Is Less *Really* More? Towards Better Metrics for Measuring Security Improvements Realized Through Software Debloating

Michael D. Brown, *Georgia Institute of Technology* Santosh Pande, *Georgia Institute of Technology*

Abstract

Nearly all modern software suffers from bloat that negatively impacts its performance and security. To combat this problem, several automated techniques have been proposed to debloat software. A key metric used in these works to demonstrate improved security is code reuse gadget count reduction. The use of this metric is based on the prevailing idea that reducing the number of gadgets available in a software package reduces its attack surface and makes mounting a gadget-based code reuse exploit such as return-oriented programming (ROP) more difficult for an attacker.

In this paper, we challenge this idea and show through a variety of realistic debloating scenarios the flaws inherent to the gadget count reduction metric. Specifically, we demonstrate that software debloating can achieve high gadget count reduction rates, yet fail to limit an attacker's ability to construct an exploit. Worse yet, in some scenarios high gadget count reduction rates conceal instances in which software debloating makes security *worse* by introducing new quality gadgets.

To address these issues, we propose new metrics based on quality rather than quantity for assessing the security impact of software debloating. We show that these metrics can be efficiently calculated with our Gadget Set Analyzer tool. Finally, we demonstrate the utility of these metrics through a realistic debloating case study.

1. Introduction

Software debloating [1-5] is an emerging field of research focused on improving software security and performance by eliminating bloat that occurs as a byproduct of modern software engineering practices. While these practices enable the rapid development of complex, widely deployable, and feature-rich software, they produce software packages (programs, libraries, etc.) with large portions of code that are unnecessary in most end use contexts. These portions of the package constitute software bloat and result in a variety of negative performance and security impacts [1, 6, 22].

Software bloat affects virtually all software and primarily occurs vertically in the software stack across layers of abstraction [1]. Programs that depend on common shared code libraries such as `libc` typically only require a small number of functions provided by the library, but load the entire library into the program's memory space at runtime.

Software bloat also occurs laterally within software packages suffering from feature creep. Examples include software such as cUrl, which can be used to transfer data via 23 different protocols, and iTunes, which features a media player, ecommerce platform, and hardware device interface within a single package. Since end users are unlikely to use every feature within these packages, the code associated with unused features contributes to software bloat.

Recently, several software debloating techniques [2-5] have been proposed that promise to improve software security by removing code bloat at various stages of the software lifecycle. A frequently utilized metric for measuring security improvements realized via debloating is the reduction in total count of code reuse gadgets available to an attacker, which we refer to as *gadget count reduction*. Several recent debloating publications [3-5] claim their methods improve security citing gadget count reduction data as evidence.

The relationship between gadget count reduction and improved security is based on the premise that reducing the total number of code reuse gadgets available in a software package reduces its attack surface. In turn, this decreases the likelihood of an attacker successfully constructing a code reuse exploit using techniques such as return, jump, or call-oriented programming (also known as ROP, JOP, and COP [7, 21, 8, 9]). At face value, gadget count reduction is an appealing security improvement metric as it is easily generated using existing automated static analysis tools [14] and is directly relevant to a class of cyberattacks that have been the focus of intense research over the last decade [7-15].

The premise linking gadget count reduction to improved security holds only if the gadgets removed by debloating are critical to the construction of an exploit, and other gadgets with equivalent functionality are not available. For an attacker attempting to construct a code reuse exploit, the total number of gadgets available is irrelevant; what truly matters is whether or not the gadgets necessary to express their desired exploit and maintain control flow are available. Recent research on gadget chaining tools [10, 11] and code reuse attack techniques [12,13] have shown that attackers do not require a large, diverse, and fully expressive set of gadgets in order to craft an exploit. As a result, it is possible that debloating can achieve high gadget count reduction and indicate an improvement in security, yet fail to remove any of the gadgets an attacker needs to express and construct an exploit.

Even worse, our research indicates that debloating techniques that remove code from a package *introduce new gadgets* at a high rate as a side effect. Except in rare cases where the total count of gadgets is increased by debloating, this poorly understood side effect is masked by gadget count reduction data. This opens the possibility that debloating "successfully" reduces the overall count of gadgets, but introduces new, useful gadgets that may negatively impact security.

1.1. Contributions

In section 3 of this paper, we present the results of our study of gadget introduction as a side effect of code-removing debloaters. We describe the root causes of gadget introduction, and show that it is occurs at a high rate using two different code-removing debloaters.

In section 4 of this paper, we propose new metrics for measuring the security impact of software debloating. Our proposed metrics, *functional gadget set expressivity* and *special purpose gadget availability*, assess the utility of the gadgets available to the attacker rather than the quantity. We present our static analysis tool capable of calculating these metrics, Gadget Set Analyzer (GSA) in section 5.

In section 6, we use GSA to demonstrate the shortcomings of gadget count reduction and show the value of our proposed metrics in realistic debloating scenarios. In each scenario, positive gadget count reduction is achieved; however, GSA reveals that a significant number of scenarios are negatively impacted by gadget introduction.

Finally, in section 7 we demonstrate through a case study that our proposed metrics can be used to mitigate the negative side effects of debloating. In this case study, we identify a scenario in which debloating had negative effects, adjust the debloating specification, generate a new variant, and use GSA to verify that these negative effects are eliminated.

2. Background

2.1. Relevant Terms

Code Reuse Attacks: Code reuse attacks are a class of attacks in which an attacker compromises the control flow of a program and redirects execution to an existing executable part of the program to cause a malicious effect, bypassing code injection defenses such as Write XOR Execute. In gadget-based code reuse attack methods such as ROP, JOP, and COP [7, 21, 8, 9], the attacker chains together short instruction sequences called gadgets present in the program in a specific order to construct a malicious payload without injecting code.

Gadget: A gadget suitable for use in a code reuse attack is a short sequence of machine instructions that end in a return, indirect jump, or indirect call instruction. Gadgets can be chained together using the control flow properties of the terminating instruction to create a malicious payload comprised entirely of existing code segments.

Gadget Types: When constructing a gadget chain, gadgets are used for one of two purposes. Functional gadgets are used as abstract instructions to express the attacker's malicious intent. Gadgets that can be used to perform important non-expressive actions such as invoking system calls or maintaining gadget chain control flow are called special purpose gadgets.

2.2. Related Work

CHISEL: Lee et al. [3] recently proposed an automatic method for debloating unnecessary features from program source code called CHISEL. CHISEL takes as input a specification script that outputs whether or not a debloated variant satisfies the desired program properties. Using an iterative, feedback-directed program reduction algorithm, CHISEL progressively removes segments of the program that are not necessary to satisfy the desired properties.

This work cites gadget count reduction data as evidence of security improvement through attack surface reduction, but does not provide further analysis of the gadgets present in their debloated programs. CHISEL's source code and benchmarks have been made publicly available [16, 17].

TRIMMER: Sharif et al. [4] recently proposed an automated method for debloating unnecessary functionality from software named TRIMMER. TRIMMER takes as input a static user defined configuration that expresses the deployment context for a particular program. Static configuration data is treated as a compile time constant and is propagated throughout the program. This is followed by custom, aggressive compiler optimizations to prune functionality from the program.

The authors provide gadget count reduction data as evidence that TRIMMER reduces the attack surface of a program by removing exploitable gadgets; However, they provide no explanation of what makes a gadget exploitable as opposed to non-exploitable. Additionally, their data indicates that syscall gadgets were introduced as a result of debloating, yet no explanation or investigation of this occurrence is provided. TRIMMER has not yet been made publicly available.

3. Gadget Introduction via Debloating

Techniques such as CHISEL and TRIMMER that debloat by altering a software package's representation (source code, intermediate representation, or binary) through code removal or progressive optimization can introduce new gadgets into the debloated variant. Since we do not have prior knowledge of which gadgets are useful to an attacker, gadget introduction can potentially offset security improvements realized through debloating, or even make a debloated package *less secure*. We describe the root causes of gadget introduction in the following two sections.

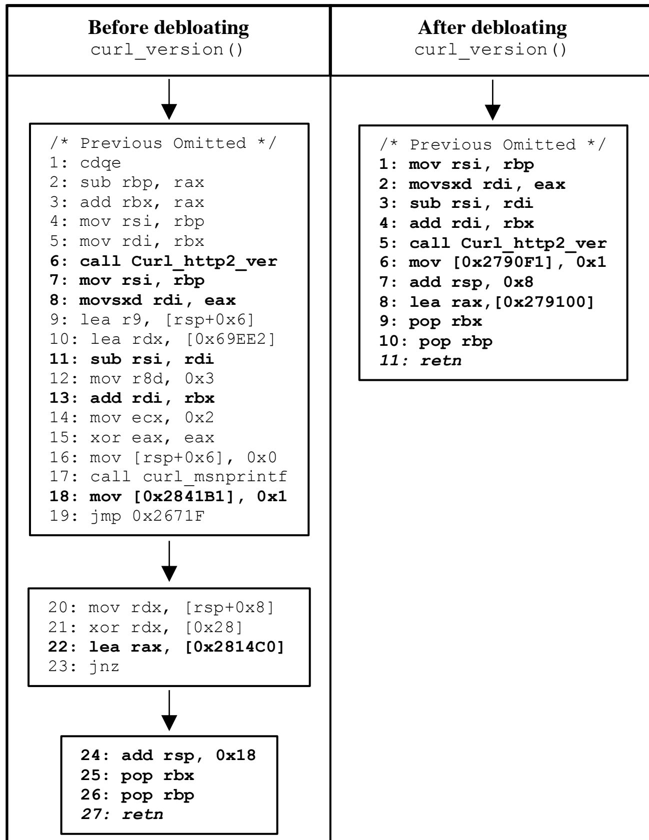
3.1. Introduction of Intended Gadgets

Gadgets comprised of compiler generated binary instructions are referred to as *intended gadgets*. Changes to package's

representation caused by debloating can cause downstream compiler stages to make different optimization and code generation decisions. This results in changes to compiler generated instructions, introducing new intended gadgets.

Consider the control flow graph (CFG) excerpts from `libcurl` shown in Figure 1. The excerpt on the left is from the original version of the function `curl_version`, and the excerpt on the right shows the results of debloating eight lines from the corresponding source code. Removing the source code results in fewer binary instructions as expected, however this shorter sequence of instructions has simpler control flow. Specifically, the `jmp` instruction on line 19 and all but one instruction in the basic block following it are removed. As a result, all three basic blocks in the original version are merged into a single basic block in the debloated version. This change in locality increases the range of instructions the compiler can reorder to maximize performance (lines 24-27 in the original version versus lines 1-11 in the debloated version). Comparing the number of unique intended ROP gadgets produced by these two sequences, the net result is an increase in the gadget count. Two gadgets, `[pop rbx; pop rbp; retn;]` and `[pop rbp; retn;]`, are present in both versions. One gadget is eliminated from the original version (lines 24-27), but two gadgets are introduced in the debloated version (lines 7-11 and 8-11).

Figure 1: Control flow graph snippets of `curl_version`.



In addition to instruction reordering, we have observed a number of other compiler operations that can be triggered by debloating. Examples include CFG generation, loop optimizations, function inlining, and dead code elimination.

3.2. Introduction of Unintended Gadgets

For ISAs with variable length instructions such as x86 and x86-64, it is possible to decode instructions from an offset other than the original instruction boundary to obtain new, potentially valid instruction sequences [7]. Code reuse gadgets found in these sequences are referred to as *unintended gadgets*. When debloating alters compiler generated instructions, it also introduces new unintended gadgets when the altered code is interpreted at unintended offsets.

For example, consider the sequence of instructions taken from the last basic block of `curl_version` before debloating, shown in the top segment of Figure 2. Unintended gadgets can be identified by interpreting the sequence at an offset of one byte and three bytes after the first instruction boundary, shown in the bottom two segments of Figure 2. After debloating, the instructions in the final basic block (top segment of Figure 3) have changed significantly. In addition to altering the intended gadgets found in this basic block, debloating also changes the unintended gadgets found at

Figure 2: Final basic block of `curl_version` before debloating, interpreted at three different byte offsets.

```
48 83 c4 18: add rsp,0x18
5b:          pop rbx
5d:          pop rbp
c3:          ret
```

```
83 c4 18:    add esp,0x18
5b:          pop rbx
5d:          pop rbp
c3:          ret
```

```
18 5b 5d:    sbb BYTE PTR[rbx+0x5d],bl
c3:          ret
```

Figure 3: Final basic block of `curl_version` after debloating, interpreted at three different byte offsets.

```
c6 05 0f 4d
24 00 01:           mov BYTE PTR[rip+0x244d0f],0x1
48 83 c4 08:        add rsp,0x8
48 8d 05 13
4d 24 00:           lea rax,[rip+0x244d13]
5b:                 pop rbx
5d:                 pop rbp
c3:                 ret
```

```
08 48 8d :          or BYTE PTR [rax-0x73],cl
05 13 4d 24 00:     add eax,0x244d13
5b:                 pop rbx
5d:                 pop rbp
c3:                 ret
```

```
13 4d 24:           adc ecx,DWORD PTR [rbp+0x24]
00 5b 5d:           add BYTE PTR[rbx+0x5d],bl
c3:                 ret
```

Table 1: Number and percentage of gadgets introduced by type in debloated software packages.

			Introduced Functional Gadgets				Introduced Special Purpose (S.P.) Gadgets		
	Debloated Variant	Gadget Count Reduction	All Gadgets	ROP Gadgets	JOP Gadgets	COP Gadgets	System Call Gadgets	JOP Specific S.P. Gadgets	COP Specific S.P. Gadgets
CARVE	libmodbus (C)	89 (14%)	222 (39%)	149 (33%)	73 (60%)	33 (66%)	N/A	2 (67%)	N/A
	libmodbus (M)	108 (16%)	221 (40%)	163 (37%)	58 (55%)	35 (67%)	N/A	0 (0%)	N/A
	libmodbus (A)	143 (22%)	252 (49%)	156 (41%)	96 (73%)	44 (73%)	N/A	0 (0%)	N/A
	Bftpd (C)	40 (5%)	249 (35%)	202 (32%)	47 (61%)	20 (80%)	N/A	0 (0%)	N/A
	Bftpd (M)	124 (16%)	260 (41%)	197 (36%)	63 (74%)	44 (90%)	N/A	0 (0%)	N/A
	Bftpd (A)	220 (29%)	196 (37%)	167 (34%)	29 (62%)	14 (78%)	N/A	0 (0%)	N/A
	libcurl (C)	214 (2%)	4727 (51%)	1858 (45%)	2864 (56%)	2165 (52%)	5 (100%)	17 (30%)	305 (99%)
	libcurl (M)	1470 (15%)	4178 (52%)	1631 (44%)	2547 (58%)	2003 (56%)	0 (0%)	10 (23%)	287 (99%)
	libcurl (A)	3766 (40%)	3334 (58%)	1422 (49%)	1911 (68%)	1664 (69%)	4 (100%)	5 (26%)	171 (99%)
	Mongoose (C)	18 (2%)	412 (33%)	307 (29%)	105 (66%)	55 (78%)	N/A	0 (0%)	N/A
	Mongoose (M)	52 (4%)	396 (33%)	277 (27%)	119 (60%)	63 (80%)	N/A	0 (0%)	1 (100%)
	Mongoose (A)	99 (8%)	405 (35%)	258 (27%)	147 (67%)	63 (80%)	N/A	0 (0%)	N/A
CHISEL	bzip2	442 (65%)	152 (64%)	99 (59%)	53 (76%)	45 (75%)	N/A	1 (100%)	N/A
	chown	327 (65%)	94 (54%)	68 (47%)	26 (84%)	14 (87.5%)	N/A	1 (100%)	N/A
	date	260 (55%)	119 (56%)	89 (51%)	30 (77%)	23 (85%)	N/A	1 (100%)	N/A
	grep	378 (36%)	479 (72%)	380 (69%)	99 (87%)	79 (93%)	N/A	1 (100%)	N/A
	gzip	195 (46%)	156 (68%)	127 (65%)	29 (83%)	25 (86%)	N/A	1 (100%)	N/A
	mkdir	101 (48%)	47 (42%)	35 (38%)	12 (67%)	5 (83%)	N/A	1 (100%)	N/A
	rm	384 (72%)	77 (52%)	47 (41%)	30 (91%)	15 (88%)	N/A	1 (100%)	N/A
	tar	1355 (84%)	144 (57%)	75 (44%)	69 (86%)	52 (88%)	N/A	2 (100%)	N/A
	uniq	176 (59%)	53 (43%)	33 (34%)	20 (77%)	5 (71%)	N/A	1 (100%)	N/A

various offsets from the original instruction boundary (bottom two segments of Figure 3). These unintended gadgets differ significantly from those found in the original sequence. The net result of this operation is the elimination of the unintended gadgets in the original sequence and the introduction of the unintended gadgets in the resulting sequence.

3.3. Prevalence of Gadget Introduction

To determine the degree to which software debloating introduces new gadgets, we debloated a variety of common software packages at varying levels of aggressiveness with two different code-removing debloaters. We analyzed each package and its variants using ROPgadget [14] to catalog its gadgets by type, and then performed a set-wise comparison to identify gadgets present in the debloated variants that were not present in the original package.

For this analysis we used our debloater, CARVE [20], and CHISEL (the only publicly available debloating tool). CARVE operates in a comparable fashion to other feature-based approaches such as CHISEL and TRIMMER, in that it removes code associated with features before generating the package binary. CARVE is implemented as a preprocessor pass on package source code that has been enriched with user-embedded feature mappings that associate segments of code with debloatable features. CARVE takes as input a user-specified list of features to debloat and scans the enriched source code for corresponding feature mappings. When found, CARVE performs syntax-aware analysis on the mapped code and intelligently removes it in a sound manner. The debloated source code is then compiled to produce a debloated binary using the same build process as the original version.

We debloated four software packages with CARVE: `libmodbus` v3.1.4, `Bftpd` v4.9, `libcurl` v7.61, and `mongoose` v.6.8. Each package was debloated at three levels of aggressiveness, defined below:

- Conservative (C): Some peripheral features in the package are removed.
- Moderate (M): Some peripheral features and some core features are removed from the package.
- Aggressive (A): All debloatable features except for a small set of core features are removed from the package.

We used CHISEL to debloat nine software packages from the author provided benchmark set [17]: `bzip2` v1.05, `chown` v8.2, `date` v8.21, `grep` v2.19, `gzip` v1.2.4, `mkdir` v5.2.1, `rm` v8.4, `tar` v1.14, and `uniq` v8.16. We used the author provided specifications (roughly equivalent to our definition of aggressive debloating) to create debloated variants.

Table 1 contains the results of our gadget count reduction and gadget introduction analysis. As shown in first column, debloating successfully reduced the total count of gadgets in all scenarios, and in most cases the total count is reduced by a large degree (>15%).

The introduction of new gadgets via debloating is neither a rare nor limited occurrence. In all scenarios, a significant portion of the functional gadgets remaining after debloating were introduced gadgets (35% was the *smallest* observed rate). In 12 of 21 scenarios, introduced gadgets accounted for the majority of remaining gadgets. Further, gadget introduction occurs at a significant rate across all gadget subcategories (ROP, JOP, COP), aggressiveness levels, and benchmarks.

Significant levels of gadget introduction were also observed for special purpose gadgets (a value of N/A indicates that no gadgets of this type were present in the variant). In 14 of the 21 scenarios, debloating resulted in the introduction of new special purpose gadgets. As was the case with functional gadgets, special purpose gadget introduction is not strongly correlated to debloating aggressiveness, gadget subcategory, or benchmark type.

The prevalence of gadget introduction has serious implications for the use of gadget count reduction as a security metric. Our data strongly indicates that the gadgets present in a debloated variant will not be a proper subset of the gadgets in the original package, rendering metrics that capture only the change in size of a set insufficient and superficial.

4. Gadget Utility Metrics

In this section, we propose two new metrics for assessing the security impact of software debloating. Our metrics are designed to measure the degree to which debloating adversely affects the construction of gadget-based code reuse exploits by assessing the *utility* of the gadgets present after debloating as opposed to the *quantity*. Since functional gadgets and special purpose gadgets are utilized in different manners, we propose a metric suited for each gadget type.

4.1. Functional Gadget Set Expressivity

Functional gadgets are used as abstract instructions that perform basic computational operations such as addition, register loading, and logical branching to construct a malicious payload. The expressivity of a set of gadgets is a measure of the computational power the set of gadgets permit.

The expressivity of a set of gadgets is typically measured against the bar of Turing-completeness. A set of gadgets is considered Turing-complete if it is sufficient to express any arbitrary program, i.e. it is computationally universal. While this level of expressivity is not difficult to achieve in practice [7-9, 12], it is not the minimum level of expressivity necessary to construct a practical ROP exploit. Exploits that mark a region of memory as writable, inject malicious code, and redirect execution to this injected code do not require Turing-complete levels of expressivity [13].

For a gadget set to achieve a certain level of expressivity, it must contain at least one gadget supporting each necessary computational class. Thus, a straightforward measure of the expressivity of a gadget set is the number of computational classes satisfied by the gadget set. For example, a gadget set that can be used for addition, register loading, and conditional branching is considered more expressive than one that supports only addition and register loading.

If debloating removes all gadgets that perform a certain computation, the number of satisfied classes decreases. As a result, an attacker may not be able to express their desired exploit. If debloating introduces gadgets that perform previously unavailable computations, the number of satisfied classes increases, indicating a negative security impact.

4.2. Special Purpose Gadget Availability

Special purpose gadgets are used to perform important non-expressive actions in an exploit gadget chain. These gadgets must meet specific criteria, and are typically encountered infrequently. Without special purpose gadgets, some exploits are not possible. For example, JOP/COP exploits do not use the stack, and instead rely on special purpose gadgets such as dispatchers and trampolines to maintain control flow from one gadget to the next. Also, exploits that must invoke system calls require special purpose gadgets.

Given their importance, the availability of special purpose gadgets is a useful metric for determining if debloating has reduced the types of exploits an attacker can construct. The availability of special purpose gadgets can be measured by maintaining counts of each type of special purpose gadget. If a debloating operation removes all of the special purpose gadgets of a particular type, then the attacker may not be able to construct their desired exploit. This metric is also capable of detecting the negative side effects of gadget introduction. If debloating introduces new special purpose gadgets of a particular type, this increase in availability is observable.

4.3. Discussion of Metrics

We do not claim that these metrics are comprehensive. We encourage further discussion that will lead to the exploration of additional useful security-oriented metrics for debloating. Other measures of functional gadget quality have been explored by Follner et al. [18] such as gadget length and memory side effects, which are potentially applicable to the problem of measuring the security impact of debloating.

5. Gadget Set Analyzer (GSA)

To assess the practicality of our metrics, we created a static binary analysis tool capable of analyzing a software package and its debloated variants to capture changes in functional gadget expressivity and special purpose gadget availability.

5.1. Operation

GSA operates in a rather straightforward manner and makes use of existing tools where possible. GSA takes as input the original package binary and one or more debloated variant binaries. First, GSA uses ROPgadget [14] to search each binary for unique ROP, JOP and Syscall gadgets. GSA then performs secondary search of these results to identify unique COP gadgets and JOP/COP specific special purpose gadgets.

Next, GSA utilizes a tool first proposed and implemented by Homescu et al. [12] for classifying short ROP gadgets (called microgadgets) into computational classes. This microgadget scanner then determines the expressive power of the gadget set relative to different levels of expressivity by analyzing the

number of computational classes satisfied. GSA uses this scanner to analyze the gadget set with respect to three levels of expressivity originally proposed in [12]: simple Turing-completeness, expressivity required for practical ROP exploits, and expressivity required for practical, ASLR-proof ROP exploits. Each level requires a different number of computational classes be satisfied by at least one gadget: 17 for simple Turing-completeness, 11 for practical ROP exploits, and 35 for ASLR-proof, practical ROP exploits.

After collecting gadget set and expressivity data using these tools, GSA compares the data for each debloated variant against the data collected on the original binary to calculate functional gadget set expressivity, special purpose gadget availability, and gadget count reduction metrics.

GSA also provides a means for the user to address gadgets introduced by debloating. Using the binary analysis library angr [19], GSA attempts to identify the names of functions containing introduced gadgets. This information can be used to identify the source of an undesirable debloating operation, potentially allowing the user alter their debloating specification to generate a new variant that does not introduce the gadget. GSA provides this as a "best effort" feature, and cannot guarantee that a function name can be retrieved. This is due to several factors including imprecision associated with generating a CFG in angr and the mechanism by which the gadget was introduced.

5.2. Limitations

GSA has the same limitations as the gadget expressivity scanner it incorporates. Specifically, the microgadget scanner used by GSA generates functional gadget expressivity data based solely on an analysis of short gadgets in the set. As a result, a gadget set may be more expressive than reported by GSA if longer gadgets excluded from analysis can satisfy additional computational classes. Also, the microgadget scanner only calculates the expressivity of ROP gadget sets, and does not calculate expressivity for JOP or COP gadget sets.

Additionally, dynamically linked external libraries are not scanned by GSA. At runtime, these libraries are loaded into memory and their code is mapped to the package's address space. Gadgets present in these libraries contribute to overall expressivity and special purpose gadget availability, and should be considered for a holistic view.

5.3. Performance

GSA and its dependencies perform binary analysis exclusively with static techniques. As such, the time required to run GSA increases with the size of the binary. GSA is performant, typically requiring less than 30 seconds to analyze a binary and three variants on a typical laptop. The maximum time required to execute GSA we observed was 2 minutes and 33 seconds, which occurred when analyzing `libcurl` and its three debloated variants.

6. Results

In this section, we present the results generated by GSA across our debloating scenarios. Our results demonstrate both the utility of our proposed metrics and the shortcomings of gadget count reduction.

6.1. Functional Gadget Set Expressivity

Table 2 contains functional gadget set expressivity data collected by GSA. Debloating was generally successful in reducing expressivity when used aggressively on simple software packages (CHISEL benchmarks). However, this was not the case in the larger, more complex packages and less aggressive debloating scenarios (CARVE benchmarks).

In two scenarios, debloating reduced the gadget count but did not decrease gadget set expressivity (*italicized* text). In these cases, gadget count reduction indicates a positive security impact, but the actual impact on security is neutral. Of greater concern are the four scenarios in which debloating increased in gadget set expressivity (**bold** text) by introducing new gadgets that satisfied previously unsatisfied computational classes. In all four scenarios, using gadget count reduction as a security metric indicates positive results, but fails to identify this negative security impact.

Table 2: Computational Classes Satisfied (and Reduction) by Gadget Set for Three Functional Expressivity Levels.

CARVE				CHISEL			
Package Variant	**Practical ROP Exploit Classes**	**ASLR-Proof ROP Classes**	**Simple Turing Complete Classes**	**Package Variant**	**Practical ROP Exploit Classes**	**ASLR-Proof ROP Classes**	**Simple Turing Complete Classes**
libmodbus (C)	6 of 11 (0)	10 of 35 (3)	6 of 17 (1)	bzip2	3 of 11 (2)	5 of 35 (2)	1 of 17 (2)
libmodbus (M)	*6 of 11 (0)*	*13 of 35 (0)*	*7 of 17 (0)*	chown	3 of 11 (0)	5 of 35 (4)	2 of 17 (2)
libmodbus (A)	*6 of 11 (0)*	*13 of 35 (0)*	*7 of 17 (0)*	date	3 of 11 (2)	5 of 35 (4)	2 of 17 (2)
Bftpd (C)	**7 of 11 (-1)**	12 of 35 (3)	6 of 17 (1)	grep	3 of 11 (2)	6 of 35 (5)	2 of 17 (5)
Bftpd (M)	**7 of 11 (-1)**	**17 of 35 (-2)**	6 of 17 (1)	gzip	3 of 11 (2)	5 of 35 (1)	1 of 17 (2)
Bftpd (A)	6 of 11 (0)	11 of 35 (4)	5 of 17 (2)	mkdir	3 of 11 (0)	6 of 35 (0)	1 of 17 (1)
libcurl (C)	9 of 11 (0)	**26 of 35 (-1)**	**11 of 17 (-1)**	rm	3 of 11 (0)	5 of 35 (4)	2 of 17 (2)
libcurl (M)	9 of 11 (0)	24 of 35 (1)	10 of 17 (0)	tar	3 of 11 (2)	5 of 35 (4)	1 of 17 (4)
libcurl (A)	**10 of 11 (-1)**	24 of 35 (1)	10 of 17 (0)	uniq	3 of 11 (0)	5 of 35 (4)	1 of 17 (3)
Mongoose (C)	7 of 11 (0)	10 of 35 (6)	8 of 17 (0)				
Mongoose (M)	7 of 11 (0)	10 of 35 (6)	8 of 17 (0)				
Mongoose (A)	7 of 11 (0)	10 of 35 (6)	8 of 17 (0)				

Table 3: Special Purpose Gadget Counts (and Reduction) by Gadget Set (excludes types not observed in any variant).

	Package Variant	Syscall Gadget	JOP Dispatcher Gadgets	JOP Data Loader Gadgets	JOP Trampoline Gadgets	COP Dispatcher Gadgets	COP Intra Stack Pivot Gadgets
CARVE	***libmodbus (C)***	*0 (0)*	*0 (0)*	***3 (-2)***	*0 (0)*	*0 (0)*	*0 (0)*
	libmodbus (M)	*0 (0)*	*0 (0)*	*1 (0)*	*0 (0)*	*0 (0)*	*0 (0)*
	libmodbus (A)	*0 (0)*	*0 (0)*	*1 (0)*	*0 (0)*	*0 (0)*	*0 (0)*
	Bftpd (C)	0 (0)	0 (0)	9 (1)	0 (0)	0 (0)	0 (0)
	Bftpd (M)	0 (0)	0 (0)	1 (9)	0 (0)	0 (0)	0 (0)
	Bftpd (A)	0 (0)	0 (0)	1 (9)	0 (0)	0 (0)	0 (0)
	libcurl (C)	**5 (-1)**	7 (1)	50 (1)	0 (1)	304 (15)	4 (0)
	libcurl (M)	0 (4)	4 (4)	41 (10)	0 (1)	287 (32)	3 (1)
	libcurl (A)	4 (0)	3 (5)	14 (37)	1 (0)	170 (149)	2 (2)
	Mongoose (C)	*0 (0)*	***1 (-1)***	*6 (0)*	*0 (0)*	*0 (0)*	*0 (0)*
	Mongoose (M)	*0 (0)*	*0 (0)*	*6 (0)*	*0 (0)*	*0 (0)*	*0 (0)*
	Mongoose (A)	*0 (0)*	*0 (0)*	*6 (0)*	*0 (0)*	*0 (0)*	*0 (0)*
CHISEL	*bzip2*	*0 (0)*	*0 (0)*	*1 (0)*	*0 (0)*	*0 (0)*	*0 (0)*
	chown	0 (4)	0 (0)	1 (0)	0 (0)	0 (0)	0 (0)
	date	0 (4)	0 (0)	1 (0)	0 (0)	0 (0)	0 (0)
	grep	0 (4)	0 (0)	1 (3)	0 (0)	0 (0)	0 (0)
	gzip	*0 (0)*	*0 (0)*	*1 (0)*	*0 (0)*	*0 (0)*	*0 (0)*
	mkdir	0 (0)	0 (0)	1 (0)	0 (0)	0 (0)	0 (0)
	rm	0 (4)	0 (0)	1 (0)	0 (0)	0 (0)	0 (0)
	tar	0 (2)	0 (0)	**2 (-1)**	0 (0)	0 (0)	0 (0)
	uniq	0 (4)	0 (0)	1 (0)	0 (0)	0 (0)	0 (0)

6.2. Special Purpose Gadget Availability

Table 3 contains special purpose gadget availability data collected by GSA. The results measured according to this metric were similarly mixed. A conclusively positive result (complete elimination of all special purpose gadgets of some type) was observed in only 8 of 21 debloating scenarios. In eight other scenarios, debloating did not reduce the availability of special purpose gadgets (*italicized* text). In four scenarios, gadget introduction caused an increase in the count of special purpose gadgets (**bold** text). In one instance, debloating introduced a special purpose gadget of a type that was not available in the original package, a conclusively negative result. As was the case with functional gadget set expressivity, measuring special purpose gadget availability captured mixed and negative debloating results that were not observable using only gadget count reduction data.

7. Case Study

We demonstrate that our metrics are actionable through a case study of the debloating scenario `libcurl` (C), which resulted in a number of negative side effects:

- Increased the gadget set expressivity with respect to ASLR-Proof practical ROP exploits
- Increased the gadget set expressivity with respect to simple Turing-completeness
- Increased the number of system call gadgets

In order to mitigate these effects, we attempted to find a less aggressive debloating configuration that does not suffer these negative side effects. In this scenario, GSA was not able to identify the containing function for the newly introduced gadgets. In the absence of this information, we used alternate heuristics to decide what adjustments to make to mitigate the negative effects. We observed the effects debloating different sets of features had on other variants of `libcurl`; These negative effects were not observed in both the aggressive and moderate debloating scenarios. By analyzing the feature sets selected for each variant, we determined that a second debloating attempt for `libcurl` (C) was likely to yield better results if we did not debloat support for two protocol families (SCP and RTSP).

We made this modification to the debloating specification, re-ran CARVE, built the new variant, and analyzed the new variant with GSA. The results were largely an improvement, as expected:

- Reduced gadget set expressivity with respect to ASLR-proof practical ROP exploits
- No change in gadget set expressivity with respect to simple Turing-completeness
- Decreased the number of system call gadgets

However, changing the debloating configuration did result in one negative impact. In the new variant, we observed an increase in the number of COP intra stack pivot gadgets by one gadget. While this is not an ideal result, we determined that it did not offset the improvements realized in this second round of debloating. We manually analyzed the newly introduced gadget and determined that it was functionally identical to another such gadget included in both the original package and our newly created variant; therefore, the newly introduced gadget did not increase the overall gadget set utility.

7.1. Discussion

This case study highlights two important consequences of the unpredictable nature of gadget introduction worthy of further discussion. First, it may not be possible to debloat packages

with code-removing debloaters without incurring some negative impacts. Good security-oriented metrics should highlight these impacts and support the user in making informed tradeoffs when debloating, including choosing not to debloat.

Second, debloating does not have a linear relationship with security improvement. Techniques that debloat a larger portion of a package are not necessarily more effective at improving security than those that debloat less. Research into code-removing debloating techniques and tools should consider the problem of debloating in manner that minimizes or mitigates the negative side effects of gadget introduction.

8. Future Work

We have identified several areas worthy of future exploration. First, our proposed metrics are intended to generate discussion and research into other viable security-oriented metrics for software debloating, such as functional gadget quality and gadget locality. Additionally, there is a need for new analysis tools that calculate gadget set expressivity data. Immediate needs raised by this work include gadget set expressivity analysis for JOP and COP gadgets, as well as expressivity analysis for arbitrary length gadgets.

9. Conclusion

We presented examples across a variety of debloating scenarios demonstrating the flaws inherent to the gadget count reduction metric. Despite achieving sizeable gadget count reductions, our scenarios revealed that debloating can introduce new gadgets, including gadgets of high value like special purpose gadgets. We proposed two new security-oriented metrics, functional gadget set expressivity and special purpose gadget availability to replace the gadget count reduction metric. We demonstrated that these metrics overcome the limitations of gadget count reduction by identifying when the side effects of debloating negatively impact security. Finally, we showed the practicality and of these metrics by introducing our tool for calculating these metrics, GSA, and using it in a realistic case study.

Artifact Sharing Statement

GSA and a selected sample of binaries analyzed in this work have been made publicly available at:

https://github.com/michaelbrownuc/GadgetSetAnalyzer

Acknowledgements

We would like to thank all of our reviewers for their helpful feedback. We also thank Andrei Homescu and his co-authors for allowing us to use their microgadget scanner in this work. Finally, we thank Joshua Kassab for his assistance in gathering our experimental data.

References

[1] QUACH, A., ERINFOLAMI, R., DEMICCO, D., AND PRAKASH, A. A multi-OS cross-layer study of bloating in user programs, kernel, and managed execution environments. In *The 2017 Workshop on Forming an Ecosystem Around Software Transformation (FEAST)* (2017).

[2] CHEN, Y., SUN, S., LAN, T., AND VENKATARAMANI, G. TOSS: Tailoring online server systems through binary feature customization. In *The 2018 Workshop on Forming an Ecosystem Around Software Transformation (FEAST)* (2018).

[3] LEE, W., HEO, K., PASHAKHANLOO, P., AND NAIK, M. Effective Program Debloating via Reinforcement Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2018).

[4] SHARIF, H., ABUBAKAR, M., GEHANI, A., AND ZAFFAR, F. TRIMMER: Application specialization for code debloating. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)* (2018).

[5] QUACH, A., PRAKASH, A., AND YAN, L. Debloating software through piece-wise compilation and loading. In *Proceedings of the 27th USENIX Security Symposium* (2018).

[6] XU, G., MITCHELL, N., ARNOLD, M., ROUNTEV, A., AND SEVITSKY, G. Software bloat analysis: finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In *Proceedings of the FSE/DSP workshop on Future of Software Engineering Research (FoSER)* (2010).

[7] SHACHAM, H. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of 14th ACM conference on Computer and Communications Security (CCS)* (2007).

[8] Bletsch, T., Jiang, X., Freeh, V.W., and Liang, Z. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS)* (2011)

[9] SADEGHI, A., NIKSEFAT, S. AND ROSTAMIPOUR, M. Pure-call oriented programming (PCOP): chaining the gadgets using call instructions. In the *Journal of Computer Virology and Hacking Techniques* (2018).

[10] SCHWARTZ, E. J., AVGERINOS, T., AND BRUMLEY, D. Q: exploit hardening made easy. In *Proceedings of the 20th USENIX security symposium* (2011).

[11] ROEMER, R. G. Finding the bad in good code: automated return-oriented programming exploit discovery. *Master's Thesis, University of California* (2009).

[12] HOMESCU, A., STEWART, M., LARSEN, P., BRUNTHALER, S., AND FRANZ, M. Microgadgets: size does matter in turing-complete return-oriented programming. In *Proceedings of the 6th USENIX conference on offensive technologies (WOOT)* (2012).

[13] ZOVI, D. D. Practical return-oriented programming. SOURCE Boston, 2010. `https://trailofbits.files.wordpress.com/2010/04/practical-rop.pdf`.

[14] SALWAN, J. ROPgadget: Gadgets finder and auto-roper, 2011. `http://shell-storm.org/project/ROPgadget/`

[15] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and Communications Security (CCS)* (2005).

[16] ASPIRE PROJECT. CHISEL: A System for Debloating C/C++ Programs, 2019. `http://github.com/aspire-project/Chisel`

[17] ASPIRE PROJECT. ChiselBench, 2019. `https://github.com/aspire-project/ChiselBench`

[18] FOLLNER, A., BARTEL, A., AND BODDEN, E. Analyzing the gadgets: towards a metric to measure gadget quality. In *Proceedings of the International symposium on Engineering Secure Software and Systems (ESSoS)* (2016).

[19] SHOSHITAISHVILI, Y., WANG, R., SALLS, C, STEPHENS, N., POLINO, M., DUTCHER, A., GROSEN, J., FENG, S., HAUSER, C., KRUEGEL, C., AND VIGNA, G. (State of) The art of war: Offensive techniques in binary analysis. In *Proceedings of the IEEE Symposium on Security and Privacy* (2016).

[20] BROWN, M. D., AND PANDE, S. CARVE: Practical security-focused software debloating using simple feature set mappings. arXiv:1907.02180 [cs.CR] (2019).

[21] CHECKOWAY, S., DAVI, L., DMITRIENKO, A., SADEGHI, A., SHACHAM, H., AND WINANDY, M. Return-oriented programming without returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)* (2010).

[22] BHATTACHARYA, S., RAJAMANI, K., GOPINATH, K., AND GUPTA, M. The interplay of software bloat, hardware energy proportionality and system bottlenecks. In *Proceedings of the 4th workshop on power-aware computing and systems (HotPower)* (2011).