The Performance Cost of Shadow Stacks and Stack Canaries

Thurston H.Y. Dang
Electrical Engineering and Computer Sciences
University of California, Berkeley
thurston@eecs.berkeley.edu

Petros Maniatis
Google Inc.
maniatis@google.com

David Wagner
Electrical Engineering and Computer Sciences
University of California, Berkeley
daw@cs.berkeley.edu

ABSTRACT

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.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

General Terms

shadow stack, stack canary, stack cookie

1. INTRODUCTION

One classic security exploit is to redirect the control flow by overwriting a return address stored on the stack [36]. Although various mitigations (e.g., NX/DEP) have made this attack and some simple refinements (e.g., return-to-libc) infeasible, the current state of the art in exploitation – return-oriented programming (ROP [44]) – continues to depend on misusing `RET` instructions, this time by chaining together short sequences of instructions ("gadgets") that end in a `RET`.[1]

[1]The generalizations of ROP – jump-oriented programming [11], or ROP without returns [14] – are not commonly used in practice and will not be considered further.

ASIA CCS'15, April 14–17, 2015, Singapore.

ACM 978-1-4503-3245-3/15/04 ...$15.00.
http://dx.doi.org/10.1145/2714576.2714635.

These attacks could largely, in principle, be prevented using control-flow integrity (CFI) schemes [7], but CFI has not been widely adopted, in part due to its non-trivial overhead [53]. Recent works that have proposed CFI for binaries [53, 55] have greatly reduced the overhead by adopting coarse-grained policies. One notable relaxation is they adopt a more permissive policy for `RET` instructions, rather than tracking the return addresses precisely using a shadow stack (as had been proposed in Abadi et al.'s original formulation [7]). Unfortunately, such weaker policies were soon shown to be insecure [13,19,24]. 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 [21], 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. In this paper, we measure this performance 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 9), 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 Related Work). To our knowledge, they have only been implemented in recompilation-based schemes – thus negating the binary-rewritability benefits of coarse-grained CFI.

2. BACKGROUND

2.1 Traditional Shadow Stacks

The purpose of a shadow stack is to protect return addresses saved on the stack from tampering. Figure 1 (left) 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.

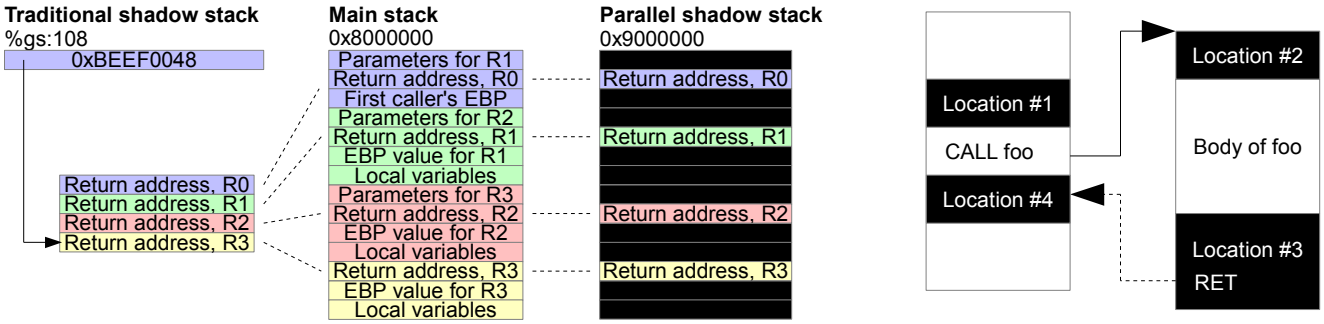

Figure 1: Left: traditional vs parallel shadow stacks. Rx refers to Routine #x. Right: possible locations for instrumentation.

```
SUB $4, %gs:108   # Decrement SSP
MOV %gs:108, %eax # Copy SSP into EAX
MOV (%esp), %ecx  # Copy ret. address into
MOV %ecx, (%eax)  #     shadow stack via ECX
```

Figure 2: Prologue for traditional shadow stack.

```
MOV %gs:108, %ecx # Copy SSP into ECX
ADD $4, %gs:108   # Increment SSP
MOV (%ecx), %edx  # Copy ret. address from
MOV %edx, (%esp)  #     shadow stack via EDX
RET
```

Figure 3: Epilogue for traditional shadow stack (overwriting).

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).

The return address can be saved before the `CALL` instruction ("Location #1" in Figure 1, right) or at the prologue of the called function ("Location #2"). The return address can be checked/overwritten before the `RET` instruction ("Location #3"); 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 [55]), we can check the return address at the return site ("Location #4").

Typically, schemes that propose changes to the hardware (e.g., Ozdoganoglu et al. [38]) find it convenient to instrument the `CALL`s, while binary-rewriting schemes often instrument function prologues and epilogues by replacing them with "trampolines" (indirect jumps) to the replacement code.

The prologue could be as per Figure 2, and the epilogue could be as per Figure 3 or 4 for the checking and overwriting policies, respectively.

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 intactness before the `RET`s of vulnerable functions [50].

```
MOV %gs:108, %ecx
ADD $4, %gs:108
MOV (%ecx), %edx
CMP %edx, (%esp) # Instead of overwriting,
JNZ abort        #     we compare
RET
abort:
    HLT
```

Figure 4: Epilogue for traditional shadow stack (checking).

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 [8, 39]. For completeness, we investigated the overhead of stack canaries.

3. CHALLENGES

3.1 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` [7].

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 [7, 40]. Some other architectures are immune, as they pass the return address using a "link register" [4].

Nonetheless, many shadow stack schemes (see Table 1) 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 per-

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

Figure 5: 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.

```
MOV %gs:108, %ecx # Copy SSP into ECX
MOV (%esp), %edx  # Copy ret. addr. into EDX
non_match:
    CMP $0, (%ecx)# Is shadow stack empty?
    JZ abort
    ADD $4, %ecx  # Increment our copy of SSP
    CMP %edx, -4(%ecx) # Check ret. addr.
    JNZ non_match # Loop until match
MOV %ecx, %gs:108 # Synchronize SSP
RET
abort:
    HLT
```

Figure 6: Epilogue for traditional shadow stack with checking and popping until a match.

formance impact of replacing `RET`s, since it is highly recommended to match `CALL`s with `RET`s [3,6]. Zhang et al. [53] 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 [55] goes to great lengths to maintain the `CALL-RET` matching despite TOCTTOU exploits, even adding extra stub function calls rather than using indirect jumps.

3.2 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 [42]) is to pop the shadow stack until a match is found, or the shadow stack is empty (denoted here with a sentinel value of zero). Figure 6 is our hand-compiled version of the C code from PSI [54].

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 [16, 38, 46].

```
POP 999996(%esp) # Copy ret addr to shadow stack
SUB $4, %esp # Fix up stack pointer (undo POP)
```

Figure 7: Prologue for parallel shadow stack.

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

Figure 8: Epilogue for parallel shadow stack.

4. DESIGN

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

4.1 Parallel Shadow Stacks

As we will show later in Section 7, 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. 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 1 (right), 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.

Our instrumentation for overwriting the return address can be as simple as adding two instructions to each uninstrumented prologue (Figure 7) and two instructions to each uninstrumented epilogue (Figure 8).

With peephole optimizations (see Appendix A), 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.

This instrumentation does not clobber any registers and can be easily modified to preserve the flags (through replacing `SUB/ADD` with `LEA`), 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.

There are two main disadvantages compared to a traditional shadow stack. Firstly, return addresses are rarely packed together, which means that each return address may even use up an entire cache line; the overhead of this depends on the calling patterns of the program. Additionally, 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; indeed, if we do not "zero out" (or corrupt) old shadow stack frame entries, then the attacker could even choose expired return addresses. 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`. Parallel shadow stacks do generally use more memory than a traditional shadow stack, but as the stack is small in comparison to the heap, and its pages can be committed only when needed, we do not consider this a major limitation.

Note that we do not claim that the overhead of a parallel shadow stack is low enough for widespread deployment.

4.2 Security Benefits

Our weakest threat model is when the attacker has a contiguous stack buffer overflow (e.g., `memcpy`), but does not have any information disclosure (i.e., while they may know the stack layout, they do not know the stack contents – such as the stack canary value or return value – nor the fixed shadow stack offset). Another threat model is where the attacker can perform arbitrary writes into the stack (e.g., repeated uses of `src[i] = s`, where `i` and `s` are under their control), but does not have any information disclosure. We also consider the previous two cases, with information disclosure (though not sufficient disclosure to defeat randomization).

Figure 5 summarizes, for each of these threat models, we consider 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).

5. AIMS

5.1 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 > 0 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).

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 [15, 42]. To provide a meaningful estimate of the overhead, it is important to use a standardized benchmark suite.

5.2 Assumptions

We assume that we have a tool similar to BinCFI [55] 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 [42]).

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 [49], though that is intended to be deployed as an assembly preprocessor in the ordinary build process.

6. METHOD

We ran SPEC CPU2006, 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 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`, using `apachebench` 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. [48], 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, 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 B): 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.

6.1 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 7), 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 ([48]), 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., multithreaded 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.

7. RESULTS

7.1 Cost of Instrumentation Schemes

The overheads (though not baseline times) from a pilot study were roughly the same for x86 vs x64 and -O2 vs -O3, hence, for brevity, we will only discuss the x86 -O3 results (Figure 9, left), as they are marginally lower, and therefore provide a lower bound.

Two of the programs (`perlbench`, `gcc`) did not work with the traditional shadow stack instrumentation; 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).[2] 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 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 9 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 9 (right).

8. DISCUSSION

8.1 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. [7] 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 10 shows the overhead of the parallel shadow stack (overwriting, zeroing out) compared to the frequency of `RET` instructions (from Isen and John [27]). 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

[2]Interestingly, Mashtizadeh et al. [31] omitted (only) these two benchmarks; they reported that they could not compile them with the vanilla `GCC` or `clang` compilers.

x86, -O3	Baseline	With RET in epilogues: Traditional shadow stack	With RET in epilogues: Parallel shadow stack: Checking	With RET in epilogues: Parallel shadow stack: Overwriting, zeroing out	With RET in epilogues: Parallel shadow stack: Overwriting, no zeroing	With RET in epilogues: -fstack -protector -all	With indirect jumps instead of RET in epilogues: Traditional shadow stack	With indirect jumps instead of RET in epilogues: Parallel shadow stack: Checking	With indirect jumps instead of RET in epilogues: Parallel shadow stack: Overwriting, zeroing out	With indirect jumps instead of RET in epilogues: Parallel shadow stack: 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 9: Benchmarks for x86, -O3 with RETs (left) or indirect jumps in the epilogues. Highest overheads are in red, and lowest overheads are in green. SPEC overheads are geometric means of individual benchmark overheads.
Overheads are only *estimates*; although we can reliably reproduce these results (minima are mostly within 1% of the medians), they do not meet certain technical requirements for "reportable" SPEC CPU2006 results (e.g., modifying the source files to add the `setupShadowStack` function violates the requirements).

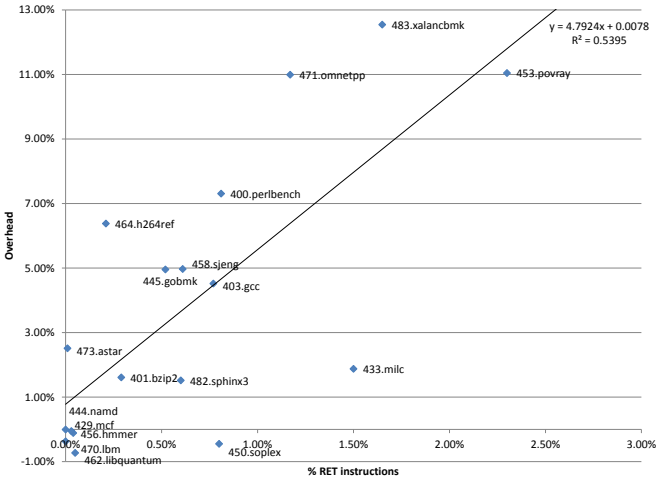

Figure 10: Correlation between the percentage of RET instructions and the overhead.

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. [10], a linear regression of the form:

Overhead $= \alpha \times$ (% RETs) $+ \beta \times$ (% RETs $\times$ % loads) $+$ $\delta \times$ (% RETs $\times$ % stores)

had a correlation of .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 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.

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 schemes [20, 54] and instrumentation schemes [40, 54, 55]. Thus, our discussion of avenues for improvement is generalizable to other shadow stack implementations and to CFI.

8.1.1 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 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.

8.1.2 Indirect Jumps vs `RET`s

With indirect jumps, the CPU can no longer predict the return address using its internal stack, but there is still dynamic branch prediction for indirect jumps. Our results show that this is noticeably imperfect, implying that the indirect jump has highly variable targets (i.e., the same function is called from multiple locations); this is somewhat similar to BinCFI [55], 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 [55]. In the context of software fault isolation (not a shadow stack), PittSFIeld [33] reported that replacing `RET`s with indirect jumps increased the overhead on SPECint2000 from 21% to 27%.

8.2 Avenues for improvement

The traditional shadow stack has close to 10% overhead – which is unlikely to be acceptable for widespread deployment [47] – 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* [32] 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 which make no calls), Crypto CFI [31] 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. [29] 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 rather than a shadow stack [18], or monitoring the taintedness of the return address in only the most recent one or two stack frames [28]. 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 [48]. Alternatively, SecondWrite's [37] return address check optimization omits the shadow stack instrumentation from functions that do not have indexed array writes. They noted that small leaf functions and recursive functions, which benefit the most from this optimization, are also the most frequently called.[3] Crypto CFI [31] 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 [31])

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 [16,25,38,52] have proposed hardware shadow-stack support. These are distinct from the return-address stack already present in modern processors for branch prediction [3, 6], 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. [18] 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 [28] proposed new registers for the Intel architecture, to store the location of the current return address, and the old value of `%ebp`.

New hardware features that are not security specific can also improve performance. For example, Crypto CFI [31] 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 [22]. 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 usually not taken, and can be predicted dynamically; the overheads of the indirect jump variants are lower-bounded by the `RET` variants, which have nearly perfect branch prediction).

[3]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.

8.3 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 security properties (as per Figure 5). Additionally, `-fstack-protector-all` was applied at the compiler level, though it is possible to add it through binary rewriting (e.g., SecondWrite [37]).

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, tail call elimination does not change the assumption that the top of the stack in the prologue is the expected return address; and the `get-EIP` idiom on x86 still works because we instrument neither the `CALL` nor `POP`. For other corner cases (e.g., exceptions, multithreading), we defer to Szekeres et al.'s [47] assessment that compatibility issues can be avoided through careful engineering.

8.4 Generalizability

Mytkowicz et al. [34] demonstrated that a narrow set of environment and compilation options etc. 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 (e.g., disabling Turbo Boost).

9. RELATED WORK

Table 1 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 [2]. 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 [37, 40, 41].

There are also many hardware assisted schemes [16, 26, 30, 38, 52]; those papers all report low overheads on SPEC2000 (or a subset thereof), when using the SimpleScalar simulator. StackGhost [23] was a proposal for a shadow stack on SPARC. We chose instead to benchmark instrumentation schemes that could be deployed on today's hardware, but hardware support may be a necessary evil.

Ozdoganoglu et al. [38] 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.

Table 1: Reported overheads of shadow stacks. Schemes are roughly sorted by the modifications involved.

Reference	Scheme	Modifications	Overhead on macro-benchmarks
Chiueh & Hsu [15]	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. [47]	Shadow stack (?)	Compiler (LLVM plugin)	5% on SPEC2006.
Mashtizadeh et al. [31]	Misc	Compiler with ABI changes	45% for leaf-optimized on SPECint 2006[a]. 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.
Vendicator [49]	Shadow stack (checking)	Assembler file processor	Unknown
Prasad & Chiueh [42]	Shadow stack (checking)	Binary rewriting with trampolines	1-3% overhead on BIND, DHCP server, PowerPoint and Outlook Express.
Baratloo et al. [8]	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. [7]	CFI + shadow stack (overwriting)	Binary rewriting with Vulcan	$\gg$ 5% on SPEC2000 for the shadow stack component; >50% for one benchmark.[b]
Gupta et al. [25]	Shadow stack (checking)	Binary rewriting with trampolines	No macro-benchmarks.
Park et al. [39]	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.[c]
Corliss et al. [16]	Shadow stacks (checking)	Binary rewriting	Average not reported, but non-trivial (overheads exceeds 40% for some SPEC2000 benchmarks)
Nebenzahl et al. [35]	Shadow stack (checking)	Binary rewriting with trampolines	4.33% on `bzip2`, 4.36% on `gzip`, and 7.09% on `mcf` from SPEC2000
Davi et al. [20]	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. [46]	Shadow stack (checking)	DynamicRIO	18.21% for SPECint 2000 on Linux; 24.82% (with compatibility issues) on Windows.
Zhang et al. [54]	General security instrumentation + shadow stack (checking)	PSI	18% overhead on a subset of SPEC2006.

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

[b] 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.

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

Corliss et al. [16] assumes that the stack pointer cannot be modified by an attacker. Such a 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 [5].

The choice of checking vs. overwriting the return address is similar to the "ensure, don't check" philosophy of SFI schemes [33, 51].

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 [43], 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 [15] Read-Only RAD accomplishes this through memory protection, albeit at a substantial overhead. Abadi et al.'s [7] 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 1). While our shadow stacks are unprotected, Szekeres et al. [47] 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).

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) [9] and XFI [21] (possibly with hardware support [12]). Importantly, due to the change in stack layout, they require significantly more code rewriting than shadow stacks. Xu et al. have separated control and data stacks (essentially a shadow stack approach, but without the return address on the main stack) [52]. 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. [29] 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. [17] and Sidiroglou et al. [45] move stack-allocated buffers to the heap. Some non-x86/x64 architectures, such as Itanium [1], have a separate register stack.

10. CONCLUSION

In this paper 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 Code Pointer Separation with their Safe Stack [29] – which unfortunately currently requires recompilation, unlike coarse-grained CFI [53, 55] – or hardware support.

11. ACKNOWLEDGEMENTS

We thank Nicholas Carlini, Mathias Payer, Úlfar Erlingsson and the anonymous reviewers for helpful comments and suggestions. This research was supported by Intel through the ISTC for Secure Computing, by the AFOSR under MURI award FA9550-12-1-0040, and by the National Science Foundation under grant CCF-0424422. We would particularly like to thank Intel for providing access to tetryl.

12. REFERENCES

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

[2] SPEC CPU2006: Read Me First. `http://www.spec.org/cpu2006/Docs/readme1st.html`, September 2011.

[3] Software Optimization Guide for AMD Family 15h Processors. January 2012.

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

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

[6] Intel(R) 64 and IA-32 Architectures Optimization Reference Manual. March 2014.

[7] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *TISSEC*, 2009.

[8] A. Baratloo, N. Singh, and T. K. Tsai. Transparent Run-Time Defense Against Stack-Smashing Attacks. In *USENIX ATC*, 2000.

[9] S. Bhatkar, D. C. DuVarney, and R. Sekar. Efficient Techniques for Comprehensive Protection from Memory Error Exploits. In *USENIX Security*, 2005.

[10] S. Bird, A. Phansalkar, L. K. John, A. Mericas, and R. Indukuru. Performance Characterization of SPEC CPU Benchmarks on Intel's Core Microarchitecture Based Processor. In *SPEC Benchmark Workshop*, 2007.

[11] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: a new class of code-reuse attack. In *CCS*, 2011.

[12] M. Budiu, Ú. Erlingsson, and M. Abadi. Architectural support for software-based protection. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, 2006.

[13] N. Carlini and D. Wagner. ROP is still dangerous: Breaking modern defenses. In *USENIX Security*, 2014.

[14] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *CCS*, 2010.

[15] T.-c. Chiueh and F.-H. Hsu. RAD: A compile-time solution to buffer overflow attacks. In *ICDCS*, 2001.

[16] M. L. Corliss, E. C. Lewis, and A. Roth. Using DISE to protect return addresses from attack. *ACM SIGARCH Computer Architecture News*, 2005.

[17] C. Dahn and S. Mancoridis. Using program transformation to secure C programs against buffer overflows. In *20th Working Conference on Reverse Engineering*, 2003.

[18] L. Davi, P. Koeberl, and A.-R. Sadeghi. Hardware-Assisted Fine-Grained Control-Flow Integrity: Towards Efficient Protection of Embedded Systems Against Software Exploitation. In *DAC*, 2014.

[19] L. Davi, D. Lehmann, A.-R. Sadeghi, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *USENIX Security*, 2014.

[20] L. Davi, A.-R. Sadeghi, and M. Winandy. ROPdefender: A detection tool to defend against return-oriented programming attacks. In *CCS*, 2011.

[21] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *OSDI*, 2006.

[22] A. Fog. The microarchitecture of Intel, AMD and VIA CPUs. `www.agner.org/optimize/microarchitecture.pdf`, August 2014.

[23] M. Frantzen and M. Shuey. StackGhost: Hardware Facilitated Stack Protection. In *USENIX Security*, 2001.

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

[25] S. Gupta, P. Pratap, H. Saran, and S. Arun-Kumar. Dynamic code instrumentation to detect and recover from return address corruption. In *International workshop on Dynamic systems analysis*, 2006.

[26] K. Inoue. Lock and Unlock: A Data Management Algorithm for A Security-Aware Cache. In *ICECS*, 2006.

[27] C. Isen and L. John. On the object orientedness of c++ programs in spec cpu 2006. In *SPEC Benchmark Workshop*, 2008.

[28] W.-F. Kao and S. F. Wu. Light-weight Hardware Return Address and Stack Frame Tracking to Prevent Function Return Address Attack. In *International Conference on Computational Science and Engineering*.

[29] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-Pointer Integrity. In *OSDI*, 2014.

[30] R. B. Lee, D. K. Karig, J. P. McGregor, and Z. Shi. Enlisting hardware architecture to thwart malicious code injection. In *Security in Pervasive Computing*. 2004.

[31] A. J. Mashtizadeh, A. Bittau, D. Mazières, and D. Boneh. Cryptographically enforced control flow integrity. In *arXiv:1408.1451*, 2014.

[32] H. Massalin. Superoptimizer: a look at the smallest program. In *ACM SIGPLAN Notices*, 1987.

[33] S. McCamant and G. Morrisett. Evaluating SFI for a CISC Architecture. In *USENIX Security*, 2006.

[34] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing wrong data without doing anything obviously wrong! In *ASPLOS*, 2009.

[35] D. Nebenzahl, M. Sagiv, and A. Wool. Install-time vaccination of Windows executables to defend against stack smashing attacks. *Dependable and Secure Computing, IEEE Transactions on*, 2006.

[36] A. One. Smashing the stack for fun and profit. *Phrack magazine*, 1996.

[37] P. O'Sullivan, K. Anand, A. Kotha, M. Smithson, R. Barua, and A. D. Keromytis. Retrofitting security in COTS software with binary rewriting. In *Future Challenges in Security and Privacy for Academia and Industry*. 2011.

[38] H. Ozdoganoglu, T. Vijaykumar, C. E. Brodley, B. A. Kuperman, and A. Jalote. SmashGuard: A hardware solution to prevent security attacks on the function return address. *Computers, IEEE Transactions on*, 2006.

[39] S.-H. Park, Y.-J. Han, S.-J. Hong, H.-C. Kim, and T.-M. 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.

[40] M. Payer and T. R. Gross. Fine-grained user-space security through virtualization. In *VEE*, 2011.

[41] M. Payer, T. Hartmann, and T. R. Gross. Safe loading-a foundation for secure execution of untrusted programs. In *IEEE S&P*, 2012.

[42] M. Prasad and T.-c. Chiueh. A Binary Rewriting Defense Against Stack based Buffer Overflow Attacks. In *USENIX ATC*, 2003.

[43] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX ATC*, 2012.

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

[45] S. Sidiroglou, G. Giovanidis, and A. D. Keromytis. A dynamic mechanism for recovering from buffer overflow attacks. In *Information security*. 2005.

[46] S. Sinnadurai, Q. Zhao, and W. 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.

[47] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal war in memory. In *IEEE S&P*, 2013.

[48] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in gcc & llvm. In *USENIX Security*, 2014.

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

[50] P. Wagle and C. Cowan. Stackguard: Simple stack smash protection for gcc. In *GCC Developers Summit*, 2003.

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

[52] J. Xu, Z. Kalbarczyk, S. Patel, and R. K. Iyer. Architecture support for defending against buffer overflow attacks. In *Workshop on Evaluating and Architecting Systems for Dependability*, 2002.

[53] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity and randomization for binary executables. In *IEEE S&P*, 2013.

[54] M. Zhang, R. Qiao, N. Hasabnis, and R. Sekar. A platform for secure static binary instrumentation. In *VEE*, 2014.

[55] M. Zhang and R. Sekar. Control Flow Integrity for COTS Binaries. In *USENIX Security*, 2013.

APPENDIX

A. PEEPHOLE OPTIMIZATIONS

Using our parallel shadow stack instrumentation as an example: Instrumented prologues will often be of the form: `POP 999996(%esp)`; `SUB $4, %esp`; `PUSH %ebp`; `MOV %esp, %ebp`; `SUB <X>, %esp` whereby the last three lines are the standard idiom for functions with frame pointers. We could replace this with `POP 999996(%esp)`; `MOV %ebp, -8(%esp)`; `LEA -8(%esp), %ebp`; `SUB <X+8>, %esp`

In the instrumented epilogue, we could replace `SUB $4, %esp`; `PUSH 999996(%esp)`; `RET` with `SUB $4, %esp`; `JMP 999996(%esp)`.

Instrumented epilogues that have frame pointers but don't use the `LEAVE` instruction will often be of the form: `MOV %ebp, %esp`; `POP %ebp`; `SUB $4, %esp`; `PUSH 999996(%esp)`; `RET` which can be converted into: `LEA 8(%ebp), %esp`; `MOV -8(%esp), %ebp`; `PUSH 999996(%esp)`; `RET`.

Alternatively, we can combine the stack pointer adjustment with the `RET` instruction i.e., `LEAVE etc.`; `SUB $4, %esp`; `PUSH 999996(%esp)`; `RET` is equivalent to: `LEAVE etc.`; `PUSH 1000000(%esp)`; `RET $4`.

The second optimization can be combined with one of the last two.

B. PROLOGUES AND EPILOGUES

- The prologues and epilogues we used are similar to, but not the same as, those described in the Introduction; for details, refer to a forthcoming technical report.
- A parallel shadow stack prologue alternative:

```
XCHGL (%esp), %ecx
MOVL %ecx, -0xb0000000(%esp)
XCHGL (%esp), %ecx
```

- A parallel shadow stack (checking) epilogue alternative:

```
POP %eax
CMPXCHG %esp, -0xa0000004(%esp)
JNZ abort
PUSH %eax
RET

abort:
    HLT
```

- For the traditional shadow stack epilogue, we can omit the comparison with the sentinel value (`CMP $0, (%ecx); JZ empty`). The latter change 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.
- For "zeroing out" (`MOV $0, ...`) the old shadow stack entry, we use `MOV %esp, ...` since it has a shorter instruction encoding. With a non-executable stack, `%esp` is an invalid return address.