Endokernel: A Thread Safe Monitor for Lightweight Subprocess Isolation

Fangfei Yang, *Rice University;* Bumjin Im, *Amazon.com;* Weijie Huang, *Rice University;* Kelly Kaoudis, *Trail of Bits;* Anjo Vahldiek-Oberwagner, *Intel Labs;* Chia-Che Tsai, *Texas A&M University;* Nathan Dautenhahn, *Riverside Research*

https://www.usenix.org/conference/usenixsecurity24/presentation/yang-fangfei

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Proceedings of the 33rd USENIX Security Symposium is sponsored by USENIX.

Endokernel: A Thread Safe Monitor for Lightweight Subprocess Isolation

Fangfei Yang
Rice University

Bumjin Im[1]
Amazon.com

Weijie Huang
Rice University

Kelly Kaoudis
Trail of Bits

Anjo Vahldiek-Oberwagner
Intel Labs

Chia-Che Tsai
Texas A&M University

Nathan Dautenhahn
Riverside Research

Abstract

Compartmentalization decomposes applications into isolated components, effectively confining the scope of potential security breaches. Recent approaches nest the protection monitor within processes for efficient memory isolation at the cost of security. However, these systems lack solutions for efficient multithreaded safety and neglect kernel semantics that can be abused to bypass the monitor.

The Endokernel is an intra-process security monitor that isolates memory at subprocess granularity. It ensures backwards-compatible and secure emulation of system interfaces, a task uniquely challenging due to the need to analyze OS and hardware semantics beyond mere interface usability. We introduce an inside-out methodology where we identify core OS primitives that allow bypass and map that back to the interfaces that depend on them. This approach led to the identification of several missing policies as well as aided in developing a fine-grained locking approach to deal with complex thread safety when inserting a monitor between the OS and the application. Results indicate that we can achieve fast isolation while greatly enhancing security and maintaining backwards-compatibility, and also showing a new method for systematically finding gaps in policies.

1 Introduction

In modern operating systems, processes serve as the fundamental unit of isolation and sharing between applications. As a result, an application has access to all the resources of this *virtual* runtime environment, including memory, file systems, networks, and code, among others. Unfortunately, this monolithic environment poses a significant security threat. Without isolation, any local problem becomes global – a buffer overflow could lead to arbitrary reads and writes, or even take full control and escalate privileges within the system [9, 10, 12, 20]. For example, the Heartbleed vulnerability, caused by an incorrect bounds check in OpenSSL, allowed attackers to overread server memory and expose sensitive information. [24]. Using a safe language appears to be a good solution, but even these are not immune (e.g., CrateDepression in Rust [29]), as they incorporate libraries of unknown origin leading to large-scale exploits [11, 25, 33, 65, 72].

Privilege separation [51] decomposes applications into compartments that are isolated by a runtime monitor. Early work successfully separated and isolated OpenSSL using process isolation [4, 30], and modern browsers employ process isolation for vulnerable components. However, process isolation incurs prohibitive overheads for components that frequently interact [60]. Thus, it can only be applied at a coarse granularity, leaving many attack surfaces exposed. For example, an integer overflow in the NGINX HTTP parser [18] could overflow the stack and allow control flow hijacking, which cannot be efficiently isolated with processes.

Recent work nests the monitor within the process using more efficient single-address space mechanisms (*e.g.,* Memory Protection Keys) that isolate at page granularity [32, 54, 60, 61]. However, their defenses are incomplete; for instance, both ERIM and Hodor [32, 60] neglect to prevent bypass through system call interfaces such as reading from the pseudo file system [15]. Donky [55] prevents these but neglects other system interfaces, and Cerberus [61] ignores multithreaded synchronization for signals [54]. Jenny [54] provides comprehensive policies, as discussed in this paper, it lacks a practical multithreaded isolation model and fails to protect against nested signals, creating inconsistencies between the monitor and the kernel that can be exploited. μSWITCH [50], through kernel modifications, avoids synchronization challenges and also delivers signals on an untrusted stack which is corruptible by untrusted attackers. Notably, all efforts evaluated security at the system call boundary and neglected to systematically assess kernel semantics that influence memory isolation (*e.g.,* `sendmsg` and the pseudo file system), while also including glibc or larger runtimes (*e.g.,* Go) in the TCB.

Our aim is to enhance nested monitor design, with secure and compatible emulation of system interfaces, while retain-

[1] Authored before joining Amazon.com

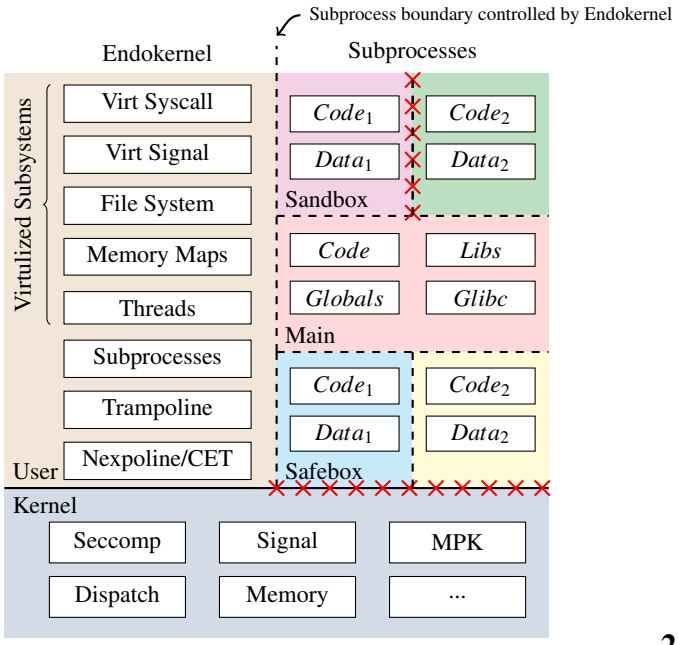


Figure 1: System Diagram for Endokernel

ing efficiency and avoiding the complexity of operating system changes. We first summarize how the Linux kernel affects the integrity of intra-process isolation by reviewing past attacks and analyzing the Linux implementation. Building on this, we derive invariants that monitors need to adhere to in order to mitigate these attacks at the system interface level. This leads to the development of novel policies, especially for multithreaded monitors. Additionally, we enhance compatibility through the virtualization of signals. We embody these invariants in the design and implementation of the Endokernel architecture that creates two privilege levels within the single-address-space using Intel® Memory Protection Keys.

We present a prototype Endokernel, including a new simple-to-use subprocess abstraction that provides process-level isolation for modules within a process, and apply it to isolate components in the NGINX server, including the HTTP parser and the OpenSSL library to demonstrate the effectiveness of the Endokernel. Figure 1 offers a detailed view of the Endokernel architecture, highlighting the Endokernel the organization of subprocesses. The monitor shows near-native performance for network benchmarks ($1.5\% - 5\%$ overhead) and cpu-intensive tasks ($< 4\%$ overhead), whereas file-based benchmarks observe overheads between $4 - 55\%$ (zip and sqlite3). The Endokernel passes 95.95% of the Linux Test Project regression tests, ensuring a high degree of compatibility with Linux. Our primary contributions include:

- A Methodology for systematically examining monitor safety based on critical internal kernel interfaces instead of the system call interface led to the identification of design flaws in synchronization between monitor and kernel states as well as unique signal management and control-flow integrity vulnerabilities (§3). We evaluate both prior work and our own against these threats (§6.1).
- The exploration of the design and implementation of the first thread-safe monitor using fine-grained locking revealed subtle inconsistencies. These occur when a thread is interrupted between verifying a condition and issuing a system call, leading to Time-of-Check to Time-of-Use (TOCTOU) violations (§3.3)."
- An embodiment of the methodology, new attack vectors, and thread safety is presented in a set of invariants that are specified and enforced in the Endokernel design (§4) and implementation (§5). A unique aspect of the prototype is backwards-compatible emulation (§6.3) while ensuring security (§6.1) and low performance degradation (§6.2) for essential system interfaces. We demonstrate the value by compartmentalizing NGINX and sudo (§7).

2 Background

We aim to produce a new in-process protection monitor that identifies invariants to ensure thread safety of the monitor and backwards-compatible system emulation while retaining security and performance. Table 1 presents an overview of the key dimensions of key related work that are described in part in this section, with novel observations and methods presented in the following section.

2.1 Memory Protection Keys

Memory Protection Key (MPK) enhances page memory protection by allowing each page to be tagged with a pkey, which serves as a privilege marker ranging from 0 to 15. The CPU features a 32-bit `PKRU` register, where every two bits correspond to a pkey's access denial (AD) and write denial (WD). The CPU performs additional checks beyond standard page protection by using the `PKRU` register to determine if reading (AD) or writing (WD) to this page is restricted. The user-accessible PKRU register can be modified using the `wrpkru` instruction, enabling lightweight memory isolation [60]. However, since MPK operates in user space, it is susceptible to threats from the operating system, such as the manipulation of `/proc/self/mem` to circumvent MPK protections [15]. Or, an attacker can abuse the `wrpkru` instruction to modify the `PKRU` register. Thus, effective isolation and monitoring are essential to maintaining MPK's integrity and supporting robust intra-process isolation policies.

2.2 Intra-process Isolation and Monitor

Intra-process isolation is a strategy that compartmentalizes different components of an application within a single process.

Aspect	Our Work	Jenny [54]	Cerberus [61]	Donkey [55]	lwC [42]	Enclosure [27]	Shreds [13]	Erim [60]	Hodor [32]	Wedge [8]	NaCl [71]	Sirius [58]	libmpk [48]	EPK [28]	CETIS [69]
Security Features and Performance Metrics															
Secure System Object	√	√[8]	√[8]	×	√[8]	√	×	×	×	×	√	√	×	×	×
Incompatibility	◔	◔	◔	◔	◔	◔	◔	◔	◔	◔	◓	◓	◔	◔	◔
Flexible Filter	√	√	√	√	×	√	×	N/A	×	N/A[1]	×	×	N/A	N/A	N/A
Granularity	Func	Func	Func	Func	Func	Library	Func	Func	Func	Thread	Exec	Func	Func	Func	Inst
Overhead	◓	◓	◕[2]	◓	◕[3]	◓	◓	◔	◓	●	◕[4]	◓[5]	◓	◓	◔
Signal and Thread Compatibility															
Signal Handling	√	√[7]	√	√	√	×	√	×	√	√	√	N/A[6]	×	√	√
Signal Security	√	×	×	×	√	×	×	×	×	√	√	N/A[6]	×	×	×
Secure Multi-threading	√	×	×	√	√	RT	√	×	√	√	√	√	×	N/A	N/A
Multi-process	√	√	√	√	√	N/A	√	×	√	√	√	√	×	N/A	N/A
Domains Isolation and Abstractions															
Multi-isolated domain	√	√	√	√	√	√	√	×	√	√	×	√	×	√	N/A
De-privileged domain	√	√	√	√	√	√	√	×	×	√	×	×	N/A	N/A	N/A
Elevated domain	√	×	×	√	√	×	√	√	√	×	×	Y	N/A	N/A	N/A

Table 1: Compartmentalization Approaches. ◔ to ● = from low to high. N/A=Not Applicable. RT=No native thread support, only thread provided by go runtime. Exec=Executable. 1=SELinux. 2=using ptrace. 3=Using virtual memory for the isolation. 4=NaCl 5=Kernel(Low)+TEE. 6=No signal. 7=Partially, non-nesting signals. 8=Not fully secure, overlooking a minority of objects.

It ensures that a compromise in one part of the application does not threaten the security of the remaining parts. MPK aids in achieving efficient intra-process isolation. Nonetheless, due to MPK's vulnerabilities, it is crucial to use an intra-process monitor alongside it, to reduce threats from the operating system and keep the isolation effective. Specifically, the operating system could circumvent MPK or compromise the integrity of the intra-process monitor in the following ways.

System Calls Bypassing MPK Protection Most system calls adhere to MPK protection. However, several system calls that enable users to manipulate the page table have already been highlighted in previous work [54, 61]. Obvious examples include `mprotect` and `mmap`, which all monitors need to restrict. The kernel also provides additional system calls to change page attributes, such as `vmsplice` and `splice` that can move physical pages and thereby bypass MPK. Furthermore, the `userfaultfd` mechanism allows handling page faults in user space, enabling users to map physical pages to virtual addresses during a page fault, thereby changing their contents. `arch_prctl` can be used to change the base address of the monitor's thread local storage and must be restricted. System calls, including `personality`, `set_thread_area`, `seccomp`, `ptrace`, `execve`, coredump and others [54, 61] that functionally affect the monitor's system calls, need to be prohibited or emulated.

System Calls Impacting Control Flow and Context Signals are necessary for compatibility and are widely used for Inter-Process Communication and process state control, such as NGINX using signals to implement timers and reload configuration files. When delivering a signal, the kernel interrupts the current process, resets `PKRU`, and jumps to the start of the signal handler. Similarly, the `sigreturn` system call interprets the data on the stack as a sigframe and uses its contents to restore the PKRU register and control flow. The problem is that altering the control flow will leak privilege and context information to untrusted domains. However, the kernel's implementation of signals does not take into account the presence of different domains within a process, making it insecure. Moreover, `sigreturn` can be abused because the kernel does not track whether a signal genuinely exists. `rseq` can also be used to alter control flow and therefore needs to be restricted.

Instructions Bypassing MPK and Altering Monitor State The following instructions can compromise the state of MPK or an intra-process monitor: `XRSTOR`, `WRPKRU`, `swapgs`, and TSX instructions. ERIM mitigates these instructions through binary scanning. That means we must ensure that all pages mapped as executable by `mmap` and `mprotect` adhere to a Write-XOR-Execute policy after being inspected by the monitor, and prevent methods such as shared memory from changing executable pages without the monitor's knowledge.

2.3 Threat Model

We assume that implementations are complete and bug-free, with a focus on attacks that exploit system interfaces. We assume that user domains may have vulnerabilities, thereby providing attackers with opportunities to undermine the monitor's integrity. We assume an attacker is present in one of the isolated user domains, with full control over that domain. Under the constraints of both the domain and the monitor, the attacker can execute arbitrary code, perform jumps, and read/write memory in an attempt to escape. We assume that the monitor is loaded and executed prior to any user code execution, thereby enabling it to run before any user code is

executed. We consider Denial-of-Service (DoS) attacks and side-channel attacks to be outside the scope of our work.

3 Motivation

Despite previous efforts to identify vulnerabilities in MPK and intra-process monitors, we found that their approaches to inspecting system calls, managing multithreading, and handling signals were flawed, leading to security risks and performance issues. This is because the nested nature of intra-process monitors presents unique challenges in achieving security and effectiveness while ensuring backward compatibility. In this section, we will outline the challenges faced by intra-process monitors and our proposed solutions to tackle these issues.

3.1 Inside-out System Calls Analysis

The kernel always has the privilege to bypass MPK. It is up to the monitor's policy to ensure that the kernel does not abuse this privilege. Apart from system calls used to manipulate page protections, the kernel can also create temporary mappings to access physical pages, which is referred to as highmem, bypassing MPK protection. The kernel temporarily maps pages into the highmem area to access the virtual memory of a remote process in these system calls. The new mappings do not include MPK information, thereby bypassing MPK protection. Typical examples include `/proc/PID/mem`, `process_vm_readv` and coredump. However, previous work has only deduced potentially threatening interfaces from the functionality of system calls. Instead, we identified that all such accesses originate from the same set of kernel APIs `kmap_*`. Therefore, we retrieved all the code in the kernel that uses this API, traced back the system calls that utilize them, and prohibited the use of these calls. Most of the usage, such as in `user_events_data`, appears in filesystem-related code or drivers triggered through `ioctl`, but we have discovered some rather special uses that cannot be directly traced back to system calls. Therefore, we conducted some manual analysis. For example, `sendmsg` with `MSG_ZEROCOPY`. Among them, the exploitation of `sendmsg` is very subtle because during the system call the kernel correctly checks the privileges of the virtual memory. After passing the check, the driver can directly access the physical pages of this virtual memory to achieve zero-copy at any moment after the system call returns. This leaves us with an exploitation opportunity. Since `mmap` and `mprotect_pkey` are two separate syscalls, we can exploit the window between these calls to bypass the kernel's checks. Consequently, when the network driver sends data, it accesses the data that has been protected by MPK. By identifying the common kernel APIs that enable these bypasses, we can formulate more targeted security policies to prevent such escapes and respond immediately as new functionalities are provided in the kernel.

3.2 Secure Signal Design

The signal provided by the kernel is not secure for the monitors. Previous monitors attempted to patch and reuse the kernel's signal infrastructure. However, all patches are limited by the semantics of system calls provided by the kernel, making security hard to achieve. For example, most monitors need to maintain the current context's information and switch the `PKRU` register upon exiting the monitor at the same time. Yet, sigreturn can only switch CPU registers without changing the monitor's context information, leaving space for attackers. Both Jenny [54] and μSWITCH [50] attempted to fix the kernel's interface but failed. This is due to their inability to effectively manage state synchronization and data security during signal delivery and sigreturn processing As an intermediary layer between the kernel and user programs, intra-process monitors require their own mechanism to maintain and deliver signals without relying on the kernel. This allows the delivery of signals and the state of sigreturn to be entirely under the monitor's control. Moreover, by avoiding additional kernel context switches, the performance of signal delivery with a monitor is also improved.

3.3 Thread-safe Monitor

Threads enable multiple execution flows within a single process using the shared memory space. Even servers with single-threaded event loops, such as NGINX, use threads to offload tasks like disk I/O. Multithreading introduces more attack vectors and synchronization challenges for the security of intra-process monitors. Consequently, some works do not support multithreading, while others overlook the security issues introduced by multithreading. Monitors track the ownership of objects such as memory pages and file descriptors, using this information to ensure the validity of system calls from users. However, this information is not always consistent, creating a gap between the kernel and userspace monitor. Figure 2 illustrates this dilemma, where a critical race exists between system calls and state updates, leading to a loss of synchronization between the kernel and the monitor. Specifically, within the monitor, we need to check the user's parameters and fulfill the application's request through one or more system calls. The kernel's state is updated during the execution of these system calls, and the kernel uses locks to ensure its consistency. However, the state in the monitor does not automatically update after the system calls finish. Attackers can disrupt the operation of the monitor within the current thread by sending signals or exploit these unsynchronized states through TOCTOU attacks, causing the monitor in other threads to use incorrect states for checks. Also, attackers can exploit the state that fails to update promptly after the system call returns. For instance, attackers can first open a legitimate file and read it. After passing the policy inspection, we can close it in another thread, and race against another

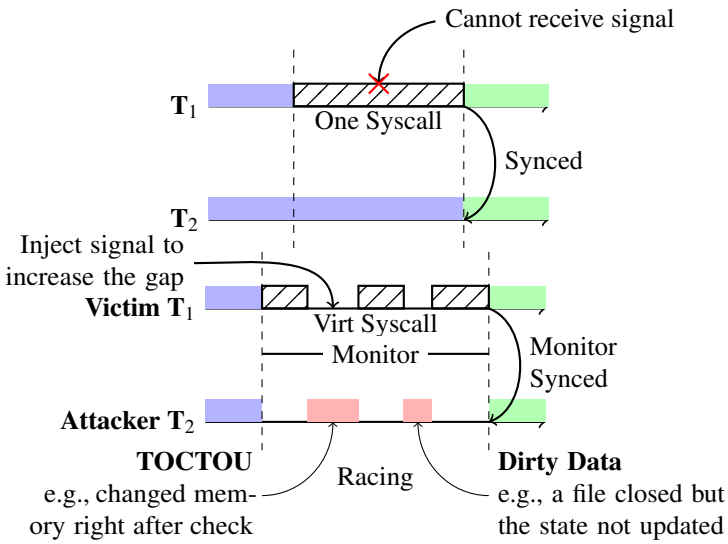


Figure 2: General Pattern of a Monitor Synchronization Problem caused by the gap between the system calls and the monitor; ▨ = Synchronized blocks (e.g., locked). ■ = State before syscall; ■ = State after syscall; ■ = Racing Area (data mismatches between kernel and monitor expose vulnerabilities);

domain to open its private file using the same file descriptor ID. Unless locks cover the entire system call, leading to issues with availability, this desynchronization is unavoidable. We developed principles for updating states that can handle desynchronization effectively. This is to ensure that the object will always use the least privileged level among the states before and after a system call. By doing so, we guarantee that the monitor always assumes the worst-case scenario, regardless of the order in which threads execute.

4 Endokernel

In this section, we define the intra-process protection model provided by Endokernel and the resources and policies it involves. This model outlines the identification of each isolated entity within the process, their methods of interaction, and the rules for allocation, sharing and destruction of resources they hold. Through this model, we define the internal boundaries of isolation within the process, identifying the objects and policies that the intra-process monitor must protect. By integrating it with security analysis, we have summarized and identified all invariants that the intra-process monitor needs to maintain during emulation or system calls. Thus we ensure the integrity of the monitor and isolation domains when handling system calls or introducing new functionalities, by using these invariants as a checklist.

4.1 Privilege Model

We define the following protection model, where a subprocess representing different privilege entities can run concurrently within a process space, orthogonal to threads. All privilege definitions are with respect to this subprocess and are divided into two parts: resources and policies. A subprocess can uniquely control certain resources: **memory**, **system objects**, and **entry points**. **System objects** include resources acquired during the subprocess's execution. **Memory** and **entry points** are established when creating a subprocess, such as isolating a segment of the program's code and data to secure it within a subprocess. This subprocess can also dynamically obtain additional memory via system calls.

Entry points are set during the initialization phase of subprocess. They are the functions through which access to the subprocess is permitted. Other subprocesses can switch to this subprocess only through these designated functions. Each subprocess implements three types of policies: **System Limitation**, **Call Permission**, and **Memory Sharing**. The **System Limitation** policy restricts access to certain system interfaces within the subprocess, such as specifying which files can be accessed, or entirely blocking system calls. The **Call Permission** policy governs whether a subprocess can call into another subprocess. And, the **Memory Sharing** policy determines the conditions for one subprocess to access the memory of another. Together, these policies outline the permissible interactions and resource access between subprocesses.

Our nested intra-process monitor, Endokernel, provides the above isolation guarantees for user programs. It ensures the integrity of its own policies and those of its subprocesses across system interfaces, system call emulation, virtualization, and multithreaded environments.

4.2 Invariants

Endokernel must ensure that the following invariants are maintained at any point during user-space execution. These invariants serve as the fundamental security guarantees provided by our monitor, and they act as the foundation for building policies for other subprocesses. We will demonstrate how we maintain these invariants within a complex process space when introducing the implementation of Endokernel in § 5.By upholding these guarantees, we can effectively identify operations that require the attention of the monitor when dealing with complex interfaces and kernel behaviors.

Synchronization guarantee

Invariant 1 (Weak Metadata Synchronization) : The memory metadata refers to the corresponding virtual memory's read, write, and execute permissions, as well as the subprocess ownership. Although the kernel can ensure synchronization between the operating system's data structures and process page tables before and after system calls, the monitor still

needs to maintain the subprocess ownership of addresses and ensure that this ownership information is correctly mapped onto the operating system's memory protection keys. Achieving synchronization in some situations can be challenging. To address this, we define the **weak synchronization** requirement, which ensures that metadata permissions are always **less than or equal to the actual permissions**. This requirement ensures that permissions used by the monitor remain appropriately constrained, preventing synchronization-related security issues. An illustrative example of a security violation resulting from failed synchronization is the bug in Jenny, detailed in §6.1.

Memory-related guarantees

Invariant 2 (Memory Policy Integrity) : All memory reads and writes occurring during the execution of any subprocess must adhere to their memory access policies or the memory policies set when sharing the memory. This includes memory reads and writes resulting from system calls and operations performed by the process in-process monitor *on behalf of* the current subprocess.

Invariant 3 (Page Integrity) : The mapping of virtual addresses to physical addresses and permission settings for memory associated with a subprocess can only be changed by the subprocess that owns the virtual address. Sharing of a page is only allowed if all subprocesses with a shared mapping to the page grant permission for sharing.

Control Flow-Related Guarantees

Invariant 4 (System Callgate and Path Integrity) : Every system call must have a path start with the Endokernel, indicating that a specific entry point for the in-process monitor needs to be active in the current context when the `syscall` instruction is executed. This invariant allows us to guarantee that checks for system calls are consistently performed, ensuring security and integrity without compromise.

Invariant 5 (endocall Integrity) : Any switch between subprocesses must either begin execution through a function call from one of the specified entry points (endocall), or, after the Subprocess has finished executing, return to the calling Subprocess and return address. Furthermore, any cross-Subprocess call must satisfy the *Call permission* defined in the privilege model.

Invariant 6 (Subprocess Atomicity) : After an endocall or invocation of the monitor, the target's execution process is invisible to the caller. This does not imply thread safety for the target; rather, under the guarantee of **I5**, the caller is blocked and only returns after the execution has completed, without being able to interfere with the target's execution state during the process. Please note that only the basic CPU state, such as registers, is protected here, without considering other shared global data or the influence of other threads.

Instruction-Related Guarantees

Invariant 7 (Instruction Blacklist) : Some instructions are considered privileged and are therefore prohibited from being used during the execution of a subprocess. For example, `WRPKRU`, `syscall`, and other instructions. Since the monitor and other subprocesses share the same process space, we require that even when jumping to a blacklisted instruction legitimately created by the monitor during execution, their execution will not have any actual effect.

Complete Mediation

Invariant 8 (Complete mediation) All system resources, during their creation, operation, loading, access, callbacks, duplication, and destruction, are intercepted and subject to inspection by the Endokernel for filtering or virtualization. This includes the restrictions on file systems, virtualization of signals, threads and analysis of the functionality and parameters of known system calls.

5 Endokernel Prototype

To meet all the requirements of the Endokernel, we have designed a basic framework for an intra-process monitor running on Linux, as shown in Figure 1. On the left side are the system components of the Endokernel, and on the right side is a simplified subprocess provided by these components for user privilege definition. In typical applications, developers can compartmentalize program modules based on their origin, importance, and trustworthiness. Modules are assigned to different subprocesses– safebox, main, and sandbox– each with a specific security level and default privileges, facilitating seamless integration. For instance, an isolated parser module can ensure that it cannot access any other modules by default. This also means a limitation on the isolated parser module's access to system calls and other subprocesses. For web engines like NGINX, protecting OpenSSL can prevent other modules from accessing its critical keys. Applications can have multiple safebox subprocesses, and access between them will also be restricted. Furthermore, developers have the ability to define custom isolation methods with specific access policies based on their application requirements, such as achieving further separation of glibc.

In a nutshell, Endokernel is built upon intercepting and securely isolating user system calls, offering fundamental protection to operating system interfaces through system call virtualization, such as filtering dangerous system calls. When encountering operations that require updating the state within Endokernel, such as threads, signals, memory, and file systems, the virtualized syscall dispatches the operation to the respective module for processing and returns the results to the user. The core objectives of the components provided by Endokernel are to: 1) Filter and inspect all system calls we have identified, ensuring the security of Endokernel and sub-

process; 2) Update and maintain Endokernel's internal state in a multithreaded environment, ensuring that decisions in (1) are based on correct information; 3) Preserve the integrity of Endokernel and subprocess's internal state and control flow, with the kernel signal delivery; 4) Offer the same interfaces and behaviors as Linux under secure conditions, maximizing system compatibility. On top of this, we have focused on examining the inevitable gap between such an intra-process monitor and the kernel, as well as the resulting security risks and mitigation methods.

Virtualization of System Calls Endokernel can virtualize system calls, which allows us to ensure **I8** by providing more complex virtualization for those system calls that cannot be simply allowed or denied across subprocesses. We classify them into four categories: those that need to be fully virtualized, partially virtualized, passed through, and prohibited. Full virtualization means that we will implement function code in the Endokernel similar to the same functionality in the kernel, somewhat like Usermode Linux, meaning that the actual providers of these features will be Endokernel. In other cases, we will pass the system call to the kernel for execution, but we will check the parameters and return values to ensure that the system call is executed safely. Alternatively, we will simply pass through the system call to the kernel if it is harmless (e.g., `getppid`). Finally, for system calls like `rseq`, we simply prohibit them. This enables us to:

1. Prohibiting specific system calls (privileged, Codemodify_ldt, `io_uring`, etc.)
2. Allowing memory map to track memory used by subprocess, ensuring proper isolation and management.
3. Enabling file system protection and restrict access to files or other system objects, enforcing access policies.
4. Supporting virtual signal and thread with Linux-compatible interfaces, ensuring compatibility with existing applications.
5. Restricting the set of allowed instructions (**I7**) that can be loaded as executable, preventing the execution of potentially harmful instructions. We implemented the instruction blacklist using ERIM's binary scanning.
6. Virtualizing `exec` to ensure that Endokernel is loaded first in new processes, maintaining the integrity of the security mechanisms.

Secure System Call Isolation We adopted nexpoline [70] and Control Flow Enforcement Technology (CET) to secure the system call interface in the userspace. Nexpoline is a secure and lightweight system call interception mechanism. We use nexpoline to partition user space into trusted and untrusted portions while ensuring **I4** of the Endokernel. With nexpoline, only the trusted portion can make system calls. Nexpoline leverages seccomp or SUD to control the privilege of syscalls, along with MPK, to transform system calls in user space into permissions determined by MPK. Nexpoline consists of a trampoline responsible for context switching, default policies necessary to protect the trusted portion, and signal redelivery. In this work, we utilize nexpoline's trampoline and extend the implemented policies and virtualization on the system interface like signal to support subprocess isolation. In addition to that, we also attempt to achieve **I4** through a global control flow integrity policy. By enabling CET, we only need to confine the range of the syscall instruction within the monitor's code segment. And, by generating the endbr64 instruction only at the only entry point of the monitor, and using seccomp or SUD to limit the caller's IP address of the syscalls.

Virtualization of Signal The Virtualized Signal is responsible for ensuring **I5** and **I6**. Due to the complexity of signals and the limitations of system calls, merely filtering parameters cannot safely permit the use of signals. Any such implementation would either be insecure, incomplete, or both. The registration, delivery, mask and return of signals must be fully emulated by Endokernel to ensure that these invariants are not violated by signals. All signals are registered with an entry-point to the Endokernel's signal handler. The `sigprocmask` is used to prevent the signal handler from being re-entered and it gets unset only when returning from the Endokernel's signal handler, ensuring **I1**. The monitor controls the delivery of signals and ensures **I5** and **I6**. When a signal occurs while the program is executing in the monitor, our signal virtualization pushes it in the pending queue in the monitor, and uses `sigreturn` immediately to continue the execution of the monitor. When a program is executing in a subprocess, the monitor attempts to deliver signals immediately if they are not masked. If a signal cannot be delivered at that moment, it is also queued. Upon returning from the subprocess or the monitor, the program checks the queue for any signals that can now be delivered. If there are deliverable signals in the queue, we create a signal context using the state of the subprocess to which it will return and delivers the signals to that subprocess. This ensures that signals do not cross subprocess boundaries, as they are only generated and handled within the context of their respective subprocesses. In order to secure the signal stacks, we utilize `sigaltstack` to provide a signal stack for signal delivery, thereby preventing threads running in the same process from tampering with the stack. To be noted here, the kernel has a flaw when delivering signals to the user when MPK is enabled and can cause kernel panic. In our design, we fixed this bug with a kernel patch, but not changing any other behaviors.

Filesystem Multiplexing The file system contributes partially to **I3** , considering the access to /proc/self/mem pointed out in Pitfall [15]. In addition, the file system implements extra access restrictions for system objects in the privilege

model, including forbidding accesses to specific files and accessing file descriptors belonging to other subprocesses. The maintenance of this information also needs to adhere to the **I1**, which we will detail in the following paragraph about memory management. It is worth noting that the file descriptors themselves are still the same within the process space, but their accesses are restricted, which is more convenient for sharing between different modules.

Importantly, considering the unique form of io_uring bypassing system calls for access, we cannot allow user programs to directly use such shared queues.

Memory Management Memory Maps provides the necessary information for the monitor to define and enforce **I2**. To achieve this, the monitor tracks or virtualizes a series of system calls that modify the memory metadata. For example, we record the ownership and access permissions of memory during `mmap`, ensuring that this ownership is not violated in subsequent accesses. The main challenge of this component is the inevitable synchronization issue. We still rely on the operating system to allocate, release, and protect memory. Simultaneously, the information also needs to be maintained within Endokernel. During the gaps between system calls, there will inevitably be moments when the access permissions recognized by the operating system and hardware are inconsistent with those maintained in Endokernel's data structures, leaving potential opportunities for attacks.

In our implementation, we adhere to the concept of **I1** by conservatively handling the information in Endokernel. The principle involves maintaining permissions at the minimum necessary level during system calls, with a priority on lowering permissions and deferring any elevation. For example, before executing the mprotect system call, we first lock the data structure within the monitor that maintains page permissions. We check the current subprocess's access permissions, then update the record to the lesser privilege before and after the mprotect call. Afterward, we unlock it and proceed with the system call. Once the call is completed, we lock and update the monitor's records to the actual permissions. This approach ensures that in all critical states, attackers will not gain more privileges than they should actually have. By doing so, we effectively maintain the integrity of memory policies and minimize potential security risks.

Multithreading The creation of a thread implies that a new subprocess will inherit the current running state and start executing. During this process, Endokernel must create the relevant thread local storage to track the new context's state based on the old state, such as the currently executing subprocess and nexpoline. This ensures that **I5** will not be forgotten or affected by the creation of new threads. Also, the synchronization and racing of the system objects and monitor's data structures among threads have profound effects on the correct virtualization of system calls and **I1**.

Trampoline and Context Switching Endokernel needs to maintain information across all subprocesses, including resources, privileges, restrictions, and entry points, while also ensuring a secure trampoline for switching between them. This serves as the source of information for the privilege model and **I5**. Here, since this information and trampoline code reside in user space and are always executable, similar to nexpoline, we also need to prevent attackers from bypassing trampoline checks or exploiting instructions such as `WRPKRU` or others through arbitrary jumps.

Additionally, we cannot prevent the kernel from delivering signals during the execution of the trampoline. Therefore, we also need to rely on signal virtualization to avoid signal delivery when the context is in a critical state. When the trampoline finishes and returns, we will perform signal delivery checks for the subprocess about to resume, thereby delaying signals that occurred during the call. From the caller's perspective, the signals appear to occur at the moment the trampoline returns. This is similar to the situation in Linux where signals are delivered during a system call, thus ensuring the implementation of **I6**.

Adapting Existing Applications To integrate Endokernel into an application, developers need to inform Endokernel about the data, program addresses, entry points, and required access permissions of the subprocess to be protected through special system calls. Due to MPK limitations, these addresses must occupy separate pages to avoid affecting other subprocesses. To simplify this process, as illustrated in Figure 3, we designed a series of header files that streamline these steps. By defining the relevant information of the subprocess to be isolated and annotating the target code using the provided macros, our header files automatically add the corresponding attributes, ensure the memory layout complies with Endokernel requirements, and generate the necessary initialization calls based on the desired isolation type. Subsequently, the function calling the subprocess can be invoked using the `xcall` macro. Furthermore, our approach supports the isolation of shared libraries. By providing Endokernel with the addresses of these libraries through system calls, Endokernel automatically scans their linkmaps, protects the relevant memory, and redirects library function calls through `xcall` using PLT hooks. Additionally, we provide a header-only allocation library that allows subprocesses to obtain protected heap memory.

6 Evaluating Endokernel Runtime

In this section, we evaluate the security, performance, and compatibility of Endokernel virtualization, *i.e.,* the Endokernel transparently executing the main application while isolating itself. We introduce a few novel attacks exposed through our exploration.

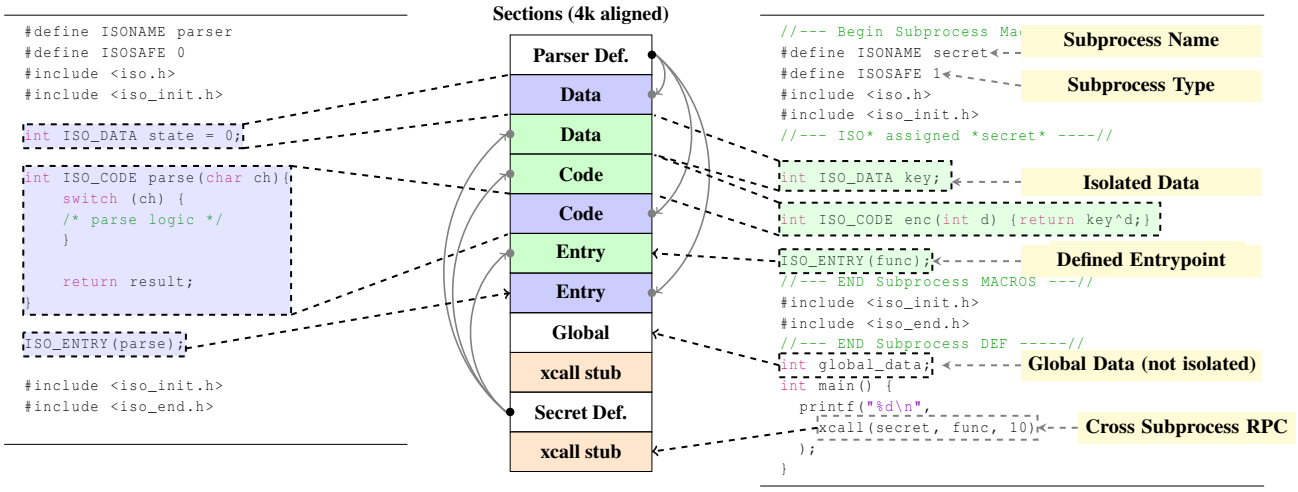


Figure 3: Compartmentalized Application using Endokernel

Attack	Our Work	Jenny [54]	Cerberus [61]	ERIM [60]
Inconsistency of PT Permissions [15]	●	●	●	○
Mutable Mappings [15]	●	●	●	○
Changing Code by Relocation [15]	●	●	●	○
Modifying PKRU via sigreturn [15]	●	●	○	○
Race condition in Scanning [15]	●	●	●	○
Determination of Trusted Mappings [15]	●	●	●	○
Influencing with seccomp [15]	●	●	●	○
Modifying Trusted Mappings [15]	●	●	●	○
Syscall TOCTOU Attack [54]	●	●	○	○
Page Table Syscalls Abuse [54]	●	○	○	○
`rseq` Control Flow Hijacks [54]	●	●	○	○
Forged Signal Delivery [70]	●	●	○	○
New Attacks				
Incorrect Signal Return Handling	●	○	○	○
Fork and Retry Attack	●	○	○	○
High Memory Access Abuse	●	●	○	○
TSX [36] Instruction Probing	●	○	○	○
Inconsistency of Monitor and Kernel Status via Syscall	●	○	○	○

Table 2: Quantitative security analysis based on Pitfalls [15] and original attacks. ○ = vulnerable ● = prevented.

6.1 Security Evaluation

As summarized in Table 2, Endokernel is able to monitor those known attack methods [15] at the system call level and define specified policies to mitigate them. In addition to that, 5 more classes of attack have been identified by this work. We explored a combination of multiple attack dimensions, including privilege escalation via unattended syscalls, signals, multi-threading, multi-domain, other CPU features like TSX instructions, and more race conditions that create inconsistency between the monitor and the OS.

Incorrect Signal Return Handling The signal stack must use a PKEY different from the user's to prevent tampering attacks [15]. But, the kernel only uses the default PKRU when delivering signals. Jenny [54] modified the kernel to alter the PKRU during signal delivery and revert it upon sigreturn. This necessitates the kernel to track whether it is awaiting a sigreturn. Jenny's method not only disrupts the kernel's ABI guarantees and restricts the use of nested signals but also imposes extra synchronization requirements during sigreturn. Jenny failed to correctly implement synchronization of signal states between the kernel and the monitor, allowing attackers to deceive the kernel into using an incorrect PKRU for sigreturn. This ultimately enables attackers to access privileged PKRU values that should only be allowed to the monitor. While we also modified the kernel to ensure signals will not cause a kernel panic, no changes have been made to its behavior. By simulating the kernel's signal behavior within Endokernel, we completely avoid the attack surface associated with signal delivery and sigreturn.

Forged Signal Delivery At the signal handler's entry point, we use `WRPKRU` to switch to the monitor. However, attackers can jump to this location and forge signals, claiming to have received them from the kernel. Thus, we are left unable to regain the PKRU state prior to the `WRPKRU` switch, complicating the monitor's ability to distinguish genuine signals from counterfeit ones. To mitigate this attack, nexpoline [70] utilizes the fact that the default PKRU from the kernel can write to the monitor's thread local storage. Before the `WRPKRU` switch, we write a flag to the for subsequent validation and clear it after the signal context switch. Because attackers lacking a privileged PKRU cannot forge this flag, they are also prevented from forging signals.

Fork and Retry Attack The fork system call allows an attacker to duplicate the current process, providing two main advantages. First, it grants the attacker in the parent process privileged access and control over the child processes. Second, and more significantly, it enables the attacker to attempt multiple retries of an attack that may not succeed in a single attempt, important for race-based attacks with narrow timing windows. By spawning a large number of child processes, the attacker can repeatedly attempt the attack without affecting the parent process complicating detection. To counter this, we propose limiting the parent process's debugging privileges over its child processes and restricting the maximum number of child processes.

Syscall TOCTOU Attack TOCTOU is a problem that must be addressed for secure multithreading monitoring. An attacker can change the pointer parameters involved in the system call in another thread. For example, use `open("/good")` for the system call and replace it with `"/secret"` after it has passed our filter. We avoid this type of attack by ensuring that all pointers in system calls are copied to Endokernel if we need to examine them.

Page Table Syscall Abuse The most straightforward system call of this type is the series of system calls represented by `mprotect`. However, there are other system calls that operate on page properties that are ignored. In addition to `userfaultfd` [54], we also find `vmsplice` that moves the underlying page directly without copying it. Therefore, the properties of the virtual memory addresses it accesses also need to be examined.

High Memory Access Syscall Abuse The `/proc/PID/mem` file, `process_vm_readv` and `process_vm_writev` system calls allow a process to read and write the memory of another process. As described in Section 3, other system calls that result in accessing high memory may also lead to similar issues and need to be prohibited or have their functionalities limited. Otherwise they can be used to bypass MPK's protection. For example, `/sys/kernel/tracing/user_events_data` can allow the attacker to write 1 or 0 to the memory of the monitor when used with the `ioctl` system call. We limit the access of these system calls using our monitor.

TSX Instruction Probing The TSX [36] instructions are designed to allow the CPU to execute a series of instructions in a transactional manner. If the transaction fails, the CPU will roll back to the restore code with an error code. This allows the attacker to probe the content of an execute-only memory created by MPK and leak information about the instructions executed. Considering that Intel® has removed TSX support in newer processors, we recommend turning off TSX.

`rseq` Control Flow Attack `rseq` allows the developer to configure a critical region where the control flow is transferred to an abort address if the process gets scheduled to another CPU. This allows the attacker to hijack the execution of other subprocesses, even the monitor, to the location specified by the attacker, while the PKRU does not change after the transfer. Although this address needs to start with a specific sequence of bytes, it is not difficult to generate such a gadget via mprotect. And the address range where `rseq` takes effect can be changed at any time by modifying only one structure in user space. Therefore, this system call needs to be blocked to prevent such an attack.

Inconsistency of Monitor and Kernel Status via Syscall Intra-process monitors inevitably need locks for synchronization. Otherwise, maintaining the correct state of system resources and domains within the monitor to decide on policies would be impossible, leading to the races and attack surfaces illustrated in Figure 2. However, we cannot simply put a lock on access to all emulated system calls because this would mean only one thread could make a system call through the monitor at any one time. For instance, when handling long-running IO system calls, we must unlock because the execution of system calls is beyond the control of user space. Synchronization issues like this are unavoidable unless we modify the kernel, similar to μSWITCH, to let the monitor and kernel use the same data structures. This would require extensive modifications to the kernel. Our solution is that each time we unlock and invoke real system calls, we require that the permissions for every resource and domain recorded inside Endokernel are less than or equal to their actual access permissions. Thus, even in cases of inconsistency, Endokernel will not make decisions that are considered incorrect in terms of security. Even if the monitor denies access, such denial is inherently a possible result of multithreaded competition, and therefore, it does not exceed the application's expectations.

6.2 Performance Evaluation

We conducted all experiments on an Intel 11th generation CPU (i7-1165G7) with 4 cores at 2.8GHz, with both Turboboost and hyper-threading disabled. The test system had 16GB of memory, ran Ubuntu 20.10, and used a kernel version 5.9.8 with both CET and SUD backported. Since CET has not yet been integrated into the kernel, we used Intel's CET patches. SUD is already available in the latest kernel. Besides that, our kernel modifications involved a net change of +82 -45 LoC to address a kernel panic bug during signal delivery with sigaltstack. For our micro and application benchmarks, we averaged results over 100 repetitions and examined various Endokernel configurations. Endokernel utilizes nexpoline [70], which allows for different configurations based on mechanisms like seccomp or syscall user dispatch, represented as nex-sec or nex-sud, respectively. nex-sec relies

solely on seccomp for enhanced compatibility, whereas nex-sud requires a more recent kernel version or the backporting of the SUD patch. We also integrated CET with seccomp and SUD to secure the system call interface, indicated as cet-sec and cet-sud. Since nexpoline already meets all the security requirements of Endokernel, and CET serves more as a reference for the performance impact under Control Flow Integrity (CFI), which represents a stronger constraint. We aimed to include a comparison with the ptrace and seccomp based MBOX [37] in our evaluation, but it failed to complete some benchmarks. Observations showed that strace performed better than MBOX by a margin of 2.7%, leading us to use strace for comparison instead. This setup provided a comprehensive framework for evaluating the performance and security enhancements offered by Endokernel in various configurations.

6.2.1 Microbenchmarks

System call overhead: We evaluated Endokernel's overhead on system calls and signal delivery in comparison to native and the ptrace-based techniques. Figure 4 depicts the latency of LMBench v2.5 [45] for common system calls. Each Endokernel configuration and the ptrace-based techniques intercepted system calls and provided a virtualized environment to LMBench. Overall, nex-sud and nex-sec that were based on nexpoline had better performance than CET-based configurations. Endokernel added a fixed cost of $0.5-2\mu$sec per system call for nexpoline based isolation. Endokernel had high overhead for protecting fast system calls, like `read` or `write` 1 byte (126%-900%, approx. 284% for nexpoline only), whereas long-lasting system calls like `open` or `mmap` only observed 29%-150% overhead. We demonstrated the difference by performing a throughput file IO experiment. Figure 5 showed high overheads for reading small buffer sizes, which amortize with larger buffer sizes. Since the overhead induced by Endokernel is per `syscall`, reading a file with larger buffers has much less overhead.

Thread scalability: Endokernel employs the weak synchronization principle to prevent races and TOCTOU attacks. The scalability of Endokernel is showcased in Figure 6 using sysbench [39], which performs concurrent reads of a 1 GB file across a varying number of threads. Due to the locks implemented in Endokernel, the number of `futex` system calls increases with the thread count, reaching optimal performance at 4 threads. The overhead associated with each configuration mirrors that observed in the microbenchmarks. cet-sec and cet-sud decrease by up to 60% because the syscall overheads of CET-based configurations are the highest.

6.2.2 Overhead on Applications

Along with the microbenchmarks, we analyze the performance of common applications such as lighttpd [41], Nginx [46], curl [17], SQLite database [56], and zip [35] protected by Endokernel. Figure 7 shows the overall overhead of each application compared to the native execution. All network traffic is handled locally to avoid any network skew or extra latency, and the client and server are assigned to separate cores to minimize interference.

`curl` [17] downloads a 1 GB file from a local web server. It is a particularly challenging workload for Endokernel, since `curl` makes a system call for every 8 KB and frequently installs signal handlers. In total it calls more than 130,000 `write` system calls and more than 30,000 `rt_sigaction()` system calls to download a 1 GB file. However, libcurl supports an option not to use signals, which reduces the overhead by about 10% on average for Endokernel, but strace gets worse by about 140%.

Lighttpd [41] and Nginx [46] serve a 64 KB file requested 1,000 times by an Apachebench tool [1] client on the same machine. All configurations perform within 94% of native. nex-sud outperforms all other configurations and highlights Endokernel's ability to protect applications at near-zero cost with a throughput degradation of 1%. In contrast, strace has about 30% overhead.

SQLite [56] runs its speedtest benchmark [56] and performs `read()` and `write()` system calls with very small buffer sizes to serve individual SQL requests. Contrary to the microbenchmarks, the difference between configurations is larger. Configurations using syscall user dispatch (nex-sud) observe about 30% less overhead when compared to their Seccomp alternatives. Strace performs poorly with more than 500% overhead.

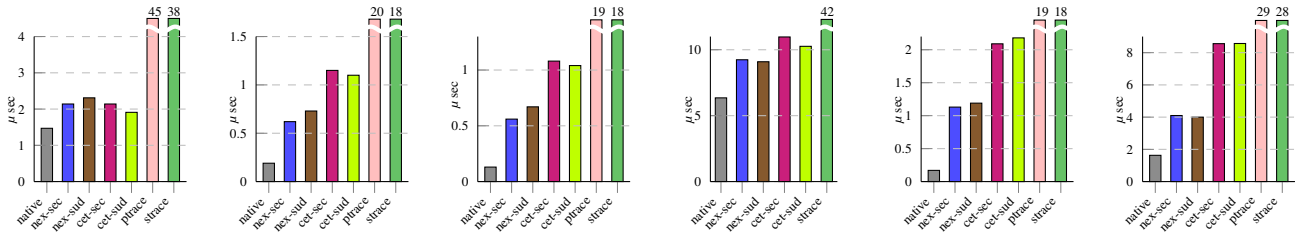
zip [35] compresses the full Linux kernel 5.9.8 source tree which opens all files in the source tree, reads their contents, compresses them, and archives them into a zip file. The observed performance degradation is similar to microbenchmarks for `openat()`, `read()`, and `write()` system calls.

Summary Network-based applications like lighttpd and Nginx perform close to native whereas file-based applications observe overheads between 4 and 55%, and small file access is the worst as in SQLite.

6.2.3 Overhead on SPEC2017

Setup We evaluated the overhead of virtualization brought by Endokernel under compute-intensive workloads on SPEC2017 intrate and intspeed version 1.1.9. All evaluations were conducted with boost and ASLR disabled, intrate running on a single core, and intspeed using 4 threads.

Result As shown in Figure 8, for most tasks, the overhead introduced by Endokernel is within 2%. This minimal overhead is primarily because the tasks in SPEC use fewer system calls. Some tasks show a negative overhead, which is due to the overhead being smaller than the variation between tests. CET configurations introduces additional performance overhead. We employ IBT solely to ensure callgate integrity, while the shadow stack is mandated for global use. Additionally, the CPU must access the legacy map to verify the indirect jumps



(a) `open()` (b) `read()` (c) `write()` (d) `mmap()` (e) `install signal` (f) `catch signal`

Figure 4: System call latency of the LMBench benchmark. Std. dev. below 5.7%.

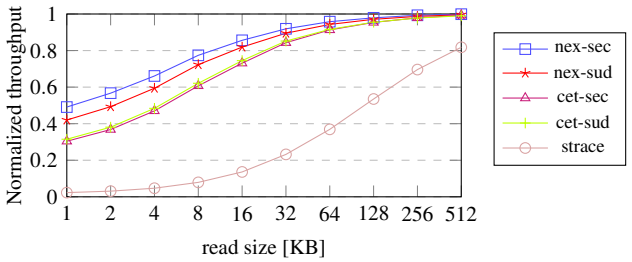


Figure 5: Normalized latency of reading a 40MB file. Std. dev. below 1.5%.

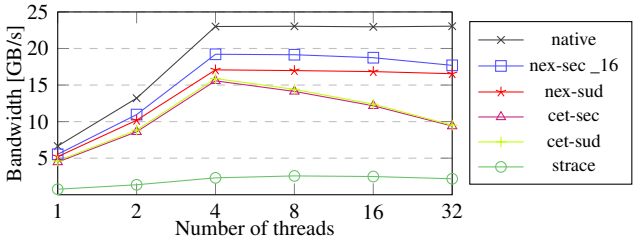


Figure 6: Random read bandwidth for diff. numbers of threads measured with `sysbench`. Std. dev. below 0.7%.

within user applications, further contributing to the overhead.

6.3 Compatibility Evaluation

We use the Linux Test Project (version 20200515) to evaluate the compatibility of Endokernel. LTP is a regression test kernel conformance test of the user's runtime environment. It evaluates whether the Linux kernel and the user runtime behave as expected. Endokernel being an intermediate level that runs in the user and crucially provides kernel functionality to the rest of the user programs, using this test project helps us ensure that the part of the kernel functionality we are emulating is correct and consistent in the boundary condition.

Baseline LTP runs 2136 test cases. The original kernel 5.9.8 reports 188 test cases skipped and 22 test cases failed. We excluded these tests going forward, since Endokernel relies on the kernel for functionality, leaving 1926 test cases in total.

Result After testing, Endokernel only failed 78 out of 1926 or 95.95% of LTP. We categorize the failed tests.

Security Implication (19 / 78): These tests fail because Endokernel protects subprocesses. Endokernel, *e.g.*,, prevents

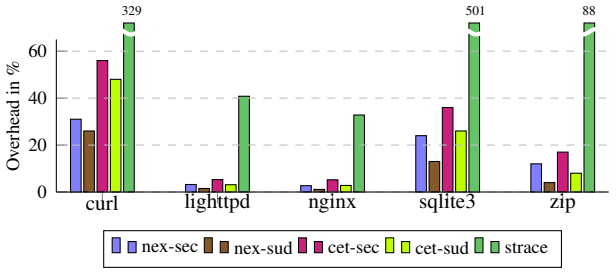


Figure 7: Normalized overhead of diff. Linux applications. Std. dev. below 2.4%.

access to core dumps, ptrace, or changes to seccomp.

Lack of Kernel Functionality (33 / 78): The kernel does not support modifying the PKEY of a shared memory segment. Endokernel cannot overcome this limitation of Intel® MPK.

Inconsistency of Kernel Functionality (11 / 78): Some tests try to probe the functionality of the kernel under certain edge cases. They depend on their own environment resulting in inconsistent or unstable results when the Endokernel is also running. E.g., in the OOM test case, sometimes the kernel not only kills the Endokernel, but also the parent process running it. We consider these tests to have failed for reasons of rigor, although they do sometimes pass.

Secondary Loader (5 / 78): For security purposes, Endokernel must be loaded as the first process. This requirement affects the `O_EXEC` flag on files, which cannot be modified in user space. Some tests use sudo, leading to issues for the kernel to properly set the user ID or require the process name to be different.

Issues Caused by Endokernel (5 / 78): In edge cases Endokernel fails which do not cause disruptions for usual applications. For example, Endokernel has to limit the arguments to an `exec`'d program to forward information to the Endokernel in the other process leading to tests with enormous argument length to fail. Cpuhotplug03 fails to find processes by name due to uniform naming under Endokernel, and mmapstress08 conflicts due to Endokernel's memory allocation strategy.

Unsupported system calls (2 / 78): Linux frequently introduces new system calls and our system successfully blocks unknown `syscalls`.

Racing causes by the kernel (1 / 78): This issue occurs in

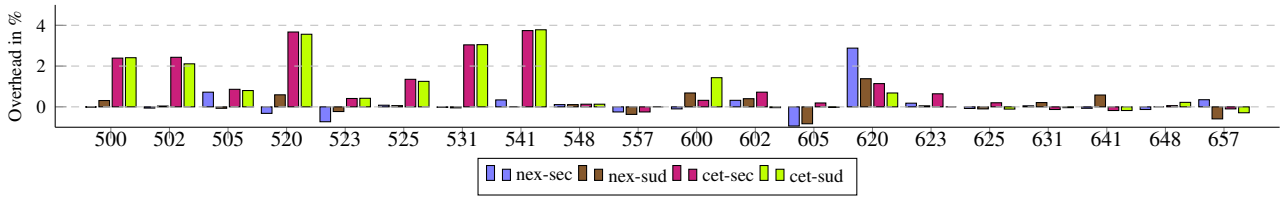


Figure 8: SPEC2017 normalized overhead for different configurations.

hugemmap06 due to the kernel's inability to perform `mmap` and `mprotect` atomically. A brief gap between these operations can lead to failures in concurrent threads. This situation is uncommon because other threads usually do not have knowledge of the addresses involved in these allocations and resembles an attack.

7 Using and Evaluating Subprocess

We demonstrate the extensibility of the Endokernel by using the subprocess virtual privilege rings to 1) eliminate the recent `sudo` vulnerability [19] and 2) implement a significant refactoring of NGINX: memory and system isolation of buggy parsers and restricting sensitive crypto access to OpenSSL while the main portion of the application maintains partial system object access.

Eliminating `sudo` Privilege Escalation. Recently, a bug was found in the `sudo` argument parser allowing an attacker to corrupt a function pointer to gain control with root access [19]. We compartmentalized `sudo` so that the parser code, in file `parse_args.c`, is isolated, and restricted to only the command line arguments and an output buffer. With these changes, the worst possible attack is to overflow the parser's internal buffer causing a segfault, but nothing harmful. In summary, by changing approximately 200 lines of code, importing our libsep in `sudo` and using Endokernel, we confined the argument parser and successfully prevented the root exploit. This applies more generally. Commonly parsers execute with too many privileges and could benefit from similar changes.

Towards a Least-Privilege NGINX We present a novel compartmentalization of NGINX, which allows us to measure the effort and performance costs of improving the security of a complex system while using the Endokernel and subprocess. Our goal is to isolate the NGINX parser and secure the OpenSSL library. By isolating the parser code in NGINX, we address the frequent vulnerabilities and excessive privileges of string handling operations which often require limited functionality. This strategy is commonly used in the work of syscall filtering [3, 22, 26, 47, 63], but often applied at the process-level. We also protected the OpenSSL library using the automatic shared libraries isolation provided by the Endokernel. By isolating OpenSSL, which handles private keys, we prevent potential key leakage. The value of isolation lies in its ability to protect against common exploits (e.g., CVE-2009-2629, CVE-2013-2028, and CVE-2013-2070) utilized

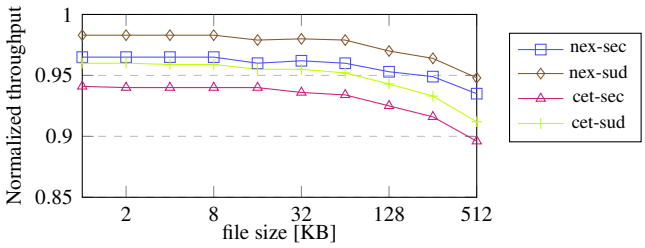


Figure 9: Normalized throughput of privilege separated NGINX using TLS v1.2 with `ECDHE-RSA-AES128-GCM-SHA256, 2048, 128`. Std. dev. below 1.5%.

by attackers to hijack control flows and to leak or compromise sensitive information (e.g., cookies, private responses data, passwords, or keys). The essence of this protection is to prevent the majority of the application from accessing key information, thereby reducing the TCB and safeguarding critical data such as session and private keys.

Performance. We evaluate the least-privilege Nginx shielding OpenSSL and isolating the HTTP parser. We measure the throughput downloading files with varying sizes and normalize to the native performance using ab (see Figure 9). All tests are performed locally with a pinned CPU core. Strace suffers from its interception costs and falls below 50% for large files. The results indicate less than 10% overhead for the different nexpoline techniques and the lowest for nex-sud at 3-5%. The number of system calls increases with file size, leading to decreased performance of Endokernel.

During the initialization of NGINX, 89 `xcalls` are made to initialize OpenSSL, and an additional 16 `xcalls` are needed to start each new HTTPS session. Furthermore, the number of `xcalls` for each connection escalates with file size, ranging from 129 for $1k$ to 312 for $1024k$.

8 Related Work

Compartmentalization enhances software security by isolating components to limit damage from breaches. Various approaches, including software-based, operating system-based, and hardware-assisted methods, offer security benefits but compromise on efficiency or complexity. This section reviews these techniques, highlighting their strengths and limitations in comparison to our intra-process security monitor, which aims for efficient and secure system interface emulation with-

out compromising backward compatibility.

Software-based techniques. Software Fault Isolation (SFI) [62] started a research field to inline security checks within application code. The goal is to translate an application via a compilation pass to enforce security properties such as Control-Flow Integrity (CFI) [2], Code-Pointer Integrity [40], or intra-process isolation [38, 52]. Similar needs arise in web applications, where protections against native code [71] vulnerabilities have spurred the development of WebAssembly [52]. However, they suffer with large performance overheads from inlined security checks and also lack safeguards against operating system-level attack vectors.

OS-based techniques. To address the limitations of traditional process-based isolation, various kernel abstractions have been developed that employ existing mechanisms such as page tables to facilitate intra-process solation. Light-weight contexts (lwC) [42], secure memory views (SMV) [34] and nested kernel [21] introduce a lighter form of isolation. These approaches enable the association of multiple virtual address spaces with a single process, allowing for quicker context switches. Shreds [13] combines OS-based and software-based techniques to enhance performance, while Wedge [8] incorporates runtime checks to strengthen security. However, the performance of these methods is still constrained by the overhead of context switching, an issue that Endokernel addresses through more efficient intra-process isolation techniques using hardware designed specifically for this purpose. SEIMI [64, 68] uses Supervisor Memory Access Prevention designed for ring 0 to restrict access to user memory, offering efficient privilege switching even compared to MPK and MPX. However, it only allows the user space to be divided into two domains and requires a complex hypervisor mechanism to run both user applications and the kernel in supervisor mode. Moreover, these OS-based techniques require the use of customized operating systems, which poses challenges for system maintenance.

Hardware-based techniques. CPU vendors have proposed methods to avoid context switches, for example, by switching address spaces with `VMFUNC`, altering memory access permissions based on domain using `WRPKRU`, specialized instructions like `WRSS`, or pointer capabilities with MTE [5] or CHERI [14, 66]. Each technique introduces its own set of limitations, performance characteristics, and security implications. Dune [7] utilizes Intel VT-x to allow a process to operate in both privileged and unprivileged CPU modes. Secage [43] achieves a similar separation by utilizing Intel®'s `VMFUNC` to switch address spaces in user space. Koning *et al.* [38] develop a compiler that isolates components with varying techniques and demonstrate the performance differences. ERIM, HODOR, Donky, Jenny, and others [32, 48, 49, 54, 55, 57, 60] enable the secure use of MPK for intra-process isolation and demonstrate efficient isolation techniques. Furthermore, EPK [28] expands the maximum number of domains by combining MPK and VMFUNC. Techniques based on MPK require coordination between user programs and the OS to ensure these mechanisms are not bypassed, making them vulnerable to OS-level attacks [15, 61]. CETIS [69] leverages Intel® CET's shadow stack for secure memory, which restricts write operations to these areas using the newly added `WRSS` instruction, avoiding context switches. Nevertheless, it only supports secure and non-secure memory domains and depends on CFI to ensure the security of these instructions. Capacity [23] uses ARM's MTE for intra-process isolation, allowing only authorized pointers to access tagged memory region. While this approach offers more flexible isolation and sharing compared to MPK, it poses challenges in the security model as it relies on the confidentiality of the pointer.

System call and signal virtualization. PKU-Pitfall [15] demonstrated how the operating system interfaces can be used to bypass intra-process isolation and motivated the need for system call virtualization. Linux security module (LSM) [67] intercepts system calls in the kernel and can be used to implement a system call filter as suggested by SELinux [44], AppArmor [6], Tomoyo [31] and Smack [53]. However, LSM as well as large kernel subsystems like the filesystem, would have to be extended to recognize different domains in userspace and its interface does not support modifying arguments making virtualization impossible. Alternatively, Seccomp [16] offers the userspace a programmable system call filter using BPF programs and can be used to limit the OS interface [71]. However, BPF can only access limited system call information, making it incapable of implementing flexible filtering or virtualization. In some configurations, Endokernel relies on Seccomp to prevent system calls outside the trusted monitor.

9 Conclusion

We propose Endokernel and its implementation, a continuation of the minimalist philosophy of microkernel that provides user space isolation. In this process, we addressed synchronization and signal issues that are inevitable for multithreaded intra-process monitors. We proposed an inside-out approach to identify more system call pathways that could lead to MPK bypass, thereby enhancing the reliability, compatibility, and efficiency of intra-process monitors. We evaluated the performance overhead of Endokernel across different use cases and conducted a comprehensive analysis and comparison of its security. We implemented subprocess isolation within Endokernel and isolated OpenSSL and the parser in NGINX to demonstrate Endokernel's practical use case.

Acknowledgments

We thank the anonymous reviewers, our shepherd, Michael LeMay for their feedback, which helped improve this paper. This work was supported in part by National Science Foundation Awards #2146537 and #2008867.

Availability

The source code of the Endokernel prototype is available at https://github.com/endokernel/endokernel-paper-ver/. The current code includes components from ERIM [60] and Graphene libos [59].

References

[1] *Ab - Apache HTTP Server Benchmarking Tool v2.3*. URL: https://httpd.apache.org/docs/2.4/en/programs/ab.html.

[2] Martin Abadi, Mihai Budiu, and Úlfar Erlingsson. "Control-Flow Integrity". In: 2005.

[3] Muhammad Abubakar, Adil Ahmad, Pedro Fonseca, and Dongyan Xu. "SHARD: Fine-Grained Kernel Specialization with Context-Aware Hardening". In: *Proceedings of the USENIX Security Symposium*. USENIX Association, 2021.

[4] AGWA. *AGWA/titus: Totally Isolated TLS Unwrapping Server*. https://github.com/AGWA/titus. (Accessed on 07/07/2023). Oct. 2020.

[5] ARM. *Armv8.5-A Memory Tagging Extension White Paper*. (Accessed on 06/04/2024).

[6] Mick Bauer. "Paranoid Penguin: An Introduction to Novell AppArmor". In: *Linux J.* (2006).

[7] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. "Dune: Safe User-level Access to Privileged CPU Features". In: *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2012.

[8] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. "Wedge: Splitting Applications into Reduced-Privilege Compartments". In: *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2008.

[9] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. "Jump-Oriented Programming: A New Class of Code-Reuse Attack". In: *Proceedings of the ACM Symposium on Information, Computer and Communications Security (Asia CCS)*. 2011.

[10] Sergey Bratus, Michael E. Locasto, Meredith L. Patterson, Len Sassaman, and Anna Shubina. "Exploit Programming: From Buffer Overflows to Weird Machines and Theory of Computation". In: *USENIX; login* 36.6 (2011).

[11] Center for Internet Security. *The SolarWinds Cyber-Attack: What You Need to Know*. en-US. Mar. 2021.

[12] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. "Return-Oriented Programming Without Returns". In: *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. 2010.

[13] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. "Shreds: Fine-Grained Execution Units with Private Memory". In: *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. 2016.

[14] David Chisnall, Brooks Davis, Khilan Gudka, David Brazdil, Alexandre Joannou, Jonathan Woodruff, A. Theodore Markettos, J. Edward Maste, Robert Norton, Stacey Son, Michael Roe, Simon W. Moore, Peter G. Neumann, Ben Laurie, and Robert N.M. Watson. "CHERI JNI: Sinking the Java Security Model into the C". In: *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2017.

[15] R. Joseph Connor, Tyler McDaniel, Jared M. Smith, and Max Schuchard. "PKU Pitfalls: Attacks on PKU-based Memory Isolation Systems". In: *Proceedings of the USENIX Security Symposium*. 2020.

[16] Jonathan Corbet. *Seccomp and sandboxing*. May 13, 2009. URL: https://lwn.net/Articles/332974/.

[17] *CURL: Command Line Tool and Library for Transferring Data with URLs v7.58.0*. URL: https://curl.haxx.se/.

[18] *CVE - CVE-2013-2028*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-2028. (Accessed on 08/10/2023).

[19] *CVE-2021-3156*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-3156. (Accessed on 06/08/2021).

[20] Dino Dai Zovi. *Practical Return-Oriented Programming*. 2010. URL: https://www.youtube.com/watch?v=AmyPzpeFN9k.

[21] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. "Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation". In: *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2015.

[22] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P. Kemerlis. "sysfilter: Automated System Call Filtering for Commodity Software". In: *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. 2020.

[23] Kha Dinh Duy, Kyuwon Cho, Taehyun Noh, and Hojoon Lee. “Capacity: Cryptographically-Enforced In-Process Capabilities for Modern ARM Architectures”. In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2023.

[24] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman. “The Matter of Heartbleed”. In: *Proceedings of the Conference on Internet Measurement Conference (IMC)*. 2014.

[25] FireEye. *Highly Evasive Attacker Leverages SolarWinds Supply Chain to Compromise Multiple Global Victims With SUNBURST Backdoor*. en. 2020.

[26] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. “Temporal System Call Specialization for Attack Surface Reduction”. In: *Proceedings of the USENIX Security Symposium*. 2020.

[27] Adrien Ghosn, Marios Kogias, Mathias Payer, James R. Larus, and Edouard Bugnion. “Enclosure: Language-Based Restriction of Untrusted Libraries”. In: *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2021.

[28] Jinyu Gu, Hao Li, Wentai Li, Yubin Xia, and Haibo Chen. “EPK: Scalable and Efficient Memory Protection Keys”. In: *Proceedings of the USENIX Annual Technical Conference (ATC)*. 2022.

[29] Juan Andrés Guerrero-Saade. *CrateDepression | Rust Supply-Chain Attack Infects Cloud CI Pipelines with Go Malware*. https://www.sentinelone.com/labs/cratedepression-rust-supply-chain-attack-infects-cloud-ci-pipelines-with-go-malware/. May 2022.

[30] H2O. *h2o/neverbleed: privilege separation engine for OpenSSL / LibreSSL*. https://github.com/h2o/neverbleed. (Accessed on 07/07/2023). Apr. 2023.

[31] Toshiharu Harada, Takashi Horie, and Kazuo Tanaka. “Task oriented management obviates your onus on Linux”. In: *Linux Conference*. Vol. 3. 2004.

[32] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. “Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries”. In: *Proceedings of the USENIX Annual Technical Conference (ATC)*. 2019.

[33] Matt Howard. *2021 State of the Software Supply Chain: Open Source Security and Dependency Management Take Center Stage*. https://blog.sonatype.com/2021-state-of-the-software-supply-chain. Sept. 2021.

[34] Terry Ching-Hsiang Hsu, Kevin Hoffman, Patrick Eugster, and Mathias Payer. “Enforcing Least Privilege Memory Views for Multithreaded Applications”. In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2016.

[35] *Info-ZIP’s Zip*. http://infozip.sourceforge.net/Zip.html. (Accessed on 06/08/2021).

[36] Intel. “Chapter 16 Programming with Intel® Transactional Synchronization Extensions”. In: *Intel® 64 and IA-32 Architectures Software Developer’s Manual*. Vol. 1. Intel, 2021, pp. 389–396.

[37] Taesoo Kim and Nickolai Zeldovich. “Practical and Effective Sandboxing for Non-Root Users”. In: Proceedings of the USENIX Annual Technical Conference (ATC). 2013.

[38] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. “No Need to Hide: Protecting Safe Regions on Commodity Hardware”. In: *Proceedings of the European Conference on Computer Systems (EuroSys)*. 2017.

[39] Alexey Kopytov et al. *Scriptable database and system performance benchmark*. https://github.com/akopytov/sysbench. (Accessed on 06/08/2021).

[40] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. “Code-Pointer Integrity”. In: *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[41] *Lighttpd v1.4.55*. URL: https://www.lighttpd.net/.

[42] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. “Light-Weight Contexts: An OS Abstraction for Safety and Performance”. In: *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 2016.

[43] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. “Thwarting Memory Disclosure with Efficient Hypervisor-Enforced Intra-Domain Isolation”. In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2015.

[44] Peter Loscocco and Stephen Smalley. “Integrating Flexible Support for Security Policies into the Linux Operating System”. In: *Proceedings of the USENIX Annual Technical Conference (ATC)*. 2001.

[45] Larry McVoy and Carl Staelin. “Lmbench: Portable Tools for Performance Analysis”. In: *Proceedings of the USENIX Annual Technical Conference (ATC)*. 1996.

[46] *NGINX v1.18.0*. nginx. URL: http://nginx.org/.

[47] Shankara Pailoor, Xinyu Wang, Hovav Shacham, and Isil Dillig. “Automated policy synthesis for system call sandboxing”. In: *Proceedings of the ACM on Programming Languages* 4 (OOPSLA 2020).

[48] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. “Libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK)”. In: PRoceedings of the USENIX Annual Technical Conference (ATC). 2019.

[49] Taemin Park, Karel Dhondt, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. “NoJITsu: Locking Down JavaScript Engines”. In: *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. 2020.

[50] D. Peng, C. Liu, T. Palit, P. Fonseca, A. Vahldiek-Oberwagner, and M. Vij. “μSwitch: Fast Kernel Context Isolation with Implicit Context Switches”. In: *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. 2023.

[51] Niels Provos, Markus Friedl, and Peter Honeyman. “Preventing Privilege Escalation”. In: *Proceedings of the USENIX Security Symposium*. 2003.

[52] Andreas Rossberg. *WebAssembly Core Specification*. Dec. 5, 2019. URL: https://www.w3.org/TR/wasm-core-1/.

[53] Casey Schaufler. “Smack in embedded computing”. In: *Linux Symposium*. Vol. 2. 2008, pp. 179–186.

[54] David Schrammel, Samuel Weiser, Richard Sadek, and Stefan Mangard. “Jenny: Securing Syscalls for PKU-based Memory Isolation Systems”. In: *Proceedings of the USENIX Security Symposium*. 2022.

[55] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. “Donky: Domain Keys – Efficient In-Process Isolation for RISC-V and X86”. In: *Proceedings of the USENIX Security Symposium*. 2020.

[56] *SQLite Database Engine v.3.32.3*. URL: https://www.sqlite.org/index.html.

[57] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. “SoK: Eternal War in Memory”. In: *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. 2013.

[58] Zahra Tarkhani and Anil Madhavapeddy. *Enclave-Aware Compartmentalization and Secure Sharing with Sirius*. Nov. 23, 2020. arXiv: 2009.01869 [cs]. URL: http://arxiv.org/abs/2009.01869.

[59] Chia-Che Tsai, Donald E. Porter, and Mona Vij. “Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX”. In: *Proceedings of the Usenix Annual Technical Conference (ATC)*. 2017.

[60] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. “ERIM: Secure, Efficient In-Process Isolation with Protection Keys (MPK)”. In: *Proceedings of the USENIX Security Symposium*. 2019.

[61] Alexios Voulimeneas, Jonas Vinck, Ruben Mechelinck, and Stijn Volckaert. “You Shall Not (by)Pass! Practical, Secure, and Fast PKU-based Sandboxing”. In: *Proceedings of the European Conference on Computer Systems (EuroSys)*. 2022.

[62] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. “Efficient Software-Based Fault Isolation”. In: *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. 1993.

[63] Zhiyuan Wan, David Lo, Xin Xia, Liang Cai, and Shanping Li. “Mining Sandboxes for Linux Containers”. In: *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 2017.

[64] Zhe Wang, Chenggang Wu, Mengyao Xie, Yinqian Zhang, Kangjie Lu, Xiaofeng Zhang, Yuanming Lai, Yan Kang, and Min Yang. “SEIMI: Efficient and Secure SMAP-Enabled Intra-process Memory Isolation”. In: *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. 2020.

[65] *What Is a Supply Chain Attack? | CrowdStrike*. https://www.crowdstrike.com/cybersecurity-101/cyberattacks/supply-chain-attacks/.

[66] J. Woodruff, R.N.M. 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”. In: *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*. 2014.

[67] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah-Hartman. “Linux Security Modules: General Security Support for the Linux Kernel”. In: *Proceedings of the Foundations of Intrusion Tolerant Systems*. 2003.

[68] Chenggang Wu, Mengyao Xie, Zhe Wang, Yinqian Zhang, Kangjie Lu, Xiaofeng Zhang, Yuanming Lai, Yan Kang, Min Yang, and Tao Li. “Dancing With Wolves: An Intra-Process Isolation Technique With Privileged Hardware”. In: *IEEE Transactions on Dependable and Secure Computing* 20.3 (2023), pp. 1959–1978.

[69] Mengyao Xie, Chenggang Wu, Yinqian Zhang, Jiali Xu, Yuanming Lai, Yan Kang, Wei Wang, and Zhe Wang. “CETIS: Retrofitting Intel CET for Generic and Efficient Intra-process Memory Isolation”. In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2022.

[70] Fangfei Yang, Anjo Vahldiek-Oberwagner, Chia-Che Tsai, Kelly Kaoudis, and Nathan Dautenhahn. *Making 'syscall' a Privilege not a Right*. 2024. arXiv: 2406.07429 [cs.CR].

[71] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. "Native Client: A Sandbox for Portable, Untrusted X86 Native Code". In: *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. 2009.

[72] Nusrat Zahan, Thomas Zimmermann, Patrice Godefroid, Brendan Murphy, Chandra Maddila, and Laurie Williams. "What Are Weak Links in the Npm Supply Chain?" In: *Proceedings of the IEEE/ACM International Conference on Software Engineering*. 2022.