Configuration-Driven Software Debloating

Hyungjoon Koo
Stony Brook University
hykoo@cs.stonybrook.edu

Seyedhamed Ghavamnia
Stony Brook University
sghavamnia@cs.stonybrook.edu

Michalis Polychronakis
Stony Brook University
mikepo@cs.stonybrook.edu

ABSTRACT

With legitimate code becoming an attack surface due to the proliferation of code reuse attacks, software debloating is an effective mitigation that reduces the amount of instruction sequences that may be useful for an attacker, in addition to eliminating potentially exploitable bugs in the removed code. Existing debloating approaches either statically remove code that is guaranteed to not run (e.g., non-imported functions from shared libraries), or rely on profiling with realistic workloads to pinpoint and keep only the subset of code that was executed.

In this work, we explore an alternative *configuration-driven software debloating* approach that removes feature-specific code that is exclusively needed only when certain configuration directives are specified—which are often disabled by default. Using a semi-automated approach, our technique identifies libraries solely needed for the implementation of a particular functionality and maps them to certain configuration directives. Based on this mapping, feature-specific libraries are not loaded at all if their corresponding directives are disabled. The results of our experimental evaluation with Nginx, VSFTPD, and OpenSSH show that using the default configuration in each case, configuration-driven debloating can remove 77% of the code for Nginx, 53% for VSFTPD, and 20% for OpenSSH, which represent a significant attack surface reduction.

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

1 INTRODUCTION

Modern software development is greatly simplified by an abundance of freely available frameworks, toolkits, and libraries. Shared libraries, in particular, are widely used due to their several benefits, including increasing productivity by using ready-made third-party modules to carry out certain tasks, simplifying code maintenance and bug fixes without the need to redistribute the whole application, and reducing space by avoiding multiple copies of the same code on disk and in memory. The downside of this flexibility, however, is that the whole library must be loaded even if just a single of its functions is needed, resulting in "code bloat" due to a large amount of code that is present but never exercised.

EuroSec '19, March 25–28, 2019, Dresden, Germany

ACM ISBN 978-1-4503-6274-0/19/03…$15.00
https://doi.org/10.1145/3301417.3312501

Although a larger code base on its own may not be a significant drawback when considering the ample resources of modern computing devices (except, perhaps, embedded systems and resource-constrained devices), from a security perspective, the much larger attack surface is definitely not welcome. As the code base of a program grows, so does the likelihood of finding (exploitable) bugs. A larger code base also increases the odds of finding sufficient "gadgets" that can be strung together to mount return-oriented programming [15] or other types of code-reuse attacks. The inclusion of more libraries also implies more ways to access private or security-sensitive data, leveraging rarely used or unneeded functionality that is still present.

The above observations have given rise to software debloating techniques that aim to reduce the attack surface by eliminating unused code. For most applications, the bulk of the code comes from shared libraries, which are either bundled with the application, or are provided by the OS to expose system interfaces and services. Applications typically use only a fraction of the functions included in those general purpose libraries, so a natural approach to reduce the attack surface of a process is to remove unneeded (i.e., non-imported) functions from all loaded libraries [11, 12, 16]. Typically, the debloating process takes place on the endpoints, where both the executable binary and its dependent libraries are present, and all final dependencies have been resolved.

Besides library customization, prior works have explored various other debloating approaches applied at different levels, including function argument specialization [10], feature-driven customization [3, 19], and kernel customization [7, 9, 21]. Other software debloating approaches specialize code for specific languages or environments, including Java [6, 17, 18], mobile systems [1, 5], containers [13], or even network protocols [2, 20].

Most of the above approaches follow one of two main strategies for identifying the code to be removed: i) deterministically identifying code that is guaranteed to be unneeded, e.g., through static code analysis, or ii) profiling the application using representative workloads, and keeping only the exercised code. In this work, we explore an alternative strategy that relies on the *configuration* of an application for identifying code that will not be needed at runtime. The insight behind our approach stems from the fact that among the multitude of configuration options provided by feature-rich applications, some of them are rarely used and are disabled by default. In many cases, a significant amount of code implementing those features is not needed by any other component, and thus could be removed whenever the corresponding feature is disabled. Existing library customization approaches, however, cannot remove that code because control flow paths to it from other parts of the program (e.g., the configuration parser) are still present.

As a first step towards developing a fully automated system for configuration-driven debloating, in this work we aim to explore the attack surface reduction potential of this debloating strategy.

To that end, we present a semi-automated approach for identifying libraries solely needed for the implementation of a particular functionality that can be tied to certain configuration directives, and deriving a mapping between such directives and their exclusively used libraries. Based on this mapping, the main executable can then be instrumented to avoid loading any libraries for which the corresponding directives are disabled.

We applied our technique to the Nginx, VSFTPD, and OpenSSH servers, and identified various configuration directives that are associated with a large number of exclusively needed libraries—all of these directives are disabled by default, and most of them are rarely used in practice (e.g., XML transformation, image filtering). The results of our experimental evaluation show that using the default configuration in each case, configuration-driven debloating can remove 77% of the code for Nginx, 53% for VSFTPD, and 20% for OpenSSH, which represent a significant attack surface reduction.

2 BACKGROUND

Applications often allow users to specify initial settings, options, parameters, and other features by editing a separate configuration file (typically in ASCII format). The types of configuration directives vary across different programs. In general, a directive consists of a variable and a single or multiple values that can be assigned to it. Of particular interest for our purposes are directives associated with specific functionality that is carried out by a standalone library—when such a directive is disabled, then the corresponding library could be completely removed.

Listing 1 shows a complete instance of an Nginx configuration. Although Nginx supports 724 different configuration options from 85 components, a simple configuration like this suffices for a basic web service. The comments highlight directives that are associated with certain libraries. For example, the `gzip` directive at line 19 is associated with the `libz.so` library. As it is specified under the `server` structure, the `gzip` directive applies to all `location` sub–structures. Similarly, the `image_filter` and `rewrite` directives in lines 26 and 27 result in the loading of a graphic library (`libgd.so`) and regular expression library (`libpcre.so`), respectively. In some cases, multiple directives have to be defined to enable a certain capability, such as the SSL-related directives in lines 18, 20, and 21.

```
1  # /etc/nginx/nginx.conf
2  worker_processes  1;
3  error_log  /var/log/nginx/error.log;
4
5  events { worker_connections  1024; }
6
7  http {
8    include  mime.types;
9    index    default.html default.htm;
10   default_type application/octet-stream;
11
12   access_log    /usr/local/nginx/logs/nginx.pid;
13   geoip_country /usr/local/nginx/conf/GeoIP.dat; #
         libGeoIP.so
14   charset UTF-8;
15   keepalive_timeout 65;
16
17   server {
18     listen 443 ssl;                  # libssl.so
19     gzip   on;                       # libz.so
20     ssl_certificate  cert.pem;       # libssl.so
21     ssl_certificate_key cert.key;    # libssl.so
22
23     location / {
24       root   /var/www/hexlab;
25       index  default.php;
26       image_filter resize 150 100;   # libgd.so
27       rewrite ^(.*)$ /msie/$1 break; # libpcre.so
28     }
29
30     location /test {
31       xml_entities    /var/www/hexlab/entities.dtd; #
             libxml2.so
32       xslt_stylesheet /var/www/hexlab/one.xslt;     #
             libxslt.so
33     }
34   }
35 }
```

Listing 1: Example of an Nginx configuration file.

3 CONFIGURATION-DRIVEN CODE DEBLOATING

Removing the code that will remain unused according to a given configuration without breaking the functionality of the program requires addressing two main requirements. First, the code that is related to a particular configuration directive needs to be precisely identified. Second, the rest of the code must be analyzed to ensure that it does not depend on the code that will be removed if that particular directive is disabled. Both of the above requirements are quite challenging to address in an automated and exhaustive way. An ideal approach would take a configuration directive as input, automatically identify all associated code, and extract the subset of that code that is not needed by the rest of the program when this particular functionality is disabled. This may be feasible using a combination of control and data flow analysis, but before investigating such a complex solution, our goal in this work is to derive a first estimate of the attack surface reduction potential that such a configuration-driven debloating scheme would offer.

Unneeded code removal can be performed at different levels of granularity, e.g., at the instruction, function, or library level. For instance, prior works remove the functions that are not imported (and thus not used) from the libraries linked to the main executable [11, 12]. Given the complexity of identifying all functions that are exclusively needed by a given configuration directive, in this work we decided instead to aim for deriving a lower bound, and perform code debloating at the *library* level. The intuition behind this decision is that many types of functionality that can be enabled or disabled through configuration directives are often carried out by third-party libraries. For instance, as shown in the Nginx configuration example of Section 2, if content compression is needed, then this will be performed by the `libz.so` library.

To pinpoint the libraries that are exclusively associated with a given configuration directive, we perform differential testing using a combination of static and dynamic analysis. The directives to be analyzed, as well as appropriate test inputs for driving the execution of the application during dynamic analysis, are manually selected after studying the configuration documentation of the application in conjunction with observing which libraries are loaded. For this work, we focused on server applications (Nginx, VSFTPD, and OpenSSH), as they typically require at least some minimal configuration specification for proper operation.

Figure 1 shows an overview of our approach. First, we compile the program by enabling code coverage profiling, and run it twice, with a given directive enabled and disabled. By comparing the two code coverage reports, we then pinpoint any extra library code that

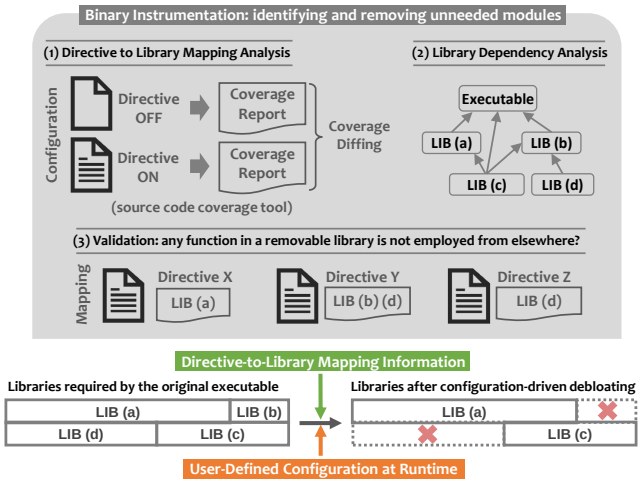

Figure 1: Overview of the configuration-driven code debloating process.

was exercised only when the directive was enabled. This allows us to derive an initial mapping between directives and libraries, which is then refined in a second dependency analysis step, which builds the dependency graph across all libraries in the program and identifies any dependencies that are exclusive to a given library. Finally, a static analysis step analyzes the whole program and verifies that the identified directive-dependent libraries are not used by any other part of the program. The whole analysis process is performed only once per configuration directive. The resulting directive-to-library mapping information can then be used as input for instrumenting the main executable to avoid loading any unneeded libraries for which the corresponding directive is disabled.

3.1 Mapping Directives to Libraries

To identify the libraries that are exclusively used by certain configuration directives, we perform differential testing by comparing the source code coverage during execution with and without a given directive. Our technique relies on the LLVM source code coverage tool (`llvm-cov` [8]), to identify the exercised code for a given combination of configuration directives and test inputs.

Before testing a given directive, we have to manually prepare i) the specially crafted configuration file that enables or disables the functionality of interest, and ii) a set of appropriate program inputs to ensure that the feature of interest will be invoked. We initially specify the simplest configuration with all directives to be tested disabled (i.e., commented out) as our base configuration. We then generate one configuration per directive (or set of directives) that enables a particular feature or functionality that is likely to be carried out by a specific library (or set of libraries).

We have implemented an analysis tool that automatically enables and disables the directive(s) for a given feature, runs the application with the appropriate test cases, and maps the directive(s) into one or more libraries. Comparing the two code coverage reports allows us to pinpoint the extra code that is associated with the tested functionality when the corresponding directive is enabled, and thus the associated libraries. For example, consider `libgd.so`, which is used by Nginx for image manipulation—a functionality that is supported by Nginx, but is *disabled* by default. When Nginx runs without this feature (the default case), we can safely exclude `libgd.so` from being loaded, as no other part of the code relies on it.

3.2 Library Dependence and Validation

The list of candidate libraries for removal from the mapping phase must be validated by checking whether i) other libraries have any dependencies from the candidate directive-related libraries, and ii) the rest of the program still uses any functions from the candidate directive-related libraries.

The first case can be easily handled by statically analyzing the imports of the main executable and all dynamic libraries. By building a library dependence graph, we can identify additional libraries that may be needed solely by a directive-dependent library, which can then be removed as well. For example, going back to Figure 1, if `libB` is directive-dependent, then `libD` can also be removed when the directive is disabled, as it is not needed by any other module. On the other hand, `libC` cannot be removed because it is still needed by other libraries. The second case is handled by identifying all exported functions of a directive-related library, and checking whether any of them are used by other parts of the source code. For instance, although differential analysis shows that the `gzip` directive depends on `libz.so`, we cannot simply remove it because a function from `libz.so` is used by other parts of the code.

4 EXPERIMENTAL EVALUATION

We evaluated the potential of configuration-driven code debloating by experimenting with three widely used server applications (Nginx, VSFTPD, and OpenSSH) running on Ubuntu 16.04. Our aim is to explore the impact of various features *that are disabled by default* on code bloat, i.e., how much code could be removed when a given feature is disabled in a certain configuration. In addition, we also compare configuration-driven code debloating with the alternative (and orthogonal) approach of code debloating based on removing any non-imported functions from libraries [11, 12].

4.1 Identifying Non-default Functionality

One way to identify promising features that are disabled by default and which may be associated with libraries that are loaded but remain unused would be to go through the various configuration directives and pinpoint the ones that seem promising. However, as discussed in Section 2, the large amount of configuration directives for some applications would make this approach quite time consuming (testing *all* directives would be even more so). Instead, we followed the opposite approach and went through the loaded libraries of each application, trying to identify those that seem specific to a certain functionality, and then looking for related configuration directives in the documentation. Once the directives corresponding to a given library are identified, a special configuration can be specified in our analysis tool to run the application with pre-defined test cases (Section 3.1).

One of the libraries Nginx loads (in its default installation, e.g., when installed by a package manager like `apt-get`) is `libGeoIP.so`, which provides code for mapping IP addresses to their geographic

Table 1: Libraries exclusively used by certain (disabled by default) features, and their corresponding footprint in terms of code size and ROP gadgets, for three server applications.

Program	Functionality	Libraries	# Functions	Code (bytes)	# Gadgets
Nginx	(Whole)	libdl.so.2, libpthread.so.0, libcrypt.so.1, libpcre.so.3*, libz.so.1*, libc.so.6, libssl.so.1.0.0, libcrypto.so.1.0.0, and all libraries from other features	38,712	13,853,649	150,914
	GeoIP	libGeoIP.so.1*	386	137,663	460
	XSLT	libxml2.so.2*, libxslt.so.1*, libexslt.so.0*, libm.so.6, libicui18n.so.55*, libicuuc.so.55*, libicudata.so.55, libstdc++.so.6	20,066	5,766,222	55,595
	Image filtering	libgcc_s.so.1, libgd.so.3*, libjpeg.so.8, libpng12.so.0*, libfreetype.so.6*, libxcb.so.1*, libfontconfig.so.1*, libXpm.so.4*, libX11.so.6, libvpx.so.3, libtiff.so.5, libexpat.so.1*, liblzma.so.5*, libjbig.so.0*, libXau.so.6*, libXdmcp.so.6*	9,240	4,800,102	49,286
VSFTPD	(Whole)	libcrypt.so.1, libc.so.6, libcap.so.0*, and all libraries from other features	9,005	2,993,778	44,221
	SSL	libssl.so.1.0.0, libcrypto.so.1.0.0	5,427	1,457,041	21,668
	PAM	libpam.so.0, libaudit.so.1*, libdl.so.2	211	65,294	1,008
	TCP wrapper	libwrap.so.0*, libnsl.so.1	230	69,169	1,175
OpenSSH	(Whole)	libcrypt.so.1, libdl.so.2, libcrypto.so.1.0.0, libutil.so.1, libresolv.so.2, libz.so.1*, libc.so.6, libgcrypt.so.20, libselinux.so.1, libsystemd.so.0, libgpg-error.so.0*, librt.so.1, liblzma.so.5*, libpcre.so.3*, libaudit.so.1*, and all libraries from other features	15,563	5,341,425	68,914
	Kerberos	libgssapi_krb5.so.2*, libkrb5.so.3*, libk5crypto.so.3*, libkrb5support.so.0*, libcom_err.so.2*, libpthread.so.0	3,823	1,043,336	10,740
	PAM	libpam.so.0	75	29,920	429

locations. We can easily assume that there may be a configuration directive related to geolocation, and we indeed identified three related directives (`geoip_city`, `geoip_country`, and `geo_org`). Similarly, the presence of `libxslt.so` is related to the `xslt_stylesheet` directive, as shown in Listing 1.

Although we begin with a single library per directive (or set of directives), it is often the case that a directive-dependent library exclusively relies on other libraries that are not used by other parts of the program, which our analysis identifies as well. For example, `libgd.so` for image filtering in Nginx subsequently loads 16 more libraries, such as `libpng12.so`, `libtiff.so`, and `libjpeg.so`. By following this approach, we identified three main features (GeoIP, XSLT, and image filtering) which are disabled by default, and account for the *vast majority* of loaded libraries of a default Nginx installation. In particular, among the 33 libraries loaded by default, 25 are solely required for the above three features—*just eight libraries are really needed* when none of those features are enabled. The left part of Table 1 (first three columns) summarizes the types of functionality that depend on certain directives, and the corresponding libraries that are exclusively required by them, for the three applications we tested, as a result of our directive-to-mapping analysis described in Section 3.

For Nginx, we could identify all libraries for the different directives without any particular test traffic, i.e., by simply starting and stopping the web server with each configuration. For the other two applications, we had to generate realistic traffic, including a complete authentication and log in process, because features related to authentication (e.e., PAM, TCP wrapper, Kerberos) require an actual login attempt to generate a meaningful code coverage report.

4.2 Attack Surface Reduction

To get a better insight on the degree of the achieved attack surface reduction, given that the code size of libraries varies widely, we provide more detailed information about the amount of code, number of functions, and number of ROP gadgets that are removed for each feature, as well as for the original program (last three columns in Table 1). We used ROPGadget [14] with its default options to discover the available ROP gadgets in each module. Note that two common libraries, the virtual dynamic shared object (`linux-vdso.so`) and the dynamic loader (`ld-linux-x86_64.so`), are omitted from the table due to their small size.

The rows denoted as "whole" in the Functionality column correspond to the original (non-debloated) binary that is typically distributed by the various Linux distributions, i.e., which contains the whole functionality that can potentially be needed by all supported configurations. For example, a default Nginx process comprises 38,712 functions across 33 libraries, which correspond to approximately 14MB of code containing around 150,914 ROP gadgets.

The rest of the rows for each application correspond only to the libraries exclusively needed for a given functionality—all listed functionalities are disabled by default. Notably, the XSLT feature of Nginx alone requires 5.7MB of code—when the whole code base of Nginx is 13.8MB—while image filtering requires 4.8MB of code. As shown in the pie chart of Figure 2, XSLT and image filtering correspond to 41% and 35% of the code. When all three features are disabled (which is very likely to be the case in many configurations), configuration-driven debloating can reduce Nginx's code to just 23% of the original. The reduced code for VSFTPD and OpenSSH with their default configurations is 47% and 80% of the original, respectively. The reduction for OpenSSH is not that significant, as just Kerberos corresponds to a significant fraction of the code (about one fifth).

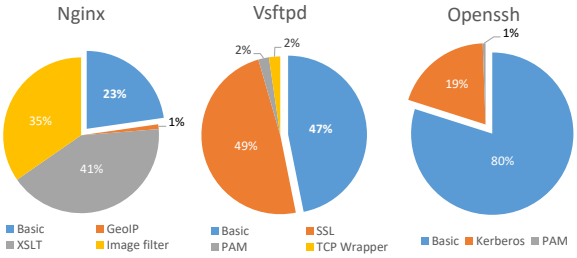

Figure 2: Breakdown of code size according to different configuration directives. "Basic" corresponds to the remaining code after configuration-driven debloating when all directives are disabled, which is the default in all cases.

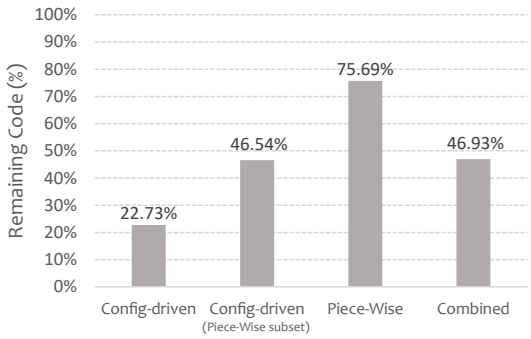

Figure 3: Remaining code for Nginx for different debloating approaches (configuration-driven, Piece-Wise [12], and their combination).

4.3 Comparison with Library Customization

In this section, we compare configuration-driven debloating with the alternative—and orthogonal—debloating approach of library customization [11, 12, 16]. Library customization works by statically analyzing the code of the application to identify which functions are imported (i.e., actually used) from shared libraries, and then remove the rest. We use the Piece-Wise Compilation implementation [12] as a representative library customization technique. For this set of experiments, we exclude `libc`, the libraries defined under `libc` (i.e., `libcrypt`, `libpthread`, `libm`, `libdl`, and `libcrypt`) and several cryptographic libraries (i.e., `libcrypto`, `libssl`, and `libgcrypt`) because Piece-Wise's modified LLVM compiler could not successfully compile them. The remaining libraries that were successfully processed (33 out of a total of 67 libraries for all three applications) are marked with an asterisk (*) in Table 1.

Figure 3 shows the remaining code for Nginx using its default configuration for i) configuration-driven debloating, ii) the same when considering only the libraries that can be successfully handled by the Piece-Wise compiler, iii) Piece-Wise compilation, and iv) the combination of the two approaches (i.e., Piece-Wise applied after configuration-driven debloating has removed the non-needed libraries). The second case is provided for a more fair comparison with Piece-Wise debloating, which shows that for Nginx, library specialization alone cannot reach the level of reduction achieved by configuration-driven debloating. Although the combination of both approaches in this case offers only a small benefit (<1%), it may be beneficial for other applications. We could not meaningfully perform the same comparison for VSFTPD and OpenSSH because Piece-Wise could only process less than half of the libraries (mostly the very small ones), which collectively do not represent a substantial amount of the whole code.

5 DISCUSSION AND LIMITATIONS

Our current implementation requires the source code of the application to collect code coverage information during the profiling phase. The reliance on source code means that the technique is not applicable on close-source software, while the profiling phase requires appropriate inputs to exercise the corresponding code paths, which may result in missed functionality, and entails a fair amount of manual preparation. For example, exercising the code for the PAM functionality in OpenSSH required realistic interaction with the server, including proper user authentication.

Our library dependence analysis and validation steps mitigate this issue, but a more principled approach may be possible by combining code and data flow analysis techniques, which we leave as part of our future work. Another aspect that currently involves manual analysis is the identification of particular configuration directives that seem promising enough to analyze. A fully automated approach would be capable of exhaustively analyzing all directives, and even certain directive combinations.

A drawback of relying on source code coverage is that its information may not be entirely accurate. Based on our experience with the LLVM source coverage tool, the coverage report is not generated properly in cases of some forking applications. In particular, when the code uses `_exit()` instead of `exit()`, the tool fails to catch the termination of the process. Therefore, we had to modify the source code as a workaround for both VSFTPD and OpenSSH. In addition, the environment variable, `LLVM_PROFILE_FILE` was not propagated to forked processes in OpenSSH, which resulted in empty report files. We resorted to running OpenSSH in debug mode, which disables forking, to extract proper coverage information.

It is worth mentioning that we excluded Apache from our evaluation because its modular design is directly exposed to the configuration file. Enabling a certain feature is performed by actually specifying the precise path to the corresponding shared library implementing that feature in the configuration.

6 RELATED WORK

One of the earliest library specialization approaches as a defense against exploitation was presented by Mulliner and Neugschwandtner [11]. Their Code Stripping and Image Freezing techniques, which operate on closed-source binaries, identify and remove all non-imported functions at load time, and then "freeze" the remaining code by modifying certain memory allocation routines to prevent the loading or injection of additional code. Quach et al. [12] proposed Piece-Wise compilation, which leverages a modified compiler and loader to perform shared library specialization. Information regarding call dependencies and function boundaries is embedded as metadata into the binary at compilation time. Song et al. [16] showed the potential of fine-grained library customization

for statically linked libraries using data dependency analysis. Shredder [10] aims to specialize further any remaining functions after applying one of the above library specialization approaches. This is achieved by restricting the scope of critical system API functions and allowing only the subset of argument values that are needed by the benign code.

Feature-oriented software specialization aims to remove unused functionality across the whole program, depending on its intended use. DamGate [19] uses both static and dynamic analysis to construct a call graph according to a set of seed functions that are given as input, and pinpoint the required code. TRIMMER [3] relies on user-defined configuration data to remove unneeded feature-related code. The code is identified using inter-procedural analysis based on the entry points specified in the initial configuration. CHISEL [4] proposes a reinforcement-learning-based approach that allows a developer to generate a reduced version of a program based on a set of example runs with the desired options.

Another line of code debloating research focuses on the kernel. FACE-CHANGE [21] generates a customized kernel view for each application to reduce the exposed kernel code based on profiling. Kurmus et al. [7] presented an automated approach for generating kernel configurations adapted to particular workloads, which can be used to compile a specialized kernel tailored for a given use case.

Other types of code specialization focus on different languages [6, 17, 18], environments [1, 5, 13], or protocols [2, 20]. Jred [18] removes unused methods and libraries based on static analysis of Java code. Similarly, Bhattacharya et al. [17] introduced a technique to detect bloated sources in Java applications. Jiang et al. [6] presented a technique for Java bytecode customization using static data flow analysis and programming slicing. RedDroid [5] eliminates unneeded methods and classes from Android apps. Apple introduced "app thinning" [1] to deliver optimized versions of apps for different devices. Cimplifier [13] performs container debloating based on system call analysis. David et al. [2] observed that vulnerabilities related to protocol implementations often reside in code that is not frequently used. TOSS [20] is an approach for automated customization of client–server systems through the removal of code related to unneeded network protocols.

7 CONCLUSION

We have presented configuration-driven software debloating, an approach that removes feature-specific shared libraries that are exclusively needed only when certain configuration directives are specified by the user, and which are typically disabled by default. Although we have identified several challenges that must be addressed for developing a fully automated configuration-driven debloating solution, our current semi-automated approach still demonstrates that the level of attack surface reduction for certain server applications is worth the effort—our results show that only 23% of the code for Nginx, 47% for VSFTPD, and 80% for OpenSSH is really required based on their default configuration. At the same time, the technique can be combined with other code debloating approaches, such as library customization.

Acknowledgments. This work was supported by the Office of Naval Research (ONR) through award N00014-17-1-2891, the National Science Foundation (NSF) through award CNS-1749895, and the Defense Advanced Research Projects Agency (DARPA) through award D18AP00045, with additional support by Accenture. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the ONR, NSF, DARPA, or Accenture.

REFERENCES

[1] Apple. 2015. What is app thinning? (iOS, tvOS, watchOS). https://help.apple.com/xcode/mac/current/#/devbbdc5ce4f.
[2] Q. A. Chen David K. Hong and Z. M. Mao. 2017. An initial investigation of protocol customization. In *Proceedings of the Second Workshop on Forming an Ecosystem Around Software Transformation (FEAST)*.
[3] Ashish Gehani Hashim Sharif, Muhammad Abubakar and Fareed Zaffar. 2018. TRIMMER: Application Specialization for Code Debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*.
[4] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective Program Debloating via Reinforcement Learning. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*.
[5] Yufei Jiang, Qinkun Bao, Shuai Wang, Xiao Liu, and Dinghao Wu. 2018. RedDroid: Android Application Redundancy Customization Based on Static Analysis. In *Proceedings of the 29th IEEE International Symposium on Software Reliability Engineering (ISSRE)*.
[6] Yufei Jiang, Can Zhang, Dinghao Wu, and Peng Liu. 2016. Feature-Based Software Customization: Preliminary Analysis, Formalization, and Methods. In *Proceedings of the 17th IEEE International Symposium on High Assurance Systems Engineering (HASE)*.
[7] Anil Kurmus, Reinhard Tartler, Daniela Dorneanu, Bernhard Heinloth, Valentin Rothberg, Andreas Ruprecht, Wolfgang Schroder-Preikschat, Daniel Lohmann, and Rudiger Kapitza. 2013. Attack Surface Metrics and Automated Compile-Time OS Kernel Tailoring. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.
[8] LLVM. 2008. Source-based Code Coverage. https://clang.llvm.org/docs/SourceBasedCodeCoverage.html.
[9] Hyungon Moon Mansour Alharthi, Hong Hu and Taesoo Kim. 2018. On the Effectiveness of Kernel Debloating via Compile-time Configuration. In *Proceedings of the Second Workshop on Forming an Ecosystem Around Software Transformation (FEAST)*.
[10] Shachee Mishra and Michalis Polychronakis. 2018. Shredder: Breaking Exploits through API Specialization. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*.
[11] Collin Mulliner and Matthias Neugschwandtner. 2015. Breaking Payloads with Runtime Code Stripping and Image Freezing.
[12] Anh Quach, Aravind Prakash, and Lok Yan. 2018. Debloating Software through Piece-Wise Compilation and Loading. In *Proceedings of the 27th USENIX Security Symposium*. 869–886.
[13] Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick D. McDaniel. 2017. Cimplifier: automatically debloating containers. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*.
[14] Jonathan Salwan. 2011. ROPGadget - Gadgets finder and auto-roper. http://shell-storm.org/project/ROPgadget/.
[15] Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and Communications security (CCS)*.
[16] Linhai Song and Xinyu Xing. 2018. Fine-Grained Library Customization. In *Proceedings of the ECOOP 1st International Workshop on SoftwAre debLoating And Delayering (SALAD)*.
[17] Kanchi Gopinath Suparna Bhattacharya and Mangala Gowri Nanda. 2013. Combining Concern Input with Program Analysis for Bloat Detection. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*.
[18] Dinghao Wu Yufei Jiang and Peng Liu. 2016. Jred: Program customization and bloatware mitigation based on static analysis. In *Proceedings of the 40th Annual Computer Software and Applications Conference (ACSAC)*.
[19] Tian Lan Yurong Chen and Guru Venkataramani. 2017. DamGate: Dynamic Adaptive Multi-feature Gating in Program Binaries. In *Proceedings of the Second Workshop on Forming an Ecosystem Around Software Transformation (FEAST)*.
[20] Tian Lan Yurong Chen, Shaowen Sun and Guru Venkataramani. 2018. TOSS: Tailoring Online Server Systems through Binary Feature Customization. In *Proceedings of the Second Workshop on Forming an Ecosystem Around Software Transformation (FEAST)*.
[21] Xiangyu Zhang Zhongshu Gu, Brendan Saltaformaggio and Dongyan Xu. 2014. FACE-CHANGE: Application-Driven Dynamic Kernel View Switching in a Virtual Machine. In *Proceedings of the 44th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*.