

USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Secpoline: A Scalable Approach to Build Secure In-Process Syscall Interposers

Ruben Sturm and Anton Schelfhout, *DistriNet, KU Leuven;* Merve Gülmez, *Ericsson Security Research;* Adriaan Jacobs and Stijn Volckaert, *DistriNet, KU Leuven*

https://www.usenix.org/conference/usenixsecurity26/presentation/sturm

This paper is included in the Proceedings of the 35th USENIX Security Symposium.

August 12–14, 2026 • Baltimore, MD, USA

ISBN 978-1-939133-58-8

Open access to the Proceedings of the 35th USENIX Security Symposium is sponsored by



جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

ARTIFACT EVALUATED
USENIX
AVAILABLE

Secpoline: A Scalable Approach to Build Secure In-Process Syscall Interposers

Ruben Sturm
DistriNet, KU Leuven

Anton Schelfhout
DistriNet, KU Leuven

Merve Gülmez
Ericsson Security Research

Adriaan Jacobs
DistriNet, KU Leuven

Stijn Volckaert
DistriNet, KU Leuven

Abstract

In-process system call interposers are increasingly used to extend or monitor application functionality without context-switching or inter-process communication (IPC) overhead. However, when embedded inside untrusted applications, existing solutions face a trade-off: they either enforce no isolation at all, or impose severe restrictions on the monitor's isolated programming environment that are incompatible with complex, real-world use cases.

This paper presents Secpoline, a new interposition platform that allows in-process monitors to support arbitrary interposer functionality without compromising isolation. Secpoline achieves this via isolated multi-program loading and a novel *meta-monitor* design that interposes the monitor itself to emulate a seamless programming environment. In addition, Secpoline implements the first fully self-contained in-process sandbox to harden its own isolation primitive, while preserving fast-path syscall interposition with zero kernel involvement.

Our evaluation shows that Secpoline matches the efficiency of state-of-the-art secure in-process interposers while enabling a significantly more expressive programming environment. We specifically demonstrate this by implementing a kernel-module-free version of the Falco intrusion detection engine. We also implement an in-process ProxySQL sidecar, embedded directly into the application process, where Secpoline transparently provides a kernel-bypass communication path to increase throughput by roughly 50%. These results confirm that Secpoline finally delivers on the promise of secure, complex application monitoring at in-process speeds.

1 Introduction

System call interposition is a fundamental mechanism in modern systems security and observability. It underpins various use cases, from sandboxing [1–6] and intrusion detection [7,8] to compatibility layers [9–16] and debugging tools [17, 18]. Traditionally, these interposers operated across process boundaries (e.g., via `ptrace` [19]) as separate user-space processes, providing a familiar programming environment that supported expressive interposer functionality. However, the performance overhead of context- and mode-switching has become a prohibitive bottleneck for syscall-intensive workloads [20, 21].

Hence, the state of practice has shifted towards *in-process* interposition. By moving the monitor directly into the application's address space, interception can fully bypass the kernel, thereby eliminating mode switches and delivering much lower latency for high-throughput networking and data processing workloads [20–22]. These interposers can be invoked via symbol overloading or binary rewriting [20, 21, 23, 24] and are already integral to modern data plane development [22].

However, moving the monitor in-process introduces severe security and usability challenges. Unlike cross-process architectures, where the kernel enforces strong isolation, in-process monitors share a single address space and process context with the application. This leads to functional interference issues (shared signal handling, synchronization, etc.) which can generally be resolved through compatibility mechanisms such as linker namespaces [20]. However, it also leads to security issues, as the application can not only directly corrupt the monitor's state, but any remaining shared resources between them, such as the dynamic linker, can also be leveraged by the application to indirectly influence or disable the monitor, as we demonstrate in Section 4. This is a real threat to interposers operating in adversarial environments, mediating access for applications that may be vulnerable, compromised, or inherently untrusted [1,2,5,7,8,25,26]. Existing solutions thus push the monitor away to a separate process [1,5,7,8], or into a higher privilege ring [1,26,27], e.g., using `seccomp` [2], to maintain isolation. However, this simply re-introduces known performance and usability issues: cross-process monitors are too slow for modern syscall-intensive workloads, virtualization-based monitors introduce interposer programmability limitations via custom Application Binary Interfaces (ABIs) [27,28], and kernel extensions face a difficult trade-off between expressiveness [23, 29] and Trusted Computing Base (TCB) impact [3,30].

Recent secure in-process monitoring proposals instead aim

to eliminate the need for sharing resources between the monitor and the application [24, 31–34]. Although this has proven effective for security [34], it isolates the interposer from many resources it would need to implement more complex functionality [33, 34]. Modern interposition use cases have evolved beyond simple system-call filtering; they increasingly resemble "sidecars"—auxiliary software components that provide rich functionality such as encryption, protocol parsing, policy enforcement, or performance optimizations (e.g., service meshes, user-space networking stacks). Supporting such functionality requires the monitor to run complex, pre-existing codebases that depend on standard libraries (libc), threading, dynamic memory allocation, etc. This is impossible in current state-of-the-art secure in-process syscall interposers [33, 34], which create a restrictive interposer programming environment without access to such features—effectively presenting a custom ABI. This forces developers to reimplement basic system infrastructure from scratch, hindering the deployment of the use cases that motivated the move to in-process execution in the first place.

This paper aims to solve this dilemma. We present *Secpoline*, the first practical, in-process interposition platform that provides robust isolation while supporting complex, unmodified monitor code. Secpoline achieves this via a new interposer architecture that combines a custom program loader with a novel meta-monitor design, which helps to restore familiar POSIX compatibility to arbitrary interposer code without depending on shared application state.

In summary, our contributions are the following:

- We present **Secpoline**, an open-source implementation of a system-call interposition framework based on the novel meta-monitor architecture we describe in Section 5. Secpoline offers strong security guarantees without imposing functionality constraints on the interposers.
- We evaluate Secpoline on standard benchmarks and find that it incurs minimal overhead, even on syscall-intensive applications (Section 7), matching the state of the art.
- We apply Secpoline to a real-world interposer use cases (ProxySQL, Falco) to demonstrate its ability to integrate complex code without compromising security (Section 8), and show that this was previously infeasible. Secpoline improves throughput by 50% relative to the standard cross-process setup by providing a transparent kernel-bypass path for high-speed communication.

The open-source Secpoline implementation is available at http://github.com/lazypoline/secpoline and at [35].

2 Background

Memory Protection Keys (MPK) are a hardware-supported mechanism for enforcing access permissions on user-space memory without requiring kernel transitions [36]. MPK partitions memory into a small number of protection domains, with access rights controlled via a thread-local register. Because permissions can be updated directly from user space, MPK enables fast domain switches and is widely used as an in-process isolation primitive [3, 6, 24, 32, 34].

The unprivileged nature of MPK permission changes also introduces security risks. In particular, attackers may attempt to bypass isolation by executing instructions that modify MPK permissions (e.g., `WRPKRU`), or by exploiting the kernel as a confused deputy [37]. As a result, MPK-based isolation mechanisms typically rely on sandboxes that restrict permission changes and mediate sensitive system calls [3, 5, 6]. These sandboxes typically rely on a combination of instruction filtering, hardware breakpoints, system call interposition, and temporary page-granular permission changes [5].

While prior work has demonstrated that MPK sandboxes can provide strong isolation guarantees, most existing designs rely on cross-process monitors or kernel support to enforce these restrictions [5, 38]. This reliance has important performance and scalability implications, which we will revisit in later sections.

System call interposition enables a monitor to observe or mediate interactions between an application and the Operating System (OS). Interposers can be broadly classified based on where the interception occurs.

Cross-process interposers run the monitor in a separate process from the monitored application, relying on mechanisms such as `ptrace` [19]. They provide strong isolation guarantees and a familiar programming environment for the monitor, but incur substantial performance overhead due to frequent context switches and inter-process communication. To mitigate this cost, some systems selectively forward only security-relevant system calls to the monitor [5, 8], while allowing "safe" syscalls to execute directly. Unfortunately, these unsafe system calls may still dominate execution in syscall-intensive workloads.

In-process interposers eliminate these overheads by redirecting syscalls directly to a monitor executing within the application's address space, typically via source or binary rewriting or symbol interposition [20, 23, 34]. This enables bypassing the kernel entirely and improves performance for high-throughput workloads. However, naive in-process interposers provide no isolation. A compromised application can directly tamper with the monitor's code or state.

Secure in-process interposers attempt to combine the efficiency of in-process interposition with isolation mechanisms such as MPK [32, 33, 38]. While these systems provide security guarantees comparable to cross-process solutions, existing designs either rely on kernel modifications, impose severe restrictions on the monitor's programming environment, or fail to fully eliminate implicit state sharing with the application. As a result, no existing system simultaneously achieves strong isolation, high performance, and support for complex, unmod-

ified monitor code. Secpoline aims to fix this problem.

3 Threat Model

Our threat model assumes that attackers may fully control the monitored application, either because the application has been compromised or because it is malicious by design (e.g., in a malware sandbox). The attackers' aim is to tamper with, leak data from, or bypass the monitor. We consider a strong attacker, who may want to inject, generate, and execute arbitrary code, alter the application's control flow at will, and read or write application memory to bypass the interception mechanism.

We assume that our implementation, the underlying OS, and the hardware are free of bugs. We also assume that the system starts in a benign, uncompromised state, that our defense mechanism is correctly loaded and initialized (e.g., via our Secpoline loader), and that any code integrated into the Secpoline interposer is free of exploitable bugs. Additionally distrusting this code is out of scope of this work.

Finally, we assume the system exposes a hardware isolation mechanism. Our prototype uses MPK, but our design could also be extended to other mechanisms [39].

Micro-architectural and side-channel attacks are considered out of scope. We also do not aim to provide an availability guarantee against malicious code, i.e., we do not protect against Denial-of-Service attacks.

All of our assumptions are consistent with those made in previous work [3, 5, 6, 34].

4 Motivation

The primary motivation for in-process interposition is performance, but real-world applications also demand scalability and monitoring expressiveness. The system call monitor must be able to execute complex logic without incurring prohibitive overhead. Supporting such use cases typically requires integrating large, pre-existing codebases into the monitor. These codebases depend on standard runtime features such as libc, dynamic memory allocation, threading, and dynamic loading.

At the same time, secure in-process interposition faces a fundamental constraint. Unlike cross-process monitors, an in-process monitor shares an address space with the untrusted application. To provide strong security guarantees under an adversarial threat model, any memory or code shared between the monitor and the application must be explicitly defined, minimal, and treated as adversarial input. Implicit sharing of runtime state and resources violates this requirement by allowing the application to indirectly influence the monitor's execution and undermine its isolation guarantees. Existing secure in-process interposers adopt one of two architectural approaches to address this tension: (i) loading the monitor as a shared library within the application, whilst explicitly isolating its dependencies from the application, or (ii) implementing the monitor as a statically linked executable that avoids exposing these dependencies to the application altogether. Both approaches fundamentally sacrifice either security or scalability, as we explain below.

4.1 Monitor as a shared library

A common way to deploy an in-process monitor is to load it as a shared library, for example, using `LD_PRELOAD` or similar mechanisms [8, 20, 23, 24, 31, 32, 38]. While convenient, this approach causes the monitor and application to share large parts of the runtime environment. Using linker namespaces can reduce direct sharing of libraries [20, 23, 40], but it does not eliminate shared runtime state entirely. The dynamic linker [41] and its associated metadata, such as the thread control block, the Procedure Linkage Table (PLT), the link map, and environment variables, remain shared. A compromised application could overwrite this metadata to change the control flow and resolution decisions within the monitor, possibly reducing its effectiveness or even disabling it entirely.

Concretely, the **thread control block** contains a dynamic thread vector that manages thread-local storage (TLS) variables [42]. While using linker namespaces ensures TLS variables are isolated from other threads, the dynamic thread vector itself remains shared. A compromised application could overwrite the dynamic thread vector to bind the monitor's TLS variables to different storage locations, thereby changing their values without overwriting them directly. The **PLT** contains addresses of dynamically bound symbols [43]. Using separate linker namespaces for the application and monitor ensures they use different PLTs. Unfortunately, the linker metadata that describes the PLT remains accessible to the application and could be tampered with to force the monitor to bind unintended or even malicious symbols. Similar attacks are possible on the **link map** [44]. Finally, the **environment variables** remain shared between the application and monitor. Prior work demonstrated several attacks that tamper with the monitor by manipulating these variables [38]. Crucially, identifying and isolating all such shared state is brittle and non-portable across library versions and runtime implementations. Consequently, monitors implemented as shared libraries cannot provide strong isolation guarantees under an adversarial threat model.

4.2 Monitor as a static executable

To avoid implicit state sharing, some secure in-process interposers implement the monitor as a statically linked executable [34]. This approach significantly strengthens isolation, but does so at the cost of scalability and monitor expressiveness. In particular, statically linked monitors cannot easily support dynamic code loading or standard runtime features such as TLS, which is typically managed by a process-wide mechanism controlled by the application. Furthermore, this ap-

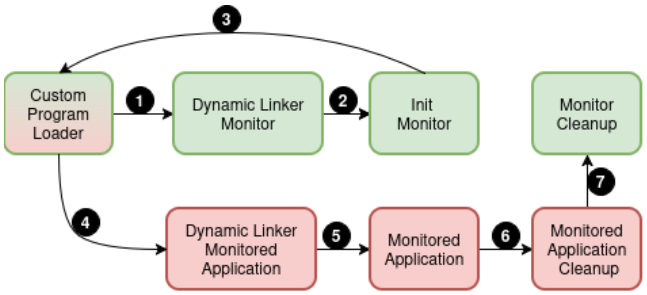


Figure 1: Secpoline program loader control-flow. All the trusted components are indicated in green, and the untrusted components are marked in red.

proach prevents the monitor from utilizing precompiled shared libraries, since shared objects cannot be statically linked into a static executable, preventing the reuse of existing binary-only libraries. As a result, large classes of existing libraries and runtime functionality become nigh impossible to use safely. This forces developers to reimplement substantial parts of the software stack and often restricts them to a highly specialized programming environment, limiting practicality and adoption.

5 Design

To address the challenges discussed in Section 4, we propose a new architecture for in-process monitors that supports securely integrating existing code. We identify two requirements that the architecture must satisfy to ensure scalability and security.

R1 The architecture must strictly limit shared memory and shared code between the monitor and the monitored application, as uncontrolled sharing undermines the assumptions required for strong security guarantees.

R2 The architecture must ensure that integrated code does not undermine the monitor's security guarantees, even if that code is not explicitly designed to operate in an in-process monitoring environment.

To satisfy both requirements, our approach comprises three components. First, a custom program loader loads the monitor and the monitored application as two separate executables, each with its own dynamic linker and set of dynamic libraries, while sharing a single process (Section 5.1). Second, a monitor component intercepts the application's system calls and enforces a base set of system-call filter rules to virtualize or isolate shared resources such as the stack, thread-local storage (TLS), and signal handlers (Section 5.2). Third, a simple meta-monitor component that intercepts system calls made by the monitor itself to preserve monitor awareness even for code not designed with this architecture in mind, enabling the execution of unmodified existing code (Section 5.3).

5.1 Loading the Monitor

To ensure that the monitor and the monitored application are loaded independently within the same process, we implement a custom program loader that loads each executable separately. For dynamically linked executables, this involves loading the executable's dynamic linker, which then acts as the program loader by loading and initializing the executable itself. This design allows us to support arbitrary monitored applications, regardless of their choice of dynamic linker. This process is illustrated in Fig. 1.

The loading process starts by loading the monitor's dynamic linker into memory ❶, which then loads the monitor and calls its initializer ❷. At this point, the monitor is entirely set up, and starts intercepting any subsequent system calls, including those made while loading the application. This ensures that the monitor has a comprehensive view of the application's resources and system calls from the start [38], without relying on custom or implementation-defined load ordering.

After the monitor has been initialized, control returns to the custom program loader ❸. The loader then loads the application's dynamic linker ❹, which subsequently initializes and starts the application ❺. During application execution, the monitor transparently intercepts all system calls and tracks shared resources in order to virtualize or isolate them.

When the application is ready to terminate, it calls the `exit_group` system call. We intercept this system call to clean up the monitor ❼ before actually invoking the `exit_group` system call and terminating the program.

A subtle challenge arises from the loader's role after monitor initialization. Once the monitor has been fully initialized, control returns to the loader to initialize the monitored application, including invoking its dynamic linker. Any system calls issued during this phase are part of the application's execution and must therefore be intercepted and mediated by the monitor. Consequently, after monitor initialization, the loader must be treated as part of the monitored application for the purpose of system call interception.

This transition introduces a critical isolation requirement. Memory allocated by the loader while loading the monitor's dynamic linker must remain isolated from the application. However, once the loader is treated as application code, it can no longer be trusted to access such memory. To preserve the monitor's security guarantees, we must therefore precisely identify and isolate all loader-allocated memory that is relevant to the monitor before transitioning the loader into the application domain.

5.2 Memory Isolation

Using separate loaders for the monitor and application ensures that each receives its own copy of any common dependencies, such as glibc, which already separates global data and provides them with different heap allocators. Still, the stack and TLS

are implemented using singular process-wide resources and must therefore be duplicated, managed, and isolated by the monitor. On x86_64, these resources are addressed via the `%rsp` and `%fs` registers, respectively, requiring the monitor to maintain its own copies and to switch it explicitly when entering and exiting the monitor.

To operate independently on each thread, the monitor must retain per-thread pointers to its own stack and TLS during application execution. However, because the application's TLS is untrusted at that point, storing monitor state there would violate **R1** by allowing the application to overwrite it. To avoid this, we introduce a separate, monitor-specific TLS that stores all thread-local state required for monitor functionality.

The monitor switches the `fs` register and stack pointer between the monitor's and the application's contexts when entering and exiting the monitor. In addition, the monitor maintains a separate `gs`-based TLS that is never swapped out during monitor transitions and holds monitor metadata , which is retrieved during monitor entries.

The `%gs` register is the only monitor resource that remains accessible during application execution. As such, it must be protected from overwrites by the application. While updates via the `arch_prctl` system call can easily be blocked, the `wrgsbase` instruction enables direct writes that require additional enforcement. We therefore employ ERIM's instruction filtering mechanism combined with hardware breakpoints [5, 6], to detect and block execution of the `wrgsbase` instruction [3].

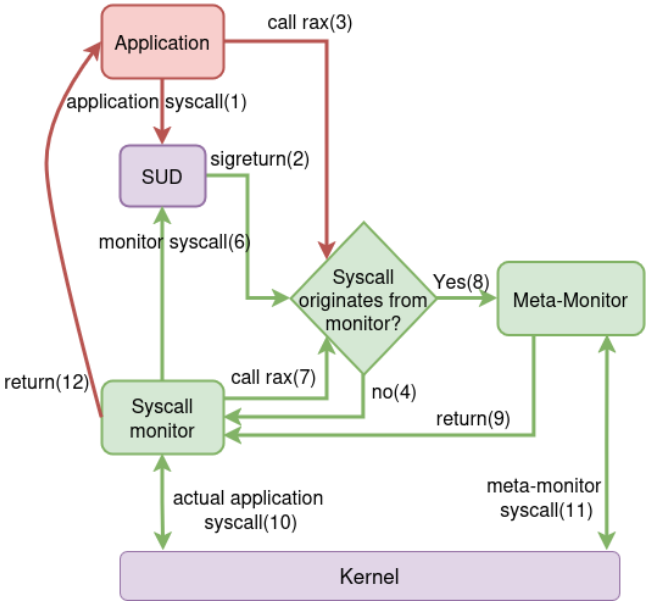


Figure 2: Secpoline system call interposition control-flow. All the trusted components are indicated in green, and transitions between trusted components are marked with a green arrow. The untrusted component is marked in red, transitions from the untrusted application to the trusted monitor and vice versa are marked in red. All kernel components are marked in purple.

5.3 Monitor Syscall Handling

Supporting complex monitor code requires not only applying system call filter rules to the calls issued by the monitored application, but also enforcing a separate set of rules on the monitor itself. This is necessary, e.g., to scan pages for unsafe instructions, or tag them with the correct MPK key (cfr. Section 6.2). To support integrated code that was not originally designed for the interposer environment, it is not a scalable option to manually instrument or redirect all syscalls [23] to implement these rules. Instead, we apply the monitor logic to itself: we execute native syscalls, but reuse our existing syscall interception mechanism to redirect execution to a *meta-monitor*, which implements the rules in a centralized place for all monitor syscalls. Syscalls made by the meta-monitor itself must be manually instrumented, but it has far fewer than the more complex application monitor and contains no interposer-unaware code. Hence, our design effectively centralizes the typical syscall constraints of the secure in-process monitor to a manageable location via the meta-monitor abstraction.

In total, the monitor must now correctly identify and handle three types of system calls: application system calls made by the application, application system calls made by the monitor, and monitor system calls.

Application system calls made by the application are the system calls executed by the untrusted application that need to be intercepted and redirected to the monitor to enforce any filter rules. When a `syscall` instruction is executed for the first time, it follows the slow path (shown by ❶ on fig. 2), where syscall user dispatch (SUD) intercepts it and forwards it to the monitor, which then activates the rewriting mechanism, so that subsequent executions of that instruction follow the fast path ❸, directly entering the monitor.

Application system calls made by the monitor are system calls issued by the monitor on behalf of the application after vetting and approving them, e.g., mapping untrusted pages in response to the application's use of `mmap`. They do not require additional interposition. In the fast path, this case would follow ❸, ❹, ❿, ⓬ on Figure 2.

Monitor system calls are system calls issued by the monitor to support its own operation and enforce monitor awareness to monitor unaware code, such as preventing the `SIGSYS` signal from being blocked by `sigprocmask`, as this would interfere with SUD. These calls are intercepted and forwarded to the meta-monitor, a minimal component responsible for applying monitor-level filtering rules. Once again, the first time a `syscall` instruction is called, it will use the slow path

(❻, ❷, ❽, ⓫, ❾) after which it is rewritten to the fast path for the following invocations (❼, ❽, ⓫, ❾).

6 Implementation of Secpoline

We implement our design in Secpoline, a prototype for x86-64 Linux written in 2231 lines of C, 2614 lines of C++, and 924 lines of x86-64 assembly. In this section, we describe how we leverage MPK to enforce isolation between the monitor and the monitored application. In particular, we focus on a novel MPK permission policy and the implementation of the first fully in-process MPK sandbox, which together ensure that the monitor can securely execute integrated code while maintaining strong isolation guarantees.

We utilize three MPK keys: one for application memory, one for monitor memory that is inaccessible to the application, and one for memory that is read-only from the application's perspective. During Secpoline initialization, we assign these keys by inspecting `/proc/self/maps` to identify all monitor memory mappings present at that time. Subsequently, the monitor and meta-monitor correctly assign MPK keys to any pages that are mapped at runtime.

6.1 A Paranoid MPK Permission Policy

Unlike traditional secure in-process monitors [32, 33], Secpoline strictly follows the principle of least privilege. By default, the monitor can only access its own memory and must explicitly switch its MPK privileges to access untrusted memory when necessary. This choice was inspired by the Supervisor Mode Access Prevention (SMAP) CPU feature that Linux uses to prevent accidental or illicit accesses to user-space memory from the kernel [36]. To facilitate safe interactions with untrusted memory, Secpoline provides an interface to copy data from an untrusted buffer into trusted memory, similar to the Linux kernel's `copy_from_user` [45] interface. At no point are the application MPK domain and the monitor MPK domain simultaneously accessible.

This design has a number of advantages. First, if accesses to untrusted memory from the monitor can only occur when permissions have been manually set to allow them, we can prevent such accesses from occurring at unintended locations, as a defense-in-depth measure that prevents the untrusted application from easily abusing bugs in the monitor, e.g. preparing a ROP chain in untrusted memory [46].

Second, by restricting monitor access solely to memory regions explicitly marked as trusted, we prevent *accidental* sharing of state with the application, e.g., through any unknown process resources that we did not explicitly duplicate or handle. As we explained in Section 4.1, any such resources can present an avenue for malicious applications to tamper with monitor execution. Our unique domain access policy instead turns such overlooked resources into detectable run-time application crashes, which can securely be handled by quitting the program or attempting to rewind to a safe point [47]. This builds confidence in the comprehensiveness of Secpoline's isolation guarantee: by default, no memory that the Secpoline monitor is able to access has ever been written or read by the untrusted application.

Finally, disjoint permissions create a mechanism for Secpoline to efficiently verify which domain a particular page belongs to. Since it is not possible to directly read the domain associated with a memory page, we instead restrict access to the expected MPK domain alone and then access the relevant page. If it belonged to a different domain, the access triggers a segmentation fault, which we detect. In the common case, e.g., when validating that application syscall arguments do not refer to monitor memory, this provides a significant speedup over standard memory-range comparisons (see Section 7).

Pointer Validation. Some system call arguments contain pointers to user-space memory, either to read from, as in the buffer used by the `write` system call, or to write to, as in the buffer used by the `read` system call. Such pointers can be exploited by using the monitor or kernel as a confused deputy to read from or write to trusted memory, thereby bypassing MPK protections [37].

We extend our paranoid permission policy to ensure that the kernel cannot read from or write to trusted memory while executing application system calls made by the monitor. We do this by configuring the `pkru` register to allow access only to untrusted memory for the duration of the system call execution. As a result, if a system call argument points to trusted memory, the call fails with an `EFAULT` error.

6.2 Self-Contained In-Process MPK Sandbox

MPK sandboxes are required to securely use MPK as an isolation primitive, preventing the monitored application from bypassing monitor memory protections, for example, by executing malicious `WRPKRU` instructions. To enforce these protections, the sandbox interposes system calls and ensures that attempts to modify MPK permissions or access monitor memory are mediated. This design allows the monitor to protect itself using the MPK sandbox while simultaneously using the monitor to intercept system calls. In contrast, existing approaches either rely on kernel modifications [32], which increase the trusted code base, or on cross-process monitors [5], which incur significant performance overhead. By combining self-protection with efficient syscall interception, our MPK sandbox implementation provides a lightweight and high-performance method for enforcing memory isolation within the same process.

A critical component of the MPK sandbox is preventing unauthorized `WRPKRU` and `XRSTOR` instructions, as these can be exploited to bypass memory isolation. Previous approaches like ERIM [3] rely on binary scanning to locate and rewrite such instructions, but this technique has been shown to be incomplete [5]. Instead of rewriting instructions, hardware

breakpoints can be used to trap execution before the instruction is executed [5,6]. Hardware breakpoints are limited in number, so Voulimeneas et al. [5] move them dynamically by marking pages containing unsafe instructions as non-executable. When the application attempts to execute such a page, the monitor places breakpoints on all contained unsafe instructions before making the page executable. To avoid scanning the entire code region at startup, all code pages are generally marked as non-executable during program loading.

Secpoline's Sandbox. We implement a similar mechanism to Voulimeneas et al. [5] (see Section 2), but with a key difference: our sandbox is implemented fully in-process, in contrast to their existing implementation, which uses a ptracer to intercept syscalls and set breakpoints. Specifically, we use the `perf_event_open` system call to set hardware breakpoints without leaving the process, eliminating the overhead and complexity of a separate tracer (see Section 7). This in-process design presents unique challenges, as the complete monitor code is mapped, executable, and available for attackers to use gadgets in. Hence, it is subject to the same protections as the application code, including hardware breakpoints and the revocation of executable permissions. We make one exception: one monitor code region always remains executable, since it contains the start of our segmentation fault handler until the syscall that marks the offending page executable. Otherwise, handling one segmentation fault would recursively trigger another, leading to an infinite segmentation fault loop.

Consequently, all unsafe instructions on these pages must retain their breakpoints. We therefore distinguish between static breakpoints, which are fixed and cannot be relocated, and dynamic breakpoints, which may be moved at runtime. This distinction introduces additional constraints on the number of available dynamic breakpoints, as only those not reserved for static breakpoints remain available. However, since only a small, well-defined portion of the code requires static breakpoints, this limitation does not pose a practical issue. Furthermore, we could manually implement this critical section in assembly, or apply gadget-free compilation techniques to break the unaligned instruction stream [48].

Fork Handling. The `perf_event_open` system call provides an option to inherit breakpoints across `fork` or `clone`, causing the child process to start with breakpoints at the same locations as the parent. However, hardware breakpoints set via `perf_event_open` are associated with a performance event, which is shared rather than copied when inherited. As a result, it is not possible to remove breakpoints independently in the child or parent, as removing a breakpoint affects both. To avoid this limitation, we do not inherit breakpoints. Instead, we manually set breakpoints when the child begins execution, creating a separate underlying performance event that can be closed independently in the child or parent.

Stop The World. A peculiarity appears in multi-threaded code. Hardware breakpoints are thread-local, so setting one in a multi-threaded program requires synchronizing all threads and updating each thread's breakpoints before making the page executable. Voulimeneas et al. [5] propose to use instruction emulation from a ptracer instead, which allows the underlying page to remain non-executable, while the monitor can inspect each instruction on the page before execution. While this case did not trigger in the workloads evaluated by Voulimeneas et al., it introduces significant complexity and would incur significant performance overhead.

In practice, however, moving breakpoints is rare in real-world applications. Therefore, we adopt a stop-the-world strategy, in which all threads synchronize to place breakpoints on a page before it is marked executable, ensuring correctness in multithreaded applications. The advantage is that, on the next execution of the same page, the breakpoints are already set correctly, while the instruction emulation approach would need to emulate again, *ad infinitum*, since the breakpoints never move anymore. Our workloads *do* trigger this case, but Section 7 confirms that the overhead is negligible when amortized over a non-trivial program run.

6.3 Signal Handling

To safely support untrusted application signal handlers, we interpose the `sigaction` system call to wrap each signal handler. Our wrapping mechanism ensures that (i) every signal handler first enters a trusted wrapper before invoking the actual untrusted application signal handler, and (ii) the kernel writes the signal frame to the `sigaltstack`. Our trusted wrapper fetches the signal frame from the `sigaltstack`, points the stack pointer to the untrusted stack, and invokes the application's original signal handler. When the original signal handler returns, our wrapper restores the stack pointer to the trusted `sigaltstack` to complete its execution.

Our trusted wrapper fully supports nested signals (i.e., signals delivered while the application is already executing a signal handler). This is not straightforward because of how the kernel writes signal frames for nested signals. Concretely, if a signal arrives while the application's stack pointer points to the `sigaltstack`, the kernel writes the signal frame just below the current stack pointer to avoid overwriting the frame for the signal that is currently being processed. However, if a signal arrives while the application's stack pointer points to the *untrusted* stack, the kernel writes the signal frame to the top of the `sigaltstack`.

This presents us with a problem. Since our trusted wrapper switches from the `sigaltstack` to the untrusted stack and back, the kernel could potentially overwrite the signal frame for the signal we are currently processing. Specifically, this happens when the untrusted signal handler has not yet returned to the trusted wrapper. We tackle this problem by reducing the apparent size of the alternate signal stack before switching to the untrusted stack. We do this by calling the `sigaltstack` system call to exclude the memory region occupied by the current signal frame. This ensures that any new signal will

not overwrite the in-use portion of the stack, preserving the integrity of nested signal handling.

This approach also allows the monitor to install signal handlers that run first and, if certain conditions are met, call the application's signal handler if one exists. For example, when a segmentation fault occurs, the monitor can first determine whether the fault was caused by execution on a non-executable page that requires placing hardware breakpoints. If the page contains no unsafe instructions, the monitor then calls the application's segmentation fault handler or allows the program to terminate. This preserves support for application-defined signal handlers even when the monitor needs to handle the same signals.

7 Evaluation

We evaluate Secpoline along two dimensions: security and performance. First, we assess the security guarantees of Secpoline by analyzing its resilience against a comprehensive set of previously known PKU bypass attacks (Section 7.1). Second, we evaluate the performance impact of Secpoline across a range of applications and compare it against nexpoline [33, 34] and strace [18], a ptrace-based cross-process monitor (Section 7.2). We further compare our sandbox implementation against Cerberus, a state-of-the-art MPK sandbox [5], to demonstrate the performance advantages of using an in-process monitor (Section 7.3). To ensure a fair comparison, we reuse the publicly available artifacts of nexpoline [34] and Cerberus [5].

Experimental Setup. We run all tests on a 12-core 12th Gen Intel Core i7-12700 processor CPU running at 2.10 GHz and 32 GiB of RAM with a L1i cache of 354 KiB, a L1 cache of 576 KiB, a L2 cache of 15 MiB, and a L3 cache of 25 MiB. We disable hyperthreading on the CPU to reduce measurement noise [49]. We use Ubuntu 20.04 with Linux kernel version 6.12.0 and glibc version 2.31.

7.1 Security Evaluation

We evaluate the security guarantees of Secpoline against a comprehensive set of previously known PKU bypass attacks. We survey the existing literature [5, 32, 34, 37, 38] to collect known attacks, and summarize the results in Table 1. For most attack categories, we implemented a corresponding Proof of Concept (PoC) exploit ourselves, either by adapting existing PoCs to specifically target our monitor, or by developing new PoCs based on prior attack descriptions. Secpoline correctly detects and rejects all these exploits, which empirically demonstrates parity with the collective security guarantees of existing PKU-protected syscall interposers [5, 32, 34, 37, 38].

Other attacks abuse broader categories of interposer weaknesses, instead of unique vulnerabilities. Secpoline takes systematic steps to eliminate these weaknesses, and empirically evaluating individual exploit examples is not the best demonstration of Secpoline's immunity against these attacks. We reason through these categories below.

Table 1: Known PKU Bypass attacks

Attack	Prevented
Inconsistency of PT Permissions / Page Table Syscalls Abuse [32, 34, 37]	●
Mutable Mappings [37]	●
Changing Code by Relocation [37]	●
Modifying PKRU via sigreturn [37]	●
Race Condition in Scanning [37]	●
Determination of Trusted Mappings [37]	●
Influencing with seccomp [37]	●
Modifying Trusted Mappings [37]	●
Syscall TOCTOU Attack [32]	◒
`rseq` Control Flow Hijacks [34]	●
Forged Signal Delivery [34]	●
Incorrect Signal Return Handling [34]	●
Inconsistency of Monitor and Kernel Status [34]	◒
Syscall Misidentification [38]	●
Vetted unsafe instruction relocation [5]	●
Incomplete debug register update [5]	●

Legend: ● = Prevented and PoC implemented ◒ = Prevented but no concrete PoC applicable ⊗ = Not prevented

Syscall TOCTOU Attacks arise whenever the monitor makes decisions based on data stored in untrusted memory [32], leaving an opportunity for the attacker to change the data between the monitor reading the data, and executing the syscall. Secpoline prevents these attacks by first copying all untrusted data into trusted memory before it is accessed by the monitor. In addition, Secpoline provides a set of helper functions for future syscall filter rules, allowing them to safely copy and validate untrusted memory as described in Section 6.1.

Inconsistency of Monitor and Kernel Status can arise from race conditions in the interposer's state management of process resources like file descriptors or memory mappings [34], and can be used to bypass other filter rules in the interposer. While it is possible to comprehensively eliminate these race conditions by modifying the kernel to implement state changes for the interposer [4], this comes at the cost of increasing the TCB. Similar to existing user-space interposers [34], Secpoline uses lock-based synchronization to maintain consistency.

7.2 Performance Evaluation

We use various real-world programs and synthetic benchmarks to measure overhead, often network-related and syscall-intensive. Figure 4 presents the overall performance overhead of each application relative to native execution. To ensure the accuracy and consistency of the results, all network communication is confined to the local system, thus eliminating potential variability due to network latency or congestion.

LMBench is a portable and straightforward benchmarking tool that we use to measure the performance impact of various commonly used system calls and signal handling [50]. Specifically, we evaluate the overhead introduced by Secpoline for the `open`, `read`, `write`, `mmap`, and `sigaction` system calls, as well as for sending and handling signals using the `kill` system call. Figure 3 shows the comparison between nexpoline, Secpoline, and strace for these system calls. We observe a significantly higher overhead for strace, as it relies on a cross-process approach using `ptrace` to intercept system calls. In contrast, Secpoline achieves performance comparable to or slightly better than nexpoline. This improvement arises because, instead of manually verifying whether a system call's arguments point to untrusted memory to prevent confused deputy attacks, Secpoline simply lowers MPK privileges (see Section 6.1), which is clearly faster.

These micro-benchmarks demonstrate that our scalable architecture does not compromise performance.

SQLite is the most used SQL database engine in the world [51]. We executed SQLite's speedtest benchmark, which uses `read()` and `write()` system calls with small buffer sizes to simulate real-world usage patterns. While Secpoline incurs 8.8% performance overhead compared to the baseline and nexpoline incurs 12.97% overhead, they remain relatively similar and significantly outperform the cross-process approach.

zip compresses the entire Linux kernel 5.9.8 source tree by recursively traversing all files, reading their contents, and archiving them into a single compressed zip file. We measure the execution time and associated overhead using the `time` command. Secpoline incurs a 1.88% performance overhead compared to the baseline, whereas nexpoline incurs an overhead of 3.12%. Unlike the other benchmarks, the majority of system calls executed by Zip are `stat`, rather than `read` or `write`. Furthermore, this benchmark performs the fewest system calls per second, resulting in lower overhead.

curl downloads a 1 GB file from a local Web server. curl issues a system call every 8 KB of data transferred and frequently installs signal handlers during execution. In total, downloading a 1 GB file results in over 13000 `write` system calls and more than 196000 rt_sigaction() system calls. It is the most syscall-intensive application and demonstrates the worst-case performance of Secpoline, with an average overhead of 25.86%, while nexpoline incurs 33.67%.

7.3 Comparison with Cerberus

To compare our MPK sandbox implementation with a cross-process approach, we chose to compare Secpoline with Cerberus [5]. Cerberus uses a small kernel module to redirect unsafe system calls to the cross-process monitor. It uses a list of dangerous syscalls that must be redirected to the cross-process monitor based on ptrace; all other syscalls are allowed to execute directly. Cerberus only provides this patch for the Linux 5.3.0 and 5.4.0 kernels, so we port it to the 6.12.0 kernel running on our machine to ensure fair comparison. We measure the performance of two server applications with different configurations. We run each server native, in Cerberus, and in Secpoline. We run client and server on the same machine. We pin them to different cores so they do not interfere with each other.

We use wrk as a client to measure the throughput of each server [52] running each configuration 10 times and calculate the average across these runs. Figure 6 and Figure 5 show the resulting throughput in requests per second.

First, we evaluate lighttpd [53], since this server is also used in the original Cerberus paper [5]. We test different sizes: 4 kB, 4.1 kB (size of index.html), and 64 kB. The results are shown in Figure 5. These results validate our setup: as reported by the authors, Cerberus performs very close to baseline.

Due to the aforementioned unsafe-syscall list, Cerberus has practically no overhead on lighttpd. All the performance-sensitive syscalls can be executed directly, without monitor interference. However, this is not always the case. On Apache, the third-most-used server worldwide according to Netcraft [54], Cerberus often drops to just half of the native throughput, as shown in fig. 6. The overhead gap in slowdown between Cerberus and the native case is largely due to the `mmap` syscall, which requires handling from an MPK sandbox to ensure that the requested memory is not both writable and executable, or that the attacker is not introducing new dangerous instructions. In Cerberus' case, this means it is forwarded to the cross-process monitor. In contrast, Secpoline does not incur this cross-process overhead and is thus able to process the `mmap` syscall significantly faster.

8 Case Studies

Finally, to demonstrate Secpoline's ability to integrate large, existing codebases that can be transparently run inside the interposer, we present two syscall interposition case studies. First, we implement an in-process intrusion detection system where Secpoline eliminates the need for a custom kernel module. Second, we use Secpoline to build a sidecar proxy as an isolated in-process component, and show how significant performance improvements are possible by doing so.

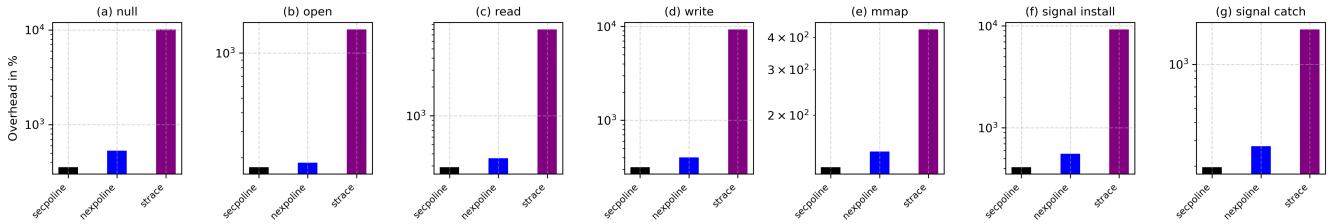


Figure 3: Relative system call latency compared to the baseline for the LMBench benchmark

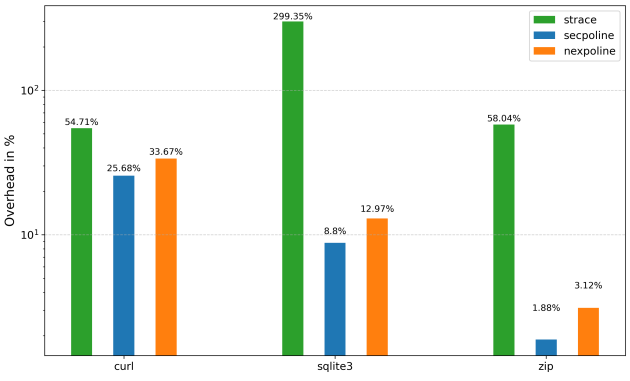


Figure 4: Normalized overhead for different applications

8.1 Falco Without Kernel Modules

First, we demonstrate that Secpoline can replace existing kernel-based interposition mechanisms in high-performance, security-sensitive applications by building a fully in-process intrusion detection system using the Falco engine [55]. Falco is a widely used intrusion detection system that combines events from multiple sources, including syscalls, to detect system compromise. It is a multi-threaded application consisting of components for event capture, parsing, and rule evaluation.

In our setup, the Falco engine is integrated into Secpoline, allowing syscall events intercepted by Secpoline to be forwarded directly to the Falco rules engine without requiring the Falco kernel module. The Falco engine itself remains isolated from the potentially compromised application by running inside the Secpoline interposer, and thus inherits the security guarantees provided by Secpoline. We implement this integration by placing intercepted syscall numbers and arguments into an event queue consumed by a custom Falco plugin [56], which forwards events to the Falco rules engine.

This case study demonstrates the flexibility of Secpoline by embedding the existing production-grade Falco intrusion detection engine into the in-process Secpoline interposer. This deployment avoids TCB-sensitive kernel modules for syscall interception, and required no deep knowledge or porting of Falco's internals.

Implementation Effort. Falco is built as a static library, and typically links statically into an executable that invokes its `run` function. We follow the same process to integrate Falco into Secpoline, in a separate trusted worker thread of the monitor. To receive events, we implemented a plugin using Falco's C++ Software Development Kit (SDK) [57] in 164 lines of C++ code. We add 100 C++ lines of filter rules to Secpoline's extendable interposition functionality to forward intercepted syscalls to the event queue. The Falco engine itself runs completely unmodified inside the protected Secpoline environment.

Experimental Setup. Because the recent version of Falco we evaluated was not compatible with Ubuntu 20.04, we use a different machine for this experiment. We run these tests on a 24-core Intel Core Ultra 9 285K processor CPU running at 5.80 GHz and 64 GiB of RAM with a L1i cache of 1.3 MiB, a L1 cache of 768 KiB, a L2 cache of 40 MiB, and a L3 cache of 36 MiB. We disable hyperthreading on the CPU to reduce measurement noise. We use Ubuntu 24.04 with Linux kernel version 6.17.0 and glibc version 2.39.

Performance Evaluation To measure the performance overhead of embedding Falco inside Secpoline, we evaluate a deployment using the `zip` system utility. Specifically, we compress the Linux kernel source code (v6.12.1) 10 times, and average the execution time across each run. We evaluate this with a traditional deployment of Falco as a separate process, using its kernel module to intercept syscalls, and with Falco running as part of Secpoline, inside the `zip` process.

As shown in Figure 7, the overhead introduced by running Falco within Secpoline is small (±5.78%). This added cost is expected, as the kernel module simply extends the default implementation of syscalls to implement asynchronous logging to a buffer, while the Secpoline implementation incurs a small additional syscall interception overhead and has to switch to the protected monitor domain to maintain isolation. In return, our approach does not extend the trusted Linux kernel at all, improving system reliability. Furthermore, this new setup can apply arbitrary *synchronous* filter rules in user space, all of which would further pollute the TCB when implemented inside Falco's kernel module.

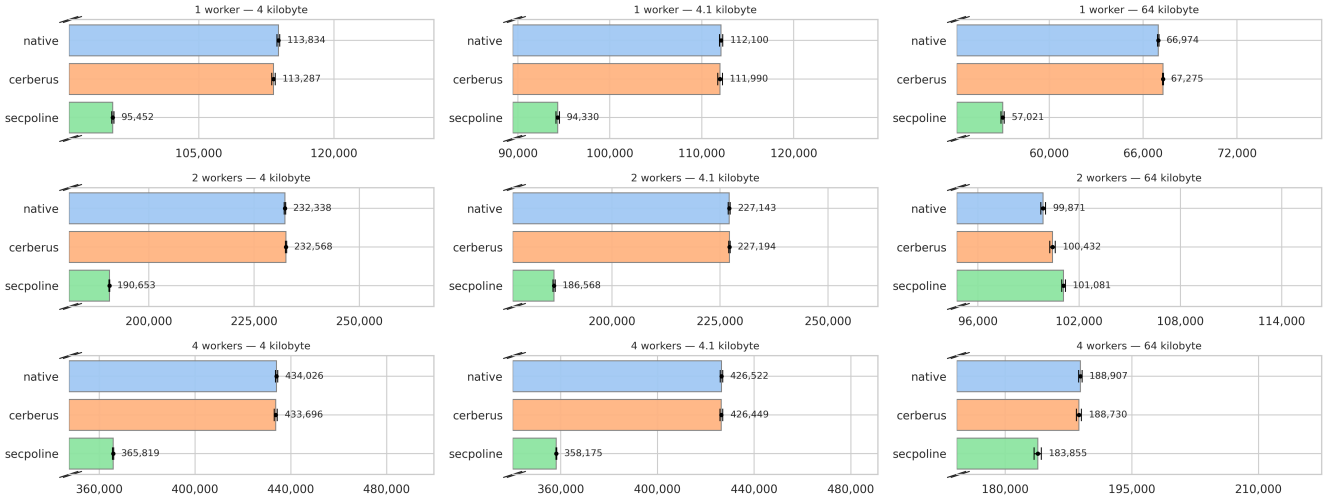


Figure 5: Lighttpd comparison (Req/Sec): between native, Cerberus, and secpoline.

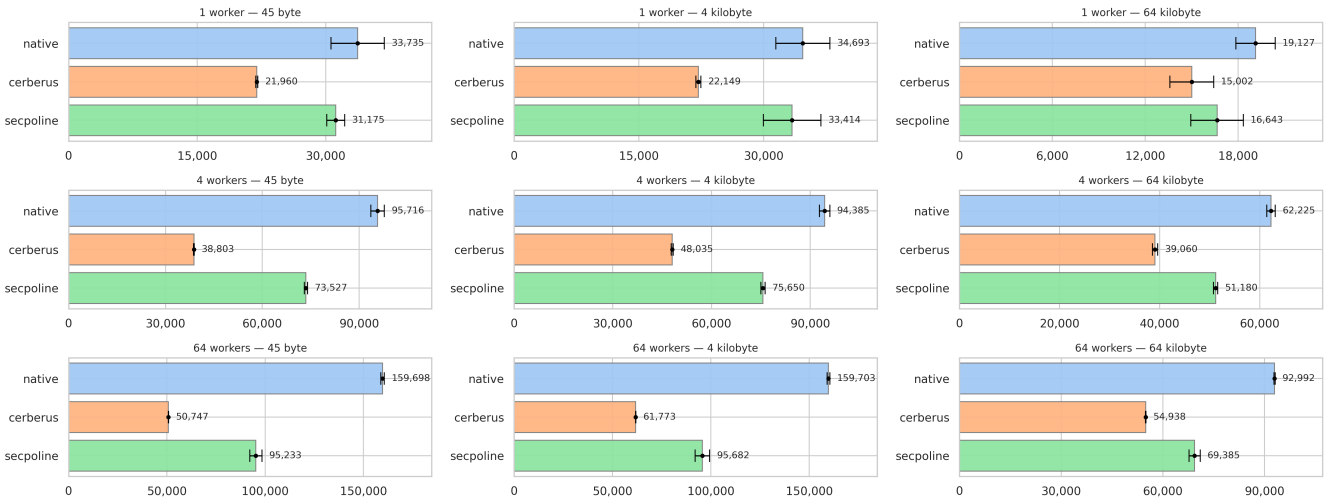


Figure 6: Apache comparison (Req/Sec): between native, Cerberus, and Secpoline.

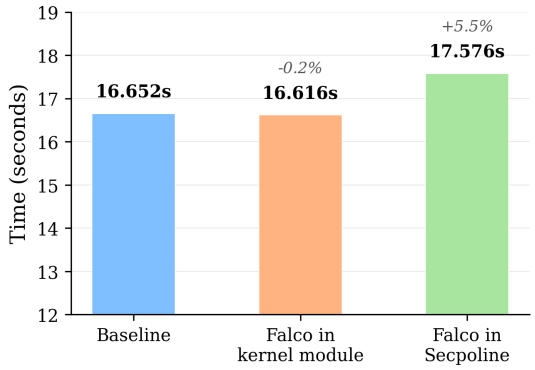


Figure 7: Average time to compress the Linux kernel source code (v6.12.0) using `zip` without using Falco (l), with Falco using a kernel module (m), and with Falco embedded in Secpoline (r).

8.2 In-Process ProxySQL

Second, we demonstrate the potential performance benefits enabled by scalable in-process syscall interposition through a production-grade sidecar. Specifically, we integrate ProxySQL v3.0 [58], which spans more than 2 million lines of C and C++ code, into Secpoline. ProxySQL is a production-grade SQL proxy that implements complex functionality, including multi-threaded network handling, query caching, and query preprocessing.

We select ProxySQL because it is a popular sidecar for application containers that want to communicate with a centralized database, in which case it often performs query validation, caching, authentication, and constrains the connection to the backend database to isolate it from accidental (or malicious) *query storms* in the application container. In a typical setup, ProxySQL is a dedicated container that receives localhost network traffic from the application container, processes it, and forwards a subset to the database container. This introduces a second network hop into the connection with the database, which presents a significant bottleneck in current service mesh architectures [59], and is the subject of ongoing innovation and research to alleviate [60].

In our setup, ProxySQL acts as a sidecar for a client issuing SQL requests. Rather than being deployed in a different process, we use Secpoline to embed ProxySQL into the monitor of the client application.

ProxySQL relies heavily on existing libraries and code for tasks ranging from handling SQL requests and encryption to parsing JSON files. Many of these components are loaded dynamically as plugins via `dl_open`. Converting ProxySQL into a static executable is therefore non-trivial: it would require re-implementing plugin loading functionality, which is normally handled by the dynamic linker, and modifying numerous dependencies that are not designed to operate in a static context. This complexity is compounded by the core ProxySQL code itself, such as its multithreading infrastructure, which partially relies on dynamic linker support. As a result, integrating ProxySQL into a static executable without major rewrites would be highly impractical.

To integrate ProxySQL into Secpoline, we first refactor it into a library by providing an entry point that initializes the proxy and spawns worker threads to run in the background. This initializer is invoked during Secpoline's startup, allowing the monitor to intercept and mediate all relevant network traffic while executing ProxySQL largely unmodified. We currently use this "librarification" method to embed ProxySQL because it sufficed for our use case, but with some additional engineering effort, Secpoline could reuse its loader infrastructure to fully load the ProxySQL binary, requiring no build system changes at all.

Then, to maximally benefit from the in-process integration, we use Secpoline to build a kernel-bypass fast path for the network traffic between client and ProxySQL, by intercepting relevant syscalls from the client and proxy, using a combination of application monitoring and meta-monitoring. Specifically, we identify when they attempt to open a socket to each other, and then intercept all subsequent operations on that socket to pass buffers directly in memory, avoiding mode switches and copies between kernel and user space [60]. Instead of socket operations, we only copy the data once to/from Secpoline-managed buffers, significantly reducing latency.

Implementation Effort. To refactor ProxySQL into a library that can be integrated into Secpoline we changed 22 lines in Makefiles and 14 lines of C code. We then changed zero lines of code to actually integrate the libProxySQL into Secpoline, aside from invoking the newly introduced ProxySQL initialization function. Finally, we added 350 lines of C++ code to Secpoline implement the kernel-bypass path, which is generic infrastructure that could be repurposed for other sidecars.

Performance Evaluation To evaluate the performance impact of integrating ProxySQL into Secpoline, we compare a baseline deployment of ProxySQL against the Secpoline-integrated version using `sysbench`. The benchmark generates concurrent SQL workloads that exercise ProxySQL's request processing and forwarding path, allowing us to assess the effectiveness of shared process implementation.

We evaluate throughput and latency using sysbench as a client that generates SQL requests to a MySQL server hosting a database with eight tables, each containing one million entries. The client issues requests for 300 seconds using a single thread. We consider two configurations. First, a baseline ProxySQL setup, in which ProxySQL runs as a separate process between sysbench and the MySQL server, acting as a conventional SQL proxy. Second, a Secpoline + libproxysql setup, where Secpoline runs with an integrated version of ProxySQL (libproxysql) and monitors the sysbench client.

Figure 8 shows the throughput and latency for both our configurations. Our integrated version of ProxySQL achieves higher throughput and lower latency compared to the baseline

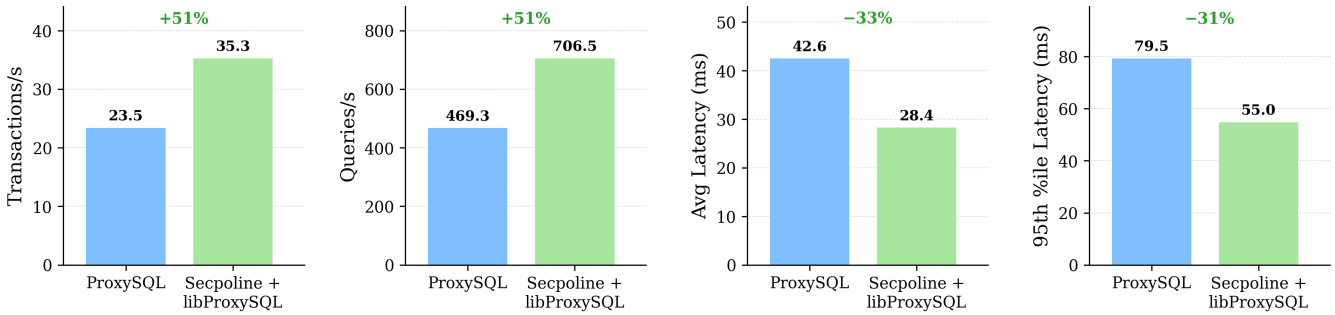


Figure 8: Throughput and latency for the sysbench benchmark using ProxySQL

ProxySQL deployment. Specifically, it processes 50.53% more transactions per second and queries per second, while having an average latency of 67.22% and a 95th-percentile latency of 69.15% relative to the baseline. These improvements result from enabling a fast path for request handling that avoids unnecessary system calls, while still mediating all security-relevant interactions through Secpoline.

9 Related Work

We now briefly compare Secpoline to prior work focusing on high-performance system call interposition and isolation.

9.1 System Call Interposition

Binary-rewriting in-process interposers (e.g., zpoline [20], lazypoline [23]) provide fast in-process syscall interception by rewriting call sites and avoiding kernel involvement, delivering near-native performance but *no* isolation. The monitored application can trivially tamper with the monitor's code and state. Secpoline preserves a similarly fast kernel-bypass path while adding hardware-enforced isolation and support for complex, unmodified monitor code thanks to its loader and meta-monitor design.

Secure in-process interposers with MPK (e.g., nexpoline [33], Endokernel [34]) harden in-process interposition with MPK (and, optionally, CET/SUD). However, they do so at the cost of a restricted programming environment with a custom libc, limited runtime features, etc. This complicates the integration of large, existing codebases into the monitor. Secpoline achieves comparable security while retaining a conventional POSIX-like programming environment for the interposer through isolated multi-program loading and its meta-monitor.

MPK sandboxes (e.g., Cerberus [5]) rely on cross-process monitors to police unsafe instructions such as `WRPKRU` and mediate sensitive system calls. This preserves isolation but incurs context-switching overhead for unsafe calls. Secpoline implements a self-contained, fully in-process MPK sandbox, avoiding these costs while achieving the same protection goals.

Positioning. Compared to the most similar system call interposition systems, Secpoline uniquely combines (i) in-process fast paths (as in zpoline and lazypoline), (ii) strong isolation with MPK (as in nexpoline/Endokernel) without a constrained ABI or custom libc, and (iii) a self-contained MPK sandbox that removes the cross-process bottlenecks that are typical in prior sandboxes. Secpoline thus supports unmodified, feature-rich monitors, while achieving high efficiency and offering strong security guarantees.

9.2 Isolation Mechanisms

A wide range of isolation mechanisms have been proposed to confine untrusted code.

Process-based isolation relies on OS primitives such as namespaces, control groups, and mandatory access control frameworks to isolate mutually untrusted components. While robust, these techniques often incur high run-time overhead due to context or mode switching, and they provide limited control over fine-grained memory access.

Software Fault Isolation (SFI) enforces memory isolation through code alignment and compiler-inserted static and dynamic checks [61–70]. Systems such as NaCl [71] and WebAssembly [72] provide strong confinement guarantees but restrict the execution of arbitrary code (e.g., no regular programming environment) or incur substantial translation overhead.

Virtualization-based isolation leverages hardware privilege rings or virtual machines to isolate execution contexts [1, 27, 66, 73–78]. This provides strong security guarantees but introduces significant complexity and performance overhead, particularly for fine-grained interactions such as system call interceptions.

Hardware-assisted in-process isolation mechanisms such as MPK [3,5,6,32,47,79], CHERI [80], ARM MTE/PAC [81], enable efficient intra-process isolation with low overhead. However, using these primitives securely requires additional sandboxing to prevent privilege escalation and confused-deputy attacks [37].

Positioning. Secpoline builds on MPK to provide hardware-assisted in-process isolation. It differs from prior systems by combining a self-contained in-process sandbox with a meta-monitor and loading architecture that eliminates implicit sharing of state with the monitored application, while achieving high efficiency and preserving compatibility with complex, unmodified monitoring code.

10 Discussion

By building on lazypoline [23], our interposer inherits strong compatibility with expressive system call features, such as `vfork` and complex `clone` flag combinations. We retain these benefits while adding our Secpoline security layer.

Red Zone However, we also inherit lazypoline's compatibility limitations. Most notably, rewriting `syscall` instructions to `call rax` pushes a return address onto the stack, potentially in a context where the program does not expect it. This can interfere with functions optimized as *leaf nodes*—those that omit a standard prologue and instead access local variables using negative offsets from `rsp`. The System V ABI [82] formally permits this optimization by defining a 128-byte "red zone" below `rsp` that must remain untouched by the kernel during syscalls or signal delivery.

Inserting a return address into this red zone violates the ABI and can overwrite local variables for such leaf functions, breaking correctness. Neither zpoline nor lazypoline can currently mitigate this compatibility issue. In practice, however, we are not aware of real-world breakages. A `syscall` instruction must either appear in inline assembly (which disables the leaf optimization by making the function's behavior opaque to the compiler), or in hand-written assembly (where the compiler is not involved). The only plausible failure case is a hand-optimized leaf function that also contains a raw `syscall`. We have not observed such code in the wild.

Architectural Dependencies Secpoline current prototype relies on x86-specific hardware breakpoints, Intel MPK and the `gs/fs` registers, but is not fundamentally limited to the x86 architecture. ARM provides similar hardware breakpoints [83] and its own in-process isolation primitives [81]. However, it only exposes a single dedicated TLS register, which conflicts with Secpoline's current design requirement of separate TLS registers to hold the monitor-specific thread-local metadata (`gsbase` on x86) and the general monitor/application TLS area (`fsbase` on x86). We can remain compatible with ARM by placing the monitor-specific TLS at a fixed offset from the application TLS, which ensures that we can locate it during interposer entry, before swapping to the monitor's general TLS during monitor execution.

Expressiveness of Application Code While this paper primarily focuses on the expressiveness of code running inside the monitor, a secondary issue concerns the expressiveness of the application code itself. Many system calls can be used to bypass the MPK sandbox and therefore must be restricted [3, 5, 34]. Some of these syscalls, such as `seccomp` and `rseq`, are fully blocked, while others are blocked or partially emulated depending on specific argument combinations, e.g., requesting a shared file-backed mapping [5]. While not observed in our evaluations, these restrictions can impact benign application behavior as well, but are currently necessary to stop known PKU bypasses, and similarly enforced by all existing approaches. Finding more fine-grained filter rules to permit more benign behavior, and increase application expressiveness, is an interesting and valuable path for future work. Secpoline facilitates this by providing a familiar and easily programmable interposer programming environment, in which more complex filter behavior can be implemented more reliably.

11 Conclusion

In this paper, we present Secpoline, a scalable solution to secure in-process syscall interposition, without sacrificing the performance advantages provided by in-process interposition. Secpoline enables safe integration of complex user code without requiring deep knowledge of the monitor or intrusive modifications to the integrated code by providing a familiar programming environment. This finally enables the practical construction of sophisticated syscall filter rules for security applications. We evaluate our design by integrating two real-world use cases, demonstrating both its ability to replace problematic kernel modules and its potential for performance gains.

Open Science

Our artifact can be accessed at https://github.com/lazypoline/secpoline and https://doi.org/10.5281/zenodo.20432296 [35].

Ethical Considerations

We identify three stakeholder groups affected by this research: operating-system and hardware vendors, system users (deployers and administrators), and malicious actors.

Vendors (OS and hardware implementers) could be harmed if previously unknown vulnerabilities in kernel interfaces, MPK implementations, or CPU instruction handling (e.g., WRPKRU/XRSTOR, perf event interfaces) are discovered and exploited before vendors can produce patches. To mitigate this risk we limited our security evaluations to controlled, synthetic workloads and microbenchmarks that exercise known classes of misuse (e.g., unauthorized PKRU changes, WRGSBASE abuse). All experiments were conducted on dedicated test systems under our control (development machines), not on production infrastructure. We target well-studied primitives and base our threat model on documented behaviors of MPK

and syscall mechanisms, we do not expect our work to introduce novel, exploitable vectors beyond those already studied in prior literature.

Users (system administrators and end-users) could be affected if vulnerabilities relating to temporal races, signal handling, or MPK enforcement are exploited in deployed systems. Our proposed techniques (loader-based isolation, per-domain MPK policies, pointer validation and copy-to-trusted strategies, and stop-the-world breakpoint synchronization) are intended to improve runtime security by preventing common bypasses of in-process monitors. To avoid harm to users we ran all testing on dedicated lab systems and did not involve production or customer-owned devices. Where our prototype requires platform features (e.g., MPK support, perf-based breakpoints, recent kernel seccomp facilities), we note these deployment constraints explicitly; we do not recommend deploying experimental instrumentation on production hosts without vendor guidance and appropriate staging. Overall, we expect our results to benefit users by reducing the risk of certain classes of syscall-interposition bypasses when adopted responsibly.

Malicious actors might attempt to misuse techniques described in our paper (for example, to discover new ways to evade existing monitors). We reduced this risk by focusing our artifact and evaluations on controlled, low-rate experiments and by emphasizing defensive mitigations (for instance, pointer-copying to avoid TOCTOU, strict MPK policies, and guarded syscall fast paths).

We encountered no unexpected ethical issues during the research. All experiments were run on systems we own or have explicit permission to use, and no user data or production systems were involved.

Decision: We decided to proceed with and publish this research because we believe the potential benefits (improved defenses for in-process syscall interposition and clearer guidance for secure MPK usage) outweigh the risks.

Acknowledgments

We thank the anonymous reviewers for their feedback that helped improve this paper. This research is partially funded by the Internal Funds KU Leuven, and by the Cybersecurity Research Program Flanders.

References

[1] gVisor Developers, "gVisor," https://gvisor.dev.

[2] *seccomp(2) — Linux manual page*, 2024, accessed: 2025-05-05. [Online]. Available: https://man7.org/linux/man-pages/man2/seccomp.2.html

[3] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg, "ERIM: Secure, efficient in-process isolation with protection keys (MPK)," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1221–1238. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/vahldiek-oberwagner

[4] D. Peng, C. Liu, T. Palit, P. Fonseca, A. Vahldiek-Oberwagner, and M. Vij, "μswitch: Fast kernel context isolation with implicit context switches," in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023, pp. 2956–2973.

[5] A. Voulimeneas, J. Vinck, R. Mechelinck, and S. Volckaert, "You shall not (by)pass! practical, secure, and fast PKU-based sandboxing," in *Proceedings of the Seventeenth European Conference on Computer Systems*, ser. EuroSys '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 266–282. [Online]. Available: https://doi.org/10.1145/3492321.3519560

[6] M. Hedayati, S. Gravani, E. Johnson, J. Criswell, M. L. Scott, K. Shen, and M. Marty, "Hodor: Intra-Process isolation for High-Throughput data plane libraries," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 489–504. [Online]. Available: http://www.usenix.org/conference/atc19/presentation/hedayati-hodor

[7] S. Volckaert, B. De Sutter, T. De Baets, and K. De Bosschere, "GHUMVEE: efficient, effective, and flexible replication," in *Foundations and Practice of Security: 5th International Symposium, FPS 2012, Montreal, QC, Canada, October 25-26, 2012, Revised Selected Papers 5*. Springer, 2013, pp. 261–277.

[8] S. Volckaert, B. Coppens, A. Voulimeneas, A. Homescu, P. Larsen, B. De Sutter, and M. Franz, "Secure and efficient application monitoring and replication," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016, pp. 167–179.

[9] "Wine," Last accessed 2023. [Online]. Available: https://www.winehq.org/

[10] J. Dike, "User-mode linux," in *Annual Linux Showcase & Conference (ALS*, 2001.

[11] H.-C. Kuo, D. Williams, R. Koller, and S. Mohan, "A linux in unikernel clothing," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3342195.3387526

[12] R. Nikolaev and G. Back, "Virtuos: An operating system with kernel virtualization," 2013.

[13] P. Olivier, D. Chiba, S. Lankes, C. Min, and B. Ravindran, "A binary-compatible unikernel," in *Conference on Virtual Execution Environments (VEE)*, 2019.

[14] A. Raza, T. Unger, M. Boyd, E. B. Munson, P. Sohal, U. Drepper, R. Jones, D. B. De Oliveira, L. Woodman, R. Mancuso, J. Appavoo, and O. Krieger, "Unikernel linux (ukl)," in *Proceedings of the Eighteenth European Conference on Computer Systems*, ser. EuroSys '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 590–605. [Online]. Available: https://doi.org/10.1145/3552326.3587458

[15] Z. Shen, Z. Sun, G.-E. Sela, E. Bagdasaryan, C. Delimitrou, R. Van Renesse, and H. Weatherspoon, "X-containers: Breaking down barriers to improve performance and isolation of cloud-native containers," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[16] L. Soares and M. Stumm, "Flexsc: Flexible system call scheduling with exception-less system calls," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[17] R. Stallman, R. Pesch, S. Shebs *et al.*, "Debugging with gdb," *Free Software Foundation*, vol. 675, 1988.

[18] *strace(1) — Linux manual page*, 2025. [Online]. Available: https://man7.org/linux/man-pages/man1/strace.1.html

[19] *ptrace(2) - Linux manual page*, 2024, accessed: 2025-05-02. [Online]. Available: https://man7.org/linux/man-pages/man2/ptrace.2.html

[20] K. Yasukata, H. Tazaki, P.-L. Aublin, and K. Ishiguro, "zpoline: a system call hook mechanism based on binary rewriting," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. Boston, MA: USENIX Association, Jul. 2023, pp. 293–300. [Online]. Available: https://www.usenix.org/conference/atc23/presentation/yasukata

[21] "syscall_intercept," Last accessed 2023. [Online]. Available: https://github.com/pmem/syscall_intercept

[22] Intel Corporation and The Linux Foundation, "Data Plane Development Kit (DPDK)," https://www.dpdk.org, 2022.

[23] A. Jacobs, M. Gülmez, A. Andries, S. Volckaert, and A. Voulimeneas, "System call interposition without compromise," in *2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2024, pp. 183–194. [Online]. Available: https://doi.org/10.1109/DSN58291

[24] X. Wang, S. Yeoh, P. Olivier, and B. Ravindran, "Secure and efficient in-process monitor (and library) protection with intel mpk," in *Proceedings of the 13th European Workshop on Systems Security*, ser. EuroSec '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 7–12. [Online]. Available: https://doi.org/10.1145/3380786.3391398

[25] H. Lefeuvre, N. Dautenhahn, D. Chisnall, and P. Olivier, "SoK: Software Compartmentalization," in *2025 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2025, pp. 3107–3126. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP61157.2025.00075

[26] S. Ghavamnia, T. Palit, S. Mishra, and M. Polychronakis, "Temporal system call specialization for attack surface reduction," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1749–1766. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/ghavamnia

[27] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis, "Dune: Safe user-level access to privileged CPU features," in *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. Hollywood, CA: USENIX Association, Oct. 2012, pp. 335–348. [Online]. Available: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/belay

[28] K. Koning, H. Bos, and C. Giuffrida, "Secure and efficient multi-variant execution using hardware-assisted process virtualization," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2016, pp. 431–442.

[29] J. Jia, Y. Zhu, D. Williams, A. Arcangeli, C. Canella, H. Franke, T. Feldman-Fitzthum, D. Skarlatos, D. Gruss, and T. Xu, "Programmable system call security with ebpf," *arXiv preprint arXiv:2302.10366*, 2023.

[30] J. Corbet, "Reconsidering unprivileged bpf," https://lwn.net/Articles/796328/, Aug. 2019.

[31] D. Schrammel, S. Weiser, S. Steinegger, M. Schwarzl, M. Schwarz, S. Mangard, and D. Gruss, "Donky: Domain keys – efficient In-Process isolation for RISC-V and x86," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1677–1694. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/schrammel

[32] D. Schrammel, S. Weiser, R. Sadek, and S. Mangard, "Jenny: Securing syscalls for PKU-based memory iso-

lation systems," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 936–952.

[33] F. Yang, A. Vahldiek-Oberwagner, C.-C. Tsai, K. Kaoudis, and N. Dautenhahn, "Making 'syscall' a privilege not a right," 2024. [Online]. Available: https://arxiv.org/abs/2406.07429

[34] F. Yang, B. Im, W. Huang, K. Kaoudis, A. Vahldiek-Oberwagner, C.-C. Tsai, and N. Dautenhahn, "Endokernel: a thread safe monitor for lightweight subprocess isolation," in *Proceedings of the 33rd USENIX Conference on Security Symposium*, ser. SEC '24. USA: USENIX Association, 2024.

[35] S. Ruben, "Secpoline: A scalable approach to build secure in- process syscall interposers," May 2026. [Online]. Available: https://doi.org/10.5281/zenodo.20432296

[36] Intel, *Intel® 64 and IA-32 Architectures Software Developer's Manual*, Intel Corporation, 2018, https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html.

[37] E. Connor, T. McDaniel, J. M. Smith, and M. Schuchard, "PKU pitfalls: Attacks on PKU-based memory isolation systems," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1409–1426. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/connor

[38] J. M. Gómez Moreno, V. Moutafis, A. Dionysiou, F. Kuipers, G. Smaragdakis, B. Coppens, and A. Voulimeneas, "Clair obscur: The light and shadow of system call interposition – from pitfalls to solutions with k23," in *Proceedings of the 26th International Middleware Conference*, ser. Middleware '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 241–255. [Online]. Available: https://doi-org.kuleuven.e-bronnen.be/10.1145/3721462.3770772

[39] ARM, "Permission overlays," https://developer.arm.com/documentation/102376/0200/Permission-indirection-and-permission-overlay-extensions/Permission-overlays.

[40] w. glibc, "LinkerNamespaces - glibc wiki." [Online]. Available: https://sourceware.org/glibc/wiki/LinkerNamespaces

[41] *ld.so(8) — Linux manual page*, 2025. [Online]. Available: https://man7.org/linux/man-pages/man8/ld.so.8.html

[42] "A deep dive into (implicit) thread local storage," 2018. [Online]. Available: https://chao-tic.github.io/blog/2018/12/25/tls

[43] *Dynamic Linking*. [Online]. Available: https://refspecs.linuxfoundation.org/ELF/zSeries/lzsabi0_zSeries/x2251.html

[44] I. Free Software Foundation, "glibc 2.42: `include/link.h`." [Online]. Available: https://elixir.bootlin.com/glibc/glibc-2.42/source/include/link.h#L95

[45] *__COPY_FROM_USER(9) Memory Management in Linux __COPY_FROM_USER(9)*, 2017. [Online]. Available: https://manpages.debian.org/testing/linux-manual-4.8/__copy_from_user.9.en.html

[46] K. Zeng, Z. Lin, K. Lu, X. Xing, R. Wang, A. Doupé, Y. Shoshitaishvili, and T. Bao, "RetSpill: Igniting User-Controlled Data to Burn Away Linux Kernel Protections," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 3093–3107. [Online]. Available: https://doi.org/10.1145/3576915.3623220

[47] M. Gülmez, T. Nyman, C. Baumann, and J. T. Mühlberg, "Rewind & Discard: Improving Software Resilience Using Isolated Domains," in *Proceedings of 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, ser. DSN '23. Washington, DC, USA: IEEE Computer Society, jun 2023, pp. 402–416. [Online]. Available: http://doi.org/10.1109/DSN58367.2023.00046

[48] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda, "G-Free: defeating return-oriented programming through gadget-less binaries," in *Proceedings of the 26th annual computer security applications conference*, 2010, pp. 49–58.

[49] D. Beyer, S. Löwe, and P. Wendler, "Reliable benchmarking: requirements and solutions," *International Journal on Software Tools for Technology Transfer*, vol. 21, no. 1, pp. 1–29, 2019.

[50] L. McVoy and C. Staelin, "lmbench: Portable tools for performance analysis," in *USENIX 1996 Annual Technical Conference San Diego, California, January 1996*, 1996.

[51] S. Developers, "SQLite." [Online]. Available: https://www.sqlite.org/index.html

[52] *wrk - a HTTP benchmarking tool*. [Online]. Available: https://github.com/wg/wrk

[53] *lighttpd wiki and documentation*. [Online]. Available: https://www.lighttpd.net/

[54] Netcraft, "May 2025 web server survey," 2025. [Online]. Available: https://www.netcraft.com/blog/may-2025-web-server-survey

[55] "Falco." [Online]. Available: https://falco.org/

[56] *Plugins Developer Guide*. [Online]. Available: https://falco.org/docs/developer-guide/plugins/

[57] *plugin-sdk-cpp*. [Online]. Available: https://github.com/falcosecurity/plugin-sdk-cpp/tree/main

[58] *A High Performance MySQL & PostgreSQL Proxy*. [Online]. Available: https://proxysql.com/

[59] A. El Malki and U. Zdun, "Guiding architectural decision making on service mesh based microservice architectures," in *European Conference on Software Architecture*. Springer, 2019, pp. 3–19.

[60] M. Butrovich, K. Ramanathan, J. Rollinson, W. S. Lim, W. Zhang, J. Sherry, and A. Pavlo, "Tigger: A database proxy that bounces with user-bypass," *Proceedings of the VLDB Endowment*, vol. 16, no. 11, pp. 3335–3348, 2023.

[61] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *Proceedings of the fourteenth ACM symposium on Operating systems principles*, 1993, pp. 203–216.

[62] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black, "Fast byte-granularity software fault isolation," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 45–58.

[63] B. Ford and R. Cox, "Vx32: Lightweight user-level sandboxing on the x86," in *2008 USENIX Annual Technical Conference (USENIX ATC 08)*. Boston, MA: USENIX Association, Jun. 2008. [Online]. Available: https://www.usenix.org/conference/2008-usenix-annual-technical-conference/vx32-lightweight-user-level-sandboxing-x86

[64] L. Deng, Q. Zeng, and Y. Liu, "Isboxing: An instruction substitution based data sandboxing for x86 untrusted libraries," in *ICT Systems Security and Privacy Protection: 30th IFIP TC 11 International Conference, SEC 2015, Hamburg, Germany, May 26-28, 2015, Proceedings 30*. Springer, 2015, pp. 386–400.

[65] Y. Mao, H. Chen, D. Zhou, X. Wang, N. Zeldovich, and M. F. Kaashoek, "Software fault isolation with api integrity and multi-principal modules," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011, pp. 115–128.

[66] L. Zhao, G. Li, B. De Sutter, and J. Regehr, "Armor: fully verified software fault isolation," in *Proceedings of the Ninth ACM International Conference on Embedded Software*, ser. EMSOFT '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 289–298. [Online]. Available: https://doi.org/10.1145/2038642.2038687

[67] S. McCamant and G. Morrisett, "Evaluating SFI for a CISC Architecture." in *USENIX Security Symposium*, vol. 10, 2006, pp. 209–224.

[68] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen, "Adapting software fault isolation to contemporary cpu architectures," in *19th USENIX Security Symposium (USENIX Security 10)*, 2010.

[69] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, "Xfi: Software guards for system address spaces," in *Proceedings of the 7th symposium on Operating systems design and implementation*, 2006, pp. 75–88.

[70] M. Payer and T. R. Gross, "Fine-grained user-space security through virtualization," *ACM SIGPLAN Notices*, vol. 46, no. 7, pp. 157–168, 2011. [Online]. Available: https://doi.org/10.1145/1952682.1952703

[71] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native client: A sandbox for portable, untrusted x86 native code," *Communications of the ACM*, vol. 53, no. 1, pp. 91–99, 2010.

[72] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with webassembly," in *Proceedings of the 38th ACM SIGPLAN conference on programming language design and implementation*, 2017, pp. 185–200.

[73] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the Linux virtual machine monitor," in *Proceedings of the Linux symposium*, vol. 1, no. 8. Dttawa, Dntorio, Canada, 2007, pp. 225–230.

[74] F. Bellard, "Qemu, a fast and portable dynamic translator." in *USENIX annual technical conference, FREENIX Track*, vol. 41, no. 46. California, USA, 2005, pp. 10–5555.

[75] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight virtualization for serverless applications," in *17th USENIX symposium on networked systems design and implementation (NSDI 20)*, 2020, pp. 419–434.

[76] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kägi, F. H. Leung, and L. Smith, "Intel virtualization technology," Intel Corporation, Tech. Rep., 2005, accessed: 2025-05-12. [Online]. Available: https://www.profsandhu.com/cs6393_s14/intel_virtualization_technology_2005.pdf

[77] L. Bassi, D. Berardi, and R. Davoli, "Vuos: A user-space hypervisor based on system call hijacking," in *International Conference on Computer Safety, Reliability, and Security*. Springer, 2024, pp. 296–307.

[78] Z. Shen, Z. Sun, G.-E. Sela, E. Bagdasaryan, C. Delimitrou, R. Van Renesse, and H. Weatherspoon, "X-containers: Breaking down barriers to improve performance and isolation of cloud-native containers," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 121–135. [Online]. Available: https://doi.org/10.1145/3297858.3304016

[79] M. Gülmez, T. Nyman, C. Baumann, and J. T. Mühlberg, "Friend or Foe Inside? Exploring In-Process Isolation to Maintain Memory Safety for Unsafe Rust," in *2023 IEEE Secure Development Conference (SecDev)*. Los Alamitos, CA, USA: IEEE Computer Society, oct 2023, pp. 54–66. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SecDev56634.2023.00020

[80] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The CHERI capability model: Revisiting RISC in an age of risk," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 457–468, 2014.

[81] K. Dinh Duy, K. Cho, T. Noh, and H. Lee, "Capacity: Cryptographically-enforced in-process capabilities for modern arm architectures," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 874–888.

[82] *System V Application Binary Interface: AMD64 Architecture Processor Supplement*, https://gitlab.com/x86-psABIs/x86-64-ABI.

[83] *Breakpoints*. [Online]. Available: https://developer.arm.com/documentation/102140/0200/Breakpoints