7% |
| | Data-flow Integrity | ✓ | R | R | R | R | 3.6% |
| | Depth Isolation (Sec. IV-D3) | | | | | | |
| | Memory Safety | ✓ | ✓ | ✓ | ✓ | ✓ | 4.5% |
| | Data-flow Integrity | ✓ | R | R | R | R | 2.4% |

✓prevents the specified access; X allows it; R denotes cases where writes are allowed but violations are detected when overwritten data or data that should be inaccessible is read.

Fig. 9: Stack threat taxonomy

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 [25]). 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 [52]. 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 [53].

Hardware-Assisted Data-flow Isolation (HDFI) [12] 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 [54]. LowRISC provides two bits of tagging in its memory system that could be used to implement HDFI with its ltag/stag operations [55], [56].

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 [57] 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 [58]. 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 [35], 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 [45], [3] can provide complete memory safety using software checks, but are expensive (116% overhead). Hardware support for bounds checking, such as HardBoud [11], Intel's MPX [15] and CHERI [14], [59] 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.

### B. 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) [17], [60], Information-Flow Control (IFC) [61], [62], Instruction and Data Tainting [17], Minimal Typing [17], Compartmentalization [60], Dynamic Sealing [60], Self Protection [60], and Heap Memory Safety [17], [60]. These previous policies did not address protecting the program stack. The previous memory safety work [17] [60] 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.

### C. Policy Applicability

Several systems provide programmable, multi-bit metadata tags that could exploit the policies we derive here [63], [23], [64], [65]. Aries [63] would need to be extended to include tags on memory. Harmoni [23] 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 [64] 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 [65] 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 [66], [67], [68]. LBA and FADE [66], [67] 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 [68]. 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.

## IX. 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.

## X. 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.

## Acknowledgements

The authors would like to thank the anonymous reviewers, as well as Cătălin Hriţcu, Benjamin Pierce, Greg Sullivan, Eli Boling, Nathan Dautenhahn, Nikos Vasilakis and Ben Karel for their valuable feedback. This research was funded by National Science Foundation grant TWC-1513854. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not reflect the official policy or position of the National Science Foundation or the U.S. Government.

## REFERENCES


[1] TIOBE. (2017) TIOBE Index for October 2017. https://www.tiobe.com/tiobe-index/. 2017-10-14.

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

[3] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic, "CETS: Compiler enforced temporal safety for C," in *International Symposium on Memory Management*, Jun. 2010.

[4] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal war in memory," in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2013, pp. 48–62. [Online]. Available: http://lenx.100871.net/papers/War-oakland-CR.pdf

[5] M. D. Schroeder and J. H. Saltzer, "A hardware architecture for implementing protection rings," *Communications of the ACM*, vol. 15, no. 3, pp. 157–170, March 1972.

[6] E. A. Feustel, "Tagged architecture and the semantics of programming languages: Extensible types," in *3rd Annual Symposium on Computer Architecture (ISCA)*. ACM, 1976, pp. 147–150.

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

[8] O. Saydjari, J. Beckman, and J. Leaman, "Lock trek: Navigating uncharted space," in *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, 1989.

[9] E. Witchel, J. Cates, and K. Asanović, "Mondrian memory protection," in *10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS X. New York, NY, USA: ACM, 2002, pp. 304–316. [Online]. Available: http://doi.acm.org/10.1145/605397.605429

[10] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004, pp. 85–96. [Online]. Available: http://csg.csail.mit.edu/pubs/memos/Memo-467/memo-467.pdf

[11] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic, "HardBound: Architectural support for spatial safety of the C programming language," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008, pp. 103–114.

[12] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek, "HDFI: Hardware-assisted data-flow isolation," in *IEEE Symposium on Security and Privacy (Oakland S&P)*. IEEE Computer Society, May 2016.

[13] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution," in *Workshop on Hardware and Architectural Support for Security and Privacy*, 2013, pp. 10:1–10:1. [Online]. Available: http://doi.acm.org/10.1145/2487726.2488368

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

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

[16] ——, "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.

[17] U. Dhawan, C. Hriţcu, R. Rubin, N. Vasilakis, S. Chiricescu, J. M. Smith, T. F. Knight, Jr., B. C. Pierce, and A. DeHon, "Architectural support for software-defined metadata processing," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015, pp. 487–502.

[18] U. Dhawan and A. DeHon, "Area-efficient near-associative memories on FPGAs," *Transactions on Reconfigurable Technology and Systems*, vol. 7, no. 4, pp. 3:1–3:22, Jan. 2015. [Online]. Available: http://doi.acm.org/10.1145/2629471

[19] E. A. Feustel, "On the Advantages of Tagged Architectures," *IEEE Transactions on Computers*, vol. 22, pp. 644–652, Jul. 1973. [Online]. Available: http://www.feustel.us/Feustel%20&%20Associates/Advantages.pdf

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

[21] M. Dalton, H. Kannan, and C. Kozyrakis, "Raksha: a flexible information flow architecture for software security," in *International Symposium on Computer Architecture (ISCA)*, 2007, pp. 482–493. [Online]. Available: http://www.engr.uconn.edu/~zshi/course/cse5302/ref/dalton07raksha_isca.pdf

[22] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic, "FlexiTaint: A programmable accelerator for dynamic taint propagation," in *Proceedings of the International Symposium on High-Performance Computer Architecture*, Feb. 2008, pp. 173–184. [Online]. Available: http://www.cc.gatech.edu/~milos/venkataramani_hpca08.pdf

[23] D. Y. Deng and G. E. Suh, "High-performance parallel accelerator for flexible and efficient run-time monitoring," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE Computer Society, 2012, pp. 1–12. [Online]. Available: http://tsg.ece.cornell.edu/lib/exe/fetch.php?media=pubs:flex-dsn2012.pdf

[24] Y.-Y. Chen, P. A. Jamkhedkar, and R. B. Lee, "A software-hardware architecture for self-protecting data," in *ACM Conference on Computer and Communications Security*. ACM, 2012, pp. 14–27. [Online]. Available: http://palms.princeton.edu/system/files/chen_ccs12.pdf

[25] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-Pointer Integrity," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[26] T. Dang, P. Maniatis, and D. 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.

[27] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. 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. [Online]. Available: https://users.ece.cmu.edu/~yoonguk/papers/kim-isca14.pdf

[28] J. A. Clause, I. Doudalis, A. Orso, and M. Prvulovic, "Effective memory protection using dynamic tainting," in *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, 2007, pp. 284–292. [Online]. Available: http://www.cc.gatech.edu/~orso/papers/clause.doudalis.orso.prvulovic.pdf

[29] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," in *Proceedings of the 7th USENIX Security Symposium*, January 1998.

[30] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-Oriented Programming without Returns," in *International Conference on Information Systems Security (CCS)*. ACM, 2010.

[31] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity," in *12th ACM Conference on Computer and Communications Security*. ACM, 2005, pp. 340–353. [Online]. Available: https://research.microsoft.com/apps/pubs/?id=69217

[32] klog. (1999) The Frame Pointer Overwrite. http://phrack.org/issues/55/8.html.

[33] M. Conti, S. Crane, L. David, M. Franz, P. Larsen, C. Liebchen, M. Negro, M. Qunaibit, and A.-R. Sadeghi, "Losing Control: On the Effectiveness of Control-Flow Integrity under Stack Attacks," in *ACM Conference on Computer and Communications Security*. ACM, 2015, pp. 952–963.

[34] P. Akritidis, C. Cadar, C. Raicui, M. Costa, and M. Castro, "Preventing memory error exploits with WIT," in *Proceedings of the 29th IEEE Symposium on Security and Privacy*. IEEE, 2008.

[35] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *USENIX Symposium on Operating System Design and Implementation*, 2006.

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

[37] A. Prakash and H. Yin, "Defeating ROP Through Denial of Stack Pivot," in *Annual Computer Security Applications Conference*. ACM, 2015.

[38] Standard Performance Evaluation Corporation. (2006) SPEC CPU 2006. https://www.spec.org/cpu2006/.

[39] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0: A tool to model large caches," HP Labs, Palo Alto, CA, HPL 2009-85, April 2009, latest code release for CACTI 6 is 6.5. [Online]. Available: http://www.hpl.hp.com/techreports/2009/HPL-2009-85.html

[40] Free Standards Group, *DWARF Debugging Information Format*, http://www.dwarfstd.org/doc/Dwarf3.pdf.

[41] D. Anderson. (2017) David A's DWARF Page. https://www.prevanders.net/dwarf.html. 2017-8-12.

[42] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," in *ACM SIGARCH Computer Architecture News*, 2014.

[43] H. Ozdoganoglu, T. Vijaykumar, C. E. Brodley, B. A. Kuperman, and A. Jalote, "SmashGuard: A Hardware Solution to Prevent Security Attacks on the Function Return Address," *IEEE Transactions on Computers*, vol. 55, pp. 1271–1284, Oct. 2006. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=1683758

[44] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-Sanitizer: A Fast Address Sanity Checker," in *USENIX Annual Technical Conference*, 2012.

[45] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdencewic, "Soft-Bound: Highly compatible and complete spatial memory safety for c," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[46] J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen, "RIPE: Runtime Intrusion Prevention Evaluator," in *27th Annual Computer Security Applications Conference (ACSAC)*. ACM, 2011.

[47] H. 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*. ACM, 2007, pp. 552–561. [Online]. Available: http://cseweb.ucsd.edu/~hovav/dist/geometry.pdf

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

[49] T. Newsham, "Bugtraq: Re: Smashing the Stack: prevention?" Apr. 1997. [Online]. Available: http://seclists.org/bugtraq/1997/Apr/129

[50] Solar Designer, "Bugtraq: Getting around non-executable stack (and fix)," Aug. 1997. [Online]. Available: http://seclists.org/bugtraq/1997/63

[51] MITRE, "CVE-2012-0769." Available from MITRE, CVE-2012-0769 CVE-2012-0769., 2012. [Online]. Available: https://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-0769

[52] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. 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.

[53] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. 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.

[54] 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)*, 2017, pp. 641–648. [Online]. Available: doi.ieeecomputersociety.org/10.1109/ICCD.2017.112

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

[56] W. Song, A. Bradbury, and R. 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.

[57] A. Limited, "ARM architecture reference manual: ARMv8, for ARMv8-A architecture profile," December 2017. [Online]. Available: https://static.docs.arm.com/ddi0487/ca/DDI0487C_a_armv8_arm.pdf

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

[59] D. Chisnall, C. Rothwell, R. N. Watson, J. Woodruff, M. Vadera, S. W. Moore, M. Roe, B. Davis, and P. 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*, ser. ASPLOS '15. ACM, 2015, pp. 117–130. [Online]. Available: http://doi.acm.org/10.1145/2694344.2694367

[60] A. Azevedo de Amorim, M. Dénès, N. Giannarakis, C. Hriţcu, B. C. Pierce, A. Spector-Zabusky, and A. Tolmach, "Micro-policies: Formally verified, tag-based security monitors," in *36th IEEE Symposium on Security and Privacy (Oakland S&P)*. IEEE Computer Society, May 2015, pp. 813–830. [Online]. Available: http://prosecco.gforge.inria.fr/personal/hritcu/publications/micro-policies.pdf

[61] A. Azevedo de Amorim, N. Collins, A. DeHon, D. Demange, C. Hriţcu, D. Pichardie, B. C. Pierce, R. Pollack, and A. Tolmach, "A verified information-flow architecture," in *Proceedings of the 41st Symposium on Principles of Programming Languages*, ser. POPL. ACM, Jan. 2014, pp. 165–178. [Online]. Available: http://www.crash-safe.org/node/29

[62] A. Azevedo de Amorim, "A methodology for micro-policies," Ph.D. dissertation, University of Pennsylvania, 2017. [Online]. Available: http://www.seas.upenn.edu/~aarthur/thesis.pdf

[63] J. Brown and T. F. Knight, Jr., "A minimally trusted computing base for dynamically ensuring secure information flow," MIT CSAIL, Tech. Rep. 5, November 2001, aries Memo No. 15. [Online]. Available: http://www.ai.mit.edu/projects/aries/Documents/Memos/ARIES-15.pdf

[64] U. Dhawan, A. Kwon, E. Kadric, C. Hriţcu, B. C. Pierce, J. M. Smith, A. DeHon, G. Malecha, G. Morrisett, T. F. Knight, Jr., A. Sutherland, T. Hawkins, A. Zyxnfryx, D. Wittenberg, P. Trei, S. Ray, and G. Sullivan, "Hardware support for safety interlocks and introspection," in *SASO Workshop on Adaptive Host and Network Security*, Sep. 2012. [Online]. Available: http://www.crash-safe.org/sites/default/files/interlocks_ahns2012.pdf

[65] A. DeHon, E. Boling, R. Nikhil, D. Rad, J. Schwarz, N. Sharma, J. Stoy, G. Sullivan, and A. Sutherland, "DOVER: A Metadata-Extended RISC-V," in *RISC-V Workshop*, Jan. 2016, accompanying talk at http://youtu.be/r5dIS1kDars. [Online]. Available: http://riscv.org/wp-content/uploads/2016/01/Wed1430-dover_riscv_jan2016_v3.pdf

[66] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. C. Mowry, V. Ramachandran, O. Ruwase, M. P. Ryan, and E. Vlachos, "Flexible Hardware Acceleration for Instruction-Grain Program Monitoring," in *35th International Symposium on Computer Architecture (ISCA)*. IEEE, 2008, pp. 377–388. [Online]. Available: http://www.cs.cmu.edu/~lba/papers/LBA-isca08.pdf

[67] S. Fytraki, E. Vlachos, Y. O. Koçberber, B. Falsafi, and B. Grot, "FADE: A programmable filtering accelerator for instruction-grain monitoring," in *20th IEEE International Symposium on High Performance Computer Architecture, HPCA 2014, Orlando, FL, USA, February 15-19, 2014*, 2014, pp. 108–119. [Online]. Available: http://dx.doi.org/10.1109/HPCA.2014.6835922

[68] 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*, Sept 2017, pp. 1–7.

## APPENDIX

### A. Implementation Challenges

**setjmp/longjmp:** 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*. These stores are violations of the stack invariants as discussed in Sec. IV-C; 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

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

Fig. 10: Return Address Protection rules

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.

### B. Return Address Protection

Figure 10 shows the rule set for the Return Address Protection. In our rule notation we use $\bot$ for the empty tag and _ to indicate a don't-care for a particular field, which means that the rule does not depend on a particular input and may match any tag.

### C. Static Authorities

In this section we explain how we tag arguments for indirect function calls (as referenced in Sec. IV-D2), as well as provide the full rules for the policy in SDMP notation.

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. Then, before an indirect call takes place, we use a special instruction tagged *BEGIN–INDIRECT–CALL*, which, with an appropriate rule, 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 *SET–ARG–FROM–PC* and use the authority identifier held in the program counter tag to set the appropriate *argument_for* field. Finally, the indirect call instruction clears the tag on the PC when it executes.

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* so that when these values are loaded the resulting registers get tagged correctly. Function pointers can also be crafted dynamically by arithmetic instructions that compute at offset from the global register. We tag these instructions with the *instruction-type CREATE–FP* along with appropriate *frame-id* for the function pointer that they are creating, so that with an appropriate rule the resulting register will contain the correct *frame-id*.

In the rules we show in Fig. 11, 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.

### D. Depth Isolation

In the rules we show in Fig. 12, we display tags on instructions as singletons of the form (*instruction-type*), tags on registers as doubles of the form (*frame-id*, *object-id*) and tags on memory as 4-tuples of the form (*frame-id*, *object-id*, *frame-id-ptr*, *object-id-ptr*).

$(1) Lda : (\bot, (MAKE\text{–}PTR, f, o), \bot, \_, \bot) \rightarrow (\bot, (f, o, \bot))$
$(2) Store : (\bot, (SET\text{–}MEM, f, o), \bot, (f, \_, \bot), EMPTY\text{–}STACK) \rightarrow (\bot, (f, o, \bot, \bot, \bot))$
$(3) Store : (\bot, (CLEAR\text{–}MEM, f, \bot), \bot, (f, \bot, \bot), \_) \rightarrow (\bot, EMPTY\text{–}STACK)$
$(4) Store : (\bot, (ACCESS\text{–}LOCAL, f, o), (f2, o2, p), (f, \bot, \bot), (f, o, \_, \_, \_)) \rightarrow (\bot, (f, o, f2, o2, p))$
$(5) Load : (\bot, (ACCESS\text{–}LOCAL, f, o), \_, (f, \bot, \bot), (f, o, f2, o2, p)) \rightarrow (\bot, (f2, o2, p))$
$(6) Arith\_prop : (\bot, (INSTR, \_), \bot, \bot, \bot) \rightarrow (\bot, \bot)$
$(7) Arith\_prop : (\bot, (INSTR, \_), \bot, (f, o, p), \bot) \rightarrow (\bot, (f, o, p))$
$(8) Arith\_prop : (\bot, (INSTR, \_), (f, o, p), \bot, \bot) \rightarrow (\bot, (f, o, p))$
$(9) Arith\_prop : (\bot, (INSTR, \_), (f1, o1, p1), (f2, o2, p2), \bot) \rightarrow (\bot, \bot)$
$(10) Arith\_no\_prop : (\bot, (INSTR, \_), (f1, o1, p1), (f2, o2, p2), \bot) \rightarrow (\bot, \bot)$
$(11) Store : (\bot, (INSTR, \_), \bot, \bot, \bot) \rightarrow (\bot, \bot)$
$(12) Load : (\bot, (INSTR, \_), \bot, \bot, \bot) \rightarrow (\bot, \bot)$
$(13) Store : (\bot, (INSTR, \_), (f2, o2, p), (f1, o1, \bot), (f1, o1, \_, \_, \_)) \rightarrow (\bot, (f1, o1, f2, o2, p))$
$(14) Load : (\bot, (INSTR, \_), \_, (f1, o1, \bot), (f1, o1, f2, o2, p)) \rightarrow (\bot, (f2, o2, p))$
$(15) Store : (\bot, (INSTR, f), (f2, o2, p2), \_, (\_, ARG, \_, \_, \_, ARGFOR = f)) \rightarrow (\bot, (f, ARG, f2, o2, p2, ARGFOR = f)$
$(16) Load : (\bot, (INSTR, f), \_, \_, (\_, ARG, f2, o2, p, ARGFOR = f)) \rightarrow (\bot, (f2, o2, p)$
$(17) Store : (\bot, (SET\text{–}ARG, f1, f2), (f3, o3, p), (f1, \bot, \bot), (f1, \_, \_, \_, \_)) \rightarrow (\bot, (f1, ARG, f3, o3, ARGFOR = f2))$
$(18) Arith\_ : (\bot, (CREATE\text{–}FP, f, p), \bot, \_, \bot) \rightarrow (\bot, p)$
$(19) Store : (\bot, (LONGJMP\text{–}CLEAR, \_), \bot, \_, (\_, \_, \_, \_, \_)) \rightarrow (\bot, EMPTY\text{–}STACK)$
$(20) Other : (\bot, (INSTR, \bot, \bot), \bot, \bot, \bot) \rightarrow (\bot, \bot)$
$(21) Arith\_prop : (\bot, (BEGIN\text{–}INDIRECT\text{–}CALL, f), \bot, (\bot, \bot, p), \bot) \rightarrow (p, \bot)$
$(22) Store : (pc, (SET\text{–}ARG\text{–}FROM\text{–}PC, f), (f2, o2, p), (f, \_, \bot), (f, \_, \_, \_, \_)) \rightarrow (pc, (f, ARG, f2, o2, p, ARGFOR = pc))$
$(23) Jsr : (\_, (\_, INSTR, \bot, \bot), \bot, \bot, \bot) \rightarrow (\bot, \bot)$

Fig. 11: Static Authorities rules

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

Fig. 12: Depth Isolation rules