Not So Fast: Understanding and Mitigating Negative Impacts of Compiler Optimizations on Code Reuse Gadget Sets

MICHAEL D. BROWN, Georgia Institute of Technology, USA
MATTHEW PRUETT, Georgia Institute of Technology, USA
ROBERT BIGELOW, Georgia Institute of Technology, USA
GIRISH MURURU, Georgia Institute of Technology, USA
SANTOSH PANDE, Georgia Institute of Technology, USA

Despite extensive testing and correctness certification of their functional semantics, a number of compiler optimizations have been shown to violate security guarantees implemented in source code. While prior work has shed light on how such optimizations may introduce semantic security weaknesses into programs, there remains a significant knowledge gap concerning the impacts of compiler optimizations on non-semantic properties with security implications. In particular, little is currently known about how code generation and optimization decisions made by the compiler affect the availability and utility of reusable code segments (called gadgets) required to mount code reuse attack methods such as return-oriented programming.

In this paper, we bridge this gap through a study of the impacts of compiler optimization on code reuse gadget populations. We analyze and compare 1,000 different variants of 17 different benchmark programs built with two production compilers (GCC and Clang) to determine how compiler optimization affects code reuse gadget sets available in program binaries. Our results expose an important and unexpected problem; compiler optimizations introduce new gadgets at a high rate and produce code containing gadgets that are generally more useful to an attacker crafting a code reuse exploit than those in unoptimized code.

Using differential binary analysis, we identify several compiler behaviors at the root of this phenomenon. In turn, we show that these impacts can be significantly mitigated through security-focused post-production software transformation passes. Finally, we provide an analysis of the performance impacts of our proposed mitigations to demonstrate that they are negligible.

CCS Concepts: • **Software and its engineering → Source code generation**; • **Security and privacy → Software security engineering**.

Additional Key Words and Phrases: Compilers, Code generation, Code optimization, Computer security, Code reuse attacks, Code reuse gadgets

1 INTRODUCTION

The design and implementation of code optimizations in production compilers is primarily concerned with the performance and correctness of the resulting optimized code. The security impact of these design choices are difficult to determine and even more difficult to quantify, which is a natural consequence of the implementation-dependent nature of security bugs. Further, security issues with compiler optimizations are also difficult to detect during correctness certification because they may not be captured by the operational semantics model used in correctness proofs [D'Silva et al. 2015]. As a result, a significant knowledge gap exists regarding the impact of code optimization on security that has been the subject of several research papers [Belleville et al. 2018; Besson et al. 2018; Deng and Namjoshi 2017, 2018; Lim et al. 2017; Proy et al. 2017; Simon et al. 2018].

Prior work by D'Silva et al. [2015] has shown that a number of compiler optimizations, despite being formally proven sound and correct, can introduce a number of semantic security weaknesses exploitable by an attacker. The security weaknesses identified in this work fall into one of three classes: information leaks, elimination of security-relevant code due to undefined behavior, and side channel introduction. However, recent work by Brown and Pande [2019a] suggests that compiler optimizations may introduce a fourth class of security weakness: increased availability and quality of code reuse gadgets used in mounting gadget-based code reuse attacks (CRAs).

As opposed to redirecting control flow to malicious code that has been injected into memory, gadget-based CRAs instead chain together snippets of the vulnerable program's code called gadgets to implement their malicious code. The diversity and utility of code reuse gadgets available to an attacker depends primarily on the code generation and optimization decisions made by the compiler when producing the executable binary and its linked libraries. Gadget-based CRAs are particularly insidious; since they do not rely on code injection, they circumvent code injection defenses and can achieve Turing-completeness. While numerous gadget-based CRA defenses have been proposed since Shacham's [2007] seminal paper, adoption of these defenses remains low due to weaknesses against increasingly complex attack patterns and runtime overhead costs [Carlini and Wagner 2014; Kayaalp et al. 2012a; Muntean et al. 2019; van der Veen et al. 2017].

In their work, Brown and Pande demonstrate that software debloating transformations can result in negative security impacts with respect to the set of gadgets available for gadget-based CRAs. Specifically, they show that removing unneeded features from program source code introduces new code reuse gadgets into debloated binaries at a high rate, despite an overall reduction in the total number of gadgets. Further, they show that in some cases the introduced gadgets are more useful to an attacker than the gadgets that are removed, resulting in overall negative security impacts. While debloating and compiler optimization are different types of transformations, the mechanisms by which new gadgets are introduced are common to both. Interestingly, the authors cite differences in the behavior of compiler optimizations on debloated code versus bloated code as one of the causes of gadget introduction [Brown and Pande 2019a].

1.1 Motivation

These findings invite questions about the impact of similar software transformations on code reuse gadget sets found in program binaries. In this paper, we are motivated to answer the following questions regarding compiler optimizations and their effects that have been raised by prior work:

(1) To what degree do compiler optimizations introduce new gadgets in optimized code?
(2) To what degree do compiler optimizations negatively impact the security of optimized code with respect to CRA gadget sets?
(3) Which specific optimizations result in negative security impacts?
(4) What are the root causes of the observed negative security impacts?
(5) How can these negative security effects be prevented or mitigated?
(6) What is the performance cost of implementing such mitigations?

To answer our first three motivating questions, we conducted a broad study of the impacts of compiler optimizations on code reuse gadget sets. Using two production compilers, GCC and Clang, we built 1,000 total binary variants across a diverse set of 17 programs. These variants were built with different optimization configurations to support both coarse-grained analysis of predefined compiler optimization levels (i.e., O0, O1, O2, O3) and fine-grained analysis which isolates the effects of individual optimizations. To identify negative impacts, we analyzed and compared these variants using a CRA gadget-specific static analysis tool, GSA [Brown 2020]. GSA catalogs the set of code reuse gadgets available in each variant, calculates gadget set utility metrics for these sets, and provides a comparison of these values across variants.

The results of our study are concerning. Our coarse-grained analysis indicates that the set of code reuse gadgets found in optimized program variants are significantly more useful to an attacker than those found in unoptimized variants of the same program. These findings are ubiquitous; we observed significant negative impacts to gadget sets across all studied compilers, optimization levels, and benchmarks. Diving deeper, our fine-grained analysis indicates that these impacts cannot be attributed to a small group of optimizations. While a small number of optimizations were observed to have clearly discernible impacts on the resulting gadget set, we also observed that nearly all optimizations have small but measurable impacts that are predominantly negative.

To answer our last three motivating questions, we conducted differential binary analysis of our variants to identify and classify the root causes of the negative security impacts we observed in our experimental data. We identified three general root causes for the negative security impacts we observed in our study: duplication of gadget producing instructions (GPIs), transformation induced gadget introduction, and special purpose gadgets introduced as optimizations. Based on our study of these phenomena, we propose potential mitigation strategies for each.

We implemented our proposed mitigation strategy to combat the negative effects of GPI duplication by developing two transformation passes for the Egalito binary recompiler [Williams-King et al. 2020]. Our passes are designed to mitigate undesirable GPI duplication behaviors while preserving the performance benefits originally intended by the compiler. When applied to O3 variants of our benchmark programs, our passes reduced the total number of useful gadgets available in transformed binaries by an average of 30.7% and successfully eliminated one or more categories of special purpose gadgets in half of our variants. Performance analysis of our transformation passes shows that these benefits can be obtained with negligible impact to execution speed and code size. The average slowdown observed was 0.28% and the average incremental increase in static code size caused by our passes was one kilobyte.

1.2 Contributions

We summarize and organize our contributions as follows:

- In Section 4, we present the results of our broad study of the impacts of compiler optimization on code reuse gadget sets.
- In Section 5, we describe the root causes of the negative impacts we observed in our study.
- In Section 6, we detail the implementation of two binary recompiler passes for mitigating a number of the negative gadget set effects we observed. Additionally, we present the results of our experimental analysis of their effects on our benchmark programs.
- In Section 7, we provide a cost-benefit analysis of the performance impacts of our proposed mitigation passes.

2 BACKGROUND

2.0.1 Gadget-Based Code Reuse Attacks (CRAs). CRA attack methods are designed to defeat code injection defenses such as W⊕X by using existing executable parts of the vulnerable program to cause a malicious effect. The canonical CRA example is return-to-libc, in which the attacker exploits an unprotected memory buffer to redirect execution of a program to a sensitive function contained within `libc`. Return-to-libc attacks are limited in expressivity, as they do not inject arbitrary code but instead rely on the malicious execution of an existing library function. To overcome this limitation, Shacham [2007] proposed the first gadget-based CRA method, called Return-Oriented Programming (ROP). In gadget-based CRAs, the attacker exploits a program in a similar manner to return-to-libc, but rather than redirecting execution to a known function, they chain together short instruction sequences found in the compromised program called gadgets to construct a malicious payload. These attack methods have been shown to be Turing-complete if the attacker has access to a sufficiently expressive set of gadgets, allowing the attacker to construct and execute an arbitrary program. Several alternative methods to ROP have been introduced, such as jump and pure call-oriented programming (JOP and COP) [Bletsch et al. 2011b; Checkoway et al. 2010; Sadeghi et al. 2018].

2.0.2 Gadget Types. A gadget suitable for use in a CRA is a sequence of machine instructions that end in a return (ROP), indirect jump (JOP), or indirect call (JOP and COP). When chained together using the control flow properties of the last instruction in each gadget, a sequence of gadgets is equivalent to an executable program built entirely from existing code segments. Gadgets are used for one of two purposes in a gadget chain. **Functional gadgets** are used to perform some computational task such as adding two values or loading a value into a register from memory. The attacker uses functional gadgets to express their malicious exploit. Gadgets that are used to perform critical non-expressive actions such as invoking system calls (i.e., syscall gadgets) or maintaining gadget chain control flow in JOP and COP exploits are called **special purpose gadgets**.

2.0.3 Unintended Gadgets. An attacker is not limited to the instructions explicitly generated by the compiler, called **intended gadgets**, when building exploit chains. Because x86 instructions are variable length, an attacker can redirect execution to any byte offset in the program and attempt to interpret the byte sequence there as a gadget. Due to the density of the x86 encoding space, programs contain large numbers of gadgets consisting of legal instruction sequences that were never explicitly generated by the compiler, called **unintended gadgets**.

2.0.4 Transformation Induced Gadget Introduction. Brown and Pande's [2019a] prior work on debloating has shown that software transformations can cause significant changes in both the quantity and quality of the gadgets present in a transformed binary. Changes to software in its source code or intermediate representation (IR) are reflected in the downstream binary produced by the compiler, which can significantly alter the composition of intended gadgets. These changes, as well as changes in binary layout that occur as a result of transformation, also have significant impacts on the composition of unintended gadgets. Their work on debloating has determined that even conservative transformations result in high gadget introduction rates; they showed that an average of 39.5% of the gadgets present in debloated binaries were not present in the original binary.

2.0.5 Gadget Set Quality Metrics. Three qualitative security-oriented metrics for measuring transformation induced changes in gadget sets have been proposed in prior work [Brown and Pande 2019a,b; Follner et al. 2016; Homescu et al. 2012]. After cataloging gadget sets via static analysis, these metrics are then calculated via set comparisons to determine the resulting security impact.

Functional gadget set expressivity is a measure of the computational tasks that can be performed with a particular set of gadgets. This metric is calculated by classifying the gadgets in each set by computation type and then comparing the gadget set's total expressive power to that required for Turing-completeness or to construct practical ROP exploits. If a transformation results in an increase in expressivity, this is considered a negative result.

Functional gadget set quality is a measure of the the overall utility of functional gadgets for constructing exploit chains. This metric is calculated by analyzing each gadget to determine if it performs a useful computational task and if it contains extraneous instructions that will interfere with exploit construction. First, the entire set of gadgets is reduced to only those that encode a useful computational task and do not contain extraneous instructions that cannot be compensated for, called **useful gadgets**. Useful gadgets are then scored based on the number of side constraints they impose on the exploit chain (e.g., stack pointer manipulations) that must be undone by successive gadgets. If a transformation increases the number of useful gadgets in a set or decreases the average number of side constraints across the gadget set, this is considered a negative result.

Special purpose gadget availability is a measure of the types of exploit chains that can be constructed with a set of gadgets. This metric is calculated by scanning gadget sets to identify which special purpose gadgets are available, defined as the presence of at least one gadget of a particular type. If a software transformation introduces types of special purpose gadgets that were

Table 1. Study Benchmarks

Common Linux Programs		SPEC 2006			
Bftpd v5.1	gzip v1.10	401.bzip2	433.milc	456.hmmer	470.lbm
libcUrl v7.65.0	httpd v2.4.39	403.gcc	444.namd	458.sjeng	482.sphinx3
git v2.21.0	libsqlite v3.28.0	429.mcf	445.gobmk	462.libquantum	

not previously available, this is considered a negative result. If a transformation removes all special purpose gadgets of a particular type, this is considered a positive result.

3 EXPERIMENTAL SETUP

3.0.1 Benchmark Selection. We selected a total of 17 benchmark programs for this study that are diverse in size, complexity, and functionality. We selected six commonly used Linux programs and a selection of eleven programs from the SPEC 2006 benchmark set. We selected benchmark programs from each group that were compatible with our selected compilers (i.e., consist entirely of C/C++ code) and the Egalito binary recompiler. These programs are displayed in Table 1.

3.0.2 Variant Generation. We used two common production compilers, GCC v7.4.0 and Clang v8.0.1, to build 1,000 different variants of these benchmark programs for our study. To conduct a coarse-grained analysis of optimization behavior, we built four variants per benchmark per compiler at optimization levels O0 through O3. To conduct a fine-grained analysis of optimization behavior, we built 864 variants that isolate the behavior of individual optimizations. To obtain realistic variants, we configured the compiler to perform the desired optimization and all optimizations at levels below the level for which the compiler includes the desired optimization by default. For example, GCC's store merging optimization is included as part of O2 by default. To generate a variant that isolates this behavior, we configured GCC to perform all O1 level optimizations and the store merging optimization. This is necessary because compiler optimization levels generally include all of the optimizations performed at lower levels. Also, many optimizations must be run after other optimizations have already made a pass on the code to be effective. A full list of the individual optimizations studied is included in Appendix A.1.

3.0.3 Variant Analysis. To determine how each optimization level or individual optimization impacts the availability and utility of the code reuse gadgets, we analyzed variants against an appropriate baseline using the Gadget Set Analyzer tool (GSA) [Brown 2020; Brown and Pande 2019a]. For coarse-grained analysis, we compared variants produced at optimization levels O1, O2, and O3 to unoptimized variants produced at level O0 as the baseline. For fine-grained analysis, we compared each variant isolating an individual optimization to its subordinate optimization level variant. Continuing with our previous example, we compared the variant that isolates GCC's store merging optimization to the O1 variant. GSA calculates four metrics of interest in our work:

(1) **Gadget Introduction Rate:** GSA compares gadget sets to determine the percentage of gadgets found in a variant binary that were not present in the baseline binary.
(2) **Functional Gadget Set Expressivity:** GSA measures and compares the expressivity of the ROP gadgets in the baseline and variant binary relative to three expressivity levels: practical ROP exploits, ASLR-proof practical ROP exploits, and Turing-completeness [Homescu et al. 2012]. Expressivity relative to each level is recorded as the proportion of computational tasks that can be performed by the set of gadgets to the total number of computational tasks necessary to achieve full expressivity at that level (11, 35, and 17 respectively).
(3) **Functional Gadget Set Quality:** GSA first identifies which gadgets in the baseline and variant binaries are considered useful. Each useful gadget is then scored using an initial score

of 0 and incrementing the score for each side constraint imposed by extraneous instructions in the gadget. Finally, GSA measures and compares both the total count of useful gadgets and the average quality of the useful gadgets for each variant against the baseline.

(4) **Special Purpose Gadget Availability:** GSA determines and compares the availability of ten different kinds of special purpose gadgets in the baseline and variant binaries. GSA identifies system call gadgets, four different types of JOP-specific gadgets [Checkoway et al. 2010; Homescu et al. 2012], and five different types of COP-specific gadgets [Sadeghi et al. 2018].

4 STUDY RESULTS

We present our study in a top-down manner. We address our first motivating question by measuring to what degree compiler optimizations introduce new gadgets into optimized binaries in Section 4.1. Next, we address our second motivating question by exploring the coarse-grained impact of optimization levels on gadget sets in Section 4.2. Next, we address our third motivating question by exploring the fine-grained impact of individual optimizations on gadget sets in Section 4.3.

4.1 Optimization Induced Gadget Introduction

To answer our first motivating question, we analyzed the gadget sets in our benchmark variants to determine the rate at which compiler optimizations introduce new gadgets. The results of our coarse-grained analysis are contained in Table 2. Table 2 contains two sets of grouped columns, each corresponding to variants produced with a different compiler. The values in each column represent the total number of code reuse gadgets present within the binary. The percentage in the parentheses indicates the gadget introduction rate for that variant, calculated as the percentage of total gadgets present in the optimized variant that are not found in the unoptimized variant.

Our results indicate that the collective effects of compiler optimizations at defined optimization levels have a large impact on the resulting gadget set. With the exception of the smallest benchmark programs, over 80% of gadgets found in optimized binaries were newly introduced. In all cases, more than two-thirds of gadgets found in the optimized binary were newly introduced. These effects were observed in both compilers and across all optimization levels.

With respect to the total count of gadget present in sets, variants built with GCC at optimization levels O1 and O2 generally contain a similar number or fewer gadgets than their unoptimized counterparts. However, eight GCC variants built at the O3 level contained a significantly higher number of gadgets (greater than a 10% increase). This is not surprising, as optimizations that perform a code size for speedup tradeoff are typically included at the O3 level in both GCC and Clang. Such optimizations create optimized copies of code segments which are likely to introduce new gadgets. Our results were significantly different for Clang, however. Significant increases in the total number of gadgets present were observed across several optimization levels for all but one benchmark (`401.bzip2`). Interestingly, unoptimized variants produced by Clang typically had smaller gadget sets than GCC, however they were far more likely to be made larger via optimization.

Our fine-grained analysis indicates that individual optimizations also introduce gadgets at a high rate. Across single optimization variants of our common Linux benchmarks, we observed gadget introduction rates of 13%, 30.9%, and 31.6% for individual GCC optimizations included at the O1, O2, and O3 levels respectively. For single-optimization Clang variants, we observed much higher average gadget introduction rates of 72.5%, 87.5%, and 87.5% for individual optimizations included at the O1, O2, and O3 levels, respectively.

Whether applied individually or used in aggregate at defined optimization levels, our results clearly indicate that compiler optimizations have a significant impact on the quantity and composition of the code reuse gadgets found in the resulting binary. However, gadget count and introduction rate data is insufficient to draw conclusions about the impact of optimizations on security [Brown

Table 2. Total Gadgets and Introduction Rate

	GCC				Clang			
Benchmark	**O0**	**O1**	**O2**	**O3**	**O0**	**O1**	**O2**	**O3**
Bftpd	686	683 (90%)	765 (91%)	781 (91%)	565	654 (91%)	622 (91%)	599 (91%)
libcUrl	7996	7891 (97%)	7670 (97%)	8470 (97%)	6373	7410 (96%)	7743 (96%)	8169 (96%)
git	30210	22424 (96%)	22462 (96%)	22853 (96%)	18259	27697 (97%)	18332 (96%)	19062 (96%)
gzip	1099	921 (88%)	963 (88%)	1052 (89%)	588	879 (90%)	697 (87%)	820 (89%)
httpd	6130	5739 (90%)	6162 (91%)	6954 (92%)	5208	5277 (90%)	5799 (90%)	5516 (90%)
libsqlite	9583	8209 (95%)	8242 (95%)	8658 (95%)	9363	11891 (96%)	10278 (95%)	10690 (96%)
401.bzip2	625	620 (91%)	645 (92%)	778 (93%)	634	643 (91%)	610 (89%)	610 (88%)
403.gcc	34431	27064 (96%)	26114 (96%)	30462 (96%)	18844	23002 (95%)	21874 (94%)	22611 (95%)
429.mcf	294	242 (82%)	228 (79%)	244 (80%)	151	218 (78.4%)	248 (82%)	263 (83%)
433.milc	1306	1388 (93%)	1322 (93%)	1494 (93%)	926	1125 (94%)	1246 (95%)	1368 (95%)
444.namd	1372	962 (92%)	701 (88%)	754 (89%)	599	1316 (95%)	1627 (96%)	1729 (96%)
445.gobmk	8785	9205 (84%)	9041 (84%)	10087 (86%)	6435	7989 (82%)	8056 (82%)	7884 (82%)
456.hmmer	3528	3009 (95%)	2945 (95%)	3375 (95%)	2343	2597 (95%)	2741 (95%)	2833 (95%)
458.sjeng	1239	1348 (93%)	1316 (94%)	1564 (94%)	716	971 (94%)	1068 (94%)	1171 (95%)
462.libquantum	544	524 (90%)	583 (90%)	694 (91%)	352	505 (89%)	569 (91%)	659 (92%)
470.lbm	401	159 (72%)	149 (69%)	172 (73%)	106	164 (77%)	168 (77%)	168 (77%)
482.sphinx3	1945	1777 (95%)	1924 (96%)	2245 (96%)	1187	1635 (95%)	1931 (95%)	2178 (96%)

and Pande 2019a]. Due to the unpredictable nature of the mechanisms by which these gadgets are introduced and the high rates of introduction observed in our benchmarks, it is likely that the overall quality of these code reuse gadget sets are similarly impacted.

4.2 Coarse-Grained Gadget Set Impacts

Given these findings, we sought to answer our second motivating question through an analysis of coarse and fine-grained impacts of optimizations gadget set quality. The results of our coarse-grained optimization analysis with respect to the three selected gadget set quality metrics follows.

4.2.1 Functional Gadget Set Expressivity. Table 3 contains the data we collected on the expressive power of the functional gadgets within each set. The first column of the table contains the expressivity level of the baseline variant, which is unoptimized code produced at level O0 for both compilers. Functional gadget set expressivity data is expressed as a 3-tuple, in which each integer indicates the number of satisfied computational classes with respect to one of the three selected expressivity levels, displayed in the following order: practical ROP exploits, ASLR-proof practical ROP exploits, and Turing-completeness. Following the first column, there are three pairs of columns containing expressivity data collected for optimized variants produced at levels O1, O2, and O3. In columns marked with the Δ symbol, the 3-tuple in parentheses indicates the difference between the unoptimized and the optimized variants. Negative values in these columns indicate a negative effect: an increase in the expressive power of the gadget set.

Our data indicates that compiler optimizations overwhelmingly resulted in negative impacts to security with respect to the expressive power of functional gadget sets. We observed an increase in expressive power for at least one expressivity level in 96% (98 of 102) of total variants, which includes *all* Clang variants. Additionally, we observed that these increases are typically significant in magnitude and affect all levels of expressivity. In approximately half of the optimized variants, the number of computational classes satisfied doubled for at least one expressivity level. In 60.8% of variants (62 of 102), optimization resulted in an increase in expressivity across all three levels.

Table 3. Coarse-Grained Functional Gadget Set Expressivity

Benchmark		O0	O1	ΔO1	O2	ΔO2	O3	ΔO3
GCC Variants	Bftpd	6/9/3	6/13/6	(0/-4/-3)	6/12/5	(0/-3/-2)	6/13/6	(0/-4/-3)
	libcUrl	7/15/4	9/28/13	(-2/-13/-9)	10/25/13	(-3/-10/-9)	9/28/14	(-2/-13/-10)
	git	11/35/16	10/33/16	(1/2/0)	11/33/15	(1/2/1)	11/34/16	(0/1/0)
	gzip	6/10/5	7/19/8	(-1/-9/-3)	6/22/11	(0/-12/-6)	8/23/10	(-2/-13/-5)
	httpd	9/18/6	8/25/12	(1/-7/-6)	8/24/12	(1/-6/-6)	8/22/11	(1/-4/-5)
	libsqlite	7/23/7	9/28/13	(-2/-5/-6)	9/29/14	(-2/-6/-7)	10/30/13	(-3/-7/-6)
	401.bzip2	6/8/2	4/11/6	(2/-3/-4)	6/10/5	(0/-2/-3)	5/17/7	(1/-9/-5)
	403.gcc	11/31/13	11/35/17	(0/-4/-4)	11/33/15	(0/-2/-2)	11/35/17	(0/-4/-4)
	429.mcf	4/7/2	5/9/4	(-1/-2/-2)	5/9/4	(-1/-2/-2)	5/9/4	(-1/-2/-2)
	433.milc	7/16/5	6/23/8	(1/-7/-3)	7/26/9	(0/-10/-4)	7/23/8	(0/-7/-3)
	444.namd	6/18/6	7/18/4	(-1/0/2)	7/19/5	(-1/-1/1)	8/23/8	(-2/-5/-2)
	445.gobmk	7/22/7	8/27/12	(-1/-5/-5)	9/26/13	(-2/-4/-6)	6/30/13	(1/-8/-6)
	456.hmmer	7/18/4	9/26/10	(-2/-8/-6)	7/26/9	(0/-8/-5)	7/30/11	(0/-12/-7)
	458.sjeng	6/8/7	8/17/8	(-2/-9/-1)	8/17/10	(-2/-9/-3)	8/17/9	(-2/-9/-2)
	462.libquantum	6/10/4	5/16/6	(1/-6/-2)	5/17/5	(1/-7/-1)	6/19/7	(0/-9/-3)
	470.lbm	4/14/2	4/17/3	(0/-3/-1)	4/15/3	(0/1/1)	4/15/3	(0/-1/-1)
	482.sphinx3	4/9/3	6/21/7	(-2/-12/-4)	9/23/10	(-5/-14/-7)	9/27/12	(-5/-18/-9)
Clang Variants	Bftpd	5/12/4	6/21/7	(-1/-9/-3)	6/20/5	(-1/-8/-1)	6/23/6	(-1/-11/-2)
	libcUrl	4/16/7	8/28/13	(-4/-12/-6)	10/29/13	(-6/-13/-6)	8/30/13	(-4/-14/-6)
	git	10/34/13	11/33/15	(-1/1/-2)	11/34/16	(-1/0/-3)	11/34/16	(-1/0/-3)
	gzip	3/6/2	6/22/9	(-3/-16/-7)	5/18/5	(-2/-12/-3)	5/20/6	(-2/-14/-4)
	httpd	5/14/4	9/25/11	(-4/-11/-7)	9/30/12	(-4/-16/-8)	9/25/9	(-4/-11/-5)
	libsqlite	6/17/8	11/32/15	(-5/-15/-7)	9/29/13	(-3/-12/-5)	9/28/13	(-3/-11/-5)
	401.bzip2	5/6/2	7/18/4	(-2/-12/-2)	6/20/5	(-1/-14/-3)	6/17/4	(-1/-11/-2)
	403.gcc	11/31/13	11/33/16	(0/-2/-3)	10/33/16	(1/-2/-3)	11/35/17	(0/-4/-4)
	429.mcf	3/5/2	6/10/6	(-3/-5/-4)	6/10/6	(-3/-5/-4)	6/10/6	(-3/-5/-4)
	433.milc	5/7/4	6/21/7	(-1/-14/-3)	7/22/8	(-2/-15/-4)	7/23/8	(-2/-16/-4)
	444.namd	3/9/2	7/22/10	(-4/-13/-8)	7/25/9	(-4/-16/-7)	8/24/9	(-5/-15/-7)
	445.gobmk	5/22/9	10/31/15	(-5/-9/-6)	10/31/14	(-5/-9/-5)	9/31/13	(-4/-9/-4)
	456.hmmer	7/12/6	8/25/9	(-1/-13/-3)	7/29/12	(0/-17/-6)	8/29/12	(-1/-17/-6)
	458.sjeng	5/6/1	8/24/10	(-3/-18/-9)	7/24/9	(-2/-18/-8)	7/23/9	(-2/-17/-8)
	462.libquantum	3/13/4	6/20/7	(-3/-7/-3)	6/22/6	(-3/-9/-2)	5/21/7	(-2/-8/-3)
	470.lbm	3/5/1	3/17/4	(0/-12/-3)	3/16/4	(0/-11/-3)	3/13/3	(0/-8/-2)
	482.sphinx3	5/11/5	6/23/11	(-1/-12/-6)	9/29/12	(-4/-18/-7)	8/26/11	(-3/-15/-6)

4.2.2 Functional Gadget Set Quality. Table 4 contains the data we collected on the number of useful functional gadgets within each set and their average quality. The first column of the table contains the expressivity level of the baseline variant, which is unoptimized code produced at level O0 for both compilers. Functional gadget set quality data is expressed as a 2-tuple, in which the first integer indicates the number of useful functional gadgets found within the set and the second value indicates the average quality score of the gadgets in the set. Following the first column, there are three pairs of columns containing quality data collected for optimized variants produced at levels O1, O2, and O3. In columns marked with the Δ symbol, the 2-tuple in parentheses indicates the difference between the unoptimized and the optimized variants. Negative values for the first item in the 2-tuple indicate a negative effect: an increase in the number of the useful gadgets in the set. Higher average gadget set quality scores indicate gadget sets with a higher number of side constraints; as such, positive values for the second item in the 2-tuple indicate a negative effect: a decrease in the average number of side constraints present in the gadgets that make up the set.

Our data indicates compiler optimizations have impacts on functional gadget set quality data that are similar in frequency and severity to functional gadget set expressivity. In 73.5% (75 of 102)

Table 4. Coarse-Grained Functional Gadget Set Quality

Benchmark		O0	O1	ΔO1	O2	ΔO2	O3	ΔO3
GCC Variants	Bftpd	268 / 1.04	302 / 0.86	(-34 / 0.17)	333 / 0.94	(-65 / .09)	347 / 0.91	(-79 / .012)
	libcUrl	3987 / 0.97	3544 / 0.86	(443 / 0.11)	3959 / 0.73	(28 / 0.23)	4284 / 0.7	(-297 / 0.26)
	git	12555 / 1.1	8058 / 0.91	(4497 / 0.19)	8018 / 0.93	(4537 / 0.18)	7401 / 0.88	(5154 / 0.23)
	gzip	431 / 1.07	358 / 0.85	(73 / 0.22)	416 / 0.99	(15 / 0.08)	401 / 0.89	(30 / 0.18)
	httpd	2725 / 1.03	2512 / 0.81	(213 / 0.22)	2899 / 0.86	(-174 / 0.17)	3278 / 0.81	(-553 / 0.22)
	libsqlite	268 / 1.04	302 / 0.86	(-34 / 0.17)	333 / 0.94	(-65 / 0.09)	347 / 0.91	(-79 / 0.12)
	401.bzip2	238 / 0.98	211 / 0.83	(27 / 0.16)	242 / 0.76	(-4 / 0.22)	301 / 0.73	(-63 / 0.26)
	403.gcc	11165 / 1.04	8799 / 0.85	(2366 / 0.19)	9389 / 0.86	(1776 / 0.17)	10601 / 0.85	(564 / 0.18)
	429.mcf	90 / 0.93	112 / 0.84	(-22 / 0.08)	107 / 0.79	(-17 / 0.13)	104 / 0.73	(-14 / 0.2)
	433.milc	512 / 1	491 / 0.98	(21 / 0.02)	507 / 0.89	(5 / 0.11)	616 / 0.8	(-104 / 0.19)
	444.namd	358 / 0.98	274 / 0.69	(84 / 0.3)	264 / 0.7	(94 / 0.28)	320 / 0.7	(38 / 0.28)
	445.gobmk	3799 / 1	4154 / 0.88	(-355 / 0.12)	4024 / 0.87	(-225 / 0.17)	4533 / 0.94	(-734 / 0.07)
	456.hmmer	1087 / 1.12	1224 / 0.89	(-137 / 0.23)	1215 / 0.86	(-128 / 0.26)	1338 / 0.82	(-251 / 0.3)
	458.sjeng	426 / 0.87	476 / 0.68	(-50 / 0.19)	489 / 0.7	(-63 / 0.17)	623 / 0.67	(-197 / 0.2)
	462.libquantum	225 / 1.11	222 / 0.82	(3 / 0.3)	233 / 0.7	(-8 / 0.4)	267 / 0.7	(-42 / 0.41)
	470.lbm	75 / 1.05	93 / 0.84	(-18 / 0.21)	83 / 0.77	(-8 / 0.28)	86 / 0.81	(-11 / 0.23)
	482.sphinx3	730 / 1.17	679 / 0.93	(51 / 0.24)	748 / 0.85	(-18 / 0.32)	867 / 0.77	(-137 / 0.4)
Clang Variants	Bftpd	295 / 1.12	339 / 0.92	(-44 / 0.2)	325 / 1	(-30 / 0.12)	337 / 0.9	(-42 / 0.22)
	libcUrl	3324 / 1	3906 / 0.7	(-582 / 0.3)	4045 / 0.68	(-721 / 0.32)	4258 / 0.68	(-934 / 0.32)
	git	8235 / 1.05	12155 / 0.78	(-3920 / 0.28)	6956 / 0.83	(1279 / 0.23)	6974 / 0.79	(1261 / 0.27)
	gzip	278 / 1.04	390 / 0.86	(-112 / 0.18)	248 / 0.81	(30 / 0.23)	292 / 0.73	(-14 / 0.31)
	httpd	2575 / 0.93	2689 / 0.7	(-114 / 0.24)	2764 / 0.72	(-189 / 0.21)	2619 / 0.7	(-44 / 0.24)
	libsqlite	3508 / 1	4540 / 0.74	(-1032 / 0.26)	3007 / 0.68	(501 / 0.32)	3322 / 0.7	(186 / 0.3)
	401.bzip2	220 / 0.85	268 / 0.77	(-48 / 0.08)	247 / 0.77	(-27 / 0.08)	208 / 0.75	(12 / 0.1)
	403.gcc	8140 / 0.94	9142 / 0.82	(-1002 / 0.11)	8328 / 0.79	(-188 / 0.15)	8635 / 0.81	(-495 / 0.12)
	429.mcf	114 / 0.98	105 / 0.94	(9 / 0.04)	120 / 0.94	(-6 / 0.04)	123 / 0.95	(-9 / 0.03)
	433.milc	380 / 1.05	455 / 0.82	(-75 / 0.23)	489 / 0.82	(-109 / 0.23)	522 / 0.8	(-142 / 0.25)
	444.namd	280 / 0.91	475 / 0.73	(-195 / 0.18)	484 / 0.66	(-204 / 0.25)	525 / 0.64	(-245 / 0.26)
	445.gobmk	3165 / 0.96	3963 / 0.8	(-771 / 0.16)	3861 / 0.84	(-696 / 0.12)	3767 / 0.81	(-602 / 0.15)
	456.hmmer	929 / 1.11	1024 / 0.86	(-95 / 0.25)	1044 / 0.87	(-115 / 0.24)	1037 / 0.84	(-108 / 0.27)
	458.sjeng	347 / 0.89	410 / 0.84	(-63 / 0.05)	445 / 0.75	(-98 / 0.14)	463 / 0.79	(-116 / 0.1)
	462.libquantum	173 / 1.01	265 / 0.8	(-92 / 0.21)	261 / 0.8	(-88 / 0.2)	267 / 0.82	(-94 / 0.19)
	470.lbm	64 / 1.04	89 / 0.86	(-25 / 0.18)	89 / 0.88	(-25 / 0.16)	86 / 1.04	(-22 / 0)
	482.sphinx3	608 / 1.12	663 / 0.95	(-55 / 0.17)	715 / 0.79	(-107 / 0.33)	772 / 0.82	(-164 / 0.3)

of variants, optimization increased the number of useful ROP, JOP, and COP gadgets. In 88.2% (90 of 102) of variants, optimizations caused the average quality score of the gadget set to improve from the attacker's perspective by a significant degree (defined as a reduction by at least 0.1). Further, we observed no instances where optimization worsened the average quality score of the gadget set from the attacker's perspective by a significant degree.

4.2.3 Special Purpose Gadget Availability. Table 5 contains the data we collected on the categories of special purpose gadgets available within each set. The first column of this table contains the number of categories available in the baseline variant, which is unoptimized code produced at level O0 for both compilers. Availability data is expressed as a single integer indicating the number of categories of special purpose gadgets with at least one gadget present within the gadget set. Following the first column, there are three pairs of columns containing expressivity data collected for optimized variants produced at levels O1, O2, and O3. In columns marked with the Δ symbol, the integer in parentheses indicates the difference between the unoptimized and the optimized

Table 5. Coarse-Grained Special Purpose Gadget Availability

	GCC							Clang						
Benchmark	**O0**	**O1**	**ΔO1**	**O2**	**ΔO2**	**O3**	**ΔO3**	**O0**	**O1**	**ΔO1**	**O2**	**Δ O2**	**O3**	**Δ O3**
Bftpd	1	1	(0)	1	(0)	2	(-1)	1	1	(0)	1	(0)	2	(-1)
libcUrl	4	4	(0)	4	(0)	4	(0)	4	4	(0)	4	(0)	4	(0)
git	6	6	(0)	6	(0)	5	(1)	6	6	(0)	6	(0)	5	(1)
gzip	1	1	(0)	2	(-1)	3	(-2)	1	1	(0)	2	(-1)	3	(-2)
httpd	6	3	(3)	5	(1)	5	(1)	6	3	(3)	5	(1)	5	(1)
libsqlite	5	4	(1)	4	(1)	4	(1)	5	4	(1)	4	(1)	4	(1)
401.bzip2	1	1	(0)	1	(0)	1	(0)	1	1	(0)	1	(0)	1	(0)
403.gcc	6	6	(0)	6	(0)	6	(0)	6	6	(0)	6	(0)	6	(0)
429.mcf	1	1	(0)	1	(0)	2	(0)	1	1	(0)	1	(0)	2	(0)
433.milc	2	2	(0)	1	(1)	1	(1)	2	2	(0)	1	(1)	1	(1)
444.namd	2	2	(0)	1	(1)	2	(0)	2	2	(0)	1	(1)	2	(0)
445.gobmk	4	5	(-1)	5	(-1)	6	(-2)	4	5	(-1)	5	(-1)	6	(-2)
456.hmmer	3	2	(1)	3	(0)	2	(1)	3	2	(1)	3	(0)	2	(1)
458.sjeng	2	1	(1)	2	(0)	2	(0)	2	1	(1)	2	(0)	2	(0)
462.libquantum	1	1	(0)	1	(0)	1	(0)	1	1	(0)	1	(0)	1	(0)
470.lbm	1	1	(0)	1	(0)	1	(0)	1	1	(0)	1	(0)	1	(0)
482.sphinx3	1	1	(0)	1	(0)	2	(-1)	1	1	(0)	1	(0)	2	(-1)

variants. Negative values in these columns indicate a negative effect: an increase in the number of special purpose gadget types present in the gadget set.

We observed overall negative impacts on special purpose gadget availability in 13.7% (14 of 102) of total variants and overall positive effects in 25.5% (26 of 102) of total variants. In the majority of variants we did not observe overall changes, due in part to the relatively small population of special purpose gadgets in our benchmarks and the calculation methodology for this metric. An overall result of no change is possible in situations where the number of gadget types eliminated is offset by introduction of an equal number of other types. A detailed analysis of the specific categories gadgets present in our variants reveals that optimizations introduced at least one new type of special purpose gadget that was not originally generated by the compiler in approximately half of variants. The most frequently introduced special purpose gadgets were syscall gadgets, which are particularly dangerous and versatile special purpose gadgets. These gadgets are useful in ROP, JOP, and COP attacks for performing sensitive actions, such as executing programs like a system shell.

4.2.4 Discussion. Our coarse-grained analysis results indicate that the sets of gadgets available in optimized program variants are significantly more useful for constructing CRA exploits than the sets found in unoptimized variants. Interestingly, our data indicates that security impacts do not follow a linear progression and are not significantly impacted by the input program. Negative gadget set impacts observed in our study did not increase with the complexity or cost of optimizations and significant impacts were observed for all optimization levels, across all benchmarks and for both compilers (more so for Clang). We conclude from this observation that **avoiding negative security impacts on CRA gadget sets is not as simple as selecting a particular optimization level**.

4.3 Fine-Grained Gadget Set Impacts

In order to answer our third motivating question, we conducted a fine-grained analysis of gadget set impacts by generating 864 different single-optimization variants of our benchmarks. Although single optimization variants were built for all benchmarks, fine-grained analysis variants were predominantly built from the common Linux program benchmark set. Tables of the individual optimizations used to generate these variants can be found in A.1. Using the coarse-grained variants produced at levels O0, O1, and O2 as baselines where appropriate, we analyzed our single optimization variants using GSA to isolate their effects on the resulting gadget sets.

4.3.1 General Observations. As was observed at coarse-grained optimization levels, our fine-grained variants exhibited a variety of negative impacts on resulting gadget sets according to our metrics. Negative impacts in the form of increases in gadget set expressivity, gadget set quality, and special purpose gadget availability were commonly observed across all benchmarks, isolated optimizations, and compilers; though these impacts were observed with more variability in our single optimization variants than our optimization level variants.

In the majority of single optimization variants, negative effects were relatively small in magnitude, manifesting as gadget set expressivity increases of less than three computational classes, an increase of less than 5% in the number of useful gadgets, or the introduction of one type of special purpose gadget (usually syscall, JOP data loader, or COP intra-stack pivot gadgets). Similar small magnitude positive impacts on gadget set metrics were also observed across our single optimization variants, albeit at a lower incidence rate than negative impacts. Brown and Pande's [2019a] work suggests that transformation induced gadget introduction is responsible for the majority of the small magnitude gadget set impacts we observed in our single-optimization variants. We detail our investigation of this potential root cause of the "background noise" we observed in Section 5.3.

4.3.2 Outlier Detection. Within this background noise, we observed a number of individual optimizations with negative impacts that were significantly larger in magnitude. To separate these instances for deeper analysis, we performed outlier detection across our single optimization variant data. We define outliers as single optimization variants with gadget set metric result changes from the baseline variant greater than 1.5 times the standard deviation from the mean gadget set metric result change on a per benchmark, per metric basis. This definition is shown in Equation 1, where A is set to 1.5, the mean variant result change is denoted by μ, the standard deviation of variant result changes is denoted by σ, and individual optimization result changes are denoted by Δ_{res}.

$$(\mu - A\sigma) \nleq \Delta_{res} \nleq (\mu + A\sigma) \tag{1}$$

For each combination of benchmark and gadget set metric, we analyzed the GSA results to identify individual optimizations with results considered outliers. We then combined the total list of identified outliers across all benchmarks and metrics and constructed a histogram to identify which isolated optimizations resulted in outlier results most frequently, and for which specific metrics they produce outlier results. Our outlier detection process identified the following isolated optimizations as frequently producing outlier gadget set impacts:

(1) **Omit Frame Pointer** (GCC `-fomit-frame-pointer`): Produces frequent outlier results across all three metrics.
(2) **Interprocedural Constant Propagation Function Cloning** (GCC `-fipa-cp-clone`): Produces frequent outlier results with respect to functional gadget set expressivity.
(3) **Jump Following Common Subexpression Elimination** (GCC `-fcse-follow-jumps`): Produces frequent outlier results with respect to special purpose gadget introduction.
(4) **Tail Call Elimination** (Clang `--tail-call-elim`): Produces frequent outlier results with respect to special purpose gadget introduction.
(5) **Peephole Optimizations** (GCC `-fpeephole2`): Produces frequent outlier results with respect to special purpose gadget introduction.

4.3.3 Outlier Analysis. Tables 6 and 7 contain the subset of our fine-grained analysis relevant to the optimizations identified as frequently producing outliers. Table 6 contains analyzer data across all three metrics for GCC's omit frame pointer optimization. This optimization causes marked

Table 6. GCC Omit-Frame-Pointer Single Optimization Variant Gadget Set Impacts

	Functional Gadget Set Expressivity			Functional Gadget Set Quality			S.P. Gadget Availability		
Benchmark	**O0**	**Opt**	**ΔOpt**	**O0**	**Opt**	**ΔOpt**	**O0**	**Opt**	**ΔOpt**
Bftpd	6/9/3	8/16/5	(-2/-7/-2)	268 / 1.04	287 / 0.74	(-19 / 0.3)	1	1	(0)
libcUrl	7/15/4	9/22/7	(-2/-7/-3)	3987 / 0.97	3595 / 0.86	(392 / 0.1)	4	6	(-2)
git	11/35/16	11/33/16	(0/2/0)	12555 / 1.1	8351 / 1.06	(4204 / 0.04)	5	5	(0)
gzip	6/10/5	7/17/5	(-1/-7/0)	431 / 1.07	365 / 0.92	(66 / 0.15)	1	1	(0)
httpd	9/18/6	7/21/8	(2/-3/-2)	2725 / 1.03	2288 / 0.87	(437 / 0.16)	6	3	(3)
libsqlite	7/23/7	8/23/10	(-1/0/-3)	4200 / 1.1	3724 / 0.93	(476 / 0.17)	5	2	(3)
401.bzip2	6/8/2	5/14/4	(1/-6/-2)	238 / 0.98	244 / 0.72	(-6 / 0.27)	1	1	(0)
403.gcc	11/31/13	11/31/14	(0/0/-1)	11070 / 1.04	9639 / 0.88	(1431 / 0.16)	5	5	(0)
429.mcf	4/7/2	5/13/3	(-1/-6/-1)	90 / 0.93	85 / 0.69	(5 / 0.23)	1	1	(0)
433.milc	7/16/5	7/19/7	(0/-3/-2)	512 / 1	473 / 0.74	(39 / 0.26)	2	2	(0)
444.namd	6/18/6	7/20/8	(-1/-2/-2)	358 / 0.98	319 / 0.72	(39 / 0.26)	2	1	(1)
445.gobmk	7/22/7	9/27/11	(-2/-5/-4)	3797 / 1	3529 / 0.82	(268 / 0.18)	4	5	(-1)
456.hmmer	7/18/4	7/21/9	(0/-3/-5)	1087 / 1.12	1122 / 0.88	(-35 / 0.23)	3	1	(2)
458.sjeng	6/8/7	7/21/9	(-1/-13/-2)	426 / 0.87	461 / 0.57	(-35 / 0.31)	2	3	(-1)
462.libquantum	6/10/4	5/14/4	(1/-4/0)	225 / 1.11	199 / 0.72	(26 / 0.39)	1	1	(0)
470.lbm	4/14/2	3/12/2	(1/2/0)	75 / 1.05	77 / 0.79	(-2 / 0.26)	1	1	(0)
482.sphinx3	4/9/3	8/17/6	(-4/-8/-3)	730 / 1.17	648 / 0.93	(82 / 0.24)	1	1	(0)

increases in functional gadget set expressivity across 82.4% (14 of 17) of variants. Interestingly, this optimization did not have similar effects on functional gadget set quality. We observed that this optimization significantly reduced the number of useful gadgets present in the resulting variant in 64.7% (11 of 17) of our benchmarks. With respect to special purpose gadget availability, this optimization introduced new categories of special purpose gadgets in three benchmarks, but was also observed to eliminate existing categories of gadgets in four benchmarks.

Table 7 contains analyzer data for optimizations with a single metric observed to exhibit a high frequency of outliers. GCC's interprocedural constant propagation function cloning optimization was observed to have mixed impacts on functional gadget set expressivity with expressivity increases and decreases occurring at roughly the same rate across all benchmarks. GCC's jump following CSE and peephole optimizations were observed to frequently introduce new types of special purpose gadgets. The most frequently observed gadgets introduced were syscall and COP intra-stack pivot gadgets. In contrast to these observations, Clang's tail call elimination optimization had a number of positive impacts on special purpose gadget availability, eliminating one or more types of special purpose gadgets in 35.3% (6 of 17) of variants. However, this optimization was also observed to introduce new types of special purpose gadgets in 23.5% (4 of 17) of variants. Deeper analysis of this unexpected result reveals this optimization frequently eliminates COP-specific special purpose gadgets, but was also observed to introduce new JOP dataloader gadgets at a high rate. The ultimate effect of the optimization (positive or negative) largely depends on the types of special purpose gadgets present in the baseline variant; for example, baseline variants without JOP dataloader or COP-specific gadget types are likely to suffer the negative impact of JOP dataloader gadgets introduction without a corresponding benefit of COP-specific gadget elimination.

4.3.4 Discussion. We draw two high-level conclusions from our fine-grained analysis of compiler optimization behaviors. First, our data indicates that nearly all optimizations have a small but measurable impact on the composition of the resulting gadget set. The majority of these impacts were observed to be negative, though positive impacts were also observed at lower frequency.

Table 7. Single Optimization Variants with Outliers in One Metric

	IPA CP Clone - GCC [1]			CSE Follow Jumps - GCC			Tail Call Elimination - Clang			Peephole - GCC		
Benchmark	**O2**	**Opt**	**ΔOpt**	**O1**	**Opt**	**ΔOpt**	**O0**	**Opt**	**ΔOpt**	**O0**	**Opt**	**ΔOpt**
Bftpd	6/12/5	6/13/6	(0/-1/-1)	1	1	(0)	1	1	(0)	1	1	(0)
libcUrl	10/25/13	9/24/12	(1/1/1)	4	5	(-1)	4	5	(-1)	4	5	(-1)
git	11/33/15	11/31/14	(0/2/1)	6	6	(0)	6	3	(3)	6	6	(0)
gzip	6/22/11	6/24/12	(0/-2/-1)	1	2	(-1)	1	1	(0)	1	1	(0)
httpd	8/24/12	9/27/13	(-1/-3/-1)	3	4	(-1)	5	4	(1)	3	4	(-1)
libsqlite	9/29/14	9/27/13	(0/2/1)	4	6	(-2)	5	3	(2)	4	6	(-2)
401.bzip2	6/10/5	5/9/4	(1/1/1)	1	1	(0)	1	1	(0)	1	1	(0)
403.gcc	11/33/15	11/34/16	(0/-1/-1)	6	6	(0)	5	6	(-1)	6	6	(0)
429.mcf	5/9/4	5/9/4	(0/0/0)	1	1	(0)	1	1	(0)	1	1	(0)
433.milc	7/26/9	7/24/9	(0/2/0)	2	2	(0)	2	1	(1)	2	2	(0)
444.namd	7/19/5	7/19/5	(0/0/0)	2	2	(0)	1	1	(0)	2	2	(0)
445.gobmk	9/26/13	7/24/12	(2/2/1)	5	5	(0)	5	3	(2)	5	6	(-1)
456.hmmer	7/26/9	7/26/9	(0/0/0)	2	2	(0)	3	4	(-1)	2	4	(-2)
458.sjeng	8/17/10	8/17/10	(0/0/0)	1	2	(-1)	1	2	(-1)	1	1	(0)
462.libquantum	5/17/5	5/17/5	(0/0/0)	1	1	(0)	1	1	(0)	1	1	(0)
470.lbm	4/15/3	4/15/3	(0/0/0)	1	1	(0)	1	1	(0)	1	1	(0)
482.sphinx3	9/23/10	9/22/9	(0/1/1)	1	1	(0)	2	1	(1)	1	1	(0)

[1] Functional gadget set expressivity data is displayed for IPA CP Clone (GCC). For all other single optimization variants, special purpose gadget availability data is displayed.

This conclusion is consistent with the findings of our coarse-grained analysis; it suggests that the high levels of negative gadget set impacts observed at predefined optimization levels are the conglomeration of smaller magnitude impacts made by individual optimizations rather than a small group of optimizations with high magnitude impacts. Second, we conclude that a relatively small number of isolated optimizations have gadget set impacts that are clearly discernible from the background noise. This conclusion is promising as it suggests that there are relatively few undesirable behaviors in existing optimizations that are negatively impacting gadget sets.

5 ROOT CAUSES

To answer our fourth motivating question, we performed differential binary analysis of our single optimization variants and their respective baseline variants determine the underlying causes for the negative impacts we observed. We did not exhaustively analyze all 864 of our single optimization variants as this would be intractable; we focused our efforts on optimizations that were observed to have frequent outlier gadget set impacts including but not limited to those identified in Section 4.3. In addition to searching for root causes of these outliers, we also sought to confirm our hypothesis that transformation induced gadget introduction is the underlying cause for the small magnitude changes in gadget set metrics we observed across our single optimization variants.

We used a number of automated tools in support of our manual differential binary analysis efforts. To identify and observe the after-effects of localized transformations performed by individual optimizations, we used the commercial disassembler IDA Pro [Hex-Rays 2020] and one of its plugins, BinDiff [zynamics 2020]. BinDiff uses a number of heuristics to match functions in different versions of the same program, enabling a visual before and after side-by-side comparison of the program's control flow graphs (CFGs). To ensure that the disassembly process produced precise CFG recovery, we built all of our program variants with full debugging metadata.

In the course of our analysis, we identified three mechanisms that result in negative impacts to gadget sets. The nature of these mechanisms varies widely, and the potential mitigation strategies to combat them vary widely as well. In the following subsections, we detail these mechanisms, the optimizations we observe that employ them, and potential mitigation strategies.

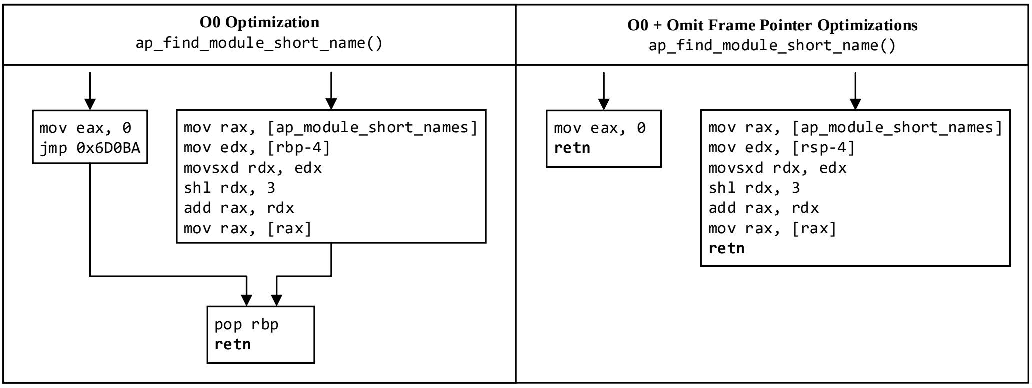
5.1 Duplication of Gadget Producing Instructions

Gadget search algorithms employed by tools such as ROPgadget [Salwan 2020] search binaries for byte sequences that encode return, indirect jump, indirect call, and system call instructions, which we refer to as gadget-producing instructions (GPIs). This search captures both GPI byte sequences intentionally inserted by the compiler (e.g., return instructions terminating functions) as well as sequences that are unintentionally encoded in headers, data sections, memory locations, constants, displacements in control-flow instructions, etc. For each GPI byte sequence found, the algorithm attempts to disassemble the bytes preceding the GPI to determine if they encode valid x86 instructions. If successful, the algorithm catalogs the valid sequence of bytes and the GPI encoding as a gadget. This is done iteratively, with each iteration attempting to disassemble a longer byte prefix to the GPI until the length of the found gadgets exceeds some useful threshold (e.g., 10 bytes in length for ROPgadget). This process identifies all intended and unintended gadgets at or below the threshold associated with each GPI, which collectively forms the binary's gadget set.

When compiler optimizations duplicate GPIs, they provide new opportunities for an attacker to find gadgets within a program. Due to the density of the x86_64 instruction set and its support for variable length instructions, each duplicated GPI is likely to introduce new, unique, and potentially useful gadgets. Unfortunately, a number of compiler optimizations increase performance by selectively duplicating code. These optimizations are typically only employed at the O3 level because code duplication increases the size of optimized binaries, however there are some notable exceptions. For example, both GCC and Clang perform inlining behavior at level O2 and above, in which a function call is replaced with the body of the called function. This allows other optimizations that are intra-procedural to optimize caller and callee code in ways that would not be possible without inlining. If the inlined function contains GPIs, these GPIs are duplicated. Due to the impacts of other optimizations and code layout on the combined code, the duplicated GPIs are likely to introduce new unique gadgets. Our differential analysis reveals that this behavior is indeed responsible for introducing new gadgets at higher optimization levels, partially explaining the negative impacts we observed in our coarse-grained analysis at levels O2 and O3.

Interestingly, our analysis revealed that this behavior also occurs with a number of GCC-specific optimizations that are employed at the O1 level. GPI duplication was most apparent for GCC's omit frame pointer optimization, which identifies functions that do not require a frame pointer and eliminates pointer setup and restore instructions from the function prologue and epilogue respectively. Due to GCC's code generation conventions, the pointer restore instruction (`pop rbp;`) is typically the lone instruction executed prior to a function's `retn` instruction. In cases where multiple code paths converge at the end of a function, eliminating the pointer restore instruction allows the return instruction to be hoisted to the end of each converging path. While this technique slightly reduces code size and execution time per path by replacing a 5-byte `jmp` instruction with a copy of the single byte `retn` instruction it targets, each duplicated `retn` instruction potentially introduces new useful gadgets. An example of this phenomenon in `httpd` is shown in Figure 1.

In the GCC variants we observed, we discovered that this behavior is not limited to the frame pointer omission optimization. We also observed this behavior with GCC's shrink wrap and tree switch conversion optimizations. Additionally, this behavior is not limited to `retn` instructions. Although rare, it is possible for indirect calls and jumps to be duplicated by optimizations for


O0 Optimization
ap_find_module_short_name()
O0 + Omit Frame Pointer Optimizations
ap_find_module_short_name()

```
mov eax, 0
jmp 0x6D0BA
```

```
mov rax, [ap_module_short_names]
mov edx, [rbp-4]
movsxd rdx, edx
shl rdx, 3
add rax, rdx
mov rax, [rax]
```

```
pop rbp
retn
```

```
mov eax, 0
retn
```

```
mov rax, [ap_module_short_names]
mov edx, [rsp-4]
movsxd rdx, edx
shl rdx, 3
add rax, rdx
mov rax, [rax]
retn
```

Fig. 1. GPI Duplication caused by GCC's Frame Pointer Omission Optimization

functions that end in another function call (i.e., a tail call). We did not observe this behavior in Clang's equivalent optimizations due to differences in how Clang generates function epilogues.

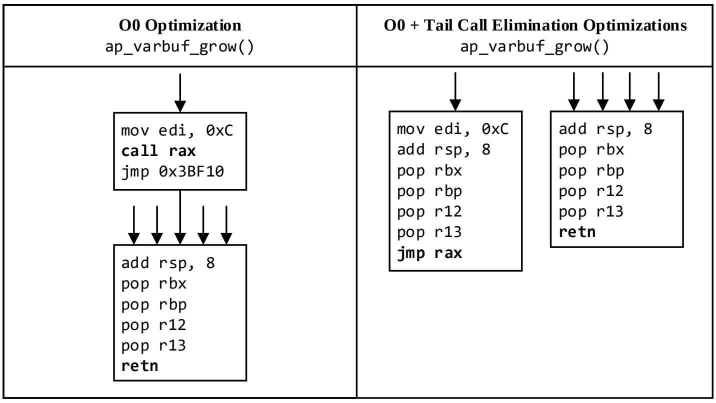
The negative effects of optimizations exhibiting this behavior occur indirectly as a result of an aggressive secondary optimization. Since a number of different optimizations exhibit this behavior, this behavior is most simply mitigated through transformation pass(es) that merge duplicated GPIs. We detail our implementation of such transformations and their effects in Section 6.

5.2 Special Purpose Gadgets Introduced as Optimizations

Complex CRA methodologies such as JOP and COP require a number of special purpose gadgets to perform important non-functional tasks in exploit chains. One such type, the JOP dataloader gadget, consists of an indirect jump preceded by a popa or multiple pop instructions. This special purpose gadget is useful for loading data injected into the stack that is necessary for the attacker's exploit into registers at the beginning of a JOP exploit. Our analysis of Clang's tail call elimination optimization revealed that the utility of this instruction sequence is not limited to exploit programming. This optimization identifies functions in which the final action is to call another function. In these cases, the caller's stack frame is no longer needed and can be reused by the callee. The optimization will replace the tail call instruction and the function terminating return instruction with a jump to the called function, eliminating the operations necessary to set up a new stack frame. When the tail call to be eliminated is an indirect call, it is replaced with an indirect jump. With respect to code reuse gadgets, the net result of this operation is the elimination of call-ending gadgets that are useful in JOP and COP exploit patterns and the introduction of jump-ending gadgets that are used in JOP exploit patterns only. Return-ending gadgets will also be eliminated if the indirect jump replaces the return instruction completely, although this only occurs if the tail call occurs on all paths to the return instruction.

However, Clang's code generation conventions for function epilogues typically include several pop instructions to restore the values of callee-saved registers (i.e., RBX, RBP, R12-R15) before returning control-flow to the calling function. When the function-terminating return instruction preceded by these pop instructions is replaced by an indirect jump, JOP dataloader gadgets are produced. An example of this behavior in httpd is shown in Figure 2.

In this case, the sequence of instructions representing the JOP dataloader gadget is intentionally placed by the compiler. The introduction of these gadgets cannot be avoided without forgoing the benefits of the optimization; either wholesale by disabling tail call elimination when building



```
mov edi, 0xC
call rax
jmp 0x3BF10
```

```
add rsp, 8
pop rbx
pop rbp
pop r12
pop r13
retn
```

```
mov edi, 0xC
add rsp, 8
pop rbx
pop rbp
pop r12
pop r13
jmp rax
```

```
add rsp, 8
pop rbx
pop rbp
pop r12
pop r13
retn
```

Fig. 2. Special Purpose Gadget Introduction caused by Clang's Tail Call Elimination Optimization

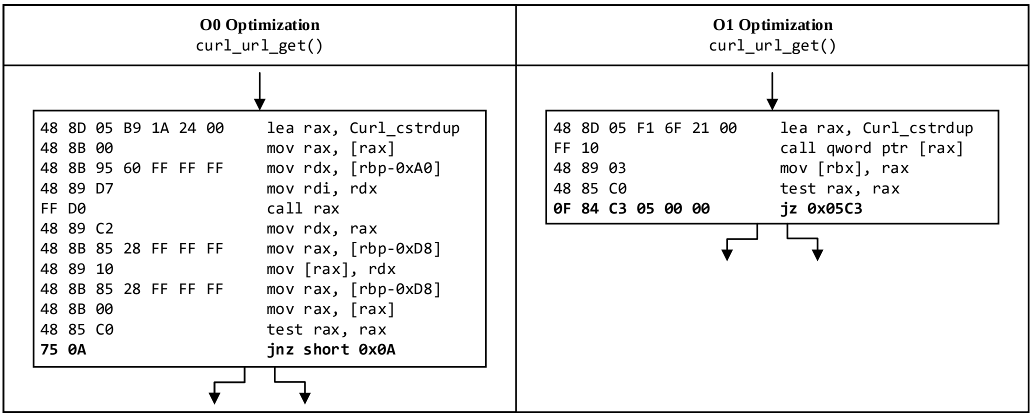
software or piecemeal with a security-sensitive version of the optimization that does not apply the transformation to indirect calls. The cost-benefit analysis of such a strategy will vary from application to application. As such, the decision to use this optimization is left to the individual building the code, as is the case with other compiler optimizations that can introduce security weaknesses (e.g., dead store elimination).

5.3 Transformation Induced Gadget Introduction

Prior work has shown that software transformation induces a significant amount of gadget introduction, even for relatively small transformations. Our analysis of single optimization variants generated by both GCC and Clang revealed that compiler optimizations also introduce a significant number of new gadgets when performing seemingly innocuous transformations. Changes to the code that precedes compiler-placed GPIs is the most readily apparent driver for changes in both intended and unintended gadgets available in optimized variants. However, optimization also induces changes in the optimized variant's layout which introduce a significant number of new, potentially useful unintended gadgets.

Layout-based unintended gadgets are primarily introduced by displacement or offset encodings for control flow instructions like unconditional jumps, conditional jumps, and function calls. If bytes corresponding to GPIs are found within these encodings, they can be used as a source of unintended gadgets. When optimizations make semantic changes to the program representation, they often introduce a number of layout changes as well, potentially altering a previously benign encoding to one containing a GPI. An example of this behavior in `libcUrl` is shown in Figure 3. In this figure, a conditional jump instruction with a short (1 byte) displacement is converted into an equivalent conditional jump instruction with a near (4-byte) displacement. This new displacement includes the byte value `0xC3`, which is the encoding for the `retn` GPI. This behavior is not limited to `retn` GPIs. We also observed that layout-based introduction was responsible for the introduction of syscall and COP intra-stack pivot special purpose gadgets, although this was observed less frequently because their encodings are multi-byte, like indirect jump and call GPIs.

We observed that changes in layout-based unintended gadgets were responsible for a significant amount of the "background noise" we observed in optimized variants. Given that this type of gadget introduction is endemic to the x86_64 instruction set architecture, it is outside the control (and concern) of the compiler's machine-independent optimizations. As such, it is not possible for



O0 Optimization
`curl_url_get()`

```
48 8D 05 B9 1A 24 00     lea rax, Curl_cstrdup
48 8B 00                 mov rax, [rax]
48 8B 95 60 FF FF FF     mov rdx, [rbp-0xA0]
48 89 D7                 mov rdi, rdx
FF D0                    call rax
48 89 C2                 mov rdx, rax
48 8B 85 28 FF FF FF     mov rax, [rbp-0xD8]
48 89 10                 mov [rax], rdx
48 8B 85 28 FF FF FF     mov rax, [rbp-0xD8]
48 8B 00                 mov rax, [rax]
48 85 C0                 test rax, rax
75 0A                    jnz short 0x0A
```

O1 Optimization
`curl_url_get()`

```
48 8D 05 F1 6F 21 00     lea rax, Curl_cstrdup
FF 10                    call qword ptr [rax]
48 89 03                 mov [rbx], rax
48 85 C0                 test rax, rax
0F 84 C3 05 00 00        jz 0x05C3
```

Fig. 3. Layout-Based Introduction of GPI and Unintended Gadgets

machine independent compiler optimizations to be designed to detect and avoid these harmful effects. An assembler injection-based solution for eliminating these types of gadgets using nop sleds has been proposed by Onarlioglu et al. [2010], however this solution is limited to `retn` GPIs and results in significant increases in code size due to the lack of direct control over the binary layout. An ideal mitigation strategy for this problem would require direct control over program layout, and would primarily avoid problematic displacements by reordering blocks and functions.

6 MITIGATION

In this section, we describe and assess two recompiler passes that implement our proposed strategy for addressing GPI duplication. With respect to our other proposed strategies, we leave the cost-benefit analysis of using Clang's tail call elimination pass to the reader as it is highly situational. Also, we consider the implementation of a broad set of binary transformations to address unintended gadgets found in binaries to be future work that is outside the scope of this paper.

6.1 Egalito Binary Recompiler

Egalito [Williams-King et al. 2020] is a binary transformation framework that lifts position-independent ELF binaries to Egalito intermediate representation (EIR). EIR is a low-level, machine-dependent, and layout-agnostic IR that allows for arbitrary transformations prior to re-compilation back to an ELF binary. Lifting to EIR requires precise disassembly of the binary, which Egalito accomplishes by using metadata present in the position-independent code (PIC) for analysis. PIC is dominant in most Linux binaries, and allows for full disassembly in a vast majority of cases.

We select Egalito as the engine for our mitigation passes for three primary reasons. First, Egalito is designed for transformation of machine-dependent EIR and therefore is readily capable of addressing code reuse gadgets, as opposed to existing compiler toolchains such as LLVM [Lattner and Adve 2004] which are primarily designed for machine-independent optimization. Second, Egalito is compiler agnostic, allowing a single implementation of our mitigation passes to address problems regardless of the compiler used. Finally, implementing our passes in Egalito allows our work to address legacy binaries, provided that they are compatible with Egalito (i.e., it is PIC).

6.2 GPI Merging

Our mitigation passes are designed to address GPI duplication behavior without significant detriment to performance. We observe that the benefits of duplicating GPIs intraprocedurally are small, saving between one and five bytes of code size per eliminated jump as well as the run time required to execute the jump. Since these benefits are secondary to the intended optimization, it is possible to re-merge split GPIs and retain the optimization's primary benefits. Our mitigation passes eliminate all but one instance of a particular GPI per function. At program points where GPIs are eliminated, an unconditional near jump to the retained GPI is inserted, effectively merging multiple GPI instances into a single one. We implemented two intraprocedural GPI merging passes: one that merges return-type instructions and one that merges indirect jump instructions.

Our return merging algorithm (Algorithm 1) operates by scanning each instruction in the function to determine if it is a return-type instruction. If multiple return-type instructions are found within the function, one is arbitrarily selected to be the target instruction. All other return type instructions are replaced with unconditional jump instructions to the target instruction. This algorithm does not differentiate between instances where return-type instructions were duplicated due to optimization versus instances where the compiler generates multiple return instructions within a function. Since the per-merge performance costs are small, we intentionally design this recompiler pass to aggressively merge return-type instructions to achieve maximum impact on the gadget set.

Our indirect jump merging algorithm (Algorithm 2) operates in a manner similar to our return merging algorithm. The primary difference is that indirect jumps can only be merged if they target the same register. During the instruction scan, if an indirect jump instruction is found, it is placed into a map data structure in which each register is mapped to a list of indirect jump instructions targeting the register. When the register scan is complete, the map is searched for any registers with more than one indirect jump instruction targeting it. If found, the indirect jump instructions are merged in the same manner as return instructions in the return merging algorithm.

Algorithm 1 Merge Return Statements

```
procedure MergeReturn(Function F)
    $ReturnSet(R_s) \leftarrow Null$
    for all $BasicBlock(BB_i) \in F$ do
        for all $instruction(I) \in BB_i$ do
            if $I$ is a return then
                $R_s \leftarrow R_s \cup I$
    $Target(T) \leftarrow$ address of some $I \in R_s$
    $R_s \leftarrow R_s - I$
    for all $return(r_i) \in R_s$ do
        Replace $r_i$ with $jmp\ T$
```

6.3 Impact on Gadget Sets

To determine how effective our passes are at reducing the availability and utility of code reuse gadgets, we used Egalito to recompile position-independent variants of our benchmark programs built at optimization level O3[1]. We then used GSA to measure the change in gadget set quality metrics after applying our passes. Our results are reported in Table 8.

[1] Several O3 program variants in our study were originally built as position-dependent binaries. These variants were rebuilt as PIC to assess our mitigations. As a result, O3 gadget set metric values in this section differ from those in Section 4.

Algorithm 2 Merge Indirect Jump Statements

```
procedure MergeIndirectJump(Function F)
    $IJMap(r, jumpList_r) \leftarrow Null$
    for all $BasicBlock(BB_i) \in F$ do
        for all $instruction(I) \in BB_i$ do
            if $I$ is a Indirect Jump then
                $R_i \leftarrow$ Indirect Jump Target Register of $I$
                $jumpList_{R_i} \leftarrow$ Jump list mapped to $IJMap(R_i)$
                Add $I$ to $jumpList_{R_i}$
    for all $jumpList_r \in IJMap$ do
        if Size of $jumpList_r > 1$ then
            $Target(T) \leftarrow$ address of some $j_r \in jumpList_r$
            $jumpList_r \leftarrow jumpList_r - j_r$
            for all $indirectJump(j_k) \in jumpList_r$ do
                Replace $j_k$ with $jmp\ T$
```

The effects of our mitigation passes on functional gadget set expressivity were mixed. While our passes did reduce the overall expressivity of the gadget set in 38.2% (13 of 34) of variants, overall expressivity increases were observed at roughly the same rate. These increases are due to transformation induced gadget introduction caused by layout changes or GPIs encoded in jump displacements introduced by our passes. These effects can be compensated for with additional mitigation passes that address unintended gadgets, which we identify as future work.

Our passes were highly effective at reducing the number of useful gadgets available after recompilation. Our passes significantly reduced the total number of quality gadgets in all 36 variants at an average rate of 30.7%. While the lowest reduction rate observed was 8%, this occurred in smaller benchmarks with very few initial gadgets. We observed a maximum reduction rate of 61%, and in 9 of 34 variants the number of quality gadgets was reduced by at least 40%. In addition to large reductions in the number of quality gadgets, the average quality score across gadgets within each set generally increased, indicating that the gadgets remaining after recompilation were generally more difficult to use in exploit chains.

Our passes were also highly effective at reducing the diversity of special purpose gadgets available in recompiled gadget sets. We observed a decrease in the number of special purpose gadget categories available in half of variants, with multiple categories of gadgets eliminated in 29.4% (10 of 34) of total variants. We observed no change in the total number of categories available in 35.3% (12 of 34) of variants. In three of these instances the unmodified O3 variant had no special purpose gadgets present to eliminate, and in the remaining instances the unmodified O3 variant had two or fewer special purpose gadget types present. We did observe an increase of one special purpose gadget type for five variants, most of which occurred in Clang produced variants.

7 PERFORMANCE IMPACTS

To answer our final motivating question, we conducted an analysis of our recompiled variants to determine the impact of our mitigating passes on execution speed and code size.

Table 8. Effects of Mitigation Passes on Gadget Set Metrics for O3 Variants

		Functional Gadget Set Expressivity			Functional Gadget Set Quality			S.P. Gadget Availability		
	Benchmark	O3	MP	Δ	O3	MP	Δ	O3	MP	Δ
GCC Variants	Bftpd	6/13/6	6/13/7	(0/0/-1)	320 / 0.93	234 / 0.90	(86 / 0.03)	2	3	(-1)
	libcUrl	9/28/14	10/27/13	(-1/1/1)	4189 / 0.70	2843 / 0.72	(1346 / -0.01)	4	4	(0)
	git	11/34/16	10/33/16	(1/1/0)	6758 / 0.90	4031 / 0.98	(2727 / -0.08)	5	3	(2)
	gzip	8/23/10	7/21/11	(1/2/-1)	360 / 0.92	269 / 1.00	(91 / -0.07)	3	2	(1)
	httpd	8/22/11	8/24/11	(0/-2/0)	2881 / 0.82	1623 / 0.87	(1258 / -0.05)	5	4	(1)
	libsqlite	10/30/13	9/30/14	(1/0/-1)	2966 / 0.86	1758 / 0.91	(1208 / -0.05)	4	2	(2)
	401.bzip2	5/16/6	5/17/6	(0/-1/0)	267 / 0.66	205 / 0.67	(62 / -0.01)	1	1	(0)
	403.gcc	11/35/17	11/35/17	(0/0/0)	10859 / 0.82	5429 / 0.95	(5430 / -0.13)	6	4	(2)
	429.mcf	5/9/4	4/8/3	(1/1/1)	88 / 0.69	74 / 0.78	(14 / -0.08)	1	1	(0)
	433.milc	7/23/8	7/25/8	(0/-2/0)	586 / 0.83	390 / 0.79	(196 / 0.03)	1	1	(0)
	444.namd	9/25/9	8/23/7	(1/2/2)	241 / 0.77	206 / 0.75	(35 / 0.02)	1	1	(0)
	445.gobmk	7/31/14	10/33/15	(-3/-2/-1)	3190 / 0.92	1294 / 1.07	(1896 / -0.16)	6	2	(4)
	456.hmmer	7/29/11	8/28/12	(-1/1/-1)	1300 / 0.82	758 / 0.97	(542 / -0.14)	3	2	(1)
	458.sjeng	8/17/9	8/21/11	(0/-4/-2)	541 / 0.80	356 / 0.84	(185 / -0.04)	3	1	(2)
	462.libquantum	6/17/6	6/16/6	(0/1/0)	236 / 0.72	175 / 0.75	(61 / -0.04)	1	1	(0)
	470.lbm	4/15/3	4/15/3	(0/0/0)	79 / 0.69	69 / 0.72	(10 / -0.03)	1	1	(0)
	482.sphinx3	9/24/11	10/27/11	(-1/-3/0)	701 / 0.84	469 / 0.87	(232 / -0.03)	2	1	(1)
Clang Variants	Bftpd	6/13/6	6/13/7	(0/0/-1)	309 / 0.91	235 / 0.90	(74 / 0.00)	2	3	(-1)
	libcUrl	8/30/13	8/26/10	(0/4/3)	4180 / 0.68	3330 / 0.67	(850 / 0.01)	4	5	(-1)
	git	10/34/17	11/34/16	(-1/0/1)	7061 / 0.80	4308 / 0.79	(2753 / 0.01)	5	3	(2)
	gzip	7/18/7	6/14/7	(1/4/0)	309 / 0.92	197 / 0.80	(112 / 0.12)	2	2	(0)
	httpd	10/27/10	8/24/10	(2/3/0)	2444 / 0.74	1479 / 0.71	(965 / 0.04)	5	3	(2)
	libsqlite	8/31/14	11/31/13	(-3/0/1)	2974 / 0.72	1895 / 0.72	(1079 / 0.00)	5	2	(3)
	401.bzip2	6/17/4	6/16/4	(0/1/0)	207 / 0.77	171 / 0.79	(36 / -0.01)	2	1	(1)
	403.gcc	11/35/17	11/34/16	(0/1/1)	9229 / 0.78	5220 / 0.88	(4009 / -0.10)	6	3	(3)
	429.mcf	6/10/6	6/10/6	(0/0/0)	108 / 0.97	90 / 0.97	(18 / 0.00)	0	0	(0)
	433.milc	8/24/10	8/25/11	(0/-1/-1)	446 / 0.92	374 / 0.87	(72 / 0.05)	0	1	(-1)
	444.namd	6/23/9	6/26/11	(0/-3/-2)	407 / 0.67	373 / 0.71	(34 / -0.04)	0	1	(-1)
	445.gobmk	10/34/15	10/32/15	(0/2/0)	3001 / 0.81	1185 / 0.91	(1816 / -0.10)	6	2	(4)
	456.hmmer	8/30/11	7/30/12	(1/0/-1)	1161 / 0.89	682 / 0.94	(479 / -0.05)	2	2	(0)
	458.sjeng	9/30/12	8/25/11	(1/5/1)	502 / 0.91	320 / 0.89	(182 / 0.01)	1	0	(1)
	462.libquantum	5/23/8	5/25/9	(0/-2/-1)	251 / 0.87	187 / 0.78	(64 / 0.09)	0	0	(0)
	470.lbm	3/13/3	3/13/3	(0/0/0)	74 / 0.84	68 / 0.88	(6 / -0.04)	0	0	(0)
	482.sphinx3	7/25/11	9/27/12	(-2/-2/-1)	691 / 0.81	519 / 0.84	(172 / -0.03)	2	1	(1)

7.1 Execution Speed

To analyze the impact of our mitigations on execution speed, we ran the position-independent O0, O3, and recompiled variants of our SPEC 2006 benchmarks using reference workloads[2]. We record the total run time for each variant across three trials to determine its average performance, shown in Figures 4 and 5. We observed that binaries recompiled with our passes enabled saw a performance slowdown of 0.28% on average, with a maximum observed slowdown of 1.9%. In five instances, performance was improved via our mitigations. These performance impacts are negligible over O3 performance, and are imperceptible considering the large performance improvements (2.3x speedup on average) O3 variants enjoy over O0 variants. Thus, recompiling production optimized

[2] While we performed functional testing of the recompiled variants of our common Linux benchmarks to ensure they were not corrupted during recompilation, we exclude them from this analysis due to a lack of standardized performance tests.

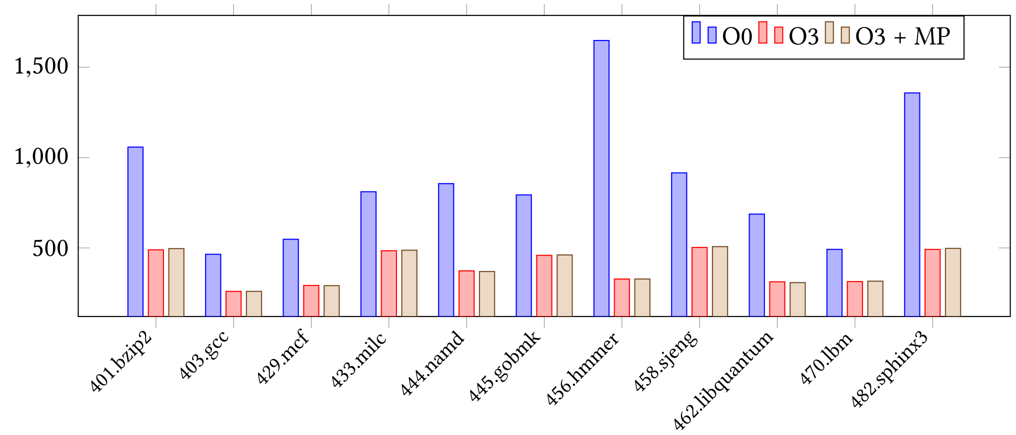


Fig. 4. Performance Comparison of SPEC 2006 GCC Variants Recompiled with Mitigation Passes (s)

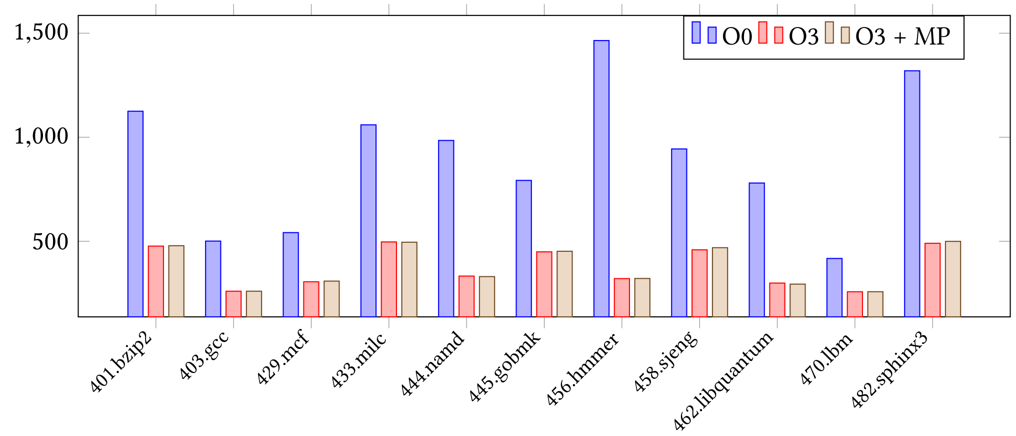


Fig. 5. Performance Comparison of SPEC 2006 Clang Variants Recompiled with Mitigation Passes (s)

code with our mitigation passes provides the best of both worlds: nearly identical performance to optimized code without the accompanying increase in gadget utility and availability.

7.2 Code Size

Recompiling binaries with Egalito can result in significant changes to code size, even when no transformation passes are run on the EIR. This is due to differences in Egalito's code generation conventions regarding program layout, function padding, and control-flow. In this analysis, we are interested both in how Egalito impacts code size as well as the incremental impact of our mitigation passes. Table 9 contains the binary sizes of the compiler-produced position-independent O3 variant (i.e., the baseline), a control variant produced by recompiling the baseline with no EIR passes selected, and a variant produced by recompiling the baseline with our passes enabled.

While the Egalito recompilation process can cause significant changes in code size (both positive and negative), our mitigation passes typically do not incrementally increase code size beyond that caused by Egalito. We observed only 7 instances across 34 variants in which our passes

Table 9. Impact of Mitigation Passes on Code Size (kB)

	GCC				Clang			
Benchmark	**O3**	**Control**	**MP**	**ΔMP**	**O3**	**Control**	**MP**	**ΔMP**
Bftpd	314.6	116.0	116.0	0	99.5	116.0	116.0	0
libcUrl	485.2	460.2	460.2	0	455.6	439.7	439.7	0
git	19403.2	3626.3	3630.4	4.1	11195.0	3151.2	3155.3	4.1
gzip	433.4	488.8	488.8	0	108.6	464.3	464.3	0
httpd	925.6	910.7	914.8	4.1	842.3	837.0	837.0	0
libsqlite	5024.5	927.1	931.2	4.1	6643.7	1328.5	1328.5	0
401.bzip2	110.6	128.3	128.3	0	98.5	116.1	116.1	0
403.gcc	4750.1	5203.3	5215.6	12.3	4021.3	4507.0	4511.1	4.1
429.mcf	26.9	54.5	54.5	0	23.0	58.7	58.7	0
433.milc	197.5	238.8	238.8	0	176.9	222.5	222.5	0
444.namd	349.5	365.9	365.9	0	333.1	353.7	353.7	0
445.gobmk	4746.6	6956.4	6956.4	0	4590.0	6804.9	6804.9	0
456.hmmer	410.6	480.5	480.5	0	376.5	452.0	452.0	0
458.sjeng	205.1	2794.8	2794.8	0	163.8	2753.9	2753.9	0
462.libquantum	55.2	75.0	75.0	0	59.7	79.2	79.2	0
470.lbm	22.1	46.3	46.3	0	22.2	50.5	50.5	0
482.sphinx3	278.8	308.5	312.6	4.1	246.8	275.8	275.8	0

incrementally increased the binary size. In all but one of these cases, the increase was less than 5 kilobytes. In the remaining case, the increase was less than 13 kilobytes. With respect to the impact of recompilation on code size, we observed that code size impacts were typically negligible. In 26.5% (9 of 34) of variants, recompilation reduced code size, and in 52.9% (18 of 34) of cases recompilation increased code size by less than 80 kilobytes. In the remaining 7 variants, large increases in code size were observed, in some cases many times over the baseline size. This potentially indicates edge cases where Egalito's code generation conventions can be improved.

8 LIMITATIONS AND CONSIDERATIONS

8.0.1 Study Limitations. Our study is limited to C/C++ source programs compiled by GCC and Clang that target x86-64 machines. Further research is necessary to determine the effects of optimizations implemented in other compilers and architectures. Also, our fine-grained analysis is limited to a single optimization per variant as it was neither feasible nor necessary to consider each permutation of optimizations. In practice, optimizations are known to interact with each other; these synergistic effects may need to be considered when applying mitigation strategies.

Linked libraries our benchmarks depend on were not analyzed in our study. Because these libraries are mapped into the program's memory space at run-time, gadgets found in library code area also available to the attacker. However, utilizing them is difficult in practice due to defensive techniques such as ASLR [PaX 2020]. Fortunately, Egalito provides a Union ELF mode in which the input binary and its linked libraries are combined into a single output ELF prior during recompilation. In this mode, EIR passes like mitigation passes we present in this paper can be readily applied to library code.

8.0.2 Tool Limitations. GSA's functional gadget set expressivity analysis is limited to small size ROP gadgets only. In practice, this covers the majority of practical security use cases as ROP exploits typically avoid long gadgets and JOP / COP exploits are rare in the wild. Future research in this area should consider expressivity impacts with respect to JOP and COP, as well as the expressive contributions of arbitrary length ROP gadgets.

Our gadget eliminating passes implemented for the Egalito binary recompiler are subject to the same limitations as the tool itself. Of particular importance, Egalito requires input programs to be position-independent code (PIC). Additionally, Egalito does not support obfuscated code or programs with inline assembly, among other limitations.

9 RELATED WORK

In response to the introduction of gadget-based CRA methods, a number of defenses against these attacks have been proposed. These techniques can be categorized into compiler-based defenses and retrofitting transformations on binaries. Both categories of approaches incur cost in the form of increased code size and run-time execution/memory overhead. These techniques have been shown to be effective at mitigating CRAs, however weaknesses in these approaches have also been identified [Conti et al. 2015; Davi et al. 2014; Evans et al. 2015; Schwartz et al. 2011].

Compiler-based defenses perform two tasks in the compiler. First, they rewrite instructions that contain unintended gadgets into semantically equivalent code that does not. Second, intended gadgets are secured by rewriting the instructions to use alternate control flow mechanisms [Li et al. 2010] or by inserting simple run-time protections such as encrypted branch targets or control-flow locks [Bletsch et al. 2011a; Onarlioglu et al. 2010]. Retrofitting transformations operate on binaries after they have been produced by the compiler. In general, these techniques identify and protect control flow branches and targets using inserted instructions that enforce run-time control-flow integrity [Abadi et al. 2005; Hawkins et al. 2016; Kayaalp et al. 2012b; Zhang and Sekar 2013] or detect anomalous run-time behavior [Chen et al. 2009; Davi et al. 2009, 2011; Yao et al. 2013].

Our approach differs from prior work primarily in that it proactively eliminates intended gadgets via post-compiler transformation techniques rather than relying on costly run-time protections or static transformations that cause large increases in code size. We consider our proposed mitigations to be complementary to these approaches; our proposed mitigations can potentially reduce the overhead costs of other defenses by reducing the number of unintended gadgets that require transformation and distinct branches that require instrumentation and run-time protection.

A recent study conducted by Louboutin et al. [2019] explored the effects of production environment on gadget density. One experiment conducted in this work compares the resulting gadget densities of two programs created with GCC and Clang. They found that compilation options produce similar effects on both binaries and that the choice of compiler influences the resulting gadget density. However, the sample size of the study is very small and the metric used in this work is derived from gadget counts, which have been shown to be a poor security-oriented metric [Brown and Pande 2019a], limiting the utility of these findings.

10 CONCLUSION

We presented the results of our broad study of the impacts of compiler optimizations on the code reuse gadget sets in optimized binaries. Through coarse and fine-grained analysis of optimization behavior across 1,000 variants of 17 different benchmark programs, we discovered that the gadget sets present in optimized code are significantly more useful for constructing CRA exploit chains than unoptimized code. We identified and detailed the root causes of our observations through differential binary analysis, and proposed potential mitigation strategies. We implemented one of these strategies, GPI merging, and showed that it can significantly reduce the availability and utility of code reuse gadget sets in optimized code. Finally, we demonstrated that these benefits can be obtained with negligible performance impact through a performance analysis of binaries transformed with our GPI merging passes.

ACKNOWLEDGMENTS

We would like to thank all of our reviewers for their helpful feedback. We also thank David Williams-King for making Egalito available for this research and for his assistance in resolving technical issues related to our work. Finally, we would also like to thank Tejas Vedantham and Chris Porter for their assistance with generating portions of the experimental data used in this study.

REFERENCES

Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow Integrity: Principles, Implementations, and Applications. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS '05)*. Association for Computing Machinery, New York, NY, USA, 340âĂŞ353.

Nicolas Belleville, Damien Couroussé, Karine Heydemann, and Henri-Pierre Charles. 2018. Automated Software Protection for the Masses Against Side-Channel Attacks. *ACM Trans. Archit. Code Optim.* 15, 4, Article 47 (Nov. 2018), 27 pages. https://doi.org/10.1145/3281662

Frédéric Besson, Alexandre Dang, and Thomas Jensen. 2018. Securing Compilation Against Memory Probing. In *Proceedings of the 13th Workshop on Programming Languages and Analysis for Security* (Toronto, Canada) *(PLAS '18)*. Association for Computing Machinery, New York, NY, USA, 29âĂŞ40. https://doi.org/10.1145/3264820.3264822

Tyler Bletsch, Xuxian Jiang, and Vince Freeh. 2011a. Mitigating Code-Reuse Attacks with Control-Flow Locking. In *Proceedings of the 27th Annual Computer Security Applications Conference* (Orlando, Florida, USA) *(ACSAC '11)*. Association for Computing Machinery, New York, NY, USA, 353âĂŞ362. https://doi.org/10.1145/2076732.2076783

Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. 2011b. Jump-Oriented Programming: A New Class of Code-Reuse Attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security* (Hong Kong, China) *(ASIACCS '11)*. Association for Computing Machinery, New York, NY, USA, 30âĂŞ40. https://doi.org/10.1145/1966913.1966919

Michael D. Brown. 2020. GadgetSetAnalyzer. https://github.com/michaelbrownuc/GadgetSetAnalyzer.

Michael D. Brown and Santosh Pande. 2019a. Is Less Really More? Towards Better Metrics for Measuring Security Improvements Realized Through Software Debloating. In *12th USENIX Workshop on Cyber Security Experimentation and Test (CSET 19)*. USENIX Association, Santa Clara, CA. https://www.usenix.org/conference/cset19/presentation/brown

Michael D. Brown and Santosh Pande. 2019b. Is Less Really More? Why Reducing Code Reuse Gadget Counts via Software Debloating Doesn't Necessarily Indicate Improved Security. arXiv:arXiv:1902.10880

Nicholas Carlini and David Wagner. 2014. {ROP} is Still Dangerous: Breaking Modern Defenses. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*. 385–399.

Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. 2010. Return-Oriented Programming without Returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (Chicago, Illinois, USA) *(CCS '10)*. Association for Computing Machinery, New York, NY, USA, 559âĂŞ572. https://doi.org/10.1145/1866307.1866370

Ping Chen, Hai Xiao, Xiaobin Shen, Xinchun Yin, Bing Mao, and Li Xie. 2009. DROP: Detecting return-oriented programming malicious code. In *International Conference on Information Systems Security*. Springer, 163–177.

Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. 2015. Losing Control: On the Effectiveness of Control-Flow Integrity under Stack Attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (Denver, Colorado, USA) *(CCS '15)*. Association for Computing Machinery, New York, NY, USA, 952âĂŞ963. https://doi.org/10.1145/2810103.2813671

Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. 2014. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, 401–416. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/davi

Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. 2009. Dynamic Integrity Measurement and Attestation: Towards Defense against Return-Oriented Programming Attacks. In *Proceedings of the 2009 ACM Workshop on Scalable Trusted Computing* (Chicago, Illinois, USA) *(STC '09)*. Association for Computing Machinery, New York, NY, USA, 49âĂŞ54. https://doi.org/10.1145/1655108.1655117

Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. 2011. ROPdefender: A Detection Tool to Defend against Return-Oriented Programming Attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security* (Hong Kong, China) *(ASIACCS '11)*. Association for Computing Machinery, New York, NY, USA, 40âĂŞ51. https://doi.org/10.1145/1966913.1966920

Chaoqiang Deng and Kedar S. Namjoshi. 2017. Securing the SSA Transform. In *Static Analysis*, Francesco Ranzato (Ed.). Springer International Publishing, Cham, 88–105.

Chaoqiang Deng and Kedar S Namjoshi. 2018. Securing a compiler transformation. *Formal Methods in System Design* 53, 2 (2018), 166–188.

Vijay D'Silva, Mathias Payer, and Dawn Song. 2015. The Correctness-Security Gap in Compiler Optimization. In *Proceedings of the 2015 IEEE Security and Privacy Workshops (SPW '15)*. IEEE Computer Society, USA, 73âĂŞ87.

Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. 2015. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (Denver, Colorado, USA) *(CCS '15)*. Association for Computing Machinery, New York, NY, USA, 901âĂŞ913. https://doi.org/10.1145/2810103.2813646

Andreas Follner, Alexandre Bartel, and Eric Bodden. 2016. Analyzing the gadgets. In *International Symposium on Engineering Secure Software and Systems*. Springer, 155–172.

B. Hawkins, B. Demsky, and M. B. Taylor. 2016. BlackBox: Lightweight security monitoring for COTS binaries. In *2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 261–272.

Hex-Rays. 2020. IDA Pro. https://www.hex-rays.com/products/ida/.

Andrei Homescu, Michael Stewart, Per Larsen, Stefan Brunthaler, and Michael Franz. 2012. Microgadgets: size does matter in turing-complete return-oriented programming. In *Proceedings of the 6th USENIX conference on Offensive Technologies*. USENIX Association, 7–7.

Mehmet Kayaalp, Meltem Ozsoy, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2012a. Branch Regulation: Low-Overhead Protection from Code Reuse Attacks. In *Proceedings of the 39th Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '12)*. IEEE Computer Society, USA, 94âĂŞ105.

Mehmet Kayaalp, Meltem Ozsoy, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2012b. Branch Regulation: Low-Overhead Protection from Code Reuse Attacks. In *Proceedings of the 39th Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '12)*. IEEE Computer Society, USA, 94âĂŞ105.

Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization* (Palo Alto, California) *(CGO '04)*. IEEE Computer Society, USA, 75.

Jinku Li, Zhi Wang, Xuxian Jiang, Michael Grace, and Sina Bahram. 2010. Defeating Return-Oriented Rootkits with âĂIJReturn-LessâĂİ Kernels. In *Proceedings of the 5th European Conference on Computer Systems* (Paris, France) *(EuroSys '10)*. Association for Computing Machinery, New York, NY, USA, 195âĂŞ208. https://doi.org/10.1145/1755913.1755934

Jay P. Lim, Vinod Ganapathy, and Santosh Nagarakatte. 2017. Compiler Optimizations with Retrofitting Transformations: Is There a Semantic Mismatch?. In *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security* (Dallas, Texas, USA) *(PLAS '17)*. Association for Computing Machinery, New York, NY, USA, 37âĂŞ42. https://doi.org/10.1145/3139337.3139343

Étienne Louboutin, Jean-Christophe Bach, and Fabien Dagnat. 2019. Statistical Measurement of Production Environment Influence on Code Reuse Availability. In *SECURWARE 2019 : The Thirteenth International Conference on Emerging Security Information, Systems and Technologies*. Nice, France. https://hal.archives-ouvertes.fr/hal-02354761

Paul Muntean, Matthias Neumayer, Zhiqiang Lin, Gang Tan, Jens Grossklags, and Claudia Eckert. 2019. Analyzing control flow integrity with LLVM-CFI. In *Proceedings of the 35th Annual Computer Security Applications Conference*. 584–597.

Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. 2010. G-Free: Defeating Return-Oriented Programming through Gadget-Less Binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference* (Austin, Texas, USA) *(ACSAC '10)*. Association for Computing Machinery, New York, NY, USA, 49âĂŞ58. https://doi.org/10.1145/1920261.1920269

PaX. 2020. Address Space Layout Randomization. https://pax.grsecurity.net/docs/aslr.txt.

Julien Proy, Karine Heydemann, Alexandre Berzati, and Albert Cohen. 2017. Compiler-Assisted Loop Hardening Against Fault Attacks. *ACM Trans. Archit. Code Optim.* 14, 4, Article 36 (Dec. 2017), 25 pages. https://doi.org/10.1145/3141234

AliAkbar Sadeghi, Salman Niksefat, and Maryam Rostamipour. 2018. Pure-Call Oriented Programming (PCOP): chaining the gadgets using call instructions. *Journal of Computer Virology and Hacking Techniques* 14, 2 (2018), 139–156.

Jonathan Salwan. 2020. ROPgadget - Gadgets finder and auto-roper. http://shell-storm.org/project/ROPgadget/.

Edward J Schwartz, Thanassis Avgerinos, and David Brumley. 2011. Q: Exploit Hardening Made Easy.. In *USENIX Security Symposium*. 25–41.

Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-Libc without Function Calls (on the X86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security* (Alexandria, Virginia, USA) *(CCS '07)*. Association for Computing Machinery, New York, NY, USA, 552âĂŞ561. https://doi.org/10.1145/1315245.1315313

L. Simon, D. Chisnall, and R. Anderson. 2018. What You Get is What You C: Controlling Side Effects in Mainstream C Compilers. In *2018 IEEE European Symposium on Security and Privacy (EuroS P)*. 1–15.

Victor van der Veen, Dennis Andriesse, Manolis Stamatogiannakis, Xi Chen, Herbert Bos, and Cristiano Giuffrdia. 2017. The dynamics of innocent flesh on the bone: Code reuse ten years later. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 1675–1689.

David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P Kemerlis. 2020. Egalito: Layout-Agnostic Binary Recompilation. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 133–147.

F. Yao, J. Chen, and G. Venkataramani. 2013. JOP-alarm: Detecting jump-oriented programming-based anomalies in applications. In *2013 IEEE 31st International Conference on Computer Design (ICCD)*. 467–470.

Mingwei Zhang and R. Sekar. 2013. Control Flow Integrity for COTS Binaries. In *22nd USENIX Security Symposium (USENIX Security 13)*. USENIX Association, Washington, D.C., 337–352. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/Zhang zynamics. 2020. zynamics BinDiff. https://www.zynamics.com/bindiff.html.

Table 10. GCC Single Optimization Variants Studied

O0	O1	O2
(1) tree-sink	(35) partial-inlining	(73) peel-loops
(2) ipa-profile	(36) ipa-icf	(74) tree-loop-vectorize
(3) tree-bit-ccp	(37) indirect-inlining	(75) inline-functions
(4) branch-count-reg	(38) tree-tail-merge	(76) predictive-commoning
(5) forward-propagate	(39) reorder-functions	(77) tree-slp-vectorize
(6) compare-elim	(40) ipa-ra	(78) split-paths
(7) ssa-phiopt	(41) isolate-erroneous-paths-dereference	(79) tree-partial-pre
(8) tree-ch	(42) ipa-cp	(80) tree-loop-distribute-patterns
(9) cprop-registers	(43) reorder-blocks-and-partition	(81) unswitch-loops
(10) tree-dse	(44) caller-saves	(82) ipa-cp-clone
(11) ipa-reference	(45) expensive-optimizations	(83) split-loops
(12) tree-sra	(46) ipa-icf-variables	(84) gcse-after-reload
(13) tree-builtin-call-dce	(47) optimize-strlen	
(14) tree-fre	(48) crossjumping	
(15) tree-coalesce-vars	(49) ipa-vrp	
(16) split-wide-types	(50) thread-jumps	
(17) tree-ccp	(51) ipa-icf-functions	
(18) tree-dce	(52) gcse	
(19) reorder-blocks	(53) code-hoisting	
(20) tree-dominator-opts	(54) strict-overflow	
(21) tree-pta	(55) devirtualize-speculatively	
(22) inline-functions-called-once	(56) devirtualize	
(23) tree-ter	(57) cse-follow-jumps	
(24) guess-branch-probability	(58) ira-remat	
(25) move-loop-invariants	(59) rerun-cse-after-loop	
(26) ipa-pure-const	(60) tree-switch-conversion	
(27) defer-pop	(61) hoist-adjacent-loads	
(28) tree-slsr	(62) store-merging	
(29) omit-frame-pointer	(63) align-labels	
(30) shrink-wrap	(64) schedule-insns2	
(31) tree-copy-prop	(65) ipa-sra	
(32) if-conversion	(66) peephole2	
(33) combine-stack-adjustments	(67) tree-vrp	
(34) if-conversion	(68) ipa-bit-cp	
	(69) optimize-sibling-calls	
	(70) inline-small-functions	
	(71) tree-pre	
	(72) strict-aliasing	

A APPENDICES

A.1 Single Optimization Variants Analyzed

Table 10 contains the single optimization variants that were generated using the GCC compiler for this study. Table 11 contains the single optimization variants that were generated using the Clang compiler for this study. In both tables, the column identifier indicates the baseline optimization that the single optimization variant was compared against using GSA.

Table 11. Clang Single Optimization Variants Studied

O0		O1	O2
(1) loop-deletion	(21) bdce	(41) mldst-motion	(47) callsite-splitting
(2) loop-distribute	(22) loop-simplify	(42) gvn	(48) aggressive-instcombine
(3) sroa	(23) instcombine	(43) slp-vectorizer	(49) argpromotion
(4) adce	(24) jump-threading	(44) constmerge	
(5) memcpyopt	(25) simplifycfg	(45) inline	
(6) deadargelim	(26) div-rem-pairs	(46) elim-avail-extern	
(7) correlated-propagation	(27) libcalls-shrinkwrap		
(8) loop-rotate	(28) globaldce		
(9) functionattrs	(29) dse		
(10) loop-idiom	(30) loop-sink		
(11) omit-frame-pointer	(31) loop-unroll		
(12) lcssa	(32) loop-vectorize		
(13) reassociate	(33) tailcallelim		
(14) loop-load-elim	(34) alignment-from-assumptions		
(15) speculative-execution	(35) licm		
(16) loop-unswitch	(36) strip-dead-prototypes		
(17) early-cse	(37) float2int		
(18) indvars	(38) prune-eh		
(19) sccp	(39) ipsccp		
(20) globalopt	(40) called-value-propagation		