

Attack Surface Reduction through Software Debloating and Specialization

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

Abstract—Attacks and defenses in software systems continue to grow. While practical defenses have evolved, attackers are still able to bypass them. The growth of software complexity has added to the complication. While best practices in software engineering suggest the reuse of existent code, they are unknowingly increasing the attack surface. Other than applications, the Linux kernel code base has also undergone a large increase in size throughout the years. Although this increase has added features and functionalities, it has also enlarged the attack surface and complicated the application of defense mechanisms.

Attack surface reduction through software debloating has recently gathered attention as an orthogonal defense mechanism to help reduce the risks of software vulnerabilities. While it does not provide a new protection for software, it can reduce the attacker’s ability in successfully mounting an attack. In this work, we first thoroughly review recent work done on attack surface reduction and then provide insight on parts which can be improved.

I. INTRODUCTION

There has been a long history of challenges regarding memory corruption attacks. As Szekeres et al. [1] call it, the “Eternal War in Memory.” While practical and effective defenses have evolved to prevent the exploitation of memory vulnerabilities, attackers have also enhanced their game. They moved away from inserting their own malicious code, to using existent code in the vulnerable application after DEP [2] was introduced. ASLR [3] was designed to randomize the placement of code in memory on each run, to prevent the attacker from jumping to known addresses. But since the default coarse-grained randomization only randomizes the base address and the placement of the rest of the code does not change it was easily bypassed after one address had been leaked. CFI [4] was proposed to defend against any type of control flow attack by keeping track of valid caller and callee addresses. Due to the high overhead, the initial implementations only performed coarse-grained CFI which only guaranteed target addresses of jumps to be valid and didn’t have any protection over the callers or returns. Many variations have followed paying attention to both sides of trying to bypass the proposed solutions [5], [6], [7] and improving the defense with a better overhead [8], [9], [10], [11]. Furthermore data-only attacks [12], [13] which focus on non-control-flow attacks proved to be both as effective and

at the same time as difficult to mitigate as control-flow attacks. While different defense mechanisms have been proposed [14], [15], they still remain an open problem.

The growth of software complexity and size has been one of the reasons which complicates protecting against memory corruptions. Best practices in software engineering suggest reusing existent code usually in the form of libraries. This has led to programs importing an entire library along with all its vulnerabilities into their address space just to use one small function.

At the same time, the code base of the Linux kernel has been expanding to support new features, protocols, and hardware. The increase in the number of exposed system calls throughout the years is indicative of the kernel’s code “bloat.” The first version of the Linux kernel just had 126 system calls, whereas the most recent version supports 326 system calls. As shown in previous works [16], [17], [18], [19], different applications leverage disparate kernel features, leaving the rest unused—and available to be potentially exploited by attackers. Kurmus et al. [17] showed that each new kernel function is an entry point to accessing a large part of the whole code, which leads to attack surface expansion.

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 [20] 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.

As a countermeasure to the ever expanding code base of modern software, *attack surface reduction* techniques have recently started gaining traction. The main idea behind these techniques is to identify and remove (or neutralize) code that is either i) completely inaccessible (e.g., non-imported functions from shared libraries), or ii) not needed for a given workload or configuration. A wide range of previous works have applied this concept at different levels, including removing unused functions

from shared libraries [21], [22], [23] or even removing whole unneeded libraries [24]; tailoring kernel code based on application requirements [18], [17]; or limiting system calls for containers [25], [26], [27], [28]. In fact, one of the suggestions in the NIST container security guidelines [29] is to reduce the attack surface by limiting the functionality available to containers.

Despite their diverse nature, a common underlying challenge shared by all these approaches is how to accurately identify and maximize the code that can be *safely* removed. On one end of the spectrum, works based on static code analysis follow a more conservative approach, and opt for maintaining compatibility in the expense of not removing all the code that is actually unneeded (i.e., “remove what is not needed”). In contrast, some works rely on dynamic analysis and training [17], [18], [25], [26], [27], [28] to exercise the system using realistic workloads, and identify the actual code that was executed while discarding the rest. (i.e., “keep what is needed”). For a given workload, this approach maximizes the code that can be removed, but does not capture exhaustively all the code that can *potentially* be needed by different workloads—let alone parts of code that are executed rarely, such as error handling routines.

In this report, we present a review of attack surface reduction. We further categorize different research done in this area based on their techniques and their target software stack.

II. BACKGROUND

A. Process Loader

The operating system uses a special program named the *loader* to start running a program. The loader is in charge of requesting the memory pages from the kernel and copying the binary to the code section of those pages. It also loads dynamic libraries required by the binary. Some attack surface reduction techniques modify the loader to modify its process and remove parts of the code or libraries from being loaded.

B. Binary Analysis











The broad field of binary analysis has been of major interest throughout the years. Extracting characteristics about the program and then modifying it in the next step have been challenging tasks which have drawn many research projects in trying to solve them.

Attack surface reduction can be done at the binary level. In this case the proposed technique must rely on binary analysis to identify segments of code which are not necessary. Doing so increases the usability of the approach, but might reduce the precision due to higher inaccuracy compared to source code analysis.

C. Seccomp System Call Filtering

Seccomp is a system call filtering mechanism built in the Linux kernel since version 2.6.12. The initial version

Table I: Proposed taxonomy applied to previous work on software debloating and specialization.

Proposed Technique	Src. or Bin.	Bloat Identification	Target	Security Benefit	Remove or Restrict
Temporal		Static Analysis		Shellcodes, Kernel CVEs	Restrict Access to System Calls
Confine		Static Analysis		Shellcodes, Kernel CVEs	Restrict Access to System Calls
Face Change		Dynamic Analysis		CVEs	Remove Code
Less is More		Dynamic Analysis		CVEs	Remove Code
Razor		Dynamic Analysis		Shellcodes, Kernel CVEs	Restrict Access to System Calls

did not provide much flexibility only allowing four system calls, `read`, `write`, `exit` and `sigreturn` to be accessed to a target application. Support for using Berkeley Packet Filters (BPF) to define and apply system call filters was added in to the kernel in version 3.5. This increased the flexibility of applying system call filtering policies significantly by allowing the user to limit access to any system call based on its name and the arguments passed to it.

Some proposed software specialization approaches [30], [31], [32] use Seccomp BPF to install filters and restrict the set of available system calls to a running process.

III. SOFTWARE SPECIALIZATION TAXONOMY

In this paper we will present a classification of different software debloating and specialization techniques. This will help show the directions the research community has already visited and which require more attention. In this section we will briefly introduce the different categories we have considered to classify previously proposed work.

IV. ACCESS TO SOURCE CODE

One of the aspects of software specialization techniques which differentiates them is whether they require the target application source code or not. While working at the binary level increases compatibility and deployability, it tends to have less precision and if not implemented correctly might lead to crashes in the software. If we suppose the proposed binary analysis framework does not have major issues and is sound, still, they can typically remove less code due compared to compiler-based approaches.

A. Compiler-based Approaches

Approaches which typically need the target application or library to be recompiled, fall in this category. These

techniques are usually implemented as a plugin in the compiler and modify the final binary to remove parts of the code or restrict its access to specific features.

The major limitation of these techniques is both having the source code available and adding the requirement of compatibility with the specific compiler. On the other hand most of the recent proposed techniques [31], [30], [21] use the LLVM [33] compiler which has large community support and is widely adopted across the industry (e.g. Google [34] uses LLVM to compile Chrome).

B. Binary-based Approaches

Techniques which do not require the source code of the target application or library fall into this category. While some approaches rely on a binary analysis framework to generate call graphs or control flow graphs which they further use to reason about code bloat, others only create traces of the executed code and use learning techniques to enrich it. For example Nibbler [35] and Sysfilter [32] rely on Egalito [36] and Razor [37] relies on their own PathFinder component for CFG construction.

V. BLOAT IDENTIFICATION

The first step in performing software debloating is identifying the code segments which can be removed in a *safe* and *sound* manner, without breaking the target application. In this section we review two main approaches used to identify bloated parts of target applications.

A. Static Analysis

Static analysis is a broad term used for assessing specific properties of a program without having to run it. Some of the main properties debloating techniques rely upon include, performing data flow analysis or generating the call graph and control flow graph. Analyzing an application without actually executing it tends to be a complicated problem. We will not go into details of the difficulties of performing static analysis, but will provide different techniques used in previous work which fall under the category of static analysis.

1) *Function Call Graph*: The call graph shows how functions depend on each other for their functionality. The nodes of the graph are functions and the edges are direct or indirect callsites which target another function. Generating the call graph is one of the major steps required in many of the previous work [30], [21], [31].

a) *Points-to Analysis*: Points-to analysis algorithms can be used to statically identify where data and code pointers actually point to. Due to the nature of static analysis this is a time-consuming task which lacks scalability and the results are overapproximated. The benefit of using points-to analysis is that the algorithms are *sound*. Examples include Andersen [38] and Steensgaard [39].

SVF [40] is an open source tool which implements Andersen's points-to analysis algorithm and works on the LLVM IR. Some of the recent work, such as piece-wise [21]

and temporal system call specialization [31] use this tool to generate the call graph for applications and their libraries. We will provide more details on how each of these two papers generate their call graph and what changes they have made to SVF.

Piece-wise Quach et al. [21] developed a modified loader and compiler to perform shared library specialization by removing unnecessary functions extracted through call dependency and function boundary identification at compile time. On top of using LLVM's call-graph analysis pass which only extracts dependencies among functions due to direct calls between them, they apply three other methods to extract indirect calls.

- **Full-Module code pointer scan**: In this approach, Piece-wise scans the entire module for code pointers and labels functions referenced by them as necessary for all the functions in that module. Although it's not the most optimal solution, it can provide an approximation in an efficient manner of time.
- **Localized code pointer scan**: This approach is similar to the previous in extracting all functions referenced by code pointers in the module, but only the ones which have their address taken at runtime are marked as necessary.
- **Pointer analysis**: They use the recently proposed algorithm for performing pointer analysis, named SVF [40] for this approach.

Using these three approaches they create a full call-graph for each library and use it at load time to remove unnecessary functions based upon each application's requirements. Although they have claimed to successfully debloat multiple unmodified executables, such as Firefox, Git and OpenOffice programs they only present detailed numbers for Curl. On average they can remove 40% of the instructions using the localized code pointer scanner approach, which is their most aggressive.

Temporal Ghavamnia et al. [31] identified that a more limited set of system calls are required in different phases of a server application's execution. They split the execution into two main phases of initialization and serving phase. They also used points-to analysis and the SVF [40] tool to generate a call graph for each application and its relevant libraries. They improved the call graph precision by adding two pruning techniques, which used the argument types along with inaccessible function pointer initializers as methods of identifying spurious edges on the call graph.

b) *Type-based Call Graph Generation*: Talk about Sea-DSA and Tea-DSA and all. No one has used these techniques, neither has anyone compared their performance.

2) *Control Flow Graph*: The control flow graph (CFG) shows the flow of execution among the different basic blocks of a program. Razor [37] is an example of a debloating technique which attempts to generate the CFG

from the binary, with no access to the source code. We have discussed Razor in Section V-B.

B. Test-driven Identification

We consider any research which needs to execute the application to identify unnecessary code sections in this category. These techniques generally have a training phase where specific test cases are used to execute the target application [41], [37], [42]. During these test cases specific features of the program are tested. Depending on the underlying technique, the remove unnecessary code based on these test cases.

1) *Dynamic Analysis*: While the use of dynamic analysis can provide an initial estimate on the amount of code used in applications and the kernel, it cannot be relied upon for actually removing code. This is mainly because programs have code paths which are only executed under certain circumstances. Ghavamnia et. al [30] have shown different examples in Nginx [43], where certain parts of the code are required but only after the program serves a configurable amount of requests. They show how the caching feature of Nginx deletes files after they occupy disk space exceeding the configured threshold. We need a deep understanding of the program and how it works semantically to cause it to enter those specific code paths. This is not a simple task and requires manual effort.

The fact that using dynamic analysis for identifying the required part of an application’s code is not a sound approach, does not rule out using them entirely. Under certain circumstances, where there is a high guarantee that the test cases used in the training phase correspond to the real world usage, this might be an acceptable approach. Combining the results of dynamic analysis with a sound static analysis can also be used to increase the reliability of the approach. We present some related works which do the same to some extent in the following section.

a) *Extending Dynamic Analysis with Heuristics*: Razor [37] uses dynamic analysis and training along with hardware tracing facilities (e.g. Intel PT [44]) to identify basic blocks which are executed when running the specified test cases. While the traces are used as their main bloat identification tool, they add other basic blocks which they presume will be required by four different heuristics, to their generated control flow graph. This significantly improves their results. While they do not use any static analysis to identify any other basic blocks required, just increasing the covered code by simple heuristics improves their performance.

2) *Reinforcement Learning*: While Chisel [42] also uses test cases to specify required features of the target program, but unlike Razor [37] it does not track the executed parts of the code and remove the rest. It applies reinforcement learning to identify which parts of the code are required to fulfill the set of test cases. Furthermore it applies a Markov decision process (MDP) to guide the algorithm in identifying which sections it should remove

to reach a correct working binary, which fulfills the test case requirements in a timely manner.

They perform debloating at the source code level, removing lines of code which do not conform to the requested test case specification. Their evaluation shows that debloating `tar` takes 12 hours and can remove more than 96% of the line of code.

VI. DEBLOATING TARGET

It has been shown in previous work [45] that most software vulnerabilities are located in parts of the code which aren’t executed. Based on their evaluation, 83% of the vulnerabilities in their dataset were located in *cold code*. This shows that removing unnecessary code can have a significant effect on reducing the attack surface. Attack surface reduction through code removal or filtering features has been applied at different levels. While some have removed extra code from applications [21], [46], [24], others have tried to reduce the attack surface of the kernel [18], [47]. There has also been attempts to shrink containers [26] and remove extra kernel functionalities through profiling the container requirements [25]. We have taken a more detailed look at each category in the following sections.

A. Application Debloating

Most of the previous work regarding debloating has been done at the application level targeting single applications. These works focus on removing unnecessary code from their process space.

Quach et al. [48] have performed dynamic analysis on multiple applications and the Linux kernel to provide an initial estimate on the amount of bloat in both the programs and the kernel. They have observed that on average only 21% of code was actually executed for the programs in their dataset. Around 31% of kernel code was executed during the boot process. Mulliner and Neugschwandtner [22] proposed one of the first approaches to library specialization. They identified and removed all non-imported library functions at load time. As we mentioned in Section V-A1, Quach et al. [21] developed piece-wise to remove functions which are not required by an application at load time by using points-to analysis to generate a complete call graph for each library. Mururu et al. [49] have proposed a context-sensitive debloating tool to load only functions required at each call site and remove the rest. They use an oracle to predict which functions from the library are required at each call site, and purge the code after it’s used. They haven’t specified how functions could be used on later accesses, though. Song et al. [46] used data dependency analysis to show the potential of fine-grained library customization of statically linked libraries. Shredder [23] limits arguments passed to critical system API functions to legitimate values extracted from each application’s source code.

Other works explore the potential of debloating software based on predefined features and dynamic analysis.

a) *CHISEL*: CHISEL [42] proposes a framework to shrink software using a reinforcement-learning-based approach based upon test cases provided by the user. It's a test case driven approach, which reduces the code size, such that none of the test cases fail.

TOSS [50] proposes an automated framework to customize online servers and software systems, using only binaries. They use dynamic tainting to track feature extraction and leverage symbolic execution to improve code coverage. They've evaluated TOSS by successfully applying it to Mosquitto, a message servicing server which can be used for lightweight IoT communications.

Kroes et al. [51] perform IR lifting on binaries and remove unnecessary parts of it based on dynamic analysis and profiling. TRIMMER [52] uses inter-procedural analysis to find unnecessary parts of code based upon user-defined configuration data. DamGate [53] uses static and dynamic analysis to rewrite binaries with *gates* preventing execution of unused features. They don't actually remove any code and rely on a set of manually provided seed functions to place their check functions. The seed functions define important feature operations and act as a starting point for their analysis.

Other works in this area have also focused on different programming languages [54], [55], [56]. Jred [54] performs static analysis on Java code to remove unused methods and classes. Jiang et al. [56] presented a feature-based debloating for Java programs using data-flow analysis.

B. Kernel Debloating

There has also been work on debloating the kernel and customizing it based on user requirements.

a) *FACE-CHANGE*: FACE-CHANGE [18] identifies necessary kernel code for each application through profiling and enforces it by using virtualization techniques. They implemented their profiler as a component of the QEMU full system emulator. The profiler would record any address range executed from the kernel code, whenever the guest OS would schedule an application to run. This allowed them to track an application's execution at a basic block granularity. The completeness of the profiling depends on the test suites run on the application, and the correctness of the generated profile can't be guaranteed. This is a general problem common among all approaches which rely on dynamic analysis to extract required kernel code. They add kernel code related to hardware interrupt handlers to the generated profile for all applications, since they can't guarantee all interrupt handlers will be used during their profiling phase. Although profiling is performed at the basic block level, entire functions are added at runtime. This means they add the entire function for any basic block required. They've implemented the runtime component of their system which performs the kernel code minimization for each process within the KVM hypervisor. They change the page table entries in the EPT

to map the kernel guest physical memory for a process to the customized version allocated in the host memory.

FACE-CHANGE has 7% overhead on average but can increase to 40% for the Apache Web Server in case the number of requests pass a certain threshold. The current implementation is limited to guest VMs with a single vCPU.

b) *KASR*: KASR [47] also uses dynamic analysis to identify unused parts of the kernel and uses the OS permissions to limit each application to its profile. They propose a hypervisor-based approach to profile the used kernel code pages by revoking executable permissions from all of them and enabling it after handling the fault. KASR has three benefits compared to FACE-CHANGE [18], first, it supports KASLR [57], second, it has a lower overhead and last, it supports multiple vCPUs whereas the former only supported one.

Kurmus et al. [17] propose a method to tailor the Linux kernel for special workloads through automatic generation of kernel configuration files. MultiK [58] addresses creating specialized kernels for containers, but relies on dynamic analysis to identify the executed parts of the kernel and needs the system call requirements of applications to be provided.

C. Container Security and Debloating

Wan et al. [25] use dynamic analysis to profile the running applications on a container and their system calls to generate seccomp rulesets for containers. DockerSlim [26] is also an open source tool which relies on dynamic analysis to generate seccomp profiles and remove unnecessary files from a docker image. However, the required system calls can't be reliably extracted through dynamic analysis alone – especially for cases that handle exceptions and errors, which are typically not part of the usual execution. Therefore, dynamic analysis cannot guarantee completely covering all the system call requirements of each application.

Cimplifier [27] splits containers running multiple applications into multiple single-purpose containers using dynamic analysis. An approach like Cimplifier can be applied on top of automated seccomp policy generation solutions to create more strict policies, by first splitting the target Docker image, into multiple single-purpose containers, and then running the automatic seccomp policy generation framework. Rastogi et al. [28] suggests further improvement to Cimplifier [27] through symbolic execution.

Previous works have also focused on container security with regards to its software protection mechanisms and vulnerabilities. Lin et al. [59] provide a dataset of security vulnerabilities and exploits which can potentially bypass the software isolation provided by the Linux Kernel. One of their recommendations is the use of stricter seccomp policies for containers. Shu et al. [60] have created a framework to perform vulnerability scanning on images found on the Docker hub. Combe et al. [61] explored the

security implications of using containers, by considering adversary models which assume complete access of an adversary to one container on a host.

VII. CONCLUSION AND FUTURE WORK

In this report we presented a survey of previous work on attack surface reduction. We have shown that the increase of features has led to a substantial amount of bloat being added to both user-space applications and the Linux kernel.

The current state of this line of research shows that an important aspect of attack surface reduction through software debloating is identifying parts of code which can be removed in a sound manner. We believe static analysis must be used to correctly identify unnecessary parts of the code to guarantee a sound and practical debloating mechanism.

REFERENCES

- [1] L. Szekeres, M. Payer, L. T. Wei, and R. Sekar, "Eternal war in memory," *IEEE Security & Privacy*, vol. 12, no. 3, pp. 45–53, 2014.
- [2] Microsoft, "A detailed description of the data execution prevention (dep) feature in windows xp service pack 2, windows xp tablet pc edition 2005, and windows server 2003." <https://support.microsoft.com/en-us/help/875352/a-detailed-description-of-the-data-execution-prevention-dep-feature-in> 2017.
- [3] PaX Team, "Address Space Layout Randomization (ASLR)," 2003. <http://pax.grsecurity.net/docs/aslr.txt>.
- [4] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, (New York, NY, USA), pp. 340–353, ACM, 2005.
- [5] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications," in *2015 IEEE Symposium on Security and Privacy*, pp. 745–762, IEEE, 2015.
- [6] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pp. 161–176, 2015.
- [7] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection," in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pp. 401–416, 2014.
- [8] V. Van der Veen, D. Andriesse, E. Göktas, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, "Practical context-sensitive cfi," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 927–940, ACM, 2015.
- [9] M. Zhang and R. Sekar, "Control flow integrity for cots binaries," in *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pp. 337–352, 2013.
- [10] J. Li, X. Tong, F. Zhang, and J. Ma, "Fine-cfi: Fine-grained control-flow integrity for operating system kernels," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 6, pp. 1535–1550, 2018.
- [11] Y. Liu, P. Shi, X. Wang, H. Chen, B. Zang, and H. Guan, "Transparent and efficient cfi enforcement with intel processor trace," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 529–540, IEEE, 2017.
- [12] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer, "Block oriented programming: Automating data-only attacks," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1868–1882, 2018.
- [13] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats," in *USENIX Security Symposium*, vol. 5, 2005.
- [14] L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi, "Pt-rand: Practical mitigation of data-only attacks against page tables," in *NDSS*, 2017.
- [15] S. Proskurin, M. Momeu, S. Ghavamnia, V. P. Kemerlis, and M. Polychronakis, "xmp: Selective memory protection for kernel and user space," in *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 584–598, 2020.
- [16] C.-T. Lee, J.-M. Lin, Z.-W. Hong, and W.-T. Lee, "An application-oriented linux kernel customization for embedded systems," *J. Inf. Sci. Eng.*, vol. 20, no. 6, pp. 1093–1107, 2004.
- [17] A. Kurmus, R. Tartler, D. Dorneanu, B. Heinloth, V. Rothberg, A. Ruprecht, W. Schroder-Preikschat, D. Lohmann, and R. Kapitza, "Attack surface metrics and automated compile-time os kernel tailoring," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2013.
- [18] X. Z. Zhongshu Gu, Brendan Saltaformaggio and D. Xu, "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)*, 2014.
- [19] H. He, S. K. Debray, and G. R. Andrews, "The revenge of the overlay: automatic compaction of os kernel code via on-demand code loading," in *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pp. 75–83, ACM, 2007.
- [20] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM conference on Computer and Communications security (CCS)*, October 2007.
- [21] A. Quach, A. Prakash, and L. Yan, "Debloating software through piece-wise compilation and loading," in *Proceedings of the 27th USENIX Security Symposium*, pp. 869–886, 2018.
- [22] C. Mulliner and M. Neugschwandtner, "Breaking payloads with runtime code stripping and image freezing," 2015.
- [23] S. Mishra and M. Polychronakis, "Shredder: Breaking Exploits through API Specialization," in *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*, 2018.
- [24] H. Koo, S. Ghavamnia, and M. Polychronakis, "Configuration-driven software debloating," in *Proceedings of the 12th European Workshop on Systems Security*, p. 9, ACM, 2019.
- [25] Z. Wan, D. Lo, X. Xia, L. Cai, and S. Li, "Mining Sandboxes for Linux Containers," in *Proceedings - 10th IEEE International Conference on Software Testing, Verification and Validation, ICST 2017*, pp. 92–102, 2017.
- [26] docker slim, "Docker slim - minify and secure docker containers (free and open source!)." <https://dockerslim.im>.
- [27] V. Rastogi, D. Davidson, L. D. Carli, S. Jha, and P. D. McDaniel, "Cimplifier: automatically debloating containers," in *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2017.
- [28] V. Rastogi, C. Niddodi, S. Mohan, and S. Jha, "New directions for container debloating," in *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation*, pp. 51–56, ACM, 2017.
- [29] K. S. Murugiah Souppaya, John Morello, "Application Container Security Guide," 2017. <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-190.pdf>.
- [30] S. Ghavamnia, T. Palit, A. Benameur, and M. Polychronakis, "Confine: Automated system call policy generation for container attack surface reduction," in *Proceedings of the International Conference on Research in Attacks, Intrusions, and Defenses (RAID)*, 2020.
- [31] S. Ghavamnia, T. Palit, S. Mishra, and M. Polychronakis, "Temporal system call specialization for attack surface reduction," in *29th USENIX Security Symposium (USENIX Security 20)*, (Boston, MA), USENIX Association, 2020.
- [32] N. DeMarinis, K. Williams-King, D. Jin, R. Fonseca, and V. P. Kemerlis, "Sysfilter: Automated system call filtering for commodity software," in *Proceedings of the International Conference on Research in Attacks, Intrusions, and Defenses (RAID)*, 2020.

- [33] LLVM, “The llvm compiler infrastructure.” <http://llvm.org>, 2008.
- [34] “Google.” <https://google.com>, 2020.
- [35] I. Agadakis, D. Jin, D. Williams-King, V. P. Kemerlis, and G. Portokalidis, “Nibbler: debloating binary shared libraries,” in *Proceedings of the 35th Annual Computer Security Applications Conference*, pp. 70–83, ACM, 2019.
- [36] D. Williams-King, H. Kobayashi, K. Williams-King, G. Patterson, F. Spano, Y. J. Wu, J. Yang, and V. P. Kemerlis, “Egalito: Layout-agnostic binary recompilation,” in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 133–147, 2020.
- [37] C. Qian, H. Hu, M. Alharthi, P. H. Chung, T. Kim, and W. Lee, “RAZOR: A framework for post-deployment software debloating,” in *Proceedings of the 28th USENIX Security Symposium*, 2019.
- [38] L. O. Andersen, *Program analysis and specialization for the C programming language*. PhD thesis, University of Copenhagen, 1994.
- [39] B. Steensgaard, “Points-to analysis in almost linear time,” in *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 32–41, 1996.
- [40] Y. Sui and J. Xue, “Svf: interprocedural static value-flow analysis in llvm,” in *Proceedings of the 25th international conference on compiler construction*, pp. 265–266, ACM, 2016.
- [41] M. Ghaffarinia and K. W. Hamlen, “Binary control-flow trimming,” in *Proceedings of the cnum26th ACM Conference on Computer and Communications Security nymCCS. forthcom-ing*, 2019.
- [42] K. Heo, W. Lee, P. Pashakhanloo, and M. Naik, “Effective program debloating via reinforcement learning,” in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, 2018.
- [43] Nginx, “Nginx.” <https://www.nginx.com/>, 2019.
- [44] J. R. Blackbelt, “Processor tracing.” <https://software.intel.com/content/www/us/en/develop/blogs/processor-tracing.html>, 2013.
- [45] J. Wagner, V. Kuznetsov, G. Candea, and J. Kinder, “High system-code security with low overhead,” in *2015 IEEE Symposium on Security and Privacy*, pp. 866–879, IEEE, 2015.
- [46] L. Song and X. Xing, “Fine-grained library customization,” in *Proceedings of the ECOOP 1st International Workshop on SoftwAre debLoating And Delaying (SALAD)*, 2018.
- [47] Z. Zhang, Y. Cheng, S. Nepal, D. Liu, Q. Shen, and F. Rabhi, “Kasr: A reliable and practical approach to attack surface reduction of commodity os kernels,” in *International Symposium on Research in Attacks, Intrusions, and Defenses*, pp. 691–710, Springer, 2018.
- [48] A. Quach, R. Erinfoami, D. Demicco, and A. Prakash, “A multi-os cross-layer study of bloating in user programs, kernel and managed execution environments,” in *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation*, pp. 65–70, ACM, 2017.
- [49] G. Mururu, C. Porter, P. Barua, and S. Pande, “Binary debloating for security via demand driven loading,” *arXiv preprint arXiv:1902.06570*, 2019.
- [50] T. L. Yurong Chen, Shaowen Sun and G. Venkataramani, “Toss: Tailoring online server systems through binary feature customization,” in *Proceedings of the Second Workshop on Forming an Ecosystem Around Software Transformation (FEAST)*, 2018.
- [51] T. Kroes, A. Altinay, J. Nash, Y. Na, S. Volckaert, H. Bos, M. Franz, and C. Giuffrida, “Binrec: Attack surface reduction through dynamic binary recovery,” in *Proceedings of the 2018 Workshop on Forming an Ecosystem Around Software Transformation, FEAST ’18*, (New York, NY, USA), pp. 8–13, ACM, 2018.
- [52] A. G. Hashim Sharif, Muhammad Abubakar and F. Zaffar, “Trimmer: Application specialization for code debloating,” in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2018.
- [53] T. L. Yurong Chen and G. Venkataramani, “Damgate: Dynamic adaptive multi-feature gating in program binaries,” in *Proceedings of the Second Workshop on Forming an Ecosystem Around Software Transformation (FEAST)*, 2017.
- [54] D. W. Yufei Jiang and P. Liu, “Jred: Program customization and bloatware mitigation based on static analysis,” in *Proceedings of the 40th Annual Computer Software and Applications Conference (ACSAC)*, 2016.
- [55] K. G. Suparna Bhattacharya and M. G. Nanda, “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)*, 2013.
- [56] Y. Jiang, C. Zhang, D. Wu, and P. Liu, “Feature-based software customization: Preliminary analysis, formalization, and methods,” in *Proceedings of the 17th IEEE International Symposium on High Assurance Systems Engineering (HASE)*, 2016.
- [57] K. Cook, “Kernel address space layout randomization,” *Linux Security Summit*, 2013.
- [58] H.-C. Kuo, A. Gunasekaran, Y. Jang, S. Mohan, R. B. Bobba, D. Lie, and J. Walker, “Multik: A framework for orchestrating multiple specialized kernels,” *arXiv preprint arXiv:1903.06889*, 2019.
- [59] X. Lin, L. Lei, Y. Wang, J. Jing, K. Sun, and Q. Zhou, “A measurement study on Linux container security: Attacks and countermeasures,” in *Proceedings of the 34th Annual Computer Security Applications Conference*, pp. 418–429, ACM, 2018.
- [60] R. Shu, X. Gu, and W. Enck, “A study of security vulnerabilities on docker hub,” in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pp. 269–280, ACM, 2017.
- [61] T. Combe, A. Martin, and R. Di Pietro, “To docker or not to docker: A security perspective.,” *IEEE Cloud Computing*, vol. 3, no. 5, pp. 54–62, 2016.