Privbox: Faster System Calls Through Sandboxed Privileged Execution

Dmitry Kuznetsov and Adam Morrison, *Tel Aviv University*

https://www.usenix.org/conference/atc22/presentation/kuznetsov

This paper is included in the Proceedings of the 2022 USENIX Annual Technical Conference.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-29-8

Open access to the Proceedings of the 2022 USENIX Annual Technical Conference is sponsored by

Privbox: Faster System Calls Through Sandboxed Privileged Execution

Dmitry Kuznetsov
Tel Aviv University

Adam Morrison
Tel Aviv University

Abstract

System calls are the main method for applications to request services from the operating system, but their invocation incurs considerable overhead, which has been aggravated by mitigation mechanisms for transient execution attacks. Proposed approaches for reducing system call overhead all break the semantic equivalence between system calls and regular function calls (e.g., by making system calls asynchronous), and so their adoption requires rearchitecting applications.

This paper proposes *Privbox*, a new approach for lightweight system calls that maintains the familiar synchronous, function-like system call model. Privbox allows an application to execute system call-intensive code in a *semi-privileged, sandboxed* execution mode, called a "privbox". Semi-privileged execution is architecturally similar to the kernel's privileged execution, which enables faster invocation of system calls, but the code is sandboxed to ensure that it cannot use its elevated privileges to compromise the system. We further propose *semi-privileged access prevention* (SPAP), a simple hardware architectural feature that alleviates much of Privbox's instrumentation overhead.

We implement Privbox based on Linux and LLVM. Our evaluation on x86 (Intel Skylake) hardware shows that Privbox (1) speeds up system call invocation by 2.2×; (2) can increase throughput of I/O-threaded applications by up to 1.7×; and (3) can increase the throughput of real-world workloads such as Redis by up to 7.6% and 11%, without and with SPAP, respectively.

1 Introduction

System calls are the de-facto method for processes to request services from the operating system (OS), but they are orders of magnitude slower than a regular function call. Much of the overhead stems from switching the processor's execution mode between unprivileged user-mode and privileged kernel execution on system call entry and exit [1]. User-to-kernel mode switches are further slowed down by the protection mechanisms [2, 3] recently added to mitigate transient execution vulnerabilities such as Meltdown [4] and Spectre [5].

Reducing system call overhead has attracted significant research attention over the years (e.g., [6, 1, 7, 8, 9, 10]), and the increased overhead imposed by the mitigations of transient execution vulnerabilities [10] underscores the importance of addressing the problem. Current approaches, however, *break the semantic equivalence between system calls and regular function calls*. For instance, FlexSC [1] and `io_uring` [8] make system calls asynchronous; the `io_uring` model and similar models [6, 11, 12] also limit how system calls can be composed. Consequently, benefitting from these system call designs requires rearchitecting applications to use the new system call models.

In this work, we propose *Privbox*: a new approach for lightweight system calls that maintains the familiar user-space programming model of synchronous, function-like system calls. In our design, an application can demarcate system call-intensive code and have it execute in a *privbox*, in which system call invocation is cheap—e.g., 2.2× faster than a regular system call on an Intel Skylake CPU. An application can thus enjoy low-overhead system calls with an unchanged synchronous system call model and only minor source code modifications to demarcate privboxed code regions.

Privboxed code runs in a "semi-privileged" mode. Semi-privileged execution consists of the processor running in privileged mode with the kernel address space mapped, which reduces user/kernel system call transition time, similarly to kernel-mode Linux (KML) [7]. But unlike KML, semi-privileged privboxed code runs *sandboxed*, so that it has the same access as the regular, unprivileged code of its process—thus it cannot violate OS security.

Our sandbox design is inspired by the software fault isolation approach of NaCl [13, 14], which uses compile-time instrumentation to generate verifiably safe code. We adapt NaCl's instrumentation approach to the circumstances and environment of semi-privileged privboxed execution. In our design, source code demarcated for privboxed execution is instrumented at compile-time to prevent it from reading/writing arbitrary kernel memory or jumping to arbitrary kernel code. The kernel verifies the correctness of the instrumentation before allowing the code to begin its privboxed execution. The privboxed code then runs natively, without any runtime environment, but under a custom page table configuration that blocks it from executing uninstrumented user-mode code.

Unfortunately, the sandbox's instrumentation slows down execution of the privboxed code, which reduces the benefit from faster system calls. We identify instrumentation of memory operations (load and store instructions) as the main culprit. Motivated by this finding, we propose *SPAP* (semi-privileged access prevention), a simple hardware architectural modification that enables omitting load/store instrumentation

from privboxed code. With SPAP, privileged mode hardware execution blocks an instruction from reading/writing a kernel address if that instruction resides in a user-mode address (as privboxed instructions do). SPAP's check can be implemented analogously to how x86-64 implements its supervisor mode access prevention (SMAP) [15] feature, which blocks privileged mode execution from accessing user-mode addresses.

We implement Privbox for x86-64 by adding support for semi-privileged execution to Linux and the musl standard C library [16], as well as extending LLVM to support generation of instrumented (sandboxed) code. We evaluate Privbox in two contexts:

1. Applications that use patterns such as I/O threads and reactors[17]. These patterns are system call intensive with little user-space logic, and are therefore less impacted by instrumentation overhead. We find that such applications can gain over 1.7× speedup, even without SPAP.
2. Applications that combine I/O and user-space logic. Specifically, we modify Redis [18], memcached [19], and SQLite [20] to use Privbox. For simplicity, we compile the entire application with instrumentation, which gives us a lower bound on Privbox's benefit. Using Privbox in these applications requires little effort (20–30 lines of code) and yields an improvement of up to 7.6% on today's hardware without SPAP and up to 11% in a configuration that approximates performance under SPAP.

Contributions We make the following contributions:

- **Privbox design (§ 3).** We design Privbox, a new systems programming mechanism for lightweight system. Privbox maintains the familiar user-space system call model and its adoption requires no application source code changes.
- **Implementation (§ 4).** We implement support for Privbox for x86-64 in Linux, the musl C library, and LLVM.
- **SPAP (§ 5).** We propose semi-privileged access prevention, a simple hardware modification that enables omitting load/store instrumentation from privboxed code.
- **Evaluation (§ 7).** We show that Privbox improves (1) system call overhead by 2.2×; (2) I/O thread execution time by over 1.7×; and (3) real-world workload throughput by as much as 7.6% without SPAP and 11% with SPAP.
- **Availability.** The Privbox implementation and benchmarks are available at `https://github.com/privbox`.

2 Background & motivation

Monolithic kernels like Linux rely on hardware *privilege modes* to enforce process isolation and to mediate I/O access to peripheral devices. The kernel offers a set of *system calls* for processes to request services requiring OS mediation, such as input/output (I/O) to devices, inter-process communication, and virtual address space modification. System calls are semantically equivalent to function calls, but are orders of magnitude slower. This problem has spurred research on reducing their overhead. These proposals, however, break the semantic equivalence between system and function calls, so adopting them requires rearchitecting applications to use a new system call model. Our goal is thus to address the problem without changing the system call programming model.

Privilege modes The basic hardware mechanism used to implement the OS isolation model is the distinction between unprivileged and privileged processor execution modes. The kernel runs in *privileged* mode, which allows full access to the entire instruction set, including *privileged instructions* for, e.g., installing a page table or enabling/disabling interrupts. Applications run in *unprivileged* mode, which only allows execution of non-privileged instructions that cannot circumvent OS isolation. Implementation of the execution privilege modes differs between hardware architectures. In this paper, we focus on the x86-64 architecture. It defines four hardware privilege levels, also called *rings*, numbered 0 to 3 in decreasing order of privilege. Most OSes execute user applications in ring 3 and the kernel in ring 0. The CPU has a *code segment* (CS) register that (indirectly) defines the CPU's current ring, also called *current privilege level* (CPL).

System calls A *system call* is a mechanism for a controlled and safe transfer of execution from untrusted unprivileged code to privileged kernel code. On x86-64, system calls are implemented by the `syscall` instruction. This instruction elevates the CPU's privilege mode and transfers control to a pre-determined kernel memory address, called the system call entry point. The entry point code determines the kernel function to service the desired call by inspecting certain CPU registers (determined by a software, OS-specific convention), executes it, and finally executes a "return to user" instruction which lowers the CPU's privilege mode and transfers control back to the unprivileged code.

System call overhead A system call is semantically equivalent to a function call from an application's perspective. But it is orders of magnitude slower, as its invocation/return is a multi-step procedure in both hardware and software, in which hardware elevates/lowers its privilege level and saves/restores certain CPU state, and kernel code saves/restores remaining CPU state and determines and executes the system call code.

System call overhead is exacerbated by mitigations of hardware microarchitecture vulnerabilities, such as Meltdown [4] and Spectre [5]. These vulnerabilities involve malicious user code abusing microarchitectural state shared by the CPU's privilege modes to read memory that is not architecturally accessible to the attacking user code. Mitigation accordingly involves modifying the relevant CPU state on system call entry, which adds overhead. For instance, on affected hardware, Linux's system call entry code flushes the CPU indirect branch predictor's state [3] to block Spectre v2 attacks. In addition, Linux's page table isolation (PTI) [2] Meltdown mit-

igation switches page tables during system call entry, which is a costly operation that also implies a TLB flush on x86-64. Indeed, our evaluation (§ 7.1) finds that while a standard system call invocation entry/exit time is 28× slower than a function call/return, PTI makes it 52× slower. And while Linux's software mitigations are not used on recent processors that mitigate the vulnerabilities in hardware, the hardware mitigation itself slows down the system call instruction [10].

Reducing system call overhead There are several proposals to reduce the overhead of system call entry/exit. They generally achieve this by compromising on the semantic equivalence between system and function calls.

Flexible system calls (FlexSC) [1] makes system calls asynchronous instead of synchronous. It offloads system call execution to a kernel "syscall thread" associated with the process, with which the process communicates over a shared-memory interface, thereby eliminating CPU cycles spent on system call entry/exit. However, FlexSC requires *rearchitecting applications to use an asynchronous programming style*.[1] It also increases CPU usage due to the added threads and polling of the shared-memory communication structures.

System call overhead can be reduced by batching. The *multi-call* approach invokes several system calls with one kernel entry/exit. Linux includes several multi-calls, such as `preadv`, which performs a sequence of seek-followed-by-read operations. Cassyopia [6] explores compiler optimizations to batch several system calls together. But since a multi-call specifies the participating calls up front, it *does not support arbitrary composition of system calls and user-space logic offered by the standard system call model*.

Recent Linux versions offer an `io_uring` [8] mechanism. `io_uring` allows submitting I/O requests through a memory interface, like FlexSC, but it does not support arbitrary system calls. Like multi-calls, multiple requests can be submitted to a submission queue, but the submitted operations can be of different kinds and interact with different file descriptors. Overall, `io_uring` can be viewed as a combination of FlexSC and batching specialized for I/O, and thus suffers from the same limitations as those approaches.

Kernel bypasses avoid system call overhead by doing away with system calls for device access. For instance, DPDK [9] and SPDK [21] allow applications to interact directly with networking and storage devices, respectively. But since the kernel no longer mediates device access, its standard interfaces such as sockets or files cannot be used, and user-space has to implement all the abstractions it requires.

Ward [10] targets overhead related to Spectre and Meltdown mitigations. It constructs process page tables which do contain mappings of kernel memory, but only memory that is safe to expose to that process. At best, Ward reduces system call overhead to that of the pre-Spectre/Meltdown baseline, which is still significantly slower than a function call.

BPF for Storage [22] is a recent approach for reducing I/O path overhead by leveraging Linux's eBPF subsystem [23], which is an in-kernel virtual machine that can execute user-loaded bytecode programs. The idea in BPF for Storage is to use eBPF programs to bypass kernel layers and avoid system calls, e.g., by searching an on-disk B+tree inside the kernel instead of via multiple system calls. However, *eBPF is a severely limiting programming model*, as the bytecode must pass static verification [24] before it can be executed in the kernel, and verification considerations limit eBPF programs to be small and to have provably bounded memory and execution time.

3 Design

Privbox is a new execution model for system call intensive code. Privbox provides standard synchronous, function-like system calls but with significantly lower invocation cost. Using Privbox thus requires no rearchitecting of application source code, as opposed to, e.g., the asynchronous system calls provided by FlexSC or `io_uring` (see § 2). We describe Privbox in the context of Linux on x86-64 hardware, but its design can be extended to other monolithic operating systems and/or hardware architectures.

Privbox provides an interface for executing code sections (e.g., ELF objects) in a *semi-privileged* execution mode (§ 3.1); we refer to such code as being *privboxed*. In semi-privileged execution, the processor is in privileged mode and kernel memory is mapped, which enables fast system call execution (§ 3.2), but for security, the code runs sandboxed, so that it has the same access as the process that loads it (§ 3.3). This sandbox is enforced by compile-time instrumentation (verified by the kernel) and by virtual memory restrictions.

Privbox and eBPF (§ 2) share some similarity in that both offer safe execution of user-supplied code in privileged processor mode, but the designs have a fundamental difference. eBPF runs code *in kernel context*, invoked to handle certain events, and so eBPF programs must be verified to terminate and be provided with interfaces for kernel operations. In contrast, privboxed code is *conceptually process code, just with faster system calls*. It can access the process' address space, be scheduled and context switched, and can only interact with the kernel via the system call interface—in particular, it invokes the kernel and not vice versa. Privbox is thus a general-purpose design, whereas eBPF requires customization for each new use case (e.g., [22]).

While we focus on the general use case, in which privboxed code must be isolated from other processes in the system, some scenarios can use Privbox without sandboxing. One such example is a workload running by itself on a dedicated virtual machine. This use case can employ Privbox without sandboxing, as the privboxed code cannot compromise any-

[1] FlexSC offers a threading library that makes its asynchronous system calls transparent to applications, but this library is relevant only for applications with many user-mode threads.

```
main:
  fd = open("priv.so")
  call priv_code_load(fd)
  call priv_code_invoke("privfunc1")
```

```
privfunc1:
  loop:
    fast_syscall ...
  call priv_code_return()
```

Listing 1: Example usage of privcall mechanism.

thing other than itself.

3.1 Execution & usage model

Privbox introduces an operating system interface with the following system calls:

- *Load privboxed code.* This method can accept a pointer to a buffer with machine instructions or a file descriptor of an ELF object file. Internally, it verifies the code's safety (§ 3.3) and relocates it to a dedicated, immutable memory region from which it will be executed.
- *Invoke privboxed code.* This method accepts a pointer or symbol from the loaded ELF file, and begins executing it in semi-privileged mode. I.e., from the calling application's perspective, this method returns only when the privboxed code returns, as explained next.
- *Return from privboxed code.* This method can only be called by privboxed code. It exits semi-privileged execution and transfers control to the code that invoked it, while making a return value available.

Listing 1 shows an example program utilizing the above interface to invoke a function from the ELF object `priv.so`.

Usage model While an entire application can be privboxed, Privbox's sandboxing instrumentation imposes overhead, so only code sections with a large fraction of cycles spent on system call entry/exit will benefit from privboxing. Privbox adoption thus consists of (1) the developer identifying system call intensive code sections for privboxing; (2) isolating such code sections into separate build units, which are built with the required instrumentation (e.g. into ELF objects); and (3) modifying application source code to load and invoke these objects as privboxed code at run time. Crucially, a privboxed code section itself requires no modification: it is simply another object file linked or loaded into the application.

We envision steps (2)–(3) being performed by tools, after developers demarcate privboxed code sections in the source code. In this paper, however, we perform them manually. For manual Privbox adoption, the "low hanging fruits" consist of applications whose software architecture already separates system call intensive code from computation code into distinct modules that communicate via some mechanism (e.g., SEDA [25]). Figure 1 depicts such an architecture. Because these architectures already isolate I/O (or other system call-heavy) code from other parts of the code, it is straightforward to surgically apply instrumentation only to that code. This approach minimizes Privbox's instrumentation overhead, as compute heavy parts remain unaffected, while the I/O parts waste less cycles on system call entry/exit.

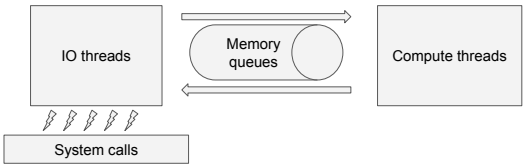

Figure 1: Software architecture that separates threads performing I/O (or system calls) from threads that perform other kinds of logic.

3.2 Semi-privileged execution

Privboxed code runs with the processor in privileged mode and kernel memory mapped, which enables it to perform a system call with a function call, without the `syscall` instruction, as detailed below. Except for having the ability for fast system calls, the OS treats privboxed code as user-space (process) code and its access is similarly restricted, hence the term *semi-privileged execution*. When privboxed code is invoked, the kernel transfers control to it with a new (ring 0) code segment (CS). Other than having a different CS value, the privboxed code runs similarly to unprivileged code—with interrupts enabled and the same priority, capabilities, and permissions. The kernel can preempt its execution at any moment and re-schedule it, as it would any other process.

Privboxed code runs with a custom page table, which modifies the standard virtual address space layout in several ways. Figure 2 shows the baseline layout of user and kernel memory in Linux. Privbox adds a special *privboxed code region* to the user part of the address space. This region is located in the lower part of the process address space and is unwritable by the process. After the kernel successfully verifies privboxed code, it relocates the code to the privboxed code region, from which it is later executed.

Normally, kernel mode execution has both kernel and user addresses mapped and accessible, and user mode execution only has user addresses accessible.[2] Privbox's custom page table maps the same memory regions as the process' page table, but using different access modes: user memory (excluding the privboxed code region) is marked not executable (see § 3.3) and kernel memory is marked accessible and executable, unlike user mode execution. (Despite kernel memory being

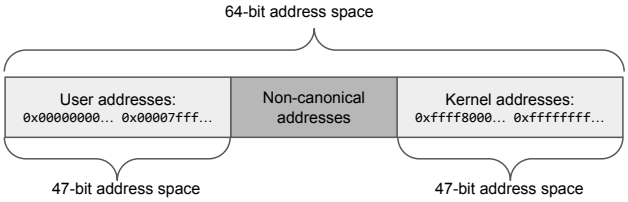

Figure 2: Example of memory map on x86-64 with 48-bit virtual address space. Lower address spaces are reserved for user program memory, higher address space are reserved for kernel memory. Access to memory at non-canonical addresses generates a trap.

[2] Without PTI, kernel addresses are still mapped, but their page table entries have a "supervisor" bit set, which allows access to the pages only when CPL < 3 (i.e., privileged mode). With PTI, kernel addresses are not mapped at all.

mapped accessible, privboxed code cannot directly access it due to the sandboxing instrumentation (§ 3.3).) Privbox's custom page table is implemented by switching the top-level page table node to one in which the page entries (PTEs) mapping the user-space and kernel address ranges have the appropriate permissions, i.e., Privbox does not create nor maintain a copy of the process' entire page table.

The execution environment of privboxed code enables a fast implementation of system calls. Privboxed code does not invoke a system call using the `syscall` instruction; it invokes a system call using a standard function call. Privbox provides a new kernel function that serves as a *gate* for system call invocation. This function has similar semantics to system call entry code: it sets up kernel execution environment (switching to kernel stack, storing registers) and routes requests to system call functions based on the system call number provided by calling code. Crucially, calls to this special function are not blocked by the sandbox instrumentation, so privboxed code can branch to it to perform a system call.

Modern x86-64 processors support supervisor mode access/execution prevention (SMAP [15]/SMEP), which disallow privileged execution from accessing/executing pages that are marked as user-space pages in the page table. Because privboxed code runs in "supervisor mode" but needs to execute and access user memory, SMAP/SMEP is disabled during privboxed execution. When privboxed code enters the kernel (e.g., a system call or interrupt), SMAP is re-enabled (this costs $\approx$ 26 cycles) but SMEP remains disabled (because toggling it costs thousands of cycles). § 6 discusses the security implications of this limitation.

A process that starts semi-privileged execution stays in that mode until it either invokes the "return from privboxed code" system call or an OS event that would result in non-sandboxed code execution occurs, such as invocation of an `exec` system call to load a new binary into the process address space. In such cases, the OS terminates the process' semi-privileged execution, moving it to standard user-mode execution.

3.3 Sandboxing semi-privileged execution

Privbox must guarantee that a semi-privileged execution (of privboxed code) cannot perform any operation or memory access that the regular unprivileged execution of its process cannot perform. Importantly, this guarantee must also hold for transient execution of privboxed code (e.g., due to indirect branch misprediction) [4, 5]. This section describes our design for enforcing this safety property, which combines run-time virtual memory restrictions and compile-time instrumentation whose safety is verified by the kernel. § 6 analyzes the security of our design.

3.3.1 Sandboxing techniques

Three types of machine instructions can violate our desired safety property: (1) memory loads and stores, which have potential to access kernel memory; (2) control flow instructions, which can be used to branch into non-instrumented code; and (3) privileged instructions, usually reserved to operating system code. In the following, we describe how Privbox mitigates each of these risks.

Execution mode: $\Rightarrow$ Accessed memory $\Downarrow$	User mode (CPL=3)	Privbox mode (CPL=0)	Kernel mode (CPL=0)
Kernel memory	$N^{\dagger}/N^{\dagger}$	$N^{\ddagger}/N^{\ddagger}$	Y/Y
Privbox code	Y/Y	Y/Y	Y^{*}/Y^{*}
User memory	Y/Y	$N^{\dagger}/Y$	Y^{*}/Y^{*}

† Restricted through page table access controls.
‡ Restricted through instrumentation.
$*$ Subject to SMAP/SMEP.

Table 1: Access to memory regions under different execution modes. Left symbol: instruction fetching, right symbol: data load/store.

Memory access and control-flow Privbox uses a combination of compile-time instrumentation and virtual memory (page table) protection to protect memory accesses. Table 1 summarizes which types of memory (rows) each type of execution mode (column) is allowed to access (for code execution/data), and how these restrictions are enforced.

We prevent privboxed code from executing user-space code outside of the (verified) privboxed code region, taking advantage of the "NX bit" feature of x86-64. On x86-64, each page table entry (PTE) has a No-Execute (NX) bit. When the NX bit is set, fetching instructions (code execution) is not permitted from the page(s) mapped by the PTE. Thus, the custom page table installed for Privbox's semi-privileged execution maps user-space addresses outside the privboxed code region as non-executable.

We use compile-time instrumentation to ensure all other types of memory accesses are safe. Indirect load/store instructions (where the operand is known only at run time) are instrumented by introducing instructions that "sanitize" the memory operand, ensuring it does not point to kernel memory. For Linux on x86-64, we use the fact that clearing the most significant bit of a virtual address is guaranteed to create either a user address or an illegal non-canonical address, whose access will generate an exception (Figure 2).

Indirect control-flow instructions, such as calls, jumps, and returns are similarly instrumented, to ensure that privboxed code does not branch to arbitrary kernel addresses, because the kernel's address space is mapped as executable. Our control-flow instrumentation also ensures that control-flow instructions can only branch to addresses which are verified when the code is loaded. Therefore, at run time, privboxed code can execute only instructions that were checked by the verifier.

Overall, our instrumentation approach guarantees that instruction memory operands are never kernel addresses, even for transient instructions, and thus blocks both non-transient and transient execution attacks (see § 6).

Privileged instructions We rely on Privbox's verifier to check that loaded code does not contain privileged instruc-

tions. Rejecting code with privileged instructions does not limit Privbox's applicability, as compilers do not emit such instructions unless specifically instructed.

3.3.2 Verification

The kernel verifies that privboxed code is correctly instrumented before allowing it to execute. Verification faces a challenge due to x86-64's variable-length instructions, which mean that decoding the same code at different offsets can yield completely different instruction sequences. It is therefore possible that a seemingly benign instruction sequence would include malicious code at certain offsets [13].

To address this problem, our design relies on code being packed into *chunks*, which enables sound disassembly and verification of any possible execution of the loaded code. We define a code *chunk* as an aligned and fixed-length byte sequence containing machine instructions that always executes from its first to last instruction (i.e., a small fixed-size basic block). For example, 32-byte code chunks are expected to be 32-byte aligned in memory and be exactly 32 bytes long. The compiler breaks long basic blocks into chunks, using no-op instructions to pad the created chunks to their desired fixed length. The compiler adds "sanitizing" instructions which align targets of control-flow instructions (including returns) to guarantee that they branch to the beginning of a chunk, i.e., to a chunk-aligned address. (See § 4.2.2 for details.)

The verifier deems the loaded code safe by verifying each code chunk individually. For each chunk, the verifier verifies that if the chunk contains a control-flow instruction, it is the last one, and that it is sanitized as described above. This ensures verified code can only branch to verified code, keeping execution inside the sandbox. In addition, the verifier checks that memory operations are preceded, within the same chunk, by the instructions sanitizing their address operands and that the chunk does not contain privileged instructions.

3.3.3 Discussion: Privbox vs. NaCl

Our sandbox design is inspired by Native Client (NaCl) [13, 14]. NaCl enables safe execution of native code downloaded from a web site inside a web browser at near native speed. The browser verifies loaded programs and makes sure the loaded code does not write to, or jump, outside of a sandboxed region. The sandboxed code is loaded into a memory region that is gapped by unmapped memory regions on both ends. To ensure sandboxed code does not write or execute memory outside of the sandbox, NaCl relies on "offset-from-known-base" operations. A *base pointer register* (immutable by sandboxed code) points at this memory region. A memory access is allowed only with an offset from base pointer register, which results in accesses always being either: (1) inside the allowed memory region; or (2) in the unmapped memory areas.

Privbox's instrumentation shares some similarities with NaCl: (1) execution is limited to code running inside the sandbox; (2) memory stores have to be instrumented (because kernel memory is accessible); and (3) instructions have to be aligned in specific manner. In contrast to NaCl, however, Privbox (1) has no need for a base pointer register—we use absolute addresses, appropriately sanitized; (2) instruments memory loads as well, because kernel memory is readable; and (3) must avoid privileged instructions.

4 Implementation

This section describes our prototype implementation of the Privbox design in Linux. § 4.1 describes OS and library modifications and § 4.2 describes the sandboxing compiler. We do not implement the verifier part of the design (§ 3.3.2), as it is not required for evaluating performance under Privbox.

4.1 OS & library support

The following describe various parts of our implementation and their size in lines of code (LOC).

Semi-privileged execution (750 LOC) To implement semi-privileged execution, we apply the techniques of kernel-mode Linux (KML [7]) to Linux v5.8 on x86-64. KML is an existing kernel patch to support execution of an entire application in kernel mode. It is based on Linux v4.0 (circa 2015), and does not isolate its in-kernel processes from each other or the kernel from them.

Linux v5.8 on x86-64 uses "legacy stack switching," where the stack is switched by the hardware only on a CPL change (i.e., an interrupt while user code is executing). However, Privbox's semi-privileged execution has CPL $= 0$ but with a user stack. This means that an interrupt received during semi-privileged execution would cause the interrupt handler to run with the user's stack, which is problematic because: (1) the user stack is accessible to user code, which might hijack execution by modifying the stack frame (from another thread); (2) writing to the stack might fault (e.g., if the stack pointer points to an unmapped page), but the page fault would not change the stack either, crashing the system; and (3) the user's stack includes a red zone [26] that must not be written to. We therefore adjust our Linux version to use x86-64's *interrupt stack table* (IST) for all interrupts and exceptions. With IST, each exception/interrupt/trap can be configured to switch to a specific stack.

Call gate (80 LOC) Privbox exposes a system call gate (§ 3.2), which is a kernel function that serves as the system call entry point for privboxed code. The gate follows Linux's `syscall` conventions for passing the system call number and parameters. It is similar to the standard system call entry code but avoids performing unnecessary steps, such as modification of page tables and toggling of interrupts. In particular, Linux's entry code (with PTI) assumes it is called from user-space and thus unconditionally switches the page

table from the user-space to the kernel page table. This is unnecessary for privboxed code, which already has kernel memory mapped in its custom page table.

Limitations For implementation simplicity, we inhibit receipt of signals during semi-privileged execution. This is not a design limitation, and there are several designs for supporting signals: (1) ensuring that the signal handler code points to verified and safe code; or (2) aborting privileged execution (i.e., having it return to the code that launched it with an `EINTR` indication). Importantly, privboxed code can still receive signals in our prototype using the `signalfd` [27] mechanism.

Library support (260 LOC) Applications usually invoke system calls through a C library function, which then invokes the `syscall` instruction. We thus create a modified standard C library, based on the musl C library [16], in which the library's system call wrappers use `syscall` or Privbox's system call gate based on the execution's CPL. The entire library is compiled with Privbox's instrumentation, so that privboxed code objects can be linked with it.

In this paper, we modify an application to use Privbox by changing its build environment to link privboxed code with the above C library. The reason is that current compilers and linkers do not support linking an entire application with both the system's C library and our modified, instrumented C library, as both export the same symbols and the tools cannot resolve which library version the application code refers to. This problem can be solved by adding compiler annotations for demarcating privboxed code; we leave this to future work.

4.2 Code instrumentation

We implement Privbox's sandboxing instrumentation by introducing a machine function pass and several other changes to the x86-64 backend of the LLVM toolchain [28], which consist of 1200 LOC. Our modified LLVM emits machine code in which unsafe instructions are replaced with equivalent but safe instruction sequences (§ 3.3).

Instrumented code is partitioned into fixed-size chunks (§ 3.3.2). Our implementation uses 32-byte chunks. The reason is that an x86-64 instruction can be up to 15 bytes long, so a 32-byte chunk can fit at least an instruction of the privboxed code plus the added instrumentation instructions that make it safe. (This is the worst case; most chunks contain more than one instruction.)

When instrumenting an instruction, it is placed in its own chunk, preceded with the instrumentation instructions, which is achieved by emitting an alignment directive in the code (`.align`). This ensures any instrumentation sequence starts at beginning of a new chunk.

4.2.1 Load/store instrumentation

Loads/stores are non-branching instructions that access memory. Their address operand is either static, verifiable at load time, or dynamic, derived from values of registers. Dynamic values cannot be verified at load time, so instrumentation is required to ensure kernel memory is not accessed. Our instrumentation "sanitizes" operands by clearing their most significant bit (MSB), which ensures it does not point to kernel memory (§ 3.3).

On x86-64, memory operands are based on four elements: scale, index, base and displacement. Scale and displacement are scalars while index and base are registers. The effective address of a memory operand is calculated by: $Displacement + Base + Scale * Index$. Either the base or index registers can be omitted and are calculated as zero in such case. Scale can be 1, 2, 4 or 8. Displacement can be either 1, 2, 4 or 8 bytes long.

The memory operand of an instruction I is sanitized by the prefixing I with the following instruction sequence: (1) computing I's effective address with a load-effective-address instruction (`lea`); (2) clearing its MSB with a bit-test-and-reset instruction (`btr`); and (3) replacing I's original memory operand with one dereferencing the sanitized value. Listing 2 shows this sequence. The `btr` instruction has a side-effect of updating the x86-64 `EFLAGS` register. Our compiler code therefore checks if the `EFLAGS` register has meaningful state at the point of instrumentation, and if so, emits `SAVE_EFLAGS` and `RESTORE_EFLAGS` around the instrumentation sequence. These are abstract operations implemented by LLVM and translated to instructions such as `sahf`/`lahf` (save/load flags).

A shorter instrumentation sequence is used for memory operands that specify (1) only one of base or index registers; and (2) 1/2/4-byte displacement. In this case, the effective address is sanitized with a single `btr` instruction (and `EFLAGS` save/restore, if needed). Appendix A.1 provides the details.

Similarly to NaCl, we avoid instrumentation of stack loads/stores by maintaining and verifying invariants on manipulations of the stack pointer, which guarantee that stack accesses always target user memory. Appendix A.2 elaborates on handling of stack accesses. This approach greatly reduces the emitted instrumentation, as stack accesses are very common.

4.2.2 Control-flow instrumentation

Control-flow instructions are instructions that can modify the instruction pointer (beyond advancing it to next instruction). As with load/store instructions, we are concerned only with instructions whose operand is unknown at load time. Control-flow instructions can compromise safety of semi-privileged execution by branching to (1) arbitrary kernel code or (2) privboxed code at the middle of a chunk. The latter is dangerous because the verifier verifies code starting at chunk boundaries.

Similarly to load/store instrumentation, kernel addresses are avoided by clearing the MSB of branch targets. Chunk-unaligned addresses are avoided by clearing the low 5 bits of the target. While this allows branching to any chunk-aligned

```
.align CHUNK_SIZE
SAVE_EFLAGS
%Reg1 = lea disp(%Idx,scale,%Base)
%Reg2 = btr $63, %Reg1
RESTORE_EFLAGS
OP operand1, (%Reg2)
```

Listing 2: Instrumentation of load/store instruction with memory operand.

```
.align CHUNK_SIZE
%Reg1 = lea *disp(%Idx,scale,%Base)
%Reg2 = btr $63, %Reg1
%Reg3 = mov *%Reg2
%Reg4 = btr $63, %Reg3
%Reg5 = and $~(CHUNK_SIZE - 1), %Reg4
call *%Reg5
.align CHUNK_SIZE
```

Listing 3: Instrumentation of memory-operand call.

```
.align CHUNK_SIZE
pop %rcx
add $(CHUNK_SIZE - 1), %rcx
and $~(CHUNKS_SIZE - 1), %rcx
btr $63, %rcx
jmp *%rcx
```

Listing 4: Instrumentation of return instruction.

```
.align CHUNK_SIZE
%Reg1 = btr $63, %Reg
%Reg2 = and $~(CHUNK_SIZE - 1), %Reg1
call *%Reg2
.align CHUNK_SIZE
```

Listing 5: Instrumentation of register-operand call.

address in user memory, only the privileged code section is mapped executable in the privboxed code's page table (§ 3.3).

Return instrumentation A return is equivalent to popping an address from stack and jumping to it. To ensure a valid destination, we replace each return instruction with an equivalent but safe sequence that pops the address into a register, clears its MSB, aligns it to the next 32 bytes, and jumps to the obtained value. Listing 4 details this instrumentation sequence. Linux's calling convention specifies that the `RCX` register is not preserved on calls, so we explicitly use it to store the return address. The `add` (addition) and `and` (logical AND) instructions are used to align-up the value in `RCX` to 32 bytes (start of next code chunk). The `btr` (bit-test-and-reset) clears the MSB.

This instrumentation ensures branching is possible only to code chunk aligned, non-kernel addresses. While in theory it is possible to hijack execution by overwriting the return address (e.g., by buffer overflow), the effects of such hijacking are very limited. Any address popped from the stack is guaranteed to be sanitized before use, so an attacker can only redirect execution to valid and verified code inside the Privbox code region or to non-executable memory (either user addresses or non-canonical). The former one does not pose a threat as privboxed code is verified as safe, and the latter causes a fault, effectively stopping the execution.

Call/jump instrumentation Call/jump instructions with an indirect destination (i.e., non embedded as an instruction-relative offset) can have their destination stored in one of two ways: (1) in a register operand and (2) in a memory operand.

Register operands are sanitized similarly to a return (sans stack pop), as shown in Listing 5. Since a return aligns addresses before branching, an alignment directive is required right after a call to push the next instruction to a chunk boundary. The `EFLAGS` register has to be preserved only in case of jumps, as the calling convention states that it is not preserved across calls.

A call/jump with a memory operand is equivalent to a memory-to-register load followed by a register operand call. Sanitization is this performed analogously: (1) the memory operand is sanitized with load/store instrumentation; (2) a `mov` instruction is used to load the address into a register; and (3) the loaded address is sanitized as a register operand call. Listing 3 shows the generated instruction sequence.

Jumps are similar to calls, except that they have to preserve the `EFLAGS` register and aligning the succeeding instruction is unnecessary, as jumps do not return. Appendix A.3 provides the details.

4.2.3 Code alignment

The instrumentation described in the previous sections deals with unsafe instructions, co-locating instrumentation sequences within same code chunks, and aligning return sites of calls. The compiler also makes sure that all branch destinations (functions, basic blocks) are aligned to the chunk boundary, because indirect branches target addressed with 5 lowest bits cleared. It additionally inserts no-op instructions before any instruction that would otherwise cross code chunk boundary, so that it moves to a chunk-aligned address. Combined, these rules partition the emitted code instructions into fixed-size chunks.

5 SPAP: Hardware support for reducing instrumentation overhead

Our analysis of Privbox's performance (§ 7.3) shows that load/store instrumentation is responsible for a considerable part of instrumentation overhead. To address this problem, we propose *semi- privileged access prevention* (SPAP), a simple hardware architectural modification that enables omitting load/store instrumentation from privboxed code.

SPAP SPAP is a hardware feature that guarantees semi-privileged (privboxed) code cannot (1) read/write kernel memory nor (2) indirect branch to kernel memory. Of course, the CPU has no notion of "kernel memory" or "semi-privileged execution"—we define *kernel memory* as any virtual address mapped by a PTE with the 'supervisor' bit set, and *semi-privileged execution* as instructions executing in privileged mode (CPL = 0) but that are located in non-kernel memory (i.e., a clear 'supervisor' bit in the code page's PTEs).

Assuming SPAP, it is possible to forgo all load/store instrumentation and limit control-flow instrumentation (§ 4.2) only to masking of jump targets to guarantee their chunk alignment. Listing 6 shows the simplified instrumentation call instruction enabled by SPAP (compared to Listing 3). The code chunk size

```
.align 16
%Reg1 = mov *disp(%Idx,scale,%Base)
%Reg2 = and $0xf, %Reg1
call *%Reg2
.align 16
```

Listing 6: Control-flow only instrumentation of memory-operand call.

is reduced to 16 bytes, which is enough to fit both the instrumented instructions and the required prefixes.

SPAP implementation We argue that SPAP can be implemented analogously to how current x86-64 processors implement SMAP/SMEP, which block *privileged mode execution* from accessing *user addresses*.[3] Restricting privboxed code data accesses and branching can happen at the same pipeline stages that enforce SMAP and SMEP, respectively. Listing 7 describes how the hardware can restrict privboxed code's data access: other than the CPL and PTE of the accessed page, which are already required by SMAP, SPAP only depends on the current instruction's PTE, which is available in the instruction TLB. Similarly, Listing 8 shows how SPAP hardware restricts privboxed branching. The information required is same as what is needed for the SMEP mechanism, plus the information of whether the current instruction is an indirect branch.

Expected overhead We claim that SPAP's additional access checks should have little to no effect on the latency of memory instructions. We base this claim on the overhead observed from enabling SMAP/SMEP, shown below, and the similarity of SPAP to them.

We evaluate SMAP overhead (on the platform described in § 7). We measure average load latency when accessing differently sized working sets from kernel space, with and without SMAP. Our test traverses each cache line in the working set buffer in random order (to prevent prefetching), with each load depending on the result of the previous one (to prevent the CPU's out-of-order execution from overlapping load execution). Preemption is disabled during the test, to ensure it has exclusive use of the CPU. We measure average cycles per load (i.e., total number of cycles divided by number of loads performed). Each test is run 31 times and we report the average of the last 30 runs.

Figure 3 shows results for working set sizes targeting the capacity of the CPU's TLB and L1/L2/L3 caches. We find that SMAP does not impact performance in a significant way, as (1) some tests still execute faster with SMAP enabled; and (2) the variance is greater than the difference between the configurations.

```
if (
  CPL < 3 and
  AccessedPage.S_bit is Set and
  CurrInstPage.S_bit is Cleared
):
  trap()
```

Listing 7: Hardware enforcement that semi-privileged execution loads/stores do not access kernel memory.

```
if (
  CurrInst is Indirect Branch and
  CPL < 3 and
  FetchPage.S_bit is Set and
  CurrInstPage.S_bit is Cleared
):
  trap()
```

Listing 8: Hardware enforcement that semi-privileged execution does not indirect branch to kernel code.

[3]The idea is to prevent exploits of kernel memory safety bugs that attempt to, e.g., jump to user-space code [15].

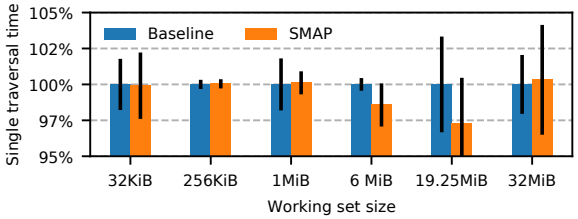

Figure 3: SMAP overhead on load instructions. Results are normalized to execution with SMAP disabled.

6 Security

We analyze Privbox's security against architectural and microarchitectural (transient execution) attacks. Recall that in privboxed code, every chunk (a 32-byte range at a 32-byte aligned address) contains a correctly instrumented, non-privileged instruction sequence of exactly 32 bytes, perhaps ending with no-ops (see § 3.3.2). This is verified by the kernel.

Semi-privileged execution has the following invariants:

Inv1 The target of *any* load/store instruction is not a kernel address. (Enforced by the instrumentation; § 4.2.1.)

Inv2 The target of *any* control-flow instruction (including returns) is a 32-byte aligned non-kernel address. (Enforced by the instrumentation; § 4.2.2.)

Inv3 The privboxed code section is read-only and user-space addresses outside of it are non-executable during semi-privileged execution. (Enforced by the virtual memory permissions; §§ 3.2–3.3.)

For normal (non-transient) instruction execution, Inv2 and Inv3 imply that semi-privileged execution can run only (instrumented) code located in the privboxed region. By Inv1, such code cannot access kernel memory, and the verifier guarantees it does not contain privileged instructions. Therefore, if regular unprivileged execution cannot perform some operation or memory access, neither can semi-privileged execution.

It remains to analyze transient execution attacks. Generally, such an attack uses architecturally-incorrect flows (whose instructions execute but do not subsequently commit) to leak memory contents via a microarchitectural side-channel [29]. Privbox's goal is thus to protect kernel memory.

The core observation is that transient execution of a *complete instrumented* chunk is safe, because it still sanitizes the operands of any memory operation in the chunk. We therefore only need to consider if transient execution can branch mid-chunk or outside of the privboxed region, either of which can happen due to indirect branch or return target misprediction. Crucially, we consider *any* "supervisor mode" transient execution—both semi-privileged and standard kernel execution—to cover attacks of privboxed code on the kernel. To this end, we analyze how the branch predictor can be "trained," i.e., which targets it observes and may mispredict execution to later:

We assume any training by user-space execution cannot affect "supervisor mode" execution, due to existing mitigations

such as Intel's enhanced indirect branch restricted speculation (eIBRS) [30] or Arm's CSV2 [31]. If this assumption does not hold, then the kernel is vulnerable regardless of Privbox.

We assume kernel execution can only train valid kernel branch/return targets (and therefore the kernel cannot transiently branch to privboxed code), because if an attacker can cause the kernel to train arbitrary addresses, they can attack the kernel regardless of Privbox. This still means that semi-privileged execution may (transiently) branch to a valid kernel function, possibly creating a speculative type confusion vulnerability [32]. Privbox's instrumentation can mitigate this problem (with run-time overhead) using retpolines [33] for indirect branches.

Finally, by Inv2, semi-privileged execution can only train non-kernel, chunk-aligned addresses. It might thus train user-mode addresses outside of the privboxed region. We assume that instructions from non-executable memory are not executed in transient execution,[4] so by Inv3, semi-privileged execution cannot be exploited by such training. However, training by semi-privileged can cause subsequent kernel execution to (transiently) branch to user-space instructions and execute them, as Privbox disabled SMEP. On current hardware, training by semi-privileged execution can be prevented from affecting the kernel's execution using a mechanism such as Intel's indirect branch predictor barrier (IBPB) [36] in the Privbox system call gate, but this will slow down system calls in privboxed code. (Our Privbox prototype does not implement this mitigation.) Future hardware could support SPAP-like extensions to eIBRS to make predictions of branches executed from supervisor pages uncontrollable by branches executed from user-level pages.

We assume SPAP can be implemented so that its restrictions apply to both normal and transient execution, as SMAP/SMEP have this guarantee [34, 35].

Limitation: Disabled SMEP Privbox's security drawbacks stem from disabling SMEP for semi-privileged execution without re-enabling it for kernel execution, as Privbox does for SMAP. As a result, semi-privileged execution can (1) exploit pre-existing kernel vulnerabilities that were mitigated by SMEP and (2) mount transient execution attacks against the kernel, as explained above. The SMEP limitation can be addressed by hardware reducing the cost of toggling SMEP. This should be possible, given that hardware has optimized SMAP toggling, an action the kernel frequently performs.

7 Evaluation

We evaluate the impact of Privbox on system call latency (§ 7.1), on system call-intensive I/O threads (§ 7.2), and the impact of privboxing complete real-world applications (§ 7.3).

[4] This holds on x86-64, where documentation states that SMEP and virtual memory execute restrictions apply to transient execution [34, 35].

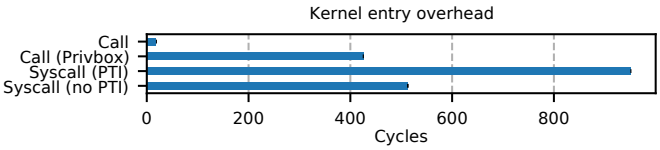

Figure 4: Cycles taken to execute a roundtrip to kernel using different entry methods. Regular call added for reference.

Platform We use a Dell PowerEdge R740 server with a 28-core Intel Xeon Gold 6132 (Skylake) CPU and 192 GiB of DRAM. Hyper-Threading is disabled. Due to current cloud computing trends, virtualized platforms represent the environments where evaluated workloads usually run. We therefore use a Linux v5.8 guest in a KVM virtual machine hosted on a Ubuntu 18.04 host. Reported measurements are averages of 10 executions after a single warmup run; error bars indicate standard deviation.

7.1 System call latency

We measure the end-to-end latency of invoking a non-existing system call, which covers user-to-kernel transition, entry code execution, and kernel-to-user return (with a "bad call" error). Our benchmark invokes the system call 100 M times and reports average invocation latency, measured with the CPU's cycle counter.

We compare the latency of a regular system call invocation with and without PTI to the latency of invoking the system call from within privboxed code. Figure 4 shows that a system call invocation alone takes about 950 and 510 cycles with and without PTI, respectively. Invocation from privboxed code takes on average 425 cycles, 2.2× and 1.2× faster, respectively, than the baseline with and without PTI.

The reason that a privboxed system call invocation is slower than a regular function call is that while Privbox eliminates hardware user/kernel transition costs, it must still manage software-related user/kernel transition steps. For instance, Privbox's system call gate (§ 3.2) switches the stack and saves/restores register state.

7.2 I/O-thread workloads

Here, we characterize the impact of Privbox on an I/O thread-based application architecture (see § 3.1 and Figure 1). We benchmark a generic server program that receives requests, processes them, and returns response. The server is composed of I/O and compute threads, which are responsible, respectively, for socket operations and the "business logic" of computing responses to incoming requests.

Our benchmark has two tunable parameters: (1) *Compute time*, the time compute thread spends on each request, which allows controlling how compute-heavy the workload is; and (2) *I/O size*, the number of bytes each for each socket I/O operation. We use fixed-sized messages, so the I/O size determines the number of system calls per message, i.e., how system call-intensive the workload is.

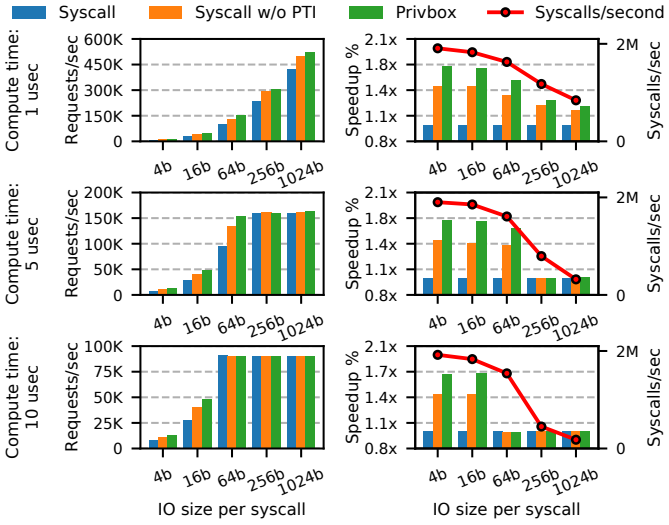

Figure 5: Privbox impact on an I/O-thread based server. Left: absolute throughput (requests/second), right: throughput normalized to default Linux (with PTI). Rows describe different compute times.

We compare between Privbox and standard execution with PTI (default Linux configuration) and without it. When using Privbox, only the I/O thread code is privboxed (and therefore compiled with instrumentation). Figure 5 shows the server's throughput (requests/second) as we vary the compute time (across rows) and I/O size (X axis in each row). For large I/O sizes (above 256 bytes) and compute time (above 5 μs), system call invocation frequency decreases and so all kernel entry methods yield similar throughput, as most CPU time is spent on compute or inside system calls (waiting for I/O). However, for fast compute and/or high rate of system calls (small I/O size), Privbox results in up to 1.72× speedup compared to regular execution with system calls.

7.3 Real-world workloads

This section analyzes the impact of Privbox on several popular real-world applications: Redis [18], memcached [19], and SQLite [20]. We modify each application to use Privbox, which requires changing/adding about 20–30 lines of code to make the application's main loop execute privboxed. All binaries are compiled with `-O2` optimizations and linked with our instrumented musl-1.2.0 C library. Importantly, we compile the entire application with Privbox's instrumentation, not only the part that runs privboxed. The reason is that our Privbox prototype does not support compiling an application with both instrumented and uninstrumented versions of the C library (see § 4.1). The upshot is that our results here are *lower bounds* of Privbox's benefit, as we instrument code that a full Privbox implementation would not.

To analyze instrumentation overhead, we measure each application with three instrumentation levels: (1) No instrumentation (*noinstr*), which shows the benefit from fast system call invocation; (2) full instrumentation (*fullinstr*, § 4.2), which shows Privbox's benefit (faster system calls, but with instrumentation overhead) on current hardware; and (3) control-flow only instrumentation (*brinstr*, § 5), which omits load/store instrumentation, thereby emulating Privbox's benefit on hardware with SPAP support.

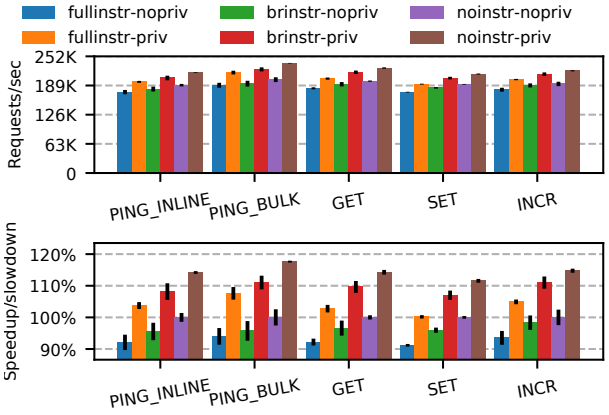

Figure 6: Redis server under load of redis-benchmark running 2 threads, 50 connections, 1 M requests. Top: throughput (requests/seconds). Bottom: throughput relative to 'noinstr-nopriv'.

To analyze the benefit from Privbox's fast system calls, we measure each instrumentation level with and without executing the privboxed code sections in privileged mode (tagged *priv* and *nopriv*, respectively). The speedup of *priv* over *nopriv* quantifies how Privbox's fast system calls offset instrumentation overhead.

Redis Redis [18] is a popular key-value store, often used as cache, document store, or for publish/subscribe messaging. We use Redis' recommended default setup of a single instance running a single thread, without persistency. We modify Redis' main loop to execute privboxed. The privboxed loop returns to user-space once per 10 K iterations to service signals (due to limitations of our prototype, see § 4.1).

We evaluate Redis using two benchmarks: (1) *redis-benchmark*, with which we simulate running various Redis commands by 50 concurrent clients that send 1 M requests, and (2) *memtier_benchmark* [37], a stress tester for NoSQL databases, which we run with a 10:1 read/write ratio of 32-byte objects.

Figure 6 shows redis-benchmark throughput of various Redis commands. Reducing system call overhead offers significant benefit: 'priv' executions have on average 13% higher throughput than their 'nopriv' variants. While Privbox's instrumentation overhead offsets some of this benefit, overall, a Privboxed Redis ('fullinstr-priv') obtains up to 7.6% higher throughput than its baseline ('noinstr-nopriv'). Had the CPU supported SPAP (enabling less instrumentation: 'brinstr-priv'), the throughput would improve to up 10% higher than the baseline. Results from memtier_benchmark (Figure 7) show similar trends, with 'fullinstr-priv' and 'brinstr-priv' obtaining 6% and 10% higher throughput than the 'noinstr-nopriv' baseline.

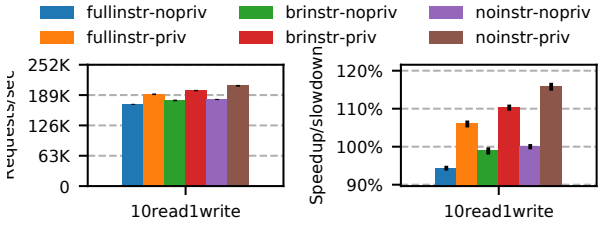

Figure 7: Redis server under load of memtier_benchmark, running 4 threads, 50 clients/thread, 10 K requests/client, no pipelining, 32 byte objects, 10:1 read/write ratio. Left: throughput (requests/seconds). Right: throughput relative to 'noinstr-nopriv'.

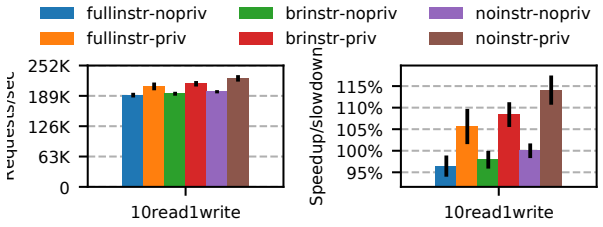

Figure 8: Memcached under load of memtier_benchmark, running 4 threads, 50 clients/thread, 10 K requests/client, no pipelining, 32 byte objects, 10:1 read/write ratio. Left: throughput (requests/seconds). Right: throughput relative to 'noinstr-nopriv'.

Memcached *memcached* [19] is a distributed object caching system offering a key-value store interface. Similarly to Redis, we modify the main loop to execute privboxed, and break out to user-space once per 10 K iterations.

We evaluate memcached by measuring the throughput obtained by memtier_benchmark, again with a 10:1 read/write ratio and keys/values of 16/32 bytes, respectively. Figure 8 shows the results, which mirror those of Redis. Specifically, 'priv' executions have on average 19% higher throughput than their 'nopriv' variants, which is sufficiently high for Privbox to outperform the baseline: 'fullinstr-priv' and 'brinstr-priv' obtain 4.5% and 6.9% higher throughput than the baseline.

SQLite SQLite [20] is a relational database engine. We evaluate it using sqlite-bench [38], a tool that measures throughput of various access patterns: writing/reading of sequential/random values in asynchronous/synchronous/batched modes. We use a RAM filesystem (`tmpfs`) to store the database files.

Figure 9 shows throughput obtained for each access sequence. Many sequences stand to benefit from Privbox's fast system calls (evidenced by an average 8% speed up of 'priv' over 'nopriv' variants), but these benefits are negated by instrumentation overhead. However, some patterns ("readrandom" and "fillseqsync") do not benefit from fast system calls.

Figure 10 explains the above results. It shows the ratio between number of system calls invoked to time spent in user code (i.e., CPU time minus system call execution time). We find a strong correlation between speedup from fast system calls and the system call/user time ratio. For example, "readrandom" suffers greatly from instrumentation because SQLite performs read queries using loads/stores (which are instrumented) without invoking system calls. The "fillseq-

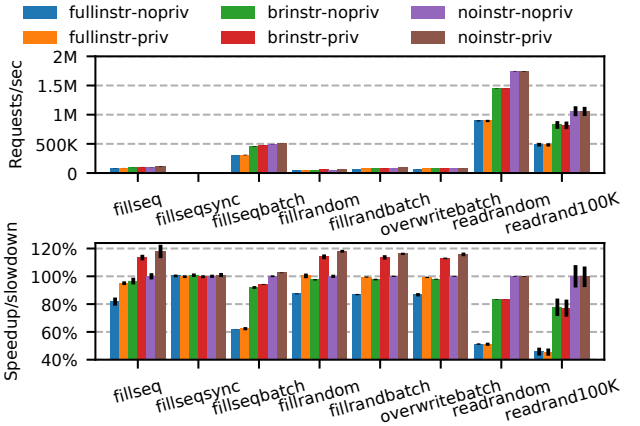

Figure 9: SQLite throughput. Top: throughput (operations/second). Bottom: throughput relative to 'noinstr-nopriv'.

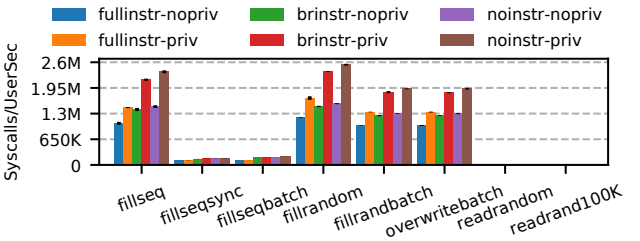

Figure 10: SQLite ratio of system calls to time spent in user-space code (log scale). Correlation can be seen with the bottom half of Figure 9: tests with a high system call to user time ratio show better speedup from 'priv' execution.

batch" sequence, which behaves the same with and without fast system calls ('priv'), batches I/O operations and thus performs fewer system calls. Finally, the "fillseqsync" sequence, which behaves the same for all instrumentation levels and execution modes, uses slow synchronous writes and so spends most of its time waiting, as opposed to running user code or entering/exiting the kernel.

8 Conclusion & future work

We propose Privbox, a design for speeding up system calls by sandboxed semi-privileged execution, without changing the underlying system call programming model. We believe Privbox can also be useful to improve isolation and fault tolerance within the kernel, e.g., by privboxing modules and device drivers to limit the memory and kernel APIs they access.

Our Privbox prototype uses simple compile-time instrumentation which incurs non-negligible overhead, offsetting some of the benefit from Privbox's fast system call invocation. There are several directions for reducing instrumentation overhead, which we leave to future work: Hardware features such as Intel's control-flow enforcement technology (CET [39]) can be useful for reducing control-flow instrumentation. Finally, a more sophisticated verifier can avoid redundant instrumentation (e.g., sanitizing a previously-sanitized register).

Acknowledgments

We thank the paper's anonymous reviewers and shepherd for their feedback and suggestions, which helped strengthen the paper. We also thank the artifact reviewers for their careful work, which helped improve the artifact.

References

[1] Livio Soares and Michael Stumm. "FlexSC: Flexible System Call Scheduling with Exception-Less System Calls." In: *OSDI*. 2010.

[2] Lars Müller. "KPTI a mitigation method against meltdown". In: *Advanced Microkernel Operating Systems* (2018), p. 41.

[3] *Fighting Spectre with cache flushes*. https://lwn.net/Articles/768418/.

[4] Moritz Lipp et al. "Meltdown: Reading kernel memory from user space". In: *USENIX Security*. 2018.

[5] Paul Kocher et al. "Spectre attacks: Exploiting speculative execution". In: *IEEE SP*. 2019.

[6] Mohan Rajagopalan et al. "Cassyopia: Compiler Assisted System Optimization". In: *HotOS*. 2003.

[7] Toshiyuki Maeda. "Kernel Mode Linux". In: *Linux J.* 2003.109 ().

[8] *Efficient io with io_uring*. https://kernel.dk/io_uring.pdf.

[9] *DPDK: Data Plane Development Kit*. https://www.dpdk.org/.

[10] Jonathan Behrens et al. "Efficiently mitigating transient execution attacks using the unmapped speculation contract". In: *OSDI*. 2020.

[11] Paul Barham et al. "Xen and the Art of Virtualization". In: *SIGOPS Oper. Syst. Rev.* 37.5 ().

[12] *VMWare: Paravirtualization API Version 2.5*. https://www.vmware.com/pdf/vmi_specs.pdf.

[13] Bennet Yee et al. "Native client: A sandbox for portable, untrusted x86 native code". In: *IEEE SP*. 2009.

[14] David Sehr et al. "Adapting Software Fault Isolation to Contemporary CPU Architectures". In: *USENIX Security*. 2010.

[15] *Supervisor mode access prevention*. https://lwn.net/Articles/517475/.

[16] *musl C library*. https://www.musl-libc.org/.

[17] Edited Jim Coplien and Douglas C Schmidt. "Reactor-an object behavioral pattern for demultiplexing and dispatching handles for synchronous events". In: (1995).

[18] *Redis*. https://redis.io/.

[19] *memcached*. https://memcached.org/.

[20] *SQLite*. https://www.sqlite.org/index.html.

[21] *SPDK: Storage Performance Development Kit*. https://spdk.io/.

[22] Yuhong Zhong et al. "BPF for Storage: An Exokernel-Inspired Approach". In: *HotOS*. 2021.

[23] *eBPF*. https://ebpf.io/.

[24] A Starovoitov, J Schulist, and D Borkmann. "Linux Socket Filtering aka Berkeley Packet Filter (BPF)". In: *Documentation/networking/filter.txt* (2016).

[25] Matt Welsh, David Culler, and Eric Brewer. "SEDA: An Architecture for Well-Conditioned, Scalable Internet Services". In: *SOSP*. 2001.

[26] Michael Matz et al. "System V Application Binary Interface". In: *AMD64 Architecture Processor Supplement, Draft v0* 99 (2013), p. 57.

[27] *Signalfd manual page*. https://man7.org/linux/man-pages/man2/signalfd.2.html.

[28] *LLVM project*. https://llvm.org/.

[29] Claudio Canella et al. "A Systematic Evaluation of Transient Execution Attacks and Defenses". In: *USENIX Security*. 2019.

[30] Intel. *Indirect Branch Restricted Speculation*. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-restricted-speculation.html.

[31] Arm. *Vulnerability of Speculative Processors*. https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability.

[32] Ofek Kirzner and Adam Morrison. "An Analysis of Speculative Type Confusion Vulnerabilities in the Wild". In: *USENIX Security*. 2021.

[33] *Retpoline: a software construct for preventing branch-target-injection*. https://support.google.com/faqs/answer/7625886.

[34] AMD. *Software Techniques For Managing Speculation On AMD Processors*. https://www.amd.com/system/files/documents/software-techniques-for-managing-speculation.pdf.

[35] Intel. *Speculative Execution Side Channel Mitigations*. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/speculative-execution-side-channel-mitigations.html.

[36] *Indirect Branch Predictor Barrier*. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-predictor-barrier.html.

[37] *memtier-benchmark*. https://github.com/RedisLabs/memtier_benchmark.

[38] *sqlite-bench*. https://github.com/ukontainer/sqlite-bench.

[39] *Intel CET*. https://newsroom.intel.com/editorials/intel-cet-answers-call-protect-common-malware-threats/.

[40] *privbox/devenv: ATC'22*. https://doi.org/10.5281/zenodo.6618853. DOI: 10.5281/zenodo.6618853.

[41] *privbox/linux: ATC'22*. https://doi.org/10.5281/zenodo.6618867. DOI: 10.5281/zenodo.6618867.

[42] *privbox/musl: ATC'22*. https://doi.org/10.5281/zenodo.6618859. DOI: 10.5281/zenodo.6618859.

[43] *privbox/llvm-project: ATC'22*. https://doi.org/10.5281/zenodo.6618847. DOI: 10.5281/zenodo.6618847.

[44] *privbox/redis: ATC'22*. https://doi.org/10.5281/zenodo.6618855. DOI: 10.5281/zenodo.6618855.

[45] *privbox/memcached: ATC'22*. https://doi.org/10.5281/zenodo.6618874. DOI: 10.5281/zenodo.6618874.

[46] *privbox/sqlite-bench: ATC'22*. https://doi.org/10.5281/zenodo.6618869. DOI: 10.5281/zenodo.6618869.

[47] *privbox/piotbench: ATC'22*. https://doi.org/10.5281/zenodo.6618857. DOI: 10.5281/zenodo.6618857.

[48] *privbox/libevent: ATC'22*. https://doi.org/10.5281/zenodo.6618872. DOI: 10.5281/zenodo.6618872.

A Instrumentation details

A.1 Special case load/store instrumentation

Listings 9 and 10 detail instrumentation for non stack-relative operations which have only one of the index/base registers specified, and displacement is either 1, 2 or 4 bytes long. In these scenarios, we can ensure any provided address value will become either a user or a non-canonical address by clearing bit 60 of the specified (base or index) register. This is sufficient because neither multiplication (in case of index register) nor addition of a 4-byte long displacement value will result in a canonical kernel address with 1s in all 16 most significant bits.

```
.align CHUNK_SIZE
SAVE_EFLAGS
%Reg1 = btr $60, %Base
RESTORE_EFLAGS
OP operand1, disp(,,%Reg1)
```

Listing 9: Instrumentation of operand containing base register

```
.align CHUNK_SIZE
SAVE_EFLAGS
%Reg1 = btr $60, %Idx
RESTORE_EFLAGS
OP operand1, disp(%Reg1,scale,)
```

Listing 10: Instrumentation of operand containing index register

```
.align CHUNK_SIZE
SAVE_EFLAGS
%Reg1 = btr $63, %Reg
%Reg2 = and $~(CHUNK_SIZE - 1),
        %Reg1
RESTORE_EFLAGS
jmp *%Reg2
```

Listing 11: Instrumentation of register operand jump

```
.align CHUNK_SIZE
SAVE_EFLAGS
%Reg1 = lea *disp(%Idx,scale,%Base)
%Reg2 = btr $63, %Reg1
%Reg3 = mov *%Reg2
%Reg4 = btr $63, %Reg3
%Reg5 = and $~(CHUNK_SIZE - 1),
               %Reg4
RESTORE_EFLAGS
jmp *%Reg5
```

Listing 12: Instrumentation of memory-operand jump

A.2 Stack accesses

Stack-based operations can be considered safe as long as we ensure that at any point in time, the stack pointer points to valid user memory. The safety of stack-relative operations is ensured by maintaining the following invariant:

- When entering semi-privileged execution, the stack pointer must be set to a known valid value.
- When the stack pointer is set to a specific value, i.e. copied from another register, the copied value must be sanitized in a similar manner to an operand of a load/store instruction (i.e., clear its MSB).
- Each operation modifying/incrementing/decrementing the stack pointer must change the value by no more than a page, and must access the memory pointed by the new stack pointer value unconditionally afterwards (e.g., in same basic block). This permits operations like `push` and `pop`, as well as operations such as `add` and `sub`, as long as the memory is accessed through the stack pointer shortly after.

  Incrementing/decrementing stack pointer without dereferencing can expose the code to an attack where the same sequence of instructions is used to modify stack pointer in small increments to an arbitrary value, until it points to kernel memory. Enforcing a stack access after the stack pointer changes makes sure that the stack pointer does not travel over inaccessible memory, such as the gap between kernel and user memory and the zero page in user memory, thereby preventing the stack pointer from overflowing/underflowing into kernel memory.

The above restrictions ensure that stack pointer always points to user memory, so loads/stores relative to the stack pointer register can be considered safe, as long as verifier successfully verifies that the above invariants hold.

A.3 Jump instructions

Listings 11 and 12 describe instrumentation of register and memory operand jumps, as mentioned in § 4.2.2.

B Artifact Description

Abstract

Our artifacts include all of the Privbox prototype code, as well as the scripts and benchmarks used to produce the results presented in this paper.

Scope

The artifacts can be used to:

- Set up a development and runtime environment for our prototype (§ 4).
- Run the experiments described in § 7, specifically, to reproduce results we present in Figures 3–9.

Refer to the artifact's README (`https://github.com/privbox/devenv/blob/privbox/README.md`) for complete instructions.

Contents

- **devenv** [40] - a repository containing a README and scripts to set up a development and evaluation environment for the Privbox prototype.
- The Privbox prototype, which consists of:
  - **Linux** [41] and **musl C library** [42] - Operating system and C library with Privbox support.
  - **LLVM** [43] - LLVM toolchain capable of creating binaries instrumented for Privbox.
- **Benchmarks** [44, 45, 46, 47] - programs we used to evaluate Privbox.

Hosting

Our artifacts are available on Github (`https://github.com/privbox/`), as well as archived on Zenodo [40, 48, 41, 43, 45, 42, 47, 44, 46].

Requirements

Evaluation of our artifact requires an Intel x86-64 machine running Linux (we have used Ubuntu 18.04). Additionally, we rely on Docker and QEMU/KVM.

- **CPU type**: Our evaluation uses an Intel Skylake CPU. While any modern Intel-architecture CPU is suitable, evaluation results might differ due to microarchitectural changes.
- **Virtualization**: Our prototype runs as a KVM-based virtual machine. In our evaluation, we use a bare-metal server as a platform. It is possible to use a virtual machine, as long as it supports nested virtualization. However, nested virtualization incurs additional overhead that might affect evaluation results.