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 for implementing 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 sets. We analyze and compare 1,187 variants of 20 different benchmark programs built with two production compilers (`GCC` and `Clang`) to determine how their optimization behaviors affect the code reuse gadget sets present in program variants with respect to both quantitative and qualitative metrics. Our study exposes an important and unexpected problem; compiler optimizations introduce new gadgets at a high rate and produce code containing gadget sets that are generally more useful to an attacker than those in unoptimized code. Using differential binary analysis, we identify several undesirable behaviors at the root of this phenomenon. In turn, we propose and evaluate several strategies to mitigate these behaviors. In particular, we show that post-production binary recompilation can effectively mitigate these behaviors with negligible performance impacts, resulting in optimized code with significantly smaller and less useful gadget sets.

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, Software security, Code reuse attacks, Code reuse gadgets, Binary recompilation

ACM Reference Format:
Michael D. Brown, Matthew Pruett, Robert Bigelow, Girish Mururu, and Santosh Pande. 2021. Not So Fast: Understanding and Mitigating Negative Impacts of Compiler Optimizations on Code Reuse Gadget Sets. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 154 (October 2021), 30 pages. https://doi.org/10.1145/3485531

Authors' addresses: Michael D. Brown, School of Computer Science, Georgia Institute of Technology, Atlanta, GA, USA, mbrown337@gatech.edu; Matthew Pruett, School of Electrical & Computer Engineering, Georgia Institute of Technology, Atlanta, GA, USA, matthew.pruett@gtri.gatech.edu; Robert Bigelow, School of Computer Science, Georgia Institute of Technology, Atlanta, GA, USA, rbigelow3@gatech.edu; Girish Mururu, School of Computer Science, Georgia Institute of Technology, Atlanta, GA, USA, girishmururu@gatech.edu; Santosh Pande, School of Computer Science, Georgia Institute of Technology, Atlanta, GA, USA, santosh.pande@cc.gatech.edu.

2475-1421/2021/10-ART154
https://doi.org/10.1145/3485531

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 impacts 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 vulnerabilities. Further, security issues caused by 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]. Of particular note is prior work by D'Silva et al. [2015] that has shown that compiler optimizations can introduce semantic security weaknesses such as information leaks, elimination of security-relevant code, and side channels despite being formally proven correct. Additionally, our recent work [Brown and Pande 2019] suggests that compiler optimizations may introduce a fourth, non-semantic, class of security weakness: increased availability and quality of code reuse attack (CRA) gadgets.

Gadget-based CRAs obviate the need to inject malicious code by chaining together snippets of the vulnerable program's code called gadgets to implement an exploit. The diversity and utility of gadgets available for creating exploits depends on the compiler-generated code in a program's binary and its linked libraries. Gadget-based CRAs are particularly insidious because they circumvent code injection defenses and can achieve Turing-completeness [Bletsch et al. 2011b; Checkoway et al. 2010; Sadeghi et al. 2018; Shacham 2007]. Numerous gadget-based CRA defenses have been proposed, however their adoption remains low due to weaknesses against increasingly complex attack patterns and runtime overhead costs [Carlini et al. 2015; Carlini and Wagner 2014; Davi et al. 2014; Evans et al. 2015; Kayaalp et al. 2012a; Muntean et al. 2019; van der Veen et al. 2017].

1.1 Motivation

Our prior work demonstrated that software debloating transformations introduce new code reuse gadgets into debloated binaries at a high rate, despite an overall reduction in the total number of gadgets [Brown and Pande 2019]. Further, we showed that in many cases the introduced gadgets are more useful to an attacker than the gadgets that were removed, negatively impacting security. Interestingly, we identified differences in compiler optimization behavior on bloated versus debloated source code as one cause of gadget introduction. In contrast to debloating transformations that are localized in nature, compiler optimizations perform a massive amount of intra- and inter-procedural code restructuring on a whole-program basis. This raises several important and as of yet unanswered questions about how these optimizations impact the code reuse gadget sets present in optimized binaries. Thus, we are motivated to answer the following questions:

(1) To what degree do compiler optimizations introduce new CRA gadgets into 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 optimization behaviors cause negative security impacts?
(4) What are the root causes of these negative security impacts?
(5) How can these negative security impacts be mitigated?
(6) What is the performance cost of implementing such mitigations?

1.2 Summary of Contributions

To answer our first three motivating questions, we developed a systematic data-driven methodology to study compiler optimization behavior. Employing this methodology, we built 1,187 total variants of 20 different benchmark programs using two production compilers, `GCC` and `Clang`. Each variant was built with a different optimization configuration to enable coarse-grained analysis of predefined optimization levels (i.e., O0, O1, O2, O3) and fine-grained analysis that isolates the impacts of individual compiler optimization behaviors. We then analyzed and compared these variants to identify negative security impacts using the Gadget Set Analyzer (GSA) [Brown 2020]. GSA is a static binary analysis tool that catalogs, filters, classifies, and scores gadgets in variant binaries. Using this information, GSA calculates quantitative and qualitative security-oriented metrics for each gadget set that measure its composition, expressive power, special purpose capabilities, and suitability for exploit creation. GSA then compares these metrics across variants to identify how security is impacted by compiler optimizations.

The results of our study are concerning. Our coarse-grained analysis (Section 4.2) revealed that code reuse gadget sets present in optimized program variants are significantly more useful to an attacker than the set in the program's unoptimized variant. These findings are ubiquitous; we observed significant negative security impacts across all compilers, optimization levels, and benchmarks. Diving deeper, our fine-grained analysis (Section 4.3) revealed that these impacts are not localized to a few problematic optimizations. While we did observe several optimizations with specific undesirable behaviors, we also observed that large majority of individual optimizations caused small but measurable negative security impacts with respect to gadget sets.

To answer our last three motivating questions, we conducted differential binary analysis of our variants to identify the root causes of these negative security impacts. Our analysis revealed three distinct root causes: duplication of gadget producing instructions (GPIs), transformation-induced gadget introduction, and special purpose gadgets introduced explicitly by optimizations. After further study of these phenomena, we formulated potential mitigation strategies for each.

We implemented our proposed mitigation strategies for GPI duplication and transformation-induced gadget introduction as a suite of five binary recompiler passes that address undesirable optimization behaviors without sacrificing their desirable performance benefits. When used to recompile optimized variants of our benchmark programs, these passes reduced the total number of special purpose gadget types available in the recompiled binaries by 34% and reduced the number of useful gadgets by an average of 31.8%. Our passes also reduced the total expressive power of the remaining gadgets in 78% of cases and successfully reduced 81% of fully expressive gadget sets in optimized variants to a less than fully expressive level. Performance analysis of our passes shows that these benefits can be obtained with negligible impact to execution speed and code size. Recompiling optimized variants with our passes resulted in an average **speedup** of 0.2% and incremental static code size increase of 0.5% (6.1 kilobytes).

Finally, we evaluated the performance impacts of a potential mitigation strategy for special purpose gadgets introduced explicitly by optimizations. Specifically, we evaluated the common practice of disabling optimizations that produce negative security impacts for `Clang`'s tail-call elimination (TCE) optimization. We observed that disabling TCE increases execution time significantly, by 14% on average. Our evaluation suggests that this basic strategy is likely cost-ineffective for preventing the introduction of certain special purpose gadget types during optimization.

Our contributions are organized as follows. In Section 3 we detail our data-driven study methodology. In Section 4, we present the results of our study of the impacts of compiler optimizations on code reuse gadget sets. In Section 5, we identify the root causes of and propose mitigation

strategies for the negative security impacts we observed. In Section 6, we detail our implementation, evaluation, and performance analysis of our proposed mitigation strategies.

2 BACKGROUND

2.1 Gadget-Based Code Reuse Attacks (CRAs)

CRA methods circumvent code injection defenses (e.g., W⊕X) by chaining existing executable snippets of a vulnerable program called gadgets into an exploit payload. Shacham [2007] proposed the first gadget-based CRA method, Return-Oriented Programming (ROP), to overcome the expressivity limitations of early CRA methods (e.g., return-to-libc) that rely on the malicious execution of an existing library function. ROP has been shown to be Turing-complete if a sufficiently expressive set of gadgets is present in the vulnerable program, meaning an attacker can construct and execute any arbitrary program. Several alternative methods to ROP have been introduced, such as jump- and call-oriented programming (JOP, COP) [Bletsch et al. 2011b; Checkoway et al. 2010; Sadeghi et al. 2018].

2.2 Gadget Types

A CRA gadget is a sequence of machine instructions that ends with one of the control-flow transfer instructions listed in Table 1, which we refer to as **Gadget-Producing Instructions (GPIs)**. Not all control-flow transfer instructions are GPIs; specifically direct jump and call instructions are not GPIs. However, conditional and unconditional jumps can be included as one of a gadget's intermediate instructions. Such gadgets are called **multi-branch gadgets**.

When chained together using the GPI's control flow properties, the gadget sequence is equivalent to an executable program built entirely from existing code segments. Individual gadgets in an exploit chain are used for one of two purposes. **Functional gadgets** perform computational tasks such as adding values or loading a value into a register. The attacker uses functional gadgets to express their malicious exploit. **Special purpose gadgets** perform critical non-expressive actions such as invoking system calls (i.e., syscall gadgets) or maintaining control flow in JOP/COP exploits.

An attacker is not limited to the instructions explicitly generated by the compiler, called **intended gadgets**, when building exploits. Because `x86-64` instructions are variable length, an attacker can redirect execution to any program byte offset and interpret the byte sequence starting there as a gadget. Due to the density of the `x86-64` encoding space, programs contain large numbers of legal instruction sequences not explicitly generated by the compiler, called **unintended gadgets**. For example, consider the byte sequence `{83 C0 5B; C3;}`, which encodes the intended gadget `{add eax, 91; ret;}`. If the attacker increments the offset of this gadget by 2 bytes, they can use the unintended gadget `{pop rbx; ret;}` encoded by the sequence `{5B; C3;}`.

2.3 Exploitation

Gadget-based CRA patterns are typically used to exploit programs with stack-based memory corruption vulnerabilities (e.g., CWE-121). First, the attacker must identify a suitable vulnerability in the target program to exploit, either by confirming the target program contains a publicly-disclosed vulnerability or by discovering a novel vulnerability via methods such as fuzzing, symbolic execution, or binary reverse engineering. Next, the attacker searches the vulnerable program for gadgets and selects those useful for expressing their malicious intent. They then use the selected gadgets to generate a payload that will overflow the vulnerable buffer, write their gadget chain and necessary data to the stack, and overwrite the return address on the stack with the address of the first gadget in the chain. Several automated tools exist that can aid attackers by simplifying

Table 1. GPIs and Exploit Patterns (x86-64)

Exploit Pattern	GPI Group

ROP	`retn; retf;`
	`retn <imm>;`
	`retf <imm>;`

JOP	`jmp <reg>;`
	`jmp [reg];`
	`jmp [reg+off];`

JOP / COP	`call <reg>;`
	`call [reg];`
	`call [reg+off];`

All (Syscall)	`int 0x80;`
	`call PTR 0x10;`
	`syscall; sysenter;`

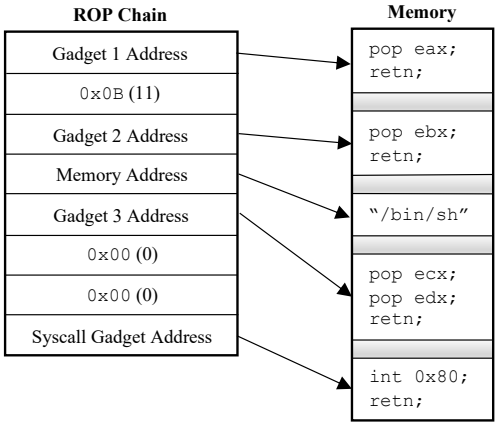

Fig. 1. Simple ROP Exploit Chain

or automating this process [Angelini et al. 2018; Follner et al. 2016b; Salwan 2020; Schwartz et al. 2011]. When delivered, the payload will hijack control-flow and execute the gadgets in sequence.

Figure 1 depicts a simple ROP chain that opens a shell. The first element of the ROP chain is the address of a functional gadget in the vulnerable program's memory. It pops an attacker controlled value, 11 (i.e., the identifier for the `execve` system call), from the stack into the `eax` register and returns. The return instruction then transfers control flow to the next gadget via the next address on the stack. This functional gadget loads the address of a string (i.e., `"/bin/sh"`) in program memory into the `ebx` register and returns. In a similar manner, the third gadget loads null values into the `ecx` and `edx` registers. Control-flow then transfers to the final syscall special purpose gadget, which invokes `execve /bin/sh`, opening a shell at the privilege level of the vulnerable program.

2.4 Defenses

Several defensive techniques against gadget-based CRA methods have been proposed and can be generally categorized as compiler-based defenses or binary retrofitting transformations. Both approaches incur increased code size and execution run-time as a trade-off.

Compiler-based defenses eliminate unintended gadgets by rewriting instructions that contain GPI encodings into equivalent instructions that do not. They then secure intended gadgets by rewriting GPIs to use alternate control-flow mechanisms [Li et al. 2010] or by inserting run-time protections such as encrypted branch targets or control-flow locks to protect them [Bletsch et al. 2011a; Onarlioglu et al. 2010]. These approaches incur significant performance degradation with average slowdowns in the 5%-15% range and code size increases of ≈26%. These impacts are primarily a factor of the number of GPIs in the program; programs with fewer GPIs generally incur less overhead to secure.

Binary retrofitting transformations generally operate by protecting control-flow branches/targets with inserted controls that enforce run-time control-flow integrity (CFI) [Abadi et al. 2005; Hawkins et al. 2016; Kayaalp et al. 2012a; Zhang and Sekar 2013] or detect anomalous run-time behavior [Chen et al. 2009; Davi et al. 2009, 2011; Yao et al. 2013]. Due to the high frequency of branches in code [McFarling and Hennesey 1986], fully-precise detection and mitigation of CRAs with these approaches is very expensive. As such, CFI implementations make precision/performance trade-offs that balance their effectiveness with their performance costs. Coarse-grained CFI implementations [Hawkins et al. 2016; Kayaalp et al. 2012a; Zhang and Sekar 2013] have performance

Table 2. Study Benchmarks

Common Linux Programs		SPEC 2006 Benchmark Set			
Bftpd v5.1	git v2.21.0	401.bzip2	403.gcc	429.mcf	433.milc
gzip v1.10	httpd v2.4.39	444.namd	445.gobmk	453.povray	456.hmmer
libcUrl v7.65.0	liblmdb v0.9.23	458.sjeng	462.libquantum	470.lbm	471.omnetpp
libsqlite v3.28.0		482.sphinx3			

overheads as low as 3% on average, however they have been shown to be ineffective due to their lack of precision [Davi et al. 2014; Schwartz et al. 2011]. More precise (and expensive) fine-grained CFI implementations incur a 16% slowdown and an 8% increase in code size on average, yet still have been shown to have serious weaknesses [Conti et al. 2015; Evans et al. 2015]. Even worse, gadget-based control-flow bending attacks have been shown to be viable against a hypothetical fully-precise CFI implementation [Carlini et al. 2015].

2.5 Transformation-Induced Gadget Introduction

Our prior work has shown that software transformations can significantly change the quantity and quality of gadgets found in a binary [Brown and Pande 2019]. Changes to source code or intermediate representation (IR) are reflected in the downstream compiler-produced binary, 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. This work has shown that high gadget introduction rates occur even in conservative debloating transformations; on average 39.5% of the gadgets present in debloated binaries were not present in the original binaries.

3 DATA-DRIVEN STUDY METHODOLOGY

Characterizing the impact of compiler optimizations on gadget availability and utility is a complex problem. Modern compilers employ a vast number of optimizations, many of which are synergistic and require careful phase ordering to be effective. Individual optimizations are also quite diverse in their features; they have varying goals (e.g., reducing code size vs. reducing execution time), make different trade-offs (e.g., increase code size to reduce execution time), and have different scopes (e.g., intra- vs. inter-procedural). As a result, compilers bundle optimizations into levels (i.e., O0 - O3) tuned for different objectives such as fast compilation or fast execution. In this section, we present the systematic, data-driven study methodology we developed to characterize the collective and individual security-oriented impacts of compiler optimizations. Our approach is similar in nature to coarse- and fine-grained studies introduced in prior work [Grant et al. 1999; Lee et al. 2006].

3.1 Benchmark Selection and Variant Generation

We selected a total of 20 C/C++ benchmark programs for this study that are diverse in size, complexity, and functionality (see Table 2). We sourced 13 of our benchmarks from the 29 programs in the SPEC 2006 benchmark set. Ten programs were excluded from this set because they were implemented in Fortran, and six were excluded because `Clang` fails when trying to build them. We then selected seven additional common Linux programs and libraries for our study to further increase the size and diversity of our benchmark set.

We then used two production compilers, `GCC` v7.4.0 and `Clang` v8.0.1, to build 1,187 different variants of our benchmarks. To conduct a coarse-grained analysis of optimization behavior, we built four variants per benchmark per compiler at optimization levels O0 through O3. While optimization level definitions vary between compilers, they are consistent for `GCC` and `Clang`. Code produced at

level O0 is almost entirely **unoptimized**. Specifying level O1 enables entry-level optimizations that reduce code size and execution time with minimal incremental compile time. At level O2, optimizations that improve execution speed without increasing code size are enabled. This increases compile time, but further reduces the binary's size and execution time over level O1. Specifying level O3 enables almost all optimizations and generally produces the fastest binaries, but they are generally larger and require more compile time versus level O2.

To conduct our fine-grained analysis, we built variants that isolate the behavior of individual optimizations. 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 at level O2. To generate a variant that isolates this behavior, we configured `GCC` to perform all level O1 optimizations and the store merging optimization. This is necessary to obtain realistic variants because specifying a particular optimization level also includes the optimizations performed at lower levels. Additionally, many optimizations must be run after other optimizations have already made a pass on the code to be effective. We have made the complete set of our study binaries publicly available to support further research in this important area (see Section 9). This repository includes a full list of the individual optimizations we generated for this study [Brown 2021a].

3.2 Variant Analysis

To determine how each optimization level or individual optimization impacts CRA gadget sets, we analyzed variants against an appropriate baseline using GSA [Brown and Pande 2019]. For coarse-grained analysis, we compared benchmark variants produced at optimization levels O1, O2, and O3 to the benchmark's unoptimized variant (i.e., 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 benchmark variants that isolate `GCC`'s store merging optimization to their respective O1 variants.

To isolate and clearly identify the effects of optimizations on our benchmarks, we do not include the benchmark's dynamically linked libraries in our analysis. In practice, gadgets in library code can be used by the attacker as they are mapped into the program's memory space at run-time. However, the library-based gadgets available depend on a build processes separate from our benchmark's; thus we consider them to be confounding variables in this study. Further, utilizing library-based gadgets is difficult in practice due to defensive techniques such as ASLR [PaX 2020], BlankIt [Porter et al. 2020], and piece-wise compilation/loading [Quach et al. 2018] that have been shown to be highly effective at neutralizing library-based gadgets.

For the baseline and each variant, GSA uses ROPgadget [Salwan 2020] to scan the binary and generate a complete catalog of its gadgets, including multi-branch gadgets. Next, GSA filters out duplicates and gadgets that are unusable (e.g., contain privileged instructions, do not generate attacker-controllable values, etc.). We refer to the gadgets that remain after filtering as **useful gadgets**. The useful gadgets are then analyzed to determine what functionality they provide, if they can be used for a special purpose, and their suitability for use in a gadget chain. This information is used to calculate four security-oriented metrics via baseline-variant gadget set comparison that measure changes caused by transformations: one of which is quantitative and three of which are qualitative [Brown and Pande 2019; Follner et al. 2016a; Homescu et al. 2012].

3.3 Gadget Set Metrics

Gadget Introduction Rate is a quantitative metric that measures how the composition of a gadget set changes following optimization. It is calculated as the percentage of gadgets in a variant binary that are not present in the baseline binary. Gadget addresses are not considered for this

metric, meaning gadgets that are not semantically altered but are relocated to a new address after optimization are not considered introduced. It is important to note that positive introduction rates do not necessarily indicate an increase in the number of unique gadgets in a set, since optimizations frequently introduce new gadgets by transforming an existing gadget.

Functional Gadget Set Expressivity is a qualitative measure of the computational tasks that can be performed with a particular set of gadgets (i.e., the set's expressive power). GSA measures functional gadget set expressivity with respect to three **levels** (listed in increasing order of overall expressive power): practical ROP exploits, ASLR-proof practical ROP exploits, and Turing-completeness. To achieve a particular level of expressivity (i.e., to be fully expressive at that level), a gadget set must contain at least one gadget that satisfies each of the level's required computational classes. The above levels require a total of 11, 35, and 17 computational classes, respectively. For example, to achieve Turing-completeness a gadget set must contain at least one gadget that adds attacker-controlled values, one that can store an attacker-controlled value to memory, one that can be used to implement a conditional branch, etc.

GSA calculates functional gadget set expressivity by scanning each ROP gadget's first instruction to determine if it implements a required computational class for any level. After the scan, the ROP gadget set's final expressivity measure is the total number of classes satisfied for each level. These expressivity measures for the baseline and variant binaries are compared on a per-level basis to determine changes in expressive power. If an optimization increases the number of satisfied computational classes with respect to one or more levels, this is considered an undesirable result.

Functional Gadget Set Quality is a qualitative measure of the overall utility of functional gadgets for constructing exploit chains. GSA assigns each gadget an initial score of 0, and then scans its intermediate instructions for side constraints (e.g., stack pointer manipulations, memory operations, conditional branches) that they impose on exploit construction. For each side constraint, the gadget's score is increased by a value proportional to the difficulty of accounting for the side constraint with another gadget in a chain [Brown 2021b]. Thus, the difficulty of using a gadget in a chain increases with its quality score. Finally, GSA computes the average quality score for the baseline and variant gadget sets and compares them. If an optimization decreases the resulting gadget set's average quality score, this is considered an undesirable result.

As an example, consider the the ROP gadget `{add eax, ebx; ret;}`. It contains no intermediate instructions and thus has no side constraints giving it a score of 0. By contrast, the ROP gadget: `{sub rsi, rcx; xor rax, rax; mov qword ptr [rdx], rsi; ret;}` has two intermediate instructions that constrain exploit construction. The first intermediate instruction (`xor rax, rax;`) will overwrite the value of the `rax` register if used in a chain, limiting the attacker's set of controllable registers. The second intermediate instruction (`mov qword ptr [rdx], rsi;`) performs a write to a memory location held in the `rdx` register, which may or may not represent a valid memory location in a hijacked execution. This gadget's score is increased by 1.0 and 2.0 for these intermediate instructions respectively, for a total score of 3.0.

Special Purpose Gadget Availability is a qualitative measure of the types of exploit patterns that can be employed with a gadget set. It is calculated by scanning the baseline and variant gadget sets to identify which special purpose gadget types are available, defined as the presence of at least one gadget of a particular type. GSA identifies ten different kinds of special purpose gadgets: syscall gadgets, four different types of JOP-specific gadgets [Checkoway et al. 2010], and five different types of COP-specific gadgets [Sadeghi et al. 2018]. If an optimization introduces types of special purpose gadgets that were not previously available, this is considered an undesirable result. Conversely, it is desirable for all special purpose gadgets of a particular type to be removed.

4 STUDY RESULTS

4.1 Optimization Induced Gadget Introduction

To answer our first motivating question {*To what degree do compiler optimizations introduce new CRA gadgets into optimized code?*}, we analyzed the gadget sets in our benchmark variants to determine the rate at which optimizations introduce new gadgets. Our results are shown in Table 3 and indicate that the collective effects of optimizations at levels O1 through O3 have a large impact on gadget sets. In all cases, more than two-thirds of useful gadgets in optimized binaries were newly introduced. We observed these effects for both compilers and all optimization levels. Excluding the smallest benchmark programs (i.e., `429.mcf` and `470.lbm`) the minimum observed introduction rate was 84%. Individual optimizations also introduce gadgets at a high rate. Across `GCC` single optimization variants, we observed an average introduction rate of 25.2%. For `Clang` single-optimization variants, we observed a much higher average rate of 82.5%.

With respect to total gadget counts, `GCC` O1 and O2 variants generally contained a similar number or fewer gadgets than their unoptimized counterparts. However, 60% of `GCC` O3 variants contained a significantly higher number of gadgets (i.e., >10% increase). This is not surprising, as optimizations that perform code size for speedup trade-offs are included at the O3 level in both compilers. Such optimizations create path-optimized copies of code segments, which are likely to introduce new unique gadgets when the copied segments contain GPIs. Our results were different for `Clang`, however. Significant increases in the total number of useful gadgets were observed for several optimization levels in all benchmarks except `httpd`, `456.hmmer`, and `458.sjeng`.

Our results clearly indicate that compiler optimizations have a significant impact on the quantity and composition of the code reuse gadgets present in their resulting binaries. While this quantitative analysis is insufficient to draw conclusions about the impact of optimizations on security [Brown and Pande 2019], the high rates of introduction observed suggest that significant impacts to gadget set quality are likely. This motivates our qualitative analysis in the following sections.

4.2 Coarse-Grained Gadget Set Impacts

To answer our second motivating question {*To what degree do compiler optimizations negatively impact the security of optimized code with respect to CRA gadget sets?*}, we conducted a coarse-grained analysis of optimization impacts on gadget sets with respect to our three qualitative metrics.

4.2.1 Functional Gadget Set Expressivity. Table 4 contains the functional gadget set expressivity data for our O0 - O3 variants. This data indicates that optimization overwhelmingly results in negative functional gadget set expressivity impacts. We observed an increase in expressive power for at least one expressivity level in 87% (91 of 105) of optimized variants that were not already fully expressive at level O0. This includes **all** `Clang` variants and 71% of `GCC` variants. Further, these increases are frequently significant in magnitude and affect all levels of expressivity. In 37% (39 of 105) of variants that were not already fully expressive, newly introduced gadgets satisfied 25% or more of the total computational classes required for at least one expressivity level. In 39% (41 of 105) of variants that were not already fully expressive, optimization increased expressivity across all three levels.

4.2.2 Functional Gadget Set Quality. Table 5 contains the functional gadget set quality data for our O0 - O3 variants. This 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 97% (116 of 120) of variants, optimization increased the number of useful gadgets and/or decreased the gadget set's average quality score. We observed significant negative impacts in 74% (89 of 120) of variants, defined as a >10% increase in the number of useful gadgets and/or a decrease in average

Table 3. Total Quality Gadgets and Introduction Rate

Benchmark	GCC				Clang			
	O0	O1	O2	O3	O0	O1	O2	O3
Bftpd	387	385 (90%)	474 (93%)	455 (92%)	335	451 (93%)	462 (94%)	449 (94%)
libcUrl	5289	5448 (97%)	6414 (98%)	6967 (98%)	4509	6058 (96%)	6506 (97%)	6825 (97%)
git	19472	12696 (96%)	13537 (97%)	13228 (97%)	13198	19177 (98%)	12266 (98%)	12362 (98%)
gzip	551	495 (90%)	537 (89%)	582 (91%)	350	495 (93%)	343 (90%)	442 (91%)
httpd	4640	4363 (93%)	4879 (93%)	5392 (93%)	4229	4368 (92%)	4432 (92%)	4478 (92%)
liblmdb	448	561 (95%)	572 (96%)	539 (95%)	421	585 (96%)	544 (96%)	559 (96%)
libsqlite	6393	5925 (95%)	6665 (96%)	6611 (96%)	5561	7863 (96%)	6209 (96%)	7007 (97%)
401.bzip2	307	301 (91%)	319 (92%)	410 (94%)	294	374 (93%)	341 (91%)	318 (90%)
403.gcc	16142	13512 (96%)	15139 (97%)	16464 (97%)	9854	14588 (97%)	13120 (97%)	13588 (97%)
429.mcf	113	151 (83%)	134 (80%)	136 (80%)	113	128 (75%)	155 (79%)	143 (78%)
433.milc	682	675 (92%)	699 (92%)	939 (94%)	519	608 (95%)	698 (95%)	739 (96%)
444.namd	770	559 (93%)	472 (91%)	570 (92%)	340	737 (95%)	875 (97%)	954 (96%)
445.gobmk	5114	5169 (84%)	5562 (85%)	6006 (86%)	4146	5596 (85%)	5296 (85%)	5400 (85%)
453.povray	8542	6021 (94%)	6932 (94%)	7278 (95%)	4475	5928 (95%)	5728 (95%)	5709 (94%)
456.hmmer	2023	1942 (96%)	2013 (96%)	2241 (96%)	1568	1530 (96%)	1568 (96%)	1585 (95%)
458.sjeng	599	697 (94%)	722 (95%)	847 (94%)	508	535 (95%)	524 (94%)	583 (95%)
462.libquantum	339	290 (90%)	320 (91%)	349 (92%)	220	304 (91%)	310 (91%)	316 (91%)
470.lbm	88	117 (74%)	92 (69%)	97 (70%)	71	111 (78%)	108 (77%)	111 (78%)
471.omnetpp	9379	8844 (96%)	8620 (97%)	8788 (97%)	8146	10054 (97%)	9125 (97%)	9162 (97%)
482.sphinx3	1033	1058 (96%)	1161 (96%)	1476 (96%)	860	1000 (95%)	1150 (96%)	1246 (96%)

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.

quality of 0.3 or greater. It is worth noting that significant negative impacts occur for all but four `Clang` variants and occur with less frequency in O1 and O2 `GCC` variants.

4.2.3 Special Purpose Gadget Availability. Table 6 contains the special purpose gadget availability data for our O0 - O3 variants. We observed overall negative impacts on special purpose gadget availability in 22% (26 of 120) of total variants and overall positive impacts in 28% (34 of 120) of total variants. We did not observe overall changes in the other half of total variants, due in part to the relatively small population of special purpose gadgets in our benchmarks and the calculation method 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 types of gadgets present in our variants revealed that optimizations introduced at least one new type of special purpose gadget category that was not originally present in 33% (39 of 120) of variants.

The most frequently introduced special purpose gadgets were syscall gadgets, JOP dispatcher gadgets, and COP dispatcher gadgets. Syscall gadgets are particularly dangerous and versatile special purpose gadgets that can be used in all exploit patterns for performing sensitive actions such as executing programs or opening shells. The introduction of these critical special purpose gadgets enables relatively simple and powerful exploits such as the one depicted in Figure 1. Dispatcher gadgets are similarly indispensable. In JOP and COP exploit patterns, the attacker cannot rely on the stack semantics of the `ret` GPI to chain gadgets together; instead they primarily rely on dispatcher gadgets to maintain control flow. While the attacker can use other special purpose gadgets (e.g., JOP and COP trampolines) to chain functional gadgets, these methods are frail and easier to detect by defensive techniques [Kayaalp et al. 2012b; Yao et al. 2013].

Table 4. Coarse-Grained Functional Gadget Set Expressivity

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

Expressivity is expressed as a 3-tuple in which each integer indicates the number of satisfied computational classes with respect to practical ROP exploits, ASLR-proof practical ROP exploits, and Turing-completeness, in that order. For example, a value of 5/5/5 in an O0 column indicates that the unoptimized binary's gadget set contains gadgets that perform five computational tasks required for each expressivity level. Columns marked with the Δ symbol contain data in parentheses that indicates the difference between the unoptimized and optimized variants (i.e., O[1,2,3] - O0). Positive values (depicted in **bold** text) in these columns indicate an undesirable outcome: increased expressive power over the unoptimized binary. For example, a value of **(1/1/1)** in a Δ column indicates that optimization increased the number of computational tasks performed by the binary's gadget set by one with respect to each expressivity level.

4.2.4 Discussion. Our coarse-grained analysis indicates that the gadget sets available in optimized program variants are significantly more useful for constructing CRA exploits than the sets found in unoptimized variants. This conclusion is consistent with the general separation of concerns in optimizing compilers. Compiler front ends responsible for lowering (i.e., translating) source code to intermediate representation (IR) focus on capturing program semantics in IR, whereas the compiler middle end optimizes the IR in place with a focus on improving performance. Many optimizations achieve performance improvements by transforming the relatively simple IR produced by the front end into IR that is more computationally diverse and complex. Coupled with our prior observation that optimizations both introduce new gadgets at a high rate and increase the total number of gadgets (see Section 4.1), it follows that optimization behaviors that increase code diversity and complexity also increase the availability and utility of code reuse gadgets found within the optimized code.

Interestingly, our data indicates that negative security impacts do not follow a linear progression, do not show biases or co-relations to optimization levels, and are not significantly impacted by the input program. We observed significant negative impacts across all optimization levels, benchmarks programs, and 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**.

Table 5. Coarse-Grained Functional Gadget Set Quality

Benchmark	GCC				Clang			
	O0	ΔO1	ΔO2	ΔO3	O0	ΔO1	ΔO2	ΔO3
Bftpd	382/1.6	(**1**/0.3)	(**80**/0.2)	(**58**/0.1)	331/2.0	**(108/-0.4)**	**(118/-0.4)**	**(104/-0.3)**
libcUrl	5218/1.7	(**155**/0.1)	(**1096**/0)	(**1648**/0)	4448/1.7	(**1519**/0.1)	(**1954**/0)	(**2302**/0)
git	19137/1.9	(-6645/**-0.1**)	(-5823/**-0.1**)	(-6120/**-0.2**)	12849/2.1	**(5945/-0.3)**	(-770/**-0.3**)	(-674/**-0.3**)
gzip	545/1.9	(-55/**-0.2**)	(-11/**-0.3**)	**(30/-0.3)**	345/1.8	(147/**-0.1**)	(-7/**-0.4**)	**(90/-0.3)**
httpd	4534/1.8	(-241/**-0.1**)	**(272/-0.1)**	**(767/-0.1)**	4141/1.9	**(136/-0.2)**	**(206/-0.2)**	**(245/-0.2)**
liblmdb	446/1.9	**(110/-0.3)**	**(125/-0.3)**	**(91/-0.3)**	419/1.8	**(163/-0.2)**	**(124/-0.1)**	**(136/-0.1)**
libsqlite	6294/1.8	(-483/**-0.1**)	(**243**/0)	**(218/-0.1)**	5472/1.7	(**2245**/0)	(**635**/0)	(**1421/-0.1**)
401.bzip2	306/1.7	(-6/**-0.1**)	(**10**/0)	**(100/-0.1)**	290/2.0	**(83/-0.4)**	**(49/-0.5)**	**(27/-0.5)**
403.gcc	15899/2.0	(-2524/**-0.3**)	(-989/**-0.2**)	**(337/-0.3)**	9768/2.0	**(4638/-0.3)**	**(3184/-0.3)**	**(3667/-0.3)**
429.mcf	112/1.7	**(35/-0.3)**	**(20/-0.3)**	**(23/-0.1)**	112/1.5	(**15**/0)	(**41**/0.2)	(**29**/0.1)
433.milc	662/1.6	(**7**/0.3)	(**32**/0.1)	(**264**/0.1)	511/1.9	**(96/-0.4)**	**(181/-0.4)**	**(226/-0.4)**
444.namd	752/2.1	(-205/**-0.2**)	(-287/**-0.5**)	(-191/**-0.5**)	339/1.6	(**392**/0.2)	**(523/-0.1)**	(**596**/0)
445.gobmk	4938/1.8	(**72**/0.1)	(**459**/0.1)	(**858**/0.1)	3976/2.0	**(1412/-0.1)**	**(1119/-0.2)**	**(1227/-0.1)**
453.povray	8273/1.6	(-2348/**0.2**)	(-1522/**0.1**)	(-1149/**0.1**)	4407/1.7	**(1414/-0.1)**	**(1220/-0.2)**	**(1198/-0.1)**
456.hmmer	1972/2.0	(-59/**-0.3**)	(0/**-0.3**)	**(237/-0.3)**	1544/2.1	(-31/**-0.5**)	**(12/-0.5)**	**(15/-0.5)**
458.sjeng	588/1.5	(**104**/0.4)	(**128**/0.4)	(**251**/0.3)	501/2.0	**(31/-0.4)**	**(19/-0.1)**	**(79/-0.1)**
462.libquantum	331/1.9	(-42/**-0.5**)	(-13/**-0.5**)	**(15/-0.5)**	217/1.8	**(85/-0.2)**	**(90/-0.2)**	**(98/-0.1)**
470.lbm	86/1.4	**(30/-0.1)**	(**4**/0)	**(10/-0.1)**	69/1.7	**(39/-0.3)**	**(38/-0.2)**	**(40/-0.2)**
471.omnetpp	9151/2.0	(-511/0)	(-875/**-0.1**)	(-682/**-0.1**)	8000/2.1	**(1734/-0.2)**	**(868/-0.1)**	**(930/-0.1)**
482.sphinx3	1018/1.8	(**34**/0)	**(133/-0.1)**	**(448/-0.1)**	852/1.7	(**135**/0.1)	(**287**/0)	**(375/-0.1)**

Gadget set quality is expressed as a 2-tuple in which the first integer indicates the number of useful functional gadgets in the variant and the second value indicates their average quality score. For example, a value of 1000/1.0 in an O0 column indicates that the unoptimized binary's gadget set contains 1000 useful gadgets and the average quality score for the set is 1.0. Columns marked with the Δ symbol contain data in parentheses that indicates the difference between the unoptimized and optimized variants (i.e., O[1,2,3] - O0). Positive values (depicted in **bold** text) for the count indicates an undesirable effect: an increase in the number of the useful gadgets. The reverse is true for the second value; negative values (depicted in **bold** text) here indicate a decrease in the average difficulty required to chain the gadgets in the variant. For example, a value of **(100/-0.5)** in a Δ column indicates that optimization increased the number of useful gadgets by 100 and decreased the average gadget set quality score by 0.5 (indicating there are fewer average side constraints per gadget).

4.3 Fine-Grained Gadget Set Impacts

To answer our third motivating question {*Which specific optimization behaviors cause negative security impacts?*}, we conducted a fine-grained analysis of gadget set impacts across 1,027 single optimization variants of our benchmarks. Using the coarse-grained variants produced at levels O0, O1, and O2 as baselines, we analyzed our single optimization variants with GSA to isolate their effects. Similarly to our coarse-grained analysis, we observed that our fine-grained variants exhibited a variety of negative security impacts on gadget sets. Increases in gadget set expressivity, gadget set quality, and special purpose gadget availability were commonly observed in variants across all benchmarks, isolated optimizations, and compilers; however, they were observed with more variability than in our coarse-grained variants.

In the majority of single optimization variants, negative impacts were relatively small in magnitude, manifesting as gadget set expressivity increases of one class, an increase of less than 5% in the number of useful gadgets, or the introduction of one type of special purpose gadget.[1] We also observed small magnitude impacts across our single optimization variants that were positive, albeit at a lower incidence rate than negative impacts. Our prior work [Brown and Pande 2019]

[1]Syscall, JOP data loader, and COP intra-stack pivot gadgets were the most frequently introduced special purpose gadgets.

Table 6. Coarse-Grained Special Purpose Gadget Availability

Benchmark	GCC				Clang			
	O0	**ΔO1**	**ΔO2**	**ΔO3**	**O0**	**ΔO1**	**Δ O2**	**Δ O3**
Bftpd	4	(-3)	(-1)	(0)	2	**(1)**	**(2)**	**(1)**
libcUrl	7	(0)	(-2)	(0)	7	(0)	(0)	(-1)
git	7	(0)	(0)	(0)	7	(0)	(0)	(0)
gzip	3	(-1)	(0)	**(2)**	4	(-2)	(0)	(0)
httpd	7	(0)	(0)	(0)	7	(0)	(-1)	(0)
liblmdb	2	(0)	(-1)	(0)	2	**(1)**	(-1)	**(1)**
libsqlite	7	(0)	(0)	(0)	7	(0)	(-1)	(0)
401.bzip2	1	(0)	**(2)**	**(1)**	3	(-2)	(-1)	(-2)
403.gcc	7	(0)	(0)	(0)	7	(0)	(0)	(0)
429.mcf	1	**(1)**	**(1)**	(0)	1	(0)	**(1)**	(0)
433.milc	5	(-2)	(-2)	(0)	4	(-3)	(-2)	(-2)
444.namd	3	**(1)**	(0)	**(1)**	1	**(3)**	**(2)**	**(2)**
445.gobmk	6	(0)	**(1)**	**(1)**	7	(-1)	(0)	(0)
453.povray	7	(0)	(0)	(0)	7	(-1)	(-1)	(0)
456.hmmer	6	(-1)	(-1)	(-1)	4	**(1)**	(0)	**(2)**
458.sjeng	3	**(2)**	(0)	**(1)**	3	(-1)	(0)	(0)
462.libquantum	3	(-2)	(-1)	(0)	3	(-1)	(-1)	(-2)
470.lbm	2	(-1)	(0)	(-1)	1	**(1)**	(0)	(0)
471.omnetpp	7	(-1)	(0)	(0)	7	(0)	(0)	(0)
482.sphinx3	4	(0)	(0)	**(1)**	4	(-1)	**(1)**	**(1)**

Special purpose gadget availability is expressed as a single integer indicating the number of special purpose gadget categories available in the gadget set. A category is considered available if at least one special purpose gadget of that type is present. For example, a value of 5 in an O0 column indicates that the unoptimized binary's gadget set has 5 types of special purpose gadget types available. In columns marked with the Δ symbol, the integer in parentheses indicates the difference between the unoptimized and the optimized variants (i.e., O[1,2,3] - O0). Positive values (depicted in **bold** text) in these columns indicate an undesirable result: an increase in the number of special purpose gadget types available in the optimized variant's gadget set. For example, a value of **(2)** in a Δ column indicates that optimization increased the number of special purpose gadget types available by 2 categories.

suggests that transformation-induced gadget introduction is responsible for the majority of this observed "background noise". We detail our investigation of this potential root cause in Section 5.2.

4.3.1 Outlier Detection. Within this "background noise", we observed a number of individual optimizations with negative impacts that were significantly larger in magnitude. To separate this "signal" for deeper analysis, we performed outlier detection across our single optimization variant data. We define outliers as variants with metric result changes from the baseline variant greater than 1.5 times the standard deviation from the mean metric result change on a per benchmark, per metric basis. We then combined the total list of identified outliers across all benchmarks and metrics into a histogram to identify which optimizations produced outliers most frequently. We identified that `GCC`'s **Interprocedural Constant Propagation (IPA CP) Function Cloning** (`-fipa-cp-clone`) optimization frequently produced outlier impacts with respect to functional gadget set expressivity. We also identified four optimizations that frequently produced outlier impacts with respect to special purpose gadget availability: `GCC`'s **Jump-Following Common Subexpression Elimination (CSE)** (`-fcse-follow-jumps`), **Peephole** (`-fpeephole2`), **Omit Frame Pointer** (`-fomit-frame-pointer`), and `Clang`'s **Tail Call Elimination (TCE)** (`--tail-call-elim`).

4.3.2 Outlier Analysis. Table 7 contains a subset of our fine-grained analysis data for the optimizations that frequently produce outliers. `GCC`'s IPA CP function cloning optimization was observed to cause negative impacts on functional gadget set expressivity in 55% of variants, with outlier

Table 7. Single Optimization Variants with Frequent Outliers

Benchmark	IPA CP Clone*		Jump Follow CSE		TCE		Peephole		Omit Frame Ptr.	
	O2	ΔOpt	O1	ΔOpt	O0	ΔOpt	O1	ΔOpt	O0	ΔOpt
Bftpd	7/21/9	(0/-1/**3**)	1	(0)	2	**(1)**	1	(0)	4	**(1)**
libcUrl	9/34/16	(0/0/0)	7	(0)	7	(-1)	7	(-1)	7	(0)
git	10/34/17	(0/0/0)	7	(0)	7	(0)	7	(0)	7	(0)
gzip	7/28/12	(-1/-2/**1**)	2	**(2)**	4	(-1)	2	(0)	3	(0)
httpd	10/34/17	(0/0/-1)	7	(0)	7	(-1)	7	(0)	7	(0)
liblmdb	7/22/8	**(1/2/1)**	2	(0)	2	**(1)**	2	(0)	2	(0)
libsqlite	11/35/16	(-1/-1/0)	7	(0)	7	(0)	7	(0)	7	(0)
401.bzip2	8/28/9	(0/**1**/0)	1	(0)	3	(-1)	1	**(1)**	1	**(2)**
403.gcc	10/34/17	(0/**1**/0)	7	(0)	7	(0)	7	(0)	7	(0)
429.mcf	7/21/6	(0/0/0)	2	(0)	1	**(1)**	2	(-1)	1	(0)
433.milc	9/30/15	(-1/-1/-2)	3	**(1)**	4	(-1)	3	**(2)**	5	(-1)
444.namd	8/21/8	(0/0/0)	4	**(2)**	1	(0)	4	**(1)**	3	**(1)**
445.gobmk	9/33/17	(**2/2**/-1)	6	**(1)**	7	(0)	6	**(1)**	6	**(1)**
453.povray	9/33/16	**(1/1/1)**	7	(0)	7	(-1)	7	(0)	7	(0)
456.hmmer	9/30/13	(-1/**1/1**)	5	**(1)**	4	**(2)**	5	**(1)**	6	(0)
458.sjeng	8/26/11	(0/**3/2**)	5	(-3)	3	(0)	5	(-3)	3	**(1)**
462.libquantum	7/26/6	(0/0/0)	1	**(2)**	3	(-1)	1	**(1)**	3	**(1)**
470.lbm	6/21/8	(0/0/0)	1	(0)	1	(0)	1	**(1)**	2	(0)
471.omnetpp	10/35/16	(0/-1/**1**)	6	**(1)**	7	(0)	6	**(1)**	7	(-1)
482.sphinx3	8/28/11	(0/**1**/0)	4	(0)	4	(-1)	4	**(1)**	4	**(1)**

* Functional gadget set expressivity data is displayed for this variant. For all others, special purpose gadget availability data is displayed.

increases to at least one expressivity level occurring in 20% of total variants. `GCC`'s jump-following CSE, peephole, and omit frame pointer optimizations frequently introduced new types of special purpose gadgets, most commonly syscall and COP intra-stack pivot gadgets. We observed new categories of special purpose gadgets introduced by these optimizations in 35%, 45%, and 35% of total variants, respectively. In contrast, `Clang`'s TCE optimization had a higher number of positive than negative impacts on special purpose gadget availability, eliminating one or more types of special purpose gadgets in 40% of total variants. New types of special purpose gadgets were introduced in 20% of variants. Deeper analysis of this unexpected result revealed that this optimization frequently eliminates COP-specific special purpose gadgets at the cost of introducing new JOP data loader gadgets at a high rate. The ultimate effect of the optimization (i.e., positive or negative) largely depends on the types of special purpose gadgets present in the baseline variant. For example, baseline variants without JOP data loader or COP-specific gadget types are likely to suffer the undesirable impact of newly introduced JOP data loader gadgets without a corresponding benefit of COP-specific gadget elimination.

4.3.3 Discussion. We draw two high-level conclusions from our fine-grained analysis. First, nearly all optimizations have a small but measurable impact on the composition of the resulting gadget set. The majority of these impacts are negative, though positive impacts were also observed at lower frequency. This conclusion is consistent with the findings of our coarse-grained analysis; it suggests that the high levels of negative impacts observed at predefined optimization levels are the conglomeration of smaller magnitude impacts made by many individual optimizations rather than a small group of troublesome optimizations with high magnitude impacts. Second, we conclude that a relatively small number of optimizations have impacts that are clearly discernible from the "background noise". This conclusion is promising as it suggests that there are relatively few undesirable optimization behaviors that are negatively impacting gadget sets.

5 ROOT CAUSES AND MITIGATION STRATEGIES

To answer our fourth motivating question {*What are the root causes of these negative security impacts?*}, we performed differential binary analysis of our single optimization variants and their respective baseline variants to determine the underlying causes for the negative impacts we observed. While we focused our efforts on the optimizations identified in Section 4.3, we analyzed a representative sample of other optimizations in order to confirm our hypothesis that transformation-induced gadget introduction is the underlying cause of the small magnitude gadget set impacts we observed. We used IDA Pro and BinDiff [Hex-Rays 2020; zynamics 2020] to identify and inspect the after-effects of individual optimizations. BinDiff uses heuristics to match functions between different versions of the same program and provides a visual before-and-after comparison of the control flow graphs (CFGs) for each version. To ensure precise CFG recovery during disassembly, we built our program variants with full debug information. Through differential analysis, we identified three root causes of the negative gadget set impacts we observed, which we detail in the following subsections. To address our fifth motivating question {*How can these negative security impacts be mitigated?*}, we also propose potential mitigation strategies for each root cause in this section.

5.1 GPI Duplication

Gadget search algorithms scan binaries for byte sequences that encode GPIs, including both byte sequences intentionally inserted by the compiler (e.g., return instructions terminating functions) as well as sequences that are unintentionally encoded in data, memory addresses, 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 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 sequence exceeds some useful threshold (e.g., 10 bytes). This process identifies all intended and unintended gadgets associated with each GPI encoding at or below the threshold, which collectively forms the binary's gadget set. When an optimization duplicates a GPI, a new set of intended and unintended gadgets associated with the GPI is created. Due to the density of the `x86-64` instruction set and its support for variable length instructions, this set is likely to contain new, unique, and useful gadgets.

A number of compiler optimizations typically employed at the O3 level selectively duplicate code in order to improve performance at the cost of increased code size. Such behavior duplicates GPIs at a high rate, however GPI duplication also occurs at lower optimization levels. For example, both `GCC` and `Clang` perform function inlining at level O2 and above, which replaces function call sites with the body of the called function. This enables intra-procedural optimizations across caller and callee code, but also duplicates GPIs present in the inlined function's body. This behavior partially explains 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 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.[2] Due to `GCC`'s code generation conventions, the pointer restore instruction (i.e., `pop rbp;`) is typically the lone instruction executed prior to a function's `retn` instruction. When multiple code paths converge at the end of a function, eliminating the pointer restore instruction allows the `retn` instruction to be hoisted to the end of each converging path. While this technique slightly reduces code size and

[2]We did not observe this behavior in Clang due to differences in how it generates function epilogues.

execution time per path by replacing a 5-byte `jmp` instruction with a copy of the single byte `retn` instruction it targets, it duplicates the GPI one or more times.

5.1.1 Concrete Example. An example of this phenomenon in `httpd` is shown in Figure 2. Before optimization, the single GPI produces two sets of gadgets: one consisting of intended and unintended gadgets sourced from the instruction sequence `{mov rax, [ap_module_short_names]; ...; pop rbp; retn;}` and one consisting of mostly intended gadgets sourced from the multi-branch instruction sequence `{mov eax, 0; jmp 0x6D0BA; pop rbp; retn;}`. The second set contains few, if any, unintended gadgets because the `jmp` in this multi-branch sequence must be executed as encoded to reach the GPI. After the optimization eliminates the `pop rbp;` instruction and duplicates the GPI, the second sequence of instructions is no longer constrained to intended execution as the `jmp` instruction has been eliminated (i.e, it is no longer a multi-branch sequence). The net result is a drastic increase in the total number of unintended gadgets sourced from the two GPIs, as well as significant changes to the composition of the intended gadgets. This behavior is not limited to `GCC`'s omit frame pointer optimization; it also occurs with `GCC`'s shrink wrap and tree switch conversion optimizations. Additionally, this behavior can occur with non-`retn` GPIs. Although rare, we observed instances in which indirect jump and call GPIs were duplicated by `GCC`'s sibling call optimizations.

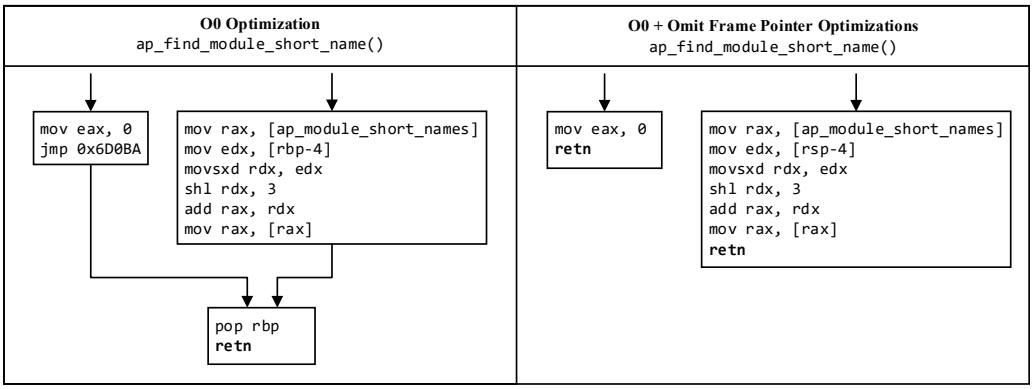

Fig. 2. GPI Duplication caused by GCC's Frame Pointer Omission Optimization

5.1.2 Mitigation Strategy. The negative effects of GPI-duplicating optimizations occur as a result of aggressive secondary optimization (i.e., GPI hoisting). One potential mitigation strategy is to patch compiler optimization passes to avoid this undesirable behavior, eliminating negative gadget set impacts while still obtaining the primary intended benefits of the original optimization. Since several distinct optimizations exhibit this behavior, we choose to evaluate this mitigation strategy in Section 6.1 with binary recompiler passes that intraprocedurally merge GPIs rather than patch multiple compiler passes and `GCC`'s code generation engine.

5.2 Transformation-Induced Gadget Introduction

Our prior work has shown that even relatively small software transformations introduce a significant number of new gadgets [Brown and Pande 2019]. Differential analysis of our single-optimization variants confirmed that seemingly innocuous and localized compiler optimizations are no exception. We refer to this phenomenon as transformation-induced gadget introduction. It is primarily driven

by changes that occur during optimization to program layout and the code preceding compiler-placed GPIs. It results in changes to both intended and unintended gadget populations and is the primary cause of the "background noise" we observed in the large majority of optimized variants in our study.

While there are several mechanisms by which transformation-induced gadget introduction occurs, the most frequent cause we observed was rooted in displacement/offset encodings in control-flow transfer instructions like direct jumps and function calls. Semantic optimizations can induce layout changes to programs, even for simple optimizations such as instruction re-ordering. The resulting changes to displacement/offset encodings can alter a previously benign encoding to one that contains a GPI, which introduces several new unintended gadgets.

5.2.1 Concrete Example. An example of this behavior in `libcUrl` is shown in Figure 3. Here, a conditional jump instruction with a short 1 byte displacement is converted into an equivalent conditional jump instruction with a near 4-byte displacement as a byproduct of optimization. This new displacement encodes the `retn` GPI (i.e., `0xC3`). This behavior is not limited to `retn` GPIs; we also observed two-byte syscall and COP intra-stack pivot special purpose gadgets introduced in this manner, although at a lower rate.

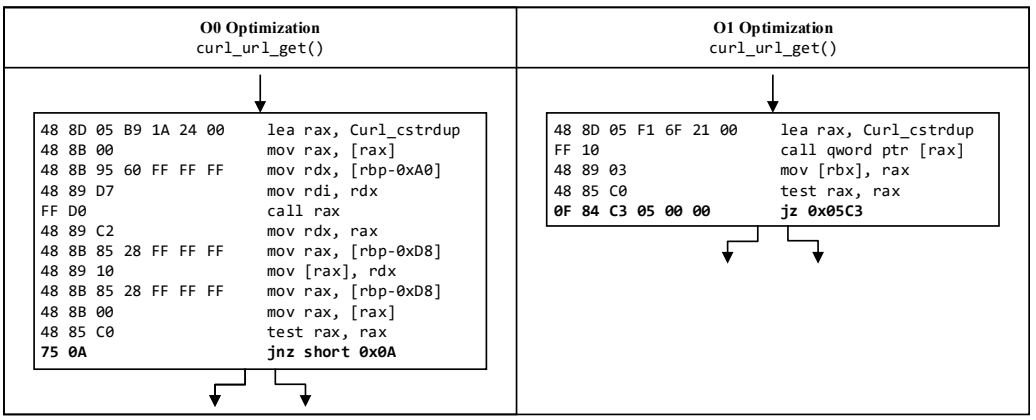

Fig. 3. Layout-Based Introduction of GPI and Unintended Gadgets

5.2.2 Mitigation Strategy. Given that this type of gadget introduction is endemic to the `x86-64` instruction set architecture, it is outside the control and concern of machine-independent optimizations. As such, it is not possible to patch them to detect and avoid these harmful effects. However, compiler back-ends responsible for producing the binary can be patched to correct problematic encodings after optimization is complete. Another viable mitigation strategy is post-production transformation passes to eliminate layout-based GPIs. An assembler injection-based solution, G-Free [Onarlioglu et al. 2010], has been proposed to address this problem, however it is limited to `retn` GPIs and suffers significant performance impacts (average 3.1% slowdown and 25.9% increase in code size) due to the lack of direct control over program layout. We propose a new approach to implement this strategy that employs transformations that exercise direct control over program layout. This strategy takes advantage of binary recompilation infrastructure to effectively eliminate unintended gadgets without suffering performance losses. We detail the implementation and evaluation of this mitigation strategy in Section 6.1.

5.3 Special Purpose Gadgets Explicitly Introduced by Optimizations

JOP and COP exploit patterns require special purpose gadgets to perform important non-functional tasks in exploit chains. JOP data loader gadgets pop the attacker's data from the stack into registers at the beginning of a JOP exploit chain. They consist of an indirect jump GPI preceded by a `popa` or multiple `pop` instructions. However, the utility of this instruction sequence is not limited to exploit programming. `Clang`'s tail call elimination (TCE) optimization identifies cases where a function's stack frame can be reused by a child function call occurring at its end. The optimization replaces the tail call and the following return instruction with a jump to the child function, eliminating stack frame set up operations. Indirect call GPIs eliminated in this manner are replaced with indirect jump GPIs, resulting in the elimination of call-ending gadgets and the introduction of jump-ending gadgets. 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. `Clang`'s code generation convention for function epilogues includes several `pop` instructions that restore the values of callee-saved registers (i.e., `RBX`, `RBP`, `R12`–`R15`) before returning to the calling function. When the function-terminating return instruction preceded by these `pop` instructions is replaced by an indirect jump GPI, JOP data loader gadgets are produced.

5.3.1 Concrete Example. An example of this behavior in `httpd` is shown in Figure 4. Prior to optimization, the code snippet contains two GPIs depicted in **bold** print. After TCE the first GPI, `call rax`, is transformed into a new GPI, `jmp rax`, which produces a JOP data loader gadget as well as functional JOP gadgets. Minimal changes are observed with respect to the other GPI, `retn`.

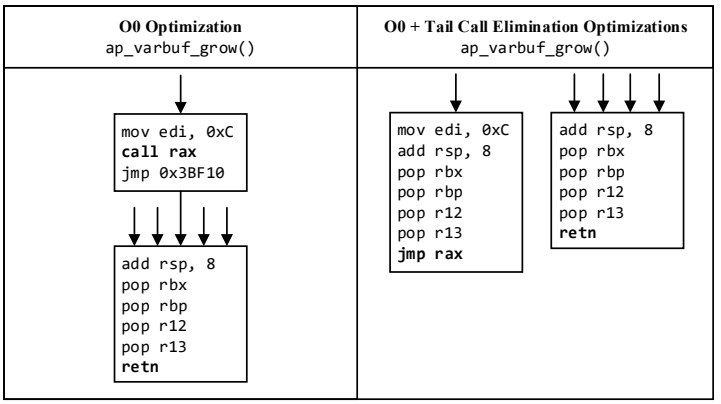

Fig. 4. Special Purpose Gadget Introduction caused by Clang's Tail Call Elimination Optimization

5.3.2 Mitigation Strategy. In this case, the sequence of instructions representing the JOP data loader gadget is intentionally placed by the compiler during optimization and cannot be avoided without forgoing its benefits. Disabling TCE during compilation is a straightforward mitigation strategy that has become common practice for other optimizations that introduce security weaknesses (e.g., dead store elimination) [Yang et al. 2017]. We evaluate the performance impacts of disabling TCE in Section 6.2. Since this behavior only occurs with indirect tail calls, `Clang`'s TCE pass could also be patched to optimize direct tail calls only. In either case, avoiding JOP data loader gadget introduction may not be worth forgoing TCE's performance benefits and is ultimately subject to risk-reward considerations that vary depending on the circumstances.

6 MITIGATION STRATEGY IMPLEMENTATION AND EVALUATION

To address our sixth motivating question {*What is the performance cost of implementing such mitigations?*}, we implemented and evaluated our proposed mitigation strategies. To mitigate GPI duplication and transformation-induced gadget introduction, we implemented five post-production binary recompiler passes that merge duplicated GPIs and eliminate layout-based unintended gadgets. To address intentional introduction of special purpose gadgets in `Clang`'s TCE optimization, we measure the performance impact of disabling it during compilation.

6.1 Post-Production Recompilation to Merge GPIs and Eliminate Unintended Gadgets

The purpose of our binary recompiler passes is to mitigate the undesirable optimization behaviors we observed without significantly impacting run-time performance or code size. We implemented two mitigation passes that address GPI duplication behaviors: one that merges `retn` GPIs and one that merges indirect `jmp` GPIs. We also implemented three mitigation passes that address layout-based gadgets introduced indirectly by optimizations: one that eliminates GPI encodings across instruction boundaries, one that eliminates GPI encodings in direct jump instruction displacements via short `nop` sleds, and one that eliminates GPI encodings in direct call displacements via function re-ordering.

As these two root cause behaviors are common to several different optimizations and compilers, our implementations target undesirable behaviors, not specific optimization algorithms. It is worth noting that these mitigation passes do not differentiate between GPIs introduced via optimization versus instances where the GPI is placed by the compiler during code generation. Since we expect the performance costs of mitigation to be negligible, we intentionally designed these mitigation passes to aggressively eliminate GPIs to achieve maximum benefit. To support further research in this important area, we have made the source code for our mitigation passes publicly available (see Section 9).

6.1.1 Implementation of GPI Merging Passes. We observe that the benefits of duplicating GPIs in the manner depicted in Figure 2 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 original optimization, it is possible to merge all occurrences of a particular GPI within a function to a single instance and retain the optimization's primary benefits. Our return merging algorithm operates by scanning each function to find all of its return-type instructions. If multiple return-type instructions are found, one is arbitrarily selected to be retained. All other return-type instructions are replaced with an unconditional jump to the retained instruction, effectively merging multiple return-type instructions into a single instance. Our indirect jump merging algorithm 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 function scan, indirect jump instructions found are placed into a map data structure in which each register is mapped to a list of indirect jump instructions targeting it. When the scan is complete, any indirect jump instructions targeting the same register are merged in the same manner as the return merging algorithm.

Both of our GPI merging passes preserve program semantics. The unconditional jump instructions inserted in place of eliminated GPIs do not alter program state other than to transfer control to the retained GPI. In both of our passes, GPIs replaced by an unconditional jump are identical to the retained GPI targeted by the inserted jump, resulting in no change to program semantics.

6.1.2 Implementation of Layout-Based Gadget Elimination Passes. In the case of layout-based GPIs introduced by transformations, such as the instance depicted in Figure 3, we observe that small non-semantic changes to the program's layout are sufficient to eliminate these GPIs. For problematic

layout encodings that are intraprocedural (i.e., instruction or basic block level), insertion of short 1 or 2 byte nop sleds can shift problematic encodings to benign ones with a negligible performance impact. We implemented two such transformations: instruction boundary widening and jump displacement sledding. For problematic layouts that are interprocedural (i.e., direct function calls) nop sleds must be much larger than 1-2 bytes in general to shift the displacement to a benign encoding due to function sizes, inter-function padding, etc. As such, this approach is not ideal as it swells code size and can have negative impacts on run-time performance. Instead, we implemented a program-wide transformation that reorders functions in the program layout in order to shift call displacements to benign encodings. This approach obviates the need for large nop sleds and thus avoids their negative performance impacts, though it must be employed carefully to avoid causing slowdowns due to compromised cache locality.

Our instruction boundary widening algorithm operates by scanning each pair of contiguous instructions in the program layout to identify instances where multi-byte GPIs are encoded by the last byte of the first instruction and the first byte of the second instruction. For example, the instruction sequence `{je 0xffcf; or PTR [rbx+0x4],0x2;}` is encoded as `{74 CD; 80 4B 04 02;}`. The byte sequence `CD 80` is contained within this sequence and encodes the syscall GPI `int 0x80`. When found, our algorithm inserts a single nop instruction to widen the boundary between the instructions (i.e., `{je 0xffcf; nop; or PTR [rbx+0x4],0x2}`), eliminating the GPI encoding.

Our jump displacement sledding algorithm first scans each function to identify problematic near or short jump displacement encodings such as the one depicted in Figure 3. For each function with problematic encodings, one encoding is selected at random to be shifted via insertion of a short (i.e., 1 or 2 bytes depending on the encoding) nop sled. Sleds are inserted before backward jumps and before a forward jump's target to shift the problematic displacement to a benign encoding. Since adjusting the function layout in this manner can affect other jump displacements within the function, this process (i.e., scan, randomly shift an encoding per function) must be conducted iteratively until each function no longer contains problematic displacements. In certain rare cases, the layout of two or more jumps may be interrelated such that correcting a problematic encoding for one jump causes an interrelated jump encoding to become problematic, and vice versa. To ensure our algorithm terminates and does not insert excessively long and ineffective sleds, our algorithm is capable of detecting repeated failures to reduce the number of problematic encodings for a given function. If detected, our algorithm excludes the function from further iterations.

Our function reordering algorithm operates by first scanning the program to identify problematic call displacement encodings, the functions they target, and the shift size in bytes required to make the encoding benign. The algorithm then chooses a function with problematic encodings at random and determines the largest shift size required to address all of that function's problematic encodings. Then, the function is moved forward or backward in the program layout by swapping its position with a neighboring function until it has moved the minimum number of bytes necessary to shift the encodings. By moving functions by the minimum distance possible, the algorithm avoids large changes to code locality. As was the case with jump displacement sledding, scanning and re-ordering must be performed iteratively since each adjustment to program layout potentially fixes or creates other problematic encodings. In certain rare cases, the layout of two or more functions may be interrelated such that correcting a problematic encoding for one call causes an interrelated call encoding to become problematic, and vice versa. To ensure termination, this algorithm detects repeated failures to reduce the global number of problematic encodings. If a sufficient number of consecutive failures is detected, our algorithm terminates having arrived at a local minimum.

All three of our layout-based gadget elimination passes preserve program semantics. In the case of instruction boundary widening and jump displacement sledding, our mitigations only alter the binary by inserting nop instructions, which by definition do not alter program behavior.

Additionally, our function reordering algorithm alters only the relative positioning of functions in the binary layout and does not insert, modify, or remove instructions. Since our passes are only compatible with position-independent code, there is no possibility of breaking position-dependent control-flow transfers when reordering functions.

6.1.3 Egalito Binary Recompiler. We developed our mitigation passes for Egalito [Williams-King et al. 2020], a binary recompiler that lifts position-independent ELF binaries to a layout-agnostic intermediate representation (EIR) that supports arbitrary transformations and recompilation 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 the vast majority of cases.

We selected Egalito as the engine for our mitigation passes for four reasons. First, it supports **machine-dependent** transformations that can readily address CRA gadgets, whereas existing compiler toolchains primarily support machine-independent optimization (e.g., LLVM [Lattner and Adve 2004]). Second, Egalito is compiler agnostic; a single implementation of our mitigation passes can be used on binaries regardless of the compiler used to produce them. Third, implementing our passes in Egalito allows our work to address legacy binaries, provided that they are PIC. Finally, Egalito provides a Union ELF mode in which the input binary and its linked libraries are combined into a single output ELF prior during re-compilation. This feature allows our mitigation passes to be readily applied to library code in a single operation and also removes unnecessary library code and their constituent gadgets as an added benefit. While not employed or evaluated in this paper, we identify exploring the benefits of Egalito's Union ELF mode as promising future work.

6.1.4 Evaluation of Gadget Set Impacts. To determine how effectively our mitigation passes reduce the availability and utility of CRA gadgets, we used Egalito to apply them to `GCC` and `Clang` O3 variants of our benchmark programs.[3] We then used GSA to measure the change in gadget set metrics after recompilation with our passes. Three of our benchmark programs were excluded from this evaluation because they were not compatible with Egalito (see Section 6.1.6). Our results are reported in Table 8.

Our passes were highly effective at reducing special purpose gadget diversity in recompiled binaries. They decreased the number of special purpose gadget categories available in 75% (24 of 32) of variants with at least one type of special purpose gadget present in the baseline O3 variant. In more than half of these cases (14 of 24), multiple categories of gadgets were eliminated. Further, we observed no instances where our mitigation passes increased the number of special purpose gadget types available. On average, our mitigation passes reduced the types of special purpose gadgets available by 34% (36.1% for `GCC` variants and 31.8% for `Clang` variants).

Our mitigation passes were also highly effective at reducing the number of useful gadgets available after recompilation. On average, our passes reduced the total number of useful gadgets by 31.8% (33% for `GCC` variants and 30.6% for `Clang` variants). We observed a maximum reduction rate of 51% and in no instances did our mitigation passes increase the number of useful gadgets. Additionally, we observed no change on average across all of our benchmarks to the average gadget set quality score. This indicates that our passes eliminate high-quality and low-quality gadgets at roughly the same rate.

Finally, our mitigation passes were also quite effective at reducing functional gadget set expressivity. Our passes reduced the overall expressivity of gadget sets in 78% (28 of 34) of total variants; however we did observe an overall increase in expressivity in four variants. Across all variants

[3]Several O3 program variants in our study were originally built as position-dependent code. They were rebuilt as PIC for this evaluation and thus metric values reported in this section may differ from those in Section 4.

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**	**ΔMP**	**O3**	**ΔMP**	**O3**	**ΔMP**
GCC Variants	Bftpd	8/21/8	(0/-3/1)	440 / 1.7	(-118 / 0.1)	4	(-2)
	libcUrl	9/33/16	(0/-2/-1)	6866 / 1.7	(-2009 / 0.1)	7	(-2)
	git	11/35/17	(-2/-1/0)	13017 / 1.7	(-4773 / 0)	7	(0)
	gzip	6/26/12	(2/3/1)	575 / 1.6	(-141 / 0)	5	(-1)
	httpd	9/33/17	(0/-1/-2)	5301 / 1.7	(-2717 / 0)	7	(-3)
	libsqlite	11/35/15	(-3/-2/0)	6512 / 1.7	(-2194 / 0.1)	7	(-1)
	401.bzip2	8/25/10	(0/2/-1)	403 / 1.6	(-101 / -0.1)	2	(-1)
	403.gcc	11/35/17	(-1/-1/0)	21636 / 1.6	(-10277 / 0.1)	7	(0)
	429.mcf	6/20/6	(-1/-1/0)	129 / 1.8	(-21 / 0.1)	1	(0)
	433.milc	8/28/12	(-1/-1/0)	891 / 1.7	(-272 / -0.1)	5	(-1)
	444.namd	8/30/12	(0/-6/-4)	512 / 1.6	(-112 / 0.2)	6	(-5)
	445.gobmk	11/35/17	(-2/-2/-2)	5262 / 1.8	(-2706 / -0.2)	7	(-3)
	456.hmmer	9/32/15	(0/0/-2)	2335 / 1.6	(-1125 / 0.1)	7	(-4)
	458.sjeng	10/29/13	(-2/-1/0)	863 / 1.5	(-289 / 0.2)	4	(-3)
	462.libquantum	7/25/9	(0/-2/-3)	323 / 1.4	(-75 / 0.2)	2	(-1)
	470.lbm	6/20/8	(0/0/-2)	107 / 1.5	(-15 / 0.1)	1	(0)
	482.sphinx3	9/31/13	(0/0/0)	1289 / 1.6	(-474 / 0)	5	(-4)
Clang Variants	Bftpd	8/21/8	(0/-3/1)	435 / 1.7	(-112 / 0.1)	3	(-1)
	libcUrl	9/32/16	(0/0/-1)	6750 / 1.7	(-1511 / 0)	6	(0)
	git	11/35/17	(-2/-1/0)	13846 / 1.7	(-5234 / 0)	7	(-1)
	gzip	7/26/8	(-1/-6/1)	417 / 1.6	(-92 / 0)	4	(-2)
	httpd	9/34/17	(0/-2/-1)	4500 / 1.6	(-1992 / 0)	7	(-3)
	libsqlite	11/34/16	(-2/-1/0)	6893 / 1.6	(-2179 / 0.1)	7	(-2)
	401.bzip2	7/26/9	(-1/-2/-1)	334 / 1.5	(-53 / 0.1)	2	(-1)
	403.gcc	11/35/17	(-2/-1/0)	18910 / 1.6	(-8205 / 0.2)	7	(0)
	429.mcf	6/15/6	(-1/-1/0)	140 / 1.8	(-23 / -0.1)	0	(0)
	433.milc	8/30/12	(-1/-3/-2)	710 / 1.6	(-135 / -0.1)	2	(0)
	444.namd	6/26/11	(0/1/1)	882 / 1.6	(-94 / 0)	2	(0)
	445.gobmk	10/34/17	(-1/-2/-3)	5162 / 1.8	(-2642 / -0.1)	7	(-1)
	456.hmmer	9/33/16	(0/0/-2)	2098 / 1.7	(-1022 / 0.1)	7	(-4)
	458.sjeng	10/33/15	(0/-3/-2)	705 / 1.8	(-211 / -0.1)	4	(-4)
	462.libquantum	7/26/11	(1/-1/-1)	339 / 1.7	(-73 / -0.2)	1	(-1)
	470.lbm	5/16/4	(0/0/0)	112 / 1.5	(-24 / 0.1)	0	(0)
	482.sphinx3	8/30/13	(0/1/1)	1210 / 1.6	(-370 / 0)	4	(-2)

and expressivity levels, our passes reduced expressivity by approximately one computational class on average. Of critical importance is our passes' performance on fully expressive variants. Of the 21 instances where a variant was fully expressive at some level, applying our passes successfully reduced 17 of these instances below the fully expressive threshold, a success rate of **81%**.

6.1.5 Evaluation of Performance Impacts. 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.[4] We recorded the total run time for each variant across three trials to determine its average performance, shown in Figures 5 and 6. We observed that binaries recompiled with our passes enabled saw an overall performance **improvement** of 0.2% on average, with a maximum observed slowdown of 3.3%. This performance improvement is not entirely unexpected, as Egalito's core recompilation process has been shown to speedup SPEC 2006 benchmarks by 1.7% on average [Williams-King et al. 2020]. We conclude that the impacts of our passes are

[4]We exclude our common Linux benchmarks from this analysis due to a lack of standardized performance tests.

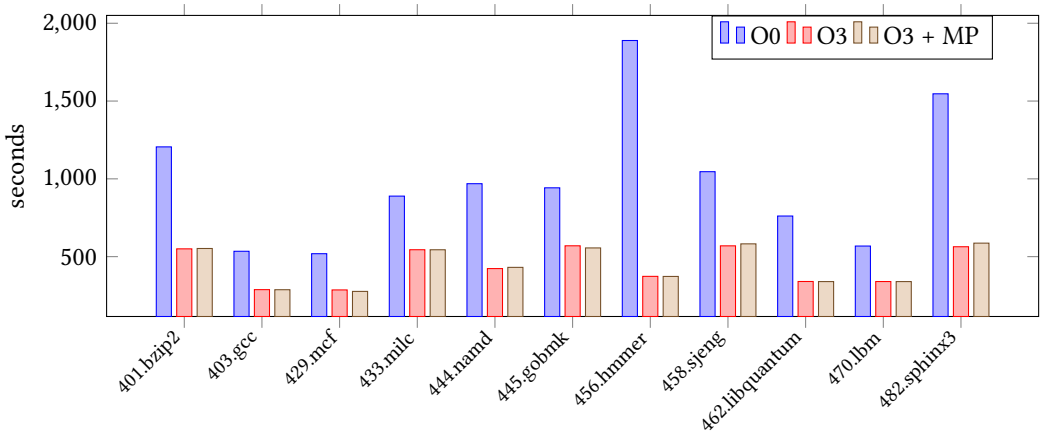

Fig. 5. Performance Comparison of SPEC 2006 GCC Variants Recompiled with Mitigation Passes (MP)

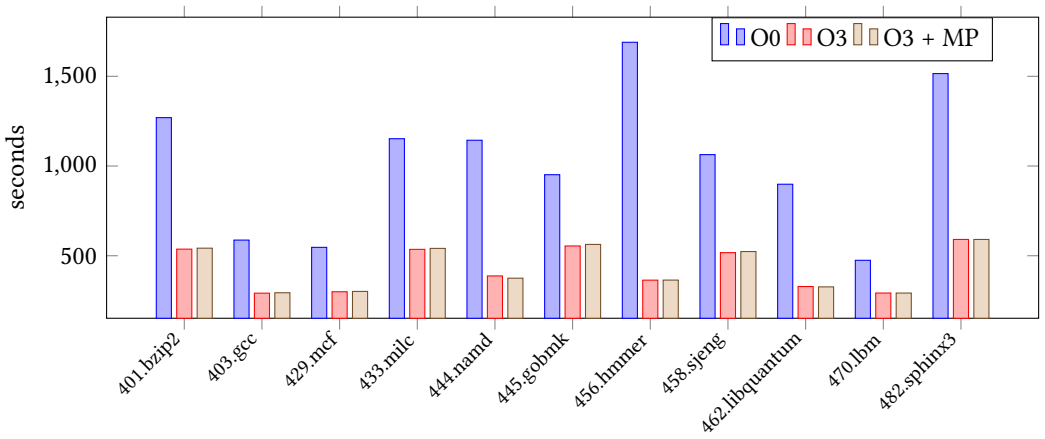

Fig. 6. Performance Comparison of SPEC 2006 Clang Variants Recompiled with Mitigation Passes (MP)

negligible over the base performance of compiler-produced O3 variants, and are imperceptible considering the large performance improvements (i.e., 53% speedup on average) O3 variants enjoy over O0 variants. Thus, recompiling compiler optimized code with our mitigation passes provides the best of both worlds: virtually identical run-time performance to compiler-optimized code with significantly reduced CRA gadget utility and availability.

To analyze the impact of our mitigation passes on code size, we compared the binary sizes of the compiler-produced O3 variants against their recompiled variants. Recompiling binaries with Egalito can result in significant changes to code size, even when no transformation passes are selected. 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 (in kB) of the compiler-produced position-independent O3 variant (i.e., the baseline), a control variant produced by recompiling the baseline with no passes selected, and a variant produced by

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

Benchmark	GCC				Clang			
	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	468.4	(8.2)	455.6	439.7	439.7	(0)
git	19403.2	3626.3	3634.5	(8.2)	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	947.6	(20.5)	6643.7	1328.5	1344.9	(16.4)
401.bzip2	110.6	128.3	132.4	(4.1)	98.5	116.1	120.2	(4.1)
403.gcc	4750.1	5203.3	5313.9	(110.6)	4021.3	4507.0	4527.5	(20.4)
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	6960.5	(4.1)	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)

recompiling the baseline with our passes enabled. In columns marked with a Δ symbol, the values in parentheses indicate the size increase caused by our mitigation passes over the control variant.

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. Our mitigation passes incrementally increased the binary size in roughly one-third of variants, with an average incremental increase of 6.1 kilobytes. With respect to the impact of recompilation on code size, we observed that code size impacts in our control variants were typically negligible. In 27% (9 of 34) of variants, recompilation reduced code size, and in 53% (18 of 34) of cases recompilation increased code size by less than 80 kilobytes. Large increases in code size were observed in the remaining seven variants, in some cases many times over the baseline size. These edge cases indicate instances where Egalito's code generation conventions can be improved.

6.1.6 Evaluation of Correctness. To ensure that recompiling binaries with our mitigation passes enabled does not produce corrupted or otherwise invalid binaries, we performed functional testing of our recompiled variants. While the transformations performed by our mitigation passes are semantics-preserving, functional testing is necessary to ensure that errors do not occur due to synergistic effects between Egalito, our mitigation passes, the benchmark implementation, and the selected compiler.

Functional testing of our SPEC 2006 variants is straightforward as the reference workloads used in our performance evaluation also perform validity checks. For example, the reference workload for `401.bzip2` compresses several test files, creating an archive file. Then, it uncompresses the archive in a temporary directory and then checks that uncompressed files are identical matches to the original test files. If they are not, the test fails there and does not report a performance result. SPEC 2006's workloads also identify when a variant under test does not terminate as expected (e.g., a segmentation fault occurs), which captures instances in which a benchmark uses language features not supported by Egalito. During functional testing of our SPEC 2006 benchmarks, we identified three such benchmarks and excluded them from our evaluation. Specifically, `liblmdb` depends on

a position-dependent code library and `453.povray` and `471.omnetpp` use C++ exceptions which are not currently supported by Egalito.

For recompiled variants of our common Linux benchmarks, we utilized developer provided functional tests to ensure their validity. For example, the larger `cUrl` code repository includes a battery of tests that run against `libcUrl` to verify its functionality. Only one of our benchmarks, `Bftpd`, does not provide functional tests. In this case, we created a custom test suite similar to those provided for our other benchmarks to validate our recompiled `Bftpd` variants.

6.2 Disabling Clang's TCE Optimization

To determine the performance impact of disabling Clang's TCE optimization, we built additional Clang variants of our SPEC 2006 benchmarks at level O3 with TCE disabled. We then ran these variants using reference workloads and recorded the total runtime for each variant across three trials to determine average performance. Our results are shown in Figure 7.[5] Our results indicate that disabling TCE has a significant impact on performance, resulting in an average slowdown of 14%. Additionally, we observed slowdowns greater than 17% in seven of the ten variants we analyzed. Interestingly, we observed one outlier result in which disabling TCE resulted in a 5.9% speedup, though this was the only benchmark in which a positive result was observed.

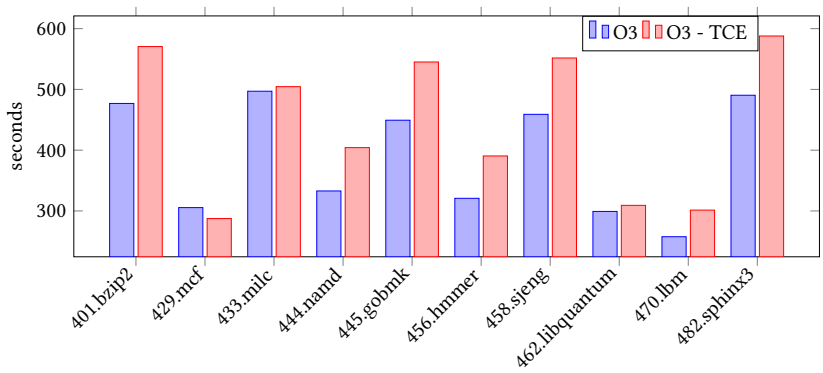

Fig. 7. Performance Comparison of SPEC 2006 Clang Variants with and without TCE Enabled

Based on these results, we assess that the performance benefits of Clang's TCE optimization generally outweigh its undesirable gadget set impacts. While JOP data loader gadgets simplify exploit chain construction by providing a convenient mechanism for stack-based data injection, functional JOP exploits can be constructed without them. Thus, we conclude that the strategy of entirely disabling TCE to avoid introducing these gadgets is non-viable in most practical scenarios. A more targeted strategy of patching Clang's TCE pass to optimize direct tail-calls only may be more practical; however we consider evaluating this strategy to be outside the scope of this paper.

7 LIMITATIONS AND CONSIDERATIONS

7.1 Study Limitations

Our selection of benchmark programs is limited to C/C++ source programs compiled by `GCC` and `Clang` into `x86-64` ELF binaries. Further research is necessary to confirm that our results generalize to build chains that utilize other programming languages (e.g., Fortran), compilers,

[5]Note that we have excluded one benchmark, 403.gcc, from our results because Clang failed to compile it with TCE disabled.

binary formats (e.g., Windows PE), or 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. However, two or more optimizations at the same level may have synergistic effects, which should be considered when applying our methodology.

Again, it is important to note that the metric calculations in our study are limited to gadgets present in our variant binaries and do not include gadgets contained in dynamically linked library code. In the context of our study, library-based gadgets are a confounding variable because shared libraries are pre-built separately from our benchmark binaries; however, these gadgets must be considered in the larger context of assessing the relative ease of crafting a code reuse attack. Additionally, defensive techniques deployed in the execution environment that prevent these gadgets from being used (e.g., position-independent code, ASLR, etc.) must also be considered when assessing gadget sets. Although our study does not include library-based gadgets in calculations, our study findings and mitigation passes are readily applicable to the build process used to create shared libraries.

7.2 Tool Limitations

GSA's functional gadget set expressivity calculation is limited to ROP gadgets. In practice, this covers the majority of practical security use cases as JOP and COP exploits are rare in the wild. Additionally, our mitigation passes implemented with Egalito are subject to the same limitations as the tool itself; namely Egalito can only recompile position-independent code and does not support obfuscated code, inline assembly, and C++ exceptions.

7.3 Security Considerations

When considering our findings in the context of overall software security, it is important to point out that gadget-based code reuse exploit patterns (e.g., ROP, JOP, COP) are not software vulnerabilities in and of themselves; they are powerful techniques for exploiting memory corruption vulnerabilities in the presence of W⊕X defenses. As such, the presence of expressive functional gadgets and special purpose gadgets in binaries does not necessarily make them less secure.

However, we cannot assume or strongly prove that software is vulnerability-free in practice. As a result, it remains important to consider impacts to gadget sets when designing code generation and transformation tools. While software vulnerabilities are the point of origin for gadget-based code reuse attacks, the malicious behaviors (e.g., DoS, privilege escalation, etc.) the attacker can manifest are dependent upon the gadgets available. Thus, reducing the availability and utility of gadgets present in binaries using techniques such as those developed in this work can limit an attacker's ability to construct attacks, which in turn positively impacts overall software security.

This work endeavors to show that compiler (and recompiler) techniques can produce performant binaries with gadget sets that are minimally useful to an attacker in the event that the program contains an unknown memory corruption vulnerability. While this approach does not eliminate software vulnerabilities, it does achieve important security objectives such as cyber hygiene and defense-in-depth.

8 RELATED WORK

The mitigation strategies we have proposed and evaluated in Section 6.1 can be classified as gadget-based CRA defenses similar in nature to those described in Section 2.4, although our approach differs significantly from prior work. In contrast to binary retrofitting transformations that add costly dynamic (i.e., run-time) control-flow protections to binaries, our approach is fully static as to avoid run-time penalties. Despite this key difference, we consider our work to be complementary to these approaches; our proposed mitigation passes can potentially reduce the overhead costs of

employing dynamic defenses by reducing the number of distinct control-flow transfers that require instrumentation and run-time protection.

With respect to compiler-based defenses, our work is similar in several ways to G-Free [Onarlioglu et al. 2010], an assembler injection-based solution for `GCC` that eliminates unintended gadgets via transformation and inserts run-time protections for compiler-placed GPIs. Two of our transformation techniques for eliminating unintended gadgets, instruction barrier widening and jump displacement sledding, operate similarly to G-Free's. However, our approach differs in many ways. First, our approach does not require source code and is narrower in scope. Our mitigation passes are designed to address undesirable optimization behaviors identified in our study, whereas G-Free addresses other sources of unintended gadgets such as problematic register allocations during code generation. Second, our approach avoids costly run-time overheads by eliminating intended GPIs where possible via merging passes rather than inserting run-time protections. Our approach also avoids costly code size overheads by exercising direct control over program layout with Egalito. G-Free's design lacks this control and may require very large numbers of `nop` instructions to eliminate unintended gadgets in call displacements and offsets. Once again, we view our approach to be complementary in that it may be used to perform gadget elimination in a manner that reduces G-Free's relatively high overhead costs (average 3.1% slowdown and 25.9% increase in code size).

9 ARTIFACT AVAILABILITY

(1) **Study Data.** Our corpus of program variants and their associated GSA output is available at: https://github.com/michaelbrownuc/compiler-opt-gadget-dataset
(2) **Recompiler Passes.** The source code for our Egalito recompiler passes is available at: https://github.com/michaelbrownuc/egalito-gadgets
(3) **Virtual Machine.** A virtual machine in `ova` format containing the above artifacts as well as other supplementary material and tools is available at: https://doi.org/10.5281/zenodo.5424844 [Brown 2021c]

10 CONCLUSION

In this work, we developed a data-driven methodology for studying of the impact of compiler optimizations on code reuse gadget sets. Employing this methodology, our coarse- and fine-grained analysis of optimization behaviors revealed that sets of code reuse gadgets in optimized binaries are significantly more useful for constructing CRA exploit chains than those in unoptimized binaries. We identified the root causes of our observations through differential binary analysis, and proposed potential mitigation strategies for them. We demonstrated that implementing our mitigation strategies for GPI duplication and transformation-induced gadget introduction as post-production recompiler passes can significantly reduce the availability and utility of code reuse gadgets in optimized code. Further, we demonstrated that these benefits can be obtained with negligible performance impact through a performance analysis of binaries transformed with our recompiler passes. Finally, we evaluated the performance costs of disabling `Clang`'s TCE optimization and determined that this strategy is likely to be too costly in practical scenarios.

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, 340–353.

M. Angelini, G. Blasilli, P. Borrello, E. Coppa, D. C. D'Elia, S. Ferracci, S. Lenti, and G. Santucci. 2018. ROPMate: Visually Assisting the Creation of ROP-based Exploits. In *2018 IEEE Symposium on Visualization for Cyber Security (VizSec)*. 1–8.

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, 29–40. 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, 353–362. 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, 30–40. https://doi.org/10.1145/1966913.1966919

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

Michael D. Brown. 2021a. Compiler Optimization Data Set README. https://github.com/michaelbrownuc/compiler-opt-gadget-dataset/blob/main/README.md.

Michael D. Brown. 2021b. GSA Gadget Criteria Reference. https://github.com/michaelbrownuc/GadgetSetAnalyzer/blob/master/Criteria.md.

Michael D. Brown. 2021c. Not So Fast: Understanding and Mitigating Negative Impacts of Compiler Optimizations on Code Reuse Gadget Sets. *PACMPL* OOPSLA 21 (Oct. 2021). https://doi.org/10.5281/zenodo.5424844

Michael D. Brown and Santosh Pande. 2019. 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

Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. 2015. Control-flow bending: On the effectiveness of control-flow integrity. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*. 161–176.

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, 559–572. 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, 952–963. 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, 49–54. 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, 40–51. 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, 73–87.

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, 901–913. https://doi.org/10.1145/2810103.2813646

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

Andreas Follner, Alexandre Bartel, Hui Peng, Yu-Chen Chang, Kyriakos Ispoglou, Mathias Payer, and Eric Bodden. 2016b. PSHAPE: Automatically Combining Gadgets for Arbitrary Method Execution. In *Security and Trust Management*, Gilles Barthe, Evangelos Markatos, and Pierangela Samarati (Eds.). Springer International Publishing, Cham.

Brian Grant, Matthai Philipose, Markus Mock, Craig Chambers, and Susan J. Eggers. 1999. An Evaluation of Staged Run-Time Optimizations in DyC. *SIGPLAN Not.* 34, 5 (May 1999), 293–304. https://doi.org/10.1145/301631.301683

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, 94–105.

Mehmet Kayaalp, Meltem Ozsoy, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2012b. Branch Regulation: Low-Overhead Protection from Code Reuse Attacks. *SIGARCH Comput. Archit. News* 40, 3 (June 2012), 94–105. https://doi.org/10.1145/2366231.2337171

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.

Han Lee, Daniel von Dincklage, Amer Diwan, and J. Eliot B. Moss. 2006. Understanding the Behavior of Compiler Optimizations. *Softw. Pract. Exper.* 36, 8 (July 2006), 835–844.

Jinku Li, Zhi Wang, Xuxian Jiang, Michael Grace, and Sina Bahram. 2010. Defeating Return-Oriented Rootkits with "Return-Less" Kernels. In *Proceedings of the 5th European Conference on Computer Systems* (Paris, France) *(EuroSys '10)*. Association for Computing Machinery, New York, NY, USA, 195–208. 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, 37–42. https://doi.org/10.1145/3139337.3139343

Scott McFarling and J Hennesey. 1986. Reducing the cost of branches. *ACM SIGARCH Computer Architecture News* 14, 2 (1986), 396–403.

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, 49–58. https://doi.org/10.1145/1920261.1920269

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

Chris Porter, Girish Mururu, Prithayan Barua, and Santosh Pande. 2020. BlankIt Library Debloating: Getting What You Want Instead of Cutting What You Don't. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 17 pages. https://doi.org/10.1145/3385412.3386017

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

Anh Quach, Aravind Prakash, and Lok Yan. 2018. Debloating software through piece-wise compilation and loading. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 869–886.

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, 552–561. 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.

Zhaomo Yang, Brian Johannesmeyer, Anders Trier Olesen, Sorin Lerner, and Kirill Levchenko. 2017. Dead store elimination (still) considered harmful. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*. 1025–1040.

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.