Towards Improved Mitigations for Two Attacks on Memory Safety

Thurston Dang
David Wagner
Petros Maniatis

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2017-209
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-209.html

December 13, 2017

Acknowledgement

Thanks to Nicholas Carlini, Ulfar Erlingsson, Mathias Payer, and David Fifield and the anonymous reviewers for helpful comments. This dissertation was supported by Intel through the ISTC for Secure Computing, by the AFOSR under MURI award FA9550-12-1-0040, the Hewlett Foundation through the Center for Long-Term Cybersecurity, the National Science Foundation under grants CCF-0424422 and CNS-1514457, and BEARS.

A special thank you to Google for allowing Petros Maniatis to serve as my co-advisor.

Towards Improved Mitigations for Two Attacks on Memory Safety

by

Thurston Hou Yeen Dang

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor David Wagner, Co-chair
Dr Petros Maniatis, Co-chair
Professor Vern Paxson
Professor John Chuang

Fall 2017

Towards Improved Mitigations for Two Attacks on Memory Safety

Abstract

Towards Improved Mitigations for Two Attacks on Memory Safety

by

Thurston Hou Yeen Dang

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor David Wagner, Co-chair

Dr Petros Maniatis, Co-chair

C, C++ and most other popular low-level languages delegate memory management to the programmer, frequently resulting in bugs. Accordingly, a longstanding problem in computer security is efficient, backwards-compatible prevention of the data and control-flow exploits that arise from writing past the end of a buffer or using memory after it has been freed.

In the first part of this dissertation, we consider protection schemes against the most popular form of control-flow hijacking: return-oriented programming (ROP), which depends on misusing `RET` instructions. Control-flow defenses against ROP either use strict, expensive, but strong protection against redirected `RET` instructions with shadow stacks or other dual-stack schemes, or much faster but weaker protections without. We study the inherent overheads of shadow stack schemes (≈10%). We then design a new scheme, the parallel shadow stack, with significantly less overhead (≈3.5%) and better compatibility. Our measurements suggest it will not be easy to further improve software-only shadow stack performance on current x86 processors, due to inherent costs associated with `RET` and memory load/store instructions.

Next, we consider defenses against heap use-after-free, which is an increasingly important class of memory safety errors. We show that, in principle, page permissions should be the most desirable approach. We then validate this experimentally by designing, implementing, and evaluating Oscar, a new protection scheme based on page permissions. Oscar does not require source code, is compatible with standard and custom memory allocators, works correctly with programs that `fork`, and performs favorably — often by more than an order of magnitude — compared to recent proposals: overall, it has similar or lower runtime overhead, and lower memory overhead than competing systems.

Yesteryear's page-permissions-based allocators, including Oscar, all place one object per virtual page, to allow physical memory to be reclaimed as soon as the object is freed. We revisit this principle in Oscar++: we place multiple objects per page, with a secure quarantine for freed objects on pages that still have other live objects, and efficient inline metadata. On average, this more than halves the overhead for allocation-intensive benchmarks. We

also consider the use of object lifetime, to prevent comingling of short-lived and long-lived objects on the same virtual page; this shows some promise for reducing memory overhead from quarantine.

In the last chapter, we conclude with some lessons and common themes from the three projects.

To my parents, Co and Tinh

Contents

List of Figures

List of Tables

Acknowledgments

Thank you to my co-advisors David Wagner and Petros Maniatis, for being exceptionally patient, kind, and supportive mentors. Thank you for giving me a chance when I was just a lost computational biology student without a klee-r direction (sorry!), and guiding me on the long and windy road that is my grad school journey; from the abyss of bochs, through the dark valley of formal verification, to this dissertation. You both rekindled my interest in computer security, and taught me the joys and challenges of systems security research. I feel exceptionally fortunate and privileged to have been able to learn from you.

Thank you to all the faculty who served on my committees and generously gave their time and expertise:

- John Chuang: dissertation committee
- Vern Paxson: qualifying exam and dissertation committees. Thanks as well for your Network Security class; it was a highly engaging and influential course, that cemented my interest in computer security.
- Dawn Song: qualifying exam committee
- Allan Sly: masters thesis, qualifying exam, and dissertation committees. Thanks for your Probability for Applications class, which laid the groundwork for my statistics masters.
- Satish Rao: masters thesis committee. Thanks for your Algorithms class, and for always being friendly and approachable even when I wasn't "your" student.
- last but not least, Elchanan Mossel. Thanks for being my masters thesis advisor, PhD temporary advisor, and a strong advocate and mentor for me throughout grad school.

Thanks to all the wonderful and helpful staff, especially La Shana Porlaris (doubly so: as grad student services advisor in CS, and then statistics), Xuan Quach, Audrey Sillers, Angela Waxman, and Shirley Salanio (Center for Student Affairs), and Angie Abbatecola and Lena Lau-Stewart (to paraphrase the plaque on the 7th floor of Soda, it definitely is home to top-shelf staff).

This PhD — and my academic journey to follow — would not have been possible without my mentors from before grad school. Thanks to Catherine Suter, Jennifer Cropley, Vanessa Venturi, and Miles Davenport, for giving me the opportunity to do computational biology research after undergrad. While I did not quite get the computational biology PhD that I envisioned, you have all been excellent role models through your inquisitive and rigorous approach to research. Thanks to Richard Buckland and Julie Henry for being my undergrad honors thesis advisors and inspiring instructors for many courses. I am also grateful to Richard for having ignited my interest in computer security (over a decade ago!), and for introducing me to the ways of teaching.

Thank you to my office mates, David Fifield, Michael McCoyd, and Richard Shin, for good company and interesting discussions.

Thanks for my parents Co and Tinh, and my siblings Lawrence, Clarence, Jennifer, Evelyn and Jocelyn, for their love and support.

Thanks to my friends for fun times and for keeping me sane during grad school. Special mentions to:

- Berkeley: Aditya Nandy (the god/infinite being of chemistry), Allen Chen, Shur Batmunkh, Michelle Leung (POTATOES POTATOES POTATOES), Cinyi Mao (the NiCEst), Yuu Ohno, Sarah Almubarak, Aziz Asaly, Chris Yamamoto, Tommy Wu
- UCLA: Ai Ohno, Julia Feng (the OG meme queen), Elena Pulkinen
- UCSD: Natalie McLain, the nugget brigade (Kerk Ang, Katie Chandra, Elodie Sandraz), Rachel Miller (best wishes for your future PhD; I trust that Elodie will find you a nice, warm volcano), Delaram Sadaghdar, Shelby Nelipovich, Jen Sun, Astrea Villarroel, Jake Shields, Megan Hayes
- Sydney: Rupert Shuttleworth, Nathania Astria Tan-Shuttleworth, YinLin Ooi, John Garland

Thanks to Tony Howard for sharing their knowledge and wisdom.

Thank you to Nicholas Carlini, Úlfar Erlingsson and the anonymous reviewers for helpful comments and suggestions on the shadow stacks and Oscar projects. These projects were supported by Intel through the ISTC for Secure Computing and by the AFOSR under MURI award FA9550-12-1-0040.

For the shadow stack project, we also thank Mathias Payer for comments, and the National Science Foundation for support under grant CCF-0424422.

For the Oscar project, we also thank David Fifield for comments, and gratefully acknowledge support from the Hewlett Foundation through the Center for Long-Term Cybersecurity, the National Science Foundation under grant NSF CNS-1514457, and BEARS.

A special thank you to Google for allowing Petros Maniatis to serve as my co-advisor.

Chapter 1

Introduction

1.1 Memory Safety

Computer systems have a finite amount of physical memory, from which allocations can be made, with a specific size and time-limited validity. Figure 1.1 illustrates how memory, spanning the space [s1, s3) during the time [t1, t3) can hypothetically be shared by objects `a`, `b`, `c` and `d`; for example, object `b` spans the space [s2, s3) during the time [t1, t2). Any memory outside of those space and time limits does not belong to object `b`, and accessing such memory via a pointer to `b` is a memory safety violation. While many modern languages such as Java and Python enforce memory safety, other languages – notably C and C++ – rely on the programmer to use memory correctly, leaving C/C++ programs prone to spatial and temporal memory errors.

1.1.1 Spatial Memory Safety

A spatial memory error occurs when code accesses memory outside of the space allocated for an object. Suppose that two variables, an array of characters `name` and an integer `age`, are located contiguously in memory (Figure 1.2).

It is intended that the array `name` be accessed only between its spatial bounds `name[0]` and `name[7]`; for example, the valid operations:

Figure 1.1: Objects subdividing physical memory across space and time.

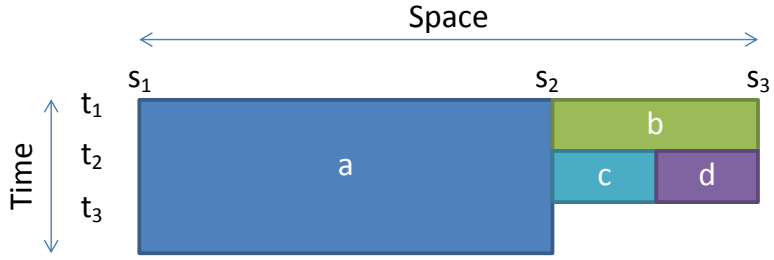


Figure 1.2: Two variables — an 8-byte character array and a 4-byte integer — spatially adjacent in memory.

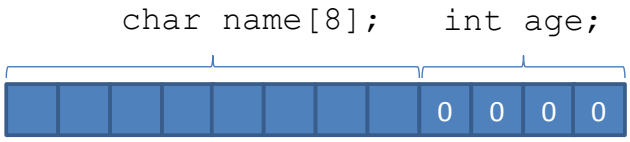


Figure 1.3: The two variables after `name` has been initialized with `Fred`.

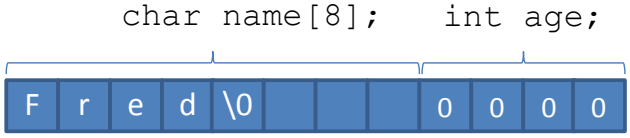
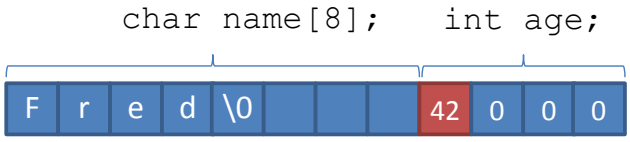


Figure 1.4: The two variables after setting `name[8]`, which is beyond the bounds of `name`.




```
name[0] = 'F';
name[1] = 'r';
name[2] = 'e';
name[3] = 'd';
name[4] = '\0'; // Null-terminator to indicate end of string
```

would result in Figure 1.3.

However, it is also possible to access beyond the end of the array, thereby using the variable `name` to access memory that belongs to `age` (Figure 1.4):

```
name[8] = '42';
```

Many functions in C/C++ can access beyond the bounds of an array; for example:

```
gets (name);
```

is given only the starting address of the string/array; in the absence of information on the size of the array, this function will store as many characters as typed in a line, even if this means writing past the end of the array. Similarly,

```
strcpy (src, name);
```

will copy every character of the string `src` into `name`, even if `src` exceeds the space allocated for `name`.

It is also possible to access beyond the start of the array:

```
name[-2] = 'x';
```

Figure 1.5: Setting `name[-2]`.

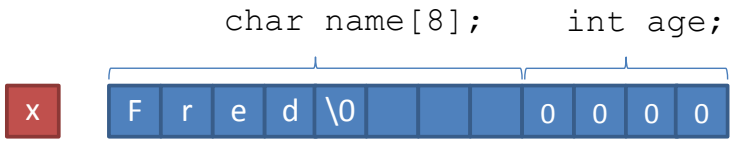
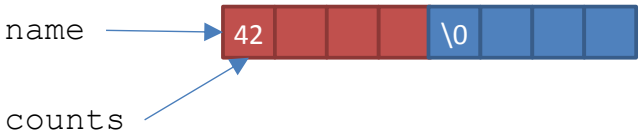


Figure 1.6: A pointer to `name`.

Figure 1.7: `counts` has been allocated the memory previously allocated to `name`.



which would overwrite whatever variable may coincidentally be located there (Figure 1.5):

1.1.2 Temporal Memory Safety

A temporal memory error occurs when code uses memory that was formerly allocated, but since `free()`'d (and therefore possibly reused for another allocation), i.e., an object is accessed outside of the time during which it was allocated.

Suppose we begin by allocating `name` from the heap (Figure 1.6):

```
1 char* name = allocate (8 * sizeof (char)); // Assume we set the name to
      Fred
```

Now suppose the space is freed via:

```
1 free (name);
```

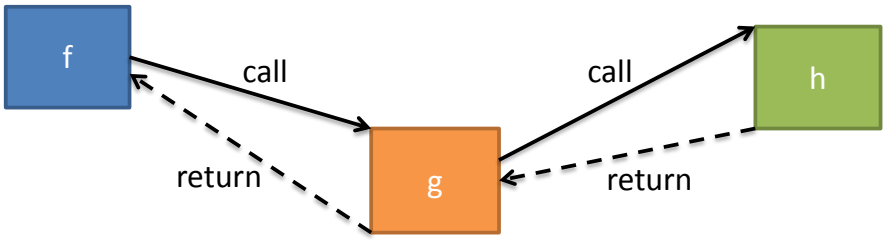
Note that the value of `name` has not changed. A future allocation may, by chance, reuse the same memory:

```
1 int* counts = allocate (2 * sizeof (int)); // Receives same address as 'name'
2 counts[0] = 42;
```

resulting in `name` and `counts` being aliased to the same memory (Figure 1.7).

If we printed the value of `name` as a character array following the write to `counts[0]`, it would no longer be `Fred`. Conversely, any updates to `name` would change the value of `counts`.

Figure 1.8: `f` calls `g` which calls `h`.



1.2 Direct defenses: enforcing memory safety

For spatial memory safety, it suffices to perform bounds checking; for example, to ensure that $0 \leq \text{index} \leq 7$ when accessing `name[index]` in Figure 1.2.

For temporal memory safety, a sufficient defense is not let users `free()` any memory, but rather to automatically `free()` when it is safe to do so. For example, after Figure 1.6, the run-time (as in Java) could detect that `name` still pointed to the object, and therefore would not allocate that memory to `counts` (or any other live allocation). If, however, there were no longer any pointers to that memory region (e.g., after `name = NULL;`), then its former object could be reused.

1.3 Partial defenses

Implementing memory safety is conceptually simple, but can have high run-time and/or space overhead and have compatibility issues with legacy code. Hence, many defenses have focused on limiting the damage from memory safety violations. In the above examples, we changed "non-control data", such as someone's age; these are roughly the inputs to functions. Attackers can often do more damage if they change the "control" data, i.e., influence what functions/code are run. One defense against these is control-flow integrity (CFI).

Consider a program where the function `f` may call `g`, which calls `h` (Figure 1.8). An attacker may manipulate the forward edges to call another function. Alternatively, they may manipulate the backward edges - making the functions return to code of their own choosing (rather than from `h` to `g`, and from `g` to `f`). The latter — "return oriented programming" is more common, and is Turing-complete [91].

1.4 This Work

Chapter 2, "The Performance Cost of Shadow Stacks and Stack Canaries", is concerned with efficient ways to protect the return address of functions on the stack, without hardware extensions.

Protecting return addresses is easy in principle — since they are (mostly) well-defined at run-time as the counterpart to `CALL` instructions — but protecting indirect jumps/calls is harder, since the correct target is hard to determine or even ambiguous. Recently, Carlini et al. [18] demonstrated that, even with the best possible control-flow integrity, "control flow bending" is still possible in some cases. Additionally, hardware advancements have made memory safety cheaper. Collectively, these developments make it worthwhile to revisit memory safety. We focus on heap temporal memory safety in Chapters 3 and 4.

Chapter 2

The Performance Cost of Shadow Stacks and Stack Canaries

Control flow defenses against ROP either use strict, expensive, but strong protection against redirected `RET` instructions with shadow stacks, or much faster but weaker protections without. In this work we study the inherent overheads of shadow stack schemes. We find that the overhead is roughly 10% for a traditional shadow stack. We then design a new scheme, the parallel shadow stack, and show that its performance cost is significantly less: 3.5%. Our measurements suggest it will not be easy to improve performance on current x86 processors further, due to inherent costs associated with `RET` and memory load/store instructions. We conclude with a discussion of the design decisions in our shadow stack instrumentation, and possible lighter-weight alternatives.

2.1 Introduction

One classic security exploit is to redirect the control flow by overwriting a return address stored on the stack [78]. Although various mitigations (e.g., No-Execute/Data Execution Prevention [70]) have made this attack and some simple refinements (e.g., return-to-libc [36]) infeasible, the current state of the art in exploitation — return-oriented programming (ROP [91]) — continues to depend on misusing `RET` instructions, this time by chaining together short sequences of instructions ("gadgets") each of which ends in a `RET`.[1]

These attacks could largely, in principle, be prevented using control-flow integrity (CFI) schemes [1], but CFI has not been widely adopted, in part due to its non-trivial overhead [106]. Some authors [108, 106] have proposed coarse-grained CFI policies, which greatly reduced the overhead, as well as making it possible to apply CFI directly to binaries, without requiring source code or recompilation. One notable relaxation is they adopt a more

[1]The generalizations of ROP — jump-oriented programming [13], or ROP without returns [21] — are not commonly used in practice [19] and will not be considered further.

permissive policy for `RET` instructions.[2], rather than tracking the return addresses precisely using a shadow stack (as had been proposed in Abadi et al.'s original formulation [1]) Unfortunately, such weaker policies were soon shown to be insecure [46, 33, 19]. Control-flow defenses against ROP either use strict, expensive, but strong protection against redirected RET instructions with shadow stacks or other dual-stack schemes such as allocation stacks [42], or much faster but weaker protections without. However, it is not clear whether the overhead seen in CFI with shadow stacks is inherent in the shadow-stack functionality, or an artifact of particular implementations. Since shadow stacks are indispensible to a strong CFI solution, one goal of this paper was to measure their inherent cost.

There is substantial literature on stand-alone shadow stacks. Some papers report low overheads, but each paper makes subtly different design decisions and/or does not use standard benchmarks (see Section 2.3), which makes it difficult to estimate the cost of adding a traditional shadow stack to coarse-grained CFI.

We do not consider schemes that have dual stacks, but which do not store the return address in a shadow stack (see Section 2.3). To our knowledge, they have only been implemented in recompilation-based schemes — thus negating the binary-rewritability benefits of coarse-grained CFI.

2.2 Background

2.2.1 Traditional Shadow Stacks

The purpose of a shadow stack is to protect return addresses saved on the stack from tampering. Figure 2.1 illustrates a traditional shadow stack, in a scenario where there are currently four nested function calls. In the main stack, each stack frame is shown with parameters, the return address, the saved frame pointer (EBP), and the local variables. In the traditional shadow stack, there is a shadow stack pointer (SSP) — which contains the address of the top of the shadow stack — and the shadow stack itself, which contains copies of the four return addresses.

In shadow stack schemes, when a function is called, the new return address is pushed onto the shadow stack.

When the function returns, it uses the return address stored on the shadow stack to ensure the integrity of the address where execution returns. This can be done by either **checking** that the return address on the main stack matches the copy on the shadow stack, or by **overwriting** the return address on the main stack with the copy on the shadow stack (equivalently, by indirectly jumping to the address stored on the shadow stack).

There are some implementation choices of where to save or check the return addresses. The return address can be saved before the `CALL` instruction ("Prologue #1" in Figure 2.2) or at the prologue of the called function ("Prologue #2"). The return address can be

[2]There are a variety of relaxed policies for forward-edges, such as allowing indirect calls/jumps to target any valid instruction boundary, or any location in the relocation table. [108]

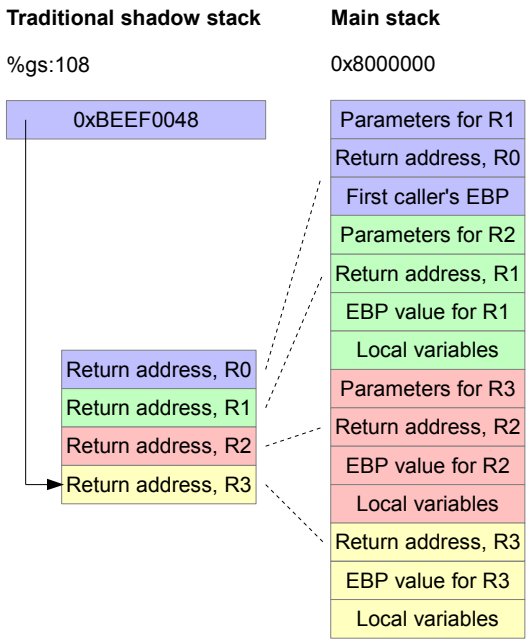


Figure 2.1: Traditional shadow stacks. Rx refers to Routine #x.

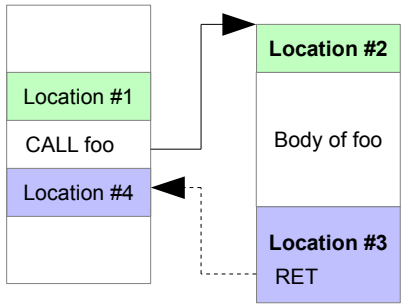


Figure 2.2: Possible locations for instrumentation.

checked/overwritten before the `RET` instruction ("Epilogue #1"); alternatively, if we ensure that every `RET` instruction can only return to a call-preceded location that is not an unintended instruction (this is the coarse-grained return policy of BinCFI [108]), we can check the return address at the return site ("Epilogue #2").

Typically, schemes that propose changes to the hardware (e.g., Ozdoganoglu et al. [81]) find it convenient to modify the `CALL` instruction to save the return address on the shadow stack (similar to "Prologue #1"), while binary-rewriting schemes often instrument function prologues and epilogues by replacing them with "trampolines" (indirect jumps) to the replacement code[6, 48, 76, 82, 86].

Figure 2.3 illustrates a sample prologue. We assume the shadow stack pointer (SSP) is

```
1 SUB $4, %gs:108   # Decrement shadow stack pointer (SSP)
2 MOV %gs:108, %eax # Copy SSP into %eax
3 MOV (%esp), %ecx  # Copy return address into
4 MOV %ecx, (%eax)  #      shadow stack via %ecx
```

Figure 2.3: Prologue for traditional shadow stack.

```
1 MOV %gs:108, %ecx # Copy SSP into %ecx
2 ADD $4, %gs:108   # Increment SSP
3 MOV (%ecx), %edx  # Copy return address from
4 MOV %edx, (%esp)  #      shadow stack via %edx
5 RET
```

Figure 2.4: Epilogue for traditional shadow stack (overwriting).

```
1 MOV %gs:108, %ecx # Copy shadow stack (SS) pointer into %ecx
2 ADD $4, %gs:108   # Increment SSP
3 MOV (%ecx), %edx  # Copy return address from SS into %edx
4 CMP %edx, (%esp)  # Compare return addresses from SS and main stack
5 JNZ abort
6 RET
7
8 abort:
9     HLT
```

Figure 2.5: Epilogue for traditional shadow stack (checking).

stored in an arbitrary memory location, `%gs:108`.[3] The code decrements the SSP [4], copies it into a scratch register (`%eax`), and then uses the SSP to copy the return address from the main stack into the shadow stack, using `%ecx` as another scratch register.

Figure 2.4 illustrates a sample overwriting-style epilogue. The code largely reverses the prologue: it copies the SSP into a scratch register, increments the SSP, then copies the return address from the shadow stack into the main stack, using `%edx` as another scratch register. The checking-style epilogue (Figure 2.5) is similar, but compares the return addresses on the main stack and shadow stack, aborting if they mismatch.

[3] Offset 108 from the segment register `%gs`; this is similar to how the stack canary is stored on Linux[49]

[4] We could also decrement the SSP at the end of the prologue, if we moved incrementing SSP to the start of the epilogue

Scenario	-fstack-protector-all				Parallel shadow stack (fixed offset)				Parallel shadow stack (randomized offset)				Protected traditional shadow stack			
	Check		Overwrite		Check		Overwrite		Check		Overwrite		Check		Overwrite	
	RA	Data	RA	Data	RA	Data	RA	Data	RA	Data	RA	Data	RA	Data	RA	Data
Contiguous writes, no info disclosure	Y	Y	N	N	Y	Y	Y	N	Y	Y	Y	N	Y	Y	Y	N
Contiguous writes, with info disclosure	N	N	N	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N
Arbitrary writes, no info disclosure	N	N	N	N	N	N	N	N	Y	N	Y	N	Y	N	Y	N
Arbitrary writes, with info disclosure	N	N	N	N	N	N	N	N	N	N	N	N	Y	N	Y	N

Table 2.1: Security properties of each mechanism.
RA = return address, Data = any data in the stack above the canary/return address of the current frame. Y = protected, N = vulnerable to overwrite.

2.2.2 Stack Canaries

Stack canaries are special values stored in stack frames between the return address and local variables. A contiguous stack buffer overflow would overwrite the stack canary, which is checked for integrity before the `RET`s of vulnerable functions [101].

Shadow stacks are sometimes argued to be a type of stack canary: instead of checking whether an added canary value has been corrupted, the return addresses (and sometimes the saved frame pointers) are used as canaries [82, 6]. For completeness, we investigated the overhead of stack canaries.

2.3 Related Work

Table 2.2 summarizes the overheads reported for various software-based shadow-stack schemes. Many of the papers use an older benchmark suite, SPEC2000; note that SPEC specifically cautions against comparing individual benchmarks between CPU2000 and CPU2006 [29]. We have omitted a number of studies where the shadow stack is a component of a security solution, for which we could not infer the cost of the shadow stack alone [85, 84, 79].

Table 2.2: Reported overheads of shadow stacks. Schemes are roughly sorted by the modifications involved.

Reference	Scheme	Modifications	Overhead on macro-benchmarks
		Compiler	

Table 2.2: Table 2.2: Reported overheads of shadow stacks. Schemes are roughly sorted by the modifications involved. (cont.)

Reference	Scheme	Modifications	Overhead on macro-benchmarks
Chiueh & Hsu [25]	Shadow stack (checking)	Compiler	Only macro-benchmarks are short-lived programs (0.63s real-time for their `ctags` benchmark, and <5s for `gcc`).
Szekeres et al. [98]	Shadow stack (?)	Compiler (LLVM plugin)	5% on SPEC2006.
Mashtizadeh et al. [66]	Misc	Compiler with ABI changes	45% on SPECint 2006 when using an optimization for leaf functions.[5]. Cost of stand-alone shadow stack is only shown in graph form; as an indication, for `xalancbmk`, it is 2.5x baseline for un-optimized.
		Assembler file processor	
Vendicator [100]	Shadow stack (checking)	Assembler file processor	Unknown
		Binary rewriting	
Prasad & Chiueh [86]	Shadow stack (checking)	Binary rewriting with trampolines	1–3% overhead on BIND, DHCP server, PowerPoint and Outlook Express.
Baratloo et al. [6]	Shadow stack (checking)	Binary rewriting with trampolines	9.5% for quicksort, which they deemed to be CPU-bound. They also measured imapd (network bound), xv (CPU and video bound), tar (I/O). All execution times were <6s.
Abadi et al.			

[5]Omitting `gcc` and `perlbench` due to compilation issues

Table 2.2: Table 2.2: Reported overheads of shadow stacks. Schemes are roughly sorted by the modifications involved. (cont.)

Reference	Scheme	Modifications	Overhead on macro-benchmarks
[1]	CFI + shadow stack (overwriting)	Binary rewriting with Vulcan	$\gg$ 5% on SPEC2000 for the shadow stack component; >50% for one benchmark.[6]
Gupta et al. [48]	Shadow stack (checking)	Binary rewriting with trampolines	No macro-benchmarks.
Park et al. [82]	Shadow stacks (checking and overwriting)	Binary rewriting with trampolines	Checking: 2.56% and 2.58% for `bzip2` and `gzip`. Overwriting: 1.56% and 1.7% respectively. Doesn't state whether this is compress or decompress.[7]
Corliss et al. [26]	Shadow stacks (checking)	Binary rewriting	Average not reported, but non-trivial (overheads exceeds 40% for some SPEC2000 benchmarks)
Nebenzahl et al. [76]	Shadow stack (checking)	Binary rewriting with trampolines	4.33% on `bzip2`, 4.36% on `gzip`, and 7.09% on `mcf` from SPEC2000
		Dynamic instrumentation	
Davi et al. [34]	Shadow stack (checking)	Pin tool	2.17x for SPECint2006, 1.41x for SPECfp. Run-time of Pin alone is 1.58x and 1.15x respectively.
Sinnadurai et al. [95]	Shadow stack (checking)	DynamicRIO	18.21% for SPECint 2000 on Linux; 24.82% (with compatibility issues) on Windows.

[6]CFI + ID check on returns cost 21%, while CFI + shadow stack (without ID check) cost 16%. But in some benchmarks (e.g., `crafty`), the shadow stack is cheaper than the ID check (roughly 45% vs. 18%). Hence, 5% is grossly underestimating the cost of a shadow stack.

[7]In our own experience, the overhead of instrumented compress is far higher than instrumented decompress.

Table 2.2: Table 2.2: Reported overheads of shadow stacks. Schemes are roughly sorted by the modifications involved. (cont.)

Reference	Scheme	Modifications	Overhead on macro-benchmarks
Zhang et al. [107]	General security instrumentation + shadow stack (checking)	PSI	18% overhead on a subset of SPEC2006.

There are also many hardware assisted schemes [26, 81, 53, 104, 63]; those papers all report low overheads on SPEC2000 (or a subset thereof), when using the SimpleScalar simulator. StackGhost [45] was a proposal for a shadow stack on SPARC. We chose to benchmark instrumentation schemes that could be deployed on today's hardware.

Ozdoganoglu et al. [81] observed that SPECint programs had higher overhead from instrumentation than SPECfp, which they attributed to the higher call frequencies of the integer benchmarks. They did not calculate a correlation, nor consider the percentage of memory loads and stores.

Corliss et al. [26] assumes that the stack pointer cannot be modified by an attacker. Such an assumption would remove the main weakness of our parallel shadow stack (compared to a traditional shadow stack); however, it is unrealistic given the increasing prevalence of stack pivots [40].

The choice of checking vs. overwriting the return address is similar to the "ensure, don't check" philosophy of SFI schemes [68, 102].

The memory-safety community, besides providing a somewhat heavy-weight solution to data- and control-flow integrity, has extensively studied how to implement *shadow memory*. In the parlance of AddressSanitizer [90], the address mapping used by traditional shadow stacks is similar to a single-level translation, while the parallel shadow stack is a direct offset (without scaling).

An ideal shadow stack would be protected from any writes by the attacker. Chiueh and Hsu's [25] Read-Only RAD accomplishes this through memory protection, albeit at a substantial overhead. Abadi et al.'s [1] protected shadow stack has much lower overhead than Chiueh and Hsu through the use of segmentation (which is not possible on 64-bit) and the security guarantees of CFI, though the overhead is still not trivial ($\gg 5\%$; see Table 2.2). While our shadow stacks are unprotected, Szekeres et al. [98] observe that even an unprotected shadow stack that *checks* for a match renders an attack "much harder", since an attacker would have to modify the return address in two distinct locations. With a shadow stack that *overwrites* the return address, an attacker would only have to modify the return address stored in the shadow stack, but this is somewhat harder than modifying the copy in the main stack (e.g., a contiguous buffer overflow would not suffice).

```
1 mov rax, [rsp]
2 mov fs:[rsp], rax

1 48 8b 04 24
2 64 48 89 04 24
```

```
1 mov rax, [rsp]
2 mov [rsp+0x1000000], rax

1 48 8b 04 24
2 48 89 84 24 00 00 00 01
```

Figure 2.6: Assembly and machine code of Return Flow Guard (left) and parallel shadow stack (right) prologues.

Some schemes use two stacks but do not duplicate the return address, hence we do not consider them to be shadow stacks, e.g., address space randomization (which uses their "shadow stack" to store buffer-type variables) [10] and XFI [42] (possibly with hardware support [16]). Importantly, due to the change in stack layout, they require significantly more code rewriting than shadow stacks. Xu et al. [104] have separated control and data stacks (essentially a shadow stack approach, but without the return address on the main stack). Their compiler implementation had up to 23% overhead on one of the SPECint 2000 benchmarks, and non-negligible overheads on most other benchmarks; they did not quote an average overhead. Kuznetsov et al. [61] have a "safe stack" that contains the return address, spilled registers, and other provably safe variables, and a separate unsafe stack. They benefit from improved locality of frequently used variables on the safe stack, thereby incurring negligible overhead on SPEC CPU2006, and can even improve performance in some cases. Dahn et al. [31] and Sidiroglou et al. [93] move stack-allocated buffers to the heap. Some non-x86/x64 architectures, such as Itanium [57], have a separate register stack.

2.3.1 Return Flow Guard

Microsoft's "Return Flow Guard" (RFG) [103] appeared in a Windows beta in October 2016. RFG is similar to parallel shadow stacks (to be discussed in Section 2.5) insofar as the "control stack" is placed at a fixed offset to the main stack, but they store the offset inside the `FS` segment register. Figure 2.6 compares the assembly and machine code of RFG and parallel shadow stacks, with the latter rewritten to highlight the similarities.

The disadvantages of RFG, compared to parallel shadow stacks, is that RFG requires use of a scratch register as well as the `FS` segment register. These disadvantages can be overcome by Microsoft, since they control the compiler and operating system.

RFG's use of the segment register does have advantages of being able to use a different offset per thread, and having a shorter instruction encoding. RFG may also have better information-hiding, by simply not exposing `FS` in regular application code (similar to the "safe region" in Code Pointer Integrity [61]), whereas parallel shadow stacks requires a randomized offset and non-readable code. It is unclear which scheme has lower overhead.

Unfortunately, RFG does have a "design-level bypass" [103] — though it is unclear what it is — resulting in RFG's removal. Due to RFG's similarities with parallel shadow stacks,

it is natural to be concerned whether parallel shadow stacks shares a similar vulnerability. Note, however, that Microsoft assumes a very strong threat model for RFG, while parallel shadow stacks are not claimed to be safe in the presence of information disclosure (Figure 2.1).

2.4 Challenges

In this section, we discuss two issues that make implementing shadow stacks securely challenging: time of check to time of use (TOCTTOU) vulnerabilities in multi-threaded programs, and deliberate mismatches between `CALL`s and `RET`s.

2.4.1 Time of Check to Time of Use Vulnerability

Epilogues that use the `RET` instruction in multi-threaded programs are vulnerable to time-of-check-to-time-of-use (TOCTTOU) attacks: the return address may be correct at the time of the shadow stack epilogue validation, but be modified by the attacker before the `RET` executes. This attack can be prevented by storing the return address inside a register (e.g., `ecx`), performing the validation on `ecx`, and converting the `RET` into `jmp *%ecx` [1].

A similar vulnerability exists for any shadow stack scheme that instruments the prologue, since between the `CALL` (when the correct return address is placed on the stack by the CPU) and the shadow stack prologue, the attacker can modify the return address. This can be avoided by instrumenting the `CALL` site, to compute and store the return address in the shadow stack [1, 84]. Some non-x86 architectures are immune, as they pass the return address using a "link register" [4].

Nonetheless, many shadow stack schemes (see Table 2.2) do instrument the prologue and epilogue (with a check performed before the `RET`); this may be because of its convenience for binary rewriting with trampolines, incremental deployability on a per-function basis, or the perceived performance impact of replacing `RET`s, since it is highly recommended to match `CALL`s with `RET`s [96, 55]. Zhang et al. [106] argue that the TOCTTOU vulnerability with their use of `RET` is difficult to exploit (due to the precise timing required), and outweighed by the benefits of return address prediction. BinCFI [108] goes to great lengths to maintain the `CALL-RET` matching despite TOCTTOU exploits, even adding extra stub function calls rather than using indirect jumps.

2.4.2 Mismatches Between `CALL`s and `RET`s

Perhaps the best known violation of `CALL-RET` matching is `setjmp/longjmp`, whereby a function may unwind multiple stack frames. For traditional shadow stacks, a typical solution (e.g., binary RAD [86]) is to pop the shadow stack until a match is found, or the shadow stack is empty (denoted with a sentinel value of zero). Figure 2.7 is our hand-compiled version of the C code from PSI [107].

```
1 MOV %gs:108, %ecx       # Copy shadow stack pointer (SSP) into %ecx
2 MOV (%esp), %edx        # Copy return address into %edx
3
4 non_match:
5     CMP $0, (%ecx)      # Is the shadow stack empty?
6     JZ abort
7     ADD $4, %ecx        # Increment cached copy of SSP
8     CMP %edx, -4(%ecx)  # Check return address
9     JNZ non_match       # Loop until match (or shadow stack empty)
10
11 MOV %ecx, %gs:108      # Synchronize SSP
12 RET
13
14 abort:
15     HLT
```

Figure 2.7: Epilogue for traditional shadow stack with checking and popping until a match.

This is already considerably more complicated than the vanilla traditional shadow stack epilogue, yet might not even be completely secure in obscure circumstances: if the same function is called multiple times before `longjmp`. This can be solved by storing both the return address and stack pointer [95, 81, 26].

2.5 Design

We devised parallel shadow stacks as a minimal, low-cost implementation of the principle of shadow stacks, albeit with some security trade-offs.

2.5.1 Parallel Shadow Stacks

As we will show later in Section 2.8, the traditional shadow stack has a non-trivial performance overhead. Our results indicate that the overhead comes mainly from the per-execution cost of the instrumentation we add, multiplied by the frequency of `RET` instructions in the program that we are protecting (we do not expect this to be a perfect predictor, due to e.g., dependencies resulting in pipeline stalls). Thus, an instrumentation with a lower execution cost would nearly directly translate into lower overhead.

We introduce a new variant on traditional shadow stacks, which we call a *parallel shadow stack*. The main idea is to place it at a fixed offset from the main stack, avoiding the overhead of maintaining the shadow stack pointer and copying it to/from memory. For example, in Figure 2.8, the shadow stack is `0x1000000` bytes above the main stack, and the return addresses in the main stack are parallel to the return addresses in the shadow stack. Single guard pages (e.g., a page marked non-present in the page table) at the top and bottom of the shadow stack protect it from contiguous buffer overflows.

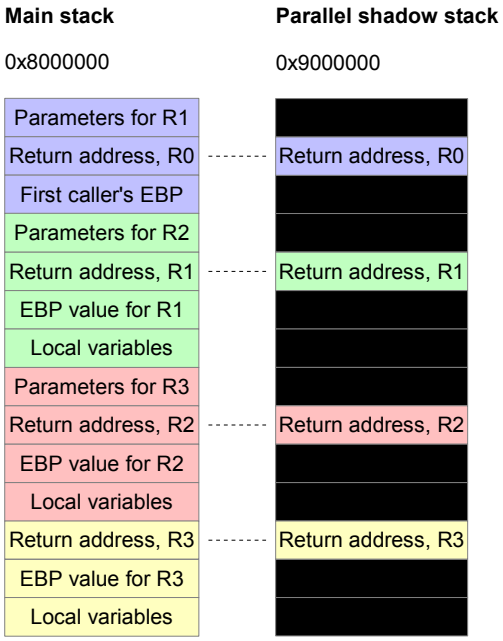


Figure 2.8: Parallel shadow stack. Rx refers to Routine #x. Compare to the traditional shadow stack of Figure 2.1.

```
1 POP 999996(%esp) # Copy return address to shadow stack
2 SUB $4, %esp     # Fix up stack pointer (undo POP)
```

Figure 2.9: Prologue for parallel shadow stack.

```
1 ADD $4, %esp       # Fix up stack pointer
2 PUSH 999996(%esp) # Copy from shadow stack
```

Figure 2.10: Epilogue for parallel shadow stack.

Our instrumentation for overwriting the return address can be as simple as adding two instructions to each uninstrumented prologue (Figure 2.9) and two instructions to each uninstrumented epilogue (Figure 2.10).

With peephole optimizations (see Appendix A.2.4), the net instruction count can be as few as one and *zero* instructions added to each prologue and epilogue, respectively. Note, however, that instruction count is not the sole, or even necessarily the major determinant of CPU overhead.

The instrumentation in Figures 2.9 and epilogue (Figure 2.10 does not clobber (overwrite) any registers and can be easily modified to preserve the flags (through replacing `SUB/ADD`

with `LEA`[8]), thus making it transparent to the rest of the program, without incurring the expense of saving/restoring any registers or EFLAGS.

All parallel shadow stacks, by definition, automatically handle any unusual changes to `%esp`: for example, when `longjmp` unwinds the stack, it also implicitly unwinds the shadow stack appropriately.

Some library functions may use `PUSH-RET` to jump to an arbitrary address. This is sometimes used to implement `longjmp` or `setcontext`. [87] In the general case where the "return" address does not match a function call, both the traditional and parallel shadow stacks would identify it as anomalous.

There are two main disadvantages compared to a traditional shadow stack: memory/-cache utilization, and the threat of modifying the shadow stack pointer.

Return addresses on the main stack are often separated by function parameters and local variables. This means that return addresses on the parallel shadow stack will likewise not be contiguous, and each return address may even use up an entire cache line; the overhead of this depends on the calling patterns of the program. The gaps in parallel shadow stacks also mean they generally use more memory than a traditional shadow stack, but as the stack is small (on our test system, the maximum stack size is 8MB`ulimit -s`) in comparison to the heap (which can use up to the 12GB of RAM on our system), and the shadow stack pages do not even consume physical memory until they are first accessed (i.e., the maximum stack size is an upper-bound, not a lower-bound, on the shadow stack size), we do not consider this a major limitation.

Due to the shadow stack pointer synchronization, an attacker who is able to pivot the stack (modify the stack pointer to point elsewhere) can choose any return address in the call stack. For example, suppose function `f` called `g` which calls `h`. The call stack contains `[f,g]` (specific instruction addresses in the functions omitted for clarity), which means the correct return address will be `g`. An attacker could change the stack pointer (and, implicitly, the shadow stack pointer) so that, at the point of the return instruction, the return address will be `f`. Furthermore, if we do not "zero out" (or corrupt) old shadow stack frame entries, then the attacker could even choose "expired" return addresses. Suppose function `h` returns correctly, which means the next return address is `f`. However, the call stack is still initialized with `[f,g]`, and an attacker could change the stack pointer (and shadow stack pointer) to use the return address `g`. A more insidious attack is to change `%esp` such that the parallel shadow stack region — say, `0xa0000000(%esp)` — is under the attacker's control; we can prevent this, at the cost of limiting the address space, by ensuring that `0xa0000000(%esp)` is mapped to the process address space only for valid values of `%esp`.

[8]This replacement can affect runtime performance, as it is not guaranteed that CPU execution units can all interchangeably be used for `SUB/ADD` and `LEA`.[43]

2.5.2 Security Benefits

We consider four threat models, based on the attacker's ability to write to the stack and to read important secrets.

The weakest write vulnerability we consider is a contiguous stack buffer overflow. For example, the vulnerable code may contain code:

```
1 strcpy (attacker_controlled, buffer_on_stack);
```

which copies an attacker-controlled input into a buffer on the stack, without any bounds checks; the input can be crafted to exceed the length of the buffer on the stack, thereby overwriting other stack variables, including the return address.

A stronger write vulnerability is where the attacker can perform arbitrary (including indexed) writes into the stack (e.g., repeated uses of `buffer_on_stack[i] = s`, where `i` and `s` are under their control).

The write vulnerabilities above do not, in general, convey the ability to read the stack contents — such as the stack canary value or return value[9], although we assume that attackers always know the general stack layout. We further bifurcate the two write vulnerability cases above based on whether the attacker has information disclosure that allows them to read the stack contents or infer the fixed shadow stack offset, though in all cases we assume the vulnerability is insufficient to defeat randomization.

Figure 2.1 summarizes, for each of these threat models, whether each mechanism can protect the return address and/or any data stored above the canary (or return address).

All these mechanisms restrict the use of gadgets that end with an instrumented `RET`, and all the parallel shadow stacks provide some limited protection against large-scale stack pivots (since they will write to the shadow stack region).

2.6 Aims

2.6.1 Research Questions

Our overarching research question is: *what are the performance costs of using a shadow stack, as seems to be necessary for security when using CFI schemes?* It is obvious that there is greater than zero overhead since it requires additional operations; our aim is to *quantify* it.

To address this question, we evaluated the overhead of: a traditional shadow stack; a no-frills parallel shadow stack; checking vs. overwriting the return address; zeroing out expired shadow stack entries (for a parallel shadow stack); `-fstack-protector-all` (which adds stack canaries to *every* function, instead of a shadow stack for every call); and replacing `RET`s with indirect jumps (to avoid TOCTTOU; see Section 2.4.1).

[9] In the limited cases where a program restarts with an identical memory layout after a crash — such as a web server that `fork()`s – it is possible to convert the write vulnerability into a read vulnerability, by repeatedly overwriting individual bytes and observing if the program executes correctly. [12]

The overhead can be made arbitrarily low by constructing test programs that perform a significant amount of computation for every function call; conversely, the overhead would be artifically high if we instrumented empty functions [86, 25]. To provide a meaningful estimate of the overhead, it is important to use a standardized benchmark suite.

2.6.2 Assumptions

We assume that we have a tool similar to BinCFI [108] that 1) does not have access to the source code; 2) but can identify the prologue and `RET`s of every function (at the assembly level); and can insert code without the need for additional trampolines (cf., Prasad and Chiueh [86]).

Compiler-based shadow stack schemes generally fulfill condition 2 but not 1, and vice-versa for binary-rewriting based shadow stack schemes. In contrast, by leveraging the address translation that is already required for BinCFI, adding a shadow stack to CFI does not require more trampolines.

Our assumptions are similar to Stack Shield [100], though that is intended to be deployed as an assembly preprocessor in the ordinary build process.

2.7 Method

We ran SPEC CPU2006[10], excluding any Fortran programs, on Ubuntu 12.04, running on a Dell Precision T5500 (Intel Xeon X5680, with 12GB of RAM). All code was compiled with gcc 4.6.3, using the standard configuration files (`Example-linux64-amd64-gcc43+.cfg`) and `Example-linux64-ia32-gcc43+.cfg`), with the exception that we disabled stack canaries entirely (i.e., `-fno-stack-protector`) since they are mostly redundant when a shadow stack is available. We tested both 32-bit and 64-bit, as they are the dominant modes; 32-bit performance may differ from 64-bit, due to the larger integers (which may affect cache/memory pressure) and increased number of general-purpose registers (R8 to R15) for the latter. We chose the `-O2` optimization level, the default setting for SPEC CPU, plus the more aggressive `-O3`. With `-O3`, inlining reduces the number of function calls (and thereby the amount of instrumentation required), though the function bodies are also faster (which makes the prologue/epilogue instrumentation relatively more expensive). Since both are reasonable optimization options, we wished to empirically compare their overhead.

We also tested `apache httpd 2.4.10`[11], using `apachebench`[12] on the same machine. We chose this as a more realistic scenario, whose performance may not be entirely CPU-bound.

As per Tice et al. [99], we disabled Turbo Boost (dynamic CPU frequency scaling) and ASLR to reduce variance in the run-times. Additionally, we observed that the SPEC benchmarks tended to run more slowly if the system had been in use for a prolonged period,

[10] `https://www.spec.org/cpu2006/`

[11] `https://httpd.apache.org/`

[12] `https://httpd.apache.org/docs/2.4/programs/ab.html`

possibly due to disk-caching effects. To avoid these carryover effects, we rebooted the system before each batch of benchmarks.

We used the following instrumentation (see Appendix A.2.1): a traditional shadow stack that checks the return address (for compatibility with `longjmp`, it pops the shadow stack until a match with the return address); a parallel shadow stack that checks the return address; a parallel shadow stack that overwrites the return address, zeroing out expired shadow stack entries; a parallel shadow stack that overwrites the return address, without zeroing; `-fstack-protector-all`, i.e., stack canaries applied to every function.

We tested multiple versions of each instrumentation, and selected the code that performed best in a pilot study.

We tried both `RET`s and indirect jumps for each of the instrumention schemes except `-fstack-protector-all`. We did not implement a traditional shadow stack (overwriting), due to its inability to protect programs using `longjmp`.

Our parallel shadow stacks used a fixed offset, for ease of implementation. However, the overhead should be similar when using a random offset, except for the time required to rewrite the offsets at load-time, which should be negligible: the number of offsets that need to be rewritten is equal to the number of function prologues and `RET`s. This is less than for relocations, which must relocate both data and functions.

In general, our implementation is intended to provide a reasonable estimate of the overhead, not production use. We discuss deployment issues in Section 2.9.3.

2.7.1 Implementation Details

We make no claim that our implementation will be robust enough to deploy widely. Nonetheless, since much of the overhead is due to inherent memory loads/stores (see Section 2.8), we believe we have implemented each of these schemes in sufficient detail to let us measure their performance overhead on SPEC CPU.

For C programs, we call `setupShadowStack` at the beginning of `main`. The `setupShadowStack` function allocates a memory region at a fixed offset from the stack (for the parallel shadow stack) or at a random location (for the traditional shadow stack). For the latter case, we copy this location into `%gs:108`.

For C++ programs, we identified which initialization function would run first, by using `gdb`: the function would crash in the prologue, as the instrumentation would attempt to write to an unallocated shadow stack region. We then prepended our own shadow stack class and instance in the same file, thus again ensuring that the shadow stack is set up prior to other function calls.

This means that we cannot instrument the prologue of functions that run before the shadow stack is set up (e.g., `main` and `setupShadowStack` for C programs). It is possible to modify the compiler to create high priority constructor init functions [99], but this would not affect the overhead since the uninstrumented functions are only called once.

The SPEC CPU build process compiles and assembles individual source files into separate object files. We wrote a wrapper script to compile each source file, then instrument

the prologues (conveniently denoted by gcc as `.cfi_startproc`) and `RET`s in the assembly, before assembling into the expected object file. This simulates the capabilities available (and overhead obtained) of a binary CFI rewriter that has access to the (dis-)assembly but not source code. It is possible, but laborious, to add `setupShadowStack` at the assembly level; we "cheated" by adding it at the source-code level, since this would not substantially affect the overhead.

Since we are instrumenting the prologue for ease of implementation, the instrumented prologue has a TOCTTOU vulnerability in the presence of concurrency (e.g., multi-threaded programs). One could avoid the TOCTTOU vulnerability by instrumenting the `CALL` instructions instead of function prologues. We expect this would have similar performance, but we have not implemented it.

2.8 Results

2.8.1 Cost of Instrumentation Schemes

Figures 2.11, 2.12, 2.13, and 2.14 show the overheads for x86 -O3, x86 -O2, x64 -O3, and x64 -O2, based on the medians of at least 9 runs per benchmark. We do not expect our instrumentation to make any programs faster; any negative overheads are likely to be experimental noise.

As the overheads (though not baseline times) were roughly the time, we we will only discuss the x86 -O3 results (Figure 2.11, left), as they are marginally lower, and therefore provide a lower bound.

Overheads are only *estimates*; they do not meet certain technical requirements for "reportable" SPEC CPU2006 results (for example, modifying the source files to add the `setupShadowStack` function violates the requirements), even though we can reliably reproduce these results (minima are mostly within 1% of the medians, and standard deviations are generally low). Figures 2.15, 2.16, 2.17, and 2.18 show the overheads for x86 -O3, x86 -O2, x64 -O3, and x64 -O2, based on the minima. Figures 2.19, 2.20, 2.21, and 2.22 show the standard deviations (in seconds) of the runtimes.

Two of the programs (`perlbench`, `gcc`) did not work with the traditional shadow stack instrumentation (for x86 with -O2 or -O3); we suspect it may be related to `fork`ing and our (mis-)use of `%gs:108` to store the shadow stack pointer (i.e., an artifact of our implementation, rather than a fundamental limitation of traditional shadow stacks).[13] To provide a fair comparison with the other schemes, we calculated the overhead of each scheme, with and without these two programs.

On SPEC CPU, excluding Fortran, `perlbench` and `gcc`, the average overhead of each scheme was as follows: a traditional shadow stack cost 9.69% overhead in CPU time; a no-frills parallel shadow stack cost 3.51%; checking the return address cost 0.8% extra, compared

[13]Interestingly, Mashtizadeh et al. [66] omitted (only) these two benchmarks; they reported that they could not compile them with the vanilla `GCC` or `clang` compilers.

to the no-frills parallel shadow stack; zeroing out expired shadow stack entries (for a parallel shadow stack) cost 0.16%; stack canaries cost 2.54%. See the bottom row of Figure 2.11 for details.

For `apache`, the parallel shadow stack (overwriting, no zeroing out) had 2.73% overhead. Had our test been network-bound or I/O-bound, we would expect the CPU overhead to be even lower.

Replacing `RET`s with indirect jumps incurs much higher overhead, except for the checking version of the parallel shadow stack, which had comparable overhead. See Figure 2.11 (right).

2.9 Discussion

2.9.1 Determinants of overhead

As a first-order approximation, we expected the overhead of the instrumentation to depend on the frequency of function calls/returns (to avoid collinearity, we only consider `RET`s); while Abadi et al.[1] reported that their overhead was "not simply correlated with the frequency of executed computed control-flow transfers", their CFI instrumentation is much more extensive than a shadow stack. Our hypothesis was supported by the data (correlation coefficient $r = .73$, which is very high). Figure 2.23 shows the overhead of the parallel shadow stack (overwriting, zeroing out) compared to the dynamic counts of `RET` instructions (from Isen and John [56]). We excluded `dealII` from our analyses, as it is an outlier with 5.3% `RET`s (over twice as many as any other program).

Since our instrumentation uses cache and memory bandwidth, we hypothesized that the overhead would depend on the percentage of load and store instructions multiplied by the percentage of `RET` instructions. For example, we would expect that a program with few `RET` instructions but many load/store instructions would still have low overhead. Using data from Bird et al. [11], a linear regression of the form:

$$\begin{aligned} \text{Overhead} = {} & 10.184 \times (\%\ \text{RETs}) \\ & +(-38.697) \times (\%\ \text{RETs} \times \%\ \text{loads}) \\ & +74.822 \times (\%\ \text{RETs} \times \%\ \text{stores}) \\ & +0.002 \end{aligned} \tag{2.1}$$

had a correlation of 0.86. This is a surprisingly high correlation, considering the complexity of modern CPUs.

Interestingly, our regression shows that the percentage of loads has a *negative* coefficient. We interpret this to mean that stores are expensive, but once a value has been stored, it is very cheap to load it due to caching.

Although correlation does not imply causation, we believe simple causality is the most parsimonious explanation.

The traditional shadow stack likely has higher overhead than the parallel shadow stack because of the extra memory transfer instructions needed for additional scratch registers and

x86, -O3	With RET in epilogues						With indirect jumps instead of RET in epilogues			
	Baseline	Traditional shadow stack	Parallel shadow stack			-fstack -protector -all	Traditional shadow stack	Parallel shadow stack		
			Checking	Overwriting, zeroing out	Overwriting, no zeroing			Checking	Overwriting, zeroing out	Overwriting, no zeroing
400.perlbench	354		9.2%	7.3%	6.7%	4.2%		7.9%	8.6%	5.5%
401.bzip2	585	2.2%	1.9%	1.6%	1.7%	-2.8%	8.2%	2.7%	2.9%	2.5%
403.gcc	300		4.9%	4.5%	4.2%	3.9%		7.8%	9.7%	7.2%
429.mcf	211	6.2%	0.0%	-0.1%	-0.1%	-0.1%	7.6%	-0.1%	1.5%	0.0%
445.gobmk	449	12.7%	5.1%	5.0%	4.0%	5.0%	26.3%	10.5%	10.9%	9.2%
456.hmmer	516	0.2%	0.0%	-0.1%	0.0%	-0.2%	4.3%	-0.1%	0.9%	0.0%
458.sjeng	518	16.9%	5.2%	5.0%	4.4%	2.0%	28.6%	9.8%	8.8%	8.5%
462.libquantum	669	-1.7%	-0.5%	-0.7%	-0.6%	3.7%	0.2%	-1.2%	-0.8%	-1.0%
464.h264ref	716	19.5%	8.7%	6.4%	6.4%	3.2%	23.2%	8.0%	8.2%	5.1%
471.omnetpp	319	25.2%	7.3%	11.0%	9.2%	5.1%	33.7%	6.3%	16.0%	6.3%
473.astar	453	7.2%	3.5%	2.5%	3.9%	0.8%	9.1%	1.2%	2.7%	1.0%
483.xalancbmk	221	33.1%	16.6%	12.5%	13.7%	9.5%	52.5%	19.6%	21.4%	17.6%
433.milc	565	4.6%	1.9%	1.9%	2.2%	1.6%	4.8%	1.0%	2.7%	1.5%
444.namd	513	-0.1%	0.0%	0.0%	0.2%	0.1%	3.1%	-0.1%	0.9%	-0.1%
447.dealII	360	16.2%	9.4%	7.3%	7.0%	5.1%	18.8%	7.5%	5.6%	5.3%
450.soplex	281	0.2%	1.1%	-0.5%	-0.1%	-0.6%	2.1%	0.1%	1.9%	0.4%
453.povray	218	29.3%	13.0%	11.0%	9.7%	10.5%	31.0%	11.6%	11.1%	8.7%
470.lbm	416	0.0%	-0.3%	-0.4%	-1.1%	0.4%	-0.9%	-1.1%	-0.1%	-0.3%
482.sphinx3	471	1.9%	2.5%	1.5%	0.6%	0.8%	7.2%	0.4%	2.8%	3.1%
SPECint: all benchmarks	5312		5.1%	4.5%	4.4%	2.8%		5.9%	7.4%	5.0%
- without perl or gcc	4657	11.6%	4.7%	4.2%	4.2%	2.6%	18.4%	5.5%	7.0%	4.8%
SPECfp: except Fortran	2823	7.0%	3.8%	2.9%	2.6%	2.5%	9.0%	2.7%	3.5%	2.6%
SPEC CPU: except Fortran	8135		4.6%	3.9%	3.7%	2.7%		4.7%	5.9%	4.1%
- and without perl or gcc	7480	9.7%	4.3%	3.7%	3.5%	2.5%	14.4%	4.3%	5.6%	3.9%

Figure 2.11: Benchmarks for x86, -O3 with `RET`s (left) or indirect jumps (right) in the epilogues. Baseline runtimes are in seconds. Highest overheads are in red, and lowest overheads are in green. SPEC overheads are geometric means of median individual benchmark overheads.

x86, -O2	Baseline	With RET in epilogues					With indirect jumps instead of RET in epilogues				
		Traditional shadow stack	Parallel shadow stack			-fstack -protector -all	Traditional shadow stack	Parallel shadow stack			
			Checking	Overwriting, zeroing out	Overwriting, no zeroing			Checking	Overwriting, zeroing out	Overwriting, no zeroing	
400.perlbench	355		8.7%	7.9%	9.5%	5.1%		9.0%	7.0%	6.9%	
401.bzip2	599	2.3%	1.8%	1.3%	1.5%	-2.7%	4.6%	3.0%	2.5%	2.8%	
403.gcc	320		6.2%	4.5%	4.7%	3.6%		8.7%	8.1%	7.2%	
429.mcf	214	6.8%	0.5%	0.5%	0.2%	0.2%	6.8%	0.5%	0.6%	0.4%	
445.gobmk	461	17.3%	4.3%	4.8%	4.1%	3.4%	25.5%	10.7%	9.2%	9.7%	
456.hmmer	517	0.4%	0.2%	0.1%	0.2%	0.0%	0.4%	0.1%	0.3%	0.3%	
458.sjeng	542	16.1%	4.6%	3.5%	1.9%	3.0%	23.7%	9.1%	8.4%	7.9%	
462.libquantum	681	-0.1%	0.1%	0.1%	0.7%	-0.2%	0.6%	-0.2%	0.1%	0.3%	
464.h264ref	733	19.9%	8.3%	6.2%	6.0%	3.7%	23.4%	7.4%	6.2%	4.9%	
471.omnetpp	324	28.0%	9.0%	13.2%	12.5%	8.6%	29.2%	10.5%	14.1%	9.4%	
473.astar	475	8.8%	5.1%	1.9%	3.8%	0.0%	9.2%	2.6%	0.9%	0.7%	
483.xalancbmk	238	41.2%	16.8%	13.3%	11.9%	10.5%	47.7%	15.7%	17.4%	13.4%	
433.milc	583	4.6%	1.0%	1.9%	1.5%	1.1%	4.0%	0.6%	0.7%	1.0%	
444.namd	518	-0.3%	-0.1%	-0.2%	-0.1%	-0.1%	-0.1%	-0.2%	-0.2%	-0.2%	
447.dealII	392	14.1%	8.1%	5.2%	5.8%	5.5%	14.5%	6.0%	3.8%	4.1%	
450.soplex	293	0.2%	1.4%	0.8%	0.1%	-1.3%	2.4%	0.7%	-1.1%	0.1%	
453.povray	235	35.6%	13.2%	12.7%	11.5%	10.3%	37.1%	11.7%	9.8%	9.2%	
470.lbm	416	0.0%	-0.2%	-0.4%	-0.5%	-0.2%	-0.2%	-1.2%	-0.3%	-0.4%	
482.sphinx3	487	4.0%	-0.1%	-3.0%	-2.2%	-3.0%	-0.5%	-3.0%	-2.0%	-3.0%	
SPECint: all benchmarks	5461		5.4%	4.7%	4.7%	2.9%		6.3%	6.1%	5.2%	
- without perl or gcc	4785	13.4%	5.0%	4.4%	4.2%	2.6%	16.3%	5.8%	5.8%	4.9%	
SPECfp: except Fortran	2924	7.7%	3.2%	2.3%	2.2%	1.7%	7.5%	2.0%	1.5%	1.5%	
SPEC CPU: except Fortran	8384		4.6%	3.8%	3.8%	2.4%		4.7%	4.4%	3.8%	
- and without perl or gcc	7709	11.0%	4.2%	3.5%	3.4%	2.2%	12.6%	4.2%	4.0%	3.5%	

Figure 2.12: Benchmarks for x86, -O2 with `RET`s (left) or indirect jumps (right) in the epilogues. Baseline runtimes are in seconds. SPEC overheads are geometric means of median individual benchmark overheads.

x64, -O3	With RET in epilogues						With indirect jumps instead of RET in epilogues			
	Baseline	Traditional shadow stack	Parallel shadow stack			-fstack -protector -all	Traditional shadow stack	Parallel shadow stack		
			Checking	Overwriting, zeroing out	Overwriting, no zeroing			Checking	Overwriting, zeroing out	Overwriting, no zeroing
400.perlbench	344		8.7%	9.0%	7.8%	8.0%		9.8%	8.9%	7.8%
401.bzip2	502	0.7%	-0.9%	0.8%	-1.2%	-1.2%	2.7%	2.3%	-0.1%	-1.2%
403.gcc	319		4.3%	4.0%	3.7%	4.2%		8.8%	7.7%	3.6%
429.mcf	282	3.7%	-0.4%	-0.2%	-0.4%	-0.2%	3.6%	-0.1%	-0.2%	-0.4%
445.gobmk	438	12.2%	4.7%	5.1%	3.7%	4.4%	21.2%	11.2%	10.9%	3.8%
456.hmmer	412	0.2%	0.0%	0.0%	0.0%	0.1%	0.3%	1.2%	0.0%	0.0%
458.sjeng	493	16.0%	4.3%	5.5%	4.3%	3.8%	18.6%	6.8%	6.7%	4.2%
462.libquantum	489	1.4%	1.3%	-0.7%	0.5%	0.8%	1.4%	0.8%	-0.2%	1.0%
464.h264ref	585	25.9%	11.1%	9.3%	9.5%	7.2%	33.2%	12.2%	11.6%	9.5%
471.omnetpp	374		11.0%	14.0%	9.5%	10.2%		10.4%	9.2%	9.5%
473.astar	404	10.7%	4.9%	4.4%	2.9%	3.6%	10.6%	4.1%	3.8%	3.8%
483.xalancbmk	244		8.7%	6.3%	10.2%	10.6%		11.8%	10.3%	10.8%
433.milc	527		1.2%	1.2%	1.2%	0.8%		6.0%	1.1%	1.2%
444.namd	437	0.0%	0.1%	0.1%	0.1%	0.1%	0.1%	0.7%	0.1%	0.1%
447.dealII	338	24.6%	10.5%	7.6%	8.0%	4.9%	26.8%	8.3%	5.5%	8.0%
450.soplex	277		1.9%	1.8%	1.7%	-0.5%		1.4%	1.2%	-0.6%
453.povray	203	36.3%	10.2%	9.3%	8.3%	6.3%	37.6%	9.4%	6.8%	7.5%
470.lbm	400	-0.1%	-0.1%	-1.3%	-0.1%	0.1%	-0.2%	0.0%	-0.3%	-0.2%
482.sphinx3	586	3.5%	1.8%	1.0%	1.3%	1.1%	4.3%	1.3%	1.7%	2.1%
SPECint: all benchmarks	4886		4.7%	4.7%	4.1%	4.2%		6.5%	5.6%	4.3%
- without perl or gcc	4223	8.5%	4.4%	4.4%	3.8%	3.9%	10.9%	6.0%	5.1%	4.0%
SPECfp: except Fortran	2768	11.9%	3.6%	2.7%	2.9%	1.8%	12.7%	3.8%	2.3%	2.5%
SPEC CPU: except Fortran	7654		4.3%	4.0%	3.7%	3.3%		5.5%	4.4%	3.6%
- and without perl or gcc	6991	9.8%	4.0%	3.7%	3.4%	3.0%	11.6%	5.1%	3.9%	3.4%

Figure 2.13: Benchmarks for x64, -O3 with `RET`s (left) or indirect jumps (right) in the epilogues. Baseline runtimes are in seconds. SPEC overheads are geometric means of median individual benchmark overheads.

x64, -O2	Baseline	With RET in epilogues					With indirect jumps instead of RET in epilogues			
		Traditional shadow stack	Parallel shadow stack			-fstack -protector -all	Traditional shadow stack	Parallel shadow stack		
			Checking	Overwriting, zeroing out	Overwriting, no zeroing			Checking	Overwriting, zeroing out	Overwriting, no zeroing
400.perlbench	353		9.2%	7.8%	8.1%	3.0%		8.4%	6.6%	6.7%
401.bzip2	492	3.0%	3.1%	1.1%	0.7%	0.5%	6.8%	3.1%	2.6%	2.8%
403.gcc	330		4.0%	3.8%	3.4%	4.1%		7.2%	6.6%	7.3%
429.mcf	283		0.0%	0.0%	0.2%	0.3%		-0.1%	-0.2%	0.3%
445.gobmk	444	16.4%	5.4%	5.5%	4.2%	4.5%	25.9%	12.3%	12.1%	11.0%
456.hmmer	429	0.3%	0.0%	0.0%	0.0%	0.1%	0.4%	0.0%	0.1%	0.1%
458.sjeng	519	17.6%	5.2%	4.0%	3.8%	4.2%	24.1%	7.9%	7.4%	6.4%
462.libquantum	496	-1.0%	-1.0%	-1.0%	0.2%	0.6%	-0.3%	-1.7%	-0.7%	-0.9%
464.h264ref	609	26.2%	9.8%	8.4%	8.6%	7.3%	29.5%	10.1%	7.9%	7.7%
471.omnetpp	375	32.1%	12.5%	14.0%	11.5%	9.2%	33.8%	13.7%	10.5%	13.0%
473.astar	440	11.7%	6.2%	4.0%	4.7%	4.0%	10.7%	3.1%	1.9%	2.3%
483.xalancbmk	251		10.9%	13.8%	10.0%	15.2%		13.2%	16.5%	12.6%
433.milc	532		0.8%	0.8%	0.8%	1.5%		0.9%	0.8%	0.9%
444.namd	437	0.0%	0.0%	0.0%	0.0%	0.1%	0.0%	0.0%	0.0%	0.0%
447.dealII	351	29.2%	10.8%	9.3%	8.7%	6.6%	33.1%	9.5%	6.3%	6.3%
450.soplex	280		0.8%	0.5%	1.1%	-1.4%		0.9%	1.1%	1.1%
453.povray	203	47.8%	15.7%	14.5%	13.6%	12.7%	50.2%	13.8%	12.9%	10.9%
470.lbm	401	-0.2%	-0.4%	-0.4%	-0.3%	-0.4%	-0.3%	-0.5%	-0.3%	-0.3%
482.sphinx3	592	1.8%	1.0%	0.5%	0.8%	0.8%	2.6%	0.1%	0.9%	0.3%
SPECint: all benchmarks	5021		5.3%	5.0%	4.6%	4.3%		6.3%	5.8%	5.7%
- without perl or gcc	4337	12.7%	5.1%	4.9%	4.3%	4.5%	15.7%	6.0%	5.7%	5.4%
SPECfp: except Fortran	2797	14.2%	3.9%	3.5%	3.4%	2.7%	15.4%	3.4%	3.0%	2.7%
SPEC CPU: except Fortran	7818		4.8%	4.4%	4.1%	3.7%		5.2%	4.8%	4.6%
- and without perl or gcc	7134	13.3%	4.6%	4.3%	3.9%	3.8%	15.6%	4.9%	4.6%	4.3%

Figure 2.14: Benchmarks for x64, -O2 with `RET`s (left) or indirect jumps (right) in the epilogues. Baseline runtimes are in seconds. SPEC overheads are geometric means of median individual benchmark overheads.

x86, -O3	Baseline	With RET in epilogues					With indirect jumps instead of RET in epilogues				
		Traditional shadow stack	Parallel shadow stack			-fstack -protector -all	Traditional shadow stack	Parallel shadow stack			
			Checking	Overwriting, zeroing out	Overwriting, no zeroing			Checking	Overwriting, zeroing out	Overwriting, no zeroing	
400.perlbench	354		9.0%	7.1%	6.9%	4.3%		7.7%	7.6%	5.3%	
401.bzip2	585	2.1%	1.9%	1.6%	1.7%	-2.9%	5.9%	2.7%	2.6%	2.4%	
403.gcc	299		4.9%	4.7%	4.4%	3.3%		7.9%	9.2%	7.2%	
429.mcf	211	6.1%	-0.1%	-0.2%	-0.2%	-0.1%	6.9%	-0.2%	0.1%	-0.1%	
445.gobmk	449	12.8%	5.0%	4.9%	4.0%	5.1%	23.2%	10.6%	10.4%	9.1%	
456.hmmer	516	0.1%	0.0%	-0.1%	0.0%	-0.2%	1.5%	-0.1%	0.2%	0.0%	
458.sjeng	518	17.0%	5.2%	5.0%	4.4%	2.0%	26.7%	9.9%	8.2%	8.5%	
462.libquantum	666	-1.5%	-0.3%	-0.6%	-0.7%	4.0%	0.2%	-1.1%	-0.7%	-1.0%	
464.h264ref	715	19.5%	8.5%	6.4%	6.4%	3.2%	23.1%	8.0%	6.9%	5.0%	
471.omnetpp	317	23.4%	7.4%	11.4%	9.3%	5.0%	26.3%	6.8%	14.4%	6.5%	
473.astar	453	5.9%	3.5%	1.9%	3.8%	0.6%	7.2%	1.1%	0.7%	0.9%	
483.xalancbmk	220	33.4%	16.7%	12.6%	13.6%	7.8%	43.8%	16.7%	17.2%	16.0%	
433.milc	565	4.6%	1.9%	1.9%	2.2%	1.6%	4.7%	1.0%	2.5%	1.5%	
444.namd	513	-0.1%	0.0%	0.0%	0.2%	0.1%	0.6%	0.0%	0.1%	-0.1%	
447.dealII	359	16.3%	9.4%	7.3%	7.0%	5.1%	17.3%	7.4%	5.0%	5.3%	
450.soplex	280	0.0%	1.1%	-0.7%	-0.4%	-1.0%	1.3%	-0.9%	0.6%	0.2%	
453.povray	218	29.3%	12.9%	10.9%	9.6%	10.4%	30.9%	11.5%	11.0%	8.7%	
470.lbm	409	1.3%	1.1%	0.9%	0.0%	0.5%	-0.2%	0.3%	0.5%	0.1%	
482.sphinx3	466	2.3%	2.8%	1.8%	0.8%	1.0%	4.3%	0.5%	0.9%	1.3%	
SPECint: all benchmarks	5303		5.0%	4.5%	4.4%	2.6%		5.7%	6.2%	4.9%	
- without perl or gcc	4650	11.4%	4.7%	4.2%	4.1%	2.4%	15.7%	5.3%	5.8%	4.6%	
SPECfp: except Fortran	2810	7.2%	4.1%	3.1%	2.7%	2.5%	7.9%	2.7%	2.9%	2.4%	
SPEC CPU: except Fortran	8113		4.7%	4.0%	3.8%	2.6%		4.6%	5.0%	4.0%	
- and without perl or gcc	7460	9.6%	4.4%	3.7%	3.5%	2.4%	12.5%	4.2%	4.6%	3.7%	

Figure 2.15: Benchmarks for x86, -O3 with `RET`s (left) or indirect jumps (right) in the epilogues. Baseline runtimes are in seconds. SPEC overheads are geometric means of **minimum** individual benchmark overheads.

x86, -O2	Baseline	With RET in epilogues					With indirect jumps instead of RET in epilogues			
		Traditional shadow stack	Parallel shadow stack			-fstack -protector -all	Traditional shadow stack	Parallel shadow stack		
			Checking	Overwriting, zeroing out	Overwriting, no zeroing			Checking	Overwriting, zeroing out	Overwriting, no zeroing
400.perlbench	354		8.4%	8.0%	9.4%	5.2%		8.9%	7.0%	6.8%
401.bzip2	599	2.3%	1.8%	1.3%	1.5%	-2.7%	4.6%	3.0%	2.5%	2.8%
403.gcc	319		6.5%	4.8%	4.9%	4.0%		9.1%	8.5%	7.5%
429.mcf	214	6.7%	0.5%	0.4%	0.3%	0.1%	6.7%	0.5%	0.5%	0.4%
445.gobmk	461	17.3%	4.2%	4.6%	4.1%	3.4%	25.5%	10.6%	9.2%	9.6%
456.hmmer	517	0.4%	0.2%	0.1%	0.2%	0.0%	0.4%	0.1%	0.2%	0.3%
458.sjeng	542	16.2%	4.6%	3.5%	1.9%	3.0%	23.7%	9.1%	8.4%	7.8%
462.libquantum	678	-0.1%	0.4%	0.3%	1.0%	-0.2%	0.4%	-0.4%	-0.3%	-0.2%
464.h264ref	732	19.9%	8.3%	6.2%	6.0%	3.5%	23.4%	7.4%	6.2%	4.9%
471.omnetpp	324	28.0%	9.0%	10.7%	10.8%	7.9%	29.2%	8.0%	9.2%	9.4%
473.astar	474	8.8%	5.1%	1.3%	3.4%	-1.4%	9.2%	1.5%	0.1%	0.7%
483.xalancbmk	238	40.7%	17.0%	12.9%	12.0%	10.5%	47.9%	15.8%	14.2%	13.5%
433.milc	582	4.6%	1.0%	1.8%	1.5%	1.1%	3.8%	0.5%	0.7%	0.7%
444.namd	518	-0.3%	-0.1%	-0.2%	-0.1%	-0.1%	-0.1%	-0.2%	-0.2%	-0.2%
447.dealII	392	14.2%	8.1%	5.2%	5.8%	5.5%	14.5%	6.0%	3.8%	4.1%
450.soplex	289	1.2%	2.3%	1.0%	1.2%	-0.5%	2.5%	0.5%	0.1%	-0.3%
453.povray	234	35.8%	13.3%	12.8%	11.6%	10.5%	37.3%	11.7%	10.0%	9.2%
470.lbm	409	1.4%	1.4%	0.4%	0.4%	0.1%	0.6%	-0.1%	0.8%	0.5%
482.sphinx3	471	5.3%	1.4%	-0.3%	0.5%	-0.2%	1.8%	-0.4%	0.3%	-1.3%
SPECint: all benchmarks	5452		5.4%	4.4%	4.6%	2.7%		6.0%	5.4%	5.2%
- without perl or gcc	4779	13.4%	5.0%	4.0%	4.0%	2.3%	16.2%	5.4%	4.9%	4.8%
SPECfp: except Fortran	2895	8.3%	3.8%	2.9%	2.9%	2.3%	8.0%	2.5%	2.2%	1.8%
SPEC CPU: except Fortran	8347		4.8%	3.9%	3.9%	2.6%		4.7%	4.2%	3.9%
- and without perl or gcc	7674	11.2%	4.5%	3.6%	3.6%	2.3%	12.8%	4.2%	3.8%	3.5%

Figure 2.16: Benchmarks for x86, -O2 with `RET`s (left) or indirect jumps (right) in the epilogues. Baseline runtimes are in seconds. SPEC overheads are geometric means of **minimum** individual benchmark overheads.

x64, -O3	Baseline	With RET in epilogues					With indirect jumps instead of RET in epilogues				
		Traditional shadow stack	Parallel shadow stack			-fstack -protector -all	Traditional shadow stack	Parallel shadow stack			
			Checking	Overwriting, zeroing out	Overwriting, no zeroing			Checking	Overwriting, zeroing out	Overwriting, no zeroing	
400.perlbench	343		8.5%	9.0%	7.9%	8.0%		10.0%	9.2%	7.8%	
401.bzip2	502	0.8%	-0.9%	0.9%	-1.2%	-1.2%	2.6%	2.3%	0.0%	-1.2%	
403.gcc	318		4.3%	3.8%	4.0%	4.2%		8.9%	7.7%	3.5%	
429.mcf	280	3.8%	-0.1%	-0.2%	-0.2%	0.1%	3.6%	0.1%	-0.1%	-0.2%	
445.gobmk	437	12.0%	4.8%	5.0%	3.6%	4.4%	21.2%	11.1%	10.8%	3.7%	
456.hmmer	412	0.2%	0.0%	0.0%	0.0%	0.0%	0.3%	0.0%	0.0%	0.0%	
458.sjeng	493	16.0%	4.3%	5.5%	4.2%	3.8%	18.6%	6.8%	6.6%	4.2%	
462.libquantum	485	0.9%	1.0%	-0.8%	0.1%	1.0%	1.4%	1.2%	-0.8%	0.7%	
464.h264ref	584	25.9%	11.1%	9.4%	9.6%	7.3%	33.2%	12.2%	11.7%	9.6%	
471.omnetpp	372		7.2%	13.9%	9.8%	10.7%		10.2%	9.4%	9.6%	
473.astar	403	10.7%	3.7%	4.3%	2.8%	3.7%	10.5%	3.9%	3.1%	3.8%	
483.xalancbmk	239		7.0%	8.4%	12.5%	8.3%		10.1%	12.6%	9.4%	
433.milc	527		1.2%	1.2%	1.2%	0.8%		6.0%	1.1%	1.1%	
444.namd	437	0.0%	0.1%	0.1%	0.1%	0.1%	0.1%	0.2%	0.1%	0.1%	
447.dealII	338	24.8%	10.7%	7.8%	8.2%	5.1%	26.7%	7.9%	5.4%	8.1%	
450.soplex	276		1.9%	1.8%	1.5%	-0.6%		-0.6%	1.3%	-0.7%	
453.povray	202	36.0%	9.7%	8.3%	7.7%	6.3%	37.2%	9.2%	6.2%	7.4%	
470.lbm	397	-0.8%	-0.7%	-0.9%	-0.7%	-0.4%	0.2%	0.3%	-0.6%	-0.6%	
482.sphinx3	583	3.2%	1.4%	0.8%	1.3%	0.5%	3.8%	1.6%	1.3%	1.5%	
SPECint: all benchmarks	4868		4.2%	4.8%	4.4%	4.1%		6.3%	5.7%	4.2%	
- without perl or gcc	4207	8.5%	3.7%	4.5%	4.0%	3.7%	10.9%	5.7%	5.2%	3.9%	
SPECfp: except Fortran	2760	11.7%	3.4%	2.7%	2.7%	1.7%	12.6%	3.4%	2.1%	2.4%	
SPEC CPU: except Fortran	7628		3.9%	4.0%	3.7%	3.2%		5.2%	4.4%	3.5%	
- and without perl or gcc	6967	9.7%	3.6%	3.8%	3.5%	2.9%	11.6%	4.8%	3.9%	3.3%	

Figure 2.17: Benchmarks for x64, -O3 with `RET`s (left) or indirect jumps (right) in the epilogues. Baseline runtimes are in seconds. SPEC overheads are geometric means of **minimum** individual benchmark overheads.

x64, -O2	Baseline	With RET in epilogues					With indirect jumps instead of RET in epilogues				
		Traditional shadow stack	Parallel shadow stack			-fstack -protector -all	Traditional shadow stack	Parallel shadow stack			
			Checking	Overwriting, zeroing out	Overwriting, no zeroing			Checking	Overwriting, zeroing out	Overwriting, no zeroing	
400.perlbench	353		9.2%	7.4%	7.9%	2.8%		8.3%	6.6%	6.5%	
401.bzip2	492	2.2%	3.1%	1.1%	0.7%	0.5%	6.8%	3.1%	2.6%	2.8%	
403.gcc	329		4.0%	3.4%	3.5%	4.2%		7.5%	7.0%	7.5%	
429.mcf	282		0.0%	0.1%	0.3%	0.5%		0.1%	0.1%	0.4%	
445.gobmk	443	16.5%	5.6%	5.6%	4.4%	4.8%	26.0%	12.4%	12.3%	11.0%	
456.hmmer	429	0.3%	0.0%	0.0%	0.0%	0.1%	0.4%	0.0%	0.1%	0.1%	
458.sjeng	519	17.6%	5.3%	3.9%	3.8%	4.2%	24.1%	8.0%	7.4%	6.5%	
462.libquantum	488	-0.8%	0.1%	-1.2%	1.0%	0.6%	0.8%	-0.7%	-1.1%	-0.3%	
464.h264ref	609	26.2%	9.8%	8.4%	8.6%	7.3%	29.6%	10.0%	7.9%	7.7%	
471.omnetpp	373	31.9%	12.6%	14.3%	11.5%	9.5%	33.4%	12.4%	10.3%	9.1%	
473.astar	436	12.1%	6.3%	4.7%	4.3%	4.8%	11.6%	3.8%	2.7%	2.0%	
483.xalancbmk	241		15.0%	13.8%	14.1%	15.2%		17.5%	19.2%	15.3%	
433.milc	531		0.8%	0.7%	0.8%	1.4%		0.9%	0.9%	0.8%	
444.namd	437	0.0%	0.0%	0.1%	0.0%	0.2%	0.0%	0.0%	0.0%	0.0%	
447.dealII	350	29.3%	11.1%	9.5%	9.0%	6.9%	33.4%	9.7%	6.5%	6.3%	
450.soplex	279		0.5%	-0.2%	1.2%	-1.5%		1.0%	1.2%	1.1%	
453.povray	203	47.4%	15.6%	14.4%	13.5%	12.6%	50.0%	13.7%	12.8%	10.7%	
470.lbm	396	-0.3%	0.6%	-0.3%	0.6%	0.5%	0.7%	-0.4%	0.8%	-0.1%	
482.sphinx3	588	2.3%	0.7%	-0.1%	-0.1%	0.5%	2.9%	0.2%	0.8%	0.6%	
SPECint: all benchmarks	4992		5.8%	5.0%	4.9%	4.5%		6.7%	6.1%	5.6%	
- without perl or gcc	4311	12.7%	5.7%	5.0%	4.8%	4.7%	15.9%	6.5%	6.0%	5.3%	
SPECfp: except Fortran	2784	14.2%	4.0%	3.3%	3.5%	2.8%	15.7%	3.5%	3.2%	2.7%	
SPEC CPU: except Fortran	7777		5.2%	4.4%	4.4%	3.9%		5.5%	5.0%	4.5%	
- and without perl or gcc	7095	13.3%	5.0%	4.3%	4.2%	3.9%	15.8%	5.2%	4.8%	4.2%	

Figure 2.18: Benchmarks for x64, -O2 with `RET`s (left) or indirect jumps (right) in the epilogues. Baseline runtimes are in seconds. SPEC overheads are geometric means of **minimum** individual benchmark overheads.

x86, -O3	With RET in epilogues					With indirect jumps instead of RET in epilogues				
	Baseline	Traditional shadow stack	Parallel shadow stack			-fstack -protector -all	Traditional shadow stack	Parallel shadow stack		
			Checking	Overwriting, zeroing out	Overwriting, no zeroing			Checking	Overwriting, zeroing out	Overwriting, no zeroing
400.perlbench	3.4		5.6	3.1	0.4	0.5		0.7	7.2	3.1
401.bzip2	0.2	20.1	0.1	0.2	0.1	0.5	7.0	1.4	6.7	5.0
403.gcc	1.1		1.1	0.6	0.3	1.3		0.6	2.1	0.7
429.mcf	0.1	0.3	0.1	0.1	0.1	0.1	2.9	0.1	1.9	0.1
445.gobmk	0.3	0.4	1.0	0.5	0.6	0.4	12.0	0.3	11.5	3.4
456.hmmer	0.1	0.3	12.9	34.1	0.1	1.0	9.7	0.0	10.3	0.2
458.sjeng	0.2	0.1	1.6	4.9	0.1	0.5	16.5	0.1	4.9	0.2
462.libquantum	2.1	3.8	4.1	1.7	2.6	4.0	1.8	2.0	2.9	2.7
464.h264ref	0.3	4.4	7.0	4.7	0.4	0.1	17.7	0.1	5.7	0.3
471.omnetpp	1.1	2.8	1.4	3.6	1.3	7.5	14.9	2.7	9.9	1.9
473.astar	0.3	2.5	0.8	1.6	0.4	0.9	5.6	1.5	3.0	1.7
483.xalancbmk	0.4	1.5	0.3	2.3	0.5	3.4	9.8	2.5	7.8	1.7
	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
433.milc	2.7	0.8	0.4	0.5	0.6	0.3	3.0	0.4	2.2	0.4
444.namd	2.6	0.1	0.1	0.3	0.1	1.1	13.2	0.0	2.7	0.1
447.dealII	0.4	0.4	0.2	0.1	0.1	0.2	10.7	0.2	2.7	0.1
450.soplex	0.3	1.1	0.4	0.6	0.9	1.9	1.2	1.3	2.2	1.1
453.povray	0.2	0.3	0.2	0.5	0.3	0.2	0.3	0.7	0.3	0.2
470.lbm	2.9	0.8	1.3	1.3	1.2	2.2	2.0	0.7	1.6	2.2
482.sphinx3	2.9	11.1	6.7	4.4	1.7	1.8	20.5	1.4	6.2	7.4

Figure 2.19: Benchmarks for x86, -O3 with RETs (left) or indirect jumps (right) in the epilogues. This table shows the standard deviation (in seconds) of each benchmark.

x86, -O2	With RET in epilogues					With indirect jumps instead of RET in epilogues				
	Baseline	Traditional shadow stack	Parallel shadow stack			-fstack -protector -all	Traditional shadow stack	Parallel shadow stack		
			Checking	Overwriting, zeroing out	Overwriting, no zeroing			Checking	Overwriting, zeroing out	Overwriting, no zeroing
400.perlbench		13.7	13.8	0.7	13.8	1.8		2.3	2.2	3.2
401.bzip2	6.6	11.4	18.0	0.3	0.1	5.5	7.5	19.0	11.8	0.1
403.gcc		3.3	3.2	0.7	4.4	1.7		1.9	2.1	1.0
429.mcf	0.1	2.5	0.1	3.0	0.1	0.1	0.4	0.3	0.1	0.2
445.gobmk	0.3	0.3	0.3	0.5	1.6	0.3	4.1	5.2	9.4	0.4
456.hmmer	0.1	0.1	0.1	0.3	11.3	0.2	0.1	18.3	19.5	0.0
458.sjeng	0.4	0.4	0.3	0.4	1.4	0.2	0.3	0.5	0.2	0.4
462.libquantum	3.4	2.3	2.6	2.1	3.0	7.9	2.4	6.8	4.5	3.1
464.h264ref	1.8	0.2	0.1	0.2	0.1	6.6	0.1	1.3	3.3	0.1
471.omnetpp	1.3	3.4	0.8	4.4	2.2	1.9	0.6	2.8	6.0	0.4
473.astar	0.5	1.5	0.4	3.0	0.9	3.8	0.2	1.9	1.7	0.2
483.xalancbmk	0.9	3.0	0.3	3.8	3.0	3.7	2.7	3.0	2.9	0.5
433.milc	0.4	0.5	1.2	2.6	0.8	0.4	0.6	1.0	1.3	1.7
444.namd	0.1	10.6	0.1	1.3	0.2	0.1	0.2	2.5	6.2	2.4
447.dealII	0.3	9.7	0.1	0.3	0.2	0.5	0.1	0.1	1.9	2.4
450.soplex	1.5	3.6	1.1	1.6	1.2	0.9	1.2	1.9	1.9	1.6
453.povray	0.3	0.2	0.4	0.2	0.1	0.2	0.5	0.2	0.2	0.7
470.lbm	2.4	0.8	1.1	2.2	2.8	2.5	2.1	3.1	1.4	2.1
482.sphinx3	7.3	6.0	8.3	1.4	12.4	1.8	6.6	2.3	2.0	4.2

Figure 2.20: Benchmarks for x86, -O2 with RETs (left) or indirect jumps (right) in the epilogues. This table shows the standard deviation (in seconds) of each benchmark.

x86, -O3	With RET in epilogues					With indirect jumps instead of RET in epilogues				
	Baseline	Traditional shadow stack	Parallel shadow stack			-fstack -protector -all	Traditional shadow stack	Parallel shadow stack		
			Checking	Overwriting, zeroing out	Overwriting, no zeroing			Checking	Overwriting, zeroing out	Overwriting, no zeroing
400.perlbench	4.4		7.4	0.9	1.1	2.2		0.9	0.6	0.9
401.bzip2	5.8	11.5	17.3	0.5	2.7	11.4	16.2	0.5	6.2	0.2
403.gcc	0.8		1.0	0.9	0.5	1.5		1.1	2.1	1.6
429.mcf	0.9	0.6	0.4	1.5	0.9	0.5	0.8	0.6	0.6	0.7
445.gobmk	6.4	13.5	0.9	1.1	1.0	0.4	9.7	1.2	4.3	0.6
456.hmmer	23.6	27.8	0.0	0.5	0.1	0.2	16.9	1.7	2.7	0.4
458.sjeng	0.7	0.5	0.3	0.1	0.1	0.3	0.7	7.4	1.7	0.2
462.libquantum	3.4	5.0	4.4	2.9	4.4	2.8	2.7	2.8	3.6	3.1
464.h264ref	6.6	7.8	0.1	0.2	0.3	0.5	0.2	0.2	0.2	0.2
471.omnetpp	3.7		8.1	1.5	4.8	0.5		1.2	3.2	8.7
473.astar	2.5	0.5	3.0	0.9	1.5	0.6	1.1	1.8	1.2	1.5
483.xalancbmk	2.6		3.0	0.3	0.3	4.4		4.7	3.5	4.6
433.milc	0.6		0.5	0.6	0.7	0.4		1.4	13.9	10.4
444.namd	0.2	5.6	0.3	0.3	0.2	0.3	0.6	1.2	5.2	7.3
447.dealII	0.7	0.3	0.6	0.3	0.1	0.2	0.7	1.0	0.4	0.3
450.soplex	1.1		0.8	0.5	0.7	1.1		2.2	0.7	3.2
453.povray	0.3	0.9	0.8	1.3	0.9	0.4	0.6	1.1	0.8	1.7
470.lbm	1.0	1.7	2.5	0.4	2.1	1.6	0.8	0.6	2.2	2.1
482.sphinx3	5.4	3.4	16.3	1.7	2.1	9.2	8.1	4.5	3.6	5.0

Figure 2.21: Benchmarks for x64, -O3 with RETs (left) or indirect jumps (right) in the epilogues. This table shows the standard deviation (in seconds) of each benchmark.

x64, -O2	With RET in epilogues						With indirect jumps instead of RET in epilogues					
	Baseline	Traditional shadow stack	Parallel shadow stack			-fstack -protector -all	Traditional shadow stack	Parallel shadow stack				
			Checking	Overwriting, zeroing out	Overwriting, no zeroing			Checking	Overwriting, zeroing out	Overwriting, no zeroing		
400.perlbench		0.1	1.5	14.6	10.2	3.0		5.4	9.4	2.2		
401.bzip2	0.1	1.5	18.7	0.4	30.3	0.4	9.6	0.2	27.3	21.6		
403.gcc		0.2	2.0	1.9	1.9	1.4		1.7	1.8	1.3		
429.mcf		0.2	1.0	0.6	0.7	0.7		0.4	1.0	0.7		
445.gobmk	12.7	0.3	4.5	16.1	5.8	0.3	2.6	6.8	10.1	2.7		
456.hmmer	17.4	0.2	0.1	0.1	0.2	0.1	0.1	28.7	17.0	0.1		
458.sjeng	1.3	0.4	1.9	0.6	0.2	0.3	0.2	0.3	1.3	4.4		
462.libquantum	3.6	3.6	1.9	3.9	4.8	4.3	3.0	4.2	4.6	3.7		
464.h264ref	3.6	0.4	0.5	0.2	3.1	8.9	0.1	7.6	5.4	5.7		
471.omnetpp	2.7	2.8	8.2	4.4	1.3	5.7	1.8	2.3	7.1	5.2		
473.astar	1.9	2.3	2.1	1.4	2.4	2.1	2.2	0.5	3.1	1.6		
483.xalancbmk		0.6	2.6	5.9	2.0	4.5		0.8	1.9	1.7		
433.milc		0.0	0.7	0.4	0.6	0.4		10.4	10.5	0.5		
444.namd	0.4	0.1	0.1	0.1	0.1	0.1	0.4	5.4	5.4	0.1		
447.dealII	0.8	0.4	0.1	0.4	0.1	0.2	0.2	0.9	4.7	0.3		
450.soplex		1.6	1.5	1.7	0.4	2.3		0.6	0.5	0.6		
453.povray	0.3	0.6	1.1	0.8	0.6	0.5	0.9	0.8	0.3	0.4		
470.lbm	2.0	2.7	0.6	2.2	0.6	1.3	0.4	2.1	0.4	1.4		
482.sphinx3	3.3	1.7	3.4	6.1	4.7	7.8	1.3	1.7	3.3	7.1		

Figure 2.22: Benchmarks for x64, -O2 with RETs (left) or indirect jumps (right) in the epilogues. This table shows the standard deviation (in seconds) of each benchmark.

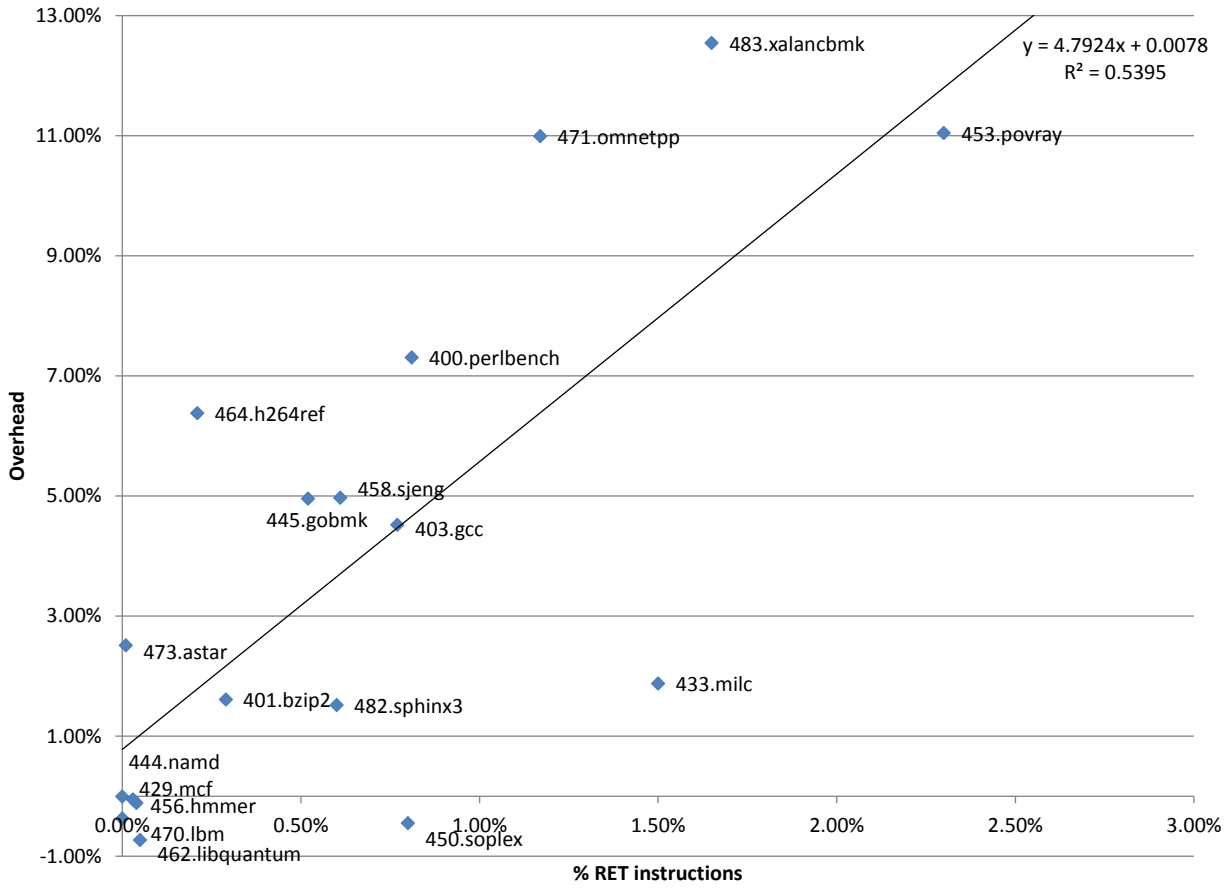


Figure 2.23: Correlation between the percentage of RET instructions and the overhead.

the shadow stack pointer. This is strongly supported by an experiment where we augmented the parallel shadow stack with those extra instructions: the overhead approached that of the traditional shadow stack. Figures 2.24, 2.25, and 2.26 show the epilogues for the parallel shadow stack where we have incrementally added instructions to save and restore `%ecx`, loop in the event of a mismatch, and load/update the pseudo-shadow stack pointer. Note that Figure 2.26 is not the same as a traditional shadow stack, because it still uses the parallel shadow stack; it simply incurs the cost of maintaining and updating a traditional shadow stack pointer, which is not meaningfully used. Figure 2.27 shows the overhead of these three augmented parallel shadow stack schemes.

This model suggests that the overheads of different instrumentation schemes should be correlated with each other: the programs that incur high overhead with one instrumentation scheme, will also tend to incur relatively high overhead on other instrumentation schemes as well. This appears to be supported by our data; for example, `xalancbmk` and `povray` have the highest overheads for every instrumentation scheme. The overhead of the schemes we investigate also appears to be correlated with that of other published shadow stack

```
1 MOV %edx, %gs:104
2
3 MOV -0xa0000000(%esp), %edx
4
5 CMP %edx, (%esp)
6 JNZ abort$$
7
8 MOV %gs:104, %edx
9
10 RET
11
12 abort$$:
13     HLT
```

Figure 2.24: Epilogue for parallel shadow stack, augmented to save and restore `%ecx`.

```
1 MOV %edx, %gs:104
2
3 MOV -0xa0000000(%esp), %edx
4
5 non_match:
6     CMP %edx, (%esp)
7     JNZ non_match
8
9 MOV %gs:104, %edx
10
11 RET
```

Figure 2.25: Epilogue for parallel shadow stack, augmented to save and restore `%ecx`, and loop in the event of a mismatch.

schemes [34, 107] and instrumentation schemes [84, 108, 107]. Thus, our discussion of avenues for improvement is generalizable to other shadow stack implementations and to CFI.

2.9.1.1 Omitted Benchmarks

We omitted the Fortran benchmarks due to the engineering effort required, relative to their relevance in a security context. These benchmarks have an extremely low percentage of `RET` instructions: 6 out of 10 have $\leq 0.02\%$, and the maximum is 0.21%. Our model suggests that shadow stacks will have low overhead on the Fortran benchmarks. Thus, our results overestimate the overhead for SPECfp and SPEC CPU.

We were not able to instrument `perlbench` and `gcc` with a traditional shadow stack. We anticipate their overhead with a traditional shadow stack would be substantial, as 0.81% and 0.77% of their instructions are `RET`s, respectively. These programs have relatively high

```
1 MOV %edx, %gs:104
2
3 MOV %gs:108, %ecx
4 MOV -0xa0000000(%esp), %edx
5
6 non_match:
7       ADD $4, %ecx
8       CMP %edx, (%esp)
9       JNZ non_match
10
11 MOV %ecx, %gs:108
12
13 MOV %gs:104, %edx
14
15 RET
```

Figure 2.26: Epilogue for parallel shadow stack, augmented to save and restore `%ecx`, loop in the event of a mismatch, and load/update the pseudo-shadow stack pointer.

x86, -O3	Baseline	With RET in epilogues: Parallel shadow stack + save/restore scratch register	+ pseudo-loop until match	+ load/update pseudo-SSP
400.perlbench	354	14.2%	13.4%	18.1%
401.bzip2	585	0.9%	0.9%	2.2%
403.gcc	300	8.1%	8.7%	12.5%
429.mcf	211	0.3%	0.2%	0.4%
445.gobmk	449	7.3%	7.2%	10.6%
456.hmmer	516	-0.1%	-0.1%	0.1%
458.sjeng	518	8.5%	9.6%	13.8%
462.libquantum	669	-1.0%	-1.2%	-0.4%
464.h264ref	716	12.4%	12.6%	17.8%
471.omnetpp	319	17.5%	17.6%	21.1%
473.astar	453	5.8%	4.6%	6.5%
483.xalancbmk	221	19.6%	21.6%	30.8%
433.milc	565	2.8%	2.6%	3.4%
444.namd	513	-0.1%	-0.1%	0.0%
447.dealII	360	12.6%	12.4%	18.5%
450.soplex	281	0.8%	1.3%	1.1%
453.povray	218	19.9%	19.5%	26.0%
470.lbm	416	-1.0%	-0.3%	-0.3%
482.sphinx3	471	2.1%	2.0%	2.4%

Figure 2.27: Benchmarks for x86, -O3 with our hybrid traditional/parallel shadow stacks.

overheads when instrumented with the parallel shadow stack or `-fstack-protector-all`.

There are a number of other CFI or shadow stack studies that omit some of the SPEC CPU benchmarks, as we have. However, in some cases, their omitted benchmarks are those which we would predict to be expensive (based on the percentage of `RET`s, and our own measurements); thus, their omission suggests that their estimate of performance overhead might be overly optimistic.

2.9.1.2 Indirect Jumps vs `RET`s

With indirect jumps, the CPU can no longer predict the return address using its internal stack. While there is still dynamic branch prediction for indirect jumps, our results show that this is noticeably imperfect. This is not surprising: we expect, for example, that `printf` will be called from many different functions (`foo`, `bar`, `baz`); and each time `printf` may "return" to any of those functions. This is somewhat similar to BinCFI [108], where its "trampoline" (address translation) routine would "return" (with an indirect jump) to many different functions. This arguably supports the prioritization of performance over TOCTTOU protection, as done by many CFI and shadow stack schemes [108]. In the context of software fault isolation (not a shadow stack), PittSFIeld [68] reported that replacing `RET`s with indirect jumps increased the overhead on SPECint2000 from 21% to 27%.

2.9.2 Avenues for improvement

The traditional shadow stack has close to 10% overhead — which is unlikely to be acceptable for widespread deployment [98] — and even a minimalist parallel shadow stack has roughly 3.5% overhead. However, these were obtained by measuring the overhead of (a) **our hand-coded assembly**, for (b) **a traditional shadow stack**, when we instrumented (c) **100%** of (d) **the (intended) `RET`s normally emitted by the compiler**. We can potentially improve performance by modifying each of those aspects:

(a) Equivalent but faster prologues and epilogues: we already tried many functionally equivalent prologues and epilogues, and even *super-optimization* [67] can provide only limited savings, as the overhead depends in part on the percentage of memory loads and stores, which are mostly unavoidable; for the limited case of leaf functions (those that make no calls), Crypto CFI [66] uses XMM4 (an SSE register) to store the return instruction pointer and frame pointer.
For the traditional shadow stack, we could modify the `setjmp/longjmp` functions or instrument their call sites (as done by Kuznetsov et al. [61] for their dual stacks), to maintain the invariant that the top of the shadow stack is always the correct return address. This means that, rather than using a loop to check the return address, we could use simpler instrumentation similar to the parallel shadow stack (albeit still with another layer of indirection in the form of the shadow stack pointer).

(b) Relaxations or variations of a shadow stack: for example, the parallel shadow stack scheme, a shadow set (which keeps track of which return addresses are valid, but does not enforce the ordering of return addresses) rather than a shadow stack [32], or monitoring the taintedness of the return address in only the most recent one or two stack frames [59]. All of these, to some extent, trade off some security for performance.

(c) Selectively instrumenting functions: choosing a random subset of functions to instrument would greatly sacrifice security — if we randomly selected $1/n^{th}$ of functions to instrument, then the *expected* overhead is $1/n^{th}$ (even if the function run-times are not uniform). A better choice might be identifying (in-)frequently called functions, using profiling [99]. Alternatively, SecondWrite's [79] return address check optimization omits the shadow stack instrumentation from functions that do not have indexed array writes.[14] They noted that small leaf functions and recursive functions, which benefit the most from this optimization, are also the most frequently called. Crypto CFI [66] also optimizes leaves, instead by storing the return address in a register rather than encrypting.

(d) Reducing the number of `RET`s through inlining (e.g., our use of `-O3`, or with link-time optimization [66])

Although reducing the number of `RET`s or selectively instrumenting functions (chosen appropriately) are valuable contributions, these are orthogonal to improving the prologues/epilogues or relaxations of the shadow stack paradigm. We should beware of conflating the speed of an instrumentation scheme with the advantage gained from a particular optimization: although shadow stack scheme A, run on benchmarks with aggressive inlining, may appear to have lower overhead than shadow stack scheme B, this might be attributable to the inlining rather than the merits of scheme A, in which case the "best" solution would be scheme B with aggressive inlining.

Since software-only shadow stacks are expensive — even with the aforementioned incremental improvements — many authors [26, 81, 48, 104] have proposed hardware shadow-stack support. These are distinct from the return-address stack already present in modern processors for branch prediction [96, 55], which are not secure: if there is a mismatch between them, the hardware reverts to using the main stack. Hardware shadow stack schemes are usually extremely fast, instrument all `RET`s (even unintended `RET`s), and do not require recompilation, but introduce complications for code that intentionally violates `CALL-RET` matching.

Davi et al. [32] proposed a hardware-assisted CFI scheme that includes a shadow *set*; this requires the addition of new labels/instructions to the code. Kao and Wu [59] proposed new registers for the Intel architecture, to store the location of the current return address, and the old value of `%ebp`. In June 2016, Intel announced their Control-flow Enforcement

[14] Unfortunately, the claim of "not sacrificing any protection" is incorrect, e.g., a bufferless function `foo` that calls `bar` could have its return address overwritten by `bar`, if `bar` has a vulnerable indexed array write.

Technology (CET) [54], which includes both coarse-grained forward-edge CFI, and a precise shadow stack.

New hardware features that are not security specific can also improve performance. For example, Crypto CFI [66] uses multiple XMM registers. Anecdotally, our parallel shadow stack appears to have lower overhead on a newer processor (a 2011 Intel Core i7-3930K) — perhaps because of an improved stack engine [44]. However, we expect that non-security specific hardware improvements will not significantly change the overheads, since there are the memory load/store costs, and there is little room for improvement with branch prediction (the indirect jumps in the `RET` variants of the shadow stacks are only taken if a return-address mismatch occurs, and therefore can be predicted dynamically; the overheads of the indirect jump variants are lower-bounded by the `RET` variants, which have nearly perfect branch prediction).

2.9.3 Deployment issues

The parallel shadow stack variants have lower overhead than the traditional shadow stack, but not sufficiently low that widespread deployment would be an obvious decision. Even faster is `-fstack-protector-all`, but its attractiveness is tempered by its strictly weaker security properties (as per Figure 2.1). Additionally, `-fstack-protector-all` was applied at the compiler level, though it is possible to add it through binary rewriting (e.g., Second-Write [79]).

Our implementation is designed only to provide accurate estimates of the overhead of shadow stacks, not shelf-ready code. Nonetheless, some seemingly tricky cases are actually non-issues. For example, consider tail call elimination (e.g., suppose function `f` calls `g` which calls `h`, and function `h`; there are some limited cases where the compiler can convert the `CALL` to `h` into a jump); this does not change the assumption that the top of the stack in the prologue is the expected return address. Another case is the `get-EIP` idiom (a `CALL` followed by a `POP` can be used to obtain the instruction pointer, as required for position-independent code; this idiom is commonly used on 32-bit x86, which lacks an instruction to directly read EIP [55]): this still works because we instrument neither the `CALL` nor `POP`.

For other corner cases (e.g., exceptions, multi-threading), we defer to Szekeres et al.'s [98] assessment that compatibility issues can be avoided through careful engineering.

2.9.4 Generalizability

Mytkowicz et al. [71] demonstrated that a narrow set of environment and compilation options may lead to invalid results. Nonetheless, we are confident in our calculated overheads due to 1) the consistency of results across a variety of parameters (x86/x64, `-O2`/`-O3`, ad-hoc tests on a different CPU); 2) the strong correlation of per-program overheads between different instrumentation options, and with the static `RET` instruction counts; 3) other steps in our methodology (disabling Turbo Boost, disabling ASLR, and rebooting between each batch of benchmarks).

```
1 POP 999996(%esp)
2 # signal could happen here
3 SUB $4, %esp
```

```
1 POP 999996(%esp)
2 # signal could happen here
3 PUSH 999996(%esp)
4 # but this restores the return address
```

```
1 XCHGL (%esp), %ecx
2 MOVL %ecx, 1000000(%esp)
3 XCHGL (%esp), %ecx
```

Figure 2.28: Unsafe prologue (left) and safe prologues (right).

2.9.5 Signal/interrupt handlers for 32-bit

Some choices of assembly code for parallel shadow stack prologues adjust the stack pointer to above where we have useful data. This is, in general, unsafe for 32-bit binaries because signal or interrupt handlers may corrupt the stack.[15] Consider, for example, the prologue in Figure 2.28 (left). The `POP` instruction moves the stack pointer so that it is above the return address. If a signal occurs, the signal handler may write to the stack, overwriting the return address. The analogous prologue for 64-bit Linux is safe because of the "red zone" [52], which states that the 128-bytes beyond `%rsp` are considered reserved. We therefore only discuss the 32-bit case in the rest of this section.

Figure 2.28 (right) we show examples of prologues that are always safe. The top-right prologue is safe because, even if the return address is corrupted by a signal handler after the `POP`, the following `PUSH` operation will restore the return address with the pristine copy from the shadow stack. The bottom-right prologue is safe because it does not move the stack pointer above where the return address is stored.

The "unsafe" prologues are actually safe when combined with an overwriting-style epilogue. This is because the correct return address was copied from the main stack to the shadow stack before the main stack possibly got corrupted, and the overwriting-style epilogue solely uses the (correct) return address on the shadow stack. With a checking-style epilogue, it is unsafe because it will be detected as a mismatch/attack, resulting in a false alarm.

Our epilogues are safe because the potentially trashed return address on the main stack will be replaced afterwards with the correct return address:

```
1 ADD $4, %esp
2 # signal could happen here
3 PUSH 999996(%esp) # but correct return address written here
```

Peephole optimizations.

[15] We thank Matthew Fernandez of Intel Labs for pointing this out.

```
1 POP 999996(%esp)
2 SUB $4, %esp
3 PUSH %ebp
4 MOV %esp, %ebp
5 SUB <X>, %esp
```

```
1 POP 999996(%esp)
2 SUB <X+8>, %esp
3 MOV %ebp, +X(%esp)
4 LEA +X(%esp), %ebp
```

```
1 POP 999996(%esp)
2 MOV %ebp, -8(%esp)
3 % signal could happen here
4 LEA -8(%esp), %ebp
5 % signal could also happen here
6 SUB <X+8>, %esp
```

Figure 2.29: Sometimes unsafe vanilla (left) and peephole optimized (middle) prologues, and a very unsafe peephole optimized prologue (right).

```
1 MOV %ebp, %esp
2 POP %ebp
3 ADD $4, %esp
4 PUSH 999996(%esp)
5 RET
```

```
1 LEA 8(%ebp), %esp
2 # signal could happen
    here
3 MOV -8(%esp), %ebp
4 PUSH 999996(%esp)
5 RET
```

Figure 2.30: Safe (left) and very unsafe peephole optimized (right) epilogues.

These can sometimes introduce or exacerbate the signal/interrupter-handler issue for prologues and epilogues. For example, Figure 2.29 (left) shows a sometimes unsafe prologue, and Figure 2.29 (center) shows a peephole-optimized variant. The "very unsafe" prologue in Figure 2.29 (right), risks corrupting the frame pointer as well as the return address, since the stack pointer is sometimes up to 8 bytes above where there is useful data stored. We consider it **very** unsafe, because instead of only resulting in a false alarm (terminating the program), a corrupted frame pointer may lead to attacks. A simple, but not inexpensive, fix would be to store both the return address and frame pointer on the shadow stack (this is done, for different security reasons, by some shadow stack schemes [82, 6]).

Figure 2.30 shows an analogous risk of frame pointer corruption during the epilogue, with the same fix possible (restoring the frame pointer from the shadow stack).

2.10 Conclusion

In this chapter, we considered the performance costs of using a shadow stack. Our results suggest that a shadow stack, even when pared to its bare minimum (the overwriting, non-zeroing version of the parallel shadow stack), has non-negligible performance overhead, due to increased memory pressure. Achieving low-overhead protection against control-flow attacks will likely require alternative paradigms, such as the Code Pointer Separation (CPS) and Safe Stack techniques proposed by Kuznetsov et al. [61] — which unfortunately currently requires recompilation, unlike coarse-grained CFI [108, 106] — or hardware support.

Chapter 3

Oscar: A Practical Page-Permissions-Based Scheme for Thwarting Dangling Pointers

Using memory after it has been freed opens programs up to both data and control-flow exploits. Recent work on temporal memory safety has focused on using explicit lock-and-key mechanisms (objects are assigned a new lock upon allocation, and pointers must have the correct key to be dereferenced) or corrupting the pointer values upon `free()`. Placing objects on separate pages and using page permissions to enforce safety is an older, well-known technique that has been maligned as too slow, without comprehensive analysis. We show that both old and new techniques are conceptually instances of lock-and-key, and argue that, in principle, page permissions should be the most desirable approach. We then validate this insight experimentally by designing, implementing, and evaluating Oscar, a new protection scheme based on page permissions. Unlike prior attempts, Oscar does not require source code, is compatible with standard and custom memory allocators, and works correctly with programs that `fork`. Also, Oscar performs favorably — often by more than an order of magnitude — compared to recent proposals: overall, it has similar or lower runtime overhead, and lower memory overhead than competing systems.

3.1 Introduction

A temporal memory error occurs when code uses memory that was allocated, but since freed (and therefore possibly in use for another object), i.e., when an object is accessed outside of the time during which it was allocated.

Suppose we have a function pointer stored on the heap that points to function `Elmo()` (see Figure 3.1) at address `0x05CADA`. The pointer is used for a bit and then de-allocated. However, because of a bug, the program accesses that pointer again after its deallocation.

This bug creates a control-flow vulnerability. For example, between the de-allocation

```
1  void(**someFuncPtr)() = malloc(sizeof(void*));
2  *someFuncPtr = &Elmo;     // At 0x05CADA
3  (*someFuncPtr)();         // Correct use.
4  void(**callback)();
5  callback = someFuncPtr;
6  ...
7  free(someFuncPtr);        // Free space.
8  userName = malloc(...); // Reallocate space.
9  ... // Overwrite object with &Grouch at 0
       x05DEAD.
10 (*callback)();            // Use after free!
```

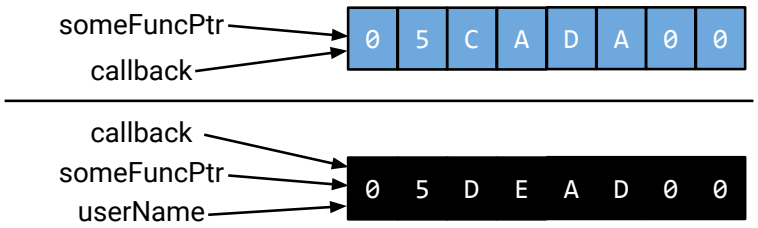


Figure 3.1: Top: `someFuncPtr` and `callback` refer to the function pointer, stored on the heap. Bottom: `userName` reuses the freed memory, formerly of `someFuncPtr/callback`.

(line 7) and faulty re-use of the pointer (line 10), some other code could allocate the same memory and fill it from an untrusted source — say a network socket. When the de-allocated pointer is faultily invoked, the program will jump to whatever address is stored there, say the address of the ROP gadget `Grouch()` at address `0x05DEAD`, hijacking control flow.

Heap temporal memory safety errors are becoming increasingly important [62, 105]. Stack-allocated variables are easier to protect, e.g., via escape analysis, which statically checks that pointers to a stack variable do not outlive the enclosing stack frame, or can be reduced to the heap problem, by converting stack allocations to heap allocations [77]. Stack use-after-free is considered rare [105] or difficult to exploit [62]; a 2012 study did not find any such vulnerabilities in the CVE database [17]. We therefore focus on temporal memory safety for heap-allocated objects in the rest of this work.

Various defenses have been tried. A decade ago, Dhurjati and Adve [37] proposed using page permissions and aliased virtual pages for protection. In their scheme, the allocator places each allocated object in a distinct virtual page, even though different objects may share the same physical page; when an object is deallocated, the corresponding virtual page is rendered inaccessible, causing pointer accesses after deallocation to fail. Although a combination of the technique with static analysis led to reasonable memory economy and performance, critics found faults with evaluation and generality, and — without quantitative comparison — summarily dismissed the general approach as impractical [75, 105], or without even mentioning it [60]. Since then, researchers have proposed more elaborate techniques (CETS [75], DangSan [60], Dangling Pointer Nullification [62] ("DangNull") and FreeSentry

[105]), relying on combinations of deeper static analysis and comprehensive instrumentation of heap operations such as object allocation, access, and pointer arithmetic. However, these schemes have yielded mixed results, including poor performance, partial protection, and incompatibility.

In this work, we first study past solutions, which we cast as realizations of a *lock-and-key* protection scheme (Section 3.3). We argue that using page permissions to protect from dangling pointers, an implicit lock-and-key scheme with lock changes, is less brittle and complex, and has the potential for superior performance. We then develop *Oscar*, a new protection mechanism using page permissions, inspired by Dhurjati and Adve's seminal work [37]. We make the following contributions:

- We study in detail the overhead contributed by the distinct factors of the scheme — shared memory mappings, memory-protection system calls invoked during allocation and deallocation, and more page table entries and virtual memory areas — using the standard SPEC CPU 2006 benchmarks (Section 3.4).
- We reduce the impact of system calls by careful amortization of virtual-memory operations, and management of the virtul address space (Section 3.5).
- We extend Oscar to handle server workloads, by supporting programs that `fork` children and the common case of custom memory allocators other than those in the standard C library (Section 3.7).
- We evaluate Oscar experimentally using both SPEC CPU2006 and the popular `memcached` service, showing that Oscar achieves superior performance, while providing more comprehensive protection than prior approaches.

Our work shows, in principle and experimentally, that protection based on page permissions — previously thought to be an impractical solution — may be the most promising for temporal memory safety. The simplicity of the scheme leads to excellent compatibility, deployability, and the lowest overhead: for example, on SPEC CPU2006, CETS and FreeSentry have 48% and 30% runtime overhead on `hmmer` respectively, vs. our 0.7% overhead; on `povray`, DangNull has 280% overhead while ours is $< 5\%$. While DangSan has runtime overhead similar to Oscar, DangSan's memory overhead (140%) is higher than Oscar's (61.5%). Also, our study of `memcached` shows that both standard and custom allocators can be addressed effectively and with reasonable performance.

3.2 Related Work

3.2.1 Dhurjati and Adve (2006)

Our work is inspired by the original page-permission with shadows scheme by Dhurjati and Adve [37]. Unlike Dhurjati and Adve's automatic pool allocation, Oscar can unmap shadows as soon as an object is freed, and does not require source code. Oscar also addresses

	One object per physical page frame	One object per shadow virtual page (core technique of Dhurjati & Adve)		
	e.g., Electric Fence	Vanilla	Automatic pool allocation (Dhurjati & Adve)	Our work
Physical memory overhead				
User-space memory	0 – 4KB per object (page align)	✓ Low overhead (O(sizeof(void*)) per object)		
Page table entry for live objects	1 page table entry per object			
Page table entry for freed objects	<Depends on implementation>	1 PTE per object	1 PTE per object in live pools	0 PTEs*
VMA struct for live objects	1 VMA struct per object			
VMA struct for freed objects	<Depends on implementation>	1 VMA struct per object		✓ None
Application source/ recompilation needed?	✓ No	✓ No	Yes: needs source + recompilation	✓ No
Compatible with fork()	✓ Yes	No; changes program semantics		✓ Mostly

Table 3.1: Comparison with Dhurjati and Adve. Green and a tick indicates an advantageous distinction. ∗ Oscar unmaps the shadows for freed objects, but Linux does not reclaim the PTE memory (see Section 3.9).

compatibility with `fork`, which appears to be a previously unknown limitation of Dhurjati and Adve's scheme.[1] They considered programs that `fork` to be advantageous, since virtual address space wastage in one child will not affect the address space of other children. Unfortunately, writes to old (pre-`fork`) heap objects will be propagated between parent and children (see Section 3.7.1), resulting in memory corruption.

While Dhurjati and Adve did measure the runtime of their particular scheme, their measurements do not let us break down how much each aspect of their scheme contributes to runtime overhead. First, their scheme relies upon static analysis (*Automatic Pool Allocation*: "APA"), and they did not measure the cost of shadow pages without APA. We cannot simply obtain "cost of syscalls" via "(APA + dummy syscalls) − APA", since pool allocation affects the cost of syscalls and cache pressure. Second, they did not measure the cost of each of the four factors we identified. For instance, they did not measure the individual cost of inline metadata or changing the memory allocation method; instead, they are lumped in with the cost of dummy syscalls. This makes it hard to predict the overhead of other variant schemes, e.g., using one object per physical page frame. Finally, they used a custom benchmark and Olden [88], which make it harder to compare their results to other schemes that are benchmarked with SPEC CPU; and many of their benchmark run-times are under five seconds, which means random error has a large impact. For these reasons, in this work

[1] We inspected their source `http://safecode.cs.illinois.edu/downloads.html` and found that they used `MAP_SHARED` without a mechanism to deal with `fork`.

we undertook a more systematic study of the sources of overhead in shadow-page-based temporal memory safety schemes.

To reduce their system's impact on page table utilization, Dhurjati and Adve employed static source-code analysis (Automatic Pool Allocation) — to separate objects into memory pools of different lifetimes, beyond which the pointers are guaranteed not to be dereferenced. Once the pool can be destroyed, they can remove (or reuse) page table entries (and associated `vm_area_struct`s) of freed objects. Unfortunately, there may be a significant lag between when the object is freed, and when its containing pool is destroyed; in the worst case (e.g., objects reachable from a global pointer), a pool may last for the lifetime of the program. Besides being imprecise, inferring object lifetimes via static analysis also introduces a requirement to have application source code, making it difficult and error-prone to deploy. Oscar's optimizations do not require application source code or compiler changes.

Oscar usually keeps less state for freed objects: they retain a page table entry (and associated `vm_area_struct`) for each freed object in live pools — some of which may be long-lived — whereas Oscar `munmap`s the shadow as soon as the object is freed (Table 3.1). Dhurjati and Adve expressly target their scheme towards server programs — since those do few allocations or deallocations — yet they do not account for `fork` or custom memory allocators.

If we are not concerned about the disadvantages of automatic pool allocation, it too would benefit from our optimizations. For example, we have seen that using `MAP_PRIVATE` greatly reduces the overhead for `mcf` and `milc`, and we expect this benefit to carry over when combined with automatic pool allocation.

3.2.2 Other Deterministic Protection Schemes

The simplest protection is to never `free()` any memory regions. This is perfectly secure, does not require application source code (change the `free` function to be no-op), has excellent compatibility, and low run-time overhead. However, it also requires infinite memory, which is impractical.

With DangNull [62], when an object is freed, all pointers to the object are set to NULL. The converse policy — when all references to a region are NULL (or invalid), automatically `free` the region — is "garbage collection". In C/C++, there is ambiguity about what is a pointer, hence it is only possible to perform conservative garbage collection, where anything that might plausibly be a pointer is treated as a pointer, thus preventing `free()`'ing of the referent. This has the disadvantages of false positives and lower responsiveness.

The Rust compiler enforces that each object can only have one owner [80]; with our lock-and-key metaphor, this is equivalent to ensuring that each lock has only one key, which may be "borrowed" (ala Rust terminology) but not copied. This means that when a key is surrendered (pointer becomes out of scope), the corresponding lock/object can be safely reused. `std::unique_ptr` provides similar semantics in C++11. It would be impractical to rewrite all legacy C/C++ software in Rust (or C++11), let alone provide Rust's guarantees to binaries that are compiled from C/C++.

MemSafe [94] combines spatial and temporal memory checks: when an object is deallocated, the bounds are set to zero (a special check is required for sub-object temporal memory safety). MemSafe modifies the LLVM IR, and does not allow inline assembly or self-modifying code. Of the five SPEC 2006 benchmarks they used, their run-times appear to be from the 'test' dataset rather than the 'reference' dataset. For example, for `astar`, their base run-time is 0.00 seconds, whereas the base run-time against which we compare Oscar is 408.9 seconds. Their non-zero run-time benchmarks have significant overhead — 183% for `bzip2`, 127% for `gobmk`, 124% for `hmmer`, and 120% for `sjeng` — though this includes spatial and stack temporal protection.

Dynamic instrumentation (e.g., Valgrind's memcheck [69]) is generally too slow other than for debugging.

Undangle [17] uses taint tracking to track pointer propagation. They do not provide SPEC results, but we expect it to be even slower than DangNull/FreeSentry, because Undangle determines how pointers are propagated by, in effect, interpreting each x86 instruction.

Safe dialects of C, such as CCured [77], generally require some source code changes, such as removing unsafe casts to pointers. CCured also changes the memory layout of pointers (plus-size pointers), making it difficult to interface with libraries that have not been recompiled with CCured.

3.2.3 Hardening

The premise of heap temporal memory safety schemes, such as Oscar, is that the attacker could otherwise repeatedly attempt to exploit a memory safety vulnerability, and has disabled or overcome any mitigations such as ASLR (nonetheless, as noted earlier, Oscar is compatible with ASLR). Thus, Oscar provides deterministic protection against heap use-after-free (barring address space exhaustion, which might necessitate reuse, as discussed in Section 3.8.1).

However, due to the high overhead of prior temporal memory safety schemes, some papers trade off protection for speed.

Many papers, starting with DieHard [8], "M-approximate" the infinite heap (using a heap that is M times larger than normally needed) and randomize where objects are placed on the heap. This means even if an object is used after it is freed, there is a "low" probability that the memory region has been reallocated. Archipelago [65] extends DieHard but uses less physical memory, by compacting cold objects. Both can be attacked by making many large allocations to exhaust the M-approximate heap, forcing earlier reuse of freed objects.

AddressSanitizer [90] also uses a quarantine pool, though with a FIFO reuse order, among other techniques. PageHeap [50] places freed pages in a quarantine, with the read/write page permissions removed. Attempted reuse will be detected only if the page has not yet been reallocated, so it may miss some attacks. These defenses can also be defeated by exhausting the heap.

Microsoft's MemoryProtection consists of Delayed Free (similar to a quarantine) and Isolated Heap (which separates normal objects from "critical" objects) [41]. Both of these

defenses can be bypassed [35].

Cling [3] only reuses memory among heap objects of the same type, so it ensures type-safe heap memory reuse but not full heap temporal memory safety.

3.2.4 Limiting the Damage from Exploits

Rather than attempting to enforce memory safety entirely, which may be considered too expensive, some approaches have focused on containing the exploit.

Often the goal of exploiting a use-after-free vulnerability is to hijack the control flow, such as by modifying function pointers per our introductory example. One defense is control-flow integrity (CFI) [1], which in its strongest form restricts the forward edges to an (often imprecise) statically computed set, and protects the backward edges using a shadow stack. However, recent work on "control-flow bending" [18] has observed that indirect branches often have many valid targets, which frequently makes it possible for an attacker to manipulate (using a memory corruption vulnerability) the indirect branches – without violating the CFI policy — to perform arbitrary computation.

Code pointer integrity (CPI) is essentially memory safety (spatial and temporal) applied only to code pointers [61]. Code pointer separation (CPS) is a weaker defense than CPI, because it does not protect pointers to code pointers, but is still stronger than CFI. Both CPI and CPS require compiler support.

CFI, CPS and CPI do not help against non-control data attacks, such as reading a session key or changing an 'isAdmin' variable [22]; recently, "data-oriented programming" has been shown to be Turing-complete [51].

3.3 Lock-and-Key Schemes

Use of memory after it has been freed can be seen as an authorization problem: pointers grant access to an allocated memory area and once that area is no longer allocated, the pointers should no longer grant access to it. Some have therefore used a lock-and-key metaphor to describe the problem of temporal memory safety [75]. In this section, we show how different published schemes map to this metaphor, explicitly and sometimes implicitly[2], and we argue that page-permission-based protection may be the most promising approach for many workloads (see Table 3.2 for a summary).

3.3.1 Explicit Lock-and-Key: Change the Lock

In this scheme, each memory allocation is assigned a lock, and each valid pointer to that allocation is assigned the matching key. Figure 3.2 shows the code from Figure 3.1, instrumented such that in line 1, the allocated object gets a new lock (say 42), and the matching key is

[2]A convenient but inaccurate simplification is to consider explicit lock-and-key to mean hardware-based pointer dereferencing checks, and implicit lock-and-key to mean software-based pointer deferencing checks.

```
1  void(**someFuncPtr)() = malloc(sizeof(void*));
2  setLock(someFuncPtr, 42); // Assume hash table or similar
3  long someFuncPtr_key = 42;
4
5  if (someFuncPtr_key != getLock(someFuncPtr) abort();
6  *someFuncPtr = &Elmo;    // At 0x05CADA
7  if (someFuncPtr_key != getLock(someFuncPtr) abort();
8  (*someFuncPtr)();        // Correct use.
9
10 void(**callback)();
11 callback = someFuncPtr;
12 setKey(callback, getKey(someFuncPtr));
13 ...
14 free(someFuncPtr);       // Free space.
15 setLock(someFuncPtr, INVALID_LOCK);
16
17 userName = malloc(...); // Reallocate space.
18 setLock(userName, 43);
19 long userName_key = 43;
20
21 ... // Overwrite object with &Grouch at 0x05DEAD.
22
23 if (callback_key != getLock(callback) abort();
24 (*callback)();          // Use after free!
```

Figure 3.2: The code from Figure 3.1, instrumented with explicit lock-and-key and changing the lock.

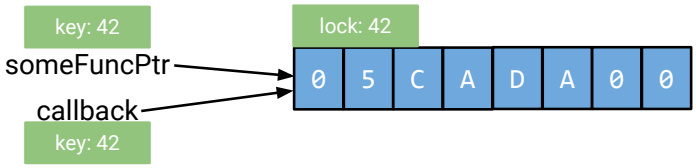


Figure 3.3: Each pointer has a key, each object has a lock.

linked to the pointer (see Figure 3.3). Similarly, in line 11, the key linked to `someFuncPtr` is copied to `callback`. The code is instrumented so that pointer dereferencing (lines 6, 8, and 24) is preceded by a check that the pointer's key matches the object's lock.

When the space is deallocated and reallocated to a new object, the new object is given a new lock (say, 43), and `userName` receives the appropriate key in line 8. The keys for `someFuncPtr` and `callback` no longer match the lock past line 7, avoiding use after free (Figure 3.4).

Since this scheme creates explicit keys (one per pointer), the memory overhead is pro-

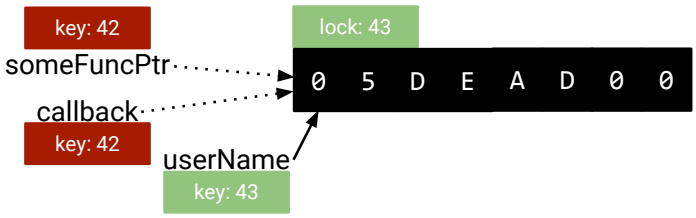


Figure 3.4: Lock change (see Figure 3.3 for the 'Before').

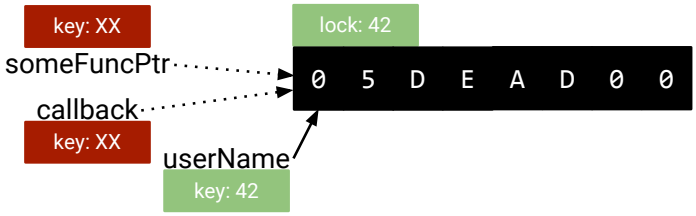


Figure 3.5: Key revocation (see Figure 3.3 for the 'Before').

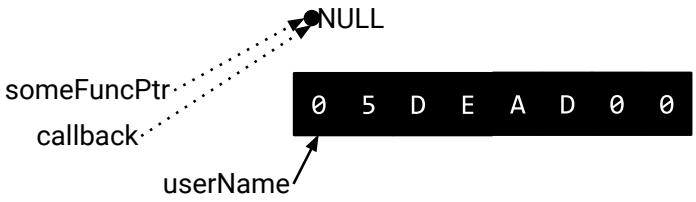


Figure 3.6: After pointer nullification (see Figure 3.1 for the 'Before'), object space can be reused safely.

portional to the number of pointers. The scheme also creates one lock per object, but the number of objects is dominated by the number of pointers.

Example Systems: Compiler-Enforced Temporal Safety for C (CETS) [75] is an example of this scheme. Although in our figure we have placed the key next to the pointer (similar to bounds-checking schemes that store both the pointer plus the size [58], called *plus-size pointers*) and lock next to the object, this need not be the case in implementations. Indeed, one of the key advances of CETS over prior lock-and-key schemes is that it uses a disjoint metadata space, with a separate entry for each pointer that stores the key and the lock location; this avoids changing the memory layout of the program.

```
1  void(**someFuncPtr)() = malloc(sizeof(void*));
2  long someFuncPtr_key = getLock(someFuncPtr);
3  registerKey(someFuncPtr,&someFuncPtr);
4
5  if (someFuncPtr_key != getLock(someFuncPtr) abort();
6  *someFuncPtr = &Elmo;    // At 0x05CADA
7  if (someFuncPtr_key != getLock(someFuncPtr) abort();
8  (*someFuncPtr)();        // Correct use.
9
10 void(**callback)();
11 callback = someFuncPtr;
12 setKey(callback, getKey(someFuncPtr));
13 registerKey(callback,&callback);
14 ...
15 free(someFuncPtr);       // Free space.
16 for key in getRegisteredKeys(someFuncPtr) {
17     setKey(key, INVALID_KEY);
18 }
19
20 userName = malloc(...); // Reallocate space.
21 long userName_key = getLock(someFuncPtr);
22 registerKey(userName,&userName_key);
23
24 ... // Overwrite object with &Grouch at 0x05DEAD.
25
26 if (callback_key != getLock(callback) abort();
27 (*callback)();           // Use after free!
```

Figure 3.7: The code from Figure 3.1, instrumented with explicit lock-and-key and revoking the keys.

3.3.2 Explicit Lock-and-Key: Revoke the Keys

Instead of changing the lock, one could revoke all keys upon reallocation. This requires tracking of keys throughout memory; for example, freeing either `someFuncPtr` *or* `callback` should revoke the keys for both pointers (Figure 3.5).

Figure 3.7 shows the code from Figure 3.1, instrumented accordingly. Upon allocation (line 1) instrumentation must maintain global metadata tracking all pointers to a given object, and this index must be updated at every relevant assignment (line 11). Deallocation (line 15) must be followed by looking up all pointers to that object, revoking (nullifying or otherwise invalidating) their keys. Revoking keys is harder than changing the lock, since it requires keeping track of where all the pointers/keys copied.

Example Systems: To our knowledge, revoking the keys has not been used for any published explicit lock-and-key scheme; but, it segues to the next idea that has been used in prior work: revoking the keys with *implicit* lock-and-key.

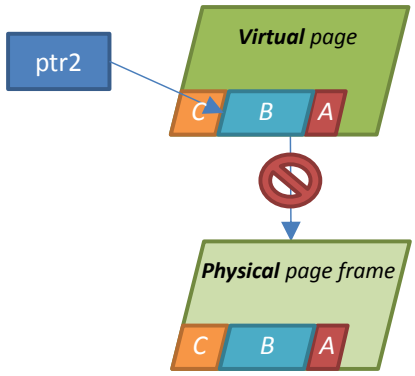


Figure 3.8: The virtual page has been made inaccessible: accesses to objects A, B or C would cause a fault.

3.3.3 Implicit Lock-and-Key: Revoke the Keys

We can view a pointer as the key, and the object as the lock. Thus, instead of revoking a key from a separate *explicit* namespace, we can change the pointer's value [62]. The relevant code instrumentation is similar to the explicit case. Upon allocation or pointer assignment, we update a global index tracking all pointers to each object. Upon deallocation, we find and corrupt the value of all pointers to the deallocated object (Figure 3.6), say by setting them to `NULL`. Pointer dereferences need not be instrumented, since the memory management unit (MMU) performs the null check in hardware.

Although this scheme does not need to allocate memory for explicit lock or key fields, it does need to track the location of each pointer, which means the physical memory overhead is at least proportional to the number of pointers.[3]

Example Systems: DangNull's dangling pointer nullification [62] is an example of this scheme. FreeSentry [105] is similar, but instead of nullifying the address, it flips the top bits, for compatibility reasons (see Section 3.8.3). DangSan [60] is the latest embodiment of this technique; its main innovation is the use of append-only per-thread logs for pointer tracking, to improve runtime performance for multi-threaded applications.

3.3.4 Implicit Lock-and-Key: Change the Lock

Implicit lock-and-key requires less instrumentation than explicit lock-and-key, and changing locks is simpler than tracking and revoking keys. The ideal scheme would therefore be implicit lock-and-key in which locks are changed.

One option is to view the object as a lock, but this lacks a mechanism to "change the lock". Instead, it is more helpful to view the *virtual address* as the lock.

[3]DangSan can use substantially more memory in some cases due to its log-based design.

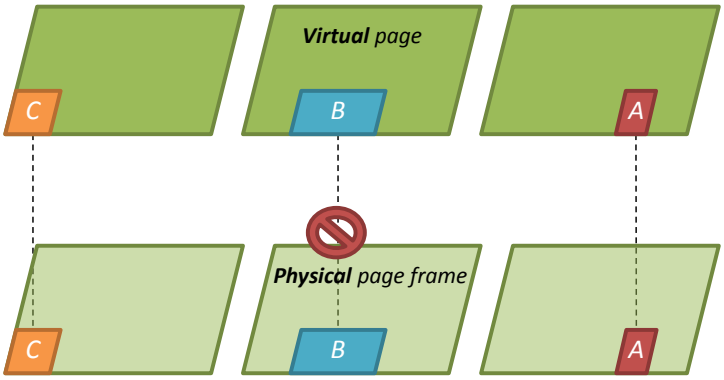


Figure 3.9: With one object per page, we can selectively disable object B.

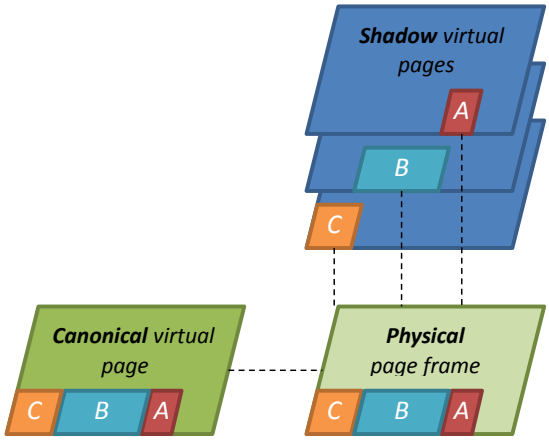


Figure 3.10: Each object has its own shadow virtual page, which all map to the same physical frame.

Recall that objects (and physical memory) are accessed via virtual addresses, which are translated (by the MMU) into physical addresses. By removing the mapping or changing the page permissions, we can make a virtual page inaccessible; the underlying physical memory can then be mapped to a different virtual address (changed lock) for reuse. A drawback is that making a virtual page inaccessible renders *all* objects on that page – often a non-trivial number, since pages are 4KB or larger while many objects are small [7] — inaccessible (Figure 3.8). Additionally, since this scheme requires fresh virtual page addresses, there is additional pressure on the translation-lookaside buffer (TLB; a cache of the mapping from virtual to physical addresses).

Placing one object per page (Figure 3.9) allows selectively disabling individual objects. Unfortunately, it has a significant physical memory overhead, since even a small 8-byte object

would take up 4KB of physical memory; furthermore, it strains the cache (cache lines may be 64-bytes each, which can often hold multiple objects; with one object per page, even an 8-byte object would take up an entire cache line) and the TLB (with one object per page, there are even more virtual addresses introduced than in the basic scheme of the preceding paragraph).

It is not strictly necessary to use page permissions to enforce page inaccessibility after deallocation. In principle, we could maintain a hashtable of live pointers, and instrument all the pointer dereferences to check that the pointer is still live, trading off instrumentation for system calls. This would still have less overhead than an explicit lock-and-key scheme, because we would not need to instrument pointer arithmetic.

Example Systems: Electric Fence [39] implements implicit lock-and-key with changing the lock, by placing one object per physical frame. Its high physical memory usage renders it impractical for anything other than debugging.

Dhurjati and Adve [37] overcame this shortcoming through virtual aliasing. Normally, `malloc` might place multiple objects on one virtual page, which Dhurjati and Adve refer to as the *canonical* virtual page. For each object on the canonical virtual page, they create a *shadow* virtual page that is aliased onto the same underlying physical page frame. This allows each object to be disabled independently (by changing the permissions for the corresponding shadow page), while using physical memory/cache more efficiently than Electric Fence (Figure 3.10). However, this still requires many syscalls and increases TLB pressure. Furthermore, creating shadows introduces compatibility issues with `fork` (Section 3.7.1).

The physical memory overhead — one page table entry, one kernel virtual memory area struct, plus some user-space allocator metadata, per object — is conceptually proportional to the number of live objects. We expect this to be more efficient than the other classes of lock-and-key schemes, which have overhead proportional to the number of pointers (albeit with a smaller constant factor). Some engineering is required to avoid stateholding of page table entries for freed objects (see Section 3.9).

3.3.5 Summary of Lock and Key Schemes

Table 3.2 compares the plausible lock-and-key schemes.

Implicit lock-and-key schemes that change the lock (i.e., one object per virtual page) are advantageous by having no overhead for any pointer arithmetic, and no direct cost (barring TLB and memory pressure) for pointer dereferences. The core technique does not require application source code: for programs using the standard allocator, we need only change the `glibc` `malloc` and `free` functions. Extensions of this scheme may require source code; for example, Dhurjati and Adve [37] requires application source code to apply their static analysis optimization, which allows them to reuse virtual addresses when it is provably safe (see Section 3.2.1).

	Explicit lock-and-key: changing the lock e.g., CETS	Implicit lock-and-key: revoking the keys e.g., DangNull/FreeSentry	Implicit lock-and-key: changing the lock e.g., Electric Fence; D&A; Oscar
Instrumentation			
`malloc ()`	Allocate lock address; Issue key; Set lock	Register pointer	Syscall to create virtual page
Simple ptr arithmetic: `p+=2`	✓ No cost		
General ptr arithmetic: `p=q+1`	Propagate lock address and key	Update ptr registration	✓ No cost
Pointer dereference: `*p`	Check key vs. lock value (at lock address)	✓ No cost	<TLB and memory pressure>
`free ()`	Deallocate lock address	Invalidate pointers	Syscall to disable virtual page
Application source/ recompilation needed?	Yes: needs source + recompilation		✓ In general, no. However, needed by Dhurjati & Adve
Physical memory overhead	O(# pointers)	O(# pointers)	✓ O(# objects)

Table 3.2: Comparison of lock-and-key schemes. Green and a tick indicates an advantageous distinction.

3.4 Baseline Oscar Design

We will develop the shadow virtual pages idea in a direction that does not require source-code analysis, with less stateholding of kernel metadata for freed objects, and with better compatibility with `fork`. We focus on `glibc` and Linux.

While we have argued that page-permissions-based protections should require less instrumentation than newer schemes, there has been no good data on the overhead of shadows (without reliance on static analysis), let alone quantitative comparisons with recent schemes. In the first part of this chapter, we quantify and predict the overhead when using only shadows. These measurements informed our approach for reducing the overhead, which are described in the second part of this chapter.

To help us improve the performance of shadow-page-based schemes, we first measure their costs and break down the source of overhead. Shadow-page schemes consist of four elements: modifying the memory allocation method to allow aliased virtual pages, inline metadata to record the association between shadow and canonical pages, syscalls to create and disable shadow pages, and TLB pressure. We measure how much each contributes to the overhead, so we can separate out the cost of each.

It is natural to hypothesize that syscall overhead should be proportional to the number of `malloc`/`free` operations, as page-permissions-based schemes add one or two syscalls per `malloc` and `free`. However, the other costs, such as TLB pressure, are less predictable, so measurements are needed.

The baseline design, which is similar to Dhurjati and Adve's [37], uses inline metadata to let us map from an object's shadow address to its canonical address. When the pro-

gram invokes `malloc(numBytes)`, we call `libc`'s internal implementation of malloc with a requested size of `numBytes + sizeof(void*)` to allocate an object within a physical page frame. We then immediately perform a syscall to create a shadow page for the object. The object's canonical address is stored as inline metadata within the additional `sizeof(void*)` bytes. This use of inline metadata is transparent to the application, unlike with plus-size pointers. Conceivably, the canonical addresses could instead be placed in a disjoint metadata store (similar to CETS), improving compactness of allocated objects and possibly cache utilization, but we have not explored this direction.

3.4.1 Measurement Methodology

We quantified the overhead by building and measuring incrementally more complex schemes that bridge the design gap from `glibc`'s `malloc` to one with shadow virtual pages, one overhead factor at a time.

Our first scheme simply changes the memory allocation method. As background, `malloc` normally obtains large blocks of memory with the `sbrk` syscall (via the macro `MORECORE`), and subdivides it into individual objects. If `sbrk` fails, `malloc` obtains large blocks using `mmap(MAP_PRIVATE)`. (This fallback use of `mmap` should not be confused with `malloc`'s special case of placing very large objects on their own pages.) We cannot create shadows aliased to memory that was allocated with either `sbrk` or `mmap(MAP_PRIVATE)`; the Linux kernel does not support this. Thus, our first change was MAP_SHARED arenas: we modified `malloc` to always obtain memory via `mmap(MAP_SHARED)` (which can be used for shadows) instead of `sbrk`. This change unfortunately affects the semantics of the program if it `fork()`s: the parent and child will share the physical page frames underlying the objects, hence writes to the object by either process will be visible to the other. We address this issue — which was not discussed in prior work[4] — in Section 3.7.1.

MAP_SHARED with padding further changes `malloc` to enlarge each allocation by `sizeof(void*)` bytes for the canonical address. We do not read or write from the padding space, as the goal is simply to measure the reduced locality of reference.

Create/disable shadows creates and disables shadow pages in the `malloc` and `free` functions using `mremap` and `mprotect(PROT_NONE)` respectively, but does not access memory via the shadow addresses; the canonical address is still returned to the caller. To enable the `free` function to disable the shadow page, we stored the shadow address inside the inline metadata field (recall that in the complete scheme, this stores the canonical).

Use shadows returns shadow addresses to the user. The *canonical* address is stored inside the inline metadata field. This version is a basic reimplementation of a shadow-page scheme.

[4]Independently of and roughly contemporaneously with our work, the LowFat scheme for stack spatial memory safety [38] also encountered the `fork()` issue and described a workaround. We thank Roland Yap for bringing our attention to their work. Their paper also has an interesting connection to our previous chapter: they used aliasing to create multiple virtual stacks, with stack pointers defined implicitly in a manner which they note is similar to parallel shadow stacks.

All timings were run on Ubuntu 14.04 (64-bit), using an Intel Xeon X5680 with 12GB of RAM. We disabled hyper-threading and TurboBoost, for more consistent timings. Our "vanilla" `malloc/free` was from `glibc 2.21`. We compiled the non-Fortran SPEC CPU2006[5] benchmarks using `gcc/g++` v4.8.4 with `-O3`. We configured `libstdc++` with `--enable-libstdcxx-allocator=malloc`, and configured the kernel at run-time to allow more virtual memory mappings.[6]

We counted `malloc` and `free` operations using `mtrace`. We placed `mtrace` at the start of `main`, which does miss a small number of allocations (e.g., static initializers and constructors for global C++ objects), but these are insignificant: as will be shown in Figure 3.12, the overhead is roughly proportional to the number of `malloc`s/`free`s, so the allocations we have missed would not, if properly instrumented, add substantially to the overhead.

3.4.2 Results

The overhead measurements of the four incrementally more complete schemes are shown in Figure 3.11 for 15 of the 19 SPEC CPU2006 C/C++ benchmarks. The remaining four benchmarks (`perlbench`, `dealII`, `omnetpp`, `xalancbmk`) exhaust the physical memory on the machine when creating/disabling shadows, due to the accumulation of `vm_area_struct`s corresponding to `mprotect`'ed pages of "freed" objects. We therefore defer discussion of them until the following section, which introduces our improvements to the baseline design. We ran at least 9 runs of each benchmark.

Even for the complete but unoptimized scheme (Use shadows), most benchmarks have low overhead. `gcc` and `sphinx` have high overhead due to creating/destroying shadows, as well as using shadows. `astar` and `povray` have a noticeable cost mainly due to using shadows, a cost which is not present when merely creating/disabling shadows; we infer that the difference is due to TLB pressure. Notably, `mcf`'s overhead is entirely due to MAP_SHARED arenas, as is most of `milc`'s. Inline padding is a negligible cost for all benchmarks.

In Figures 3.12 and 3.13, we plot the run-time of creating/disabling shadows, against the number of shadow-page-related syscalls.[7] We calculated the y-values by measuring the runtime of Create/disable shadows (to ensure all benchmarks complete successfully, we used an optimization – the "high water mark" optimization – which will be discussed later in Section 3.5) *minus* MAP_SHARED with padding: this discounts runtime that is not associated with syscalls for shadows. The high correlation matches our mental model that each syscall has an approximately fixed cost, though it is clear from `omnetpp` and `perlbench` that it is not perfectly fixed. Also, we can see that `perlbench`, `dealII`, `omnetpp` and `xalancbmk` each create over 100 million objects, which is why they could not run to completion using the unoptimized implementation: they would require over 19 gigabytes of physical memory (see Section 3.5.1).

[5] `https://www.spec.org/cpu2006/`

[6] `sudo sysctl -w vm.max_map_count=100000000`

[7] A `realloc` operation involves both creating a shadow and destroying a shadow, hence the number of `malloc/free` operations is augmented with `(2 * realloc)`.

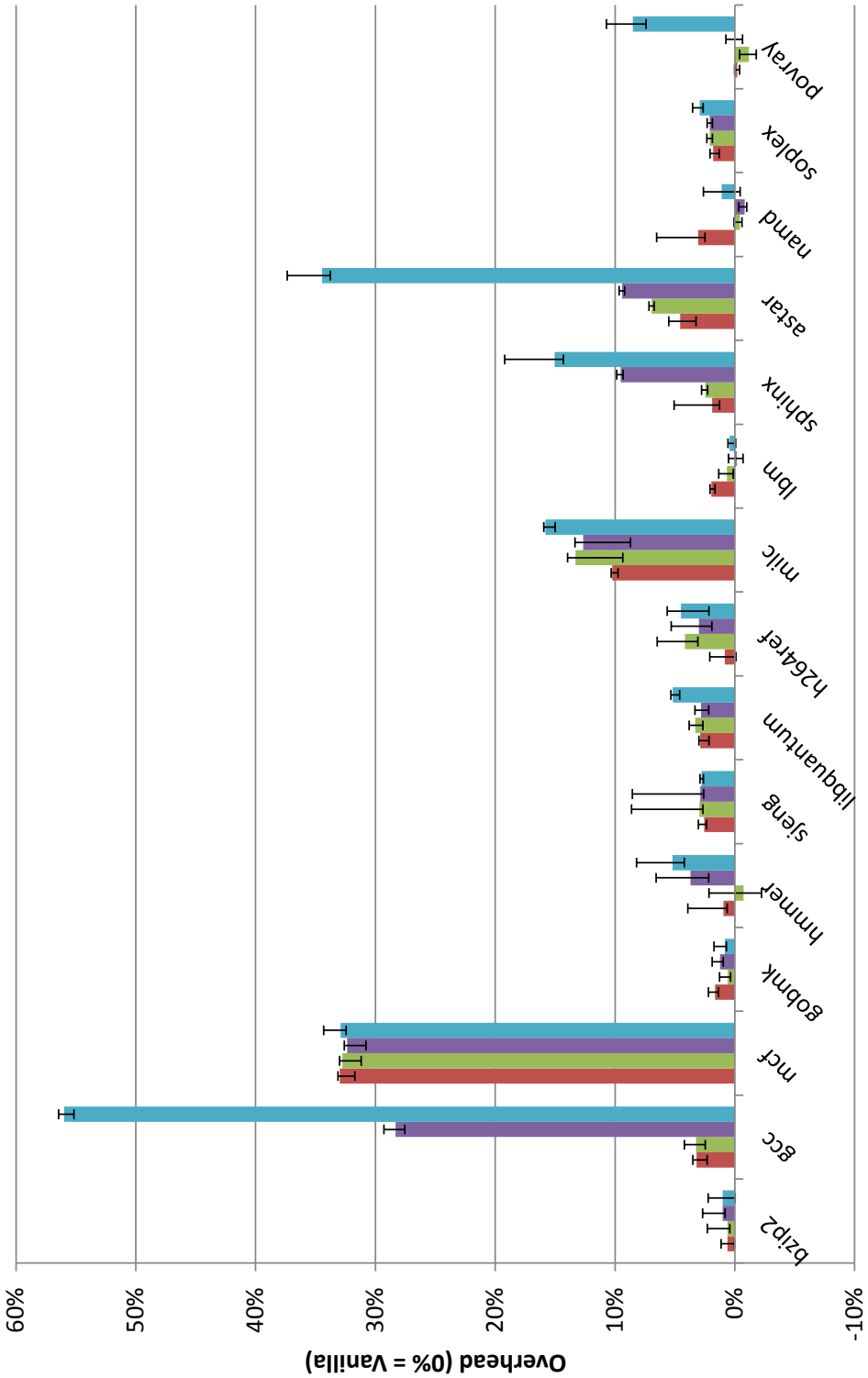


Figure 3.11: SPEC CPU2006 C/C++ benchmarks, showing the overhead as we reach the full design.

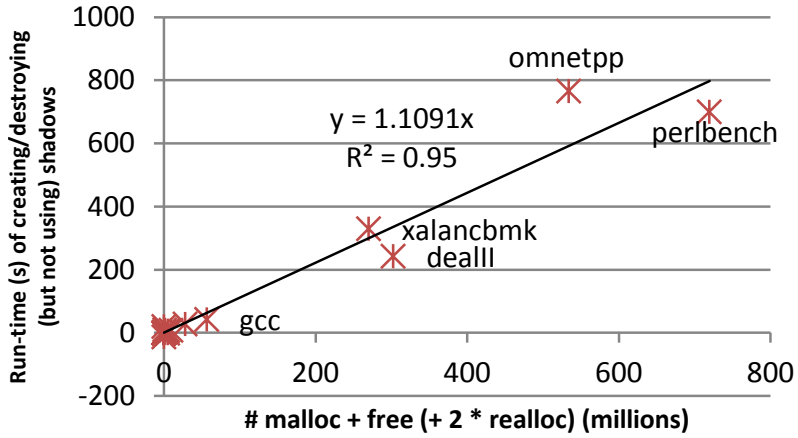


Figure 3.12: Predicting syscall overhead. See Figure 3.13 for a magnified view of the bottom-left.

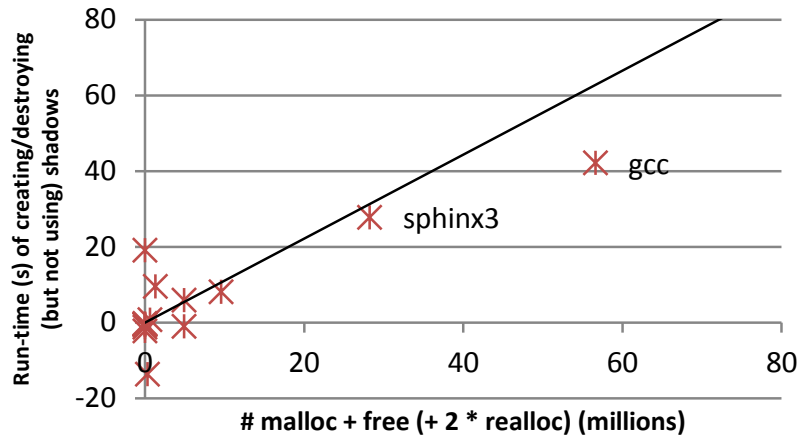


Figure 3.13: Predicting syscall overhead, magnifying the bottom-left of Figure 3.12.

3.5 Lowering Overhead Of Shadows

The previous section shows that the overhead is due to `MAP_SHARED`, creating/destroying shadows, and using shadows. The TLB pressure cost of using shadows can be reduced with hardware improvements, such as larger TLBs (see Section 3.8.2). In this section, we propose, implement, and measure three optimizations for reducing the first two costs.

3.5.1 High water mark

The naïve approach creates shadows using `mremap` without a specified address and disables shadows using `mprotect(PROT_NONE)`. Since disabled shadows still occupy virtual address space, new shadows will not reuse the addresses of old shadows, thus preventing use-after-free of old shadows. However, the Linux kernel maintains internal data structures for these

shadows, called `vm_area_structs`, consuming 192 bytes of kernel memory per shadow. The accumulation of `vm_area_structs` for old shadows prevented a few benchmarks (and likely would prevent many real-world applications) from running to completion: we observed that the system was thrashing.

We introduce a simple solution. Contrary to conventional wisdom [37], with a small design modification, Oscar can both unmap and prevent reuse of a virtual page. We use a "high water mark" for shadow addresses: when Oscar creates a shadow, we specify the high water mark as the requested shadow address, and then increment the high water mark by the size of the allocation. This is similar to the `sbrk` limit of `malloc`. Oscar can now safely use `munmap` to disable shadows and free kernel memory, without risk of reusing old shadows. As we show in Section 3.8.1, virtual address space exhaustion is an unlikely, tractable problem.

Our scheme, including the high water mark, is compatible with address space layout randomization (ASLR) [83]. At startup, we initialize the high water markat a fixed offset to the (randomized) heap base address. To reduce variability in run-times, all benchmarks, including the baseline, were measured without ASLR, as is typical in similar research [99].

3.5.2 Refreshing shadows

Figure 3.14 (left) depicts the simplified circle of life of a heap-allocated chunk of physical memory. Over the lifetime of a program, that chunk may be allocated, freed, allocated, freed, and so forth, resulting in syscalls to create a shadow, destroy a shadow, create a shadow, destroy a shadow. Except for the very first time a chunk has been created by `malloc`, every shadow creation is preceded by destroying a shadow.

Oscar therefore speculatively creates a new shadow each time it destroys a shadow, in Figure 3.14 (right). This saves the cost of creating a new shadow, the next time an object is allocated on that canonical page. The opportunistically renewed shadow is stored in a hash table, keyed by the size of shadow (in number of pages) and the address of the canonical *page* (not the canonical object). This means the shadow address can be used for the next similarly-sized object allocated on the canonical page(s), even if the new object does not coincide precisely with the old object's size or offset within the page. It also improves the likelihood that the shadow can be used when objects are coalesced or split by the allocator.

Up to now, we have used `mremap` to create shadows. `mremap` actually can be used to both destroy an old mapping and create a new virtual address mapping (at a specified address) in a single system call. We use this ability to both destroy the old shadow mapping and create a new one (i.e., refresh a shadow) with one system call, thereby collapsing 2 system calls to 1 system call. This optimization depends on the high water mark optimization: if we called `mremap` with `old_size = new_size` without specifying a `new_address`, `mremap` would conclude that there is no need to change the mappings at all, and would return the old shadow virtual address.

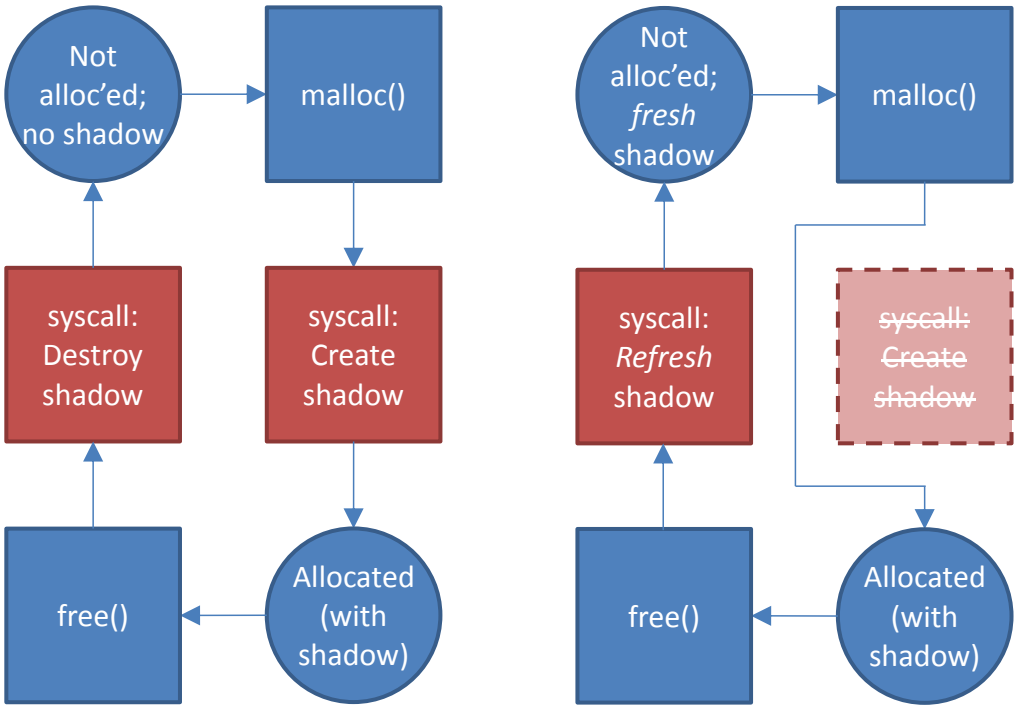


Figure 3.14: Left: Simplified lifecycle of a chunk of memory. Right: The `destroyShadow` syscall has been modified to simultaneously destroy the old shadow and create a new one.

3.5.3 Using MAP_PRIVATE when possible

As mentioned earlier, `MAP_SHARED` is required for creating shadows, but sometimes has non-trivial costs. However, for large objects that `malloc` places on their own physical page frames, Oscar does not need more than one shadow per page frame; in fact, Oscar can avoid using shadow addresses at all — since the canonical addresses will never be reused — though we use shadows for consistency of implementation. For these large allocations, Oscar uses `MAP_PRIVATE` mappings.

Implementing `realloc` correctly requires care. Our ordinary `realloc_wrapper` is, in pseudo-code:

```
1 munmap(old_shadow);
2 new_canonical = internal_realloc(old_canonical);
3 new_shadow = create_shadow(new_canonical);
```

This works when all memory is `MAP_SHARED`. However, if the reallocated object `new_canonical` is large enough to be stored on its own `MAP_PRIVATE` pages, `create_shadow` will allocate a different set of physical page frames instead of creating an alias; this then necessitates copying the contents of the object from `new_canonical` to the new page frames for `new_shadow`. It is possible to avoid this inefficient copying step if Oscar simply did not use the (unnecessary) shadows for large allocations. Our prototype of Oscar incurs this copying cost,

but it does not substantially affect our results, since few of the SPEC CPU2006 benchmarks use `realloc` extensively.[8]

The overhead reduction is upper-bounded by the original cost of MAP_SHARED arenas.

3.5.4 Abandoned approach: Batching system calls

We tried batching the creation or deletion of shadows, but did not end up using this approach in Oscar.

We implemented a custom syscall (loadable kernel module ioctl) to create or destroy a batch of shadows. When we have no more shadows for a canonical page, we call our `batchCreateShadow` ioctl once to create 100 shadows, reducing the amortized context switch cost per malloc by 100x. However, this does not reduce the overall syscall cost by 100x, since `mremap`'s internals are costly. In a microbenchmark, creating and destroying 100 million shadows took roughly 90 seconds with individual `mremap`/`munmap` calls (i.e., 200 million syscalls) vs. ≈80 seconds with our batched syscall. The savings of 10 seconds was consistent with the time to call a no-op ioctl 200 million times.

In our pilot study, batching did not have a significant benefit. It even slowed down some benchmarks, due to mispredicting which shadows will be needed in the future. For example, we may create 100 shadows for a page that contains solely of a single object which is never freed, wasting 99 shadows.

We also tried batch-disabling shadows: any objects that are `free()`'d are stored in a "quarantine" of 100 objects, and when the quarantine becomes full, we disable all 100 shadows with a single batched syscall, then actually free those 100 objects. This approach maintains temporal memory safety, unlike the standard use of quarantine (see Section 3.2). Unlike batch-creating shadows, with batch-deletion we need not predict the future.

In our pilot study, batch deletion had mixed effects on runtime overhead. We hypothesize this is due to disrupting favorable memory reuse patterns: `malloc` prefers to reuse recently freed objects, which are likely to be hot in cache; quarantine prevents this.

Batch deleting could also replaced with batch refreshing; this inherits the misprediction issues of both batch creation and refreshing.

3.6 Performance Evaluation

The effect of these improvements on the previous subset of 15 benchmarks is shown in Figure 3.15.

Our first two optimizations (high water mark, refreshing shadows) greatly reduce the overhead for `gcc` and `sphinx`; this is not a surprise, as we saw from Figure 3.11 that much of `gcc` and `sphinx`'s overhead is due to creating/destroying shadows. These two optimizations do not benefit `mcf`, as its overhead was entirely due to MAP_SHARED arenas; instead, fortuitously, the overhead is eliminated by the `MAP_PRIVATE` optimization. The `MAP_PRIVATE`

[8] We ran each benchmark while logging the `realloc` operations.

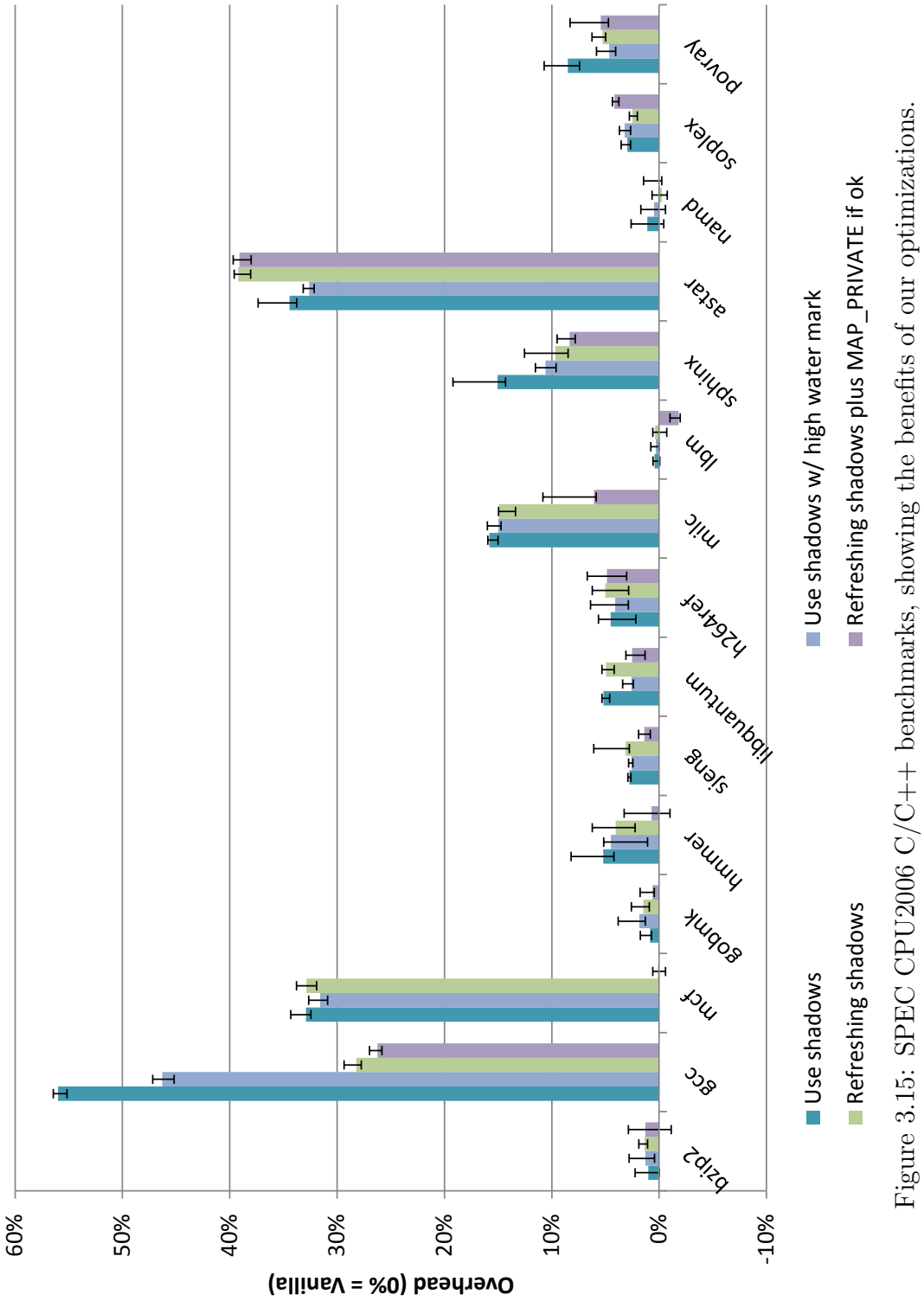


Figure 3.15: SPEC CPU2006 C/C++ benchmarks, showing the benefits of our optimizations.

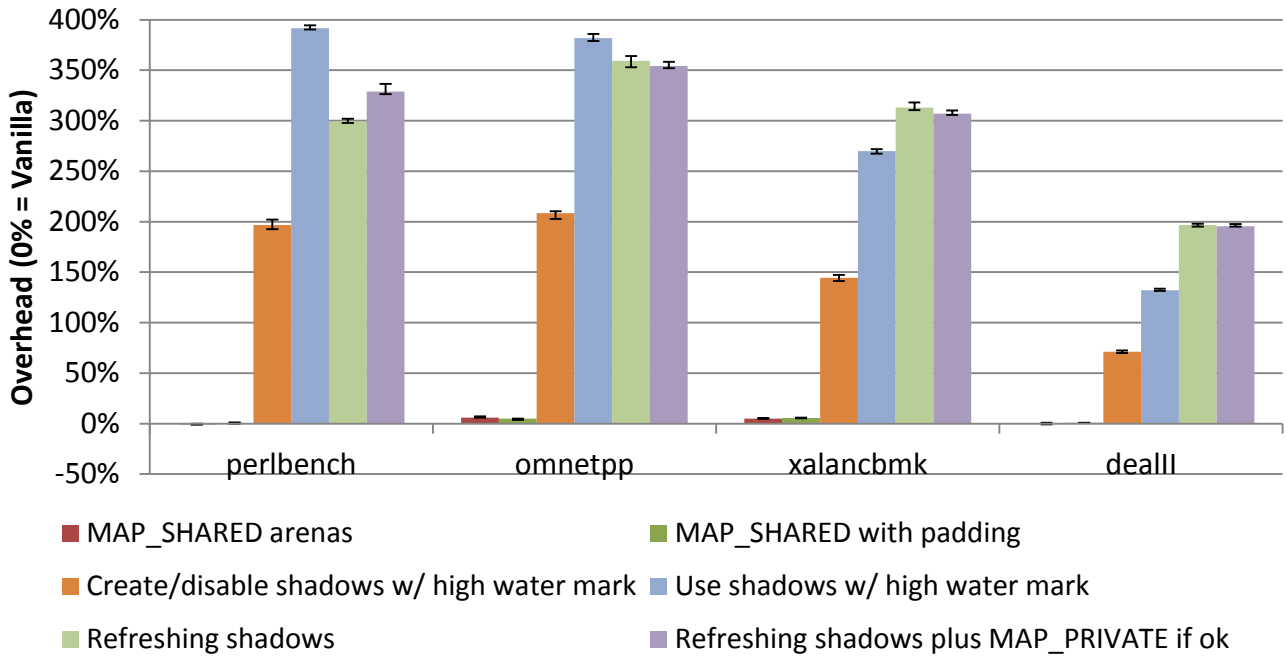


Figure 3.16: The 4 allocation-intensive benchmarks. Note that the first two columns are near zero for each benchmark.

optimization also reduces the overhead on `milc` by roughly ten percentage points, almost eliminating the overhead attributed to `MAP_SHARED`.

The four allocation-intensive benchmarks (`perlbench`, `omnetpp`, `xalancbmk`, `dealII`) are shown in Figure 3.16. Recall that for these benchmarks, the baseline scheme could not run to completion, because of the excessive number of leftover `vm_area_structs` for `mprotect`'ed shadows corresponding to "freed" objects. The high water mark optimization, which permanently `munmaps` the shadows, allows Linux to reclaim the `vm_area_structs`, reducing the memory utilization significantly and enabling them to complete successfully. To separate out the cost of syscalls from TLB pressure, we backported the high water mark change to Create/disable shadows.

For all four benchmarks, `MAP_SHARED` and inline metadata costs (the first two columns) are insignificant compared to creating/disabling and using shadows. Refreshing shadows reduces overhead somewhat for `perlbench` and `omnetpp` but increases overhead for `xalancbmk` and `dealII`.

The `MAP_PRIVATE` optimization had a negligible effect, except for `perlbench`, which became 30 percentage points slower. This was initially surprising, since in all other cases, `MAP_PRIVATE` is faster than `MAP_SHARED`. However, recall that Oscar also had to change the `realloc` implementation. `perlbench` uses `realloc` heavily: 11 million calls, totaling 700GB of objects; this is 19× the `reallocs` of all other 18 benchmarks *combined* (by calls or GBs of objects). We confirmed that `realloc` caused the slowdown, by modifying Refreshing shadows

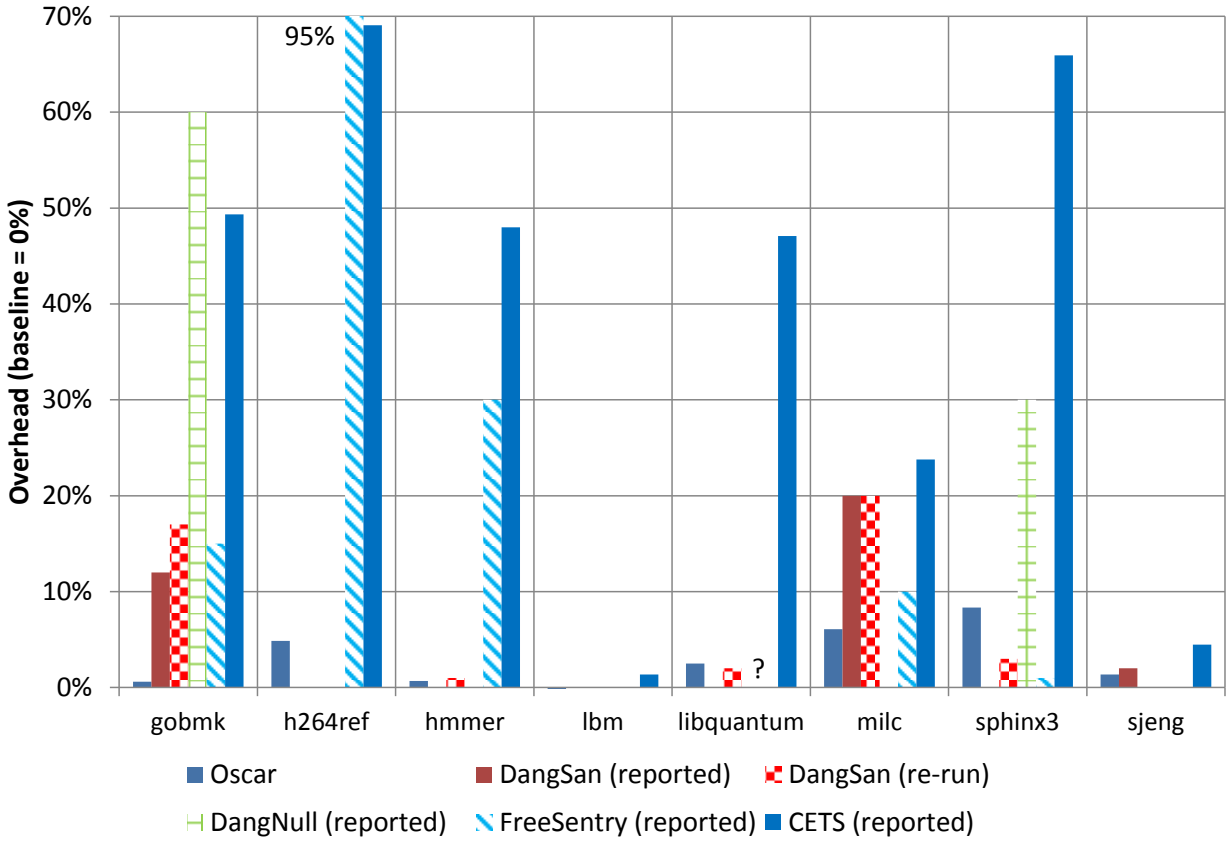


Figure 3.17: Runtime overhead of SPEC benchmarks against DangSan, DangNull, FreeSentry, and CETS. Some overheads are based on results reported in the papers, not re-runs (see legend). '?' indicates that FreeSentry did not report results for `libquantum`.

to use the inefficient `realloc` but with `MAP_SHARED` always; this was marginally slower than refreshing shadows and using `MAP_PRIVATE` where possible.

3.6.1 Runtime and Memory Overhead Comparison

Figure 3.17 compares the runtime overhead of Oscar against DangSan, DangNull, FreeSentry, and CETS. Figure 3.18 shows the remaining SPEC benchmarks, for which results were reported by DangSan and DangNull, but not by FreeSentry or CETS.

A caveat is that CETS' reported overheads are based on providing temporal protection for both the stack and heap, which is more comprehensive than Oscar's heap-only protection. However, since CETS must, to a first approximation, fully instrument pointer arithmetic and dereferencing instructions even if only heap protection is desired, we expect that the overhead of heap-only CETS would still be substantially higher than Oscar.

All other comparisons (DangSan, DangNull, FreeSentry) are based on the appropriate

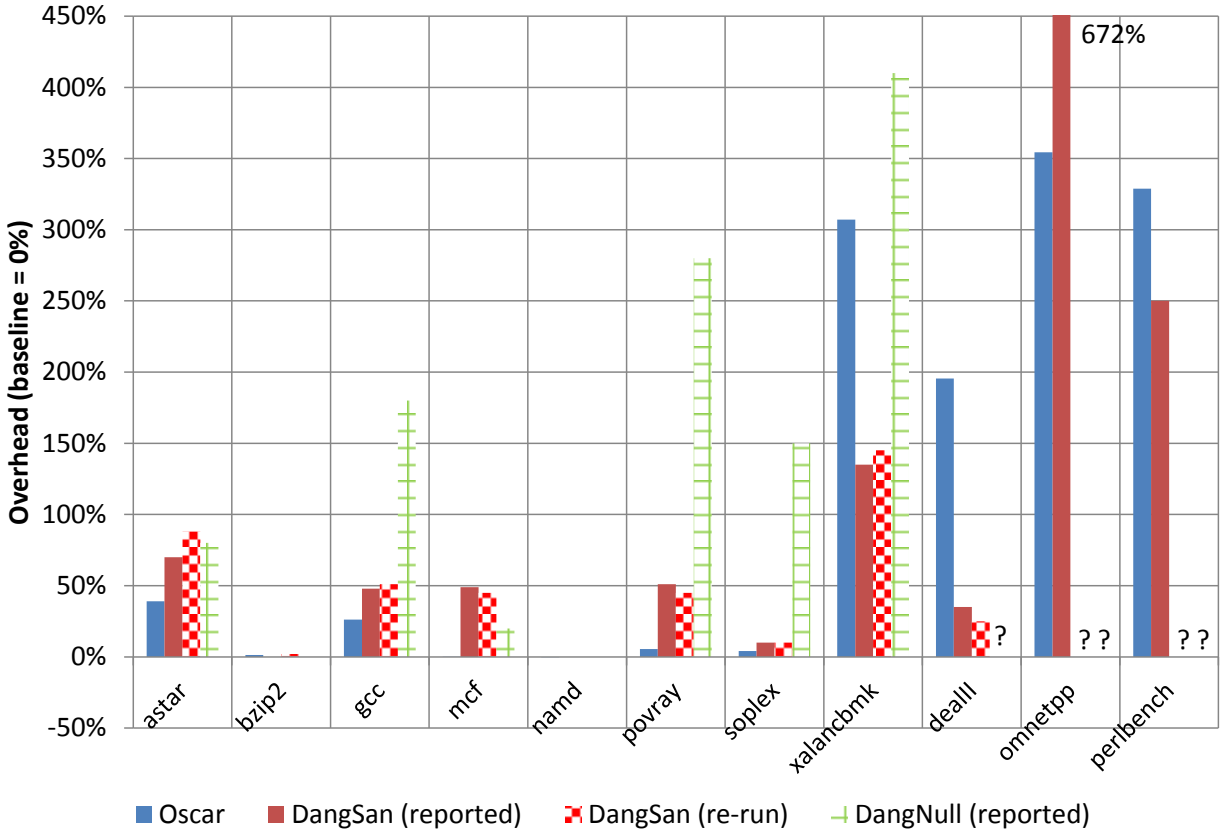


Figure 3.18: Runtime overhead of the remaining SPEC benchmarks. The y-axis differs from Figure 3.17. Results were reported by DangSan and DangNull, but not FreeSentry or CETS. Some overheads are based on results reported in the papers, not re-runs (see legend). '?' indicates that DangNull did not report results for `dealII`, `omnetpp`, or `perlbench`, and we could not re-run DangSan on `omnetpp` or `perlbench`.

reported overheads for heap-only temporal protection.

Comparison to DangSan

We re-ran the latest publicly available version of DangSan[9] on the same hardware as Oscar. DangSan re-run overheads were normalized to a re-run with their "baseline LTO" script. We were unable to re-run `perlbench` due to a segmentation fault, or `omnetpp` due to excessive memory consumption.[10] As seen in the graphs, our re-run results are very similar to DangSan's reported results; thus, unless otherwise stated, we will compare Oscar against the latter.

[9]March 19, 2017,
`https://github.com/vusec/dangsan/commit/78006af30db70e42df25b7d44352ec717f6b0802`

[10]We estimate that it would require over 20GB of memory, taking into account the baseline memory usage on our machine and DangSan's reported overhead for `omnetpp`.

Across the complete set of C/C++ SPEC CPU2006 benchmarks, Oscar and DangSan have the same overall overhead, within rounding error (geometric means of 40% and 41%). However, for all four of the allocation-intensive benchmarks, as well as `astar` and `gcc`, the overheads of both Oscar and DangSan are well above the 10% overhead threshold [97], making it unlikely that either technique would be considered acceptable. If we exclude those six benchmarks, then Oscar has average overhead of 2.5% compared to 9.9% for DangSan. Alternatively, we can see that, for five benchmarks (`mcf`, `povray`, `soplex`, `gobmk`, `milc`), Oscar's overhead is 6% or less, whereas DangSan's is 10% or more. There are no benchmarks where DangSan has under 10% overhead but Oscar is 10% or more.[11]

Comparison to DangNull/FreeSentry

We emailed the first authors of DangNull and FreeSentry to ask for the source code used in their papers, but did not receive a response. Our comparisons are therefore based on the numbers reported in the papers rather than by re-running their code on our system. Nonetheless, the differences are generally large enough to show trends. In many cases, Oscar has almost zero overhead, implying there are few `malloc`s/`free`s (the source of Oscar's overhead); we expect the negligible overhead generalizes to any system. Oscar does not instrument the application's pointer arithmetic/dereferencing, which makes its overhead fairly insensitive to compiler optimizations. We also note that DangSan – which we were able to re-run and compare against Oscar — *theoretically* should have better performance than DangNull because of DangSan's optimized pointer tracking.[12]

Oscar's performance is excellent compared to FreeSentry and DangNull, even though DangNull provides less comprehensive protection: DangNull only protects pointers to heap objects *if* the pointer is itself stored on the heap. Figure 3.17 (left) compares all SPEC CPU2006 benchmarks for which DangNull and FreeSentry both provide data. FreeSentry has higher overhead for several benchmarks (`milc`, `gobmk`, `hmmer`, `h264ref`) – especially higher for the latter three. FreeSentry is faster on the remaining three benchmarks, but in all those cases except for `sphinx3`, our overhead is negligible anyway. DangNull has much higher overhead than Oscar for `gobmk` and `sphinx3`. For other benchmarks, DangNull often gets zero overhead, though it is not much lower than Oscar's, and comes with the caveat of their weaker protection.

Our comparisons are based on our overall "best" scheme with all three optimizations. For some benchmarks, using just the high water mark optimization and not the other two optimizations would have performed better. Even the basic shadow pages scheme without optimizations would often beat DangNull/FreeSentry.

Figure 3.18 shows additional SPEC CPU2006 benchmarks for which DangNull reported their overhead but FreeSentry did not. For the two benchmarks where DangNull has zero overhead (`bzip2`, `namd`), Oscar's are also close to zero. For the other six benchmarks, Oscar's overhead is markedly lower. Two highlights are `soplex` and `povray`, where DangNull's

[11] Of course, there is a wide continuum of "under 10%", and those smaller differences may matter.

[12] However, DangSan's empirical comparisons to DangNull and FreeSentry were also based on reported numbers rather than re-runs.

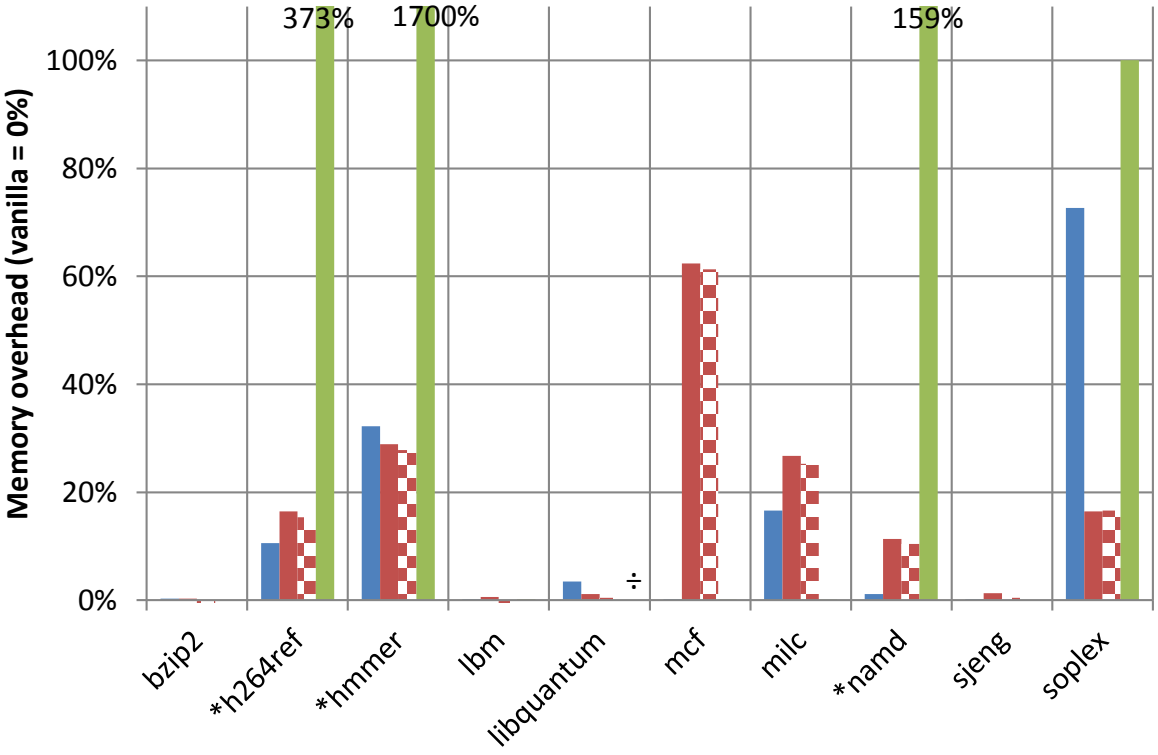


Figure 3.19: Memory overhead on CPU2006. DangNull reported a baseline of 0MB for libquantum, so an overhead ratio is not calculable.

overhead is 150%/280%, while Oscar's is under 6%.

When considering only the subset of CPU2006 benchmarks that DangNull reports results for (i.e., excluding `dealII`, `omnetpp` and `perlbench`), Oscar has a geometric mean runtime overhead of 15.4% compared to 49% for DangNull. For FreeSentry's subset of reported benchmarks, Oscar has just 2.8% overhead compared to 18% for FreeSentry.

Comparison to CETS

We compare Oscar to the temporal-only mode of SoftBoundCETS [73] (which we will also call "CETS" for brevity), since that has lower overhead and a more comprehensive dataset than the original CETS paper.

The latest publicly available version of SoftBoundCETS for LLVM 3.4[13] implements both temporal and spatial memory safety. We received some brief advice from the author of SoftBoundCETS on how to modify it to run in temporal-only mode, but we were unable to get it to work beyond simple test programs. Thus, our comparisons rely on their reported numbers rather than a re-run.

We have omitted the `bzip2` and `mcf` benchmarks, as CETS' bzip2 is from the CPU2000

[13]September 19, 2014, `https://github.com/santoshn/softboundcets-34/commit/9a9c09f04e16f2d1ef3a906fd138a7b89df44996`

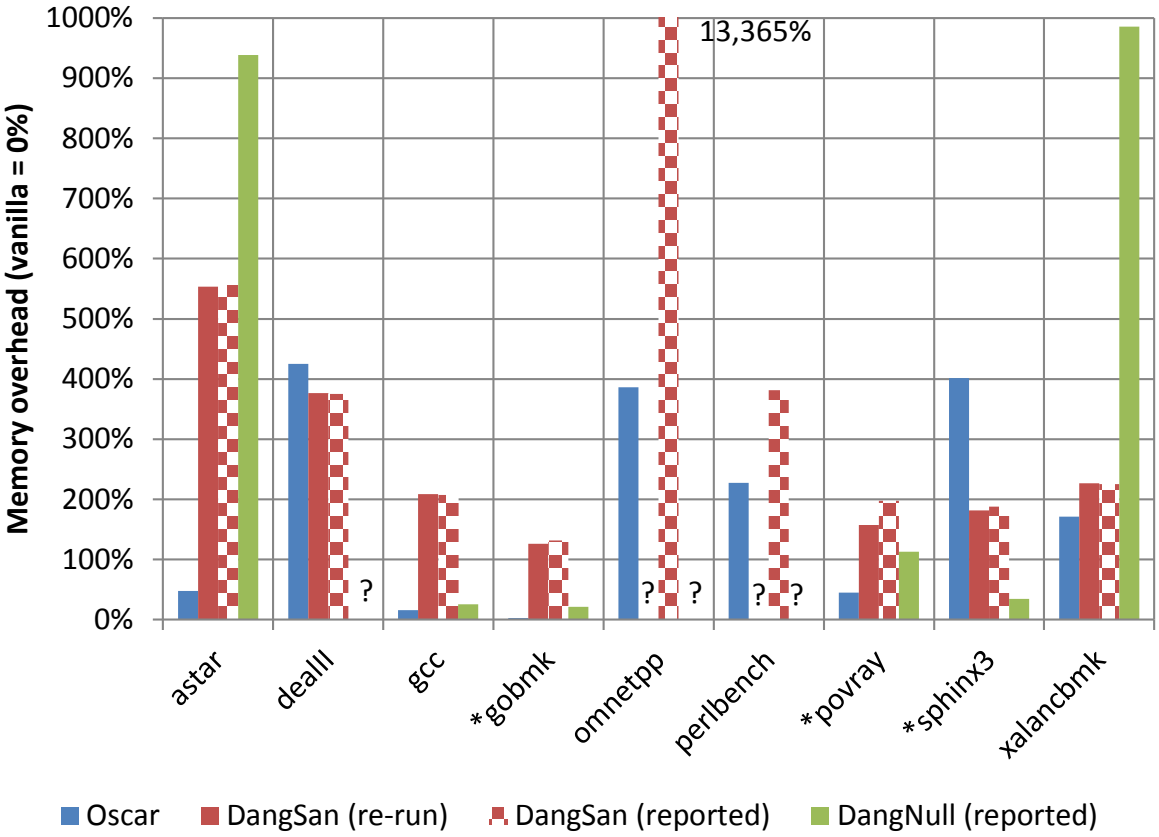


Figure 3.20: Memory overhead on CPU2006 (continued). '?' indicates that DangNull did not report memory usage for `dealII`, `omnetpp`, or `perlbench`, and we could not re-run DangSan on the latter two.

suite [72] and we suspect their mcf is as well.[14] SPEC specifically cautions that, due to differences in the benchmark workload and/or source, the results on CPU2000 vs. CPU2006 might not be comparable [28].

Figure 3.17 shows the overhead of CETS vs. our overall best scheme. We are faster than CETS for all benchmarks, often by a significant margin. For example, CETS has >48% overhead on `gobmk` and `hmmer`, compared to less than 1% for Oscar. The geometric mean across CETS' subset of CPU2006 benchmarks is 2.8% for Oscar compared to 36% for CETS.

3.6.2 Memory Overhead Comparison

Figures 3.19 and 3.20 show the memory overhead of Oscar, DangSan (re-run and reported), and DangNull (reported only). We did not find any reported data for FreeSentry, CETS or

[14] In any case, since CETS has 23% and 114% overhead on `bzip2` and `mcf` respectively — compared to less than 1.5% on each for Oscar – including them in the comparison would not be favorable to CETS.

SoftBoundCETS temporal-only. The graphs have different y-axes to highlight differences in overheads in the lower-overhead benchmarks of Figure 3.19.

We calculated the memory overhead based on the combined maximum resident set size (RSS)[15], size of the page tables[16], and approximate size of the `vm_area_structs`[17]. Our polling approach introduces some minor inaccuracies with regard to obtaining the maxima and baseline values. For DangSan, which does not greatly increase the number of page table entries or `vm_area_structs`, this is very similar to their maximum resident set size metric. It is unclear what memory consumption metric DangNull used, so some care should be taken when interpreting their overheads.

The RSS values reported in `/proc/pid/status` are misleading for Oscar because it double-counts every shadow page, even though many of them are aliased to the same canonical. We know, however, that the physical memory usage of Oscar — and therefore the resident set size when avoiding double-counting — is essentially the same as the MAP_SHARED with padding scheme (from Section 3.4.1). We therefore calculated the maximum RSS for that scheme, but measured the size of the page tables and `vm_area_structs` for the full version of Oscar. We were unable to measure the size of the hashtables used for caching refreshed shadows, although our second-best scheme (omitting the refreshing shadows optimization) has very similar runtime overhead while not requiring those hashtables.

For the complete suite of CPU2006 benchmarks, Oscar has 61.5% memory overhead, far lower than DangSan's 140%. Even if we omit DangSan's pathological case of `omnetpp` (reported overhead of over 13,000%), Oscar is still far more memory-efficient with 52% overhead vs. 90% for DangSan. The only benchmarks on which Oscar performs substantially worse than DangSan are `sphinx3` and `soplex`. `sphinx3` with Oscar has a maximum RSS of ≈50MB (compared to a baseline of ≈45MB), maximum page-table size of ≈130MB, and maximum `vm_area_structs` of ≈45MB. In Section 3.9, we propose methods to reduce the memory overhead by garbage collecting old page table entries (which would reduce Oscar's overhead on `sphinx3`), and sharing inline metadata (which would reduce Oscar's overhead on `soplex` with its many small allocations).

DangNull has roughly 127% memory overhead, but, as also noted by the DangSan authors, DangNull did not report data for many of the memory-intensive benchmarks. If we use the same subset of SPEC benchmarks that DangNull reported, then Oscar has only 36% memory overhead (vs. ≈75% for DangSan).

3.7 Extending Oscar for Server Applications

When applying Oscar to server applications — which are generally more complex than the SPEC CPU benchmarks — we encountered two major issues that resulted in incompatibility

[15] `VmHWM` (peak RSS) in `/proc/pid/status`

[16] `VmPTE` and `VmPMD` in `/proc/pid/status`

[17] We counted the number of mappings in `/proc/pid/maps` and multiplied by `sizeof(vm_area_struct)`.

and incomplete protection: forking and custom memory allocators. Additionally, we modified Oscar to be thread-safe when allocating shadows.

3.7.1 Supporting shadows + fork()

Using `MAP_SHARED` for all allocations is problematic for programs that `fork`, as it changes the semantics of memory: the parent and child's memory will be shared, so any post-`fork` writes to pre-`fork` heap objects will unexpectedly be visible to both the parent and child. In fact, we discovered that most programs that `fork` and use `glibc`'s `malloc` will crash when using `MAP_SHARED`. Surprisingly, they may crash even if neither the parent nor child read or write to the objects post-`fork`.[18]

Oscar solves this problem by wrapping `fork` and emulating the memory semantics the program is expecting. After `fork`, in the child, we make a copy of all heap objects, unmap their virtual addresses from the shared physical page frames, remap the same virtual addresses to new (private) physical page frames (with `MAP_SHARED`), and repopulate the new physical page frames with our copy of the heap objects. The net effect is that the shadow and canonical virtual addresses have not changed — which means old pointers (in the application, and in the allocator's free lists) still work — but the underlying physical page frames in the child are now separated from the parent.

Method.

Oscar instruments `malloc` and `free` to keep a record of all live objects in the heap and their shadow addresses. Note that with a loadable kernel module, Oscar could avoid recording the shadow addresses of live objects and instead find them from the page table entries or `vm_area_struct`s.

Then, Oscar wraps `fork` to do the following:

1. call the vanilla `fork()`. After this, the child address space is correct, except that the `malloc`'d memory regions are aliased with the parent's physical page frames.

2A. [naïve algorithm] in the child process:

   a) map a new page at any unused address `t`
   b) for each `canonical_page` in the heap:
      i. copy the contents of `canonical_page` to `t`
      ii. unmap `canonical_page`
      iii. allocate a new page at address `canonical_page` using `mmap(MAP_SHARED | MAP_ANONYMOUS)`

[18] We observed this behavior in `glibc` 2.21; we expect it to generalize to other versions, based on `glibc`'s design whereby `malloc` stores the main heap state in a static variable (not shared between parent and child), but also partly through inline metadata of heap objects (shared); thus, when the parent or child allocates memory post-`fork`, the heap state can become inconsistent or corrupted. A program that simply `malloc()`s 64 bytes of memory, `fork()`s, and then allocates another 64 bytes of memory in the child, is sufficient to cause an assertion failure.

iv. copy the contents of `t` to `canonical_page`

c) for each live object, recreate a shadow at the same virtual address as before (using the child's new physical page frames):

i. `munmap (oldShadow, length);`

ii. `newShadow = mremap (canonical, 0, length, MREMAP_MAYMOVE | MREMAP_FIXED, oldShadow);`

d) unmap page `t`

2B. [optimized algorithm] in the child process:

a) for each `canonical_page` in the heap:

i. allocate a new page at any unused address `t` using `mmap(MAP_SHARED | MAP_ANONYMOUS)`

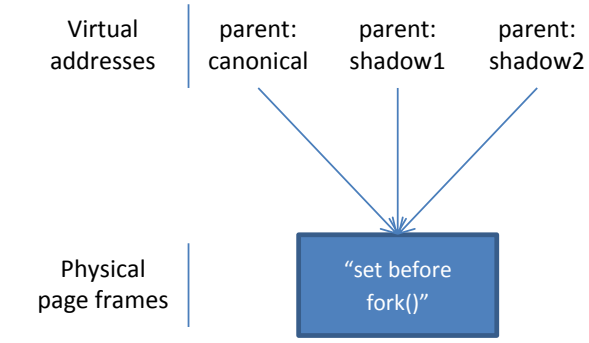
ii. copy the contents of `canonical_page` to `t`

iii. call `mremap(old_address=t, new_address=canonical_page)`. Note that `mremap` automatically removes the previous mapping at `canonical_page`.

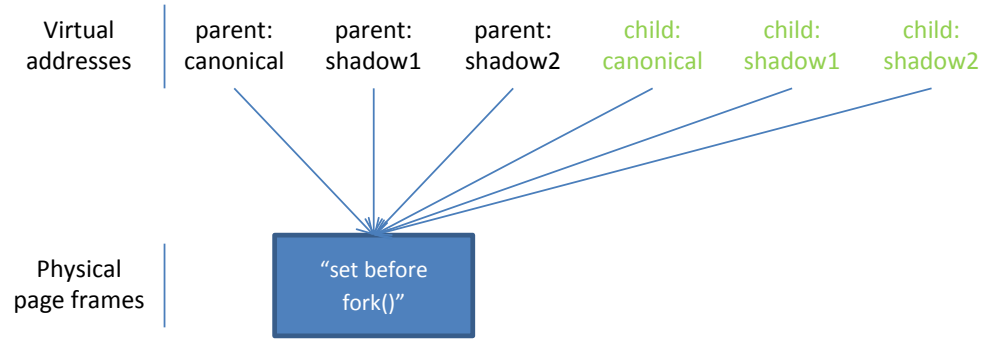
b) for each live object: as per corresponding step of the naïve algorithm

Compared to the naïve algorithm, the use of `mremap` in the optimized algorithm halves the number of memory copy operations. We illustrate the algorithm in Figure 3.3.

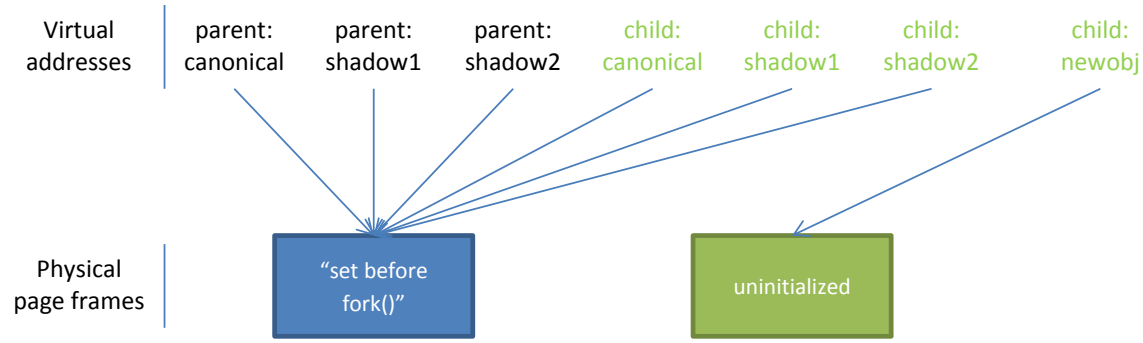
Table 3.3: Illustrated guide to laundering.



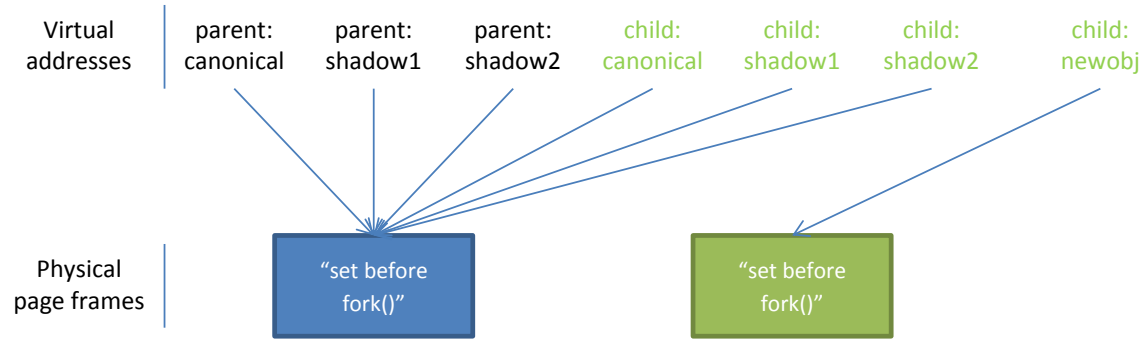
Step 1: initial configuration.



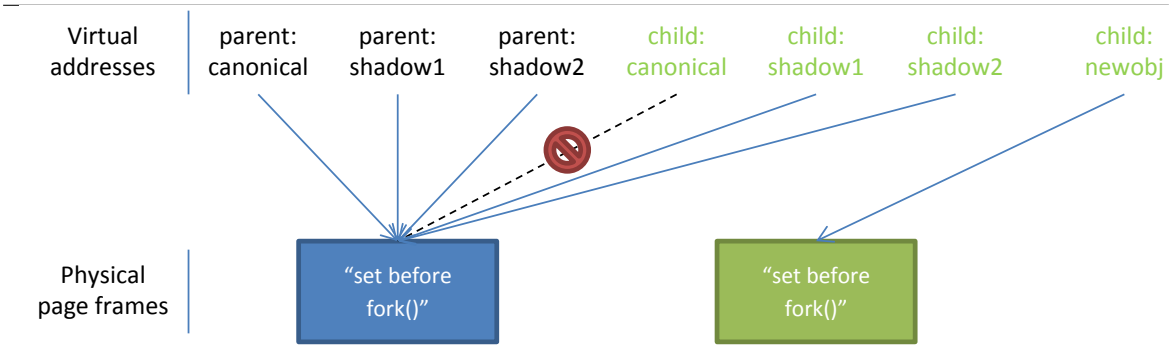
Step 2: after the vanilla `fork`. The child's canonical and shadow addresses are aliased to the parent's physical page frames.



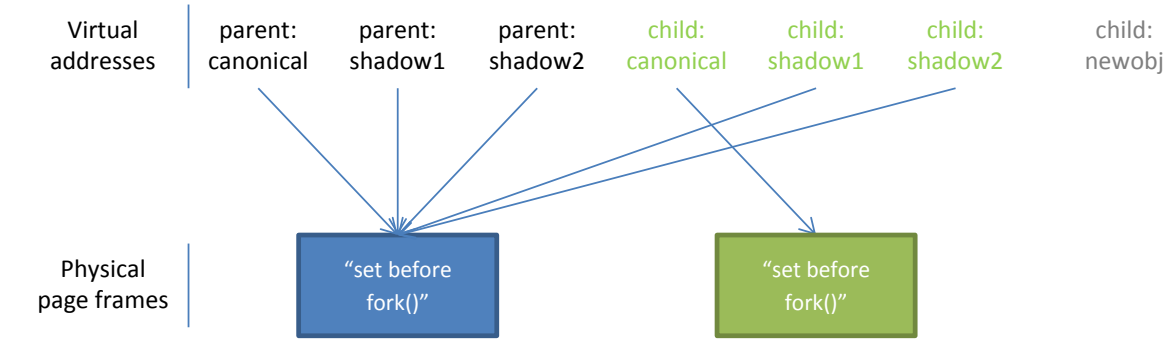
Step 3: the child allocates a new page, `newobj`.



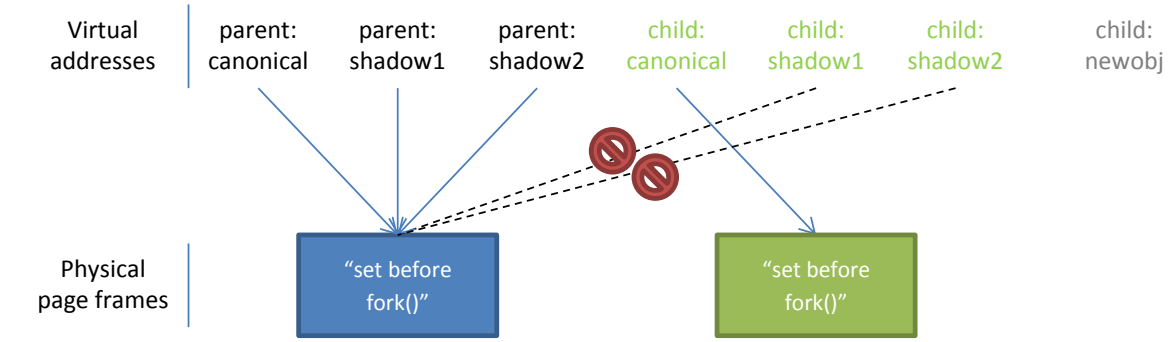
Step 4: the child uses the `canonical` (or `shadow1`/`shadow2`) pointer to copy the contents to `newobj`. After this step, the parent can resume execution.



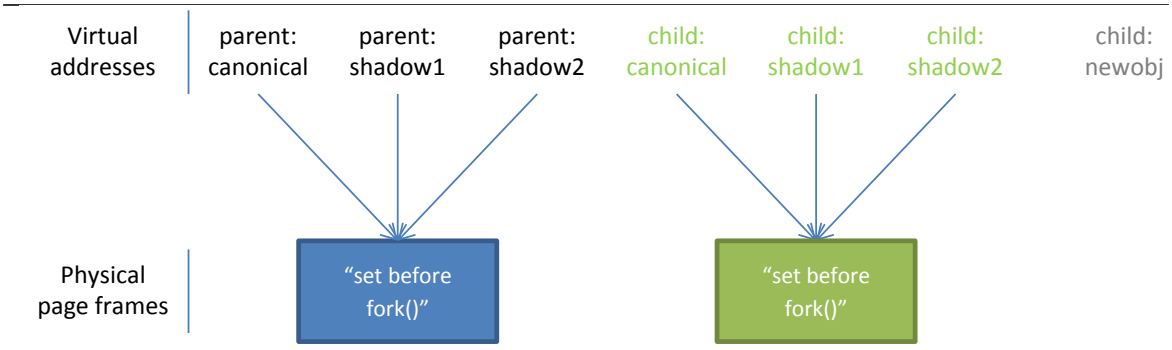
Step 5: the child unmaps `canonical`.



Step 6: the child remaps `newobj` to a "new" address, `canonical`.



Step 7: the child unmaps `shadow1` and `shadow2`.



Step 8: the child maps `shadow1` and `shadow2` as aliases of `canonical`. After this step, the child can resume execution.

We can further reduce the number of system calls by observing that the temporary pages `t` can be placed at virtual addresses of our choice. In particular, we can place all the temporary pages in one contiguous block, which lets us allocate them all using just one `mmap` command.

The parent process must sleep until the child has copied the canonical pages (and physical page frames). Fortunately, the parent does not need to wait while the child patches up the child's shadows, because although the child's shadows are still transiently pointing to the parent's physical page frames, the child application code will not access those pointers until after our wrapper has patched them. Oscar blocks signals for the duration of the `fork()` wrapper.

This algorithm suffices for programs that have only one thread running when the program forks. This covers most reasonable use cases; it is considered poor practice to have multiple threads running at the time of `fork` [14]. For example, `apache`'s event multi-processing module forks multiple children, which each then create multiple threads. To cover the remaining, less common case of programs that arbitrarily mix threads and `fork`, Oscar could "stop the world" as in garbage collection, or LeakSanitizer (a memory leak detector) [47].

Our algorithm could readily be modified to be "copy-on-write" for efficiency. Additionally, batching the remappings of each page might improve performance; since the intended mappings are known in advance, we could avoid the misprediction issue that plagued regular batch mapping. With kernel support we could solve this problem more efficiently, but our focus is on solutions that can be deployed on existing platforms.

Results.

We implemented the basic algorithm in Oscar. In cursory testing, `apache`, `nginx`, and `openssh` run with Oscar's `fork` fix, but fail without. These applications allocate only a small number of objects pre-`fork`, so Oscar's `fork` wrapper does not add much overhead (tens or hundreds of milliseconds).

3.7.2 Custom Memory Allocators

The overheads reported for SPEC CPU are based on instrumenting the standard `malloc`/`free` only, providing a level of protection similar to prior work. However, a few of the SPEC benchmarks [23] implement their own custom memory allocator (CMAs), which we consider to be any memory allocator other than the `glibc` `malloc`/`free`.

Since standard schemes for temporal memory safety require instrumenting memory allocation and deallocation functions, without special provisions none of them — including Oscar — will protect objects allocated via arbitrary CMAs.

There are two main reasons why CMAs are not automatically protected by Oscar. First, Oscar does not know about the CMAs. For example, if the application uses a duplicate copy of `glibc`'s allocator, but named `malloc2/free2`, this would be unknown to Oscar, and therefore not instrumented with a "lock-and-key". This can be addressed by manually identifying the CMAs (CMA identification could also be done automatically [24]) and wrapping them with Oscar as well. Second, CMAs typically obtain a large block of memory (from `sbrk`, `mmap`, or even `malloc`), and then partition (and even reuse) the block internally into multiple allocations; from Oscar's perspective, this is a single allocation, which means there is only one "lock" assigned during the lifetime of the large block, which are then shared by multiple allocations. CMAs that simply invoke and wrap a `malloc` call (for example, `xmalloc` calls `malloc` and aborts if the result is `NULL`) avoid this issue, and are automatically protected by Oscar.

We found that CMAs seem to be even more common in server programs, such as `apache`[19], `nginx`[20], and `proftpd`[21]. Prior work typically ignores the issue of CMAs.

If we do not wrap a CMA with Oscar, any objects allocated with the CMA would obviously not be resistant to use-after-free. However, there are no other ill effects; it would not result in any false positives for any objects, nor would it result in false negatives for the non-CMA objects.

3.7.3 Case Study: malloc-like custom memory allocator in memcached

`memcached`[22] is a memory object caching system that exports a get/set interface to a key-value store. We compiled `memcached 1.4.25` (and its prerequisite, `libevent`) and benchmarked performance using `memaslap`[23].

When we wrapped only `glibc`'s `malloc`, the overhead was negligible: throughput was reduced by 0–3%, depending on the percentage of set operations (Figure 3.21). However, this is misleadingly low, as it fails to provide temporal memory safety for objects allocated by the CMA. Therefore, we applied Oscar to wrap the CMA, in the same way we wrapped glibc's `malloc/free`.

Method

To support wrapping the CMA, we had to ensure that Oscar's `malloc` always used `MAP_SHARED` even for large objects. This covers the case where a CMA obtains a large block of memory from Oscar's `malloc` and partitions that block into multiple objects, each of which Oscar must protect by creating individual shadows.

We partitioned the address space to use separate high water marksfor the `malloc` wrapper and CMA wrapper.

[19] `https://httpd.apache.org/`
[20] `http://nginx.org/`
[21] `http://nginx.org/`
[22] `http://memcached.org/`
[23] `http://docs.libmemcached.org/bin/memaslap.html`

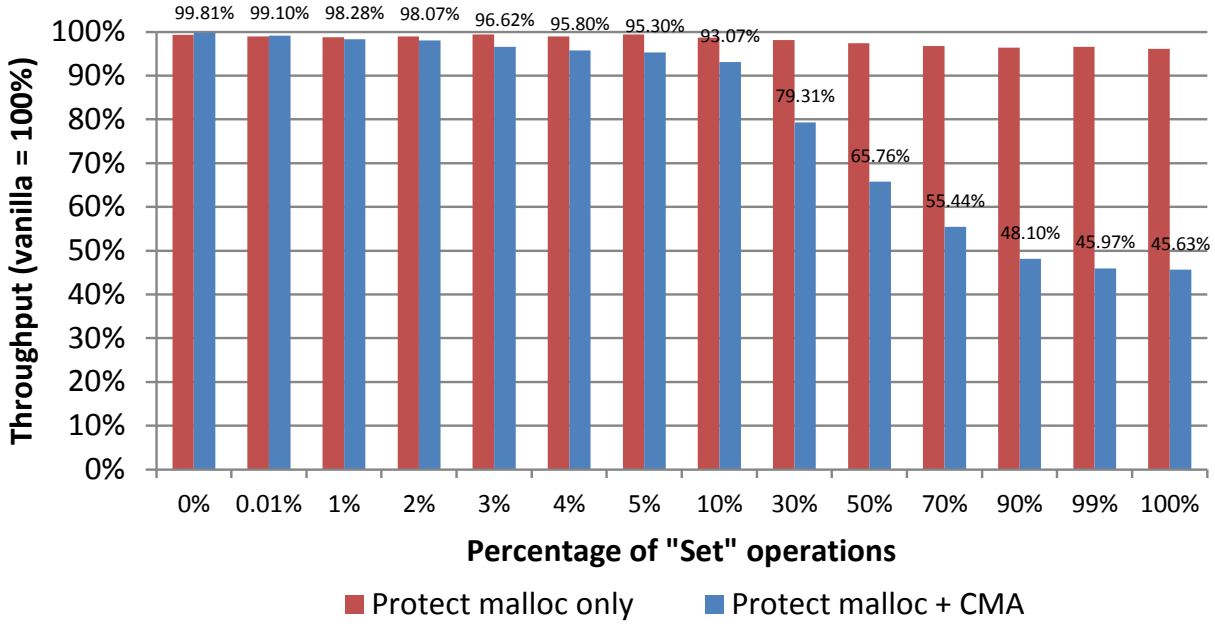


Figure 3.21: Throughput of Oscar on memcached.

We identified that allocations and deallocations via memcached's slab allocator are all made through the `do_item_alloc` and `item_free` functions. Thus, it is sufficient to add shadow creation/deletion to those functions.

For ease of engineering, we made minor changes directly to the slab allocator [15], similar to those we applied to `glibc`'s `malloc`: inserting a canonical address field in the memcached item struct, and modifying the allocation/deallocation functions. In principle, we only need to override the CMA allocate/deallocate symbols, without needing to recompile the main application.

With Oscar, the per-object metadata (e.g., the canonical address) is stored inline. This works for CMAs with `malloc`-like interfaces, where, given an allocation request of size `s`, Oscar can ask the CMA for an allocation of size `s + sizeof(inline_metadata)`. Some CMAs may, however, make this less practical (for example, a CMA that only allocates objects of a fixed size). If Oscar switched to a disjoint metadata store (e.g., a hashtable), similar to CETS [75], it would be easy to extend Oscar to protect any custom memory allocators that are identified: as with `glibc`'s `malloc`, the allocator function simply needs to be wrapped to return a new shadow, and the deallocator function wrapped to destroy the shadow. This would be a better long-term approach than individually dealing with each CMA that is encountered.

Results

When set operations are 3% of the total operations (a typical workload [5]), the performance overhead is roughly 4%. The overhead is higher for set operations because these require allocations (via the CMA), which involves creating shadows. Get operations have

almost no overhead because they do not perform memory allocation or deallocation and consequently do not require any system calls.[24] Unlike SPEC CPU, which is single-threaded, we ran `memcached` with 12 threads. This shows that Oscar's overhead is low even for multi-threaded applications, despite our naïve use of a mutex to synchronize part of Oscar's internal state (namely, the high water mark; see Section 3.9).

3.7.4 Special case: Region-based allocators

We have found several server programs that use region-based custom memory allocators [9]. Region-based allocators are particularly favorable for page-permissions-based schemes such as Oscar.

Typically, region-based allocators obtain a large block of memory from `malloc`, which they carve off into objects for their allocations. The distinguishing feature is that individual objects within a region cannot be freed; only the entire region can be freed en masse.

Region-based allocators by themselves are not resistant to use after free, since the blocks from `malloc` may be reused, but they provide temporal memory safety when the underlying `malloc/free` is protected by a lock-and-key scheme. Thus, there is no need to explicitly identify region-based CMAs; merely wrapping glibc's `malloc/free` with Oscar suffices to provide temporal memory safety for such programs, i.e., Oscar would provide full use-after-free protection for a region-based allocator, without the need for any custom modifications.

Oscar's performance is especially good for programs that use region-based allocators: since there are few `malloc()`s or `free()`s to instrument, and correspondingly low memory or TLB pressure, Oscar imposes negligible overhead. Other classes of lock-and-key schemes also provide full protection to programs with region-based allocators, but they often have high overhead, since they must instrument all pointer arithmetic operations (and possibly pointer dereferences).

3.8 Discussion

Our results show that shadow-page-based schemes with our optimizations have low overhead on many benchmarks. From Table 3.2, we argue that changing the lock is theoretically easier than revoking all the keys, and implicit lock-and-key is better than explicit. Our experimental results confirm that prediction: Oscar's runtime overhead is lower than CETS, DangNull, and FreeSentry overall and on most benchmarks, and comparable to DangSan (but with lower memory overhead for Oscar), even though they all need source code while Oscar does not.

[24] Technicality: `memcached` lazily expires entries, checking the timestamp only during the get operation. Thus, the overhead of destroying shadows may be attributed to get operations. This means when there are not "Get" operations, we might not be measuring the overhead of destroying shadows.

3.8.1 Virtual Address Space Considered Hard to Fill

A concern might be that Oscar would exhaust the $2^{47}B$ =128TB =32 billion 4K page user-space virtual address space, necessitating reuse of addresses belonging to freed pages. This is unlikely in common scenarios. Based on extrapolating the CPU2006 benchmarks, it would take several *days* of continuous execution even for allocation-intensive programs. For example, with `perlbench`, which allocates 361 million objects (>99% of objects fit in one page) over 25 minutes, it would take 1.6 days (albeit less on newer, faster hardware) to exhaust the address space. `dealII`, `omnetpp` and `xalancbmk` would take over 2.5 days each, `gcc` would take 5 days, and all other CPU2006 benchmarks would take at least 2 *weeks*. An analysis of `memcached` servers at Facebook shows between 5-60 billion requests (predominantly for very small objects) per week [5], corresponding to between roughly 3.5 and 40 days to exhaust the virtual address space. We expect that most programs would have significantly shorter lifetimes, and therefore would never exhaust the virtual address space. It is more likely that they would first encounter problems with the unreclaimed page-table memory (see Section 3.9). Nonetheless, it is possible to ensure safe reuse of virtual address space, by applying a conservative garbage collector to old shadow addresses (note that this does not affect physical memory, which is already reused with new shadow addresses); this was proposed (but not implemented) by Dhurjati and Adve [37].

Recently, Intel has proposed 5-level paging, enabling a 57-bit virtual address space [27]; implementation of Linux support is already underway [92]. This 512-fold increase would make virtual address space exhaustion take *years* for every CPU2006 benchmark.

3.8.2 Hardware Extensions

Due to the high overhead of software-based temporal memory safety for C, some have proposed hardware extensions (e.g., Watchdog [74]). Oscar is fast because it *already* utilizes hardware – hardware that is present in many generations of x86 CPUs: the memory management unit, which checks page table entries. We believe that, with incremental improvements, shadow-page-based schemes will be fast enough for widespread use, without the need for special hardware extensions. For example, Intel's Broadwell CPUs have a larger TLB and also a second TLB page miss handler [2], which are designed to improve performance for general workloads, but would be particularly useful in relieving Oscar's TLB pressure. Intel has also proposed finer-grained memory write protection [89]; if future CPUs support write *and read* protection on subpage regions, Oscar could be adapted to one-object-per-*subpage*, which would reduce the number of shadows (and thereby TLB pressure).

3.8.3 Compatibility

Barring virtual address-space exhaustion (discussed in Section 3.8.1), Oscar will crash a program if and only if the program *dereferences* a pointer after its object has been freed. It does not interfere with other uses of pointers. Unlike other lock-and-key schemes, page-

permissions-based schemes do not need to instrument pointer arithmetic or dereferencing (Table 3.2).

Accordingly, Oscar correctly handles many corner cases that other schemes cannot handle. For example, DangNull/FreeSentry do not work correctly with encrypted pointers (e.g., PointGuard [30]) or with typecasting from non-pointer types. CETS has false positives when casting from a non-pointer to pointer, as it will initialize the key and lock address to invalid values.

Additionally, DangNull does not allow pointer arithmetic on freed pointers. For example, suppose we allocate a string `p` on the heap, search for a character, then free the string:

```
char* p = strdup("Oscar"); // Memory from malloc
char* q = strchr(p, 'a');  // Find the first 'a'
free(p);
```

Computing the index of "a" (`q - p == 3`) fails with DangNull, since `p` and `q` were nullified. It does work with DangSan and FreeSentry (since they only change the top bits) and with Oscar.

DangSan, DangNull and FreeSentry only track the location of pointers when they are stored in memory, but not registers. This can lead to false negatives: DangSan notes that this may happen with pointers spilled from registers onto the stack during function prologues, as well as race conditions where a pointer may be stored into a register by one thread while another thread frees that object. DangSan considers both issues to be infeasible to solve (for performance reasons, and also the possibility of false positives when inspecting the stack). Oscar does not need to track pointers, and therefore does not have this vulnerability.

Benign use-after-free

There are two classes of use-after-free that are arguably benign, which we catch — though so do most other schemes. If an object has been freed and reallocated to a new pointer, but the new pointer has not yet accessed the memory:

```
char* old = malloc(...);
free (old);
char* new = malloc(...); // Assume 'new' aliases 'old'
```

then it is safe to read/write with the old pointer. Alternatively, if we have read, but not written, with the new pointer, then it is safe to read, but not write, using the old pointer.

We could change our implementation to catch only verifiably unsafe cases, as follows. When the old object is freed, we do not change its shadow page permissions; this means we still allow read/write using the old pointer, for the moment. When the new object is allocated, we set the new shadow page permissions to no-read/no-write. If the new pointer is used to read, it segfaults; we catch this with a custom signal handler that grants read permissions to the new object's shadow page, and removes write permissions from the old object's shadow page. Similarly, if the new pointer is used to write, it will segfault; our signal handler can add read+write permissions to the new object's shadow page, and remove

read+write permissions from the old object's shadow page. We did not implement this, as it requires roughly two to three times more syscalls than the ordinary approach — greatly increasing the overhead — with a questionable benefit.

Nonetheless, should avoiding these somewhat benign cases be considered desirable, we note that it is cheaper to add the above workaround to our scheme, than it is to patch alternative lock-and-key schemes (e.g., CETS, DangNull, FreeSentry). Our simple workaround is possible only with page-permissions-based schemes, since it relies on each object having its own virtual address.

3.9 Limitations and Future Work

Oscar is only a proof-of-concept for measuring the overhead on benchmarks, and is not ready for production, primarily due to the following two limitations.

Reclaiming page-table memory takes some engineering, such as using `pte_free()`. Alternatively, the Linux source mentions they "Should really implement gc for free page table pages. This could be done with a reference count in struct page."[25] Not all page tables can be reclaimed, as some page tables may contain entries for a few long-lived objects, but the fact that most objects are short-lived (the "generational hypothesis" behind garbage collection) suggests that reclamation may be possible for many page tables. Note that the memory overhead comparison in Section 3.6.2 already counts the size of paging structures against Oscar, yet Oscar still has lower overall overhead despite not cleaning up the paging structures at all.

We did not encounter any issues with users' `mmap` requests overlapping Oscar's region of shadow addresses (or vice-versa) — which would overwrite the existing mapping — but it would be safer to deterministically enforce this by intercepting the users' `mmap` calls.

Currently, all threads share the same high water markfor placing new shadows, and this high water markis protected with a global mutex. A better approach would be to dynamically partition the address space between threads/arenas; for example, when a new allocator arena is created, it could split half the address space from the arena that has the current largest share of the address space. Each arena could therefore have its own high water mark, and allocations could be made independently of other arenas. This could lower the overhead of the `memcached` benchmarks, but not the SPEC CPU benchmarks (which are all single-threaded).

We can vary the threshold of when an object is considered large enough to qualify for its own sole-occupancy page(s) (and therefore `MAP_PRIVATE`). If the threshold is too high, then few objects will benefit; but if the threshold is too low, many small objects will each occupy an entire physical page (with a zero-byte threshold, this devolves to the one object per physical page frame approach). One of the reasons this threshold is set fairly high by `glibc` is because that `malloc` and `free` would each require an additional syscall (`mmap`/`munmap`)

[25] http://lxr.free-electrons.com/source/arch/x86/include/asm/pgalloc.h

for every object. However, each "small" object (sharing a page) would require these syscalls anyway (for the shadows), which partly balances out the cost.

If we are willing to modify `internal_malloc`, Oscar can be selective in how to refresh (or batch-create) shadows. For example, objects that are small enough (among other conditions) to fit in `internal_malloc`'s "small bins" are reused in a first-in-first-out order, which means that a speculatively created shadow is likely to be used eventually. Other bins are last-in-first-out or even best-fit, which makes their future use less predictable. This optimization may particularly benefit `xalancbmk` and `dealII`, for which the ordinary refresh shadow approach was a net loss.

We could take advantage of the short-lived nature of most objects to experiment with placing multiple objects per shadow; fewer shadows means lower runtime and memory overhead. To further reduce memory overhead, we could change `internal_malloc` to place the canonical address field at the start of each page, rather than the start of each object. All objects on the page would then share the canonical address field, which could drastically reduce the memory overhead for programs with many small allocations (e.g., `soplex`). We explore this direction in Chapter 4.

3.10 Conclusion

Efficient, backwards compatible, temporal memory safety for C programs is a challenging, unsolved problem. By viewing many of the existing schemes as lock-and-key, we showed that page-permissions-based protection schemes were the most elegant and theoretically promising. We built upon Dhurjati and Adve's core idea of one shadow per object. That idea is unworkable by itself due to accumulation of `vm_area_struct`s for freed objects and incompatibility with programs that `fork()`. Dhurjati and Adve's combination of static analysis partially solves the first issue but not the second, and comes with the cost of requiring source-code analysis. Our system Oscar addresses both issues and introduces new optimizations, all without needing source code, providing low overheads for many benchmarks and simpler deployment. Oscar thereby brings page-permissions-based protection schemes to the forefront of practical solutions for temporal memory safety.

Chapter 4

Oscar++: Extending Oscar with Multiple Objects per Alias

4.1 Introduction

Oscar advanced the state of the art for heap temporal memory safety, but its overhead was very high for some allocation-intensive programs (Figure 3.16). Oscar's optimizations reduced the overhead of `MAP_SHARED` (by using `MAP_PRIVATE` memory for large mappings) and updating aliases (by using the high-water mark, and refreshing aliases), but did not reduce TLB pressure.

To a first-order approximation, TLB pressure occurs because there are N objects, with one object per alias resulting in N aliases, exceeding the TLB capacity of k objects. Increasing the TLB size requires hardware changes, while decreasing the number of objects requires application software changes; both are out of scope. This means the only lever to reduce the TLB pressure is to place multiple objects per alias. Multiple objects per alias also reduces the overhead of syscalls to maintain the mappings.

Page-permissions based schemes have historically used one object per page, so that the resources associated with an object (physical memory on the canonical page, and the alias mapping for alias-based schemes) can be reclaimed immediately when the object is freed by the user. With multiple objects per alias, when the user frees an object, Oscar++ considers it "quarantined" and does not call `internal_free()`, nor does it unmap the alias. When all the objects on the alias are in the quarantined state, Oscar++ unmaps the alias and `internal_free()` all the objects; this is similar to the abandoned approach of batching frees in Section 3.5.4.

When using quarantine (more directly, when delaying the `munmap` of an alias), some attempted attacks (e.g., writing to an object after it has been freed, but before the memory has been reallocated to another object) may not be detected. However, this does not compromise the security of the scheme, since the memory has not been reused for a fresh allocation: as discussed in Section 3.8.3, it is *reuse* after free that is dangerous. We note that the infinite

heap (i.e., never `free()` objects) or garbage collection — which are both considered secure — also do not detect these benign cases of use after free.

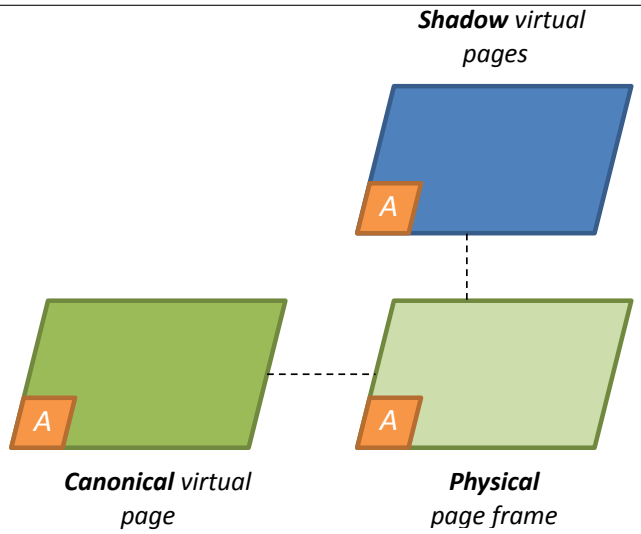
The main downside of quarantine is memory usage: in the worst case, it can increase memory usage by over a hundred-fold due to stateholding. For example, consider an alias that contains a small object (1-byte of user data plus 16 bytes of allocator metadata) and a large object (4,000+16 bytes), where the large object is freed/quarantined but the small object is not; on account of the 1-byte object, the 4,000 bytes of the large object cannot be truly freed and reused. In practice, memory usage is not a concern because the vast majority of objects are short-lived (the generational hypothesis in garbage collection), which means it is rare that short-lived and long-lived objects will be placed on the same alias. Additionally, placing multiple objects per alias reduces the number of aliases, and thereby the amount of memory consumed by page table entries and `vm_area_structs`. Finally, the number of objects per page is parameterizable, ranging from the low memory overhead of vanilla Oscar (but high runtime overhead for allocation-intensive programs) to a higher level of memory overhead with lower runtime overhead.

In the first half of this chapter, we implement and evaluate Oscar++, a basic multiple objects per alias scheme, and demonstrate that it substantially reduces the runtime overhead of allocation-intensive benchmarks. In the second half, we will consider using the lifetime of an object (the time from which it is allocated to "freed") to reduce the stateholding that arises when objects of different lifetimes are placed on the same alias. We will conclude with a discussion of directions for future work, such as using the "hotness" of an object (frequency of memory accesses) to optimize TLB and data cache utilization.

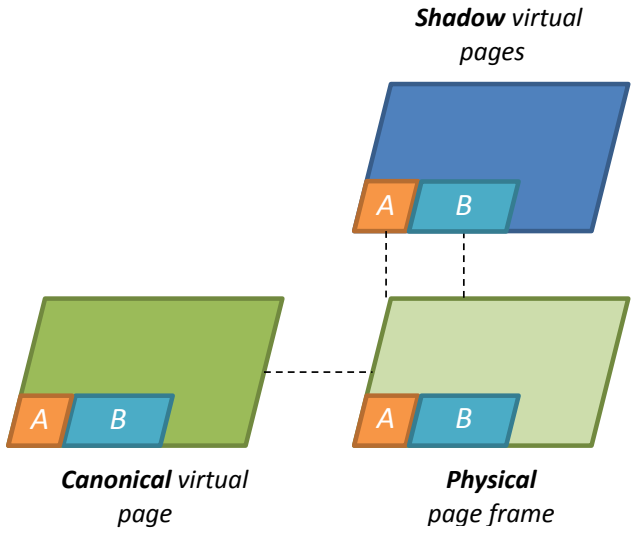
4.2 Implementation

Oscar++ maintains the standard glibc malloc interface, receiving allocation requests (possibly interspersed with `free()` requests) individually. This means Oscar++ must be able to incrementally allocate additional objects to an existing alias. Furthermore, Oscar++ must be able to keep track of which objects on the alias the user has previously requested to `free()`— which can happen in any order — and then, when the alias has reached its lifetime maximum number of additional objects, and all objects were `free()`'d by the user, unmap the alias and `internal_free()` all the objects. In Figure 4.1, we illustrate how a series of `malloc` and `free()` events are handled by Oscar++.

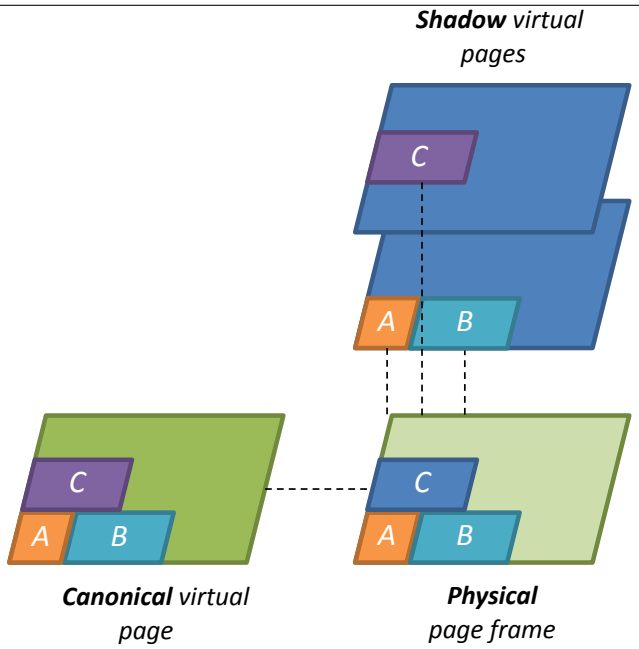
Table 4.1: Illustration of how `malloc` and `free()` events are handled by Oscar++.



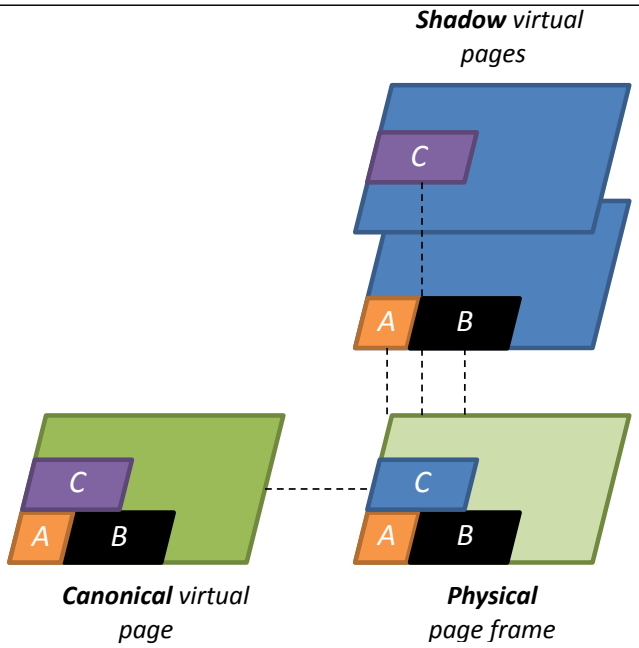
Step 1: Object A allocated. We assume 2 objects per alias.



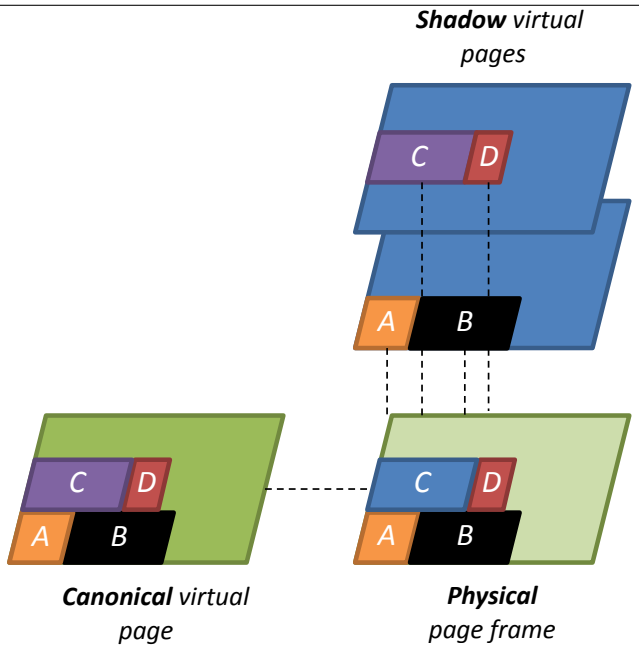
Step 2: Object B allocated; it is assigned to the same alias as object A.



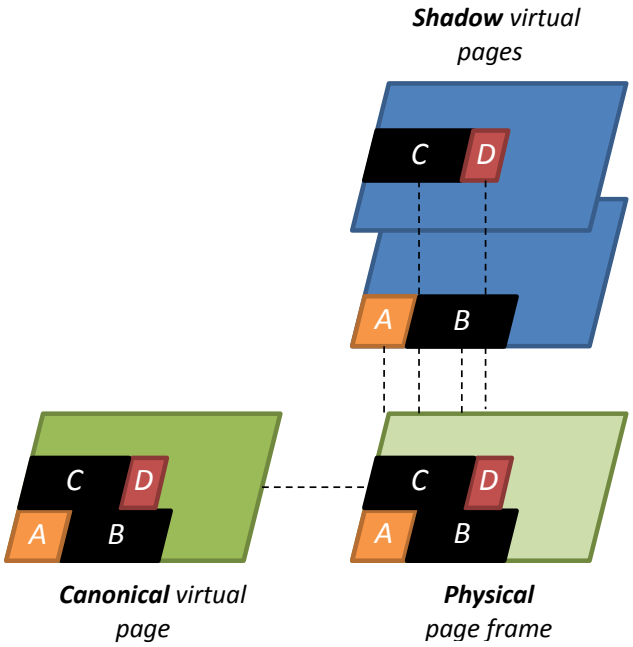
Step 3: Object C allocated; requires a new alias.



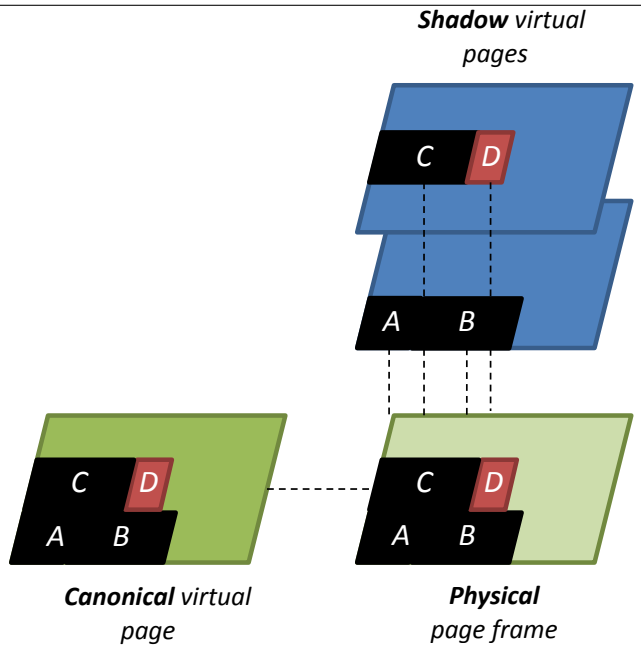
Step 4: Object B freed by the user; Oscar++ quarantines it (`glibc`'s allocator considers it still allocated).



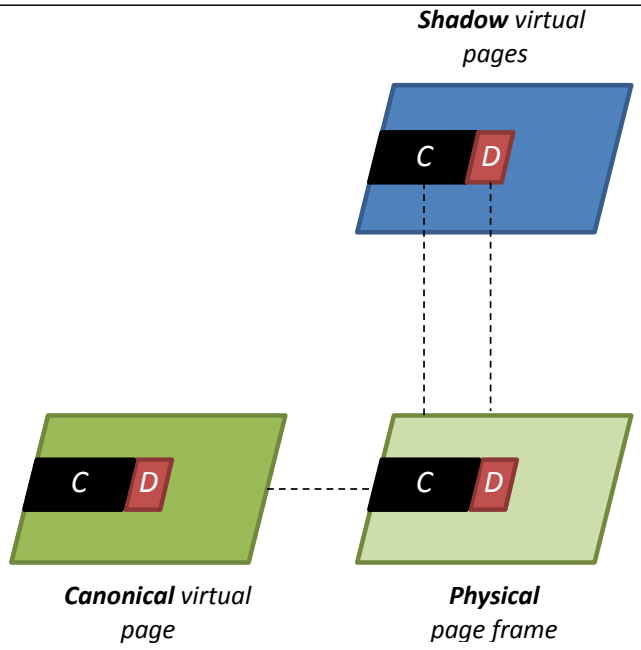
Step 5: Object D allocated; it is assigned to the same alias as object C.



Step 6: Object C freed by the user; Oscar++ quarantines it.



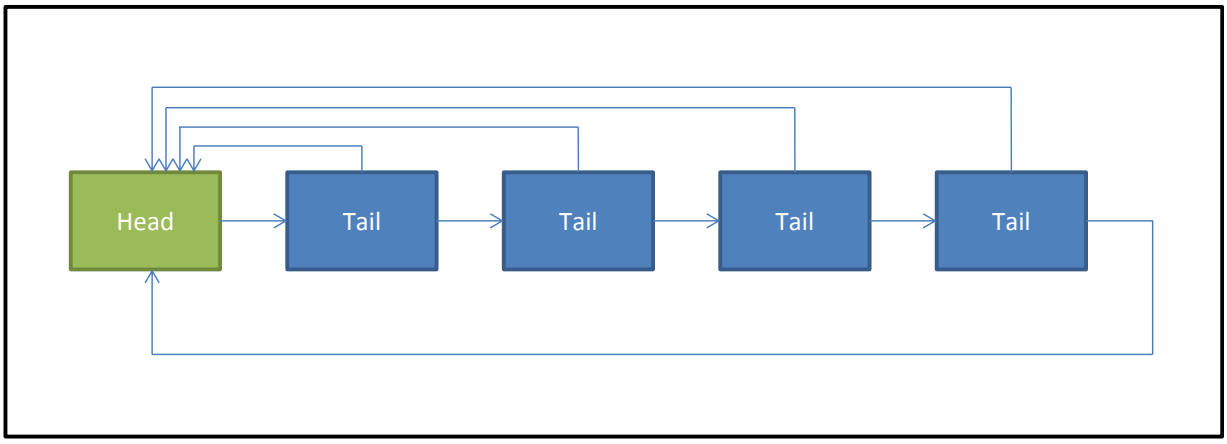
Step 7: Object A freed by the user.



Step 8: The alias containing objects A and B can be unmapped, and both canonicals are finally `free()`'d.

For simplicity, we only consider objects that fit fully within one page. This is not a major

Figure 4.1: Oscar++ linked list



restriction, as objects that span multiple pages are either larger than a page (and therefore do not benefit from multiple objects per page) or are uncommon edge cases (spanning the edge between two contiguous pages).

Oscar++ keeps track of all the objects on an alias by chaining them together in a singly-linked list (Figure 4.1). It also stores inline metadata for each object, to indicate whether it has been freed. The first object to be allocated on the alias, which becomes the head of the list, maintains a counter of the number of additional objects that can be allocated on the alias ("remaining allocation slots"), and a counter of the number of live objects (including itself) on the alias.

When we allocate an object on an alias that already has a head, we attach the new object to the linked list and decrement the number of remaining allocation slots. When we free any object on the alias, we must decrement the counter of the number of live objects; since this counter is stored on the head, all tail (non-head) objects store a pointer to the head, allowing O(1) access to the counter. For the `free()` function call, Oscar++ is only given the address of the object, and not whether it is a head or tail, so we use an additional bit to indicate whether an object is the head or not.

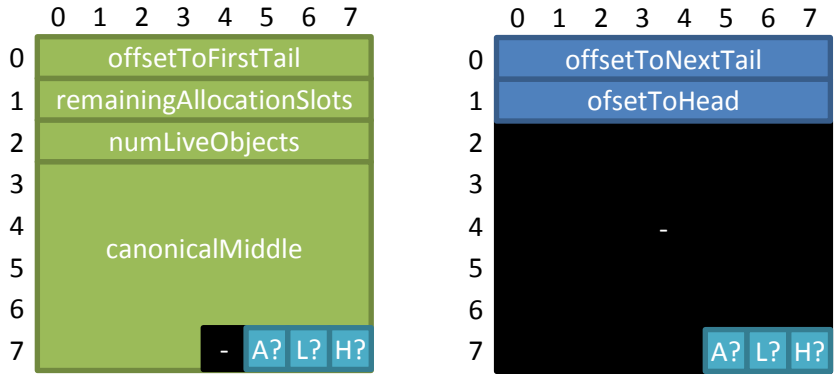
4.2.1 Packing the metadata

Implemented naively, the above approach entails a significant amount of additional inline metadata, especially for smaller objects. With some careful bitpacking, Oscar++ requires only 8-bytes of inline metadata per object, the same as Oscar.

In Oscar, the canonical address field stores a regular 64-bit pointer. Current implementations of x64 CPUs only support a 48-bit virtual address space, with the top 16 bits set to zero.[1] Additionally, we only need to store the start of the canonical page, which means the

[1]We can save one additional bit by observing that, on Linux, the top-half of the virtual address space is reserved for the kernel.

Figure 4.2: Oscar++ metadata for head (left) and tail (right) objects. A = truly aliased (see Section 4.2.3), L = live object, H = head.



bottom 12 bits are set to zero (assuming a 4KB page), freeing up a total of 28 bits per page. This is similar to DangSan's [60] "pointer compression".

We can apply similar pointer compression to the linked-list of objects within an alias. Since they are all within the same 4KB page, we only need to store the page offset (lower 12 bits); and with the assumption that objects are aligned to 16-byte addresses (which holds for the version of `glibc` that we modified), we can omit the bottom 4 bits. Thus, each linked-list node only requires 8 bits.[2]

With pointer compression, we can no longer `NULL`-terminate the linked list, since any 8-bit value is potentially the offset of a valid object on the page. Instead, we use the offset of the head to indicate the end of the linked list (Figure 4.1), i.e., it is actually a ring, though we refer to it as a linked list for simplicity.

We use 8-bit counters for the number of remaining allocation slots and the number of live objects. This exceeds the maximum number of objects that can be placed on a 4KB page, considering that the minimum object size (including `glibc` and Oscar's inline metadata) is 24 bytes.

Oscar++ uses the least-significant bit of its inline metadata to store if it is the head; this bit is in the same location for both head and tail objects. We also use single bits to indicate whether the object is live, and one bit to store whether the "alias" virtual page is truly aliased to the canonical virtual page, for implementation reasons described later. The inline metadata fields are shown in Figure 4.2.

Intel has recently proposed a 57-bit virtual address space, which would require 9 additional bits for the canonical address. We currently only have 1 unused bit. To obtain the remaining 8 bits, we could reduce the `remainingAllocationSlots` and `numLiveObjects` fields to 4 bits each. As our results will show, a small maximum number (< 10) of objects per alias generally provides optimal runtime.

[2]If we only assume 8-byte addresses, we require an additional bit, which we can afford. However, it is more performant to load and store an 8-bit value.

4.2.2 Caching partly filled aliases

As part of the "Refreshing shadows" optimization (Section 3.5.2), Oscar maintains a hashtable that maps from each canonical page range (1 or more pages) to an array of empty aliases. Oscar++ uses the same array to store partly-filled aliases (aliases that contain one or more objects), i.e., when malloc is called, Oscar++ checks the hashtable for a cached alias. If it is empty, then it initializes the head node. If it is a partly filled alias, it chains on a non-head node. If no cached alias is available, it creates one on demand and initializes the head node. In all three cases, it decrements the number of remaining allocation slots, and if that is still non-zero, it stores the alias in the cache.

We us a last-in-first-out policy for reusing cached aliases. This can lead to the array of cached aliases containing a mix of empty and partly filled aliases. For example, suppose there is a maximum of two objects per alias lifetime, and we allocate three objects (A, B, C). Objects A and B will be assigned to alias1, and object C will be assigned to alias2. If objects A and B are freed, alias1 will be refreshed into alias3, which will be added to the end of the cache; thus, the next object D will be assigned to alias3, not alias2. To distinguish between the empty and partly filled aliases (an important distinction because it results in a head or tail node respectively), we tag the least-significant bit of the alias address for the latter case. There is no information loss since addresses are 16-byte aligned, which means the bottom four bits are always zero. Alternatively, our hashtable could map to an array of empty aliases, and a partly filled alias (possibly `NULL`). This scheme would ensure that there is at most one partly filled alias per canonical page.

4.2.3 The isAliased bit

When an alias has exhausted its lifetime maximum number of objects, and all objects were `free()`'d by the user, Oscar++ conceptually unmaps the alias and then traverses the linked list using the canonical page to find and `free()` the objects. This does not work for the large objects that are backed by `MAP_PRIVATE` memory: since the canonical and "aliased" pages are mapped to different sets of physical page frames, Oscar++'s metadata is only stored on the "aliased" page, which is unavailable after we have unmapped the alias. This is similar to the issue encountered with `realloc` (Section 3.5.3). We cannot defer unmapping the alias until after `free()`'ing the objects, since that returns the canonical memory to the heap and therefore permits use-after-free in multi-threaded applications.

Our workaround is to walk the linked list and make a local copy of Oscar++'s metadata, unmap the alias, and then use that copy of the metadata to free the objects. This workaround is correct even for properly aliased (`MAP_SHARED`) memory[3], but is unnecessarily slow, so we only use it for `MAP_PRIVATE` objects. The "isAliased" bit is set when the object is initially allocated, so that in the common case (`MAP_SHARED`), we can use the faster, naïve algorithm;

[3]When implemented as described. `MAP_PRIVATE` implies it is a large, multi-page object, and therefore does not share the alias with any other objects, so it is possible to simplify this algorithm for the `MAP_PRIVATE` case, i.e., to only handle a single object.

it could also potentially be used to provide a fastpath for the realloc function. We determine whether the pages are aliased by first principles (compare the first byte, set the first byte, compare again) rather than using the specifics ("`chunk_is_mmapped`") of the glibc allocator. Some care is required to prevent the compiler from optimizing out the alias check.

4.3 Results

Figure 4.3 shows the runtime of the six SPEC CPU2006 benchmarks that had >10% overhead with Oscar, for different settings of the maximum number of additional objects on an alias over its lifetime ("n"). The y-axis is the runtime, normalized to vanilla `glibc malloc`. The Oscar results are a rerun, so they differ slightly from the results in Chapter 3. They also differ slightly from the n=0 case of Oscar++, due to minor implementation differences (e.g., different inline metadata management) and experimental error. Although "n" is a discrete parameter, we have added dashed lines between data points to make it easier to view the runtime trend when increasing "n". The graph shows that the overhead drops significantly even with just n=1, and decreases further for larger values of n, albeit with diminishing returns. The diminishing returns is to be expected, since the remaining overhead of syscalls for creating/disabling shadows is roughly $\frac{1}{n+1}$ of the overhead of Oscar (or Oscar++ with n=0). For example, with n=2, there are up to 3 objects per alias, so the overhead may be as low as 1/3.

The best value of n differs for each benchmark, but n=5 provides close to optimal overhead. The runtime overhead increases for very large values of n; this is to be expected as many aliases will contain at least one long-lived object, preventing reuse of quarantined objects and thereby disrupting locality of reference.

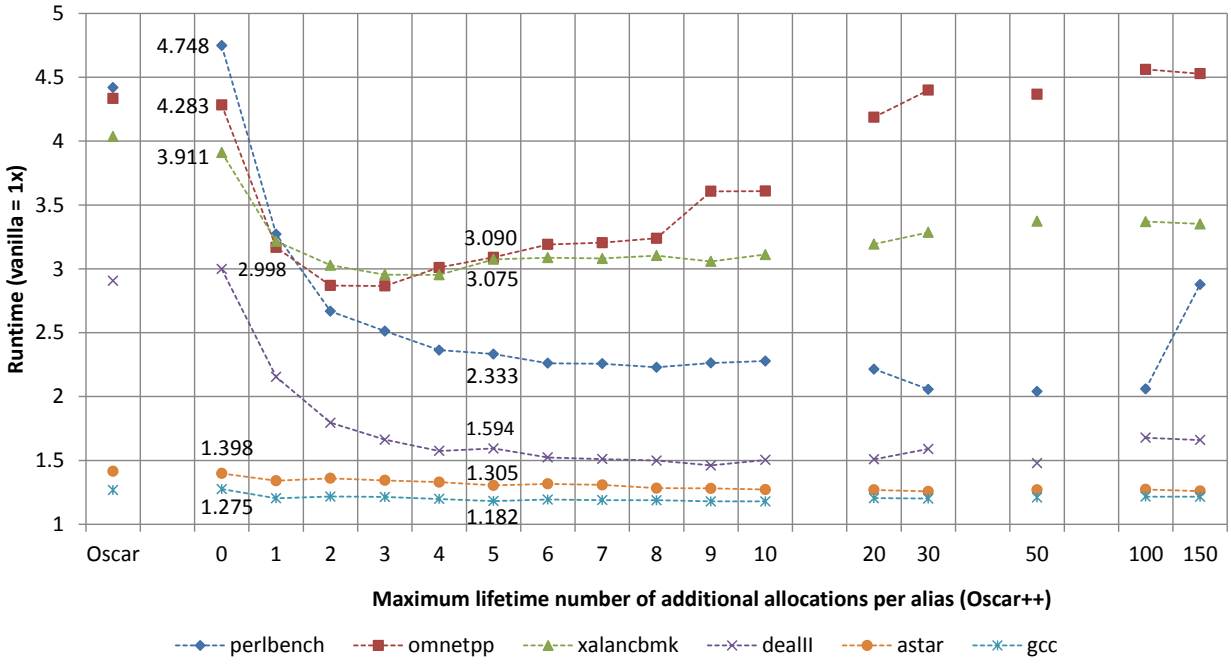
The remaining non-Fortran SPEC CPU2006 benchmarks, which all have <10% overhead, are shown in Figure 4.4. `sphinx3`, `milc` or `povray` (the most expensive of this group) may have some non-trivial benefits from Oscar++. The other benchmarks have very low Oscar overhead and do not benefit from Oscar++.

Figure 4.5 shows the geometric mean overhead of the six higher-overhead benchmarks from Figure 4.3, the remaining benchmarks from Figure 4.4, and combined. These graphs show what overhead reduction is achievable if we set a single n value for all benchmarks, rather than tailoring n to each benchmark. Overall, this graph shows the same trend as Figure 4.3, with n=5 being approximately optimal, with runtime overhead of 24.7% for Oscar++ compared to 39.4% for Oscar.

4.4 Oscar++LP: Oscar++ with Lifetime Prediction

Lifetime prediction has been proposed for general-purpose allocators [7], with simplified, faster handling of short-lived objects. We adapt this idea to reduce stateholding in Oscar++ that we had identified in Section 4.1.

Figure 4.3: Oscar++ runtimes for the six higher-overhead non-Fortran SPEC CPU2006 benchmarks. n = maximum lifetime number of additional allocations per alias, for Oscar++.



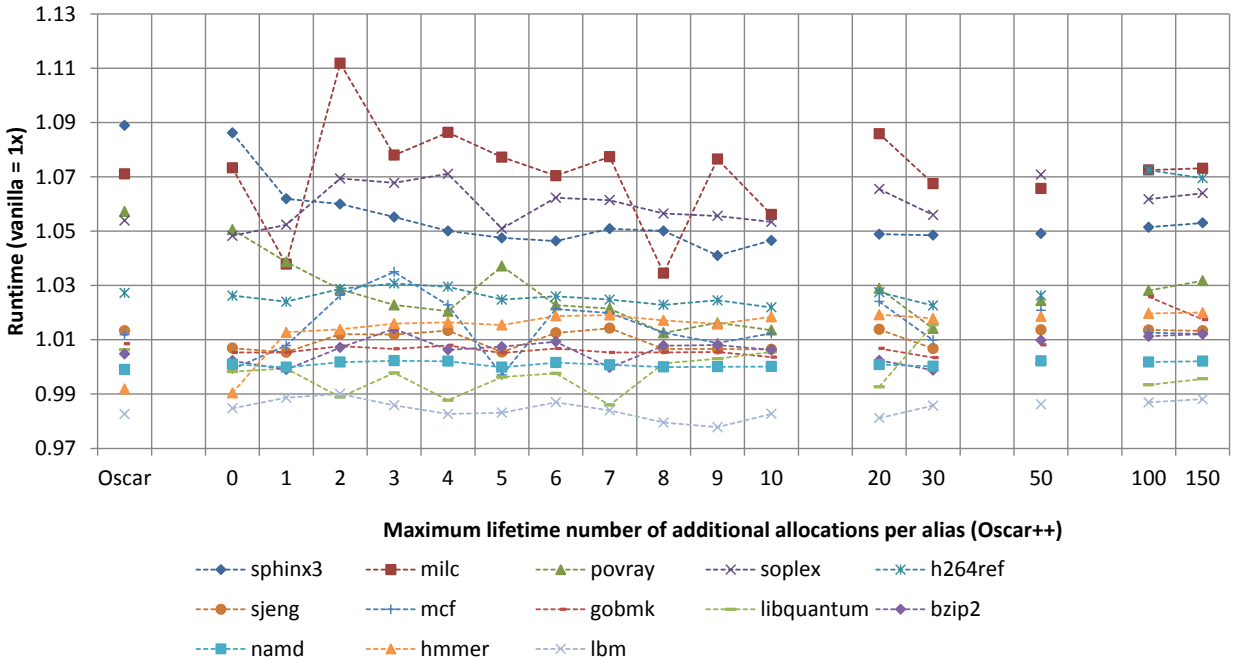
Stateholding of memory occurs only because objects of different lifetimes are co-located on the same alias. If an alias hosted only short-lived objects, or only long-lived objects, the duration of stateholding would be greatly reduced, since all objects would be freed at roughly the same time.[4] In addition to preventing stateholding of memory on canonical pages, this also reduces the number of live memory mappings (and therefore memory used for `vm_area_structs`) and TLB pressure. For example, suppose objects S1 and S2 are short-lived, while objects L1 and L2 are long-lived. If we placed objects S1,L1 onto alias1 and S2,L2 onto another alias2, then both aliases would be needed. The placement of S1,S2 and L1,L2 onto alias1 and alias2 respectively would allow alias1 to be retired earlier.

We cannot readily consolidate or migrate long-lived objects onto a different page, because we would need to know where all the pointers to the object are, and update them accordingly. This would necessitate a pointer tracking mechanism such as the compiler-based instrumentation of DangNull/FreeSentry/DangSan [62, 105, 60] (which we have seen incurs significant overhead) – or the Pintool-based instrumentation of RuntimeASLR [64] (which is extremely slow when tracking pointers).

In this section, we consider the benefit of segregating objects/aliases by lifetime. We assume we have access to an oracle that returns the lifetime of an allocation; this provides a

[4]The long-lived objects would, by definition, not be freed for a long time, regardless of whether we are using Oscar++ or no aliases at all. Thus, Oscar++ does not induce much stateholding.

Figure 4.4: Oscar++ runtimes for the non-Fortran SPEC CPU2006 benchmarks, other than the six higher-overhead benchmarks of Figure 4.3. n = maximum lifetime number of additional allocations per alias, for Oscar++. The y-axis is different from Figure 4.3.



loose upper-bound on the benefit of using lifetime, since practical systems can only approximately predict lifetime.

4.4.1 Implementation

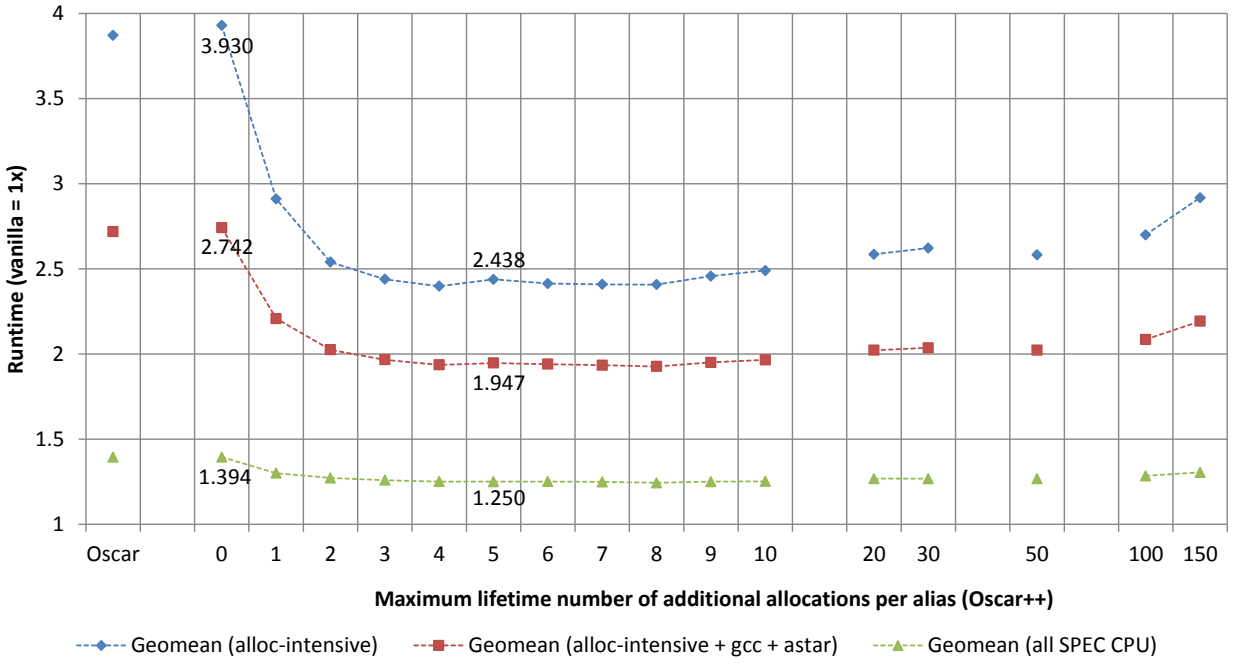
In this section, we describe how we obtain the allocation event log needed for the oracle, how we use the log to implement the oracle, and how we store the cached aliases.

Obtaining the log for the oracle

In Section 3.4.1 of Oscar, we already obtained, using `mtrace`, a log of all allocation and free events. We encountered some minor inconsistencies between mtrace's log and the order in which Oscar++ received `malloc/realloc/free()` calls ("allocator calls"); for example, `mtrace` lists `realloc` as a `free` followed by a `malloc`, whereas Oscar++ implements it as allocate new object — copy — free old object. To improve synchronization between the log and the program run, we essentially reimplemented `mtrace`: we open a file for writing, `mmap` it into memory, and then write to the log during each `malloc/free()` operation. We avoid using any logging functions that would result in any Oscar++ memory allocations.

Using the log

Figure 4.5: Oscar++ runtimes: geometric means. n = maximum lifetime number of additional allocations per alias, for Oscar++. The y-axis is different from previous figures.



Given the log of all alloc/free events and ample memory, it is trivial to calculate the object lifetimes by storing all allocations in a hashtable, and printing the allocation and lifetime when we encounter the matching free.

We modified `malloc`/`realloc` to call our oracle function, which reads the precomputed lifetime from the log (`mmap`'ed into memory), to determine which alias to place the object on. We add assertions to check that the parameters of allocator calls match the log. This is important because the allocation calls are highly sensitive to minor changes in the environment (for example, `perlbench` calls `getenv`, so even an apparently inconsequential choice, such as the previous working directory[5] will affect the parameters), and seemingly non-deterministic as well: we force recompilation and rerun until the assertions pass.

Storing the cached aliases

For any given canonical page, there are three types of cached aliases: aliases containing 1 or more short-lived objects, aliases containing 1 or more long-lived objects, and empty aliases. Accordingly, we convert the array of cached aliases (which contained a mix of empty and partly filled aliases) into a struct containing a partly filled alias (`NULL` if not present) for short-lived objects, a partly filled alias for long-lived objects (`NULL` if not present), and an array of cached empty aliases:

```
void** shadows_empty;
```

[5] `OLDPWD`

```
void* shadow_partlyFilled_shortLived;
void* shadow_partlyFilled_longLived;
```

When Oscar++ allocates an object, it preferentially selects the cached partly filled alias of the appropriate lifetime. This means that, for any canonical page, there is only at most one partly filled short-lived shadow and at most one partly filled long-lived shadow. If a partly filled alias is not available, it selects a cached empty alias; and if those are also not available, it creates an alias on demand. In all cases, if the number of additional objects allowed on the alias is non-zero, it caches the partly filled alias in the slot of appropriate lifetime.

4.4.2 Case Study: first perlbench benchmark

We found in a brief pilot study that it is surprisingly difficult, even given the lifetime oracle, for Oscar++ to improve upon Oscar++. We present, as a case study, the first (of three) `perlbench` benchmarks from SPEC CPU2006; this is one of the most promising benchmarks because it was allocation-intensive (57.4 million allocations), with 6.8 million of size 1 and lifetime 0 (i.e., allocate — use — free with no intervening allocations); a further 2.9 million allocations have size between 2 and 99 and lifetime 0. We configured Oscar++ to consider objects of lifetime 0 to be short-lived, and other objects to be long-lived. Figure 4.6 shows the runtime (x-axis) and memory (y-axis; user-mode only) usage of Oscar++ with and without lifetime prediction, for different values of n (maximum lifetime number of allocations per alias). Both variants of Oscar++ show a similar runtime-memory tradeoff: more objects per page result in more memory usage (due to stateholding of quarantined objects), with lower runtime. For n=0 (one object per page), both Oscar and Oscar++ are theoretically equivalent; the minor differences on the graph are due to experimental error. We did not explore extremely large values of n, for which we would predict both runtime and memory would be higher. For values of n between 1 and 10, Oscar++ with lifetime prediction generally offers runtime similar to Oscar++, but with lower memory overhead (usually equivalent to Oscar++ for n-1).

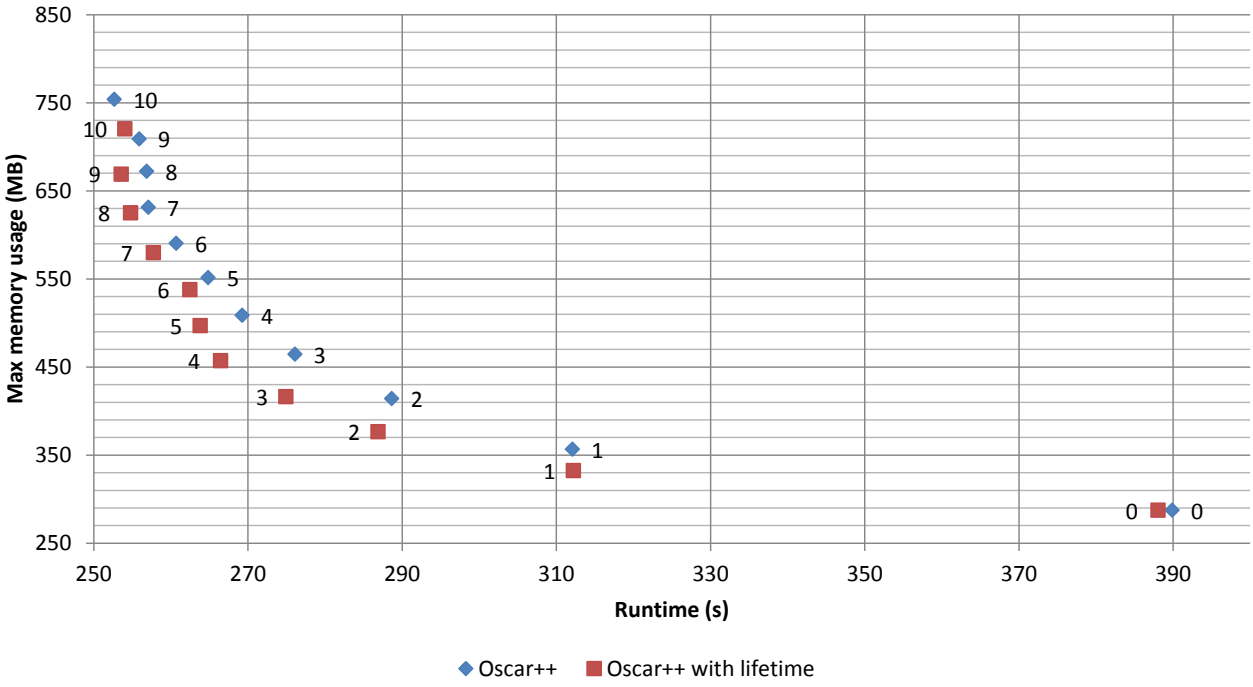
4.4.3 Discussion

The results for lifetime prediction are disappointing if we consider runtime: for example, given a fixed memory budget of 420MB, Oscar++ runs in 288.7s (n=2, reads the lifetime prediction data but does not use it[6]) or 274.9s (n=3, reads and uses the lifetime prediction data), which is a roughly 4.8% speedup. We chose these n values because they show a comparatively large runtime difference, with similar memory usage.

The runtime-memory curve does show, however, that it is simply very difficult to reduce runtime beyond n=2; for example, n=10 only reduces the overhead by 12.5% (compared to

[6]This gives a comparison that solely evaluates the benefit of segregating allocations by lifetime, without conflating the cost of lifetime prediction.

Figure 4.6: Runtime and user-mode memory of Oscar++ with and without lifetime prediction on the 1st perlbench benchmark. Individual data points show different values of n (maximum lifetime number of additional allocations per alias).



n=2), while increasing memory usage by 82%. The results are more favorable if we consider memory usage while holding runtime constant: for example, if our runtime budget is 270s, we can either use 551.6MB of memory (n=5, without using lifetime prediction) or somewhere between 457.2-497.0MB of memory (n=4 or 5, using lifetime prediction), which is a memory saving of between 9.9-17.1%.

These results assumed the presence of a lifetime oracle. A practical system could predict lifetime using the allocation size and allocation site (including unwinding allocation wrappers, similar to Cling [3]), under the assumption that allocations of the same type are likely to have similar lifetimes. In the absence of a reliable oracle for predicting lifetime, there are two techniques to mitigate the pathological memory stateholding scenario described in section 4.1. One method is to enforce that all objects on a particular alias are within a size range [sizeMin, sizeMax]. With n objects per alias, in the worst case, (n-1) objects of sizeMax are quarantined while one object of sizeMin is still live, resulting in a memory usage blowup of $\frac{(n-1)*sizeMax}{sizeMin}$. This can be efficiently implemented in the cache of aliases, by changing the array of aliases (some empty, some non-empty) to an array of empty aliases plus scalars for partly-filled aliases for objects in different classes of allocation sizes (e.g., [0,15], [16,31], [32,63], ...) i.e.,

```
void** shadows_empty;
void* shadow_partlyFilled_verySmall; // Size [0,15]
void* shadow_partlyFilled_small;     // Size [16,31]
void* shadow_partlyFilled_med;       // Size [32,63]
... // Can also use array: void** shadows_partlyFilled [...]
```

Another method is to avoid placing additional objects on an alias that contains live objects, i.e., we wait until all objects on the alias are freed before we place a new object on the alias. If the object on an alias is short-lived, then it will be freed and we can add another object to it in due time; but if the object is long-lived, then we have avoided colocating another object, which may be short-lived. There is still some remaining stateholding risk: suppose we have cumulatively placed k (short-lived) objects on an alias, with each of them having been sequentially freed/quarantined, and then we place another object which is long-lived. Overall, this method roughly halves the amount of memory stateholding.[7]

4.5 Future Work: Hotness

Next, we discuss some possible directions for further improvement that we have not yet explored.

The strategy of co-locating objects based on lifetime reduced the amount of stateholding of memory, and also partly reduces TLB pressure. We might be able to further reduce TLB pressure if we consider whether objects are "hot" (frequently accessed) or cold. For example, suppose we have four long-lived objects, two of which are hot (H1, H2) and two are cold (C1, C2). Since all objects are long-lived, the amount of physical memory usage is fixed regardless of placement onto aliases; but if we place H1, H2 onto alias1 and C1, C2 onto alias2, then we will rarely need to access alias2.

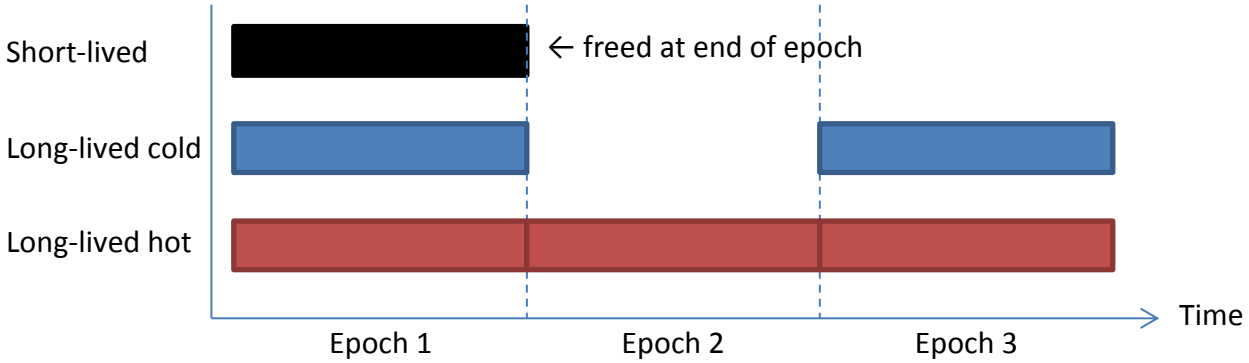
This generalizes when also considering short-lived objects. Suppose we have three access patterns for objects: short-lived[8], long-lived cold, and long-lived hot. Figure 4.7 shows whether these objects are accessed during different epochs.

Epoch 1 imposes no constraints: all objects are in use, so any placement (onto a fixed set of aliases) is optimal. In epoch 2, only the long-lived hot objects are accessed, so we want all of them to share a set of aliases. Epoch 2 does not impose any constraint on short-lived

[7]Suppose there is a small probability p of an object being long-lived, and we place k objects per page. Ordinarily with Oscar++, if there is a long-lived object, then there is stateholding of the other (k-1) objects. If we use this sequential placement strategy, there is the same probability that a long-lived object exists, but there is only stateholding of (k-1) objects if the long-lived object is the last to be added to the alias; if it is the first object on the alias, there is zero stateholding. On average, there will be stateholding of $\frac{(k-1)}{2}$ objects. Since p is small, the probability of two or more long-lived objects is negligible.

[8]As a simplifying assumption, it is not necessary to distinguish between short-lived hot and short-lived cold: a short-lived object is accessed at least once (during initialization), and does not live long enough for hotness to matter.

Figure 4.7: Access patterns: short-lived, long-lived cold, and long-lived hot objects.



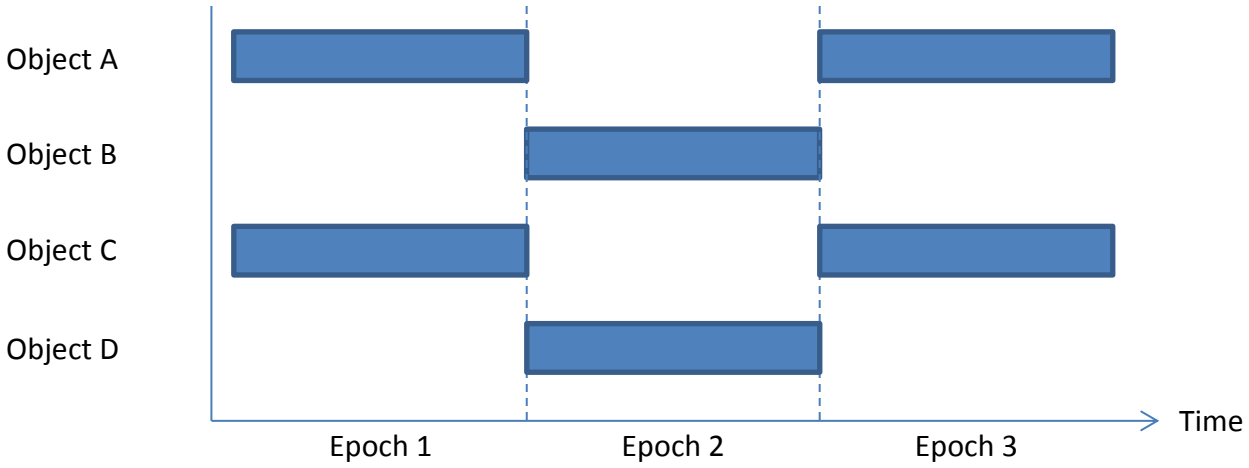
and long-lived cold objects from sharing aliases. In epoch 3, long-lived cold and long-lived hot objects can share aliases, while short-lived objects should be segregated. Under the assumption that every type of object can be tightly packed onto the desired aliases, this leads to the "obvious" conclusion that short-lived, long-lived cold, and long-lived hot objects should be placed on aliases with their kindred objects.

The aforementioned is only a first-order approximation. A smarter algorithm could consider which long-lived cold objects are accessed together in each epoch, e.g., in Figure 4.8, objects A,C should be preferentially co-located, and B,D should be co-located. Long-lived hot and short-lived objects require less analysis: long-lived hot objects are, by definition, frequently accessed, so they are always accessed together (i.e., we can just arrange them onto any hot alias); and for short-lived objects, a greedy algorithm (pack objects onto the non-empty alias) would preferentially co-locate objects that are accessed within the same epoch.

Unfortunately, recording all memory accesses with tools such as MadT, pinatrace, or lackey/valgrind commonly introduces a 1,000–10,000x slowdown [20]. Since we only need to determine hotness of an object, and not the precise number of accesses, a sampling approach may suffice; this would require some adaptation of those tools.

We have not implemented any of the ideas in this section, but believe that they may make interesting directions for future work.

Figure 4.8: Access patterns: a finer-grained look at long-lived cold objects.



Chapter 5

Conclusion

This dissertation tackled a subset of the longstanding, ongoing problem of memory safety vulnerabilities in legacy programs: quantifying and reducing the cost of shadow stacks to tackle the key attack vector of return-oriented programming, and reducing the cost of directly enforcing heap temporal memory safety.

Our results show that "well-known", old techniques may be worth revisiting: we devised parallel shadow stacks, advanced page-permissions-based protection using Oscar, and then further reduced the overhead with Oscar++ by changing the assumption of one object per alias.

Our work on shadow stacks showed that instrumenting every instance of a large class of operations (e.g., prologues and `RET`s) is expensive, despite our minimalist instrumentation. Likewise, instrumenting every `malloc` and `free()` can be costly (for allocation-intensive applications), but we are able to avoid instrumenting any pointer arithmetic or dereferencing operations.

Both shadow stacks and Oscar are comparatively simple schemes, with less indirection than many alternative schemes in the literature. This lends itself to lower overhead (shown empirically) and better compatibility (e.g., with `setjmp`/`longjmp` for shadow stacks, and typecasts and source-code requirements for heap protection).

Additionally, our work also shows that preventing instead of detecting attacks can be a worthwhile trade-off, with checking vs. overwriting return values for shadow stacks, or using quarantine for Oscar++.

Appendix A

Shadow Stacks

A.1 Traditional Shadow Stacks

For the traditional shadow stack epilogue, we can omit the comparison with the sentinel value (`CMP $0, (%ecx); JZ empty`). This is secure if we assume that the logical bottom of the shadow stack is filled with zeros (which would not be a useful return address for an attacker) until a page boundary, below which is a guard page. Hence, if no match is found, it will eventually hit the guard page and abort.

A.2 Parallel Shadow Stacks

A.2.1 Prologues

No scratch registers assumed:

- P1:[1]

```
1 POPL -0xb0000004(%esp) # Copy ret addr to shadow stack
2 LEAL -4(%esp), %esp    # Fix up stack pointer
```

- P2:

```
1 XCHGL (%esp), %ecx             # Ret addr in %ecx; old %ecx in stack
2 MOVL  %ecx, -0xb0000000(%esp) # Copy ret addr to shadow stack
3 XCHGL (%esp), %ecx             # Restore ret addr to main stack
4                                #     and restore old %ecx
```

[1] Internal names

- P3:
  - This is more complex than P1, but has the advantage of being safe in the presence of signal/interrupt handlers on 32-bit Linux (see Section 2.9.5).

```
1 POPL  -0xb0000004(%esp) # Copy ret addr to shadow stack
2 PUSHL -0xb0000004(%esp) # Fix up stack pointer
```

- P4:
  - `SUB` is a shorter and simpler alternative to prologue P1's `LEA`, but this has the disadvantage of setting flags.

```
1 POPL -0xb0000004(%esp) # Copy ret addr to shadow stack
2 SUB  $4, %esp          # Fix up stack pointer
```

- P5* (a naïve implementation that uses a scratch register, saving it to and restoring it from the stack):

```
1 PUSHL %ecx
2 MOVL  4(%esp), %ecx             # Copy ret addr to %ecx
3 MOVL  %ecx, -0xaffffffc(%esp) # Copy ret addr to shadow stack
4 POPL  %ecx
```

Assumes `%ecx` is available (e.g., due to calling convention) as a scratch register:

- P6 (compare to P5):

```
1 MOVL (%esp), %ecx              # Copy ret addr to %ecx
2 MOVL %ecx, -0xb0000000(%esp) # Copy ret addr to shadow stack
```

A.2.2 Epilogues: Overwriting

No scratch registers assumed:

- E1/E4:
  - The alternate epilogue is obtained by replacing `$0` with `%esp`. With a non-executable stack, `%esp` is an invalid return address, so it can be used for "zeroing-out" the stack, with a shorter instruction encoding than `$0`.

```
1 PUSH -0xa0000000(%esp)      # Copy ret addr from shadow stack
2 MOVL $0, -0x9ffffffc(%esp) # Zero out; or MOVL %esp, ...
3 RET  $4
```

- E2/E3:

```
1 ADDL  $4, %esp              # Fix up stack pointer
2 PUSHL -0xa0000004(%esp)     # Copy ret addr from shadow stack
3 MOVL  $0, -0xa0000000(%esp) # Zero out; or MOVL %esp, ...
4 RET
```

- E5/E6:
  - Prologue E2/E3's `ADD` is a shorter and simpler alternative to `LEA`, but has the disadvantage of setting flags.

```
1 LEAL  4(%esp), %esp         # Fix up stack pointer
2 PUSHL -0xa0000004(%esp)     # Copy ret addr from shadow stack
3 MOVL  $0, -0xa0000000(%esp) # Zero out; or MOVL %esp, ...
4 RET
```

- E7:

```
1 XCHGL -0xa0000000(%esp), %ecx # Ret addr in %ecx; old %ecx in shadow
2 MOVL  %ecx, (%esp)            # Main stack has correct ret addr
3 XOR   %ecx, %ecx              # Zero out %ecx
4 XCHGL -0xa0000000(%esp), %ecx # Restore old %ecx; shadow stack zeroed
5 RET
```

- E8 (a naïve implementation that uses a scratch register, saving it to and restoring it from the stack):

```
1 PUSHL %ecx
2 MOVL  -0x9ffffffc(%esp), %ecx # Copy ret addr to %ecx
3 MOVL  $0, -0x9ffffffc(%esp)   # Zero out; or MOVL %esp, ...
4 MOVL  %ecx, 4(%esp)           # Copy ret addr to main stack
5 POPL  %ecx
6 RET
```

- E30 (identical to E2/E3 but uses an indirect jump instead of a `RET`):

```
1 ADD  $4, %esp            # Fix up stack pointer
2 JMPL *-0xa0000004(%esp) # Use ret addr from shadow stack
```

Assumes `%ecx` is available (e.g., due to calling convention) as a scratch register:

- E9/E10* (compare to E8):

```
1 MOVL -0xa0000000(%esp), %ecx #
2 MOVL $0, -0xa0000000(%esp) # Zero out; or MOVL, %esp, ...
3 MOVL %ecx, (%esp)
4 RET
```

- E10_jmp* (identical to E10 but uses an indirect jump instead of a `RET`):

```
1 MOVL -0xa0000000(%esp), %ecx
2 MOVL %esp, -0xa0000000(%esp)
3 ADD  $4, %esp
4 JMPL *%ecx
```

- E16:
  - We store the shadow stack address in `%ecx`.

```
1 LEAL  -0xa0000000(%esp), %ecx # Store address of shadow stack entry
2 PUSHL (%ecx)                  # Copy shadow stack entry
3 MOVL  %esp, 4(%ecx)           # ``Zero out'' shadow stack entry
4 RET   $4
```

- E20 (compare to E8):

```
1 MOVL -0xa0000000(%esp), %ecx # Copy ret addr to %ecx
2 MOVL %ecx, (%esp)           # Copy ret addr to main stack
3 RET
```

Assumes `%ecx` and `%edx` are available (e.g., due to calling convention) as scratch registers:

- E11:

```
1 XORL %edx, %edx             # %edx = 0
2 MOVL -0xa0000000(%esp), %ecx # Copy ret addr to %ecx
3 MOVL %edx, -0xa0000000(%esp) # Zero out shadow stack
4 MOVL %ecx, (%esp)           # Copy ret addr to main stack
5 RET
```

- E12:
  - We store the shadow stack offset in `%edx`.
  - `(%esp,%edx,1)` is equivalent to `%esp + %edx` i.e., the address of the shadow stack entry.

```
1 MOVL $-0xa0000000, %edx  # Store shadow stack offset in %edx
2 MOVL (%esp,%edx,1), %ecx # Copy ret addr to %ecx
3 MOVL $0, (%esp,%edx,1)   # Zero out shadow stack
4 MOVL %ecx, (%esp)        # Copy ret addr to main stack
5 RET
```

- E13:
  - We store the shadow stack address in `%edx`.

```
1 LEAL -0xa0000000(%esp), %edx # Store address of shadow stack entry
2 MOVL (%edx), %ecx            # Copy ret addr to %ecx
3 MOVL $0, (%edx)              # Zero out shadow stack
4 MOVL %ecx, (%esp)            # Copy ret addr to main stack
5 RET
```

- E14:
  - We store the shadow stack address in `%edx`.

```
1 LEAL -0xa0000000(%esp), %edx # Store address of shadow stack entry
2 MOVL (%edx), %ecx           # Copy ret addr to %ecx
3 MOVL %ecx, (%esp)           # Copy ret addr to main stack
4 XORL %ecx, %ecx             # %edx = 0
5 MOVL %ecx, (%edx)           # Zero out shadow stack
6 RET
```

- E15:
  - We store the shadow stack address in `%edx`.

```
1 LEAL -0xa0000000(%esp), %edx # Store address of shadow stack entry
2 MOVL (%edx), %ecx           # Copy ret addr to %ecx
3 MOVL %esp, (%edx)           # ``Zero out'' shadow stack entry
4 MOVL %ecx, (%esp)           # Copy ret addr to main stack
5 RET
```

A.2.3 Epilogues: Checking

No scratch registers assumed:

- E101:
  - `abort$$` is a placeholder.

```
    PUSH %ecx
    MOV 4(%esp), %ecx            # Store ret addr in %ecx
    CMP %ecx, -0x9ffffffc(%esp) # Compare to shadow stack
    POP %ecx
    JNZ abort$$
    RET
abort$$:
    HLT
```

- E102:

```
    XCHG -0xa0000000(%esp), %ecx # Ret addr in %ecx; old %ecx in shadow
    CMP %ecx, (%esp)             # Compare to  main stack
    XCHG -0xa0000000(%esp), %ecx # Restore old %ecx and shadow stack
    JNZ abort$$
    RET
abort$$:
    HLT
```

- E104:

```
    XCHG %eax, (%esp)                 # Ret addr in %eax
    CMPXCHG %esp, -0xa0000000(%esp) # See explanation below
                                      # %eax will have correct ret addr
    XCHG %eax, (%esp)                 # Restore %eax (and move ret addr
                                      #     to main stack -- redundant)
    JNZ abort$$
    RET
abort$$:
    HLT
```

The `CMPXCHG %esp, -0xa0000000(%esp)` instruction operates as:

```
    if (%eax == -0xa0000000(%esp)) {
        // Ret address matches shadow stack
        ZeroFlag = 1
        -0xa0000000(%esp) = %esp // ``Zero out'' shadow stack
    } else {
```

```
6           // Ret address does not match
7           ZeroFlag = 0
8           %eax = -0xa0000000(%esp) // %eax contains correct return address
9       }
```

At the end of both code paths, `%eax` will contain the correct return address. However, the purpose of the `XCHG` instruction is *not* to save the correct return address to the main stack – if the copy on the main stack was incorrect, it will abort anyway – but to restore the original value of `%eax`.

- E105:
  - This is similar to E104, but it saves and restores the contents of `%eax` using `PUSH/POP` instead of `XCHG`s, and it does not bother to save the correct return address to the main stack in the case of a mismatch (since it will abort anyway).

```
1       PUSH %eax
2       MOV 4(%esp), %eax                   # Ret addr in %eax
3       CMPXCHG %esp, -0x9ffffffc(%esp) # See explanation above
4       JNZ abort$$
5       POP %eax
6       RET
7   abort$$:
8       HLT
```

Assumes `%ecx` is available (e.g., due to calling convention) as a scratch register:

- E100*:

```
1       MOV (%esp), %ecx                # Ret addr in %ecx
2       CMP %ecx, -0xa0000000(%esp) # Compare ret addr to shadow stack
3       JNZ abort$$
4       RET
5   abort$$:
6       HLT
```

- E100_jmp*:

```
1       MOV -0xa0000000(%esp), %ecx # Ret addr in %ecx
2       CMP %ecx, (%esp)                # Compare ret addr to main stack
3       JNZ abort$$
4
5       ADD $4, %esp                    # Fix up stack pointer
6       JMPL *%ecx                      # Indirect jump instead of RET
```

```
7 abort$$:
8     HLT
```

Assumes %eax is available (e.g., due to calling convention) as a scratch register:

- E103:

```
1     POP %eax                              # Ret addr in %eax
2     CMPXCHG %esp, -0xa0000004(%esp) # See explanation above
3                                           # %eax will have correct ret addr
4     JNZ abort$$
5     PUSH %eax                             # Copy ret addr to main stack
6     RET
7 abort$$:
8     HLT
```

A.2.4 Peephole Optimizations

- See Section 2.9.5 for caveats about signal/interrupt handlers on 32-bit Linux.
- We use our parallel shadow stack instrumentation as an example.

Prologue: basic optimization.

Instrumented prologues will often be of the form:

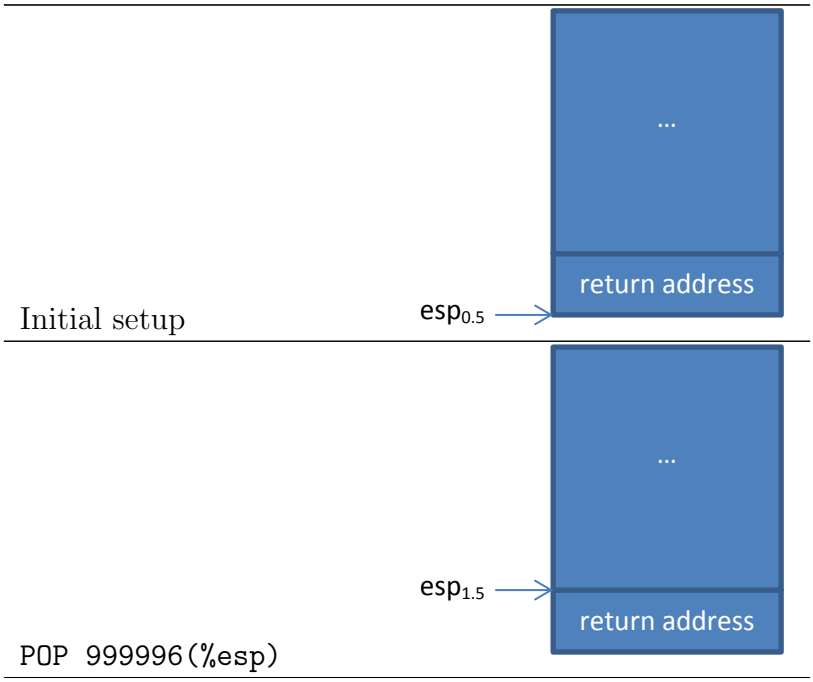
```
1 POP   999996(%esp)
2 SUB   $4, %esp
3 PUSH %ebp
4 MOV   %esp, %ebp
5 SUB   <X>, %esp
```

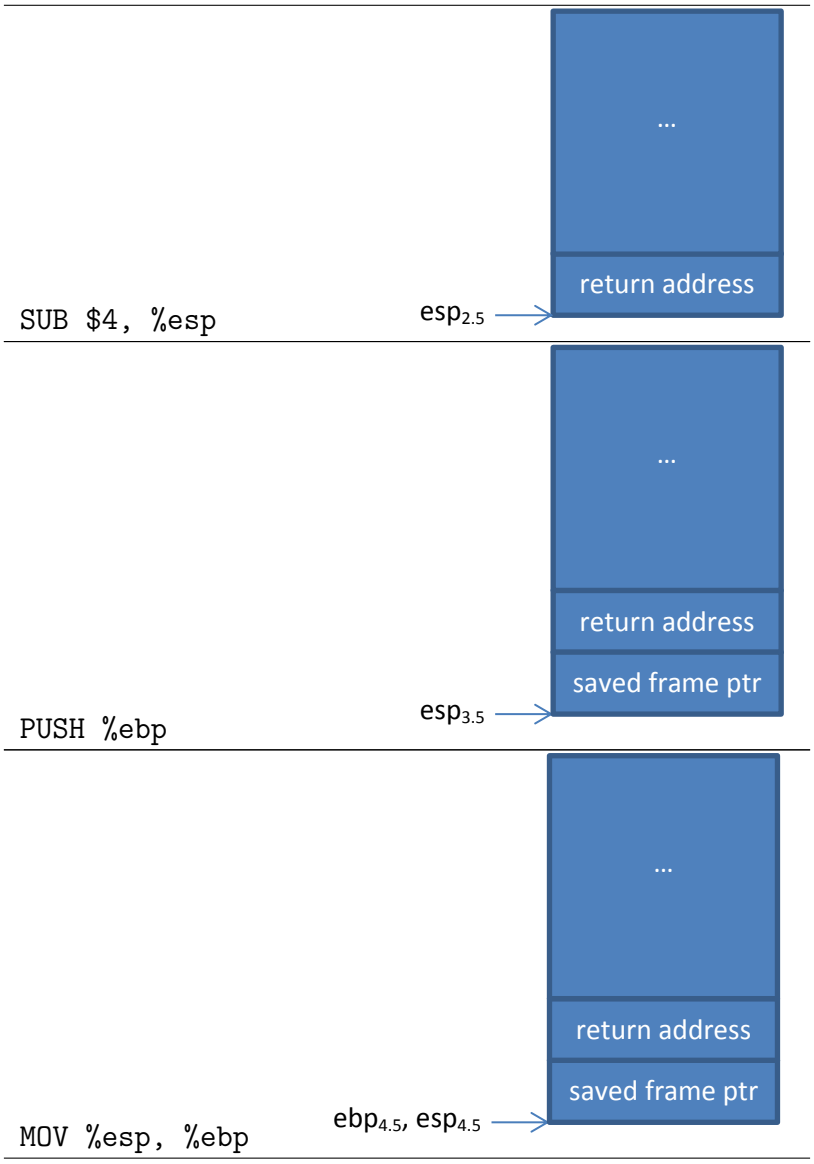
whereby the last three lines are the standard idiom for functions with frame pointers. We could replace this with:

```
1 POP 999996(%esp)
2 MOV %ebp, -8(%esp)
3 LEA -8(%esp), %ebp
4 SUB <X+8>, %esp
```

Tables A.1 and A.2 show step-by-step walkthroughs of the instrumented prologues before and after peephole optimization.

Table A.1: Instrumented prologue before peephole optimization.




...
return address
SUB $4, %esp
esp2.5
...
return address
saved frame ptr
PUSH %ebp
esp3.5
...
return address
saved frame ptr
MOV %esp, %ebp
ebp4.5, esp4.5

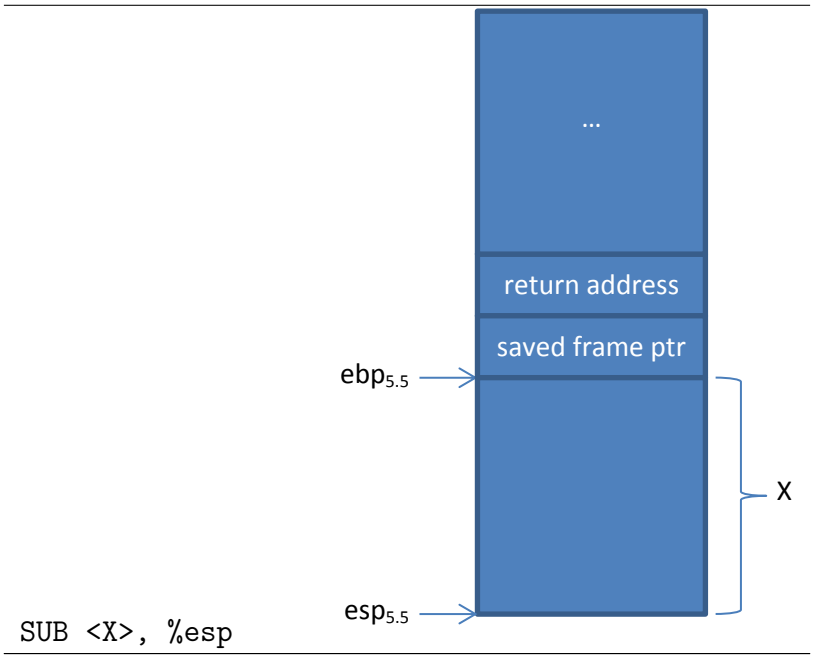
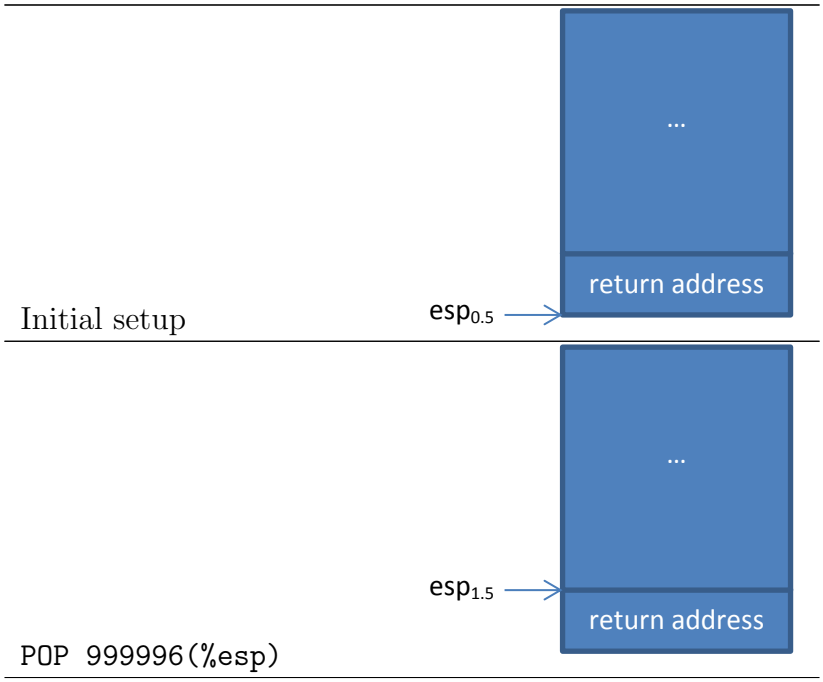


Table A.2: Instrumented prologue after peephole optimization.



Epilogues without `LEAVE`: basic optimization.

Instrumented epilogues that have frame pointers but don't use the `LEAVE` instruction will often be of the form:

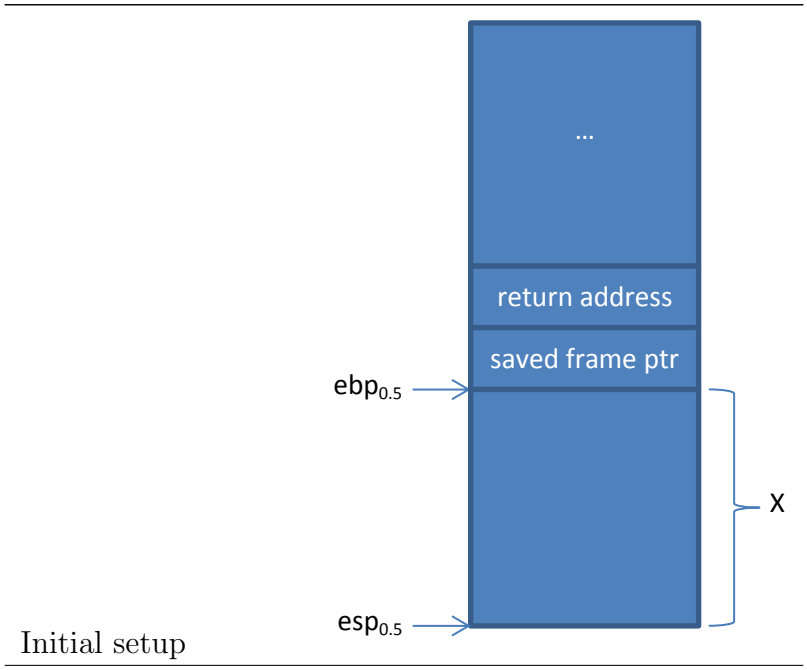
```
1 MOV  %ebp, %esp
2 POP  %ebp
3 SUB  $4, %esp
4 PUSH 999996(%esp)
5 RET
```

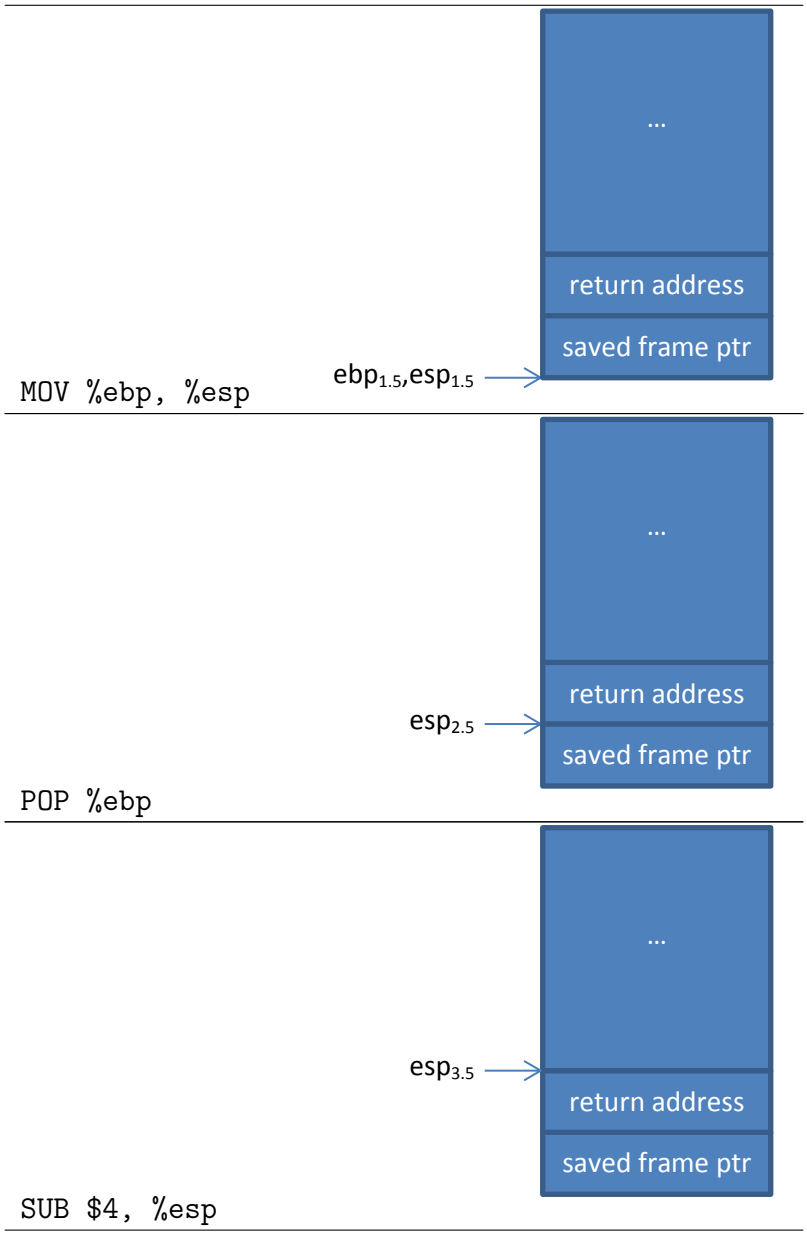
which can be converted into:

```
1 LEA  8(%ebp), %esp
2 MOV  -8(%esp), %ebp
3 PUSH 999996(%esp)
4 RET
```

Tables A.3 and A.4 show step-by-step walkthroughs of the instrumented epilogues before and after peephole optimization.

Table A.3: Instrumented epilogue before peephole optimization. "retaddr from PSS" refers to return address from parallel shadow stack.




...
return address
saved frame ptr
ebp1.5,esp1.5
MOV %ebp, %esp
...
return address
esp2.5
saved frame ptr
POP %ebp
...
esp3.5
return address
saved frame ptr
SUB $4, %esp

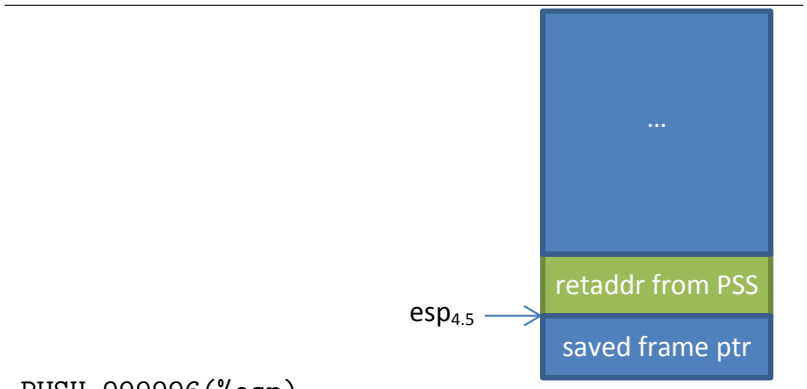
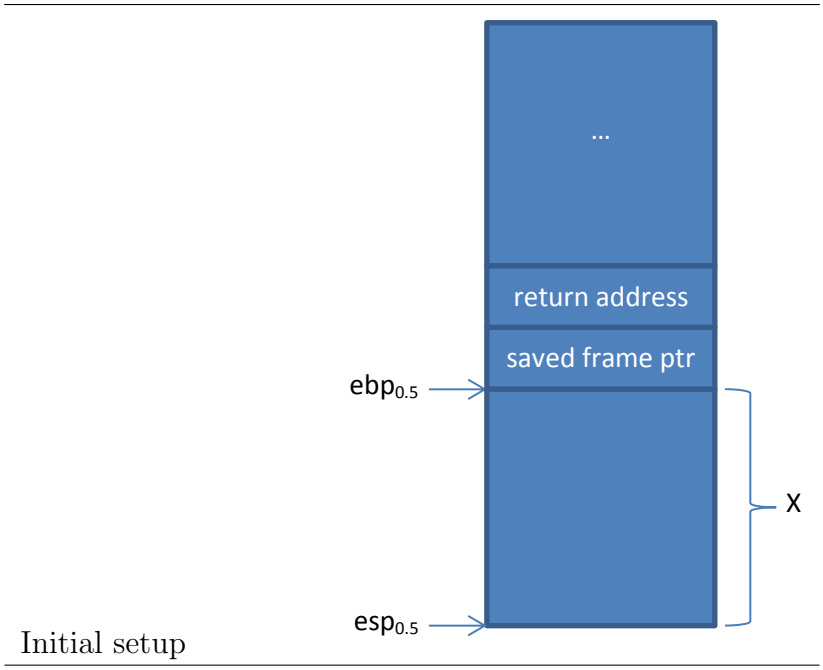
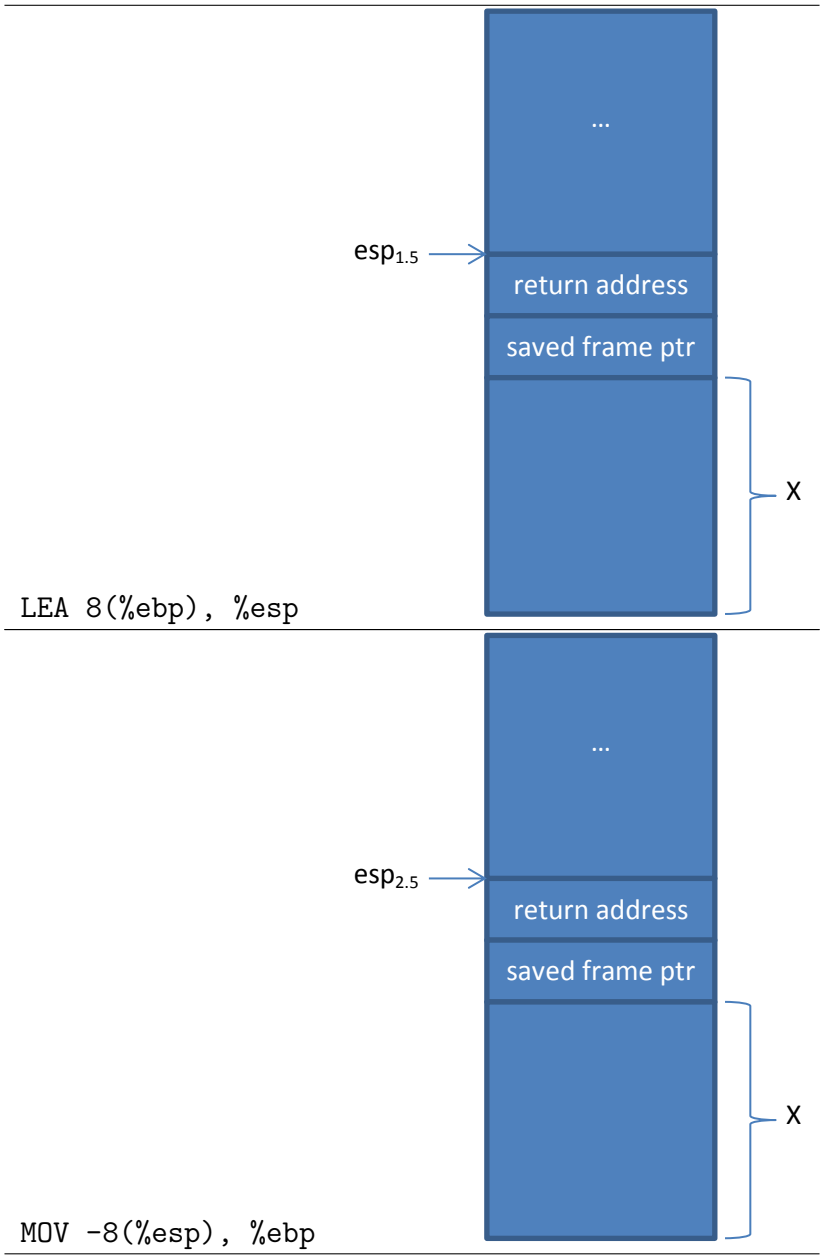
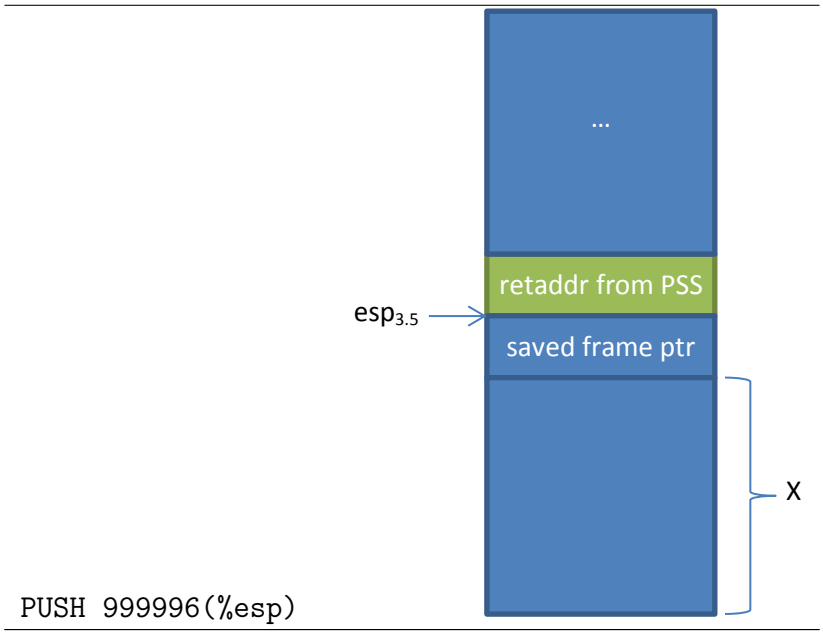


Table A.4: Instrumented epilogue after peephole optimization. "retaddr from PSS" refers to return address from parallel shadow stack.




...
esp1.5
return address
saved frame ptr
X
LEA 8(%ebp), %esp
...
esp2.5
return address
saved frame ptr
X
MOV -8(%esp), %ebp



Epilogues with `LEAVE`: `RET` with offset.

We can combine the stack pointer adjustment with the `RET` instruction i.e.,

```
1 LEAVE  ...
2 SUB    $4, %esp
3 PUSH   999996(%esp)
4 RET
```

is equivalent to:

```
1 LEAVE  ...
2 PUSH   1000000(%esp)
3 RET    $4
```

Note that the return address from the shadow stack is copied *below* the return address on the main stack.

Epilogue: indirect jump.

In the instrumented epilogue, we could replace the `RET`:

```
1 # standard epilogue omitted
2 ...
3 SUB   $4, %esp
4 PUSH  999996(%esp)
5 RET
```

with an indirect jump:

```
1 # standard epilogue omitted
2 ...
3 SUB $4, %esp
4 JMP 999996(%esp)
```

This optimization can be applied/combined to the peephole optimized epilogues presented above (with or without `LEAVE`).

Bibliography

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. "Control-flow integrity principles, implementations, and applications". In: *TISSEC* (2009).

[2] *Advancing Moore's Law in 2014!* `http://www.intel.com/content/dam/www/public/us/en/documents/presentation/advancing-moores-law-in-2014-presentation.pdf`. Aug. 2014.

[3] Periklis Akritidis. "Cling: A Memory Allocator to Mitigate Dangling Pointers." In: *USENIX Security*. 2010, pp. 177–192.

[4] *ARM Information Center*. `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0439d/Chdedegj.html`. Sept. 2013.

[5] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. "Workload analysis of a large-scale key-value store". In: *ACM SIGMETRICS Performance Evaluation Review*. Vol. 40. 1. ACM. 2012, pp. 53–64.

[6] Arash Baratloo, Navjot Singh, and Timothy K. Tsai. "Transparent Run-Time Defense Against Stack-Smashing Attacks." In: *USENIX ATC*. 2000.

[7] David A. Barrett and Benjamin G. Zorn. "Using lifetime predictors to improve memory allocation performance". In: *ACM SIGPLAN Notices*. Vol. 28. 6. ACM. 1993, pp. 187–196.

[8] Emery D. Berger and Benjamin G. Zorn. "DieHard: probabilistic memory safety for unsafe languages". In: *ACM SIGPLAN Notices* 41.6 (2006), pp. 158–168.

[9] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. "Reconsidering custom memory allocation". In: *ACM SIGPLAN Notices* 48.4S (2013), pp. 46–57.

[10] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. "Efficient Techniques for Comprehensive Protection from Memory Error Exploits". In: *USENIX Security*. 2005.

[11] Sarah Bird, Aashish Phansalkar, Lizy K. John, Alex Mericas, and Rajeev Indukuru. "Performance Characterization of SPEC CPU Benchmarks on Intel's Core Microarchitecture Based Processor". In: *SPEC Benchmark Workshop*. 2007.

[12] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazieres, and Dan Boneh. "Hacking Blind". In: *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE. 2014, pp. 227–242.

[13] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. "Jump-oriented programming: a new class of code-reuse attack". In: *CCS*. 2011.

[14] Linux Programming Blog. *Threads and fork(): think twice before mixing them*. June 2009. URL: `https://www.linuxprogrammingblog.com/threads-and-fork-think-twice-before-using-them`.

[15] Jeff Bonwick et al. "The Slab Allocator: An Object-Caching Kernel Memory Allocator." In: *USENIX summer*. Vol. 16. Boston, MA, USA. 1994.

[16] Mihai Budiu, Úlfar Erlingsson, and Martín Abadi. "Architectural support for software-based protection". In: *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*. 2006.

[17] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. "Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities". In: *International Symposium on Software Testing and Analysis*. ACM. 2012, pp. 133–143.

[18] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. "Control-flow bending: On the effectiveness of control-flow integrity". In: *USENIX Security*. 2015, pp. 161–176.

[19] Nicholas Carlini and David Wagner. "ROP is still dangerous: Breaking modern defenses". In: *USENIX Security*. 2014.

[20] Marco Cesati, Renato Mancuso, Emiliano Betti, and Marco Caccamo. *MadT: A memory access detection tool for symbolic memory profiling*. Tech. rep. UIUC, 2015.

[21] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. "Return-oriented programming without returns". In: *CCS*. 2010.

[22] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K. Iyer. "Non-Control-Data Attacks Are Realistic Threats." In: *USENIX Security*. Vol. 5. 2005.

[23] Xi Chen, Asia Slowinska, and Herbert Bos. "On the detection of custom memory allocators in C binaries". In: *Empirical Software Engineering* (2015), pp. 1–25.

[24] Xi Chen, Asia Slowinska, and Herbert Bos. "Who allocated my memory? Detecting custom memory allocators in C binaries." In: *WCRE*. 2013, pp. 22–31.

[25] Tzi-cker Chiueh and Fu-Hau Hsu. "RAD: A compile-time solution to buffer overflow attacks". In: *ICDCS*. 2001.

[26] Marc L. Corliss, E. Christopher Lewis, and Amir Roth. "Using DISE to protect return addresses from attack". In: *ACM SIGARCH Computer Architecture News* (2005).

[27] Intel Corporation. *5-Level Paging and 5-Level EPT*. May 2017. URL: `https://software.intel.com/sites/default/files/managed/2b/80/5-level_paging_white_paper.pdf`.

[28] Standard Performance Evaluation Corporation. *Readme 1st CPU2006*. URL: https://www.spec.org/cpu2006/Docs/readme1st.html#Q21.

[29] Standard Performance Evaluation Corporation. *SPEC CPU2006: Read Me First*. http://www.spec.org/cpu2006/Docs/readme1st.html. Sept. 2011.

[30] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. "PointGuard: protecting pointers from buffer overflow vulnerabilities". In: *USENIX Security*. Vol. 12. 2003, pp. 91–104.

[31] Christopher Dahn and Spiros Mancoridis. "Using program transformation to secure C programs against buffer overflows". In: *20th Working Conference on Reverse Engineering*. 2003.

[32] Lucas Davi, Patrick Koeberl, and Ahmad-Reza Sadeghi. "Hardware-Assisted Fine-Grained Control-Flow Integrity: Towards Efficient Protection of Embedded Systems Against Software Exploitation". In: *DAC*. 2014.

[33] Lucas Davi, Daniel Lehmann, Ahmad-Reza Sadeghi, and Fabian Monrose. "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection". In: *USENIX Security*. 2014.

[34] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. "ROPdefender: A detection tool to defend against return-oriented programming attacks". In: *CCS*. 2011.

[35] Jared DeMott. *UaF: Mitigation and Bypass*. Jan. 2015. URL: https://bromiumlabs.files.wordpress.com/2015/01/demott_uaf_migitation_and_bypass2.pdf.

[36] Solar Designer. *lpr LIBC RETURN exploit*. http://insecure.org/sploits/linux.libc.return.lpr.sploit.html. 1997.

[37] Dinakar Dhurjati and Vikram Adve. "Efficiently detecting all dangling pointer uses in production servers". In: *Dependable Systems and Networks*. IEEE. 2006, pp. 269–280.

[38] Gregory J Duck, Roland HC Yap, and Lorenzo Cavallaro. "Stack bounds protection with low fat pointers". In: *Symposium on Network and Distributed System Security*. 2017.

[39] *Electric Fence*. http://elinux.org/index.php?title=Electric_Fence&oldid=369914. Jan. 2015.

[40] *Emerging 'Stack Pivoting' Exploits Bypass Common Security*. http://blogs.mcafee.com/mcafee-labs/emerging-stack-pivoting-exploits-bypass-common-security. May 2013.

[41] HP Enterprise. *Efficacy of MemoryProtection against use-after-free vulnerabilities*. URL: http://community.hpe.com/t5/Security-Research/Efficacy-of-MemoryProtection-against-use-after-free/ba-p/6556134#.VsFYB8v8vCK (visited on 2014).

[42] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C Necula. "XFI: Software guards for system address spaces". In: *OSDI*. 2006.

[43] Agner Fog. *Microarchitecture of Intel, AMD and VIA CPUs*. `http://www.agner.org/optimize/microarchitecture.pdf`. 2017.

[44] Agner Fog. *The microarchitecture of Intel, AMD and VIA CPUs*. `www.agner.org/optimize/microarchitecture.pdf`. Aug. 2014.

[45] Michael Frantzen and Michael Shuey. "StackGhost: Hardware Facilitated Stack Protection." In: *USENIX Security*. 2001.

[46] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Gerogios Portokalidis. "Out of control: Overcoming control-flow integrity". In: *IEEE S&P*. 2014.

[47] Google. *AddressSanitizerLeakSanitizer*. URL: `https://github.com/google/sanitizers/wiki/AddressSanitizerLeakSanitizer`.

[48] Suhas Gupta, Pranay Pratap, Huzur Saran, and S. Arun-Kumar. "Dynamic code instrumentation to detect and recover from return address corruption". In: *International workshop on Dynamic systems analysis*. 2006.

[49] Tejun Heo. *Patchwork [11/11] x86: implement x86_32 stack protector*. `https://patchwork.kernel.org/patch/6217/`. 2009.

[50] *How to use Pageheap.exe in Windows XP, Windows 2000, and Windows Server 2003*. `https://support.microsoft.com/en-us/kb/286470`.

[51] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. "Data-Oriented Programming: On the Expressive of Non-Control Data Attacks". In: *IEEE S&P*. 2016.

[52] Jan Hubicka, Andreas Jaeger, and Mark Mitchell. *System V Application Binary Interface: AMD64 Architecture Processor Supplement*. 2005.

[53] Koji Inoue. "Lock and Unlock: A Data Management Algorithm for A Security-Aware Cache." In: *ICECS*. 2006.

[54] Intel. *Control-flow Enforcement Technology Preview*. `https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf`. 2017.

[55] *Intel(R) 64 and IA-32 Architectures Optimization Reference Manual*. Mar. 2014.

[56] Ciji Isen and Lizy John. "On the Object Orientedness of C++ programs in SPEC CPU 2006". In: *SPEC Benchmark Workshop*. 2008.

[57] *Itanium(R) Processor Family Performance Advantages: Register Stack Architecture*. `https://software.intel.com/en-us/articles/itaniumr-processor-family-performance-advantages-register-stack-architecture`. Oct. 2008.

[58] Trevor Jim, J. Gregory Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. "Cyclone: A Safe Dialect of C." In: *USENIX ATC*. 2002, pp. 275–288.

[59] Wen-Fu Kao and S. Felix Wu. "Light-weight Hardware Return Address and Stack Frame Tracking to Prevent Function Return Address Attack". In: *International Conference on Computational Science and Engineering*.

[60] Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. "DangSan: Scalable Use-after-free Detection." In: *EuroSys*. 2017, pp. 405–419.

[61] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. "Code-Pointer Integrity". In: *OSDI*. 2014.

[62] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. "Preventing Use-after-free with Dangling Pointers Nullification." In: *NDSS*. 2015.

[63] Ruby B. Lee, David K. Karig, John P. McGregor, and Zhijie Shi. "Enlisting hardware architecture to thwart malicious code injection". In: *Security in Pervasive Computing*. 2004.

[64] Kangjie Lu, Wenke Lee, Stefan Nürnberger, and Michael Backes. "How to Make ASLR Win the Clone Wars: Runtime Re-Randomization." In: *NDSS*. 2016.

[65] Vitaliy B. Lvin, Gene Novark, Emery D. Berger, and Benjamin G. Zorn. "Archipelago: trading address space for reliability and security". In: *ACM SIGOPS Operating Systems Review* 42.2 (2008), pp. 115–124.

[66] Ali Jose Mashtizadeh, Andrea Bittau, David Mazières, and Dan Boneh. "Cryptographically Enforced Control Flow Integrity". In: *arXiv:1408.1451*. 2014.

[67] Henry Massalin. "Superoptimizer: a look at the smallest program". In: *ACM SIGPLAN Notices*. 1987.

[68] Stephen McCamant and Greg Morrisett. "Evaluating SFI for a CISC Architecture". In: *USENIX Security*. 2006.

[69] *Memcheck: a memory error detector*. `http://valgrind.org/docs/manual/mc-manual.html`.

[70] Microsoft. *A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003*. `https://support.microsoft.com/en-us/help/875352/a-detailed-description-of-the-data-execution-prevention-dep-feature-in`. n.d.

[71] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. "Producing wrong data without doing anything obviously wrong!" In: *ASPLOS*. 2009.

[72] Santosh Nagarakatte. personal communication. June 22, 2017.

[73] Santosh Ganapati Nagarakatte. *Practical low-overhead enforcement of memory safety for C programs*. Doctoral dissertation. University of Pennsylvania, 2012.

[74] Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. "Watchdog: Hardware for safe and secure manual memory management and full memory safety". In: *ACM SIGARCH Computer Architecture News* 40.3 (2012), pp. 189–200.

[75] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. "CETS: compiler enforced temporal safety for C". In: *ACM SIGPLAN Notices* 45.8 (2010), pp. 31–40.

[76] Danny Nebenzahl, Mooly Sagiv, and Avishai Wool. "Install-time vaccination of Windows executables to defend against stack smashing attacks". In: *Dependable and Secure Computing, IEEE Transactions on* (2006).

[77] George C. Necula, Scott McPeak, and Westley Weimer. "CCured: Type-safe retrofitting of legacy code". In: *ACM SIGPLAN Notices* 37.1 (2002), pp. 128–139.

[78] Aleph One. "Smashing the stack for fun and profit". In: *Phrack magazine* (1996).

[79] Pádraig O'Sullivan, Kapil Anand, Aparna Kotha, Matthew Smithson, Rajeev Barua, and Angelos D. Keromytis. "Retrofitting security in COTS software with binary rewriting". In: *Future Challenges in Security and Privacy for Academia and Industry*. 2011.

[80] *Ownership and moves*. `https://rustbyexample.com/scope/move.html`.

[81] Hilmi Ozdoganoglu, T.N. Vijaykumar, Carla E. Brodley, Benjamin A. Kuperman, and Ankit Jalote. "SmashGuard: A hardware solution to prevent security attacks on the function return address". In: *Computers, IEEE Transactions on* (2006).

[82] Seon-Ho Park, Young-Ju Han, Soon-Jwa Hong, Hyoung-Chun Kim, and Tai-Myoung Chung. "The Dynamic Buffer Overflow Detection and Prevention Tool for Windows Executables Using Binary Rewriting". In: *The 9th International Conference on Advanced Communication Technology*. 2007.

[83] PaX. *Address Space Layout Randomization*. `https://pax.grsecurity.net/docs/aslr.txt`. 2001.

[84] Mathias Payer and Thomas R. Gross. "Fine-grained user-space security through virtualization". In: *VEE*. 2011.

[85] Mathias Payer, Tobias Hartmann, and Thomas R. Gross. "Safe loading-a foundation for secure execution of untrusted programs". In: *IEEE S&P*. 2012.

[86] Manish Prasad and Tzi-cker Chiueh. "A Binary Rewriting Defense Against Stack based Buffer Overflow Attacks." In: *USENIX ATC*. 2003.

[87] Rui Qiao, Mingwei Zhang, and R. Sekar. "A Principled Approach for ROP Defense". In: *Proceedings of the 31st Annual Computer Security Applications Conference*. ACM. 2015, pp. 101–110.

[88] Anne Rogers, Martin C. Carlisle, John H. Reppy, and Laurie J. Hendren. "Supporting dynamic data structures on distributed-memory machines". In: *TOPLAS* 17.2 (1995), pp. 233–263.

[89] Ravi L. Sahita, Vedvyas Shanbhogue, Gilbert Neiger, Jonathan Edwards, Ido Ouziel, Barry E. Huntley, Stanislav Shwartsman, David M. Durham, Andrew V. Anderson, Michael Lemay, et al. *Method and apparatus for fine grain memory protection*. US Patent 20,150,378,633. Dec. 2015.

[90] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. "AddressSanitizer: A Fast Address Sanity Checker". In: *USENIX ATC*. 2012.

[91] Hovav Shacham. "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)". In: *CCS*. 2007.

[92] Kirill A. Shutemov. *[RFC, PATCHv1 00/28] 5-level paging*. Dec. 8, 2016. URL: `http://lkml.iu.edu/hypermail/linux/kernel/1612.1/00383.html`.

[93] Stelios Sidiroglou, Giannis Giovanidis, and Angelos D. Keromytis. "A dynamic mechanism for recovering from buffer overflow attacks". In: *Information security*. 2005.

[94] Matthew S. Simpson and Rajeev K. Barua. "MemSafe: ensuring the spatial and temporal memory safety of C at runtime". In: *Software: Practice and Experience* 43.1 (2013), pp. 93–128.

[95] Saravanan Sinnadurai, Qin Zhao, and Weng Fai Wong. *Transparent runtime shadow stack: Protection against malicious return address modifications*. `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.120.5702&rep=rep1&type=pdf`. 2008.

[96] *Software Optimization Guide for AMD Family 15h Processors*. Jan. 2012.

[97] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. "SoK: Eternal war in memory". In: *IEEE S&P*. IEEE. 2013, pp. 48–62.

[98] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. "SoK: Eternal war in memory". In: *IEEE S&P*. 2013.

[99] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. "Enforcing forward-edge control-flow integrity in gcc & llvm". In: *USENIX Security*. 2014.

[100] Vendicator. *Stack Shield*. `http://www.angelfire.com/sk/stackshield/info.html`. 2000.

[101] Perry Wagle and Crispin Cowan. "Stackguard: Simple stack smash protection for gcc". In: *GCC Developers Summit*. 2003.

[102] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. "Efficient software-based fault isolation". In: *SOSP*. 1993.

[103] David Weston and Matt Miller. *Microsoft's strategy and technology improvements toward mitigating arbitrary native code execution*. `https://cansecwest.com/slides/2017/CSW2017_Weston-Miller_Mitigating_Native_Remote_Code_Execution.pdf`. 2017.

[104] Jun Xu, Zbigniew Kalbarczyk, Sanjay Patel, and Ravishankar K. Iyer. "Architecture support for defending against buffer overflow attacks". In: *Workshop on Evaluating and Architecting Systems for Dependability*. 2002.

[105] Yves Younan. "FreeSentry: protecting against use-after-free vulnerabilities due to dangling pointers." In: *NDSS*. 2015.

[106] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, László Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. "Practical control flow integrity and randomization for binary executables". In: *IEEE S&P*. 2013.

[107] Mingwei Zhang, Rui Qiao, Niranjan Hasabnis, and R. Sekar. "A platform for secure static binary instrumentation". In: *VEE*. 2014.

[108] Mingwei Zhang and R. Sekar. "Control Flow Integrity for COTS Binaries". In: *USENIX Security*. 2013.