LibFilter: Debloating Dynamically-Linked Libraries through Binary Recompilation

Benjamin Shteinfeld
Brown University

Abstract

Both standard and newly written applications hardly ever implement all functionality they use from scratch; rather they rely on prewritten shared libraries, which implement useful and reusable functionality, such as threading, common file/networking operations, and cryptographic primitives. Shared libraries are widely used because they decrease application development time, reduce binary size by allowing for reuse of code and make distribution of applications simpler. However, from a security perspective, they facilitate code reuse attacks. Regardless of how many symbols an application uses from a shared library, the library's code is mapped into the application's address space in its entirety. This increases the amount of code marked as executable and thus available in the address space for a potential attacker to use in an exploit.

In this paper, we propose LibFilter: a system that, given an x86-64 application, identifies unused functions in its dynamically-linked libraries and erases them without access to source code. This makes code-reuse attacks harder by removing unused code from the address space. We implement and test a prototype of this system and found that it can reduce the number of functions available in the address space by 76.5% in Coreutils and 52.7% in SQLite. LibFilter can also work in conjunction with other hardening mechanisms such as control-flow integrity, execute-only memory, and continuous code re-randomization.

1 Introduction

With the widespread adoption of data execution prevention (DEP), which prevents malicious data injected into the address space from being executed, many modern exploits rely on code-reuse attacks, such as return-oriented programming (ROP) [8]. This technique typically involves gaining control of a code pointer such as a return address or function pointer in order to redirect the control flow of the program to a specific address. Through an analysis of program code, an attacker can find short snippets of instructions ending in `ret`, called gadgets, and chain them together to perform arbitrary execution. This attack technique can be generalized to jump-oriented and call-oriented programming (JOP/COP), which use indirect `jmp` and `call` instructions to perform the chaining of gadgets [9].

While ROP/JOP exploits are becoming increasingly popular, at the same time modern applications are getting larger and relying on more libraries. For example, curl v7.52 has 35 dependencies, and Chromium v73.0 has 153 dependencies. More and larger libraries lead to more code being mapped into an application's address space at run-time because shared libraries are mapped in their entirety even if only a few functions in them are actually called from the application. Larger amounts of code in the address space facilitate code reuse attacks because of the increase in the number of potential gadgets an attacker can use to construct an exploit.

There have been a plethora of approaches trying to mitigate this problem: control flow integrity [1], destructive code reads [10], continuous code rerandomization [11], and many more. All of these approaches attempt to make it harder to find and use existing ROP/JOP gadgets available in the address space. Despite these attempts, they still leave many gadgets in the attack space available for use by attackers. For example, consider an application hardened with coarse-grained control flow integrity. This restricts an attacker to using only call-preceded gadgets, but against an application linked against many shared libraries, they will be many call-preceded gadgets which will allow an attacker to construct an exploit.

This becomes the starting point for this project. Orthogonally from all of these other defenses, our goal is to reduce the number of gadgets available for an attacker to use. This will make it more difficult for an attacker to find the gadgets necessary to construct a successful ROP exploit. The key observation we make to safely reduce the number of gadgets is that it is harmless from the application's perspective to remove code that is unreachable.

We perform static analysis on binaries and their shared libraries using Egalito [6], a binary analysis and recompiler, to construct a function call graph. We use this to find a set of

functions which are possibility reachable in the binary and all its shared libraries. We compare this reachable set with the set of all functions in all shared libraries. The difference contains the set of functions which are unreachable and can thus be removed. We create new versions of the shared libraries with the unreachable functions replaced with one byte `hlt` instructions, which will cause a segmentation fault if executed. This means that many ROP gadgets previously available for attackers to use would now be detected and cause the application to terminate.

Our contributions in this project are developing and testing a prototype LibFilter system which reduces the number of bytes in Coreutils by 37.2% and in SQLite by 29.7%. All functionality tests pass after our debloating, which indicates that we did not remove code that is actually used and are not affecting the functionality of applications which we debloat.

2 Background and Related Work

2.1 Control Flow Integrity

Control Flow Integrity (CFI) was introduced as an attempt to restrict code reuse attacks by preventing control flow hijacking [1]. In most exploits, an attacker hijacks the control flow of a vulnerable program by overwriting the value of a code pointer, typically a function pointer or return address, to cause the control flow of the program to begin executing either injected or existing code. Because of this, CFI hardens control flow transfers with checks to ensure that the destination of the transfer is valid.

This is done through enforcement of the CFI safety property. It states that execution flow in a program follows a predefined set of paths which can be analyzed at compile time. This set of paths is called a Control Flow Graph (CFG) and can be determined by static analysis. Successful enforcement of the property would prevent the control flow of the program from executing along a path that was not intended, such as calling arbitrary functions and jumping into the middle of functions.

Abadi et al. implement CFI by inserting instrumentation in binaries before any indirect control flow transfer to check that the jump corresponds to a valid edge in the CFG. During the analysis phase, for every transfer, all possible destinations must be calculated and given an ID which is embedded into the binary text as `prefetch` instructions. Before every `call` and `ret` instruction a check is performed to ensure the expected ID is at the destination. If one of these run-time ID checks fails, a violation is triggered. These run-time checks enforce that transfers of control only occur between edges in the CFG.

Notably, the CFG is an over-approximation because all indirect calls have edges to all address taken functions. An exact representation of the CFG is an unsolvable problem, and under-approximation could lead to breaking functionality. Thus, even finer grained CFI will always leave attackers with a window to hijack control flow using edges present in the CFG [2–4].

2.2 Shared Library Debloating

Nibbler. With a similar motivation, Nibbler [5] attempts to improve security by library thinning. Nibbler identifies unused code in shared libraries and removes it. Nibbler does this by analyzing the Function Call Graph (FCG) of a binary and all its required libraries, and composes the FCGs into an application-level FCG. Using this, all code that is not reachable by the application need not be shipped with the application. Thus, Nibbler chooses to replace all unreachable code with the `int3` instruction. This instruction will trap and kill the process, meaning a mistake from an attacker will likely be spotted. This instruction is also a single byte, meaning that it can be used to fill up precisely the amount of space that was once taken up by functions, and thus no code relocation needs to take place. Additionally, a single byte instruction does not introduce any new or misaligned gadgets that an attacker could exploit.

By removing unused code, Nibbler reduces the number of gadgets available in typical real-world binaries by 32%, in SPEC CPU2006 by 52% and 64% in when CFI is present.

Piecewise. Piecewise Compilation and Loading [13] attempts to debloat libraries using a compiler and loader based approach. Using the LLVM toolchain, this work creates a compiler pass which outputs call-graph information from the compiler into a new `.dep` ELF section. At load time, the FCG information in the `.dep` section is used by their custom loader to determine at run-time which functions in the library are needed and which are not. The loader erases the functions which it determines are not reachable by the binary. It erases the code by invoking `mprotect` on the pages containing the code to be erased to make it writeable/non-executable, overwrites code with illegal instruction opcodes, and maps the code pages back to executable. This approach is able to remove on average 79% of code from libraries used by Coreutils.

This approach only requires libraries to be compiled and stored once, whereas other approaches likely require multiple versions of debloated libraries per binary. The main disadvantages are the performance overhead of having to erase code during load time, the need to recompile source code, and its dependency on LLVM and thus `musl` instead of `glibc`.

2.3 Binary Recompilation

Many hardening techniques that operate on binaries require accurate disassembly, symbol locations and internal code references to safely and confidently implement the desired hardening instrumentation. Many COTS stripped binaries are

believed to not have enough information to be able to successfully identify all this information. In fact, there are works attempting to add more information to binaries at the compiler level to be able to help harden binaries [7].

Egalito [6] is a binary recompiling framework which claims that modern binaries contain enough information to successfully identify all code references and safely rearrange code, introducing little overhead. Egalito's analysis gives the programmer access to a compiler-like intermediate representation (IR) of binary which she can manipulate to implement generic forms of hardening. For example, Egalito has been used to implement function/data reordering, CFI, binary-level retpolines, and continous code randomization. Importantly, Egalito is an egalitarian defense as all hardening mechanisms developed using Egalito can be applied to Egalito itself.

3 Design and Implementation

3.1 Call-graph Extraction

We use Egalito to implement our analysis to extract the application's FCG to determine the reachable set of functions in all of its shared libraries. In this project, we built increasingly accurate FCGs using a variety of analysis techniques.

Static FCG. The most basic FCG we can extract is a static FCG. This includes all functions that are reachable from the binary entry point and can be found by analyzing all control flow transfer instructions such as `call`s and `jmp`s in the application. An edge exists between two functions if the caller function contains such an instruction with the callee function as a target. As seen in Figure 2, function calls to shared libraries go through the procedure-linkage table (PLT) which invokes the dynamic loader to place the address of the library function in the global offset table (GOT). This affects our static reachability analysis because the address of the callee is only available at run-time. To handle this case, we run a `ResolvePLT` pass in Egaltio, which inserts links between the PLT stub function and the library function. Using this technique we can determine the set of statically reachable functions in the binary and all of its shared libraries.

Indirect FCG. The static FCG is an under-approximation of the true set of reachable functions due to the problem of function pointers. The issue arises when we try to determine the target of an indirect function call, we do not know the exact target(s) until run-time. Nearly every non-trivial program will make use of function pointers, and any program that links against `libc` does. To address this, we construct an indirect FCG, which adds edges to the static FCG. We use Egalito to get all address taken (AT) functions (i.e., f = &func), and add all AT functions as targets of all indirect function calls as seen in Figure 3.

This approach results in an over-approximation of the FCG

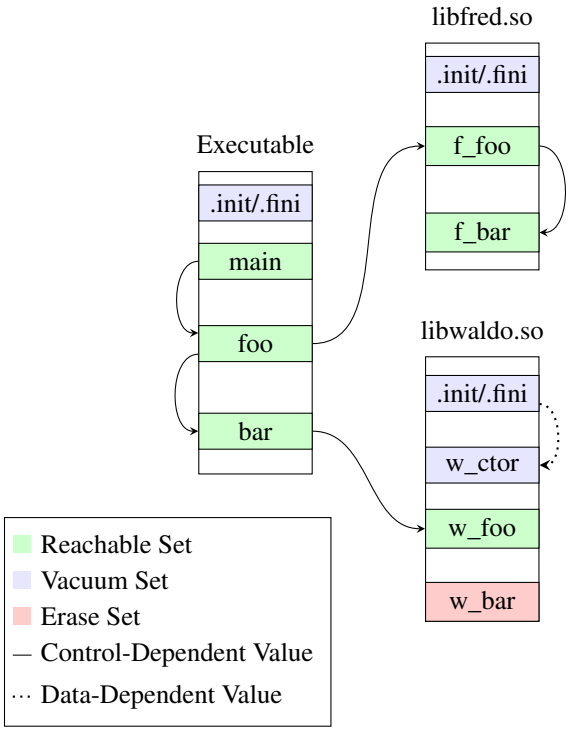

Figure 1: Overview of LibFilter approach. Given an application and its shared libraries, LibFilter performs static analysis to determine the set of reachable functions, the set of initialization and destruction functions (vacuum), and the set of functions to be erased. In the example shown, `w_bar` is not reachable from the main binary and thus can safely be erased. The solid lines represent links we determine through control flow instructions, whereas the dotted lines represent links to functions we determine are reachable via data (`.init` and `.fini` arrays).

because there are edges in the graph which would never occur in the actual run-time of the program. Unnecessary edges and nodes in the FCG hinder our ability to erase as many functions as safely possible. To address this, we implemented an optimization based on AT function pruning used in Nibbler. If a function is address taken only inside of a function which is statically unreachable, then we know that address taken function will never be the target of an indirect function call and thus we can safely remove it (Figure 4).

This is only an optimization and still provides us with an over-approximation of the real FCG, as determining the actual FCG from static analysis of the binary is impossible. It would be interesting to see how future work in this area can allow us to safely further decrease the reachable set of address taken functions. A potential heuristic to remove edges is function signatures. For example, a function pointer of type `int (int, char*)` will never dereference to a function

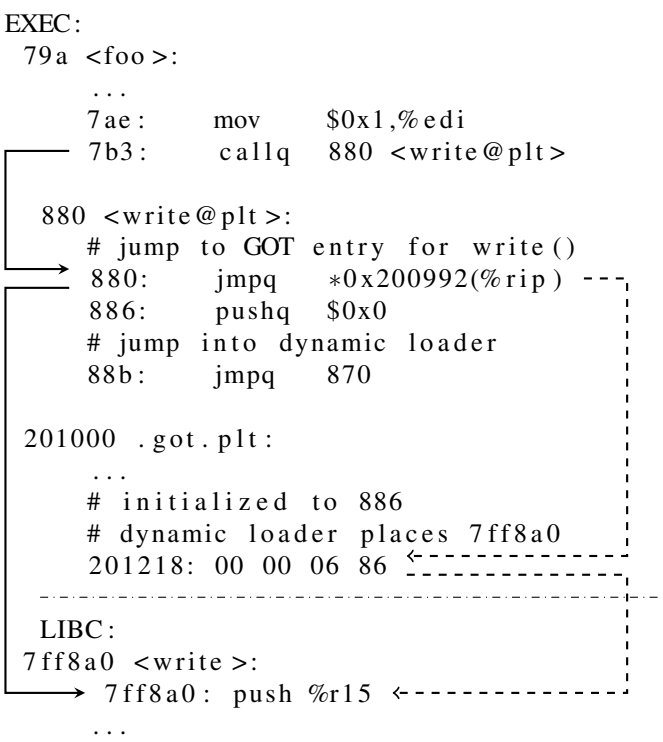

Figure 2: Flow of control of a call to libc's `write()` from an application. This requires calling the PLT stub which on it's first invocation will invoke the dynamic linker and place the correct address into the GOT. On subsequent calls, the PLT stub will jump directly to `write()`. Through Egalito's analysis we are able to determine control-flow links (solid lines) from `foo` to `write@plt` to `write` and data-flow links (dashed lines) from `write@plt` to `.got.plt` to `write`.

with a signature of `void (short, float)`. Implementing more heuristics like this will reduce the over-approximation of the FCG.

Vacuum FCG. The indirect FCG described in the section above is an over-approximation of the FCG rooted from the main binary's entry point, `_start`, but it does not take into account functions used in program initialization. Before `_start` is called (when the binary takes control of execution), the loader must do a lot of work to set the application and its libraries in the address space, for example set up relocations. As part of this setup, the loader executes functions stored as function pointers in the `.init` section of the binary and after the program terminates, it executes the function pointers stored in the `.fini` section. Since these function pointers are never explicitly address taken, they will not be picked up using the indirect FCG. If we do not include them in our FCG, they will be erased and as a result, erased binaries would crash

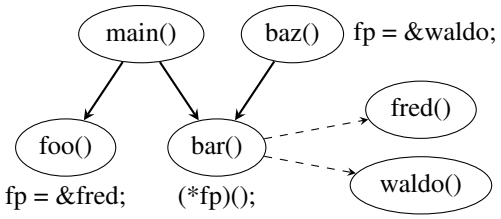

Figure 3: Construction of Indirect FCG. `fred()` and `waldo()` are address taken functions. When `bar()` dereferences the `fp` function pointer, its possible targets are `fred()` and `waldo()`.

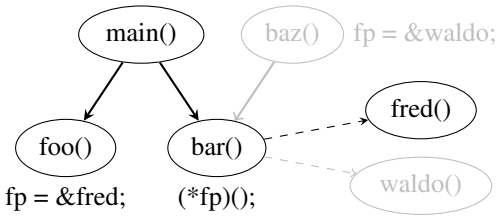

Figure 4: `waldo()` is only address taken inside `baz()` which is unreachable from `main()`. Thus, `waldo()` cannot be a valid target of the function pointer in `bar()`. This pruning reduces the reachable set of functions, allowing for more functions to be erased.

on startup.

For this issue, we developed a Vacuum FCG which encapsulates all of this analysis. In particular, we treat all the function pointers in the `.init` and `.fini` sections as entries points to the binary, so that later in our FCG analysis, we add all of the transitive dependencies of these functions to our reachable set, so that they are not erased.

3.2 Dynamic Loader Challenges

In this process, we discovered that `ld`, the linux dynamic loader, is tightly intertwined with `glibc`. In particular, during the course of program startup, the dynamic loader invokes seven functions inside of `libc` which we do not capture in our analysis. For this reason, as a workaround we had to hardcode those functions inside of `libc` as always in the reachable set because the dynamic loader depends on them.

Another issue we ran into during the development of this project is the order of symbol resolution of symbols with the same name by the loader. When debloating SQLite, we noticed that `libreadline` made a call to `read()` and the loader bound that symbol to the `libpthread` implementation of `read()`, while Egalito parsed that symbol and bound it to the `libc` implementation of read. This bug caused our framework to incorrectly erase the `libpthread` implementation of `read()` and caused a crash. As a workaround to this bug in Egalito's symbol resolution, we decided to always keep

`read()` in the reachable set of functions and have yet to encounter another case like this. This bug will soon be patched in Egalito.

3.3 Erasing Code

Our Egalito analysis computes the reachable set of functions per library and finds the difference between that and the set of all functions in the library. This leaves us with a list of functions and their sizes to erase for each library. We erase functions by overwriting them with x86 `hlt` instructions. We do this by using the `readelf` utility to convert virtual addresses of functions to their byte offset in the library ELF file on disk. With this conversion, we can overwrite a function with a single `write()` call.

There is a large benefit to using the `hlt` instruction to erase functions. If this instruction is executed, perhaps during an attacker's exploit, the program will terminate, thus alerting the client of an attack. The `hlt` instruction is only one-byte which means it works for erasing functions of both odd and even byte length, does not change the size of the binary, and does not introduce any new gadgets along unaligned instruction boundaries.

After we have made copies of an application's libraries and erased them, we indicate to the loader that the binary should use the erased libraries by using the `patchelf` utility to change the binary's and all the libraries' `rpath` to a directory with the newly erased ELF files. This allows us to move the binary around and execute it from any directory without the need for any environment variables. We found this to be convenient when testing as passing along the required arguments to the `LD_PRELOAD` environment variable to use the erased libraries is not trivial and required a different solution for each application with a different test framework.

3.4 `IFUNC`s

GNU `libc` introduces the concept of `IFUNC`s as a mechanism for developers to provide multiple implementations of a function, each one optimized for a particular system. The `IFUNC` itself is a resolver function which at run-time determines which implementation to use. This creates a problem for our analysis similar to that of the `.init` and `.fini` sections. Because the loader invokes all `IFUNC` resolvers at load time, we consider all `IFUNC`s as valid entry points into the library when constructing the FCG. As a result we are unable to erase any `IFUNC`s or their transitive dependencies.

Potential future work could be to optimize this case. Because the loader executes all the resolver functions on startup, we cannot remove any of those. However, if in our FCG analysis, we see that the `IFUNC` is never called, then we can safely erase all implementations of that `IFUNC` and its transitive dependencies. We found that `libc` v2.27 exports 56 `IFUNC`s, but that the average Coreutil binary only uses 7. If we see that an `IFUNC` is in fact called from our FCG, then we must keep all implementations of that `IFUNC`. Even though only one of the implementations will be called on the current machine, if we erase the others, we will cause a crash on a different machine which invokes another implementation.

	Coreutils	**SQLite**
Average KB Erased	590	595
Average Byte % Erased	37.2%	29.7%
Average Functions Erased	1914	2165
Average % Function Erased	76.5%	52.7%

Table 1: The amount of code and functions erased from the address space of Coreutils applications and SQLite.

3.5 Dynamically-Loaded Code

Code loaded dynamically through `LD_PRELOAD` or calls to `dlopen()` and `dlsym()` create a problem in FCG analysis. The code loaded at run-time could reference code from libraries which we erased because we determined it was unreachable. The safest solution would be to not erase applications which load code at run-time because there is no way to get a complete analysis. In the cases that we know what is being loaded, we can augment our FCG analysis to include the required dependencies. This was necessary to get the Coreutils tests to pass, as there were a few test cases which dynamically loaded code. An analysis in the Nibbler project showed that 13.8% of Debian v9's 25,256 packages use `dlopen()` and `dlsym()` to dynamically load code as run-time. This is a known limitation of our approach.

In this project we found that `libc` dynamically loads code from several sub-libraries, called Network Security Services (NSS) libraries. To avoid potentially erasing a function from `libc` which one of these libraries could potentially need, we add NSS libraries as dependencies to our application inside of Egalito, so that it is able to include that code in the reachabilty analysis. We take the same approach for `libpthread` which dynamically loads codes from `libgcc` for stack unwinding purposes.

4 Evaluation

We evaluate LibFilter on Linux x86-64: Coreutils (v8.30) and SQLite (v3.27.2), using GNU libc (v2.27). We ran LibFilter over all 107 binaries included in Coreutils and over SQLite. For each binary, this process resulted in a set of newly debloated libraries which erased unused functions. We ran all the test suites in Coreutils and SQLite, all of which passed.

Library	**Coreutils** (avg. of 107 binaries)		**SQLite**	
	Functions Erased	**Bytes Erased**	**Functions Erased**	**Bytes Erased**
libc	1813 (77.7%)	567 KB (37.6%)	1623 (69.5%)	494 KB (32.8%)
libpthread	194 (56.5%)	33 KB (57.8%)	146 (42.4%)	19 KB (33.2%)
libdl	17 (40.5%)	3 KB (77.6%)	7 (16.7%)	1 KB (25.2%)
librt	64 (54.5%)	13 KB (95.4%)	65 (55.1%)	13 KB (96.2%)
libnss_compat	32 (59.5%)	18 KB (77.3%)	27 (50.0%)	15 KB (61.3%)
libnss_systemd	28 (18.4%)	5 KB (3.3%)	28 (18.4%)	19KB (33.2%)
libgmp	413 (62.3%)	180 KB (47.5%)	N/A	N/A
libpcre	20 (31.3%)	7 KB (2.2%)	N/A	N/A
libselinux	223 (58.3%)	48 KB (51.5%)	N/A	N/A
libreadline	N/A	N/A	89 (12.9%)	9 KB (7.9%)
libtinfo	N/A	N/A	109 (43.8%)	17 KB (34.7%)
libz	N/A	N/A	71 (58.7%)	22 KB (28.1%)

Table 2: Functions and code removed per library in Coreutils and SQLite.

4.1 Debloating

Tables 1 and 2 summarize debloating results with LibFilter. On average we are able to erase 590 KB of library code from Coreutils binaries and 595 KB from SQLite. In both of these cases, the majority of the erased code came from `libc`. In Coreutils, we find that we can erase 37% of bytes in `libc`, which corresponds to 77% of functions. We see the same phenomenon in the SQLite case. We find that roughly 25% of functions in `libc` in general-use applications are needed and make up 60-70% of the bytes in the `.text` section. This suggests that critical parts of `libc` used by all applications can be identified and placed in their own library, apart from all other functionality to help mitigate bloat.

Comparison with Nibbler. Nibbler decided to debloat a set of libraries for all Coreutils binaries, which limited the amount of which can be removed from each library because if even one binary used a function in a library, it would be left despite being unused in many other binaries. Nibbler was able to erase 58.8% of functions and 32.9% of code of libraries used by Coreutils. LibFilter debloats libraries per binary, instead of a set of binaries, and as a result is able to erase on average 76.5% of functions and 37.2% code of libraries used by Coreutils. While this is not a direct comparison because Nibbler debloats for a set of binaries, we achieve very similar code removal rates. It takes Nibbler 2 hours to analyze and erase all Coreutils binaries, whereas it takes LibFilter less than 20 minutes running on a Ubuntu 18.10 VM on a 4 core Macbook Pro with 8GB of RAM. This demonstrates the power of Egalito's analysis.

Comparison with Piecewise Debloating. Quach et al. reports that for Coreutils, their mean function reduction is 79%. LibFilter achieves a 76.5% reduction rate on the same metric. The piecewise approach is able to remove more functions because of the additional information that the compiler has about the usage of AT functions. Despite this, LibFilter is able to achieve a similar reduction, which working exclusively from the binary.

4.2 Memory Pressure

When using a vanilla library, it is only stored on disk once and loaded into memory once, shared by all applications using it. In Piecewise, only one version of a library is stored on disk and at load time, sections are debloated by erasing unused functions. The pages which include erased code to be copied (COW) because they are modified and no longer shared between processes. Under this approach, two applications using the same library running on the same machine would have the same library code paged in twice. LibFilter suffers from the same issue, as all binaries are debloated separately. However, not all of `libc` is paged in when in use, only the working set is, making this issue very difficult to measure accurately. The loading step in Piecewise causes a 20x performance hit in small applications such as Coreutils, as the erasing code during load time take much longer than running the application itself. LibFilter does not incur any load time performance overhead because all debloating takes place offline.

A potential avenue to mitigate the additional memory pressure that is created from loading the same libraries into memory is changing the way we erase functions from libraries. The reason we erased functions by overwriting instructions with `hlts` is for ease of implementation and ability to detect an attack. Instead, we can remove unreachable code completely from the library. To avoid gaps in the library, we would need to move functions to be contiguous in memory, updating all headers and relocations to be consistent. Egalito provides this functionality when recompiling binaries, however it is not stable enough yet to do so predictably. In the case of Coreutils, on average, we would be able to compress `libc` by 567 KB, or 141 4K pages. We would see this reduction for every

debloated application running which is linked against `libc`.

5 Conclusion

We presented LibFilter, a system utilizing Egalito to debloat shared libraries. When applications use shared libraries, they inadvertently add large amounts of code which can be used for ROP gadgets into the address space. We have shown that we can remove 37.2% of code from the address space of the average Coreutils application as an orthogonal avenue of defense against code reuse attacks. Debloated libraries do not contain code that is not reachable from the binary, which means our changes do not affect the functionality of programs. We are able to achieve code reduction rates close to compiler based approaches, indicating that source code is not strictly necessary to implement good security guarantees.

Acknowledgments

I would like to thank Vasileios P. Kemerlis for sparking my interest in computer security research, introducing me to the field, guiding me through this project, and giving me constant feedback and suggestions. Without you, I never would have started this project.

I also want to thank Kent Williams-King for your constant willingness to help me with debugging, testing, and implementing this project. I learned more about systems from our meetings than from many classes; your deep knowledge of Unix systems never ceases to amaze me.

I want to thank my parents for always being there for me. Thank you for everything you have sacrificed to bring me to this country and for supporting me through Brown. I am blessed and truly grateful to have such loving and amazing parents like you.

Lastly, I want to thank Alyssa for always keeping me motivated and focused on this project. You were there for me from start to finish. Thank you for encouraging me to pursue this project, keeping me motivated, and always being eager to listen to me ramble on about my research. Thank you for surprising me and coming to my defense. I am so lucky to have you in my life.

This project truly would not have been possible without all of you.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. *Proceedings of the 12th ACM conference on Computer and communications security (CCS)*, 2005.

[2] L. Davi, A. Sadeghi, D. Lehmann, and Fabian Monrose. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. *23rd USENIX Security Symposium*, 2014.

[3] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. Gross. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. *24th USENIX Security Symposium*, 2015.

[4] V. van der Veen, D. Andriesse, M. Stamatogiannakis, X. Chen, H. Bos, and C. Giuffrdia. The Dynamics of Innocent Flesh on the Bone: Code Reuse Ten Years Later. *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[5] Anonymous submission. Nibbler: Shared Library Debloating without Recompilation. 2019.

[6] Anonymous submission. Egalito: Program Hardening through Layout-Agnostic Binary Recompilation. 2019.

[7] H. Koo, Y. Chen, L. Lu, V. P. Kemerlis and M. Polychronakis, Compiler-assisted Code Randomization. *IEEE Symposium on Security and Privacy*, 2018.

[8] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal war in memory. *IEEE Symp. on Security and Privacy*, 2013.

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

[10] A. Tang, S. Sethumadhavan, and S. Stolfo, Heisenbyte: Thwarting memory disclosure attacks using destructive code reads. *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 256–267.

[11] D. Williams-King, G. Gobieski, K. Williams-King, J.P. Blake, X. Yuan, P. Colp, M. Zheng, V.P. Kemerlis, J. Yang, and W. Aiello. Shuffler: Fast and deployable continuous code rerandomization. *USENIX Symposium on Operating Systems Design and Implementation* (2016).

[12] D. L. C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, 2000, p. 168–177.

[13] Anh Quach, Aravind Prakash, and Lok Yan. Debloating Software through Piece-Wise Compilation and Loading. *Proceedings of the 27th USENIX Security Symposium*. 869–886. 2018