System Call Interposition Without Compromise

Adriaan Jacobs
DistriNet, KU Leuven
Belgium

Merve Gülmez
Ericsson Security Research
Sweden
DistriNet, KU Leuven
Belgium

Alicia Andries
DistriNet, KU Leuven
Belgium

Stijn Volckaert
DistriNet, KU Leuven
Belgium

Alexios Voulimeneas
TU Delft
Netherlands

***Abstract*—Syscall interposition is crucial for tools that monitor/modify application behavior. Mainstream OSes have, therefore, provided syscall interposition APIs for years, but these often incur prohibitive performance penalties in syscall-intensive applications. Recent work showed how to reduce this overhead by rewriting syscall instructions[1] to invoke the interposer directly, avoiding expensive mode/context switches. However, these methods may not locate/rewrite all relevant instructions, which is essential for many applications.**

Our key insight is to combine the aforementioned techniques to efficiently intercept *all* system calls. We present lazypoline, a tool that uses slow kernel interfaces to exhaustively locate valid syscall instructions upon their first use, and then lazily rewrites them to invoke the interposer directly in all subsequent executions. We extensively evaluate lazypoline on micro- and macrobenchmarks and show that it is non-intrusive, fully exhaustive, and it achieves the efficiency of pure rewriting, even for datacenter-scale syscall-intensive workloads.

I. Introduction

Many user-space programs interact continuously and frequently with the operating system (OS) kernel. They do this, for instance, to access system resources or to perform actions that affect other processes. Since system calls (syscalls) are the primary means for these interactions, one can easily monitor or manipulate an application's behavior by installing a syscall interposer. There are various use cases for such interposers, including but not limited to (i) tracing and debugging [1–3], (ii) improving program reliability [4–8] and security [9–27], (iii) emulating a different OS [28, 29], (iv) adding binary compatibility support for new OS subsystems [30–35], and (v) transparently switching to a custom network stack [36, 37].

Most of these use cases require fast and effective syscall interposition, but creating a suitable, generic, and widely applicable mechanism has proven to be a significant challenge in practice. Naturally, the kernel is able to reliably intercept and redirect *all* system calls, but it should not run arbitrary interposer code with elevated privileges for reasons of reliability and security [38]. Hence, most kernel-based interposition mechanisms switch back to user mode before invoking the interposer. `ptrace`'s system call tracing mode, for instance, transfers control to a tracer process when a tracee process invokes or returns from a system call [39]. These control-flow transfers require expensive context switches. By contrast, Syscall User Dispatch (SUD) [40] and some `seccomp` filters (see `seccomp-user` in Table I) transfer control to interposer functions that execute within the context of the tracee process [25]. These mechanisms still add additional mode switches to every intercepted system call, decreasing the *efficiency* of the tracee process [25, 36].

As an alternative, some systems allow programs to install kernel-space interposers using Berkeley Packet Filters (BPF), shown as `seccomp-bpf` in Table I. These interposers maintain high performance, but, for security reasons, are severely restricted in which actions they can perform [41]. These restrictions can hurt the interposers' *expressiveness* to the point where they cannot implement the required functionality of many interposition use cases [3, 4, 23, 25, 42].

Finally, some more recent approaches side-step the efficiency and expressiveness issues [6, 36, 43], by rewriting syscall instructions up front such that the interposer code is directly invoked without first passing through the kernel at all. While such methods achieve maximal efficiency without restricting the interposer's expressiveness, their reliance on correct static binary disassembly and rewriting prevents them from *exhaustively* interposing all syscalls in practice. In addition, they cannot interpose syscall instructions created after the initial rewriting phase. These restrictions are incompatible with some popular ways to develop [44], deploy [45], protect [46], and obfuscate [47] software, which rely on dynamically loading or generating new code [48], and modifying or rewriting existing code [49]. At the same time, a vast number of application monitoring scenarios *require* the exhaustive interpositioning of syscalls to ensure security [23, 24, 42], reliability [1, 3], or even basic functionality [4, 5, 7, 50].

In short, **the current state of the art does not simultaneously support *expressive*, *exhaustive*, and *efficient* user-space syscall interposition.** As a result, some projects with extreme security or performance requirements now modify OS kernels to provide first-class support for their particular use case [21, 25, 42], or even develop custom hardware modifications to achieve their goals [22, 25]. We believe that such *intrusive* changes to the Trusted Computing Base (TCB) should be avoided to maintain a secure, reliable, and maintainable software stack in the long term.

This paper presents the first non-intrusive approach that enables expressive, exhaustive, and efficient syscall interposition. Our key idea is to create a *hybrid* interposition mechanism that combines the exhaustiveness of kernel interfaces like SUD [40]

1. Throughout this paper, we will use the term "syscall instruction" to refer to both the x86 `SYSCALL` and `SYSENTER` instructions.

Characteristics	`ptrace`	`seccomp-bpf`	`seccomp-user`, SUD	Binary Rewriting [6, 36]	Our Approach
Expressiveness	Full	Limited	Full	Full	Full
Exhaustiveness	✓	✓	✓	✗	✓
Efficiency	Low	High	Moderate	High	High

TABLE I: Characteristics of popular non-intrusive syscall interposition solutions. `seccomp-bpf` refers to `seccomp`-based syscall interposition that operates entirely in kernel space using Berkeley Packet Filters (BPF), while `seccomp-user` defers the handling of a syscall back to user space, similar to SUD.

with the efficiency of binary rewriting techniques [6, 36] to invoke a user-space interposer that is maximally expressive.

We implemented our design in an open-source tool called lazypoline[2]. We extensively evaluated lazypoline using microbenchmarks and real-world applications, and show that our approach enables exhaustive, expressive, and efficient syscall interposition.

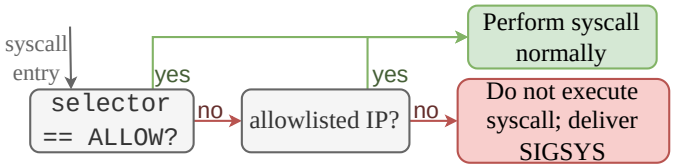


Fig. 1: The Linux syscall kernel entry path under SUD.

II. Background

Various techniques for syscall interposition exist, such as binary rewriting [6, 36, 43], kernel interfaces [39–41], and hardware or kernel modifications [21, 22, 25]. For this work, we mainly focus on approaches that depend on binary rewriting and existing kernel interfaces, summarized in Table I, since they do not require intrusive changes to the Trusted Computing Base (TCB). We briefly discuss approaches that require kernel or hardware modifications in Section VII.

A. Linux Kernel Interfaces for Syscall Interposition

Linux has historically offered a number of interfaces to monitor and control the application's use of syscalls.

`ptrace` allows one *tracer* thread to attach to multiple *tracee* threads [39], and synchronously be notified of all the syscalls they perform. The tracer can inspect and modify the syscall number, arguments, and return value, as well as any tracee memory and registers while it is in the syscall stop state. Although this provides a powerful interface for debugging [1] and simple tracing [3], `ptrace` is infamous for the significant slowdown it typically incurs [25, 42, 51]. Its overhead stems both from the many additional syscalls required to perform even basic operations on the tracee, as well as the context switch necessary to schedule the tracer on every tracee syscall.

`seccomp` presents a more efficient interface [41], specifically targeted at sandboxing. Users can supply Berkeley Packet Filter (BPF) programs that take superficial syscall information as input, and output an action for the kernel to take, e.g. permit or reject the syscall. The kernel then directly runs these BPF programs whenever a syscall is invoked, which yields highly efficient syscall interposition without any redundant mode switches (see `seccomp-bpf` in Table I). However, BPF has limited expressiveness. For example, it does not allow simple operations such as dereferencing pointers. This makes BPF an ill fit in scenarios that require deep inspection or modification of invoked syscalls, e.g., dynamic software updating [7] and more thorough sandboxing [23–26]. Despite tremendous effort to bring more expressive filtering to `seccomp`, e.g., using eBPF [52], upstreaming efforts have traditionally met resistance from Linux kernel developers [53] hesitant to add complexity, and thus potentially exploitable bugs [54–58], in what is considered the kernel's "most important security isolation boundary" [59]. Newer `seccomp` actions allow filters to delegate handling back to user space for increased expressiveness [25, 41] (see `seccomp-user` in Table I), but naturally involve extra mode switches, hurting performance [60].

Syscall User Dispatch (SUD) [40] is Linux' most recent mechanism to intercept syscalls and handle them in user space. Similar to `seccomp-user`, the core idea is to generate a signal (SIGSYS) whenever a syscall is invoked, which allows the application to implement the interposition via a signal handler. Users can enable and configure SUD on a per-task basis via the `prctl` interface. The interception behavior of SUD is flexibly controllable through a user-space `selector` byte, that applications can set to enable/disable syscall interposing at will. In addition, SUD never triggers a signal for syscalls invoked from an application-specified code address range, regardless of the `selector` value. The location of the `selector` byte and the allowlisted code address range are specified when enabling SUD through `prctl`. Figure 1 schematically represents the updated kernel syscall entry path when SUD is enabled.

While SUD was initially developed as a more elegant alternative to `seccomp-user` in the Wine compatibility layer [28], it has since found its way into other use cases as well, such as sandboxing [60]. A typical SUD deployment sets the `selector` to `ALLOW` at the start of the SIGSYS handler, invokes the syscall interposer function from within the signal handler, and resets the `selector` to `BLOCK` at the end, such that subsequent application syscalls are also intercepted. To avoid recursively triggering SUD when exiting the SIGSYS handler and returning to the application context via `rt_sigreturn`, the syscall instruction performing the sigreturn is typically included in the allowlisted code address range [40].

Despite being one of the most efficient available kernel interfaces for syscall interposition, recent work has shown that

2. https://github.com/lazypoline/lazypoline

SUD still incurs a significant slowdown for syscall-intensive applications like web servers [36].

B. Binary Rewriting for Syscall Interposition

Various approaches use binary rewriting techniques for syscall interposition [5–8, 36, 43]. They largely perform the following steps: (i) disassemble the binary, (ii) identify syscall instructions, and (iii) replace the identified instructions with an invocation of the syscall interposer's code. These approaches eliminate multiple mode switches between kernel and user mode, while still running the interposers' code in user space. Hence, they increase the efficiency of a syscall interposer while maintaining full expressiveness and avoiding an increase in the size of the Trusted Computing Base (TCB).

Despite these advantages, each of the steps involved in this binary rewriting procedure has historically come with complications and incompatibilities [34, 43, 61, 62], often exacerbated on architectures with variable-length instructions like x86-64 [6, 36]. For instance, syscall instructions may inadvertently appear as part of other instructions or data, and they occupy just two bytes on x86-64, while `jmp/call` instructions with an arbitrary destination address are much larger than that.

To overcome this latter issue, previous binary rewriters make assumptions about the surrounding code [6, 43], and often rewrite or move a whole sequence of instructions. In doing so, they rely on current-day code generator properties or common practices to match their rewriting rules. Instead, Yasukata et al. recently proposed a new binary rewriting approach for x86-64 binaries [36], called zpoline, that resolves the aforementioned rewriting issue. zpoline replaces the two-byte `syscall/sysenter` instruction with the two-byte `call rax` instruction and instantiates some trampoline code at virtual address 0. x86-64 applications have to store the syscall number in `rax` before invoking a syscall, according to the calling convention. Hence, these `call rax` instructions jump to a virtual address between 0 and the max syscall number *N*, typically under 500. zpoline populates this virtual address range with a `nop` sled that leads into a jump to user-defined code, i.e., the syscall interposer's code.

The major advantage of the zpoline technique is that, by design, it *cannot fail to rewrite a syscall instruction*, which definitively solves the final step in the rewriting procedure. However, being a static binary rewriter, zpoline still struggles with the challenges involved in every other step of the rewriting procedure, just like prior work [6, 43]. For instance, it cannot discover syscall instructions that are loaded or crafted *after* its static disassembly phase, such as those found in dynamically generated code. Furthermore, zpoline still depends on accurate static binary disassembly, which is extremely difficult to achieve in practice without using assumptions and heuristics [6, 63, 64].

III. Goal & Design

The overarching goal of this work is to provide a multi-purpose syscall interposition solution that is suitable for various scenarios with different requirements. We define three important properties interposers must have to satisfy this goal, and find that existing interposers never have all three, as shown in Table I. This section describes these properties in more detail, presents our novel, hybrid interposition method, and explains how it overcomes the limitations of previous work *by design*.

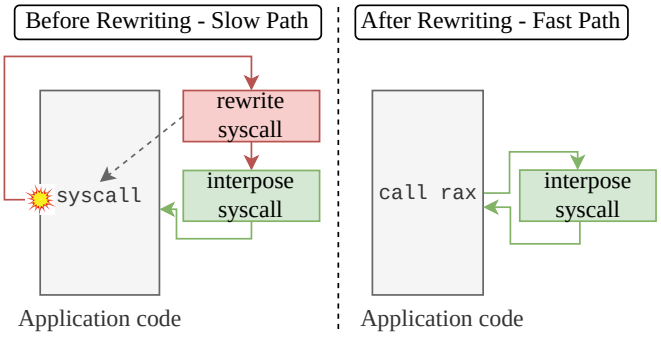


Fig. 2: The fast and slow path of our lazy rewriting design.

Expressiveness refers to the capabilities of the syscall interposer. For instance, `seccomp`-based interposition techniques that completely operate in kernel space (`seccomp-bpf`) cannot perform deep argument inspection of a syscall. Consequently, the expressiveness of such approaches is limited.

Exhaustiveness refers to the ability to interpose *all* invoked syscalls. For instance, binary rewriting-based techniques like zpoline [36] and SaBre [6] cannot interpose syscalls triggered from syscall instructions located in dynamically loaded or generated code, e.g., JIT-compiled code, and rely on heuristics and assumptions to correctly disassemble and identify legitimate syscall instructions. Consequently, they are *not* exhaustive.

Efficiency refers to the performance penalty that the syscall interposer imposes on applications. For instance, when SUD is enabled the kernel raises a signal for each invoked syscall. Previous work [36], as well as our own evaluation (see Section V), shows that SUD has a significant performance impact on syscall-intensive workloads due to the overhead of the required signal handling. Consequently, we classify SUD's performance as moderate.

We find that the kernel's involvement in identifying valid syscall invocation sites is critical to achieve our goal. An exhaustive kernel interface like SUD or `ptrace` can reliably identify the precise code bytes that represent a real syscall instruction at the moment it is first executed. This insight leads us to design a *hybrid* interposer. We use a kernel interface such as SUD as a *slow-path*, *catch-all* mechanism, and rewrite any encountered syscall instructions to install a fast-path mechanism that invokes the user-space interposer code directly on subsequent executions, bypassing the slow path. We illustrate our approach in Figure 2.

The slow path has a dual purpose: (i) it rewrites any previously not encountered syscall, and (ii) it interposes the first syscall triggered from such instructions. This is shown as "Before Rewriting" in Figure 2. When the application tries to use a rewritten syscall instruction, it will actually jump

to our syscall interposer. This is shown as "After Rewriting" in Figure 2. The slow path remains constantly enabled to discover new syscall invocation sites.

Our approach achieves expressiveness, exhaustiveness, and efficiency because (i) both the slow and the fast path are fully expressive, (ii) the slow path is exhaustive and does not depend on assumptions or heuristics to identify legitimate syscall instructions, and (iii) our fast path *eventually* interposes the vast majority of invoked syscalls. Note that we overcome the primary limitations of prior rewriting work *by design*. Our slow path only ever rewrites real, aligned syscall instructions, avoiding compatibility issues. At the same time, *all* syscall invocations are reliably interposed, regardless of the way in which they were conceived into the code.

Security Considerations: Similar to other expressive and efficient syscall interposition mechanisms [6, 36], our approach does not provide any security guarantees against an attacker that aims to either execute syscalls without being detected or access potential sensitive internal state maintained by the interposer. We discuss techniques to enhance the security of our approach in Section VI.

IV. Implementation

We implemented our tool named lazypoline in 1.4k LoC of C/C++ code and 200 lines of x86-64 assembly. In this section, we outline implementation details of our lazy rewriting mechanism, as well as various implementation challenges we had to overcome with both the fast and slow interposition path.

A. The Slow Path

Based on our review of eligible exhaustive interception mechanisms in Section II-A, we opted to use SUD to implement the slow path, primarily because of its simple and flexible control via the `selector` byte, such that any syscalls made from within the interposer can efficiently bypass interposition.

a) Motivation of SUD choice: Under `ptrace`, the tracer process would still be notified of such syscalls, even if it chooses not to interpose them, resulting in significant runtime overhead [23, 42, 51]. `seccomp` does provide a more efficient option to permit interposer syscalls, by filtering on the code address of the syscall invocation, similar to SUD's allowlisted code address range [60, 65]. However, this still requires loading and executing a BPF program for every syscall, which previous work has shown to be slower than SUD's more direct filtering [60]. In addition, `seccomp` proves to be much less flexible to work with in practice due to its security orientation. For instance, filters can never be uninstalled, even after calling `execve` to load a new application binary, which developers may not want to run with any syscall interposition enabled [66]. These flexibility issues were the main motivation for the Wine project to develop SUD in the first place [65].

b) Rewriting to the fast path: Once our SUD SIGSYS handler is invoked, it will rewrite the invoked syscall instruction to `call rax`, establishing the fast path for that particular syscall invocation site. We implement the rewrite by temporarily changing the page permissions to `RW`, modifying the code page, and restoring its original page permissions afterward. We hold a spinlock throughout this procedure to prevent race conditions where one thread revokes write permissions while another thread is busy rewriting.

c) selector-only SUD: Our usage of SUD is somewhat unconventional compared to the typical deployment described in Section II-A, because we wanted to avoid pitfalls and implementation complexities faced by previous work [25, 60]. Concretely, after finishing the syscall instruction rewrite, we do not directly invoke the syscall interposer function from inside the SIGSYS handler. Rather, we modify the application's provided register context from within the signal handler to resume execution at the start of our fast-path entry point upon returning out of the signal handler. We do this by setting `REG_RIP` to the address of our generic interposer entry point. We sigreturn out of the signal handler with the `selector` byte still set to `ALLOW`.

This approach has several advantages compared to the more traditional design of allowlisting a code address range for the SIGSYS `rt_sigreturn` and interposing the syscall from the SIGSYS handler. First off, we are able to share a single syscall handling implementation between the fast and slow path, which reduces development and maintenance effort, especially for interposing more complex syscalls such as `vfork`, `clone`, and `rt_sigreturn` itself. Secondly, it allows us to avoid excluding *any* code addresses from SUD interception, since our implementation allows us to sigreturn from the SUD SIGSYS handler with the `selector` byte set to `ALLOW`. This sidesteps the primary reason why SUD is not recommended for exhaustive syscall interception in a security context [25, 40], since attackers could simply jump to any allowlisted syscall instruction. Instead, we reduce the problem of attacker-robust SUD interception to one of isolating the `selector` byte from malicious overwrites, which is a simple intra-process memory isolation problem resolvable through a breadth of existing techniques [22–24, 26, 42, 60, 67, 68]. We discuss potential security extensions of lazypoline in Section VI.

B. The Fast Path

While any of the existing state-of-the art binary rewriters would suffice to build our fast path [6, 36, 43], we choose zpoline due to its unique ability to rewrite *any* syscall instruction, provided it knows its location (see Section II). Since we already solve the problem of exhaustively locating all valid syscall instructions through the slow path, this is the most desirable quality to base our selection on. We reimplement part of the open-source zpoline prototype to serve as our fast path for syscall interposition [69]. We improve the prototype in a number of ways to handle more complex applications, adhere more thoroughly to application's expectations of the syscall ABI, and be robust enough to avoid imposing restrictions on the interposer's code and abilities. We briefly overview the most important changes here, as well as our handling of application complexities in general.

a) Multiprocessing and Multithreading: SUD can be activated on a per-task basis, and it is deactivated on every

Listing 1: Simplified disassembly of pthread initialization routine. `r12` contains `&__stack_user`, a list of threads with user-provided stacks in use.

```
mov xmm0, r12          ; load into both
punpcklqdq xmm0, xmm0  ; halves of xmm0
; ... irrelevant
syscall                ; set_tid_address
; ... irrelevant
syscall                ; set_robust_list
; ... irrelevant
movups [r12], xmm0     ; write `&__stack_user`
                       ; to `prev` + `next`
```

`(v)fork`, `clone`, and `execve` [40]. Hence, we re-enable SUD in the child to continue intercepting syscalls from any new invocation sites. We supply a different `selector` byte per task, and store it in a `%gs`-relative memory region, which we map and initialize whenever a new task is created. This ensures that even tasks that share their virtual memory subsystem using `CLONE_VM`, e.g., threads, can separately enable/disable SUD.

b) ABI Compatibility: On x86-64 Linux, a syscall does not clobber any registers apart from `rax`, `rcx`, and `r11`. To maximize compatibility with existing applications, any interposer must strictly adhere to this ABI. Similar to existing interposers, we preserve all general purpose registers across the interposition. However, we found that existing open-source interposers do not always preserve all extended state components [6, 36, 42, 43], primarily those that use binary rewriting. In practice, this precludes the interposer from using any of the SSE/AVX vector or legacy x87 FPU registers [70], which modern compilers and libraries, including libc, may do ad libitum for performance reasons.

To quantify this issue in practice, we wrote an Intel Pin [71] tool that tracks at run time whether a syscall is executed between a consecutive write to and read from the same register. This indicates that the application expected the register contents to remain preserved across the syscall. Interposers should respect these expectations. Note that, as the Pin tool performs a dynamic analysis, it will generally underestimate the frequency of such occurrences. Still, when we evaluated popular coreutils (see Section V) using this tool, we found that, apart from general purpose registers, many programs also expect the kernel to preserve extended state. Listing 1 presents a representative example, taken from the pthread initialization routine of glibc 2.31. The compiler uses a single SSE `movups` to initialize two adjacent struct fields at once. Because the function contains no instructions that clobber SSE registers, the compiler populates the relevant `xmm0` register up front. The main body of the initialization routine performs two syscalls before actually using `xmm0` to initialize the struct fields.

By default, lazypoline preserves all the SSE, AVX, and x87 legacy FPU state, since we aim to provide the interposer with a fully expressive environment without arbitrary restrictions. At the same time, we recognize that interposers which operate at the extreme performance levels we scale to (see Section V) might well be willing to incorporate *some* restrictions to maximize efficiency, or at least tune the interposition to their specific workload. We support those users through a configurable option that controls which extended state components are preserved, if any. By using this option, interposers accept that they must either not clobber the extended state, or preserve it themselves. On interposer entry, we use the `xsave` instruction [70] to save any extended state components to a dedicated per-task, `%gs`-relative memory region, because they can amount to a sizable chunk of data that could overflow the application stack when pushed on top of it. The per-task `xstate` memory region is nonetheless managed as a stack by lazypoline to support nested interposer invocations (see *Signal Handling*). On interposer exit, we use the `xrstor` instruction [70] to restore the register state from the top of our `xstate` component stack.

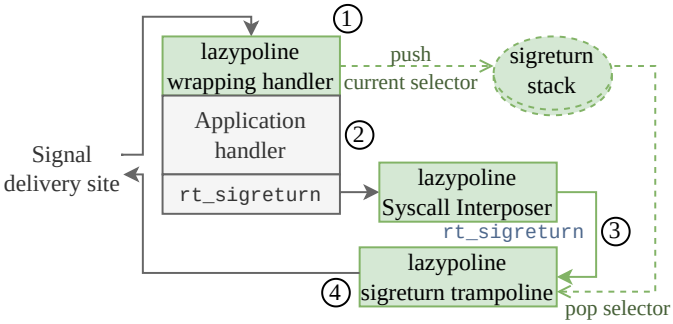


Fig. 3: Application signal handler execution and interposition under lazypoline. The dotted lines represent data flow, the full lines represent control flow.

c) Signal Handling: Whenever a signal is delivered, we need to ensure that SUD interception is enabled and the `selector` is set to `BLOCK`. This allows us to catch any new syscalls invoked by the signal handler. While we trivially meet these conditions when signals are delivered during regular application execution, we must specially handle signals delivered while the application executes our interposer.

Using our exhaustive syscall interposition, we intercept all of the application's attempts to register custom signal handlers. We modify the `sigaction` structure passed to the kernel to register our own wrapper handler instead, and keep track of the original application handlers in a separate table. Figure 3 visualizes the way lazypoline handles signal delivery from this point on. Whenever a signal is delivered for which we need to invoke an application signal handler, our wrapper handler pushes the current value of the `selector` to a separate, `%gs`-relative *sigreturn stack* for later use, before setting the `selector` to `BLOCK` and invoking the relevant application handler from the table (①). From then on, all syscalls executed from the application handler are interposed normally by our hybrid slow- and fast-path mechanisms (②).

When our interposer intercepts the application signal handler's `rt_sigreturn` call, it must restore the `selector` value we previously pushed to our `%gs`-relative *sigreturn stack* before handing control back to the original signal delivery context.

However, it cannot restore the `selector` before invoking the `rt_sigreturn` itself, since that might recursively trigger interposer invocations when the `selector` is restored to `BLOCK`. Hence, our interposer calls `rt_sigreturn` with the `selector` set to `ALLOW` (③), but forces the application to return to its regular execution context through an indirection called the *sigreturn trampoline*. This trampoline restores the original `selector` value and subsequently transfers control to the original signal delivery context (④).

V. Evaluation

We evaluate lazypoline on its ability to comprehensively interposition all system calls in dynamically generated code, as well as its performance overhead in real-world, syscall-intensive workloads, e.g., web servers.

A. Exhaustiveness

We test lazypoline on the Tiny C Compiler (tcc) [72], notable for its just-in-time (JIT) compilation within the C programming environment. We introduce a singular, non-libc `getpid` syscall into a C application, which we then ran under tcc (`tcc -run`). We evaluate the exhaustiveness of lazypoline's different components by JIT-ing the same program under SUD, zpoline, and lazypoline. All interposers perform the same interposition function; they print the current system call with all its arguments, then execute the syscall without modification and return the result. As expected, lazypoline and SUD print the exact same syscalls, in the same order, including our introduced `getpid` syscall, satisfying our goal of matching SUD's exhaustiveness. zpoline's trace, on the other hand, does not include the relevant `getpid`, since the syscall instruction from which it was invoked did not exist yet at load time, when zpoline scanned the `tcc` binary.

Note that, while our `tcc` evaluation validates lazypoline's ability to interpose dynamically loaded or generated code, our exhaustiveness advantages over zpoline span beyond this alone. As mentioned in Section II-B, static binary scanning may additionally fail to identify syscall instructions that were not or incorrectly disassembled, e.g., due to coverage limitations during code discovery and disambiguation [64]. Moreover, our live syscall identification resolves issues with static misidentification due to faulty disassembly, yielding a higher-fidelity tool without the risk of accidentally destroying misidentified code.

B. Performance

We evaluate the performance impact of lazypoline's exhaustive syscall interpositioning by benchmarking different configurations and scenarios, and we re-evaluate the existing state of the art for a direct comparison. Throughout the evaluation, we use a "dummy" interposition function that simply executes the syscall with its original arguments and returns the result, to benchmark the overhead of lazypoline's interposition alone. As such, the baseline for all experiments is the native execution of the benchmark without any interposition. We ran all experiments on a 48-core Intel Xeon Gold 5318S CPU running at 2.10 GHz and 1.0 TiB of RAM. We disable hyperthreading on the CPU to reduce measurement noise [73]. The machine runs Ubuntu 22.04.3 LTS with version 5.15.0-83 of the Linux kernel.

First, we evaluate the overhead of lazypoline's fast path in a synthetic microbenchmarking setup, since the fast path case matches zpoline's pure rewriting configuration. We evaluate whether the overhead is comparable, and explain the differences in detail. The later macrobenchmarks will evaluate whether the added initial execution of the slow path significantly degrades the aggregated efficiency of our hybrid design in real-world, datacenter-scale workloads.

a) Microbenchmarks: We measure the CPU cycles required to interpose a non-existent syscall (number `500`) 100M times. A non-existent syscall gives a lower bound on the round trip time of entering and exiting the kernel. Hence, the overhead differences between the different interposers will be maximally enlarged, which yields the clearest results. Additionally, syscall number `500` will cause zpoline's `nop` sled to be entered at its very tail, which minimizes zpoline's overhead, allowing us to precisely evaluate the additional overhead of our hybrid design on top of zpoline's pure rewriting. We measure the overhead over the baseline 10 times, and report the geomean and maximal standard deviation in Table II for zpoline, SUD, and lazypoline with and without preservation of extended state components (`xstate`). For the lazypoline measurements, we manually rewrote the syscall instruction up front, so there is no initial execution of the slow path. This way, the measured overhead does not depend on the number of microbenchmark iterations, but solely represents lazypoline's steady state. Hence, lazypoline's additional slowdown compared to zpoline in Table II is not due to any live rewriting, but solely due to the different, slower syscall entry path the kernel takes when any of its interception interfaces are enabled, even when that specific syscall is exempt from interception. This overhead stems from kernel code that checks if syscall interception is enabled, and, in the case of SUD, also reads out the user-space `selector` byte. Table II shows the significance of this overhead, even on non-interposed, native syscalls. To verify that this is our only overhead over zpoline, we run the microbenchmark of lazypoline's fast path again with SUD disabled. Figure 4 shows that, without the SUD overhead, lazypoline's fast path matches zpoline's performance. Hence, the overhead labeled as "enabling SUD" precisely represents the added cost of our exhaustiveness guarantee over prior work.

We separately measure the cost of properly adhering to the syscall ABI by preserving the extended state components. As Figure 4 shows, this preservation is responsible for the majority of lazypoline's overhead over baseline. To get a better idea of whether this cost actually benefits compatibility in practice, we run our Pin tool, which dynamically tracks register preservation expectations across syscalls, on ten popular coreutils shown in Table III. Given that most syscalls will be executed by libc, we run the evaluation on two different libc versions in two different Linux distributions, targeting two different x86-64 micro-architectural support levels [74]. The first instance

zpoline	1.23x
lazypoline without `xstate` preservation	1.66x
lazypoline	2.38x
SUD	20.8x
baseline with SUD enabled (`selector=ALLOW`)	1.42x

TABLE II: Microbenchmarking overhead compared to baseline. Standard deviation is below 0.19%.

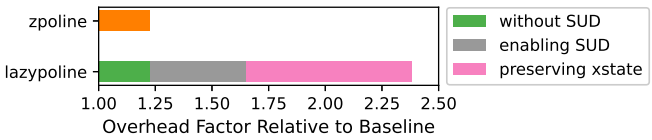


Fig. 4: lazypoline's overhead breakdown.

Coreutils	Ubuntu 20.04	Clear Linux
ls	✓	✓
pwd	✗	✓
chmod	✗	✓
mkdir	✓	✓
mv	✓	✓
cp	✓	✓
rm	✗	✓
touch	✗	✓
cat	✗	✓
clear	✗	✓

TABLE III: Ten popular coreutils evaluated with our Pin tool on two different Linux distributions. ✓ indicates that the program expected an extended state component to be preserved across at least one syscall. ✗ means we found no such issues.

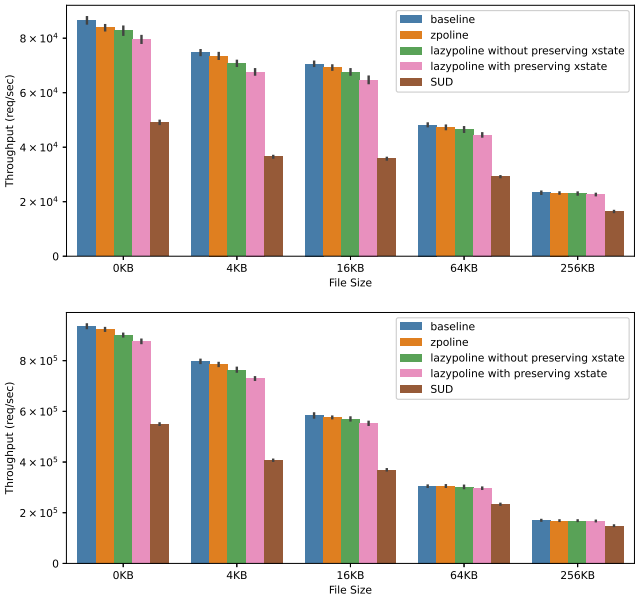
runs Ubuntu 20.04 with glibc 2.31, targeting x86-64-v1, which only assumes the availability of the legacy x87 FPU state and the first eight `xmm` registers. The second instance runs Intel's Clear Linux (version 41040) with glibc 2.39, targeting up to x86-64-v4, but containing dynamic CPU feature checks that only enable x86-64-v3 code paths on the AMD Ryzen 7 PRO 5850U (Zen 3) CPU we used for this evaluation. The higher micro-architectural support level reflects a deployment scenario where an optimized libc is able to use more of the extended state components.

Table III summarizes the results. Overall, across both distributions and all evaluated utilities, we found that the vast majority of executed syscalls do not suffer from any compatibility issues when only preserving general purpose registers. However, many programs contain at least one affected syscall for which some extended state components should be preserved. In Ubuntu 20.04, 40% of the evaluated coreutils are affected by the same pthread initialization issue, which we described in Listing 1. Similarly, in Clear Linux, *all* programs are affected by a singular issue, this time in `ptmalloc_init`, when prepopulating an `xmm` register to intialize some of the `main_arena` state. The program expects an intervening `get_random` syscall to preserve the relevant `xmm` register. Our evaluation indicates that there exists a large potential for users of lazypoline to avoid needlessly suffering the `xstate` preservation cost in a majority of cases. Our Pin tool can help users make informed implementation choices in that regard.

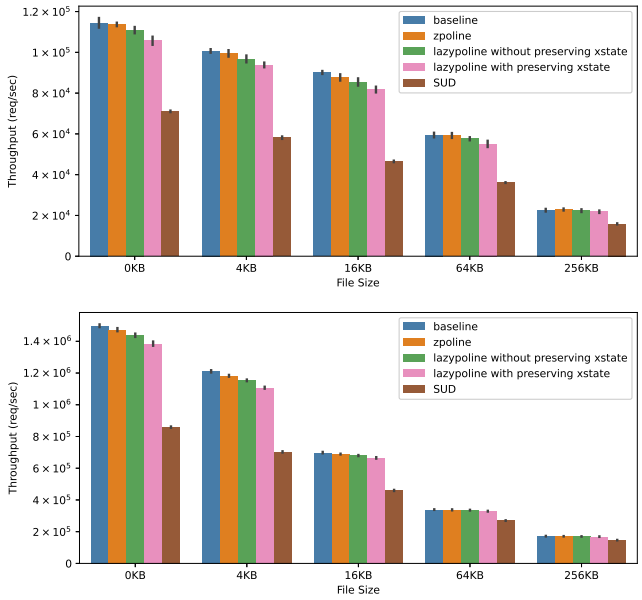
b) Macrobenchmarks: Following the microbenchmarks, we want to evaluate whether our exhaustiveness and compatibility improvements have a significant performance impact on real-world, syscall-intensive workloads. We selected two representative workloads we set out to scale to: lighttpd [75] version 1.4.73, and nginx [76] version 1.25.3. We evaluate the overhead of interposing all syscalls when serving static content of different sizes, once using only a single worker, and once in a more realistic, 12-worker configuration. Syscall interposition traditionally impacts single-threaded performance, since the additional work involved does not introduce additional processes or threads, and does not voluntarily yield to the scheduler. Hence, the overhead will be most clear on the single-worker deployment, with the multi-worker configuration included as reference.

We used the `wrk` client [77] with 36 threads to continuously request the same static resource for 30 seconds via a `keepalive` connection, repeating the test 10 times. The client and server run on a different set of physical cores on the same machine; 12 for the server and 36 for the client, set via `taskset` [78]. They communicate over localhost to create a maximally intensive workload that is not artificially slowed down by arbitrary throughput limits. Apart from lazypoline, we also evaluate zpoline and a typical SUD deployment in the same scenario. Figure 5 visualizes the results.

For fairness, we first compare zpoline against lazypoline without `xstate` preservation. In the very worst case out of all single-worker benchmarks, our hybrid design still maintains 94.72% of the baseline throughput in nginx, and 94.81% in lighttpd, which is respectively only 3.60pp and 2.40pp slower than zpoline when serving the same content, while guaranteeing exhaustive syscall interposition. The results also show that the overhead of preserving `xstate` is at most 4.70pp compared to not preserving it, in the most syscall-intensive single-worker configuration. This `xstate`-preserving deployment provides us with an equally expressive environment as SUD, at almost twice the throughput in many cases. As expected, the overhead difference between all evaluated interposition mechanisms becomes smaller as the size of the served content increases, since that diminishes the syscall intensity of the web server. For instance, from 64 KB on, the overhead difference between zpoline and lazypoline practically vanishes. Yet, when serving 256 KB files, the slowdown of SUD still remains noticeable in both nginx and lighttpd. Even at such larger file sizes, where there is little discernible overhead for the `xstate` preservation, the benefit of using lazypoline instead of SUD for interposition is apparent. Hence, we conclude from our evaluation that lazypoline achieves its main goal of interposing syscalls with the exhaustiveness and expressiveness of SUD at an efficiency level similar to pure binary rewriting.



(a) nginx with 1 (upper) and 12 (lower) worker processes. Standard deviation is below 1.82%.



(b) lighttpd with 1 (upper) and 12 (lower) worker processes. Standard deviation is below 2.92%.

Fig. 5: Performance impact of lazypoline and prior art on web servers.

VI. Discussion

In this section, we discuss possible extensions, and opportunities for future research enabled by our contributions.

Multi-OS Support: Our binary rewriting techniques are applicable to various OSes similar to previous work [36]. However, we are not aware of exhaustive mechanisms with similar characteristics to SUD on other OSes. Hence, we believe lazypoline can currently not support other OSes without using hardware or OS modifications.

Multi-Architecture Support: SUD is a recent kernel interface and we expect it to be fully supported on multiple architectures in the near future. Similar to previous work [36], our binary rewriting techniques are not applicable to architectures that have fixed-length instructions and prohibit jumps to unaligned virtual addresses. However, we can use different binary rewriting techniques to overcome such limitations [6, 43].

Security: Our work, like other efficient user-space syscall interposition approaches [6, 36, 43], does not provide any security guarantees against attackers who specifically target the user-space interposer. If we wish to do so, we need to protect the interposer's sensitive state, e.g., the SUD `selector` byte. Similar to recent work [24–27, 60, 67, 79–91], we can leverage commodity hardware primitives like MPK [92] to isolate the interposer's memory from the attacker-controlled application code. Interestingly, memory isolation itself typically requires expressive interposition of some potentially harmful syscalls [23–25, 51, 67, 93], such as `open` and `mmap`. Several of these are executed very frequently by some applications [51], and failing to interpose even a single one may allow attackers to completely bypass the intended memory isolation [23, 25, 51, 93]. Hence, where previous work resorted to kernel and even hardware changes to meet these demands, our contribution of simultaneously *efficient* and *exhaustive* user-space syscall interposition may itself be used to secure the interposer.

VII. Related Work

In this section, we discuss some additional syscall interposition work that does not solely rely on binary rewriting or standard kernel interfaces (see Section II for those).

Function-Level Interposition: Some work interposes syscall wrapper functions instead of syscalls directly [2, 79, 94, 95]. The performance impact of these solutions is minimal but comes at the cost of exhaustiveness, since syscall instructions can also appear outside of wrapper functions. In addition, function-level interposers must identify all syscall wrapper functions and map them to the syscalls they perform, which does not scale in practice [6, 36].

Intrusive Approaches: A vast amount of existing work with stringent security and/or performance requirements resorts to kernel or hardware modifications to achieve the common goal of efficient, exhaustive, and expressive syscall interposition [21–26, 42, 50, 66, 96–104]. The major disadvantage of hardware and kernel changes is, however, that they are error-prone, hard to maintain, and frequently do not see widespread adoption. In addition, these approaches increase the Trusted Computing Base (TCB), which majorly impacts the security guarantees of not only the interposer, but of the entire system.

VIII. Conclusion

In this work, we present the first non-intrusive syscall interposition design that is simultaneously exhaustive, expressive, and efficient. We achieve this through a novel, hybrid design that combines the efficiency of binary rewriting with the exhaustive

syscall interception of standard kernel interfaces. We implement our design in the lazypoline prototype, and extensively evaluate its performance impact in both micro- and macrobenchmarks. The results indicate that lazypoline is a suitable interposition tool even for workloads with extreme syscall intensity.

We open-source lazypoline to foster future research and allow the community to reuse and extend our work. Our prototype and benchmarks are available at https://github.com/lazypoline/.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Timo Hönig, for their helpful feedback. This research is partially funded by the Research Fund KU Leuven, by the Flemish Research Programme Cybersecurity, and by the EU H2020 MSCA-ITN action 5GhOSTS, grant agreement no. 814035.

References

[1] "gdb," Last accessed 2023. [Online]. Available: https://man7.org/linux/man-pages/man1/gdb.1.html

[2] "ltrace," Last accessed 2023. [Online]. Available: https://man7.org/linux/man-pages/man1/ltrace.1.html

[3] "strace," Last accessed 2023. [Online]. Available: https://man7.org/linux/man-pages/man1/strace.1.html

[4] P. Hosek and C. Cadar, "Safe software updates via multi-version execution," in *International Conference on Software Engineering (ICSE)*, 2013.

[5] L. Pina, D. Grumberg, A. Andronidis, and C. Cadar, "A dsl approach to reconcile equivalent divergent program executions," in *USENIX Annual Technical Conference*, 2017.

[6] P.-A. Arras, A. Andronidis, L. Pina, K. Mituzas, Q. Shu, D. Grumberg, and C. Cadar, "Sabre: Load-time selective binary rewriting," *International Journal on Software Tools for Technology Transfer*, 2022.

[7] L. Pina, A. Andronidis, M. Hicks, and C. Cadar, "Mvedsua: Higher availability dynamic software updates via multi-version execution," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[8] P. Hosek and C. Cadar, "Varan the unbelievable: An efficient n-version execution framework," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[9] D. Bruschi, L. Cavallaro, and A. Lanzi, "Diversified process replicæ for defeating memory error exploits," in *IEEE Performance, Computing, and Communications Conference (IPCCC)*, 2007.

[10] S. Volckaert, B. De Sutter, T. De Baets, and K. De Bosschere, "GHUMVEE: efficient, effective, and flexible replication," in *International Symposium on Foundations and Practice of Security (FPS)*, 2012.

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

[12] A. Voulimeneas, D. Song, F. Parzefall, Y. Na, P. Larsen, M. Franz, and S. Volckaert, "Distributed heterogeneous n-variant execution," in *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2020.

[13] B. Salamat, T. Jackson, A. Gal, and M. Franz, "Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space," in *European Conference on Computer Systems (EuroSys)*, 2009.

[14] N. DeMarinis, K. Williams-King, D. Jin, R. Fonseca, and V. P. Kemerlis, "sysfilter: Automated system call filtering for commodity software," in *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2020.

[15] T. Kim and N. Zeldovich, "Practical and effective sandboxing for non-root users," in *USENIX Annual Technical Conference*, 2013.

[16] C. Canella, M. Werner, D. Gruss, and M. Schwarz, "Automating seccomp filter generation for linux applications," in *Cloud Computing Security Workshop (CCSW)*, 2021.

[17] S. Ghavamnia, T. Palit, and M. Polychronakis, "C2c: Fine-grained configuration-driven system call filtering," in *ACM Conference on Computer and Communications Security (CCS)*, 2022.

[18] S. Ghavamnia, T. Palit, A. Benameur, and M. Polychronakis, "Confine: Automated system call policy generation for container attack surface reduction," in *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2020.

[19] S. Ghavamnia, T. Palit, S. Mishra, and M. Polychronakis, "Temporal system call specialization for attack surface reduction," in *USENIX Security Symposium*, 2020.

[20] V. L. Rajagopalan, K. Kleftogiorgos, E. Göktaş, J. Xu, and G. Portokalidis, "Syspart: Automated temporal system call filtering for binaries," in *ACM Conference on Computer and Communications Security (CCS)*, 2023.

[21] T. Garfinkel, "Traps and pitfalls: Practical problems in system call interposition based security tools," in *Symposium on Network and Distributed System Security (NDSS)*, 2003.

[22] D. Schrammel, S. Weiser, S. Steinegger, M. Schwarzl, M. Schwarz, S. Mangard, and D. Gruss, "Donky: Domain keys – efficient in-process isolation for risc-v and x86," in *USENIX Security Symposium*, 2020.

[23] A. Voulimeneas, J. Vinck, R. Mechelinck, and S. Volckaert, "You shall not (by) pass! practical, secure, and fast pku-based sandboxing," in *European Conference on Computer Systems (EuroSys)*, 2022.

[24] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, P. Druschel, and D. Garg, "Erim: Secure, efficient in-process isolation with memory protection keys," in *USENIX Security Symposium*, 2019.

[25] D. Schrammel, S. Weiser, R. Sadek, and S. Mangard, "Jenny: Securing syscalls for PKU-based memory isolation systems," in *USENIX Security Symposium*, 2022.

[26] M. Hedayati, S. Gravani, E. Johnson, J. Criswell, M. L.

Scott, K. Shen, and M. Marty, "Hodor: Intra-process isolation for high-throughput data plane libraries," in *USENIX Annual Technical Conference*, 2019.

[27] M. Gülmez, T. Nyman, C. Baumann, and J. T. Mühlberg, "Rewind & discard: Improving software resilience using isolated domains," in *IEEE/IFIP Conference on Dependable Systems and Networks (DSN)*, 2023.

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

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

[30] H.-C. Kuo, D. Williams, R. Koller, and S. Mohan, "A linux in unikernel clothing," in *European Conference on Computer Systems (EuroSys)*, 2020.

[31] R. Nikolaev and G. Back, "Virtuos: An operating system with kernel virtualization," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

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

[33] A. Raza, T. Unger, M. Boyd, E. B. Munson, P. Sohal, U. Drepper, R. Jones, D. B. De Oliveira, L. Woodman, R. Mancuso, J. Appavoo, and O. Krieger, "Unikernel linux (ukl)," in *European Conference on Computer Systems (EuroSys)*, 2023.

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

[35] L. Soares and M. Stumm, "{FlexSC}: Flexible system call scheduling with {Exception-Less} system calls," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[36] K. Yasukata, H. Tazaki, P.-L. Aublin, and K. Ishiguro, "zpoline: a system call hook mechanism based on binary rewriting," in *USENIX Annual Technical Conference*, 2023.

[37] R. Jansen, J. Newsome, and R. Wails, "Co-opting linux processes for High-Performance network simulation," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 327–350. [Online]. Available: https://www.usenix.org/conference/atc22/presentation/jansen

[38] T. Garfinkel, B. Pfaff, M. Rosenblum *et al.*, "Ostia: A delegating architecture for secure system call interposition." in *Symposium on Network and Distributed System Security (NDSS)*, 2004.

[39] "ptrace," Last accessed 2023. [Online]. Available: http://man7.org/linux/man-pages/man2/ptrace.2.html

[40] "Syscall user dispatch," Last accessed 2023. [Online]. Available: https://www.kernel.org/doc/html/latest/admin-guide/syscall-user-dispatch.html

[41] "seccomp," Last accessed 2023. [Online]. Available: https://man7.org/linux/man-pages/man2/seccomp.2.html

[42] S. Volckaert, B. Coppens, A. Voulimeneas, A. Homescu, P. Larsen, B. D. Sutter, and M. Franz, "Secure and efficient application monitoring and replication," in *USENIX Annual Technical Conference*, 2016.

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

[44] "V8 javascript engine," Last accessed 2023. [Online]. Available: https://v8.dev/

[45] "Upx – the ultimate packer for executables," Last accessed 2023. [Online]. Available: https://upx.github.io/

[46] J. Zhang, Z. Li, Y. Liu, Z. Sun, and Z. Wang, "Safte: A self-injection based anti-fuzzing technique," *Computers and Electrical Engineering*, vol. 111, p. 108980, 2023.

[47] "Tigress jitter," Last accessed 2023. [Online]. Available: https://tigress.wtf/jitter.html

[48] X. Xu, M. Ghaffarinia, W. Wang, K. W. Hamlen, and Z. Lin, "CONFIRM: Evaluating compatibility and relevance of control-flow integrity protections for modern software," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1805–1821.

[49] "Tigress jitdynamic," Last accessed 2023. [Online]. Available: https://tigress.wtf/jitDynamic.html

[50] J. Vinck, B. Abrath, B. Coppens, A. Voulimeneas, B. D. Sutter, and S. Volckaert, "Sharing is caring: Secure and efficient shared memory support for mvees," in *European Conference on Computer Systems (EuroSys)*, 2022.

[51] R. J. Connor, T. McDaniel, J. M. Smith, and M. Schuchard, "PKU pitfalls: Attacks on pku-based memory isolation systems," in *USENIX Security Symposium*, 2020.

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

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

[54] "CVE-2020-8835." Available from MITRE, CVE-ID CVE-2020-8835., 2020. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-8835

[55] "CVE-2021-31440." Available from MITRE, CVE-ID CVE-2021-31440., 2021. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-31440

[56] "CVE-2021-33200." Available from MITRE, CVE-ID CVE-2021-33200., 2021. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-33200

[57] "CVE-2021-3490." Available from MITRE, CVE-ID CVE-2021-3490., 2021. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-3490

[58] "CVE-2021-29154." Available from MITRE, CVE-ID CVE-2021-29154., 2021. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-29154

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

[60] I. Bumjin, Y. Fangfei, T. Chia-Che, L. Michael, V.-O. Anjo, and D. Nathan, "The endokernel: Fast, secure, and programmable subprocess virtualization," *arXiv preprint arXiv:2108.03705*, 2021.

[61] B. Chamith, B. J. Svensson, L. Dalessandro, and R. R. Newton, "Instruction punning: Lightweight instrumentation for x86-64," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[62] G. J. Duck, X. Gao, and A. Roychoudhury, "Binary rewriting without control flow recovery," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2020.

[63] D. Andriesse, X. Chen, V. van der Veen, A. Slowinska, and H. Bos, "An In-Depth analysis of disassembly on Full-Scale x86/x64 binaries," in *USENIX Security Symposium*, 2016.

[64] C. Pang, R. Yu, Y. Chen, E. Koskinen, G. Portokalidis, B. Mao, and J. Xu, "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)*, 2021.

[65] G. K. Bertazi, "Efficient syscall emulation on linux," *Open Source Summit Europe*, Oct. 2020. [Online]. Available: https://static.sched.com/hosted_files/osseu2020/5c/Gabriel_Krisman_Bertazi-Efficient-syscall-emulation-for-gaming-on-linuxsyscall-emulation-dark.pdf

[66] A. J. Gaidis, V. Atlidakis, and V. P. Kemerlis, "SysX-CHG: Refining Privilege with Adaptive System Call Filters," in *ACM Conference on Computer and Communications Security (CCS)*, 2023.

[67] M. Xie, C. Wu, Y. Zhang, J. Xu, Y. Lai, Y. Kang, W. Wang, and Z. Wang, "Cetis: Retrofitting intel cet for generic and efficient intra-process memory isolation," in *ACM Conference on Computer and Communications Security (CCS)*, 2022.

[68] C. Wu, M. Xie, Z. Wang, Y. Zhang, K. Lu, X. Zhang, Y. Lai, Y. Kang, M. Yang, and T. Li, "Dancing with wolves: An intra-process isolation technique with privileged hardware," *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2022.

[69] "zpoline," Last accessed 2023. [Online]. Available: https://github.com/yasukata/zpoline/tree/master

[70] Intel Inc., *Intel 64 and IA-32 Architectures. Software Developer's Manual*, 2021.

[71] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *Acm sigplan notices*, vol. 40, no. 6, pp. 190–200, 2005.

[72] "Tiny c compiler," Last accessed 2024. [Online]. Available: https://bellard.org/tcc/

[73] D. Beyer, S. Löwe, and P. Wendler, "Reliable benchmarking: requirements and solutions," *International Journal on Software Tools for Technology Transfer*, vol. 21, pp. 1–29, 2019.

[74] x86-64 psABI Authors, "System v application binary interface – amd64 architecture processor supplement (with lp64 and ilp32 programming models) version 1.0," 2024. [Online]. Available: https://gitlab.com/x86-psABIs/x86-64-ABI/-/jobs/artifacts/master/raw/x86-64-ABI/abi.pdf?job=build

[75] "lighttpd," Last accessed 2023. [Online]. Available: https://www.lighttpd.net/

[76] "nginx," Last accessed 2023. [Online]. Available: https://nginx.org/

[77] "wrk," Last accessed 2023. [Online]. Available: https://github.com/wg/wrk

[78] "taskset," Last accessed 2024. [Online]. Available: https://man7.org/linux/man-pages/man1/taskset.1.html

[79] X. Wang, S. Yeoh, P. Olivier, and B. Ravindran, "Secure and efficient in-process monitor (and library) protection with intel mpk," in *European Workshop on System Security (EuroSec)*, 2020.

[80] A. Ghosn, M. Kogias, M. Payer, J. R. Larus, and E. Bugnion, "Enclosure: Language-based restriction of untrusted libraries," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[81] T. Park, K. Dhondt, D. Gens, Y. Na, S. Volckaert, and M. Franz, "Nojitsu: Locking down javascript engines," in *Symposium on Network and Distributed System Security (NDSS)*, 2020.

[82] P. Kirth, M. Dickerson, S. Crane, P. Larsen, A. Dabrowski, D. Gens, S. Volckaert, and M. Franz, "Pkru-safe: Automatically locking down the heap between safe and unsafe languages," in *European Conference on Computer Systems (EuroSys)*, 2022.

[83] M. Gülmez, T. Nyman, C. Baumann, and J. T. Mühlberg, "Friend or foe inside? exploring in-process isolation to maintain memory safety for unsafe rust," in *2023 IEEE Secure Development Conference (SecDev)*, 2023, pp. 54–66.

[84] X. Jin, X. Xiao, S. Jia, W. Gao, D. Gu, H. Zhang, S. Ma, Z. Qian, and J. Li, "Annotating, tracking, and protecting cryptographic secrets with cryptompk," in *IEEE Symposium on Security and Privacy (S&P)*, 2022.

[85] I. Bang, M. Kayondo, H. Moon, and Y. Paek, "Trust: A compilation framework for in-process isolation to protect safe rust against untrusted code," in *USENIX Security Symposium*, 2023.

[86] W. Blair, W. Robertson, and M. Egele, "Mpkalloc: Efficient heap meta-data integrity through hardware memory protection keys," in *Conference on Detection of Intrusions and Malware & Vulnerability Assessment*

(DIMVA), 2022.

[87] ——, "Threadlock: Native principal isolation through memory protection keys," in *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2023.

[88] E. Rivera, S. Mergendahl, H. Shrobe, H. Okhravi, and N. Burow, "Keeping safe rust safe with galeed," in *Annual Computer Security Applications Conference (ACSAC)*, 2021.

[89] H. Lei, Z. Zhang, S. Zhang, P. Jiang, Z. Zhong, N. He, D. Li, Y. Guo, and X. Chen, "Put your memory in order: Efficient domain-based memory isolation for wasm applications," in *ACM Conference on Computer and Communications Security (CCS)*, 2023.

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

[91] L. Maar, M. Schwarzl, F. Rauscher, D. Gruss, and S. Mangard, "Dope: Domain protection enforcement with pks," in *Annual Computer Security Applications Conference (ACSAC)*, 2023.

[92] J. Corbet, "Intel memory protection keys," https://lwn.net/Articles/643797/, 2015.

[93] R. Rudd, R. Skowyra, D. Bigelow, V. Dedhia, T. Hobson, S. Crane, C. Liebchen, P. Larsen, L. Davi, M. Franz *et al.*, "Address oblivious code reuse: On the effectiveness of leakage resilient diversity." in *Symposium on Network and Distributed System Security (NDSS)*, 2017.

[94] B. Chamith, B. J. Svensson, L. Dalessandro, and R. R. Newton, "Instruction punning: Lightweight instrumentation for x86-64," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*

[100] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser, "N-variant systems: A secretless framework for security through diversity." in *USENIX Security Symposium*, 2006.

(PLDI), 2017.

[95] "LD_PRELOAD," Last accessed 2023. [Online]. Available: https://man7.org/linux/man-pages/man8/ld.so.8.html

[96] "falco," Last accessed 2023. [Online]. Available: https://falco.org/

[97] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis, "Dune: Safe user-level access to privileged CPU features," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[98] A. Voulimeneas, D. Song, P. Larsen, M. Franz, and S. Volckaert, "dmvx: Secure and efficient multi-variant execution in a distributed setting," in *European Workshop on System Security (EuroSec)*, 2021.

[99] M. Xu, K. Lu, T. Kim, and W. Lee, "Bunshin: compositing security mechanisms through diversification," in *USENIX Annual Technical Conference*, 2017.

[101] S. Volckaert, B. Coppens, B. De Sutter, K. De Bosschere, P. Larsen, and M. Franz, "Taming parallelism in a multi-variant execution environment," in *European Conference on Computer Systems (EuroSys)*, 2017.

[102] K. Lu, M. Xu, C. Song, T. Kim, and W. Lee, "Stopping memory disclosures via diversification and replicated execution," *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2018.

[103] S. Österlund, K. Koning, P. Olivier, A. Barbalace, H. Bos, and C. Giuffrida, "kMVX: Detecting kernel information leaks with multi-variant execution," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[104] X. Wang, S. Yeoh, R. Lyerly, P. Olivier, S.-H. Kim, and B. Ravindran, "A framework for software diversification with ISA heterogeneity," in *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2020.