Optimizing Control Data Isolation on GCC

Misiker Aga misiker@umich.edu
Sherin A Hazboun shazboun@umich.edu
Leif Millar ltmillar@umich.edui
Austin Yarger ayarger@umich.edu

Abstract

Computer security has long struggled against the ramifications of indirect control flow. Operations that read modifiable data, and branch based on said data, are indirect, and thus vulnerable to adversaries wielding buffer overflow, return-oriented-programming, and heapspray attacks. While Research has long focused to sanitize user-input, through the likes of techniques such as Address Space Layout Randomization (ASLR)[2] and Data Execution Prevention (DEP) [3], such techniques fall short in addressing the fundamental vulnerability of indirect jumps.

Control Data Isolation (CDI) was a step in a far different direction. It sought to hard code addresses and use comparison "sleds" to enforce a known and expected control flow graph. However, the performance penalties of CDI present a threat to it's wide-spread adoption. In this paper, we present a novel optimization technique that shortens the length of comparison sleds at the cost of additional memory through the use of function cloning and dynamic hot-path profiling.

1 Introduction

In this section we will discuss control flow attacks and some of the previously proposed solutions and there drawbacks.

1.1 Control Flow

At the heart of classic program exploitation lies the desire to invalidate a program's expected Control Flow. That is to say, attackers seek to alter the expected order of execution of instructions within a program. An attacker that can accomplish this could, for example, cause an otherwise innocuous program to execute system calls to delete one's files against a user's will, register a keylogger to collect user credentials, or open a root shell for the attacker to do what he pleases. Enforcing expected control-flow within a program ensures that the program only does what the programmer intended (barring a bug here or there).

Control flow is traditionally modeled via a control flow graph (CFG) that represents a program's branches as edges and "Basic Blocks" (atomic, contiguous collections of operations) as nodes.

1.2 Traditional Approaches to Control Flow Security

Buffer Overflows, Heapspray, and return-oriented-programming attacks all rely on indirect branching within a program as a means of subverting the intended control flow of a program. To combat this, several solutions have been proposed.

Address Stack Layout Randomization (ASLR): ASLR determines where in memory executable files are loaded into memory [2]. This makes it much less likely an adversary will be able to successfully determine where to jump. While ASLR does improve security, it can still be circumvented by either brute-forcing all possible 256 locations or heap spraying.

Data Execution Prevention (DEP): In order to prevent attackers from injecting executable code directly onto the stack, data execution prevention is used to prevent the system from executing code from memory locations that are writable [3]. While this prevents naive attacks, return-oriented programming (ROP) can still be used.

Stack canaries: Stack canaries are random nonces that are placed onto the stack and then checked when returning from a call to ensure that the stack has not been overwritten. This makes it much more difficult for an adversary to overwrite (also called smashing) the stack as they would need to be sure any canaries are preserved in their injected attack.

Shadow stacks: A shadow stack is where the return address is saved to an alternate location and then the return address on the stack is then checked against the saved value before returning. This prevents directly replacing the return address with a malicious location, however it does not prevent other stack corruption which can still be used to divert control flow, for example overwriting a function pointer on the stack.

1.3 Control Data Isolation

The techniques for mitigating control flow attacks mentioned previously are effective against many common exploits, however they can still be circumvented by a sophisticated adversary. In order to prevent control flow attacks entirely, control data isolation has been proposed [1]. It works by replacing indirect jumps/calls and return instructions with a check to determine if the function call or return address is valid.

1.4 Motivation: CDI Performance

Even though Control-Data Isolation gives a security guarantee for protecting control-flow attacks by enforcing the program to flow in a programmer blessed control-flow graph, it has an inherent performance overhead. Every indirect instruction is converted to a series of compare and jump instruction (called a 'sled') to all the allowable targets of the indirect instruction being considered. Previous work has shown an optimization by rearranging sled entries based on dynamic profile of programs [1]. Even though this optimization reduces the overhead by bringing the frequently taken branches to the top of the return sled, it lacks when the execution count of the sled entries is evenly distributed. In this project we show that by using a function cloning optimization on top of profile based rearranging of sleds the performance overhead can be further reduced at the cost of a slight increase in binary size.

2 Function Cloning Optimization

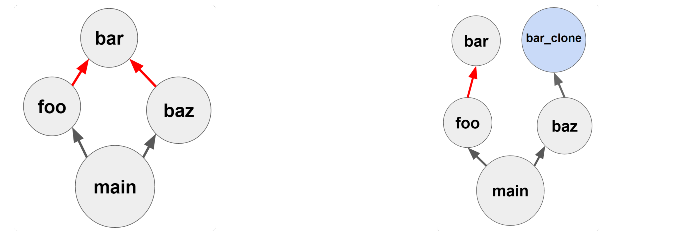
In order to enable CDI's CFG-preserving security properties, programs will necessarily need additional comparison instructions and jumps. In fact, for every callsite of a given function, said function will need an additional two lines– one comparison and one jump– to ensure that the return is to an expected address. For functions with few callsites, these additional instructions have negligible performance impact. For functions that are called from more callsites, or called many, many times within a program, the additional comparison operations can heavily degrade performance.

This can be witnessed in figure 1. The function "bar" has two call sites, one in function "foo", and one in function "baz". These callsites become return sites in the assembly code, as seen in figure 2, resulting in four additional lines and two additional comparisons.

The goal, then, is to reduce the number of comparisons in function "return sleds". With fewer comparisons to make, functions will gain in performance. This is accomplished with a technique called "function cloning". The technique proceeds as follows...

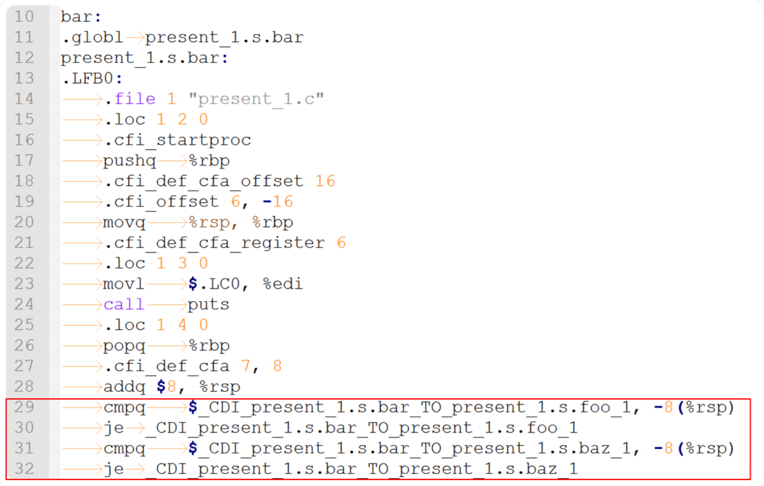
(1) create a clone of a function for every one of the function's callsites, minus one.

(2) Take the "versions" of that function (the original in addition to all of its clones) and distribute the original function's callsites among the versions as evenly as possible.



(a) Two call sites (b) One call site per clone

Figure 1: CFGs with and without function cloning



```
bar:
.globl  present_1.s.bar
present_1.s.bar:
.LFB0:
        .file 1 "present_1.c"
        .loc 1 2 0
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        .loc 1 3 0
        movl    $.LC0, %edi
        call    puts
        .loc 1 4 0
        popq    %rbp
        .cfi_def_cfa 7, 8
        addq $8, %rsp
        cmpq    $_CDI_present_1.s.bar_TO_present_1.s.foo_1, -8(%rsp)
        je  _CDI_present_1.s.bar_TO_present_1.s.foo_1
        cmpq    $_CDI_present_1.s.bar_TO_present_1.s.baz_1, -8(%rsp)
        je  _CDI_present_1.s.bar_TO_present_1.s.baz_1
```

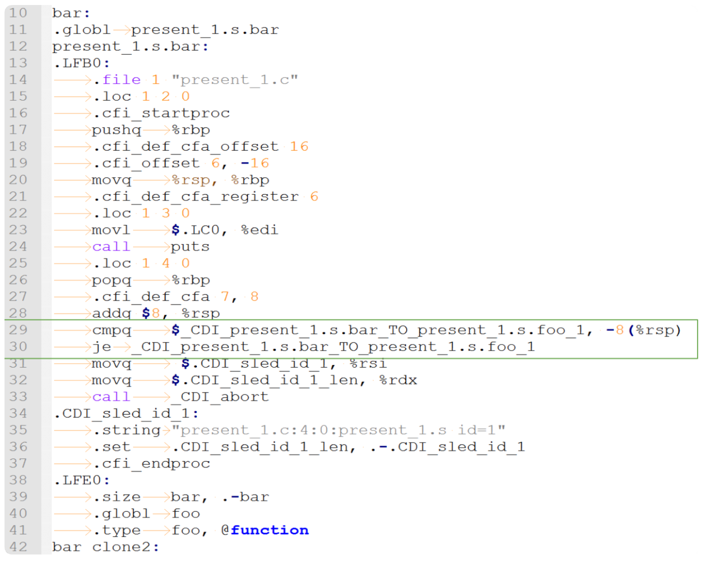
Figure 2: Return sled with two return sites

(3) Eliminate unused return sites in each version, reducing the length of return sleds and increasing performance.

This technique, performed on the "bar" function from figures 2 and 1a, results in the CFG and assembly seen in figures 1b and 3 respectively. This technique purchases better performance at the cost of larger binaries. The cost-benefit analysis is performed in section 5.3.

3 Multi-Way Branch Target Ordering Optimization

When the number of call or return sites for a function is particularly large, it can add several instructions to the critical path of compares further down the 'sled'. By first profiling the binary and then reordering the sites we allow frequent paths to jump quickly and push infrequent paths further down, effectively reducing the time to reach the taken branch.



```
bar:
.globl	present_1.s.bar
present_1.s.bar:
.LFB0:
	.file 1 "present_1.c"
	.loc 1 2 0
	.cfi_startproc
	pushq	%rbp
	.cfi_def_cfa_offset 16
	.cfi_offset 6, -16
	movq	%rsp, %rbp
	.cfi_def_cfa_register 6
	.loc 1 3 0
	movl	$.LC0, %edi
	call	puts
	.loc 1 4 0
	popq	%rbp
	.cfi_def_cfa 7, 8
	addq $8, %rsp
	cmpq	$_CDI_present_1.s.bar_TO_present_1.s.foo_1, -8(%rsp)
	je	_CDI_present_1.s.bar_TO_present_1.s.foo_1
	movq	 $.CDI_sled_id_1, %rsi
	movq	$.CDI_sled_id_1_len, %rdx
	call	_CDI_abort
.CDI_sled_id_1:
	.string	"present_1.c:4:0:present_1.s id=1"
	.set	.CDI_sled_id_1_len, .-.CDI_sled_id_1
	.cfi_endproc
.LFE0:
	.size	bar, .-bar
	.globl	foo
	.type	foo, @function
bar clone2:
```

Figure 3: Reduction to one return site

4 Implementation

4.1 Environment

Both optimizations outlined in this paper were implemented in Python 2.7, and integrated with a GCC-version of CDI developed at the University of Michigan. The tests are performed on Ubuntu 16.04, with GCC 6.1.1 as the primary target. We used SPECINT2000 and SPECINT2006 to evaluate the performance of CDI-GCC with and without optimizations.
Shared Libraries: Shared libraries are handled by a custom dynamic linker(custom-ld.so) which makes indirect jumps from the Procedure Linkage Table(PLT) sections to direct calls to the library. Return instructions on the other hand are converted to PC-relative jumps to the caller. To avoid the run time overhead of binding we enforce non-lazy binding by setting LD_BIND_NOW environment variable. The GCC-version of CDI has a verifier to check if a binary is CDI compliant.

5 Evaluation

In this section we show the performance improvements and the overheads of our optimizations on SPECINT2000/2006 and custom benchmarks.

5.1 Evaluation of Function Cloning Optimization

The function cloning technique reduced the length of function return sleds at the expense of larger binaries. Cloning code unsurprisingly increases the size of an application's code base. The important question is whether this performance improvement justifies the cost in memory.

Table 1: Performance(in seconds) and binary size(in bytes) for frequent function cloning optimized and an optimized custom benchmark program.

	Non-optimized		Optimized	
Call sites	Speed	Size	Speed	Size
1	2.77	1608	2.79	1608
5	15.32	1744	11.68	2160
10	36.55	1880	23.34	2836
50	631.54	3000	120.38	8276
100	2368.83	4392	260.5	15276

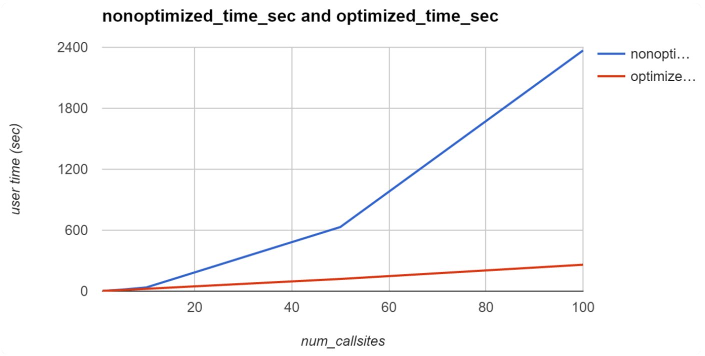


Figure 4: Function cloning performance (custom benchmark)

Performance and program text size were evaluated using Ubuntu's /usr/bin/time. We used SPECINT benchmarks[1] as well as custom benchmarks to show the effects of function call cloning. You can find the custom benchmark in the project repository. `https://github.com/Shereen92/cdi/blob/master/cdi_generator/improved_benchmark/ybench100.c`

[1]We didn't have results of function cloning experiment on SPEC benchmarks in time for the presentation.

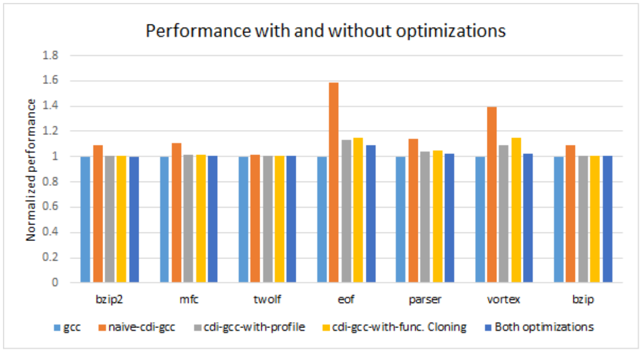


Figure 5: Optimization performance comparisons

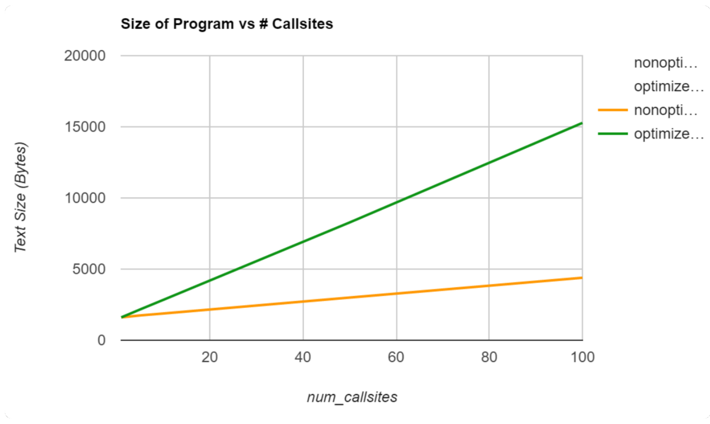


Figure 6: Function cloning memory overhead

5.2 Performance

As seen in figure 4, benchmark running times when we do clone functions (red line) are vastly shorter than when we don't clone functions (blue line). For example, in table 1 we can see that, at 100 callsites, our benchmark runs for over 2,000 seconds non-optimized, whereas with function cloning, the same test takes less than 300. The difference between non-optimized and optimized running times grows as the number of callsites for a function increases, and this has an intuitive explanation. With fewer callsites, the impact of cloning a function to reduce the length of a callsite sled is minimal (reduction from 2 comparisons to 1 comparison likely won't register unless the function is called very often during program execution. With 100 callsites however, function cloning may reduce a return sled of 100 comparison operations down to only 1, which is far more likely to have a measurable impact on performance.

5.3 Binary Size

As seen in Figure 6, as the number of callsites for a particular function increases, the size of the program's binaries text section increases in linear fashion. With function cloning enabled (green line), this rate of increase is larger. This is intuitive, as cloning a function for every new callsite increases a program's size far more than just the additional function call would.

5.4 Evaluation of Multi-Way Branch Target Ordering Optimization

We performed the sled entry reordering by using cachegrind tool of valgrind to annotate the assembly file for execution count of taken branches. We use the execution count to rearrange the sled entries during binary rewriting to order the sled in descending order of profiled execution count.

5.5 Discussion

Figure 5 shows the performance improvements with each of the optimization as well as when they are applied in tandem for SPECINT2000 and SPECINT2006 benchmarks. Multi-way branch reordering benefits the common case by bringing the frequently taken branches to the top of return sleds. It reduced the average performance overhead of the naive cdi-gcc from 21% to 5.1%. Function cloning on the other hand reduces the performance overhead by cloning frequently called functions, resulting in a performance overhead reduction of naive cdi-gcc's 21% to 6%. When used in tandem these optimizations resolve both the common case and worst case scenarios, reducing the total CDI performance overhead to a measly 2.6%.

6 Conclusion

Control-Data Isolation is a promising technology for defending against the dangers of indirect jumps, and we hope our contributions will broaden its adoption. In this work, we have implemented the CDI technique on GCC, with two main optimizations, multi-way branch target ordering and frequent function cloning. Results show that we were able to achieve substantial performance improvement while keeping the space growth at a reasonable rate. We hope for a wide-spread adoption of this technology as it is a highly capable approach to protect computing systems from a variety of control-flow attacks.

7 Future Work

For future extension of this work, we seek to further reduce the performance overhead of Control-Data Isolation with more aggressive optimizations towards its wide spread adoption. In addition to that, we're considering the use of non-linear search techniques as replacement of a a linear set of conditional branches. Such mechanisms have either constant or logarithmic time complexity, e.g. a binary search tree.

8 Contributions

The optimizations in this project were started from work by Misiker Aga and his team at the University of Michigan. Misiker and Leif worked on the multi-way branch target ordering optimization, while Sherin and Austin worked on the function cloning optimization. Everyone contributed to testing the optimizations and writing the paper.

Thanks to the staff of EECS 583– Professor Lingjia Tang and Animesh Jain– for a wonderful semester!

References

[1] William Arthur, Ben Mehne, Reetuparna Das, and Todd Austin. Getting in control of your control flow with control-data isolation. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 79–90. IEEE Computer Society, 2015.

[2] Cristiano Giuffrida, Anton Kuijsten, and Andrew S Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 475–490, 2012.

[3] Nenad Stojanovski, Marjan Gusev, Danilo Gligoroski, and Svein J Knapskog. Bypassing data execution prevention on microsoftwindows xp sp2. In *Availability, Reliability and Security, 2007. ARES 2007. The Second International Conference on*, pages 1222–1226. IEEE, 2007.