Nibbler: Debloating Binary Shared Libraries

Ioannis Agadakos*
ioannis.agadakos@sri.com
SRI International

Di Jin
di_jin@brown.edu
Brown University

David Williams-King
dwk@cs.columbia.edu
Columbia University

Vasileios P. Kemerlis
vpk@cs.brown.edu
Brown University

Georgios Portokalidis
gportoka@stevens.edu
Stevens Institute of Technology

ABSTRACT

Developers today have access to an arsenal of toolkits and libraries for rapid application prototyping. However, when an application loads a library, the entirety of that library's code is mapped into the address space, even if only a single function is actually needed. The unused portion is *bloat* that can negatively impact software defenses by unnecessarily inflating their overhead or increasing their attack surface. Recent work has explored debloating as a way of alleviating the above problems, when source code is available. In this paper, we investigate whether debloating is possible and practical at the binary level. To this end, we present *Nibbler*: a system that identifies and erases unused functions within shared libraries. Nibbler works in tandem with defenses like continuous code re-randomization and control-flow integrity, enhancing them without incurring additional run-time overhead. We developed and tested a prototype of Nibbler on x86-64 Linux; Nibbler reduces the size of shared libraries and the number of available functions, for real-world binaries and the SPEC CINT2006 suite, by up to 56% and 82%, respectively. We also demonstrate that Nibbler benefits defenses by showing that: (i) it improves the deployability of a continuous re-randomization system for binaries, namely Shuffler, by increasing its efficiency by 20%, and (ii) it improves certain fast, but coarse and context-insensitive control-flow integrity schemes by reducing the number of gadgets reachable through returns and indirect calls by 75% and 49% on average.

CCS CONCEPTS

• **Security and privacy** → **Systems security**; **Software and application security**; *Software security engineering*; *Software reverse engineering*; Information flow control.

KEYWORDS

Code debloating, Static binary analysis, Software security

*Work done while at Stevens Institute of Technology.

ACM Reference Format:
Ioannis Agadakos, Di Jin, David Williams-King, Vasileios P. Kemerlis, and Georgios Portokalidis. 2019. Nibbler: Debloating Binary Shared Libraries. In *2019 Annual Computer Security Applications Conference (ACSAC '19), December 9–13, 2019, San Juan, PR, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3359789.3359823

1 INTRODUCTION

Software developers rely heavily on shared libraries for rapid application prototyping and development. However, as they are utilized by more and more diverse applications, they grow in complexity and size, accumulating an abundance of new features, while retaining old, potentially unused ones. When an application loads a shared library, all of this functionality is included in all of the application's processes, even if only a single function is actually used.

This *bloat* of code affects binary programs and libraries, which frequently suffer from critical vulnerabilities [18] that enable attackers to compromise them despite broadly-deployed defenses, such as data execution prevention (DEP) [4] and address-space layout randomization (ASLR) [52]. Code bloat impedes the adoption of novel defenses, like continuous code re-randomization [6, 14, 74, 76] (e.g., because of increased run-time overhead), while it can also restrict the effectiveness of others, like control-flow integrity (CFI) [1] (i.e., because of over-permissiveness).

Recent work [53] has explored debloating for applications and libraries by proposing an LLVM-based framework that analyzes code at compile time and embeds function-dependency metadata in the emitted binaries. That information is used by a modified loader to *debloat* libraries, dynamically-loaded by the application, by overwriting unused shared-library code. Its results confirmed that a large part of library code is indeed not needed by applications, and, therefore, it is possible to debloat them without restricting their functionality. The question this paper aims to answer is: *is it possible to debloat binary-only software, to what extent, and what are the security benefits?* Binary-only, *dynamically-linked* or *shared* libraries can still be found in many settings: commercial software is usually distributed without source code, and even open-source software may depend on legacy binary-only, shared libraries.

To answer this question, we design and implement *Nibbler*: a system that analyzes binary applications and the libraries they depend on to identify and erase *unused* library code. Nibbler generates *thin* versions of shared libraries, which can be used instead of the original, bloated ones with any of the analyzed applications. Nibbler focuses on shared libraries as they have a series of advantages over static libraries in real-world deployments: (i) application binaries are smaller, (ii) the code of shared libraries is efficiently shared by applications, so it is not duplicated in physical memory, (iii) shared libraries can be developed in different programming languages than C/C++ (e.g., Go), (iv) they facilitate maintainability (e.g., updating and patching), (v) their load-time addresses can be individually randomized by ASLR, and (vi) shared libraries with (L)GPL license can be used by applications without distribution complications.

Previous attempts to debloat binaries [46] used bounded address tracking [34] to statically determine the set of used functions, which was prone to errors, requiring the manual whitelisting of certain functions to avoid program crashes. In antithesis, Nibbler over-approximates the function-call graph (FCG) of applications to conservatively include *all* code that could potentially be used (assuming no manual library loading occurs). So, even though Nibbler also predominantly relies on static analysis, it does not lead to application crashes nor require maintaining a whitelist.

As binary analysis is an undecidable problem in general [75], we focus on non-obfuscated compiler-generated code, and leverage symbol and relocation information—produced during compilation—to correctly disassemble binaries. We expect that software vendors will be willing to provide (anonymized) symbols and relocations for their libraries to facilitate debloating and retrofitting defenses. For instance, relocation information is already included in many modern libraries to support ASLR, and various operating system vendors offer symbol files [19, 43] for their most popular libraries. If such information is not available, disassembly may still be possible using advanced reverse-engineering tools [28, 48, 62, 70, 73].

With Nibbler, we overcome various challenges pertaining to FCG reconstruction of binaries. For example, certain compiler optimizations make transitions between functions implicit. The treatment of function pointers is another challenge, as failure to detect the usage of one could lead to incorrectly excluding used code. We propose a novel analysis for detecting *address-taken (AT) functions* (i.e., functions that have their address referenced as a constant) [53], which are not unused and iteratively eliminate them, while we include all others. Finally, a challenge of more technical nature is precisely mapping the policies applied by the system loader when resolving symbols, which includes things like special symbols resolved based on the actual configuration of the system (e.g., the CPU model). We found that this intricacy is not addressed in earlier studies [46, 53].

We developed a prototype of Nibbler for x86-64 Linux and tested it with real-world applications, including the GNU Coreutils, the Nginx web server, the MySQL database server, and the SPEC CINT2006 benchmark suite. Our evaluation shows that Nibbler reduces library code size and functions in scope, including the notoriously hard to analyze GNU `libc` (`glibc`), by up to 56% and 82%, respectively. While Nibbler does not focus on applications that manually load libraries with `dlopen()`, we also developed a profiling tool for collecting symbols loaded by applications at run time, similarly to training approaches employed by earlier studies [53]. We evaluate Nibbler with run-time profiling using the Chromium browser, which extensively loads libraries at run time. On average, we reduce code size and functions in scope by 25.98% and 34.95%, respectively.

We evaluate the security benefits of debloated code, by running the Nginx web server with thinned libraries under Shuffler [76], a continuous re-randomization system for binaries. We observe a throughput improvement of 20%, which increases the *deployability* of the defense. We also developed an analysis framework to determine the effect of debloated code on certain CFI techniques [42, 68, 81, 82], including *real-world* CFI solutions, like Microsoft's Control-Flow Guard [42] and LLVM's CFI enforcement [68]. For coarse, context-insensitive techniques [81, 82], we found that the number of gadgets that can be targeted by function returns is reduced by 75% on average. The number that can be targeted by indirect function calls is reduced by 49% on average, because our analysis detects and removes unused address-taken functions. While the number of gadgets remaining in the application is still significant, the analysis performed by Nibbler clearly improves the effectiveness of some CFI defenses. Finally, we look at whether debloating can reduce attack surface by removing vulnerabilities. Unlike what is suggested by previous work [53], we argue that this type of debloating cannot reduce attack surface, as, by design, it only removes code that is *never used* by applications.

Below, we summarize the contributions of this paper:

- We design and develop a practical system, Nibbler, which removes bloat from binary shared libraries without requiring source code and recompilation.
- We devise a novel method for detecting unused address-taken functions, which allows Nibbler to detect and eliminate, safely, more unused code.
- We evaluate the debloating capabilities of Nibbler with real-world binaries and the SPEC CINT2006 suite, and show that it removes 33%–56% of code from libraries, and 59%–82% of the functions in scope.
- We demonstrate the benefit of debloating to security techniques, like continuous code re-randomization, by integrating Nibbler with an existing system [76], where we observe a 20% run-time improvement for Nginx.
- We demonstrate the benefits to coarse-grained CFI [42, 81, 82] by analyzing the evaluated applications to find that, on average, Nibbler removes 75% of the available gadgets.
- We discuss the limitations of this type of debloating, which only eliminates unused code, in terms of attack surface reduction and vulnerability removal.

The rest of this paper is organized as follows. Sec. 2 provides background information and motivates our work by discussing how Nibbler improves existing defenses. We present the design of Nibbler and our methodology for thinning shared libraries in Sec. 3. In Sec. 4, we briefly discuss how we implemented Nibbler and some challenges we had to overcome. Sec. 5 presents the results from evaluating Nibbler, Sec. 6 discusses limitations of debloating, in general, and Sec. 7 summarizes related work. We conclude in Sec. 8.

2 BACKGROUND AND MOTIVATION

2.1 Software Exploitation Techniques

Attacks against software written in C and C++ are currently employing multiple vulnerabilities to overcome defenses like ASLR [52] and DEP [4]. They first reveal the layout of the targeted application, either by exploiting information leakage vulnerabilities [63] or using other guessing techniques [64] to bypass ASLR. Then, they exploit memory-safety bugs (e.g., use-after-free) to take control of a code pointer, hijack control flow, and, ultimately, perform code-reuse to achieve arbitrary code execution, despite DEP.

Such attacks employ techniques like *ROP* [60] and *return-to-*`libc` (`ret2libc`) [20]. The first reuses entire functions, while the latter chains arbitrary pieces of code terminating in indirect control-flow instructions, called gadgets. Other techniques, inspired by the above, include JOP [13], COP [25], COOP [58], CFB [11], and Control Jujutsu [22]. Code bloat is a boon for attackers, as more code implies more potential gadgets to pick from, making development of payloads easier and faster, and facilitating automation [12].

2.2 Continuous Code Re-randomization

Continuous code re-randomization techniques [6, 14, 24, 74, 76] mitigate exploits by continuously moving code at run time with high frequency. This introduces a real-time deadline for attackers, who only have milliseconds between exposing the layout of the process and mounting a code-reuse attack. Essentially, they aim to invalidate the leaked information before they can be used by exploits. A high re-randomization frequency can be pivotal against browser exploits [63] that utilize malicious JavaScript (JS) to execute the whole locally, using the leaked information almost immediately. Run-time overhead is also crucial, as lightweight defenses are a lot more likely to be adopted than heavyweight ones. By removing unneeded code, there is less code that needs to be shuffled at run time, so we can improve continuous re-randomization solutions both in terms of frequency and overhead.

2.3 Control-flow Integrity Defenses

CFI is a technique proposed by Abadi et al. [1], which aims to enforce the control flow of the original program, forbidding arbitrary transitions. It aims to prevent the control-flow hijacking part of attacks, after code pointers are taken over. There have been multiple instantiations of CFI [50, 51, 71, 81, 82] with different granularity, overhead, and requirements.

Applying CFI on binaries and achieving low overhead has been particularly problematic. The most deployable solutions enforce a coarse version of CFI [50, 81, 82], without employing context in their enforcement of the control-flow graph (CFG). These defenses only allow functions to return to code segments that follow a function invocation (i.e., `CALL`-preceded gadgets) and indirect function calls to address-taken and library-exported functions (which can be called through a pointer). Nibbler enhances these CFI techniques in two ways by removing unnecessary code: (i) there are less `CALL`-preceded gadgets for returns to target, and (ii) there are less AT functions that can be targeted by indirect calls.

3 DESIGN

Nibbler is designed primarily for the (Linux) ELF file format [69]. We believe our techniques are applicable to other settings, such as Microsoft Windows and the PE file format [77], but leave this for future work. We focus on the x86-64 architecture, but Nibbler's requirements (disassembly, library symbols, etc.) are also available on other contemporary architectures, such as x86 and ARM.

3.1 Overview

Figure 1 depicts a high-level overview of Nibbler. Given a set of binary applications, Nibbler processes the shared libraries they use, disassembles them, and statically analyzes them to reconstruct the FCG of each library. Then, the functions required by applications and the already-extracted library FCGs are composed to determine functions that are never called (i.e., unreachable code), by any of the applications of the set. At this point, Nibbler considers all functions that may be called through a function pointer as used.

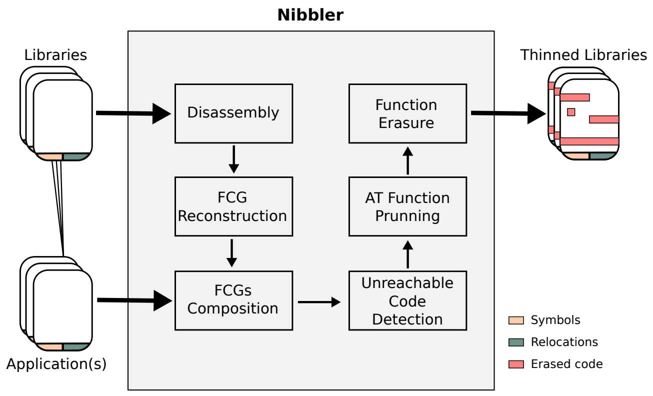


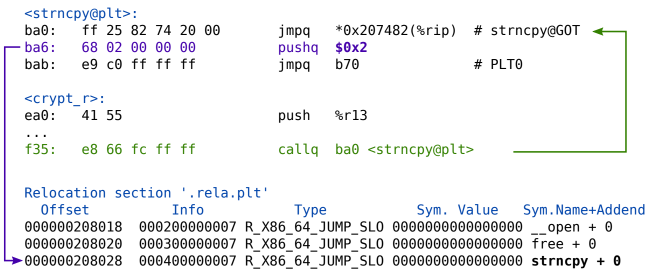
Figure 1: Approach overview.

The analysis over-approximates the set of functions that could (potentially) be used to *eliminate* the possibility of error, assuming no manually-loaded libraries. We then perform an iterative analysis that detects functions pointers that can never be used and also remove them. Finally, Nibbler produces a set of new (thinned) libraries that can be used with the input binaries, where the extra code has been erased by overwriting it with a trapping instruction [17]. Each library only needs to be analyzed once and the results of the analysis are cached in a database.

3.2 Disassembly

Obtaining the complete disassembly of an arbitrary binary program is an undecidable problem [75]. However, modern compiler-generated binaries can be linearly disassembled (verified on GCC and Clang [5]), especially since we use symbol information to accurately identify function boundaries. For functions that are exported by libraries, we also record the following additional metadata: (i) the type of the function's symbol (`FUNC` or `IFUNC`), (ii) its binding, which dictates its scope (`GLOBAL` or `WEAK` for externally-visible symbols), and (iii) its version (e.g., `memcpy@@GLIBC_2.14`). Note that this information is *always available* in shared libraries for exported symbols, by definition.

GNU `IFUNC`-type symbols. These allow for the run-time selection of a target, decided by a gateway function commonly referred to as a resolver function. Typically, this mechanism is used to select between different function implementations that use processor features such as `SSE`, `AVX`, etc which are more efficient but not always available. `IFUNC` symbols point to resolver functions which themselves contain references to multiple targets. To avoid specializing an application to a specific environment, Nibbler preserves all possible `IFUNC` targets if the `IFUNC` symbol is called.



```
<strncpy@plt>:
ba0:   ff 25 82 74 20 00       jmpq   *0x207482(%rip)  # strncpy@GOT
ba6:   68 02 00 00 00          pushq  $0x2
bab:   e9 c0 ff ff ff          jmpq   b70              # PLT0

<crypt_r>:
ea0:   41 55                   push   %r13
...
f35:   e8 66 fc ff ff          callq  ba0 <strncpy@plt>


Relocation section '.rela.plt'
  Offset          Info           Type           Sym. Value    Sym.Name+Addend
000000208018  000200000007 R_X86_64_JUMP_SLO 0000000000000000 __open + 0
000000208020  000300000007 R_X86_64_JUMP_SLO 0000000000000000 free + 0
000000208028  000400000007 R_X86_64_JUMP_SLO 0000000000000000 strncpy + 0
```

Figure 2: `call` to function in shared library in `crypt_r()`, `libcrypt-2.19.so`.

3.3 Library FCG Reconstruction

We use the disassembly and symbol information to statically reconstruct the FCG of each library. The goal is to resolve the targets of function calls. On x86, compilers use two classes of instructions to perform this task, specifically `CALL` and `JMP` instructions; we handle both the same way. Function calls are further classified into three categories: (i) calls targeting library-local functions, (ii) calls targeting functions in other shared libraries, and (iii) indirect calls that use pointers. We ignore cases where the targeted function is the same function (recursion)—i.e., multiple edges between functions are collapsed to a single one.

3.3.1 Calls to Local Functions. To resolve these calls, we go over the disassembled code and search for `CALL` and `JMP` instructions with an immediate value (i.e., a constant) as operand/argument. During execution, the CPU adds the value of the immediate to the address of the next instruction to calculate the address to transfer control to (PC-relative addressing). When the target address matches the starting address of a function, we add an edge between these two functions (caller-callee) in the FCG.

3.3.2 Calls to Functions in Shared Libraries. These calls are (usually) resolved lazily at run time, when a function is first invoked. The mechanism employed on Linux and other Unix-like systems uses two specially crafted sections called the Procedure Linkage Table (PLT) and the Global Offset Table (GOT) [38]. Without going into too many details, calls to external functions are performed through the PLT. For example, in Figure 2, the call at address `0x0f35` targets an entry in the PLT that corresponds to the `strnpy()` function. PLT entries are also code, which on the first invocation call the dynamic linker/loader to resolve the desired symbol. Subsequent calls direct control into the resolved function.

The dynamic linker/loader (`ld.so(8)`) resolves external functions by name (e.g., `strnpy` in the example above). The name of the targeted function is indicated by the second instruction of a PLT entry, the one at address `0x0ba6` in Figure 2. This instruction pushes an offset in another table onto the stack, `0x02` in our example. That table essentially contains a list of references to symbol names (i.e., the string "`strncpy`" in our example). We have analyzed the steps taken by the loader and mirrored the steps in Nibbler to link such functions calls with symbol names. Resolution of these symbols occurs during the FCG composition step.

3.3.3 Calls using Pointers. These function invocations are performed using a pointer, which can be dynamically computed at run time. In binary form, they correspond to `CALL` or `JMP` instructions (with register or memory as an operand).

Unfortunately, statically resolving the set of potential targets of such calls is a hard problem [34]. Instead of attempting to do so and risk introducing errors, like CodeFreeze [46], Nibbler is designed to identify *all* the functions that could be *potentially* called through a pointer, and assumes that *any* of them may indeed be invoked. We compute the set of indirectly-invoked functions by analyzing the disassembly and relocation information to identify where the address of a function is *taken*, and a pointer is generated. A function used as a callback, for example, will have its address taken at least once which will add the function to the list of (all) indirect targets. This over-approximation circumvents the limitations of static analysis to accurately track pointers in memory. Note that in Sec. 3.5 we present a method for further trimming the set of indirectly-invoked functions, thereby producing more tight FCGs. We employ two strategies to detect AT functions:

1) Function pointers in disassembled code. When a program assigns a function pointer to a variable, instructions are generated to obtain its address and store it in memory or a register. The address of the function is either directly used as an immediate (mostly on 32-bit systems) or expressed using PC-relative addressing (x86-64). Nibbler scans the disassembled code for move (`MOV`) and load-effective address (`LEA`) instructions, looking for operands that match function addresses. Since the set of function addresses is known, this heuristic works very well, especially on x86-64 where PC-relative addressing is used extensively. However, an optimizing compiler could perform arithmetic to compute target addresses, and detecting such cases would require data-flow (e.g. value set) analysis. We do look at operands for `ADD`/`SUB` arithmetic instructions, because they may occasionally contain function references. But in our experiments we never saw an address computation split between multiple instructions (and hence data-flow analysis was not required).

2) Function pointers via relocation information. Relocations are usually created to facilitate relocating code and data at load-time (e.g., for enabling ASLR). Each entry corresponds to a particular offset in the binary, typically a pointer to a function or global data object. Relocation entries describe how that address should be adjusted when the target entity is relocated. Modern systems support multiple relocation types [30, Chapter 4.4], which define different ways of calculating the "fix" to the targeted offset. Nibbler parses them to identify any function pointers that were not discovered in the previous step, mainly, in data sections. Relocations of type `R_X86_64_IRELATIVE` require more complex handling, as they specify that the targeted address should be patched based on the *return value* of a resolver function. Nibbler handles them by scanning the body of resolver functions, adding any code pointers referenced there in the list of targets that could be potentially returned at run time. Essentially, it connects all of the implementations to the FCG to retain support for indirectly-invoked functions.

3.4 Unreachable Code Detection

We analyze the application binaries and compose the FCGs of used libraries to conservatively estimate the library functions that are needed by the applications. Assuming the set of used libraries is known, our algorithm is complete by design and ensures that unused functions can be *safely* removed. Applications that manually load additional libraries at runtime are discussed in Sec. 3.7.

3.4.1 Identification of Required Symbols. At this stage, Nibbler calculates which library symbols are required by the input applications. It does so by processing them to determine the library symbols they refer to. These are obtained by scanning the PLT sections of the binaries to obtain the required symbol names, similarly to the process described in Sec. 3.3.2. Initialization and cleanup routines defined in libraries are also added in the set of required symbols, since they are called by the dynamic linker/loader during library loading or unloading. Such functions are defined (as arrays of function pointers) in special sections in binaries. Some of these are: `.preinit_array`, `.init`, `.init_array`, `.ctors`, `.fini`, `.fini_array`, and `.dtors`. Nibbler essentially manages to capture the non-trivial startup procedure of x86 ELF-compliant systems [29]. At this stage, we still consider that all AT functions are required.

3.4.2 Composition of Function-Call Graphs. We compose the FCGs of libraries, adding edges between callers and callees, the same way the dynamic linker/loader does when a program executes. To connect the various graphs, we start by resolving each graph's calls to external functions. In the simplest case, this requires looking for a function symbol with the same name as the one referenced by a call site. At load-time this process is performed by `ld.so`, which enforces various rules that Nibbler replicates faithfully.

In particular, we enforce the following: (a) `LOCAL` symbols are ignored; (b) `GLOBAL` symbols have precedence over `WEAK` ones; and (c) when the particular version of a symbol requested is not found, we use the one defined as default. Default symbols are denoted by the two '`@@`' characters (e.g., `putwchar@@GLIBC_2.2.5`). Note that there can only be one default version [21]. When resolving symbols, it is possible that multiple symbols with the same name exist. There may be multiple local symbols with the same name, or a local symbol with the same name as a global one. For inter-library symbol resolution, all symbols except weak or global ones are ignored. The dynamic linker/loader resolves `WEAK` and `GLOBAL` symbol references according to library load order. We did not implement all intricacies of library load order—a complex process, e.g., dependencies can be recursive—but rather create links to all weak/global functions of the same name in our graph. This approach produces a super-graph that may include more code than necessary, but it is guaranteed to include all functions that could be possibly used.

3.4.3 Collection of Unused Functions. At this point, we can use the composed FCG and the required symbols extracted in the previous steps to create an over-approximated set of used functions. The graph actually consists of multiple, potentially disconnected, sub-graphs; we focus on the ones that include required symbols, such as library functions invoked by one of the applications or AT functions. These nodes act as starting points that allow us to designate their whole sub-graph as *used*. All other functions are unreachable code that we can remove from libraries.

Listing 1: AT functions within function defined in `crypto/bn/bn_exp.c`:of `openssl-1.1.0j`.

```
int BN_mod_exp_mont_consttime(BIGNUM *rr, const BIGNUM *a, ...)
{
    ...
    static const bn_mul_mont_f mul_funcs[4] = {
                bn_mul_mont_t4_8, bn_mul_mont_t4_16,
                bn_mul_mont_t4_24, bn_mul_mont_t4_32
        };
    ...
}
```

3.5 AT Function Pruning

To reduce the number of AT functions included in the FCG, we introduce an analysis that takes into account the *location* a code pointer was found. Initially, we separate pointers found in data (e.g., `.rodata`) and code (e.g., `.text`) segments. For the latter, we iterate every function that has been classified as unused by our algorithm, and check if an AT function's address is *only* taken within unused functions. If this condition is true, we mark the respective AT function as unused. Note that this may result in additional (function) sub-graphs to be deemed unused, and so we iteratively perform this process until no additional functions can be classified as unused. For example, consider the function shown in Listing 1, which defines and uses the static array of function pointers `bn_mul_mont_f[]`. If Nibbler detects that the function is unused, the pointers contained in the array, which will actually be stored in the data segment, can also be ignored.

To eliminate AT functions in data segments (e.g., in `.(ro)data`), we proceed as follows. First, we leverage symbol information to identify the bounds (`[OBJ_BEGIN – OBJ_END]`) of global data objects (i.e., symbols of type `OBJECT/GLOBAL`). Next, we check for relocation entries that: (a) correspond to AT functions; and (b) fall within the bounds of any global object. Our approach basically identifies *statically-initialized* arrays of function pointers or data structures that contain function pointers. Lastly, like before, we iterate every function that has been classified as unused by our algorithm, and check if `OBJ_BEGIN` is taken *only* within unused functions. Again, if this condition is true, we mark the AT functions that correspond to the object beginning at `OBJ_BEGIN` as unused, and we iteratively perform this process until no additional function sub-graphs can be classified as unused. Note that the above process is complete; it only excludes AT functions that are used by unreachable code.

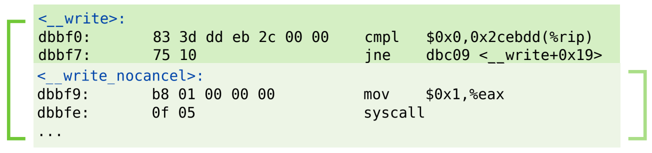
3.6 Function Erasure

Nibbler erases functions that are not part of the application FCG by overwriting them with a single byte instruction, namely `int3`, which causes a trap and interrupts execution.[1] Attempts to use the removed code will lead to termination of the running process [17].

3.7 Application-loaded Libraries

Application-loaded libraries are libraries which are explicitly loaded through calls to `dlopen()`. Pointers to functions in such a library

[1]Both `int3` and `hlt` [76] are used in related work for "erasing" code. We chose `int3` as it raises a `SIGTRAP`, rather than a `SIGSEGV`, signal; `SIGSEGV` has many potential causes.



```
<__write>:
dbbf0:       83 3d dd eb 2c 00 00    cmpl   $0x0,0x2cebdd(%rip)
dbbf7:       75 10                   jne    dbc09 <__write+0x19>
<__write_nocancel>:
dbbf9:       b8 01 00 00 00          mov    $0x1,%eax
dbbfe:       0f 05                   syscall
...
```

Figure 3: A fall-through function reuses another's code.

can be dynamically retrieved using `dlsym()`. It is very hard to statically determine the set of libraries and functions that are invoked in this way, and, as such, all the libraries and symbols that are required by the program. Nibbler cannot guarantee the safety of debloating an application which calls `dlopen`. However, in our experience, profiling the application with common workloads reveals the additional dependencies. Previous approaches concur with this [53].

As an alternative, we can be conservative and avoid debloating any applications that manually load libraries, or leverage software packaging semantics to include additional code in scope (e.g., all the `.so` files included in a particular package). To determine how frequently application-loaded libraries are used, we examined the source code of 25,526 Debian (v9) packages, using `apt-src`. After processing, we determined that 9,792 contain at least one file of C or C++ code, and 1,351 of them (13.8% of the C/C++ packages) call `dlopen()`/`dlsym()` and hence may perform manual library loading.

3.8 Challenges

Function Aliases. One function may be encompassed by many symbols of the same size (but often different type or scope), in effect creating aliases for the same function. We treat all aliases as a single entity and a reference to any name is sufficient to prevent the function from being removed.

Fall-through Functions. Some symbols share code (or overlap) with other symbols. This requires that we employ caution when erasing an unused function, as its bytes may be shared by another symbol. A frequent case in GNU `libc` is fall-through functions, shown in Figure 3, where one function performs a few checks and then drops into the beginning of another function. We carefully identify each function which does not terminate in a control-flow transfer, forming a reference to the following function and preventing its removal if the previous function is used.

Noreturn Functions. Functions that the compiler knows will terminate can be marked with the `__noreturn__` GCC attribute, which will be recursively propagated if possible. When generating a call to such a function, like `__fortify_fail`, the compiler may simply stop generating code afterwards (which would be unreachable). Luckily, we always observed the compiler generating a `nop` following the `CALL` in this case, which allows us to avoid (incorrectly) classifying this case as a fall-through function.

GNU `libc` Sub-libraries. While we view `libc` as a single library that is used by C/C++ programs, its most popular version, GNU `libc` (`glibc`), actually consists of sub-libraries that implement back-ends to common interfaces. For example, different name services, like the Network Information Service (NIS) and the Domain Name Service (DNS), are implemented in shared libraries (`libnss_dns.so` and `libnss_nis.so` in this case). These are loaded by `libc` at run time, when a particular API is accessed. To avoid erasing any `libc` functions that may be used by functions in those libraries, we include all their symbol requirements in our analysis.

Zero-sized Symbols. Certain internal functions, like `_start` (GCC-inserted), have a symbol of size zero. These functions have known semantics (e.g., `_start` calls `__libc_start_main`), and so we add them to the FCG for completeness, marking them as non-removable.

4 NIBBLER IMPLEMENTATION

We developed a prototype of Nibbler using Python on Linux. In this section, we provide some information on the implementation of Nibbler's core components, and discuss certain noteworthy challenges that we had to overcome.

4.1 Components

The disassembly and static analysis components of Nibbler were written in Python (≈7 KLOC). We used the `objdump` Linux utility for linear disassembly and symbol information, and the `pyelftools` Python package to access ELF files. The algorithms described in Sec. 3, regarding FCG reconstruction and AT function elimination, were developed from scratch.

Obtaining Symbols. Since by default all binaries installed on Linux are stripped of symbols, we developed a tool for fetching the debug packages corresponding to the binaries and libraries we want to process. It uses the `build-id` of an installed library to find a corresponding match in the debug repositories, and automatically download it using Debian's package management tools (i.e., `apt(8)`).

4.2 Application-loaded Libraries

We implemented a tool to collect the libraries and symbols which are manually loaded by programs with `dlopen()`. We exploited the linker's auditing interface in Linux [39] to introduce "hooks" that are called before any operation is performed, such as searching and opening a library, resolving a symbol, etc. Specifically, we developed a shared library that sets up hooks to receive all pertinent information from the loader, filters events unrelated to dynamic loading, and logs the rest. The tool can be easily activated by setting the `LD_AUDIT` environment variable before running an application.

5 EVALUATION

We evaluate Nibbler, using the following application suites on Debian GNU/Linux x86-64 (v9, Stretch): Coreutils (v8.26), Nginx (v1.10.3), MySQL (v5.5.8), and SPEC CINT2006; we also used the stock GNU `libc` (v2.24). We applied Nibbler on Nginx and MySQL individually, as well as on all the binaries in Coreutils and SPEC CINT2006[2], treating them as *sets* of applications. In addition, we applied Nibbler on all the above applications, considering them as a single set totaling 117 binaries. The end result is *five* sets of thinned libraries. Note that for every application set, we generate *one* set of libraries to satisfy the requirements of *all* the included binaries.

We verified if Nibbler correctly removes only unused code by running the following tests; they all completed successfully.

[2]Excluding `perlbench` that did not compile successfully.

- **Coreutils**: We run the built-in high-coverage test suite, invoked through '`make check`'.
- **Nginx**: We used Siege [33] to perform requests on a running server, as well as Nginx's official test suite.
- **MySQL**: We used the officially-provided test suite, invoked through '`mysql-test-run.pl`'.
- **SPEC**: We used the '`ref`' workload.

5.1 Debloating

5.1.1 Library-code Reduction with Nibbler. Table 1 summarizes the code reduction achieved by Nibbler on the application sets that do not manually load libraries. It performs the best with Nginx, where 55.95% of library code, in terms of bytes, is removed. On the other hand, for Coreutils, which include a large number of diverse utilities, we are able to eliminate 32.85% of library code. If we combine all 117 applications (last row), we achieve a reduction of 47.80%. If we instead focus on removed functions, reduction is between 58.81% – 81.57%, as many smaller functions are removed.

Bloat is not equally distributed in libraries. In the worst case (`libcrypto`), we found that 93.82% of its functions are not used by any of the four application sets. In the best case (`libpcre`), we removed 13.29% of its functions, however, this was also one of the smallest libraries in our set. Table 2 highlights the per-library code reduction achieved by Nibbler in libraries used by all four application sets. We think that these results demonstrate the heavy bloat in certain libraries, making their thinned versions great candidates even for system-wide replacement.

Comparison with Piece-wise [53]. While direct comparison is not possible, because Quach et al. focus on debloating individual programs, we highlight our differences using Coreutils and SPEC to provide some perspective to the reader. The mean reduction, in the number of functions, Piece-wise achieves, with respect to these two program sets, is 79% and 85%, respectively; Nibbler achieves 58.81% and 73.37%. This difference is primarily due to the fact that we debloat libraries for sets of applications, instead of individual binaries. In addition, the lack of semantics in binary code prevents us from effectively using analyses to eliminate more AT functions, while we preserve multiple versions of a used symbol in thinned libraries (Piece-wise keeps only one).

5.1.2 Reduction with Application-loaded Libraries. Chromium (web browser; v57) is a large, complex application that performs manual library loading. To debloat it, we profile it by: i) visiting the top sites in Alexa's "Top 500 Global Sites" list [2], exercising a broad range of functionality, such as video playback, animations, etc., and ii) using Chromium's comprehensive test suite [67], which includes a plethora of tests related to layout and rendering, conformance to certain web standards, UI events, and Chrome-specific APIs. During profiling the browser loaded 63 additional libraries. After including the used symbols in Nibbler, we included an additional 3241 functions or approximately 1MB of code. Table 3 lists the 10 (out of 84) most thinned libraries and total code reduction in Chromium. Finally, we browsed the top-10 sites in Alexa's list, which did not result in new libraries being loaded. Our experiment confirms the results of previous work [53], which showed that profiling can be sufficient to debloat manually-loaded libraries in certain applications.

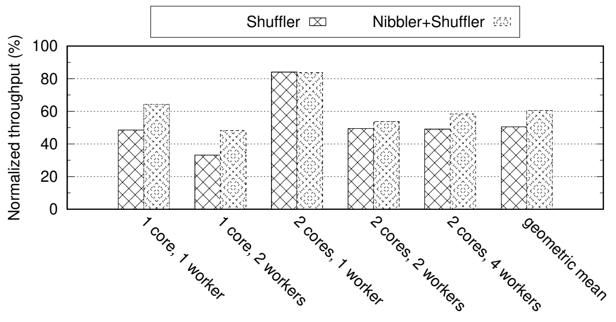


Figure 4: Nginx throughput for vanilla Shuffler, and Nibbler+Shuffler, over an undefended baseline.

5.2 Benefits on Defenses

5.2.1 Continuous Code Re-randomization. By identifying and removing unused code, Nibbler can reduce the overhead of certain security techniques, thereby easing their adoption and improving software security. Shuffler [76] is a system realizing such a technique: continuous re-randomization of a target program and all its libraries, including its own code. It does so asynchronously, from a background thread, preparing a new copy of the code every 20 ms (in case of Nginx), and then signaling all other threads to migrate to this new copy. Because of this asynchronous design, all functions must be re-randomized, during each shuffle period, as the system cannot determine in advance what will be required. Nibbler's library thinning can combine excellently with Shuffler's defense. We fused Nibbler with Shuffler, on Nginx 1.4.6, trimming functions that would never be used during execution, reducing the amount of work that Shuffler must perform. Overall, we nibbled ≈1.6K functions (out of ≈6.2K) or 26%.

In our experiment, we used 4 Nginx worker processes, pinned to 2 CPU cores; Shuffler threads (one per worker) were also pinned to the same cores. Shuffler's asynchronous overhead will take CPU time away from the target program, reflecting in a throughput drop. We ran 32 client threads (using the benchmark tool Siege [33]) pinned to 4 other cores on the same system, which was sufficient to saturate the server. This experiment is a smaller scale version of the Nginx experiment included in the original paper [76].

Results are shown in Figure 4. Nibbler+Shuffler performance improves substantially when there are more Shuffler workers, and hence more CPU time is being spent on asynchronously copying code. In the 2-cores, 1-worker case, one core runs the Nginx worker and one executes the Shuffler thread, so Nibbler has little impact. However, if we assume a carefully provisioned system with few resources to spare, *Nibbler can improve Shuffler's performance significantly.* Overall, the geometric mean of throughput improved from 50.51% (Shuffler) to 60.54%, a relative increase of 19.87%.

This makes sense since we trimmed 26% of the code; due to Shuffler's design, we expect a linear increase in performance as the amount of code decreases. Additionally, we expect these results to scale to larger experiments, since Shuffler spends very little time on coordination (0.3% of overall runtime [76]). If the server is multiprocess, every process gets its own independent Shuffler thread, and Nibbler still reduces the overhead linearly. If the server is

Table 1: The effect of Nibbler on various application sets. The table summarizes library-code reduction in terms of code bytes and functions removed. The *Vanilla* column corresponds to the original libraries.

Application Set			Code Reduction (Lib. Set)					
App(s)	Set Size		Functions			Code (KB)		
	# of Bin.	# of Lib.	Vanilla	Removed	Reduction (%)	Vanilla	Removed	Reduction (%)
a) **Coreutils**	104	11	4754	2796	58.81%	2164.61	711.01	32.85%
b) **SPEC**	11	5	7808	5729	73.37%	2431.15	1104.93	45.44%
c) **Nginx**	1	7	8599	7015	81.57%	2917.77	1632.71	55.95%
d) **MySQL**	1	8	7979	5524	69.23%	2522.66	1010.17	40.04%
a) + b) + c) + d)	117	16	14438	10622	73.57%	4621.53	2208.93	47.80%

Table 2: Code removed from common libraries in our application set (Coreutils, Nginx, MySQL, and SPEC CINT2006).

Library	Unused Code			
	Functions		Bytes	
librt	52	(72.22%)	8.88 KB	(71.64%)
libattr	17	(53.12%)	3.35 KB	(38.17%)
libgcc	113	(66.86%)	36.02KB	(56.10%)
libdl	8	(33.33%)	0.55 KB	(26.70%)
libcrypto	4770	(93.82%)	1043.96 KB	(83.57%)
libz	83	(40.84%)	40.84 KB	(60.91%)
libpthread	139	(46.80%)	19.42 KB	(36.89%)
libpcre	21	(13.29%)	33.26 KB	(10.09%)
libc	1539	(53.27%)	297.90 KB	(24.93%)
libm	426	(66.56%)	172.16 KB	(39.76%)
libstdc++	2719	(71.31%)	309.14 KB	(45.04%)
libgmp	438	(66.46%)	176.60 KB	(46.43%)
libselinux	225	(63.20%)	50.51 KB	(53.13%)
libacl	42	(60.00%)	11.75 KB	(69.99%)
libcap	21	(75.00%)	3.38 KB	(49.97%)
libcrypt	9	(23.08%)	0.91 KB	(4.29%)

multithreaded, Shuffler's overhead decreases as more cores become available, and Nibbler still reduces the overhead proportionally.

5.2.2 Control-flow Integrity. Nibbler improves low-overhead, coarse CFI schemes in two ways. First, it reduces the number of (CALL-preceded) ROP gadgets that are accessible to attackers. To quantify this gain, we built a gadget analysis framework (details in App. A), atop the Capstone disassembler [54] to calculate the reduction of CFI-resistant [78] gadgets in thinned libraries.

Table 4 reports the results of our analysis. The *Suite* column corresponds to the different applications used, along with their thinned libraries; the numbers in parentheses indicate code reduction. *Total* reports the overall reduction of the gadgets (thinned vs. vanilla) in each library, whereas the rest of the columns (*Stack – NOP*) present the reduction of certain gadget classes. For the different gadget types, we used the semantic definitions of Snow et al. [63], with additional (sub)categories for precision. (Gadget reduction when CFI is not in place can be found in App. B.)

Table 3: 10 most debloated libraries in Chromium.

Library	Unused Code			
	Functions		Bytes	
libgtk	4886	(49.06%)	994.70 KB	(41.03%)
libxml2	1260	(48.26%)	423.87 KB	(46.70%)
libc	1606	(46.93%)	316.35 KB	(26.25%)
libgio	1749	(38.65%)	289.00 KB	(35.13%)
libgnutls	826	(36.26%)	212.32 KB	(27.76%)
libnss3	2763	(70.56%)	172.14 KB	(18.05%)
libbglib	840	(40.64%)	154.70 KB	(32.37%)
libasound	1053	(36.15%)	152.99 KB	(27.61%)
libm	375	(61.68%)	137.77 KB	(32.00%)
libstdc++	842	(28.45%)	115.45 KB	(22.47%)
libX11	433	(22.29%)	111.08 KB	(19.95%)
Total	20946	(34.95%)	4198.22 KB	(25.98%)

Second, Nibbler reduces the number of functions that can be targeted by indirect functions calls, by eliminating function pointers that are never used in applications (Sec. 3.5). Specifically, our analysis eliminates 45.19%, 57.86%, 57.75%, and 36.60% of AT functions in the four tested applications suites: Coreutils, SPEC, Nginx, and MySQL; 49.08% when we combine all of them.

All in all, the reduction of the CFI-resistant gadgets is analogous to the achieved code reduction, but, on average, the gadget reduction rate(s) are higher, suggesting that *Nibbler can increase considerably the precision of backward-edge CFI schemes* [81, 82]. More importantly, Nibbler can eliminate certain gadget classes in various libraries (100% reduction), like Load Reg. (load a register from another register or the stack), Memory (memory read/write), Arithmetic+Mem. (arithmetic computations with memory operands), Logic (logic computations), and Branch (indirect JMP/CALL), even on applications that experience moderate code reduction rates, such as Coreutils, as shown in Table 4 (underlined entries). As expected, when CFI is not present, Nibbler can achieve compelte gadget elimination in less cases (see Table 6 in App. B).

5.3 Performance Overhead

Memory. Nibbler can be applied on an entire system or on smaller sets of applications. In the latter case, if the original version of a

Table 4: CFI-resistant gadget reduction results. The percentages correspond to removed gadgets (thinned vs. vanilla). Entries marked with 'N/A' indicate absence of certain gadget classes in vanilla.

Suite		Total	Gadget Type													
			Stack		Load Reg.		Memory		Arithmetic		Logic		Branch		Syscall	NOP
			Pivot	Lift	Reg.	Stack	Load	Store	Reg.	Mem.	Reg.	Mem.	jmp	call		
SPEC																
libpthread	(92.79%)	97.0%	N/A	N/A	95.7%	100.0%	100.0%	86.5%	97.5%	100.0%	100.0%	100.0%	N/A	N/A	88.4%	97.5%
libm	(39.76%)	52.3%	N/A	N/A	N/A	18.5%	0.0%	0.0%	50.7%	0.0%	100.0%	N/A	N/A	N/A	N/A	57.7%
libc	(42.01%)	75.8%	91.7%	0.0%	75.6%	78.0%	72.6%	66.2%	75.9%	74.7%	62.6%	76.9%	25.0%	62.1%	73.3%	79.7%
libgcc	(72.35%)	45.7%	0.0%	N/A	39.8%	60.3%	21.1%	11.1%	69.8%	N/A	40.0%	N/A	N/A	N/A	N/A	59.4%
libstdc++	(48.88%)	80.1%	0.0%	50.0%	48.3%	79.5%	76.4%	69.7%	72.1%	77.8%	44.8%	100.0%	100.0%	35.7%	N/A	78.6%
Coreutils																
libpthread	(39.94%)	56.8%	N/A	N/A	61.1%	57.7%	62.5%	56.8%	42.5%	100.0%	55.8%	0.0%	N/A	N/A	55.8%	56.2%
libdl	(75.41%)	97.8%	N/A	N/A	100.0%	100.0%	100.0%	100.0%	93.8%	N/A	100.0%	N/A	N/A	100.0%	N/A	96.8%
libselinux	(53.13%)	70.3%	N/A	N/A	80.7%	64.0%	90.0%	66.7%	76.1%	N/A	78.4%	N/A	N/A	94.4%	N/A	67.2%
libacl	(69.99%)	58.8%	N/A	N/A	63.9%	71.7%	N/A	60.0%	36.8%	100.0%	57.9%	N/A	N/A	N/A	N/A	61.1%
librt	(71.64%)	69.4%	N/A	N/A	64.8%	76.3%	60.0%	50.0%	65.4%	N/A	62.5%	0.0%	N/A	66.7%	72.7%	78.1%
libpcre	(10.15%)	17.1%	N/A	N/A	25.0%	16.9%	10.0%	30.0%	7.7%	N/A	33.3%	N/A	50.0%	0.0%	N/A	21.4%
libc	(28.28%)	59.8%	50.0%	0.0%	63.5%	56.7%	58.8%	55.5%	62.6%	63.7%	44.6%	38.5%	25.0%	57.6%	58.7%	65.3%
libattr	(38.17%)	63.4%	N/A	N/A	50.0%	100.0%	100.0%	100.0%	45.5%	100.0%	100.0%	N/A	N/A	N/A	N/A	38.5%
libgmp	(46.43%)	65.5%	N/A	N/A	86.3%	59.6%	79.7%	73.8%	71.4%	38.9%	72.4%	0.0%	100.0%	85.7%	N/A	70.9%
libgcc	(92.26%)	95.0%	80.0%	N/A	95.5%	100.0%	86.0%	88.9%	96.2%	N/A	100.0%	N/A	N/A	N/A	N/A	96.9%
libcap	(49.97%)	60.2%	N/A	N/A	40.9%	50.0%	N/A	87.5%	61.5%	N/A	40.0%	N/A	N/A	N/A	N/A	61.9%
MySQL																
libm	(44.25%)	52.8%	N/A	N/A	N/A	19.2%	0.0%	0.0%	52.2%	0.0%	100.0%	N/A	N/A	N/A	N/A	58.9%
libpthread	(34.33%)	49.4%	N/A	N/A	53.5%	54.2%	62.5%	43.2%	37.5%	100.0%	51.2%	0.0%	N/A	N/A	46.5%	46.2%
libz	(29.37%)	58.5%	N/A	N/A	53.1%	62.0%	37.0%	60.0%	52.9%	50.0%	72.7%	N/A	N/A	37.5%	N/A	56.2%
libc	(34.23%)	67.0%	91.7%	0.0%	67.0%	66.4%	64.3%	56.3%	67.6%	63.2%	54.7%	61.5%	25.0%	50.0%	60.0%	72.5%
libgcc	(56.10%)	42.3%	0.0%	N/A	39.8%	49.2%	19.3%	11.1%	66.0%	N/A	40.0%	N/A	N/A	N/A	N/A	56.2%
libstdc++	(45.04%)	77.5%	0.0%	0.0%	46.2%	76.8%	70.4%	60.5%	69.1%	58.3%	37.9%	100.0%	100.0%	33.3%	N/A	76.7%
libdl	(26.70%)	40.2%	N/A	N/A	42.9%	31.8%	0.0%	33.3%	50.0%	N/A	0.0%	N/A	N/A	50.0%	N/A	41.9%
libcrypt	(4.30%)	20.2%	N/A	N/A	25.0%	20.7%	N/A	0.0%	21.4%	N/A	N/A	N/A	N/A	N/A	N/A	23.1%
Nginx																
libpthread	(47.75%)	50.3%	N/A	N/A	51.2%	47.9%	62.5%	54.1%	43.8%	100.0%	41.9%	0.0%	N/A	N/A	46.5%	55.0%
libz	(60.91%)	77.2%	N/A	N/A	68.8%	87.0%	51.9%	64.0%	67.6%	50.0%	81.8%	N/A	N/A	50.0%	N/A	78.1%
libc	(40.78%)	73.8%	91.7%	0.0%	71.0%	74.0%	71.8%	70.2%	74.8%	70.9%	62.6%	69.2%	25.0%	62.1%	60.0%	79.2%
libdl	(26.70%)	40.2%	N/A	N/A	42.9%	31.8%	0.0%	33.3%	50.0%	N/A	0.0%	N/A	N/A	50.0%	N/A	41.9%
libpcre	(10.09%)	17.1%	N/A	N/A	25.0%	16.9%	10.0%	30.0%	7.7%	N/A	33.3%	N/A	50.0%	0.0%	N/A	21.4%
libcrypt	(6.42%)	20.2%	N/A	N/A	25.0%	20.7%	N/A	0.0%	21.4%	N/A	N/A	N/A	N/A	N/A	N/A	23.1%
libcrypto	(83.56%)	93.1%	N/A	0.0%	92.0%	93.5%	86.7%	91.0%	93.4%	100.0%	96.1%	97.1%	82.4%	72.5%	N/A	94.4%

Table 5: Memory overhead (KB) comparison.

Application Set	Nibbler Max. Total	Piece-wise [53] Estimate per execution
a) **Coreutils**	1900 KB	1024 KB
b) **SPEC**	1148 KB	580 KB
c) **Nginx**	1668 KB	1292 KB
d) **MySQL**	2108 KB	1248 KB
a) + b) + c) + d)	3256 KB	1816 KB

library is also in use, then two versions of the same library will be present in memory (i.e., vanilla and thinned library). Calculating exactly how much memory overhead the thinned library will impose is not straightforward, as the OS dynamically pages-in code pages used by applications. We can calculate, however, the additional memory required when *all* library code is used (worst case analysis). This corresponds to all code pages (of the thinned library) that have at least one byte of code that was not erased.

On the other hand, Piece-wise [53] keeps a single version of each library on disk, and removes code at load time. As a result, each memory page that has code erased—but not removed entirely—is no longer shared with any other executing application that uses the same library, due to copy-on-write (COW).

Consequently, the overhead increases as more applications execute concurrently. This includes multiple invocations of the same application, but not multiple processes resulting from a `fork()` system call. As a reference, there are approximately 39 distinct applications running in a fresh installation of Debian v9. Comparison of memory overheads are shown in Table 5. We estimate the per-invocation overhead of Piece-wise using the code-reduction numbers of Nibbler (more code removed likely means higher overhead for Piece-wise). Our approach incurs lower and more predictable memory overhead.

Load Time. Nibbler does not incur any load-time overhead. In contrast, previous work [53], which only removes code at load time, reported a 20x slowdown on average with the small programs in Coreutils (20ms on average, for a process usually taking under 2ms).

Analysis Time. Processing the 104 programs in Coreutils and their libraries took ≈2 hours with Nibbler. Timings were collected on a VM running Debian 9 (using VMware Workstation 12 Pro), hosted on top of Windows 10 on a mid-range workstation featuring a 3.5GHz AMD FX-6300 CPU and 16GB RAM. Our experiment indicated that processing time is primarily influenced by code size; a secondary factor is the number of relocation entries. While the speed of the analysis is not critical, Nibbler is a prototype, so we are confident there is room for improvement.

6 LIMITATIONS

Exploit Disruption. In App. D we evaluate Nibbler against *pre-compiled*, real-world core-reuse exploits. In all cases, Nibbler disrupts the exploits; however, attackers can modify them to use other gadgets and potentially restore their capabilities. Debloating, even when combined with CFI, cannot block all code-reuse attacks, but it does make libraries a less fertile ground for gadget harvesting. Note that compiler-based approaches [53] performing the same type of debloating have a similar limitation. Exploits are more likely to be prevented by combining Nibbler with a system like Shuffler [76].

Attack Surface Reduction. Previous work [53] suggested that debloating can reduce the attack surface by removing vulnerabilities contained in the erased library code. Nibbler performs a similar type of debloating on binary code. However, we did not reach the same conclusion. By design, both works remove code only when there is no viable execution path to that code for a given application. Consequently, any code removed is essentially unreachable code, so vulnerabilities contained within are not relevant because they can never be triggered by external input(s). We do agree, however, that such vulnerabilities can potentially be used by multistage exploits, where later stage exploit components (ab)use vulnerabilities in unused code to escape sandboxing or further elevate privileges [15].

7 RELATED WORK

7.1 Code Reduction

Recent work from Quach et al. [53] proposes a compiler-based framework for debloating applications when source code is available, while Nibbler targets binary-only software. Other differences with Nibbler include the following: (i) their approach is unable to work with (one of) the most commonly used libraries, `GNU libc` (`glibc`), which requires the GNU C compiler, while Nibbler is compiler-agnostic, (ii) their approach opts to debloat each application individually, which incurs significant memory overhead when applied to numerous applications, as it breaks the sharing of memory pages that include erased code per-application instantiation, (iii) they choose to debloat applications at load time, which incurs a slowdown of 20x, and, even though the overhead for launching one application is negligible in absolute terms, it compounds in applications that spawn others (e.g., shell scripts), and (iv) Nibbler goes beyond CFI by demonstrating one of the key benefits of debloating by integrating it with continuous code re-randomization.

CodeFreeze [46] aims to reduce the attack surface of Windows binaries by removing unused code in shared libraries (DLLs). It utilizes bounded address tracking [34] to resolve function pointers, which leads to over-restrictive CFGs. As a result, while it is more aggressive at removing code, it can erroneously remove needed functions (e.g., constructors) and it depends on whitelisting to avoid crashes. Instead, Nibbler's analysis is conservative and attempts to err on the safe side by over-approximating. Our evaluation shows that we can correctly trim libraries without the need for a whitelist.

Perses [65] and C-Reduce [55] are state-of-the-art program reduction tools that build upon the concept of (hierarchical) delta debugging [44, 80]. Specifically, by specifying a program to be minimized and an arbitrary property test function, both these tools return a minimized version of the input program that is also correct with respect to the given property. Chisel [27] further improves this approach, by leveraging reinforcement learning. In particular, via repeated trial and error, Chisel builds (and further rectifies) a model that determines the likelihood of a candidate (minimal) program to pass the property test.

In antithesis to tools like the above, Nibbler does not require any high-level specification regarding the functionality of the input program/library. Our thinned libraries are guaranteed to be correct under any given input to the set of applications that uses them. Kurmus et al. [36] focus on reducing the attack surface of the Linux kernel by removing unnecessary features. Unlike Nibbler, they develop a tool-assisted approach for identifying and removing unnecessary features during the kernel's configuration phase, hence, omitting code during compilation.

A series of works [31, 32, 72] have focused on reducing bloat in Java programs and the Java Virtual Machine (JVM). JRed [31] employs static analysis to extract the FCG of applications and identify, and remove, the bytecode that corresponds to unused classes and methods from the Java runtime. Similarly, Wagner et al. [72] propose "slimming" the JVM by removing code that does not execute frequently and dynamically fetching it from a server only when it is required. The goal is to reduce the amount of code that needs to be deployed in thin clients, such as embedded systems, by dynamically deploying what is required.

Jiang et al. [32], instead of targeting the JVM, aim to cut specific features that are not needed from Java programs. Starting from a small set of methods responsible for implementing a feature, they use static analysis and backwards slicing to identify and remove all the code corresponding to the feature. While these approaches also utilize static analysis, decompiling and reconstructing the FCG of Java programs is less challenging than that of binaries [26].

Landsborough et al. [37] also propose removing unused features from programs to reduce their attack surface. Their approach involves manually disabling features in binaries and a genetic algorithm that is applied in toy programs. Malecha et al. propose software winnowing [41], an approach that uses partial evaluation of function arguments during compilation to "specialize" code, eliminating some unused code in the process.

In the same vein, TRIMMER [61] specializes program code, and debloats applications, by leveraging user-defined configurations, while Shredder [45] further introduces constant propagation analyses to specialize system API functions. Lastly, Koo et al. [35] propose the concept of configuration-based software debloating: i.e., the removal of feature-specific code, which is exclusively used only when certain configuration directives are specified/enabled. These approaches are orthogonal to Nibbler, looking at software thinning from a different perspective, while most of them (with the exception of Shredder) require source code.

Other works approach debloating from a performance angle, focusing on reducing memory consumption [9, 49, 79]. Despite the similarity in name, slim binaries [23], proposed by Franz et al., refer to programs that are represented in an way that allows their translation to multiple architectures.

7.2 FCG Extraction

Sound and complete extraction of the FCG from binaries is an open problem. Murphy et al. [47] perform an empirical analysis of static call-graph extractors that operate on source code or at compile time. Their findings indicate that there is significant variance, based on the tool, and the potential for false negatives. The latter correspond to undiscovered but existing call edges, which would be problematic for our approach, as removal of used code can be catastrophic. As existing FCG extraction methods are insufficient, we developed our own method that is complete.

There are also various promising binary analysis and augmentation frameworks [3, 7, 8] that reconstruct the FCG of binaries. Even though these tools keep improving, errors are still possible per their authors, as well as other researchers [5]. As such, their analyses are not appropriate for Nibbler. Instead, the methods described in control-flow integrity works for binaries [81, 82] are more related to our approach. Unlike them though, we introduce a novel methodology for eliminating AT functions—thereby deleting extraneous CFG edges—and reconstruct a complete FCG that also includes directs calls within and across modules.

8 CONCLUSION

In this paper, we presented Nibbler, a system which demonstrates that debloating binary-only applications is possible and practical. Nibbler identifies unused code in shared libraries and erases it. We use a conservative FCG reconstruction algorithm to initially only remove functions without pointers to them, which we refine by introducing an optimization for eliminating functions with unused pointers. We evaluated the debloating capabilities of Nibbler with real-world binaries and the SPEC CINT2006 suite, where we eliminate 56% and 82% of functions and code, respectively, from used libraries. Nibbler is able to correctly analyze binary software, by only leveraging symbol and relocation information produced by existing compilers.

Nibbler, and debloating generally, improves security of software indirectly, by benefiting defenses. Continuous code re-randomization systems get a performance boost, which we demonstrated by integrating Nibbler with Shuffler to lower overhead by 20%. Lower overheads make such defenses more attractive for deployment on production systems, or can be used to provide stricter security guarantees (e.g., by raising re-randomization frequency) in critical systems. Control-flow integrity defenses also benefit, because we remove code involved in allowable control-flows. Our evaluation shows that Nibbler reduces the number of gadgets reachable through returns and indirect calls by 75% and 49% on average.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and Maverick Woo, our shepherd, for their valuable comments. This work was supported by the Office of Naval Research (ONR) under awards N00014-16-1-2261 and N00014-17-1-2788. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the US government or ONR.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-Flow Integrity. In *Proc. of ACM CCS*. 340–353.
[2] Alexa. 2018. The top 500 sites on the web. https://www.alexa.com/topsites.
[3] Kapil Anand, Matthew Smithson, Khaled Elwazeer, Aparna Kotha, Jim Gruen, Nathan Giles, and Rajeev Barua. 2013. A Compiler-level Intermediate Representation Based Binary Analysis and Rewriting System. In *Proc. of EuroSys*. 295–308.
[4] Starr Andersen and Vincent Abella. 2004. Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies, Data Execution Prevention. Microsoft TechNet Library. http://technet.microsoft.com/en-us/library/bb457155.aspx
[5] Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. 2016. An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries. In *Proc. of USENIX SEC*. 583–600.
[6] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. 2015. Timely Rerandomization for Mitigating Memory Disclosures. In *Proc. of ACM CCS*. 268–279.
[7] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. 2011. BAP: A Binary Analysis Platform. In *Proc. of CAV*. 463–469.
[8] David Brumley, JongHyup Lee, Edward J. Schwartz, and Maverick Woo. 2013. Native x86 Decompilation Using Semantics-Preserving Structural Analysis and Iterative Control-Flow Structuring. In *Proc. of USENIX SEC*. 353–368.
[9] Yingyi Bu, Vinayak Borkar, Guoqing Xu, and Michael J. Carey. 2013. A Bloat-aware Design for Big Data Applications. In *Proc. of ISMM*. 119–130.
[10] Amat Cama. 2014. Tool to generate ROP gadgets for ARM, AARCH64, x86, MIPS, PPC, RISCV, SH4 and SPARC. https://github.com/acama/xrop.
[11] Nicolas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. 2015. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *Proc. of USENIX SEC*. 161–176.
[12] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing Mayhem on Binary Code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*. 380–394.
[13] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. 2010. Return-Oriented Programming Without Returns. In *Proc. of ACM CCS*. 559–572.
[14] Xi Chen, Herbert Bos, and Cristiano Giuffrida. 2017. CodeArmor: Virtualizing the Code Space to Counter Disclosure Attacks. In *Proc. of IEEE EuroS&P*. 514–529.
[15] Chromium Blog . 2012. A Tale of Two Pwnies. https://blog.chromium.org/2012/05/tale-of-two-pwnies-part-1.html.
[16] Corelan. 2011. Corelan Repository for mona.py. https://github.com/corelan/mona.
[17] Stephen Crane, Per Larsen, Stefan Brunthaler, and Michael Franz. 2013. Booby Trapping Software. In *Proc. of NSPW*. 95–106.
[18] National Vulnerability Database. 2019. BlueKeep Vulnerability (CVE-2019-0708). NIST. https://nvd.nist.gov/vuln/detail/CVE-2019-0708
[19] Bruce Dawson. 2013. Symbols on Linux update: Fedora Fixes. https://randomascii.wordpress.com/2013/03/05/symbols-on-linux-update-fedora-fixes/
[20] Solar Designer. 1997. Getting around non-executable stack (and fix). BugTraq. https://seclists.org/bugtraq/1997/Aug/63
[21] Ulrich Drepper. [n.d.]. ELF Symbol Versioning. https://www.akkadia.org/drepper/symbol-versioning.
[22] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. 2015. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *Proc. of ACM CCS*. 901–913.
[23] Michael Franz and Thomas Kistler. 1997. Slim Binaries. *Commun. ACM* 40, 12 (Dec. 1997), 87–94.
[24] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. 2012. Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization. In *Proc. of USENIX SEC*. 475–490.
[25] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. 2014. Out of Control: Overcoming Control-Flow Integrity. In *Proc. of IEEE S&P*. 575–589.
[26] J. Hamilton and S. Danicic. 2009. An Evaluation of Current Java Bytecode Decompilers. In *Proc. of IEEE SCAM*. 129–136.
[27] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective Program Debloating via Reinforcement Learning. In *Proc. of ACM CCS*. 380–394.
[28] Hex-Rays. 2016. The IDA Pro Disassembler and Debugger. https://www.hex-rays.com/products/ida/
[29] Patrick Horgan. [n.d.]. Linux x86 Program Start Up. http://dbp-consulting.com/tutorials/debugging/linuxProgramStartup.html.
[30] Intel. 2013. System V Application Binary Interface. https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf.
[31] Y. Jiang, D. Wu, and P. Liu. 2016. JRed: Program Customization and Bloatware Mitigation Based on Static Analysis. In *Proc. of IEEE COMPSAC*. 12–21.

[32] Y. Jiang, C. Zhang, D. Wu, and P. Liu. 2016. Feature-Based Software Customization: Preliminary Analysis, Formalization, and Methods. In *Proc. of IEEE HASE*. 122–131.

[33] JoeDog. 2017. Siege – an http load tester and benchmarking utility. https://github.com/JoeDog/siege.

[34] Johannes Kinder and Helmut Veith. 2010. Precise Static Analysis of Untrusted Driver Binaries. In *Proc. of FMCAD*. 43–50.

[35] Hyungjoon Koo, Seyedhamed Ghavamnia, and Michalis Polychronakis. 2019. Configuration-Driven Software Debloating. In *Proceedings of the 12th European Workshop on Systems Security*. ACM, 9.

[36] 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. In *Proc. of NDSS*.

[37] Jason Landsborough, Stephen Harding, and Sunny Fugate. 2015. Removing the Kitchen Sink from Software. In *Proc. of ACM GECCO*. 833–838.

[38] John R. Levine. 1999. *Linkers and Loaders* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[39] Linux Programmer's Manual. 2018. rtld-audit – auditing API for the dynamic linker. http://man7.org/linux/man-pages/man7/rtld-audit.7.html

[40] LLVM Project. 2018. LLVM Link Time Optimization: Design and Implementation.

[41] Gregory Malecha, Ashish Gehani, and Natarajan Shankar. 2015. Automated Software Winnowing. In *Proc. of ACM SAC*. 1504–1511.

[42] Microsoft. 2015. Control Flow Guard. Windows Dev Center. https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065(v=vs.85).aspx

[43] Microsoft. 2017. Symbols and Symbol Files. Microsoft Developer Network. https://msdn.microsoft.com/en-us/library/ff558825.aspx

[44] Ghassan Misherghi and Zhendong Su. 2006. HDD: Hierarchical Delta Debugging. In *Proc. of ICSE*. 142–151.

[45] Shachee Mishra and Michalis Polychronakis. 2018. Shredder: Breaking Exploits through API Specialization. In *Proc. of ACSAC*. ACM, 1–16.

[46] Collin Mulliner and Matthias Neugschwandtner. 2015. Breaking Payloads with Runtime Code Stripping and Image Freezing. BHUSA. https://www.blackhat.com/us-15/briefings.html#breaking-payloads-with-runtime-code-stripping-and-image-freezing

[47] Gail C. Murphy, David Notkin, William G. Griswold, and Erica S. Lan. 1998. An Empirical Study of Static Call Graph Extractors. *ACM Trans. Softw. Eng. Methodol.* 7, 2 (April 1998), 158–191.

[48] National Security Agency. 2019. Ghidra. nsa.gov. https://www.nsa.gov/resources/everyone/ghidra/

[49] Khanh Nguyen and Guoqing Xu. 2013. Cachetor: Detecting Cacheable Data to Remove Bloat. In *Proc of ACM ESEC/FSE*. 268–278.

[50] Ben Niu and Gang Tan. 2014. Modular Control-Flow Integrity. In *Proc. of ACM PLDI*. 577–587.

[51] Ben Niu and Gang Tan. 2015. Per-Input Control-Flow Integrity. In *Proc. of ACM CCS*. 914–926.

[52] PaX Team. 2003. Address Space Layout Randomization (ASLR). https://pax.grsecurity.net/docs/aslr.txt

[53] Anh Quach, Aravind Prakash, and Lok Yan. 2018. Debloating Software through Piece-Wise Compilation and Loading. In *27th USENIX Security Symposium*. USENIX Association, Baltimore, MD, 869–886.

[54] Nguyen Anh Quynh. 2014. Capstone: Next-Gen Disassembly Framework. In *BHUSA*.

[55] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case Reduction for C Compiler Bugs. In *Proc. of PLDI*. 335–346.

[56] Jonathan Salwan. 2011. ROPgadget - Gadgets finder and auto-roper. http://shell-storm.org/project/ROPgadget/.

[57] Sascha Schirra. 2014. Ropper - rop gadget finder and binary information tool. https://scoding.de/ropper/.

[58] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *Proc. of IEEE S&P*. 745–762.

[59] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2011. Q: Exploit Hardening Made Easy. In *Proc. of USENIX SEC*. 25–41.

[60] Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *Proc. of ACM CCS*. 552–561.

[61] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. 2018. TRIMMER: Application Specialization for Code Debloating. In *Proc. of ACM/IEEE ASE*. ACM, 329–339.

[62] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*. 138–157.

[63] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2013. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *Proc. of IEEE S&P*. 574–588.

[64] Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund, and Thomas Walter. 2009. Breaking the Memory Secrecy Assumption. In *Proc. of EuroSec*. 1–8.

[65] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. 2018. Perses: Syntax-guided Program Reduction. In *Proc. of ICSE*. 361–371.

[66] t0x0sh. 2014. A tool to help you write binary exploits. https://github.com/t00sh/rop-tool.

[67] The Chromium Projects. 2018. Testing and infrastructure. https://www.chromium.org/developers/testing

[68] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-edge Control-Flow Integrity in GCC & LLVM. In *Proc. of USENIX SEC*. 941–955.

[69] TIS Committee. 1995. Tool Interface Standard (TIS)– Executable and Linking Format (ELF)– Specification– Version 1.2. http://refspecs.linuxbase.org/elf/elf.pdf.

[70] Trail of Bits. 2019. McSema. GitHub. https://github.com/trailofbits/mcsema

[71] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. 2015. Practical Context-Sensitive CFI. In *Proc. of ACM CCS*. 927–940.

[72] Gregor Wagner, Andreas Gal, and Michael Franz. 2011. "Slimming" a Java Virtual Machine by Way of Cold Code Removal and Optimistic Partial Program Loading. *Sci. Comput. Program.* 76, 11 (Nov. 2011), 1037–1053.

[73] Shuai Wang, Pei Wang, and Dinghao Wu. 2015. Reassembleable Disassembling. In *24th USENIX Security Symposium (USENIX Security)*. 627–642.

[74] Zhe Wang, Chenggang Wu, Jianjun Li, Yuanming Lai, Xiangyu Zhang, Wei-Chung Hsu, and Yueqiang Cheng. 2017. ReRanz: A Light-Weight Virtual Machine to Mitigate Memory Disclosure Attacks. In *Proc. of ACM VEE*. 143–156.

[75] Richard Wartell, Yan Zhou, Kevin W. Hamlen, Murat Kantarcioglu, and Bhavani Thuraisingham. 2011. Differentiating Code from Data in x86 Binaries. In *Proc. of ECML-PKDD*. 522–536.

[76] David Williams-King, Graham Gobieski, Kent Williams-King, James P. Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P. Kemerlis, Junfeng Yang, and William Aiello. 2016. Shuffler: Fast and Deployable Continuous Code Re-Randomization. In *USENIX OSDI*. 367–382.

[77] Windows Dev Center. 2018. PE Format. https://msdn.microsoft.com/library/windows/desktop/ms680547(v=vs.85).aspx.

[78] Patrick Wollgast, Robert Gawlik, Behrad Garmany, Benjamin Kollenda, and Thorsten Holz. 2016. Automated Multi-Architectural Discovery of CFI-Resistant Code Gadgets. In *Proc. of ESORICS*. 602–620.

[79] Guoqing Xu. 2012. Finding Reusable Data Structures. In *Proc. of ACM OOPSLA*. 1017–1034.

[80] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Softw. Eng.* 28, 2 (February 2002), 183–200.

[81] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. 2013. Practical Control Flow Integrity and Randomization for Binary Executables. In *Proc. of IEEE S&P*. 559–573.

[82] Mingwei Zhang and R. Sekar. 2013. Control Flow Integrity for COTS Binaries. In *Proc. of USENIX SEC*. 337–352.

A GADGET COLLECTION

Given an x86-64 ELF binary as input, our tool identifies its executable sections, by parsing the respective ELF header(s), and proceeds as follows. First, it pinpoints *all* the byte sequences (in the previously-identified executable sections) that correspond to `RET`, indirect `JMP` and `CALL` instructions; the location of every such instruction is marked, as `GAD_END`, because it indicates the end of a gadget, while the instruction opcode (`RET`, `JMP`, `CALL`) specifies the type of the gadget (i.e., ROP, JOP, or COP).

Second, `GAD_BEGIN` is set to `GAD_END - 1`, and Capstone is used to linearly disassemble the region `[GAD_BEGIN, GAD_END]`; every resulting code snippet is by definition a gadget (as it ends with an indirect branch instruction), and the type of the instruction that starts at `GAD_BEGIN` is used to further classify the whole gadget (more about this below). Next, `GAD_BEGIN` is set to `GAD_END - 2`, and step 2 is repeated; the process is executed recursively for `GAD_BEGIN = GAD_END - 3`, ..., `GAD_BEGIN = GAD_END - k`, where `k` (bytes) is an input parameter, typically set to 10, in accordance to modern automated gadget finding tools, like ROPgadget [56], Ropper [57], xrop [10], and rop-tool [66].

(We had to develop our own analysis framework, as none of the aforementioned gadget finding tools is intended for quantitative analyses [78].) The above procedure discovers instruction (sub)sequences, of size 1, 2, …, `k` bytes, which are prefixes of an indirect branch, thereby constituting gadgets. For example, `|mov (%rdi), %rax; pop r14; pop r15; pop rbp; ret|` will be accounted as 4 separate ROP gadgets: (a) `|pop rbp; ret|`, (b) `|pop r15; pop rbp; ret|`, (c) `|pop r14; pop r15; pop rbp; ret|`, and (d) `|mov (%rdi), %rax ... ret|`. Also, gadgets (a) – (c) will be classified as loading a register with a value from the stack, whereas (d) will be classified as a memory load. Lastly, gadgets that include invalid instructions sequences, like privileged instructions (`hlt`, `in`/`out`, `rdmsr`/`wrmsr`, etc.), instructions that access non-general-purpose, registers (`dr#`, `cr#`), and vendor-specific ISA extensions (MMX, SSE, AVX, TSX), are all filtered out.

B GADGET REDUCTION WITHOUT CFI

Table 6 summarizes our findings regarding conventional gadgets. Similar to the previous case, the reduction of gadgets is analogous to the achieved code reduction. Again, Nibbler seems to be *very effective on certain gadget classes*, like Branch, where the achieved gadget reduction is (on average) a bit higher than the respective code reduction. Similarly, Stack Pivot and Stack Lift gadgets, in certain real-world applications, such as Coreutils and MySQL, are reduced considerably (again on average) or eliminated completely (e.g., `libdl` in Coreutils). Our results indicate that although Nibbler does not entirely protect against code reuse, it raises the bar significantly for an attacker that tries to automatically stitch together code snippets to mount a ROP/JOP/COP attack [16, 56, 57, 59]; and it achieves this with practically zero run-time performance overhead.

C STATIC LINKING

We argued that shared libraries have numerous advantages over static ones in real-world deployments. Nevertheless, static linking, specially with the addition of link-time optimization [40] (LTO), has the potential to eliminate even more code. We compare Nibbler with static linking to highlight its debloating capabilities, as well as the large overhead associated with static linking. We statically linked all applications, and compared the total size of the statically-linked applications against the dynamically-linked ones, with all their required libraries, after we have applied Nibbler. Figure 5 shows the total size of each application set. On average, static linking reduces the size of the application by 50.32%, while Nibbler by 55.29%, indicating that our code elimination techniques approximate what can be achieved through recompilation and static linking. If we, however, look at the total code in the system, the first (statically-linked) take 78.57 MB, while the latter (nibbled) only 14.34 MB.

Effect of LTO. LTO corresponds to inter-procedural optimizations, performed during linking, which may eliminate even more code. To measure its effect, we built Nginx with LLVM/Clang (its LTO implementation is more mature than GCC's). We found that LTO does not significantly eliminate code, when compared to static linking without LTO. Table 7 summarizes the debloating effects on library code. We notice that LTO does not significantly eliminate code, as code attributed to libraries is reduced only by 2.1%, compared to conventional static linking. Interestingly, certain libraries may require additional functions/code when they are built statically, allowing Nibbler to achieve better results than static linking with LTO in such cases (see `libc` and `libcrypt` in Table 7). In general, both types of static linkage result in ≈10% less library code than Nibbler's thinned libraries, indicating that our code elimination techniques approximate the "optimal" reduction rate(s) sufficiently.

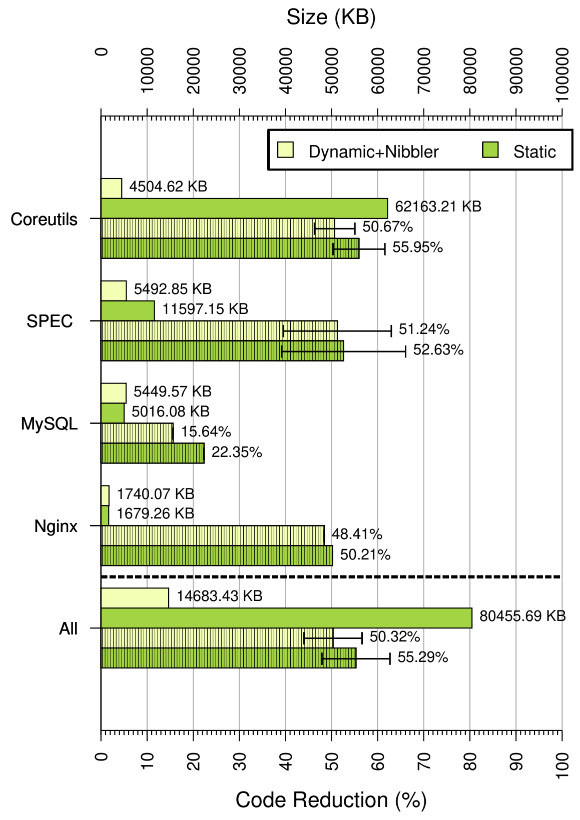


Figure 5: Comparison of *total app. size* (in KB; binaries + required libraries), when using Nibbler with dynamically-linked binaries (shared libraries) vs. statically-linked binaries (library code is duplicated)—top x-axis. The hatched bars correspond to average code reduction and standard deviation of *total app. size* across a set—bottom x-axis.

D REAL-WORLD EXPLOITS

We evaluated Nibbler against *pre-compiled*, real-world core-reuse exploits. Specifically, we replicated: (1) a ROP-based exploit against Nginx (CVE-2013-2028[3]), (2) a ROP/`ret2libc`-based exploit against mcrypt (CVE-2012-4409[4]), and (3) a `ret2libc`-based exploit against Tinyproxy/`glibc` (CVE-2015-7547[5]). Next, we nibbled the libraries of the applications and re-tested the exploits. In all cases, Nibbler managed to stop the attack, by removing gadgets, required by the exploit(s), or whole (`libc`) functions. Note that the attacker can

[3]https://github.com/danghvu/nginx-1.4.0
[4]https://www.exploit-db.com/exploits/22928/
[5]https://researchcenter.paloaltonetworks.com/2016/05/how-cve-2015-7547-glibc-getaddrinfo-can-bypass-aslr/

Table 6: Gadget reduction results. The percentages correspond to removed gadgets (thinned vs. vanilla). Entries marked with 'N/A' indicate absence of certain gadget classes in vanilla.

Suite		Total	Gadget Type													
			Stack		Load Reg.		Memory		Arithmetic		Logic		Branch		Syscall	NOP
			Pivot	Lift	Reg.	Stack	Load	Store	Reg.	Mem.	Reg.	Mem.	jmp	call		
SPEC																
libpthread	(92.79%)	96.1%	100.0%	95.9%	97.8%	99.1%	100.0%	90.2%	93.1%	96.1%	93.3%	99.2%	96.2%	80.0%	96.5%	95.8%
libm	(39.76%)	48.4%	28.6%	44.3%	53.5%	30.5%	75.6%	39.4%	54.3%	47.3%	72.2%	44.9%	45.9%	55.9%	71.4%	40.8%
libc	(42.01%)	51.8%	68.9%	30.2%	63.5%	68.8%	38.3%	39.5%	53.4%	47.1%	59.7%	56.8%	41.0%	57.8%	76.6%	48.9%
libgcc	(72.35%)	75.8%	55.4%	84.9%	62.9%	53.7%	32.0%	75.5%	83.9%	74.8%	91.0%	74.0%	86.2%	14.3%	0.0%	77.7%
libstdc++	(48.88%)	63.9%	67.4%	46.2%	60.4%	67.5%	51.1%	69.3%	72.5%	68.5%	68.2%	72.9%	56.5%	52.9%	50.0%	68.2%
Coreutils																
libpthread	(39.94%)	50.4%	51.5%	46.5%	53.1%	50.1%	44.1%	59.8%	44.6%	49.9%	56.9%	43.2%	48.8%	0.0%	44.2%	55.2%
libdl	(75.41%)	85.5%	100.0%	100.0%	92.3%	88.9%	100.0%	90.0%	85.2%	93.3%	91.3%	50.0%	76.9%	80.0%	N/A	77.3%
libselinux	(53.13%)	67.3%	64.7%	52.9%	62.0%	63.2%	80.7%	84.7%	65.2%	64.2%	84.1%	79.9%	62.2%	91.8%	N/A	63.8%
libacl	(69.99%)	70.4%	60.0%	74.4%	80.0%	65.9%	69.2%	57.1%	52.7%	86.5%	78.2%	73.5%	73.9%	91.4%	N/A	67.6%
librt	(71.64%)	70.3%	80.8%	85.2%	70.3%	77.2%	40.0%	79.7%	62.6%	59.7%	66.7%	73.7%	81.0%	50.0%	66.7%	69.3%
libpcre	(10.15%)	12.1%	22.6%	10.2%	15.2%	7.9%	16.7%	21.2%	11.5%	11.3%	22.6%	12.9%	8.1%	12.8%	0.0%	16.0%
libc	(28.28%)	38.6%	45.0%	21.8%	49.0%	45.9%	33.4%	31.0%	44.5%	35.2%	47.3%	43.7%	28.5%	50.1%	49.1%	36.2%
libattr	(38.17%)	32.4%	68.4%	50.0%	10.0%	71.1%	0.0%	22.2%	37.9%	25.7%	18.2%	25.0%	16.7%	0.0%	N/A	46.1%
libgmp	(46.43%)	57.3%	61.9%	49.2%	65.7%	60.0%	42.6%	48.3%	66.0%	55.6%	55.0%	63.2%	52.2%	48.2%	N/A	55.1%
libgcc	(92.26%)	98.2%	95.9%	99.6%	99.0%	99.0%	92.8%	98.0%	97.6%	99.3%	99.2%	98.0%	98.7%	85.7%	100.0%	97.5%
libcap	(49.97%)	56.8%	62.5%	60.0%	75.9%	54.1%	100.0%	78.6%	50.8%	29.2%	50.0%	50.0%	54.5%	20.0%	100.0%	51.9%
MySQL																
libm	(44.25%)	49.9%	29.9%	45.4%	54.7%	32.4%	75.6%	41.8%	55.8%	47.9%	71.9%	46.4%	51.6%	55.9%	71.4%	42.4%
libpthread	(34.33%)	44.4%	47.0%	37.8%	48.6%	45.6%	44.1%	50.4%	39.9%	41.4%	50.5%	43.9%	38.7%	0.0%	40.7%	46.6%
libz	(29.37%)	47.8%	38.9%	22.6%	63.6%	40.7%	79.8%	57.8%	57.3%	43.3%	66.0%	27.3%	39.6%	59.3%	N/A	51.6%
libc	(34.23%)	42.5%	55.5%	26.1%	53.2%	54.3%	30.4%	30.7%	45.3%	38.0%	48.9%	45.3%	34.3%	44.5%	57.0%	41.0%
libgcc	(56.10%)	61.1%	44.6%	59.9%	60.9%	40.1%	22.7%	46.9%	72.1%	53.6%	81.2%	62.0%	73.3%	14.3%	0.0%	63.7%
libstdc++	(45.04%)	59.0%	60.5%	39.1%	54.4%	62.2%	46.9%	64.8%	68.9%	63.4%	64.4%	64.9%	53.2%	49.7%	50.0%	63.6%
libdl	(26.70%)	38.6%	0.0%	0.0%	35.9%	27.8%	78.3%	43.3%	37.0%	68.3%	56.5%	0.0%	46.2%	33.3%	N/A	33.3%
libcrypt	(4.30%)	7.3%	14.3%	5.0%	6.2%	9.5%	25.0%	0.0%	10.9%	4.0%	0.0%	8.7%	0.0%	0.0%	0.0%	5.7%
Nginx																
libpthread	(47.75%)	49.3%	48.5%	49.4%	47.2%	49.7%	42.6%	54.1%	43.2%	51.5%	55.1%	47.0%	52.5%	0.0%	43.0%	53.5%
libz	(60.91%)	69.5%	63.9%	53.8%	80.3%	60.7%	87.9%	67.5%	81.1%	62.2%	78.7%	78.2%	66.7%	70.4%	N/A	74.3%
libc	(40.78%)	49.2%	64.0%	32.3%	58.2%	64.0%	38.1%	36.1%	51.9%	44.5%	56.5%	58.0%	39.3%	58.0%	59.4%	46.7%
libdl	(26.70%)	38.6%	0.0%	0.0%	35.9%	27.8%	78.3%	43.3%	37.0%	68.3%	56.5%	0.0%	46.2%	33.3%	N/A	33.3%
libpcre	(10.09%)	11.5%	22.6%	10.2%	14.2%	7.9%	16.2%	17.6%	11.1%	10.3%	19.2%	12.2%	7.9%	12.8%	0.0%	15.0%
libcrypt	(6.42%)	11.1%	25.0%	7.1%	6.2%	13.9%	25.0%	0.0%	10.9%	6.0%	25.0%	8.7%	0.0%	0.0%	0.0%	10.6%
libcrypto	(83.56%)	89.2%	90.8%	80.7%	87.7%	92.3%	87.1%	85.8%	90.5%	79.3%	87.6%	88.3%	90.3%	83.3%	91.7%	89.3%

easily modify their exploits to use gadgets, or whole functions, from the residual code. The purpose of this experiment, however, was to demonstrate that Nibbler can thwart *canned* exploits with no additional run-time overhead.

Table 7: Per-library debloating in Nginx with Nibbler, static linking, and LTO.

	Code in Scope			
	# of Functions (Size in KB)			
	Dynamic	Nibbler	Static	Static+LTO
libcrypto	5085	317	33	33
	(1155.25)	(105.76)	(22.14)	(22.14)
libc	2889	921	1025	1025
	(1172.84)	(683.24)	(697.08)	(697.08)
libpthread	297	142	55	55
	(49.43)	(25.69)	(6.01)	(6.01)
libz	140	46	66	41
	(84.55)	(31.83)	(40.01)	(28.35)
libpcre	74	45	48	22
	(158.82)	(115.56)	(122.57)	(108.49)
libcrypt	39	28	37	37
	(20.45)	(19.29)	(30.38)	(30.38)
libdl	24	16	9	9
	(2.54)	(1.69)	(0.06)	(0.06)
Total (Lib.)	8548	1581	1270	1222
	(2643.89)	(990.07)	(911.24)	(892.51)