Negativa: A Practical Debloater for Shared Libraries

Abstract

Software bloat describes code, features, and functions of a software that are not required by the application during runtime. Bloat does not only waste resources but also increases the attack surface of applications increasing their vulnerabilities. Despite significant research in the area of software debloating, existing debloating tools often fail to handle complex applications and in some cases decrease security by introducing new gadgets. In addition, many debloating tools focus on debloating software binaries, but not the shared libraries these binaries use. Shared libraries are however a large contributor to bloat of the overall system, and have been recently shown to be a major source of vulnerabilities. To fill this gap, we propose Negativa, a novel runtime shared library debloater that does not require access to source code or recompilation, a common weakness in the few tools that consider shared library debloating. Negativa utilizes a novel dynamic tracing technique to detect used functions within shared libraries. Using a novel algorithm, Negativa safely removes unused functions while maintaining memory addresses intact. We evaluate Negativa with 20 applications from the *Debloater-Eval Benchmark*, a new benchmark released for evaluating debloating tools. Compared to ten debloating tools tested with the benchmark, Negativa is the only tool able to safely debloat all applications in the benchmark with a 100% passing score. We also test Negativa on the highly complex PyTorch framework, comprising 91 shared libraries. Our results show that Negativa removes up to 91% of unused functions, reducing file sizes by up to 59% and for some applications up to 20% reduction in memory usage, while simultaneously improving security by removing gadgets and vulnerabilities.

1 Introduction

"It's simple. I just remove everything that is not David."

— Michelangelo, *when asked how he carved the statue of David.*

Today's software is bloated. Software bloat is a byproduct of developers reusing code, *libraries*, and containerized applications while continuously adding features and modules to support multiple functionalities. This reuse introduces code that is not relevant to the developed application, with many of the added features and modules only relevant to a (small) subset of users at runtime. In addition, shared libraries, which most applications rely on, are developed for usage across a wide number of use-cases, resulting in much of the code of these libraries not being used at all. Software bloat increases resource usage and the attack surface of applications with no or limited benefits to the users [29].

Bloat has been studied across different layers of software with many suggested debloating tools to remove bloat from these layers. For example, a recent study has shown that up to 90% of popular Machine Learning (ML) container images is bloat, with this bloat contributing to up to 90% of all the Common Vulnerabilities and Exposures (CVE) [58] in these containers. Similarly, for operating systems, it has been shown that strict debloating exposes 181× less kernel code preventing up to 90% of a set of popular attacks [12]. Similar results have been obtained for executable binaries [14, 45], shared libraries [13, 46], web-applications [18, 19, 30], Android applications [36, 37], and even firmware [26].

Bloat not only increases the memory footprint of an application significantly, it also increases the attack surface of software considerably [58]. The problem of bloat is exacerbated by the prevalent use of memory-unsafe programming languages such as C and C++. Using memory-unsafe languages was highlighted as a major security risk in a recent report from the US Cybersecurity and Infrastructure Security Agency, the US Federal Bureau of Investigation, the Australian Signals Directorate's Cyber Security Centre, and the Canadian Centre for Cyber Security, reporting that 52% of 172 critical open source projects contain code written in memory-unsafe languages [27]. In these projects, 55% of all the code is written in a memory-unsafe language. Amongst the 45% of projects written in a memory-safe language, a majority depends on software packages, mostly in the form of shared libraries, that are written in a memory-unsafe language, resulting in almost all the studied critical open source projects having memory safety issues.

The prevalence of memory-unsafe languages in developing shared libraries poses security risks. Many of today's software is packaged as shared libraries to enable developers to share commonly used functions across multiple applications. From shared libraries in Android applications [17], to large ML frameworks [43], shared libraries have become ubiquitous. A large framework such as the CPU version of PyTorch has a total of 91 shared libraries. A simpler software such as Nginx depends on up to seven shared libraries. Many of these libraries are bloated. For example, only 5% of `libc` was reportedly used on average across the Ubuntu Desktop environment (based on programs from 2016) [46]. Given how bloat can contain many vulnerabilities that utilize memory unsafety, it is of paramount importance to remove bloat in software in general, and in shared libraries in particular as they are shared across a large number of applications.

While there has been extensive research on debloating shared libraries and other parts of the software [13, 20, 42, 44, 55, 57], Brown et al. [21] have recently shown that debloating

is far from a solved problem. We identify the following three main shortcomings with almost all of the tools available.

- Debloating has been shown to introduce new quality gadgets in the debloated software [23] making the software less secure. In many other instances, debloating fails to improve the security of software.
- Almost all available debloating tools we are aware of were tested on simple programs. For example, the total size of Nginx shared libraries, which are used to evaluate many debloating tools, is around 3MB. Many of today's shared libraries are orders of magnitude larger, e.g., the total size of the CPU version of PyTorch shared libraries is around 564 MB. Most state-of-the-art debloating tools fail to produce a working debloated program as has been shown by Brown et al. [21].
- Many debloating tools suffer from usability issues, as they either require: the program's source code; significant time to produce a debloated version; or have other technical issues [21].

To solve the above shortcomings, in this paper we present *Negativa*, a novel debloating tool for shared libraries. We focus on shared library debloating as shared libraries constitute a large fraction of many of today's most popular software. *Negativa* removes bloat based on the end-use without requiring access to the source code. To do so, *Negativa* first detects the used libraries and used functions within software using dynamic analysis, locating the file ranges of each function in the shared libraries. The tool then removes unused functions, compacting the shared libraries. During compaction, the tool needs to reconstruct the library such that any memory offsets used in the compacted library are correct when the operating system loads the program in memory. To do so, *Negativa* utilizes a novel approach that maintains the memory offsets intact while reducing the file size and memory usage of shared libraries. Our contributions can be summarised as follows:

- We propose a novel lightweight dynamic analysis method to detect all used libraries and all used functions in software.
- We develop a novel approach to compacting shared libraries that preserves the correct memory mapping while reducing the file size and memory usage.
- We develop a prototype of *Negativa* which we test on 21 popular software packages with various complexities evaluating the tool comprehensively. Our evaluations show that *Negativa* successfully debloats 21 applications, removing up to 91% of the unused functions, resulting in up to 59% reduction in file size, and for some applications up to 20% reduction in memory usage.
- We evaluate *Negativa* using seven different security metrics measuring security improvements. We conduct a real-world case study to demonstrate how *Negativa* removes a critical vulnerability and improves software security.

2 Background and Related Work

There has been a huge body of research on debloating, with more than 70 papers published in the past 10 years, and many companies aiming to provide debloating tools and services. In this section, we provide an overview of this research, discussing some limitations of the previous work.

2.1 Bloat and Debloating

There is no formal definition of what constitutes "bloat". Most of the debloating literature assumes that there is a workload input set that is used with a program to find out which parts of the software are being used, with everything else considered bloat. This means that most debloating tools will remove parts of the program that are not activated by the selected input set. Software bloat can be classified into two categories [21]. In the first category, the discovered bloat is universally unnecessary code and can be removed without impacting the program's behavior for all intended end uses (Type I bloat). The second category of bloat is end-use dependent; code may or may not be bloat depending on how its user(s) uses the program (Type II bloat).

Recently, there have been increased efforts to systemize the knowledge around bloat and debloating tools. Brown et al. [21] survey the field of program debloating, evaluating 10 debloating tools using 20 benchmark programs, across 12 performance, security, and correctness metrics. They show that the debloating tools success rate is only 42.5% across all tools and all benchmarks. For medium- and high-complexity programs, the success rate is only 21%, with only 3.3% success rate in removing Type II bloat. Furthermore, none of the debloating tools Brown et al. evaluate debloated all 20 benchmark programs successfully. Their results indicate that the available debloating tools are not mature enough to provide sound and working debloated programs.

2.2 Static and Dynamic Analysis in Debloating

Debloating tools can be categorized by the types of bloat they remove: Type I and Type II debloaters. Type I debloaters often do not require running target software with workloads, relying on static or compiler-assisted analysis [13, 30, 31, 47, 48, 56] to identify and remove universally unnecessary bloat. Some Type I debloaters may also incorporate dynamic analysis by running the software to improve static analysis results [24, 53, 59]. Type II debloaters require executing certain workloads or specifying features of the software to

detect and remove unnecessary code for the specific end-use [14, 15, 22, 28, 32, 40, 41, 45, 52]. Our work, Negativa, is a Type II debloater.

Static Analysis. Static analysis involves analyzing the software without executing it. This method typically constructs a Function Call Graph (FCG), where any functions not reachable within the graph are considered to be bloat. FCGs can be built by analyzing the source code directly [30, 31, 56], using compilers to create dependency graphs that show which functions depend on each others [47], or analyzing the binaries by disassembling them and leveraging symbol and relocation information [13]. Additionally, static analysis can also construct the Control Flow Graph (CFG). A Control Flow Graph (CFG) provides finer granularity than an FCG, as it tracks basic blocks (sequences of code) within functions rather than focusing solely on function call [48]. Any code that is unreachable in the CFG is similarly identified as bloat. However, the generation of the ideal CFG is proven to be an undecidable problem [34].

Dynamic Analysis. Dynamic analysis based methods require executing the application with prespecified workloads to determine which code is used at run time. It is often used alongside static analysis to improve its results or act as a starting point for constructing the FCG. One type of dynamic analysis approach involves recompiling the target application and inserting logs at function calls [20]. These logs are then used to construct the FCG. However, this approach requires access to the source code, limiting its applicability. Another approach to dynamic analysis uses binary tracing, which records the binary's execution and its interactions with the operating system without needing source code or recompilation [35, 45, 59]. Tools like DynamoRIO [6] and PIN [39] are two popular binary tracing frameworks. Based on these frameworks, some library function call tracing tools, such as `drltrace` [3] (based on DynamoRIO) and `TinyTracer` [10] (based on PIN), are developed to detect used library functions. However, all tools we studied either fail to correctly detect all used library functions or have high overheads to the target application, as we show in § 4.2.

Relying solely on dynamic analysis can lead to wrong detections of used code, which is why traditional debloating methods often combine dynamic and static analysis. This combination aims to cover more functions, particularly those that dynamic analysis might overlook. However, the combined approach can result in overestimation, where unused functions are mistakenly marked as necessary. As noted by Ali et al. [16], although static analysis debloaters produce more correct applications, dynamic analysis debloaters achieve better bloat reduction. We argue that dynamic analysis, if done correctly, is sufficient to accurately and efficiently detect all used functions in a shared library, achieving both better correctness and bloat reduction.

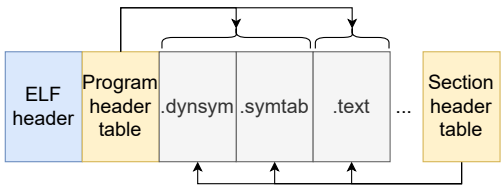


Figure 1. ELF format.

2.3 Shared Library Debloating

Many debloating tools have been proposed in the literature, mostly focusing on debloating executable binaries [14, 22, 32, 40, 41, 45, 52]. Binary debloaters have a straightforward scope since each application typically involves a single binary. These tools however ignore the shared libraries used by these binaries. From our evaluation of several popular software packages, we found that shared libraries are much larger than the binaries that use them. For example, the shared libraries used by the Nginx binary are 3× larger than the binary itself. Most of the tools that support shared library debloating require access to the source code to recompile the library [20, 47], or cannot remove Type II bloat [13, 44]. In addition, tools that support the debloating of type II bloat run target applications with prep-selected workloads offline, using, e.g., test cases, and then remove any parts of the program not activated by the pre-selected workloads. However, this offline approach has an intrinsic limitation: the workloads may not represent the actual end-use of the application. When deployed in a production environment, the actual usage of the application may differ from the workloads, leading to crashes or other undesired behavior.

2.4 Shared Library structure: The ELF format

To build practical shared library debloating tools, we realised that the previously described shortcomings can only be solved by rethinking how shared libraries are built, loaded, and run in memory during runtime. To understand how shared libraries are loaded in memory, we study the Executable and Linkable Format (ELF) file structure, the standard file format for shared libraries in Linux [2].

Figure 1 shows the structure of an ELF file. ELF headers contain metadata about the ELF file, such as the ELF file type and the architecture. The ELF file consists of multiple *sections*. Each section is well-defined, for instance, the `.dynsym` section holds symbol information, while the `.text` section holds the program code.

The section header table and the program header table provide two different views of the file by organizing the sections in different ways. The section header table specifies each entry for each section, including the metadata of that section, such as the section name, size, and memory address. Hence, the section header table provides a view of the file's structure. The program table header defines how

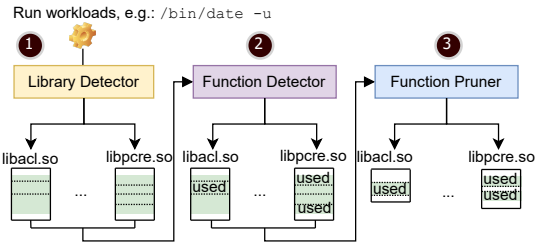


Figure 2. Negativa overview.

the ELF file is loaded into memory. One entry in the program header table may point to multiple sections, specifying which sections to load and which memory addresses to load them at. This section hence provides a view of the memory layout when the ELF file is loaded into memory. Before a shared library is loaded, the system loader decides where and how to load the shared library into memory based on the program header table. Another table utilized by Negativa (the `.dynsym` and `.symtab` section) is the symbol table. The symbol table includes information about functions in the library, e.g., their names, addresses, and sizes. The symbol table helps the dynamic linker identify and resolve symbol addresses, ensuring that the correct functions are accessed when shared libraries are loaded.

3 Negativa Design

Figure 2 shows an overview of Negativa with its three main components: Library Detector, Function Detector, and Function Pruner. When a program is debloated with Negativa, Negativa calls Library Detector to detect all libraries used by the workload. Then Function Detector detects used functions in each shared library, marking it as "used" as shown in the figure. Finally, Function Pruner removes all unused functions and compacts the shared library, resulting in a debloated shared library with a reduced size. We next describe the details of each of the three components.

3.1 Library Detector

Library Detector detects all the shared libraries used by a program's workload at runtime. To better understand our design, we first describe how existing approaches detect shared libraries in programs. Existing debloaters rely on static analysis to detect used libraries [13, 59]. They analyze the executable binaries and read the ELF Dynamic section to find the shared libraries that the executable binaries depend on. However, executable binaries can also load shared libraries dynamically at runtime (loaded by `dlopen`). Existing debloaters do not debloat such shared libraries due to their inability to detect them [13].

To address this issue, Library Detector detects used shared libraries at runtime, utilizing a hook function provided by system loaders, `_dl_debug_state`. The function is called by the system loader when a new shared library is loaded into memory. Before running the workload, we insert a trap instruction at the beginning of the hook function. Each time a new shared library is loaded, the workload process will be paused, and the process memory map, which contains the shared libraries used, will be read by Library Detector to identify all used shared libraries, including dynamically loaded ones.

After detecting all shared libraries used by the workload, we need to extract the file ranges (the start and end file offsets) of each function in each shared library. The file ranges are used by Function Detector and Function Pruner. We utilize the symbol table of each shared library to get the file ranges of each function. The symbol table includes the name, file offset[1], and the size of each function. Theoretically, given a function name, we can get the file range of that function by checking its file offset and size in the symbol table. However, we observed that the size of a function in the symbol table is not always set. This is because the symbol size is an optional field in the ELF format, i.e., the compiler may not always fill it. Relying on the file offset and size in the symbol table can cause the removal of used code, resulting in a broken shared library.

To address this issue, we first sort the functions by their offsets in ascending order. Then the file range of a function starts from the offset of the function and ends at the offset of the next function. Formally, for each function f, there is a start offset $f.s$. Given a list of *all* functions in a shared library, $\{f_1, f_2, \ldots, f_{n-1}, f_n\}$, sorted by their offsets in ascending order, the corresponding file ranges of all functions are $\{[f_1.s, f_2.s), [f_2.s, f_3.s), \ldots, [f_n.s, e)\}$, where e is the end offset of the program code section. The approach may overestimate function sizes, but it avoids the risk of removing used code due to the incorrect function size in the symbol table.

3.2 Function Detector

The most critical part for debloaters is the correct identification of all functions (or code) in a shared library. Existing approaches to detect used functions are usually based on static analysis, dynamic analysis or a combination of both. However, these approaches face some limitations. First, they cause significant performance overhead. Second, they either miss some used functions (false negative) or report unused functions as used (false positive). To address these limitations, Negativa uses the Function Detector component, a novel approach to detect used functions in shared libraries, with negligible performance overheads and better accuracy. Additionally, it supports multi-processes applications and the detection of indirect function calls.

Based on the function file ranges obtained from Library Detector, Function Detector disassembles the shared library to get all instructions for each function. Then for each function, Function Detector finds the *branch-related* instructions,

[1]Symbol tables include the virtual address of a function, which can be easily converted to a file offset.

such as CALL, RET, JMP and JE. A mapping M is created from the memory address of the branch-related instructions to the first byte of the instructions and the function name, i.e.,

$$M = \{address : (first\_byte, function\_name)\}$$

We then run two processes in parallel, a tracer process, and a tracee process. The tracer process forks and attaches to the tracee process using Linux PTRACE. The tracee is the process that uses the shared libraries and runs the target application. Once the tracee is forked, and before it runs any instructions the tracer iterates over all memory addresses in M and modifies the first byte of the address to 0xCC, which is a trap instruction (INT3). When the tracee hits the trap instruction, it notifies the tracer of the current instruction memory address of the tracee. The tracer then looks up the memory address in M and gets the function name and reports the function as used. The tracer then recovers *all* the modified bytes of that function.

The main difference between Function Detector and traditional binary tracing approaches is that it traps *all branch-related instructions* in a function, which is also the reason why Function Detector can detect more used functions. To detect used functions, traditional binary tracing approaches utilize the rules of *Call Convention* [5]. One approach is to trace the CALL instruction and analyze the operand of the CALL instruction to get which function is called. Another approach inserts trap instructions at the beginning of a function and the RET instruction, assuming that the execution starts from the first instruction of a function and ends at the RET instruction [3]. However, from our observation of many real-world shared libraries, we found that *Call Convention* is not always followed. Some functions are not called by the CALL instruction, instead, they are called by the JMP instruction directly, which nullifies the first approach. On the other hand, the entrance and exit of a function are not always at the beginning and the RET instruction, which invalidates the other approach. Based on this observation, we assume that if a function is executed, then the function must exit at branch-related instructions, such as RET, JMP and CALL. Therefore, we trap all the branch-related instructions to detect the function usage. In doing so, the effectiveness of detecting used functions is improved. As for efficiency, a used function will only be trapped once and the function will be executed as normal after the trap instruction is hit. Thus, the performance overhead is also reduced.

To support multi-processes applications, we need to handle the fork and exec system calls. If a tracee process forks a child process, the child process inherits the memory map of the tracee process. A new tracer process is created for the child process. The mapping M is reused for the new tracer process. If a tracee process execs a new binary, the memory map is changed. Therefore, a new mapping M will be created for the new memory map.

Another observation we made is that existing binary tracing tools fail to detect indirect functions in shared libraries [3, 8, 10]. The GNU indirect function support (IFUNC) is a feature allowing developers to create multiple implementations of a given function and select amongst them at runtime using a resolver function [4]. The resolver function and all the functions called by it are executed by the system loader during shared library loading into memory. Existing tracing tools start tracing after the shared library has been loaded, which is too late to detect these functions. To address this issue, Function Detector uses dlopen to load the shared library and uses dlsym to trigger the resolver function after the shared library has been loaded. In this way, Function Detector postpones the execution of the resolver function to the time after the shared library has been loaded. Therefore, these indirect functions can be detected by Function Detector just as normal functions.

The output of Function Detector is a mapping of used shared libraries to file ranges of their used functions. Listing 1 shows an example output of Function Detector.

Listing 1. Example output of Function Detector. One file range of `libacl.so` is used; Two file ranges of `libpcre.so` is used. Step 2 in Figure 2 also illustrates these ranges, which are marked as "used".

```
libacl.so: {[0x400, 0x461)}
libpcre.so: {[0x70, 0xBA), [0x1920, 0x1930)}
```

3.3 Function Pruner

Function Pruner removes unused code and compacts the shared library to reduce the size while maintaining the correct memory mapping. After obtaining the file ranges from Function Detector, Function Pruner first zeros out the rest code that is not in the file ranges which is considered bloat. This results in gaps between remained file ranges. To reduce the size of the shared library, we need to move the remained file ranges closer to each other. However, this is a non-trivial task. First, moving a file range to a new location will invalidate the offsets in the code. For example, if a function is moved to a new location, the offsets of the function in the code should be updated to the new location. Otherwise, errors will be incurred when executing the code. Updating all related offsets in the code is not possible because some offsets are calculated dynamically at runtime. Second, for an ELF format file, the file offset and memory address where the file offset is loaded should meet the congruence constraint:

$$\texttt{file\_offset} \equiv \texttt{memory\_address} \pmod{P} \qquad (1)$$

In other words, the file offset and the memory address should have the same modulo of the page size P. The page size is configurable and usually set to 0x1000 bytes.

Our solution to the first challenge is as follows: we map the file offsets to their original memory addresses where the original shared library is loaded into memory. In doing so,

the memory addresses of the file offsets are still valid even if the file offsets are moved, and it will not cause errors when executing the code. This mapping can be done by creating new program headers into the program header table [59].

The second challenge on the other hand, based on the solution we address the first challenge, can be formally described as an optimization problem as follows: Given a list of file ranges $\{[s_1, e_1), [s_2, e_2), \ldots, [s_n, e_n)\}$ output by Function Detector and their original memory addresses $\{[m_1, m_1+s_1-e_1), [m_2, m_2+s_2-e_2), \ldots, [m_n, m_n+s_n-e_n)\}$, the file ranges are sorted by s_i in ascending order and there is no overlap between two file ranges. Our goal is to move the file ranges to new locations, denoted as $\{[s'_1, e'_1), [s'_2, e'_2), \ldots, [s'_n, e'_n)\}$, such that the span of the file ranges is minimized under the congruence constraint, i.e., for any i, $s'_i \equiv m_i \pmod P$.

This constraint can be further simplified: since the original shared library already meets the congruence constraint, we have $s_i \equiv m_i \pmod P$. So the congruence for the problem can be simplified to for any i, $s'_i \equiv s_i \pmod P$. In other words, the new file offset s'_i should have the same modulo of the page size P as the original file offset s_i. Based on this analysis, we propose an $O(n)$ algorithm to approximately solve the problem.

Algorithm 1: Function Pruner Algorithm

```
   /* R is sorted by s_i in ascending order   */
   Data: R = {[s_1, e_1), [s_2, e_2), ..., [s_n, e_n)}
   Result: R' = {[s'_1, e'_1), [s'_2, e'_2), ..., [s'_n, e'_n)}
1  s'_1 = s_1;
2  e'_1 = e_1;
3  R' = {[s'_1, e'_1)};
4  for i from 2 to n do
5  |  s'_i = s_i - ⌊(s_i - e_{i-1}) / P⌋ × P;
6  |  e'_i = s'_i + e_i - s_i;
7  |  R' = R' ∪ {[s'_i, e'_i)};
8  |  s_i = s'_i;
9  |  e_i = e'_i;
10 end
```

As shown in Algorithm 1, Function Pruner iterates over all original file ranges. A file range $[s_i, e_i)$ will be moved forward by $\lfloor \frac{s_i - e_{i-1}}{P} \rfloor \times P$ bytes. The reason is that since $s'_i \equiv s_i \pmod P$, we have $s'_i = s_i - k * P$, where k is a non-negative integer. In other words, a file range will not be moved if the gap between two adjacent file ranges is smaller than the page size ($k = 0$). A file range will be moved forward by an integer multiple of the page size, if the gap is larger than the page size $k > 0$; By moving the file range forward, it is closer to the previous file range and the total size of the shared library is reduced. Also note that after the file range is moved forward, the space that the file range occupied before will also be zeroed out and treated as a new gap (lines 8-9).

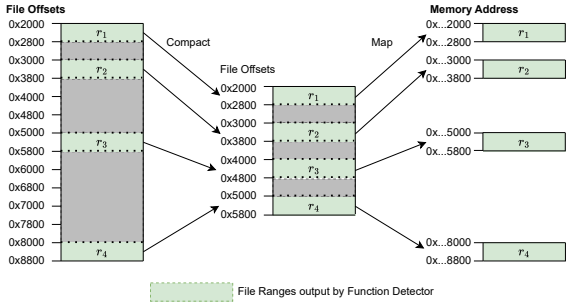


Figure 3. An example of function removal.

In doing so, the congruence constraint is met, and the shared library is compacted.

Figure 3 shows an example of Algorithm 1. P is set to `0x1000` bytes. Initially, there are 4 file ranges of the shared library, (r_1 to r_4), which are obtained from Function Detector. These file ranges need to be kept in the shared library. The rest code is zeroed out, as illustrated by the grey areas. The shared library is compacted by moving the file ranges closer to each other using Algorithm 1. The gap between r_1 and r_2 is smaller than P. So r_2 is not moved. The gap between r_2 and r_3 is `0x1800` bytes, so r_3 is moved forward by `0x1000` ($1 \times P$) bytes. Since r_3 is moved forward, the space it occupied originally is zeroed out too. Therefore, the gap between new r_3 and r_4 is `0x3800` bytes, so r_4 is moved forward by `0x3000` ($3 \times P$) bytes. After the compaction, the size of the shared library is reduced by `0x3000` bytes. When the compacted shared library is loaded into memory, the file offsets are mapped to their original memory addresses as shown on the right side of the figure. The congruence constraint is met and the size of the shared library is reduced. We note that there are four discontinuous memory segments. The memory gap between any two segments is not occupied and can be re-used by the operating system. In theory, the used memory is reduced from `0x6800` to `0x2000` bytes. However, in practice, the memory reduction may be less than that due to memory fragmentation. We leave addressing the memory fragmentation problem for future work.

3.4 LibGuard

Although Function Tracer is designed to detect all used functions, some used functions still may be missed due to the complexity of shared libraries. This issue is also faced by other Type II debloaters [13, 45, 59]. To mitigate this issue, we propose a post-debloating verification tool called LibGuard to ensure no used functions are removed. LibGuard is considered as an experimental feature and is mostly not used in our evaluation, so it is not shown in Figure 2. LibGuard works as follows: LibGuard runs the application with the debloated library produced by Function Pruner. If any necessary functions are mistakenly removed, the workload crashes, generating a core dump file. LibGuard analyzes the

core dump to get the address causing the crash, maps this address to the file offset within the shared library, and determines the corresponding function name. The identified function is deemed used and kept in the shared library. This process iterates until the workload runs normally, indicating no missing functions.

In §4, we evaluate 21 applications involving hundreds of shared libraries. Negativa effectively debloats all libraries *without* requiring LibGuard, with one exception: a shared library injected with malicious code, which will be discussed in §4.4.

3.5 Implementation

We develop a prototype of Negativa with over 9,000 lines of Rust code. Negativa has two primary components: Function Tracer and Function Pruner. The Function Tracer integrates the capabilities of both Library Detector(§3.1) and Function Detector(§3.2). It identifies used libraries and the functions within shared libraries. The Function Pruner, as the name implies, removes unused functions and compacts the shared libraries as described in §3.3. This implementation allows Negativa to be used solely as a dynamic analysis tracing tool without Function Pruner feature, which can be integrated with existing debloating tools. Additionally, it can be used for tasks like debugging or security analysis.

4 Experiments

We evaluate Negativa's performance using a wide set of applications and use cases. We focus on evaluations of the performance w.r.t. the overheads, bloat reductions, robustness, and security. All experiments were conducted on an AWS instance with 8×3.1GHz CPUs and 32 GB of memory.

4.1 Evaluation on Debloater-Eval Benchmark

We first evaluate Negativa's performance using **Debloater-Eval Benchmark** [21], a recently released debloating benchmark containing 20 applications. The applications are categorized according to their complexity levels: low (10), medium (6), and high (4). Each of the applications has a set of preselected workloads to be executed for debloating. Additionally, the benchmark includes the debloating results of nine *state-of-the-art* debloaters[2]. We compare the debloating results of Negativa with those of other debloaters reported using the benchmark. In line with the metrics used by the benchmark, we assessed the following three aspects; evaluating the performance of the debloaters; measuring the performance of the debloated applications; and the correctness of the debloated applications.

Tool Performance. Table 1 presents the performance metrics of the debloaters. Each debloater was executed 10 times on each application. The #Benchmark column shows the number of applications successfully debloated by each debloater. An application is considered successfully debloated if the retained features can be executed correctly. Negativa is the only debloater that successfully debloated all low-, medium- and high-complexity applications. The Run Time and Peak Memory columns report the average run time and average peak memory usage of a debloater to debloat an application without encountering errors. If a debloater fails to successfully debloat all applications of a particular complexity level, its runtime and memory usage for that level are marked as "-". To produce the debloated applications, Negativa uses the least time across all complexity levels and uses a maximum of 37 MB of memory for all applications. For high-complexity applications, Negativa uses the least peak memory among all debloaters. For other applications, Negativa is comparable to RAZOR which uses slightly lower memory. However, RAZOR fails to debloat half the medium-complexity applications. We note that for each application, RAZOR only debloats a single executable binary while Negativa debloats all used shared libraries. Negativa uses much less memory and time if it only debloats a single library for each application.

Program Performance. Table 2 displays the performance results of the debloated applications compared to their original versions. As Negativa is a shared library debloater, the static binary size for Negativa is computed by comparing the aggregate file sizes of the debloated shared libraries with their original versions. Negativa demonstrates notable static binary size reductions, with the size after debloating being 63.6% (2nd best), 74.3% (2nd best) and 67.6% (the best) of the original size across the three complexity levels, respectively. However, the actual average program run time increases slightly when using Negativa due to the increased number of program headers in the debloated shared libraries. We note that all debloating tools have produced debloated binaries with increases in run time. In terms of peak memory usage, Negativa achieves slight reductions for the low- and medium-complexity applications. For the high-complexity applications, Negativa performs the best, with the debloated applications using only 88.3% of the peak memory of their original versions. We further investigated the change in peak memory for each of the four high-complexity applications included in the benchmark: Although `nginx` has no meaningful reduction in the peak memory usage, `poppler`, `imagemagick`, and `nmap` use only 81.2%, 80.4%, and 91.4% of the original peak memory, respectively. This may suggest that debloating tends to yield more benefits for high-complexity applications.

Correctness. It is important to note that Negativa aims to retain user-needed features and remove unnecessary functions for those features. Therefore, we evaluate the correctness of a debloated application by assessing the retained features: A debloated application *must* successfully execute the

[2]Debloater-Eval Benchmark evaluated 10 debloaters. One of them failed to generate any functional applications. Therefore, the results of that debloater is not included in this paper

Table 1. Number of successfully debloated applications, average Run Time (CPU minutes) and average Peak Memory per debloater. The best results are marked in bold.

	#Benchmarks			Run Time(Min)			Peak Memory(MB)		
Debloater	Low	Medium	High	Low	Medium	High	Low	Medium	High
CHISEL [28]	7	2	0	282	224	-	306	91	-
CHISEL-GT [28]	7	2	1	4078	4058	13350	2145	1675	3011
RAZOR [45]	**10**	3	2	**<1**	**<1**	10	**32**	**31**	71
Binary Reduce (Dyn.) [1]	0	0	1	-	-	3	-	-	1085
Trimmer (v2) [Agg.] [14, 52]	6	1	2	2	90	18	3442	6518	5270
OCCAM (v2) [Agg.] [41]	**10**	1	0	**<1**	150	-	73	5565	-
LMCAS-SIFT [Agg.] [15]	7	0	0	**<1**	-	-	458	-	-
Binary Reduce (Static) [1]	9	4	0	1	7	-	415	1577	-
Libfilter [13]	8	2	0	**<1**	**<1**	-	304	503	-
Negativa	**10**	**6**	**4**	**<1**	**<1**	**2**	37	37	**37**

Table 2. Average Change in Static Binary Size, application Run Time and Peak Memory. The best results are marked in bold. Only the results of debloaters that successfully debloated 50% or more of the applications are listed.

	Static Binary Size			Run Time			Peak Memory		
Debloater	Low	Medium	High	Low	Medium	High	Low	Medium	High
CHISEL-GT	80.9%	79.7%	96.6%	97.6%	100.7%	101.8%	99.6%	99.7%	100.4%
RAZOR	117.2%	101.5%	107.3%	95.8%	106.3%	**100.9%**	99.8%	99.4%	100.6%
OCCAM (v2) [Agg.]	80.5%	74.7%	-	128.2%	175.7%	-	110.3%	101.4%	-
Binary Reduce (Static)	**22.2%**	**46.5%**	-	**95.2%**	101.0%	-	**97.2%**	98.2%	-
Libfilter	101.3%	101.1%	-	102.4%	**100.4%**	-	100.3%	100%	-
Negativa	63.6%	74.3%	**67.6%**	108.8%	101.2%	106.3%	99.5%	**97.9%**	**88.3%**

retained features. To do that, we use Differ [21] to fuzz test the retained features for a debloated application. As reported by Brown et al. [21], none of the existing debloaters pass the fuzz tests for all applications in Debloater-Eval Benchmark, which pose a significant concern for the correctness of these debloaters. In our testing, all debloated applications produced by Negativa successfully pass the fuzz test for retained features. Moreover, we also observed that removing unnecessary functions for user-needed features also disabled the features not needed by users. For example, for the debloated `objdump` application, the user-needed features involve using the application with the `-x` option only. The debloated `objdump` passes all fuzz tests related to this feature. Additionally, it fails to execute the `-D` and `–info` options, which is the desired behavior as these options are not user-needed.

4.2 Tracer Evaluation

Function Tracer is a critical component of Negativa. Since it can be used as a standalone component, in this section we evaluate the effectiveness and efficiency of Function Tracer by comparing it with three other function tracers: `ltrace` [8], `drltrace` [3] and `TinyTracer` [10]. `drltrace` and `TinyTrace` utilizes the same tracing frameworks used by RAZOR [45]. We evaluate the effectiveness of a tracer based on two key metrics, namely, the number of used shared libraries and the number of used functions detected by each tracer. An effective tracer should be able to detect all the shared libraries used by an application, as well as all the functions used within those libraries.

We use an Nginx server to perform our evaluation. We start the server while tracing it with each tracer. We send a request to the server, wait for a response, and then stop the server. This represents a minimal sample workload for Nginx. This minimal workload is sufficient for comparison because our goal here is to compare the effectiveness of the tracers rather than the debloating of the shared libraries.

The results from our experiments are shown in Table 3. Negativa's Function Tracer detects six shared libraries, while the other tracers only detect four; `libdl.so` and `libz.so` are not detected by any of the other three tracers, yet these libraries are essential for Nginx, as their removal causes startup failures. For the detected shared libraries, we use the tracers to detect the used functions in these libraries. Negativa detects more functions than the other tracers for all the detected libraries. If we remove the functions detected by Negativa but missed by the other tracers, the sample workload fails. For example, in the `libssl` library, Negativa detects three additional functions compared to the other tracers: `SSL_rstate_string`, `SSL_add_dir_cert_subjects_to_`

Table 3. Number of detected functions of each shared library by each tracer. A dash indicates the shared library is not detected by the tracer.

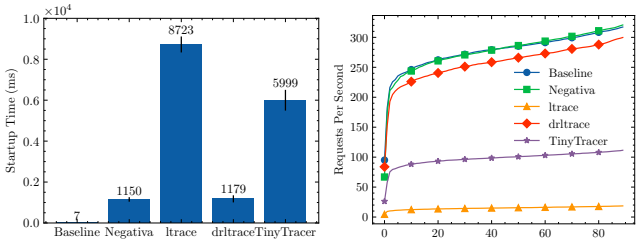
Library	ltrace	TinyTracer	dltrace	Function Tracer
libpthread.so	28	9	25	**47**
libc.so	105	52	126	**224**
libcrypto.so	309	2	309	**330**
libdl.so	-	-	-	**1**
libssl.so	2	1	2	**5**
libz.so	-	-	-	**1**

stack, and ERR_load_SSL_strings. Removing these functions causes the Nginx server to fail to handle the request. These results demonstrate that Negativa is more effective than the other tracers, which is crucial for correct debloating.

We also utilize the Nginx server to evaluate efficiency, i.e., the performance overhead introduced by each tracer. Similar to the previous experiment, we start the server with the tracers and perform load tests using Locust [7] to measure the maximum requests per second (RPS) the server can handle. We also run the same workload on an untraced Nginx server to obtain a baseline. We repeat the experiment ten times.

Figure 4a presents a comparison of the startup times between the traced and original servers. All tracers introduce noticeable overhead to the startup process of Nginx. However, Negativa and drltrace show significantly less overhead compared to the other two tracers. For the throughput (RPS), Figure 4b shows that Negativa has comparable performance to the original server, whereas the other three tracers exhibit noticeable performance degradation. Note that Negativa only traces a function once when it is called the first time. Hence, for the first few requests of a workload, Negativa will introduce some overheads. After that, only negligible overheads occur. This is why Negativa shows a noticeable overhead at the server startup process, but the RPS is comparable to the original server. These results indicate that Negativa is both efficient and effective at tracing used functions within shared libraries, introducing minimal performance overhead while maintaining accurate tracing capabilities.

An effective and efficient tracer has important implications for software debloating. The intrinsic limitation of offline-debloating is that offline running workloads for debloating may not reflect the actual usage of the application, leading to a broken application when the debloated application is deployed in a production environment. In contrast, Function Tracer, due to its low overhead, offers the potential for online debloating. Online debloating involves deploying the application in a production environment and using Function Tracer to trace it. Negativa debloats the shared libraries based on the actual usage in the production environment. This will result in more reliable debloating, as it is based on the actual usage of the application. For example, consider a developer who wants to debloat an Nginx server. Instead of running



(a) Startup Time **(b)** RPS

Figure 4. Server startup time and RPS comparison.

the server with pre-selected workloads, the developer can deploy it in a production environment and trace it using Function Tracer. After running for a period of time, Negativa debloats the shared libraries based on the actual usage of the server. Thus, in addition to the traditional offline debloating approach, we anticipate that Negativa can also be effectively utilized for online debloating.

4.3 Debloating an ML Framework: PyTorch

Machine learning frameworks are significantly more complex than traditional applications[50, 58]. A typical ML training task may rely on hundreds of shared libraries, with a combined size of several gigabytes [58]. To date, no existing debloaters have been evaluated on applications of such complexity. In this section, we demonstrate Negativa's capability to debloat such complex applications. We debloat the CPU version of the popular PyTorch framework [43] by running two typical ML workloads: training and inference, respectively. Specifically, the training workload involves using PyTorch to train MobileNetV2 [49] on the CIFAR-10 dataset [33] for two epochs. MobileNetV2 is a lightweight deep learning model that is widely used in mobile and edge devices [25, 38, 54]. The inference workload involves using the trained model to classify the test set of the CIFAR-10 dataset.

The debloating results for the two workloads are shown in Table 4. Both the training and inference workloads use 91 shared libraries. The original total file size of the 91 shared libraries is 564 MB, which is much larger than that of the applications evaluated in Debloater-Eval Benchmark which have a maximum size of only 9 MB. Negativa reduced the file size by 58.2% and 58.9% for both training and inference workloads, respectively. The original number of functions in the shared libraries is 451,376. Negativa removes 91.3% and 91.7% of the functions for the training and inference workloads, respectively. The inference workload has a slightly higher reduction in file size and function number than the training workload. This is because training a model involves both the forward and backward pass, requiring more functions than the inference workload, which only includes the forward pass.

We ran the same workloads 10 times using the debloated shared libraries and compared the runtime performance with

Table 4. Static Binary Size, function count, Peak Memory, Run Time (and reductions in percentage) of the debloated PyTorch .

Workload	#Libs	Static Binary Size (MB)	#Functions	Peak Memory (MB)	Run Time (s)
Training	91	564 (58.2%)	451,376 (91.3%)	610 (6.7%)	1533(-0.9%)
Inference	91	564 (58.9%)	451,376 (91.7%)	330 (11.8%)	106 (-3.6%)

the original libraries. As shown in Table 4, the peak memory usage of the debloated libraries is reduced by 6.7% for training workloads and 11.8% for inference workloads. Inference time experienced a slight increase, averaging 3.6% slower than with the original libraries, while training time saw a smaller increase of just 0.9%. The overhead from multiple program headers in the debloated libraries occurs only during the loading phase and happens only once. The longer the workload runs, the less significant the relative overhead becomes. We expect debloating to result in even lower performance overheads for long-running workloads, such as ML model training or web server operations.

4.4 Security Impacts

In this section, we study how Negativa improves the security of debloated applications.

We first evaluate the gadget count reduction in the debloated applications for the four high-complexity applications. Gadgets are short instruction sequences present in the program that end in a return, indirect jump, or indirect call instruction. Gadgets can be chained together to perform code-reuse attacks. Based on the way to use gadgets, gadgets are categorized into four types: Return-Oriented Programming (ROP), Jump-Oriented Programming (JOP), Call-Oriented Programming (COP), and Special Purpose (S.P.) gadgets [23, 51]. As Table 5 shows, in all applications and for all gadget types, Negativa achieves a significant reduction in the gadget count, with the total reduction ranging from 50.2% to 78.0%. Although previous work shows that debloating introduces new gadgets from 30% to 70% [23], Negativa introduces less than 1% for all gadget types in all applications.

We also use three metrics suggested in the literature [21] to evaluate the security impacts of debloating: Gadget Set Expressivity, Gadget Set Quality, and S.P. Gadget Availability. Gadget Set Expressivity measures the collective expressive power of functional gadgets in a binary (or library) to determine if they are sufficient to launch practical exploits. A *positive* value is desired, indicating that the gadget set in the debloated application has lower expressivity compared to its original version, thereby reducing the potential for exploits. A change of 2 or more is considered significant [21]. Gadget Set Quality assesses the side constraints imposed on gadgets, which affect their usability for launching practical exploits. A *negative* value is desirable for this metric, signifying that the number of side constraints per gadget has increased after debloating. A change of 0.5 or more in either direction is considered significant. S.P. gadget availability measures the availability of special purpose code reuse gadgets. A *positive* value is desired for this metric, indicating that the availability of special purpose gadgets in the debloated application is lower than its original version. We refer the reader to the details of the definitions in the original work [23]. Table 6 presents the average changes for the above security metrics for Negativa in comparison to the other debloaters considered in the debloating benchmark. Negativa is the only debloater that shows significant improvements for both expressivity and S.P. gadget availability, while not degrading the gadget set quality.

4.5 A Case Study on XZ Backdoor

In our final experiment, we present a case study showing how Negativa handles the XZ backdoor vulnerability, i.e., CVE-2024-3094 [9]. XZ is a set of open-source command-line tools and libraries for data compression, which is installed by default on most Linux distributions. Recently, it was discovered that an attacker injected malicious code into a shared library of XZ, `liblzma.so` [11]. The malicious code is loaded into memory when an SSH server is started, allowing the attacker to connect to the SSH server with his own private key and execute arbitrary code. This vulnerability has a CVSS score of 10 out of 10, indicating the most critical security risk. Assuming users have installed the compromised `liblzma.so` library on their systems, we investigate whether Negativa can remove the malicious code from the shared library.

To do this, we first reproduce the backdoor. We started an SSH server with the compromised `liblzma.so` library so that the server can be connected in two ways: (1) using a normal SSH connection, and (2) using the backdoor connection. To debloat the shared library, we connect to it using a normal SSH connection, which acts as the normal debloating workload. Then we use Negativa with the LibGuard feature to debloat the shared library. Subsequently, we start the SSH server with the debloated library. We successfully connected to the SSH server using a normal SSH connection, but failed to connect using the backdoor connection. This indicates that Negativa successfully removed the malicious code from the shared library, preventing the backdoor from being exploited.

Moreover, we compare the functions used by a normal connection and the backdoor connection. Traced by Function Tracer, we start the SSH server with the compromised library. Then we connect the server using a normal SSH connection. Function Tracer detected 26 functions in the

Table 5. Gadget count (and reduction in percentage) after debloating, categorized by gadget types for the four high-complexity applications.

Application	ROP Gadgets	JOP Gadgets	COP Gadgets	S.P. Gadgets	Total
nginx	2,370 (75.4%)	333 (72.7%)	637 (91.1%)	713 (77.7%)	4,053 (78.0%)
poppler	19,690 (70.8%)	4,730 (77.8%)	14,834 (67.9%)	1,318 (81.0%)	40,572 (70.9%)
imagemagick	14,520 (52.6%)	4,628 (54.5%)	5,177 (38.1%)	1,405 (56.2%)	25,730 (50.2%)
nmap	19,754 (74.7%)	3,330 (85.1%)	7,271 (78.7%)	1,475 (81.3%)	31,830 (77.0%)

Table 6. Average changes in Gadget Set Expressivity, Quality and S.P. Gadget Availability. Only the results of debloaters that successfully debloated 50% or more of the applications are listed.

	Gadget Set Expressivity			Gadget Set Quality			S.P. Gadget Types Available		
Debloater	Low	Medium	High	Low	Medium	High	Low	Medium	High
CHISEL-GT	0.3	0.7	0.5	0.1	0.2	**0.1**	-0.6	-2.3	-1.5
RAZOR	-0.3	-0.3	-0.5	0.2	0	**0.1**	-0.9	0	**0.5**
OCCAM (v2) [Agg.]	1.3	0.5	-	0.2	0	-	0.4	-0.5	-
Binary Reduce (Static)	0.3	-0.5	-	0.3	0.1	-	**2.9**	**2**	-
Libfilter	1	1	-	0	0	-	0	0	-
Negativa	**2**	**1.7**	**0.8**	**-0.1**	**0.0**	**0.1**	1.2	0.8	0.25

compromised `liblzma.so` used by this workload. Similarly, we repeat the process but connect the server using the backdoor connection. Surprisingly, 39 functions are detected by Function Tracer. This indicates that the backdoor connection used 13 additional functions in `liblzma.so` compared to the normal connection. And these additional functions were successfully removed by Negativa, which is why the backdoor connection failed to connect to the SSH server.

5 Discussion

There has been extensive research on software debloating. Existing debloating tools struggle to improve security, fail to produce functioning debloated applications for complex applications, and suffer from usability issues. In this study, we propose Negativa, a novel shared library debloater that neither requires access to source code nor recompilation.

We evaluated Negativa on various applications. The results are promising. To the best of our knowledge, Negativa is the only debloater that successfully debloats all applications in Debloater-Eval Benchmark. The reduction in file size and memory usage offers several advantages, including lower storage and memory requirements, which is crucial for mobile and edge devices with limited resources. Additionally, if the application needs to be transferred over a network—such as being packaged into a container or software distribution package—the smaller file size saves network bandwidth and energy. Furthermore, the intrinsic limitation of offline debloating - the workloads may not represent the actual end-use of the application - is one of the main reasons why debloating is not widely used in industry. The low overhead of Function Tracer makes it possible for online debloating, thus addressing this limitation of offline debloating.

One anticipated use case for Negativa is in serverless services. Serverless services package applications with well-defined features into containers that often include numerous shared libraries, leading to bloat [58]. Negativa can be applied to debloat these serverless services, reducing file size, network bandwidth usage, energy consumption, memory overhead, and potential security risks.

Like all debloaters, Negativa incurs some overhead. It introduces additional program headers into debloated shared libraries, resulting in longer loading times. This overhead is primarily due to current operating systems not being considering software debloating. A new system loader, designed to support debloated shared libraries, could be developed to mitigate this overhead. This highlights that debloating is not a standalone issue but part of a broader set of challenges within the current software stack that must be addressed collectively.

6 Conclusion

In this work, we proposed Negativa, a novel shared library debloater without requiring source code or recompilation. First, Negativa utilizes a novel dynamic tracing techniques to detect used functions used shared libraries. Then, it uses a novel algorithm that safely removes unused functions and reduce the shared library size while maintaining memory address mapping correct. Negativa was evaluated on Debloater-Eval Benchmark, successfully debloating all 20 applications, as well as the complex PyTorch framework, which includes 91 shared libraries. It was able to eliminate up to 91% of unused functions, resulting in a reduction of up to 59% in file size and 20% in memory usage, all while enhancing security.

References

[1] Binary Reduction — grammatech.github.io. https://grammatech.github.io/prj/binary-reduce/. [Accessed 10-09-2024].

[2] ELF Header (Linker and Libraries Guide) — docs.oracle.com. https://docs.oracle.com/cd/E19683-01/816-1386/chapter6-43405/index.html. [Accessed 08-10-2024].

[3] GitHub - mxmssh/drltrace: Drltrace is a library calls tracer for Windows and Linux applications. — github.com. https://github.com/mxmssh/drltrace. [Accessed 28-08-2024].

[4] GNU_IFUNC - glibc wiki — sourceware.org. https://sourceware.org/glibc/wiki/GNU_IFUNC. [Accessed 27-09-2024].

[5] Guide: Function Calling Conventions — delorie.com. https://www.delorie.com/djgpp/doc/ug/asm/calling.html. [Accessed 29-08-2024].

[6] Home — dynamorio.org. https://dynamorio.org/. [Accessed 28-08-2024].

[7] Locust.io — locust.io. https://locust.io/. [Accessed 07-10-2024].

[8] ltrace — ltrace.org. https://ltrace.org/. [Accessed 08-10-2024].

[9] NVD - CVE-2024-3094 — nvd.nist.gov. https://nvd.nist.gov/vuln/detail/CVE-2024-3094. [Accessed 26-09-2024].

[10] Tiny tracer: A pin tool for tracing api calls. https://github.com/hasherezade/tiny_tracer. [Accessed 16-09-2024].

[11] xz-utils backdoor situation (CVE-2024-3094) — gist.github.com. https://gist.github.com/thesamesam/223949d5a074ebc3dce9ee78baad9e27. [Accessed 18-10-2024].

[12] Muhammad Abubakar, Adil Ahmad, Pedro Fonseca, and Dongyan Xu. {SHARD}:{Fine-Grained} kernel specialization with {Context-Aware} hardening. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2435–2452, 2021.

[13] Ioannis Agadakos, Di Jin, David Williams-King, Vasileios P Kemerlis, and Georgios Portokalidis. Nibbler: debloating binary shared libraries. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 70–83, 2019.

[14] Aatira Anum Ahmad, Abdul Rafae Noor, Hashim Sharif, Usama Hameed, Shoaib Asif, Mubashir Anwar, Ashish Gehani, Fareed Zaffar, and Junaid Haroon Siddiqui. Trimmer: an automated system for configuration-based software debloating. *IEEE Transactions on Software Engineering*, 48(9):3485–3505, 2021.

[15] Mohannad Alhanahnah, Rithik Jain, Vaibhav Rastogi, Somesh Jha, and Thomas Reps. Lightweight, multi-stage, compiler-assisted application specialization. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*, pages 251–269. IEEE, 2022.

[16] Muaz Ali, Muhammad Muzammil, Faraz Karim, Ayesha Naeem, Rukhshan Haroon, Muhammad Haris, Huzaifah Nadeem, Waseem Sabir, Fahad Shaon, Fareed Zaffar, et al. Sok: A tale of reduction, security, and correctness-evaluating program debloating paradigms and their compositions. In *European Symposium on Research in Computer Security*, pages 229–249. Springer, 2023.

[17] Sumaya Almanee, Arda Ünal, Mathias Payer, and Joshua Garcia. Too quiet in the library: An empirical study of security updates in android apps' native code. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1347–1359. IEEE, 2021.

[18] Babak Amin Azad, Rasoul Jahanshahi, Chris Tsoukaladelis, Manuel Egele, and Nick Nikiforakis. {AnimateDead}: Debloating web applications using concolic execution. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 5575–5591, 2023.

[19] Babak Amin Azad, Pierre Laperdrix, and Nick Nikiforakis. Less is more: Quantifying the security benefits of debloating web applications. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1697–1714, 2019.

[20] Priyam Biswas, Nathan Burow, and Mathias Payer. Code specialization through dynamic feature observation. In *Proceedings of the Eleventh ACM Conference on Data and Application Security and Privacy*, pages 257–268, 2021.

[21] Michael D Brown, Adam Meily, Brian Fairservice, Akshay Sood, Jonathan Dorn, Eric Kilmer, and Ronald Eytchison. A broad comparative evaluation of software debloating tools. pages 3927–3943, 2024.

[22] Michael D Brown and Santosh Pande. Carve: Practical security-focused software debloating using simple feature set mappings. In *Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation*, pages 1–7, 2019.

[23] Michael D Brown and Santosh Pande. 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)*, 2019.

[24] Bobby R Bruce, Tianyi Zhang, Jaspreet Arora, Guoqing Harry Xu, and Miryung Kim. Jshrink: In-depth investigation into debloating modern java applications. In *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 135–146, 2020.

[25] Yu-Chen Chiu, Chi-Yi Tsai, Mind-Da Ruan, Guan-Yu Shen, and Tsu-Tian Lee. Mobilenet-ssdv2: An improved object detection model for embedded systems. In *2020 International conference on system science and engineering (ICSSE)*, pages 1–5. IEEE, 2020.

[26] Jake Christensen, Ionut Mugurel Anghel, Rob Taglang, Mihai Chiroiu, and Radu Sion. {DECAF}: Automatic, adaptive de-bloating and hardening of {COTS} firmware. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1713–1730, 2020.

[27] CISA. Exploring memory safety in critical open source projects. Technical report, CISA, FBI, ASD-ACSC, and CCCS, 2024.

[28] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. Effective program debloating via reinforcement learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 380–394, 2018.

[29] Bert Hubert. Why bloat is still software's biggest vulnerability: A 2024 plea for lean software. *IEEE Spectrum*, 61(4):22–50, 2024.

[30] Rasoul Jahanshahi, Babak Amin Azad, Nick Nikiforakis, and Manuel Egele. Minimalist: Semi-automated debloating of {PHP} web applications through static analysis. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 5557–5573, 2023.

[31] Igibek Koishybayev and Alexandros Kapravelos. Mininode: Reducing the attack surface of node. js applications. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pages 121–134, 2020.

[32] Hyungjoon Koo, Seyedhamed Ghavamnia, and Michalis Polychronakis. Configuration-driven software debloating. In *Proceedings of the 12th European Workshop on Systems Security*, pages 1–6, 2019.

[33] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[34] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4):323–337, 1992.

[35] James R. Larus. Efficient program tracing. *Computer*, 26(5):52–61, 1993.

[36] Jiakun Liu, Xing Hu, Ferdian Thung, Shahar Maoz, Eran Toch, Debin Gao, and David Lo. Autodebloater: Automated android app debloating. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 2090–2093. IEEE, 2023.

[37] Jiakun Liu, Zicheng Zhang, Xing Hu, Ferdian Thung, Shahar Maoz, Debin Gao, Eran Toch, Zhipeng Zhao, and David Lo. Minimon: Minimizing android applications with intelligent monitoring-based debloating. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[38] Xuefeng Liu, Zhenqing Jia, Xiaoke Hou, Min Fu, Li Ma, and Qiaoqiao Sun. Real-time marine animal images classification by embedded system based on mobilenet and transfer learning. In *OCEANS 2019-Marseille*, pages 1–5. IEEE, 2019.

[39] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices*, 40(6):190–200, 2005.

[40] Gregory Malecha, Ashish Gehani, and Natarajan Shankar. Automated software winnowing. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 1504–1511, 2015.

[41] Jorge A Navas and Ashish Gehani. Occam-v2: combining static and dynamic analysis for effective and efficient whole-program specialization. *Communications of the ACM*, 66(4):40–47, 2023.

[42] Pardis Pashakhanloo, Aravind Machiry, Hyonyoung Choi, Anthony Canino, Kihong Heo, Insup Lee, and Mayur Naik. Pacjam: Securing dependencies continuously via package-oriented debloating. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, pages 903–916, 2022.

[43] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[44] Chris Porter, Girish Mururu, Prithayan Barua, and Santosh Pande. Blankit library debloating: Getting what you want instead of cutting what you don't. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 164–180, 2020.

[45] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. {RAZOR}: A framework for post-deployment software debloating. In *28th USENIX security symposium (USENIX Security 19)*, pages 1733–1750, 2019.

[46] Anh Quach, Aravind Prakash, and Lok Yan. Debloating software through {Piece-Wise} compilation and loading. In *27th USENIX security symposium (USENIX Security 18)*, pages 869–886, 2018.

[47] Anh Quach, Aravind Prakash, and Lok Yan. Debloating software through {Piece-Wise} compilation and loading. In *27th USENIX security symposium (USENIX Security 18)*, pages 869–886, 2018.

[48] Nilo Redini, Ruoyu Wang, Aravind Machiry, Yan Shoshitaishvili, Giovanni Vigna, and Christopher Kruegel. B in t rimmer: Towards static binary debloating through abstract interpretation. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19–20, 2019, Proceedings 16*, pages 482–501. Springer, 2019.

[49] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[50] David Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. Hidden technical debt in machine learning systems. *Advances in neural information processing systems*, 28, 2015.

[51] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561, 2007.

[52] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. Trimmer: application specialization for code debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 329–339, 2018.

[53] César Soto-Valero, Thomas Durieux, Nicolas Harrand, and Benoit Baudry. Coverage-based debloating for java bytecode. *ACM Transactions on Software Engineering and Methodology*, 32(2):1–34, 2023.

[54] Qian Xiang, Xiaodan Wang, Rui Li, Guoling Zhang, Jie Lai, and Qingshuang Hu. Fruit image classification based on mobilenetv2 with transfer learning technique. In *Proceedings of the 3rd international conference on computer science and application engineering*, pages 1–7, 2019.

[55] Qi Xin, Qirun Zhang, and Alessandro Orso. Studying and understanding the tradeoffs between generality and reduction in software debloating. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–13, 2022.

[56] Renjun Ye, Liang Liu, Simin Hu, Fangzhou Zhu, Jingxiu Yang, and Feng Wang. Jslim: Reducing the known vulnerabilities of javascript application by debloating. In *International Symposium on Emerging Information Security and Applications*, pages 128–143. Springer, 2021.

[57] Haotian Zhang, Mengfei Ren, Yu Lei, and Jiang Ming. One size does not fit all: security hardening of mips embedded systems via static binary debloating for shared libraries. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 255–270, 2022.

[58] Huaifeng Zhang, Mohannad Alhanahnah, Fahmi Abdulqadir Ahmed, Dyako Fatih, Philipp Leitner, and Ahmed Ali-Eldin. Machine learning systems are bloated and vulnerable. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 8(1):1–30, 2024.

[59] Andreas Ziegler, Julian Geus, Bernhard Heinloth, Timo Hönig, and Daniel Lohmann. Honey, i shrunk the elfs: Lightweight binary tailoring of shared libraries. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(5s):1–23, 2019.