IBTpoline: Flexible, Robust, and Intel IBT-compliant System Call Hooking

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

Ioulios Patmanidis
TU Delft
Delft, Netherlands
i.patmanidis-1@student.tudelft.nl

Isabelle Shapiro
Brown University
Providence, Rhode Island, USA
isabelle_shapiro@alumni.brown.edu

Georgios Portokalidis
IMDEA Software Institute
Madrid, Spain
georgios.portokalidis@software.imdea.org

Alexios Voulimeneas
TU Delft
Delft, Netherlands
a.voulimeneas@tudelft.nl

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

Abstract

System call interposition underpins essential systems tooling, from tracers and sandboxes to compatibility layers. While binary rewriting-based approaches offer state-of-the-art performance by transforming `syscall` instructions into indirect branches that target a zero-page trampoline, this design is fundamentally incompatible with modern, hardware-assisted protection. Specifically, Intel IBT—an increasingly deployed hardware-based forward-edge CFI mechanism—requires all indirect branches to target valid `endbr64` landing pads, causing existing zero-page trampolines to fault. Consequently, the fastest interposition mechanisms are currently broken on modern, IBT-enabled platforms.

We present `IBTpoline`: a lightweight, high-performance syscall interposition framework that is IBT-compliant by construction. Rather than relying on a zero-page mapping, `IBTpoline` rewrites both the `syscall` and its preceding number-loading instruction, seamlessly redirecting execution to a localized, IBT-compliant trampoline stub. This eliminates the security risks of mapping page zero while flexibly supporting pre-syscall, post-syscall, and replacement hooks. Furthermore, `IBTpoline` introduces a hardware-backed lockdown mode, uniquely combining `seccomp-bpf` with Intel CET shadow stacks to guarantee that installed hooks cannot be bypassed, even by an attacker with arbitrary read/write capabilities. Evaluated on IBT-enabled hardware, `IBTpoline` incurs just 1.49% overhead on a worst-case microbenchmark (compared to 10.69% for state-of-the-art approaches) and sub-1.5% overhead across SQLite, Redis, and Nginx.



This work is licensed under a Creative Commons Attribution 4.0 International License.

SIGOPS ATC 2026, Hong Kong, China
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2818-1/2026/11
https://doi.org/10.1145/3828523.3857066

CCS Concepts: • **Security and privacy** → **Systems security**; *Software security engineering*; • **Software and its engineering** → Software notations and tools.

Keywords: system call interposition, Intel CET/IBT

ACM Reference Format:
Alexander J. Gaidis, Ioulios Patmanidis, Isabelle Shapiro, Georgios Portokalidis, Alexios Voulimeneas, and Vasileios P. Kemerlis. 2026. IBTpoline: Flexible, Robust, and Intel IBT-compliant System Call Hooking. In *The 2026 ACM SIGOPS Annual Technical Conference (SIGOPS ATC 2026), November 15–18, 2026, Hong Kong, China.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3828523.3857066

1 Introduction

System calls (syscalls) form the narrow, well-defined boundary through which user programs request services from the operating system (OS): spawning processes, performing I/O, mapping memory, and communicating over the network, all funnel through a single entry point in the kernel before being dispatched to appropriate handlers. This makes the syscall boundary a unique and appealing vantage point for *observing* and *manipulating* program behavior. *Syscall interposition* capitalizes on this boundary by inserting a programmable layer between a user application and the kernel, allowing syscalls to be *inspected*, *transformed*, or *blocked* in flight. Consequently, syscall interposition underpins a wide range of systems: tracers and debuggers [48, 76, 104], sandboxes and security monitors [13, 26, 63, 85, 91, 92, 105, 108–112], compatibility layers such as Wine [116] and Proton [106], fuzzers [58, 96], reliability and fault-injection tooling [52, 72], and user-space file systems and OS extensions [7].

The myriad benefits of syscall interposition have prompted a long line of interposition mechanism designs, each navigating the fundamental tradeoff between robustness and performance. *Kernel-mediated approaches*—e.g., `ptrace` [98], `seccomp` [9], `seccomp-bpf` [101], and Syscall User Dispatch (SUD) [103]—can transparently intercept all syscalls made by an unmodified binary, but pay for that "perfect" coverage with expensive user–kernel context switches.

These mechanisms impose overheads orders of magnitude higher than the cost of a raw `syscall` instruction [49, 56, 119]. At the other end of the spectrum, techniques like `LD_PRELOAD`-*based symbol hooking* exhibit near-zero overhead but are inherently incomplete: they hook at coarse function-call boundaries rather than `syscall` instructions, missing inline syscalls and non-exported functions [44].

Binary rewriting techniques close this gap by transforming `syscall` instructions into branches to user-space interposition logic, achieving both granular, syscall-level coverage *and* excellent performance. The current state-of-the-art mechanisms—e.g., zpoline [119] and its successors, lazypoline [56] and `K23` [49]—capitalize on shared instruction lengths: on the x86-64 architecture, the 2-byte `syscall` instruction is exactly the same size as an indirect branch, `call *%rax`. Because the Linux syscall ABI dictates that the target syscall number resides in `%rax` prior to execution, replacing the instruction results in a branch to a virtual address exactly equal to the syscall number. However, using this technique as a generic interposition mechanism requires mapping a trampoline at virtual address `0x0`, consisting of a massive `nop` sled spanning every possible syscall number to "slide" execution into the interposition logic. This design carries significant drawbacks, most notably breaking the ubiquitous assumption that a `NULL` pointer dereference will crash a program. Both zpoline [119] and recent work [49] leverage Intel MPK [55] to mitigate this, yet MPK and similar schemes providing XOM come with their own shortcomings [26, 66, 112].

At a more fundamental level, binary-rewriting-based interposition mechanisms are on a collision course with modern hardware advancements. Intel Control-flow Enforcement Technology (CET) [53], first shipped in 2020, introduced a forward-edge control-flow integrity (CFI) [15] feature known as Indirect Branch Tracking (IBT). IBT mandates that all indirect control-flow transfers must target a valid `endbr64` "landing pad" instruction; otherwise, the processor immediately raises a `#CP` fault. This hardware feature is fundamentally incompatible with the zpoline-style approach, where `call *%rax` targets a sled of `nop` instructions rather than a valid `endbr64` landing pad (as `nop` is a single byte, it cannot encode a 4-byte `endbr64`). IBT-enabled binaries are now emitted by default by modern toolchains [18, 37], distributions are actively shipping IBT-ready userlands [17, 37], and the Linux kernel supports IBT in kernel space [34] with user-space support on the way [81]. *Consequently, the fastest existing syscall interposition mechanisms [49, 56, 119] are actively broken on the platforms where modern deployment is heading, and no existing scheme addresses this incompatibility.*

This paper presents `IBTpoline`: a syscall interposition framework that is *lightweight*, *flexible*, and—by construction—strictly *compliant with Intel IBT*.

The key insight behind `IBTpoline` is that the syscall number stored in `%rax` does not *need* to serve as the branch target for `call *%rax`; a completely different value can be loaded instead. Thus, in addition to replacing the `syscall` instruction itself, `IBTpoline` safely rewrites each syscall site's preceding number-loading instruction to instead load the *address* of a uniquely designated, per-syscall trampoline stub. The region that stubs reside in is a single, compact, contiguous memory region—N slots of 128 bytes, with each slot corresponding to a syscall number—that can be mapped anywhere within the lowest 2GB of the address space. Crucially, every trampoline stub begins with a valid `endbr64` instruction, guaranteeing IBT compatibility with the rewritten branch. Furthermore, each stub can be patched at load time to support one of *four* hook flavors, dynamically dispatching developer-provided hook functions *around*, or *in place of*, the syscall. To prevent infinite recursion if a hook re-enters the interposition layer, hook libraries are safely `dlmopen`ed into a separate, isolated *link namespace*, granting developers unhindered access to shared library functions. In short, `IBTpoline` provides an interposition framework that retains high performance while offering IBT compatibility, hooking flexibility, and the elimination of zero-page mapping complications (i.e., reliance on Intel MPK or other XOM-based solutions [23, 24, 28, 80]).

Two additional design decisions distinguish `IBTpoline` from its predecessors. First, we decouple *what* to rewrite from *how* to rewrite: the patching process is driven by a human-auditable *rules file* that specifies exactly how to patch target bytes and includes built-in structural validation checks. This turns the historically fragile problem of identifying and modifying x86-64 `syscall` sites into an offline, per-build, verifiable artifact rather than a runtime gamble. Second, `IBTpoline` introduces a hardware-backed *lockdown mode* for security-sensitive deployments. By combining a `seccomp-bpf` filter to strictly confine the `syscall` instruction pointer to the trampoline region with Intel CET shadow stacks—the other half of the hardware feature that breaks prior mechanisms—`IBTpoline` verifies that control flow legitimately traversed the trampoline region before executing any syscalls. Under lockdown mode, even an attacker equipped with arbitrary read/write primitives cannot issue a syscall that bypasses an installed hook.

We implemented `IBTpoline` in approximately 1,200 lines of code (LoC) in C, alongside a comprehensive rules file covering all 423 syscall sites in Debian 13's `(g)libc`. On a CET-enabled Tiger Lake processor supporting both IBT and shadow stacks, a worst-case microbenchmark (a non-implemented syscall in a tight loop) demonstrates that `IBTpoline` adds just 1.49% overhead atop a native, uninstrumented syscall. In contrast, zpoline incurs up to 10.69% overhead even in its most favorable configuration.

Across real-world applications (SQLite, Redis, and Nginx), IBTpoline's overhead never exceeds 1.43% in any hooking configuration and typically remains well below 1%, consistently outperforming zpoline. We also run a suite of correctness tests exercising 305 Linux syscalls across 270 target-able sites, triggering zero #CP faults and exhibiting no semantic regressions. Finally, we demonstrate IBTpoline's API expressiveness by implementing 6 hook libraries spanning tracing, profiling, compatibility, and testing use cases.

In summary, this paper makes the following contributions:

- **An IBT-compliant syscall hooking design.** We present the first binary rewriting-based, syscall-interposition framework natively compatible with Intel CET. By leveraging per-syscall trampoline stubs marked with endbr64 instructions that can reside anywhere in the low 2 GB, IBTpoline permanently eliminates the need for zero-page trampolines and reliance on MPK/XOM.
- **A hardware-backed lockdown mode.** We demonstrate how pairing IBTpoline with seccomp-bpf instruction-pointer filtering and CET shadow stacks guarantees that every syscall is strictly mediated by the trampoline—even when the target application is under attack.
- **Auditable and verifiable patching via rules files.** By decoupling rewrite-target identification from the runtime engine via an external rules file, we establish a robust pipeline for offline, verifiable patching.
- **A comprehensive implementation and evaluation.** We implement IBTpoline in ≈ 1.2K LoC and thoroughly evaluate its performance and correctness against microbenchmarks, macrobenchmarks, and 6 practical hook libraries.

2 Background and Related Work

2.1 Syscall Interposition

When a user application requires a *service* from the OS kernel—such as spawning processes, executing programs, opening network connections, or modifying the file system—it invokes the system call (syscall) API. On x86-64, syscalls are typically issued via the syscall instruction, which transitions the CPU from user to kernel mode and jumps to a kernel-side dispatch routine whose address is held in a dedicated model-specific register (MSR). Linux v6.17, for example, implements 402 syscalls. The well-defined boundary of the syscall API—together with its role as the kernel gateway—makes it a rich surface on which to observe and manipulate system interactions. The set of techniques that attach logic to syscall entry or exit is known as *syscall interposition*.

Importantly, we make a distinction between the syscall interposition *mechanism* and the use cases it enables. For example, UFO [7] provides a global file system that transparently allows access to remote files by extending OS functionality, in user space, with syscall interposition. In this case, the application—i.e., the file system implementation—is built atop the syscall interposition mechanism that uses ioctl calls on a process' PID to control it. The remainder of this section dives into popular syscall interposition mechanisms.

ptrace. Linux provides the ptrace syscall [98], which lets a *tracer* process attach to and control one or more *tracee* processes and observe every syscall they issue. The kernel mediates the relationship end-to-end: it records the tracer/tracee link in each task_struct and stops the tracee at syscall entry, syscall exit, and signal delivery so the tracer can inspect or rewrite its registers and memory contents, naturally enabling syscall interposition on *unmodified* binaries. Because the tracer is itself a user process, the interposed logic is unconstrained—it can run arbitrary code, hold persistent state, and freely call into libraries or the network. Crucially, that logic lives in a separate address space from the tracee, so a compromised or malicious target cannot tamper with the interposition policy or its state. This tamper-resistance has made ptrace the foundation for debuggers and tracers [48, 76, 104], security mechanisms [13, 26, 63, 85, 105, 109, 111, 112], fuzzing [58, 96], and reliability and fault-injection tooling [52, 72]. The same cross-address-space isolation, however, is the source of ptrace's overhead: each intercepted syscall requires a pair of context switches between tracer and tracee, plus additional syscalls to access the tracee's registers and memory, making ptrace unsuitable for performance-sensitive interposition [49, 56, 72, 94, 105, 119].

seccomp-bpf. Linux v2.6.12 (2005) introduced seccomp [9] as a coarse-grained sandboxing primitive: a process can transition (one-way) into a restricted mode in which the only permitted syscalls are read, write, _exit, and sigreturn. The original target was compute jobs that operate on untrusted data and have no need for general kernel services. Linux v3.5 (2012) generalized this design into seccomp-bpf [101], which lets a process install an arbitrary syscall-filtering policy expressed as a classic BPF program [74]. The filter runs in-kernel on the syscall entry path and can inspect the syscall number, the calling architecture, the scalar argument values, and the user-space instruction pointer; it returns one of a fixed set of actions—typically ALLOW, ERRNO, KILL, or TRAP (which raises SIGSYS in the calling thread). Each installed filter is permanent—it cannot be removed or weakened—and successive filter installations stack rather than replace; filters are inherited across fork and exec. Because filtering happens in-kernel on the syscall hot path and avoids any cross-process round trips, seccomp-bpf has become a foundational security primitive in sandboxing and syscall filtering [14, 16, 29, 45–47, 62, 68, 84, 88, 117, 118, 121].

However, the design of seccomp-bpf has well-known limitations. Classic BPF cannot dereference user-space memory—the in-kernel filter would have to walk untrusted pointers under a TOCTTOU race against the calling thread [11]—so filters can examine pointer arguments only as opaque addresses, never as the structures they reference.

Additionally, the hierarchical design of `seccomp-bpf` lends itself to overprivilege as filters are inherited across process creation and program execution without the ability to dynamically adjust filters to new execution contexts [41, 79].

Beyond pure allow/deny filtering, `seccomp-bpf` doubles as a generic syscall-interposition mechanism through a handful of its return actions. `SECCOMP_RET_TRACE` notifies an attached `ptrace`-based tracer of the intercepted syscall, letting the tracer inspect or rewrite arguments and return values exactly as it would at an ordinary syscall stop—but only for the syscalls the filter selects, sidestepping the per-syscall cost of stopping on every kernel entry. Similarly, `SECCOMP_RET_USER_NOTIF` forwards the intercepted syscall to an attached user process that is waiting on an associated notification file descriptor. Finally, `SECCOMP_RET_TRAP` directs the kernel to send a `SIGSYS` signal to the triggering thread where a signal handler implements custom interposition logic. Importantly, these generic syscall-interposition enablers can result in expensive user–kernel context switching; the performance advantage of `seccomp-bpf` typically only applies to syscalls that can be resolved entirely in-kernel.

SUD. Syscall User Dispatch (SUD) [103] is a per-thread syscall-interposition mechanism introduced in Linux v5.11 to support compatibility layers like Wine [116] and Proton [106] that need to emulate syscalls issued from specific regions of their address space. A process sets up SUD via a single `prctl` call that registers a contiguous memory region: syscalls whose instruction pointer lies within that region pass through the kernel normally, while syscalls from any other address raise `SIGSYS` on the calling thread, handing control to an in-process signal handler that can inspect, emulate, or redirect the call. (This region can also be defined as the inverse.) SUD also exposes a per-thread *selector* byte in user memory that the kernel consults on every syscall; this byte can be toggled in user space without a syscall to enable or disable interception, both for performance and to let the signal handler safely issue its own syscalls. Notably, while `seccomp-bpf` *can* be used to achieve a similar effect, it is more rigid than SUD; `seccomp-bpf` has limited support for address-range filtering and filters, once installed, cannot be removed or toggled off. SUD's `prctl`, in contrast, can be re-issued to update the range at runtime, which matters for any deployment where target ranges change dynamically (e.g., with JIT or `dlopen`). However, SUD is *not* a security mechanism: because the target shares an address space with the interposition handler, an attacker can disarm interposition by flipping the selector, calling `prctl` again, or simply jumping into the "pass through" range.

LD_PRELOAD. When a program is executed, the kernel loads both it and the dynamic linker (`ld.so`) it specifies into a fresh process address space and transfers control to the linker, which recursively loads the program's dependencies, applies relocations, runs constructors, and finally jumps to the executable's entry point that eventually calls `main`.

The `LD_PRELOAD` environment variable extends this load-time machinery: any library named in `LD_PRELOAD` is loaded by `ld.so` first and inserted into the global symbol-resolution scope *before* the executable's own required dependencies. This placement has two consequences: the preloaded library's constructors fire before any of the executable's regular dependencies, and—more importantly—any symbol the preloaded library exports shadows the same symbol in libraries loaded later, so a call to, e.g., `open` from the main executable resolves to the preloaded definition rather than the one in `libc`.

Since applications typically reach the kernel through `libc` wrappers, rather than raw `syscall` instructions, hooking those wrappers via `LD_PRELOAD` is a valid and commonly used approach to syscall interposition [113, 120]. The mechanism is lightweight (no binary rewriting, no expensive user–kernel context switches), broadly available (`glibc`, `musl`, and the BSDs all implement `LD_PRELOAD`), and trivial to deploy—a single environment variable, with no recompilation. As a *robust* interposer, however, `LD_PRELOAD` falls short of the other mechanisms in this section in two main ways [44]. First, it intercepts at function-call boundaries, not at the syscall boundary: any code path that reaches the kernel without going through a dynamically-resolved symbol—e.g., inline `syscall` instructions, statically-linked binaries, and even `libc`'s own internal syscall sites that bypass its public wrappers—is invisible to a preloaded hook. Second, the hook can be disabled by tampering with the (meta)data of the dynamic linker [31]—a problem that cannot be trivially addressed as it requires integrity-protecting the (meta)data of `ld.so`. Consequently, `LD_PRELOAD` remains convenient for cooperative interposition (debuggers, profilers, allocators), but security-sensitive interposers—which must observe every syscall regardless of how it was issued—avoid it.

Binary Rewriting. Recent work on syscall interposition mechanisms has largely focused on binary rewriting techniques, where `syscall` instructions in a binary are rewritten to direct control flow to some interposition logic. Rewriting can happen either offline or online during program loading. The main benefit of this approach is avoiding kernel modifications and expensive user–kernel context switches. zpoline [119] introduced an elegant variant of this idea for x86-64: it rewrites the 2-byte `syscall` instruction as the same-sized `call *%rax`. Because Linux's syscall calling convention requires the target syscall number to sit in `%rax` immediately before the `syscall` instruction, the rewritten `call *%rax` branches to the address numerically equal to that number. zpoline exploits this by mapping a sled at virtual address `0x0` that spans the possible values of `%rax` and terminates in a stub that invokes the interposition logic. While clever, mapping virtual address `0x0` results in three main issues: (1) putting code at address `0x0` breaks common programming assumptions—e.g., that a `NULL`-pointer dereference will cause a fault and halt the program—resulting in a new requirement for hardware features (e.g., Intel MPK [55]);

(2) mapping memory at address `0x0` requires administrative privileges to adjust the value of `/proc/sys/vm/mmap_min_addr`; and (3) mapping address `0x0` is not possible on some OSes, such as Windows [22] and macOS [73].

lazypoline [56] is an alternative approach that shares zpoline's page-zero trampoline but addresses the disassembly problem by taking a "lazy," delayed method: it uses SUD to intercept a given `syscall` the first time it executes and only then rewrites that site to `call *%rax` in zpoline-like fashion. The program's own `syscall` executions identify the rewrite targets with perfect accuracy and cover `syscall` instructions that appear dynamically (e.g., via JIT or `dlopen`). Because lazypoline reuses zpoline's `call *%rax` encoding, however, it inherits zpoline's requirement of a trampoline mapped at virtual address `0x0`; its runtime rewriting is also performed under multi-threaded execution and is exposed to a class of concurrency hazards—non-atomic writes, missing instruction-cache coherence, and fragile permission handling—that zpoline's single-pass load-time rewrite avoids [49].

`K23` [49] systematizes the limitations of zpoline and lazypoline, identifying a family of soundness, completeness, and security flaws: incomplete interposition (e.g., of vDSO-based syscalls and syscalls issued before the interposer is injected via `LD_PRELOAD`), data corruption from misidentifying embedded data as `syscall` bytes, and the rewriting hazards noted above. `K23` responds with a hybrid design that combines an offline observation phase, a brief `ptrace`-based startup phase, and a single online rewriting pass restricted to call sites pre-validated during the offline phase. The offline phase runs the target under an SUD-based logger that records the region and offset of every observed `syscall` site; then, at runtime, `ptrace` handles startup syscalls, and once `K23`'s runtime library loads, only the sites logged offline are rewritten. Notably, `K23` also uses an SUD fallback scheme to catch any sites the offline phase missed. Like zpoline and lazypoline, `K23` also maps the trampoline at virtual address `0x0`; however, it uses Intel MPK [55] to protect the trampoline from `NULL` read/write accesses.

Crucially, none of these schemes preserve their interposition guarantees in the presence of an active adversary. Their dispatch logic is ordinary, unprotected code at a fixed, known location (virtual address `0x0`): an attacker capable of control-flow hijacking can branch directly to the `syscall` instruction(s) inside the trampoline—bypassing hook logic entirely—or into the middle of the interposer itself, corrupting the state it operates on.

2.2 Intel CET

Intel debuted Control-flow Enforcement Technology (CET) with its 11th-generation "Tiger Lake" processors in 2020 [53]: a hardware feature aimed at providing efficient control-flow integrity (CFI) to mitigate control-flow hijacking. CET has two components. The first, a *shadow stack*, protects backward edges (e.g., `ret`): every `call` pushes its return address onto both the normal stack and a protected shadow stack, and every `ret` pops both and compares them, raising a `#CP` (Control Protection) exception on mismatch. The second component, *Indirect Branch Tracking* (IBT), protects forward-edge indirect transfers (e.g., `call *` and `jmp *`). IBT introduces a 4-byte instruction, `endbr64`, and requires that every indirect branch land on an `endbr64`; any other target raises a `#CP`. On older processors without CET, the `endbr` instruction is treated as a `nop`. Mainline Linux currently supports user-space shadow stacks and kernel-space IBT—as well as a fine-grained IBT variant, FineIBT [34, 43]. User-space IBT support is currently being planned for the kernel [81].

3 Threat Model

Our threat model concerns `IBTpoline`'s hooking mechanism itself and makes no assumptions about interposition logic implemented atop the tool—i.e., we assume developer-written hook code is benign and not actively malicious. We assume that a process augmented with `IBTpoline` may contain vulnerabilities exploitable by an attacker to gain arbitrary read and/or write primitives [100, 107], or control-flow "bending" capabilities [10, 12, 20, 35, 36, 95, 123]. Our goal under this model is to ensure that `IBTpoline`'s interposition continues to function as intended while under attack.

We assume standard user-process mitigations that are correctly implemented and cannot be disabled by the attacker: in particular, non-executable memory [70, 75] enforcing a W^X policy [77] (i.e., code injection [59] is not possible) and Intel CET—both IBT and shadow stacks [53] (i.e., forward-edge indirect branches must land on an `endbr64` and backward-edge control flow cannot stray from the ordering imposed by the shadow stack). Other defenses, such as stack-smashing protection [27], ASLR [40, 83], code diversification [64, 67, 114], CPI [65], (fine-grain) CFI [15, 43], and protection against data-only attacks [6, 60, 61, 78, 86, 89], are orthogonal to `IBTpoline`'s correctness: we neither preclude nor depend on them. Finally, we deem micro-architectural attacks [50, 99] out-of-scope and hardening against such attacks [21, 25, 82, 99, 122] orthogonal to `IBTpoline`; however, due to the additional indirect calls `IBTpoline` injects into a program, we still recommend using a host with hardware-based Spectre-v2 mitigations [54] commonly found on CPUs with Intel CET. Our threat model is on par with the state-of-the-art in syscall hooking [13, 26, 63, 85, 105, 109, 111, 112].

4 Design

`IBTpoline` is a *syscall hooking framework* that is *lightweight* (incurring 1.49% overhead on a syscall-intensive microbenchmark where zpoline [119] incurs 10.69%—over 7× more; §6.1.1), *flexible* (providing four programmable hook types that observe, alter, or wholly replace syscall behavior, demonstrated with realistic use cases in Section 7.1), and *IBT compatible* (interposing on IBT-protected code via `endbr64`-fronted

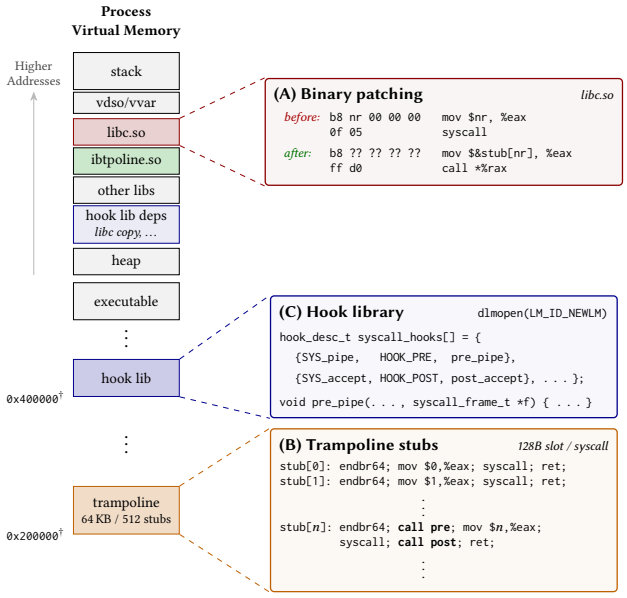


Figure 1. IBTpoline design overview. Addresses marked † are the default placement; under IBTPOLINE_ASLR, both regions are mapped at a random, kernel-chosen address below 2GB via MAP_32BIT.

Table 1. Rules file grammar.

rules	::=	(*comment* | *blank*)* *section*+
section	::=	*header* (*instr* | *comment* | *blank*)*
header	::=	#soname: SP *path* SP *patchStart* SP *patchEnd*
instr	::=	SP* *offset* SP *size* SP *expected* SP *replacement* *modifier*?
modifier	::=	+
path	::=	*filesystem-path*
patchStart, *patchEnd*	::=	*hex*
offset, *size*	::=	*hex*
expected, *replacement*	::=	*hex*
hex	::=	0x [0-9a-fA-F]+
comment	::=	SP* #¬soname: .*

```
1  #soname: libc.so.6 0x28000 0x163000
2   # rt_sigprocmask
3     0x28501 0x4 0x0e000000 0xe+       # mov $0xe,%eax -> mov $&ts[0xe],%eax
4     0x28505 0x2 0x0f05     0xffd0     # syscall -> call *%rax
5  ...
6   # read
7     0x104ce0 0x2 0x31c0       0xeb23       # xor %eax,%eax -> jmp 0x104d05
8     0x104d04 0x1 0x66         0x90         # nop after ret
9     0x104d05 0x5 0x2e0f1f8400 0xb800000000 # mov $0x0,%eax
10    0x104d06 0x4 0x00000000   0x0+         # mov $nr,%eax
11    0x104d0a 0x2 0x0000       0xebd6       # jmp 0x104ce2 (back to syscall)
12    0x104d0c 0x4 0x00006690   0x0f1f4000   # nopl 0x0(%rax)
13    0x104ce2 0x2 0x0f05       0xffd0       # syscall -> call *%rax
14 ...
```

Listing 1. Example rules file snippet.

trampoline stubs while the state-of-the-art designs of zpoline, lazypoline [56], and K23 [49] break outright or must disable IBT to run; §4.2). At a high level, IBTpoline patches binaries (e.g., libc) at load time, rewriting syscall sites into indirect calls that target per-syscall trampoline stubs; each stub can invoke user-defined hooks *around*, or *in place of*, the intercepted syscall, and when no hook is registered for a syscall, a default "pass-through" stub preserves its original semantics. IBTpoline's IBT compatibility is no accident: with IBT-enabled binaries now emitted by default by modern toolchains and shipped in mainstream distributions (§2.2), prior interposition mechanisms [49, 56, 119] are destined for compatibility issues or insecure operating modes. We thus design IBTpoline from the ground up with IBT compatibility as a primary consideration.

IBTpoline is designed and implemented as a *runtime library*, dubbed ibtpoline.so, which is integrated into a process via LD_PRELOAD to allow for portability as well as easy testing and incremental adoption. Like other similar systems first constructed around a discrete runtime library [42], the design of IBTpoline can easily be applied in other settings, such as the dynamic linker (ld.so).

The remainder of this section details the intricacies of IBTpoline's design, beginning with our syscall patching mechanism (§4.1) before moving on to describe trampoline setup (§4.2), hook installation (§4.3), and a security-focused *lockdown* mode (§4.4). The first three phases—including how they affect a process' virtual address space—are depicted in Figure 1, blocks **(A)**–**(C)**.

4.1 Binary Patching

When a new program is executed, before the main function runs, a constructor from the LD_PRELOADed ibtpoline.so is invoked. This constructor is the main conductor of IBTpoline functionality. It first sets up the process' address space—loading the trampoline stubs (§4.2) and hooks (§4.3)—prior to *patching code* across the main executable and its dependencies. Patching transforms a given syscall site into an indirect call targeting its corresponding trampoline stub. This is illustrated in block **(A)** of Figure 1, where we assume only libc.so requires rewriting. Notably, this assumption is not arbitrary. A large-scale study performed by DeMarinis et al. [29] that recorded syscall profiles for 30K C/C++ binaries in Debian sid showed that the vast majority of syscall instruction invocations originate from libc. After libc, other standard libraries claim the remaining top-5 spots for the most popular syscall-bearing libraries, including: libpthread, libstdc++, libnss_resolve, and libglib. Given this, we contend that applications rarely perform raw syscalls themselves, justifying our libc focus.

Before patching is performed on a given syscall site in libc, a target syscall number ($nr) is moved into %eax followed by the syscall instruction. After patching, the value moved into %eax is the (constant-valued) address of the syscall's corresponding trampoline stub and the syscall instruction is replaced with call *%rax (similar to ∗poline [56, 87, 119]). As a result of executing these new instructions, libc.so will call into the given syscall's trampoline stub instead of directly executing the syscall.

Patching is done *surgically* and *securely* using a *rules* file specified by an environment variable (`IBTPOLINE_RULES`), which is parsed by `ibtpoline.so` during load time. During rewriting, code pages are set to be writable and then immediately reverted to be read and execute only once patching is done. (Note that the process has not yet handled any untrusted input at this stage.) The grammar of the rules file is given in Table 1, with an example implementation snippet for `libc.so.6` depicted in Listing 1. A rules file is made up of a collection of comments, blank lines, and *section*s; each *section* represents an object in virtual memory, containing a *header* that describes the object followed by a series of instructions (*instr*) that dictate how to patch it.

In Listing 1, the *header* (ln. 1) describes a patch *section* for `libc.so.6` starting at offset `0x28000` from the base load address of `libc.so.6` and ending at offset `0x163000`. Ln. 2–4 show instructions for patching a `rt_sigprocmask` `syscall` location, with ln. 3 addressing the rewrite of the `mov` into `%eax` and ln. 4 addressing the rewrite of the `syscall` instruction. Specifically, each instruction line begins with an offset to the bytes that need to be patched—e.g., the `mov` sits at `0x28501` bytes from the base load address of `libc.so.6`—followed by the size of the required patch, the bytes expected to be found at the given location (for sanity checking), and finally, the new bytes to patch in. For example, on ln. 4, at offset `0x28505`, we expect to see 2 bytes, `0x0f05` (the `syscall` instruction), which `IBTpoline` will replace with `0xffd0` (`call *%rax`). Notably, the replacement bytes may also contain a modifier, as shown on ln. 3 (i.e., `0xe+`), which is used to express a value that is not known until run time. In this case, we are rewriting `mov $0xe, %eax`—which moves the syscall number for `rt_sigprocmask` (i.e., `0xe`) into `%eax`—to `mov $&ts[0xe], %eax`—which moves the address of `rt_sigprocmask`'s trampoline stub (i.e., `&ts[0xe]`) into `%eax`; however, the address of the trampoline stub cannot be resolved until run time.

We manage this uncertainty with a replacement modifier, `+`, that we call a *trampoline-address* modifier. When a `+` augments a *replacement* field, the field no longer describes the actual bytes to patch the binary with; instead, the *replacement* field is treated as a syscall number. While the rules file is getting parsed, this syscall number is used to resolve the address of the corresponding trampoline stub:

$$\&trampoline_stub_{nr} = \&trampoline_base + nr \cdot stub_size$$

This address is then used as the replacement when patching. Importantly, since the trampoline code is placed in the first 2GB of the address space (§4.2), a `mov` of a 4-byte value into `%eax` can easily be overwritten without special consideration—a 4-byte value covers up to 2^{32} possible addresses and a `mov` into `%eax` also clears the upper 32 bits of `%rax`. In our experience, this is the most common case when patching libraries. However, there are a handful of special cases where the typical 6-byte rewrite (i.e., the trampoline stub address and `call *%rax`) does not work because either: (1) the syscall number is not known statically, or (2) there is not enough room to perform inline rewriting.

In both cases (1) and (2), `IBTpoline` must carve space out of the executable section(s) of the respective binary (e.g., `libc.so`) to insert additional instructions. To achieve this, `IBTpoline` finds a stash spot that can be (short) `jmp`ed to—typically in the padding bytes between functions. The `read` syscall rewrite from ln. 6–13 of Listing 1 is an example of using a stash spot after the end of a function. A 2-byte instruction (`xor %eax, %eax`) that sets the value of `%eax` to 0 (i.e., `read`'s syscall number) for the subsequent `syscall` instruction will be overwritten with a short `jmp` that targets a stash spot after `read`'s `ret`. The code written to this stash spot (ln. 9–12) loads the address of `read`'s trampoline stub into `%eax` before jumping back to the function's body where the `syscall` instruction will be replaced with `call *%rax`.

For cases where the syscall number is not known until run time, we write code to a stash spot that dynamically computes the trampoline address based on the value in `%eax` (e.g., `shl $0x7, %eax; add $&ts_base, %eax`).

Rules-file Generation. The load-time patching engine of `IBTpoline` ingests an offline-computed rules file to selectively validate and rewrite target syscall sites. Crucially, `IBTpoline` performs no dynamic disassembly or complex runtime code analysis. The *soundness* of the binary rewriting process is therefore entirely dependent on the external generation of this rules file, rendering target identification strictly *orthogonal* to `IBTpoline`'s core architectural design. Recent advancements in frameworks such as Egalito [115], RetroWrite [32], and ARMore [30] have consistently demonstrated that sound, static binary analysis of code emitted by modern toolchains (e.g., GCC and LLVM) is highly reliable. By building atop these frameworks, developers can generate rules files fully automatically, resolve irregular assembly edge cases via LLM-assisted analysis, or even author rules manually. Because a rules file is tied to a specific binary build and only needs to be generated once, it represents a one-time, offline cost, falling in line with `IBTpoline`'s design goals.

Rules-file Validation. Even though rules-file generation is external to `IBTpoline`, we provide three complementary validation mechanisms to test the correctness of the rules file and `IBTpoline`'s patching. First, `IBTpoline` performs *static post-patch verification*: once rewriting completes, `ibtpoline.so` rescans the executable pages of every patched object for residual `0x0f 0x05` byte pairs (i.e., the `syscall` instruction). Because this check operates on raw bytes rather than decoded instructions, it may flag `syscall` occurrences that are not instruction boundaries, but it will not miss a `syscall` instruction that a rules file did not cover. Second, `IBTpoline` supports an *audit mode* that installs the lockdown filter (§4.4) with `SECCOMP_RET_LOG` in place of `SECCOMP_RET_KILL_PROCESS`, converting each escaping syscall into a logged event attributed to a concrete location.

Third, we provide a correctness suite (§6.3) that exercises patched binaries end-to-end, detecting both semantic regressions and IBT faults. In combination, these mechanisms render rules-file omissions diagnosable rather than silent: the first bounds the problem statically, the second localizes any residual escape at run time, and the third confirms that patching preserves program semantics.

Stash-spot Availability. Importantly, IBTpoline's rewriting engine depends on the ability to rewrite syscall sites: if patching a syscall site inline is not feasible due to byte restrictions, we rely on the presence of stash spots in the executable section of the binary—typically at the end of the function containing the target syscall site. Theoretically, there is no way to guarantee the *completeness* of this approach, as a binary can exist that uses the entirety of its executable sections. However, in practice, we have noticed that modern binaries are not optimized for space (i.e., `-Os`) [30, 115], but rather speed. As a result, padding is liberally applied to a binary for instruction alignment—a side effect that benefits IBTpoline by creating stash spots. In the event that there are truly no stash spots in a binary, a post-compilation transformation can be applied to the binary to expand an executable section (e.g., the `.text` section) to create rewriting space [32, 115].

4.2 Trampoline-based Interception

At load time, in the constructor of `ibtpoline.so`, IBTpoline maps a *trampoline* region in the process' virtual memory (see Figure 1, **(B)**). The trampoline region is a single, contiguous, aligned, non-writable, executable region that all rewritten syscalls (indirectly) call into. The trampoline is placed in the bottom 2GB of the address space to ensure it is reachable from the patched syscall instructions (i.e., with `call *%rax` where `%rax` holds a 32-bit address). By default, the trampoline is located at `0x200000`; however, through optional environment variables its base address can be specified by a user (i.e., with `IBTPOLINE_ADDR`) or randomized by the kernel (i.e., with `IBTPOLINE_ASLR`) via `mmap`'s `MAP_32BIT` flag.

Randomizing trampoline placement is a capability unique to IBTpoline; other ∗poline schemes (zpoline, lazypoline, and K23) hard-wire their trampolines to virtual address `0x0`. As a result, IBTpoline makes it more difficult for an attacker to (ab)use the trampoline region as a code-reuse target—this is especially important since each trampoline entry is prefixed with an `endbr64` instruction, permitting indirect branches to trampoline entries. With the `MAP_32BIT` flag, the kernel confines mappings to $[2^{30}, 2^{31})$ and randomizes its base within a 2^{25}-byte (32MB) window at page (2^{12}) granularity, so the number of candidate bases is:

$$N = 2^{25}/2^{12} = 2^{13} = 8192,$$

yielding $\log_2 N = 13$ bits of entropy and a $2^{-13} \approx 0.0122\%$ probability of successfully determining the base address per blind guess. By extending `ibtpoline.so` with a custom base-address randomizer (forgoing `MAP_32BIT`), the base can instead be drawn page-uniformly from $[$`mmap_min_addr`$, 2^{31})$ and mapped with `MAP_FIXED_NOREPLACE`. Under the default `mmap_min_addr` of 2^{16}, this yields:

$$N = (2^{31} - 2^{16})/2^{12} = 2^{19} - 2^4 \approx 2^{19}$$

candidate bases, 19 bits of entropy, and a ≈0.0002% chance of a correct blind guess. Importantly, in either case, a wrong guess faults, and thus locating the trampoline typically requires a separate memory-disclosure primitive.

```
1  endbr64
2  push %r9
3  push %r8
4  push %r10
5  push %rdx
6  push %rsi
7  push %rdi
8  push %rsp
9  mov %r10, %rcx
10 call <pre_hook>
11 pop %rsp
12 pop %rdi
13 pop %rsi
14 pop %rdx
15 pop %r10
16 pop %r8
17 pop %r9
18 mov $nr, %eax
19 syscall
20 ret
```

(a) Pre stub.

```
1  endbr64
2  mov $nr, %eax
3  syscall
4  push %rax
5  push %r9
6  push %r8
7  push %r10
8  push %rdx
9  push %rsi
10 push %rdi
11 push %rsp
12 mov %r10, %rcx
13 call <post_hook>
14 pop %rsp
15 pop %rdi
16 pop %rsi
17 pop %rdx
18 pop %r10
19 pop %r8
20 pop %r9
21 pop %rax
22 ret
```

(b) Post stub.

```
1  endbr64
2  push %r9
3  push %r8
4  push %r10
5  push %rdx
6  push %rsi
7  push %rdi
8  push %rsp
9  mov %r10, %rcx
10 call <replace_hook>
11 pop %rsp
12 pop %rdi
13 pop %rsi
14 pop %rdx
15 pop %r10
16 pop %r8
17 pop %r9
18 ret
```

(c) Replace stub.

Figure 2. Trampoline stub variants.

After mapping the region, `ibtpoline.so`'s constructor formats it as an array of 128-byte stubs, one per syscall number, making its total size: 128B × 512 = 64KB (assuming 512 syscalls at the time of writing). Thus, calculating the address of a given syscall stub is simple: $\&trampoline_stub_{nr} = \&trampoline_base + nr \cdot 128$. Further, the constructor fills each 128-byte slot with the *default* ("pass-through") *stub*:

```
endbr64; mov $nr, %eax; syscall; ret
```

This 12-byte stub consists of an `endbr64` instruction for IBT compatibility—allowing the rewritten `call *%rax` instruction target the stub—followed by three instructions that make the intended syscall without additional effects and return to the stub's call site. This default stub preserves the intended functionality of the syscall, in the event that it is not targeted for hooking, with minimal overhead. From here, `ibtpoline.so` loads the user-defined hook library and parses its exported descriptor array (§4.3) to determine how to overwrite trampoline slots with one of four stubs that dispatch user-provided hook functions. Importantly, the cost of writing the trampoline stubs is paid during load time; at run time, hooking a syscall only adds overhead equal to the cost of calling the hook function(s). The remainder of this section describes the four other possible trampoline stubs.

Pre-syscall stub. The 40-byte *pre-syscall stub*, depicted in Figure 2a, calls a user-defined hook (ln. 10) prior to invoking the intended syscall (ln. 19).

The leading `endbr64` instruction (ln. 1) ensures compatibility with IBT, preventing #CP faults. The following six `push` instructions (ln. 2–7) save the six syscall argument registers to the stack. The seventh `push` instruction pushes `%rsp` to the stack (ln. 8). This final push is not for register preservation, but to place a pointer to the saved-register area on the stack where the System V ABI expects the seventh argument. This allows hook authors to modify syscall arguments in their hooks. Following that, ln. 9 moves the value in `%r10` (the fourth syscall argument) into `%rcx` (the fourth function-call argument) to account for calling-convention discrepancies. Finally, after calling the hook (ln. 10), the preserved register state is restored (ln. 11–17), the syscall is made, and control flow is returned to the stub's caller.

Post-syscall stub. The *post-syscall stub* (Figure 2b) is similar to the pre-syscall stub but with the syscall invoked before the user-defined hook function runs. In order to support this, an additional `push %rax` (ln. 4) and `pop %rax` (ln. 21) are inserted into the register save and restore sequences, respectively. This preserves the return value of the syscall—in addition to allowing its modification (§4.3)—when returning back to the stub's caller. These additional instructions add 2 bytes, making the post-syscall stub's size 42 bytes.

Prepost-syscall stub. The *prepost-syscall stub* is the largest of the variants, combining both a pre-syscall hook and a post-syscall hook. It has a size of 70 bytes, motivating our choice to make each stub slot 128 bytes and thus 64-byte aligned [33]. Notably, the stub size is largely the result of register state being preserved and restored twice: once around the pre-syscall hook and again around the post-syscall hook.

Replace stub. The *replace stub* (Figure 2c) substitutes the syscall with a call to a user-defined hook. Its size is 33 bytes.

4.3 Hook Model

After mapping the trampoline region and filling it with default stubs (§4.2), `ibtpoline.so`'s constructor loads the user-defined *hook library* named in the `IBTPOLINE_LIB` environment variable. The hook library is mapped into the process' virtual memory within `rel32` range of the trampoline region (see Figure 1, **(C)**); with the `IBTPOLINE_ASLR` flag set, the hook library is mapped at a random location in the low 2GB of the address space similar to the trampoline region. Importantly, the library is mapped using `dlmopen` with the `LM_ID_NEWLM` flag to place the hook library and all of its dependencies (e.g., `libc.so`) in a separate *namespace*—in Figure 1 this is rendered as blue boxes in the process' virtual address space. These segregated dependencies are never rewritten, which prevents hook *re-entry*: when the hook library calls into its private `libc.so`, syscalls issued from that copy execute normally and are not intercepted. This property is what lets developers write hooks using standard-library calls without risking infinite recursion or having to manually handle syscall re-entry in the hook functions themselves.

Table 2. Capabilities of each hook variant.

Variant	Arguments		Return Value		Issues syscall
	Read	Mutate	Read	Mutate	
`pre`	✓	✓	–	–	yes
`post`	✓	–	✓	✓	yes
`prepost`	✓	✓[a]	✓[b]	✓[b]	yes
`replace`	✓	✓	–	✓[c]	no

[a] Pre-context only. [b] Post-context only. [c] Hook's return value is syscall's.

Each hook library must export two symbols: `syscall_hooks`, an array of *hook descriptors*—one per hooked syscall—and `syscall_hooks_len`, the array's length. A given descriptor records the target syscall number, the hook type, and one or two function pointers to the hook implementation(s). The four *hook types* mirror the trampoline stub flavors: `HOOK_PRE` fires before the syscall, `HOOK_POST` after it, `HOOK_PREPOST` on both sides, and `HOOK_REPLACE` supplants the syscall entirely with a user-defined function. Table 2 summarizes each variant's capabilities; together they let developers inspect or mutate arguments and return values around a syscall, or elide the syscall altogether. `ibtpoline.so` consults these two exported symbols to patch the trampoline stubs in accordance with each descriptor.

`IBTpoline` declares the hook function-pointer type with an empty parameter list: i.e., `void (*hook_fn_t)()`. Under pre-C23 rules [38], the unspecified parameter list will suppress call-site type checking so that hooks of any arity can share the function-pointer type; under C23 [39], this syntax translates to a function-pointer type with zero parameters, requiring a cast when creating the hook descriptor. In either case, argument passing relies on the System V convention: the trampoline stub will pass the six syscall argument register values to the hook function in the standard argument registers. Given this, a hook function only needs to define parameters it needs access to. For example, a post-syscall hook function for `write` might have the following signature, ignoring the latter three unused syscall argument registers:

```
void __write_post(int fd, const void *buf, size_t n);
```

As described in Section 4.2, the trampoline stubs also pass a seventh argument on the stack by pushing `%rsp`. Prior to pushing `%rsp`, it points to a stack region containing the syscall's saved register state, which we call the *syscall frame*. Thus, as a result of pushing `%rsp`, the seventh argument is a pointer to the syscall frame, in the form of:

```
typedef struct { unsigned long a0, a1, a2, a3, a4, a5, rv; } syscall_frame_t;
```

This seventh argument can then be passed to one of several *helper functions* `IBTpoline` provides to mutate syscall arguments or get/set the syscall return value. For example, a post-syscall hook function for `read` that prints its arguments and return value could be coded as follows:

```
void __read_post(int fd, const void *buf, size_t n,
        unsigned long a3, unsigned long a4, unsigned long a5,
        syscall_frame_t *sf)
{ fprintf(stdout, "[+] fd = %d, buf = %p (%s), n = %zu | rv = %ld\n",
                  fd, buf, (char *)buf, n, syscall_post_get_rv(sf)); }
```

```
1  #define TRAMP_LEN   (STUB_SZ * SYSCALL_MAX) /* 128 * 512 = 64 KB */
2  #define IP_LO       offsetof(struct seccomp_data, instruction_pointer)
3  #define IP_HI       (IP_LO + 4)
4  /* tramp_base = trampoline base, resolved at load time */
5
6  struct sock_filter lockdown_filter[] = {
7      /* Load upper 32-bits of instruction pointer (i.e., IP[63:32]) */
8      BPF_STMT(BPF_LD  | BPF_W   | BPF_ABS, IP_HI),
9      /* If nonzero, jump to KILL b/c trampoline region is < 2^32 */
10     BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, 0, 0, 4),
11
12     /* Load lower 32-bits of instruction pointer (i.e., IP[31:0]) */
13     BPF_STMT(BPF_LD  | BPF_W   | BPF_ABS, IP_LO),
14     /* If IP < tramp_base then jump to KILL */
15     BPF_JUMP(BPF_JMP | BPF_JGE | BPF_K, tramp_base, 0, 2),
16     /* If IP >= tramp_base + TRAMP_LEN then jump to KILL */
17     BPF_JUMP(BPF_JMP | BPF_JGE | BPF_K, tramp_base + TRAMP_LEN, 1, 0),
18
19     /* ALLOW syscall */
20     BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_ALLOW),
21     /* KILL the process */
22     BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_KILL_PROCESS),
23 };
```

Listing 2. `seccomp-bpf` lockdown program template.

4.4 Lockdown Mode

`IBTpoline`'s hooking mechanism, as described thus far, is *cooperative*, faithfully interposing on syscalls made by benign code. While this is sufficient for certain use cases such as tracing, profiling, and testing (§7.1), it is inadequate for security-focused use cases such as auditing syscall activity and sandboxing, where an attacker (§3) will search for paths to make a syscall that circumvents `IBTpoline`'s trampoline.

`IBTpoline` features a *lockdown mode* to address this scenario, which ensures that all syscalls originating from the main program or its dependencies are strictly routed through a trampoline stub and any registered hooks. Notably, `IBTpoline`'s entire hooking scheme is "baked into" program code: patched syscall sites encode each trampoline-stub address as an immediate value (§4.1), stubs reside in non-writable memory and invoke hook functions via direct calls, and the hook descriptor array (§4.3) is consulted only at load time, playing no role in runtime dispatch. In short, `IBTpoline` does not expose any data (e.g., function pointers, `ld.so` (meta)data) that can be tampered with to bypass or uninstall a hook. Yet, this is not sufficient to guarantee an attacker cannot issue a syscall without traversing the trampoline.

To ensure an attacker is confined to performing syscalls through the tamper-proof trampoline, `IBTpoline` configures the kernel to block any syscall whose instruction pointer (`%rip`) lies outside the trampoline region. We implement this by installing a `seccomp-bpf` filter as the final step in `ibtpoline.so`'s constructor, after mapping the trampoline region, loading the hook library, and rewriting the trampoline stubs. This ordering is critical: because the trampoline's base address may be user-defined or kernel-randomized, `IBTpoline` must first resolve this address before embedding it into the `seccomp-bpf` program and applying the filter.

Listing 2 illustrates the `seccomp-bpf` program template used for lockdown mode. Because classic BPF instructions operate strictly on 4-byte words, the 8-byte instruction pointer must be split into its lower (`IP_LO`) and upper (`IP_HI`) 32 bits for the address-range check. `IBTpoline` mandates that the trampoline region resides within the lowest 2GB of the address space, simplifying this verification: the upper 32 bits must strictly equal zero (ln. 7–10). The lower 32 bits must then fall within the dynamically resolved trampoline bounds, specifically [`tramp_base`, `tramp_base`+`TRAMP_LEN`) (ln. 12–17). If the instruction pointer is valid, the filter yields `SECCOMP_RET_ALLOW` (ln. 20); otherwise, it returns `SECCOMP_RET_KILL_PROCESS` (ln. 22) to terminate the offending process. For debugging purposes, developers can temporarily replace the `KILL` action with `SECCOMP_RET_TRAP` to identify legitimate syscall sites omitted by a rules file.

Furthermore, `seccomp-bpf` filters are hierarchical by design (§2.1): once installed, they cannot be removed and are permanently inherited across `execve`. If the trampoline is mapped to a fixed location (whether default or user-defined), this inheritance does not pose a problem for new processes. However, supporting `IBTPOLINE_ASLR` necessitates special handling. We recommend employing adaptive syscall filters [41] to dynamically swap filters during `execve`. Alternatively, the system can randomize the trampoline region once prior to filter installation, signaling to subsequent programs that the trampoline address must remain constant for the remainder of the execution chain.

`IBTpoline`'s lockdown mode faces an additional challenge: permitting legitimate syscalls from the namespaced dependencies of the hook library, while strictly ensuring that all syscalls originating from the main program (even under exploitation) remain unconditionally hooked. At a high level, this requires solving two sub-problems: #1 allowing syscalls from namespaced libraries, whose addresses naturally fall outside the `seccomp-bpf` filter range; and #2 guaranteeing that control flow legitimately traversed a primary trampoline stub prior to invoking a namespaced-library syscall.

To resolve problem #1, `IBTpoline` rewrites syscall sites within namespaced libraries to target a *shadow trampoline region*. This region contains stubs that execute syscalls, mirroring the design of the primary trampoline's default stubs. `IBTpoline` then extends the `seccomp-bpf` filter to permit syscalls originating from this shadow trampoline range. Importantly, only the specific syscalls invoked by the hook library require rewriting. By tailoring a rules file explicitly for the hook library's dependencies, we drastically reduce the attack surface. For example, a hook library that solely calls `printf` from its namespaced `libc` can rely on a targeted rules file that is a strict subset of the rules applied to the main program's `libc`.

To resolve problem #2, `IBTpoline` leverages Intel CET's shadow stacks (§2.2). When shadow stacks are enabled and a primary trampoline stub calls a hook function, a return address residing within the primary trampoline region is safely pushed to the hardware-protected shadow stack.

```
1     endbr64                     /* IBT landing pad (call target) */
2     rdsspq  %rcx                /* %rcx = SSP (walk cursor)      */
3  .walk:
4     mov     (%rcx), %r11        /* load entry; guard #PF = deny  */
5     btr     $63, %r11           /* signal-frame marker? (bit 63) */
6     jnc     .range
7     lea     8(%rcx), %rax
8     cmp     %rax, %r11
9     je      .deny               /* it is a marker => deny        */
10 .range:
11    sub     $TRAMP_BASE, %r11   /* entry in trampoline range?    */
12    cmp     $0x10000, %r11
13    jb      .allow              /* yes => allow                  */
14    add     $8, %rcx            /* keep walking with next entry  */
15    jmp     .walk
16 .allow:
17    mov     $nr, %eax           /* slot's syscall number         */
18    syscall
19    ret                         /* %rax = kernel return value    */
20 .deny:
21    ud2                         /* check failed                  */
```

Listing 3. Shadow trampoline stub.

IBTpoline capitalizes on this property by augmenting the shadow trampoline stubs with *verification code* that walks the shadow stack, confirming that control flow legitimately passed through the primary trampoline before reaching the shadow stub (Listing 3). To prevent runaway walks of the shadow stack, IBTpoline maps a *guard page* immediately above each thread's shadow stack; if the walk exhausts the stack, this page gracefully triggers a fault that denies the syscall (ln. 4). We favor using a guard page over planting a sentinel value at the top of each shadow stack as the latter requires toggling `wrss`, which can introduce gadgets into the process that may allow an attacker to write to the shadow stack and bypass the walk. Finally, the walk carefully accounts for signals that might interrupt trampolines and hook functions (ln. 5–9). IBTpoline checks whether the shadow stack value is a marker denoting a separate signal-handling context [102]; if so, it conservatively denies the syscall.

Special care is needed for threads spawned from hook functions in lockdown mode that perform syscalls: IBTpoline cannot reliably distinguish whether the syscalls made by such threads were dispatched legitimately by the hook library or maliciously forged by an attacker.

5 Implementation

Prototype. We implemented IBTpoline's runtime patching, trampoline setup, and hook installation within a constructor of the `ibtpoline.so` runtime library, which is injected into the target process' address space via `LD_PRELOAD`. The core `ibtpoline.so` library is highly compact, consisting of ≈ 1.2K lines of code (LoC) in C. To demonstrate IBTpoline's practicality, our prototype includes a comprehensive rules file (≈ 1.3K lines) that patches 423 distinct syscall sites within Debian 13's ("Trixie") `libc.so.6` (`v2.41-12+deb13u2`).

Currently, our prototype only patches objects present at load time; Section 7.4 discusses how our prototype can be extended to handle `dlopen`ed libraries and JIT-generated code. To generate rules semi-automatically, we relied on custom command-line scripts built atop the Egalito binary analysis framework [115] for the standard cases, while 8 irregular cases were resolved by LLM-assisted analysis. Crucially, every generated rule underwent rigorous manual verification, allowing us to guarantee *complete* and accurate static syscall coverage for this `(g)libc` build. Adapting an established rules file for incremental updates requires minimal effort; our empirical testing shows that modern LLMs (e.g., Claude [8]) can extrapolate and port these verified rules across minor version bumps, neutralizing the long-term maintenance burden typically associated with binary rewriting. We found that it took under an hour for a Ph.D. student to build an initial rules file for an unseen `libc`.

6 Evaluation

Testbed. We conducted all evaluations on a host equipped with an 8-core, Intel Core i7-11800H ("Tiger Lake") processor locked at 2GHz, 16GB of DDR4 RAM, and a 1TB NVMe SSD. The system ran Debian 13 ("Trixie") powered by Linux kernel v6.17.1, which we patched to support user-space Intel IBT—shadow stacks are already natively supported. All benchmarked applications were linked against the GNU C Library (i.e., `glibc`) `v2.41-12+deb13u2`. To minimize measurement variance and ensure reproducibility, we placed the host in a quiescent state during benchmarking by disabling the desktop environment and terminating all non-essential background processes. Furthermore, we disabled dynamic voltage and frequency scaling (DVFS) and restricted processor C-states strictly to `C0`.

6.1 Performance

Our performance evaluation of IBTpoline comprises a *custom microbenchmark* and a *suite of real-world application macrobenchmarks*. Across all workloads, we measure the overhead introduced by various IBTpoline and zpoline [119] configurations relative to a native, uninstrumented baseline. Specifically, we evaluate *five* IBTpoline configurations and *two* zpoline configurations:

- **IBTpoline Default**: The default trampoline stub directly executes the syscall; no hook library is loaded.
- **IBTpoline Pre**: Trampoline stubs are rewritten using the *pre-syscall* stub, which invokes an empty hook function immediately before the syscall.
- **IBTpoline Post**: Trampoline stubs are rewritten using the *post-syscall* stub, which invokes an empty hook function immediately after the syscall.
- **IBTpoline Prepost**: Trampoline stubs are rewritten using the *prepost-syscall* stub, invoking an empty hook function both before and after the syscall.
- **IBTpoline Replace**: Trampoline stubs are rewritten using the *replace-syscall* stub, redirecting execution to a unique hook function that manually issues the syscall.
- **zpoline Default**: No hook library is loaded; zpoline dispatches syscalls using its default instrumentation.

Table 3. Macrobenchmark results.

Variant	SQLite	Redis	Nginx		
			1K	100K	1M
IBTpoline Default	0.29%	0.49%	0.44%	0.15%	0%
IBTpoline Pre	0.29%	0.49%	0.44%	0.15%	0.11%
IBTpoline Post	0.29%	0.88%	0.71%	0.15%	0.01%
IBTpoline Prepost	0.29%	0.88%	0.76%	0.15%	0.11%
IBTpoline Replace	0.29%	1.43%	1.11%	0.19%	0.08%
zpoline Default	0%	2.18%	2.85%	0.58%	0.05%
zpoline Replace	0%	2.32%	2.98%	0.58%	0.05%

- **zpoline Replace**: Syscalls are strictly dispatched through an installed hook function that issues the syscall.

Our performance metrics are designed to capture the baseline cost of using IBTpoline and zpoline as interposition frameworks. Because developers can write arbitrarily complex, use-case-specific hooks, evaluating custom hook logic would obscure the underlying system overhead. Therefore, we utilize empty hooks to isolate the framework's architectural cost. The only exceptions are the IBTpoline Replace and zpoline Replace configurations, wherein the hook functions perform the bare minimum work required to execute the syscall and return. Importantly, our evaluation does not compare IBTpoline against lazypoline [56] or K23 [49] since zpoline represents a lower bound in terms of overhead across the three schemes—both lazypoline and K23 perform additional computation(s) beyond just hooking `syscall` instructions in a zpoline fashion.

6.1.1 Microbenchmark. To isolate the fundamental cost of our interposition mechanism, we designed a custom microbenchmark modeled after the methodology presented in lazypoline [56]. We measure the total CPU cycles required to issue a non-implemented syscall 100M times across 5 iterations. Specifically, we target syscall #500; this configuration represents a best-case scenario for zpoline and establishes a strict lower bound on its overhead, as higher syscall numbers minimize the length of the `nop` sled required to reach the hook. Furthermore, issuing a non-implemented syscall deliberately minimizes kernel-side execution time, ensuring our measurements almost exclusively reflect the architectural overhead of the interposition framework.

Relative to the uninstrumented baseline, the IBTpoline Default introduces just 1.49% overhead, whereas zpoline Default imposes 10.69%. These results confirm that even when zpoline is evaluated under its most favorable conditions, IBTpoline *delivers superior performance*—all while providing architectural flexibility and strict compliance with Intel IBT.

6.1.2 Macrobenchmarks. To evaluate IBTpoline and zpoline in realistic deployment scenarios, we benchmarked three widespread applications: a SQL database engine (SQLite v3.51.3 [4]), an in-memory, key-value store (Redis v8.6.3 [3]), and a web server (Nginx v1.30.2 [2]).

As shown in Table 3, IBTpoline imposes marginal overhead atop the uninstrumented baseline (peaking at 1.43%), consistently outperforming zpoline (which reaches up to 2.98%). The configuration and methodology for each macrobenchmark are detailed below. We also verified that all syscalls that were issued by the applications during benchmarking originate from `libc.so`. We executed each benchmark 10 times and took the average. (Overheads listed as 0% indicate measurements < 0.01%.)

SQLite. We evaluated SQLite using its built-in benchmarking tool, `speedtest` [97]: a standalone executable that statically links SQLite's core with a benchmark driver. A single `speedtest` invocation times common relational database operations (e.g., `SELECT`, `INSERT`, `UPDATE`). We used default configurations, with the exception of scaling the relative test size to 1000 and forcing an in-memory database to eliminate disk I/O bottlenecks. Across all 10 iterations, both IBTpoline and zpoline exhibited negligible overhead (< 0.3%).

Redis. We utilized the `memtier_benchmark` [90], a high-throughput traffic generator, to issue concurrent `GET` and `SET` requests against a local Redis instance. Redis was configured to use its default single I/O thread, and it listened exclusively on the loopback device to minimize network latency and maximize CPU saturation. The `memtier_benchmark` client was configured with 2 threads, 128 clients per thread, and a 10 : 1 ratio of `GET` to `SET` requests for 32-byte payloads. Each iteration measured sustained throughput over a 1-minute window. The results in Table 3 demonstrate that IBTpoline incurs minimal throughput degradation (typically < 1%), solidly outperforming zpoline (> 2.1% overhead).

Nginx. To evaluate Nginx, we generated HTTP traffic using `wrk` [1]. We configured Nginx to communicate solely over the loopback interface to prevent network hardware latency from obscuring the interposition overhead. We benchmarked Nginx by serving statically generated, random-data files of three distinct sizes: 1KB, 100KB, and 1MB. Nginx was configured with 8 worker processes for the 1KB and 100KB workloads, and 2 workers for the more memory-intensive 1MB workload. Across all file sizes, `wrk` was configured with 8 threads maintaining 1024 concurrent HTTP connections. Each iteration measured throughput over a 1-minute duration. As detailed in Table 3, IBTpoline introduced negligible overhead across all workloads, peaking at 1.11% for the 1KB file size, whereas zpoline reached up to 2.98%.

6.2 Shadow Trampoline Performance

To quantify the cost of the shadow trampoline (lockdown), we built a microbenchmark that compares IBTpoline's Default stub against the shadow trampoline across a range of shadow-stack walk depths. The benchmark issues an unimplemented syscall (#500) so minimal time is spent in the kernel, isolating the verification cost, which includes the trampoline stub, the (shadow) stack walk, and the `syscall` instruction itself.

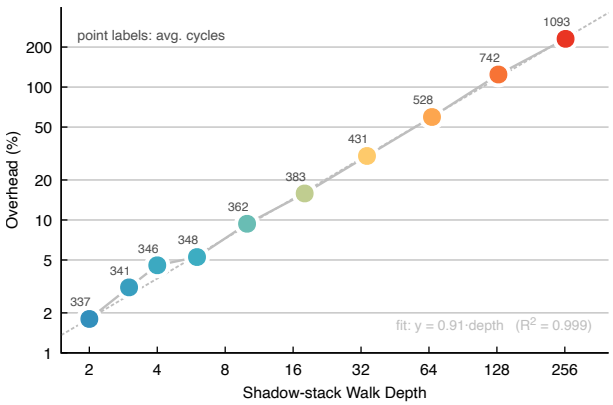


Figure 3. Per-frame shadow-stack walk cost. Overhead scales linearly with walk depth (≈ 0.91% per frame, R^2 = 0.999). Marker color encodes overhead magnitude; numbers above each point are the raw cycle counts.

To control walk depth, a stub invokes a function that recurses to a specified depth before issuing the syscall; we begin timing only after the target depth is reached, so the recursive descent does not litter the measurement. Each depth is measured over 5 runs of 10M walks.

We report the overhead percentage of each shadow-stack walk at various depths compared to IBTpoline's Default stub in Figure 3 and show mean cycles per walk as individual data-point labels. Figure 3 shows that overhead grows linearly in walk depth across the entire tested range. Every additional frame traversed by the verification code adds roughly 3 additional cycles atop the ≈ 331-cycle Default baseline. In practice, we expect the call depth at a syscall site to rarely approach a number as high as 10; even at $depth = 10$ the 9.36% overhead amounts to only ≈ 31 cycles—an overhead easily masked by the subsequent kernel-side syscall handling and any I/O the program performs around it.

6.3 Correctness

Our correctness evaluation of IBTpoline sought to verify two primary criteria: (1) that IBTpoline strictly adheres to Intel IBT hardware-enforcement rules, and (2) that its binary rewriting and syscall interposition mechanisms are precise and preserve original program semantics. To evaluate both criteria simultaneously, we developed a comprehensive syscall test suite that exercises 305 of the 375 syscalls advertised in our (g)libc headers. The majority (56) of the missing syscalls are either unavailable in our environment, require elevated privileges, or are handled by vDSO instead of their libc implementation. Excluding these syscalls raises our coverage to 95.6% (305/319). Our suite manages to hit 270 of the 423 target-able syscall sites (382 target-able sites if we exclude the 56 unavailable syscalls). This lower site coverage arises because libc contains multiple sites for the same syscall, including uncommon paths that are difficult to trigger deterministically.

For example, the evaluated libc contains 70 distinct futex syscall sites. The suite is architected as a collection of modular test cases that group related syscalls to mirror realistic behavior; for instance, our memory test case sequentially invokes mmap, madvise, mincore, and other memory-management operations. We executed this entire test suite across all IBTpoline configurations. In all cases, we successfully verified that no control-flow enforcement faults occurred (i.e., no crashes resulting from missing endbr64 instructions) and that the runtime patching and hooking processes introduced zero application regressions or unintended semantic side-effects.

7 Discussion

7.1 Use Cases

To demonstrate IBTpoline's versatility and API expressiveness, we implemented six hook libraries spanning tracing and profiling, compatibility shimming, and dynamic testing; these collectively exercise all four of IBTpoline's stub variants and are summarized in Table 4.

7.2 Alternate Patching Mechanisms

IBTpoline's main contribution is the *IBT-compliant, per-syscall trampoline* (§4.2) and the syscall-site rewrite that reaches it (§4.1)—not the mechanism that applies the rewrite. Whatever mechanism performs the rewrite—e.g., our load-time approach, an offline binary rewriter, or a compiler—has no bearing on the system's run-time cost: the overheads reported in Section 6.1 are properties of the trampoline design, not of how the bytes got there. The patching mechanism does, however, trade off between deployment friction, applicability to commercial off-the-shelf (COTS) binaries, run-time cost, text-page sharing, re-entrancy, and generality beyond libc.

Load-time Patching. Our prototype (§4.1, §5) has the lowest deployment friction: LD_PRELOAD enables it per process, with no rebuilds and no on-disk changes, making it applicable to any COTS binary. Its cost is text-page sharing: rewriting a shared object in memory triggers copy-on-write (CoW), replacing shared pages with process-private ones. Under our glibc policy, syscall-site rewrites span 64 pages (256KB).

Pre-patched Binaries. Alternatively, patching can be moved offline, with pre-patched binaries loaded via LD_LIBRARY_PATH. This avoids load-time work and lets patched objects stay shared across instrumented processes—but only with a *fixed* trampoline address (IBTPOLINE_ASLR disabled); otherwise .text relocations at every syscall site trigger CoW on the same 64 pages. Additionally, deployment friction is higher: every vanilla libc (or syscall-making binary) needs a separately maintained IBTpoline build, and LD_PRELOAD is still required to install the trampoline absent another initializer. However, given such an initializer, this is the only approach that naturally extends to statically linked programs, where neither LD_PRELOAD nor LD_LIBRARY_PATH applies.

Table 4. Example hook libraries implemented atop `IBTpoline`.

Use Case	Stub	Analog	Hooked Syscalls	Hook Behavior
Syscall tracer	`post`	`strace` [5, 57]	all	Logs each syscall's number, args, and return value.
Syscall profiler	`prepost`	DeBox [93], DTrace [19]	all	Records invocations and kernel compute time.
Path remapping	`pre`	CDE [51]	`open{at}`, `stat`, `access`	Rewrites path arguments before kernel reads.
Fault injection	`replace`	LFI [71]	`read`, `write`, `connect`, `sendto`	Returns a plausible error instead of executing the syscall.
Latency injection	`pre`	Lu et al. [69]	`read`, `write`, `sendto`, `recvmsg`	Sleeps before syscalls, emulating network/disk issues.
Deterministic replay	`post`	RR [76]	`getrandom`	Overwrites syscall's output with fixed constants.

Source-level Patching. Patching code at the source level shares the pre-patched approach's text-sharing advantage and its fixed-address-versus-relocation dilemma, while eliminating the need for stash spots (§4.1). However, these gains merely relocate cost from run time to the binary's development and release pipeline. Further, re-entrancy becomes a bigger problem: to allow hook logic to run against an unpatched dependency—preventing infinite recursion—a source-patched program must either undergo substantial refactoring so hook and instrumented entry points diverge, or retain a second, unpatched copy of the program for hooks to call into. Finally, matching `IBTpoline`'s coverage means repeating the effort for every code base that issues raw syscalls (e.g., `libpthread`, `libstdc++`, language runtimes [29]), and even then source-level patching is not applicable to legacy, third-party, or closed-source binaries.

7.3 vDSO Support

Our current `IBTpoline` prototype does not handle vDSO calls, similar to other syscall tracing frameworks like zpoline [119] and `strace` [5]. However, `IBTpoline` can be extended to handle them in one of two ways: either by disabling the vDSO—which makes `libc` fall back to ordinary `syscall` instructions that `IBTpoline` can rewrite—or by intercepting vDSO fast paths through rewriting linker-resident data (e.g., pointers in `glibc`'s `_rtld_global_ro` or `vdso_func` pointers in `musl`). We leave this for future work.

7.4 Dynamically Loaded and Generated Code

`IBTpoline` rewrites the syscall sites of the main executable and its dependencies at load time (§4.1), but syscall-bearing code can still enter the address space afterwards—e.g., via `dlopen` or JIT compilation. Our prototype handles neither; critically, neither weakens lockdown mode (§4.4): a `syscall` executed from a late-loaded library or a JIT region falls outside the allowed syscall region enforced by `seccomp-bpf`, and thus crashes the program by design. In no scenario does new code silently escape interposition. Neither case is a fundamental limitation, either. Shared objects mapped via `dlopen` can be supported by interposing on `dlopen` itself and applying that library's rules file with NOPoutNG's live-patching scheme [42] before any initializers execute.

Additionally, JIT engines can be made `IBTpoline`-aware by emitting `mov $&ts[nr], %rax; call *%rax` in place of the non-compliant `mov $nr, %rax; syscall`, directly targeting the IBT-compliant stub and preserving lockdown mode's invariants. To achieve this, the engine only needs the trampoline base address, which `ibtpoline.so` can export. In practice, raw syscall emission in JIT-compiled code is exceedingly rare, as most engines wrap system interactions in calls to the already-instrumented runtime or `libc`.

8 Conclusion

We presented `IBTpoline`: a flexible, natively IBT-compliant interposition framework. By rethinking how rewritten `syscall` instructions resolve their branch targets, `IBTpoline` safely redirects execution to localized, `endbr64`-prefixed trampoline stubs. This design cleanly eliminates the need for legacy zero-page mappings and CET exemptions. More importantly, `IBTpoline` transforms CET from an obstacle into a security asset: by combining `seccomp-bpf` with hardware-enforced CET shadow stacks, `IBTpoline`'s lockdown mode guarantees that no syscall can bypass an installed hook. Our evaluation also demonstrates that strict IBT compliance need not come at the expense of performance. Supported by a lightweight runtime of $\approx$ 1.2K LoC in C and an auditable, offline rules-file pipeline, `IBTpoline` provides a robust and verifiable foundation for binary rewriting-based interposition. As CET-enabled hardware and IBT-protected userlands become the ubiquitous default, we believe architectures like `IBTpoline`—which treat hardware-enforced CFI as a foundation to build upon rather than an obstacle to bypass—chart a path forward for high-performance syscall hooking.

Availability

Our `IBTpoline` prototype is available at: https://gitlab.com/brown-ssl/ibtpoline. The evaluated artifact can be found at: https://zenodo.org/records/23201689.

Acknowledgments

We thank our shepherd Ruslan Nikolaev, the anonymous reviewers, and Kex Huang for their valuable feedback. This work was supported by the National Science Foundation (NSF) through award 2238467. Opinions, findings, and conclusions expressed herein are those of the authors and do not necessarily reflect views of the U.S. government or NSF.

References

[1] 2021. wrk – a HTTP benchmarking tool. https://github.com/wg/wrk.

[2] 2023. nginx. https://nginx.org.

[3] 2023. Redis. https://redis.io.

[4] 2023. SQLite. https://www.sqlite.org.

[5] 2026. strace: linux syscall tracer. https://strace.io.

[6] Salman Ahmed, Hans Liljestrand, Hani Jamjoom, Matthew Hicks, N Asokan, and Danfeng Daphne Yao. 2023. Not All Data are Created Equal: Data and Pointer Prioritization for Scalable Protection against Data-Oriented Attacks. In *USENIX Security Symposium (SEC)*. 1433–1450.

[7] Albert D Alexandrov, Maximilian Ibel, Klaus E Schauser, and Chris J Scheiman. 1998. UFO: A Personal Global File System Based on User-Level Extensions to the Operating System. *ACM Transactions on Computer Systems (TOCS)* 16, 3 (1998), 207–233.

[8] Anthropic. 2026. Claude Opus 4.8. https://www.anthropic.com/claude.

[9] Andrea Arcangeli. 2005. seccomp: secure computing support. Linux kernel commit `d949d0ec9c60`.

[10] Marcos Bajo, Ritvik Goyal, Apostolos Chatzianagnostou, and Christian Rossow. 2026. Crashing Through Defenses: Exploiting Segfaults and Chaining around Intel CET. In *IEEE Symposium on Security and Privacy (S&P)*. 4656–4675.

[11] Atri Bhattacharyya, Uros Tesic, and Mathias Payer. 2022. Midas: Systematic Kernel TOCTTOU Protection. In *USENIX Security Symposium (SEC)*. 107–124.

[12] Erik Bosman and Herbert Bos. 2014. Framing Signals—A Return to Portable Shellcode. In *IEEE Symposium on Security and Privacy (S&P)*. 243–258.

[13] Danilo Bruschi, Lorenzo Cavallaro, and Andrea Lanzi. 2007. Diversified Process Replicæ for Defeating Memory Error Exploits. In *IEEE International Performance, Computing, and Communications Conference (IPCCC)*. 434–441.

[14] Alexander Bulekov, Rasoul Jahanshahi, and Manuel Egele. 2021. Saphire: Sandboxing PHP Applications with Tailored System Call Allowlists. In *USENIX Security Symposium (SEC)*. 2881–2898.

[15] Nathan Burow, Scott A Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. 2017. Control-flow Integrity: Precision, Security, and Performance. *ACM Computing Surveys (CSUR)* 50, 1 (2017), 1–33.

[16] Claudio Canella, Mario Werner, Daniel Gruss, and Michael Schwarz. 2021. Automating Seccomp Filter Generation for Linux Applications. In *ACM Cloud Computing Security Workshop (CCSW)*. 139–151.

[17] Canonical Ltd. 2026. Default Compiler and Linker Flags. https://documentation.ubuntu.com/security/security-features/process-memory/compiler-flags/.

[18] Canonical Ltd. 2026. ToolChain/CompilerFlags. https://wiki.ubuntu.com/ToolChain/CompilerFlags.

[19] Bryan Cantrill, Michael W Shapiro, and Adam H Leventhal. 2004. Dynamic Instrumentation of Production Systems. In *USENIX Annual Technical Conference (USENIX ATC)*. 15–28.

[20] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. 2015. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *USENIX Security Symposium (SEC)*. 161–176.

[21] Chandler Carruth. 2024. Speculative Load Hardening. https://llvm.org/docs/SpeculativeLoadHardening.html.

[22] Raymond Chen. 2017. On Generating Sentinel Pointer Values in Windows. https://devblogs.microsoft.com/oldnewthing/20170420-00/?p=96005.

[23] Yaohui Chen, Dongli Zhang, Ruowen Wang, Rui Qiao, Ahmed M Azab, Long Lu, Hayawardh Vijayakumar, and Wenbo Shen. 2017. NORAX: Enabling Execute-only Memory for COTS Binaries on AArch64. In *IEEE Symposium on Security and Privacy (S&P)*. 304–319.

[24] George Christou, Grigoris Ntousakis, Eric Lahtinen, Sotiris Ioannidis, Vasileios P Kemerlis, and Nikos Vasilakis. 2023. BinWrap: Hybrid Protection against Native Node.js Add-ons. In *ACM Asia Conference on Computer and Communications Security (ASIA CCS)*. 429–442.

[25] Neophytos Christou, Alexander J Gaidis, Vaggelis Atlidakis, and Vasileios P Kemerlis. 2024. Eclipse: Preventing Speculative Memory-error Abuse with Artificial Data Dependencies. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 3913–3927.

[26] Emma Connor, Tyler McDaniel, Jared M Smith, and Max Schuchard. 2020. PKU Pitfalls: Attacks on PKU-based Memory Isolation Systems. In *USENIX Security Symposium (SEC)*. 1409–1426.

[27] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. 1998. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *USENIX Security Symposium (SEC)*. 63–78.

[28] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. 2015. Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *IEEE Symposium on Security and Privacy (S&P)*. 763–780.

[29] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P Kemerlis. 2020. Sysfilter: Automated System Call Filtering for Commodity Software. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. 459–474.

[30] Luca Di Bartolomeo, Hossein Moghaddas, and Mathias Payer. 2023. ARMore: Pushing Love Back Into Binaries. In *USENIX Security Symposium (SEC)*. 6311–6328.

[31] Alessandro Di Federico, Amat Cama, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2015. How the ELF Ruined Christmas. In *USENIX Security Symposium (SEC)*. 643–658.

[32] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. 2020. RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization. In *IEEE Symposium on Security and Privacy (S&P)*. 1497–1511.

[33] Ulrich Drepper. 2007. *What Every Programmer Should Know About Memory*. Technical Report.

[34] Jake Edge. 2025. Enhancing FineIBT. https://lwn.net/Articles/1039633/.

[35] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. 2015. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 901–913.

[36] Reza Mirzazade Farkhani, Saman Jafari, Sajjad Arshad, William Robertson, Engin Kirda, and Hamed Okhravi. 2018. On the Effectiveness of Type-based Control Flow Integrity. In *Annual Computer Security Applications Conference (ACSAC)*. 28–39.

[37] Fedora Project. 2026. Changes/HardeningFlags28. https://fedoraproject.org/wiki/Changes/HardeningFlags28.

[38] International Organization for Standardization. 2018. *ISO/IEC 9899:2018, Information technology — Programming languages — C*.

[39] International Organization for Standardization. 2024. *ISO/IEC 9899:2024, Information technology — Programming languages — C*.

[40] Stephanie Forrest, Anil Somayaji, and David H Ackley. 1997. Building Diverse Computer Systems. In *Workshop on Hot Topics in Operating Systems (HotOS)*. 67–72.

[41] Alexander J Gaidis, Vaggelis Atlidakis, and Vasileios P Kemerlis. 2023. SysXCHG: Refining Privilege with Adaptive System Call Filters. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 1964–1978.

[42] Alexander J Gaidis, Jamie Gabbay, Joao Moreira, and Vasileios P Kemerlis. 2025. NOPoutNG: Improving the Effectiveness of Hardware-assisted Control-flow Integrity via Dynamic Landing Pad Elision. In *Annual Computer Security Applications Conference Workshops (ACSAC*

Workshops). 531–545.

[43] Alexander J Gaidis, Joao Moreira, Ke Sun, Alyssa Milburn, Vaggelis Atlidakis, and Vasileios P Kemerlis. 2023. FineIBT: Fine-grain Control-flow Enforcement with Indirect Branch Tracking. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. 527–546.

[44] Tal Garfinkel. 2003. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In *Network and Distributed System Security Symposium (NDSS)*.

[45] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. 2020. Temporal System Call Specialization for Attack Surface Reduction. In *USENIX Security Symposium (SEC)*. 1749–1766.

[46] Seyedhamed Ghavamnia, Tapti Palit, and Michalis Polychronakis. 2022. C2C: Fine-Grained Configuration-Driven System Call Filtering. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 1243–1257.

[47] Adrien Ghosn, Marios Kogias, Mathias Payer, James R Larus, and Edouard Bugnion. 2021. Enclosure: Language-Based Restriction of Untrusted Libraries. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 255–267.

[48] GNU Development Tools. 2026. gdb(1) — Linux manual page. https://www.man7.org/linux/man-pages/man1/gdb.1.html.

[49] Jesús María Gómez Moreno, Vissarion Moutafis, Antreas Dionysiou, Fernando Kuipers, Georgios Smaragdakis, Bart Coppens, and Alexios Voulimeneas. 2025. Clair Obscur: The Light and Shadow of System Call Interposition–From Pitfalls to Solutions with K23. In *ACM/IFIP International Middleware Conference (Middleware)*. 241–255.

[50] Daniel Gruss. 2018. Software-based Microarchitectural Attacks. *it – Information Technology* 60, 5–6 (2018), 335–341.

[51] Philip J Guo and Dawson Engler. 2011. CDE: Using System Call Interposition to Automatically Create Portable Software Packages. In *USENIX Annual Technical Conference (USENIX ATC)*. 247–252.

[52] Petr Hosek and Cristian Cadar. 2013. Safe Software Updates via Multi-version Execution. In *IEEE/ACM International Conference on Software Engineering (ICSE)*. 612–621.

[53] Intel. 2020. A Technical Look at Intel® Control-Flow Enforcement Technology. https://www.intel.com/content/www/us/en/developer/articles/technical/technical-look-control-flow-enforcement-technology.html.

[54] Intel. 2021. Speculative Execution Side Channel Mitigations. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/speculative-execution-side-channel-mitigations.html.

[55] Intel Corporation. 2026. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide.* Section 5.6.2, "Protection Keys".

[56] Adriaan Jacobs, Merve Gülmez, Alicia Andries, Stijn Volckaert, and Alexios Voulimeneas. 2024. System Call Interposition Without Compromise. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 183–194.

[57] Michael B Jones. 1993. Interposition Agents: Transparently Interposing User Code at the System Interface. In *ACM Symposium on Operating Systems Principles (SOSP)*. 80–93.

[58] Joonkyo Jung, Jisoo Jang, Jo Yongwan, Jonas Vinck, Alexios Voulimeneas, Stijn Volckaert, and Dokyung Song. 2025. Moneta: Ex-Vivo GPU Driver Fuzzing by Recalling In-Vivo Execution States. In *Network and Distributed System Security Symposium (NDSS)*.

[59] Gaurav S Kc, Angelos D Keromytis, and Vassilis Prevelakis. 2003. Countering Code-Injection Attacks With Instruction-Set Randomization. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 272–280.

[60] Sharjeel Khan, Bodhisatwa Chatterjee, and Santosh Pande. 2024. Pythia: Compiler-Guided Defense Against Non-Control Data Attacks. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 850–866.

[61] Juhee Kim, Jinbum Park, Yoochan Lee, Chengyu Song, Taesoo Kim, and Byoungyoung Lee. 2024. PeTAL: Ensuring Access Control Integrity against Data-only Attacks on Linux. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2919–2933.

[62] Sungjin Kim, Byung Joon Kim, and Dong Hoon Lee. 2021. Prof-gen: Practical Study on System Call Whitelist Generation for Container Attack Surface Reduction. In *IEEE International Conference on Cloud Computing (CLOUD)*. 278–287.

[63] Taesoo Kim and Nickolai Zeldovich. 2013. Practical and Effective Sandboxing for Non-root Users. In *USENIX Annual Technical Conference (USENIX ATC)*. 139–144.

[64] Hyungjoon Koo, Yaohui Chen, Long Lu, Vasileios P Kemerlis, and Michalis Polychronakis. 2018. Compiler-assisted Code Randomization. In *IEEE Symposium on Security and Privacy (S&P)*. 461–477.

[65] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. 2014. Code-Pointer Integrity. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 147–163.

[66] Donghyun Kwon, Jangseop Shin, Giyeol Kim, Byoungyoung Lee, Yeongpil Cho, and Yunheung Paek. 2019. uXOM: Efficient eXecute-Only Memory on ARM Cortex-M. In *USENIX Security Symposium (SEC)*. 231–247.

[67] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. 2014. SoK: Automated Software Diversity. In *IEEE Symposium on Security and Privacy (S&P)*. 276–291.

[68] Lingguang Lei, Jianhua Sun, Kun Sun, Chris Shenefiel, Rui Ma, Yuewu Wang, and Qi Li. 2017. SPEAKER: Split-Phase Execution of Application Containers. In *International Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*. 230–251.

[69] Ruiming Lu, Yunchi Lu, Yuxuan Jiang, Guangtao Xue, and Peng Huang. 2025. One-Size-Fits-None: Understanding and Enhancing Slow-Fault Tolerance in Modern Distributed Systems. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 359–378.

[70] LWN.net. 2004. x86 NX support. https://lwn.net/Articles/87814/.

[71] Paul D Marinescu and George Candea. 2009. LFI: A Practical and General Library-level Fault Injector. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 379–388.

[72] Matthew Maurer and David Brumley. 2012. TACHYON: Tandem Execution for Efficient Live Patch Testing. In *USENIX Security Symposium (SEC)*. 617–630.

[73] Karol Mazurek. 2025. NULL Pointer Dereference on macOS: Exploitation History. https://afine.com/history-of-null-pointer-dereferences-on-macos.

[74] Steven McCanne and Van Jacobson. 1993. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *USENIX Winter Conference*. 259–270.

[75] Microsoft Docs. 2023. Data Execution Prevention. https://learn.microsoft.com/en-us/windows/win32/memory/data-execution-prevention.

[76] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. 2017. Engineering Record and Replay for Deployability. In *USENIX Annual Technical Conference (USENIX ATC)*. 377–389.

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

[78] Tapti Palit, Jarin Firose Moon, Fabian Monrose, and Michalis Polychronakis. 2021. DynPTA: Combining Static and Dynamic Analysis for Practical Selective Data Protection. In *IEEE Symposium on Security and Privacy (S&P)*. 1919–1937.

[79] Meghna Pancholi, Andreas Kellas, Kostis Kaffes, Steven M Bellovin, Simha Sethumadhavan, and Vasileios P Kemerlis. 2026. Santa: Language Agnostic Automated System Call Policy Learning for Cloud

Microservices. In *International Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*.

[80] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. 2019. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *USENIX Annual Technical Conference (USENIX ATC)*. 241–254.

[81] Richard Patel. 2026. [PATCH 1/7] x86: add userspace IBT config option. Linux Kernel Mailing List (LKML). https://www.spinics.net/lists/kernel/msg6210584.html

[82] Marco Patrignani and Marco Guarnieri. 2021. Exorcising Spectres with Secure Compilers. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 445–461.

[83] PaX Team. 2003. Address Space Layout Randomization. https://pax.grsecurity.net/docs/aslr.txt.

[84] Dinglan Peng, Congyu Liu, Tapti Palit, Pedro Fonseca, Anjo Vahldiek-Oberwagner, and Mona Vij. 2023. uSWITCH: Fast Kernel Context Isolation with Implicit Context Switches. In *IEEE Symposium on Security and Privacy (S&P)*. 2956–2973.

[85] Georgios Portokalidis, Philip Homburg, Kostas Anagnostakis, and Herbert Bos. 2010. Paranoid Android: Versatile Protection for Smartphones. In *Annual Computer Security Applications Conference (ACSAC)*. 347–356.

[86] Sergej Proskurin, Marius Momeu, Seyedhamed Ghavamnia, Vasileios P Kemerlis, and Michalis Polychronakis. 2020. xMP: Selective Memory Protection for Kernel and User Space. In *IEEE Symposium on Security and Privacy (S&P)*. 563–577.

[87] Kailun Qin and Dawu Gu. 2024. One System Call Hook to Rule All TEE OSes in the Cloud. In *IEEE International Conference on Cloud Computing (CLOUD)*. 205–216.

[88] Vidya Lakshmi Rajagopalan, Konstantinos Kleftogiorgos, Enes Göktas, Jun Xu, and Georgios Portokalidis. 2023. SysPart: Automated Temporal System Call Filtering for Binaries. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 1979–1993.

[89] Prabhu Rajasekaran, Stephen Crane, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. 2020. CoDaRR: Continuous Data Space Randomization against Data-only Attacks. In *ACM Asia Conference on Computer and Communications Security (ASIA CCS)*. 494–505.

[90] Redis. 2023. memtier_benchmark. https://github.com/RedisLabs/memtier_benchmark.

[91] André Rösti, Stijn Volckaert, Michael Franz, and Alexios Voulimeneas. 2024. I'll Be There for You! Perpetual Availability in the A8 MVX System. In *Annual Computer Security Applications Conference (ACSAC)*. 520–533.

[92] André Rösti, Alexios Voulimeneas, and Michael Franz. 2024. The Astonishing Evolution of Probabilistic Memory Safety: From Basic Heap-Data Attack Detection Toward Fully Survivable Multivariant Execution. *IEEE Security & Privacy* 22, 4 (2024), 66–75.

[93] Yaoping Ruan and Vivek S Pai. 2004. Making the "Box" Transparent: System Call Performance as a First-Class Result. In *USENIX Annual Technical Conference (USENIX ATC)*. 1–14.

[94] David Schrammel, Samuel Weiser, Richard Sadek, and Stefan Mangard. 2022. Jenny: Securing Syscalls for PKU-based Memory Isolation Systems. In *USENIX Security Symposium (SEC)*. 936–952.

[95] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *IEEE Symposium on Security and Privacy (S&P)*. 745–762.

[96] Sirus Shahini, Mu Zhang, Mathias Payer, and Robert Ricci. 2023. Arvin: Greybox Fuzzing Using Approximate Dynamic CFG Analysis. In *ACM Asia Conference on Computer and Communications Security (ASIA CCS)*. 232–246.

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

[98] System Calls Manual. 2025. ptrace(2) — Linux manual page. https://man7.org/linux/man-pages/man2/ptrace.2.html.

[99] Jakub Szefer. 2019. Survey of Microarchitectural Side and Covert Channels, Attacks, and Defenses. *Journal of Hardware and Systems Security (HaSS)* 3, 3 (2019), 219–234.

[100] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *IEEE Symposium on Security and Privacy (S&P)*. 48–62.

[101] The Linux Kernel. 2023. Seccomp BPF (SECure COMPuting with filters). https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html.

[102] The Linux Kernel. 2026. Control-flow Enforcement Technology (CET) Shadow Stack. https://www.kernel.org/doc/html/next/x86/shstk.html.

[103] The Linux Kernel. 2026. Syscall User Dispatch. https://docs.kernel.org/admin-guide/syscall-user-dispatch.html.

[104] User Commands. 2026. ltrace(1) — Linux manual page. https://www.man7.org/linux/man-pages/man1/ltrace.1.html.

[105] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *USENIX Security Symposium (SEC)*. 1221–1238.

[106] Valve Corporation. 2026. Proton. https://github.com/ValveSoftware/Proton.

[107] Victor Van der Veen, Nitish Dutt-Sharma, Lorenzo Cavallaro, and Herbert Bos. 2012. Memory Errors: The Past, the Present, and the Future. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. 86–106.

[108] Jonas Vinck, Adriaan Jacobs, Alexios Voulimeneas, and Stijn Volckaert. 2025. Divide and Conquer: Introducing Partial Multi-Variant Execution. In *IEEE European Symposium on Security and Privacy (EuroS&P)*. 1049–1066.

[109] Stijn Volckaert, Bart Coppens, and Bjorn De Sutter. 2015. Cloning Your Gadgets: Complete ROP Attack Immunity with Multi-Variant Execution. *IEEE Transactions on Dependable and Secure Computing (TDSC)* 13, 4 (2015), 437–450.

[110] Alexios Voulimeneas, Dokyung Song, Per Larsen, Michael Franz, and Stijn Volckaert. 2021. dMVX: Secure and Efficient Multi-Variant Execution in a Distributed Setting. In *European Workshop on Systems Security (EuroSec)*. 41–47.

[111] Alexios Voulimeneas, Dokyung Song, Fabian Parzefall, Yeoul Na, Per Larsen, Michael Franz, and Stijn Volckaert. 2020. Distributed Heterogeneous N-Variant Execution. In *International Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*. 217–237.

[112] Alexios Voulimeneas, Jonas Vinck, Ruben Mechelinck, and Stijn Volckaert. 2022. You Shall Not (by)Pass! Practical, Secure, and Fast PKU-based Sandboxing. In *European Conference on Computer Systems (EuroSys)*. 266–282.

[113] Xiaoguang Wang, SengMing Yeoh, Pierre Olivier, and Binoy Ravindran. 2020. Secure and Efficient In-process Monitor (and Library) Protection with Intel MPK. In *European Workshop on Systems Security (EuroSec)*. 7–12.

[114] David Williams-King, Graham Gobieski, Kent Williams-King, James P Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P Kemerlis, Junfeng Yang, and William Aiello. 2016. Shuffler: Fast and Deployable Continuous Code Re-Randomization. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 367–382.

[115] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P Kemerlis. 2020. Egalito: Layout-agnostic Binary Recompilation. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 133–147.

[116] WineHQ. 2026. WineHQ: Run Windows applications on Linux, BSD, Solaris and macOS. https://www.winehq.org.

[117] Yunlong Xing, Jiahao Cao, Kun Sun, Fei Yan, and Shengye Wan. 2022. The Devil is in the Detail: Generating System Call Whitelist for Linux Seccomp. *Future Generation Computer Systems (FGCS)* 135 (2022), 105–113.

[118] Yunlong Xing, Xinda Wang, Sadegh Torabi, Zeyu Zhang, Lingguang Lei, and Kun Sun. 2023. A Hybrid System Call Profiling Approach for Container Protection. *IEEE Transactions on Dependable and Secure Computing (TDSC)* 21, 3 (2023), 1068–1083.

[119] Kenichi Yasukata, Hajime Tazaki, Pierre-Louis Aublin, and Kenta Ishiguro. 2023. zpoline: a system call hook mechanism based on binary rewriting. In *USENIX Annual Technical Conference (USENIX ATC)*. 293–300.

[120] Sengming Yeoh, Xiaoguang Wang, Jae-Won Jang, and Binoy Ravindran. 2024. sMVX: Multi-Variant Execution on Selected Code Paths. In *ACM/IFIP International Middleware Conference (Middleware)*. 62–73.

[121] Dongyang Zhan, Zhaofeng Yu, Xiangzhan Yu, Hongli Zhang, Lin Ye, and Likun Liu. 2022. Securing Operating Systems Through Fine-Grained Kernel Access Limitation for IoT Systems. *IEEE Internet of Things Journal (IoT-J)* 10, 6 (2022), 5378–5392.

[122] Zhiyuan Zhang, Gilles Barthe, Chitchanok Chuengsatiansup, Peter Schwabe, and Yuval Yarom. 2023. Ultimate SLH: Taking Speculative Load Hardening to the Next Level. In *USENIX Security Symposium (SEC)*. 7125–7142.

[123] Zhechang Zhang, Hengkai Ye, Song Liu, and Hong Hu. 2026. SACK: Systematic Generation of Function Substitution Attacks against Control-flow Integrity. In *Network and Distributed System Security Symposium (NDSS)*.