

**COMPILER AND MACHINE LEARNING-BASED PREDICTIVE TECHNIQUES
FOR SECURITY ENHANCEMENT THROUGH SOFTWARE DEBLOATING**

A Dissertation
Presented to
The Academic Faculty

By

Chris Porter

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science
College of Computing

Georgia Institute of Technology

August 2023

© Chris Porter 2023

**COMPILER AND MACHINE LEARNING-BASED PREDICTIVE TECHNIQUES
FOR SECURITY ENHANCEMENT THROUGH SOFTWARE DEBLOATING**

Approved by:

Dr. Santosh Pande, Advisor
School of Computer Science
Georgia Institute of Technology

Dr. Vivek Sarkar
School of Computer Science
Georgia Institute of Technology

Dr. Rajiv Gupta
Department of Computer Science and
Engineering
University of California, Riverside

Dr. Qirun Zhang
School of Computer Science
Georgia Institute of Technology

Dr. Alessandro Orso
School of Computer Science
Georgia Institute of Technology

Date approved: July 28, 2023

ACKNOWLEDGMENTS

I must first thank my advisor, Professor Santosh Pande, for his decade of support. He has funded me throughout this entire degree program, and he has dedicated countless evenings and hours working with me. It is through him that I have learned how to do research. I am forever grateful for his mentorship.

I would like to thank my labmates over the years for every bit of help: Kaushik, Girish, Chao, Prithayan, Vincent, Amit, Michael, Sharjeel, Nathan, Bodhi, and Alex. We have shared a few wins, a few disappointments, and way too much fluorescent light.

Thank you immensely to the committee members for your feedback and guidance in improving this thesis: Professors Rajiv Gupta, Alex Orso, Vivek Sarkar, and Qirun Zhang. Professors Hadi Esmailzadeh, Ada Gavrilovska, and Fisayo Omojokun, thank you for helping me pursue this program in the first place. Thank you also to Georgia Tech and its faculty for seeing me through the BS, MS, and now PhD.

To friends, family, and my partner, Lorena, I would not have made it through this program without all of you. This work takes its toll. Thank you for your love and support.

The author gratefully acknowledges that this work was partially supported by the Office of Naval Research (ONR) via Grant No. N00014-17-1-2895, Period: September 1, 2017 to August 31, 2022. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the Office of Naval Research.

TABLE OF CONTENTS

Acknowledgments	iii
List of Tables	viii
List of Figures	x
Summary	xii
Chapter 1: Introduction	1
1.1 Problem Statement	1
1.2 Background	1
1.2.1 Code Reuse Attacks	1
1.2.2 Traditional Defenses	4
1.2.3 State-of-the-art Industry Defenses	6
1.2.4 Debloating Defenses	8
1.3 A Mixed Compiler and Predictive Approach	10
1.4 Thesis Statement	12
1.5 Contributions	12
Chapter 2: BLANKIT: Binary-level, Predictive Library Debloating	14
2.1 Introduction	14

2.1.1	Debloating	15
2.1.2	BLANKIT: “Get what you want”	15
2.1.3	Overcoming CFI Limitations	17
2.1.4	Threat Model	19
2.1.5	Limitations	20
2.1.6	Contributions	20
2.2	BLANKIT’s Overall Framework	21
2.3	BLANKIT’s Runtime	23
2.3.1	BLANKIT Design	23
2.3.2	Handling Misprediction	25
2.4	Call Graph Prediction	27
2.4.1	Overall Prediction Framework	27
2.4.2	Static Call Flow Divergence Detection	28
2.4.3	Reverse Dominance Frontier-Based Prediction	29
2.4.4	Argument Value-Based Prediction	30
2.4.5	Decision Trees	31
2.5	Evaluation 1: SPEC CPU 2006	33
2.5.1	Security	34
2.5.2	Performance	37
2.5.3	Prediction	39
2.6	Evaluation 2: nginx and sshd	41
2.6.1	Security	42
2.6.2	Performance	43

2.6.3	Prediction	44
2.6.4	Defense Against Real-world Attacks	45
2.7	Related Work	46
2.8	Conclusion	47
Chapter 3: DECKER: Source-level Application Debloating		49
3.1	Introduction	49
3.2	Overview	50
3.2.1	Proposed Solution	50
3.2.2	Soundness	52
3.2.3	Threat Model	53
3.3	Framework	54
3.3.1	Compiler Component	54
3.3.2	Runtime Component	62
3.4	Evaluation	64
3.4.1	SPEC CPU 2017	64
3.4.2	GNU coreutils	68
3.4.3	nginx	69
3.4.4	Binary Size Growth	71
3.4.5	Breaking ROP Gadget Chains	72
3.4.6	Breaking JOP Gadget Chains	72
3.5	Related Work	77
3.6	Conclusion	79

Chapter 4: PDSG: Predictive Debloat with Static Guarantees	80
4.1 Introduction	80
4.2 Overview	81
4.2.1 Challenges	83
4.3 Framework	84
4.3.1 Prediction	85
4.3.2 Rectification	88
4.3.3 Path Checking	91
4.3.4 Putting It Back Together	96
4.4 Evaluation	97
4.4.1 Gadget Reduction	98
4.4.2 Performance Overhead	99
4.4.3 Rectification and Prediction	101
4.4.4 Datalog	103
4.5 Conclusion	104
Chapter 5: Conclusion	106
5.1 Future Work	107
References	110

LIST OF TABLES

1.1	Broad properties of 3 state-of-the-art debloating techniques for security (Piece-wise, Chisel, and Razor).	9
2.1	Key graph metrics for SPEC CPU 2006 and glibc	33
2.2	Security-related improvements in SPEC CPU2006	35
2.3	Prediction accuracy and audit overhead	41
2.4	Security-related improvements for nginx and sshd	43
2.5	Transfer/sec degradation with BLANKIT under varying, sustained load times.	44
2.6	Transfer/sec degradation with BLANKIT under varying, sustained request sizes.	44
2.7	Call chain prediction accuracy and audit overhead for nginx and sshd	45
3.1	SPEC CPU 2017 total gadget reduction as a percentage (higher is better). . .	66
3.2	GNU coreutils total gadget reduction as a percentage (higher is better). . . .	69
3.3	JOP gadget metrics over all available page sets running nginx on Linux (baseline shown in average column).	74
3.4	JOP reduction on Windows nginx (higher is better).	76
4.1	SPEC CPU 2017 total gadget reduction as a percentage (higher is better). . .	99
4.2	SPEC CPU 2017 static callgraph properties. The percent of edges that are instrumented with RPs is shown in the rightmost column.	102

4.3	SPEC CPU 2017 prediction and rectification occurrences. The rightmost column is the percentage of predictions that require rectification.	103
4.4	Datalog results for SPEC CPU 2017.	104

LIST OF FIGURES

2.1	BLANKIT compared with other debloating mechanisms	17
2.2	An application running (A) without BLANKIT and (B) with it. The example is based on the return-to-libc attack from [3].	22
2.3	An example of how the copy probe works. Execution flows in order along the arrows.	24
2.4	A prediction call that passes an ID of 4 for the upcoming predicted call chain. Within the prediction probe, the last chain of exposed functions will be blanked if the predicted set has changed. In this case, it has changed (from <code>malloc</code> and <code>mmap64</code> to <code>free</code>), so blanking occurs.	26
2.5	Example decision tree for Listing 2.1	32
2.6	Runtime slowdown for BLANKIT on SPEC 2006 CPU, normalized against native.	39
3.1	High-level view of the compiler part. The LLVM pass outputs a custom linker script and instrumented object file.	51
3.2	High-level runtime example from GNU coreutils' <code>date</code> . The set of RX-mapped pages increases at P2 to include a called function.	52
3.3	Simplified callgraph from the <code>xz</code> data compression application. This illustrates 4 types of edges, each of which requires different handling by the instrumentation pass.	56
3.4	Slowdown for SPEC CPU 2017 using DECKER.	65
3.5	Transfer/sec degradation for <code>nginx</code> using DECKER.	70

4.1	High-level flow diagram of PDSG’s profiling and release stages. Dashed arrows indicate “feeds into;” solid arrows indicate “creates.”	81
4.2	High-level depiction of execution flow (X nodes) and prediction (donut nodes) through a callgraph, and a corresponding memory layout of the predicted functions.	83
4.3	Example callgraph for demonstrating rectification.	89
4.4	Slowdown for SPEC CPU 2017 using PDSG.	100

SUMMARY

Code reuse attacks continue to be a serious threat to software. Attackers today are able to piece together short sequences of instructions in otherwise benign code to carry out malicious actions. Eliminating these reusable code snippets, known as gadgets, has become one of the prime focuses of attack surface reduction research. The aim is to break these chains of gadgets, thereby making such code reuse attacks impossible or substantially less common. Recent work on attack surface reduction has attempted to eliminate these attacks by subsetting the application, e.g. via user-specified inputs, configurations, or features, to achieve high gadget reductions. However, such approaches suffer from the limitations of soundness (meaning the software might crash or produce incorrect output during no-attack executions on regular inputs), or the techniques may be conservative and leave a large amount of attack surface untackled. This thesis develops three techniques that combine static analysis with dynamic predictions based on machine learning (ML) to address the above shortcomings. They are fully sound, obtain strong gadget reduction, and are shown to break shell-spawning gadget chains and stop real-world attacks arising out of known Common Vulnerabilities and Exposures (CVEs). The techniques reduce attack surface by activating a (minimal) set of functions at chosen callsites and then deactivating them upon return.

In the first work, BLANKIT, we target library code and achieve $\sim 97\%$ attack surface reduction. The technique uses arguments to library function calls and their static single assignment-based backward slices for training an ML model, which then predicts reachable functions at the callsite using runtime values. In particular, we are able to debloat GNU libc, which is notorious for housing gadgets for code reuse attacks. In the second work, DECKER, we target application code and achieve $\sim 73\%$ total gadget reduction. The percentage reduction is similar to prior art but without sacrificing soundness. Decker works by instrumenting the program at compile-time at key points to enable and

disable code pages; then at runtime, the framework executes these permission-mapping calls with minimal overhead ($\sim 5\%$). In the third work, PDSG, we show how to augment the whole-application technique with an accurate predictor to further reduce the potential attack surface. ML-based predictive techniques do not offer guarantees and suffer from mispredictions; thus, the predictions are sanitized with lightweight checks. The checks rely on statically derived *ensue* relations (i.e. valid call sequence relations) that are used for separating mispredictions from actual attacks. PDSG achieves $\sim 83\%$ total gadget reduction with $\sim 11\%$ runtime overhead. Its predictions trigger runtime checking in $\sim 4\%$ of cases.

In conclusion, the thesis empirically shows that it is possible to devise precise and sound attack surface reduction techniques by combining static analysis and ML to overcome their inherent limitations. ML prediction aids purely static analysis by improving its precision, and static techniques augment the ML models by providing mechanisms for identifying when a misprediction is truly an attack.

CHAPTER 1

INTRODUCTION

1.1 Problem Statement

Modern software systems are still susceptible to code reuse attacks. In these attacks, the executable code of the program *itself* is repurposed to construct a successful attack. To address this, a heavily researched area lately is to simply remove unneeded code in the program, i.e. “debloat” the software. The idea is to essentially hamstring the attacker by removing the code they need. This defense has been effective to some extent, but there are shortcomings with current approaches. Broadly, known debloating techniques are either too conservative, leaving too much code available to attackers; or they compromise soundness by specializing the application for only certain inputs or features, which can lead to crashes or incorrect output. This motivates us to find a set of debloating techniques that removes enough of the attack surface to break real-world attacks, while still leaving all features of a program intact.

1.2 Background

1.2.1 Code Reuse Attacks

Code injection could be considered the predecessor of modern code reuse attacks. Early code injection attacks could simply inject code into memory such as the heap and execute it. This was countered by data execution prevention (DEP) [1]. DEP enforces the write XOR execute ($W \oplus X$) property on pages, which is sufficient for stopping such blatant attacks.

Attackers grew to overcome this. Perhaps the most basic code reuse attack is the classic return-to-libc attack [2, 3]. In a return-to-libc attack, the attacker exploits some memory

vulnerability to compromise “control data,” i.e. an indirect branch instruction (return, indirect jump, or indirect call). Then the attacker redirects control flow to jump into GNU libc (glibc) code, which is full of functionality that can be very useful to an attacker. For example, one useful place to jump can be the `mprotect` function. If the attacker can control the parameters to this function, then they can remap a page as writable, breaking the $W \oplus X$ property, and allowing them to carry out code injection. Address space layout randomization (ASLR) [4] can make it more difficult to locate target code such as the glibc library functions, but then there are also known attacks that get around it [5, 6, 7, 8].

Reuse attacks have only become more sophisticated. Return-oriented programming [9], jump-oriented programming [10, 11], and call-oriented programming [12] are all techniques that leverage existing code to perform attacks. They rely on “gadgets” in the code base, which are sequences of instructions that can be strung together to perform malicious control flow and eventually some malicious computation. In fact, an attacker can construct a Turing-complete program from gadgets that exist in the code base, but this is often not even needed. Turing-incomplete functionality may be sufficient if it still allows the attacker to carry out some useful action (e.g. leaking a secret, which could be the end goal, or spawning a shell, to give the attacker more control).

Listing 1.1: An ROP gadget chain from `nginx` that spawns a shell with `execve`.

```
gadget 1 addr  -> pop rax; ret;
'/bin/sh'
gadget 2 addr  -> pop rcx; ret;
bss addr       -> 0x00800000
gadget 3 addr  -> mov qword ptr [rcx], rax; ret;
gadget 4 addr  -> pop rax; ret;
0x00000000
gadget 5 addr  -> pop rcx; ret;
bss addr + 8   -> 0x00800008
gadget 6 addr  -> mov qword ptr [rcx], rax; ret;
gadget 7 addr  -> pop rdi; ret;
```

```
bss addr      -> 0x00800000
gadget 8 addr -> pop rsi; ret;
bss addr + 8  -> 0x00800008
gadget 9 addr -> pop rdx; ret;
bss addr + 8  -> 0x00800008
gadget 10 addr -> pop rax; ret;
0x0000003b
gadget 11 addr -> syscall;
```

Listing 1.1 shows an example of a return-oriented programming (ROP) gadget chain. These chains can be used as part of an attack to leak secrets, hijack processes, or otherwise cause damage. Snippets of code (gadgets) are strung together with return statements to jump from one to the next, in order to carry out some malicious computation. In the above example, the chain is designed to compute relatively little but to a powerful end: It will launch a shell via `execve`. This is a real chain that has been automatically generated by the Ropper tool [13] on the `nginx` server application [14]. Launching such an attack requires exploiting a memory vulnerability, which is also a realistic possibility in this and many C/C++ code bases due to the languages' weak memory models. CVE-2013-2028, for example, is a bug in the decoding functionality of `nginx`. Carlini et al. [15] show a technique called control-flow bending (CFB) that exploits this bug to write to arbitrary locations. Note that as with any code reuse attack that we consider, we assume the attacker has some way of initiating the attack. Here we assume, for example, that the attacker can exploit some memory vulnerability to write this chain into the stack, overwrite the return address to point to gadget 1's address, and freely return along the ROP chain.

Referring to Listing 1.1, the ROP chain proceeds as follows. It stores the address of the `"/bin/sh"` string into some location in the `.bss` section (gadgets 1-3). Then, 8 bytes beyond that, it stores the value 0 (gadgets 4-6). Then it places the address of `"/bin/sh"` into the `rdi` register, and it places the 0 value into `rsi` and `rdx` (gadgets 7-9). Lastly, it stores the `execve` syscall number `0x3b` into the `rax` register and then executes the syscall

instruction (gadgets 10-11). The control flow for executing this attack is return-based. The gadget addresses are the return targets. Thus, each `ret` instruction does the attacker's bidding, popping the gadget address and jumping to the next malicious snippet.

There are several important details about the strength of gadget chains. First, as in this case, the chain does not need to be Turing-complete to be effective. Here the gadgets perform a specific set of actions, all in the service of setting up an `execve` call that will launch `/bin/sh`. Second, the gadget chain does not need to be built from "intended" instructions on architectures such as x86. Though not shown in the listing, gadget addresses from this `nginx` example leverage "unintended" instructions. They are at unintended offsets in the original instructions (which is allowed in x86's variable-length instruction set architecture). Third, the gadgets are not particularly rare instruction sequences, so scavenging these gadgets (or computationally equivalent sequences) from the executable code ought to be achievable given a sufficient set of instructions.

These points illustrate the importance as well as the difficulty of the problem. Not only is it possible to create an attack from only a handful of gadgets from a large pool of instructions, but on x86 the gadgets can be cherry-picked from what is essentially a byte stream of executable code. And this attack represents a very serious security breach. Once the attacker hijacks the process and launches a shell, they can perform any action with that process' permissions.

1.2.2 Traditional Defenses

Hardware and software defense mechanisms have been developed to detect and mitigate these types of attacks, but even state of the art techniques have limitations. For instance, address space layout randomization [4] (ASLR) and data execution prevention [1] (DEP) cannot protect against advanced control-flow hijack techniques like ROP and JOP. To defend against the exploitation of these advanced vulnerabilities, specific and general mechanisms have been invented. One such well-known defense mechanism that has been researched

extensively is control-flow integrity (CFI) [16, 17, 18, 19, 20, 21, 22, 23], a mechanism that attempts to restrict the program execution to legal paths only.

While coarse-grained CFI such as bin-CFI (a binary-level CFI technique) [24] restricts gadget jumps to the starting addresses of functions, it over-approximates the legal execution paths possible through calls and returns. This allows for malicious program paths and thus overlooks certain attacks. TypeArmor [25] tries to improve on bin-CFI by reducing the over-approximation of forward edges. To further reduce the approximation of the allowed set of calls from a call site and to maintain legal returns, a number of fine-grained CFI techniques such as π CFI [26] and MCFI [27] leverage the source code of libraries. However, computing a fully-precise CFI is an intractable problem because of the inherent limitations in pointer analysis [28], irregular control flow in the presence of `setjmp` and `longjmp`, or an inherent inability to distinguish between a legal and illegal dynamic control flow base with regard to the current program input. μ CFI [21] leverages Intel Processor Trace and static analysis to enforce its unique target code property for indirect control-flow transfers. It broadens the scope of CFI by handling non-control data corruption that can alter function pointers.

Memory safety is a defense mechanism that prevents memory corruption by verifying the correctness of memory operations. While memory safety techniques like Soft-bound+CETS [29] can thwart almost all control-flow hijacking attacks, they incur overheads $\geq 2x$. Due to the large overhead, it is impractical to deploy it online. Code pointer integrity (CPI) [30] is a defense mechanism that uses memory safety only on pointers that can directly or indirectly modify code (control-flow), thus lowering the overheads of a full memory safety mechanism. Control-flow bending (CFB) via generalization of non-control data attacks shows that a fully precise, static CFI approach cannot defend against many non-control data attacks, and similarly CPI suffers from such attacks. CFB demonstrates that with the availability of a dispatcher function, commonly available in `glibc`, a CFI mechanism can be fooled to successfully carry out an attack. CFI also does not handle certain

types of privilege escalation attacks [31, 32]. Due to this, techniques like CFI and CPI still need to be complemented with defense mechanisms for these specific attacks that fall outside their scope. Moreover, as defenses strengthen, new attacks are revealed that undermine the current best practices of defense mechanisms. Due to the sophistication of attack construction tools, the number of newly discovered attacks are growing at a much faster rate, which may lead to a losing battle [33].

1.2.3 State-of-the-art Industry Defenses

To understand the extent to which code reuse attacks continue to be serious and credible threats, we can look at the state of affairs for two prominent companies in the software and hardware space, Microsoft and Intel. Both companies address code reuse attacks with control flow integrity (CFI) solutions. CFI is designed to restrict forward- and backward-edge control flow to only valid paths. Microsoft and Intel's defenses are also outlined well in [34].

Microsoft touts two main CFI defenses: Control Flow Guard (CFG) and Return Flow Guard (RFG). Control Flow Guard is coarse-grained, forward-edge CFI. Return Flow Guard is the backward-edge CFI solution. CFG is implemented as follows: Prior to an indirect call, the Windows portable executable (PE file) contains a check that the address is valid. RFG is a software-based shadow stack implementation: Prior to a return, a check ensures that the return target is on the stack.

Intel's latest defense is called Control-flow Enforcement Technology (CET). It is a hardware CFI defense that requires integration with the operating system to work properly. Currently CET is supported in Windows 10. Only Intel Tiger Lake CPUs (or later) have this hardware support. (The AMD equivalent of this technology is in AMD Zen 3 Ryzen CPUs and later.) CET has forward- and backward-edge support. The forward-edge support is called Indirect Branch Tracking (IBT). The backward-edge support is a hardware shadow stack.

Bypasses have been discovered for Microsoft's CFI line of defense [35, 36, 37]. Another valid strategy for attackers is to just avoid it. Because PE files have to be compiled with the `/guard:cf` flag, it is possible to launch attacks on binaries that are never compiled with CFG support. Moreover, even if the binary is instrumented with CFG support, its supporting dynamic-linked libraries (DLLs) may not have been. This allows attackers to target the modules without CFG support. Similarly, even in CFG-supported binary code, attackers can target inline assembly (which the compiler cannot protect), or look at opcode splitting (i.e. using unintentional instructions in the variable-length x86 instruction stream).

In Intel's case, one problem with IBT is that in addition to being only in the newest hardware, it is also only now getting integrated with compilers and operating systems. This means that in order for it to even be effective, the computing environment needs to have all three (the hardware, operating system, and compiler support). Thus this defense is not present in slightly older binaries or systems, which will continue to run into the near future. Beyond the short term, many devices will still not meet this criteria (consider embedded devices, edge devices, digital signal processors, etc.).

A recent 2019 Black Hat presentation from McAfee [38] lists multiple techniques for attacking systems that are running with Intel CET. These include: code replacement attacks, counterfeit object-oriented programming [39], data-only corruption, function pointer hijacking through race condition attacks, and thread context hijacking by abusing the NT-Continue mechanism. The authors also point out that Windows 10 still relies on software-based CFG (and is not fully integrated with Intel's IBT), which is more susceptible to bypasses.

In short, how to defend against reuse attacks is still an open problem. This is true at the cutting edge of new operating systems and hardware; it is true for programs running on legacy systems; and it is true for programs running on modern hardware and systems that do not contain these latest defenses.

1.2.4 Debloating Defenses

It can be safely concluded that attackers have either managed to dodge defense mechanisms for certain kinds of gadgets (ROP), or mechanisms do not even exist for certain other types of gadgets (JOP). In response, attack surface reduction [40] is one class of defense that has gained in prominence lately. Piece-wise compiler [41], Chisel [42], and Razor [43] are three recently developed debloating/attack surface reduction techniques that motivate our work. They successfully show how to reduce applications' attack surfaces, but they also have shortcomings.

Piece-wise compiler modifies the loading stage at process start-up to remove unreachable library code. It is sound, removing only unneeded functionality from libraries (for which it requires libraries' source code). No user input is needed, but piece-wise-compiled libraries must be provided to a program before running it. This approach removes function(s) in the library only if they are proved unreachable on a whole-application basis, i.e. from nowhere in the driving application can they ever be invoked. Due to complex control flow in the libraries and conservative limitations of static analysis, proving such a property becomes extremely difficult. As a result this work has not been demonstrated on glibc, a critical real-world library with vulnerabilities. Secondly, conservative static analysis leads to limited success in terms of attack surface reduction.

Chisel uses reinforcement learning (RL) to learn which parts of a program are actually used and needed and then builds a trimmed version of it. Chisel is an unsound technique. It relies on test cases and may learn a model that eliminates *needed* functionality. This can induce crashes and so is not practical for use in the real world since software under a no-attack condition could become unusable. Chisel works in a kind of compile-test-refine loop, as its learner identifies which parts of the program are needed in order not to crash or provide bad output. It requires a user specification (supplied as test inputs) and source code. Chisel is designed to work with application code.

Razor uses heuristics and test inputs to debloat binaries. Like Chisel, Razor is unsound

and can lead to crashes, which they report and tally in their experiments. Razor is designed for and works on applications, though they include a discussion on its current effectiveness on libraries. Similar to Chisel and Piece-wise, it also cannot debloat **may-use** code, i.e. code that may be used by the program under certain inputs/execution conditions. Neither of the previous two techniques perform verification nor model checking, leading to unsoundness.

Table 1.1: Broad properties of 3 state-of-the-art debloating techniques for security (Piece-wise, Chisel, and Razor).

	PW	Chsl	Rzr
Works on application		✓	✓
Works on library	✓		✓
Works on binary			✓
No user input needed	✓		
No training needed	✓		✓
Is sound	✓		
Can debloat may-use code			

We summarize and compare the characteristics of these three approaches in Table 1.1. With regard to practical usage, we find the last two rows strikingly important (i.e. whether it is sound, and whether it can debloat may-use code). Guaranteeing soundness under normal execution conditions is a must for any practical usage of software. An important issue with debloating *may-use* code is that the technique should be able to dynamically adapt as per the input. That is, during certain execution conditions, a complete segment of some code may be needed, but under certain others, none of that code may be needed. A debloating technique should ideally be able to handle both scenarios. *Solutions such as Razor, Chisel, and Piece-wise take zero-sum approaches towards may-use code: It is removed, leading to unsoundness and potentially crashes or bad output; or it is left in place under all execution conditions, leading to a higher amount of attack surface.*

This state of affairs stems from these frameworks’ underlying techniques. Piece-wise depends on program analysis and load-time dead code elimination from libraries, which, as mentioned, is sound but cannot overcome the inherent limitations of static analysis. Both

Chisel and Razor require a user specification of the functionality needed by the application. In both frameworks, this specification is provided as test cases. Not only can this requirement limit adoption for real-world use, in both of these works it is tied directly to the soundness. If the user does not provide the “right” set of test cases, then the debloated program may crash. In their evaluations, several such cases are shown, which is a significant issue. In Chisel’s particular case, it uses these test cases to build an RL model, which has the additional drawback of mispredictions. Chisel issues predictions at compile time, and mispredictions lead to unsound cuts to the program. It does not have any support to handle mispredictions in order to ensure soundness. *In short, the techniques from prior art have compelling aspects, but the conservative approaches are restricted by static analysis’ inherent limitations, and the more aggressive and predictive approaches do not offer any soundness guarantees.*

1.3 A Mixed Compiler and Predictive Approach

This thesis adopts the view that a *mixed compiler and predictive approach* to software debloating can improve on the limitations of prior work and ultimately thwart or substantially weaken code reuse attacks. To understand the reasons for adopting this view, we first consider the general tradeoffs of static and dynamic techniques in the context of software debloating.

Static analysis’ benefits are mainly twofold: performance and soundness. The cost of doing static analysis is typically paid only once at compilation time. Even a high cost may be acceptable, so long as the analysis has not impacted the application’s runtime. Soundness is not inherently guaranteed by static analysis and static transformations, but traditionally, it is an assumed property of the compiler – it errs on the conservative side. Soundness is fundamental for real-world scenarios. Without it, applications will crash or produce bad output.

On the other hand, dynamic techniques offer one key benefit for debloating: improv-

ing the attack surface reduction by leveraging runtime information. In the case of function pointers, for example, a dynamic technique has access to the actual address value at runtime, which opens the opportunity to restrict or debloat the program to only that section of code. Similarly, the dynamic context surrounding the function call has been used in multiple CFI techniques to reduce points-to sets and improve the defense. This has not been exploited as heavily in debloating techniques, which is one area this thesis attempts to tackle.

Static and dynamic techniques each have their drawbacks. In terms of static analysis' soundness guarantee, compilers overapproximate to obtain reasonable compilation times and also to "solve" undecidable problems. Pointer analysis is applicable to debloating and is a good example of both. It is undecidable [44, 45], but algorithms such as Andersen's analysis [46] manage to produce good results in a reasonable amount of time. The limitations of pointer analysis ultimately restrict the debloating one can do at build-time, because the points-to sets are conservative; function pointer analysis produces target sets that may include many more functions than dynamically possible. As for dynamic techniques, they typically come at a cost to performance. By definition they perform some action at runtime and so must pay some unavoidable overhead.

Machine learning (ML) is just one tool that can be leveraged as a dynamic technique. What makes it attractive, however, is that it can improve the debloating even further. ML models are capable of predicting control flow paths at runtime using the rich dynamic information available, and the unlikely paths can be eliminated from the code (thereby reducing the attack surface). In other words, a static technique can cut from the program unused code; a dynamic technique can temporarily cut from a running program code that is not immediately needed; but an ML-based predictive technique can temporarily cut from a running program code that is not *expected* to be immediately needed. The major downside of ML is that it comes with a degree of error. That error, if not handled properly, can lead to worse security, crashes, or other misbehavior like bad output.

In this thesis we contend that it is possible to perform software debloating in ways that reap the benefits of both (static) compiler and (dynamic) ML-based predictive techniques. We want to show that novel static techniques and prediction-enabled runtime frameworks can be combined in such a way that the solutions are sound; the attack surfaces are reduced significantly; and the performance degradation can be kept within limits for real-world application. We want to show how static analysis is capable of carving out substantially reduced subsections of the callgraph that a runtime can enforce. Static analysis can also give us deep insights about the program, such as the reverse dominance frontiers of values, and dynamic binary instrumentation can leverage those values at runtime. We can leverage ML to predict subsets of the (statically known) callgraph and course correct at runtime in the event of mispredictions. Furthermore, when there are mispredictions, we can check against static properties of the program, made available via compiler instrumentation, to ensure ML-based predictions conform to the original program’s semantics (and are not attacks). These are the types of synergistic combinations between the compiler and ML-based predictive techniques that we wish to explore in this thesis, and which we intend to show can be beneficial to software debloating.

1.4 Thesis Statement

This work aims to support the following hypothesis:

Compiler and machine learning-based predictive techniques can enable on-the-fly code remapping and dynamic callgraph prediction to debloat software in a sound and performant manner with lightweight static guarantees in order to mitigate code reuse attacks.

1.5 Contributions

1. We present BLANKIT, a binary-level framework for loading and unloading third-party libraries on demand. To reduce over-approximations, it leverages machine learning to predict the required functions that must be loaded at each library call

site. It can handle the GNU libc library, which historically has been challenging to protect and which has been used in notorious code reuse attacks.

2. We present DECKER, a source-level framework for mapping and unmapping application code on demand. It separates the static callgraph into function sets called “decks” which together can be mapped executable and later unmapped such that performance degradation is minimal and software debloating is still effective. Like BLANKIT, this technique is also sound.
3. We present PDSG, a source-level framework that combines static analysis with machine learning prediction for whole-application debloating. Using the insights from both BLANKIT and DECKER (in terms of function set prediction and static function set organization, respectively), PDSG is able to: (1) to refine whole-application debloating by predicting at runtime at specific points in the program only the *expected* subset of upcoming functions that will execute; and (2) to *verify* at runtime that the execution path through these subsets of the callgraph adhere to statically valid call sequences of the program.

CHAPTER 2

BLANKIT: BINARY-LEVEL, PREDICTIVE LIBRARY DEBLOATING

2.1 Introduction

Modern software relies heavily on libraries that are often built for supporting a large amount of functionality. In a given application, however, only a small amount of such functionality may get used. For example, programmers leverage Android libraries, lean on machine learning and AI tools, and build on top of web frameworks to improve software development productivity [47, 48, 49, 50, 51]. Although these frameworks can be dauntingly large, it is normal to use only a small subset of the APIs. A recent study [52] shows in userspace programs that ship with Ubuntu Desktop 16.04, only about 10% of the shared library functions are used. For performance reasons, these underlying libraries are inevitably written in C/C++. The lack of memory safety in C/C++ leads to a large number of exploits that expose the applications or frameworks that use the libraries, in turn leaving them vulnerable. One such library that forms the core of C/C++ libraries and applications is the GNU version of libc, or “glibc” [53]. Glibc also acts as a basic building block of other libraries and their higher-level APIs in GNU/Linux systems. Unfortunately, the list of vulnerabilities and exploits in glibc keeps growing, with 108 known and published vulnerabilities at the time of this writing [54], of which nine were published just last year. Moreover, glibc is a mine of return-oriented programming [9] (ROP) gadgets that can be stitched together to create semantically valid and malicious attacks. Although known CVEs are patched and remedied, new/future attacks are built over these gadgets. Hence, reducing the quantity and expressiveness of such gadgets for libraries like glibc is an important problem for which numerous strategies are still being invented.

2.1.1 Debloating

Current compiler-based link-time optimizations can remove unwanted or dead code from the application or library. They statically determine a set of functions to be removed from the linked library using a combined analysis of unreachable functions, global constant propagation, and inter-procedural evaluation of branch outcomes using the fixed points of unreachable code and (conditionally propagated) constants [55, 56, 57, 58]. Software debloating at call sites that use function pointers faces the same limitation of over-approximation seen in CFI, allowing construction of gadgets from available functions. Although static program debloating is a positive step towards thwarting unknown possibilities of attacks and towards reducing attack surface area, it is quite ineffective due to the limitations of static analyzability of programs. A lot of code remains linked.

Piece-wise compilation [52] improves purely static techniques by collecting address-taken functions of each module. At load time, it brings in only those functions that are deemed required by the program. While it reduces the number of functions at an attacker's disposal, dynamically unused functions are still loaded. It debloats between 60% to 86% for `musl libc`, leaving a number of ROP and JOP gadgets linked to the program. Further, most software debloating mechanisms require *source code analysis* and hence omit important, widely used libraries such as `glibc`, which is notorious for extremely complex control flow, poses a huge obstacle for inter-procedural analysis of reasonable sophistication (static or dynamic), and cannot even be compiled with `clang/LLVM` at present. In fact, the current `glibc` is not a realistic target for static linking. It uses `dlopen` to pull in additional libraries that *themselves* depend on `glibc`. Thus, even if one could statically link the main application with `glibc`, at runtime inadvertently another copy of `glibc` is dynamically loaded and linked.

2.1.2 BLANKIT: “Get what you want”

Current security debloating approaches can be classified into two categories: conservative ones that remove only those functions whose removal is deemed safe under the soundness

assumption (e.g. piece-wise compilation) [52], or aggressive ones, which break soundness and can lead to crashes (e.g. Razor or Chisel) [43, 59]. Conservative approaches leave a considerable amount of unwanted code linked to an application, significantly degrading security. On the other hand, aggressive but unsound approaches suffer from usability issues (such as crashes, and/or incorrect execution), making such schemes impractical. Guaranteeing soundness requires verification, which is intractable for any reasonably sized programs.

In response to the limitations above, we introduce BLANKIT. **BLANKIT is designed to get only the functions required at a library call site in the current dynamic calling context, rather than cut the statically unwanted functions.** This fundamental departure yields a high amount of debloating (almost 97%). BLANKIT relies on the following core ideas:

- It loads only the set of needed/predicted functions from the library at a call site on demand at runtime. The predictor, a decision tree-based learner, acts as an input-sensitive oracle and provides the list of required functions. The other functions in the library have been blanked out with invalid instructions.
- Library functions run under the supervision of probes, which fire before each function’s invocation. If more functions are needed for execution, they are loaded on demand.
- When the library execution returns to the call site, the complete call chain of loaded functions is purged.
- When a probe detects a misprediction, a mitigation mechanism is triggered. We list several schemes here and include an evaluation of a parallel audit process that reviews the alarm.

Differences between BLANKIT and other debloating schemes are illustrated in Figure 2.1. BLANKIT’s function set is often a small subset of the functions needed by a

program. This subset dynamically shrinks and grows, mostly staying within the innermost boundary of the needed functions. The prediction accuracy of BLANKIT is very high, but if BLANKIT under-predicts the set of needed functions, it is able to load them in a safe manner. In a very small number of occurrences, BLANKIT over-predicts by a small degree. Due to these reasons, BLANKIT achieves very aggressive debloating, diminishing the attack construction surface in libraries to near-zero while guaranteeing a very high amount of usability of continuity of execution (without crashes or incorrect execution). We show in section 2.5 and section 2.6 that BLANKIT has a very high security benefit to avoid currently known CVEs and vulnerabilities.

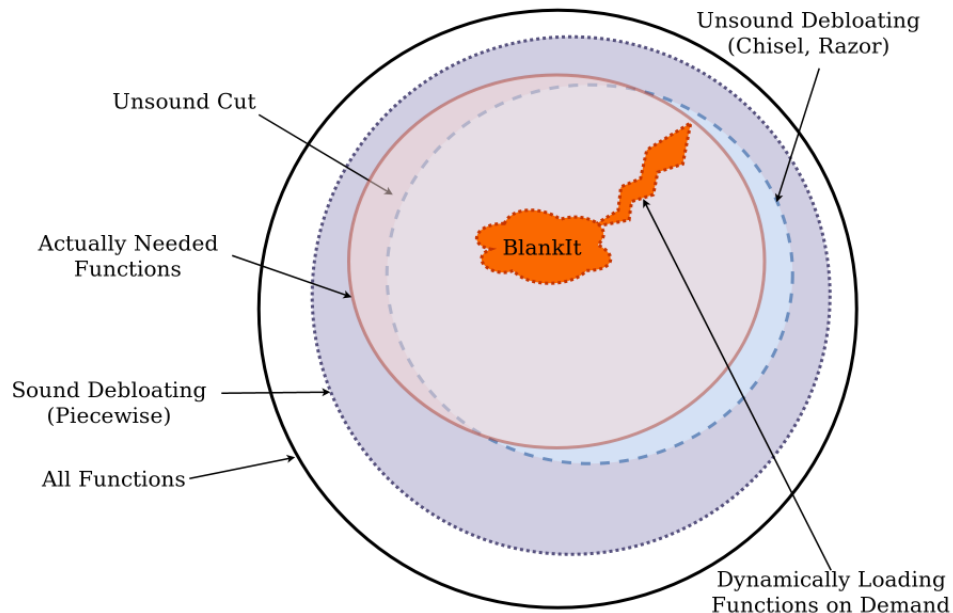


Figure 2.1: BLANKIT compared with other debloating mechanisms

2.1.3 Overcoming CFI Limitations

BLANKIT can be combined with CFI and should not be seen as a replacement. Though there is some functional overlap, BLANKIT can complement CFI and handle scenarios that CFI would otherwise miss. For instance, there are environments where many CFI techniques do not penetrate, such as glibc. BLANKIT's predictive approach can identify valid

jumps for `set jmp/long jmp` easily compared to CFI mechanisms. And most crucially, the prediction approach enables BLANKIT to tackle non-control data attacks, as well, which is outside the scope of all CFI implementations and CPI [21].

To illustrate, consider the non-control data attack from [31] shown in Listing 2.1. A user’s credentials are verified with an input key in function `service`, and the result is assigned to the variable `user`. The user is frequently checked to provide access to certain operations. Between checks, however, the function interacts with the program user. Input `someinput` can be manipulated by the program user, causing a buffer overflow at line 7, and leading to superuser access. This attack is not handled by CPI or the latest CFI (π CFI, μ CFI), but in the case of BLANKIT, the `call_super_user` function is not predicted, and the attack is detected in the audit and mitigation phase that checks for violations. Note that CFI combined with a software-debloating mechanism like piece-wise compilation [52] also cannot handle this attack. Another of BLANKIT’s advantages is that it is a binary defense mechanism and works on glibc, unlike most state-of-the-art defense mechanisms like CPI, π CFI, or CETS. BLANKIT is limited, however, to handling such attacks only when a function call is involved, since that is the prediction granularity it currently operates on.

Listing 2.1: Attack missed in CPI and μ CFI but caught in BLANKIT

```
void service(ulong key) {
    char str[SIZE], user[SIZE]; ...
    verify_user(user, key);
    if(strncmp(user, "admin", 5)) { ... }
    else{ ... }
    strcpy(str, someinput);
    if(strncmp(user, "admin", 5))
        call_super_user();
    else
        call_normal_user();
}
void access_service(char pwd[20]) {
```

```
int auth = verify_pwd(pwd);
if(auth)
B1:   key = calculate_key_su(pwd);
else
B2:   key = calculate_key_nm(pwd);
      service(key);
}
```

2.1.4 Threat Model

We assume that the program is not self-modifying, and an attacker can read/write the data section and read/execute the code section of a vulnerable program. We assume that the application source, LLVM compiler that instruments the application, the dynamic BLANKIT runtime, and the underlying hardware, OS, and loader are not compromised. Prediction calls to BLANKIT from user applications are protected against any argument tampering using pass-by-register semantics; further, these arguments are guaranteed to fall within a compile-time known prediction set. Application code pages are $W \oplus X$ (writable XOR executable), and BLANKIT maintains this property between copying and blanking operations to guard against code injection (and potentially probe insertion). All external libraries can be a source of threat, and any such external call must be protected. Untrusted application code can be modularized, however, into shared libraries and protected with BLANKIT, just like untrusted third-party libraries. We make no assumptions on obfuscated code or hand-written binaries or any other calling conventions.

We assume a protected runtime environment for BLANKIT, independent of its current prototype in Pin. These solutions have already been explored, e.g. CPI's in-process isolation [30], and a BLANKIT runtime can leverage hardware segmentation or software fault isolation [60] that guards runtime against attacks such as [61]. This prevents attackers from jumping into BLANKIT's runtime probes. Repeatedly calling BLANKIT's copy and blank functions does not change the state of the application nor increase dynamically linked func-

tions. Influencing or somehow steering the decision trees is impossible. The decision trees are embedded and compiled into application code and use read-only, scalar arguments. BLANKIT-inserted trampolines cannot be triggered by an attacker either. Such an arbitrary trampoline would be caught in BLANKIT the same way any arbitrary jump is caught (due to illegal control flow). Note that some of these attacks are also caught by CFI mechanisms with which BLANKIT can be augmented.

2.1.5 Limitations

Like other debloating techniques, vulnerabilities against loaded instructions are not protected by BLANKIT. For example, if an attack exploits a buffer overflow inside of a loaded library call chain, and then uses only the gadgets within that call chain, then the attack will go unnoticed by BLANKIT. Only by landing in unloaded memory, or by triggering a mispredict, does BLANKIT thwart attacks.

2.1.6 Contributions

The following are the contributions of this chapter:

1. A framework for dynamically loading and unloading library functions on demand at binary level.
2. A prediction mechanism within that framework for predicting the required library functions to be loaded at each call site, thereby reducing over-approximation.
3. An alarm and audit technique that on misprediction runs a memory safety check on the library call.
4. An evaluation of our framework using SPEC CPU 2006 benchmarks and its supporting libraries (glibc, libm, libgcc, and libstdc++), some of which are not handled in other approaches that depend on library source; and an evaluation on sshd and

nginx, with treatment of over 10 more libraries and support for multiprocessing and multithreading.

These result in the following security implications:

1. The set of active dynamically linked functions is reduced to protect against future, unforeseen exploits that mainly rely on code reuse and replay-based attack construction, and we provide reasonable metrics and results for the same.
2. The glibc library is addressed. It is notorious for vulnerabilities and not tackled by most state-of-the-art defense techniques. It is ubiquitous and a significantly higher-impact library than alternatives like musl [62].
3. Certain non-control data attacks are thwarted (as seen in the example in Listing 2.1 or the nginx case in subsection 2.6.4), which fall outside the scope of CFI and CPI.

2.2 BLANKIT's Overall Framework

BLANKIT is a compiler-aided runtime environment for demand-driven loading of library functions. At runtime, BLANKIT is initialized at program-start, when libraries are first brought into memory, and BLANKIT immediately does the following for every function: It inserts a probe at the function entry, copies the function code after the probe into a safe read-only memory, and wipes (blanks) out all the code after the inserted probe. Later during runtime, BLANKIT loads library functions on demand.

The application itself is instrumented beforehand. A decision tree learnt through off-line profiling is instrumented for every library call site in the application by the compiler (LLVM) pass. At runtime, the decision tree simply outputs the set of functions within the library that it expects to invoke at a given call site under the current dynamic calling context. That context consists of call site ID, reverse dominance frontier (RDF) of static single assignment (SSA) values of arguments to the call, and the library function's arguments themselves. When the application calls a library function, the probe (inserted when

the library was imported) gets invoked. This probe checks if the functions called are as predicted. If they are, execution continues in the loaded functions. If, however, the call is not through legitimate means (non-predicted and not legal) then an attack is detected. Any attempt to bypass the probe results in a fault or crash, because the code section has been blanked. Finally, when the control returns to the application, the functions that were copied are reset back to a blanked state. This purging is what minimizes dynamically linked functions.

A simplified diagram of BLANKIT’s defense is depicted in Figure 2.2. It shows that in the normal runtime A, an overflow vulnerability can be exploited to jump to an arbitrary address within glibc, but in runtime B with BLANKIT, since the library functions are wiped out, jumping to an arbitrary address results in a fault. The prediction mechanism reduces the set of functions that can be reached through the vulnerability.

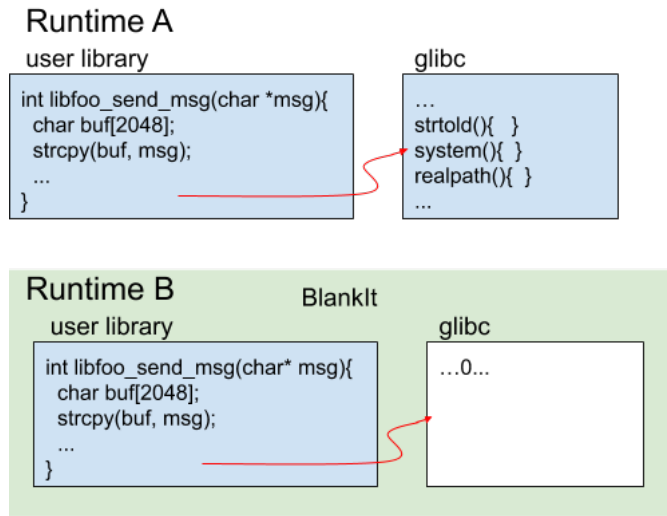


Figure 2.2: An application running (A) without BLANKIT and (B) with it. The example is based on the return-to-libc attack from [3].

A highly accurate yet lightweight prediction framework is critical to BLANKIT’s success. We compute the predicates on which the call site and call arguments are control-dependent (using reverse dominant frontier), and generate the model based on these predicates and argument values. We empirically show (based on the intuition of static value

separation via SSA) that a model based on reverse dominance frontiers and argument values is accurate and lightweight. This approach provides a highly likely subset of the may-reachable set of functions from a given call site, which itself is a function of arguments of the call site summarized by the static-dynamic artifacts in the application. A lightweight decision tree based on these artifacts allows us to predict the may-reachable set of functions at each call site. More details on this are in section 2.4. Both under- and over-predictions can occur. An under-prediction is when a function is invoked that was not loaded. It *may* be an attack, so it raises an alarm. An over-prediction is when a function is predicted and loaded but not invoked (which could lead to reduction in security since it is an over-approximation). Over-prediction is very rare (as shown later in section 2.5) and is much better than static approximation of reachable functions, as only valid callees are loaded. When under-prediction occurs (which is also rare due to high prediction accuracy), BLANKIT starts an alarm and audit phase to check for an attack. The library function and its arguments are handed over to a process that runs the library function with a memory safety mechanism, which then checks if the misprediction is an attack.

2.3 BLANKIT’s Runtime

BLANKIT’s runtime wipes out all functions in a library and loads them only on demand. The predictor that guides this Pin-based runtime is described in section 2.4.

2.3.1 BLANKIT Design

BLANKIT leverages Intel Pin [63], and its design adheres to Pin’s probe mode and programming model. Probe mode allows for fast, native execution (as opposed to Pin’s JIT mode). At initialization time, BLANKIT iterates over all of the executable’s shared objects. Then for each function within a shared object, it overwrites the first few bytes with a trampoline and blanks the remaining bytes with illegal or invalid instructions. BLANKIT maintains a separate copy of functions in a trusted cache. At runtime, trampolines bounce execution

into a generic handler, or in Pin-speak, a probe. The BLANKIT probe copies the function's bytes back into place (i.e. from the end of the trampoline to the end of the function) and returns to the function.

Figure 2.3 depicts this mechanism for a two-function call chain. When the application calls a library function (`malloc` in this case), execution proceeds along arrow 1 to the original function location in `libc`. Note that during initialization, however, BLANKIT had replaced it with a trampoline. The trampoline causes execution to flow along arrow 2 to the `probe_copy` function. Focusing first on lines 6-9, a loop over a map `m` copies and remembers all functions of the currently predicted set. `malloc` and `mmap64` are needed, so they are copied back into place in `libc`. After returning from the `probe_copy` function along arrow 3, execution flows along arrows 4, 5, 6, and 7 when `malloc` invokes `mmap64`.

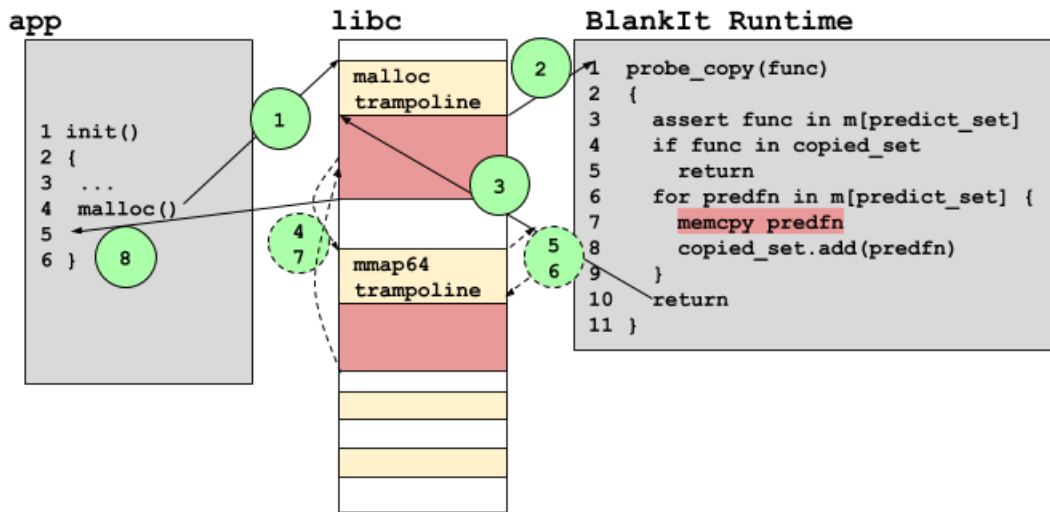


Figure 2.3: An example of how the copy probe works. Execution flows in order along the arrows.

Notice that the trampoline in `mmap64` is still taken (the two small dashed arrows at 5 and 6), even though its code is already in place and no additional copying is needed. This is handled in lines 4-5 in `probe_copy`. There is no blanking at the end, either. That is, arrow 8 proceeds back to the application. The copied functions are blanked during the prediction of the next library function call chain, but only if the predicted set has changed. This *lazy*

blanking optimization trades some security (by allowing the last call chain in the library to remain unblanked and exposed until the next library call) for speed. It still enforces security across call sites, though, since no two call chains remain linked to the application at the same time.

A second type of probe (Figure 2.4) is used for both blanking and prediction. In a BLANKIT-enabled application, prediction calls are hoisted before the library calls. The predictions are *themselves* library calls (to libBlankIt.so), though, and BLANKIT simply replaces them with a probe during initialization. In Figure 2.4, the application calls a library function (`free`), so at compile-time, a call to `blankit_predict` was inserted with an ID of the profiled/learned call chain at that call site. The initialization for BLANKIT replaced all `blankit_predict` calls with `probe_blankit_predict`, which allows it to blank the last set of copied functions (lines 4-5) (lazy blanking) and update the ID of the new prediction set (line 6).

`probe_blankit_predict` then returns along arrow 2 to the application, where it invokes the library function, `free`. At this point, the behavior is similar to that described in Figure 2.3. Any misprediction raises an alarm and invokes an audit mechanism.

2.3.2 Handling Misprediction

The call flow deviates from the predicted path either because of an attack or misprediction. BLANKIT's runtime must distinguish between mispredictions and attacks, and there are several existing approaches that can be plugged into BLANKIT with zero to few changes.

Different misprediction handling approaches have different security-performance trade-offs. The right approach for handling the alarms depends on the application and the level of security required. If the application is 100% predictable, then even killing the process is an option. In our experiments, we show our prediction is 100% accurate in a few applications. Another option is to use a virtual machine (VM) with checkpointing, and on mispredict, restore the application with heightened monitoring [64]. Yet another option is to feed the

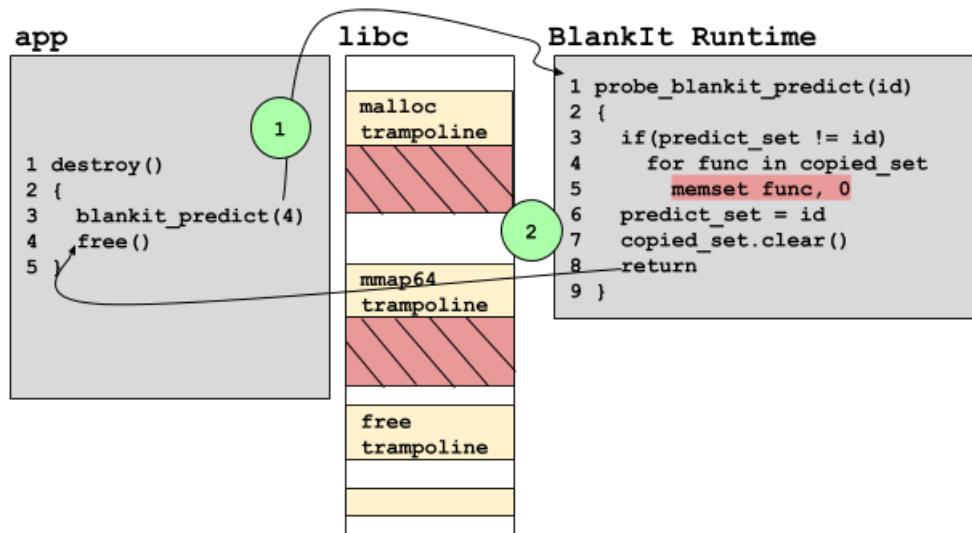


Figure 2.4: A prediction call that passes an ID of 4 for the upcoming predicted call chain. Within the prediction probe, the last chain of exposed functions will be blanked if the predicted set has changed. In this case, it has changed (from `malloc` and `mmap64` to `free`), so blanking occurs.

application state into a memory forensics tool.

Memory forensics is an active area of research with a number of tools that could serve as a drop-in solution for handling BLANKIT alarms. One recent tool in particular, CREDAL [65], is a forensics tool that, given a core dump, can identify a data dependency mismatch (indicating an attack or corruption) with 80% success rate. 6% of the failure cases can be resolved by unwinding the stack as required by CREDAL. Further improvements in CREDAL-like systems will reduce the gap in the remaining 14% failures.

In this chapter, we illustrate one more option: an audit mechanism that checks for memory safety [29]. BLANKIT enters an audit and alarm phase on mispredict, in which the library function arguments and the function name are passed to an auditor process. The auditor loads the library with `dlopen` and calls the library function while executing with a memory safety mechanism. Here we consider Valgrind’s Memcheck [66, 67]. Memcheck detects only pointer targets that fall outside of valid allocations, but an experimental extension, SGCheck, can be enabled with `--tool=exp-sgcheck` to handle global and stack overruns. So while Memcheck is not complete, we chose to demonstrate BLANKIT with it

because it is an easily available off-the-shelf tool for verifying the feasibility of a memory safety mechanism. It also suffers from high overhead ($\sim 20x$), making it a useful data point for a “heavy-weight” solution.

We outline the three cases that have to be handled for Memcheck integration to work properly: library calls that have scalar arguments, that have pointer arguments, or that maintain state. Scalar arguments are the simplest. At runtime and on mispredict, the arguments are passed directly to the auditor. Pointer arguments require the auditor to track metadata of the pointer’s start address, type, and size. These can be captured from calls such as `malloc`. An example is passing metadata on the `src` and `dest` pointers in a call to `memcpy`. Library calls that maintain state cannot be independently repeated by Memcheck with just the input arguments. These calls have to be identified prior to runtime, and Memcheck must repeat all calls needed to recreate the state. In order to reduce work, stateful calls only need to be repeated up to the last “reconciliation point” (i.e. wherever the last mispredict occurred and the global state was last synchronized within the auditor).

2.4 Call Graph Prediction

BLANKIT runtime is driven by the prediction component, which is concerned with determining the expected flow through the libraries. Distinguishing between input-specific, legitimate dynamic control-flow and an illegitimate counterpart is critical to thwarting attempts to construct a malicious security apparatus inside the library. The goal is to build an oracle that takes into account the input relatedness and predicts the functions that correspond to the valid control flow.

2.4.1 Overall Prediction Framework

1. The static divergence analysis finds statically non-divergent functions for which no prediction is needed.
2. We instrument the reverse dominance frontiers (RDFs) corresponding to every argu-

ment at every call site to construct a call context for the library call.

3. The instrumented application is profiled, and the call context and argument values are logged along with the call chain.
4. This profile information is fed into a machine learning model that constructs a decision tree to predict the call chain depending on the call context and argument values.
5. The generated decision tree is then instrumented into the application at the application call site for prediction.
6. Constant folding and dead code elimination is carried out to remove redundant predicates.
7. The instrumented application feeds the predicted call chain into the BLANKIT runtime system.

2.4.2 Static Call Flow Divergence Detection

In the first step of the prediction framework, we find library functions that do not have statically divergent call flow. If a library function is non-divergent, all dynamic calls of the function will result in the same call chain, making prediction trivial. We find non-divergent functions by checking if every call site within a function post-dominates the entry block of the function, as shown in Algorithm 1. If the callee is not on a control-divergent path but is itself divergent, then the caller is also marked as a divergent function, and all other functions that call this function are also marked as divergent functions.

In glibc, we found that among all the functions analyzed, 737 functions (roughly 27%) are statically non-divergent. For the remaining 1985 functions (73%), dynamic prediction is a must.

Algorithm 1 Static Divergence Detection

```
1: procedure NON-DIVERGENT(Function F)
2:    $BB_e \leftarrow$  entry basic block in  $F$ 
3:   for each Call site  $C_i$ :
4:      $f_{ci} \leftarrow$  function called in  $C_i$ 
5:     if NON-DIVERGENT( $f_{ci}$ ) &&  $C_i$  post-dominates  $BB_e$  then
6:       F is non-divergent
7:   End for
```

2.4.3 Reverse Dominance Frontier-Based Prediction

After generating a list of statically non-divergent functions, we now look at predicting the call graph for the diverging ones. Normally, whole program path profiling [68] would need instrumentation of every branch. This would result in a path prefix of a fixed window W , entailing the context of the last W branches. This path prefix can then be used for prediction of the call chain within the function call, but this elaborate model could also pose very high overheads. We leverage the fact that library functions are usually control-divergent due to their arguments and contend that the static context required to predict the call chain is dependent upon the control dependence of the arguments passed to the library.

This motivation leads us to investigate the reverse dominance frontier (RDF) of the arguments of a call site, because the divergence in the library call path can be caused by the different definitions of arguments (determined by their respective RDF) reaching the call site. For example in Listing 2.1, the input to function `service` in `access_service` has one argument `key`, which is defined at basic blocks $B1$ and $B2$. If $B1$ is executed, then `key` has admin access, and if $B2$ is executed, it does not. In this case, both $B1$ and $B2$ are control dependent on the branch `if(auth)` at Line 14. Thus the branch condition decides the functions invoked within `service`. The intuition here is that the call chain inside a library call will be dependent on arguments sent to the library, which in turn could be statically separated into corresponding SSA variables of a backward slice that are finally guarded by the control-dependent branches. With RDF-based prediction, the context leading up to a call site consists of only the last executed RDF for every function

argument. This minimal instrumentation has enough information to predict the call chain within a function with reasonable accuracy. When the profiling is done, the context for each call site is constructed separately, depending on the RDFs of its arguments. The instrumentation methodology is outlined in Algorithm 2.

Algorithm 2 RDF-based Instrumentation

```

1: procedure INSTRUMENT_FUNCTION(Func)
2:   BBs_to_instrument  $\leftarrow$  Null
3:   for all callsite(Ci)  $\in$  Func do
4:     Fci  $\leftarrow$  function called in Ci
5:     for all argument Ai  $\in$  Fci do
6:       def_worklist  $\leftarrow$  get definition of Ai
7:     for all instruction defi  $\in$  worklist do
8:       phi_instr_set  $\leftarrow$  TRACE_PARENT_PHI(defi)
9:        $\triangleright$  call returns a set of phi instructions
10:    for all  $\phi_i \in$  phi_instr_set do
11:      for all incoming_BBi  $\in$   $\phi_i$  do
12:        rdf_set  $\leftarrow$  GET_RDF(incoming_BBi)
13:      for all rdf_BBi  $\in$  rdf_set do
14:        for all Successor Succi  $\in$  rdf_BB do
15:          BBs_to_instrument  $\leftarrow$  Succi
16:    return BBs_to_instrument
17: procedure TRACE_PARENT_PHI(instruction)
18:   phi_instr_set  $\leftarrow$  Null
19:   for all Operands opi  $\in$  instruction do
20:     if opi is_a phi instruction then
21:       phi_instr_set  $\leftarrow$  opi
22:     else
23:       phi_instr_set  $\leftarrow$  TRACE_PARENT_PHI(opi)
24:   return phi_instr_set

```

2.4.4 Argument Value-Based Prediction

The control path in library functions can diverge because of the various values of the arguments that are reaching the call sites in the application from the same RDF. Since the RDF remains constant, RDF-based prediction might not be sufficient. For example, we have observed that math library functions like `sqrt(x)` and `exp(x)` have different call chains

depending on the range of the parameters. In order to handle such cases, we must classify the values of the function arguments that reach from a single RDF, so we also include the values of the arguments captured by profiling in training the model. The value-based profiling also captures the values of the function pointers. Thus if a function pointer is passed as an argument to another function, then we are able to predict the function invocation based on the function pointer values.

2.4.5 Decision Trees

To predict the library call chain at each call site using RDF and call arguments, we chose decision trees as our machine learning model. Decision trees can be easily embedded into the application as few predicates without increasing the application runtime overhead. First, using LLVM, we instrumented the calls for recording the call context for every library call site within the application. The instrumented application was run using Pin JIT to generate the profile trace (the library callgraph), which along with the call context is then parsed to create the training data. The output of our model is the list of functions called within each library. The training data consists of the following features:

- Application call site
- Top-level library function called from application (callee name)
- Reverse dominance frontier of arguments
- Arguments passed to the library call.

For example in Listing 2.1, the decision tree learns that on a call to `service` from function `access_service`, `call_superuser` is called only when RDF B1 is taken or `key < 750`, as shown in Figure 2.5. We leverage the python-based scikit library [69] (`sklearn.tree.DecisionTreeClassifier` API), to learn a decision tree over the training data. The decision tree model is then written to a file in a special text format, which

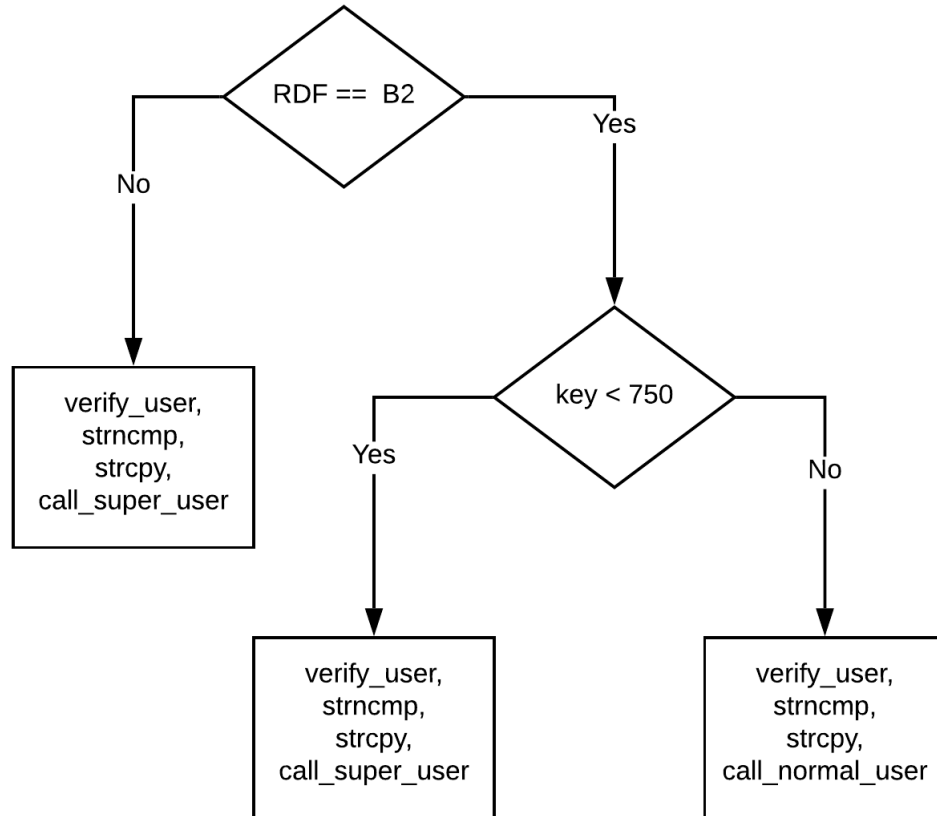


Figure 2.5: Example decision tree for Listing 2.1

is read by an LLVM pass that embeds the decision tree within the application for each library call. The decision tree is basically a binary tree of predicates of input features. The tree follows a certain path at each predicate based on the predicates’ outcome, and halts when it reaches a leaf node. The leaf’s “class label” is the set of functions within the library predicted to be called at the call site. We experimentally found that a max tree-depth of 10 provided good accuracy with acceptable performance. Higher depth would only be required when there are too many rows of the same input features but different outcomes, though this would also add the dilemma of overfitting the training data. We did not use boosting for learning the decision tree.

2.5 Evaluation 1: SPEC CPU 2006

We evaluate BLANKIT using SPEC CPU 2006, followed by two real-world applications in the subsequent section.

SPEC has several properties that make it a good candidate for exercising and evaluating BLANKIT. First, it is performance-oriented, so we can measure runtime overheads on realistic CPU-intensive benchmarks. Second, it has a standard library profile, which carries a relatively large CVE list. Lastly, callgraphs of SPEC benchmarks and their supporting libraries are significantly divergent, which is needed to stress the prediction framework (see Table 2.1).

Table 2.1: Key graph metrics for SPEC CPU 2006 and glibc

SPEC benchmark and glibc graph metrics	Avg
1. # dynamically called glibc funcs from SPEC	63
2. # statically reachable funcs in glibc based on (1)	171
3. # callgraph edges based on static callgraph in (2)	378
4. # non-divergent functions in (2)	8
5. max static callchain depth of (2)	7

With the exception of the linker shared object and the vdso image, we have instrumented all the benchmarks’ shared objects. The most important is glibc-2.26 [53], a fundamental library for programs compiled on GNU/Linux systems. The library provides POSIX, BSD, OS-specific and other APIs including utilities like print, login, and crypt. Historically, it has also been targeted and exploited heavily. The other libraries are libm 2.26, libgcc_s from the libgcc1 package v1:7.2.0-1ubuntu1 16.04, and libstdc++ 6.0.24, which are all tackled by BLANKIT. Our runtime uses Pin v3.6. We have built SPEC with version 5 of LLVM, which fails to compile 400.perlbench. It generates multiple definitions for `gnu_dev_major`, `minor`, and other functions, so we have elided treatment of it here. All other C/C++ benchmarks are presented. The raw decision trees have multiple compile-time constant checks. Constant folding reduces these trees to relatively few *if-else* checks, increasing the binary sizes by at most 1%. All runtime overhead experiments are on an Ubuntu 16.04.3

LTS machine with an AMD Ryzen 7 1800X 8-Core 3.6 GHz processor and 32 GB DDR4 2666 MHz RAM. All the SPEC benchmarks were trained on “test” (small) and “train” (medium) input data sets and were tested using the (large) “ref” input data sets. All runtime experiments are averaged over three runs.

2.5.1 Security

What constitutes an attack surface, especially when preparing for unknown future attacks, is difficult to capture. We present three results for reducing dynamically linked functions. Taken together, they offer three different axes for understanding how BLANKIT is behaving. They summarize (1) the number of functions that are exposed, (2) the number of gadgets in a known type of attack that are exposed, and (3) the number of known functions from a vulnerable list that are exposed. As additional evidence, we also provide a measure of gadget expressiveness, i.e. how easy-to-use or useful a gadget is to an attacker. We show how BLANKIT not only dramatically reduces the number of gadgets available to an attacker but also makes the available gadgets more difficult to use for an attacker.

The first measure is a dynamic metric that describes the maximum number of library functions exposed at any given time during execution. This can be described by the following formula:

$$exposed = p + s + c \tag{2.1}$$

where *exposed* is the maximum number of loaded functions at runtime; *p* is the number of functions that Pin is unable to instrument and thus must be left as they are (without blanking them); *s* is the number of functions that are less than 14 bytes and therefore too small for blanking; and *c* is the maximum-length call chain for a given benchmark that our framework dynamically loads at any call site during execution. In other words, the number of exposed functions in the worst case is the maximum number of unblanked functions that

could be leveraged in some attack. The percent reduction of code surface is then given by:

$$reduction = \frac{\sum_l n_l - exposed}{\sum_l n_l} * 100 \quad (2.2)$$

where l is some library and n_l is the total number of functions available at runtime in some library l . This metric does not capture anything about the inherent weaknesses or strengths of the exposed functions - only that they are exposed. Table 2.2 presents this result in the first column. The worst-case reduction is, on average, 97.1%, which is significantly higher than any static or dynamic cutting/debloating technique. For example, worst-case function reduction in piece-wise [52] is 60% and 46.09% on musl-libc and C++ libraries, respectively.

Table 2.2: Security-related improvements in SPEC CPU2006

Benchmark	% Exposed Code Surface Reduction	% ROP Gadget Reduction	% glibc CVE Function Reduction
401.bzip2	97.7	98.9	95.7
403.gcc	97.2	97.8	95.7
429.mcf	94.5	95.9	95.7
433.milc	98	98.5	95.7
444.namd	97	96.4	93.6
445.gobmk	95.7	96.2	93.6
450.soplex	97.4	99.3	97.9
453.povray	96.9	97.7	93.6
456.hmmer	97.9	97.9	93.6
458.sjeng	97.8	98.9	95.7
462.libquant	97.9	98.6	95.7
464.h264ref	97.9	98.7	95.7
470.lbm	97.8	98.6	95.7
471.omnetpp	95.6	96.9	95.7
473.astar	96.6	96.6	95.7
482.sphinx3	97.6	98.2	95.7
483.xalanc	96.9	98.1	91.5

Return-oriented programming depends on ROP gadgets in the code in order to carry out an effective attack. To measure the reduction in gadgets, we leveraged ROPgadget [70],

an analysis tool for enumerating the ROP gadgets in a binary. Because BLANKIT is a dynamic technique, the number of gadgets varies over time. Thus, similar to before, we choose the worst-case scenario at any given point during the program execution and report its ROP gadget reduction. We calculate the maximum number of exposed gadgets (similar to equation 1). Then we calculate the reduction by summing over the number of gadgets in the text section in each library, subtracting out the exposed gadgets, and gathering the percentage (as in equation 2). Table 2.2 shows the benefits of BLANKIT on ROP gadget reduction. The average is 97.8%.

The last metric is a measure of the functions in glibc CVEs that are removed by BLANKIT. We reviewed the list of all the CVEs for glibc, which reach back to year 2000, and we identified all unique functions mentioned in the descriptions. Of these, we identified 47 that are unequivocally loaded by glibc in the SPEC suite. That is, there must be an exact match in the dynamically loaded list of glibc functions for a function related in a glibc CVE to be considered. For example, “alloca” is mentioned in CVE-2015-1473, but it is not explicitly exported with that name and so is discarded, even though there are multiple allocation functions. Then the number of exposed CVE functions is obtained as follows:

$$exposedcve = p + s + a \tag{2.3}$$

where p and s are uninstrumented due to Pin or too small to instrument (as before), but a represents *any* function that is called that is in the CVE function list. The percentage is then taken out of the 47 CVE functions. This metric should not be misunderstood as the number of CVEs since 2000 that are thwarted by BLANKIT. Rather, it classifies functions as vulnerable based on their CVE history, and then asks how many such functions are exposed under BLANKIT running a common benchmark. The results are shown in the last column of Table 2.2. The average reduction is 95.1%.

We also measure the quality of gadgets with and without BLANKIT using Gality [71].

Gality provides an overall score for a notion of the ease with which one could construct an attack. Gality first categorizes the gadgets (arithmetic, data move, etc.), finding the distribution of different gadgets that an attacker could use in their arsenal, and then checks for side effects of a gadget. A side effect is measured by the number of preconditions that must be satisfied for using a gadget. In other words, a high-quality gadget is one with no preconditions or that does not affect any other register or memory. The Gality score starts at 0 and is increased for side effects and preconditions. Thus, a higher score indicates worse gadget quality and hence better security. The authors of Gality use the value of 1 as a cutoff and indicate that gadget quality below 1 is of “high quality” to an attacker. To summarize, the BLANKIT-enabled SPEC applications have higher scores in all cases (with an average of 1.09, versus 0.77 without BLANKIT), signifying improved security. In 15 out of 17 cases, scores are near or above 1, implying high security.

The metrics in the community for assessing the gadget quality are still limited, a concern that has been expressed in previous CFI works (e.g. [15]), and other metrics have been proposed (e.g. the AIR metric in [24] for indirect jump reduction). [72] recently attempts to carefully quantify gadget expressiveness. Although only one gadget can be sufficient to hijack a system, our gadget reduction and quality results are nevertheless useful for comparison with other similar debloating works such as [43, 73, 52, 74].

Lastly, note that multiple applications have their own private copy of shared libraries at runtime. Thus, even if multiple BLANKIT-enabled applications run simultaneously and unblank disjoint library function sets, process isolation prevents any “union of exposure” among them.

2.5.2 Performance

Figure 2.6 shows the runtime overhead. The most crucial library for security purposes is glibc. The average slowdown for a BLANKIT-enabled application on only glibc is 16%. There is a slight speedup in the case of lbm attributed to instrumentation effects on mem-

ory alignment as in other security works [30, 26, 24]. The worst case is gcc, with a 1.76x overhead. Note that runtime overhead is not a function of misprediction in the current implementation, as alarms are handled in parallel. Thus, while prediction accuracy was poorest for libquantum (discussed in the next section), the performance was good. Covering all libraries costs little in overhead for the extra security. The average slowdown for BLANKIT over all libs is 18%, demonstrating that BLANKIT likely scales well across programs' dynamic library sets. (Note that in bzip2, gcc, mcf, and sjeng, libc is the only dynamically linked library.)

The runtime overhead for BLANKIT depends on several factors. We see that BLANKIT invokes (and handles) a non-trivial number of dynamic library calls for SPEC: All but four benchmarks invoke over 1M calls, and another four make over 100M calls. The timing is more complex than just the dynamic call counts, though. For instance, sjeng makes 24,766 dynamic library calls, but its performance is 1.5x worse than the native baseline; compare this with povray, with 4,323,690,635 dynamic calls and 1.1x slowdown. The overhead depends on the ratio of the program's runtime spent in libraries, and it also depends on the ratio of the time to copy a function (which depends on its size and other factors like alignment) versus the time to execute that function. Optimizations like lazy blanking can help, by leaving functions unblanked when they are called successively. We looked at the effect of the lazy blanking optimization, and hmmer suffered the most without it (4.61x slowdown when disabling lazy blanking); gcc saw the least benefit from it (1.05x slowdown when disabling it).

We looked at the latency of the BLANKIT probes to see where it was adding time. The trampolines and probe logic are low costs ($\approx 1\mu\text{s}$), but un/blanking dominates due to memset/cpy (up to 10s of microseconds without lazy blanking). For example, in gcc, the average runtime cost of a full blanking probe is roughly 5x compared to a lazy one (i.e. a trampoline + memset vs. just the trampoline). Similarly, a full unblanking probe versus its lazy counterpart costs over 10x. For perspective, the absolute time, on average, to unblank

code was $2.57\mu s$ in our gcc experiments (max of $37\mu s$), versus $0.22\mu s$ in lazy cases. We considered a simple dynamic runtime scheme that would keep in memory only the union of functions seen during profiling (on a call-site basis). This increased loaded library functions in SPEC by 58%. This hurts security, but the increased footprint also hurts performance. Because un/blanking costs dominate, prediction is necessary to reduce the memory copying and setting.

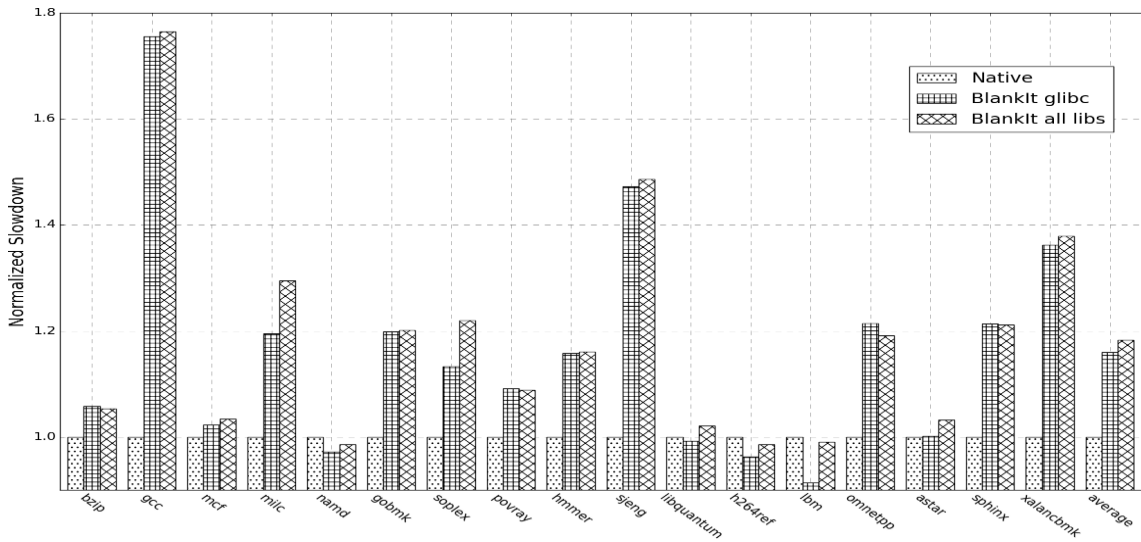


Figure 2.6: Runtime slowdown for BLANKIT on SPEC 2006 CPU, normalized against native.

2.5.3 Prediction

Our treatment of prediction includes our observations while working with the framework, a breakdown of the mispredictions, and our accuracy results and analysis on SPEC.

Observations

In the case of library functions that usually do not have an argument context for library function calls, we have observed that the set of functions called within the library can be predicted based on the dynamic calling context. The necessary dynamic calling context is discovered by the decision tree, which can be as simple as just the name of the function

and/or call site. However, sometimes the call chain is dependent on the arguments passed to the function, which could include a function pointer. For example, there might be a math library function that calls a different set of internal functions based on the arguments. The decision tree is able to handle such cases by training on the values passed to the function. It is able to model the different call chains that are invoked based on the range of values of the function parameters encountered during profiling. The decision tree is currently unable to capture other use cases like certain special system states or error conditions that were not triggered during profiling, which is a challenge of any machine learning approach.

Breakdown of Misprediction in SPEC

We classify the mispredictions into over-prediction (where more functions were loaded than necessary) and under-prediction (where fewer functions were loaded than necessary). In all but two cases, under-predictions accounted for 100% of mispredictions. The exceptions are gcc, where the under- and over-predictions were almost equal (0.5% difference), and soplex, where under-predictions accounted for 93% of the cases. The main reason is that when the decision tree encounters a call site which was not exercised during training, it is extremely conservative and bases the prediction on whether the entry function was seen during training, and predicts a callee set which is a subset of the static callee set for an entry function. This in turn provides good security, since no spurious functions are brought in that could increase the exposed code surface.

Accuracy and Audit Overhead

Table 2.3 shows the call chain prediction accuracy and audit overhead for the SPEC2006 benchmark suite. The audit overheads are based on the expected runtime of *parallel* Valgrind threads on each mispredict. In summary, only two benchmarks would require more than 1 parallel Valgrind process to keep pace with mispredicts. Over half of the accuracy figures are 97% or greater; three are 100%; and the average is 94.3%. In fact, the decision

tree’s prediction ability was tested for predicting every function in SPEC2006 at every call site and was found to be very close to the one reported here for the libraries.

Table 2.3: Prediction accuracy and audit overhead

Benchmark	% Accuracy	Audit (μs)
401.bzip2	91	287
403.gcc	99	12
429.mcf	94	92
433.milc	100	27
444.namd	99	11
445.gobmk	84	8
450.soplex	92	26
453.povray	97	6
456.hmmer	98	6
458.sjeng	97	27
462.libquantum	60	41
464.h264ref	100	28
470.lbm	98	171
471.omnetpp	99	6
473.astar	100	58
482.sphinx3	99	11
483.xalancbmk	97	18

One important reason for very high prediction accuracy stems from the fact that libraries follow software design patterns; the use cases of libraries are therefore finite and can be siloed. Our decision trees are able to learn these input-related patterns and during testing pinpoint the silo and its underlying set of function calls. The only application for which the decision tree accuracy was relatively low is libquantum, in which a call chain involving function `cfree` in training is followed by `munmap` during testing because of the size of the memory that is freed in `cfree` in the train vs. ref input.

2.6 Evaluation 2: nginx and sshd

We evaluate two real world applications: nginx v1.15.10 [14], a common server and reverse proxy tool, and sshd v8.0p1 [75], the server daemon for the openssh communication and encryption tool suite. Both of these applications serve as critical infrastructure in today’s

software landscape, and both of them suffer from real-world attacks that exploit vulnerabilities in linked libraries. We evaluate them on security, performance, and prediction, and study BLANKIT’s defense against well-known attacks. Note that nginx is both a multiprocessing and multithreaded application. In a multithreaded environment, un/blanking can be implemented with `mremap` without marking the page writable. This protects against a nefarious thread attempting to write to a page that is being unblanked.

To train BLANKIT’s prediction mechanism for nginx, we used a basic test page (only html and body tags that reads “site under construction”), and made requests for 3s. To perform the requests, we used a traffic generation tool called wrk [76] (used by Nginx, Inc. for benchmarking). We ran wrk using 12 threads with 400 connections. To train sshd, we used a simple ssh payload script which echos “hello world” on the server and sends it back to the ssh client; we also spawned 1000 such concurrent connections in order to train with sshd’s error handling.

2.6.1 Security

Our security results for nginx are based on running a 30s test with a standard 80KB file, the main page for Wikipedia [77]. In the case of sshd, we use a shell script that implements Dijkstra’s algorithm to solve a shortest path graph problem that executes for approximately 40s. Table 2.4 shows security results for nginx and sshd similar to those achieved in SPEC. Exposed code surface reduction is 95.8% and 89.5% for nginx and sshd, respectively. Less than 2% of the ROP gadgets remain exposed. Gality gadget quality under BLANKIT is again greater than 1 (while the unprotected version is ~25% worse). The number of gadgets in the nginx and sshd libraries are 27,523 and 30,752, respectively, and the maximum number of exposed gadgets under BLANKIT are merely 299 and 562. These two applications also offer a look at the effectiveness of BLANKIT on additional libraries. Linked libraries in nginx were libc, libcrypt, libdl, libpcre, libpthread, and libz; in sshd, they were libc, libcrypto, libdl, libutil, libz, libcrypt, libresolv, libssh, libnss_compat, libnsl, libnss_nis, and

libnss_files. We ignored only the vdso image and the linker shared object.

Table 2.4: Security-related improvements for nginx and sshd

Benchmark	% Exposed Code Surface Reduction	% ROP Gadget Reduction	% glibc CVE Function Reduction
nginx	95.8	98.9	93.6
sshd	89.5	98.2	91.5

2.6.2 Performance

For nginx, we ran three performance experiments. Table 2.5 shows the effect of running nginx under a constant load for varying amounts of time, verifying that it is not degrading. BLANKIT overhead is in the range of 1.06x to 1.09x. We used the Wikipedia main page for this experiment. Table 2.6 shows the result of running BLANKIT under the same 12-thread, 400-connection workload, but for 30s under varying payloads. Here we used random, binary payloads of size 1MB, 10MB, and 100MB. We see that running nginx with BLANKIT scales with the sizes of the requests. We checked the CPU utilization and instructions per cycle (IPC) with perf for these experiments. We found no noticeable difference in CPU utilization. In the former experiment (varying length of sustained time), the IPC degraded by 3.3% on average. In the latter experiment (varying the payload size), the IPC degraded by 0.9% on average. There appears to be no mounting overhead for running nginx with BLANKIT for an extended time, but shorter payloads (e.g. the size of a single HTML page) affect the IPC slightly more. We ran one additional performance experiment on nginx: a glibc-only protected version. We ran the same wrk configuration as before, for 30s, using the main page for Wikipedia. The slowdown of bytes transferred per second was 1.07x, which is comparable to the overhead when supporting all libraries.

To test sshd’s performance, we ran the Dijkstra shell script against vanilla and BLANKIT-enabled sshd. We measured no noticeable runtime difference. For this experiment we also

checked CPU utilization and IPC. We found a negligible difference in CPU utilization (1% worse with BLANKIT) and detected no difference in IPC. We tested with concurrent ssh connections and again found no noticeable effect. The reason appears to be that the interaction with ssh clients are completed immediately and the control passed to spawned shells. We tried with more connections (up to 1000) and still saw no difference. Thus, for openssh, the security benefit of BLANKIT comes with little to no performance penalty.

Table 2.5: Transfer/sec degradation with BLANKIT under varying, sustained load times.

Benchmark	3s	30s	300s
nginx	1.07x	1.09x	1.06x

Table 2.6: Transfer/sec degradation with BLANKIT under varying, sustained request sizes.

Benchmark	1MB	10MB	100MB
nginx	1.02x	1.04x	1.00x

2.6.3 Prediction

For nginx, we tested against 2 kinds of requests: a normal-sized html page (the main page for Wikipedia) and a randomized binary file (1MB). In both cases we ran for 30s. The results are shown as part of Table 2.7. BLANKIT’s prediction for nginx is 100% in both cases. In other words, the call chains within dynamic libraries for nginx are easily predictable under normal execution for GET requests, and this is predictable even with minimal training.

To test prediction on sshd, we designed several scripts. The first manipulates a few folders, changes directories, and writes “hello world” strings to files and reads them back. In Table 2.7, this is the “cmds” test. The “cmds-100” test is the same but with 100 concurrent connections. The Dijkstra test is the same script used in performance and security testing. We performed a 100 concurrent-connections test for it, as well. Lastly, there is a “6MB” test that simply reads from a 6MB text file (retrieved from Peter Norvig’s website [78]) and sends it back to the client through the terminal. Our results show that with

minimal training, BLANKIT predicts with 95% accuracy and above for these typical ssh interactions.

There is no measurable audit overhead for nginx because the accuracy was perfect in our experiments. This is with non-trivial dynamic library call counts (11,521,794 and 2,547,380 library calls in 30s for the Wikipedia and 1mb.bin tests, respectively). In sshd, the call counts are lower (ranging from roughly 2,000 to 50,000, depending on the input), and the highest mispredict / second ratio is under 300. Table 2.7 shows audited functions take up to $\sim 100\mu s$ to execute. Thus with 300 mispredicts / second, an alarm handling mechanism only needs about 30ms of that 1 second to process alarms, which is more than enough to keep pace in parallel.

Table 2.7: Call chain prediction accuracy and audit overhead for nginx and sshd

Benchmark	Test	% Accuracy	Audit (μs)
nginx	wiki	100	36
nginx	1MB	100	40
sshd	cmds	97	53
sshd	cmds-100	95	47
sshd	dijkstra	97	56
sshd	dijkstra-100	98	73
sshd	6MB	97	55

2.6.4 Defense Against Real-world Attacks

We describe two known attacks on these applications and show how BLANKIT defends against them: the attack demonstrated in “Control Jujutsu” [28] against nginx, and CVE-2015-7547 for sshd [79].

The Control Jujutsu [28] attack on nginx proceeds as follows: A memory vulnerability is exploited, allowing an attacker to modify a function pointer and its arguments; when the function pointer is eventually invoked, execution jumps to the attacker’s target function with the corrupted arguments. In the paper, an nginx global object called `ctx` is corrupted, modifying the `ctx`’s `output_filter()` function to target `ngx_execute_proc()`,

which executes `execve()`. The arguments, which are passed via `ctx's filter_ctx` member, are also corrupted. This allows for the attackers to carry out remote code execution. The authors show this is all achievable using just HTTP GET requests. As we observed, BLANKIT's prediction on nginx is very strong (100% in our GET experiments using simple training), which suggests any kind of function pointer manipulation will be caught by BLANKIT.

The `sshd` vulnerability allows “remote attackers to cause a denial of service (crash) or possibly execute arbitrary code via a crafted ... call to the `getaddrinfo` function [in `glibc`].” As we showed in our evaluation, BLANKIT can be enabled for `sshd` to protect `libnss` modules and `libresolv`, which were involved in the attack. In short, the attack can be triggered by `ssh`, and it is based off a mistake in `glibc` that uses an undersized stack buffer when it should be using its heap; this leads to an overflow and potentially arbitrary code execution. With BLANKIT a jump to an arbitrary location for using library gadgets would enter a blanked region, and the unpredicted function execution would raise an alarm.

2.7 Related Work

The two main lines of research that are closely related to our work in this chapter are security-based software debloating and security techniques that mitigate memory corruption.

In the Introduction, we contrasted BLANKIT with sound debloating techniques like static debloating and load-time mechanisms like piece-wise [52], and with unsound techniques like Razor [43] and Chisel [59]. BLANKIT differs in goals and achievements, and direct comparisons are challenging. To summarize, however, BLANKIT's ROP gadget reduction is 97% or greater on SPEC with `glibc`, with 18% runtime overhead (all libs). To our knowledge, no other security debloating mechanisms handle `glibc`. Piece-wise uses `musl-libc` (2 CVEs to `glibc`'s 108) and achieves up to 86% ROP gadget reduction on SPEC. Chisel and Razor are debloating only applications; they achieve 85.1% and 68.19% code

reduction on SPEC; they are unsound.

Within the software engineering community several works [80, 81, 82, 83, 84, 85, 86] have targeted debloating to improve performance. The embedded software community performed debloating based on link-time optimizations to reduce code size. The key goal of compiler-based link-time optimizations is to compact the code size [87, 88] [89, 90]. While these works have paved the way for debloating, they are not geared towards solving the security issues due to bloated libraries.

We also discussed the different CFI and memory safety mechanisms that avoid change in control-flow due to memory corruption and prevent memory corruption through memory safety, respectively. While BLANKIT is not a stand-alone system and depends on CFI for thwarting attacks, BLANKIT strengthens CFI by reducing the code surface and disallowing the use of gadgets in the library code for attacks that bypass CFI. BLANKIT also catches attacks such as non-control data attacks that involve change in call flow by running an audit mechanism with memory safety on mispredictions. An online full memory safety mechanism can incur overheads greater than 100%. However, full memory safety mechanisms like AddressSan [91] or EffectiveSan [92] can be employed offline to detect bugs in the code and are used for a different purpose.

2.8 Conclusion

In this chapter, we propose and demonstrate a “get what you want” security defense scheme that effectively reduces the exposed code surface of vulnerable linked libraries and defends against attacks. This is a departure from current debloat techniques that “cut what you don’t want.” As a security technique, BLANKIT advances the state of the art in binary library debloating and in defenses against non-control data attacks. As a debloating technique within the security space, BLANKIT achieves aggressive, sound function and gadget removal. At the crux of our approach is a decision-tree based call path predictor. The predicted call graph is used by a binary-level mechanism that dynamically loads the needed

functions at a given call site while the unneeded ones are blanked, which significantly weakens an attacker's ability to construct gadgets. Our results on the SPEC2006 benchmark suite (evaluated with glibc) show prediction accuracy in most cases to be at least 97%, and dynamically linked function and ROP gadget reduction likewise to be at least 97%. On average, the runtime overhead with BLANKIT is 18% across all libraries. We also evaluate two critical server applications, nginx and sshd, with over 10 more libraries, and show similar security and prediction results, with overhead being either within 10% or negligible, respectively. Due to near perfect reduction in the gadgets and CVEs with reasonable overheads, we conclude that BLANKIT provides a practical debloating mechanism.

CHAPTER 3

DECKER: SOURCE-LEVEL APPLICATION DEBLOATING

3.1 Introduction

Improving the state-of-the-art for defenses on whole applications is a critical issue. BLANKIT shows one approach for tackling libraries, but it does not address the applications themselves. A question that still begs attention for whole applications is whether the attack surface can be reduced substantially enough to break chains while maintaining the soundness property during normal (non-attack) execution.

Thus, we propose DECKER, an attack surface reduction technique for reducing reusable gadgets and breaking their chains. This chapter makes the following contributions:

1. The first sound, whole-application technique for on-demand loading and purging for attack surface reduction which also has low runtime overhead.
2. An evaluation on SPEC CPU 2017, GNU coreutils, and `nginx` that is comparable to unsound techniques in terms of gadget reduction.
3. Evidence that short but highly detrimental gadget chains can be broken by this technique.

The remainder of this chapter is laid out as follows: an overview and assumptions of our solution (section 3.2), framework details (section 3.3), evaluation (section 3.4), related work (section 3.5), and conclusion (section 3.6).

3.2 Overview

3.2.1 Proposed Solution

We propose a compiler and runtime solution for attack surface reduction called DECKER. It is sound, has a static and runtime component, requires no user input, requires no hardware changes, and works on application code.

DECKER embraces the idea that only code that is currently needed by a running program should be available for execution; the rest should be made inaccessible such that any attempt to access it should trigger a runtime exception. Active sections of code form “decks” that the program can effectively stand on. When a deck is unneeded, it can be removed. To take the analogy further, DECKER is a technology for attack surface reduction, but it can be viewed as *constructive*. It achieves attack surface reduction not by cutting down the program, but by putting together the active code that it needs at any particular execution point. A deck could be a group of functions, a subset of which are guaranteed to execute from a given program point (taking into account the program intra- and inter-procedural control flow). Since such a set cannot be precisely evaluated at runtime without causing heavy overheads (especially inside loops), DECKER will turn this into a tight over-estimation problem inside respective regions.

DECKER depends heavily on static analysis. The key idea of DECKER is simple: to determine a group of functions that may execute at a given program point (called decks), enable their executable permission just in time, and disable their executable permission at a subsequent program point. Ideally, decking could be performed right before a call site, but the overheads of doing so could be very high, especially if the call site is located inside a loop. In order to reduce overheads, decking is placed at key program points such as at the entrances of outermost loops or at the entrances of long static call chains. At each decking point, a conservative approximation of all functions reachable until the next decking point is applied (including multiple targets of function pointers, if any) to maintain the soundness

condition. In our implementation, granularity of the disabling/enabling mechanism is at the system page level. Creating and tearing down a deck corresponds to marking code pages read-execute (RX) and read-only (RO), respectively. In other words, if a deck consisting of functions `foo()` and `boo()` is to be enabled at some program point, one must execute calls to mark the respective pages that contain `foo()` and `boo()`'s code as RX.

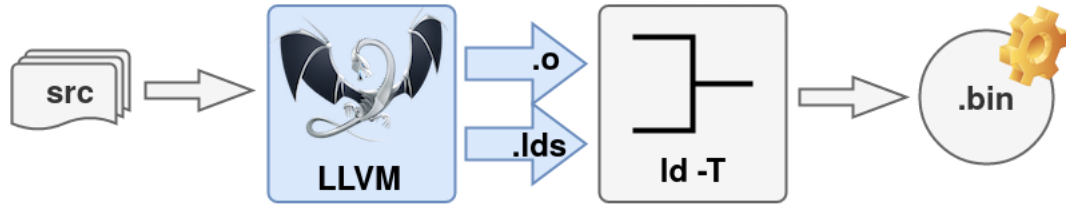


Figure 3.1: High-level view of the compiler part. The LLVM pass outputs a custom linker script and instrumented object file.

Figure 3.1 shows a high-level view of the compiler step of the DECKER solution. Application source code is fed into the LLVM compiler pass. The compiler performs static analysis and first identifies programs points for decks to be enabled (RX) and disabled (RO); it also performs partitioning of such sets of functions to improve security benefits, as discussed later. Additionally, the pass creates a custom linker script. Both the object files and linker script are fed into the linker (where the `-T` option consumes the custom linker script). The linker produces the final binary. DECKER is represented by the blue part in the figure, and the linker itself is unmodified.

Figure 3.2 illustrates the runtime by way of an example. It is drawn directly from the GNU coreutils' `date` program [93], which provides a command-line option for reading dates from a file (given by the `-f` switch). When this option is given, `main` invokes `batch_convert`. At runtime, DECKER will create and tear down a deck for this single function. In Figure 3.2, there are two code pages in memory (for simplicity). Page A contains `main`, and page B contains `batch_convert`. The call to `batch_convert` has been instrumented by the compiler, so that before it is invoked, its page will be mapped RX, and after it returns, its page will be mapped RO. These mapping steps are done by

`deck_single` and `deck_single_end`, respectively. `bc_funcid` is the function ID for `batch_convert` assigned by DECKER at compile-time. The 4 program points, P1-P4, indicate which pages are mapped RX at each step. One can see that gadgets in unmapped decks and their pages are thus inaccessible to the attacker; thus, the finer the granularity of the deck (finest granularity being one function), the better the security.

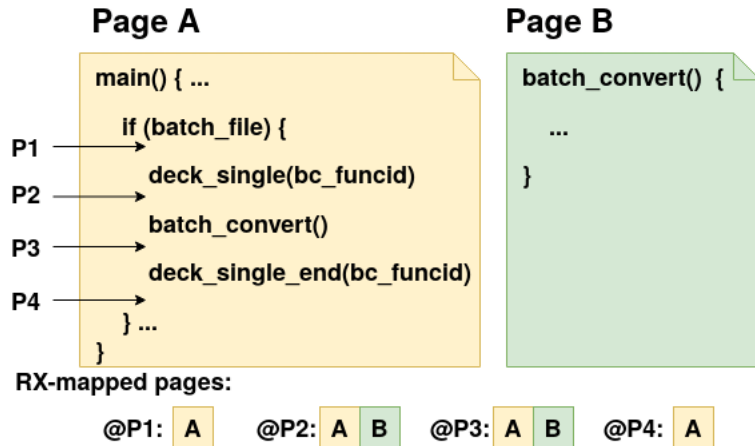


Figure 3.2: High-level runtime example from GNU coreutils’ `date`. The set of RX-mapped pages increases at P2 to include a called function.

3.2.2 Soundness

Guaranteeing soundness is a key contribution of DECKER. We define a sound transformation as one that does not change the semantics of a program. A DECKER program transformation is simply one that enables code pages before use (and disables them afterward). We claim that DECKER is sound because it always enables page(s) containing any forthcoming function calls (either single functions or a group of them) before execution can enter that region. To demonstrate this, consider a simplified implementation of DECKER which instruments before and after *every* interprocedural control flow transfer. Provided all transfers are accounted for, such a program transformation is sound by design – but it would be prohibitively slow. DECKER’s soundness follows from this base case. Instead of instrumenting at every transfer, however, DECKER’s analysis removes superfluous instru-

mentation based on the static reachability of functions and loops in the (conservative) static callgraph using the concept of encompassed functions. This will be discussed in detail in subsection 3.3.1.

DECKER’s soundness guarantee comes with a critical (but realistically met) assumption, namely that the callgraph at the intermediate representation in the compiler is correct. One example where this may not hold for LLVM (DECKER’s underlying compiler) is inline assembly code, which can hide callgraph edges. Inline assembly is typically for performance, missing language support, or compiler barriers. We did not encounter missing edges in any of our benchmarks on Linux or Windows. DECKER can easily support warning or hard-failing when encountering inline assembly that contains control flow transfer. DECKER can also uncover some missing edges by locating those functions which are not reachable from `main()`.

3.2.3 Threat Model

We assume the operating system and compiler are trusted. The source code and any third-party libraries may contain bugs. For simplicity, we do not handle dynamically generated code or self-modifying programs; we focus on C/C++ source.

We are focused solely on attack surface reduction and assume the attacker has some way of initiating and propagating the attack (e.g. that the attacker can exploit a memory vulnerability and trigger a gadget chain). Given today’s state-of-the-art defenses, we find this assumption reasonable.

We assume the runtime is protected (a similar assumption in [94]), and which can be implemented with in-process isolation [95], hardware segmentation or software fault isolation [96]. This prevents attackers from jumping into the runtime and guarantees, along with the trusted loader, that the statically computed function IDs and framework metadata are protected.

Arguments to the runtime API are statically evaluated and passed by register and so

cannot be tampered with, except in the case of indirect calls. How to guarantee the integrity of indirect call targets is precisely the problem handled by orthogonal schemes like CFI and CPI (see section 3.5). DECKER does *not* tackle this problem, which we consider to be orthogonal to the core problem of this chapter to disable gadget chains through decking. In short, DECKER is focused on reducing the attack surface available to an attacker when an attack occurs. DECKER's main goal is to incrementally expose the executable surface of a program by following its interprocedural control flow. Such an approach breaks chains of gadgets, because all the gadgets that compose a chain are never dynamically exposed at the same time. Repeatedly invoking the same instrumented runtime call that is mapped RX is disallowed by construction (see section 3.3, which details how instrumented calls will only execute exactly once for every paired teardown call).

As described earlier, the threat is an attacker exploiting the memory vulnerabilities of an application executing under the DECKER system, attempting to string together a gadget chain to launch an attack. Due to needed gadgets residing in multiple decks that are disabled, however, the attack will lead to a runtime exception and will be caught.

3.3 Framework

3.3.1 Compiler Component

DECKER's compiler component is an LLVM [97] pass that can be divided further into two parts: instrumentation and linker script output. During instrumentation, DECKER identifies function calls and loops and instruments them appropriately with calls to the runtime. As the pass does this, it collects critical static information for organizing the text section, which it uses to create a custom linker script. A key idea of the work in this chapter is to keep the decks as lean as possible. We define a deck as a group of functions that are enabled at a program point by turning their page permissions to RX. For the best security, each deck could consist of one function. Such a scheme would incur very high overheads, however, especially for call chains that execute inside loops, and would make the scheme untenable.

Thus, based on the context surrounding a program point, static analysis identifies which functions should be part of a deck at a given program point and inserts calls to the runtime accordingly.

Analyzing for Decks

Analysis and instrumentation for decks is heavily organized around loops. Loops are problematic because adding code for enabling or disabling a deck inside them can cause significant performance degradation. We define two terms: **encompassed** and **non-encompassed** functions to distinguish between two kinds of a function's static loop context. A function is encompassed if it is called inside of a loop, or if it is reachable through the callgraph by some function that is called within a loop through a caller-callee relation. To determine the encompassed function set, the pass first identifies all functions called within a loop, and then takes the transitive closure of any functions reachable from that set using the caller-callee relation shown by the callgraph. The non-encompassed function set is simply the set of all functions minus the encompassed function set.

DECKER's default treatment of loops is to bear on the side of performance. Therefore it tries to avoid adding decks inside of loops, because if this deck instrumentation is invoked in a surrounding intra- or interprocedural loop at runtime, the instrumentation will incur repeated invocations, leading to high overheads. Interprocedurally this implies that DECKER cannot instrument inside of encompassed functions, either. This raises a problem for loop-enclosed indirect calls, whose static target set can be large, and whose precise dynamic value is often unknown until execution is inside of the loop. Note that a given function might be both encompassed as well as non-encompassed, depending upon its loop calling context. Such cases are tackled by edge-based placement of RX-RO.

To handle these different cases, the pass instruments four different types of decks which will invoke the runtime: (1) Single, (2) Loop, (3) Reachable, and (4) Indirect. The **single** deck is used when a non-encompassed function calls a non-encompassed function. Because

the callee is known to be non-encompassed (i.e. not part of some transitive closure that lies within a loop), only a single function needs to be mapped RX (i.e. the callee itself). The **loop** deck is placed at the outermost loop header for any loop nest in any non-encompassed function. It is designed to map all functions that can be reached interprocedurally within that loop. The **reachable** deck is placed in a non-encompassed function before any calls to encompassed functions.

The **indirect** deck is for function pointers. The challenge of function pointers is that their exact targets are often not known statically, and therefore the compiler cannot determine precisely what needs to be mapped RX until runtime. Function pointer analysis can help narrow the possible targets but would still be an overapproximated set at compile time (which would limit attack surface reduction). DECKER originally attempted to invoke static function pointer analysis but found the results very unacceptable (especially in a C/C++ setting), so it instead opts to solve this at runtime. The instrumentation passes the function pointer to the runtime library, which then maps the appropriate page(s). Indirect decks may need to be placed inside of loops if static analysis fails to hoist the value outside of it. Such cases must be optimized to avoid drastically degrading performance (discussed at the end of subsection 3.3.2).

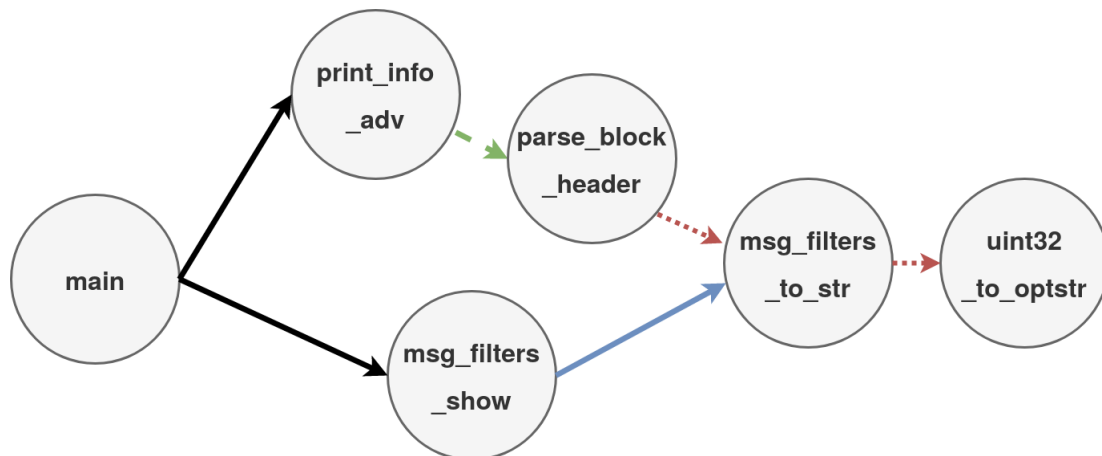


Figure 3.3: Simplified callgraph from the xz data compression application. This illustrates 4 types of edges, each of which requires different handling by the instrumentation pass.

Figure 3.3 depicts a sub-callgraph from SPEC CPU 2017’s xz, a data compression ap-

plication [98]. It illustrates all but the **indirect** case. Each node in the figure is a function, and each edge is a call. Only the call from `print_info_adv` to `parse_block_header` (dashed, green) is inside of a loop. The set of encompassed functions is therefore $\{\text{parse_block_header}, \text{msg_filters_to_str}, \text{uint32_to_optstr}\}$, and the set of non-encompassed functions is $\{\text{main}, \text{msg_filters_show}, \text{print_info_adv}\}$. Instrumentation will be treated as follows:

1. The solid-black edges from `main` require **single** decks.
2. The dashed-green edge from `print_info_adv` requires a **loop** deck; instrumentation will be at the loop preheader that dominates the call to `parse_block_header`.
3. The solid-blue edge from `msg_filters_show` requires a **reachable** deck.
4. The dotted-red edges from `parse_block_header` and `msg_filters_to_str` will have no instrumentation, because they are encompassed.

Basic pseudocode for the compiler instrumentation pass is shown in Algorithm 3. It shows the function `run_on_func`, which is a hook called by the pass manager on each function. The pseudocode shows the general logic for how decks are selected and inserted. Each deck needs only one key piece of runtime information, namely a unique ID that is generated statically for each loop or function. At runtime the library maps this parameter to a set of functions and their corresponding pages in memory. (In the case of indirect calls, the only difference is that the runtime target address is used instead of a statically known ID.) Several details not shown in the pseudocode but that should be noted include: the insertion of the deck teardown calls; the insertion of an initialization call at program start; construction of functions' static reachability; and construction of the encompassed function set.

DECKER's compiler instrumentation supports non-trivial C and C++ behavior. It handles recursion and strongly connected components (SCCs) similar to loops by adding such

Algorithm 3 Pseudocode for DECKER’s compiler pass.

```
function RUN_ON_FUNC(func)
  for instr in func do
    switch instr.getType() do
      case LOOP_START
        INSERT_DECK(LOOP, instr.id)
      case DIRECT_CALL_ENCOMPASSED
        target ← instr.GET_CALL_TARGET()
        INSERT_DECK(REACHABLE, target.id)
      case DIRECT_CALL_SINGLE
        target ← instr.GET_CALL_TARGET()
        INSERT_DECK(SINGLE, target.id)
      case INDIRECT_CALL
        target_addr ← instr.GET_FUNC_PTR()
        INSERT_DECK(INDIRECT, target_addr)
```

functions to the set of encompassed functions. Thus, DECKER does not instrument inside of SCCs (which avoids overheads due to repeated execution). Other general features that DECKER must support include the following: In addition to handling LLVM IR’s call instructions, it must also handle invoke instructions and therefore landing pads. It handles external libraries that take and then invoke a callback to the application. It handles C++ destructors that can be invoked when an exception is thrown (via `_cxa_throw`). It handles `libc_nonshared.a`. It handles start-up C++ code before `main`. It handles signal handlers, including `atexit` and `on_exit`. Though the details for these cases are unimportant, it is important to stress that the approach is general.

Function Pointers

We follow the callgraph construction of LLVM, which safely handles function pointers. As stated in the LLVM source code, the callgraph is implemented as a “conservative superset of all of the caller-callee relationships, which is useful for transformations;” an extension/improvement in the future could be to “prove (through pointer analysis) that an indirect call site can call only a specific set of functions” [99]. As currently implemented, however, LLVM creates two “external” nodes in a module’s callgraph. The first node rep-

resents all external calls that can enter the module: If a function’s linkage is external/visible outside of the module, it must have an edge from this node; and, *if a function has its address taken, it must have an edge from this node*. The second node represents all external calls that can leave the module: If a function is external to the module, it must have an edge to this node; and, *if a function invokes an indirect call, it must have an edge to this node* [99]. In other words, all functions that are either indirectly callable or that call function pointers are connected by an edge to one of these nodes, which is sufficient (albeit conservative) for DECKER to statically identify all functions that must be mapped when any particular deck (i.e. portion of the callgraph) is added.

As we will show in the evaluation (see subsection 3.4.1), unfortunately the static function pointer analysis in LLVM is incredibly insufficient for reducing gadgets, due to its low precision. DECKER does not attempt to ameliorate it with flow, field, or context sensitivity techniques. Rather, DECKER uses the dynamic target of a function pointer to enable the target function’s page(s). Note that the dynamic pointer target in no-attack scenarios must be valid; thus, the applications do not crash, and the technique is still sound. Also, as mentioned earlier, we do not address the orthogonal issue of function pointer integrity but rely on other approaches such as CFI [16, 100]. Other function pointer analysis works (such as for Java virtual methods [101]) could form the basis for extending LLVM or DECKER in the future. In fact, the programming language plays an important part in the callgraph construction. Dynamic facilities like Java’s reflection or JavaScript’s eval would have to be carefully handled if applying techniques like DECKER beyond LLVM.

Linking

At the end of the compiler pass, DECKER outputs a custom linker script. Intuitively, the goal of the linker script is to separate functions into different pages so that marking a given function as RX does not “activate” unrelated functions that reside in the same page (i.e. make unrelated functions and their gadgets available for use). Thus, the goal of this step is

to factor out such functions into separate pages. An example and pseudocode may help to understand this more clearly.

Example: Referring again to the callgraph in Figure 3.3, the deck sets for this callgraph $Dk.*$ are:

$$Dk.S1 = \{print_info_adv\}$$

$$Dk.S2 = \{msg_filters_show\}$$

$$Dk.L = \{parse_block_header, msg_filters_to_str, \\ uint32_to_optstr\}$$

$$Dk.R = \{msg_filters_to_str, uint32_to_optstr\}$$

Any of these functions can arbitrarily belong to the same system page at runtime. For example, without any enforcement, `parse_block_header` can occupy the same system page as either of the functions in $Dk.R$. That is, invoking `msg_filters_to_str` from `msg_filters_show` at runtime could inadvertently activate the gadgets in `parse_block_header`. The solution is to rely on the fact that DECKER instrumentation ensures that each deck will be mapped RX independently at runtime, and to leverage the custom linker script to avoid this security penalty. The custom linker script should separate the intersection $Dk.L \cap Dk.R = \{msg_filters_to_str, uint32_to_optstr\}$ into its own disjoint set. Thus, the full disjoint sets $Dj.*$ are as follows:

$$Dj.S1 = \{print_info_adv\}$$

$$Dj.S2 = \{msg_filters_show\}$$

$$Dj.L.1 = \{parse_block_header\}$$

$$Dj.I.LR = \{msg_filters_to_str, uint32_to_optstr\}$$

—where $Dj.I.LR$ is the disjoint set formed by $Dk.L \cap Dk.R$ and $Dj.L.1$ is the disjoint set formed by $Dk.L \setminus (Dk.L \cap Dk.R)$. Each of these disjoint sets will be page-aligned by the custom linker script. Such a separation allows finer control over the activation of decks,

although it increases the code size.

Pseudocode: The pseudocode for creating these disjoint sets is shown in Algorithm 4. The algorithm begins with the deck sets ($Dk.*$ in our example). A deck set corresponds directly to 1 of the 4 types of decks: the function of a **single** deck forms a singleton; the functions of a **loop** deck form their own set; any encompassed function that can be called from some non-loop path has itself and any **reachable** functions as part of a set; and any functions that have their addresses taken and can be invoked by some **indirect** call form a set with their statically reachable callees. The algorithm begins with a list of these sets and then iterates, attempting to separate functions into their own disjoint sets ($Dj.*$ in our example). To find the disjoint sets, each pair of the decks is intersected with each other. If the intersection is non-null, those shared members are removed from the pair of decks and form their own disjoint set. This pairwise intersection-removal process is repeated until no more disjoint sets can be formed. Once the disjoint sets are known, each one is assigned its own page-aligned section in the linker script.

Algorithm 4 Pseudocode for creating disjoint sets for DECKER’s custom linker script.

```
function CREATE_DISJOINT_SETS(deck_sets)
  disjoint_sets  $\leftarrow$   $\emptyset$ 
  while !deck_sets.EMPTY() do
     $A \leftarrow$  deck_sets[0]
     $tmp \leftarrow$  deck_sets[1 :]
    deck_sets  $\leftarrow$   $\emptyset$ 
    for  $B$  in tmp do
       $I \leftarrow A \cap B$ ;  $A_I \leftarrow A \setminus I$ ;  $B_I \leftarrow B \setminus I$ 
      if  $|I| == 0$  then
        deck_sets.PUSH( $B$ )
      else
        if  $|B_I| > 0$  then
          deck_sets.PUSH( $B_I$ )
        deck_sets.PUSH( $I$ )
         $A \leftarrow A_I$ 
      disjoint_sets.PUSH( $A$ )
  return disjoint_sets
```

Precision and Recall

In the context of DECKER, we define false positives (precision) as alarms for attacks which never occurred. This case is not possible under DECKER; DECKER faithfully follows program semantics under caller-callee relations, so any control flow outside a deck under a no-attack scenario is impossible. We define false negatives (recall) as attacks that occur but were not caught. False negatives are a possibility under DECKER (see subsection 3.4.1 for experimental results). One way to quantify it is as the additional code surface that is mapped RX by DECKER but which is never executed. This can happen because the mapped code is a (tight) overapproximation of the forthcoming code at any particular program point. The false negative rate is also related to the soundness properties and function pointer analysis of DECKER. In terms of soundness, an underapproximated mapping of the forthcoming code would reduce false negatives, but it would be unsound. In terms of function pointer analysis, replacing DECKER's dynamic technique with pure static analysis would lead to much larger code mappings and therefore a higher false negative rate.

3.3.2 Runtime Component

The runtime support is exposed as a library to the application. It is responsible for enabling and disabling pages by marking them RX or RO. The API calls align directly with the 4 types of decks mentioned previously (single, loop, reachable, and indirect), plus library initialization. They also have a corresponding deck teardown call to remap the relevant pages RO.

Two important steps are necessary for library initialization. The first is identifying the binary's base address. Function offsets are known at build-time, but at runtime DECKER still needs to determine the text section's base address. The second step is to protect all of the text pages by marking them RO. This happens at the start of main, and main is left RX.

Regarding the primary API endpoints, `deck_single` takes as argument the function ID of an impending callee. The runtime uses the ID to look up the actual page addresses of

this function, and it marks them as RX. When that function returns, `deck_single_end` will mark the pages as RO. `deck_reachable` is similar. Because the callee is an encompassed function, though, all pages of statically reachable functions must be marked RX, as well. These are compile-time known, so the runtime library only needs to issue a map lookup to find which set of functions to mark RX for that particular callee. `deck_loop` takes a single loop ID parameter as its argument. A unique loop ID is assigned by the compiler to each interprocedural, outermost loop in the program. It is a simple lookup to find which functions are part of that loop, and to map them RX. `deck_indirect` takes a runtime address as its argument. This is mapped by the library to the corresponding function, in order to determine whether that function is an encompassed or non-encompassed function. If it is encompassed, then the library leverages its own `deck_reachable` support for that function. Similarly, non-encompassed functions are handled by the `deck_single` support.

DECKER maintains a reference counter for the text pages. When a function is needed, the reference count for each of its pages is incremented; when that function is no longer needed, the reference count for each of its pages is decremented. Whenever a page's count changes from 0 to 1, the page must be marked RX; and whenever a page's count changes from 1 to 0, it can be marked RO again. We define the set of pages at runtime with reference counts greater than 0 as the **available pages**. Adding a deck at runtime will either increase the cardinality of the available pages (if new pages are needed), or have no effect on its size (if all needed pages are already available); the opposite holds for removing a deck.

There is one critically important optimization we make that is called indirect deck caching (IDC). When there is an indirect call inside of a loop, IDC inlines a check against its address, and that deck is “cached” for the duration of the loop. We find that this reduces slowdown by up to 80% in some cases. It is also an argument against static treatment of function pointers, which is fast but enables too many gadgets at loop headers (~40% worse).

3.4 Evaluation

We perform experiments on two commodity desktops. Almost all experiments are performed on Linux with an AMD Ryzen 7 1800X with 32GB RAM; Ubuntu 18.04 LTS; and LLVM v11.0.0. The experiments on Windows (subsubsection 3.4.6) are performed with an AMD Ryzen 7 2700X with 32GB RAM; Windows 10 Education v21H2; and LLVM v11.0.0 (built with cl.exe v19.31.31104). We perform experiments on the SPEC CPU 2017 suite [98], 10 programs from the GNU coreutils package [93], and the nginx web server v1.20.1 [14]. We choose these programs partly for their inherent qualities, but also because other debloating techniques experiment with them, which allows for comparison. Unless stated otherwise, DECKER’s results are with the IDC optimization enabled (details in subsection 3.3.2); baseline results are with the same compiler and optimization levels (but without the framework).

Our evaluation focuses on the following questions:

1. How much slowdown does an application incur when using the DECKER framework?
What is the code growth due to the linker step for segregating function sets?
2. What is the gadget reduction for applications using DECKER?
3. Can DECKER (a) break real gadget chains in the benchmarks and real-world applications to be able to stop gadget-based attacks; and (b) render JOP gadgets ineffective in practical scenarios, including Windows?

3.4.1 SPEC CPU 2017

SPEC CPU 2017 [98] is a staple suite for CPU-bound performance benchmarking, making it useful for stressing the performance of binaries running under DECKER. It also includes a diverse group of applications that give us insight into DECKER’s effect on security, too. We use C and C++ applications from the suite, used in domains including route planning,

discrete event simulation, video compression, alpha-beta tree search, molecular dynamics, and ray tracing.

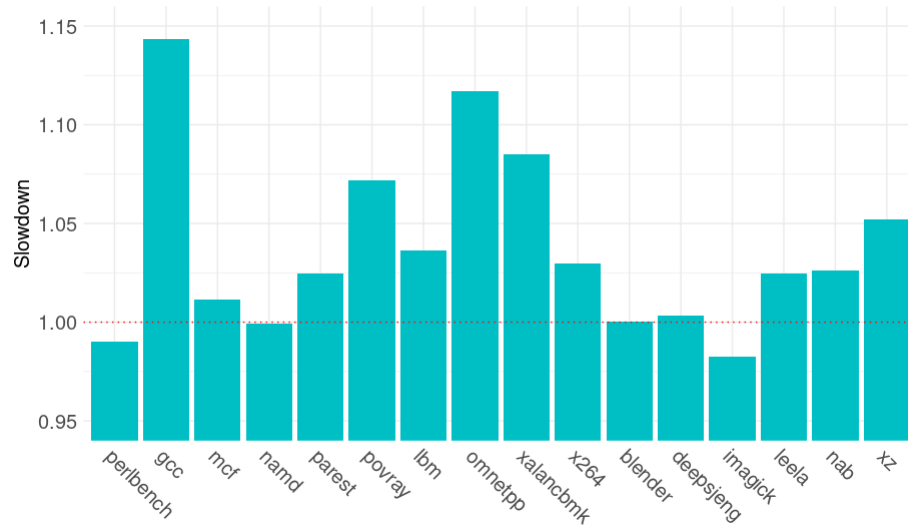


Figure 3.4: Slowdown for SPEC CPU 2017 using DECKER.

Performance

The normalized performance results are reported in Figure 3.4. We compile and run a baseline version of each benchmark, optimized at `-O3`. Then we recompile and run the benchmark with DECKER. The worst-case slowdown is 14% for `gcc`. Both `imagick` and `perlbench` have slight speedups, which can happen in instrumentation-based works that affect memory alignment [95, 26, 24, 94]; DECKER also modifies function layout across pages which may play a role. The average slowdown across SPEC is 5.2%.

In comparison, BlankIt achieves 18% overhead on SPEC CPU 2006 by debloating libraries (not the application). Razor achieves 1.7% overhead on average on SPEC CPU 2006, with a worst-case of 16%. Piece-wise adds only negligible load-time overheads but deals only with libraries and not whole applications. Thus, we find DECKER’s 5.2% average slowdown (on whole applications) in SPEC CPU 2017 to be reasonable compared with existing approaches.

Table 3.1: SPEC CPU 2017 total gadget reduction as a percentage (higher is better).

Application	Min	Max	Avg
perlbench	51.1	98.8	68.4
gcc	44.4	99.6	73.8
mcf	25.6	68.4	54.0
namd	75.4	94.0	88.5
parest	76.1	99.8	94.6
povray	36.6	97.4	53.0
lbm	47.9	62.9	57.4
omnetpp	52.4	98.4	79.1
xalancbmk	58.9	99.6	72.8
x264	17.2	99.9	32.5
blender	73.9	99.8	98.5
deepsjeng	24.4	68.2	64.9
imagick	39.3	99.4	88.7
leela	54.6	87.7	84.2
nab	68.6	91.9	86.6
xz	51.7	94.9	74.1
AVERAGE	49.9	91.3	73.2

Security

With DECKER, the gadgets that are available to an attacker change dynamically during runtime based on which pages DECKER has mapped RX, i.e. based on the **available pages** (discussed in subsection 3.3.2). The average reduction over all available page sets is given by the formula:

$$avg_reduction = \frac{\sum reduction_{AP}}{num_APs}$$

—where $reduction_{AP}$ is the total gadget reduction for some available page set AP versus the baseline. We simply sum over all such reductions and divide by the number of available page sets num_APs .

To capture these reduction metrics, we first enable DECKER’s logging and run the program under all test inputs. This dumps the available pages on every DECKER API call to a log file. Next, we scan every log line and identify the gadgets across each available page set. Each set of available pages is considered “equal” to another for the purposes of gadget-

counting. The average reduction is the average proportion of gadgets reduced by each set of available pages over the logs.

The per-benchmark gadget reductions are shown in Table 3.1. DECKER achieves an average of 73.2% total gadget reduction across all SPEC CPU 2017 benchmarks. For SPEC (and other experiments, as well) the custom linker script contributes $\sim 15\%$ to the total reduction. The total gadget reductions are representative of the individual (ROP, JOP, COP, and special-purpose) results, which is expected. For example, DECKER’s average reduction of ROP gadgets specifically is 77.3%.

We capture these same gadget reduction results for a DECKER scheme without dynamic pointer target resolution. In other words, when DECKER uses LLVM’s built-in function pointer analysis, we want to see the impact on gadget reduction. The average total gadget reduction across all benchmarks is 39.9% (27.6% and 41.5% for minimum and maximum, respectively). This is nearly half as effective as a dynamic technique, which helps clarify that dynamic function pointer resolution is crucial to gadget reduction.

Direct comparisons to prior work are difficult because of differences in technique or reporting. Piece-wise reduces total gadgets by an average of 72.88% on SPEC CPU 2006 benchmarks for musl-libc but does not perform whole application debloating. BlankIt reports an average of 97.8% ROP gadget reduction on SPEC CPU 2006 benchmarks for all libraries (and using glibc) and does not do whole application debloating either. Razor reports 68.19% code reduction (not gadgets) for applications in SPEC CPU 2006. Thus, DECKER’s SPEC security result appears to be similar to the other application-focused technique, Razor, without sacrificing soundness. Compared with the library-only techniques, DECKER appears to reduce applications’ gadgets equally as well as the load-time technique (Piece-wise), but not as thoroughly as an on-demand runtime technique (BlankIt).

Compilation Time

We capture the slowdown in different stages of DECKER’s compilation of SPEC. In terms of the absolute time of the LLVM pass itself, in 11 out of 16 benchmarks, DECKER completes in a negligible amount of time (≤ 1 s); in 2 cases, it completes in ≤ 10 s; and in the remaining 3 cases, it takes 17s, 36s, and 186s. The 3-minute outlier is gcc, which is much more complex than the others. Comparing the overall time to build the binaries, we see 9.3% slowdown with DECKER compared with the baseline.

False Negative Rate

Lastly, we report a false negative metric when running SPEC CPU 2017 under DECKER. We perform a tracing run so that whenever DECKER adds a deck, we can track what percentage of the newly added surface is executed. The average percentage of pages used after a deck/growth step is 88.0% (86.3% for functions). Thus, as an estimate of false negatives, DECKER supplies 12% extra (unused) system pages that could be potentially used by an attacker but which would not be caught. Note that this metric is imperfect, since it merely measures extra code surface enabled; it is unknown if this code surface contains gadget chain components that could be exploited by an attacker. Further evaluation of gadget chain removal sheds light on this important aspect of DECKER’s attack prevention capability. Please refer to subsection 3.4.5 and subsection 3.4.6 for details.

3.4.2 GNU coreutils

We measure our technique on a subset of GNU coreutils. This package contains roughly 100 tools, including `grep`, `mkdir`, and `rm`. These utilities are relevant to software debloating for several reasons, including their real-world ubiquity, and that they have a history of CVEs. Chisel and Razor also report results for coreutils that DECKER can compare against.

The authors of Razor made their tool available [102], so we use the same application

versions and inputs as them. Their inputs were designed to cover the same functionality tested by Chisel. We use only the test inputs, as we do not require any training. The number of inputs per benchmark ranges between 17-40, and the number of options that any given input may exercise ranges from 1-7 (see [43] for more details).

Runtime overheads are negligible for coreutils. Every test completes in under 1 second and is trivially performant. (In contrast, SPEC CPU 2017 tests each take 3-10 minutes.)

Table 3.2: GNU coreutils total gadget reduction as a percentage (higher is better).

Application	Min	Max	Avg
bzip2	42.7	78.8	70.8
chown	88.3	97.3	95.9
date	95.0	97.5	96.9
grep	65.0	90.9	82.8
gzip	34.7	75.7	64.6
mkdir	90.4	96.6	94.5
rm	88.4	98.7	96.9
sort	79.0	91.9	90.5
tar	49.0	86.8	83.4
uniq	93.0	96.0	95.4
AVERAGE	72.5	91.0	87.2

As with SPEC, we present the total gadget reduction numbers (Table 3.2). The average decrease across coreutils is 87.2%. The worst-case scenario occurs at one point during `gzip`, where only 34.7% of the application’s gadgets are unavailable. The best case occurs for `rm` at 98.7%. Razor and Chisel achieve 61.9% and 85.1% ROP gadget reduction on coreutils, respectively. Thus, DECKER fares better than two other state-of-the-art techniques that reduce ROP gadgets in application code, and more importantly, does not compromise soundness.

3.4.3 nginx

`nginx` is by some metrics the most popular web server today [103]. It is used to serve web content, as a reverse proxy, and as a load balancer. As a common multitool in today’s web infrastructure, security is a real concern for `nginx`. It is multithreaded and multi-

processed, unlike the SPEC benchmarks and coreutils applications we have evaluated, and this can break or stress frameworks. `nginx`'s performance is a critical factor in certain deployments, so it is important that any security techniques not interfere too heavily with it. It is also recently evaluated by another debloating technique, BlankIt, which will serve as a good comparison point. For all these reasons, we choose to evaluate `nginx` with DECKER.

We faithfully reproduce the security and performance experiments described in the BlankIt evaluation [94]. We use the same `nginx` workload generator, `wrk`, which runs 12 threads in parallel and creates 400 concurrent connections to the server. For the performance experiment, there are 4 inputs: the home page of Wikipedia, and 3 randomized binaries of 1MB, 10MB, and 100MB. The performance experiment includes 2 separate tests. In the first test, `wrk` requests the Wikipedia home page for 3s, then 30s, then 300s. The experiment tests that `nginx` can serve a normal-sized page (80KB) under high load for extended periods of time without degrading. In the second test, `wrk` requests the 1MB binary for 30s, then the 10MB binary for 30s, and finally the 100MB binary for 30s. This experiment tests that as the request size scales, there is still no degradation. For the security experiment, only the home page of Wikipedia is used. `wrk` makes requests for 30s.

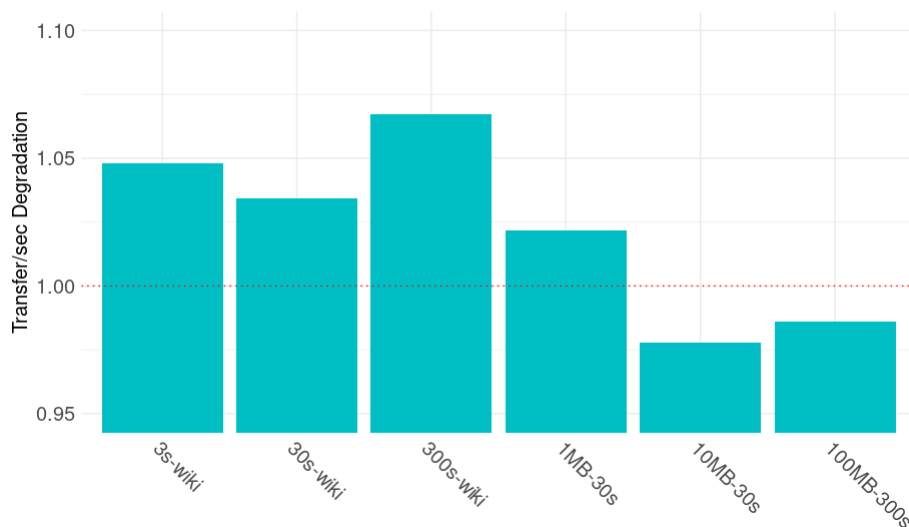


Figure 3.5: Transfer/sec degradation for `nginx` using DECKER.

The normalized performance result is shown in Figure 3.5. The slowdown is reported as the transfer/sec degradation, normalized against the baseline. DECKER achieves 1.023x slowdown on average. The total gadget reduction on `nginx` is 80.3% on average (50.3% and 95.3% minimum and maximum, respectively). The average is an improvement over SPEC but less than that for `coreutils`.

In comparison, BlankIt averages 1.047x runtime overhead and 98.9% ROP gadget reduction on `nginx`'s libraries. As with SPEC, DECKER outperforms BlankIt at runtime but with less ROP gadget reduction. BlankIt copies needed library code into place before use and zeroes it out after use. This accounts for BlankIt's higher gadget reduction, and also explains why, despite only being used on libraries, BlankIt is slower than DECKER's page-mapping scheme. Neither Chisel nor Razor has been evaluated by their respective authors on `nginx`, making it difficult to compare DECKER against them.

3.4.4 Binary Size Growth

We measure the binary size increase over every application. There is 2.9x increase across SPEC, 1.5x across `coreutils`, and 1.8x for `nginx`. In absolute terms, the modified binaries are 18.3MB, 1.2MB, and 7.1MB on average. Thus, for these applications, the binaries are still reasonably sized, despite the growth, and the performance measurements have confirmed that this has not adversely affected runtime. The improvement of `coreutils` over SPEC is due to fewer disjoint sets in `coreutils`. All binaries in `coreutils` have fewer than 200 sets, whereas the majority in SPEC have 200 or more.

The worst-case growth without any custom linking would assign a page-aligned section to every function in the program (i.e. every disjoint set would be a singleton). We estimate this case, lower-bounding it by ignoring weak function symbols in the baseline applications. Our custom linking script improves over lower-bounded, worst-case growth by 1.8x, 2.7x, and 1.3x for SPEC, `coreutils`, and `nginx`, respectively.

3.4.5 Breaking ROP Gadget Chains

Gadget reduction is a common metric in security-focused debloating works [41, 94, 43, 42, 74], but it is still difficult to draw certain conclusions. We need to investigate whether an attack surface reduction technique actually removes attacks. We start by looking at DECKER’s ability to break ROP chains.

Ropper [13] is an open source tool that can automatically build gadget chains. It allows us to automatically test if we can successfully build a ROP gadget chain that spawns `/bin/sh` via an `execve` syscall. We use Ropper to identify which binaries from all our previous experiments have this ROP gadget chain. Then we test every binary over all test inputs with DECKER and check every available page set for the ROP gadget chain. We run Ropper on SPEC, coreutils, and `nginx`. It finds that 9 of them, including `nginx`, have the full ROP gadget chain (write-what-where gadgets, argument staging gadgets, and the syscall gadget); 12 have an incomplete chain; and the rest have no components of the chain.

We analyze every available page set over all inputs across all applications with Ropper and find that DECKER does not allow the ROP chain under any set of dynamic decks. This important result includes 6,453 unique dynamic deck sets (with 14,378 dynamic execution count) over all applications and inputs. On further analysis, the breaking component in the chain is repeatedly the syscall gadget, which is rare and not loaded by DECKER in any of the 9 applications.

3.4.6 Breaking JOP Gadget Chains

Next we look at breaking JOP chains, which are more complex than ROP and also resilient against modern ROP defense mechanisms such as shadow stacks. A typical JOP chain consists of, among other details, a dispatcher gadget (DG) and dispatch table (DT) (refer to [10]).

nginx on Linux

We are not aware of any tools like Ropper that can automatically create JOP chains on Linux. Thus, we perform manual analysis for this chain-breaking study to provide a mix of quantitative and qualitative results. Our analysis is guided by two questions. First, is it possible to form a JOP chain that executes the notorious `execve` attack on `nginx` while running under DECKER? Second, is it possible to build some other useful JOP chain with the gadgets that are still available when running under DECKER?

Our first question is already answered by our previous ROP study, because, even when replacing the main portion of the ROP chain with JOP, we still fail to find the `0x0F05` syscall opcode needed for the final `execve` invocation. ***Thus, the shell-spawning JOP chain is infeasible under DECKER for JOP chains, as well.***

To answer our second question, we run ROPGadget [70] on all dynamically captured available page sets running under DECKER for `nginx` and output all JOP gadgets. Then we scan for DGs. We find none that have the most useful, traditional form (add immediate + indirect jump). There are, however, length-2 gadgets of the form `add reg1 reg0; jmp *reg0`. This is sufficient, provided that `reg1` can be set to `0x8`, for example (which could be done with a properly constructed attack payload with `0x8` and `pop reg1`). Thus, we assume these gadgets are feasible for use as DGs.

Next, for each dynamically active page set, we analyze the available JOP gadgets that would be active if a given DG were chosen. In other words, we throw out gadgets that could potentially have side effects on the two registers that are being used for a chosen DG (`reg0` and `reg1` in our example); and we do this for all DGs available for a page set. This yields several results. First, it tells us if, at some point during execution, there was even a DG available for an attacker. Second, it tells us – after setting the DG registers – which JOP gadgets remain available for launching an attack.

We report 5 metrics for this analysis, shown in Table 3.3. These report the total number of gadgets, the number of DGs, and the number of unique first operands of those gadgets.

Also, as described above in terms of choosing a DG, we report the total number of usable gadgets and the number of unique first operands that are usable once a DG and its registers are set.

Table 3.3: JOP gadget metrics over all available page sets running nginx on Linux (baseline shown in average column).

Metric	Min	Max	Avg	Stdev
all	2	106	33.9 (357)	27.9
dispatcher	1	21	6.1 (28)	5.6
uniq first op	1	28	12.1 (62)	7.4
usable	0	74	25.7 (200)	17.5
uniq-first-op usable	0	23	12.1 (50)	6.3

The standard deviation for these metrics is instructive. For example, the total gadgets for a given page set averages 33.9 (low, indicating great defense), but the standard deviation is 27.9. When there are very few gadgets available, a quick inspection shows that there is nearly nothing that could conceivably be built. We must inspect the worst-case page sets, though, to understand if JOP chaining would still be difficult under these circumstances, even without a syscall gadget.

A good candidate is the available page set that contains the most number of unique first operands in its gadget list (after choosing the DG). Table 3.3 shows that the max observed value is 23. This occurs in our page set with ID=319. Note that we have not yet enforced more sophisticated restrictions, such as choosing a register to hold the address of the DG itself (which would be necessary for a real attack), nor checking for stack pivot capability (often needed for a real attack). In the former case, this must be taken into account. For page set 319, despite having 23 unique operands, these operands are not available or substantially abundant once we select the register that will hold the DG. For instance, there is a useful indirect write gadget that requires `rcx` to hold the DG address. If we want to use it, however, there are no `pop` gadgets. Similarly, to benefit from the most `mov` gadgets (7 total), one must choose the `rsi` register to hold the DG's address, but there are again no `pop` gadgets that use `rsi` for jumping back to the DG.

In contrast, the baseline contains a much richer set of gadgets that could conceivably be used for more complex chains (Table 3.3, Avg column in parentheses). For example, there are 52 indirect write gadgets (instead of 3 for page set 319), and it is easy to identify `pop` gadgets that can use the same address for the DG. We ran another third-party tool on `nginx` to help validate this result. GSA [104] has some support for JOP gadgets and for ranking their “quality.” GSA’s total JOP score for the baseline is 789. For page set 319 it is 543.5 (31% reduction).

To summarize, an `execve`-invoking JOP chain is thwarted due to the missing `syscall` gadget. And, on closer inspection, a more novel, complex JOP chain is very difficult (no write or `mov` gadgets) or impossible to build (no usable gadgets), even when the dynamically available set of pages is most exposed. The lack of availability of a gadget breaks the necessary condition for constructing a JOP chain; we thus contend that DECKER is able to stop JOP chain-based attacks. In contrast, the baseline supports a substantially richer set of JOP gadgets; these provide the ability to launch an `execve` JOP chain; and they comprise a realistic set of *simultaneously present gadgets* for choosing registers used for the DG and DT, which is necessary to build chains beyond `execve`-invoking ones (e.g. the recent shellcodeless JOP technique [105]).

nginx on Windows

We conduct a JOP case study on Windows, as well. This is another critical platform, and it provides evidence that DECKER generalizes well beyond Linux. The only powerful open source tool for JOP gadgets is also a Windows-only tool (JOP Rocket [34]), so we can leverage it for our study.

Our goals are similar to the previous subsection. We check whether JOP chains are broken by DECKER, and whether other novel JOP chains can still be built. We use the same workloads as before and capture the available page sets. Then we run JOP Rocket on every page set recorded at runtime, as well as on a statically compiled `nginx` baseline.

In all cases, including the baseline, JOP Rocket is unable to automatically build a complete JOP chain. This does not imply that there are no JOP chains – only that JOP Rocket could not automatically construct one. (There are multiple reasons why a chain may not be found. The JOP Rocket paper at Black Hat is a useful source to learn more [34].) It is for this reason that fine-grained metrics are extremely useful in gauging the effectiveness of DECKER on debloating JOP gadgets. JOP Rocket makes this task much easier than on Linux, so we provide a more interesting set of reduction metrics in Table 3.4.

Table 3.4: JOP reduction on Windows nginx (higher is better).

Metric	Min	Max	Avg
call gadgets	0.65	1	0.78
len-2 gadgets	0.05	1	0.47
len-2 mov gadgets	0.78	1	0.81
max len-3 add ops	0.88	1	0.99
stack pivot gadgets	0.5	1	0.52
2-gadget dispatchers	0.5	1	0.6
uniq addends for add op	0.38	1	0.54
uniq dec ops	0.96	1	0.97
uniq gadgets	0.25	1	0.6
uniq inc ops	0.42	1	0.76
uniq jump regs for add op	0.5	1	0.77
uniq len-2 addends to esp	0	1	0.99
uniq mov imm insts	1	1	1
uniq sub ops	0.26	1	0.46

As with Linux, this list of metrics shows how DECKER can make JOP-based attacks much more difficult. For example, the maximum reduction for DECKER under the `nginx` workloads is 100% for each of these cases. In the average case, the results are similar to the overall numbers presented previously. For example, we report `call gadgets` here as well, showing that on average 78% are eliminated from the available page sets. Length-2 gadgets (which are usually desirable to attackers because of the limited side effects) are reduced by 47%. In one case, DECKER succeeds in eliminating an entire class of gadget under the given workload, namely the `mov-immediate` gadgets present in the baseline. Stack pivot gadgets are crucial for many JOP chains (e.g. those which need to set up arguments for a

function call). DECKER eliminates 100% of these in the best case and reduces this by 52% on average. This can be sufficient for thwarting an attack if the attacker cannot properly pivot the stack during the chain. One other interesting metric is the number of 2-gadget dispatchers, which are reduced by 60% on average. These dispatchers are introduced in [34], and JOP Rocket is capable of finding these gadgets for manually building chains from them; but as seen here, there are 60% fewer such gadgets on average due to DECKER.

3.5 Related Work

Control flow integrity (CFI) [16, 100] is a traditional defense that limits forward and backward control flow transfers to legal edges in the control flow graph and callgraph. CFI has a rich history, addressing a variety of scenarios, contexts, and attacks [18, 19, 108, 21, 109, 23, 20, 17, 25, 106, 27, 107]. Despite these advancements, current state-of-the-art still has its shortcomings. There are numerous examples of how to bypass CFI (e.g. [28]) or what its limits are (e.g. [15]). In fact recent work [110] thoroughly categorizes the shortcomings of several CFI systems, and which they broadly characterize as: imprecise analysis methods, improper runtime assumptions, unprotected corner code, unexpected optimization, incorrect implementation, mismatched specification, and unintended targets. For example, μ CFI [21] cannot protect against code pointer reuse and VTable attacks. OS-CFI [111] fails to protect against tail calls that are optimized for indirect calls.

State-of-the-art, industry solutions such as Windows' CFG and RFG [112, 113, 114] and Intel's CET [115] are still vulnerable to code reuse attacks [34]. Code replacement attacks, counterfeit object-oriented programming [39], data-only corruption, function pointer through race condition attacks, and thread context hijacking by abusing the NTContinue mechanism are all techniques for attacking Intel CET [38]. Windows 10 still relies on software-based CFG (not Intel's IBT [116]), which is more susceptible to bypasses [35, 36, 37]. Attackers can also avoid the defenses (targeting Windows 7 systems, different hardware, applications and libraries built without support, inline assembly, opcode splitting,

etc.).

Control pointer integrity (CPI) [95, 96] attempts to guarantee the integrity of all code pointers. Though effective, certain attacks may fall out of its scope (e.g. control-flow bending [15]). CPI’s memory safety is just at the level of code pointers. CFI, in contrast, elides memory safety but attempts to verify code targets. Both are about ensuring correct control flow.

Techniques such as CFI and CPI are *orthogonal* and *complementary* to debloating and attack surface reduction techniques. Approaches such as DECKER, Piece-wise [41], or Razor [43] do not try to solve control flow misdirection. Rather, they try to remove vulnerable code that can be used to launch an attack, or which could be leveraged once an attack occurs. These aims are complementary because a weakness in a CFI technique, for example, could be mitigated by attack surface reduction like DECKER, and vice versa. For example, if a control-flow bending attack or privilege execution attack succeeds in bypassing a CFI defense, there is still a chance that the gadgets needed to complete the attack are debloated. Likewise, if a debloated program still contains a full gadget chain at a certain point during its execution, the attacker may not be able to trigger that code due to a CFI defense.

Other attack surface reduction works not yet mentioned include several feature-based techniques. Slimium [117] debloats Chromium features based on a static-, dynamic-, and heuristic-based analysis. Koo et al. take a configuration-driven approach to remove feature-specific code [118], achieving 77% debloat on `nginx`. Trimmer [74] is another technique that takes as input a user configuration and uses it to drive the debloating process. Soto-Valero et al. [119] leverage existing coverage tools to achieve similar debloating percentages for Java library bytecode.

ELFbac [120] is similar to DECKER in that it attempts to restrict segments of ELF files. Users create data or execution policies that specify access relations among ELF sections. These policies work at the level of ELF metadata. It does not work at DECKER’s more granular level of control flow at function calls. The ELF format supports 30 sections,

and the programmer can manually create more within the code. In DECKER, there is no programmer intervention, and DECKER more easily extends to other formats (like Windows PE).

Lastly, software engineering researchers have worked on debloating, but the focus has not traditionally been on security. For instance, [80, 81, 82, 83, 84, 85, 86] leverage debloating to improve performance, and [87, 88, 89, 90] use it to reduce code size.

Contrasted with the above approaches, the main contribution of this chapter is to develop a whole application debloating technique which is sound and has low runtime overheads with high mitigation capability. In terms of adoption, code reuse attacks are recognized as serious threats in industry, and multiple products already employ some type of defense (e.g. see Intel CET/IBT and Windows CFG/RFG above). We believe that DECKER has an edge over such efforts, as well, due to it being a software-only technique with full automation.

3.6 Conclusion

We present DECKER, an attack surface reduction technique that works on full programs and can enable may-use code on-demand. It achieves state-of-the-art gadget reduction results without compromising soundness or requiring training, user inputs, or specifications. Total gadget reduction across SPEC CPU 2017, GNU coreutils, and the nginx server average 73.2%, 87.2% and 80.3%, respectively. In our performance experiments, the runtime slowdown on SPEC is 5.2% and negligible for GNU coreutils; the transfer/sec degradation for nginx is only 2%. In an additional study over these applications, we show that for all test inputs, DECKER eliminates the presence of a ROP chain that spawns a shell via `execve`. We show similar results in an nginx JOP study for both Linux and Windows. Based on these results and the generality of the approach, we find DECKER to be a promising technique for attack surface reduction in practice.

CHAPTER 4

PDSG: PREDICTIVE DEBLOAT WITH STATIC GUARANTEES

4.1 Introduction

We have so far presented two frameworks for attack surface reduction. DECKER currently achieves 73%, 87%, and 80% overall gadget reductions on SPEC 2017, GNU coreutils, and nginx. BLANKIT is in the vicinity of 95% or higher on a similar suite of programs. BLANKIT’s reductions are better, and we would like to achieve similar results in DECKER. One reason for the difference is related to the problems they tackle: BLANKIT works on libraries, whereas DECKER works on application code. Because only small portions of a library are only ever needed at run time, library debloating lends itself to high-percentage reductions. Another factor that works in favor of BLANKIT is its prediction capability, which is not inherently restricted to debloating library code. *It is an open question how to effectively apply prediction at the whole-application level for debloating.*

In this piece of work, our goal is to refine debloating of application code using runtime predictions and callgraph path checking. Our idea is to train a model that will predict the upcoming regions of code (in the same spirit as BLANKIT), but to do it on a whole-application basis (in the same context as DECKER). The solution must still be performant, and it must still be sound. In addition, we want to leverage static analysis techniques to check that mispredicted control flow still conforms to the original program’s callgraph at runtime to be able to distinguish between mispredictions and attacks. *In other words, we want to combine static analysis techniques with machine learning to improve attack surface reductions and still guarantee certain control flow properties of the program are met.*

In summary, this work makes the following contributions:

1. To the best of our knowledge, it is the first prediction-based approach for whole-

application debloating that is performant and sound.

2. It is the first debloating technique to leverage static analysis techniques that guarantee certain program properties are met whenever mispredictions due to ML occur.
3. It includes an empirical evaluation that shows the technique improves attack surface reduction beyond the state of the art and with overheads that are in line with prior art.

4.2 Overview

We present a high-level overview of our technique, predictive debloat with guarantees (PDSG). There are two modes/stages to the tool: profiling and release. A user first builds and runs their application with PDSG in profiling mode to capture logs of its behavior and to train a model. Then the user rebuilds and runs their application with PDSG in release mode. The release build performs additional static analysis, and running in release mode leverages that analysis, along with the model created during the profiling stage, to reduce the program’s attack surface.

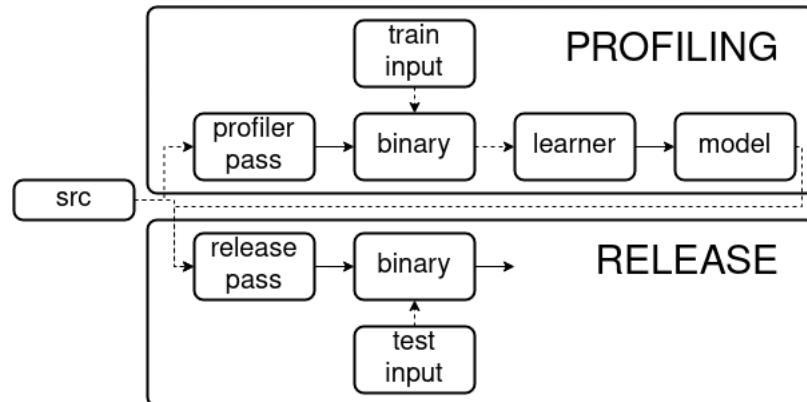


Figure 4.1: High-level flow diagram of PDSG’s profiling and release stages. Dashed arrows indicate “feeds into;” solid arrows indicate “creates.”

Figure 4.1 depicts a high-level flow diagram of the framework. The profiling component is shown in the top half of the figure. It is responsible for instrumenting the application source code (using PDSG’s custom LLVM profiler pass), producing logs for model training

(by running the profiling binary on training input), and training the model for the release stage (by using the logs generated during the binary’s training runs and passing this through a learner). The release stage is shown in the bottom half of Figure 4.1. It is responsible for instrumenting the application source code (using PDSG’s custom LLVM release pass and the model generated during profiling), and running the binary on test input (i.e. real input for exercising the model and debloating functionality).

In a release run, PDSG’s runtime system is responsible for actually reducing the attack surface. It works at the granularity of functions and system code pages. As the program executes, the runtime system issues predictions of upcoming functions that it expects will be invoked in the near future. Predicted functions’ code pages are marked read-or-execute (RX), and unneeded functions’ code pages are marked read-only (RO). The runtime system must also handle mispredictions by checking the dynamic behavior against static properties of the program.

Figure 4.2 and Listing 4.1 illustrate the release mode’s prediction mechanism at a high level. In the figure, assume that the execution has flowed from function *A* to *B* to *D*, at which point a prediction mechanism is triggered; and in the listing, the pseudocode for function *D* is shown. *D* has a call to `predict` just above the while loop. This `predict` call is instrumented by PDSG’s LLVM release pass. During the application’s execution, `predict` invokes PDSG’s runtime system, which predicts the forthcoming functions. In this example, the runtime predicts *F*, *J*, and *M*. The runtime will mark these functions as RX before entering the loop. The right side of Figure 4.2 makes it explicit that these functions occupy some pages in system memory (not necessarily continuous or in order), and it is only these upcoming sections of the memory that will be enabled by this particular prediction.

Listing 4.1: Example code for demonstrating release instrumentation and callgraph prediction.

```
D (...) {
```

```

...
predict (...)
while (...)
    if (...)
        F (...)
    else
        E (...)
...
}

```

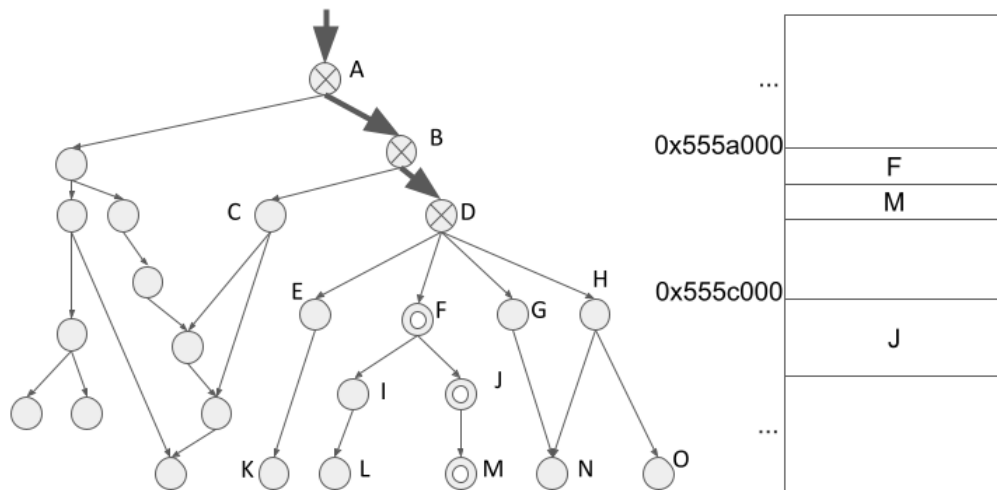


Figure 4.2: High-level depiction of execution flow (X nodes) and prediction (donut nodes) through a callgraph, and a corresponding memory layout of the predicted functions.

4.2.1 Challenges

Applying a prediction technique with runtime checking on whole applications raises several new challenges that must be solved:

- **Prediction:** It is unknown what kind of model is effective for this problem, what the features should be, and where the release pass should instrument the `predict` calls.

- **Rectification:** On misprediction, the program will attempt to call a function that is not yet mapped executable. PDSG must identify these “rectification points” in the program and map the necessary functions to avoid crashes. It is unclear which functions should be mapped at these points or how functions in the .text section should be laid out to accommodate all of this. Again, it is also unclear where the release pass should instrument the rectification points.
- **Path checking:** Finally, on a misprediction, it is important to distinguish between a misprediction and an attack. In order to do so, one must answer questions such as: Which properties of the program can and should be checked? How does one construct such a static model that is also fast for runtime-checking?

4.3 Framework

PDSG has three technical components, which address the three challenges described at the end of section 4.2:

1. **Prediction:** PDSG profiles the application using realistic inputs and uses the training data to build an ML model. This model is then embedded back into the release version of the application and issues predictions at runtime. The model predicts upcoming regions of the callgraph that are expected to execute.
2. **Rectification:** The runtime framework is relying on predictions to determine what code should be mapped active, but it is still possible for mispredictions to occur. When they do, the runtime must rectify this by mapping the needed code.
3. **Path checking:** In addition to mapping necessary code at a misprediction point, the execution path of the program leading up to that point should be checked and validated. That is, the misprediction point is an opportunity to compare the dynamic execution trace with statically known invariants of the program structure, which could catch additional attacks.

4.3.1 Prediction

We consider a naive prediction scheme, followed by a more robust approach, and then discuss the model itself.

Naive Prediction Scheme

One could implement a method to predict callgraph behavior without taking any of the callgraph structure into account. We describe such an approach, because its shortcomings motivate the need to develop a more sophisticated technique.

The idea of our naive approach is to treat the function calls of an executing program as a sequence (ignoring program structure entirely). That is, given a sequence of the last X previously executed functions, the goal is to train a model that will predict the next Y functions in the future. Sequence-based prediction is a well-studied problem in machine learning, and we choose one of its most popular, basic solutions to model this problem, long short-term memory (LSTM) [121]. Using offline traces, we first study the efficacy of such an approach.

We run SPEC CPU 2006 benchmarks with training input and with function tracing enabled. This outputs an ordered sequence of the executed functions. We feed this as training data to a simple keras implementation of an LSTM, and this produces a trained model. We then rerun the benchmark suite using its reference inputs, and we use its output trace to test the model. On average, the prediction accuracy is 91.1%.

There are several critical shortcomings of these results. The accuracy may be sufficient for the problem context ($\sim 10\%$ misprediction rate could still be handled, potentially). To achieve these results, however, the scheme takes several shortcuts. The traces are very large (10s to 100s of GBs) – so large, in fact, that they cannot reasonably be managed on a commodity desktop machine. One engineering technique could be to pipe these logs on-the-fly into a learner, which complicates design but is feasible. Instead, we choose a simpler technique for this naive scheme, which is to increase the inlining threshold substantially (which

reduces function calls and therefore the log sizes). Note that this increases compilation time and modifies the application beyond what the original programmer may expect (i.e. is probably not tenable in a real-world scenario). Nevertheless, this approach allows us to train a model for most of the benchmarks in the suite. Unfortunately, the logs are still too large for several benchmarks, even on SPEC CPU 2006’s smallest input. In these cases, we must short-circuit during tracing and ignore certain callsites that are invoked too frequently. This is an unrealistic strategy for real-world applications, and it also negatively impacts prediction accuracy.

Another serious shortcoming is how to reasonably handle the mispredictions. Because of the sequential nature of this approach, *every* function is a candidate for misprediction. If we were to implement this kind of prediction in a real application with debloating enabled, we would need instrumentation and prediction-handling mechanisms at the entry-point of all functions. This is not realistic from a performance standpoint.

In summary, this naive approach shows that whole applications seem to exhibit predictable behavior with even a simple call sequence-based predictor, but it is also untenable. It shows that by ignoring the program structure (loops, essentially), the profiling data becomes unmanageable for normal use cases. There is also no clear way of handling mispredictions when function calls are simply flattened into a kind of stream or sequence.

Program Structure-based Profiling

Based on our results from the naive scheme, we are motivated to look at an approach that takes the program structure into account. We adopt a similar static analysis approach as the one in DECKER, which describes how to identify “good” instrumentation points, i.e. points in the program that will still lead to effective debloating while limiting the impact on performance. In brief, we need to take special care to instrument interprocedurally outermost loops and indirect function calls, and to identify a “sub-callgraph” (SCG) reachable from those points.

The model we are training answers the following question: *Given a particular sub-callgraph and features, which functions of the sub-callgraph are expected to execute?* More precisely, the target of the model is an integer ID that represents a subset of functions in the sub-callgraph that will execute. We call this the “predicted sub-callgraph” (**PSCG**). The features of the model are the arguments to the function entry-point of the SCG, or, when a loop is the entry-point to an SCG, the arguments to the parent function of that loop.

The purpose of our profiling instrumentation is to provide these features and target values as sample data for training. To capture the features, we instrument a logging function at the entry-point to SCGs that prints the arguments to the function being called (or, in the case of loops, the arguments to its parent function). To capture the target values, we trace all functions – as we would in a naive, sequence-based scheme, but with the critical difference that we also provide call stack information in the trace. The call stack information allows us to reconstruct the functions that are executed within the SCG, and therefore to assign IDs (the target values) for training.

Note that the compiler instrumentation pass for release is simpler than the pass for profiling (described above). In short, a release binary does not need any instrumentation for call stack-based tracing, and instead of instrumenting a logging function at the entry-point to SCGs, it must instrument a model-invoke function.

Model

We use decision trees as our model for several reasons. Empirical evidence in BLANKIT already shows that they can be effective predictors of execution path behavior based on dynamic program features (such as the runtime arguments passed to a function call). Second, it is straightforward to instrument them as if-else branches. This is important for performance, because the new framework must support model invocation within loops and critical paths of the program. In contrast, invoking a complex model in a third-party library at runtime to predict the upcoming functions would be prohibitively expensive.

Note that the training data for PDSG is also fundamentally different from input or test cases for debloating works with unsound transformations. That is, the training data for PDSG is not a stand-in for the desired features, functionality, or configuration, nor must its fidelity be perfect because of potential soundness issues. For example, in Razor [43], the test cases are used to uncover the features and ultimately yield a useful binary that hopefully will not crash. Similarly, in Chisel [42], the hope is that the training data captures the expected features as best as possible, because the model can make unsound cuts to the program. In contrast, the training data for PDSG, though it ought to be good, does not affect the soundness of the resulting binary. The worst-case scenario for PDSG is that it always mispredicts, but it would still be sound because of its rectification component.

4.3.2 Rectification

The model and its supporting instrumentation are designed to activate functions at runtime of the predicted sub-callgraphs (PSCGs) rather than the entire sub-callgraphs (SCGs) at each instrumentation point, thereby further reducing the attack surface of the program. Rectification is about how to handle mispredictions so that the program does not crash. That is, when the program attempts to execute code outside of the activated PSCG, the rectification component must activate the necessary code in the SCG.

Rectifying mispredictions at runtime has two characteristics:

1. It guarantees soundness of the application even when using prediction-guided code mappings.
2. It adds security beyond that of on-the-fly, binary-level debloating techniques like BLANKIT that effectively have to check for rectification at *all* function calls.

Rectification guarantees soundness by ensuring that before a function executes, it is guaranteed to have an active mapping. For predicted functions, this is trivially true (no rectification is needed). For functions that fall outside of the PSCG, PDSG's release pass

must instrument all edges from the PSCG to the SCG. The PDSG runtime will map those functions as active before continuing execution. Rectification yields additional security beyond schemes like BLANKIT because PDSG instruments only at select points in the callgraph, which we call “rectification points” (**RP**s). In schemes such as BLANKIT, all function calls are allowed to proceed, though mispredictions trigger a mitigation strategy. In PDSG, however, *only function calls at RPs are allowed to proceed* (and mispredictions will trigger a path-checking strategy, detailed in the next subsection). We show an example to help illustrate these properties and aid understanding of our rectification algorithm.

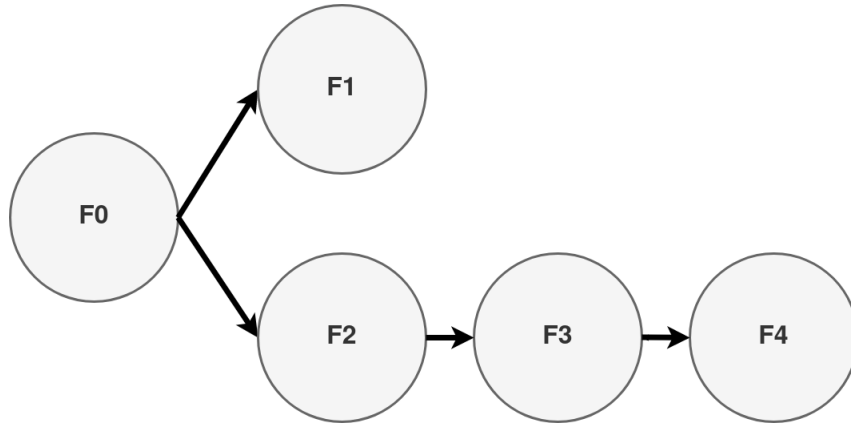


Figure 4.3: Example callgraph for demonstrating rectification.

Figure 4.3 shows an example callgraph of five functions, $F0 - F4$. For the purposes of this example, we assume that this callgraph forms an SCG and that the root of the SCG is $F0$. Assume also that our training leads to two PSCGs that get called within the SCG: $A = \{F0, F1, F2\}$ and $B = \{F0, F2, F3\}$. We will show that in this example, we need 3 RPs: $RP_{A0} = F2 \rightarrow F3$, $RP_{B0} = F0 \rightarrow F1$, and $RP_{B1} = F3 \rightarrow F4$.

The high-level reasoning for these RPs is as follows. The PSCGs will only ever activate set A or set B (never both, and never anything outside of those sets). Therefore, whenever the predicted set is A , a misprediction means that $F3$ and $F4$ must be mapped at an RP, namely RP_{A0} (because otherwise a crash could occur). Similarly, whenever the predicted set is B , a misprediction means that $F1$ and $F4$ must be mapped at an RP; in this case, they are not connected, so two RPs are required, namely RP_{B0} and RP_{B1} .

The exact implementation mechanisms and semantics for RPs can vary. For example, at RP_{B0} , the framework can mark both $F1$ and $F4$ as active (and disable RP_{B1}). Alternatively, it could service only $F1$ and leave RP_{B1} enabled (which will service $F4$, should it be needed). Both are reasonable schemes depending on the expected runtime overheads and prediction accuracy. This can be refined through further empirical studies, but for the purposes of this work, we use the former approach.

To return to our prior point that this rectification can provide security *beyond* that of on-the-fly, binary-level debloating techniques like BLANKIT, notice that all possible prediction sets are known statically. (Likewise, all misprediction sets are known statically.) Therefore, if we take the example of prediction set A , we know at build time that a misprediction includes only functions $F3$ and $F4$, and that $F4$ is dominated by $F3$ in the callgraph. In other words, there is no way for execution to go from $F0$, $F1$, or $F2$ directly to $F4$, and thus there is no RP to handle code activation along such a path. This “gate-like” behavior is not present in techniques like BLANKIT which map any function on demand and depend on a mitigation strategy to catch any misprediction.

The pseudocode for instrumenting at RPs is presented in Algorithm 5. The algorithm iterates over the PSCGs and for each one conducts a depth-first search over its functions. If it finds a callsite to a callee that is not part of the PSCG, it inserts an RP at the callsite (`add_rp` in Algorithm 5). `add_rp` simply instruments a call to the PDSG runtime, which will actually map the statically known, remaining functions of the SCG during execution.

Algorithm 5 Pseudocode for instrumenting at the rectification points for all PSCGs of a given SCG.

```

function INSTRUMENT_AT_RPS( $SCG, PSCGs$ )
  for  $PSCG$  in  $PSCGs$  do
     $rectify\_SCG \leftarrow SCG \setminus PSCG$ 
    for  $func$  in  $PSCG.DFS\_NEXT\_NODE()$  do
      for  $callsite$  in  $func.CALLSITES()$  do
        if  $callsite.callee$  not in  $PSCG$  then
           $callsite.ADD\_RP(rectify\_SCG)$ 

```

There is a precedent for this kind of approach in CFI-based security techniques. In

π CFI [26], the authors implement a control flow graph (CFG) called an “enforced CFG” (ECFG). The ECFG is an application’s original CFG with indirect call edges disabled. When an indirect call is invoked at runtime, π CFI verifies that its function target conforms to the static callgraph and the dynamic execution environment. If it does, π CFI enables that edge in the ECFG (and leaves it enabled for the duration of the program). Rectification is similar but with some key differences. One difference is that rectification does not always grow the attack surface. It is cleared when an SCG is torn down. A second difference is that π CFI activates indirect call edges along the ECFG, whereas the proposed rectification scheme activates functions within SCGs.

4.3.3 Path Checking

PDSG’s path-checking component handles dynamic, ML-based mispredictions based on program properties derived from static program analysis. Path checking is a key motivation for this work, because other prediction mechanisms for debloating ([42, 94]) do not give any formal guarantees for checking mispredictions. It offers a security improvement *beyond* that already provided by the attack surface reduction from the prediction component, and from the gating benefit from the rectification component. The insight is that static analysis can “package” a multitude of useful program properties for the runtime to check whenever there is a misprediction. PDSG records a dynamic call trace so that on misprediction, it can compare that trace against static properties of the program. This allows PDSG to handle mispredictions with less overhead and clearer guarantees than a heavy-weight mitigation strategy like in BLANKIT.

The shape of this solution is as follows: At build time, PDSG encodes static program properties in Datalog, which serves as a deductive database. Then at runtime, PDSG runs the program with tracing enabled and on misprediction, queries the database to check that the trace satisfies the properties.

The major challenges to be tackled are:

1. What static properties are useful, and how do we define them?
2. How do we capture those properties in a practical compiler setting?
3. How do we perform tracing? It must not degrade security, and it should have little to no effect on performance.
4. How do we perform the checking online?

Defining Static Program Properties

For our initial prototype, we aim to capture one key property, the **ensue relation**, which we can use for basic validation on mispredictions. We define the *ensue* relation as follows:

- A *call sequence* is an ordered list of callsite IDs, as they are encountered in an executing program.
- An *ensue relation* is simply a pair of adjacent callsite IDs within a valid call sequence.

An example will make this clearer. Consider the code in Listing 4.2. Possible call sequences include: $0.main \rightarrow 1.A \rightarrow 5.C \rightarrow 3.D \rightarrow 4.E$ and $0.main \rightarrow 2.B \rightarrow 3.D \rightarrow 4.E$. *Ensue* relations include: $ensue(0.main, 1.A)$, $ensue(0.main, 2.B)$, $ensue(1.A, 5.C)$, $ensue(5.C, 3.D)$, $ensue(2.B, 3.D)$, $ensue(3.D, 4.E)$. Notice that the *ensue* relation is determined by both the callgraph and the control flow graph.

Listing 4.2: Example code for demonstrating various static relations.

```
main () {
    if (...) {
        1. A()
    } else {
        2. B()
    }
}
```

```

    3. D()
    4. E()
}
A() {
    5. C()
}
B(){} C(){} D(){} E(){}

```

We identify seven static program properties in order to fully capture the ensue relation:

head(f, i): The first subroutine invoked by function f is at callsite i . Referring to Listing 4.2 as an example, the head relations are: $head(main, 1)$, $head(main, 2)$, and $head(A, 5)$.

tail(f, i): The last subroutine invoked by function f is at callsite i . The tail relations are: $tail(main, 4)$, $tail(A, 5)$

next(f, i, j): The callsites i and j are adjacent in function f . The next relations are: $next(main, 1, 3)$, $next(main, 2, 3)$, and $next(main, 3, 4)$.

leaf(f): The function f does not invoke any subroutine. The leaf relations are: $leaf(B)$, $leaf(C)$, $leaf(D)$, and $leaf(E)$.

belong(i, f): The function called at callsite i is function f . The belong relations are: $belong(0, main)$, $belong(1, A)$, $belong(2, B)$, $belong(3, D)$, $belong(4, E)$, and $belong(5, C)$.

last(f, i): The function called at callsite i is the last function to be pushed to the stack before the stack starts popping all the way down to function f . This is a derived relation:

$$last(f, i) :- tail(f, i), belong(i, g), leaf(g).$$

$$last(f, i) :- tail(f, j), belong(j, g), last(g, i).$$

ensue(i, j): The function called at callsite i immediately precedes the function called at

callsite j in some valid call sequence. This is a derived relation:

$$ensue(i, j) :- head(f, j), belong(i, f).$$

$$ensue(i, j) :- next(g, i, j), belong(i, f), leaf(f).$$

$$ensue(i, j) :- next(g, k, j), belong(k, f), last(f, i).$$

Capturing Static Program Properties

We implement the first five of these properties within LLVM. The remaining two (*last* and *ensue*) are derived relations that we capture solely in Datalog once we prime its database with the other five facts.

The *leaf* and *belong* relations are straightforward. PDSG simply needs an LLVM pass that scans all functions and enumerates their callsites. Each callsite i in function f is added as *belong*(i, f), and any function f with no callsites is added as *leaf*(f).

The *head* and *tail* relations could be written as classic graph traversals of the CFG and reverse CFG. The *next* relation, however, is non-trivial once the CFG is moderately complex. Thus, for these three relations, we lean on dataflow techniques and define *IN* and *OUT* transfer functions.

We define two sets of transfers functions, *AHEAD* and *BEHIND*. These transfer functions capture the callsites immediately ahead and behind of a given point in the CFG, which is precisely the information needed to capture the *head*, *tail*, and *next* relations. Intuitively, to find out the callsites immediately ahead of a block, the information in front must be propagated backward. Similarly, to find out the callsites immediately behind a block, the information behind must be propagated forward. Thus, the *AHEAD* transfer functions (backward flow) are:

$$OUT_AHEAD[B] = \cup_{S \text{ succ of } B} IN_AHEAD[S]$$

$$IN_AHEAD[B] = Callsite_B \parallel OUT_AHEAD[B]$$

The *BEHIND* transfer functions (forward flow) are:

$$\begin{aligned}
 IN_BEHIND[B] &= \cup_{P \text{ pred of } B} OUT_BEHIND[P] \\
 OUT_BEHIND[B] &= Callsite_B \parallel IN_BEHIND[B]
 \end{aligned}$$

We do not show the pseudocode for building these sets, but they are similar to algorithms for other dataflow equations. The algorithm iterates over the basic blocks of a function, performing updates to the sets according to the above. Iteration ceases once there are no more changes to the sets.

After the PDSG release pass creates these sets, it must actually build the *head*, *tail*, and *next* facts from them. The pseudocode for this is shown in Algorithm 6. Briefly, the *head* of a function is simply *IN_AHEAD* of the entry block. *tail* and *next* require us to traverse the CFG and visit each block. The local functions `t_visit` and `n_visit` build the *tail* and *next* facts, respectively. *tail* consists of all callsites in the *OUT_BEHIND* set of a function’s exit blocks. A *next* relation is added between every block’s callsite and the callsites in its *OUT_AHEAD* set.

Online Path Checking

We have described seven static program properties, including one derived from the others called the *ensue* relation, and we have shown how to capture these using Datalog and LLVM. To perform online path checking, we also need tracing support at function granularity. There are multiple options for this, including hardware support (e.g. Intel PT), dynamic binary instrumentation (e.g. Intel Pin [63]), or simply compiler instrumentation. For the PDSG prototype, we use the compiler to add basic tracing. Then during execution, and on misprediction, the PDSG runtime must verify that the (traced) call sequence history is a valid chain of *ensue* relations. The checking can be done sequentially (i.e. in the critical path of the application), in parallel, or written to file and processed offline. In our

Algorithm 6 Capturing *head*, *tail*, and *next* sets from the transfer functions.

```
function BUILD_DL_FACTS(F)
  head[F] ← IN_AHEAD[F.GETENTRYBLOCK()]
  stack.PUSH(F.entry_block)
  while !stack.EMPTY() do
    B ← stack.POP()
    if B.visited then continue
    T_VISIT(B)
    N_VISIT(B)
    B.visited ← true
    stack.PUSH(B.successors)

function T_VISIT(B)
  if B.ISEXITBLOCK() then
    tail[F].INSERT(OUT_BEHIND[B])

function N_VISIT(B)
  if B.callsite then
    for callsite in OUT_AHEAD[B] do
      next[F].INSERT(B.callsite, callsite)
```

experiments, we assume the checks are done sequentially. The goal is to check that all adjacent pairs of executed functions in the call sequence adhere to the ensue relation. PDSG maintains the ensue relations in memory as a map of sets (i.e. to check $ensue(a, b)$, one can check if b is in the set returned by the map $ensue$ at key a). Under no-attack scenarios, all queries should return true.

Note that the ensue relation does not capture function pointers. This is a fundamental limitation due to static pointer analysis, which cannot always identify a must-points-to singleton. If there is a call sequence $1.A \rightarrow 2.B \rightarrow 3.C$ in the trace, and function B is indirectly called, then neither $ensue(1, 2)$ nor $ensue(2, 3)$ will be checked.

4.3.4 Putting It Back Together

Now that we have described the framework's components in detail, we want to step back to clarify again how they fit together. From the user's standpoint, they build an application with PDSG and profile it using representative inputs. This creates a log that gets automat-

ically fed into a learner for training a model. The user then recompiles their application with PDSG and the ML model. This final binary contains the necessary instrumentation and linking so that when it runs, PDSG will dynamically reduce its attack surface.

To achieve this attack surface reduction, several of PDSG’s features are working in concert. As the program is executing, the function calls are being tracked. Then at key points in the callgraph, the prediction mechanism is invoked, and PDSG enables the upcoming region of code that it expects to execute (i.e. the predicted sub-callgraph, PSCG). As mentioned, the output of the predictor is a single-value integer ID, which represents the *set* of functions in the PSCG. By design, this set is always a subset of the SCG. In other words, at runtime PDSG is periodically enabling upcoming functions that it expects will execute, and these functions are always within the statically known and reachable portion from the current point in the callgraph.

When execution flows outside of the current PSCG, this is a misprediction, and it must be handled. PDSG checks that its call sequence history conforms to the database of ensue relations. That is, if a pair of calls is an invalid ensue relation, then PDSG has detected an attack (or bug), and can act accordingly (alert, exit, etc.). If, on the other hand, the call sequence history is verified against the ensue relations, then the program continues as normal.

4.4 Evaluation

All experiments are run on one machine using an AMD Ryzen 7 with 32GB RAM. All experiments are on SPEC CPU 2017. We use LLVM v11, python v3.6.9 with sklearn v0.22.1, and Datalog v2.6. Our evaluation is focused on the following questions:

1. Does a predictive technique on whole applications significantly improve attack surface reduction in terms of gadget count?
2. What is the performance overhead of PDSG?

3. What are the rectification and prediction characteristics, both offline and online (count, frequency, accuracy, etc.)?
4. Does capturing the ensue relation with Datalog scale for complex program behavior?

4.4.1 Gadget Reduction

The gadget reductions are shown in Table 4.1. This is for all gadget types (ROP, JOP, COP, and special-purpose) reported by the GSA tool [104]. Because of PDSG’s dynamic behavior, the gadget reductions fluctuate over the duration of the program, depending on the active PSCGs and SCGs. Thus, the minimum and maximum columns represent the worst- and best-case reductions during each application’s execution. The lowest reduction across all benchmarks at any point is for `mcf` with 24% reduction; and the highest reduction is nearly 100% (after rounding) for both `perlbench` and `xalanbmk`. On average, however, the minimum and maximum reductions tend to fall between 60 and 90%, with the average being 82.5%.

Compared with other state-of-the-art techniques, PDSG’s gadget reductions appear to be an advancement. Of the other sound debloating techniques, Piece-wise debloats 72.9% of total gadgets from libraries; BLANKIT debloats 97.8% ROP gadgets from libraries; LMCAS debloats 55.9% ROP gadgets from applications; and DECKER debloats 73.2% total gadgets from applications. Thus, PDSG is nearly a 10% improvement over the nearest, closely related prior work, DECKER, and more than 25% improvement over the other sound, application-debloating technique, LMCAS.

On closer inspection, PDSG appears to be acting (as intended) to restrict the functions within the SCGs to a narrow predicted subset. For example, on SPEC CPU 2017, DECKER reports that on average, functions can statically reach ~ 45 other functions; similarly, interprocedurally outermost loop headers encompass ~ 35 functions. Though these averages collapse the differences among the benchmarks, it is useful to compare these with the average cardinality of the PSCGs reported by PDSG, which is ~ 6 functions; simi-

larly the complement sets of the PSCGs hold ~ 35 functions on average. In other words, PDSG’s trained model is much more restrictive than enabling the otherwise large, statically reachable function sets; and the benefit to attack surface reduction is limited by the mispredictions and unseen inputs not available during training.

Table 4.1: SPEC CPU 2017 total gadget reduction as a percentage (higher is better).

Application	Min	Max	Avg
perlbench	38.8	100.0	67.9
gcc	45.1	99.8	79.9
mcf	24.8	70.0	59.5
namd	77.0	96.1	92.2
parest	90.4	99.9	98.3
povray	59.1	99.5	78.5
lbm	53.0	65.5	59.2
omnetpp	73.1	99.3	90.3
xalancbmk	82.8	100.0	92.9
x264	41.4	96.9	63.5
blender	93.2	99.7	99.6
deepsjeng	51.9	87.5	84.6
imagick	52.6	99.4	94.8
leela	48.8	87.8	81.8
nab	79.0	99.7	94.3
xz	66.4	95.3	82.6
AVERAGE	61.1	93.5	82.5

4.4.2 Performance Overhead

The performance overhead is shown in Figure 4.4. The average overhead is 10.9%. The other debloating techniques with runtime components, DECKER and BLANKIT, report 5.2% and 18% overhead on SPEC, respectively. (Note that BLANKIT protects libraries, however, and that it reports for SPEC 2006, not 2017). Compared with these frameworks, PDSG appears to fall somewhere between the two techniques.

In Figure 4.4, there seem to be roughly two “categories” of benchmarks. Several have low overheads (less than 5%), while others are high (more than 15%). This suggests that for the class of benchmarks where the overhead is low, PDSG seems to be providing sub-

stantial benefit for little cost. For others, the overhead may be too high at present and not justifiable. To take one example of each, `deepsjeng` has nearly no overhead when running with PDSG, and the average total gadget reduction is 84.6% (Table 4.1). In contrast, PDSG achieves 59.5% total gadget reduction for `mcf` but with almost 20% overhead.

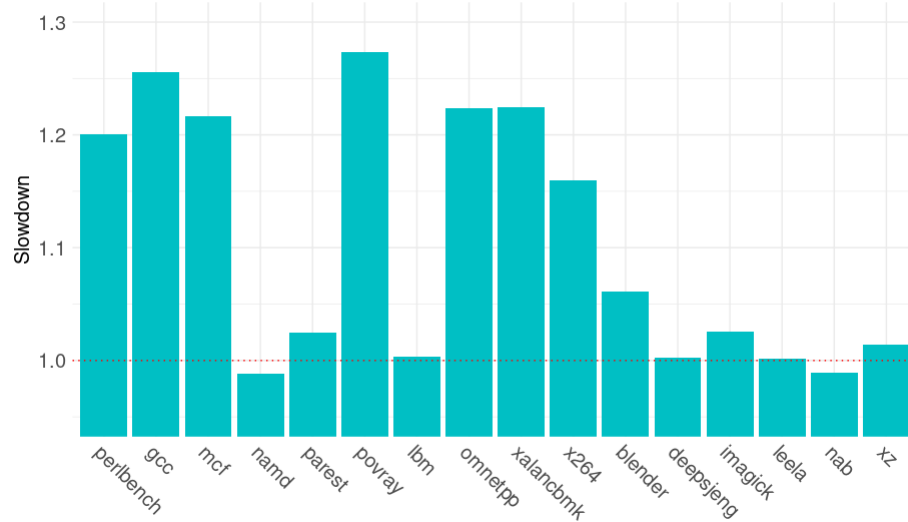


Figure 4.4: Slowdown for SPEC CPU 2017 using PDSG.

The PDSG release binaries have several components that add to the overhead beyond the slowdowns already seen in a system like DECKER. These are the predictions and rectifications, the tracing, and the ensue querying. The cost of the instrumentation (both outside and inside of loops) is otherwise similar to DECKER. In other words, PDSG’s additional $\sim 5.7\%$ slowdown over DECKER is from these components. We isolate these components and measure the contributions. Tracing adds $\sim 0.5\%$, and there is $\sim 0\text{-}2\%$ overhead from the ensue queries. The remaining overhead (slightly over half) is due to the prediction and rectification. We also test with larger ensue buffers (in the critical path and without blocking), and see ensue querying add up to $\sim 4\%$ overhead for 16-element ensue buffers (vs. a simpler two-element history). We notice seven of the SPEC benchmarks have decision trees with max depths over 10 levels: `perlbench`, `gcc`, `parest`, `omnetpp`, `xalancbmk`, `x264`, and `blender`. We find this corresponds closely with worst-case slowdowns from Figure 4.4, which merits further investigation (i.e. more closely measuring and then check-

ing how limiting their tree depths affects the overall security and performance).

Another focus of our performance overhead study is the contribution from indirect function calls. Of all the predictions issued by PDSG, $\sim 32\%$ of these are due to indirect calls. More importantly, because they require instrumentation inside of loops, they are heavy contributors to the slowdown. Note that like DECKER, PDSG “caches” active code pages inside of loops (i.e. keeps them active until loop exit) to avoid overhead. Thus, the prediction cost to performance should only be paid once per indirect call inside of a loop, and PDSG and DECKER should therefore see similar slowdowns from indirect calls. We disable tracing and ensue querying for this study and see only $\sim 3\%$ overhead from PDSG over baseline SPEC CPU 2017. This is partly expected but also remarkable. It demonstrates the heavy penalty for instrumenting inside of loops and strongly suggests that the PDSG profile data could be leveraged to avoid instrumenting in particularly hot loop sections.

4.4.3 Rectification and Prediction

We capture the percent of callgraph edges that have rectification instrumentation, and we record at runtime how frequently these edges are traversed. The first metric gives a sense of how many RPs are actually needed to ensure sound behavior at runtime. “Lower is better” does not necessarily hold for this metric. A high count of RPs could be due to a benchmark’s particular callgraph structure and how it is exercised by its training inputs (which ultimately determines the predicted sets and thus the RPs). Nevertheless, a low count of RPs implies that there are fewer “gates” through which an attack must pass if it is attempting to load functions outside of the predicted set.

The second metric is a useful way to report the prediction accuracy. When execution reaches an RP (i.e. when it traverses a callgraph edge with rectification instrumentation), it is because execution is flowing outside of the predicted set of functions – a misprediction. In the current implementation of PDSG, the rectification behavior is to map all remaining functions in that SCG. Therefore, whenever execution reaches an RP, the misprediction is

handled once (and only once) for that SCG. The misprediction rate is thus the percentage of all rectification events over all prediction events.

Table 4.2 reports the static callgraph properties for the SPEC benchmarks. #Nodes is the number of functions and is reported for completeness. The percentage of edges that are instrumented is shown in the rightmost column, which is simply $100 * \#RPs / \#Edges$. All but two benchmarks have 10% or less of their edges instrumented with RPs. `lbm` has 13.3%, and `x264` has 44.8%. Perhaps the most interesting case is `mcf`, which has 0% of its edges instrumented. Upon inspection, the application is so simple that the complements to the PSCGs within the SCGs are empty sets. That is, the predicted subcallgraphs are equivalent to the subcallgraphs, and thus there is no need for rectification.

Table 4.3 reports the frequency of the rectification events vs. the prediction events. The rightmost column reports this value as a percent. On average, PDSG must rectify 3.8% of all predictions that happen for SPEC CPU 2017. The worst case is `leela`, which has only 4 predictions, 1 of which triggers rectification.

Table 4.2: SPEC CPU 2017 static callgraph properties. The percent of edges that are instrumented with RPs is shown in the rightmost column.

Application	#Nodes	#Edges	#RPs	% E-Instr
perlbench	2112	572195	1971	0.3
gcc	10399	2261256	24523	1.1
mcf	17	46	0	0
namd	104	196	18	9.2
parest	12017	72832	2678	3.7
povray	1309	73023	2250	3.1
lbm	6	15	2	13.3
omnetpp	5684	33480	1632	4.9
xalancbmk	12237	113962	3743	3.3
x264	406	869	389	44.8
blender	31145	682535	985	0.1
deepsjeng	56	349	18	5.2
magick	1530	156511	980	0.6
leela	195	1337	18	1.3
nab	121	774	58	7.5
xz	227	449	8	1.8

Table 4.3: SPEC CPU 2017 prediction and rectification occurrences. The rightmost column is the percentage of predictions that require rectification.

Application	#Predicts	#Rectifies	% Rectifies
perlbench	175	7	4
gcc	449	6	1.3
mcf	5	0	0
namd	19	0	0
parest	777	17	2.2
povray	43	2	4.7
lbm	3	0	0
omnetpp	513	1	0.2
xalancbmk	342	6	1.8
x264	60	4	6.7
blender	8687	17	0.2
deepsjeng	4	0	0
imagick	26	2	7.7
leela	4	1	25
nab	95	2	2.1
xz	48	2	4.2

4.4.4 Datalog

The PDSG compiler pass outputs a Datalog program which includes all *head*, *tail*, *next*, *leaf*, and *belong* facts, along with the derived relations for *last* and *ensue*, and, crucially, a single query as its final line: $ensue(A, B)?$. We execute this Datalog program offline. Datalog reads all of the facts, derives *last* and *ensue*, and returns the result of the final query, which is a list of all *ensue* relations.

The results are shown in Table 4.4. The column headers mean the following: Time is the length of time for Datalog to produce all *ensue* relations for the application, rounded to the nearest minute; Size is the filesize (reported by `du -sm`) of the output from Datalog of all *ensue* relations; #Facts is the number of facts in the Datalog program (input); #Ensue is the number of *ensue* relations Datalog reports (output).

In half of the benchmarks, Datalog completes in less than 1 minute. The most striking case is `gcc`, which takes nearly 8 hours to complete. Though all of this processing is offline, this result shows that calculating *ensue* relations for highly complex callgraphs can

Table 4.4: Datalog results for SPEC CPU 2017.

Application	Time	Size	#Facts	#Ensnue
perlbench	16m	8MB	270764	392007
gcc	478m	232MB	1318536	11468987
mcf	0m	1MB	123	46
namd	0m	1MB	1357	711
parest	43m	4MB	152680	172228
povray	2m	4MB	62619	175976
lbm	0m	1MB	73	26
omnetpp	2m	2MB	56165	108442
xalancbmk	33m	13MB	142410	667789
x264	0m	1MB	9145	26541
blender	76m	11MB	227995	507865
deepsjeng	0m	1MB	1336	1644
imagick	9m	8MB	95182	414880
leela	0m	1MB	2164	3210
nab	0m	1MB	1613	2419
xz	0m	1MB	1674	1207

be compute-intensive. Furthermore, the Size result (output filesize) matters for runtime, because it must be loaded into memory as a fast data structure and queried during execution. In all cases but `gcc`, the data size is reasonable. The benchmark with the most Datalog facts after `gcc` is `perlbench`. Though `gcc` has $\sim 5x$ more facts than `perlbench`, its output (in terms of filesize, which is correlated with the number of ensue relations), is $\sim 30x$, and it takes $\sim 30x$ longer to run, as well. In other words, the relative complexity of `gcc` is exceptionally high compared with the other benchmarks, as seen in its ensue count, and this drives up its computational costs for Datalog.

4.5 Conclusion

This work presents a new framework to advance whole-application debloating. We describe a novel prediction scheme that takes advantage of program structure. To handle mis-predictions, we introduce a statically verifiable checking mechanism based on valid call sequences. We show how and where to invoke predictions within the callgraph, and how to enable the checking mechanism in a lightweight manner. The resulting programs that

use this framework maintain full feature support, and all transformations are sound. We improve attack surface reduction beyond state-of-the-art (82.5% total gadget reduction on SPEC CPU 2017) with reasonable runtime overhead (10.9%) and minimal online checking (3.8% of predictions).

CHAPTER 5

CONCLUSION

In this thesis, we start by describing code reuse attacks and the traditional techniques that have been developed in response. We show how the challenging nature of these attacks has made them resilient, even in the face of state-of-the-art industrial defenses. Software debloating is seen as a promising angle to address some of these challenges, and recent research has made progress. Nevertheless, these new debloating techniques have been incomplete in some way. In general, they tend to be overly conservative, leaving too much code available for attack; or overly aggressive, performing transformations that can cause crashes or bad output in the modified program.

To address this state of affairs, this thesis adopts a mixed compiler-predictive strategy for debloating software. Our goal is to show that this strategy is key for overcoming limitations in current attack surface reduction techniques. To this end we present three key pieces of work. In the first, `BLANKIT`, we show a binary-level library debloating technique. We show that static analysis can illuminate library calls' argument values and the paths leading up to the library call invocations; and we show how to use this information within a binary instrumentation framework to predict at runtime the library functions that will execute. In our second work, `DECKER`, we focus on source-level application code. We show how static analysis techniques can be used to identify and instrument specific program points so that a runtime can enable and disable related code pages without degrading performance or sacrificing program features or soundness. In the third work, `PDSG`, we combine insights from the previous frameworks to improve whole-application debloating. We adopt a source-level strategy with instrumentation points similar to `DECKER`, and we use ML-based prediction to narrow the attack surface like in `BLANKIT`. Furthermore, we show how to leverage static call sequence information to perform lightweight checks on mispredictions at runtime.

This thesis attempts to show how this approach leads to effective software debloating. In particular, the reductions at the binary level for libraries are striking (upwards of 97%), without sacrificing soundness. The reductions at the source level for application code are competitive with other approaches (73% on SPEC CPU 2017), without reducing the feature set or compromising soundness. Lastly, adding predictive support for whole-application debloating improves attack surface reduction further (83% on SPEC CPU 2017); and by continuing to embrace a mixed compiler-predictive approach, any mispredictions can be checked against certain program properties derived from program analysis and exposed to the runtime.

5.1 Future Work

There are several open questions that we wish to explore next. We discuss four – one regarding each of hardware, performance, security, and prediction. In terms of hardware, there are several examples today of hardware-based security modules whose main goal is to serve as a trusted platform for more secure software. A few examples include Intel’s Software Guard Extensions (SGX) and Trusted Domain Extensions (TDX), or AMD’s Secure Encrypted Virtualization technologies (SEV, SEV-ES, SEV-SNP). These trusted execution environments work in tandem with system software to enable different types of trust boundaries, and these are enforced through some type of encryption. These technologies demonstrate that on-the-fly encryption and decryption of memory is actually achievable in real-world cases. A comparable area is hardware-enabled compression/decompression. In other words, it is reasonable to expect that the kinds of dynamic debloating techniques in this thesis could be supported by or incorporated into hardware in a similar fashion. A bit today can be used to mark a page as encrypted/decrypted. With hardware support for debloating, we can imagine another bit being used to mark a page as active/inactive. We feel this question of how to provide hardware support for debloating opens several interesting research directions.

In terms of performance, we observe that PDSG’s overhead is less than BLANKIT’s, but we expect that it will need to be $\sim 5\%$ (similar to DECKER’s) for it to be considered viable for real-world adoption. One strategy in particular that we wish to explore involves a new concept that we term “may-must execute sets.” This is based on the previously mentioned idea of “may-use code,” but we formalize it as an anticipability problem for callsites. That is, given some point p within a block in the callgraph, we can statically calculate to some call depth d the anticipatable function callsites A that will execute. This is more precise than our prior concept of “static reachability” (i.e. *all* functions that can be statically reached from some point p). This concept will allow us to improve performance by issuing fewer page-mapping calls into our runtime. For example, wherever we currently issue a sequence of single-function mapping calls, we can instead map all must-execute functions to some depth d (handling any may-execute cases and subsequent must-execute cases as appropriate). Though this may have a slightly negative effect on security, by construction this technique only maps functions that are known to execute (and therefore would have exposed their gadgets anyway).

A third open question is how to “sink” instrumentation inside of loops. When mapping all reachable functions at the top of a loop, the interprocedural set of functions can be quite large, which can expose a substantial amount of attack surface. However, from a static perspective, the point at which to sink the instrumentation is non-obvious. If it is placed inside a “tight” loop, the instrumentation will execute repeatedly (and hurt performance). With PDSG’s introduction of a profiling phase for gathering training data, there is an opportunity now to use that phase to collect information about loop timings, as well. For example, if the profiling data from PDSG suggests a tight loop exists with a substantial code surface, then the prediction scheme seems already to be quite effective. If, however, profiling data suggests the outermost loops have a relatively low frequency, then predictions could be sunk deeper into the callgraph. This could even have a positive knock-on effect on the prediction accuracy, because the models would be trained on a more focused,

smaller subset of the loop.

In terms of prediction, we would like to take advantage of stronger ML models. The decision tree has several characteristics that make it appealing. For example, it is small and fast; it can be expressed as blocks of if-else statements that can be directly inlined into the C code. On the other hand, more accurate models may reduce the number of mispredictions, though at the cost of runtime overhead. It is unclear, however, whether *varying* the ML model can give a best of both worlds. For example, for predictions that must occur inside of loops, a decision tree may still be necessary to achieve reasonable performance; but for predicting large function sets at interprocedurally outermost loops, a more complex classifier may be preferable. The intuition is that it is worth paying a higher runtime cost for a better prediction if it only happens once (i.e. outside of any loops) and if that prediction is for a large portion of the callgraph.

In conclusion, the thrust of this thesis, namely that the compiler and ML-based predictions can be complementary and used in tandem to improve upon state-of-the-art attack surface reduction, still offers substantial runway for future research questions.

REFERENCES

- [1] S. Andersen and V. Abella, *Data execution prevention: Changes to functionality in microsoft windows xp service pack 2, part 3: Memory protection technologies*, <http://technet.microsoft.com/en-us/library/bb457155.aspx>, Accessed: 2018 May 7, 2004.
- [2] Nergal, *The advanced return-into-lib(c) exploits: Pax case study. phrack magazine*, <http://phrack.org/issues/58/4.html>, Accessed: 2018 Apr 24, 2001.
- [3] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning, “On the expressiveness of return-into-libc attacks,” in *Recent Advances in Intrusion Detection*, R. Sommer, D. Balzarotti, and G. Maier, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 121–141, ISBN: 978-3-642-23644-0.
- [4] *Pax address space layout randomization*, <https://pax.grsecurity.net/docs/aslr.txt>, Accessed: 2018 May 7, 2003.
- [5] T. Durden, *Bypassing pax aslr protection*, <http://phrack.org/issues/59/9.html>, Accessed: 2021 Sept 14, 2002.
- [6] D. Evtuyshkin, D. V. Ponomarev, and N. B. Abu-Ghazaleh, “Jump over ASLR: attacking branch predictors to bypass ASLR,” in *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*, IEEE Computer Society, 2016, 40:1–40:13.
- [7] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh, “On the effectiveness of address-space randomization,” in *Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS 2004, Washington, DC, USA, October 25-29, 2004*, V. Atluri, B. Pfitzmann, and P. D. McDaniel, Eds., ACM, 2004, pp. 298–307.
- [8] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, “Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization,” in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, ser. SP ’13, Washington, DC, USA: IEEE Computer Society, 2013, pp. 574–588, ISBN: 978-0-7695-4977-4.
- [9] H. Shacham, “The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86),” in *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*, P. Ning, S. D. C. di Vimercati, and P. F. Syverson, Eds., ACM, 2007, pp. 552–561, ISBN: 978-1-59593-703-2.

- [10] T. K. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, “Jump-oriented programming: A new class of code-reuse attack,” in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ASIACCS 2011, Hong Kong, China, March 22-24, 2011*, B. S. N. Cheung, L. C. K. Hui, R. S. Sandhu, and D. S. Wong, Eds., ACM, 2011, pp. 30–40.
- [11] S. Checkoway, L. Davi, A. Dmitrienko, A. Sadeghi, H. Shacham, and M. Winandy, “Return-oriented programming without returns,” in *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*, E. Al-Shaer, A. D. Keromytis, and V. Shmatikov, Eds., ACM, 2010, pp. 559–572.
- [12] A. Sadeghi, S. Niksefat, and M. Rostamipour, “Pure-call oriented programming (PCOP): chaining the gadgets using call instructions,” *J. Comput. Virol. Hacking Tech.*, vol. 14, no. 2, pp. 139–156, 2018.
- [13] S. Schirra, *Ropper*, <https://github.com/sashes/ropper>, Accessed: 2021 Sept 10, 2021.
- [14] *Nginx*, <https://nginx.org/>, Accessed: 2019 June 10, 2019.
- [15] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, “Control-flow bending: On the effectiveness of control-flow integrity,” in *Proceedings of the 24th USENIX Conference on Security Symposium*, ser. SEC’15, Washington, D.C.: USENIX Association, 2015, pp. 161–176, ISBN: 978-1-931971-232.
- [16] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, “Control-flow integrity,” in *Proceedings of the 12th ACM Conference on Computer and Communications Security, CCS 2005, Alexandria, VA, USA, November 7-11, 2005*, V. Atluri, C. A. Meadows, and A. Juels, Eds., ACM, 2005, pp. 340–353.
- [17] C. Zhang *et al.*, “Practical control flow integrity and randomization for binary executables,” in *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, IEEE Computer Society, 2013, pp. 559–573.
- [18] T. K. Bletsch, X. Jiang, and V. W. Freeh, “Mitigating code-reuse attacks with control-flow locking,” in *Twenty-Seventh Annual Computer Security Applications Conference, ACSAC 2011, Orlando, FL, USA, 5-9 December 2011*, R. H. Zakon, J. P. McDermott, and M. E. Locasto, Eds., ACM, 2011, pp. 353–362.
- [19] J. Criswell, N. Dautenhahn, and V. S. Adve, “Kcofi: Complete control-flow integrity for commodity operating system kernels,” in *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, IEEE Computer Society, 2014, pp. 292–307.

- [20] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz, “Opaque control-flow integrity,” in *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, The Internet Society, 2015.
- [21] H. Hu *et al.*, “Enforcing unique code target property for control-flow integrity,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’18, Toronto, Canada: ACM, 2018, pp. 1470–1486, ISBN: 978-1-4503-5693-0.
- [22] X. Ge, W. Cui, and T. Jaeger, “GRIFFIN: guarding control flows using intel processor trace,” in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi’an, China, April 8-12, 2017*, Y. Chen, O. Temam, and J. Carter, Eds., ACM, 2017, pp. 585–598.
- [23] 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)*, Feb. 2017, pp. 529–540.
- [24] M. Zhang and R. Sekar, “Control flow integrity for cots binaries,” in *Proceedings of the 22Nd USENIX Conference on Security*, ser. SEC’13, Washington, D.C.: USENIX Association, 2013, pp. 337–352, ISBN: 978-1-931971-03-4.
- [25] V. van der Veen *et al.*, “A tough call: Mitigating advanced code-reuse attacks at the binary level,” in *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, IEEE Computer Society, 2016, pp. 934–953.
- [26] B. Niu and G. Tan, “Per-input control-flow integrity,” in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, I. Ray, N. Li, and C. Kruegel, Eds., ACM, 2015, pp. 914–926.
- [27] B. Niu and G. Tan, “Modular control-flow integrity,” in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’14, Edinburgh, United Kingdom - June 09 - 11, 2014*, M. F. P. O’Boyle and K. Pingali, Eds., ACM, 2014, pp. 577–587.
- [28] I. Evans *et al.*, “Control jujutsu: On the weaknesses of fine-grained control flow integrity,” in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’15, Denver, Colorado, USA: ACM, 2015, pp. 901–913, ISBN: 978-1-4503-3832-5.
- [29] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, “Cets: Compiler enforced temporal safety for c,” in *Proceedings of the 2010 International Symposium*

on Memory Management, ser. ISMM '10, Toronto, Ontario, Canada: ACM, 2010, pp. 31–40, ISBN: 978-1-4503-0054-4.

- [30] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, “Code-pointer integrity,” in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14, Broomfield, CO: USENIX Association, 2014, pp. 147–163, ISBN: 978-1-931971-16-4.
- [31] X. Zhuang, T. Zhang, and S. Pande, “Using branch correlation to identify infeasible paths for anomaly detection,” in *39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-39 2006), 9-13 December 2006, Orlando, Florida, USA*, IEEE Computer Society, 2006, pp. 113–122.
- [32] S. Chen, J. Xu, and E. C. Sezer, “Non-control-data attacks are realistic threats,” in *Proceedings of the 14th USENIX Security Symposium, Baltimore, MD, USA, July 31 - August 5, 2005*, P. D. McDaniel, Ed., USENIX Association, 2005.
- [33] W. Tounsi and H. Rais, “A survey on technical threat intelligence in the age of sophisticated cyber attacks,” *Computers & Security*, vol. 72, pp. 212–233, 2018.
- [34] B. Brizendine and A. Babcock, *Pre-built jop chains with the jop rocket: Bypassing dep without rop*, <https://i.blackhat.com/asia-21/Thursday-Handouts/as-21-Brizendine-Babcock-Prebuilt-Jop-Chains-With-The-Jop-Rocket-wp.pdf>, Accessed: 2022 Jun 06, 2021.
- [35] Z. Yunha, *Bypass control flow guard comprehensively*, <https://www.blackhat.com/docs/us-15/materials/us-15-Zhang-Bypass-Control-Flow-Guard-Comprehensively-wp.pdf>, Accessed: 2022 Jun 06, 2015.
- [36] M. Schenk, *Bypassing control flow guard in windows 10*, <https://improsec.com/tech-blog/bypassing-control-flow-guard-in-windows-10>, Accessed: 2022 Jun 06, 2017.
- [37] S. Thomas, *Object oriented exploitation: New techniques in windows mitigation bypass*, https://www.slideshare.net/_s_n_t/object-oriented-exploitation-new-techniques-in-windows-mitigation-bypass, Accessed: 2022 Jun 06, 2016.
- [38] B. Sun, J. Liu, and C. Xu, *How to survive the hardware-assisted control-flow integrity enforcement*, <https://i.blackhat.com/asia-19/Thu-March-28/bh-asia-Sun-How-to-Survive-the-Hardware-Assisted-Control-Flow-Integrity-Enforcement.pdf>, Accessed: 2022 Jun 06, 2019.
- [39] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A. 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, SP 2015, San Jose, CA, USA, May 17-21, 2015*, IEEE Computer Society, 2015, pp. 745–762.
- [40] P. K. Manadhata and J. M. Wing, “An attack surface metric,” *IEEE Trans. Software Eng.*, vol. 37, no. 3, pp. 371–386, 2011.
- [41] A. Quach, A. Prakash, and L. Yan, “Debloating software through piece-wise compilation and loading,” in *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD: USENIX Association, 2018, pp. 869–886, ISBN: 978-1-931971-46-1.
- [42] K. Heo, W. Lee, P. Pashakhanloo, and M. Naik, “Effective program debloating via reinforcement learning,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’18, Toronto, Canada: ACM, 2018, pp. 380–394, ISBN: 978-1-4503-5693-0.
- [43] C. Qian, H. Hu, M. Alharthi, S. P. H. Chung, T. Kim, and W. Lee, “RAZOR: A framework for post-deployment software debloating,” in *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, N. Heninger and P. Traynor, Eds., USENIX Association, 2019, pp. 1733–1750.
- [44] W. Landi and B. G. Ryder, “A safe approximate algorithm for interprocedural aliasing,” in *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, ser. PLDI ’92, San Francisco, California, USA: Association for Computing Machinery, 1992, pp. 235–248, ISBN: 0897914759.
- [45] G. Ramalingam, “The undecidability of aliasing,” *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 5, pp. 1467–1471, Sep. 1994.
- [46] L. O. Andersen, *Program analysis and specialization for the c programming language*, PhD thesis, DIKU, University of Copenhagen, May 1994.
- [47] *Android ndk*, <https://developer.android.com/ndk/guides/cpp-support.html>, Accessed: 2017 Feb 5, 2017.
- [48] *Tensorflow for c*, https://www.tensorflow.org/install/install_c, Accessed: 2017 Feb 5, 2017.
- [49] *Caffe*, <http://caffe.berkeleyvision.org/>, Accessed: 2017 Feb 5, 2017.
- [50] *Flask*, <http://flask.pocoo.org/>, Accessed: 2018 Apr 24, 2018.
- [51] *Node.js*, <https://nodejs.org/en/>, Accessed: 2018 Apr 24, 2018.

- [52] A. Quach, A. Prakash, and L. Yan, “Debloating software through piece-wise compilation and loading,” in *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD: USENIX Association, 2018, pp. 869–886, ISBN: 978-1-931971-46-1.
- [53] *Glibc*, <https://www.gnu.org/software/libc/>, Accessed: 2017 Feb 5, 2017.
- [54] *Cve details*, https://www.cvedetails.com/vulnerability-list/vendor_id-72/product_id-767/GNU-Glibc.html, Accessed: 2017 Feb 5, 2017.
- [55] S. Debray and W. Evans, “Profile-guided code compression,” in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, ser. PLDI ’02, Berlin, Germany: ACM, 2002, pp. 95–105, ISBN: 1-58113-463-0.
- [56] B. D. Sutter, B. D. Bus, and K. D. Bosschere, “Link-time binary rewriting techniques for program compaction,” *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 5, pp. 882–945, 2005.
- [57] B. D. Sutter, L. V. Put, D. Chanet, B. D. Bus, and K. D. Bosschere, “Link-time compaction and optimization of ARM executables,” *ACM Trans. Embedded Comput. Syst.*, vol. 6, no. 1, p. 5, 2007.
- [58] V. Le, C. Sun, and Z. Su, “Randomized stress-testing of link-time optimizers,” in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015, Baltimore, MD, USA: ACM, 2015, pp. 327–337, ISBN: 978-1-4503-3620-8.
- [59] K. Heo, W. Lee, P. Pashakhanloo, and M. Naik, “Effective program debloating via reinforcement learning,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, 2018, pp. 380–394.
- [60] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, and D. Song, “Poster : Getting the point(er): On the feasibility of attacks on code-pointer integrity,” 2015.
- [61] I. Evans *et al.*, “Missing the point(er): On the effectiveness of code pointer integrity,” in *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, IEEE Computer Society, 2015, pp. 781–796, ISBN: 978-1-4673-6949-7.
- [62] *Musl*, <https://www.musl-libc.org/>, Accessed: 2019 June 15, 2019.
- [63] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi, “Pin-pointing representative portions of large intel #174; itanium #174; programs with

dynamic instrumentation,” in *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on*, Dec. 2004, pp. 81–92.

- [64] I. Zhang, T. Denniston, Y. Baskakov, and A. Garthwaite, “Optimizing vm checkpointing for restore performance in vmware esxi,” in *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC’13, San Jose, CA: USENIX Association, 2013, pp. 1–12.
- [65] J. Xu, D. Mu, P. Chen, X. Xing, P. Wang, and P. Liu, “CREDAL: towards locating a memory corruption vulnerability with your core dump,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds., ACM, 2016, pp. 529–540, ISBN: 978-1-4503-4139-4.
- [66] N. Nethercote and J. Seward, “Valgrind: A framework for heavyweight dynamic binary instrumentation,” in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’07, San Diego, California, USA: ACM, 2007, pp. 89–100, ISBN: 978-1-59593-633-2.
- [67] J. Seward and N. Nethercote, “Using valgrind to detect undefined value errors with bit-precision,” in *Proceedings of the 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA*, USENIX, 2005, pp. 17–30.
- [68] T. Ball and J. R. Larus, “Efficient path profiling,” in *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 29, Paris, France: IEEE Computer Society, 1996, pp. 46–57, ISBN: 0818676418.
- [69] F. Pedregosa *et al.*, “Scikit-learn: Machine learning in Python,” *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.
- [70] *Ropgadget v5.4*, <https://github.com/JonathanSalwan/ROPgadget>, Accessed: 2018 May 7, 2018.
- [71] A. Follner, A. Bartel, and E. Bodden, “Analyzing the gadgets - towards a metric to measure gadget quality,” in *Engineering Secure Software and Systems - 8th International Symposium, ESSoS 2016, London, UK, April 6-8, 2016. Proceedings*, J. Caballero, E. Bodden, and E. Athanasopoulos, Eds., ser. Lecture Notes in Computer Science, vol. 9639, Springer, 2016, pp. 155–172.
- [72] M. D. Brown and S. Pande, “Is less really more? towards better metrics for measuring security improvements realized through software debloating,” in *12th USENIX Workshop on Cyber Security Experimentation and Test, CSET 2019, Santa Clara, CA, USA, August 12, 2019.*, R. Jansen and P. A. H. Peterson, Eds., USENIX Association, 2019.

- [73] K. Heo, W. Lee, P. Pashakhanloo, and M. Naik, “Effective program debloating via reinforcement learning,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’18, Toronto, Canada: ACM, 2018, pp. 380–394, ISBN: 978-1-4503-5693-0.
- [74] H. Sharif, M. Abubakar, A. Gehani, and F. Zaffar, “TRIMMER: application specialization for code debloating,” in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, M. Huchard, C. Kästner, and G. Fraser, Eds., ACM, 2018, pp. 329–339.
- [75] *Openssh*, <https://www.openssh.com/>, Accessed: 2019 June 11, 2019.
- [76] *Wrk*, <https://github.com/wg/wrk>, Accessed: 2019 June 10, 2019.
- [77] *Wikipedia*, https://en.wikipedia.org/wiki/Main_Page, Accessed: 2019 June 10, 2019.
- [78] *Norvig*, <https://norvig.com/big.txt>, Accessed: 2019 June 12, 2019.
- [79] *Sshdcve*, <https://nvd.nist.gov/vuln/detail/CVE-2015-7547>, Accessed: 2019 June 12, 2019.
- [80] N. Mitchell and G. Sevitsky, “The causes of bloat, the limits of health,” in *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, R. P. Gabriel, D. F. Bacon, C. V. Lopes, and G. L. S. Jr., Eds., ACM, 2007, pp. 245–260.
- [81] N. Mitchell, E. Schonberg, and G. Sevitsky, “Making sense of large heaps,” in *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings*, S. Drossopoulou, Ed., ser. Lecture Notes in Computer Science, vol. 5653, Springer, 2009, pp. 77–97.
- [82] G. (Xu, N. Mitchell, M. Arnold, A. Rountev, and G. Sevitsky, “Software bloat analysis: Finding, removing, and preventing performance problems in modern large-scale object-oriented applications,” in *Proceedings of the Workshop on Future of Software Engineering Research, FoSER 2010, at the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*, G. Roman and K. J. Sullivan, Eds., ACM, 2010, pp. 421–426.
- [83] N. Mitchell, E. Schonberg, and G. Sevitsky, “Four trends leading to java runtime bloat,” *IEEE Software*, vol. 27, no. 1, pp. 56–63, 2010.

- [84] G. (Xu, N. Mitchell, M. Arnold, A. Rountev, E. Schonberg, and G. Sevitsky, “Finding low-utility data structures,” in *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, B. G. Zorn and A. Aiken, Eds., ACM, 2010, pp. 174–186.
- [85] G. Xu and A. Rountev, “Detecting inefficiently-used containers to avoid bloat,” in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’10, Toronto, Ontario, Canada: ACM, 2010, pp. 160–173, ISBN: 978-1-4503-0019-3.
- [86] G. (Xu, “Finding reusable data structures,” in *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, G. T. Leavens and M. B. Dwyer, Eds., ACM, 2012, pp. 1017–1034.
- [87] Á. Beszédes, R. Ferenc, T. Gyimóthy, A. Dolenc, and K. Karsisto, “Survey of code-size reduction methods,” *ACM Comput. Surv.*, vol. 35, no. 3, pp. 223–267, Sep. 2003.
- [88] S. K. Debray, W. Evans, R. Muth, and B. De Sutter, “Compiler techniques for code compaction,” *ACM Trans. Program. Lang. Syst.*, vol. 22, no. 2, pp. 378–415, Mar. 2000.
- [89] R. Muth, S. K. Debray, S. Watterson, and K. De Bosschere, “Alto: A link-time optimizer for the compaq alpha,” *Softw. Pract. Exper.*, vol. 31, no. 1, pp. 67–101, Jan. 2001.
- [90] M. Franz and T. Kistler, “Slim binaries,” *Commun. ACM*, vol. 40, no. 12, pp. 87–94, Dec. 1997.
- [91] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, “Addresssanitizer: A fast address sanity checker,” in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC’12, Boston, MA: USENIX Association, 2012, p. 28.
- [92] G. J. Duck and R. H. C. Yap, “Effectivesan: Type and memory error detection using dynamically typed c/c++,” *SIGPLAN Not.*, vol. 53, no. 4, pp. 181–195, Jun. 2018.
- [93] *Coreutils - gnu core utilities*, <https://www.gnu.org/software/coreutils/>, Accessed: 2021 Jul 27, 2021.
- [94] C. Porter, G. Mururu, P. Barua, and S. Pande, “Blankit library debloating: Getting what you want instead of cutting what you don’t,” in *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Im-*

plementation, *PLDI 2020, London, UK, June 15-20, 2020*, A. F. Donaldson and E. Torlak, Eds., ACM, 2020, pp. 164–180.

- [95] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, “Code-pointer integrity,” in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’14, Broomfield, CO: USENIX Association, 2014, pp. 147–163, ISBN: 978-1-931971-16-4.
- [96] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, and D. Song, “Poster : Getting the point(er): On the feasibility of attacks on code-pointer integrity,” 2015.
- [97] C. Lattner and V. S. Adve, “LLVM: A compilation framework for lifelong program analysis & transformation,” in *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, IEEE Computer Society, 2004, pp. 75–88.
- [98] S. P. E. Corporation, *Spec cpu 2017*, <https://www.spec.org/cpu2017/>, Accessed: 2021 Jul 27, 2021.
- [99] *Callgraph.h*, <https://github.com/llvm/llvm-project/blob/llvmorg-11.0.0/llvm/include/llvm/Analysis/CallGraph.h>, Accessed: 2022 Nov 9, 2020.
- [100] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, “XFI: software guards for system address spaces,” in *7th Symposium on Operating Systems Design and Implementation (OSDI ’06), November 6-8, Seattle, WA, USA*, B. N. Bershad and J. C. Mogul, Eds., USENIX Association, 2006, pp. 75–88.
- [101] V. Sundaresan *et al.*, “Practical virtual method call resolution for java,” in *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA ’00, Minneapolis, Minnesota, USA: Association for Computing Machinery, 2000, pp. 264–280, ISBN: 158113200X.
- [102] cxreet, *Razor*, <https://github.com/cxreet/razor>, Accessed: 2021 Sept 30, 2021.
- [103] W3Techs, *Usage statistics of web servers*, https://w3techs.com/technologies/overview/web_server, Accessed: 2021 Oct 10, 2021.
- [104] M. D. Brown, M. Pruett, R. Bigelow, G. Mururu, and S. Pande, “Not so fast: Understanding and mitigating negative impacts of compiler optimizations on code reuse gadget sets,” *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, pp. 1–30, 2021.
- [105] B. Brizendine and A. Babcock, *Shellcodeless jop: Advanced code-reuse attacks with jump-oriented programming*, <https://files.athack.com/files/2021-12/AtHack->

Advanced % 20Code - Reuse % 20Attacks % 20Whitepaper_0 . pdf ? VersionId = D . HV08jcDUeclPb4FcliC2jpOuK07GR8, Accessed: 2022 Jun 23, 2021.

- [106] R. Ding, C. Qian, C. Song, W. Harris, T. Kim, and W. Lee, “Efficient protection of path-sensitive control security,” in *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, E. Kirda and T. Ristenpart, Eds., USENIX Association, 2017, pp. 131–148.
- [107] M. Khandaker, A. Naser, W. Liu, Z. Wang, Y. Zhou, and Y. Cheng, “Adaptive call-site sensitive control flow integrity,” in *IEEE European Symposium on Security and Privacy, EuroS&P 2019, Stockholm, Sweden, June 17-19, 2019*, IEEE, 2019, pp. 95–110.
- [108] X. Ge, W. Cui, and T. Jaeger, “GRIFFIN: guarding control flows using intel processor trace,” in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi’an, China, April 8-12, 2017*, Y. Chen, O. Temam, and J. Carter, Eds., ACM, 2017, pp. 585–598.
- [109] V. van der Veen *et al.*, “Practical context-sensitive CFI,” in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, I. Ray, N. Li, and C. Kruegel, Eds., ACM, 2015, pp. 927–940.
- [110] Y. Li, M. Wang, C. Zhang, X. Chen, S. Yang, and Y. Liu, “Finding cracks in shields: On the security of control flow integrity mechanisms,” in *CCS ’20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, J. Ligatti, X. Ou, J. Katz, and G. Vigna, Eds., ACM, 2020, pp. 1821–1835.
- [111] M. Khandaker, W. Liu, A. Naser, Z. Wang, and J. Yang, “Origin-sensitive control flow integrity,” in *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, N. Heninger and P. Traynor, Eds., USENIX Association, 2019, pp. 195–211.
- [112] J. Tang, “Exploring control flow guard in windows 10,” Trend Micro, Tech. Rep., 2015.
- [113] Microsoft, *Control flow guard for platform security*, <https://learn.microsoft.com/en-us/windows/win32/secbp/control-flow-guard>, Accessed: 2022 Oct 21, 2022.
- [114] D. Wei, Lywang, and FlowerCode, *Return flow guard*, <https://xlab.tencent.com/en/2016/11/02/return-flow-guard/>, Accessed: 2022 Oct 21, 2016.

- [115] *Intel control-flow enforcement technology (cet)*, <https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf>, 2017.
- [116] J. Corbet, *Indirect branch tracking for intel cpus*, <https://lwn.net/Articles/889475/>, Accessed: 2022 Oct 21, 2022.
- [117] C. Qian, H. Koo, C. Oh, T. Kim, and W. Lee, “Slimium: Debloating the chromium browser with feature subsetting,” in *CCS ’20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, J. Ligatti, X. Ou, J. Katz, and G. Vigna, Eds., ACM, 2020, pp. 461–476.
- [118] H. Koo, S. Ghavamnia, and M. Polychronakis, “Configuration-driven software debloating,” in *Proceedings of the 12th European Workshop on Systems Security, EuroSec@EuroSys 2019, Dresden, Germany, March 25, 2019*, ACM, 2019, 9:1–9:6.
- [119] C. Soto-Valero, T. Durieux, N. Harrand, and B. Baudry, “Coverage-based debloating for java bytecode,” *ACM Trans. Softw. Eng. Methodol.*, Jun. 2022, Just Accepted.
- [120] J. Bangert *et al.*, “Elfbac: Using the loader format for intent-level semantics and fine-grained protection,” Dartmouth University, Tech. Rep. TR2013-727, 2013.
- [121] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural computation*, vol. 9, pp. 1735–80, Dec. 1997.
- [122] Y. Chen, O. Temam, and J. Carter, Eds., *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi’an, China, April 8-12, 2017*, ACM, 2017.