Bloat Factors and Binary Specialization

Anh Quach, Aravind Prakash
Binghamton University
{aquach1,aprakash}@binghamton.edu

ABSTRACT

Code bloating in software has been proven to be pervasive in recent research. However, each study provides a different approach to measure bloat. In this paper, we propose a system of metrics to effectively quantify bloat in binaries called bloat factors. Subsequently, we conducted an extensive study to calculate bloat factors for over 3000 Linux applications and 896 shared libraries. Using these metrics as pointers, we introduce a static approach to perform debloating for closed-source binaries by creating corresponding specialized versions to cater for a specific program requirements. We evaluated our debloating technique on large programs and achieved a maximum code reduction of 19.7%.

KEYWORDS

debloat; binary analyis

ACM Reference Format:
Anh Quach, Aravind Prakash. 2019. Bloat Factors and Binary Specialization. In *3rd Workshop on Forming an Ecosystem Around Software Transformation (FEAST'19), November 15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3338502.3359765

1 INTRODUCTION

With the dramatic increase in application size and complexity, code bloating has become a prominent and prevalent problem in the modern software scene. This poses as a negative effect on both performance and security of software as extraneous code is a valuable source of gadgets and vulnerabilities for exploits.

Code bloating can happen in both program as well as shared libraries or Dynamic-link library (DLL). Bloat in software happens when developers excessively pack it with features that may not be used by users while providing them with little to no ability to remove unwanted functionalities. For example, a web server program may be used to host only a particular website that may not require all features that come with it. At the same time, bloat in shared libraries or DLLs stems from current library development model that heavily favors a one-size-fit-all model where a library is often designed to facilitate different types of application. This methodology while enabling the production of complex applications often leads to negative impact on both performance and security. Debloating has been proven to improve security by removing gadgets and vulnerabilities and trimming the bloated portions.

FEAST'19, November 15, 2019, London, United Kingdom

ACM ISBN 978-1-4503-6834-6/19/11…$15.00
https://doi.org/10.1145/3338502.3359765

Although there are some attempts in recent research to provide an insight into this problem in the wild, there is insufficient work on a set of universal, indicative, and systematic metrics to quantify it. Furthermore, while debloating open-source software is often straight-forward, it is challenging to debloat COTS or legacy counterpart mostly due to known complications in analysis techniques for closed-source binaries.

In this paper, we first introduce a set of comprehensive and generic metrics called bloat factors to systematically measure code bloating in binaries and use them as pointers to debloat binaries. Our metrics are designed to show bloat in programs as well as in libraries. From these metrics, we can infer different bloat-specific aspects of each component. Specifically, we propose three metrics called bloat factors. Program bloat factor (PBF) shows the amount of bloat in a program *including* all the shared libraries that it uses. Library bloat factor (LBF) shows the amount of bloat in a shared library for a specific program. Finally, system library bloat factor (SLBF) measures bloat in a shared library for a set of programs. To calculate bloat factors, we leverage a set of conservative binary analysis techniques that identify all inter-modular (program and libraries) dependencies. Using these dependencies, we are able to calculate the bloat portions of binaries that will not be executed at runtime across user input space. Secondly, we propose a conservative and static approach to debloat closed-source software using bloat factors as guidance by performing a simple and elegant binary rewriting technique.

Our contributions are as follow:

- We define a set of three descriptive metrics to measure bloat in programs and shared libraries. To the best of our knowledge, we are the first to propose and systematize such metrics.
- We conduct a comprehensive study on bloat in binaries across over 3000 programs and 896 shared libraries in Ubuntu 16.04. Using PBF, LBF, and SLBF to quantify bloat in those programs and shared libraries.
- We propose a technique to perform static debloating on a closed-source binary guided by bloat factors to generate its corresponding specialized version for a particular program called a fragment. A fragment contains only part of code that is required by the program.
- We implement our proposed technique in a framework and evaluate it on real-world programs and successfully debloat at most 25% of code in shared libraries with no performance cost.

2 MEASURING CODE BLOATING WITH BLOAT FACTORS

Code bloating occurs when there exists extraneous code in address space that is unused by a program. This is a pervasive problem that has been studied in the literature (e.g. [10, 11]). However, there is a lack of effort to establish a universal, accurate and descriptive method to systematically quantify this problem. We introduce a

set of metrics to measure code bloating for both programs and shared libraries called bloat factors. Bloat factor is a metric that gives us a lower bound to bloat and therefore to how much code we can eliminate. It is an ideal limit towards which debloating solutions must strive. To the best of our knowledge, we are the first to propose such metrics. Accordingly, we conducted an extensive study to measure code bloating for 3110 programs and 896 shared libraries in Ubuntu Desktop 16.04.

Bloat Factor Input Scope. All bloat factors are defined based on all possible benign inputs as intended by developers. Henceforth, the term "all possible input" in our paper refers to these inputs. They are calculated based on what the program itself declares as requirements from other code modules. This is intended to capture all requirements to process user inputs, including malformed user inputs. Malicious inputs are out of this work's scope .

Definition 1 (Program Bloat Factor (PBF)) Given a program P, let K be the set of all code paths over all possible inputs to P, and let I_i be the set of all the instructions executed from all the modules (libraries + executable) loaded in $P's$ memory in path K_i. Then:

$$PBF(P) = \frac{S_P - |\bigcup_{i \in K} I_i|}{S_P} \quad (1)$$

where the notation $|I|$ denotes the total number of bytes occupied by the instructions in set I. S_P is the total number of bytes of executable code in the program's memory. Since program bloat factor (PBF) covers both the main program and shared libraries, it is expected to be positive.

Definition 2 (Library Bloat Factor (LBF)) Given a program P and a library L loaded in P's memory, let K be the set of all code paths over all possible inputs to P, and let $I_i(L)$ be the set of all the instructions in L that are executed in path K_i. Then:

$$LBF(L, P) = \frac{S_L - |\bigcup_{i \in K} I_i(L)|}{S_L} \quad (2)$$

where the notation $|I|$ denotes the total number of bytes occupied by the instructions in set I and S_L is the total number of bytes of executable code in the library L.

Definition 3 (System Library Bloat Factor (SLBF)) Given a set of programs PS installed on a system or all programs included in the working set that use a shared library L, let K be the set of all possible code paths over all inputs to PS, and let $I_i(L)$ be the set of all the instructions in L that are executed in path K_i. Then:

$$SLBF(L, PS) = \frac{S_L - |\bigcup_{i \in K} I_i(L)|}{S_L} \quad (3)$$

where the notation $|I|$ denotes the total number of bytes occupied by the instructions in set I and S_L is the total number of bytes of executable code in the library L.

Using Bloat Factors as Directives for Debloating. Intuitively, LBF is the fraction of unused code in a library while SLBF of a library is its LBF for all programs in the set. Similarly, PBF is the fraction of unused code in a program and libraries that it uses. LBF provides insight into how a library is utilized by a particular program and serves as an indication of prolific debloating space when its LBF is high. Essentially, it directly reflects the absolute minimum amount of code that can be trimmed from the library without affecting its functionality for a program. Meanwhile, SLBF can be used to show how programs with similar functionality use a particular library. For example, Ubuntu packages are categorized into groups with similar functionality. Each group is called a section. By calculating SLBF for each section, we can visualize how a particular shared library is used by groups of similar applications. This provides valuable judgment for software developer to perform debloating at an early stage (i.e. development stage) to design shared libraries such that it can be efficiently utilized by several programs. For example, a generic library with high SLBF across different sections can be broken down into smaller versions that can facilitate a section or group of sections with low or negligible bloat factor. In other words, we are proposing a shift in library development philosophy from "more is better" to "usage-based targeted feature set".

3 CALCULATING BLOAT FACTORS FOR CLOSED-SOURCE BINARIES

3.1 Challenges

Calculating bloat factors requires knowledge of all inter-modular dependencies at both module-level (which module depends on what other modules) and function-level dependency (which function in a module is required by all other modules). We do not consider finer granularities because it requires runtime execution context although they may deliver better debloating results. While our solution is not restricted by execution platform, we choose Linux to explain the challenges:

1. **Precise inter-modular dependency.** A program can depend on multiple shared libraries which depend on more shared libraries. It is essential to identify correct module-level and function-level dependencies to create a precise slice of a shared library that includes all requirements for an entire process. This is a challenging problem for binary level solution as it requires accurate binary analysis.

2. **Indirect control flows.** Performing accurate control-flow analysis including indirect branches is essential to create correct library fragments. Moreover, challenges arise when function invocations between two modules happen through code pointers. A practical solution must include all code pointers across two modules.

3. **Dependency introduced at runtime with `fork` and `exec`.** Extra module-level and function-level dependencies can be introduced at runtime by either creating a new process using `fork` or dynamically loading of new shared libraries using `dlopen`, `dlclose`, and `dlsym`. A new process created using only `fork` will inherit resources (code and data) from its parent and therefore require the parent to account for the child's dependencies. Meanwhile, a child process created using `exec` does not cause problem as it comes with its own set of resources. Conversely, dynamically loaded libraries' dependencies must be identified and included in library fragments statically.

4. **Code-data dependency.** A shared library can have exported global variables. If imported by other modules, they can be accessed directly or indirectly through provided methods (e.g. mutator and accessor functions). Therefore, if a global variable is imported, its accompanied mutator and accessor methods must also be identified as dependencies. A practical solution must correctly identify the dependency between a global variable and these functions.

5. **Process initialization and termination.** During a module's runtime initialization and termination phase, some of its functions (e.g. `ifunc`, global variable initialization methods) are invoked by the loader to perform various tasks. These functions must be included as dependencies.

6. **Symbol Versioning.** An ELF binary, especially `glibc`, may contain multiple definitions of a symbol to support backward ABI compatibility. If an application is built against a shared library with symbol versioning, its import table will contain both function name and version string in each entry. At runtime, the GNU loader will perform strict symbol version matching based on versioning information embedded in the binary to import the correct implementation of a symbol. Failure to handle ELF's symbol versioning mechanism during static binary analysis leads to incorrect dependency identification and thus deficient specialization.

3.2 Generating Module's Dependence Graph (MDG)

The majority of runtime code bloating happens in shared libraries since executables are often heavily optimized by dead code elimination at compile time. To calculate bloat factors, we aim to identify the amount of code for each shared object that cannot be executed on all user inputs. First, for each module involved, we generate a module dependence graph (MDG) that captures both its control-flow and data dependencies. Next, we stitch all MDGs together to generate a system dependence graph (SDG). Both bloat factor calculation and library fragmentation will rely on the control-flow and data-flow represented in the SDG.

A module's dependence graph is a modified version of program dependence graph [3] that captures both the control-flow and data flow of a binary which are needed in our analysis for library fragmentation. To generate a module dependence graph, we first create a call graph from the binary. We then apply a combination of various binary analysis techniques including code pointer analysis, code-data dependence analysis and heuristics (process initialization and termination, dynamic dependence) to generate the complete MDG.

Code pointer analysis. To handle indirect calls, we perform code pointer analysis on a binary to recover both code pointer constants and code references. For code pointer constants, we use heuristics similar to the work by Zhang and Sekar [14] to identify them in a binary: code pointer constants are the ones that point to valid instructions in the code section. First, we scan different read-only sections in a binary to recover code pointers in places such as jump tables for `switch` statement, vtables, and pointer arrays. Functions in a shared library referenced by these addresses will always be included as dependencies. Next, we scan the entire code section for both code pointer constants that appear in address-taking instructions and code references. Particularly, for each function, we check each instruction for any code references that a call graph fails to capture. In more detail, instructions that take address of a function and instructions that compute code pointer. We include all valid addresses extracted within a function as its dependency in MDG.

Code-data dependence analysis. An imported object from a shared libraries may require additional functions to initialize and destroy it during process startup and termination. For example, if an exported C++ global object is imported by a program, its constructors are expected to be invoked for object creation and therefore should be included as dependencies. As a result, there is an additional level of dependency between those objects and code within a module. We use data flow analysis to identify those code-data dependencies. Particularly, for each exported variable, we traverse its definition-usage chain to identify all functions that modify it, then include them as dependencies of that object. If imported, those functions will not be counted towards bloat and will be included in the library fragment by our debloating framework.

Process initialization and termination. During process initialization and termination time, a number of functions in a binary will be invoked by the loader at all time and therefore must be counted included in the library fragment and thus not be considered as bloat regardless of whether or not they are explicitly imported. These functions come from three sources:

- **Shared object `.init` and `.fini` code.** At load time, for each mapped shared object, the loader performs relocation and then executes its initialization code found in the `.init` section. Similarly, when a process begins its termination procedure, the loader sequentially executes its termination functions found in `.fini` section. All code within these sections must be included as dependencies. If they contain code pointer addresses, we create a set of functions that they point to. Next, for every function in this set, we also include all other functions within the same module that it depends on as described in the MDG.
- **C++ global constructors and destructors.** These functions are invoked by the loader at process initialization and termination time to create and destroy static global variables. Pointers to these functions are typically found in the `.ctors` and `.dtors` sections. We scan each of those sections for valid code pointers addresses, identify other functions within the same module that they depend on, then include all of them in the library fragments.
- **GNU indirect functions.** Indirect functions called `ifunc` allow user to choose which implementation of a symbol to use at load time. When the loader performs symbol resolution, all `ifuncs` may be invoked and therefore must be retained in the library fragment.

Handling Runtime Dependencies. Dynamic loading of new code modules at runtime introduces additional dependencies that static analysis may fail to capture. There are two methods in which a process can load new module at runtime: creating a new process using `fork` and `exec` or dynamically loading new shared library using `dlopen`. Accurate handling of these cases is required to ensure correctness. In the case of creating new process, forking a new process of the same program does not introduce new dependency as the child process will inherit the entire process image from its parent. However, when `exec` is used in combination with `fork` to execute a new program, new dependencies from a shared library are introduced as the child process imports different functions from the same library. To handle executing new program at runtime, we create separate fragments of the same library for both the parent and the child process, then patch each program with the appropriate `rpath` values to allow them to use their own version of library fragment. At runtime, when loading new program, the loader will prioritize `rpath`'s value and load the correct fragment for the child.

Dynamically loaded shared libraries introduce new dependencies unknown during static analysis that may not be included in a library fragment. We first scan the binaries for strings that represent valid shared libraries, then we take a training approach to record all shared objects loaded at runtime using `dlopen`. We use a representative set of inputs that reflect average use case to record `dlopen` arguments. Any given program would be run in its intended form (i.e., use the app as it would normally be used). This approach takes advantage of how `dlopen` is used in the wild. Firstly, `dlopen` is rarely used with variable argument (i.e. library name is generated and thus not statically decodeable). Secondly, in our experience, `dlopen` is used to load environment-specific libraries once the environment is established. This process completes during the initialization stage of the program, and is therefore captured even with simple training inputs. After gathering the names of all dynamically loaded shared libraries, we include all their dependencies in the final library fragment.

3.3 Generating System Dependence Graph (SDG)

A system dependence graph (SDG) is the combination of all MDGs involved. Every two MDGs are connected using import-export relationship between two binaries. For example, if module A imports function `foo` from module B, then an edge is created to connect between node `foo` in A's import table and node `foo` in B's export table. To gather import-export information, a naive approach is to explore each import table in all code modules involved, including the executable and shared libraries to identify all imported functions from each module. However, this technique falls short due to a variety of challenges. Firstly, the load order of code modules which is decided at runtime by the loader determines which module an imported function can be found. Secondly, to choose the correct definition for a symbol at runtime, the dynamic loader must match both the symbol names and the version names. Choosing a function to import without considering version requirement may introduce false positive and result in packing the wrong definition to the shared library fragment. To generate accurate explicit imports, it is essential to replicate how the loader performs symbol binding at runtime.

3.4 Calculating Bloat Factors

At a high level, to calculate bloat factors we perform graph traversal on SDG with a set of nodes as starting points. All reachable nodes are counted as part of the dependency while unreachable ones are counted towards bloat. Each type of bloat factor has a different set of starting points. To calculate PBF and LBF, we use entries in the main program's import table as starting points since it fully reflects what a program requires from all modules. To calculate SLBF for a library, we first identify all programs in our test set that use it. Then, we collect all entries in their import tables and use them as starting points.

3.5 Scope and Limitations

Dynamic Loading. To handle dynamically loaded shared libraries, we take a training approach to record all shared objects loaded at runtime similar to the one used in PieceWise [11]. Upon examining 3174 programs and 4292 shared libraries, dynamic loading of code modules is rarely used. Only 3.1% of programs and 2.2% of shared libraries use this feature. While the set of recorded dynamically loaded shared libraries depends on the completeness of the input set, our experiments in this paper using a representative set of inputs (including inputs shown in Table 4) show that the majority of them can be captured. This is a known limitation and a complete solution for cases that are not covered within our experiments is being pursued.

Binary Analysis. Binary analysis, including control-flow graph generation is a challenging and ongoing research problem. Obstacles such as binary complexity, lack of high-level semantics, and optimizations lead to incomplete control-flow graph. Yet, pointer analysis gives an over-approximation version of control-flow graph hindering debloating quality in closed-source binaries. In this work, we rely on state-of-the-art binary analysis techniques.

Scope. Our system specifically tackles situations where only binaries are available. We assume the absence of source code and other debug information. While this work considers ELF binaries, our technique can be extended to other environments provided the appropriate binary analysis infrastructure.

4 BINARY SPECIALIZATION

4.1 Linux Binary Specialization

Ideally, the code available for a program during runtime is *no more than* what is required by the program across all possible inputs to the program. The technique for binary debloating we present in this paper is best applied to shared libraries.

One solution to debloat a shared library is to partition it into multiple smaller functionality-specific shared libraries (e.g., *printf()* and *sprintf()* would be in a single formated printing library), and load only those shared libraries that a program requires. However, identifying these boundaries is not straightforward. Furthermore, introducing new libraries is not backwards compatible; programs linked to libc will now have to be linked to the smaller libraries that make up `libc`. Such a solution lacks compatibility and fails to support programs that already depend on libc, and designing functionality-based shared libraries could be challenging. Code is often shared between functionalities, which could result in a vast number of indirect calls through PLT lookups during runtime.

Debloating all shared libraries is based on the fundamental principle that code that is not used should not be retained in the memory. However, debloating executables and shared libraries requires different techniques. For executables, firstly, the assumption is that programs usually shipped heavily optimized by compilers with techniques such as static dead code elimination. In other words, all the code in the executable is expected to be exercised by some or the other input to the program. Therefore, effectively debloating executables requires context-sensitive runtime solution and is better achieved dynamically. Debloating shared libraries, on the other hand, can be determined regardless of user inputs, albeit a challenging problem. An effective solution must achieve partial debloating wherein certain modules can be debloated while others (e.g., legacy binaries) can be retained without any changes.

At a high level, to debloat a shared library for a program, we create a specialized version of that shared library called a library

fragment for the given program. Ideally, a library fragment will contain only the functions that are required by the main program as well as all other loaded shared libraries in the same process. Specifically, we develop a framework that takes a program and its shared library, performs binary analysis on all code modules involved, identifies all necessary dependencies, then creates fragments for the shared libraries of choice. While our prototype is currently restricted to Linux binary specialization, this technique at a high level is applicable to all platforms where binary analysis (including pointer analysis) is sufficiently accurate.

Our approach maintains the following design goals:

• **Deployability.** Our framework works seamlessly with COTS binaries without access to neither program nor shared library source code, making it an ideal system to debloat legacy software.

• **Correctness.** A library fragment created for a particular program shall be self-sufficient and provide all the necessary code to ensure the correct execution of the said program. To achieve this, binary and dependency analysis technique should not include false negatives while false positives are inevitable due to known challenges in reverse engineering COTS binaries.

Creating a shared library fragment for a program requires the precise identification of all dependencies that the program has for that shared library. We take advantage of the LBF and use it as a directive to create a specialized library fragment.

4.2 Generating Library Fragments

To generate a library fragment for a program, the high level approach contains two steps: identifying the program's dependencies and package the required code into a library fragment. Using the SDG, identifying program's dependency is simply a graph traversal problem in which we find all reachable nodes starting from the main program's import table. Once a set of reachable nodes is generated, we sort all nodes into their respective shared library. As a result, we now have a list of dependencies for each shared library.

Given a list of functionalities to include, there are a few methods for creating a library fragment, each with its own advantages and disadvantages. One possible approach is to lift the binary to an intermediate representative (IR) level, remove unused code then recompile it. This approach introduces complex challenges in decompilation process and in designing an IR that is suitable to represent the binary for the purpose of debloating. Furthermore, recompiling the library may negate one important benefit from debloating which is improving security by removing vulnerabilities and potential gadgets for code reuse attacks as proven by Brown and Pande [2]. Alternatively, we can pick needed functions and repackage them as-is to a new binary. While this approach may reduce the size of the final fragment, it requires delicate and accurate handling of binary rewriting such as reference patching since code and data references are relative as all Linux shared libraries are position-independent. This is a non-trivial problem. In our approach, we first create an identical copy of the shared library then replace unused code with invalid bytes. A good candidate for this invalid byte must satisfy two requirements to guarantee security benefit of debloating. Firstly, it should not introduce new gadgets for code reuse attack or new vulnerabilities for exploits in general. Secondly, it shall terminate illegal execution of code that was not indented to be used by the program. In other words, it shall terminate the process upon execution realizing that an exploit (such as an attempt to hijack control-flow) is happening. On a Linux system, byte `\xd6` would satisfy all of the above requirements and therefore is used in our framework.

Table 1: Program and Library Bloat Factors for Large Programs

Program	PBF	Average LBF
chrome	12.19%	64.37%
chromium	10.91%	64.74%
firefox	37.23%	42.79%
thunderbird	36.04%	41.09%
make	61.94%	61.29%
g++-5	26.32%	35.99%
evince	68.57%	68.19%
vlc	68.89%	73.47%

Table 2: Program Bloat Factors (PBF) and Library Bloat Factors (LBF) for SPECrate 2017 Integer and Floating Point Suites

Benchmark	PBF	Average LBF	libc LBF
perlbench	29.72%	47.32%	57.97%
gcc	11.70%	51.07%	52.71%
mcf	60.16%	33.48%	66.31%
omnetpp	26.01%	36.37%	44.98%
xalancbmk	15.40%	37.06%	47.63%
x264	50.28%	52.40%	60.12%
deepsjeng	53.61%	31.22%	61.79%
leela	47.63%	45.44%	54.79%
xz	51.73%	31.33%	62.01%
namd	38.94%	42.71%	54.72%
parest	12.56%	35.59%	54.72%
povray	32.09%	37.19%	52.33%
lbm	67.70%	54.19%	64.09%
imagick	25.25%	37.80%	53.18%
nab	54.06%	47.03%	57.51%

5 EVALUATION

In this section, we first aim to compute bloat factors for a large test set of over 3000 programs and 800 shared libraries in Linux to demonstrate their capabilities in illustrating bloat and its characteristics in Linux binaries. Then, we evaluate our static debloating method guided by these metrics for its effectiveness, performance, and correctness.

5.1 Bloat Factor Evaluation

Bloat factors for individual Programs (PBF and LBF). To calculate bloat factors, it is important to capture all dependencies between code modules (program and shared libraries and between shared libraries) which are inherently symbol binding information available at load time. To acquire this, we begin by identifying all code modules (shared library or new programs) to be loaded for

Table 3: Libc Code Size Reduction for Large Programs

Program	% Instruction Reduction	% Function Reduction	% Code Reduction (bytes)
firefox	12.66%	28.77%	12%
g++-5	20.81%	43.48%	19.69%
vlc	13.77%	31.73%	13.88%

Table 4: Workload for correctness experiment on real-world programs

Program	Workload
firefox	Run browser benchmark at https://web.basemark.com/ and open Alexa top 500 websites [1].
vlc	VLC fundamental tests in src/test and all VLC benchmarks found at http://streams.videolan.org/benchmark/
g++-5	Compile SPEC CPU 2017 C++ benchmarks: 508.namd_r, 510.parest_r, 511.povray_r, 620.omnetpp_s, 623.xalancbmk_s, 641.leela_s, 520.omnetpp_r, 523.xalancbmk_r, 541.leela_r

a program then recored precise symbol resolution using the GNU loader (`ld.so`). Our test set includes large programs such as popular web browsers, `thunderbird, make, g++, evince`, and `vlc` as well as all benchmarks in SPEC CPU 2017. For each program, we calculate both PBF, LBF for each shared library that they use, and LBF for `libc` since it is the most popular shared library in Linux.

The results from our study are presented in Tables 1, 2, and Table 5 in Appendix A. Overall, our results outline a significant debloat space for all programs in our test set in which we can remove at most 68% of program code and at most 66% of shared library code among the programs in our set.

Bloat Factors for System of Programs (SLBF). To illustrate the correlation and impact of SLBF in inferring bloat-specific characteristics of libraries, we conduct an extensive study across 3110 programs, 896 shared libraries, and 50 sections (each section contains packages with similar functionality) installed on Ubuntu 16.04 LTS. For each pair (S, L) of section S and library L, we calculate SLBF that shows how much of code in L is unused by all programs in S.

Overall, we generated the SLBF for 2739 section-library pairs. On average, each section uses 55 shared libraries. There are 2523 instances where a library has a negligible SLBF of less than 10%. Amongst them, we found 405 instances where a library is fully utilized by a section with a SLBF of value zero. However, our study finds that there are 136 cases where a shared library contains at least 30% unused code by a section. Table 6 in Appendix B shows a list of libraries with the highest SLBF. Each SLBF number represents how much code in each library is currently unused in all programs of a section.

5.2 Binary Specialization Evaluation

Code Size Reduction. We evaluated our framework on large programs (`firefox, g++, vlc`) and SPEC CPU benchmarks and generate `libc` fragments for each of them. We then calculated how much of code (in bytes) that we were able to eliminate from the original `libc` for each fragment. The result is presented in Table 3.

Correctness. To evaluate the correctness of the `libc` fragments for SPEC CPU 2017 benchmarks, we run each of them with the new `libc` fragment on the reference workload and did not observe any failures according to `runcpu`. To run each SPEC CPU 2017 benchmark using the new library fragment, we modify each program's `rpath` variable to redirect the loader to use the new fragment at runtime. We use the program `patchelf` for this step.

To evaluate the correctness of library fragments for a set of real-world programs, we ran each of them against a large workload and did not observe any failures. Details for this experiment can be found in Table 4.

Performance Overhead. We measure the performance overhead for all SPEC CPU 2017 benchmarks running the new specialized versions `libc`. The results are presented in Table 7 in Appendix C. Overall, the performance overhead captured in our experiments are within negligible error margins specified by SPEC, with some benchmarks reporting negative performance cost.

6 RELATED WORKS

Code bloating has been shown to be a prominent and prevalent problem [10]. Several approaches have been proposed that aim to eliminate code bloat for both open source and closed-source applications. Use-based configurations have been widely used in recent research as a directive to debloat a program. Koo et al. [5] and Sharif et al. [12] take configurations provided by users to produce a trimmed down version of a library or program. Chisel [4] applies machine learning to debloat in which it takes a program's desired functionality as an input and leverages delta debugging and model-base reinforcement learning to produce a debloated version of the program. Similarly, software winnowing [8] generates specialized versions of applications given configuration or deployment context using aggressive compiler-based optimization. Another research direction proposes debloating approaches that preserve a program's original functionalities. Shredder [9] debloats closed-source binaries by limiting execution of security-critical API calls to their specialized versions. Piecewise [11] is another approach to debloat a library by using embedded control-flow information in a binary to direct the loader to remove unused code at load time.

Debloating approaches have also been introduced for the Linux kernel. kRazor [6] records how a process uses the kernel then introduces a kernel module to enforce the recored function calls. Kurmus et al. [7] propose a configuration-based approach that first records a typical workload of the kernel for the entire system and then modifies compilation configuration to generate a kernel specialized for that system workload. Similarly, Tartler et al. [13] automatically generate a kernel configuration tailored for a particular workload.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their feedback and our shepherd Maverick Woo for helping us improve the paper. This research was supported in part by ONR Award #N00014-17-1-2929, NSF Award #1566532, and DARPA Award #8119. Any opinions, findings and conclusions in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

[1] 2019. Top Sites in United States. https://www.alexa.com/topsites/countries/US. (2019). Accessed: 2019-08-24.

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

[3] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9, 3 (1987), 319–349.

[4] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective Program Debloating via Reinforcement Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. ACM, New York, NY, USA, 380–394. https://doi.org/10.1145/3243734.3243838

[5] Hyungjoon Koo, Seyedhamed Ghavamnia, and Michalis Polychronakis. 2019. Configuration-Driven Software Debloating. In *Proceedings of the 12th European Workshop on Systems Security (EuroSec '19)*. ACM, New York, NY, USA, Article 9, 6 pages. https://doi.org/10.1145/3301417.3312501

[6] Anil Kurmus, Sergej Dechand, and Rüdiger Kapitza. 2014. Quantifiable run-time kernel attack surface reduction. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 212–234.

[7] Anil Kurmus, Reinhard Tartler, Daniela Dorneanu, Bernhard Heinloth, Valentin Rothberg, Andreas Ruprecht, Wolfgang Schröder-Preikschat, Daniel Lohmann, and Rüdiger Kapitza. 2013. Attack Surface Metrics and Automated Compile-Time OS Kernel Tailoring. http://www.internetsociety.org/sites/default/files/03_2_0.pdf

[8] Gregory Malecha, Ashish Gehani, and Natarajan Shankar. 2015. Automated Software Winnowing. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing (SAC '15)*. ACM, New York, NY, USA, 1504–1511. https://doi.org/10.1145/2695664.2695751

[9] Shachee Mishra and Michalis Polychronakis. 2018. Shredder: Breaking Exploits through API Specialization. In *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM, 1–16.

[10] Anh Quach, Rukayat Erinfolami, David Demicco, and Aravind Prakash. 2017. A Multi-OS Cross-Layer Study of Bloating in User Programs, Kernel and Managed Execution Environments. In *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation (FEAST '17)*. ACM, New York, NY, USA, 65–70. https://doi.org/10.1145/3141235.3141242

[11] Anh Quach, Aravind Prakash, and Lok Kwong Yan. 2018. Debloating Software through Piece-Wise Compilation and Loading. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD. https://www.usenix.org/conference/usenixsecurity18/presentation/quach

[12] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. 2018. TRIMMER: Application Specialization for Code Debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*. ACM, New York, NY, USA, 329–339. https://doi.org/10.1145/3238147.3238160

[13] Reinhard Tartler, Anil Kurmus, Bernhard Heinloth, Valentin Rothberg, Andreas Ruprecht, Daniela Dorneanu, Rüdiger Kapitza, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2012. Automatic OS Kernel TCB Reduction by Leveraging Compile-Time Configurability. In *Presented as part of the Eighth Workshop on Hot Topics in System Dependability*. USENIX, Hollywood, CA. https://www.usenix.org/conference/hotdep12/workshop-program/presentation/Tartler

[14] Mingwei Zhang and R. Sekar. 2013. Control Flow Integrity for COTS Binaries. In *Proceedings of the 22nd USENIX Security Symposium (Usenix Security'13)*. 337–352.

A APPENDIX

Table 5: Program Bloat Factors (PBF) and Library Bloat Factors (LBF) for SPECspeed 2017 Integer and Floating Point Suites

Benchmark	Program BF	Average LBF	libc LBF
perlbench	29.81%	58.99%	55.64%
gcc	11.78%	61.92%	50.02%
mcf	57.50%	54.08%	61.75%
omnetpp	25.66%	40.60%	42.41%
xalancbmk	15.27%	41.16%	44.96%
x264	49.86%	63.58%	57.43%
deepsjeng	53.61%	31.22%	61.79%
leela	47.63%	45.44%	54.79%
xz	36.85%	45.80%	40.87%
lbm	52.70%	54.56%	40.85%
imagick	21.26%	47.30%	37.19%
nab	42.86%	51.14%	37.29%

B APPENDIX

Table 6: Top 20 Most Bloated Libraries per Section

Section	Library	Size	SLBF
universe/libs	libaudio.so.2	98.61 KB	100.00%
universe/tex	libaudio.so.2	98.61 KB	100.00%
utils	libxapian.so.22	1.98 MB	100.00%
admin	libxapian.so.22	1.98 MB	100.00%
admin	libappstream.so.3	313.09 KB	99.91%
universe/sound	libglibmm-2.4.so.1	485.05 KB	99.84%
gnome	libglibmm-2.4.so.1	485.05 KB	99.84%
universe/libs	libKF5WindowSystem.so.5	289 KB	99.70%
contrib/misc	libcurl.so.4	442.38 KB	99.46%
universe/libs	libQtXml.so.4	279.07 KB	99.18%
misc	libseccomp.so.2	282.27 KB	99.12%
universe/kde	libdbusmenu-qt.so.2	207.98 KB	98.98%
editors	liblangtag.so.1	130.43 KB	97.21%
gnome	libtracker-sparql-1.0.so.0	119.3 KB	97.07%
universe/admin	libseccomp.so.2	282.27 KB	96.24%
universe/kde	libQtXml.so.4	279.07 KB	95.91%
libs	libsamba-util.so.0	414.4 KB	95.75%
utils	libnl-genl-3.so.200	23.23 KB	95.46%
gnome	libcolumbus.so.1	270.77 KB	95.09%

C APPENDIX

Table 7: SPECrate and SPECspeed Performance Overhead running libc fragment

Benchmark	Overhead	Benchmark	Overhead
500.perlbench	-0.77%	600.perlbench	-2.81%
502.gcc	-1.32%	602.gcc	-2.02%
505.mcf	0.41%	605.mcf	-1.79%
520.omnetpp	1.29%	620.omnetpp	-0.79%
523.xalancbmk	0.26%	623.xalancbmk	-2.56%
525.x264	-0.18%	625.x264	-1.81%
531.deepsjeng	0.11%	631.deepsjeng	-0.64%
541.leela	-0.68%	641.leela	-1.43%
557.xz	0.57%	657.xz	-1.73%
508.namd	-0.98%	619.lbm	-1.00%
510.parest	-1.97%	638.imagick	-1.02%
511.povray	-0.90%	644.nab	-1.22%
519.lbm	-1.38%		
526.blender	-1.22%		
538.imagick	-1.09%		
544.nab	-0.91%		