zpoline: a system call hook mechanism based on binary rewriting

Kenichi Yasukata, Hajime Tazaki, and Pierre-Louis Aublin, *IIJ Research Laboratory;*
Kenta Ishiguro, *Hosei University*

https://www.usenix.org/conference/atc23/presentation/yasukata

This paper is included in the Proceedings of the
2023 USENIX Annual Technical Conference.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-35-9

zpoline: a system call hook mechanism based on binary rewriting

Kenichi Yasukata[1], Hajime Tazaki[1], Pierre-Louis Aublin[1], and Kenta Ishiguro[2]
[1]*IIJ Research Laboratory*
[2]*Hosei University*

Abstract

This paper presents zpoline, a system call hook mechanism for x86-64 CPUs. zpoline employs binary rewriting and offers seven advantages: 1) low hook overhead, 2) exhaustive hooking, 3) it does not overwrite instructions that should not be modified, 4) no kernel change and no additional kernel module are needed, 5) source code of the user-space program is not required, 6) it does not rely on specially-modified standard libraries, and 7) it can be used for system call emulation. None of previous mechanisms achieve them simultaneously.

The main challenge, this work addresses, is that it is hard to replace `syscall`/`sysenter` with `jmp`/`call` for jumping to an arbitrary hook function because `syscall` and `sysenter` are ***two-byte*** instructions, and usually more bytes are required to specify an arbitrary hook function address.

zpoline resolves this issue with a novel binary rewriting strategy and special trampoline code; in a nutshell, it replaces `syscall`/`sysenter` with a two-byte **`callq *%rax`** instruction and instantiates the trampoline code at virtual address 0. We confirmed zpoline is functional on the major UNIX-like systems: Linux, FreeBSD, NetBSD, and DragonFly BSD. Our experiments show that zpoline achieves 28.1~761.0 times lower overhead compared to existing mechanisms which ensure exhaustive hooking without overwriting instructions supposed not to be modified, and Redis and a user-space network stack bonded by zpoline experience only a 5.2% performance reduction compared to the minimum overhead case while the existing mechanisms degrade 72.3~98.8% of performance.

1 Introduction

System calls are the primary interface for user-space programs to communicate with Operating System (OS) kernels. Since user-space programs almost always go through system calls to perform important actions, system call hooks can be the vantage point to trace and change their behavior. Therefore, there are many use cases, such as tracing tools [6, 19], sandboxes [18, 25], OS emulation layers [1, 8], and binary compatibility supports of new OS subsystems [22, 29, 30, 33, 36, 37].

Motivating use case. Past studies demonstrated that user-space OS subsystems [10, 13, 17, 23, 24, 27], backed by kernel-bypass frameworks [15, 34, 38], are highly performant. In principle, system call hooks enable us to *transparently* apply user-space OS subsystems to the legacy software artifacts through the POSIX standard (as demonstrated in § 3.3), and the transparency is an important factor for the applicability of user-space OS subsystems.

Problem and related work. However, in UNIX-like systems on x86-64 CPUs, the representative platforms for server systems, there is no perfect system call hook mechanism.

1. Existing kernel supports (e.g., `ptrace` (§ 3.1.1) and Syscall User Dispatch (SUD) [20] (§ 3.1.3)) and the legacy binary rewriting technique using `int3` signaling (§ 3.1.2) **cause unacceptable performance degradation** to hook-applied user-space programs (§ 3.3).
2. Other binary rewriting mechanisms (e.g., instruction punning [7], E9Patch [9], and the technique applied in X-Containers [36]) (explained in § 2.1) and function call replacement (e.g., LD_PRELOAD (§ 3.1.4)) **cannot exhaustively hook system calls**. Thus, they cannot be used for systems requiring reliability.
3. Another type of binary rewriting technique (e.g., Detours [14]) **overwrites instructions that are supposed not to be modified** (explained in § 2.1).
4. Solutions based on specific changes to the kernel or additional kernel modules such as Dune [3], which are not merged to the mainline, substantially **diminish the portability** of applications relying on them.
5. Approaches requiring recompilation of the source code of a user-space program, typically seen in Unikernel [26] systems [5, 21], are **unusable in many cases** because users often do not have access to the source code.
6. The approach, which links application binaries with a standard library (e.g., libc) specially modified for replacing the invocations of system calls with function calls of specific OS subsystems [22, 29, 30, 33, 37], **narrows down the range of choice for applicable standard**

library implementations, moreover, **cannot hook system calls which are invoked from the outside of standard libraries**.

7. Although BSD Packet Filter (BPF) [28] and its extended version, eBPF, allow users to apply hooks to the kernel-space functions, they **cannot be used for changing and emulating the behavior of system calls without modifying the kernel source code**.

In summary, every existing system call hook mechanism has a significant downside. Due to the lack of an ideal system call hook mechanism, there have been no practical means of transparently applying user-space OS subsystems to existing user-space programs. Consequently, the applicability of user-space OS subsystems has been significantly limited, regardless of their great advantages.

Contributions. To solve this problem, we present a novel system call hook mechanism for x86-64 CPUs named *zpoline* that is free from all drawbacks mentioned above (§ 2). We demonstrate the benefits of zpoline through microbenchmarks (§ 3.2) and experiments transparently applying user-space OS subsystems to user-space programs (§ 3.3).

2 zpoline

zpoline is based on binary rewriting; it replaces `syscall` and `sysenter`, which are ***two-byte*** instructions (`0x0f 0x05` and `0x0f 0x34` in opcode respectively) that trigger a system call, to jump to an arbitrary hook function address.

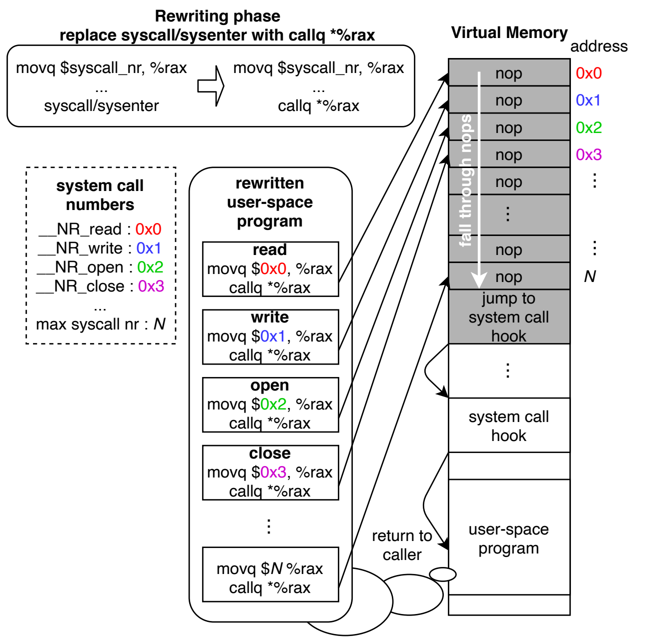
2.1 Challenge and Goal

The challenge of this work is that the two-byte space, originally occupied by a `syscall`/`sysenter` instruction, is too small to locate a `jmp`/`call` instruction along with an arbitrary destination address; typically, two bytes are occupied by the opcode of `jmp`/`call` and eight bytes are necessary for a 64-bit absolute address, or another possibility is one byte for a `jmp`/`call` instruction and four bytes of a 32-bit relative address. Due to this issue, existing binary rewriting techniques give up the replacement in some cases and fail to ensure exhaustive hooking [7, 9, 36], exceed the two-byte space originally occupied by `syscall`/`sysenter` to put the code bigger than two bytes while a jump to the exceeded part causes unexpected behavior [14], or take the `int3` signaling approach (§ 3.1.2) that imposes a significant overhead (§ 3.2). The goal of zpoline is to be free from these drawbacks.

2.2 Design

The overview of zpoline is shown in Figure 1.

System call and calling convention. zpoline employs the calling convention of system calls. In UNIX-like systems on x86-64 CPUs, when a user-space program executes



Rewriting phase
replace syscall/sysenter with callq *%rax
movq $syscall_nr, %rax ... syscall/sysenter
movq $syscall_nr, %rax ... callq *%rax
system call numbers
__NR_read : 0x0
__NR_write : 0x1
__NR_open : 0x2
__NR_close : 0x3
...
max syscall nr : N
rewritten user-space program
read movq $0x0, %rax callq *%rax
write movq $0x1, %rax callq *%rax
open movq $0x2, %rax callq *%rax
close movq $0x3, %rax callq *%rax
movq $N %rax callq *%rax
return to caller
Virtual Memory
address
nop
0x0
0x1
0x2
0x3
N
fall through nops
jump to system call hook
system call hook
user-space program

Figure 1: zpoline overview. The trampoline code is shaded.

`syscall`/`sysenter`, the context is switched into the kernel, then, a pre-configured system call handler is invoked. To request the kernel to execute a particular system call, a user-space program sets a system call number (e.g., 0 is `read`, 1 is `write`, and 2 is `open` in Linux on x86-64 CPUs) to the `rax` register before triggering a system call, and in the kernel, the system call handler executes one of the system calls according to the value of the `rax` register.

Binary rewriting. To hook system calls, zpoline replaces `syscall`/`sysenter` with **`callq *%rax`** which is represented by two bytes `0xff 0xd0` in opcode. Since the instruction sizes of `syscall`/`sysenter` and `callq *%rax` are the same two bytes, *the replacement does not break the neighbor instructions.* What `callq *%rax` does is to push the current instruction pointer (the caller's address) to the stack, and jump to the address stored in the `rax` register. Our insight is that, according to the calling convention, the `rax` register always has a system call number. Therefore, the consequence of `callq *%rax` is the jump to a virtual address between 0 and the maximum system call number which is more or less 500[1].

Trampoline code. To redirect the execution to a user-defined hook function, zpoline instantiates the trampoline code at virtual address 0; in the trampoline code, the virtual address range between 0 and the maximum system call number is filled with the single-byte `nop` instruction (`0x90`), and at the next to the last `nop` instruction, a piece of code to jump to a particular hook function is located.

[1]In Linux 5.15, the maximum system call number of the x86-64 ABI, seen in `unistd_64.h`, is 448.

Execution flow. After the trampoline code instantiation and binary rewriting are completed, the rewritten part (`callq *%rax`) will jump to one of the `nop`s in the trampoline code while pushing the caller's address on the stack. The execution slides down the subsequent `nop`s; after executing the last `nop`, it jumps to the hook function. Here, the hook function will have the same register state as the kernel-space system call handler. The return of the hook function jumps back to the caller address that is pushed on the stack by `callq *%rax`.

Security notes. We note that, like other system call hook mechanisms based on binary rewriting, zpoline itself does not offer security enhancement. On the other hand, if users wish to improve the security of zpoline-applied systems, they can employ existing mechanisms; for instance, seccomp [2] can filter the execution of kernel-space system calls triggered by a zpoline-applied user-space program, and CPU supports such as Memory Protection Keys (MPK) [16] can isolate the implementation of a hook function.

2.3 Implementation

Our current prototype focuses on Linux. The core implementation of zpoline consists of trampoline code instantiation and binary rewriting. We implement these in a shared library called `libzpoline.so` and a special loader named `zpoline_loader`; we assume a user uses either[2]. They perform the setup procedure of zpoline (§ 2.3.1) before the main function of a user-space program starts.

2.3.1 Setup Procedure

The trampoline code setup procedure first allocates memory at virtual address 0 by using the `mmap` system call[3]. Afterward, it fills the allocated memory region with the content described in § 2.2. The binary rewriting procedure initially obtains the memory mapping information from `procfs`. Then, it traverses CPU instructions on the executable memory regions, and replaces `syscall`/`sysenter` with `callq *%rax` (§ 2.2). The memory regions for the trampoline code and the code binary of the user-space program are configured to be writable during this setup phase, and they are restored to be non-writable before the setup procedure exits. After the setup completes, the main function of the user-space program starts as usual, but, all system calls are hooked by zpoline. We note that this implementation does not change the binary files of user-space programs since binary rewriting is done on the code binary loaded onto the memory.

[2]`libzpoline.so` assumes to be loaded through LD_PRELOAD and used when the application binary is dynamically linked. LD_PRELOAD allows `libzpoline.so` to run the setup procedure before the main function of the user-space program starts. `zpoline_loader` is complementary and assumes to be used when the application binary is statically linked and the LD_PRELOAD feature does not work.

[3]In Linux, by default, the memory mapping to virtual address 0 is only allowed for the root user, but it can be permitted for all non-root users by setting 0 to `/proc/sys/vm/mmap_min_addr` (confirmed in Linux 5.15).

2.3.2 Hook Function Development

zpoline users can implement an arbitrary system call hook function as part of `libzpoline.so` or `zpoline_loader`. However, there is an issue that the hook function falls into an infinite loop when it calls a function that originally executes `syscall`/`sysenter` because the replaced code (`callq *%rax`) brings the execution back to the hook function. Users encounter this issue especially when they use `libzpoline.so` because the default dynamic linker/loader automatically associates library calls used in the hook function with the libraries whose `syscall`/`sysenter` instructions are replaced with `callq *%rax`.

Use of `dlmopen`. We avoid this issue by using `dlmopen`, an extended version of `dlopen`. `dlopen` loads a library file onto the memory of a user-space process. On top of this basic feature, `dlmopen` allows users to specify a *namespace* where the library is loaded, and it conducts the association in the same namespace. Thus, `dlmopen` enables us to avoid the automatic undesired association by loading the hook function in a new namespace. To use `dlmopen`, we assume a zpoline user builds the core of the hook implementation as an independent shared library. During the setup phase, `libzpoline.so` loads the library using `dlmopen`, and obtains the pointer to the core implementation of the hook function by using `dlsym`. The hook function, implemented in `libzpoline.so`, calls it through the obtained pointer.

2.3.3 NULL Access Termination

Typically, a memory access to virtual address 0, namely the NULL pointer access, causes a page fault because of the lack of physical memory mapping at virtual address 0, and it results in the termination of the user-space program. The NULL access termination is important for stopping buggy programs. On the other hand, the use of virtual address 0 in zpoline brings about the issue that the NULL access of a user-space program does not cause a fault. To cope with this issue, zpoline employs a set of techniques.

Terminate NULL read and write. To terminate NULL read and write, zpoline configures the trampoline code to be the eXecute-Only Memory (XOM)[4]; a user-space program, that attempted a read/write access to XOM, will be terminated by the kernel because of a fault.

Terminate NULL execution. To trap unintentional NULL execution, zpoline collects the virtual addresses of `syscall`/`sysenter` which are replaced during the setup phase of zpoline (§ 2.3.1), and it checks, at the entry point

[4]In Linux running on a CPU supporting the Memory Protection Keys (MPK) [16] feature, the `mprotect` system call configures XOM when only `PROT_EXEC` is set in the access flag; if MPK is not supported by the CPU, `mprotect` does not configure XOM.

of the hook function, if the caller of the hook function is one of the replaced virtual addresses or not. If it is not, zpoline terminates the user-space program because it is not the jump from the replaced `callq *%rax`, meaning an unintentional jump to NULL. To maintain the replaced virtual addresses while achieving a low-overhead NULL execution check, we use a bitmap that covers the entire 256 TB (48-bit) virtual address range that is typical in x86-64 CPUs. This bitmap allows us to conduct the NULL execution check with a few bit operations, and this cost is evaluated in § 3.2. The bitmap occupies 32 TB of virtual address space, however, its physical memory consumption is substantially smaller because the virtual address pages, whose all bits are clear, do not need to have underlying physical memory pages[5]. We note that if a user prefers to avoid occupying 32 TB of virtual address for the bitmap, we can alternatively use a hash table at the cost of the higher overhead of the NULL execution check.

2.4 Limitations

Here, we discuss the limitations of zpoline.

`syscall`/`sysenter` loaded at runtime. The current prototype of zpoline cannot hook `syscall`/`sysenter` loaded or crafted after the completion of the setup (§ 2.3.1). We can resolve this issue by borrowing the idea of online binary rewriting presented in the X-Containers [36] work that traps an invocation of a system call and rewrites, on the fly, the `syscall`/`sysenter` instruction that triggered the system call.

vDSO (virtual dynamic shared object). Kernels provide user-space programs with several system calls by directly exposing the code for them through vDSO. Like other system call hook mechanisms, zpoline cannot hook vDSO-based system calls by default; however, we can enable zpoline to hook them by disabling vDSO[6].

Unusable virtual address 0. zpoline is not applicable if memory at virtual address 0 is unusable; for instance, the virtual address 0 is already used for other purposes, or the kernel does not allow the mapping at virtual address 0.

Other OSes. We confirmed that zpoline is functional on FreeBSD 13.0, NetBSD 9.2, and DragonFly BSD 6.0[7]. We could not use zpoline on OpenBSD 7.0 because the minimum mappable virtual address is hard-coded as the page size. In Windows, `VirtualAlloc` is conceptually equivalent to `mmap`. On Windows 10, `VirtualAlloc` fails when the specified virtual address is lower than 0x10000, therefore, we could not apply zpoline. But, Windows offers a compatibility layer for Linux called Windows Subsystem for Linux (WSL). We confirmed that zpoline works on WSL2 while it did not on WSL1 whose `mmap` to virtual address 0 returns successfully but actually does not conduct the memory mapping. On macOS, the virtual address 0 of a user-space program is used by a special segment named `__PAGEZERO`, therefore, we could not apply zpoline on macOS.

Other CPU architectures. zpoline is not compatible with CPU architectures which assume the instructions to be aligned by architecture-specific sizes on the memory and consider a jump to an unaligned virtual address as an invalid operation (e.g., ARM); this is because, when zpoline is applied, the execution can jump to an unaligned virtual address between 0 and the maximum system call number (§ 2.2)[8]. However, we believe zpoline is applicable to a large number of servers because x86-64 CPUs are very popular.

[5]In many cases, most of `syscall`/`sysenter` instructions come from libc and ld.so. We found the bitmap uses 22 and 5 physical 4 KB pages to maintain 544 and 50 of `syscall`/`sysenter` in libc and ld.so respectively (glibc-2.35).

[6]Linux disables vDSO when the kernel boot option specifies `vdso=0`.

[7]In FreeBSD and NetBSD, users can use `sysctl` to permit memory mapping at virtual address 0. DragonFly BSD allows it by default.

[8]Besides the issue of the instruction alignment, binary rewriting techniques need to pay attention to architecture-specific factors; for example, on ARM CPUs, the simple replacement from `SVC` to `BL` overwrites/breaks the return address saved in a specific register [31].

3 Evaluation

This section evaluates zpoline through a comparison with existing hook mechanisms (§ 3.1). Particularly, we quantify the hook overhead of zpoline (§ 3.2) and the performance penalty experienced by application programs and user-space OS subsystems bonded by zpoline (§ 3.3).

Experiment setup. For the experiments, we use two machines; each has two 16-core Intel Xeon Gold 6326 CPUs clocked at 2.90 GHz and 128 GB of DRAM. The two machines are directly connected via Mellanox ConnectX-5 100 Gbps NICs. In the experiments in § 3.3, we use one of the two as the *server machine*, and the other as the *client machine*. Both machines run Linux 5.15.

3.1 Comparison

We compare zpoline with `ptrace` (§ 3.1.1), `int3` signaling (§ 3.1.2), SUD (§ 3.1.3), and LD_PRELOAD (§ 3.1.4). Here, we describe the mechanisms and properties of them.

3.1.1 `ptrace`

UNIX(-like) OSes offer the `ptrace` system call that enables a *tracer* process to hook system calls attempted by a *tracee* process. Since `ptrace` is a kernel feature, it can hook system calls exhaustively. However, its hook overhead is enormous due to the context switch between the tracer and tracee; the tracer sleeps while the tracee is running, and the tracee sleeps during the tracer runs its hook function. Therefore, at every system call invocation, the tracee experiences a long latency that includes the wake-up time of the tracer, the execution time of the hook function, and the wake-up time of the tracee. This latency results in *significant performance degradation of the user-space program running on the tracee.*

3.1.2 int3 Signaling

int3 is a one-byte instruction (0xcc) that invokes a software interrupt. On Linux, the kernel handles it and raises SIGTRAP to the user-space process that executed int3. The int3 signaling technique exploits this behavior to hook system calls; it replaces syscall/sysenter with int3 and employs the signal handler for SIGTRAP as the hook function. Since int3 is one byte, it can replace an arbitrary instruction without breaking the neighbor instructions. This technique is traditionally used in debuggers to implement breakpoints. However, *signal handling incurs a large overhead* because it involves context manipulation by the kernel.

3.1.3 Syscall User Dispatch (SUD)

Syscall User Dispatch (SUD) [20] was added in Linux 5.11, and it offers a way to redirect system calls to arbitrary user-space code. For the SUD feature, the kernel implements a hook point at the entry point of system calls. A user-space process can activate SUD via the prctl interface. When SUD is activated, the hook point raises SIGSYS to the user-space process. This mechanism allows a user-space program to leverage the SIGSYS signal handler as the system call hook. However, similarly to the int3 signaling technique, SUD *imposes a significant performance penalty on the user-space program due to the overhead of the signal handling.*

3.1.4 Function Call Replacement by LD_PRELOAD

The dynamic linker/loader (ld.so) offers the LD_PRELOAD feature that allows users to specify shared objects to be loaded before the main part of a program starts, and it can be used for selectively overriding function calls implemented in other shared objects. Users can employ this mechanism to replace the system call wrapper functions, which are typically implemented in standard libraries, with arbitrary function calls. The performance penalty of LD_PRELOAD is very small because the hooks are applied through function pointer replacement.

A function call hook is not a system call hook. However, precisely, the function call replacement for a system call wrapper function is not the hook for a system call; in the first place, the syscall and sysenter instructions are not directly associated with any function calls, and LD_PRELOAD cannot hook a syscall/sysenter instruction which does not have a dedicated and exported wrapper function.

The case where LD_PRELOAD fails to hook. glibc [11] is a representative example where LD_PRELOAD cannot apply system call hooks exhaustively. In many cases, glibc does not use the well-known system call wrapper functions to invoke system calls; instead, glibc directly embeds syscall/sysenter in its internal functions which are marked as invisible from the outside of glibc, and LD_PRELOAD cannot apply hooks to syscall/sysenter instructions wrapped by such internal function calls.

Mechanism	Time [ns]
ptrace	31201
int3 signaling	1342
SUD	1156
zpoline	41
zpoline (no NULL execution check (§ 2.3.3))	40
LD_PRELOAD	6

Table 1: The overhead for hooking a system call.

Potential but impractical approach. Although it is possible for users to apply hooks using LD_PRELOAD by entirely replacing library calls that contain syscall/sysenter instructions, this approach does not scale because users must give up the use of the original library call implementations; in other words, they need to reimplement the equivalent functionalities by themselves, however, it is not realistic to reimplement large part of glibc. Moreover, this reimplementation approach cannot be applied if, unlike glibc, the source code of a shared library file is not available.

Limitation of LD_PRELOAD. In short, *LD_PRELOAD cannot exhaustively hook system calls*, thus, is not an appropriate option to apply user-space OS subsystems to existing user-space programs; for instance, a file descriptor, which is opened by a user-space OS subsystem, will be passed to a kernel-space OS subsystem if a system call is not properly hooked, and it leads to unexpected behavior of the system.

Similarity to binary rewriting techniques. We note that, in our experiments, the cases of other binary rewriting techniques [7, 9, 14, 36] are represented by the LD_PRELOAD case because they share the same characteristics: their performance overhead is very small, however, as described in § 2.1, they cannot hook system calls exhaustively.

3.2 System Call Hook Overhead

We quantify the system call hook overhead by measuring the time to hook getpid, one of the simplest system calls. Our primary interest here is the hook overhead itself; to avoid the overhead of the kernel-crossing system call, we use a hook function that returns a dummy value without actually executing the getpid system call. Table 1 shows the results. First, the overhead of LD_PRELOAD is negligible as expected (§ 3.1.4). The overhead of zpoline is 6.8 times higher than LD_PRELOAD, and this is primarily due to the nops in the trampoline code (§ 2.2). The cost of the NULL execution check (§ 2.3.3) is 1 ns out of 41 ns. zpoline is 761.0, 32.7, and 28.1 times lighter than ptrace, int3 signaling, and SUD respectively. The major overheads of int3 signaling (§ 3.1.2) and SUD (§ 3.1.3) derive from the signal handling for SIGTRAP and SIGSYS. ptrace exhibits the biggest overhead due to the cost of scheduling between the tracer and tracee processes (§ 3.1.1).

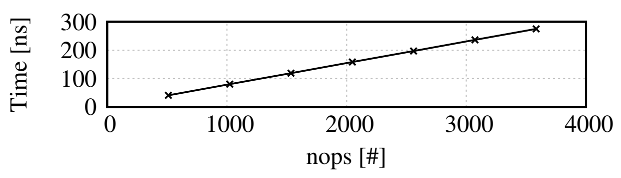


Figure 2: The overhead to hook a system call depending on the number of `nops` at the beginning of the trampoline code.

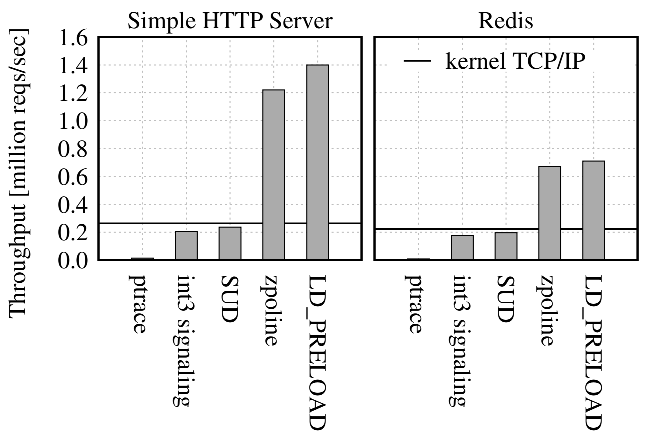


Figure 3: Performance of applications running on lwIP and DPDK with different system call hook mechanisms. For reference, the throughput of the Linux kernel TCP/IP stack is shown by the non-dotted horizontal line.

`nop` overhead. In zpoline, `nops` in the trampoline code increase the hook overhead. The number of `nops` depends on the number of system calls implemented by the kernel (§ 2.2). To see how the overhead grows, we run the same `getpid` test above while changing the number of `nop` instructions in the trampoline code. Figure 2 shows that the overhead linearly increases according to the number of `nops`. However, in the first place, the kernel development communities are very wary to add system calls. Therefore, we believe the `nop` overhead of zpoline will not increase drastically in the future. Moreover, although 3.5 K `nops` are located, the overhead of zpoline is still 113.4, 4.8, and 4.2 times lower than that of `ptrace`, `int3` signaling, and SUD respectively.

3.3 User-space OS Subsystem Performance

This section evaluates how zpoline affects the performance of application programs backed by user-space OS subsystems; we employ zpoline and the existing hook mechanisms described in § 3.1 to *transparently* apply a portable TCP/IP stack, lwIP [10], backed by Data Plane Development Kit (DPDK) [15], to a simple HTTP server and Redis [35]. Normally, kernel-bypassing lwIP achieves higher networking performance than the kernel TCP/IP stack of Linux [4, 32]; for reference, we run the same benchmarks using the kernel TCP/IP stack of Linux and report its performance by non-dotted horizontal lines in Figure 3. We note that the simple HTTP server and Redis are chosen for the experiments because LD_PRELOAD could apply hooks to them, and as explained in § 3.1.4, LD_PRELOAD can fail to hook system calls in other systems.

Simple HTTP server. Commonly, a server program triggers network-relevant system calls more frequently when its application logic gets lighter because it can serve a lot of requests in a short time. To stress the hook mechanisms with lightweight application logic, we made a simple HTTP server that replies a static 64-byte content; we run it on the server machine. As the benchmark client, we run wrk [12] on the client machine; it sends requests through 32 persistent concurrent connections. The results are shown in Figure 3 (left). First, the LD_PRELOAD result represents the minimum overhead case (§ 3.2), and it demonstrates the potential of lwIP on DPDK, which is 5.2 times faster than the Linux kernel TCP/IP stack whose throughput is shown by the non-dotted horizontal line in Figure 3 (left). Comparison with the LD_PRELOAD case sheds light on the overhead of each hook mechanism. The percentages of performance reduction in `ptrace`, `int3` signaling, and SUD compared to LD_PRELOAD are 98.9%, 85.3%, and 83.0% respectively. Contrarily, zpoline causes only 12.7% of performance reduction. These results are explained by the hook overheads shown in Table 1.

Redis. We evaluate how a real-world application performs on zpoline. For benchmarking, we use Redis [35], a widely used key-value store; we run a Redis server process on the server machine. As the benchmark client, we use `redis-benchmark`, which is distributed as part of the Redis source, on the client machine; we run the GET 100% workload so that the Redis server will spend most of its time on networking operations rather than disk operations. Requests are sent over 32 persistent concurrent connections. Figure 3 (right) shows a similar trend to the simple HTTP server experiment, and the overall results reflect the overheads shown in Table 1. Compared to LD_PRELOAD, the throughput results of `ptrace`, `int3` signaling, and SUD are 98.8%, 75.0%, and 72.3% lower respectively. In contrast, zpoline imposes only 5.2% of throughput reduction.

4 Conclusion

This paper has presented zpoline, a system call hook mechanism for x86-64 CPUs, that can exhaustively hook system calls at a low overhead without overwriting instructions that are supposed not to be modified. zpoline is a practical means of transparently applying user-space OS subsystems to existing user-space programs and contributes to the applicability of user-space OS subsystems.

Acknowledgments

We are grateful to anonymous USENIX ATC 2023 and 2022 reviewers and Pierre Olivier, our shepherd at USENIX ATC 2023, for their insightful comments.

References

[1] Bob Amstadt and Eric Youngdale. Wine. https://www.winehq.org/, 1993.

[2] Andrea Arcangeli. seccomp. https://man7.org/linux/man-pages/man2/seccomp.2.html, 2005.

[3] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 335–348, Hollywood, CA, October 2012. USENIX Association.

[4] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, Broomfield, CO, October 2014. USENIX Association.

[5] Alfred Bratterud, Alf-Andre Walla, Hårek Haugerud, Paal E Engelstad, and Kyrre Begnum. IncludeOS: A minimal, resource efficient unikernel for cloud services. In *2015 IEEE 7th international conference on cloud computing technology and science (cloudcom)*, pages 250–257. IEEE, 2015.

[6] Juan Cespedes. ltrace. https://ltrace.org/, 1997.

[7] Buddhika Chamith, Bo Joel Svensson, Luke Dalessandro, and Ryan R. Newton. Instruction punning: Lightweight instrumentation for x86-64. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 320–332, New York, NY, USA, 2017. Association for Computing Machinery.

[8] Jeff Dike. User-mode linux. In *5th Annual Linux Showcase & Conference (ALS 01)*, 2001.

[9] Gregory J. Duck, Xiang Gao, and Abhik Roychoudhury. Binary rewriting without control flow recovery. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 151–163, New York, NY, USA, 2020. Association for Computing Machinery.

[10] Adam Dunkels. Design and Implementation of the lwIP TCP/IP Stack. *Swedish Institute of Computer Science*, 2(77), 2001.

[11] Free Software Foundation. The GNU C Library (glibc). https://www.gnu.org/software/libc/, 1988.

[12] Will Glozer. wrk: Modern HTTP benchmarking tool. https://github.com/wg/wrk, 2012.

[13] Michio Honda, Felipe Huici, Costin Raiciu, Joao Araujo, and Luigi Rizzo. Rekindling network protocol innovation with user-level stacks. *SIGCOMM Comput. Commun. Rev.*, 44(2):52–58, apr 2014.

[14] Galen Hunt and Doug Brubacher. Detours: Binary interception of win32 functions. In *Proceedings of the 3rd Conference on USENIX Windows NT Symposium - Volume 3*, WINSYM'99, page 14, USA, 1999. USENIX Association.

[15] Intel. Data Plane Development Kit. https://www.dpdk.org/, 2010.

[16] Intel. Intel 64 and IA-32 Architectures Software Developer Manuals, 2023.

[17] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a highly scalable user-level TCP stack for multicore systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 489–502, Seattle, WA, April 2014. USENIX Association.

[18] Taesoo Kim and Nickolai Zeldovich. Practical and effective sandboxing for non-root users. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 139–144, San Jose, CA, June 2013. USENIX Association.

[19] Paul Kranenburg. strace. https://strace.io/, 1991.

[20] Gabriel Krisman Bertazi. Syscall User Dispatch. https://www.kernel.org/doc/html/latest/admin-guide/syscall-user-dispatch.html, 2021.

[21] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, Cristian Banu, Laurent Mathy, Răzvan Deaconescu, Costin Raiciu, and Felipe Huici. Unikraft: Fast, specialized unikernels the easy way. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 376–394, New York, NY, USA, 2021. Association for Computing Machinery.

[22] Hsuan-Chi Kuo, Dan Williams, Ricardo Koller, and Sibin Mohan. A linux in unikernel clothing. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[23] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 460–477, New York, NY, USA, 2017. Association for Computing Machinery.

[24] Jing Liu, Anthony Rebello, Yifan Dai, Chenhao Ye, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Scale and performance in a filesystem semi-microkernel. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 819–835, New York, NY, USA, 2021. Association for Computing Machinery.

[25] Google LLC. gVisor. https://gvisor.dev/, 2018.

[26] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 461–472, New York, NY, USA, 2013. Association for Computing Machinery.

[27] Ilias Marinos, Robert N.M. Watson, and Mark Handley. Network stack specialization for performance. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 175–186, New York, NY, USA, 2014. Association for Computing Machinery.

[28] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *USENIX winter*, volume 46, 1993.

[29] Ruslan Nikolaev and Godmar Back. Virtuos: An operating system with kernel virtualization. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 116–132, New York, NY, USA, 2013. Association for Computing Machinery.

[30] Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. A binary-compatible unikernel. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE 2019, page 59–73, New York, NY, USA, 2019. Association for Computing Machinery.

[31] Pierre Olivier, Hugo Lefeuvre, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. A syscall-level binary-compatible unikernel. *IEEE Transactions on Computers*, 71(9):2116–2127, 2022.

[32] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 1–16, Broomfield, CO, October 2014. USENIX Association.

[33] 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. Unikernel linux (ukl). In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 590–605, New York, NY, USA, 2023. Association for Computing Machinery.

[34] Luigi Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 101–112, Boston, MA, June 2012. USENIX Association.

[35] Salvatore Sanfilippo. Redis - Remote Dictionary Server. https://redis.io/, 2009.

[36] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert Van Renesse, and Hakim Weatherspoon. X-Containers: Breaking Down Barriers to Improve Performance and Isolation of Cloud-Native Containers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 121–135, New York, NY, USA, 2019. Association for Computing Machinery.

[37] Livio Soares and Michael Stumm. FlexSC: Flexible system call scheduling with Exception-Less system calls. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, Vancouver, BC, October 2010. USENIX Association.

[38] Ziye Yang, James R Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E Paul. Spdk: A development kit to build high performance storage applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 154–161. IEEE, 2017.