Clair Obscur
The Light and Shadow of System Call Interposition - From Pitfalls to Solutions with K23

Gómez Moreno, Jesús María; Moutafis, Vissarion; Dionysiou, Antreas; Kuipers, Fernando; Smaragdakis, Georgios; Coppens, Bart; Voulimeneas, Alexios

DOI
10.1145/3721462.3770772

Licence
CC BY

Publication date
2025

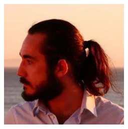
Document Version
Final published version

Published in
Middleware 2025 - Proceedings of the 26th ACM International Middleware Conference

RESEARCH-ARTICLE

Clair Obscur: The Light and Shadow of System Call Interposition – From Pitfalls to Solutions with K23

JESÚS MARÍA GÓMEZ MORENO, Delft University of Technology, Delft, Zuid-Holland, Netherlands



Currently working in security tools leveraging systems knowledge.

VISSARION MOUTAFIS, Delft University of Technology, Delft, Zuid-Holland, Netherlands

ANTREAS DIONYSIOU, Delft University of Technology, Delft, Zuid-Holland, Netherlands

FERNANDO A KUIPERS, Delft University of Technology, Delft, Zuid-Holland, Netherlands

GEORGIOS SMARAGDAKIS, Delft University of Technology, Delft, Zuid-Holland, Netherlands

BART COPPENS, Ghent University, Ghent, VOV, Belgium

View all

Open Access Support provided by:

Delft University of Technology

Ghent University

PDF Download
3721462.3770772.pdf
14 January 2026
Total Citations: 0
Total Downloads: 164

Published: 15 December 2025

Citation in BibTeX format

Middleware '25: 26th International Middleware Conference
December 15 - 19, 2025
TN, Nashville, USA

Clair Obscur[*]: The Light and Shadow of System Call Interposition – From Pitfalls to Solutions with K23

Jesús María Gómez Moreno
TU Delft
Delft, The Netherlands
j.m.gomezmoreno@tudelft.nl

Vissarion Moutafis
TU Delft
Delft, The Netherlands
V.Moutafis@tudelft.nl

Antreas Dionysiou
TU Delft & Frederick University
Delft & Nicosia, The Netherlands & Cyprus
A.Dionysiou@tudelft.nl

Fernando Kuipers
TU Delft
Delft, The Netherlands
F.A.Kuipers@tudelft.nl

Georgios Smaragdakis
TU Delft
Delft, The Netherlands
g.smaragdakis@tudelft.nl

Bart Coppens
Ghent University
Ghent, Belgium
bart.coppens@ugent.be

Alexios Voulimeneas[†]
TU Delft
Delft, The Netherlands
A.Voulimeneas@tudelft.nl

Abstract

System call interposition is a widely used technique to trace and modify application behavior. Over the years, numerous interposition mechanisms have been proposed, each with distinct strengths and trade-offs. Recently, advances in binary rewriting—specifically targeting `x86-64` `syscall` and `sysenter` instructions—have led to new techniques that take important steps forward, with some claiming to support general-purpose use.

We analyze state-of-the-art interposers in depth and uncover several fundamental design and implementation flaws—pitfalls that we collectively term System Call Interposition Pitfalls. For example, prior work cannot reliably interpose all system calls and may even corrupt code and data. These flaws undermine the practicality of existing solutions in real-world scenarios, rendering them unsuitable as universal interposition mechanisms.

Motivated by our findings, we design and implement a new plug-and-play system call interposition approach named `K23`, targeting `x86-64` platforms. `K23` addresses the uncovered pitfalls via a hybrid design that unifies the strengths of prior methods, combining offline and online phases that leverage multiple Linux interfaces and binary rewriting. Our evaluation shows that `K23` overcomes the key limitations of state-of-the-art solutions while remaining highly efficient. To our knowledge, `K23` is the first general-purpose interposer suitable for a wide range of use cases and environments, from low-end devices to performance-critical, datacenter-scale workloads.

ACM Reference Format:
Jesús María Gómez Moreno et al.. 2025. Clair Obscur[*]: The Light and Shadow of System Call Interposition – From Pitfalls to Solutions with `K23`. In *26th ACM Middleware Conference (Middleware '25), December 15–19, 2025, Nashville, TN, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3721462.3770772

1 Introduction

System calls serve as the primary interface between user-space applications and the OS kernel, enabling operations such as file I/O, network communication, and process creation. Because nearly all OS interactions involve system calls, they present a natural interception point for observing and altering application behavior through system call interposition.

System call interposition techniques have been applied across a wide range of use cases, including (i) the construction of sophisticated debugging and tracing tools [9, 15, 23, 29]; (ii) the enhancement of system security [43, 47, 54, 60, 67, 68, 80, 103, 107, 118, 122, 123, 135, 143] and reliability [75, 76, 100, 114, 115]; and (iii) the emulation of alternative OS environments [27, 38, 61]. Beyond these, system call interposition can also (iv) support binary compatibility layers for emerging OS subsystems [86, 94, 105, 108, 117, 119, 130, 131], (v) transparently redirect network operations to custom user-space stacks [78, 157], and (vi) support forensic analysis [56, 63, 145, 146].

System call interposition has many flavours, but no single mechanism offers a universal solution suitable for all use cases. Linux provides several interfaces for this purpose, including `ptrace` [21], `seccomp` [24], and Syscall User Dispatch (`SUD`) [30]. However, each of these mechanisms has notable limitations, such as performance overhead or ***constraints on the interposer's expressiveness (i.e., the capability of the interposer to access application state and execute actions in response).*** For example, `ptrace` and `SUD` introduce additional context or mode switches that degrade efficiency [57, 76, 77, 125, 140, 157], whereas `seccomp` either incurs comparable performance overheads or restricts the interposer's expressiveness—such as lacking support for deep inspection of pointer arguments—depending on how it is configured [64, 77, 125].

Alternatively, other approaches leverage binary rewriting by replacing ***`x86-64` `syscall`/`sysenter` instructions that trigger system calls with `jmp`/`call` instructions to the interposer's code [31, 45]***. This allows system call interposition without additional context/mode switches, achieving maximum efficiency while

[*] *Clair Obscur* refers to the artistic use of strong contrasts between light and shadow. We use it metaphorically to highlight the antithesis between pitfalls and solutions in system call interposition. The term also appears in *Clair Obscur: Expedition 33*, an acclaimed video game by Sandfall Interactive.
[†] Corresponding author.

preserving the interposer's full expressiveness. However, these techniques rely on precise and correct static binary disassembly and rewriting—a well-known hard problem, especially for architectures with variable instruction lengths like `x86-64`—making it infeasible to reliably interpose all system calls in real-world scenarios [77, 157]. This limitation is particularly problematic in settings that require ***exhaustive interposition (i.e., the ability to reliably interpose all system calls)***, including sandboxing [72, 125, 135, 144, 156], introspection and analysis tools [9, 15, 23, 29], vulnerability discovery [80], incremental development of new OS layers [94], automated software updates [114], and sanitization mechanisms [153].

To overcome the aforementioned limitations, both industry and academia frequently turn to intrusive approaches such as OS or hardware modifications [72, 125, 126, 135]. While often effective, these solutions are painstaking to maintain, expand the Trusted Computing Base (TCB), and ultimately hinder broader adoption and usability. More recently, advances in binary rewriting—namely `zpoline` [157] and `lazypoline` [77]—have claimed to enable ***flexible (i.e., adaptable to a wide range of use cases)*** system call interposition while sidestepping the challenges of traditional binary rewriting approaches [31, 44, 45, 76, 109, 114].

However, our in-depth investigation of `zpoline` and `lazypoline` uncovers several design and implementation issues that undermine their claims. ***For example, both techniques fail to reliably interpose all system calls and can corrupt code and data***. We collectively refer to these shortcomings as ***System Call Interposition Pitfalls***[1], as they restrict the applicability of these approaches in real-world deployments.

Our findings lead us to the conclusion that no existing system call interposition mechanism serves as a general-purpose solution.

The main focus of this work is twofold: first, to shed light on the limitations of existing state-of-the-art system call interposers; and second, to provide a general-purpose solution that addresses these limitations for `x86-64` platforms. To that end, we analyze the designs of both `zpoline` and `lazypoline`, along with their corresponding open-source prototypes [11, 40]. Building on these insights, we develop `K23`[2], a new interposer that overcomes the identified challenges through a combination of offline and online phases, leveraging multiple Linux interfaces, binary rewriting, and principled engineering choices.
Our paper makes the following contributions:

- We identify System Call Interposition Pitfalls—fundamental flaws in state-of-the-art interposition techniques—and develop targeted Proof-of-Concept (PoC) programs that expose them. We also demonstrate real-world use cases impacted by these pitfalls.
- We introduce `K23`, the first plug-and-play interposer for `x86-64` that overcomes these pitfalls—ranging from lack of exhaustiveness to code and data corruption—while providing flexibility across a wide range of use cases.
- We conduct a comprehensive evaluation of `K23`, comparing it with state-of-the-art interposers [77, 157]. `K23` achieves consistently high efficiency while uniquely addressing the pitfalls that affect these prior approaches.

Our goal is to provide a general-purpose solution that the community can adopt and extend. To that end, we release our code at **https://gitlab.com/tudelft-ssl/k23**, including the `K23` prototypea and our evaluation framework.

2 Background

In this section, we provide background on the Linux interfaces that `K23` relies on (`ptrace` [21] and Syscall User Dispatch (`SUD`) [30]), as well as on state-of-the-art system call interposition techniques—`zpoline` [157] and `lazypoline` [77].

2.1 `ptrace` and `SUD`

`ptrace` is a Linux interface that allows a ***tracer*** thread to observe and control the execution of one or more ***tracee*** threads. It enables exhaustive interposition of system calls, signals, and even individual instructions, allowing fine-grained control over each tracee's execution. Additionally, the tracer can access the state of tracees using system calls such as `process_vm_readv` and `process_vm_writev` [19]. Due to its exhaustiveness and expressiveness, `ptrace` has been widely used for debugging [9, 15, 23], security [51, 57, 80, 82, 138, 141], and reliability purposes [75, 100]. While `ptrace` offers the level of control required for a general-purpose system call interposer, ***it suffers from prohibitive performance overhead*** due to frequent context switches and the need to issue multiple system calls even for basic operations, such as accessing a tracee's memory [57, 76, 77, 125, 140, 157]. Consequently, it is not a viable all-around solution, particularly in use cases where performance is critical.

`SUD` is a relatively recent Linux mechanism that enables system call interposition by delivering a `SIGSYS` signal to a user-space handler when a system call is invoked. SUD operates at the thread level and can be enabled using the `prctl` system call [18]. Each thread can enable or disable interposition via a dedicated `selector` byte in user space, which dictates whether system calls made by that thread should trigger a signal or proceed normally. Optionally, `SUD` bypasses interposition entirely for system calls made from within a designated address range in the application, regardless of the `selector`'s current value. This allowlisted address range is typically reserved for trusted or internal code paths where interposition is either unnecessary or could interfere with control flow.

In a typical setup, the `SIGSYS` handler begins by disabling interposition through the `selector`, executes the interposer logic, and then re-enables interposition before returning—ensuring that subsequent system calls made by the application are intercepted as expected. To prevent recursive triggering of `SUD` during the return from the signal handler (via the `rt_sigreturn` system call [25]), the return `syscall`/`sysenter` instruction is usually placed within the allowlisted address range that bypasses interposition. Recent works [77, 157], however, have shown that interposer logic can also be performed entirely outside the signal handler by modifying the signal context directly, eliminating the need to exclude specific `syscall`/`sysenter` instructions from interposition.

[1]Throughout this paper, "System Call Interposition Pitfalls" and "pitfalls" are used interchangeably.
[2]A reference to the legendary perfume in Tom Robbins' novel *Jitterbug Perfume*, which is pursued for centuries and believed to possess rare and transformative qualities.

Due to its exhaustiveness and expressiveness, SUD has been adopted in a variety of contexts, including OS compatibility layers [27, 38] and security mechanisms [156]. However, despite these advantages, **SUD *still incurs substantial overhead in user-space applications due to signal delivery and handling***—especially for system call-heavy workloads typical of datacenter environments (e.g., high-performance servers and databases) [77, 157]. Even though SUD outperforms ptrace by avoiding expensive context switches, its signal-based interposition mechanism still introduces a significant performance bottleneck in these scenarios.

2.2 System Call Interposition via Binary Rewriting

To avoid the performance costs associated with kernel involvement—such as additional context/mode switches—recent approaches have explored user-space system call interposition through binary rewriting [31, 45, 76, 114, 115, 130]. Specifically on x86-64, these techniques work by rewriting syscall/sysenter instructions that trigger system calls to jmp/call instructions that redirect execution to the interposer code, allowing system call interposition entirely within user space. However, this is widely regarded as an open and challenging problem [44, 45, 109], particularly on architectures with variable instruction lengths like x86-64, where precise disassembly and rewriting are both difficult and error-prone. ***Among recent efforts,* zpoline *[157] and* lazypoline *[77] take important steps toward addressing long-standing challenges in binary rewriting.*** In the following paragraphs, we take a closer look at these two systems.

2.2.1 A Closer Look at zpoline. Binary rewriting techniques have traditionally faced a number of challenges, often resulting in incorrect transformations or unpredictable execution behavior. These difficulties are especially pronounced on architectures with variable-length instructions, such as x86-64 [31, 44, 45, 77, 109, 157]. For instance, syscall and sysenter instructions are only two bytes long, whereas the jmp and call instructions typically used for redirection to arbitrary addresses are significantly larger. This mismatch in instruction size forces rewriting tools to make optimistic assumptions about code and memory layout, often requiring them to relocate and transform multiple instructions [45, 76, 114, 115].

Yasukata et al. proposed zpoline, a binary rewriting approach for x86-64 that specifically addresses the instruction size mismatch between two-byte syscall/sysenter instructions and the longer redirection instructions [157]. zpoline replaces each syscall (0x0f 0x05 opcode) and sysenter (0x0f 0x34 opcode) instruction with a two-byte callq *%rax instruction (0xff 0xd0 opcode). This transfers control to a virtual address between 0 and N (typically $N < 500$), leveraging the fact that x86-64 applications store the system call number in rax before invoking a system call.

zpoline constructs a trampoline region starting at virtual address 0, beginning with a nop sled that leads into a jump to the interposer's code. The interposer retrieves the system call number and arguments from the registers, and obtains the address of the instruction following the replaced syscall/sysenter instruction-from the stack. As a result, zpoline naturally enables per-system call and argument-specific logic—a fundamental requirement for general-purpose system call interposition.

2.2.2 A Closer Look at lazypoline. Despite its advantages, zpoline ***still*** depends on precise static binary disassembly [10] and only operates on code that is present at load time, including the main executable and any initially loaded shared libraries. Such disassembly is inherently challenging, particularly on architectures with variable-length instructions such as x86-64, where the bytes representing a syscall/sysenter instruction may be embedded within a larger, unrelated instruction encoding, or may be misidentified due to alignment issues [44, 45, 109]. In addition, zpoline only identifies, and consequently rewrites, syscall/sysenter instructions in this initial set of code, failing to interpose code that is generated or loaded later at runtime—such as dynamically generated code or code loaded via mechanisms like dlopen and dlmopen [4]—which is common in various use cases [32, 33, 35, 36, 154, 160]. To address these limitations, Jacobs et al. [77] proposed lazypoline, a system call interposer that combines zpoline-style rewriting with SUD. Unlike zpoline, lazypoline does not rely on static binary disassembly. Instead, it uses SUD to interpose the first time a syscall/sysenter instruction is executed and subsequently rewrites that instruction in a zpoline-like fashion. This design sidesteps the challenges of imprecise disassembly and supports interposition of code that is dynamically generated or loaded later at runtime.

3 Threat Model

We assume an application running under an in-process system call interposer such as zpoline or lazypoline. We make no assumptions about the application itself. We consider an adversary whose goal is to exploit vulnerabilities in the target program to subvert or abuse the interposer. Because the interposer shares the application's address space, attackers can do so by manipulating its *internal* state (e.g., the selector) or its *external* state (e.g., environment variables). Following prior work [77, 157], we assume that an orthogonal isolation mechanism protects the interposer's internal state. For example, Protection Keys for Userspace (PKU) [49, 50, 69, 70, 83, 92, 111–113, 124, 127, 135, 144, 156] can be used to enforce such protection. Attacks on the interposer's internal state are therefore out of scope. We instead focus on adversaries who manipulate the interposer's external state. Importantly, all of the System Call Interposition Pitfalls remain relevant regardless of the isolation mechanism protecting the internal state, as they explicitly target the external environment itself. Mitigations like software diversity [88] and CFI [42, 53] raise the bar against manipulation of the interposer's external state, but we do not rely on such defenses.

4 System Call Interposition Pitfalls

Both zpoline and lazypoline claim to provide flexible system call interposition [77, 157]. We conduct an in-depth analysis of both solutions and identify several design and implementation flaws. Although a few of these shortcomings have been previously acknowledged by their authors, we also uncover a range of previously undocumented issues. We describe these ***pitfalls*** (P1–P5) in detail below. To support our analysis, we developed Proof-of-Concept (PoC) programs and reference real-world use cases affected by these issues. ***While other binary rewriting techniques are also susceptible to these pitfalls, our examination centers on* lazypoline**

and* `zpoline`*, as they overcome several limitations of alternative binary rewriting approaches [45, 157].

4.1 P1—Interposition Bypass

Both `zpoline` and `lazypoline` rely on `LD_PRELOAD` [13] to inject their fast interposition libraries into target processes. This simple yet powerful technique allows to interpose transparently—without kernel patches or special compiler support—by instructing the dynamic linker/loader to load specified libraries before others. Although `LD_PRELOAD` is most often used to override existing symbols in the standard library, it can also introduce entirely new functionality even when no symbol collisions occur. As a result, it provides a convenient mechanism for bootstrapping user-space system call interposition at process startup. However, the mechanism is fragile: `LD_PRELOAD` can be cleared or modified before launching a new process, preventing the interposition library from being loaded and thereby bypassing system call interposition. Attackers can exploit this behavior in several ways, such as by calling `unsetenv("LD_PRELOAD")` [34] or by invoking `execve` with a `NULL` or sanitized environment [6, 7]. As a result, the interposer is silently disabled in the new process. This behavior is illustrated in Listing 1.

```c
pid_t pid = fork();
if (pid == 0) {
  char *args[] = {"/bin/ls", NULL};
  char *env[]  = {NULL};

  // Empty environment: LD_PRELOAD
  // not inherited from parent
  execve("/bin/ls", args, env);
  perror("execve failed");
  exit(EXIT_FAILURE);
} else {
  wait(0);
  printf("Child completed.\n");
}
```

Listing 1: `execve` invoked with a `NULL` environment clears all environment variables, including `LD_PRELOAD`, thereby preventing the interposition library from being loaded.

```c
#define PR_SET_SYSCALL_USER_DISPATCH    59
#define PR_SYS_DISPATCH_OFF             0
#define PR_SYS_DISPATCH_ON              1

// Disable SUD-based system call interposition
syscall(__NR_prctl, PR_SET_SYSCALL_USER_DISPATCH,
PR_SYS_DISPATCH_OFF, 0, 0, 0, 0);
```

Listing 2: Disabling `SUD`-based system call interposition.

Crucially, these patterns also occur in benign software. For instance, launching a process with an empty environment disables all interposition mechanisms that rely on environment variables. We first encountered this in our own test suite: a utility designed to verify interposition functionality inadvertently disabled our interposing library by invoking `execve` with an empty environment. This demonstrates that even non-malicious or debugging-oriented code can unintentionally bypass `LD_PRELOAD`-based mechanisms. ***We refer to this class of interposition bypass as* `P1a`*. Notably, this behavior affects not only system call interposition but any mechanism that specifically depends on* `LD_PRELOAD`.** As a result, any mechanism relying on `LD_PRELOAD` can be disabled using the same technique—unless proper safeguards are in place (see Section 5.2). ***Additionally,* `lazypoline` *enables another avenue for bypass:*** even if `LD_PRELOAD` successfully injects the interposing library, system call interposition can still be disabled by explicitly deactivating `SUD` via the `prctl` system call. ***We refer to this bypass technique as* `P1b`**, and illustrate it in Listing 2.

> ***`zpoline`'s and `lazypoline`'s system call interposition can be bypassed either intentionally (e.g., by an attacker) or unintentionally (e.g., due to common application behavior).***

4.2 P2—System Call Overlook

Both `zpoline` and `lazypoline` claim to provide exhaustive system call interposition—a claim that does not hold in practice. First, as shown in [77], `zpoline` fails to rewrite `syscall`/`sysenter` instructions generated after its initial disassembly and rewriting routine. Additionally, it may overlook `syscall`/`sysenter` instructions that are not identified due to well-known limitations in binary disassembly [44, 45, 109]. ***We refer to system call overlooks caused by these disassembly limitations as* `P2a`.** Moreover, both interposers fail to intercept any system calls issued before their interposition library loads. ***This limitation affects not only* `zpoline` *and* `lazypoline` *but any mechanism relying solely on library injection—unless additional measures are employed (see Section 5.2).*** In addition, both `lazypoline` and `zpoline` fail to intercept `vdso`-based calls [37], which execute entirely in user space without a traditional `syscall`/`sysenter` instruction. ***We refer to these system call overlooks—those occurring prior to or during library loading and those while using the* `vdso` *interface—as* `P2b`.**

While these limitations may not appear critical at first glance, they introduce blind spots that undermine the exhaustiveness of interposition. A survey of prior literature reveals several scenarios that typically necessitate interposition of ***all*** system calls—including those issued before and during library loading, as well as those made via `vdso`. These scenarios span domains such as reliability [76, 115] and security [80, 123, 137–139, 141, 143, 153]. For example, `Bunshin` [153] employs an N-variant execution engine that distributes runtime security checks across multiple program variants, ensuring checks never conflict while minimizing performance penalties through parallel execution. To guarantee consistency across variants, `Bunshin` requires exhaustive system call interposition—capturing calls issued both before and during library loading, as well as `vdso`-based calls. Traditionally, N-variant execution engines rely on `ptrace` for this purpose [51, 75, 100, 138, 141], incurring prohibitively high overhead. `Bunshin` instead adopts OS-level modifications to achieve exhaustive interposition without `ptrace`'s performance drawbacks. Moreover, the dynamic linker/loader has been identified as a frequent target for attackers [66, 106], reinforcing the need for interposition mechanisms that operate reliably from the very start of process execution—especially in security-critical contexts such as sandboxing [72, 135, 144, 156].

> ***Neither `zpoline` nor `lazypoline` reliably interpose all system calls, creating critical blind spots. This limitation undermines use cases that demand exhaustive interposition.***

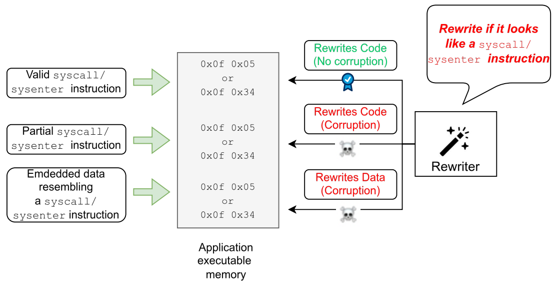

Valid syscall/ sysenter instruction
Partial syscall/ sysenter instruction
Emdedded data resembling a syscall/ sysenter instruction
0x0f 0x05 or 0x0f 0x34
0x0f 0x05 or 0x0f 0x34
0x0f 0x05 or 0x0f 0x34
Application executable memory
Rewrites Code (No corruption)
Rewrites Code (Corruption)
Rewrites Data (Corruption)
Rewrite if it looks like a syscall/ sysenter instruction
Rewriter

Figure 1: `zpoline` and `lazypoline` may misidentify embedded data or partial instructions containing `syscall`/`sysenter` opcodes as legitimate `syscall`/`sysenter` instructions.

4.3 P3—Instruction Misidentification

In addition to overlooking system calls from specific `syscall`/`sysenter` instructions (see Section 4.2), we also found that both interposers are prone to misidentifying `syscall`/`sysenter` instructions—either by confusing them with other instructions or by mistaking embedded data for `syscall`/`sysenter` instructions. This can result in unintended and potentially harmful rewrites of memory regions that should remain untouched.

`zpoline` inherits the well-known limitations of static binary disassembly [44, 45, 109]. It may incorrectly rewrite ***partial instructions, i.e., byte streams where `syscall`/`sysenter` opcodes appear inside other instructions***, or data. On architectures like x86, where instructions can begin at any byte boundary, such subsequences may be executed as genuine `syscall`/`sysenter`. Likewise, `zpoline` may misidentify data as instructions. Prior work [99, 110] shows that embedding data in code pages is widespread (e.g., jump tables), which amplifies the risk of misidentification and corruption, threatening security, correctness, and stability of the target application. ***We refer to these misidentifications as* `P3a`.**

`lazypoline` improves upon this by avoiding reliance on imprecise static disassembly. Instead, it uses SUD to dynamically intercept and rewrite `syscall`/`sysenter` instructions at runtime. However, we found that persistent attackers can still exploit this rewriting mechanism. Because data embedded within executable code pages [99, 110] may coincidentally match the byte pattern of a `syscall`/`sysenter` instruction, an attacker who hijacks control flow can redirect execution to such data. The CPU then treats it as a valid `syscall`/`sysenter` instruction, causing `lazypoline`'s SUD-based handler to intercept and rewrite it—corrupting legitimate application data. Likewise, an attacker can redirect control flow to partial instructions with `syscall`/`sysenter` instruction opcodes (partial `syscall`/`sysenter` instructions), again resulting in unintended code rewriting. ***We refer to these attack-induced misidentifications as* `P3b`**.

Figure 1 illustrates an application containing valid `syscall`/`sysenter` instructions, along with partial instructions containing `syscall`/`sysenter` instruction opcodes (i.e., partial `syscall`/`sysenter` instructions) and embedded data that resembles `syscall`/`sysenter` instructions. Unfortunately, both interposers may misidentify the latter two, which can result in corruption of code and data.

Both `zpoline` and `lazypoline` are susceptible to `syscall`/`sysenter` instruction misidentification, which can lead to code or data corruption—due to disassembly limitations (`P3a`) and attacker-controlled control-flow redirection (`P3b`), respectively.

4.4 P4—NULL Access Termination Pitfalls

In typical Linux applications, the page at virtual address 0 is unmapped. Any access to this region—what the `zpoline` authors call a NULL memory access [157]—triggers a segmentation fault, terminating the process. Both `zpoline` and `lazypoline`, however, repurpose this page by mapping a trampoline at address 0, so reads, writes, or instruction fetches no longer reliably fault. To compensate, both interposers mark the trampoline page as eXecute-Only Memory (XOM) [58, 59, 87, 99, 104, 144] using Protection Keys for Userspace (PKU) [1], preserving the fault-on-NULL-read/write behavior—but not execution. **`lazypoline` *implements no guard against unintended code fetches into the trampoline, a shortcoming we call* `P4a`**. Because of `P4a`, bugs that would ordinarily raise a segmentation fault now divert control into the trampoline, turning simple NULL-code-pointer errors into inscrutable debugging nightmares. Furthermore, many kernel and low-level mechanisms assume page 0 is off-limits—for example, they rely on NULL faults to detect or halt exploits—so remapping it without proper runtime checks risks silently undermining these critical mechanisms.

In contrast, `zpoline` performs a runtime check at the interposer's entry point to verify whether the call originated from a known, rewritten system call site, terminating the process if the check fails. This mechanism uses a bitmap that spans the entire virtual address space, allowing fast validation through bitwise operations. Although physical memory is only allocated for portions of the bitmap that are actually used, the reserved virtual memory can still introduce non-negligible overhead. This memory overhead becomes more pronounced in multi-process settings, where each process maintains its own bitmap instance. In particular, it can pose challenges for low-end devices or scenarios where system call interposition is applied broadly across many applications. ***We refer to this problem as* `P4b`**.

`lazypoline` performs no checks on unintended execution within the memory page starting at virtual address 0, while `zpoline` introduces fast runtime checks—at the cost of added memory overhead.

4.5 P5—Runtime Rewriting Pitfalls

`lazypoline` performs code rewriting on the fly, replacing two-byte `syscall`/`sysenter` instructions with two-byte `callq *%rax` instructions. To prevent race conditions, it uses synchronization primitives to ensure that no two threads concurrently rewrite the same instruction. However, upon analyzing its implementation, we identified several serious flaws in the rewriting mechanism—primarily related to inter-thread concurrency—despite the authors' efforts to mitigate such issues.

First, the two-byte replacement is not guaranteed to be written atomically, potentially leading to the execution of partially rewritten instructions, as demonstrated in previous work [73, 74]. Second, the rewriting process does not ensure proper instruction visibility or coherence across cores: the I-cache is not explicitly flushed, and no instruction stream serialization is enforced (e.g., via `mfence`, `cpuid`, or similar barriers). As a result, modified instructions may not become visible to the CPU pipeline in a timely or consistent manner—an essential requirement for correctness in self-modifying code. Finally, memory access permissions of the pages containing `syscall/sysenter` instructions targeted for modification are neither properly saved before rewriting nor reliably restored afterward—restoration instead relies on error-prone assumptions, exposing potential risks. For instance, the mechanism does not account for XOM [87, 99, 104, 144].

In contrast, `zpoline` avoids all of the above issues by performing binary rewriting once at load time—before any concurrency challenges arise. It also properly saves existing page permissions before rewriting and restores them afterward, preserving memory access permissions. However, this approach ***comes at the cost of missing system calls invoked by any `syscall/sysenter` instructions generated after this single rewriting step (see Section 4.2)***.

On-the-fly binary rewriting is fundamentally challenging in modern multi-threaded and multi-core systems. These challenges illustrate the complexity and fragility of runtime rewriting approaches such as `lazypoline`.

5 K23: Making System Call Interposition Resilient to Pitfalls

The identified pitfalls challenge several claims made by state-of-the-art solutions [77, 157]. For instance, as demonstrated in Section 4, both `zpoline` and `lazypoline` can corrupt code and data, and fail to reliably interpose all system calls. To address these issues, we introduce K23, a new interposer that is resilient to these pitfalls while maintaining high efficiency. K23 employs a two-phase strategy—comprising an ***offline phase*** and an ***online phase***—and combines two Linux interfaces, `ptrace` and SUD, with `zpoline`-like binary rewriting. Specifically, K23 includes three interposition components as shown in Table 1. At a high level, the offline phase executes the application with representative inputs to identify and log legitimate `syscall/sysenter` instructions. During the online phase, K23 enforces exhaustive system call interposition and selectively accelerates handling of system calls invoked via instructions identified during the offline phase. ***By restricting binary rewriting to pre-validated sites, K23 matches the performance of state-of-the-art schemes***[3] ***while sidestepping those pitfalls.*** In the following sections, we detail each phase of K23 and its corresponding interposition components, demonstrating how they address the identified pitfalls.

5.1 A Closer Look at K23's Offline Phase

During K23's offline phase, we run the target program in a controlled environment with benign inputs. The main steps are shown in Figure 2 (steps ①–④). When a system call is invoked ①, the kernel traps it and redirects it to `libLogger` ②. `libLogger` first disables SUD-based interposition via the `selector` (see Section 2.1) to avoid recursive traps. It then logs the `syscall/sysenter` instruction that triggered the call ③. Finally, `libLogger` invokes the original system call and returns its result, re-enables SUD-based interposition, and returns control to the application ④. This sequence continues until the program terminates. To improve coverage, we can repeat the process with different inputs, generating additional logs.

[3]Our evaluation verifies our claims (see Section 6.2).

Interposition Component	When	Where	How
`libLogger`	Offline Phase	In-Process	SUD
`ptracer`	Online Phase	Cross-Process	`ptrace`
`libK23`	Online Phase	In-Process	SUD & Binary Rewriting

Table 1: K23's interposition components, showing *when* they run, *where* they operate, and *how* they achieve system call interposition.

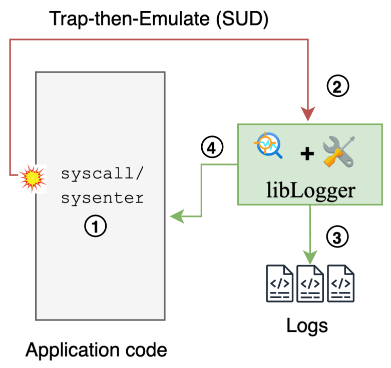
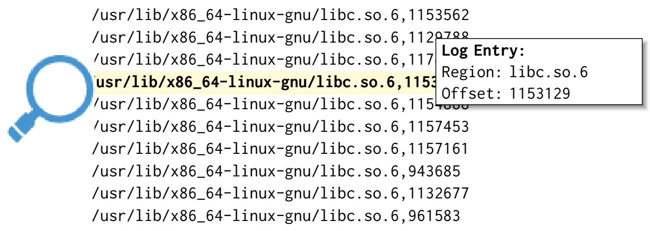


Figure 2: Main steps of K23's offline phase. The kernel traps each system call and forwards it to `libLogger`, which logs the exact triggering `syscall/sysenter` instruction. `libLogger` then invokes the original system call and returns its result, and then returns control to the application.

Any exhaustive system call interposition mechanism may be used during the offline phase. Since performance is not a concern, we use `LD_PRELOAD` to inject an SUD-based interposition library (alternatives include `ptrace` or `seccomp`). With SUD, the kernel traps each system call and raises a SIGSYS signal in user space, invoking `libLogger`'s preinstalled handler. `libLogger` then extracts the virtual address of the triggering `syscall/sysenter` instruction from the signal context, determines its containing memory region (e.g., `libc.so.6`) and its offset by parsing `/proc/$PID/maps` [20], and records each unique (region, offset) pair. The online phase can later map these logged pairs back to actual virtual addresses, since offsets within a given region remain consistent across runs—even under Address Space Layout Randomization (ASLR) [128]. Furthermore, to avoid issues with code that may not exist during K23's single rewriting step (see Section 5.2), e.g., dynamically generated code, `libLogger` records only instructions from expected



```
/usr/lib/x86_64-linux-gnu/libc.so.6,1153562
/usr/lib/x86_64-linux-gnu/libc.so.6,1129788
/usr/lib/x86_64-linux-gnu/libc.so.6,117
usr/lib/x86_64-linux-gnu/libc.so.6,1153
/usr/lib/x86_64-linux-gnu/libc.so.6,1154
/usr/lib/x86_64-linux-gnu/libc.so.6,1157453
/usr/lib/x86_64-linux-gnu/libc.so.6,1157161
/usr/lib/x86_64-linux-gnu/libc.so.6,943685
/usr/lib/x86_64-linux-gnu/libc.so.6,1132677
/usr/lib/x86_64-linux-gnu/libc.so.6,961583
```

```
Log Entry:
Region: libc.so.6
Offset: 1153129
```

Figure 3: Log file generated for `ls`. Each log entry records the memory region and the relative offset within that region of the `syscall`/`sysenter` instruction that triggered a system call.

Application	#Instructions
`pwd`	7
`touch`	9
`ls`	10
`cat`	11
`clear`	13
`sqlite`	20
`nginx`	43
`lighttpd`	44
`redis`	92

Table 2: Number of unique `syscall`/`sysenter` instructions logged during the offline phase for five coreutils and four real-world applications.

executable and non-writable regions, e.g., `libc.so.6` and the application binary. An example log generated for `ls` coreutil [120] is shown in Figure 3.

Because K23’s offline phase runs in a controlled environment, ***we can ensure that all log entries correspond to legitimate*** **`syscall`/`sysenter`** ***instructions***. That said, `libLogger` is simply an injected SUD-based interposition library. Consequently, it cannot log instructions executed during or before the program’s library loading (see Section 4.2). Moreover, `libLogger` cannot interpose `vdso`-based system calls. Importantly, the goal of this phase is not to capture every possible instruction but rather those most frequently used. As we show in Section 5.2, the online phase is responsible for ensuring the ***reliable interposition of all system calls***.

We executed K23’s offline phase on five coreutils [120]—`pwd`, `touch`, `ls`, `cat`, and `clear`—as well as on four real-world applications: `nginx` (branch `stable-1.26`) [16], `lighttpd` (tag `lighttpd-1.4.76`) [14], `sqlite` (tag `version-3.50.4`) [26], and `redis` (branch `8.0`) [22]. Specifically, for the real-world applications, we adopted representative workloads proposed by their developers or from prior work [77, 135, 157]. The results are summarized in Table 2. For the coreutils, we observed only a small number of unique `syscall`/`sysenter` instructions—ranging from 7 for `pwd` to 13 for `clear`—which is expected given their simplicity and short execution time. For the real-world applications, we observed between 20 and 92 instructions. This too is expected, as server and database applications typically run tight loops that repeatedly invoking the same code paths and system calls. ***Overall, our experiments demonstrate that a small number of*** **`syscall`/`sysenter`** ***instructions are responsible for triggering the vast majority of system calls***.

5.2 A Closer Look at K23’s Online Phase

K23’s online phase leverages the logs generated during its offline phase (see Section 5.1), along with Linux interfaces and binary rewriting, to enable flexible system call interposition. An overview of the main steps involved in this phase is illustrated in Figure 4.

During startup, we must interpose every system call—including those issued before or during library loading ①. To do this, we employ `ptracer`, a `ptrace`-based interposer: to our knowledge, `ptrace` is the only mechanism that provides this capability without OS or hardware modifications. **`ptracer`** ***can interpose system calls from the program’s very first instruction.*** Moreover, `ptracer` disables `vdso`, forcing all `vdso`-based calls to use `syscall`/`sysenter` instructions[4]. As a result, K23 can interpose those calls as well. ***Thus,*** **`K23`** ***fully addresses*** **`P2b`*****: it traps system calls invoked before or during library loading into the kernel ①, redirects them to*** **`ptracer`** ***②, executes each call’s handler, and then returns control to the application ③—even for calls that*** **`zpoline`** ***and*** **`lazypoline`** ***miss (see Section 4.2)***.

Although `ptrace` introduces substantial runtime overhead (see Section 2.1), library loading typically accounts for only a small fraction of an application’s execution time[5]. Consequently, we employ `ptracer` only at startup (“`ptracer`: Interposition” in Figure 4). Once K23’s fast interposition library `libK23` is loaded, K23 switches to it for efficient interposition (“`libK23`: Interposition” in Figure 4)—thereby rendering `ptracer`’s impact on overall performance negligible (see Section 6.2). `ptracer` also ensures that `libK23` is injected into the program via `LD_PRELOAD`. To achieve this, `ptracer` intercepts the `execve` system call and checks the `LD_PRELOAD` environment variable. If `LD_PRELOAD` does not already include our library, `ptracer` overwrites it to force injection. Consequently, `ptracer` thwarts any attacker or benign code that attempts to modify environment variables to silently disable injection of our fast interposition library—***effectively addressing*** **`P1a`** ***(see Section 4.1). Such safeguards apply not only to system call interposition but to any mechanism relying on*** **`LD_PRELOAD`**.

Once `libK23` is loaded, its initialization routine notifies `ptracer`, and `ptracer` detaches itself (see Section 5.3). Then, `libK23` installs a trampoline at virtual address 0—similar to `zpoline` and `lazypoline` (see Section 2.2)—and performs a one-time, `zpoline`-style rewrite ④ of each valid `syscall`/`sysenter` instruction identified during K23’s offline phase (see Section 5.1). In addition, `libK23` saves existing page permissions before rewriting and restores them afterward, following a strategy similar to `zpoline` (see Section 4.5). ***By restricting rewriting to these pre-validated sites in a single***

[4]This applies throughout the program’s execution. We also used library interposition and a custom `glibc` to achieve the same result.
[5]An exception is programs with extremely short execution times.

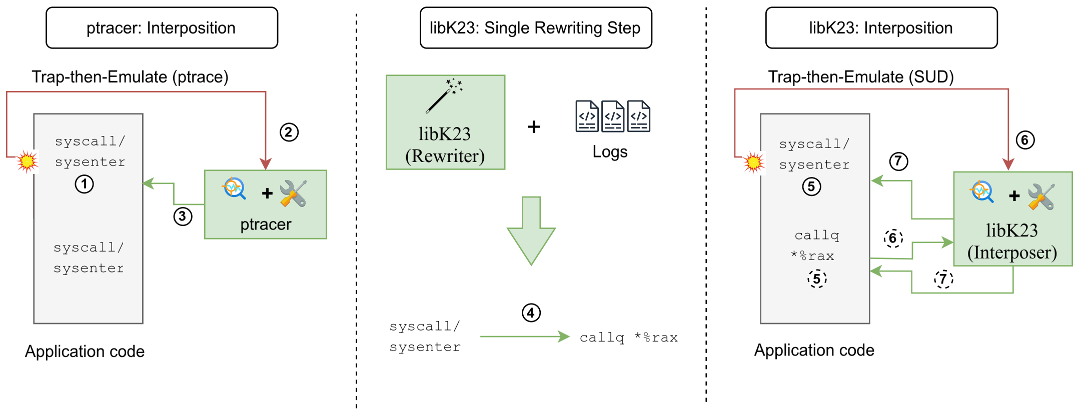


Figure 4: Main steps of K23's online phase. First, ptracer interposes every system call before and during library loading ("ptracer: Interposition") and then detaches once libK23 is loaded. Next, libK23 installs a trampoline (similar to zpoline/lazypoline), performs a single, selective rewrite of the instructions logged in the offline phase ("libK23: Single Rewriting Step"), and configures an SUD-based fallback for any syscall/sysenter sites missed during the offline phase. In the example, one system call is interposed via rewriting and another via the fallback; in both cases, each call is redirected to the same interposition code ("libK23: Interposition").

step and preserving memory access permissions,* K23 *simultaneously addresses* P3a*,* P3b*, and* P5 *(see Sections 4.3 and 4.5).

As noted in Section 5.1, K23's offline phase may still miss some syscall/sysenter instructions. Consequently, immediately after step ④ (not shown in Figure 4), libK23 installs an SUD-based interposer as a fallback—similar to lazypoline [77]. Unlike lazypoline, however, libK23 does not use SUD for discovering or rewriting instructions, since it is prone to attack-induced misidentifications (see Section 4.3). Instead, libK23 employs SUD solely to interpose any syscall/sysenter instructions missed during the offline phase. ***This hybrid catch-all mechanism ensures no system call is overlooked and thereby effectively addresses* P2a *(see Section 4.2).***

Once libK23 completes rewriting and SUD setup, it takes over all system call interposition ("libK23: Interposition" in Figure 4). If the triggering syscall/sysenter instruction was logged during the offline phase, it has already been rewritten to call *%rax ⑤, so invoking it jumps directly into libK23 ⑥. Then libK23 disables SUD-based interposition via the selector (see Section 2.1), avoiding recursive traps. It then handles the system call, and finally re-enables SUD-based interposition just before returning control to the application ⑦. On the other hand, if a syscall/sysenter instruction was ***not*** encountered during the offline phase ⑤, the SUD-based fallback solution redirects execution into libK23 ⑥. After handling the call, libK23 returns control to the application ⑦. During ⑥ and ⑦, libK23 again uses the selector to disable and re-enable SUD-based interposition as needed. Regardless of whether a syscall/sysenter instruction is rewritten, every system call reaches the same interposition code, thereby guaranteeing exhaustiveness. Moreover, libK23 prevents the SUD-based mechanism from being silently disabled by interposing the prctl system call and inspecting its arguments. ***If* libK23 *detects any attempt to disable* SUD*-based interposition (see Listing 2), it aborts immediately—effectively addressing* P1b *(see Section 4.1).***

5.3 Implementation Details

In this section, we describe several key implementation aspects omitted above but critical to K23's functionality.

First, K23 protects its trampoline (at virtual address 0) from NULL read/write accesses using Protection Keys for Userspace (PKU) [1], just as zpoline and lazypoline do (see Section 4.4). Second, we leverage dlmopen [5] to load libK23 into its own namespace—again following the approach used in prior work [77, 157]. This prevents recursive redirection when the interposer invokes shared libraries that the application also uses—libraries which may themselves contain rewritten syscall/sysenter instructions. For more on namespace isolation, see Yasukata et al. [157].

Regarding the issues described in Section 4.4, libK23 performs a runtime check at its entry point—verifying that each call originates from a known, rewritten site—and aborts the process if the check fails. Unlike zpoline, which maintains a large bitmap covering the entire address space, libK23 uses a hash set containing only the instructions logged during the offline phase (see Section 5.1). Because the set is bounded by the offline logs (see Table 2), its memory overhead is negligible. ***Consequently,* K23 *effectively addresses both* P4a *and* P4b.** We employ tsl::robin_set, an alternative high-performance hash set, to store valid instruction addresses and perform these checks [3].

Although not stated earlier, ptracer is a cross-process interposer, whereas libK23 operates in-process (see Table 1). Before detaching, ptracer hands off any accumulated state—such as the number of system calls issued during startup, open file descriptors, and so on—to libK23. To accomplish this, libK23 issues a ***fake system call***—i.e., a non-existent system call number— that the kernel naturally redirects to ptracer. ptracer then transfers its state via OS primitives (e.g., the process_vm_writev system call [19]). Once that completes, libK23 issues a second ***fake system call*** to signal ptracer to detach, after which libK23 actively performs exhaustive system call interposition.

Pitfall		zpoline [157]	lazypoline [77]	K23
P1 - Interposition Bypass (Section 4.1)	P1a	✗	✗	✓
	P1b	✓	✗	✓
P2 - System Call Overlook (Section 4.2)	P2a	✗	✓	✓
	P2b	✗	✗	✓
P3 - Instruction Misidentification (Section 4.4)	P3a	✗	✓	✓
	P3b	✓	✗	✓
P4 - NULL Access Termination Pitfalls (Section 4.4)	P4a	✓	✗	✓
	P4b	✗	✓	✓
P5 - Runtime Rewriting Pitfalls (Section 4.5)	P5	✓	✗	✓

Table 3: Comparison of interposers against System Call Interposition Pitfalls (see Section 4). ✓ indicates the pitfall is either specifically handled or not relevant to the interposer, while ✗ indicates it is not handled.

This mechanism is highly flexible and easily extensible—fake system calls can carry arguments, and `ptracer` can leverage various OS abstractions to access `libK23`'s state (memory, registers, etc.). For security, `ptracer` verifies that both fake system calls originate from `libK23` and not from potentially compromised code (e.g., the dynamic linker/loader). In addition, because K23 relies critically on the integrity of the offline logs, we mark the log directory immutable once the offline phase completes, and keep it so for the program's entire lifetime. ***Therefore, we close the door to new attack surfaces.*** To further harden our solution, `libK23` switches to a dedicated stack upon entry (regardless of whether the triggering instruction was rewritten). This stack-switching technique has proven effective in prior security work [72, 113], and we rely on orthogonal in-process isolation mechanisms to protect that stack and other sensitive interposer's internal state, such as the `selector` and the hash set containing legitimate `syscall`/`sysenter` instructions identified during the offline phase (see Section 3).

Moreover, if the application invokes `execve` again to spawn a new process, `libK23` restarts and re-attaches `ptracer` just before executing the `execve` call. This ensures that the entire online phase can be repeated for the newly spawned process (see Section 5.2). Similarly, although not previously discussed or shown in Figure 2, a simple `ptracer`-like component guarantees that `libLogger` is always injected, even if benign code clears or modifies `LD_PRELOAD`. This component does not record any instructions; its sole purpose is to prevent silent disabling of `libLogger` in newly spawned processes (see Section 4.1). Note that this is purely to maximize our coverage of system calls, not for security enforcement. Finally, to avoid executable stack issues, we adopt practices proposed in previous work while building `libK23` [158].

6 Evaluation

We evaluate K23 along two dimensions: first, we assess whether it successfully addresses the System Call Interposition Pitfalls introduced in Section 4; second, we conduct an extensive performance evaluation. Throughout our evaluation, we also compare K23 against `zpoline` [41] and `lazypoline` [12].

6.1 Pitfall-Oriented Comparison

To evaluate each pitfall, we (i) analyzed the publications and open-source prototypes of `zpoline` and `lazypoline`, and (ii) developed Proof-of-Concept (PoC) programs that trigger these issues. In Table 3, we illustrate the comparison of `zpoline`, `lazypoline`, and K23 against pitfalls.

Specifically, P1a affects both `zpoline` and `lazypoline`, whereas P1b only affects `lazypoline`—it is irrelevant to `zpoline` because `zpoline` does not use SUD (see Sections 2.2 and 4.1). In contrast, K23 effectively addresses both pitfalls (see Section 5.2). Likewise, prior work [77, 157] highlights `zpoline`'s shortcomings with respect to P2a, and neither `zpoline` nor `lazypoline` address P2b (see Section 4.2); again, K23 successfully handles both (see Section 5.2). ***Interestingly, we found that even simple utilities like `ls` [120] issue over 100 system calls during startup before the interposition library is loaded, demonstrating that any mechanism delaying interposition until after library load will inevitably miss these calls (see Section 4.2).***

Both `zpoline` and `lazypoline` rewrite executable memory when it resembles `syscall`/`sysenter` instructions—due to limitations in disassembly (P3a) or hijacked control flow (P3b), respectively (see Section 4.3). By comparison, K23 performs a single selective rewrite of only those `syscall`/`sysenter` instructions pre-validated during the offline phase, eliminating both classes of misidentification (see Sections 5.1 and 5.2).

Meanwhile, `lazypoline` fails to handle P4a, though it is not affected by P4b, as it does not retain a bitmap of valid rewritten instructions. The `zpoline` authors acknowledge P4b and propose alternative, slower strategies that reduce memory overhead [157]. K23 resolves both P4a and P4b by rewriting only `syscall`/`sysenter` instructions identified during the offline phase, and maintaining an optimized hash set of their virtual addresses (see Section 5.3). As Table 2 shows, K23 identified only a handful of instructions (between 7 and 44 in our experiments) during its offline phase—keeping the memory state required for instruction checks extremely low.

Lastly, `lazypoline` does not address P5. By contrast, `zpoline` sidesteps this pitfall by rewriting every detected `syscall`/`sysenter` instruction in one upfront pass (see Section 4.5)—at the cost of missing any system calls invoked by instructions introduced afterward (see Section 4.2). K23 addresses P5 through a single rewriting step and careful design and implementation choices (see Section 5.2).

These limitations constrain the applicability of both `zpoline` and `lazypoline` (see Section 4). K23, on the other hand, addresses all of these pitfalls while matching the efficiency of `zpoline` and `lazypoline` (see Section 6.2), making it a viable, universal system call interposition solution. Importantly, the pitfalls we identify are not exclusive to `zpoline` and `lazypoline`; they may also affect other approaches, even beyond system call interposition. ***For instance, any mechanism relying on `LD_PRELOAD` is similarly affected (see Section 4.1). This broadens the impact of our findings and highlights the generality of our proposed solutions.***

6.2 Performance Evaluation

We conducted our experiments on a DELL Precision 7960 workstation equipped with a 12-core Intel Xeon w5-3425 CPU running at 3.20 GHz (latest firmware) and 64 GB of RAM. To minimize measurement noise, we disabled Turbo Boost, Hyper-Threading, and CPU frequency scaling [48]. The system ran Ubuntu 22.04.5 LTS with the Linux 6.8.0-85 kernel. We evaluated the performance

Variants	Extra Features
`zpoline-default`	–
`zpoline-ultra`	`NULL` Execution Check
`K23-default`	–
`K23-ultra`	`NULL` Execution Check
`K23-ultra+`	`NULL` Execution Check & Stack Switch

Table 4: Variants of `zpoline` and `K23` with their additional features beyond the respective default configurations. The "`NULL` Execution Check" and "Stack Switch" are discussed in Section 4.4 and Section 5.3, respectively. `-default` variants are best suited for high-performance, low-overhead environments, while `-ultra` and `-ultra+` are more suitable for security- and debugging-critical scenarios.

Mechanism	Overhead
`zpoline-default`	1.1267× (±0.042%)
`zpoline-ultra`	1.1576× (±0.083%)
`lazypoline`	1.3801× (±0.040%)
`K23-default`	**1.2788× (±0.056%)**
`K23-ultra`	**1.3919× (±0.072%)**
`K23-ultra+`	**1.3948× (±0.036%)**
`SUD`-no-interposition	1.2269× (±0.045%)
`SUD`	15.3022× (±0.036%)

Table 5: Microbenchmarking overhead relative to native execution (lower is better). Overhead is shown as a multiplicative factor; standard deviation is in parentheses.

of `K23` against `zpoline` and `lazypoline` using both microbenchmarks and macrobenchmarks. Each experiment was executed 10 times; we discarded the maximum and minimum values as outliers, then computed the geometric mean of the overhead relative to a native execution baseline. To capture variability, we also report the standard deviation as a percentage of the mean. For `K23`, we first performed its offline phase by running the relevant microbenchmarks and macrobenchmarks multiple times. For macrobenchmarks, we used widely adopted workloads from prior work [77, 135]. Overall, the offline phase completed within seconds to a few minutes, depending on the specific benchmark.

Additionally, we evaluated different configurations of `zpoline` and `K23`. Specifically, we considered two variants of `zpoline`: `zpoline-default`, which omits `NULL` execution checks, and `zpoline-ultra`, which includes them. For `K23`, we evaluated three variants: `K23-default`, which performs neither `NULL` execution checks nor stack switching; `K23-ultra`, which adds `NULL` execution checks; and `K23-ultra+`, which includes both `NULL` execution checks and stack switching. Aside from these specific differences, all variants behave identically to their respective `default` configuration. We evaluate these variants for two reasons: (i) to isolate and quantify the performance cost of individual features, and (ii) to demonstrate the flexibility of each interposer in adapting to specific use cases, e.g., `-ultra` and `-ultra+` variants are best suited for security- and debugging-critical scenarios. The variants are summarized in Table 4. We also evaluated `SUD` in depth, both when it actively interposes system calls and when interposition is disabled using the `selector`. This allows us to demonstrate: (i) that `SUD` is unsuitable for use cases where interposition performance is critical, and (ii) to better understand the additional overhead introduced by `lazypoline` and `K23` in comparison to `zpoline`.

Since all interposers offer equivalent fine-grained control over applications (e.g., deep argument inspection, fast access to application memory, etc.), we measure overhead using an empty interposition function that simply invokes the original system call and returns its result, following the methodology of prior work [77]. This setup isolates the cost of the interposition mechanism itself, which is the primary focus of our evaluation.

Finally, we note that all benchmarks are designed to reflect high-intensity scenarios—such as system call stress tests and data center–like workloads—to evaluate interposers under extreme operating conditions.

6.2.1 Microbenchmarks. For microbenchmarking, we created a system call stress test using a non-existent system call (system call number 500), which we invoked 100M times. We selected this call because it spends minimal time in the kernel, thereby emphasizing the overhead introduced by each interposition technique. All results are presented in Table 5. As shown in Table 5, `SUD` incurs the highest overhead at 15.3022× relative to native execution. Both `zpoline-default` and `zpoline-ultra` are the most efficient interposers, with overheads of 1.1267× and 1.1567×, respectively. Next is `K23-default` at 1.2788×, followed by `lazypoline` at 1.3801×. The `K23-ultra` (1.3919×) and `K23-ultra+` (1.3948×) variants introduce slightly higher overhead. Nonetheless, all `K23` variants—as well as `lazypoline`—significantly outperform `SUD`.

Interestingly, `K23-default` is faster than `lazypoline`, due to optimizations in `K23`'s trampoline code that save CPU cycles. In particular, `K23` takes advantage of the fact that the kernel clobbers the `rcx` and `r11` registers during system call execution on `x86-64` platforms, allowing it to reuse them without the need for preservation. Furthermore, the runtime overhead of `zpoline-ultra` relative to `zpoline-default` is smaller than that of `K23-ultra` relative to `K23-default`. This discrepancy is attributed to our decision to use a hash set rather than a bitmap, trading slightly higher runtime cost for reduced memory overhead (see Section 4.4).

Following prior work [77], we also measured the overhead of `SUD` when initialized but with interposition disabled via the `selector` ("`SUD`-no-interposition" in Table 5). These results confirm that the performance degradation observed in both `lazypoline` and all `K23` variants stems primarily from relying on `SUD` as a fallback mechanism, even when it does not actively interpose system calls. Specifically, once `SUD` is initialized, ***all*** system calls follow a slower path upon entering the kernel. Our findings align with those reported in previous work [77, 157].

6.2.2 Macrobenchmarks. Following the microbenchmarks, we evaluated the performance impact of system call interposition techniques on real-world, system call–intensive workloads. Specifically, for macrobenchmarking, we assessed the performance of `zpoline` (in two variants; see Table 4), `lazypoline`, `K23` (in three variants; see Table 4), and `SUD` on four real-world applications:

Application (workload)	Native (req/s)	zpoline-default rel. (%)	zpoline-ultra rel. (%)	lazypoline rel. (%)	K23-default rel. (%)	K23-ultra rel. (%)	K23-ultra+ rel. (%)	SUD rel. (%)
nginx (1 worker, 0 KB)	184762 (±0.65%)	99.05 (±0.40%)	98.40 (±1.09%)	97.85 (±0.65%)	97.94 (±0.68%)	97.29 (±0.59%)	96.70 (±0.62%)	51.29 (±0.45%)
nginx (1 worker, 4 KB)	139709 (±0.42%)	96.73 (±0.53%)	96.14 (±0.39%)	96.04 (±0.39%)	96.24 (±0.35%)	95.89 (±0.84%)	95.76 (±0.39%)	45.95 (±0.99%)
nginx (10 workers, 0 KB)	1214421 (±1.84%)	99.62 (±0.50%)	99.34 (±0.32%)	98.79 (±0.86%)	99.52 (±0.40%)	98.39 (±0.69%)	97.83 (±0.87%)	53.93 (±0.29%)
nginx (10 workers, 4 KB)	830426 (±0.24%)	98.83 (±0.45%)	98.76 (±0.26%)	98.14 (±0.38%)	98.59 (±0.31%)	98.12 (±0.24%)	98.23 (±0.27%)	53.97 (±0.10%)
lighttpd (1 worker, 0 KB)	189729 (±0.59%)	98.76 (±0.65%)	99.48 (±0.88%)	98.23 (±0.73%)	99.15 (±0.64%)	97.89 (±1.46%)	97.50 (±0.69%)	61.25 (±0.13%)
lighttpd (1 worker, 4 KB)	147927 (±0.42%)	99.28 (±0.86%)	98.37 (±0.58%)	97.93 (±0.59%)	98.56 (±0.67%)	98.01 (±0.47%)	97.62 (±0.55%)	61.62 (±0.36%)
lighttpd (10 workers, 0 KB)	1444141 (±0.34%)	98.77 (±0.69%)	98.60 (±0.63%)	98.18 (±0.69%)	98.16 (±0.76%)	98.36 (±0.39%)	97.69 (±0.49%)	59.83 (±0.19%)
lighttpd (10 workers, 4 KB)	976989 (±0.19%)	99.17 (±0.26%)	98.98 (±0.24%)	98.67 (±0.16%)	99.01 (±0.32%)	98.65 (±0.34%)	98.62 (±0.37%)	65.06 (±0.18%)
redis (1 I/O thread)	174613 (±0.64%)	100.00 (±0.21%)	99.93 (±0.21%)	99.98 (±0.50%)	100.21 (±0.31%)	100.17 (±0.60%)	99.90 (±0.46%)	96.15 (±0.47%)
redis (6 I/O threads)	398804 (±0.19%)	99.94 (±0.18%)	99.80 (±0.00%)	99.80 (±0.00%)	99.97 (±0.20%)	99.97 (±0.20%)	99.95 (±0.19%)	35.75 (±0.07%)
sqlite (speedtest1, size 800)	N/A	98.12 (±0.19%)	97.80 (±0.19%)	97.31 (±0.18%)	97.56 (±0.16%)	97.13 (±0.12%)	97.20 (±0.14%)	55.90 (±0.60%)
geomean	N/A	**98.93**	**98.27**	**98.26**	**98.62**	**97.96**	**97.90**	**56.70**

Table 6: Macrobenchmark results for interposers across server and database workloads. Each row reports the native throughput (requests per second) and the throughput relative to native (% of native; native = 100%) for zpoline, lazypoline, K23, and SUD. For sqlite, which is not throughput-oriented, we instead report relative runtime performance versus native (computed as $\frac{\text{native\_benchmark\_completion\_time}}{\text{under\_interposer\_benchmark\_completion\_time}} \times 100$). We evaluated multiple variants of zpoline and K23 (see Table 4). Standard deviations across runs are shown in parentheses. The bottom row reports the geometric mean of the relative metrics for each interposer across all workloads. N/A indicates values that are not applicable or meaningful to calculate.

nginx (branch stable-1.26), lighttpd (tag lighttpd-1.4.76), sqlite (tag version-3.50.4), and redis (branch 8.0).

For nginx and lighttpd, we tested four configurations: (i) a single-worker server serving a static 0 KB file, (ii) a single-worker server serving a static 4 KB file, (iii) a 10-worker server serving a static 0 KB file, and (iv) a 10-worker server serving a static 4 KB file. We benchmarked redis in two configurations: the default single-threaded mode (1 I/O thread) and with 6 I/O threads enabled for request handling. Finally, we tested a single sqlite configuration: a fresh 4 KiB-page sqlite database in WAL mode with synchronous=NORMAL and no auto-checkpointing.

These configurations are designed to place interposers under extreme system call–intensive conditions. For nginx, lighttpd, and redis, we matched the number of benchmarking client threads to the number of server workers or I/O threads. Specifically, we used wrk for nginx and lighttpd, configured with 16 connections per client thread and a 30-second run, and redis-benchmark (distributed with redis) in the 100% GET workload, similar to previous work [77, 157]. Following prior work [77], we run both clients and servers on the same physical machine. This eliminates the variability and overhead of the network hardware/software stack, so that our measurements focus on the overhead of system call interposition. In this setup, the benchmarking client(s) and server worker(s) communicate directly over localhost. For sqlite, which is not a client–server system, the benchmark naturally executes locally on the same machine. We used the single-threaded speedtest1 benchmark provided upstream, with database size parameter -size=800. Throughout all experiments, we ensured that the relevant CPU cores were fully saturated.

Table 6 presents the results of our macrobenchmark evaluation, reporting native throughput (in requests per second) and relative throughput under each interposition technique. For sqlite, which is not throughput-oriented, we instead report relative runtime performance versus native (computed as $\frac{\text{native\_benchmark\_completion\_time}}{\text{under\_interposer\_benchmark\_completion\_time}} \times 100$). The geometric mean (bottom row) shows that macrobenchmark trends closely match the microbenchmark results in Table 5. For comparison, we also include SUD, again confirming its unsuitability for system call–intensive workloads. Overall, our findings show that K23 matches the performance of prior approaches while addressing their limitations. By offering multiple variants, K23 lets developers to select configurations that best meet their requirements (Table 4).

7 Discussion

System calls are the gateways from user space to kernel space [28]. As a result, numerous systems have been built around the system call interface. However, as OSs and hardware continue to evolve, these changes inevitably affect all solutions that rely on system call interposition. For example, prior work on sandboxing [57, 72, 125, 135, 144, 156] has shown how attackers can exploit the OS as a confused deputy—e.g., by using the open system call [17]—to bypass hardware-enforced isolation. Our work is built around the principle ***"Every System Call Counts"*** and aims to provide a flexible foundation for developers building system call interposition-based solutions.

While our solution is general in design, the current prototype targets x86-64 Linux platforms. Below, we outline several extensions that would enable K23 to support alternative environments. First, although our implementation uses PKU to apply XOM, it does not depend on it. K23 is, in principle, compatible with alternative XOM approaches [58, 59, 87]. Second, K23 currently leverages ptrace and SUD, both of which are mature OS abstractions supported across various architectures. In cases where these mechanisms are unavailable, kernel modifications—similar to those proposed in prior work [52, 84, 121, 153]—could be used to support equivalent interposition functionality. Similarly, our rewriting technique is currently specific to x86-64. However, alternative binary disassembly and rewriting techniques proposed in prior work [45, 129] could be adopted. In particular, for architectures with fixed instruction lengths, such as ARM, disassembly-based rewriting is expected to be less challenging than on variable-length architectures like x86-64. Porting K23 to such architectures and evaluating its effectiveness in those environments is an interesting direction for future work. Likewise, supporting non-Linux operating systems that lack essential abstractions like SUD and ptrace could be achieved through kernel or hardware modifications.

Our evaluation focuses on applications, which benefit from widely available and standardized benchmarking suites such as wrk [39] and ab [2]. We used these tools during K23's offline phase to generate instruction logs. As shown in Section 6.2.2, these logs let us optimize the handling of most system calls invoked under realistic workloads. However, not all applications have well-structured or comprehensive benchmark suites. In such cases, a promising future direction is to combine dynamic and static analysis to reliably identify syscall/sysenter instructions during the offline phase, e.g., via fuzzing and binary/source code analysis.

Finally, as discussed in Section 3, and consistent with prior work [77, 157], we do not consider attacks targeting the interposer's internal state. A wealth of lightweight intra-process isolation mechanisms exist with ultra-low runtime overhead. Prior work has shown that such mechanisms can enforce effective isolation [93, 155], including hardware virtualization support [72, 85, 97, 116, 135], Protection Keys for Userspace (PKU) [49, 50, 69, 70, 83, 92, 111–113, 124, 127, 135, 144, 156], underused x86 intermediate privilege levels [89, 90], ARM Memory Domains [133], Intel Total Memory Encryption Multi-Key (TME-MK) [134], Supervisor Mode Access Prevention (SMAP) [91, 149, 150], Intel Control-flow Enforcement Technology (CET) [151, 156], ARM Memory Tagging Extension (MTE) and Pointer Authentication Codes (PAC) [62, 81, 102], ARM Privileged Access Never (PAN) and load/store unprivileged (LSU) [152], as well as custom hardware designs [125, 126]. K23 is compatible with such techniques, and we assume that developers will deploy their preferred solution as needed.

8 Related Work

In this section, we discuss intrusive and function hooking-based system call interposition approaches.

Intrusive approaches rely on hardware and OS modifications [8, 46, 52, 65, 72, 84, 98, 121, 125, 126, 135, 136, 142, 147, 153, 161, 162]. These techniques are highly efficient and are typically suited for use cases where performance is critical. However, they are often error-prone, difficult to maintain, and consequently lack flexibility. Moreover, they increase the Trusted Computing Base (TCB), potentially compromising the security of the entire system.

Function hooking-based approaches interpose wrapper functions that invoke system calls, rather than intercepting the system calls themselves [13, 15, 55, 148, 159]. This incurs minimal performance overhead but fails to interpose system calls issued outside these wrapper functions. Additionally, such techniques limit the interposer's expressiveness, as it can only access system call arguments and results via the wrapper's function parameters and return values. Identifying and mapping all relevant wrapper functions to their corresponding system calls is also non-trivial, particularly in large or complex codebases [45, 77, 157].

The Berkeley Packet Filter (BPF) [101] and its extended version (eBPF) [71] allow custom hooks into kernel code, enabling system call interposition and fine-grained runtime observability. However, writing eBPF programs is more complex than writing user-space code due to eBPF's restricted execution environment and its immature ecosystem compared to user-space tooling—such as debuggers, libraries, and APIs. Moreover, despite the presence of a verifier intended to ensure the safety of eBPF programs, these programs still execute with kernel privileges, which exposes the system to risk [95]. Prior research has shown that attackers have repeatedly bypassed the verifier, exploiting vulnerabilities in the eBPF subsystem to gain arbitrary kernel execution [132]. Notably, eBPF has been used to enhance attackers' capabilities [96], and to extend the reach of existing kernel exploits [79].

9 Conclusion

In this work, we identify several fundamental design and implementation flaws—System Call Interposition Pitfalls—that affect state-of-the-art system call interposers. For example, prior approaches cannot reliably interpose all system calls and may even corrupt code and data. To illustrate their impact, we develop PoC programs that reliably trigger these issues and reference real-world use cases affected by them. Guided by these findings, we design and implement K23, a new plug-and-play interposer for x86-64 that overcomes these pitfalls. Our evaluation demonstrates that K23 delivers performance comparable to state-of-the-art interposers, while fully addressing all their identified shortcomings.

Acknowledgments

We thank our reviewers, and especially our shepherd, Antonio Barbalace, for their valuable comments and suggestions, which substantially improved this paper. We are also grateful to Vasileios Kemerlis, Marios Kogias, Georgios Portokalidis, Anjo Vahldiek-Oberwagner, and the authors of zpoline for many helpful discussions on this and related topics, as well as for their feedback on earlier drafts of this paper. This research was supported by the Netherlands' National Growth Fund through the Dutch 6G flagship project "Future Network Services". This research has received funding from the European Union's Horizon Europe research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 101206668, and under projects SafeHorizon (Grant Agreement #101168562) and RECITALS (Grant Agreement #101168490). This project was also funded by the Cybersecurity Research Program Flanders.

References

[1] 2025. Intel. Intel 64 and IA-32 Architectures Software Developer Manuals.
[2] Last accessed 2025. ab - Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/current/programs/ab.html
[3] Last accessed 2025. A C++ implementation of a fast hash map and hash set using robin hood hashing. https://github.com/Tessil/robin-map
[4] Last accessed 2025. dlclose, dlopen, dlmopen - open and close a shared object. https://man7.org/linux/man-pages/man3/dlopen.3.html
[5] Last accessed 2025. dlmopen. https://man7.org/linux/man-pages/man3/dlmopen.3.htmll
[6] Last accessed 2025. environ, execl, execv, execle, execve, execlp, execvp - execute a file. https://linux.die.net/man/3/execve
[7] Last accessed 2025. execve - execute program. https://man7.org/linux/man-pages/man2/execve.2.html
[8] Last accessed 2025. falco. https://falco.org/
[9] Last accessed 2025. gdb - The GNU Debugger. https://man7.org/linux/man-pages/man1/gdb.1.html
[10] Last accessed 2025. GNU Binutils. https://www.gnu.org/software/binutils/
[11] Last accessed 2025. lazypoline. https://github.com/lazypoline/lazypoline
[12] Last accessed 2025. lazypoline - commit bb098e8ca. https://github.com/lazypoline/lazypoline
[13] Last accessed 2025. ld.so, ld-linux.so - dynamic linker/loader. https://man7.org/linux/man-pages/man8/ld.so.8.html
[14] Last accessed 2025. lighttpd. https://www.lighttpd.net/
[15] Last accessed 2025. ltrace - A library call tracer. https://man7.org/linux/man-pages/man1/ltrace.1.html
[16] Last accessed 2025. nginx. https://nginx.org/
[17] Last accessed 2025. open, openat, creat - open and possibly create a file. https://man7.org/linux/man-pages/man2/open.2.html
[18] Last accessed 2025. prctl - operations on a process or thread. https://man7.org/linux/man-pages/man2/prctl.2.html
[19] Last accessed 2025. process_vm_readv, process_vm_writev - transfer data between process address spaces. https://man7.org/linux/man-pages/man2/process_vm_readv.2.html
[20] Last accessed 2025. /proc/pid/maps - mapped memory regions. https://man7.org/linux/man-pages/man5/proc_pid_maps.5.html
[21] Last accessed 2025. ptrace - process trace. http://man7.org/linux/man-pages/man2/ptrace.2.html
[22] Last accessed 2025. redis. https://redis.io/
[23] Last accessed 2025. rr. https://github.com/rr-debugger/rr
[24] Last accessed 2025. seccomp - operate on Secure Computing state of the process. https://man7.org/linux/man-pages/man2/seccomp.2.html
[25] Last accessed 2025. sigreturn, rt_sigreturn - return from signal handler and cleanup stack frame. https://man7.org/linux/man-pages/man2/sigreturn.2.html
[26] Last accessed 2025. sqlite. https://sqlite.org/
[27] Last accessed 2025. Steam. https://store.steampowered.com/
[28] Last accessed 2025. Steven M. Bellovin's retirement talk. https://www.cs.columbia.edu/~smb/talks/farewell.mp4
[29] Last accessed 2025. strace - trace system calls and signals. https://man7.org/linux/man-pages/man1/strace.1.html
[30] Last accessed 2025. Syscall User Dispatch. https://www.kernel.org/doc/html/latest/admin-guide/syscall-user-dispatch.html
[31] Last accessed 2025. syscall_intercept. https://github.com/pmem/syscall_intercept
[32] Last accessed 2025. Tigress JitDynamic. https://tigress.wtf/jitDynamic.html
[33] Last accessed 2025. Tigress Jitter. https://tigress.wtf/jitter.html
[34] Last accessed 2025. unsetenv — remove an environment variable. https://man7.org/linux/man-pages/man3/unsetenv.3p.html
[35] Last accessed 2025. UPX – the Ultimate Packer for eXecutables. https://upx.github.io/
[36] Last accessed 2025. V8 JavaScript engine. https://v8.dev/
[37] Last accessed 2025. vdso - overview of the virtual ELF dynamic shared object. https://man7.org/linux/man-pages/man7/vdso.7.html
[38] Last accessed 2025. Wine. https://www.winehq.org/
[39] Last accessed 2025. wrk - a HTTP benchmarking tool. https://github.com/wg/wrk
[40] Last accessed 2025. zpoline. https://github.com/yasukata/zpoline
[41] Last accessed 2025. zpoline - commit 56aec8797. https://github.com/yasukata/zpoline
[42] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. 2005. Control-Flow Integrity. In *ACM Conference on Computer and Communications Security (CCS)*.
[43] Bert Abrath, Bart Coppens, and Bjorn De Sutter. 2025. MVX-based mitigation of position-independent code reuse. *Comput. Secur.* 159 (2025), 104655. doi:10.1016/J.COSE.2025.104655
[44] Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. 2016. An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries. In *USENIX Security Symposium*.
[45] Paul-Antoine Arras, Anastasios Andronidis, Luis Pina, Karolis Mituzas, Qianyi Shu, Daniel Grumberg, and Cristian Cadar. 2022. SaBRe: Load-time Selective Binary Rewriting. *International Journal on Software Tools for Technology Transfer* (2022).
[46] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. 2012. Dune: Safe User-level Access to Privileged CPU Features. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.
[47] Emery D. Berger and Benjamin G. Zorn. 2006. DieHard: probabilistic memory safety for unsafe languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.
[48] Dirk Beyer, Stefan Löwe, and Philipp Wendler. 2019. Reliable benchmarking: requirements and solutions. *Int. J. Softw. Tools Technol. Transf.* 21, 1 (Feb. 2019), 1–29. doi:10.1007/s10009-017-0469-y
[49] William Blair, William Robertson, and Manuel Egele. 2022. MPKAlloc: Efficient Heap Meta-data Integrity Through Hardware Memory Protection Keys. In *Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*.
[50] William Blair, William Robertson, and Manuel Egele. 2023. ThreadLock: Native Principal Isolation Through Memory Protection Keys. In *ACM Asia Conference on Computer and Communications Security (ASIA CCS)*.
[51] Danilo Bruschi, Lorenzo Cavallaro, and Andrea Lanzi. 2007. Diversified process replicæ for defeating memory error exploits. In *IEEE Performance, Computing, and Communications Conference (IPCCC)*.
[52] Quinn Burke, Ryan Sheatsley, Yohan Beugin, Eric Pauley, Owen Hines, Michael Swift, and Patrick McDaniel. 2025. Efficient Storage Integrity in Adversarial Settings. *IEEE Symposium on Security and Privacy (S&P)* (2025).
[53] Nathan Burow, Scott A Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. 2017. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys (CSUR)* 50, 1 (2017), 16.
[54] Claudio Canella, Mario Werner, Daniel Gruss, and Michael Schwarz. 2021. Automating seccomp filter generation for linux applications. In *Cloud Computing Security Workshop (CCSW)*.
[55] Buddhika Chamith, Bo Joel Svensson, Luke Dalessandro, and Ryan R. Newton. 2017. Instruction Punning: Lightweight Instrumentation for X86-64. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.
[56] Zijun Cheng, Qiujian Lv, Jinyuan Liang, Yan Wang, Degang Sun, Thomas Pasquier, and Xueyuan Han. 2024. Kairos: Practical intrusion detection and investigation using whole-system provenance. In *IEEE Symposium on Security and Privacy (S&P)*.
[57] Emma Connor, Tyler McDaniel, Jared M. Smith, and Max Schuchard. 2020. PKU Pitfalls: Attacks on PKU-based Memory Isolation Systems. In *USENIX Security Symposium*.
[58] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. 2015. Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *IEEE Symposium on Security and Privacy (S&P)*.
[59] Stephen J. Crane, Stijn Volckaert, Felix Schuster, Christopher Liebchen, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, Bjorn De Sutter, and Michael Franz. 2015. It's a TRaP: Table Randomization and Protection against Function-Reuse Attacks. In *ACM Conference on Computer and Communications Security (CCS)*.
[60] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P. Kemerlis. 2020. sysfilter: Automated System Call Filtering for Commodity Software. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*.
[61] Jeff Dike. 2001. User-Mode Linux. In *Annual Linux Showcase & Conference (ALS)*.
[62] Kha Dinh Duy, Kyuwon Cho, Taehyun Noh, and Hojoon Lee. 2023. Capacity: Cryptographically-Enforced In-Process Capabilities for Modern ARM Architectures. In *ACM Conference on Computer and Communications Security (CCS)*.
[63] Asbat El Khairi, Marco Caselli, Andreas Peter, and Andrea Continella. 2024. REPLICAWATCHER: Training-less Anomaly Detection in Containerized Microservices. In *Symposium on Network and Distributed System Security (NDSS)*.
[64] Alexander J. Gaidis, Vaggelis Atlidakis, and Vasileios P. Kemerlis. 2023. SysXCHG: Refining Privilege with Adaptive System Call Filters. In *ACM Conference on Computer and Communications Security (CCS)*.
[65] Tal Garfinkel. 2003. Traps and pitfalls: Practical problems in system call interposition based security tools. In *Symposium on Network and Distributed System Security (NDSS)*.
[66] Xinyang Ge, Mathias Payer, and Trent Jaeger. 2017. An Evil Copy: How the Loader Betrays You.. In *Symposium on Network and Distributed System Security (NDSS)*.
[67] Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and Michalis Polychronakis. 2020. Confine: Automated system call policy generation for container attack surface reduction. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*.

[68] Seyedhamed Ghavamnia, Tapti Palit, and Michalis Polychronakis. 2022. C2C: Fine-grained configuration-driven system call filtering. In *ACM Conference on Computer and Communications Security (CCS)*.

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

[70] Spyridoula Gravani, Mohammad Hedayati, John Criswell, and Michael L. Scott. 2021. IskiOS: Intra-kernel Isolation and Security using Memory Protection Keys. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*.

[71] Brendan Gregg. Last accessed 2025. Linux Extended BPF (eBPF) Tracing Tools. https://www.brendangregg.com/ebpf.html

[72] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. 2019. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *USENIX Annual Technical Conference*.

[73] Quan Hong, Jiaqi Li, Wen Zhang, and Lidong Zhai. 2024. NanoHook: An Efficient System Call Hooking Technique with One-Byte Invasive. In *International Symposium on Dependable Software Engineering: Theories, Tools, and Applications*. Springer, 363–381.

[74] Quan Hong, Jiaqi Li, Wen Zhang, and Lidong Zhai. 2025. DataHook: An Efficient and Lightweight System Call Hooking Technique without Instruction Modification. *Proc. ACM Softw. Eng.* 2, ISSTA (June 2025), 21 pages. doi:10.1145/3728874

[75] Petr Hosek and Cristian Cadar. 2013. Safe software updates via multi-version execution. In *International Conference on Software Engineering (ICSE)*.

[76] Petr Hosek and Cristian Cadar. 2015. Varan the unbelievable: An efficient n-version execution framework. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

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

[78] Rob Jansen, Jim Newsome, and Ryan Wails. 2022. Co-opting Linux Processes for High-Performance Network Simulation. In *USENIX Annual Technical Conference*.

[79] Di Jin, Vaggelis Atlidakis, and Vasileios P. Kemerlis. 2023. EPF: Evil Packet Filter. In *USENIX Annual Technical Conference*.

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

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

[82] Taesoo Kim and Nickolai Zeldovich. 2013. Practical and effective sandboxing for non-root users. In *USENIX Annual Technical Conference*.

[83] Paul Kirth, Mitchel Dickerson, Stephen Crane, Per Larsen, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. 2022. PKRU-safe: automatically locking down the heap between safe and unsafe languages. In *European Conference on Computer Systems (EuroSys)*.

[84] Koen Koning, Herbert Bos, and Cristiano Giuffrida. 2016. Secure and efficient multi-variant execution using hardware-assisted process virtualization. In *IEEE/IFIP Conference on Dependable Systems and Networks (DSN)*.

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

[86] Hsuan-Chi Kuo, Dan Williams, Ricardo Koller, and Sibin Mohan. 2020. A Linux in Unikernel Clothing. In *European Conference on Computer Systems (EuroSys)*.

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

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

[89] Hojoon Lee, Chihyun Song, and Brent Byunghoon Kang. 2018. Lord of the x86 Rings: A Portable User Mode Privilege Separation Architecture on x86. In *ACM Conference on Computer and Communications Security (CCS)*.

[90] Hojoon Lee, Chihyun Song, and Brent Byunghoon Kang. 2023. Harnessing the x86 Intermediate Rings for Intra-Process Isolation. *IEEE Transactions on Dependable and Secure Computing* 20, 4 (2023), 3251–3268. doi:10.1109/TDSC.2022.3192524

[91] Seongman Lee, Seoye Kim, Chihyun Song, Byeongsu Woo, Eunyeong Ahn, Junsu Lee, Yeongjin Jang, Jinsoo Jang, Hojoon Lee, and Brent Byunghoon Kang. 2024. GENESIS: A Generalizable, Efficient, and Secure Intra-kernel Privilege Separation. In *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing*.

[92] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Alexander Jung, Stefan Teodorescu, Sebastian Rauch, Felipe Huici, Costin Raiciu, and Pierre Olivier. 2022. FlexOS: Towards Flexible OS Isolation. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[93] Hugo Lefeuvre, Nathan Dautenhahn, David Chisnall, and Pierre Olivier. 2025. SoK: Software Compartmentalization. In *IEEE Symposium on Security and Privacy (S&P)*.

[94] Hugo Lefeuvre, Gaulthier Gain, Vlad-Andrei Bădoiu, Daniel Dinca, Vlad-Radu Schiller, Costin Raiciu, Felipe Huici, and Pierre Olivier. 2024. Loupe: Driving the Development of OS Compatibility Layers. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[95] Matan Liber. Last accessed 2025. The good, bad and compromisable aspects of linux ebpf. https://pentera.io/wp-content/uploads/2022/07/penteralabs-the-good-bad-and-compromisable-aspects-of-linux-ebpf.pd

[96] Qirui Liu, Wenbo Shen, Jinmeng Zhou, Zhuoruo Zhang, Jiayi Hu, Shukai Ni, Kangjie Lu, and Rui Chang. 2024. Interp-flow Hijacking: Launching Non-control Data Attack via Hijacking eBPF Interpretation Flow. In *European Symposium on Research in Computer Security (ESORICS)*.

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

[98] Kangjie Lu, Meng Xu, Chengyu Song, Taesoo Kim, and Wenke Lee. 2018. Stopping Memory Disclosures via Diversification and Replicated Execution. *IEEE Transactions on Dependable and Secure Computing (TDSC)* (2018).

[99] Chenke Luo, Jiang Ming, Mengfei Xie, Guojun Peng, and Jianming Fu. 2025. Retrofitting XoM for Stripped Binaries without Embedded Data Relocation. In *Symposium on Network and Distributed System Security (NDSS)*.

[100] Matthew Maurer and David Brumley. 2012. TACHYON: tandem execution for efficient live patch testing. In *USENIX Security Symposium*.

[101] Steven McCanne and Van Jacobson. 1993. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Usenix Winter Conference*.

[102] Derrick Paul McKee, Yianni Giannaris, Carolina Ortega, Howard E Shrobe, Mathias Payer, Hamed Okhravi, and Nathan Burow. 2022. Preventing Kernel Hacks with HAKCs.. In *Symposium on Network and Distributed System Security (NDSS)*.

[103] A. Melvin, G. Jaspher Kathrine, Andrew Jeyabose, and Cenitta David. 2025. A Deep Learning Model Leveraging Time-Series System Call Data to Detect Malware Attacks in Virtual Machines. *International Journal of Computational Intelligence Systems* 18 (03 2025). doi:10.1007/s44196-025-00781-z

[104] Daiping liu Mingwei Zhang, Ravi Sahita. 2018. eXecutable-Only-Memory-Switch (XOM-Switch). In *Black Hat Asia Briefings (Black Hat Asia)*.

[105] Ruslan Nikolaev and Godmar Back. 2013. VirtuOS: An Operating System with Kernel Virtualization. In *ACM Symposium on Operating Systems Principles (SOSP)*.

[106] Dario Nisi, Mariano Graziano, Yanick Fratantonio, and Davide Balzarotti. 2021. Lost in the Loader:The Many Faces of the Windows PE File Format. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*.

[107] Gene Novark and Emery Berger. 2010. DieHarder: Securing the Heap. In *ACM Conference on Computer and Communications Security (CCS)*.

[108] Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. 2019. A Binary-Compatible Unikernel. In *Conference on Virtual Execution Environments (VEE)*.

[109] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. 2021. SoK: All You Ever Wanted to Know About x86/x64 Binary Disassembly But Were Afraid to Ask. In *IEEE Symposium on Security and Privacy (S&P)*.

[110] Chengbin Pang, Tiantai Zhang, Ruotong Yu, Bing Mao, and Jun Xu. 2022. Ground truth for binary disassembly is not easy. In *USENIX Security Symposium*.

[111] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. 2019. Libmpk: software abstraction for intel memory protection keys (intel MPK). In *USENIX Annual Technical Conference*.

[112] Taemin Park, Karel Dhondt, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. 2020. Nojitsu: Locking down javascript engines. In *Symposium on Network and Distributed System Security (NDSS)*.

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

[114] Luís Pina, Anastasios Andronidis, Michael Hicks, and Cristian Cadar. 2019. MVEDSUa: Higher Availability Dynamic Software Updates via Multi-Version Execution. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[115] Luís Pina, Daniel Grumberg, Anastasios Andronidis, and Cristian Cadar. 2017. A DSL approach to reconcile equivalent divergent program executions. In *USENIX Annual Technical Conference*.

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

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

[118] Vidya Lakshmi Rajagopalan, Konstantinos Kleftogiorgos, Enes Göktaş, Jun Xu, and Georgios Portokalidis. 2023. SYSPART: Automated Temporal System Call Filtering for Binaries. In *ACM Conference on Computer and Communications Security (CCS).*

[119] Ali Raza, Thomas Unger, Matthew Boyd, Eric B Munson, Parul Sohal, Ulrich Drepper, Richard Jones, Daniel Bristot De Oliveira, Larry Woodman, Renato Mancuso, Jonathan Appavoo, and Orran Krieger. 2023. Unikernel Linux (UKL). In *European Conference on Computer Systems (EuroSys).*

[120] Branden Robinson, David MacKenzie, and Jim Meyering. Last accessed 2025. *GNU Coreutils: File, Shell and Text Manipulation Utilities.* Free Software Foundation. https://www.gnu.org/software/coreutils/

[121] André Rösti, Stijn Volckaert, Michael Franz, and Alexios Voulimeneas. 2024. I'll Be There for You! Perpetual Availability in the A 8 MVX System. In *Annual Computer Security Applications Conference (ACSAC).* IEEE.

[122] André Rösti, Alexios Voulimeneas, and Michael Franz. 2024. The Astonishing Evolution of Probabilistic Memory Safety: From Basic Heap-Data Attack Detection Toward Fully Survivable Multivariant Execution. *IEEE Security and Privacy* 22, 4 (July 2024), 66–75. doi:10.1109/MSEC.2024.3407648

[123] Babak Salamat, Todd Jackson, Andreas Gal, and Michael Franz. 2009. Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space. In *European Conference on Computer Systems (EuroSys).*

[124] Vasily A. Sartakov, Lluís Vilanova, and Peter Pietzuch. 2021. CubicleOS: A Library OS with Software Componentisation for Practical Isolation. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

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

[126] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. 2020. Donky: Domain Keys – Efficient In-Process Isolation for RISC-V and x86. In *USENIX Security Symposium.*

[127] Leon Schuermann, Jack Toubes, Tyler Potyondy, Pat Pannuto, Mae Milano, and Amit Levy. 2025. Building bridges: safe interactions with foreign languages through Omniglot. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI).*

[128] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. 2004. On the effectiveness of address-space randomization. In *ACM Conference on Computer and Communications Security (CCS).*

[129] Yang Shen, Min Xie, Wenzhe Zhang, and Tao Wu. 2024. ASC-Hook: fast and transparent system call hook for Arm. *arXiv preprint arXiv:2412.05784* (2024).

[130] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert Van Renesse, and Hakim Weatherspoon. 2019. X-containers: Breaking down barriers to improve performance and isolation of cloud-native containers. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

[131] Livio Soares and Michael Stumm. 2010. {FlexSC}: Flexible system call scheduling with {Exception-Less} system calls. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI).*

[132] Hao Sun, Yiru Xu, Jianzhong Liu, Yuheng Shen, Nan Guan, and Yu Jiang. 2024. Finding correctness bugs in ebpf verifier with structured and sanitized program. In *European Conference on Computer Systems (EuroSys).*

[133] Zahra Tarkhani and Anil Madhavapeddy. 2020. μTiles: Efficient Intra-Process Privilege Enforcement of Memory Regions. *arXiv preprint arXiv:2108.03705* (2020).

[134] Martin Unterguggenberger, Lukas Lamster, David Schrammel, Martin Schwarzl, and Stefan Mangard. 2024. TME-Box: Scalable In-Process Isolation through Intel TME-MK Memory Encryption. *arXiv preprint arXiv:2407.10740v2.*

[135] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, Efficient In-process Isolation with Memory Protection Keys. In *USENIX Security Symposium.*

[136] Jonas Vinck, Bert Abrath, Bart Coppens, Alexios Voulimeneas, Bjorn De Sutter, and Stijn Volckaert. 2022. Sharing is Caring: Secure and Efficient Shared Memory Support for MVEEs. In *European Conference on Computer Systems (EuroSys).*

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

[138] Stijn Volckaert, Bart Coppens, and Bjorn De Sutter. 2016. Cloning your gadgets: Complete ROP attack immunity with multi-variant execution. *IEEE Transactions on Dependable and Secure Computing (TDSC)* (2016).

[139] Stijn Volckaert, Bart Coppens, Bjorn De Sutter, Koen De Bosschere, Per Larsen, and Michael Franz. 2017. Taming parallelism in a multi-variant execution environment. In *European Conference on Computer Systems (EuroSys).*

[140] Stijn Volckaert, Bart Coppens, Alexios Voulimeneas, Andrei Homescu, Per Larsen, Bjorn De Sutter, and Michael Franz. 2016. Secure and Efficient Application Monitoring and Replication. In *USENIX Annual Technical Conference.*

[141] Stijn Volckaert, Bjorn De Sutter, Tim De Baets, and Koen De Bosschere. 2012. GHUMVEE: efficient, effective, and flexible replication. In *International Symposium on Foundations and Practice of Security (FPS).*

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

[143] Alexios Voulimeneas, Dokyung Song, Fabian Parzefall, Yeoul Na, Per Larsen, Michael Franz, and Stijn Volckaert. 2020. Distributed heterogeneous N-variant execution. In *Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA).*

[144] Alexios Voulimeneas, Jonas Vinck, Ruben Mechelinck, and Stijn Volckaert. 2022. You shall not (by) pass! practical, secure, and fast PKU-based sandboxing. In *European Conference on Computer Systems (EuroSys).*

[145] Qi Wang, Wajih Ul Hassan, Ding Li, Kangkook Jee, Xiao Yu, Kexuan Zou, Junghwan Rhee, Zhengzhang Chen, Wei Cheng, Carl A Gunter, et al. 2020. You are what you do: Hunting stealthy malware via data provenance analysis.. In *Symposium on Network and Distributed System Security (NDSS).*

[146] Ruihua Wang, Yihao Peng, Yilun Sun, Xuancheng Zhang, Hai Wan, and Xibin Zhao. 2023. Tesec: Accurate server-side attack investigation for web applications. In *IEEE Symposium on Security and Privacy (S&P).*

[147] Xiaoguang Wang, SengMing Yeoh, Robert Lyerly, Pierre Olivier, Sang-Hoon Kim, and Binoy Ravindran. 2020. A Framework for Software Diversification with ISA Heterogeneity. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID).*

[148] Xiaoguang Wang, SengMing Yeoh, Pierre Olivier, and Binoy Ravindran. 2020. Secure and Efficient In-Process Monitor (and Library) Protection with Intel MPK. In *European Workshop on System Security (EuroSec).*

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

[150] Chenggang Wu, Mengyao Xie, Zhe Wang, Yinqian Zhang, Kangjie Lu, Xiaofeng Zhang, Yuanming Lai, Yan Kang, Min Yang, and Tao Li. 2023. Dancing With Wolves: An Intra-Process Isolation Technique With Privileged Hardware. *IEEE Trans. Dependable Secur. Comput.* 20, 3 (May 2023), 1959–1978. doi:10.1109/TDSC.2022.3168089

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

[152] Jiali Xu, Mengyao Xie, Chenggang Wu, Yinqian Zhang, Qijing Li, Xuan Huang, Yuanming Lai, Yan Kang, Wei Wang, Qiang Wei, and Zhe Wang. 2023. PANIC: PAN-assisted Intra-process Memory Isolation on ARM. In *ACM Conference on Computer and Communications Security (CCS).*

[153] Meng Xu, Kangjie Lu, Taesoo Kim, and Wenke Lee. 2017. Bunshin: compositing security mechanisms through diversification. In *USENIX Annual Technical Conference.*

[154] Xiaoyang Xu, Masoud Ghaffarinia, Wenhao Wang, Kevin W Hamlen, and Zhiqiang Lin. 2019. CONFIRM: Evaluating compatibility and relevance of control-flow integrity protections for modern software. In *USENIX Security Symposium.*

[155] Nikita Yadav, Franziska Vollmer, Ahmad-Reza Sadeghi, Georgios Smaragdakis, and Alexios Voulimeneas. 2024. Orbital Shield: Rethinking Satellite Security in the Commercial Off-the-Shelf Era. In *IEEE Security for Space Systems (3S).*

[156] Fangfei Yang, Bumjin Im, Weijie Huang, Kelly Kaoudis, Anjo Vahldiek-Oberwagner, Chia che Tsai, and Nathan Dautenhahn. 2024. Endokernel: A Thread Safe Monitor for Lightweight Subprocess Isolation. In *USENIX Security Symposium.*

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

[158] Hengkai Ye and Hong Hu. 2025. Too Subtle to Notice: Investigating Executable Stack Issues in Linux Systems. In *Symposium on Network and Distributed System Security (NDSS).*

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

[160] Jianyi Zhang, Zhenkui Li, Yudong Liu, Zezheng Sun, and Zhiqiang Wang. 2023. SAFTE: A self-injection based anti-fuzzing technique. *Computers and Electrical Engineering* 111 (2023), 108980.

[161] Qihang Zhou, Wenzhuo Cao, Xiaoqi Jia, Peng Liu, Shengzhi Zhang, Jiayun Chen, Shaowen Xu, and Zhenyu Song. 2025. RContainer: A Secure Container Architecture through Extending ARM CCA Hardware Primitives. In *Symposium on Network and Distributed System Security (NDSS).*

[162] Sebastian Österlund, Koen Koning, Pierre Olivier, Antonio Barbalace, Herbert Bos, and Cristiano Giuffrida. 2019. kMVX: Detecting Kernel Information Leaks with Multi-variant Execution. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*