NOPoutNG: Improving the Effectiveness of Hardware-assisted Control-flow Integrity via Dynamic Landing Pad Elision

Alexander J. Gaidis
Brown University
Providence, USA
agaidis@cs.brown.edu

Jamie Gabbay
Brown University
Providence, USA
jamie_gabbay@brown.edu

Joao Moreira
Microsoft
Redmond, USA
joaomoreira@microsoft.com

Vasileios P. Kemerlis
Brown University
Providence, USA
vpk@cs.brown.edu

***Abstract*—Hardware-assisted Control-flow Integrity (CFI), such as Intel's Indirect Branch Tracking (IBT), provides an efficient defense against control-flow hijacking attacks. However, the protection it provides is fundamentally coarse-grained, leaving a large attack surface of valid indirect branch targets that an attacker can (ab)use. Recently, FineIBT introduced a novel, load-time feature called NOPout that aims to reduce this attack surface by removing `endbr` instructions from unused, exported functions. Unfortunately, NOPout suffers from several design limitations—including insecure fail-safe defaults, performance overhead from excessive code patching and copy-on-write (COW) memory effects, and a cursory design for dynamically loaded libraries—as well as an evaluation narrow in scope. To address the shortcomings of NOPout, we present `NOPoutNG`: a novel mechanism that reverses NOPout's insecure model with a secure-by-default approach. It begins by inserting inert `nop` instructions into exported functions, only promoting them to active `endbr` "landing pads" when there is concrete evidence—discovered at compile, link, or load time—that a function is actually used (i.e., address-taken). We implement `NOPoutNG` as a collection of minor modifications to the LLVM toolchain, as well as a load-time shared library, and demonstrate its practicality and security via a comprehensive evaluation where it achieves up to 86% `endbr` reduction vs. vanilla IBT with minimal overhead.**

***Index Terms*—Intel CET/IBT, FineIBT, CFI enforcement**

1. Introduction

Software systems are woven into nearly every aspect of contemporary life—powering economies, shaping politics, and transforming science and education. Unfortunately, modern software is increasingly large and complex due to requirements on backwards compatibility [1] and supporting a rich set of features for expanding, heterogeneous user-bases [2]–[5]. As a result, code bases are riddled with weaknesses that attackers routinely exploit—typically for financial or social gain. Among the myriad of vulnerabilities that exist, *memory errors* [6] continue to be among the most malefic, dominating the SANS institute's list of the "*top 25 most dangerous software errors*" [7]. These software bugs can be (ab)used by attackers to corrupt or leak the contents of memory [8] or cause denial-of-service (DoS) [9]. Their threat is far from theoretical: Microsoft and Google report that ≈70% of all vulnerabilities in their products stem from memory-safety problems [10]–[12].

Applications written in memory- or type-unsafe languages, such as C and C++, are often exploited through spatial *spatial* [13] or *temporal* [14] memory-safety violations. By targeting these vulnerabilities, attackers can manipulate control data [15] (e.g., return addresses, function pointers) to hijack execution and run arbitrary code [16]. Presently, to achieve arbitrary code execution, attackers now commonly employ code-reuse techniques, including ROP [17], JOP [18], [19], COP [20], JIT-ROP [21], COOP [22] (see Appendix A). In response, numerous mitigations have been developed, with one of the most prominent being Control-flow Integrity (CFI) [23]—a security mechanism that confines a program's control flow to a set of "legal" code paths.

The pioneering CFI work by Abadi et al. [24] ignited a surge of innovation, with the following two decades seeing an abundance of CFI-based mitigations ranging from software-only schemes [24]–[62], to hardware-assisted [63]–[79] and hardware-only solutions [80]–[83]. Recently, CFI has caught the attention of industry, with Microsoft using it to secure Windows [84], Google using it to harden Android [40], [85] and Chrome [86], and Intel and ARM adding new hardware extensions—e.g., CET [87] and BTI [88]—to accelerate it. (See Appendix B for a literature review.)

Indirect Branch Tracking (IBT) constitutes the forward-edge, control-flow protection—i.e., protection of indirect `call`/`jmp` instructions—offered by Intel's Control-flow Enforcement Technology (CET) hardware feature [89], [90]. At a high-level, IBT inserts a new, 4-byte instruction called an `endbr` at the beginning of all indirect-branch targets, "pinning" the indirect branches to predetermined locations; if an indirect branch targets a non-`endbr` instruction the CPU raises an exception. While IBT is efficient, being implemented in hardware, its protection model is fundamentally coarse-grained, allowing an attacker with control of an indirect branch to "bend" the control-flow of the program, in arbitrary ways, steering it toward any location marked with an `endbr` [22], [23], [91], [92]. To improve the granularity of IBT, a more recent scheme, FineIBT [93], builds upon IBT by leveraging its `endbr` "landing pads" to pin indirect branches to certain locations where additional instrumentation further restricts allowed targets.

Importantly, irrespective of the sensitivity of the policy applied to FineIBT, there will typically be more `endbrs` in a program than necessary to support dynamic linking. For example, the exported symbols in a shared library must have an `endbr` in their prologue since they *may* be called indirectly by another program that dynamically links with the library and invokes an exported function via its PLT/GOT. As a result, an attacker can "bend" the control-flow of the program to extraneous `endbr` "landing pads" in a victim process' address space (e.g., unused, exported functions). To combat this issue, FineIBT includes a load-time feature called NOPout that *elides* `endbr` instructions from unused, address-taken functions, replacing them with `nop` instructions. This further limits the available targets of indirect branches, irrespective of the policy being applied.

While NOPout is a step in the right direction, it suffers from four major problems [93]. First, NOPout liberally adds `endbr` to *all* exported functions, only to replace them with `nops` in unused functions at load-time. This design does not provide fail-safe defaults—i.e., if the NOPout library does not get loaded, or has a bug, an attacker will silently gain access to many more extraneous, but legal, `endbr` targets. Additionally, libraries typically export many more functions than are ever used by a given program, leading to increased performance overhead due, in part, to function patching triggering copy-on-write (COW) for shared objects. Second, to support this problematic order of operations, NOPout propagates metadata, collected during compilation and (static) linking, to the NOPout run-time library to avoid removing `endbr` instructions from exported functions that are address-taken locally. This metadata is stored in a new ELF section that increases both the storage requirements for a protected binary and its in-memory footprint. Third, NOPout's support for dynamically loaded shared libraries is not thoroughly studied: its design is limited and it was not implemented. Finally, there is no performance evaluation of NOPout and only a minimal effectiveness evaluation.

To address all four limitations of NOPout, we present NOPout Next Generation (`NOPoutNG`): an efficient, secure, and thoroughly studied load-time mechanism to reduce a program's attack surface by limiting available indirect-branch targets. Instead of starting with an insecure default state, `NOPoutNG` compiles `nop` instructions at the beginning of all exported functions and only replaces them with `endbrs` when knowledge of their use is made available during compilation, linking, or loading via information already present in the ELF file. Further, we provide a comprehensive design for handling dynamically loaded shared objects, and we improve upon NOPout's evaluation, considering the performance and effectiveness of `NOPoutNG`. Regarding performance, `NOPoutNG` exhibited negligible overhead atop IBT in several industry-standard, and real-world, benchmark applications; regarding effectiveness, `NOPoutNG` is highly effective reducing available `endbrs` in IBT-hardened programs up to 86% with fewer COW effects than NOPout.

2. Background

2.1. (Fine)IBT

IBT. Control-flow Enforcement Technology (CET) [87], [89], [90] is a recent hardware extension available from 11th generation (i.e., "Tiger Lake") Intel CPUs. It provides two new hardware features to confine indirect branches: (1) a *shadow stack* for confining backward-edge control-flow transfers, and (2), Indirect Branch Tracking (*IBT*) for confining forward-edge transfers. Notably, the shadow stack CET provides is an efficient, fine-grained, and *compact* [94], [95] implementation that protects backward-edge control-flow transfers. This contrasts IBT, which is performant, but only offers coarse-grained, forward-edge protection.

To support IBT, CET introduced a new 4-byte instruction: `endbr`. When IBT is enabled, every indirect, forward-edge control-flow transfer (i.e., indirect `call`/`jmp` instructions) must target an `endbr` "landing pad," else a `#GP` exception is raised. On older hardware that does not provide CET, `endbr` instructions are treated as `nop` instructions.

Notably, when compiling source code into an executable or shared library (DSO) with IBT protection, the compiler adds an `endbr` at the beginning of every function that is address-taken locally (i.e., within the current compilation unit), or is exported and could thus be address-taken in the future when object files are linked together to create an executable or DSO. As a result, IBT wildly over-approximates the set of allowed targets for all indirect branches; an attacker-controlled indirect branch can target any `endbr`, and since these are placed at least at the beginning of all exported library functions, they can easily perform full-function code reuse [20], [96].

FineIBT. To address the inherent weakness in IBT, Gaidis et al. introduced FineIBT [93], improving the granularity of IBT by restricting the number of legal locations a given indirect branch can target. FineIBT provides a *mechanism* to apply various CFI policies to a binary, ranging from coarse-grained policies to finer-grained ones. At a high-level, FineIBT leverages IBT to confine indirect control-flow transfers to the beginning of functions (via `endbrs`) before performing a set-ID (SID) check that compares an (immediate) SID value loaded into a register at the branch site to an (immediate) SID value at the branch target. If the SID values match then execution continues; if not, an exception is raised. FineIBT provides policy agnosticism through SIDs, with different policies providing different SIDs to restrict an indirect branch's allowed targets.

System Support. IBT and FineIBT have been gaining traction and growing in popularity. Since Linux kernel v6.2, IBT is now enabled by default in the kernel [97], and toolchain and runtime support for building and running user-space programs with IBT is already quite mature [98]–[101]. Additionally, preliminary FineIBT support has also been added to the Linux kernel in v6.2 [102].

2.2. NOPout

FineIBT [93] can apply arbitrary CFI policies to a binary via its SID-checking instrumentation that is added atop IBT. While this improves the granularity of IBT, FineIBT inherits some of the limitations that arise from how IBT is implemented in modern toolchains. In particular, irrespective of the granularity of the underlying policy, many more functions will be instrumented than are necessary to run a given program. For example, out of 1664 unique functions (i.e., not including aliases) exported by `musl libc`, `redis-server` only uses 233, but all 1664 are still instrumented with `endbrs` and SID checks. While this may not appear bad if the applied policy pairs every indirect branch with only one target (e.g., UCT [77]), the vast majority of policies create equivalence classes based on a property like types. In cases like this, two unrelated library functions that share the same signature can both be targeted by an indirect branch, even if the current program only ever uses one of them. Regardless of the policy being applied, functions that are never used by a program should not contain valid `endbr` landing pads, potentially increasing the program's attack surface.

To address this problem, FineIBT proposed a load-time feature, dubbed NOPout, to *elide* `endbrs` from unused functions at run time. To achieve this, NOPout starts by recording every: (a) non-address-taken, (b) non-local, and (c), (Fine)IBT-protected `FUNC` symbol in a new ELF section called `.plt.nopout` during compilation. Then, at load time, the section is loaded and parsed by the dynamic linker/loader (i.e., `ld.so`) and its entries are compared against PLT/GOT slots. If an entry in `.plt.nopout` is not present in a PLT/GOT slot, then the function is deemed unused and a runtime library (i.e., `libnopout.so`) will replace the function's 4-byte `endbr` with a 4-byte `nop`. This effectively prevents any indirect branch from targeting the function, irrespective of any instituted policy. Notably, this also minimizes speculative code-reuse attacks [103].

3. Motivation

While the initial debut of NOPout was interesting and well-intentioned, it suffers from several limitations. These problems span its *design* and *implementation*, as well as its brief *evaluation*. The rest of this section details four of these issues, and, in addition, how `NOPoutNG` aims to solve them, filling the respective gaps in the design space.

Problem #1: Order of Operations. As exemplified in Figure 1, NOPout liberally adds `endbr` instructions to all exported functions, and then later, at runtime, replaces the `endbr` of unreferenced, exported functions with an inert 4-byte `nop` instruction. The problem here is two-fold: (a) this order of operations does not provide *fail-safe defaults*—if the NOPout runtime library is not loaded, or has a bug, an attacker will have many more extraneous, but legal, `endbr` targets to choose from; and (b), typically, libraries export many more functions than an executable will use—for example, `musl libc` exports 1664 unique (i.e., not aliased) functions, but `redis-server` only requires 233—making NOPout perform many `nop` writes at load time. In short, patching (oftentimes many) `endbr` instructions with `nop` instructions, at load time or run time, is not ideal for the vast majority of cases, in part due to COW.

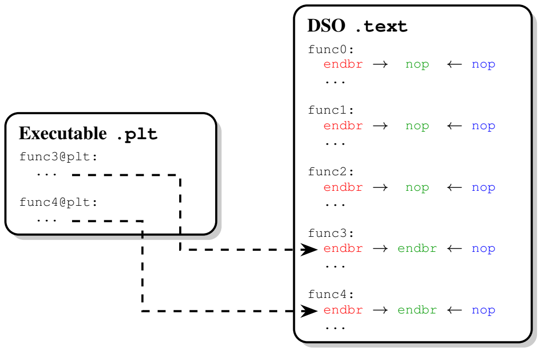


Figure 1. Example of an executable calling external library functions under NOPout and `NOPoutNG`. Under NOPout, all exported functions in a DSO start with an `endbr`, which is replaced with a `nop` if the function is *unused*. In contrast, `NOPoutNG` starts all exported functions with a `nop` instruction that is replaced with an `endbr` if the function is *used*.

Solution #1: Compile-time `nops`. To solve Problem #1, `NOPoutNG` trades backwards-compatibility—i.e., being able to run a `NOPoutNG`-compiled binary on a system without the `NOPoutNG` runtime library—for *improved performance* and stronger *secure-by-default guarantees*. `NOPoutNG` compiles `nop` instructions at the beginning of all exported functions (that are not locally address-taken) and waits until run time (specifically, load time) to patch the inert `nops` in the used functions' prologues with active `endbrs`. In essence, Solution #1 switches the order of operations from adding `nops` at *run time*—as in NOPout—to adding `nops` at *compile time*.

Problem #2: Extraneous Metadata. To support NOPout's goal of replacing `endbr` instructions with `nops` at run time, it collects every function that is non-address-taken, non-local, and (Fine)IBT-protected in a new ELF section, dubbed `.plt.nopout`. The section contents are then compared with PLT/GOT slots at run time to determine which functions should have their `endbrs` replaced. However, this section, which slows down compilation and increases executable file sizes, is *completely unnecessary* given the new order of operations in `NOPoutNG`.

Solution #2: Relocations Mining. `NOPoutNG` *leverages relocation data*—which are present in regular object files—to determine what functions are always address-taken locally in the executable (or DSO), and thus should not be patched with `nop` instructions.

Problem #3: Dynamic Loading. While the description of NOPout's design in FineIBT [93] does mention that NOPout can support dynamically loaded shared libraries, it does not verify this through implementation and evaluation. Further, details are omitted regarding how `dlopen`, `dlsym`, and `dlclose` are handled, and how NOPout identifies functions that subsequently need their `endbrs` replaced or elided.

Solution #3: Verified Dynamic Loading Support. We address the shortcomings of NOPout by providing a more substantial, *concrete design* (§5.5) and *implementation* (§6) for handling dynamically loaded DSOs in `NOPoutNG`, as well as an *evaluation* of its performance overhead (§7.1.3).

Problem #4: Limited Evaluation. The original NOPout work contains only a brief evaluation of its effectiveness, as NOPout is not the primary focus of the work; the limited evaluation provides quantitative metrics for how many `endbr` instructions NOPout is able to remove at run time. While this is a good starting point, it falls significantly short of the mark for wholistically understanding how NOPout would perform in a production system. There is no additional analysis or discussion of the results—e.g., where over-approximation might still be present, if at all. Also, there is no performance evaluation of NOPout, including the handling of dynamically loaded libraries (via `dlopen`/`dlsym`).

Solution #4: Full Evaluation of `NOPoutNG`. We greatly supplement NOPout's evaluation with a thorough effectiveness (§7.2) and performance (§7.1) evaluation of `NOPoutNG`, including the handling of dynamically loaded shared objects (§7.1.3). Further, our evaluation compares the new design of `NOPoutNG` against NOPout's design.

4. Threat Model

Adversarial Capabilities. We assume a capable, state-of-the-art adversary aiming to hijack the control flow of a program by exploiting one, or multiple, memory errors [6] in its codebase. We put no restrictions on the type of memory errors the adversary has at their disposal (e.g., spatial and/or temporal [13], [14]), nor do we restrict the frequency at which the adversary can trigger memory errors. Formally, we consider an adversary who can disclose or corrupt any readable or writable memory, respectively, whenever and however many times they need during a program's execution in order to hijack its control flow. These adversarial capabilities represent the current state-of-the-{art, practice} regarding CFI [23], [104], [105] and mirror the capabilities laid out in the original NOPout work [93].

Hardening Assumptions. Similar to the hardening assumptions presented in the original NOPout work [93], we assume an adversary is targeting an x86 platform that has support for non-executable memory [106]—and correctly enforces the W^X memory policy [107]—and Intel CET [87], [89], [90] (i.e., the CPU is 11th generation—"Tiger Lake"—or later). Further, we assume that both IBT and shadow stacks are enabled properly to protect forward- and backward-edge transfers, respectively, and cannot be disabled or "turned off." Given these hardening assumptions, an attacker is still able to tamper with code pointers and forward-edge transfers to perform code reuse (e.g., JOP and COP [18]–[20]). Other mitigations such as ASLR [108], stack-smashing protection [109], code randomization/diversification [110]–[112], protection against data-only attacks [113], and speculative memory-error abuse [103] are all orthogonal to our work: `NOPoutNG` neither precludes nor requires them.

5. Design

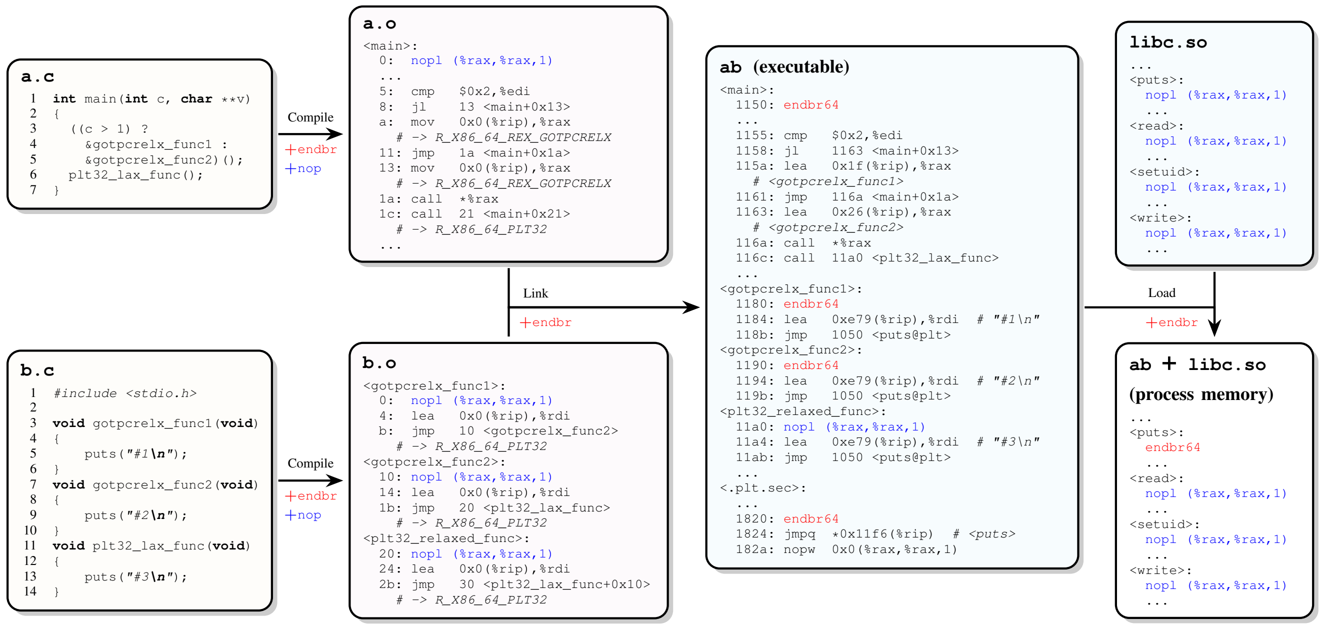
5.1. Overview

`NOPoutNG` is a toolchain-assisted, {*load, run*}*-time* mechanism that aims to reduce the number of over-approximated (Fine)IBT targets in a process, and thus reduce its overall attack surface. At a high-level, `NOPoutNG` slightly modifies the compiler and (static) linker to delay determining whether a function can be a legal indirect-branch target—i.e., it is address-taken and should begin with an `endbr`—until conclusive evidence is found. Meanwhile, `NOPoutNG` inserts an inert 4-byte `nop` instruction at the beginning of the function in question, preventing an indirect branch from targeting it, but saving space for the `nop` to be turned into an `endbr` as more information becomes available. Since a function can be the target of an indirect branch (1) *locally* (i.e., within the same compilation unit), (2) *across object files*, or (3), *across DSOs* (i.e., libraries), `NOPoutNG` adds logic to compilation, (static) linking, and loading to determine what functions require the insertion of an `endbr` for the program to run. (For compiling, linking, and loading background, see Appendix C.) Since the majority of indecision arises from run-time behavior—e.g., different programs might use a different set of exported library functions—the bulk of the work done by `NOPoutNG` is at load time in the form of an "injected" (i.e., `LD_PRELOAD`ed) library.

While similar in objective to its predecessor, NOPout, `NOPoutNG` follows a fundamentally different approach, exploring different trade-offs in addition to solving several of its shortcomings (§3). `NOPoutNG` ultimately tries to provide *fail-safe defaults* and *enhanced performance* by starting with a set of *invalid* indirect-branch targets (i.e., they begin with `nop`s instead of `endbr`) and only adding them to a valid set (i.e., exchange a `nop` for an `endbr`) when there is concrete evidence to indicate they are address-taken somewhere in the currently running process. In contrast, NOPout starts with a large set of *valid* indirect-branch targets and prunes functions from there, replacing their `endbr`s with `nop`s. This has the benefit of being backwards-compatible—i.e., a NOPout-compiled binary will work on IBT-equipped systems without the corresponding NOPout runtime library—at the cost of security and performance.

5.2. Compilation

In the compiler, `NOPoutNG` interposes on IBT's `endbr` insertion code, iterating over all functions in the compilation unit to determine if they are address-taken and have local linkage (i.e., `ST_BIND == STB_LOCAL`); if true for a given function, `NOPoutNG` inserts an `endbr` in its prologue, else a `nop`. Notably, a function is considered address-taken if any of its uses are not direct calls or invocations, e.g., storing the address of the function into memory, passing the function address as an argument to a call, or casting the function's address to another pointer type. Thus, the result of feeding source code to `NOPoutNG`'s compilation phase



a.c

```
int main(int c, char **v)
{
  ((c > 1) ?
    &gotpcrelx_func1 :
    &gotpcrelx_func2)();
  plt32_lax_func();
}
```

a.o

```
<main>:
  0:  nopl (%rax,%rax,1)
  ...
  5:  cmp   $0x2,%edi
  8:  jl    13 <main+0x13>
  a:  mov   0x0(%rip),%rax
    # -> R_X86_64_REX_GOTPCRELX
  11: jmp   1a <main+0x1a>
  13: mov   0x0(%rip),%rax
    # -> R_X86_64_REX_GOTPCRELX
  1a: call  *%rax
  1c: call  21 <main+0x21>
    # -> R_X86_64_PLT32
  ...
```

b.c

```
#include <stdio.h>

void gotpcrelx_func1(void)
{
    puts("#1\n");
}
void gotpcrelx_func2(void)
{
    puts("#2\n");
}
void plt32_lax_func(void)
{
    puts("#3\n");
}
```

b.o

```
<gotpcrelx_func1>:
  0:  nopl (%rax,%rax,1)
  4:  lea   0x0(%rip),%rdi
  b:  jmp   10 <gotpcrelx_func2>
    # -> R_X86_64_PLT32
<gotpcrelx_func2>:
  10: nopl (%rax,%rax,1)
  14: lea   0x0(%rip),%rdi
  1b: jmp   20 <plt32_lax_func>
    # -> R_X86_64_PLT32
<plt32_relaxed_func>:
  20: nopl (%rax,%rax,1)
  24: lea   0x0(%rip),%rdi
  2b: jmp   30 <plt32_lax_func+0x10>
    # -> R_X86_64_PLT32
```

ab (executable)

```
<main>:
  1150: endbr64
  ...
  1155: cmp   $0x2,%edi
  1158: jl    1163 <main+0x13>
  115a: lea   0x1f(%rip),%rax
    # <gotpcrelx_func1>
  1161: jmp   116a <main+0x1a>
  1163: lea   0x26(%rip),%rax
    # <gotpcrelx_func2>
  116a: call  *%rax
  116c: call  11a0 <plt32_lax_func>
  ...
<gotpcrelx_func1>:
  1180: endbr64
  1184: lea   0xe79(%rip),%rdi  # "#1\n"
  118b: jmp   1050 <puts@plt>
<gotpcrelx_func2>:
  1190: endbr64
  1194: lea   0xe79(%rip),%rdi  # "#2\n"
  119b: jmp   1050 <puts@plt>
<plt32_relaxed_func>:
  11a0: nopl (%rax,%rax,1)
  11a4: lea   0xe79(%rip),%rdi  # "#3\n"
  11ab: jmp   1050 <puts@plt>
  ...
<.plt.sec>:
  ...
  1820: endbr64
  1824: jmpq  *0x11f6(%rip)  # <puts>
  182a: nopw  0x0(%rax,%rax,1)
```

libc.so

```
...
<puts>:
  nopl (%rax,%rax,1)
  ...
<read>:
  nopl (%rax,%rax,1)
  ...
<setuid>:
  nopl (%rax,%rax,1)
  ...
<write>:
  nopl (%rax,%rax,1)
  ...
```

ab + libc.so (process memory)

```
...
<puts>:
  endbr64
  ...
<read>:
  nopl (%rax,%rax,1)
  ...
<setuid>:
  nopl (%rax,%rax,1)
  ...
<write>:
  nopl (%rax,%rax,1)
  ...
```

Figure 2. Example compiling {a, b}.c to object files {a, b}.o, linking them into executable ab, and running ab. Notable relocations are commented.

is an output object file that has endbrs at the beginning of any function that is address-taken somewhere within the compilation unit. Functions that require more information to determine if they are address-taken (e.g., from other compilation units) have nops inserted in their prologue.

From Figure 2: compiling {a, b}.c under NOPoutNG yields a.o and b.o with nop instructions in each function prologue as no function is address-taken locally.

5.3. Static Linking

The main goal of NOPoutNG's additions to the (static) linker is to replace the nop instructions inserted by the compiler with endbrs in address-taken functions. Since the linker has additional context that the compiler was missing (i.e., information about all the compilation units that will comprise a binary), it can refine the set of allowed indirect branch targets prior to runtime. The key insight here is that *the static relocation information emitted by the compiler is sufficient to identify all functions that are address-taken across compilation units*. This means that NOPoutNG does not require link-time optimization (LTO) nor does it need to propagate additional metadata to the dynamic linker/loader during loading as NOPout required. In NOPout's case, the additional section, .plt.nopout, was necessary to identify functions that could have their endbr elided so, for example, an exported function that is also address-taken locally does not get elided (see Problem #2; §3). The additive approach of NOPoutNG contrasts this: if an exported function is address-taken locally it will already have an endbr in its prologue and no more work is required—only the functions whose address-taken property is decided at run time will ever be marked with nop instructions. For detailed information regarding what relocations NOPoutNG uses to identify address-taken functions, refer to Appendix D.

In Figure 2, linking {a, b}.o into executable ab under NOPoutNG adds endbrs to several functions, including: main, due to its address being taken in _start of Scrt1.o (omitted for space), and gotpcrelx_func{1, 2}, due to their their address being taken in main of a.o. In all three cases, NOPoutNG knows concretely during (static) linking that the functions are address-taken due to corresponding R_X86_64_REX_GOTPCRELX relocations present in Scrt1.o and a.o, emitted during compilation. Additionally, NOPoutNG opts to improve performance by maintaining endbrs in PLT entries (e.g., puts in ab) and disallowing access at the target function.

5.4. Dynamic Linking

The main component of NOPoutNG is a load-time feature that replaces nop instructions with endbrs in address-taken function prologues. Similar to the (static) linking phase, the dynamic linker/loader (ld.so) has access to additional context not available during compilation and (static) linking. Specifically, the dynamic linker loads a main executable and any shared libraries that are needed (i.e., marked DT_NEEDED in an ELF file's .dynamic section), and thus has access to all functions that are required (i.e., in the ELF .dynsym sections as an SHN_UNDEF, STT_FUNC symbol) as well as provided (i.e., a defined, STT_FUNC symbol) by the loaded binaries. NOPoutNG capitalizes on this by hooking the loading procedure to collect all symbols that are required and provided, before adding endbrs in all provided symbols that are required. Notably, this process is not stateless: NOPoutNG maintains metadata for each binary's required and provided symbols throughout the execution of a program to support dynamically-loaded shared libraries—i.e., those loaded with `dlopen` (§5.5).

5.4.1. Metadata Collection. `NOPoutNG` uses two main structures to represent: (1) libraries loaded into a process' address space, and (2), the symbols present in those libraries. During load-time, `NOPoutNG` allocates a library structure for each loaded library as well as a symbol structure for each of its `STT_FUNC` symbols.

Library Collection. `NOPoutNG` maintains a linked list of library structures that each represent a given library loaded into the process' address space. Each structure records: the name of library; addresses of relevant dynamic sections (i.e., those that are marked `SHF_ALLOC` in the ELF file) including the `.dynamic`, `.dynsym`, `.dynstr`, and `.gnu.hash` sections; the address range the library covers; and a hash table of all the library's provided and required symbols. Notably, we allocate symbol structures for required symbols in a library only if a provided symbol is not found previously parsed libraries; once the corresponding provided symbol is found, any relevant required symbol structures are replaced with references to the provided symbol structure.

Symbol Collection. For every symbol in a library's `.dynsym` section that is marked as `STT_FUNC` (i.e., a function), `NOPoutNG` creates a symbol structure for it that gets added to the library structure's provided or required hash table depending on if it is defined in that library or not, respectively. This symbol structure records the symbol's name, address, library, whether it is defined, and the number of references to the symbol. The reference counter for a given symbol gets increased, indicating it is required, if it is ever seen as undefined (`SHN_UNDEF`) across any of the loaded libraries, or if the function the symbol refers to begins with an `endbr` prior to loading—i.e., it is address-taken locally. This latter condition ensures that functions address-taken locally always have a reference count of at least 1, preventing the `endbr` from ever getting removed (e.g., during `dlclose`; see Section 5.5).

From Figure 2, `NOPoutNG` creates a library object for `libc.so` and populates its provided symbols hash table with an object representing `puts`, `read`, `write`, and all other exported functions. Having previously processed executable `ab`, `NOPoutNG` marks `puts` as being required by incrementing its reference count—the reference count of all other `libc.so` symbols *in the figure* remains 0, indicating that they should remain inaccessible to indirect branches.

5.4.2. Replacing `NOP`s. After parsing every binary loaded into a process' address space (still during load time), `NOPoutNG` exchanges the 4-byte `nop` instruction at the beginning of every required function's definition (assuming it is provided) with a 4-byte `endbr` instruction, allowing it to be targeted by an indirect branch. To do this, `NOPoutNG` first iterates through the linked list of library metadata structures previously collected. For each library, `NOPoutNG` performs an `mprotect` system call to mark the executable segment as writeable and enable patching the necessary points in the code. Notably, since this procedure works at load-time it is resistant to exploitation (i.e., the process has not yet received any input), an assumption used by similar mechanisms such as `RELRO` (relocation read-only; `-z relro` or `-Wl,-z,relro`) [114].

While the library's code is set as writeable, `NOPoutNG` iterates through its provided symbols, replacing `nop` instructions with `endbr` instructions if the reference count of the provided symbol is > 0 (i.e., at least one function requires it). If a provided symbol already has an `endbr` (indicating it is address-taken locally), or no symbol requires it, `NOPoutNG` continues to the next symbol. For the latter case, this means the provided symbol already begins with a 4-byte `nop` instruction and *cannot* be targeted by an indirect branch. After every provided symbol in a library has been considered, there are no further changes needed, and the library's executable segment permissions are returned to read, execute (`R-X`) via another `mprotect` call.

In Figure 2, the executable `ab` requires `puts`, defined in `libc.so` and external to `ab`. Since `libc.so` is used by many programs, all of its exported functions (that are not address-taken locally) begin with a 4-byte `nop` instruction. When `ab` is run and `libc.so` is loaded, `NOPoutNG` identifies that `puts` is required (i.e., address-taken) by `ab` and replaces the `nop` in the prologue of `puts` with an `endbr`. All other functions (`read`, `write`, *etc.*) are untouched, keeping their `nop`s and disallowing any indirect branch from targeting them. Importantly, this procedure `NOPoutNG` uses to reduce a program's attack surface by replacing previously inserted `nop` instructions with `endbr`s directly addresses Problem #1 (§3). The original NOPout work did the inverse, substituting `endbr` instructions for `nop`s at run time. Solely from this simple example, the efficiency of `NOPoutNG` can be seen: rather than adding `nop`s to almost every exported function in `libc.so`, `NOPoutNG` simply adds a single `endbr` to `puts`. Not only does this improve load-time performance due to fewer `nop`/`endbr` exchanges, but it also reduces the number of pages affected by COW.

5.5. Dynamic Loading

Loading a DSO into a process' address space dynamically *after* load time happens through `dlopen`. Typically, `dlopen` is used to load modules that augment or enhance a program's functionality; e.g., Nginx provides the ability to integrate custom, third-party modules [115]. The process of `dlopen`ing a DSO is nearly identical to loading a library at load time: the library is mapped into the process' address space and symbol resolution is performed. After a new library is `dlopen`ed, its symbols can be accessed using `dlsym`, which takes a handle to a loaded library, and a symbol name, and returns the symbol's address.

Importantly, `dlopen`ing a library brings new required (and provided) symbols into `NOPoutNG`'s analysis scope. Additionally, a successful `dlsym` for a function will *always* mark the function as required. As a result, `NOPoutNG` needs to handle both cases accordingly to maintain compatibility and ensure program execution does not break. For both dlopen and dlsym, NOPoutNG wraps each function, providing a new definition that is linked against, which adds logic around a call to the original function. For dlopen, a nearly identical process to NOPoutNG's standard library loading technique (§5.4) is performed; for dlsym, we mark the requested symbol as required (by increasing its reference count) and insert an endbr in its prologue. Notably, the logic for handling code patching at run time is more complex, as we can no longer assume an attacker cannot interfere with the process—e.g., tampering with code while (code) pages are writeable. To address this problem, NOPoutNG follows a *secure live-patching scheme* similar to NOPout; see Appendix E for details.

To compliment dlopen, there is a corresponding function, dlclose, which performs the inverse operations: i.e., unloads the specified library from a process' address space. Similar to dlopen, NOPoutNG wraps dlclose with additional logic that iterates through the library's required symbols, decrementing the reference count in their corresponding provided object. If the reference count for a provided symbol reaches 0 we perform secure live patching to overwrite the existing endbrs with 4-byte nops.

6. Implementation

Toolchain Modifications. We implemented the compiler and (static) linker components of NOPoutNG atop the LLVM toolchain (v12). Our compiler changes amounted to ≈100 C++ lines of code (LoC) primarily in the X86IndirectBranchTrackingPass class that operates as a MachineFunctionPass [116] in the compiler's backend to avoid any optimizations from mangling endbr and nop instructions that we insert. The linker changes that we applied to lld amount to ≈100 C++ LoC and interpose on writing the output file to add endbr instructions based on available relocation information. Importantly, while we implement NOPoutNG in the LLVM toolchain, no specific feature of LLVM is required to make NOPoutNG work; NOPoutNG can be easily ported to GCC and GNU Binutils or other toolchains.

Runtime Library. We implemented NOPoutNG's runtime component as a self-contained (i.e., without dependencies to other DSOs, like libc.so) shared library, libnopoutng.so, which can be LD_PRELOADed into a process' address space where a constructor will then perform metadata collection (§5.4.1) and nop replacement (§5.4.2). In addition, libnopoutng.so hooks dlopen and dlsym by preempting their resolution to libc's implementation. To prevent infinite recursion when calling the "real" dlopen/dlsym from the hook, we save their real addresses while NOPoutNG iterates through libc's symbol table. In total, libnopoutng.so was implemented in ≈1250 C LoC, with a file-system footprint of ≈42KB and a memory footprint of ≈19KB. Notably, while we could fully integrate the functionality of libnopoutng.so into the dynamic linker, ld.so, we opted to keep it a discrete runtime library for testing against other dynamic linkers.

TABLE 1. NOPoutNG PERFORMANCE OVERHEAD RESULTS.

	Benchmark	IBT	IBT + NOPoutNG
Real-world	Nginx (1KB)	0.07%	0.07%
	Nginx (100KB)	0.25%	0.25%
	Nginx (1MB)	0.30%	1.56%
	Redis (GET)	≈0%	≈0%
	Redis (SET)	≈0%	≈0%
	SQLite	≈0%	0.06%
SPEC CPU 2017	600.perlbench	5.03%	5.03%
	602.gcc	0.77%	0.77%
	605.mcf	0.06%	0.06%
	625.x264	0.08%	0.15%
	657.xz	0.43%	0.43%
	619.lbm	≈0%	3.67%
	638.imagick	0.08%	0.08%
	644.nab	0.45%	0.45%

7. Evaluation

Testbed. We performed all of our experiments on a host armed with an 11th generation Intel Core i7-11800H CPU, which has support for CET/IBT. Our host is also equipped with 16GB of DDR4 RAM, running Debian v11 with Linux kernel v5.13 (patched to support IBT [117]). Importantly, we instrumented and hardened *all* benchmarked applications, including their dependencies (i.e., any DSOs they link with), with IBT and NOPoutNG. Further, all programs were built as position independent (i.e., compiled with the -f{PIC, PIE} flags and linked with the -pie or -shared flags), with -O2 optimization level, and with full RELRO, which includes eager binding (i.e., linked with -z relro -z now or -Wl,-z,relro,-z,now). Unfortunately, due to glibc's considerable use of GCC- and GNU-specific features [118], it cannot be built with Clang/LLVM v12; thus, all benchmarked applications are (dynamically) linked with musl libc [119]. Finally, when running benchmarks, the host was run in a "quieted" single-user mode to limit noise. We also disabled any dynamic frequency and voltage scaling (DVFS) and fixed the CPU's frequency to constantly run at 2.3GHz in the C0 C-state to promote result reproducibility.

7.1. Performance

7.1.1. SPEC CPU 2017. We used the 8 C benchmarks from the SPEC CPU 2017 [120] SPECspeed Integer and Floating Point suites to evaluate the runtime slowdown of IBT and NOPoutNG compared to an uninstrumented baseline. Notably, SPEC CPU 2017 measures program initialization (i.e., everything that happens before the main function executes) as well as the program's characteristic functionality, making it an ideal benchmark to measure the load-time additions introduced by NOPoutNG. Table 1 presents the results of the 8 C benchmarks (col. 1) as percentage change atop the uninstrumented baseline when IBT is applied (col. 2) and when NOPoutNG (plus IBT) is applied (col. 3). The results are averaged over 5 runs of the ref workload—double the required amount for a "reportable" run [121]. (≈0% corresponds to $< 0.01\%$ overhead).

From Table 1: IBT overhead is minimal, ranging from ≈0%–5.03%. NOPoutNG adds negligible overhead to IBT, except for a ≈3.67% increase in 619.lbm.

7.1.2. Real-world Applications. We also evaluated the performance of NOPoutNG on three real-world applications: a web server (Nginx v1.28.0 [115]), an in-memory key–value store (Redis v7.0.15 [122]), and a relational database (SQLite v3.49.2 [123]). Our results for IBT and NOPoutNG are shown in Table 1, as percentages atop a uninstrumented baseline. (≈0% corresponds to $< 0.01\%$ overhead.)

Nginx. We used the wrk [124] benchmarking tool to generate HTTP requests to drive Nginx's execution. We performed 10 benchmark iterations for each variant (vanilla, IBT, and NOPoutNG) for 3 file sizes (filled with random data): 1KB, 100KB, and 1MB. In each iteration, wrk continuously sent HTTP requests to Nginx locally, over the loopback (lo) interface for 1 minute. For the 1KB file size, wrk used 4 threads, each making 1024 simultaneous HTTP requests; for the 100KB and 1MB file sizes, wrk used 8 threads, each making 512 connections. To service incoming requests, Nginx was configured to use 8 worker threads in all cases. We chose these settings to saturate the CPU (i.e., hit maximum utilization) so that any overhead from IBT or NOPoutNG is not masked by I/O. As shown in Table 1, the effects of IBT and NOPoutNG on Nginx are minimal, peaking at 1.56% for the 1MB file size for NOPoutNG.

Redis. To drive Redis, we used a high-throughput traffic generator, memtier_benchmark [125], which sends GET and SET requests to a given Redis instance. Using memtier_benchmark we perform 10 benchmark iterations for each variant, where each iteration sends GET and SET requests continuously to Redis for a 32-byte object in a $10 : 1$ ratio for 1 minute. We configured memtier_benchmark to use 2 execution threads, each making up to 32 simultaneous requests; Redis was configured to use a single worker thread. We determined these settings maximized the CPU's utilization during the benchmark. Further, all network I/O was performed over the loopback device to minimize I/O latency and increase CPU utilization. From Table 1, it can be seen that applying IBT and NOPoutNG results in no performance consequences.

SQLite. To evaluate SQLite, we use speedtest [126]: a built-in benchmarking tool for SQLite that is the result of (statically) linking SQLite's application code with a benchmark driver into a single executable. Each invocation of speedtest measures the time taken to complete a series of common relational database operations (CREATE TABLE, SELECT, *etc.*). Notably, to ensure that we only measure the overhead of the SQLite application and not the benchmark driver, we omit IBT instrumentation from the benchmark driver during compilation by adding a nocf_check attribute to each of the driver's functions. Further, we configure speedtest to use an in-memory database to prevent disk I/O from masking any overhead; all other configuration options were left as their default value. The overhead averaged over 10 iterations is negligible.

7.1.3. Dynamic Loading Microbenchmark. We crafted a microbenchmark to evaluate the overhead incurred by NOPoutNG's handling of dynamic library loading. The microbenchmark measures: dlopening zlib [127], dlsyming (de)compression and checksum (CRC-32) functions, and running the functions on data sizes derived from the Calgary corpus [128]. This mimics common compression library use cases; e.g., a compression library is dynamically chosen for checksumming and (de)compressing data for storage on disk. For a full description of the microbenchmark, refer to Appendix F. Comparing NOPoutNG to a vanilla baseline across 100 iterations yields an average overhead of 13.23%.

TABLE 2. NOPoutNG EFFECTIVENESS RESULTS.

Application	# endbr IBT	# endbr NOPoutNG	NOPout +nop	NOPoutNG +endbr	NOPoutNG Pages
Nginx	3244	1466 (-54.47%)	1602 (92.44%)	131 (7.56%)	42 (172KB)
Redis	5413	3600 (-33.49%)	1656 (91.34%)	157 (8.66%)	53 (217KB)
SQLite	2366	1111 (-53.04%)	1101 (94.67%)	62 (5.33%)	25 (102KB)
600.perlbench	3758	1341 (-64.32%)	1089 (93.56%)	75 (6.44%)	27 (111KB)
602.gcc	11512	4505 (-60.87%)	1115 (95.46%)	53 (4.54%)	20 (82KB)
605.mcf	1746	403 (-76.92%)	1156 (99.40%)	7 (0.60%)	5 (20KB)
625.x264	2141	646 (-69.83%)	1158 (99.40%)	7 (0.60%)	4 (16KB)
657.xz	2057	519 (-74.77%)	1150 (98.88%)	13 (1.12%)	9 (37KB)
619.lbm	1726	397 (-77.00%)	1158 (99.57%)	5 (0.43%)	4 (16KB)
638.imagick	3639	495 (-86.40%)	1156 (99.40%)	7 (0.60%)	4 (16KB)
644.nab	1891	429 (-77.31%)	1138 (97.51%)	29 (2.49%)	18 (74KB)

7.2. Effectiveness

endbr Reduction. The primary metric to evaluate the effectiveness of NOPout and NOPoutNG is endbr reduction; the fewer endbr a program has, the fewer available "landing pads" an indirect branch can target, reducing the overall attack-surface of the program. Notably, NOPout and NOPoutNG should both achieve a similar level of endbr reduction when compared with vanilla IBT, which is shown for NOPoutNG in Table 2 (col. 2–3). These columns present the number of functions that begin with endbrs *at run time* for each application and its (shared library) dependencies. As can be seen, NOPoutNG exhibits a substantial reduction in endbrs—and thus attack surface—compared to vanilla IBT, ranging from 33.49%–86.40%.

Comparison with NOPout. While similarly effective, NOPout elides endbrs at run time while NOPoutNG inserts them. Beyond the performance angle, NOPout's approach also reduces the effectiveness—or "shareability"—of shared libraries due to COW. As shown in Table 2 (col. 4–6), removing endbrs at run time will require patching many more code pages than NOPoutNG's approach of adding endbrs. Each patched code page requires COW, slowing startup and increasing the system's overall memory needs.

8. Conclusion

In this paper, we addressed the problems in NOPout through NOPoutNG, a toolchain-assisted, load-time mechanism that hardens programs by minimizing the number of available endbr "landing pads." The core of NOPoutNG is its secure-by-default design philosophy: it begins with inert nop instructions in exported function prologues and only promotes them to "active" endbr targets when they are demonstrably used by the application. Our evaluation shows that NOPoutNG is both effective and practical, successfully reducing attack surface with negligible runtime overhead.

Availability

`NOPoutNG` is available at:
https://gitlab.com/brown-ssl/nopoutng

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This work was supported by the National Science Foundation (NSF), through award CNS-2238467. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the US government or NSF.

References

[1] A. Quach, A. Prakash, and L. Yan, "Debloating Software through Piece-Wise Compilation and Loading," in *USENIX Security Symposium (SEC)*, 2018, pp. 869–886.

[2] G. J. Holzmann, "Code Inflation," 2015. [Online]. Available: https://spinroot.com/gerard/pdf/Code_Inflation.pdf

[3] A. Quach, R. Erinfolami, D. Demicco, and A. Prakash, "A Multi-OS Cross-Layer Study of Bloating in User Programs, Kernel and Managed Execution Environments," in *ACM Workshop on Forming an Ecosystem Around Software Transformation (FEAST)*, 2017, pp. 65–70.

[4] C. Soto-Valero, N. Harrand, M. Monperrus, and B. Baudry, "A Comprehensive Study of Bloated Dependencies in the Maven Ecosystem," *Empirical Software Engineering (EMSE)*, vol. 26, no. 3, p. 45, 2021.

[5] R. Pike and B. Kernighan, "Program Design in the UNIX Environment," *AT&T Bell Laboratories Technical Journal*, vol. 63, no. 8, pp. 1595–1605, 1984.

[6] V. v. d. Veen, L. Cavallaro, H. Bos *et al.*, "Memory Errors: The Past, the Present, and the Future," in *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2012, pp. 86–106.

[7] SANS Institute, "CWE/SANS TOP 25 Most Dangerous Software Errors," https://www.sans.org/top25-software-errors/, 2023.

[8] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal War in Memory," in *IEEE Symposium on Security and Privacy (S&P)*, 2013, pp. 48–62.

[9] J. Xu, D. Mu, P. Chen, X. Xing, P. Wang, and P. Liu, "Credal: Towards Locating a Memory Corruption Vulnerability with Your Core Dump," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016, pp. 529–540.

[10] Microsoft Security Response Center, "A proactive approach to more secure code," https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/, 2019.

[11] The Chromium Projects, "Memory safety," https://www.chromium.org/Home/chromium-security/memory-safety/, 2023.

[12] Google Security Blog, "Mitigating Memory Safety Issues in Open Source Software," https://security.googleblog.com/2021/02/mitigating-memory-safety-issues-in-open.html, 2021.

[13] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "SoftBound: Highly Compatible and Complete Spatial Memory Safety for C," in *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2009, pp. 245–258.

[14] ——, "CETS: Compiler Enforced Temporal Safety for C," in *International Symposium on Memory Management (ISMM)*, 2010, pp. 31–40.

[15] V. Kuznetsov, L. Szekeres, M. Payer, G. C. nd R. Sekar, and D. Song, "Code-Pointer Integrity," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014, pp. 147–163.

[16] A. One, "Smashing The Stack For Fun And Profit," *Phrack Magazine*, vol. 7, no. 49, 1996.

[17] H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86)," in *ACM Conference on Computer and Communications Security (CCS)*, 2007, pp. 552–561.

[18] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-Oriented Programming without Returns," in *ACM Conference on Computer and Communications Security (CCS)*, 2010, pp. 559–572.

[19] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-Oriented Programming: A New Class of Code-Reuse Attack," in *ACM Asia Symposium on Information, Computer and Communications Security (ASIACCS)*, 2011, pp. 30–40.

[20] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out Of Control: Overcoming Control-Flow Integrity," in *IEEE Symposium on Security and Privacy (S&P)*, 2014, pp. 575–589.

[21] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization," in *IEEE Symposium on Security and Privacy (S&P)*, 2013, pp. 574–588.

[22] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications," in *IEEE Symposium on Security and Privacy (S&P)*, 2015, pp. 745–762.

[23] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-Flow Integrity: Precision, Security, and Performance," *ACM Computing Surveys (CSUR)*, vol. 50, no. 1, pp. 1–33, 2017.

[24] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-Flow Integrity," in *ACM Conference on Computer and Communications Security (CCS)*, 2005, p. 340–353.

[25] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, "XFI: Software Guards for System Address Spaces," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006, pp. 75–88.

[26] S. McCamant and G. Morrisett, "Evaluating SFI for a CISC Architecture," in *USENIX Security Symposium (SEC)*, 2006, pp. 209–224.

[27] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen, "Adapting Software Fault Isolation to Contemporary CPU Architectures," in *USENIX Security Symposium (SEC)*, 2010, pp. 1–11.

[28] Z. Wang and X. Jiang, "HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity," in *IEEE Symposium on Security and Privacy (S&P)*, 2010, pp. 380–395.

[29] T. Bletsch, X. Jiang, and V. Freeh, "Mitigating Code-Reuse Attacks with Control-Flow Locking," in *Annual Computer Security Applications Conference (ACSAC)*, 2011, pp. 353–362.

[30] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.-R. Sadeghi, "MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones," in *Network and Distributed System Security Symposium (NDSS)*, 2012.

[31] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis, "kGuard: Lightweight Kernel Protection against Return-to-User Attacks," in *USENIX Security Symposium (SEC)*, 2012, pp. 459–474.

[32] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical Control Flow Integrity and Randomization for Binary Executables," in *IEEE Symposium on Security and Privacy (S&P)*, 2013, pp. 559–573.

[33] M. Zhang and R. Sekar, "Control Flow Integrity for COTS Binaries," in *USENIX Security Symposium (SEC)*, 2013, pp. 337–352.

[34] B. Niu and G. Tan, "Monitor Integrity Protection with Space Efficiency and Separate Compilation," in *ACM Conference on Computer and Communications Security (CCS)*, 2013, pp. 199–210.

[35] J. Pewny and T. Holz, "Control-flow Restrictor: Compiler-based CFI for iOS," in *Annual Computer Security Applications Conference (ACSAC)*, 2013, pp. 309–318.

[36] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, and R. H. Deng, "ROPecker: A Generic and Practical Approach for Defending against ROP Attack," in *Network and Distributed System Security Symposium (NDSS)*, 2014.

[37] D. Jang, Z. Tatlock, and S. Lerner, "SafeDispatch: Securing C++ Virtual Calls from Memory Corruption Attacks," in *Network and Distributed System Security Symposium (NDSS)*, 2014.

[38] J. Criswell, N. Dautenhahn, and V. Adve, "KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels," in *IEEE Symposium on Security and Privacy (S&P)*, 2014, pp. 292–307.

[39] B. Niu and G. Tan, "Modular Control-Flow Integrity," in *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2014, pp. 577–587.

[40] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM," in *USENIX Security Symposium (SEC)*, 2014, pp. 941–955.

[41] B. Niu and G. Tan, "RockJIT: Securing Just-In-Time Compilation Using Modular Control-Flow Integrity," in *ACM Conference on Computer and Communications Security (CCS)*, 2014, pp. 1317–1328.

[42] R. Gawlik and T. Holz, "Towards Automated Integrity Protection of C++ Virtual Function Tables in Binary Programs," in *Annual Computer Security Applications Conference (ACSAC)*, 2014, pp. 396–405.

[43] C. Zhang, C. Song, K. Z. Chen, Z. Chen, and D. Song, "VTint: Protecting Virtual Function Tables' Integrity," in *Network and Distributed System Security Symposium (NDSS)*, 2015.

[44] A. Prakash, X. Hu, and H. Yin, "vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries," in *Network and Distributed System Security Symposium (NDSS)*, 2015.

[45] M. Payer, A. Barresi, and T. R. Gross, "Fine-Grained Control-Flow Integrity Through Binary Hardening," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2015, p. 144–164.

[46] P. Team, "RAP: RIP ROP," in *Hackers 2 Hackers Conference (H2HC)*, 2015.

[47] B. Niu and G. Tan, "Per-Input Control-Flow Integrity," in *ACM Conference on Computer and Communications Security (CCS)*, 2015, pp. 914–926.

[48] M. Wang, H. Yin, A. V. Bhaskar, P. Su, and D. Feng, "Binary Code Continent: Finer-Grained Control Flow Integrity for Stripped Binaries," in *Annual Computer Security Applications Conference (ACSAC)*, 2015, pp. 331–340.

[49] I. Haller, E. Göktaş, E. Athanasopoulos, G. Portokalidis, and H. Bos, "ShrinkWrap: VTable Protection without Loose Ends," in *Annual Computer Security Applications Conference (ACSAC)*, 2015, pp. 341–350.

[50] D. Bounov, R. G. Kici, and S. Lerner, "Protecting C++ Dynamic Dispatch Through VTable Interleaving," in *Network and Distributed System Security Symposium (NDSS)*, 2016.

[51] C. Zhang, D. Song, S. A. Carr, M. Payer, T. Li, Y. Ding, and C. Song, "VTrust: Regaining Trust on Virtual Calls," in *Network and Distributed System Security Symposium (NDSS)*, 2016.

[52] X. Ge, N. Talele, M. Payer, and T. Jaeger, "Fine-Grained Control-Flow Integrity for Kernel Software," in *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2016, pp. 179–194.

[53] V. Van Der Veen, E. Göktas, M. Contag, A. Pawoloski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida, "A Tough `call`: Mitigating Advanced Code-Reuse Attacks at the Binary Level," in *IEEE Symposium on Security and Privacy (S&P)*, 2016, pp. 934–953.

[54] A. Pawlowski, M. Contag, V. van der Veen, C. Ouwehand, T. Holz, H. Bos, E. Athanasopoulos, and C. Giuffrida, "MARX: Uncovering Class Hierarchies in C++ Programs," in *Network and Distributed System Security Symposium (NDSS)*, 2017.

[55] M. Elsabagh, D. Fleck, and A. Stavrou, "Strict Virtual Call Integrity Checking for C++ Binaries," in *ACM Asia Symposium on Information, Computer and Communications Security (ASIACCS)*, 2017, pp. 140–154.

[56] J. Moreira, S. Rigo, M. Polychronakis, and V. P. Kemerlis, "DROP THE ROP: Fine-grained Control-flow Integrity for the Linux Kernel," *Black Hat Asia (BHASIA)*, 2017.

[57] W. Wang, X. Xu, and K. W. Hamlen, "Object Flow Integrity," in *ACM Conference on Computer and Communications Security (CCS)*, 2017, pp. 1909–1924.

[58] N. Burow, D. McKee, S. A. Carr, and M. Payer, "CFIXX: Object Type Integrity for C++ Virtual Dispatch," in *Network and Distributed System Security Symposium (NDSS)*, 2018.

[59] J. Grossklags and C. Eckert, "τCFI: Type-Assisted Control Flow Integrity for x86-64 Binaries," in *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2018.

[60] A. Pawlowski, V. van der Veen, D. Andriesse, E. van der Kouwe, T. Holz, C. Giuffrida, and H. Bos, "VPS: Excavating High-Level C++ Constructs from Low-Level Binaries to protect Dynamic Dispatching," in *Annual Computer Security Applications Conference (ACSAC)*, 2019, pp. 97–112.

[61] K. Lu and H. Hu, "Where Does It Go? Refining Indirect-Call Targets with Multi-Layer Type Analysis," in *ACM Conference on Computer and Communications Security (CCS)*, 2019, pp. 1867–1881.

[62] V. Duta, C. Giuffrida, H. Bos, and E. Van Der Kouwe, "PIBE: Practical Kernel Control-Flow Hardening with Profile-Guided Indirect Branch Elimination," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021, pp. 743–757.

[63] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native Client: A Sandbox for Portable, Untrusted x86 Native Code," in *IEEE Symposium on Security and Privacy (S&P)*, 2009, pp. 79–93.

[64] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent ROP Exploit Mitigation using Indirect Branch Tracing," in *USENIX Security Symposium (SEC)*, 2013, pp. 447–462.

[65] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz, "Opaque Control-Flow Integrity," in *Network and Distributed System Security Symposium (NDSS)*, 2015.

[66] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières, "CCFI: Cryptographically Enforced Control Flow Integrity," in *ACM Conference on Computer and Communications Security (CCS)*, 2015, pp. 941–951.

[67] V. Van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, "Practical Context-Sensitive CFI," in *ACM Conference on Computer and Communications Security (CCS)*, 2015, pp. 927–940.

[68] T. Abera, N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik, "C-FLAT: Control-Flow Attestation for Embedded Systems Software," in *ACM Conference on Computer and Communications Security (CCS)*, 2016, pp. 743–754.

[69] Y. Gu, Q. Zhao, Y. Zhang, and Z. Lin, "PT-CFI: Transparent Backward-Edge Control Flow Violation Detection using Intel Processor Trace," in *ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2017, pp. 173–184.

[70] X. Ge, W. Cui, and T. Jaeger, "GRIFFIN: Guarding Control Flows using Intel Processor Trace," *ACM SIGPLAN Notices*, vol. 52, no. 4, pp. 585–598, 2017.

[71] R. Ding, C. Qian, C. Song, B. Harris, T. Kim, and W. Lee, "Efficient Protection of Path-Sensitive Control Security," in *USENIX Security Symposium (SEC)*, 2017, pp. 131–148.

[72] T. Nyman, J.-E. Ekberg, L. Davi, and N. Asokan, "CFI CaRE: Hardware-supported Call and Return Enforcement for Commercial Microcontrollers," in *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2017, pp. 259–284.

[73] A. Abbasi, T. Holz, E. Zambon, and S. Etalle, "ECFI: Asynchronous Control Flow Integrity for Programmable Logic Controllers," in *Annual Computer Security Applications Conference (ACSAC)*, 2017, pp. 437–448.

[74] J. Li, X. Tong, F. Zhang, and J. Ma, "Fine-CFI: Fine-Grained Control-Flow Integrity for Operating System Kernels," *IEEE Transactions on Information Forensics and Security (TIFS)*, vol. 13, no. 6, pp. 1535–1550, 2018.

[75] J. Zhang, B. Qi, Z. Qin, and G. Qu, "HCIC: Hardware-Assisted Control-Flow Integrity Checking," *IEEE Internet of Things Journal*, vol. 6, no. 1, pp. 458–471, 2018.

[76] D. Kwon, J. Seo, S. Baek, G. Kim, S. Ahn, and Y. Paek, "VM-CFI: Control-Flow Integrity for Virtual Machine Kernel Using Intel PT," in *International Conference on Computational Science and Its Applications (ICCSA)*, 2018, pp. 127–137.

[77] H. Hu, C. Qian, C. Yagemann, S. P. H. Chung, W. R. Harris, T. Kim, and W. Lee, "Enforcing Unique Code Target Property for Control-Flow Integrity," in *ACM Conference on Computer and Communications Security (CCS)*, 2018, pp. 1470–1486.

[78] R. J. Walls, N. F. Brown, T. Le Baron, C. A. Shue, H. Okhravi, and B. C. Ward, "Control-Flow Integrity for Real-Time Embedded Systems," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2019.

[79] N. S. Almakhdhub, A. A. Clements, S. Bagchi, and M. Payer, "μRAI: Securing Embedded Systems with Return Address Integrity," in *Network and Distributed System Security Symposium (NDSS)*, 2020.

[80] L. Davi, P. Koeberl, and A.-R. Sadeghi, "Hardware-Assisted Fine-Grained Control-Flow Integrity: Towards Efficient Protection of Embedded Systems Against Software Exploitation," in *Design Automation Conference (DAC)*, 2014.

[81] L. Davi, M. Hanreich, D. Paul, A.-R. Sadeghi, P. Koeberl, D. Sullivan, O. Arias, and Y. Jin, "HAFIX: Hardware-Assisted Flow Integrity Extension," in *Design Automation Conference (DAC)*, 2015.

[82] P. Yuan, Q. Zeng, and X. Ding, "Hardware-Assisted Fine-Grained Code-Reuse Attack Detection," in *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2015, pp. 66–85.

[83] J. Zhang, R. Hou, J. Fan, K. Liu, L. Zhang, and S. A. McKee, "RAGuard: A Hardware Based Mechanism for Backward-Edge Control-Flow Integrity," in *ACM International Conference on Computing Frontiers (CF)*, 2017, pp. 27–34.

[84] M. D. S. R. Team, "Analysis of the Shadow Brokers release and mitigation with Windows 10 virtualization-based security," https://www.microsoft.com/en-us/security/blog/2017/06/16/analysis-of-the-shadow-brokers-release-and-mitigation-with-windows-10-virtualization-based-security/?source=mmpc, 2017.

[85] Android Open Source Project, "Control Flow Integrity," https://source.android.com/docs/security/test/cfi, 2022.

[86] The Chromium Projects, "Control Flow Integrity," https://www.chromium.org/developers/testing/control-flow-integrity/, 2023.

[87] I. Corporation, *Intel 64 and IA-32 Architectures Software Developer's Manual*, 2021.

[88] ARM Limited, *ARM A64 Instruction Set Architecture – Branch Target Identification*, 2020.

[89] V. Shanbhogue, D. Gupta, and R. Sahita, "Security Analysis of Processor Instruction Set Architecture for Enforcing Control-Flow Integrity," in *International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2019.

[90] I. Corporation, *Control-flow Enforcement Technology Specification*, 2019.

[91] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-Flow Bending: On the Effectiveness of Control-Flow Integrity," in *USENIX Security Symposium (SEC)*, 2015, pp. 161–176.

[92] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, "Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity," in *ACM Conference on Computer and Communications Security (CCS)*, 2015, pp. 901–913.

[93] A. J. Gaidis, J. Moreira, K. Sun, A. Milburn, V. Atlidakis, and V. P. Kemerlis, "FineIBT: Fine-Grain Control-Flow Enforcement with Indirect Branch Tracking," in *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2023, pp. 527–546.

[94] T.-C. Chiueh and F.-H. Hsu, "RAD: a compile-time solution to buffer overflow attacks," in *International Conference on Distributed Computing Systems (ICDCS)*, 2001, pp. 409–417.

[95] N. Burow, X. Zhang, and M. Payer, "SoK: Shining Light on Shadow Stacks," in *IEEE Symposium on Security and Privacy (S&P)*, 2019, pp. 985–999.

[96] Bugtraq, "Getting around non-executable stack (and fix)," https://seclists.org/bugtraq/1997/Aug/63, 1997.

[97] J. Corbet, "Kernel release status," https://lwn.net/Articles/924113/, 2023.

[98] H. J. Lu, "Control-Flow Enforcement Technology," in *Linux Plumbers Conference (LPC)*, 2018.

[99] H. J. Lu *et al.*, "x86: Support Intel IBT with IBT property and IBT-enable PLT," Patch to GNU Binutils mailing list, 2017. [Online]. Available: https://sourceware.org/legacy-ml/binutils/2017-06/msg00285.html

[100] J. Corbet, "Indirect branch tracking for intel cpus," *LWN.net*, 2018. [Online]. Available: https://lwn.net/Articles/889475/

[101] T. G. C. L. project, "Intel CET support in glibc: '–enable-cet' (IBT + shadow stack)," glibc patch / NEWS entry, 2018. [Online]. Available: https://sourceware.org/legacy-ml/libc-alpha/2018-07/msg00550.html

[102] The Linux Foundation, "Linux 6.2 Released," Linux Kernel release notes / KernelNewbies, 2023. [Online]. Available: https://kernelnewbies.org/Linux_6.2

[103] N. Christou, A. J. Gaidis, V. Atlidakis, and V. P. Kemerlis, "Eclipse: Preventing Speculative Memory-error Abuse with Artificial Data Dependencies," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2024, pp. 3913–3927.

[104] Clang 17.0.0git documentation, "Control Flow Integrity," https://clang.llvm.org/docs/ControlFlowIntegrity.html, 2023.

[105] Microsoft technical documentation, "Control Flow Guard for platform security," https://docs.microsoft.com/en-us/windows/win32/secbp/control-flow-guard, 2022.

[106] J. Corbet, "x86 NX support," https://lwn.net/Articles/87814/, 2004.

[107] OpenBSD, "i386 W^X," https://marc.info/?l=openbsd-misc&m=105056000801065, 2003.

[108] S. Forrest, A. Somayaji, and D. H. Ackley, "Building Diverse Computer Systems," in *Workshop on Hot Topics in Operating Systems (HotOS)*, 1997, pp. 67–72.

[109] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," in *USENIX Security Symposium (SEC)*, vol. 98, 1998, pp. 63–78.

[110] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "SoK: Automated Software Diversity," in *IEEE Symposium on Security and Privacy (S&P)*, 2014, pp. 276–291.

[111] D. Williams-King, G. Gobieski, K. Williams-King, J. P. Blake, X. Yuan, P. Colp, M. Zheng, V. P. Kemerlis, J. Yang, and W. Aiello, "Shuffler: Fast and Deployable Continuous Code Re-Randomization," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016, pp. 367–382.

[112] H. Koo, Y. Chen, L. Lu, V. P. Kemerlis, and M. Polychronakis, "Compiler-assisted Code Randomization," in *IEEE Symposium on Security and Privacy (S&P)*, 2018, pp. 461–477.

[113] S. Proskurin, M. Momeu, S. Ghavamnia, V. P. Kemerlis, and M. Polychronakis, "xMP: Selective Memory Protection for Kernel and User Space," in *IEEE Symposium on Security and Privacy (S&P)*, 2020, pp. 563–577.

[114] Red Hat Blog – Huzaifa Sidhpurwala, "Security Technologies: RELRO," https://www.redhat.com/en/blog/hardening-elf-binaries-using-relocation-read-only-relro, 2019.

[115] "nginx," https://nginx.org, 2023.

[116] LLVM Project, "Machinefunctionpass class reference," https://llvm.org/doxygen/classllvm_1_1MachineFunctionPass.html, 2025.

[117] Intel Corporation, "Linux Intel Quilt," https://github.com/intel/linux-intel-quilt/tree/mainline-tracking-v5.13-yocto-210727T062416Z, 2021.

[118] MaskRay, "When can glibc be built with Clang?" https://maskray.me/blog/2021-10-10-when-can-glibc-be-built-with-clang, 2021.

[119] "musl libc," https://musl.libc.org, 2023.

[120] J. Bucek, K.-D. Lange, and J. v. Kistowski, "SPEC CPU2017: Next-generation Compute Benchmark," in *ACM/SPEC International Conference on Performance Engineering (ICPE)*, 2018, pp. 41–42.

[121] Standard Performance Evaluation Corporation, "Using SPEC CPU 2017: the 'runcpu' Command," https://www.spec.org/cpu2017/Docs/runcpu.html#size, 2021.

[122] "Redis," https://redis.io, 2023.

[123] "SQLite," https://www.sqlite.org, 2023.

[124] "wrk – a HTTP benchmarking tool," https://github.com/wg/wrk, 2021.

[125] Redis, "memtier_benchmark," https://github.com/RedisLabs/memtier{_}benchmark, 2023.

[126] SQLite, "Database Speed Comparison," https://www.sqlite.com/speed.html, 2023.

[127] "zlib," https://www.zlib.net, 2025.

[128] T. Bell, I. H. Witten, and J. G. Cleary, "Modeling for Text Compression," *ACM Computing Surveys (CSUR)*, vol. 21, no. 4, pp. 557–591, 1989.

[129] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering Code-Injection Attacks With Instruction-Set Randomization," in *ACM Conference on Computer and Communications Security (CCS)*, 2003, pp. 272–280.

[130] Microsoft Docs, "Data Execution Prevention," https://docs.microsoft.com/en-us/, 2022.

[131] Wikipedia, "NX bit," https://en.wikipedia.org/w/index.php?title=NX_bit&oldid=1309719290, 2023, accessed 2025-11-06.

[132] Virus Bulletin, "Code injection via return-oriented programming," https://www.virusbulletin.com/virusbulletin/2012/10/code-injection-return-oriented-programming, 2012.

[133] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking Blind," in *IEEE Symposium on Security and Privacy (S&P)*, 2014, pp. 227–242.

[134] E. Bosman and H. Bos, "Framing Signals—A Return to Portable Shellcode," in *IEEE Symposium on Security and Privacy (S&P)*, 2014, pp. 243–258.

[135] R. Gawlik, B. Kollenda, P. Koppe, B. Garmany, and T. Holz, "Enabling Client-Side Crash-Resistance to Overcome Diversification and Information Hiding," in *Network and Distributed System Security Symposium (NDSS)*, 2016.

[136] R. Rudd, R. Skowyra, D. Bigelow, V. Dedhia, T. Hobson, S. Crane, C. Liebchen, P. Larsen, L. Davi, M. Franz, A.-R. Sadeghi, and H. Okhravi, "Address Oblivious Code Reuse: On the Effectiveness of Leakage Resilient Diversity," in *Network and Distributed System Security Symposium (NDSS)*, 2017.

[137] E. Göktas, B. Kollenda, P. Koppe, E. Bosman, G. Portokalidis, T. Holz, H. Bos, and C. Giuffrida, "Position-independent Code Reuse: On the Effectiveness of ASLR in the Absence of Information Disclosure," in *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2018, pp. 227–242.

[138] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006, pp. 147–160.

[139] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, "Preventing Memory Error Exploits with WIT," in *IEEE Symposium on Security and Privacy (S&P)*, 2008, pp. 263–277.

[140] S. Bhatkar and R. Sekar, "Data Space Randomization," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2008, pp. 1–22.

[141] P. Chen, J. Xu, Z. Lin, D. Xu, B. Mao, and P. Liu, "A Practical Approach for Adaptive Data Structure Layout Randomization," in *European Symposium on Research in Computer Security (ESORICS)*, 2015, pp. 69–89.

[142] K. Sinha, V. P. Kemerlis, and S. Sethumadhavan, "Reviving Instruction Set Randomization," in *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2017, pp. 21–28.

[143] P. Rajasekaran, S. Crane, D. Gens, Y. Na, S. Volckaert, and M. Franz, "CoDaRR: Continuous Data Space Randomization against Data-Only Attacks," in *ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2020, pp. 494–505.

[144] K. Koning, X. Chen, H. Bos, C. Giuffrida, and E. Athanasopoulos, "No Need to Hide: Protecting Safe Regions on Commodity Hardware," in *European Conference on Computer Systems (EuroSys)*, 2017, pp. 437–452.

[145] V. Kiriansky, D. Bruening, and S. Amarasinghe, "Secure Execution via Program Shepherding," in *USENIX Security Symposium (SEC)*, 2002.

[146] PaX Team, "What the future holds for PaX," https://pax.grsecurity.net/docs/pax-future.txt, 2003.

[147] M. Khandaker, A. Naser, W. Liu, Z. Wang, Y. Zhou, and Y. Cheng, "Adaptive Call-Site Sensitive Control Flow Integrity," in *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019, pp. 95–110.

[148] M. R. Khandaker, W. Liu, A. Naser, Z. Wang, and J. Yang, "Origin-sensitive Control Flow Integrity," in *USENIX Security Symposium (SEC)*, 2019, pp. 195–211.

[149] K. Kleftogiorgos, P. Zielinski, S. Huang, J. Xu, and G. Portokalidis, "Sidecar: Leveraging Debugging Extensions in Commodity Processors to Secure Software," in *Annual Computer Security Applications Conference (ACSAC)*, 2024, pp. 534–547.

[150] Erlingsson, Úlfar and Schneider, Fred B., "SASI Enforcement of Security Policies: A Retrospective," in *New Security Paradigms Workshop (NSPW)*, 1999, p. 87–95.

[151] U. Erlingsson and F. B. Schneider, "IRM Enforcement of Java Stack Inspection," in *IEEE Symposium on Security and Privacy (S&P)*, 2000, pp. 246–255.

[152] M. Werner, T. Unterluggauer, D. Schaffenrath, and S. Mangard, "Sponge-Based Control-Flow Protection for IoT Devices," in *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2018, pp. 214–226.

[153] T. H. Dang, P. Maniatis, and D. Wagner, "The Performance Cost of Shadow Stacks and Stack Canaries," in *ACM Asia Symposium on Information, Computer and Communications Security (ASIACCS)*, 2015, pp. 555–566.

[154] N. Carlini and D. Wagner, "ROP is Still Dangerous: Breaking Modern Defenses," in *USENIX Security Symposium (SEC)*, 2014, pp. 385–399.

[155] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection," in *USENIX Security Symposium (SEC)*, 2014, pp. 401–416.

[156] E. Göktaş, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis, "Size Does Matter: Why Using Gadget-Chain Length to Prevent Code-Reuse Attacks is Hard," in *USENIX Security Symposium (SEC)*, 2014, pp. 417–432.

[157] F. Kasten, P. Zieris, and J. Horsch, "Integrating Static Analyses for High-Precision Control-Flow Integrity," in *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2024, pp. 419–434.

[158] H. Xiang, Z. Cheng, J. Li, J. Ma, and K. Lu, "Boosting Practical Control-Flow Integrity with Complete Field Sensitivity and Origin Awareness," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2024, pp. 4524–4538.

[159] T. Xia, H. Hu, and D. Wu, "{DEEPTYPE}: Refining indirect call targets with strong multi-layer type analysis," in *USENIX Security Symposium (SEC)*, 2024, pp. 5877–5894.

[160] G. Li, M. Sridharan, and Z. Qian, "Redefining Indirect Call Analysis with KallGraph," in *IEEE Symposium on Security and Privacy (S&P)*, 2025, pp. 2957–2975.

[161] D. Liu, S. Ji, K. Lu, and Q. He, "Improving {Indirect-Call} analysis in {LLVM} with type and {Data-Flow}{Co-Analysis}," in *USENIX Security Symposium (SEC)*, 2024, pp. 5895–5912.

Appendix

1. Memory-error Exploits

Memory errors arise in software written in memory- and/or type-unsafe languages—e.g., C, C++, Objective-C, and assembly (ASM)—allowing an attacker to (ab)use memory within a victim program's (virtual) address space [6]. Typically, in real-world exploits, attackers aim to *corrupt* or *leak control data*, such as return addresses, function pointers, and dynamic dispatch tables (i.e., *code pointers* [15]), to hijack the *control-flow* of a program and ultimately achieve *arbitrary code execution* [16].

Originally, arbitrary code execution was achieved via *code injection* [129], where an attacker leverages the lack of fine-grained memory permissions to inject code as (writable and executable) data and subsequently drive control flow to it. To mitigate code injection, contemporary platforms [106], [130] introduced non-executable memory [131] and the corresponding W^X memory protection policy [107] to prevent the execution of data as code. Today, code injection is typically only found as a part of multi-stage exploits [132].

The current state-of-the-{art, practice} exploit technique for achieving arbitrary code execution is *code reuse* [96]: a technique where an attacker directs control-flow to pieces of existing, benign code in a victim process' address space, executing them "out-of-context" to perform arbitrary computation. There has been a plethora of work exploring different ways pieces of existing code can be (ab)used and strung together to perform code reuse, with popular examples being: ROP [17], JOP [18], [19], COP [20], JIT-ROP [21], BROP [133], SROP [134], COOP [22], CROP [135], AOCR [136], and PIROP [137].

2. Control-flow Integrity

To combat the torrent of code-reuse-based exploitation techniques, a diverse set of mitigations were developed [8] that can largely be grouped into four categories: Control-flow Integrity (CFI) [23], Data-flow Integrity (DFI) [138], [139], automated diversification [110]–[112], [140]–[143], and memory isolation [15], [113], [144].

Control-flow integrity was first introduced by Abadi et al. [24] and is a form of *program shepherding* [145], [146] at its core. At a high level, CFI confines a program's execution to "legal" paths typically identified a priori. Notably, CFI does not attempt to mitigate memory errors; it aims to stop their exploitation when (ab)used to hijack a program's control flow. Since the foundational work of Abadi et al., a myriad of CFI schemes have been developed which can be roughly classified into *software-based* [24]–[62], *hardware-assisted* [63]–[79], [93], [147]–[149], and *hardware-only* [80]–[83] schemes. Despite covering different points in the CFI design space, the majority of CFI schemes share a commonality: a control-flow *enforcement mechanism* that adjudicates on whether control-flow transfers are allowed by referring to a pre-constructed control-flow graph (CFG). In essence, CFI enforcement mechanisms act as *Inlined Reference Monitors* (IRMs) [150], [151] that enforce policies related to the target(s) of indirect branch instructions. CFI schemes can be further classified according to three criteria: *coverage*, *granularity*, and *compatibility*.

Coverage. Indirect branches can be categorized into *forward-* and *backward-edge* control-flow transfers based on the paths they represent in a program's CFG. Examples of instructions that perform forward-edge control-flow transfers are indirect `call` and `jmp` instructions, and examples of backward-edge transfers are `ret` instructions (in the x86 architecture). In general, CFI schemes that cover both control-flow transfer directions [24]–[36], [38], [39], [41], [45]–[48], [53]–[57], [59], [62]–[68], [70]–[78], [80]–[82], [152] offer the best security guarantees. However, some schemes only protect forward edges [37], [40], [42]–[44], [51], [58], [60], [61], [79], [147], [148] or backward edges [69], [83], with the assumption that the non-covered edges are handled by an existing scheme; for example, many forward-edge schemes assume the presence of a fine-grain, backward-edge protection mechanism such as a *shadow stack* [95], [153].

Granularity. The number of locations a CFI scheme allows an indirect branch to target defines the scheme's *granularity*. *Coarse-grained* schemes over-approximate the number of legal targets for an indirect branch, providing a larger attack surface to an adversary. Designs of coarse-grained schemes vary from: (a) restricting indirect branches to aligned instructions [26], [27], [63], (b) restricting `ret` instructions to `call`-preceded instructions and `call`/`jmp` instructions to function-entry points [32], (c) restricting the targets of `call`/`jmp` instructions to function-entry points of only address-taken functions [33], (d) confining indirect function calls to address-taken functions that have the same arity [53], and (e), various other points in the design space between (a) and (d) [24], [25], [28], [29], [31], [34]–[39], [41]–[44], [50], [64], [65], [75]. Unfortunately, given the over-approximation of legal targets, coarse-grain CFI has been shown to be bypassable [20], [154]–[156].

In contrast, *fine-grained* schemes [30], [40], [45]–[49], [51], [54]–[62], [66]–[74], [76]–[79], [93], [147], [148], [152] provide a tighter approximation of an indirect branch's legal targets, enhancing security by reducing a program's attack surface. This improved granularity is typically achieved with advanced static and/or dynamic analyses—e.g., modern points-to analysis [47], [57], [62], [71], [74], [77], [157], [158], type analysis [40], [56], [59], [61], [159]–[161], and class hierarchy analysis [49], [51], [54], [55], [58], [60]—to reduce the set of allowed indirect branch targets. However, similar to coarse-grained CFI schemes, attackers have also proven it possible to bypass fine-grained schemes by "bending" the control flow to branch targets in the same *equivalence class* [22], [23], [91], [92].

Compatibility. CFI schemes also differ based on the environments they can operate in. For example, some CFI schemes require access to *source code* [26]–[29], [31], [34], [35], [37]–[41], [46], [47], [49], [51], [56]–[58], [61]–[63], [66], [67], [74], [77], [79], [147], [148], [152], [157]–[161], while others operate directly on *binary code* [24], [25], [30], [32], [33], [36], [42]–[45], [48], [53]–[55], [59], [60], [64], [65], [68]–[73], [75], [76], [78], [80]–[83], [149]. There is a trade-off here, with source-code-agnostic solutions prioritizing compatibility with commercial off-the-shelf (COTS) software [23] in lieu of the increased effectiveness and coverage gained when source code is available to perform advanced analyses [40], [47], [49], [51], [57], [61], [77], [157], [159]–[161]. Additionally, while a majority of CFI schemes aim to be language-agnostic or target C software [24]–[34], [36], [38]–[41], [45]–[48], [53], [56], [59], [61]–[83], [152], [157]–[161], a handful of other schemes target specific languages, such as C++ [37], [42]–[44], [49], [51], [54], [55], [57], [58], [60] and Objective-C [35].

3. Compiling, Linking, and Loading

When building an executable from C source code, the source code is first compiled to object files which are then linked together to form the final binary. For example, in Figure 2, C files, `a.c` and `b.c`, are compiled into separate object files, `a.o` and `b.o`, which are then linked together to create executable `ab`. We will now expound on each stage: *compiling*, *linking*, and *loading*.

Compiling. `a.c` and `b.c` each represent separate compilation units, and are thus compiled into separate object files by the compiler without knowledge of the other source file or any libraries. For `a.c`, the compiler is unaware of the location of `gotpcrelx_func{1,2}` and `plt32_relaxed_func`, so it emits relocations against the `.text` section in `a.o` that the (static) linker will attempt to resolve once more information is present. The external functions `gotpcrelx_func{1,2}` are address-taken in the conditional (ternary) operator in `main` and then (indirectly) invoked. As a result, the compiler safely assumes that the addresses of `gotpcrelx_func{1,2}` will be present in the final executable's `.got` section at runtime; however, since there is a chance that `a.o` might link with another object file that defines `gotpcrelx_func{1,2}`, the compiler emits a `R_X86_64_REX_GOTPCRELX` relocation to inform the (static) linker that, if the function definition is present in the final executable, *relaxation* can be performed and the GOT indirection can be rewritten to a direct access (e.g., `mov` $\rightarrow$ `lea`). Additionally, the address of the external function `plt32_relaxed_func` is unknown during compilation, and so a `R_X86_64_PLT32` relocation is emitted that informs the linker to amend the (direct) `call` instruction to target the function's `.plt` slot in the final executable. There are similar `R_X86_64_PLT32` relocations emitted for the each of the aforementioned functions that are defined in `b.c` since each function calls an external (library) function (`puts`) that the compiler has no knowledge of.

Linking. The (static) linker will ingest object files `a.o` and `b.o`, and merge them together to create `ab`. It is in this stage that relocations present in the input object files are used to generate the executable's `.got` and `.plt` sections, and relaxations are applied to optimize certain code-paths now that more information is present. The three functions defined in `b.o` all use the (external) library function `puts` and will be transformed in the output executable to "jump" to the corresponding `.plt` slot emitted for `puts`. In order to resolve the address of `puts` at runtime, the linker will emit a relocation for `puts`'s `.got.plt` slot that the loader will resolve. The three `extern` functions used by `a.o` are all defined in `b.o`, and thus the linker can perform relaxation. For the `R_X86_64_REX_GOTPCRELX` relocations, the `mov` instructions that were made to move the *contents* of the `gotpcrelx_func{1,2}` `.got` slots into `%rax` are replaced with `lea` instructions that directly compute the addresses of `gotpcrelx_func{1,2}` without an additional memory read. Similarly, the call to `plt32_relaxed_func` can avoid indirection through the `.plt` and directly call the function.

Loading. When `ab` is executed, the (dynamic) linker/loader (`ld.so`) is first run to load `ab`, resolve its runtime dependencies, and apply dynamic relocations (assuming eager binding; interested readers are referred to Gaidis et al. [93] for a detailed description of delayed and eager binding). In this case, the loader applies the relocations for `puts`, filling the `.got.plt` slot of `puts` with its `libc` address.

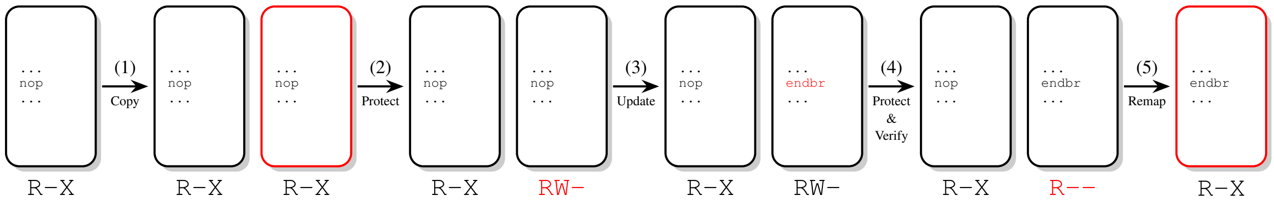


Figure 3. Demonstration of the secure live-patching scheme of NOPoutNG with changes between each stage highlighted in red.

TABLE 3. STATIC RELOCATION TYPES THAT CAN INDICATE AN ADDRESS-TAKEN FUNCTION.

Relocation	Calculation
R_X86_64_64	$value + addend$
R_X86_64_PC32	$value + addend - offset$
R_X86_64_PC64	$value + addend - offset$
R_X86_64_GOTPCREL	$GOT.offset + \&GOT + addend - offset$
R_X86_64_GOTPCRELX	$GOT.offset + \&GOT + addend - offset$
R_X86_64_REX_GOTPCRELX	$GOT.offset + \&GOT + addend - offset$

4. Notable Relocation Types

Table 3 lists the relocations that NOPoutNG looks for to indicate an address-taken function. Relocation R_X86_64_64 is typically found in *position-dependent* code to load an absolute 64-bit address; however, it is also found in *position-independent* code (i.e., PIC/PIE) to reference the address of a function from a global variable (e.g., in the .data section). Relocations R_X86_64_PC{32, 64} are used to calculate a PC-relative address, which can be part of a function-address computation whose result is then stored or passed around. This signals that the function is address taken; however, this relocation is also used to compute the address of a function relative to the PC for direct calls. In this case, we disregard the relocation, preserving the nop instruction associated with the relocation's target function. R_X86_64_GOTPCREL can be used to access a function address stored in the GOT section; this relocation is always indicative of an address-taken function. Similarly, R_X86_64_GOTPCRELX is a relaxable version of R_X86_64_GOTPCREL (as noted by the 'X' suffix)—i.e., if the relocation entry's target function ends up in the same binary as the place being relocated, then an optimization can be performed to eliminate a memory load. This relocation, in addition to its prefixed version, R_X86_64_REX_GOTPCRELX, which is meant for extended registers and 64-bit operations, indicate that a function's address is being taken.

5. Live Patching

Following Figure 3: initially, a code page marked R–X contains a 4-byte nop instruction that must be replaced with an endbr. In step (1), NOPoutNG creates a copy of the target page and subsequently changes the copy's memory permissions to RW– in step (2). Now, with a writeable page, NOPoutNG patches the code in step (3), replacing the nop with an endbr. In step (4), the page permissions are set read-only (R––) and its contents are compared against the original, unmodified page that has remained non-writable. If any changes have been made to the copy outside of adding the single endbr instruction, an error is thrown indicating a potential attack. Finally, if the verification succeeds, step (5) remaps the updated page atop the original.

6. Microbenchmark Description

The microbenchmark we crafted measures the overhead of dlopen and dlsym in a realistic setting. A single run of the microbenchmark: (1) dlopens zlib [127], a standard compression library; (2) dlsyms compress, uncompress, crc32, and crc32_combine; and (3) uses the dlsymed functions to checksum, compress, and uncompress 4 separate buffers of 16KB, 48KB, 96KB, and 256KB, derived from the Calgary corpus [128]. Notably, we populate the data buffers with non-zero, non-random data to ensure that the operations will perform similarly over multiple runs. Regarding checksumming, we use crc32 twice—once on either half of an uncompressed buffer—before passing the result of both operations to crc32_combine. We do this because zlib allows preloading the crc32 symbol, resulting in it being both defined in the library but also in its PLT. As a result, when zlib is dlopened, crc32 gets an endbr added to its prologue, making the dlsym of crc32 not incur the live-patching scheme that the other dlsymed functions do. Finally, the microbenchmark performs setup and tear down—e.g., (de)allocating buffers and populating them with data—outside of the timed portion.