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 outline a set of measures for accurately assessing the security impact of software debloating with respect to gadget-based code reuse attacks. Further, we address complications in implementing these measures arising from the shortcomings of automated debloating models by proposing a security oriented human-in-the-loop model that overcomes these limitations.

1. Introduction

As software security breaches continue to increase in frequency, severity and sophistication [1], there has been a corresponding increase in the research and development of new methods for improving software security. One such method that has seen a surge of new research recently is software debloating [2, 3, 4, 5, 7].

Software bloat is the result of software engineering practices designed to make software modular, re-usable, and feature rich. While these practices enable the rapid development of complex, widely deployable software packages (user-level programs, shared libraries, etc.) they create software with large portions of code that are never used and are unnecessary in most end use contexts. This unnecessary portion of the package is called software bloat, and results in a variety of negative performance and security impacts [10, 11].

Software bloat affects virtually all software and primarily occurs vertically throughout the layers of the software stack across layers of abstraction [2]. For example, 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 in software packages suffering from feature creep [11]. Examples include software such as cUrl [9], which can be used to transfer data via 23 different protocols, and iTunes, which features a media player, media library manager, ecommerce platform, advertising portal, and hardware device interface within a single package. End users of cUrl and iTunes are unlikely to use every feature within these software packages, and the code associated with unused features contributes to software bloat.

In recent years, several software debloating techniques [3, 4, 5, 6, 7, 8] 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. Gadget count reduction is calculated by subtracting the total number of code reuse gadgets available in a debloated variant of a software package from the total number of code reuse gadgets available in the original package. Several recent publications in the software debloating corpus [4, 5, 7] make claims of improved security through software debloating 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 and decreases the likelihood of an attacker successfully constructing a code reuse exploit [4, 5, 7] using techniques such as return, jump, call, or data-oriented programming (ROP, JOP, COP or DOP [12, 13, 14, 15]). At face value, gadget count reduction is an appealing security improvement metric as it is easily generated using existing automated static analysis tools and directly relevant to a class of cyberattacks that have been the focus of intense research over the last decade [12-20, 23, 24, 28-36]. It also provides a direct means for comparing debloating techniques, and supports the prevailing argument that the more code that is debloated from a package, the more secure it becomes.
Unfortunately, the base premise linking gadget count reduction to improved security is inherently flawed due to the superficial and overly simplistic nature of gadget count reduction. Gadget count reduction captures only the change in the total number of gadgets available in a package after debloating. For an attacker attempting to construct a code reuse exploit, the total number of gadgets available is irrelevant; what truly matters to the attacker is whether or not the gadgets necessary to construct the desired exploit and maintain control flow are available. Recent advances in gadget chaining tools [16, 17, 18] and code reuse attack techniques [19] show that attackers do not require a large, diverse, and fully expressive set of gadgets in order to craft an exploit. Considering the large amounts of bloat present in software [2], it is possible that debloating can achieve high gadget count reduction yet fail to remove the gadgets an attacker needs. Worse yet, debloating can simultaneously remove gadgets that the attacker does not need and introduce new gadgets that the attacker does need, negatively impacting security.

In this paper, we challenge the value of using gadget count reduction as a security improvement metric for software debloating techniques. We present examples across a variety of debloating scenarios where sizeable gadget count reductions are achieved, but deeper analysis reveals that debloating failed to improve security, and in some cases caused measurable decreases in security against code reuse attacks. In order to overcome gadget count reduction’s failures, we propose a set of measures for gadget analysis to replace gadget count reduction as a metric and explore how these measures can be used in our scenarios to identify when debloating fails to improve security or negatively impacts it. We discuss concerns with implementing these measures in automated debloating models and propose a human-in-the-loop debloating model that incorporates them into an iterative process that utilizes human intuition and insight to improve security. To show the potential of this model, we demonstrate its effectiveness with a proof of concept.

The contributions of our work are organized as follows:

- In section 3, we explain our motivation through a discussion of the gadget count reduction metric and its use in software debloating research.
- In section 4, we describe in detail the approach and analysis methodology we used to generate our example debloated packages used in our case studies.
- In sections 5, 6, and 7, we demonstrate the flaws in the gadget count reduction metric by showing how software debloating techniques can result in sizeable gadget count reductions, but fail to improve security or even make it worse. Specifically, we show that “successful” debloating operations can result in newly introduced gadgets and increased availability of high-quality gadgets. Further, we show that software debloating can fail to limit the attacker’s ability to construct an exploit.
- In section 8, we summarize our observations and discuss their ramifications for software debloating. We propose a replacement for gadget count reduction that combines several measures to accurately assess the security impact of debloating operations and address concerns with implementing these measures.
- In section 9, we propose a human-in-the-loop debloating model that addresses the implementation concerns with our proposed measures. We discuss the advantages of this model over the existing automated model for security oriented debloating. We demonstrate that this model works in principle through a proof of concept.

2. Background

Code Reuse Attacks: Code reuse attacks are a class of cyberattacks 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 W\*X (Write XOR Execute). A commonly cited example of a code reuse attack is return-to-libc, in which the attacker typically exploits an unprotected buffer to redirect execution of a program to a sensitive function contained within libc. Return-to-libc attacks are limited in expressivity, as they do not inject arbitrary code but instead rely on the malicious execution of a library function.

Gadget-Based Code Reuse: To overcome the expressivity limitations of return-to-libc attacks, Shacham [12] proposed the first gadget-based code reuse attack method, called Return-Oriented Programming (ROP) in 2007. In gadget-based code reuse attack methods such as ROP, JOP, and COP (defined below), the attacker exploits a program in a similar manner to a return-to-libc attack, but rather than redirecting execution to a known function they chain together short instruction sequences called gadgets present in the program in a specific order to construct a malicious payload without injecting code. These attack methods have been shown to be Turing-complete if the attacker has access to a sufficiently expressive set of gadgets, allowing the attacker to construct and execute an arbitrary program.

Gadget: A gadget suitable for use in a code reuse attack is a sequence of instructions that end in a return, unconditional jump, or function call. When chained together using the control flow properties of the last instruction in each gadget, a chain of gadgets is equivalent to an executable program comprised entirely of existing code segments.

ROP: In ROP [12], the attacker takes control of the stack and chains gadgets ending in return instructions together to create a malicious payload. ROP Gadgets that make system calls are particularly useful in crafting exploits, and are called syscall
gadgets. Syscall gadgets belong to a class of gadgets called special purpose gadgets. Due to their unique nature and use across multiple code reuse attack types, we separate syscall gadgets into their own category in this paper.

**JOP/COP:** In JOP [13], the attacker chains gadgets ending in unconditional jump or call instructions together to create a malicious payload. This approach is not dependent on the stack and return instructions for control flow, and as such circumvents many ROP defenses. To control the flow of execution from gadget to gadget, JOP relies on a special purpose gadget called a dispatcher. The dispatcher maintains an ordered table of functional gadgets and ensures each functional gadget’s jump or call targets the dispatcher gadget after execution. COP [14] is a specialized variant of JOP, in which functional gadgets are limited to sequences ending in call instructions only. Gadget to gadget control flow in COP is accomplished using a dispatcher gadget similar to a JOP dispatcher, or through the use of other special purpose gadgets such as trampoline gadgets.

**DOP:** Other gadget-based code reuse attack methods such as DOP [15] have also been proposed. DOP differs from ROP, JOP, and COP in that it compromises data flow integrity as opposed to control flow integrity. In this paper, we limit our analysis to ROP, JOP and COP attack techniques.

### 3. Motivation

Security improvements have become one of the chief motivations for software debloating research. While early analysis of the software bloat problem was primarily focused on performance [10, 21], nearly all recent research into software bloat and debloating techniques cites security concerns with software bloat or states improving security as a motivating goal for debloating [2-8, 10, 11, 22]. As is typical with research in software security, justifying claims of improved security is difficult and the supply of security metrics relevant to software debloating are in short supply. While other metrics such as software diversity and vulnerability elimination have also been used to substantiate claims of increased security in debloated packages, gadget count reduction is increasingly being used as the metric of choice.

One driving force behind the adoption of this metric is the availability of static binary analysis tools such as ROPgadget [23] and Ropper [24] that can automatically scan a target binary for multiple types of code reuse gadgets, catalog them, and assist in chaining these gadgets into exploits. While these tools make it easy for researchers to quickly generate gadget count reduction data points in their experiments, these data points do not capture the complex effects debloating has on the gadgets present in a software package after debloating.

One of the key effects of debloating is the introduction of new gadgets. Debloating methods that remove instructions at the source, intermediate representation, or binary level introduce new gadgets into the debloated variant in a manner that is difficult to predict. Since gadgets can vary greatly in terms of usefulness to an attacker, debloating can potentially introduce valuable gadgets (including special purpose gadgets) while removing gadgets with little value.

Gadget count reduction also fails to capture how debloating impacts the attacker’s ability to construct exploits. The set of gadgets available in a package does not need to be large and diverse to contain a Turing-complete set of gadgets. Debloating does not necessarily produce a variant with less expressive power than the original program. For methods that introduce gadgets, it is even possible for debloating to increase expressive power.

Despite these flaws, recently proposed debloating techniques justify claims of improved security using gadget count reduction data as evidence. Our motivation is to highlight the issues surrounding the use of gadget count reduction as a security improvement metric and suggest an alternative set of measures to counter this trend. We briefly describe these techniques and their use of gadget count reduction:

**CHISEL:** Lee et al. [4] recently proposed an automatic method for debloating unnecessary features from program source code. This method, named 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, source code aware program reduction algorithm, CHISEL removes segments of the program that are not necessary to satisfy the desired properties. Across ten different programs, CHISEL reduced the total gadget count by 54.1% on average.

In a section entitled “Security Hardening”, the authors claim that this reduction reduces the potential code reuse attack surface. Their work cites the total gadget count reduction as the sole evidence of attack surface reduction, and their data does not indicate that gadget increases or introduction occurs. Further, the authors do not conduct analysis of how this reduction actually impacts a potential attacker.

The source code for CHISEL was made publicly available while we were conducting this research. Unfortunately, the authors did not include their debloated test cases or specification scripts with their source code, preventing us from replicating their published results and examining them at a deeper level. However, their source code for CHISEL does include 28 small toy programs (approximately 50 LOC or less) used as test cases for verifying that CHISEL is working properly. These test cases contain both the original and debloated toy programs. We built and analyzed these test cases and found that gadgets were introduced by debloating in 20 of the 28 test cases. In 6 of these test cases, the debloated program had a *higher* total count of gadgets. While these test cases are far from real-world examples, they show that CHISEL may in fact introduce new gadgets via debloating.
**TRIMMER**: Sharif et al. [5] recently proposed an automated method for deboating 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 report an average reduction in gadget count of 20% on 16 different case studies. The authors claim this reduction reduces the attack surface of a deboated program by reducing the number of exploitable gadgets. The authors provide gadget count reduction data in support of this claim; however, they provide no explanation of what makes a gadget exploitable as opposed to non-exploitable. Their data indicates that syscall gadgets were introduced as a result of deboating, yet no explanation or investigation of this occurrence is provided.

**PCL**: Quach et al. [7] recently proposed an automated technique for eliminating software bloat from shared libraries called Piece-wise Compilation and Loading (PCL). PCL performs what is described as a load time version of dead code elimination, in which a specialized compiler generates a dependency graph of external functions which is later used by a specialized loader to eliminate unreachable library code in memory at runtime. This code is deboated by marking dead pages as nonexecutable and individual dead functions as non-executable and rewriting them as invalid instructions. Across a variety of benchmark programs, PCL achieved an average total gadget count reduction of 71%. The authors state that gadget elimination hampers code reuse exploits in principle but do not determine if the gadgets removed actually impact the attacker’s ability to mount a code reuse attack with the remaining gadgets available.

### 4. Approach

#### 4.1. Deboater Selection and Operation

In order to highlight the flaws of gadget count reduction as a security metric for software deboating, we deboated several common software packages at varying levels of aggressiveness using our own custom deboater. We used our own deboater in this work as the other debloaters referenced in this paper are not publicly available, with the exception of CHISEL. We determined that CHISEL was not suitable for our work because of the high manual effort required to deboat programs in a sound manner. Additionally, the creators of CHISEL have not made their benchmark deboated programs or the test scripts used to create them publicly available, so we could not work directly with their deboater’s output to circumvent its limitations.

Our deboater operates in a comparable fashion to feature-focused approaches such as CHISEL and TRIMMER, in that it removes code associated with deboated features before generating the package binary. It takes as input a list of features to deboat and the program source augmented with a mapping of features to their associated source code. These mappings are generated by a human using custom preprocessor commands embedded in the source code in a manner similar to documentation generation tools such as Doxygen [38]. The mappings only need to be generated once, as they are left embedded in the source code, though they must be updated as new versions are created if changes affect a feature that is considered deboatable. The deboater itself is implemented as a preprocessor pass on the source code, which searches for embedded mappings associated with features to be deboated. When such a mapping is encountered, our deboater intelligently removes the source code specified by the mapping.

This approach allows the human operator the ability to reason directly and statically on the feature set to be deboated in source code with a high degree of precision and control. This contrasts CHISEL, in which a human operator must reason indirectly on the feature set to be deboated by writing a test scripts that dynamically exercise code associated with features that should be kept. For the purposes of this work, our deboater required less manual effort (however, we do not claim in this work that this is the case in general).

#### 4.2. Deboating Scenarios

In selecting software packages to deboat, we chose packages that varied in size, structure and operational complexity. The software packages we selected are:

- **libmodbus v3.1.4** [27], a software library implementing a common industrial network protocol.
- **Bftpd v4.9** [26], an FTP server utility program.
- **libcurl v7.61.0** [9], a data transfer utility library.

We created three different sets of features to deboat for each package corresponding to aggressive, moderate, and conservative deboating scenarios. We define these levels in common language:

- **Conservative**: Some peripheral features in the package are targeted for deboating.
- **Moderate**: Some peripheral features and some core features are targeted for deboating.
- **Aggressive**: All deboatable features except for a small set of core features are targeted for deboating.

These feature sets were selected to reflect reasonable real-world use cases. Detailed feature set information for each scenario is included in Appendix A.

For each combination of package and aggressiveness level, we ran the deboater and then built the deboated code using the default package build configuration. All of the software packages and their deboated variants were built on the same platform, a virtual machine running the 64-bit Ubuntu 18.04.1 LTS. GCC version 7.3.0 was used for each software build, and build configurations were kept constant for each
build with the exception of build flags associated with features that were debloated prior to compilation. We tested all debloated variants using a combination of package test scripts and custom testing scripts to ensure that the debloated packages were correct with respect to kept features, and that user input specifying debloated features did not cause the package to crash.

4.3. Analysis
First, we compared the gadgets available in the original package to the gadgets available in each of its debloated variants and used ROPgadget [23] to calculate the gadget count reduction achieved for each variant. We then compared the gadget count reduction values achieved by our debloater, and found that it performed on par with CHISEL, TRIMMER, and PCL. Aggressive debloating scenarios reduced the total number of gadgets by 30.2% on average, followed by moderate debloating scenarios at 15.2% on average and conservative debloating scenarios at 8% on average.

Next, we conducted a deeper analysis of the gadget sets in each debloated variant using advanced analysis tools such as IDA [25], Gality [36], and an advanced gadget scanner [19] to determine the impact debloating had on security. We focused our analysis on three security sensitive impacts that gadget count reduction fails to address: gadget introduction, gadget quality, and gadget set expressivity. We present the details of our analysis and findings in the following sections.

5. New Gadgets Introduced via Debloating
Attack surface reduction is the most commonly cited security benefit of gadget count reduction. Setting aside the fact that the term attack surface is misapplied (gadgets are not attack vectors or vulnerabilities, they are an abstract instruction set), attack surface reduction is generally considered to be good for security. However, debloating techniques that remove code from the package can introduce new gadgets into the debloated variant. When gadget introduction occurs, it alters the attack surface to cover a different area (even if it is smaller in overall size), which is not necessarily good for security. Since we do not have prior knowledge of which gadgets are useful to an attacker, gadget introduction can potentially make a debloated package more vulnerable to due to the possibility that gadgets the attacker does not need are removed and gadgets that they do need are introduced. Gadget count reduction masks this, providing a false sense of security even in cases where large reduction rates are obtained.

5.1. Introduction of Intended Gadgets
During compilation, source code is translated to an intermediate representation and optimized before the resulting binary is created during code generation. Gadgets formed by using the compiler generated binary instructions are referred to as intended gadgets.

Debloating techniques that remove or transform the source code of a software package can cause the downstream compiler stages to make different optimization and code generation decisions. This potentially results in the introduction of different instruction sequences, and by extension, new intended gadgets. Even in simple debloating operations, such as removing a short segment of code or a portion of static data, the resulting debloated binary code can differ from the original version in unexpected ways.

Consider the code snippet from libcurl shown in Figure 1. The curl_version function is used to generate a version string based in part on the version data from various dependent libraries. In the conservative debloating scenario for libcurl, support for the RTMP family of protocols is debloated resulting in the removal of the marked code segment. With respect to this single segment deletion, we expect that the resulting binary code will be the same as the original, less the instructions associated with segment. If true, then this function would contribute strictly fewer intended gadgets to the overall total. The actual result is more complicated. Removing the source code results in fewer instructions in the resulting binary as expected, however this shorter sequence of instructions is more easily optimized by the compiler. The result is the reordering of some instructions and simplification of the function’s control flow, shown in Figure 2. The instruction in bold italics indicates the return instruction that forms the end of a ROP gadget, and instructions in bold are common between both versions.

As shown in Figure 2, debloating the original version eliminates several instructions. In particular, the JMP instruction on line 19 and all but one instruction in the basic block following it are removed. As a result, the first and third basic blocks are merged into a single basic block in the debloated version. This change in locality increases the range of instructions (lines [1:11]) the compiler can reorder to maximize performance. In this case, the result is a significant change to the order of the instructions preceding the return instruction on line 11 in the debloated version. In the original version, the return instruction has three non-control flow instructions preceding it, but in the debloated version it is preceded by five non-control flow instructions. With respect to unique intended gadgets produced by these two sequences, the net result is an increase in unique gadgets. Two ROP gadgets (lines [9:11] and [10:11]) are present in both versions. One ROP gadget is eliminated from the original version (lines [24:27]), but two ROP gadgets are introduced in the debloated version (lines [7:11] and [8:11]).

Modern compilers are very complex, and it is impossible to predict the unintended effects of software debloating. In addition to the previous example, our analysis revealed a variety of potential sources of introduced gadgets caused by downstream compiler code generation and optimization (not an exhaustive list):
Debloating cases from a switch statement may alter which case falls through during execution, potentially altering the gadgets created by an instruction in the next block.

Eliminating branches via debloating (especially those close to an indirect branch or a procedure boundary) can alter control flow and introduce new gadgets.

Debloating entries in static arrays can trigger or prevent compile time optimizations for loops that iterate through these arrays. This has a significant effect on the code the compiler generates and the control flow used to implement the loop.

Inline function expansion can be triggered by debloating function call sites, altering control flow and potentially introducing new gadgets.

Debloating can trigger dead code elimination across basic blocks, resulting in changes to control flow such as the merging of basic blocks that can introduce new gadgets.

5.2. Introduction of Unintended Gadgets

As originally shown by Shacham [12], software packages contain gadgets that were not specifically generated by the compiler during code generation. 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 a different, but valid instruction sequence. Gadgets obtained from these instruction sequences are referred to as unintended gadgets.

Since many debloating techniques involve removing or transforming instructions in the source code, IR, or in the binary, the resulting debloated package binary is significantly altered. Even in the simplest case where code is removed entirely, new unintended gadgets are potentially introduced due to reinterpretation of code from unintended offsets.

For example, consider the sequence of instructions taken from the last basic block of the function `curl_version` shown at the top of Figure 3. When interpreted at an offset of one byte and three bytes after the first instruction boundary, unintended gadgets can be accessed.

In the aggressive debloating scenario for `libcurl`, this function has source code for determining the version numbers of libraries required only for undesired features removed. This affects the composition of the final basic block of this function for various reasons explained in the previous section, resulting in a slightly different final instruction sequence. In addition to the changes in intended gadgets, this change also causes a change in the unintended gadgets contained at different offsets, shown in Figure 4.
To capture these instances, gadget counts, even when broken down by type, do not capture gadget introduction instances where a removed gadget is replaced by an introduced gadget. To capture these instances, we compared the sets of unique gadgets available in each software package before and after debloating and identified gadgets present in the debloated package, but absent in the original package. This data is shown in Table 2. In this table, introduction rate is the proportion of introduced gadgets to the total gadgets in a variant.

The introduction of gadgets via debloating is neither a rare occurrence nor a limited one. In our scenarios, introduced gadgets account for at least 34.9% of the gadgets remaining after debloating in all software packages. In some cases, introduced gadgets account for over 50% of the gadgets remaining after debloating. Gadgets of all types were introduced at significant rates, the smallest of which was 31.5%, excluding syscall instances. Still, in the case of conservatively debloating libcurl, debloating increased the total number of syscall gadgets, and each gadget was newly introduced. On average, our data shows that debloating results in a variant that is comprised of 44.6% newly introduced gadgets. While it is possible that introduced gadgets do not negatively impact security, further analysis of the quality and expressivity of these gadgets is required to make that determination.

Table 1: Gadget counts data by type.

| Package Variant | ROP Gadgets | JOP Gadgets | COP Gadgets | Syscall Gadgets | Total Gadgets |
|-----------------|-------------|-------------|-------------|-----------------|---------------|
| libmodbus (O)   | 561         | 99          | 67          | 0               | 655           |
| libmodbus (C)   | 444         | 103         | 73          | 0               | 547           |
| libmodbus (M)   | 447         | 119         | 62          | 0               | 566           |
| libmodbus (A)   | 382         | 129         | 60          | 0               | 511           |
| Bftpd (O)       | 655         | 101         | 79          | 0               | 755           |
| Bftpd (C)       | 638         | 79          | 52          | 0               | 717           |
| Bftpd (M)       | 546         | 85          | 89          | 0               | 631           |
| Bftpd (A)       | 488         | 49          | 39          | 0               | 535           |
| libcurl (O)     | 4327        | 5219        | 4200        | 4               | 9536          |
| libcurl (C)     | 4155        | 5148        | 4248        | 5               | 9304          |
| libcurl (M)     | 3668        | 4389        | 3619        | 0               | 8057          |
| libcurl (A)     | 2923        | 2849        | 2476        | 4               | 5768          |
Gadgets are not created equal, and objectively measuring the relative quality of one gadget to another is easier said than done. There is limited prior work in assessing the relative quality of functional gadgets [36], and gadget quality can also vary widely based on external factors such as the implementation of defenses against certain types of attacks [28, 29, 30, 31, 32, 33]. However, importance of special purpose gadgets in constructing exploits make them an important measure of gadget quality that is directly affected by debloating.

Alternatives to ROP such as JOP and COP eliminate the need for stack-based control flow via return instructions by instead relying on special purpose gadgets such as dispatchers. These special purpose gadgets are rare and form the backbone for mounting these attacks, and must meet specific criteria. Without them, these exploits are not possible, and as such these gadgets are of higher value to an attacker.

### 6. Gadget Quality

In order to determine the effect debloating has on the availability of special purpose gadgets, we manually analyzed the set of JOP and COP gadgets in each of the debloated packages created in our scenarios to identify potential special purpose gadgets for both attack types (ROP gadget does not identify special purpose gadgets except for syscall gadgets).

Our results are shown in Table 3. This table does not include counts for every type of special purpose gadget; It excludes columns for special purpose gadget types that were not observed in the original package or any of its debloated variants. **Bold underlined** values indicate increases in special purpose gadget counts after debloating.

Even though software debloating generally reduced the availability of special purpose gadgets, there was only one instance where debloating eliminated all gadgets of a particular type. In three of the nine scenarios, including the only scenario in which elimination occurred, new special purpose gadgets were introduced, negatively impacting security.

Moderate debloating of **libcurl** eliminated all syscall gadgets, however debloating also introduced a COP trampoline gadget into **libcurl**, a gadget type that was not available in the original package. Additionally, there were two instances (**libcurl C and M**) in which the count of a special purpose gadget type increased. As a result, we cannot claim that debloating will generally deprive an attacker of a previously available special purpose gadget type. It is also not possible to claim any significant increase in security based on this data, as we cannot know if the gadgets we removed will prevent an attacker from mounting an exploit or simply require them to use an alternate gadget of the same type.

The introduction of special purpose gadgets represents a serious security concern. Despite the fact that each debloating scenario resulted in significant gadget count reduction rates, in no scenario can we state for certain that debloating eliminated the population of special purpose gadgets without negative side effect.

### 7. Gadget Expressivity

When constructing an exploit, an attacker uses functional gadgets as abstract instructions that perform basic operations such as addition, register loading, and logical branching to construct a malicious payload. The expressivity of a given set of gadgets is a measure of the computational power the set of gadgets permit. The expressive power of a given 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.

In practice, a software package typically contains many gadgets that perform the same class of computation. Unless a debloating operation removes all gadgets that perform a certain computation, the expressivity of the gadget set is not affected. Due to the relatively low number of gadgets necessary to construct an exploit and the relatively large size of software packages [19], it is possible for debloating to achieve high gadget count reduction rates yet fail to reduce the expressivity of a gadget set. In cases where new gadgets are introduced, debloating may increase expressivity.

---

| Debloated Variant | Total Gadgets Introduced | Intro Rate | ROP Gadgets Introduced | Intro Rate | JOP Gadgets Introduced | Intro Rate | COP Gadgets Introduced | Intro Rate | Syscall Gadgets Introduced | Intro Rate |
|-------------------|--------------------------|------------|------------------------|------------|------------------------|------------|------------------------|------------|--------------------------|------------|
| libmodbus (C)     | 221                      | 40.4%      | 163                    | 36.7%      | 58                     | 56.3%      | 56                     | 76.7%      | 0                       | 0.0%       |
| libmodbus (M)     | 222                      | 39.2%      | 149                    | 33.3%      | 73                     | 61.3%      | 45                     | 72.6%      | 0                       | 0.0%       |
| libmodbus (A)     | 250                      | 48.9%      | 156                    | 40.9%      | 94                     | 72.9%      | 44                     | 73.3%      | 0                       | 0.0%       |
| Bftp (C)          | 250                      | 34.9%      | 201                    | 31.5%      | 49                     | 62.0%      | 42                     | 80.8%      | 0                       | 0.0%       |
| Bftp (M)          | 259                      | 41.0%      | 196                    | 35.9%      | 63                     | 74.1%      | 79                     | 88.8%      | 0                       | 0.0%       |
| Bftp (A)          | 195                      | 36.4%      | 166                    | 34.0%      | 31                     | 63.3%      | 30                     | 76.9%      | 0                       | 0.0%       |
| libcurl (C)       | 4735                     | 50.9%      | 1859                   | 44.7%      | 2863                   | 55.6%      | 2275                   | 53.6%      | 5                       | 100.0%     |
| libcurl (M)       | 4190                     | 52.0%      | 1630                   | 44.4%      | 2556                   | 58.2%      | 2059                   | 56.9%      | 0                       | 0.0%       |
| libcurl (A)       | 3350                     | 58.0%      | 1422                   | 48.6%      | 1927                   | 67.6%      | 1736                   | 70.1%      | 4                       | 100.0%     |

---

| Libcurl C          | 1422                     | 52.0%      | 1859                   | 44.7%      | 58                     | 56.3%      | 56                     | 76.7%      | 0                       | 0.0%       |
| Libcurl M          | 1950                     | 50.9%      | 1630                   | 44.4%      | 73                     | 61.3%      | 45                     | 72.6%      | 0                       | 0.0%       |
| Libcurl A          | 3350                     | 58.0%      | 1422                   | 48.6%      | 94                     | 72.9%      | 44                     | 73.3%      | 0                       | 0.0%       |
| Bftp C             | 250                      | 34.9%      | 201                    | 31.5%      | 49                     | 62.0%      | 42                     | 80.8%      | 0                       | 0.0%       |
| Bftp M             | 259                      | 41.0%      | 196                    | 35.9%      | 63                     | 74.1%      | 79                     | 88.8%      | 0                       | 0.0%       |
| Bftp A             | 195                      | 36.4%      | 166                    | 34.0%      | 31                     | 63.3%      | 30                     | 76.9%      | 0                       | 0.0%       |
| Libcurl C          | 4735                     | 50.9%      | 1859                   | 44.7%      | 2863                   | 55.6%      | 2275                   | 53.6%      | 5                       | 100.0%     |
| Libcurl M          | 4190                     | 52.0%      | 1630                   | 44.4%      | 2556                   | 58.2%      | 2059                   | 56.9%      | 0                       | 0.0%       |
| Libcurl A          | 3350                     | 58.0%      | 1422                   | 48.6%      | 1927                   | 67.6%      | 1736                   | 70.1%      | 4                       | 100.0%     |
Turing-completeness represents the high-water mark of computational expressivity and is not difficult to achieve in practice. Handpicked sets of ROP, JOP, COP, and DOP gadgets have all been shown to be Turing-complete [12, 13, 14, 15], establishing that Turing-completeness is achievable using any of these attack types. Further, it has been shown that versions of libc and many different binaries in common Linux distributions contain Turing-complete sets of ROP gadgets [12, 19]. Practical ROP exploits have been proposed that do not require a Turing-complete level of expressivity [19]. These exploits use gadgets to mark a region of memory as writable, inject an arbitrary payload into this region, and then branch execution to the region [20].

### 7.1. Analysis and Results

Determining the expressivity of a gadget set is typically a difficult and manual process as there are not tools widely available for this purpose. However, Homescu et al. [19] recently proposed and implemented a tool for automatically discovering and classifying sets of short ROP gadgets (referred to as microgadgets) in order to determine their expressive power.

In order to assess the impact of debloating on gadget set expressivity, we analyzed our binaries before and after debloating in order to determine their expressive power. Weexclude gadgets contributed by package dependencies, and in the case of libraries such as libcurl and libmodbus, gadgets from user level programs that would link these libraries.

We analyzed each debloating scenario with respect to three levels of expressivity: simple Turing-completeness, the expressivity required to construct a practical ROP exploit such as the exploit described in previous section, and expressivity required to create a practical, ASLR [37] resistant ROP exploit. Each level of expressivity requires a different number of gadget classes which correspond to different computational operations. There are 17 classes required for simple Turing-completeness, 11 classes required for practical ROP exploits, and 35 classes required for ASLR-proof practical ROP exploits.

The microgadget scanner [19] statically analyzes the variant binary to find gadgets and determine which classes they satisfy. To achieve a particular level of expressivity, a set of gadgets must have at least one gadget per class. Since we include gadgets contributed by external libraries and referencing programs in our analysis, we refer to our data in Table 4 as the package’s partial contribution towards expressivity. Bold underlined values indicate increases after debloating.

### Table 3: Availability of JOP and COP special purpose gadgets before and after debloating.

| Package Variant | JOP Dispatcher Gadgets | JOP Load Data Gadget | COP Dispatcher Gadgets | COP Trampoline Gadgets | COP Intra Stack Pivot Gadgets | Syscall Gadgets |
|-----------------|------------------------|----------------------|------------------------|------------------------|-----------------------------|----------------|
| libmodbus (O)   | 0                      | 1                    | 0                      | 0                      | 0                           | 0              |
| libmodbus (C)   | 0                      | 1                    | 0                      | 0                      | 0                           | 0              |
| libmodbus (M)   | 0                      | 2                    | 0                      | 0                      | 0                           | 0              |
| libmodbus (A)   | 0                      | 1                    | 0                      | 0                      | 0                           | 0              |
| Bftpd (O)       | 0                      | 10                   | 0                      | 0                      | 0                           | 0              |
| Bftpd (C)       | 0                      | 9                    | 0                      | 0                      | 0                           | 0              |
| Bftpd (M)       | 0                      | 1                    | 0                      | 0                      | 0                           | 0              |
| Bftpd (A)       | 0                      | 1                    | 0                      | 0                      | 0                           | 0              |
| libcurl (O)     | 8                      | 53                   | 320                    | 0                      | 4                           | 4              |
| libcurl (C)     | 7                      | 51                   | 304                    | 0                      | 3                           | 5              |
| libcurl (M)     | 4                      | 41                   | 287                    | 1                      | 3                           | 0              |
| libcurl (A)     | 3                      | 14                   | 175                    | 0                      | 2                           | 4              |

### Table 4: Proportion of gadget classes fulfilled by variant.

| Package Variant | Non-ASLR Practical Set | ASLR Proof Practical Set | Simple Turing Complete Set |
|-----------------|------------------------|--------------------------|----------------------------|
| libmodbus (O)   | 6 of 11                | 13 of 35                 | 7 of 17                    |
| libmodbus (C)   | 6 of 11                | 13 of 35                 | 7 of 17                    |
| libmodbus (M)   | 6 of 11                | 10 of 35                 | 6 of 17                    |
| libmodbus (A)   | 6 of 11                | 13 of 35                 | 7 of 17                    |
| Bftpd (O)       | 6 of 11                | 15 of 35                 | 7 of 17                    |
| Bftpd (C)       | 7 of 11                | 12 of 35                 | 6 of 17                    |
| Bftpd (M)       | 7 of 11                | 17 of 35                 | 6 of 17                    |
| Bftpd (A)       | 6 of 11                | 11 of 35                 | 5 of 17                    |
| libcurl (O)     | 9 of 11                | 25 of 35                 | 10 of 17                   |
| libcurl (C)     | 9 of 11                | 26 of 35                 | 11 of 17                   |
| libcurl (M)     | 9 of 11                | 24 of 35                 | 10 of 17                   |
| libcurl (A)     | 10 of 11               | 24 of 35                 | 10 of 17                   |

In general, software debloating did not have a large effect on the expressivity of the debloated gadget sets. The majority (44.4%) of data points saw no change in the number of gadget classes satisfied, and only 33% of data points saw a reduction in classes satisfied. Surprisingly, 22% of data points saw an increase in gadget classes satisfied after debloating. Increases in satisfied classes were noted at all aggressiveness levels. Surprisingly, there were no debloating scenarios in which a reduction of classes satisfied for all three expressivity levels was observed. Only one third of the debloating scenarios resulted in a clear security improvement (at least one reduction in the number of satisfied classes and no increases). In the
remaining two-thirds of scenarios, deboating did not reduce the partial contribution of the package or resulted in at least one increase in partial contribution. In these scenarios, security cannot be considered improved as result of deboating. Our data suggests that deboating does not usually hinder the expressive power of a software package with respect to this measure of expressivity. We observed that it was fairly common for deboating to fail to improve security or negatively impact it. Our suspicion is that larger software packages will see this effect exaggerated, due to the larger starting pool of available gadgets. It is also worth noting that all three of these packages link libc across all deboating scenarios. Libc has been shown to contain a Turing-complete set of ROP gadgets [12] without the partial contributions of these packages, further undermining the claimed security benefits of deboating techniques that do not target shared libraries.

7.2. Limitations

The results of this analysis are limited to ROP gadget sets, however this approach can be extended to assess the expressivity of sets of other gadget classes. We leave this as future work. Also, it should be noted that the microgadget scanner searches for specific sets of short gadgets to determine if an expressivity level is reached. It does not consider all gadgets, and as a result a gadget set may be more expressive than reported by this tool if longer gadgets are included in analysis.

8. Discussion

8.1. Applicability of Our Findings

We could not directly compare our findings with respect to these deeper analyses to the other deboaters referenced in this paper, as they and/or their benchmarks are not publicly available. Our deboater does not operate in a manner that suggests our findings are limited to our implementation only. This is supported by the fact that CHISEL introduces gadgets in their test cases and TRIMMER reports gadget introduction in their work. Since PCL rewrites instructions in memory rather than removing them, it does not introduce new gadgets. However, our findings with respect to gadget set expressivity still apply to PCL (as well as CHISEL and TRIMMER).

8.2. Summary of Findings

We summarize the full results of our analysis in Table 5. For each scenario, we made an overall assessment of the impact deboating had on security by considering the results of our previous analyses in aggregate. Scenarios assessed to have neutral overall impact had no clear positive or negative impact on security in our analyses. Scenarios in which both positive and negative impacts were observed are assessed to have mixed overall impact. Scenarios in which strictly positive or negative impacts were observed are labelled accordingly. The results of our aggregate assessment clearly run counter to the prevailing notion that gadget count reduction is a useful metric for measuring the security impact of software deboating. While each scenario resulted in positive gadget count reduction results, there was only one scenario (aggressive deboating of `Bftpd`) in which deboating resulted in a clearly positive security impact. Likewise, only one scenario (conservative deboating of `libcurl`) resulted in a clearly negative impact on security. The other seven scenarios had either no impact on security or mixed results.

Had we only analyzed our deboated variants for gadget count reduction, we would have misinterpreted the success of deboating as wholly positive. The serious flaws in the gadget count reduction metric would have led to a superficial sense of security in most cases, and in some a false one.

8.3. Assessing the Security Impact of Software Deboating

Accurately assessing the security impact of deboating requires a deeper and multifaceted analysis than gadget count reduction provides. We enumerate and describe a set of core measures introduced in this work that should be captured when assessing the security impact of a deboating operation:

| Package Variant | Gadget Count Reduction Rate | Gadget Intro Rate | Gadget Type Availability Impact | Special Purpose Gadget Availability Impact | Partial Expressivity Impact | Overall Impact Assessment |
|-----------------|-----------------------------|------------------|---------------------------------|-------------------------------------------|-----------------------------|--------------------------|
| libmodbus (C)   | 16.5%                       | 40.4%            | ROP ↓ JOP, COP ↑               | None Eliminated                           | None                        | Neutral                  |
| libmodbus (M)   | 13.6%                       | 39.2%            | ROP, COP ↓ JOP ↑               | JOP Load Data Increase                    | ASLR, Turing ↓             | Mixed                    |
| libmodbus (A)   | 22.0%                       | 48.9%            | ROP, COP ↓ JOP ↑               | None Eliminated                           | ASLR, Turing ↓             | Mixed                    |
| Bftpd (C)       | 5.0%                        | 34.9%            | ROP, JOP, COP ↓                | None Eliminated                           | ASLR, Turing ↓             | Mixed                    |
| Bftpd (M)       | 16.4%                       | 41.0%            | ROP, JOP ↓ COP ↑               | None Eliminated                           | Turing ↓                   | Mixed                    |
| Bftpd (A)       | 29.1%                       | 36.4%            | ROP, JOP, COP ↓                | None Eliminated                           | ASLR, Turing ↓             | Mixed                    |
| libcurl (C)     | 2.4%                        | 50.9%            | ROP, JOP ↓ COP ↑               | Syscall Increased                         | ASLR, Turing ↑             | Negative                 |
| libcurl (M)     | 15.5%                       | 52.0%            | ROP, JOP, COP ↓                | COP Trampoline Introduced Syscall Eliminated | ASLR ↓                     | Mixed                    |
| libcurl (A)     | 39.5%                       | 58.1%            | ROP, JOP, COP ↓                | None Eliminated                           | ASLR ↓ Non-ASLR ↑          | Mixed                    |

Table 5: Summarized impacts of deboating on security. ↓ indicates decreased value, ↑ indicates increased value.
Gadget Count Reduction by Gadget Type: For this measure, gadget count reduction counts should be calculated for all functional gadget types (ROP, JOP etc.) as well as special purpose gadget types (syscall, dispatcher, etc.). This fine-grained extension of gadget count reduction is useful for determining if debloating has increased or decreased the availability of a particular gadget type (including high quality gadgets).

Gadget Introduction Rate by Gadget Type: The rate of gadget introduction should be calculated for all functional gadget types as well as special purpose gadget types. Determining the rate of gadget introduction is useful for identifying the change in diversity of gadgets caused by debloating.

Partial Expressivity of Functional Gadgets by Type: For this measure, the partial expressivity of the sets of functional gadgets for each type (ROP/JOP/COP) should be determined for all relevant levels of expressivity. Determining the partial expressivity of a gadget set is a useful measure of how debloating impacts the attacker’s ability to craft an exploit.

Contributions by Dependencies: Dependent libraries that are not statically linked to the package contribute to gadget counts, the availability of special purpose gadgets, and the expressivity of the available gadgets at runtime. Analyzing these dependencies is necessary to obtain a holistic view. We do not claim that the measures we have proposed are comprehensive. On the contrary, we encourage the extension of this set of measures with more advanced gadget-based security analyses. One relevant open research area that we leave as future work is relative functional gadget quality. Functional gadgets are not created equal, and as such some gadgets are more useful to an attacker than others. Some research in this area has been conducted [36], however it is specific to a certain class of exploits and is not generally applicable.

8.4. Implementation Concerns
The primary concern with implementing these proposed measures is scalability. Currently, manual effort is required to identify some special purpose gadget types and tools for determining the expressivity of a gadget set are limited in scope and availability. These scalability concerns can be addressed by developing more advanced static analysis or refining existing tools like ROPgadget [23] and gadget chaining tools [16, 17, 18].

Of greater concern for scalability are the resources necessary to run these tools for each unique debloating operation. While these analyses are generally not computationally intensive, it may be infeasible to address all of these measures for large packages and their dependencies in a reasonable timeframe. A secondary concern for implementing these measures is determining what actions should be taken if debloating fails to improve security. Gadget count reduction data in recent work suggested that security improved monotonically with the aggressiveness of the debloating operation, and as such remediation options for failed debloating runs have not been addressed by the research community. Our research indicates that the relationship between debloating and security impact is not monotonic, and naively reverting back to the original package when debloating fails may miss opportunities to improve security using alternative debloating configurations. Options to address both of these outstanding concerns are limited due to the nature of the automated debloating model employed by current techniques. In the automated model, inputs are static and debloating is a single use, pipelined, mechanical process. This model works well for applications like compilers, however it is not well suited for applications that are situationally dependent like security.

9. A Human-in-the-Loop Model for Debloating
In this section, we propose a new model for security oriented debloating that leverages human knowledge and intuition to overcome the limitations of the automated debloating model. In our model, shown in Figure 5, one or more debloating techniques are applied at different stages in the software lifecycle. The resulting debloated variant is then built and analyzed by a human with the support of automatic, security oriented static analysis tools implementing our proposed measures. The human has two primary functions in this model. First, the human determines which analyses are relevant in the given scenario, prioritizes them by importance, and uses them to generate decision support information. This addresses the scalability concerns of multifaceted gadget analysis by using human intuition to determine which analyses are most useful in a given debloating scenario. For example, if the software package is to be built using ASLR, the human can limit the gadget expressivity analysis to ASLR proof levels of expressivity only. Additionally, the human can short-circuit this stage if certain conditions that are cause for immediate rejection (such as the introduction of syscall gadgets) are present, conserving resources for remediation efforts.

Second, the human uses this generated data to determine the impact the debloating operation had on security and what further actions to take. At this decision point, the human can make appropriate tradeoffs according to their high-level security goals. If debloating had negative effects on security, the human can choose to revert to the original package, or if resources permit, perform another iteration with a modified debloating configuration. If debloating was successful or had no effect on security, the human can choose to accept the debloated variant, or if resources permit, perform another iteration.

9.1. A Human-in-the-Loop Proof of Concept
As a proof of concept, we applied our human-in-the-loop model to the libcurl conservative debloating scenario. We
selected this scenario as it was observed to have an overall negative impact on security due to an increase in the availability of syscall gadgets and two levels of partial expressivity. Taking the role of the human analyst in this loop, we decided to run a second debloating iteration with a configuration that debloated three fewer features (RTSP, TFTP, and SCP) than the first iteration. The decision to keep these features rather than debloat them in the second iteration was driven by human intuition. We observed that these features had the greatest deal of interaction with other features not selected for debloating, and we hypothesized that they were causing a significant amount gadget introduction. After running the debloater and building the new variant, we analyzed it using the same measures utilized in the first iteration. The full results of this analysis follow:

- Gadget counts: Total increased to 9594; ROP increased to 4371; JOP increased to 5224; COP increased to 4247, Syscall decreased to 3.
- Gadget introduction rates: Total: 50.5%; ROP: 45.1%; JOP: 54.8%; COP: 52.2%; Syscall 100%.
- Special purpose gadget counts: JOP Dispatcher decreased to 7; COP Data Load decreased to 45; COP Dispatcher decreased to 306; COP trampoline and stack pivot: no change; Syscall decreased to 3.
- Partial expressivity of gadget sets: Non-ASLR Practical: 9 of 11; ASLR Proof Practical: 24 of 35; Simple Turing Complete: 10 of 17.

We then summarized these results and compared them to those obtained in the first iteration. These results are shown in Table 6, items in **bold underlined** text indicate key differences. While the debloated variant created in the second iteration ended up with a higher total gadget count and higher gadget type counts for ROP, JOP, and COP gadgets, improvements over the first iteration were realized where it really matters. Instead of increasing the number of syscall gadgets available, the second debloating iteration reduced the count. Additionally, the second iteration reduced the partial expressivity of the set of available gadgets with respect to one expressivity level, rather than raising it with respect to two as was the case in the first iteration. Further, no special purpose gadget introductions were observed. As a result, the second iteration of debloating had an overall positive impact on security as opposed to the negative impact the first iteration had despite its poor performance according to the gadget count reduction metric. Having achieved our goals of improving security through debloating, we decided to accept this variant.

### 9.2. Compatibility and Extension

Our proposed model wholly contains the automated debloating model is therefore compatible with and can leverage multiple existing automated debloating techniques. In the worst case, the resource budget for human analysis and iteration is zero, the human-in-the-loop debloating model reduces to the automated debloating model. Additionally, our debloating model is sufficiently general to permit extension. For example, this model can be extended to consider additional software security improvement metrics like known vulnerability...
elimination by incorporating additional tools into the security impact assessment phase. Finally, this model can be used for orthogonal concerns in software debloating, such as correctness and soundness verification of debloated software variants. We leave the development, enrichment and extension of this model as future work.

10. 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. Our scenarios also revealed that debloating does not necessarily eliminate the availability of special purpose gadgets or reduce the partial expressivity of the set of gadgets available in a software package. We proposed a set of gadget analysis measures to replace the gadget count reduction metric that can overcome the limitations we identified in our examples by identifying when debloating fails to improve security or makes it worse. Finally, we discussed the difficulties in implementing these measures in automated debloating models and proposed an iterative human-in-the-loop debloating model well suited for these measures.

A. Appendix

The following tables detail which high level software features were debloated in each scenario referenced in this paper. For brevity, the complete list of fine grained debloatable features in the package are condensed into the categories in the leftmost column. For each scenario, our debloater removed the code associated with features marked with an X in that scenario’s column.

### Table A.1: Debloated features per scenario for libmodbus.

| Debloatable Feature | Conservative Scenario | Moderate Scenario | Aggressive Scenario |
|---------------------|-----------------------|-------------------|---------------------|
| RTU Read Operations | X                     | X                 |                     |
| RTU Write Operations| X                     | X                 |                     |
| RTU Raw Operations  | X                     | X                 |                     |
| TCP (IPv4) Read Operations | X     | X                 |                     |
| TCP (IPv4) Write Operations | X     | X                 |                     |
| TCP (IPv4) Raw Operations | X     | X                 |                     |
| TCP (IPv4/6) Read Operations | X     | X                 |                     |
| TCP (IPv4/6) Write Operations | X     | X                 |                     |
| TCP (IPv4/6) Raw Operations | X     | X                 |                     |

### Table A.2: Debloated features per scenario for Bftpd.

| Debloatable Feature          | Conservative Scenario | Moderate Scenario | Aggressive Scenario |
|------------------------------|-----------------------|-------------------|---------------------|
| Admin Commands               | X                      | X                 | X                   |
| Read Commands                |                        |                   |                     |
| Write Commands               |                        |                   | X                   |
| Directory Commands           | X                      | X                 |                     |
| Server Config Commands       | X                      | X                 |                     |
| Miscellaneous Commands       | X                      | X                 |                     |
| Server Info Commands         |                        |                   | X                   |

### Table A.3: Debloated features per scenario for libcurl.

| Debloatable Feature          | Conservative Scenario | Moderate Scenario | Aggressive Scenario |
|------------------------------|-----------------------|-------------------|---------------------|
| Uncommon API Elements        | X                      | X                 | X                   |
| HTTP                         |                        |                   |                     |
| HTTPS                        |                        |                   |                     |
| RTSP                         | X                      | X                 |                     |
| FTP Read Commands            |                        |                   |                     |
| FTP Write Commands           | X                      | X                 |                     |
| FTPS                         | X                      | X                 |                     |
| Telnet                       | X                      | X                 | X                   |
| LDAP                         |                        |                   |                     |
| TFTP                         | X                      | X                 |                     |
| IMAP                         | X                      | X                 |                     |
| SMB                          | X                      | X                 |                     |
| SMTP                         | X                      | X                 |                     |
| POP3                         | X                      | X                 | X                   |
| RTMP                         | X                      | X                 | X                   |
| File                         | X                      | X                 | X                   |
| Gopher                       | X                      | X                 | X                   |
| Dict                         | X                      | X                 | X                   |
| SCP                          | X                      | X                 | X                   |
| SFTP                         | X                      | X                 |                     |
References

[1] University of Cambridge Centre for Risk Studies. Cyber Risk Outlook 2018. https://www.jbs.cam.ac.uk/fileadmin/user_upload/research/centres/risk/downloads/crs-cyber-risk-outlook-2018.pdf.

[2] 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).

[3] 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).

[4] Lee, W., Heo, K., Pashakhanloo, P., and Naik, M. Effective Program Debolating via Reinforcement Learning. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS) (2018).

[5] 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).

[6] Malecha, G., Gehani, A., and Shankar, N. Automated software winnowing. In Proceedings of the 30th Annual ACM Symposium on Applied Computing (SAC) (2015).

[7] Quach, A., Prakash, A., and Yan, L. Debolating software through piece-wise compilation and loading. In Proceedings of the 27th USENIX Security Symposium (2018).

[8] Jang, Y., Wu, D., and Liu, P. JRed: Program customization and bloateware mitigation based on static analysis. In Proceedings of the 2016 IEEE 40th annual Computer Software and Applications Conference (COMPSAC) (2016).

[9] cUrl: command line tool and library for transferring data with URLs, 2018. https://curl.haxx.se.

[10] 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).

[11] Jang, Y., Zhang, C., Wu, D., and Liu, P. Feature-based software customization: preliminary analysis, formalization, and methods. In 2016 IEEE 17th international symposium on High Assurance Systems Engineering (HASE) (2016).

[12] 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).

[13] Blelch, 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 (ASI-ACCS) (2011).

[14] Sadeghi, A., Niksefat, S. and RostamiPourn, M. Pure-call oriented programming (PCOP): chaining the gadgets using call instructions. In the Journal of Computer Virology and Hacking Techniques (2018).

[15] Hu, H., Shinde, S., Adrian, S., Chua, Z.L., Saxena, P., and Liang, Z. Data-oriented programming: on the expressiveness of non-control data attacks. In Proceedings of the 2016 IEEE symposium on Security and Privacy (SP) (2016).

[16] Dullien, T., Kornau, T., and Weinmann, R. A framework for automated architecture-independent gadget search. In Proceedings of the 4th USENIX conference on offensive technologies (WOOT) (2010).

[17] Schwartz, E. J., Avgéronos, T., and Brumley, D. Q: exploit hardening made easy. In Proceedings of the 20th USENIX security symposium (2011).

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

[19] Homescu, A., Stewart, M., Larsen, P., Brunkhaler, 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).

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

[21] 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).
[22] HONG, H. System debloating via compile-time configuration and hybrid binary rewriting. FEAST, 2018. https://feastworkshop.github.io/program.html

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

[24] SCHIRRA, S. Ropper: rop gadget finder and binary information tool, 2018. https://scoding.de/ropper/

[25] IDA, 2018. https://www.hex-rays.com/products/ida/index.shtml

[26] Bftpd: A fast, secure, easy to configure FTP server, 2018. http://bftpd.sourceforge.net/

[27] libmodbus: a Modbus library for Linux, Mac OS X, FreeBSD, QNX and Win32, 2018. https://libmodbus.org/

[28] DAVI, L., SADEGHI, A., AND WINANDY, M. Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks. In Proceedings of the 2009 ACM workshop on Scalable Trusted Computing (STC) (2009).

[29] CHEN, P., XIAO, H., SHEN, X., YIN, X., MAO, B., AND XIE, L. DROP: Detecting return-oriented programming malicious code. In Proceedings of the International Conference on Information Systems Security (ICISS) (2009).

[30] LI, J., WANG, Z., JIANG, X., GRACE, M., AND BAHRAM, S. Defeating return-oriented rootkits with “return-less” kernels. In Proceedings of the 5th European conference on computer systems. (2010)

[31] DAVI, L., SADEGHI, A., AND WINANDY, M. ROPdefender: a detection tool to defend against return-oriented programming attacks. In Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS) (2011).

[32] 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).

[33] DAVI, L., SADEGHI, A., LEHMANN, D., AND MONROSE, F. Stitching the gadgets: on the effectiveness of coarse-grained control-flow integrity protection. In Proceedings of the 23rd USENIX security symposium (2014).

[34] YAO, F., CHEN, J., AND VENKATARAMANI, G. JOP-alarm: detecting jump-oriented programming-based anomalies in applications. In Proceedings of the IEEE 31st International Conference in Computer Design (ICCD) (2013).

[35] KAYAALP, M., OZSOY, M., ABU-GHAZALEH, N., AND PONOMAREV, D. Branch regulation: low-overhead protection from code reuse attacks. In Proceedings of the 39th annual International Symposium on Computer Architecture (ISCA) (2012).

[36] 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).

[37] Address Space Layout Randomization, 2003. https://pax.grsecurity.net/docs/aslr.txt

[38] Doxygen: Generate documentation from source code, 2018. http://www.doxygen.nl