





ultra-light weight execution tracer that records only the code that was executed within the kernel context. The overview of our measurement system is presented in Figure 1. In a nutshell, like most other dynamic translators (e.g., DynamoRIO and Intel’s Pin), Qemu maintains a translation code cache. Whenever any code (kernel or user) must be executed, during the first instance, it is translated into a “Translation Block” (TB) and is maintained within the translation cache. Each TB is analogous to a basic block in the code. This one time transformation is key to the performance benefits of the emulator.

We intercept the translation stage in Qemu and whenever a TB is generated, we record the virtual address and size of the code that corresponds to the TB. Next, we filter the blocks that correspond to the kernel address region to identify kernel code that was executed. Note that this is different from regular tracing because unlike regular tracing (e.g., DECAF [8]), the TBs themselves are unmodified and we do not frequently seek CPU state. As an additional benefit, such an approach allows us to measure multiple kernels without any modification to the measurement code.

We make two specific measurements:

- (1) *Boot Process*: We seek to evaluate how much of the kernel code actually executes during the boot process. This information will help determine the attack surface available for malware that target the boot process. In this case, we gather all the kernel code that was executed irrespective of the process context under which the execution occurred.
- (2) *System calls and bloat*: We seek to evaluate the footprint of different system calls in the kernel. User programs interact with the kernel through system calls. Therefore, by examining the footprint of commonly used system calls in the kernel, we can estimate the amount of kernel code that typically executes during program execution. This also reveals the scope to debloat the kernel. In order to measure the system calls, we:
  - (a) We run a synthetic program that invokes specific system call(s).
  - (b) Upon reaching the entry point of the program, we clear the translation cache. That is, we delete all the TBs to force translation of any future code. This process ensures that any system calls executed by the loader are excluded.
  - (c) We record the kernel code that resulted in emission of TBs, and stop recording when the servicing of system call completes.

Translation cache is a per-CPU data structure. Therefore, to eliminate interference due to kernel code execution from another process context, we use the CR3 register to filter the code that corresponds to the synthetic program we created. Because the CR3 register uniquely represents a process context, we retrieve the kernel code that executed in the context of our synthetic program.

## 2.4 Measuring Bloat in Execution Engines

We aim to measure the amount of code in the runtime/interpreter that is actually triggered during the execution of typical language-specific programs. We subject Java virtual machine and Python interpreter to different—typical—payloads and measure the amount

of code that executes during the execution of a Java/Python program. Next, we identify the code in runtime that is common to multiple payloads. This gives us the core component of the runtime that is exercised during a typical execution (e.g., bytecode reader). We hope to obtain a sense of how much code exists in the runtime vis-a-vis the amount of code that is needed for typical program execution.

## 3 RESULTS

**Test Set:** Our test set comprises of a wide range of programs ranging from complex browsers (e.g., Firefox), media player (vlc), compilers (clang++, g++), text editors (sublime), and utility programs (make). For the kernels, we pick Debian Wheezy FreeBSD 64bit, Debian Wheezy AMD 64bit, and Windows 8.1 OSes. We use the Java virtual machine (JDK 1.8.0) and Python interpreter 2.7.6 to evaluate bloating in managed execution engines.

### 3.1 Bloating in User-Level Programs

**Static measurement of bloat—lower bound.** The results of our static measurement experiments are presented in Table 1. As an average overestimate (as described in Section 4.1), programs in our test set require only 65% of the code in the libraries and 73% of the code including the libraries and the executable. Chrome is an exceptional case where the executable is relatively large and contributes a large amount of code to the overall code in the memory. While all the libraries loaded by Chrome add up to 4.04 million instructions, chrome.exe alone accounts for 28.3 million instructions. On average, only 36% of functions in the memory are used by programs. This indicates that there exists space for research on optimal code organization within modules. For example, we may be able to group frequently used functions across different libraries into a single library so as to reduce the overall bloat across programs.

**Dynamic measurement of bloat for typical loads.** We ran the programs in our test set on Pin, a dynamic translator, and we examined the execution profile of the programs. Results are tabulated in Table 2. Programs were subjected workloads that the programs are typically run on (column 2). Only about 21% (average) of code in the memory executes for typical payloads, which corresponds to 12.12% of functions. Note that even a heavy program like firefox executes less than 30% of code. That is, most of the code is never executed in most of the cases. This provides us with insights on potential approaches to debloat in a context-specific manner (more in Section 4.2).

### 3.2 Bloating in Kernel

Below, we present our findings from kernel experiments. The overall kernel code in our study includes code in the kernel and all the loaded kernel modules.

**Boot time code execution.** We measured the fraction of kernel code that was executed during the boot process. Our results are tabulated in Table 3. In each case, we monitored the unique TBs that were created between the system being turned on and the login prompt being displayed to the user. Across three kernels tested, on

**Table 1: Static bloat measurement. LIR: Percentage of instructions in all the dependent libraries that a program *may* execute. OIR: Percentage of instructions in all the libraries + executable that a program *may* execute. LFR: Percentage of functions in all the dependent libraries that a program *may* execute. OFR: Percentage of functions in all the libraries + executable that a program *may* execute.**

Program	% Library Instructions Required (LIR)	% Overall Instructions Required (OIR)	% Library Functions Required (LFR)	% Overall Functions Required (OFR)
firefox	67.20%	68.37%	36.42%	38.60%
chrome	69.72%	95.67%	33.57%	36.75%
webbrowser-app	58.86%	59.03%	29.34%	30.22%
vlc	78.22%	78.25%	42.44%	42.79%
rhythmbox	77.92%	77.92%	29.83%	29.83%
evince	70.84%	71.34%	33.61%	36.19%
sublime	68.88%	84.95%	39.13%	41.42%
gnome calculator	68.59%	69.21%	34.02%	36.18%
git	62.70%	78.11%	22.75%	29.11%
clang	53.99%	73.91%	34.32%	56.83%
g++	52.36%	64.37%	23.90%	29.58%
make	52.13%	56.06%	23.11%	27.75%
<b>Average</b>	<b>65.11%</b>	<b>73.01%</b>	<b>31.87%</b>	<b>36.27%</b>

**Table 2: Runtime bloat measurement. Percentage of code executed in user-level programs for typical usage. For each programs, we list the number of shared libraries loaded, % of library instructions executed, % instructions in process executed, % library functions executed, and number of unique system calls invoked by both program and shared libraries.**

Program	Workload	#Libraries loaded	% Instructions Executed in Libraries	% Overall Instructions Executed	% Functions Executed in Libraries	#Syscalls
firefox	Open top 10 websites in Alexa list	146	28.66%	28.70%	17.04%	101
webbrowser-app	Open google.com. Open and play a video youtube.com.	182	12.70%	12.76%	15.28%	93
vlc	Play 1 song	681	12.44%	12.44%	10.54%	80
libreoffice	Create, write and save a new word file.	191	23.41%	23.41%	16.03%	86
sublime	Create, write and save a new word file.	77	26.66%	38.12%	16.85	67
gnome-calculator	Add and subtract numbers.	81	35.18%	36.25%	21.35%	59
git	Clone a repository	41	12.61%	11.78%	6.71%	47
clang++	Compile a C++ program	10	6.63%	10.32%	8.62	23
g++	Compile a C++ program	9	4.52%	17.57%	2.53%	17
make	Run make on a C++ project.	9	11.97%	18.20%	6.22%	26
<b>Average</b>			<b>17.48%</b>	<b>20.96%</b>	<b>12.12%</b>	<b>59.9</b>

average only 31.25% of the kernel code executes during the boot process.

**System-call-specific code execution.** We also measured the kernel code that is invoked within individual system calls that are frequently used in Linux [16]. Our findings are tabulated in Table 4. Most system calls exercise under 10% of kernel code.

### 3.3 Bloating in Managed Execution Engines

We also measured the bloat in Java Virtual Machine and Python interpreter. Specifically, for JVM, we ran Java programs and examined the footprint in the JVM. For Python, we ran five programs and recorded the interpreter and shared library code that was executed. Our findings are tabulated in Table 5 and Table 6. Our results show that only about 30% of functions and (32%) of code is executed in the JVM.

**Table 3: Percentage of kernel code executed at boot time**

Operating System	Code Executed During Boot (B)	Kernel Size (B)	% Kernel Code Executed During Boot
Debian Wheezy	2192166	7494595	29.25%
kFreeBSD Wheezy	2445095	10556370	23.16%
Windows 8.1	1112279	2691056	41.33%
<b>Average</b>	<b>1916513</b>	<b>6914007</b>	<b>31.25%</b>

**Table 4: Percentage of code executed in kernel space in kFreeBSD and Debian for popular system calls.**

System Call	Kernel Code Executed (B)		% Kernel Code Executed	
	kFreeBSD	Debian	kFreeBSD	Debian
exit	941961	778361	8.92%	7.89%
exit_group	Not Supported	462053	Not Supported	4.68%
open + close	1076732	964614	10.20%	9.78%
getuid	792612	575879	7.51%	5.84%
execve	1453331	1388650	13.77%	14.07%
getcwd	681763	903860	6.46%	9.16%
write	792519	713358	7.51%	7.23%
getpid	670177	533857	6.35%	5.41%
<b>Average</b>	<b>938293</b>	<b>803343</b>	<b>8.89%</b>	<b>8.14%</b>

**Table 5: Bloating in Java programs.**

Program	Workload	# Modules in JVM	% Instructions Executed	% Functions Executed
Eclipse	Create, compile, and execute a program	6	10.50%	25.13%
		8	33.65%	35.45%
Jabref	Create a bibliography file	13	31.88%	30.42%
Jenkins	Install plugins, create an admin user, and create a pipeline	12	34.51%	32.39%
<b>Average</b>		<b>10</b>	<b>27.63%</b>	<b>30.85%</b>

## 4 DEBLOATING—APPROACHES AND CHALLENGES

We explore two approaches to debloating—a static and a runtime approach.

**Table 6: Bloating in Python programs.**

Program	Workload	# Modules in python	% Instructions Executed	% Functions Executed
Calibre	Open and read a book	34	6.75%	4.97%
		34	7.08%	7.66%
Mercurial	Clone a repository	4	39.76%	69.23%
		6	40.84%	44.44%
Pip	Install a package	11	14.52%	60.49%
		11	9.21%	60.49%
Ubuntu software center	Install and remove a program	2	11.33%	53.85%
		16	32.75%	14.32%
		17	44.88%	30.07%
Gramps	Create a family tree	8	54.25%	36.67%
<b>Average</b>		<b>14</b>	<b>26.14%</b>	<b>38.22%</b>

### 4.1 Static Approach

In the first approach, we could determine the static dependencies for a program and ensure that remaining code is eliminated from the memory. In this approach, we would first identify the static bloat using the technique described in Section 3.1, and then employ a late-stage component (e.g., a modified loader) that would selectively eliminate unwanted functionality. The main advantage of such an approach is that it imposes near-zero runtime overhead (except for the one-time load overhead).

*Challenges:* There are multiple challenges that need addressing. First, the static dependencies must be complete to prevent runtime errors. In our approach, we take a conservative approach to obtain the overestimate of the amount of code required by the program. While this guarantees correctness, it does not yield optimal debloating. Precise analysis will be needed to achieve optimal debloating. Such an analysis is known to be hard especially for binaries. Second, language-specific details must be specifically handled. For example, the virtual function targets in C++ are obtained from a class-specific table called the VTable. A precise analysis must be object-sensitive in order to handle C++ code and precisely identify the potential targets for virtual function calls. Similar challenges exist for execution engines like Java and Python. Third, modules can be loaded during the program runtime (e.g., using `dlopen`) or code may be generated during runtime through Just-In-Time (JIT) compilation. These code modules may exhibit dependencies on code that has already been removed. Special handling is required to handle such JIT code. Failure to handle such cases will result in undesirable program crashes. Finally, selectively disabling/removing code from the memory will hinder code sharing. Shared code is typically memory-mapped into multiple address spaces and a single physical copy is loaded into the memory. Due to copy-on-write, deleting a copy in memory for one process will force duplication of data, which could in turn hinder performance. Special handling will be needed to allow code sharing without significant performance impact.

### 4.2 Dynamic Approach

More ambitious debloating can be achieved through a dynamic approach that is context sensitive. In essence, the actual code that

the program requires to process a given input is known upon concretization of input. That is, when the concrete input is known, the precise dependencies can be computed for that input. Therefore, for that input, other code in the memory is unwanted and can therefore be safely removed/disabled. This approach is context sensitive and requires a runtime component that selectively enables and disables functionality in the memory depending on the program input. On the one hand, such a technique has potential for high level of precision whereas it imposes runtime overhead due to the per-input interception performed by the runtime component. Analogous approaches have been experimented upon in the context of CFI [7, 11].

*Challenges:* A key challenge in dynamic approach is to contain the runtime overhead imposed by the solution. Additionally, challenges faced by static approach (JIT code, code sharing, language specific challenges) are also faced by the dynamic approach.

Additionally, debloating closed-source executables requires analyzing the binary, which is hard. Correct disassembly, accurate control-flow graph recovery, correct binary instrumentation, etc. are all practical challenges faced in binary analysis, and must be solved in order to successfully debloat binaries.

## 5 RELATED WORK

Managed programming languages suffer from significant runtime overhead or bloating due to the extra logic added to manage the execution environment. On the one hand, Xu et al. [17] and Bu et al. [4] delegate the debloating task to developers, classifying this problem as purely software engineer related. On the other hand, Jiang et al. [10] propose a feature-based solution that allows a developer to remove certain features in Java bytecode by performing static analysis. Further, Jiang et al. [9] introduce an automatic approach to statically analyze and remove unused code in both Java application and JRE.

Major impact of bloating is felt in the realm of software security, particularly in code-reuse attacks where an attacker reuses existing code in the memory to achieve malicious computation. ROP [3, 5, 14] and COOP [15] are examples of code-reuse attacks. Modern defenses have focused on enforcing runtime program properties (e.g., CFI [2], SPI [12, 13]) to defeat code-reuse attacks. Generally, debloating reduces the amount of code that needs protection, and therefore strengthens the overall security of the system.

The techniques used in this paper, particularly virtual-machine introspection on Qemu has been previously explored (e.g., DECAF [8]) to provide analysis frameworks that allow fine-grained introspection. These solutions modify the code cache to embed callbacks to analysis code such that timely analysis can be performed upon specific events in the guest OS. While we borrow ideas from those solutions, our approach is unique in the sense that we do not modify individual translation blocks. Instead, we rely on monitoring the translation cache.

## 6 CONCLUSION

We conducted experiments to measure bloat at multiple levels of abstraction—user mode programs, managed execution (JVM and Python interpreter) and the OS kernel. For the user level programs,

we also measure the static lower bound for bloat. Our findings show that (1) bloating is pervasive and a cross-layer problem, and (2) there exists significant research space for systematic debloating at multiple layers in the execution stack.

## 7 ACKNOWLEDGEMENT

We would like to thank anonymous reviewers for their feedback. This research was supported in part by Office of Naval Research Grant #N00014-17-1-2929. Any opinions, findings and conclusions in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

## REFERENCES

- [1] Alexa top 500 sites on the web. <https://www.alexa.com/topsites>. (????). Accessed: 2017-09-17.
- [2] Martin Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS'05)*. 340–353.
- [3] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. 2011. Jump-Oriented Programming: A New Class of Code-Reuse Attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ACM, 30–40.
- [4] Yingyi Bu, Vinayak Borkar, Guoqing Xu, and Michael J Carey. 2013. A bloat-aware design for big data applications. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 119–130.
- [5] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. 2010. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 559–572.
- [6] Chris DiBona and Sam Ockman. 1999. *Open sources: Voices from the open source revolution*. O'Reilly Media, Inc.
- [7] Ren Ding, Chenxiong Qian, Chengyu Song, Bill Harris, Taesoo Kim, and Wenke Lee. 2017. Efficient Protection of Path-Sensitive Control Security. In *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC.
- [8] Andrew Henderson, Aravind Prakash, Lok Kwong Yan, Xunchao Hu, Xujiewen Wang, Rundong Zhou, and Heng Yin. 2014. Make it work, make it right, make it fast: Building a platform-neutral whole-system dynamic binary analysis platform. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 248–258.
- [9] Yufei Jiang, Dinghao Wu, and Peng Liu. 2016. JRed: Program Customization and Bloatware Mitigation Based on Static Analysis. In *Computer Software and Applications Conference (COMPSAC), 2016 IEEE 40th Annual*, Vol. 1. IEEE, 12–21.
- [10] Yufei Jiang, Can Zhang, Dinghao Wu, and Peng Liu. 2015. A preliminary analysis and case study of feature-based software customization. In *Software Quality, Reliability and Security-Companion (QRS-C), 2015 IEEE International Conference on*. IEEE, 184–185.
- [11] Ben Niu and Gang Tan. 2015. Per-input control-flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 914–926.
- [12] Aravind Prakash and Heng Yin. 2015. Defeating ROP Through Denial of Stack Pivot. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC 2015)*.
- [13] Anh Quach, Matthew Cole, and Aravind Prakash. 2017. Supplementing Modern Software Defenses with Stack-Pointer Sanity. In *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC 2017)*.
- [14] 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*. ACM, 552–561.
- [15] Felix Shuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit Object-oriented Programming, On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *Proceedings of 36th IEEE Symposium on Security and Privacy (Oakland'15)*.
- [16] Chia-Che Tsai, Bhushan Jain, Nafees Ahmed Abdul, and Donald E. Porter. 2016. A Study of Modern Linux API Usage and Compatibility: What to Support when You're Supporting. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. ACM, New York, NY, USA, Article 16, 16 pages. <https://doi.org/10.1145/2901318.2901341>
- [17] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, and Gary Seivitsky. 2010. Software bloat analysis: finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*. ACM, 421–426.