



Decker: Attack Surface Reduction via On-Demand Code Mapping

Chris Porter
porter@gatech.edu
Georgia Institute of Technology
Atlanta, Georgia, USA

Sharjeel Khan
smkhan@gatech.edu
Georgia Institute of Technology
Atlanta, Georgia, USA

Santosh Pande
santosh.pande@cc.gatech.edu
Georgia Institute of Technology
Atlanta, Georgia, USA

ABSTRACT

Modern code reuse attacks take full advantage of bloated software. Attackers piece together short sequences of instructions in otherwise benign code to carry out malicious actions. Mitigating these reusable code snippets, known as gadgets, has become one of the prime focuses of attack surface reduction research. While some debloating techniques remove parts of software that contain such gadgets, other methods focus on making them unusable by breaking up chains of them, thereby substantially diminishing the possibility of code reuse attacks. Third-party libraries are another main focus, because they exhibit a high number of vulnerabilities, but recently, techniques have emerged that deal with whole applications. Attack surface reduction efforts have typically tried to eliminate such attacks by subsetting (debloating) the application, e.g. via user-specified inputs, configurations, or features to achieve high gadget reductions. However, such techniques suffer from the limitations of soundness, i.e. the software might crash during no-attack executions on regular inputs, or they may be conservative and leave a large amount of attack surface untackled.

In this work we present a general, whole-program attack surface reduction technique called DECKER that significantly reduces gadgets which are accessible to an attacker during an execution phase (called a deck) and has minor performance degradation. DECKER requires no user inputs and leaves all features intact. It uses static analysis to determine key function sets that should be enabled/disabled at runtime. The runtime system then enables these function sets at the specified program points during execution. On SPEC CPU 2017, our framework achieves 73.2% total gadget reduction with 5.2% average slowdown. On 10 GNU coreutils applications, it achieves 87.2% reduction and negligible slowdown. On the nginx server it achieves 80.3% reduction with 2% slowdown. We also provide a gadget chain-breaking case study, including detailed JOP gadget metrics on both Linux and Windows, and show that our framework breaks the shell-spawning chain in all cases.

CCS CONCEPTS

• Security and privacy → Software security engineering.

KEYWORDS

software debloating, program security

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASPLoS '23, March 25–29, 2023, Vancouver, BC, Canada

© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-9916-6/23/03...\$15.00

<https://doi.org/10.1145/3575693.3575734>

ACM Reference Format:

Chris Porter, Sharjeel Khan, and Santosh Pande. 2023. Decker: Attack Surface Reduction via On-Demand Code Mapping. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLoS '23)*, March 25–29, 2023, Vancouver, BC, Canada. ACM, New York, NY, USA, 15 pages. <https://doi.org/10.1145/3575693.3575734>

1 INTRODUCTION

Attack surface reduction has gained in importance lately. A recent investigation [51] showed that on average, 95% of GNU libc code [26] is never used by user applications in a typical Ubuntu Desktop installation. This bloated software has security implications. The excess code can contain bugs and is typically not maintained well. This allows it to become a landmine of vulnerabilities, or to be repurposed for malicious ends in a code reuse attack.

In this work, we aim to reduce the exposed, reusable portions of code that are available when an attacker launches a code reuse attack. We motivate our work with a simple but powerful example. It shows why attack surface reduction is important, and also why it is challenging to make a meaningful dent in the reusable code used for staging the attack.

Listing 1: An ROP gadget chain from nginx that spawns a shell with `execve`.

```
gdgt 1 addr  -> pop rax; ret;
'/ bin / sh '
gdgt 2 addr  -> pop rcx; ret;
bss addr     -> 0x00800000
gdgt 3 addr  -> mov qword ptr [rcx], rax; ret;
gdgt 4 addr  -> pop rax; ret;
0x00000000
gdgt 5 addr  -> pop rcx; ret;
bss addr + 8 -> 0x00800008
gdgt 6 addr  -> mov qword ptr [rcx], rax; ret;
gdgt 7 addr  -> pop rdi; ret;
bss addr     -> 0x00800000
gdgt 8 addr  -> pop rsi; ret;
bss addr + 8 -> 0x00800008
gdgt 9 addr  -> pop rdx; ret;
bss addr + 8 -> 0x00800008
gdgt 10 addr -> pop rax; ret;
0x0000003b
gdgt 11 addr -> syscall;
```

Listing 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 to launch an attack. 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 [55] on the `nginx` server application [44]. 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. [11] 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 in this work, 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, 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.

Defending against code reuse attacks is an area of research lately. A rich set of gadgets (ROP, JOP and COP) exist in almost every application, resulting in a number of possibilities to concoct an attack¹. The attackers are able to creatively chain the gadgets together, and current defenses are not able to handle them. Debloating or subsetting unused functionality of an application is one way to prevent the use of gadget chains, because it purges the gadgets altogether. Unfortunately, such techniques suffer from soundness issues. We define a **sound** technique as one whose program transformations preserve the original semantics, and this will be discussed in more detail later. Given the above state-of-the-art, *a critical question that begs attention is whether the attack surface can be reduced substantially enough to break such 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 paper 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.

¹Although we have made several assumptions in this simple example, more complex gadget-based attacks exist.

The remainder of our paper is laid out as follows: further background and motivation (Section 2), an overview and assumptions of our solution (3), framework details (4), evaluation (5), related work (6), and conclusion (7).

2 BACKGROUND AND MOTIVATION

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) [2]. 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 [43, 67]. Address space layout randomization (ASLR) [47] 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 [20, 23, 58, 61].

Reuse attacks have only become more sophisticated. Return-oriented programming [57], jump-oriented programming [6, 12], and call-oriented programming [53] 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.

Traditional and industry defenses have had some success, but they have also had their shortcomings (see Section 6). 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 is one class of defense that has gained in prominence lately. Piece-wise compiler [51], Chisel [27], Razor [49], and BlankIt [48] are four 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 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.

BlankIt is a binary runtime technique that dynamically loads library code before use and purges it afterward. It is capable of handling may-use code. It includes a machine learning-based predictor for minimizing the number of functions loaded at each library call-site. While it is an effective solution for library debloating, it does not perform whole application debloating, which is much more challenging. It works well in terms of predicting the set of reachable functions at runtime, but, being based on machine learning, cannot give guarantees on these sets (resulting in a low false positive rate).

We summarize and compare the characteristics of these four approaches in Table 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.*

Table 1: Broad properties of 4 state-of-the-art debloating techniques for security (Piece-wise, Chisel, Razor, and BlankIt).

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

To the best of our knowledge, there is no general technique today that (1) works on the applications as a whole instead of libraries, (2) is sound, and (3) can effectively debloat may-use code using dynamic contexts. Current techniques either tackle libraries to achieve strong attack surface reduction, or they tackle applications and compromise soundness. Furthermore, an ideal solution would not require any test cases or specification from the user; and it

would either avoid prediction or handle its security challenges gracefully. All these limitations motivate the current work.

3 OVERVIEW

3.1 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 interprocedural 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 overestimation 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.

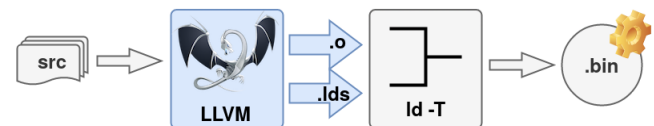


Figure 1: High-level view of the compiler part. The LLVM pass outputs a custom linker script and instrumented object file.

Figure 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 2 illustrates the runtime by way of an example. It is drawn directly from the GNU `coreutils`' `date` program [14], 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 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.

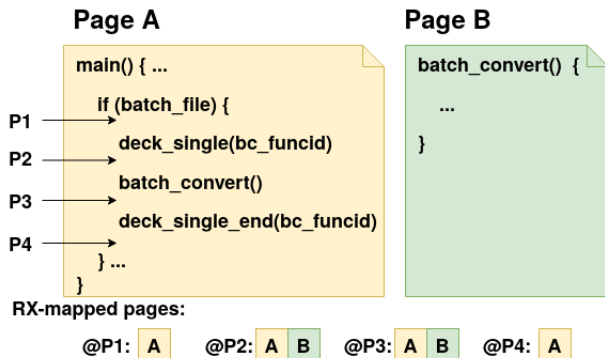


Figure 2: High-level runtime example from GNU `coreutils`' `date`. The set of RX-mapped pages increases at P2 to include a called function.

3.2 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 instrumentation 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 Section 4.1.1.

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()`.

3.3 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 [48]), and which can be implemented with in-process isolation [32], hardware segmentation or software fault isolation [33]. 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 6). DECKER does *not* tackle this problem, which we consider to be orthogonal to the core problem of this work 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 4, 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.

4 FRAMEWORK

4.1 Compiler Component

DECKER’s compiler component is an LLVM [34] 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 this work 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.

4.1.1 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 Section 4.2).

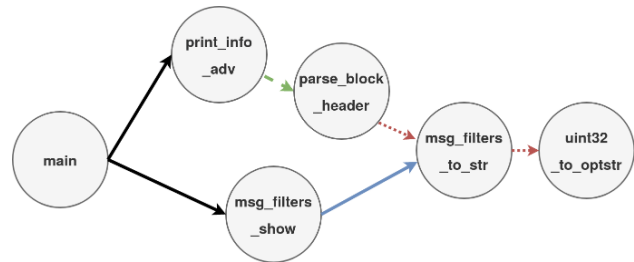


Figure 3: Simplified callgraph from the xz data compression application. This illustrates 4 types of edges, each of which requires different handling by the instrumentation pass.

Figure 3 depicts a sub-callgraph from SPEC CPU 2017’s xz, a data compression application [15]. 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 `{parse_block_header, msg_filters_to_str, uint32_to_optstr}`, and the set of non-encompassed functions is `{main, msg_filters_show, 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 1. 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.

Algorithm 1 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)
    end for
  end function

```

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 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.

4.1.2 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" [10]. As currently implemented, however, LLVM creates two "external" nodes in a module's callgraph. The first node represents 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* [10]. 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 Section 5.1.2), 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 [1, 21]. Other function pointer analysis works (such as for Java virtual methods [64]) 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.

4.1.3 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, the deck sets for this callgraph $Dk.*$ are:

```

Dk.S1 = {print_info_ado}
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_adu}
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 2. 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 2 Pseudocode for creating disjoint sets for DECKER’s custom linker script.

```
function CREATE_DISJOINT_SETS(deck_sets)
  disjoint_sets ← ∅
  while !deck_sets.EMPTY() do
    A ← deck_sets[0]
    tmp ← deck_sets[1:]
    deck_sets ← ∅
    for B in tmp do
      I ← A ∩ B; A_I ← A \ I; B_I ← B \ I
      if |I| == 0 then
        deck_sets.PUSH(B)
      else
        if |B_I| > 0 then
          deck_sets.PUSH(B_I)
        end if
        deck_sets.PUSH(I)
        A ← A_I
      end if
    end for
    disjoint_sets.PUSH(A)
  end while
  return disjoint_sets
end function
```

4.1.4 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 Section 5.1.4 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.

4.2 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).

5 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 (Section 5.6.2) 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 [15], 10 programs from the GNU coreutils package [14], and the nginx web server v1.20.1 [44]. 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 Section 4.2); 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?

5.1 SPEC CPU 2017

SPEC CPU 2017 [15] 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.

5.1.1 Performance. The normalized performance results are reported in Figure 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

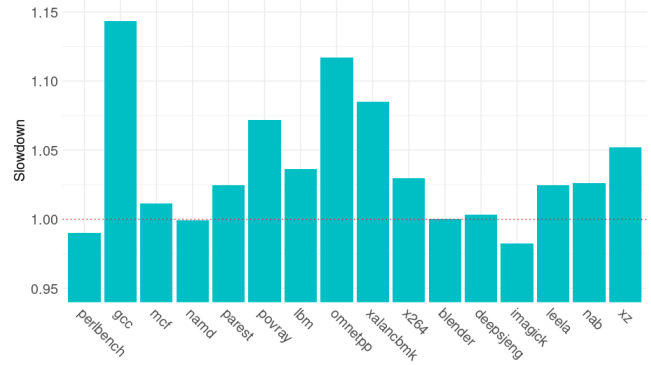


Figure 4: Slowdown for SPEC CPU 2017 using DECKER.

[32, 46, 48, 78]; 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 2: 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

5.1.2 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 Section 4.2). 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 2. 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 ~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).

5.1.3 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 (<1s); in 2 cases, it completes in <10s; 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.

5.1.4 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 sections 5.5 and 5.6 for details.

5.2 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 [17], 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 [49] 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: 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). 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.

5.3 nginx

nginx is by some metrics the most popular web server today [70]. 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 [48]. 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.

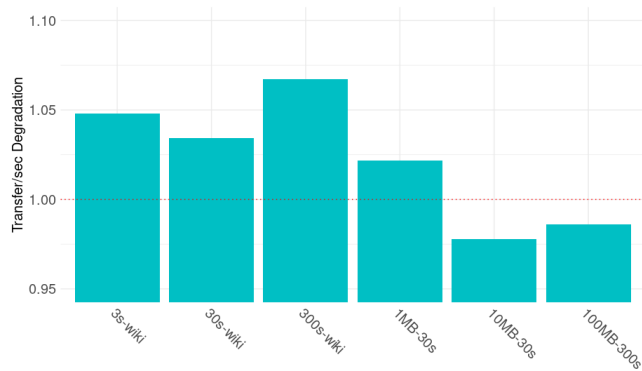


Figure 5: Transfer/sec degradation for `nginx` using `DECKER`.

The normalized performance result is shown in Figure 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.

5.4 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.

5.5 Breaking ROP Gadget Chains

Gadget reduction is a common metric in security-focused debloating works [27, 48, 49, 51, 60], 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` [55] 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.

5.6 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 [6]).

5.6.1 `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` [52] 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 4. 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 4: 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 4 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 4, 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` [9] 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 [8]).

5.6.2 `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 [7]), 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 [7].) 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 5.

As with Linux, this list of metrics shows how `DECKER` can make JOP-based attacks much more difficult. For example, the maximum

Table 5: 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

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 [7], 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.

6 RELATED WORK

Control flow integrity (CFI) [1, 21] 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 [5, 16, 19, 25, 28, 30, 36, 41, 45, 68, 69, 77]. Despite these advancements, current state-of-the-art still has its shortcomings. There are numerous examples of how to bypass CFI (e.g. [22]) or what its limits are (e.g. [11]). In fact recent work [35] 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 [28] cannot protect against code pointer reuse and VTable attacks. OS-CFI [29] fails to protect against tail calls that are optimized for indirect calls.

State-of-the-art, industry solutions such as Windows' CFG and RFG [37, 65, 71] and Intel's CET [59] are still vulnerable to code

reuse attacks [7]. Code replacement attacks, counterfeit object-oriented programming [56], 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 [63]. Windows 10 still relies on software-based CFG (not Intel's IBT [13]), which is more susceptible to bypasses [54, 66, 76]. 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) [32, 33] attempts to guarantee the integrity of all code pointers. Though effective, certain attacks may fall out of its scope (e.g. control-flow bending [11]). 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 [51], or Razor [49] 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 [50] 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 [31], achieving 77% debloat on nginx. Trimmer [60] is another technique that takes as input a user configuration and uses it to drive the debloating process. Soto-Valero et al. [62] leverage existing coverage tools to achieve similar debloating percentages for Java library bytecode.

ELFbac [3] 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, [38–40, 72–75] leverage debloating to improve performance, and [4, 18, 24, 42] use it to reduce code size.

Contrasted with the above approaches, the main contribution of this paper 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.

7 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.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and the shepherd for their invaluable comments to improve the paper significantly. The authors gratefully acknowledge 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(s) and do not necessarily reflect the views of the Office of Naval Research.

A ARTIFACT APPENDIX

A.1 Abstract

The Decker framework consists of a compiler pass and runtime library. Its main objective is to debloat software at runtime. The artifact includes a Docker image that encapsulates basic dependencies, the Decker code itself, benchmarks, and the scripts to drive artifact evaluation.

A.2 Artifact Checklist (Meta-Information)

- **Compilation:** LLVM
- **Transformations:** Instrumentation, linker sections
- **Run-time environment:** Runtime library
- **Hardware:** x86_64
- **Metrics:** Gadgets, runtime slowdown
- **How much disk space required (approximately)?:** 25GB
- **How much time is needed to prepare workflow (approximately)?:** 1 hour
- **How much time is needed to complete experiments (approximately)?:** 6 hours
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** Creative Commons Attribution 4.0 International
- **Archived (provide DOI)?:** 10.5281/zenodo.7222072

A.3 Description

A.3.1 How to access. The artifact is available on Zenodo at the following link: <https://doi.org/10.5281/zenodo.7222072>. Its DOI is 10.5281/zenodo.7222072.

A.3.2 Hardware dependencies. A commodity x86_64 desktop is sufficient. We used an AMD Ryzen 7 1800X (8 cores, 16 with hyperthreading) with 32GB RAM. (No GPU or special hardware is needed.)

A.3.3 Software dependencies. We tested on a host machine with Ubuntu 18.04.5 LTS, running Docker version 20.10.7, build f0df350. All software dependencies should be included in the Docker image.

A.4 Installation

Docker must be installed on the host machine. Refer to official Docker documentation for steps on installing Docker. All other dependencies should be encapsulated within the Docker image.

A.5 Experiment Workflow

The artifact includes a top-level README file that explains the steps for carrying out the experiment. Briefly, the artifact contains three top-level scripts: one to build the software, one to run all of it, and one to print the results.

A.6 Evaluation and Expected Results

The script that prints the results should produce output that closely matches the values in the paper. The script's output includes headers so that users know which section of the paper a result is referring to. In addition, the artifact includes a set of test results produced by the artifact itself. That is, the programs were built and run once, and their output was stored as part of the artifact. Thus, it is possible to re-run the script for printing results and verify that it matches the reference output, as well.

A.7 Experiment Customization

The source code for the framework is included in the artifact. Thus, users can extend or customize the evaluation.

A.8 Methodology

Submission, reviewing and badging methodology:

- <https://www.acm.org/publications/policies/artifact-review-badging>
- <http://cTuning.org/ae/submission-20201122.html>
- <http://cTuning.org/ae/reviewing-20201122.html>

REFERENCES

- [1] Martin Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security, CCS 2005, Alexandria, VA, USA, November 7–11, 2005*. Vijay Atluri, Catherine A. Meadows, and Ari Juels (Eds.). ACM, 340–353. <https://doi.org/10.1145/1102120.1102165>
- [2] S. Andersen and V. Abella. 2004. 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: 2021 Oct 10.
- [3] Julian Bangert, Sergey Bratus, Rebecca Shapiro, Michael E. Locasto, Jason Reeves, Sean W. Smith, and Anna Shubina. 2013. *ELFbac: Using the Loader Format for Intent-Level Semantics and Fine-Grained Protection*. Technical Report TR2013-727. Dartmouth University. https://digitalcommons.dartmouth.edu/cs_tr/345
- [4] Árpád Beszédés, Rudolf Ferenc, Tibor Gyimóthy, André Dolenc, and Konsta Karsisto. 2003. Survey of Code-size Reduction Methods. *ACM Comput. Surv.* 35, 3 (Sept. 2003), 223–267. <https://doi.org/10.1145/937503.937504>
- [5] Tyler K. Bletsch, Xuxian Jiang, and Vincent W. Freeh. 2011. 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, Robert H'obbes' Zakon, John P. McDermott, and Michael E. Locasto (Eds.). ACM, 353–362. <https://doi.org/10.1145/2076732.2076783>
- [6] Tyler K. Bletsch, Xuxian Jiang, Vincent W. Freeh, and Zhenkai Liang. 2011. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ASIACCS 2011, Hong Kong, China, March 22–24, 2011*, Bruce S. N. Cheung, Lucas Chi Kwong Hui, Ravi S. Sandhu, and Duncan S. Wong (Eds.). ACM, 30–40. <https://doi.org/10.1145/1966913.1966919>
- [7] Bramwell Brizendine and Austin Babcock. 2021. 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.
- [8] Bramwell Brizendine and Austin Babcock. 2021. 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.HV08jcdUeclPb4FciC2jpOuK07GR8. Accessed: 2022 Jun 23.
- [9] Michael D. Brown, Matthew Pruet, Robert Bigelow, Girish Mururu, and Santosh Pande. 2021. Not so fast: understanding and mitigating negative impacts of compiler optimizations on code reuse gadget sets. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–30. <https://doi.org/10.1145/3485531>
- [10] CallGraph.h 2020. CallGraph.h. <https://github.com/llvm/llvm-project/blob/llvmorg-11.0.0/llvm/include/llvm/Analysis/CallGraph.h>. Accessed: 2022 Nov 9.
- [11] Nicolas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. 2015. Control-flow Bending: On the Effectiveness of Control-flow Integrity. In *Proceedings of the 24th USENIX Conference on Security Symposium* (Washington, D.C.) (SEC'15). USENIX Association, Berkeley, CA, USA, 161–176. <http://dl.acm.org/citation.cfm?id=2831143.2831154>
- [12] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. 2010. Return-oriented programming without returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4–8, 2010*, Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov (Eds.). ACM, 559–572. <https://doi.org/10.1145/1866307.1866370>
- [13] Jonathan Corbet. 2022. Indirect branch tracking for Intel CPUs. <https://lwn.net/Articles/889475/>. Accessed: 2022 Oct 21.
- [14] coreutils 2021. Coreutils - GNU core utilities. <https://www.gnu.org/software/coreutils/>. Accessed: 2021 Jul 27.
- [15] Standard Performance Evaluation Corporation. 2021. SPEC CPU 2017. <https://www.spec.org/cpu2017/>. Accessed: 2021 Jul 27.
- [16] John Criswell, Nathan Dautenhahn, and Vikram S. Adve. 2014. 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, 292–307. <https://doi.org/10.1109/SP.2014.26>
- [17] cxreel. 2021. Razor. <https://github.com/cxreel/razor>. Accessed: 2021 Sept 30.
- [18] Saumya K. Debray, William Evans, Robert Muth, and Bjorn De Sutter. 2000. Compiler Techniques for Code Compaction. *ACM Trans. Program. Lang. Syst.* 22, 2 (March 2000), 378–415. <https://doi.org/10.1145/349214.349233>
- [19] Ren Ding, Chenxiong Qian, Chengyu Song, William Harris, Taesoo Kim, and Wenke Lee. 2017. Efficient Protection of Path-Sensitive Control Security. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16–18, 2017*, Engin Kirda and Thomas Ristenpart (Eds.). USENIX Association, 131–148. <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/ding>
- [20] Tyler Durden. 2002. Bypassing PaX ASLR protection. <http://phrack.org/issues/59/9.html>. Accessed: 2021 Sept 14.
- [21] Úlfar Erlingsson, Martín Abadi, Michael Vrbale, Mihai Budiu, and George C. Necula. 2006. XFI: Software Guards for System Address Spaces. In *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6–8, Seattle, WA, USA*, Brian N. Bershad and Jeffrey C. Mogul (Eds.). USENIX Association, 75–88. <http://www.usenix.org/events/osdi06/tech/erlingsson.html>
- [22] Isaac Evans, Fan Long, Ulzii Bayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. 2015. Control Jujuitsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (Denver, Colorado, USA) (CCS '15). ACM, New York, NY, USA, 901–913. <https://doi.org/10.1145/2810103.2813646>
- [23] Dmitry Evtvushkin, Dmitry V. Ponomarev, and Nael B. Abu-Ghazaleh. 2016. 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, 40:1–40:13. <https://doi.org/10.1109/MICRO.2016.7783743>
- [24] Michael Franz and Thomas Kistler. 1997. Slim Binaries. *Commun. ACM* 40, 12 (Dec. 1997), 87–94. <https://doi.org/10.1145/265563.265576>
- [25] Xinyang Ge, Weidong Cui, and Trent Jaeger. 2017. 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*, Yunji Chen, Olivier Temam, and John Carter (Eds.). ACM, 585–598. <https://doi.org/10.1145/3037697.3037716>
- [26] glibc 2021. glibc. <https://www.gnu.org/software/libc/>. Accessed: 2021 Oct 10.
- [27] Kihong Heo, Woosuk Lee, Pardis Pashakhanlou, and Mayur Naik. 2018. Effective Program Deobfuscation via Reinforcement Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) (CCS '18). ACM, New York, NY, USA, 380–394. <https://doi.org/10.1145/3243734.3243838>
- [28] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R. Harris, Taesoo Kim, and Wenke Lee. 2018. Enforcing Unique Code Target Property for Control-Flow Integrity. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) (CCS '18). ACM, New York, NY, USA, 1470–1486. <https://doi.org/10.1145/3243734.3243797>
- [29] Mustakimur Khandaker, Wenqing Liu, Abu Naser, Zhi Wang, and Jie Yang. 2019. Origin-sensitive Control Flow Integrity. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14–16, 2019*, Nadia Heninger and Patrick Traynor (Eds.). USENIX Association, 195–211. <https://www.usenix.org/conference/usenixsecurity19/presentation/khandaker>
- [30] Mustakimur Khandaker, Abu Naser, Wenqing Liu, Zhi Wang, Yajin Zhou, and Yueqiang Cheng. 2019. 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, 95–110. <https://doi.org/10.1109/EuroSP.2019.00017>
- [31] Hyungjoon Koo, Seyedhamed Ghavamnia, and Michalis Polychronakis. 2019. Configuration-Driven Software Deobfuscation. In *Proceedings of the 12th European Workshop on Systems Security, EuroSec@EuroSys 2019, Dresden, Germany, March 25, 2019*. ACM, 9:1–9:6. <https://doi.org/10.1145/3301417.3312501>
- [32] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2014. Code-pointer Integrity. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) (OSDI'14). USENIX Association, Berkeley, CA, USA, 147–163. <http://dl.acm.org/citation.cfm?id=2685048.2685061>
- [33] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, and Dawn Song. 2015. Poster : Getting The Point(er): On the Feasibility of Attacks on Code-Pointer Integrity.
- [34] Chris Lattner and Vikram S. Adve. 2004. 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, 75–88. <https://doi.org/10.1109/CGO.2004.1281665>
- [35] Yuan Li, Mingzhe Wang, Chao Zhang, Xingman Chen, Songtao Yang, and Ying Liu. 2020. 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*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM, 1821–1835. <https://doi.org/10.1145/3372297.3417867>
- [36] Y. Liu, P. Shi, X. Wang, H. Chen, B. Zang, and H. Guan. 2017. Transparent and Efficient CFI Enforcement with Intel Processor Trace. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 529–540. <https://doi.org/10.1109/HPCA.2017.18>
- [37] Microsoft. 2022. Control Flow Guard for platform security. <https://learn.microsoft.com/en-us/windows/win32/secbp/control-flow-guard>. Accessed: 2022 Oct 21.
- [38] Nick Mitchell, Edith Schonberg, and Gary Sevitsky. 2009. Making Sense of Large Heaps. In *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6–10, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5653)*, Sophia Drossopoulou (Ed.). Springer, 77–97. https://doi.org/10.1007/978-3-642-03013-0_5
- [39] Nick Mitchell, Edith Schonberg, and Gary Sevitsky. 2010. Four Trends Leading to Java Runtime Bloat. *IEEE Software* 27, 1 (2010), 56–63. <https://doi.org/10.1109/MS.2010.7>
- [40] Nick Mitchell and Gary Sevitsky. 2007. 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*, Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr. (Eds.). ACM, 245–260. <https://doi.org/10.1145/1297027.1297046>
- [41] Vishwath Mohan, Per Larsen, Stefan Brunthaler, Kevin W. Hamlen, and Michael Franz. 2015. 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. <https://www.ndss-symposium.org/ndss2015/opaque-control-flow-integrity>
- [42] Robert Muth, Saumya K. Debray, Scott Watterson, and Koen De Bosschere. 2001. Alto: A Link-time Optimizer for the Compaq Alpha. *Softw. Pract. Exper.* 31, 1 (Jan. 2001), 67–101. [https://doi.org/10.1002/1097-024X\(200101\)31:1<67::AID-SPE357>3.0.CO;2-A](https://doi.org/10.1002/1097-024X(200101)31:1<67::AID-SPE357>3.0.CO;2-A)
- [43] Nergal. 2001. The Advanced Return-into-lib(c) Exploits: PaX Case Study. Phrack Magazine. <http://phrack.org/issues/58/4.html>. Accessed: 2021 Oct 10.
- [44] nginx 2019. nginx. <https://nginx.org/>. Accessed: 2021 Oct 10.

- [45] Ben Niu and Gang Tan. 2014. Modular control-flow integrity. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 577–587. <https://doi.org/10.1145/2594291.2594295>
- [46] Ben Niu and Gang Tan. 2015. 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*, Indrajit Ray, Ninghui Li, and Christopher Kruegel (Eds.). ACM, 914–926. <https://doi.org/10.1145/2810103.2813644>
- [47] PAX ASLR 2003. PaX Address Space Layout Randomization. <https://pax.grsecurity.net/docs/aslr.txt>. Accessed: 2021 Oct 10.
- [48] Chris Porter, Girish Mururu, Prithayan Barua, and Santosh Pande. 2020. 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 Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 164–180. <https://doi.org/10.1145/3385412.3386017>
- [49] Chenxiong Qian, Hong Hu, Mansour Alharthi, Simon Pak Ho Chung, Taesoo Kim, and Wenke Lee. 2019. RAZOR: A Framework for Post-deployment Software Debloating. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, Nadia Heninger and Patrick Traynor (Eds.). USENIX Association, 1733–1750. <https://www.usenix.org/conference/usenixsecurity19/presentation/qian>
- [50] Chenxiong Qian, Hyungjoon Koo, ChangSeok Oh, Taesoo Kim, and Wenke Lee. 2020. 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*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM, 461–476. <https://doi.org/10.1145/3372297.3417866>
- [51] Anh Quach, Aravind Prakash, and Lok Yan. 2018. Debloating Software through Piece-Wise Compilation and Loading. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 869–886. <https://www.usenix.org/conference/usenixsecurity18/presentation/quach>
- [52] ROPgadget 2018. ROPgadget v5.4. <https://github.com/JonathanSalwan/ROPgadget>. Accessed: 2021 Oct 10.
- [53] AliAkbar Sadeghi, Salman Niksefat, and Maryam Rostampour. 2018. Pure-Call Oriented Programming (PCOP): chaining the gadgets using call instructions. *J. Comput. Virol. Hacking Tech.* 14, 2 (2018), 139–156. <https://doi.org/10.1007/s11416-017-0299-1>
- [54] Morten Schenk. [n.d.]. Bypassing Control Flow Guard in Windows 10. <https://improsec.com/tech-blog/bypassing-control-flow-guard-in-windows-10>.
- [55] Sascha Schirra. 2021. Ropper. <https://github.com/sashes/ropper>. Accessed: 2021 Sept 10.
- [56] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 745–762. <https://doi.org/10.1109/SP.2015.51>
- [57] Hovav Shacham. 2007. 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*, Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson (Eds.). ACM, 552–561. <https://doi.org/10.1145/1315245.1315313>
- [58] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. 2004. 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*, Vijayalakshmi Atluri, Birgit Pfiftzmann, and Patrick D. McDaniel (Eds.). ACM, 298–307. <https://doi.org/10.1145/1030083.1030124>
- [59] Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. 2019. Security Analysis of Processor Instruction Set Architecture for Enforcing Control-Flow Integrity. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy (Phoenix, AZ, USA) (HASP '19)*. Association for Computing Machinery, New York, NY, USA, Article 8, 11 pages. <https://doi.org/10.1145/3337167.3337175>
- [60] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. 2018. 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*, Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.). ACM, 329–339. <https://doi.org/10.1145/3238147.3238160>
- [61] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2013. 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 (SP '13)*. IEEE Computer Society, Washington, DC, USA, 574–588. <https://doi.org/10.1109/SP.2013.45>
- [62] César Soto-Valero, Thomas Durieux, Nicolas Harrand, and Benoit Baudry. 2022. Coverage-Based Debloating for Java Bytecode. *ACM Trans. Softw. Eng. Methodol.* (jun 2022). <https://doi.org/10.1145/3546948> Just Accepted.
- [63] Bing Sun, Jin Liu, and Chong Xu. 2019. 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.
- [64] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. 2000. Practical Virtual Method Call Resolution for Java. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Minneapolis, Minnesota, USA) (OOPSLA '00). Association for Computing Machinery, New York, NY, USA, 264–280. <https://doi.org/10.1145/353171.353189>
- [65] Jack Tang. 2015. *Exploring Control Flow Guard in Windows 10*. Technical Report. Trend Micro. <https://documents.trendmicro.com/assets/wp/exploring-control-flow-guard-in-windows10.pdf>
- [66] Sam Thomas. [n. d.]. 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.
- [67] Minh Tran, Mark Etheridge, Tyler Bletsch, Xuxian Jiang, Vincent Freeh, and Peng Ning. 2011. On the Expressiveness of Return-into-libc Attacks. In *Recent Advances in Intrusion Detection*, Robin Sommer, Davide Balzarotti, and Gregor Maier (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 121–141.
- [68] Victor van der Veen, Dennis Andriese, Enes Göktas, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. 2015. Practical Context-Sensitive CFI. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, Indrajit Ray, Ninghui Li, and Christopher Kruegel (Eds.). ACM, 927–940. <https://doi.org/10.1145/2810103.2813673>
- [69] Victor van der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. 2016. 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, 934–953. <https://doi.org/10.1109/SP.2016.60>
- [70] W3Techs. 2021. Usage statistics of web servers. https://w3techs.com/technologies/overview/web_server. Accessed: 2021 Oct 10.
- [71] Danny Wei, Lywang, and FlowerCode. 2016. Return Flow Guard. <https://xlab.tencent.com/en/2016/11/02/return-flow-guard/>. Accessed: 2022 Oct 21.
- [72] Guoqing Xu and Atanas Rountev. 2010. Detecting Inefficiently-used Containers to Avoid Bloat. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (Toronto, Ontario, Canada) (PLDI '10)*. ACM, New York, NY, USA, 160–173. <https://doi.org/10.1145/1806596.1806616>
- [73] Guoqing (Harry) Xu. 2012. 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*, Gary T. Leavens and Matthew B. Dwyer (Eds.). ACM, 1017–1034. <https://doi.org/10.1145/2384616.2384690>
- [74] Guoqing (Harry) Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. 2010. 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*, Benjamin G. Zorn and Alexander Aiken (Eds.). ACM, 174–186. <https://doi.org/10.1145/1806596.1806617>
- [75] Guoqing (Harry) Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, and Gary Sevitsky. 2010. 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*, Gruia-Catalin Roman and Kevin J. Sullivan (Eds.). ACM, 421–426. <https://doi.org/10.1145/1882362.1882448>
- [76] Zhang Yunha. [n. d.]. Bypass Control Flow Guard Comprehensively. <https://www.blackhat.com/docs/us-15/materials/us-15-Zhang-Bypass-Control-Flow-Guard-Comprehensively-wp.pdf>.
- [77] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. 2013. 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, 559–573. <https://doi.org/10.1109/SP.2013.44>
- [78] Mingwei Zhang and R. Sekar. 2013. Control Flow Integrity for COTS Binaries. In *Proceedings of the 22nd USENIX Conference on Security (Washington, D.C.) (SEC '13)*. USENIX Association, Berkeley, CA, USA, 337–352. <http://dl.acm.org/citation.cfm?id=2534766.2534796>

Received 2022-07-07; accepted 2022-09-22