Fast Intra-kernel Isolation and Security with IskiOS

Spyridoula Gravani
Mohammad Hedayati
{sgravani,hedayati}@cs.rochester.edu
University of Rochester

John Criswell
Michael L. Scott
{criswell,scott}@cs.rochester.edu
University of Rochester

ABSTRACT

The kernels of operating systems such as Windows, Linux, and MacOS are vulnerable to control-flow hijacking. Defenses exist, but many require efficient intra-address-space isolation. Execute-only memory, for example, requires read protection on code segments, and shadow stacks require protection from buffer overwrites. Intel's Protection Keys for Userspace (PKU) could, in principle, provide the intra-kernel isolation needed by such defenses, but, when used as designed, it applies only to user-mode application code.

This paper presents an unconventional approach to memory protection, allowing PKU to be used within the operating system kernel on existing Intel hardware, replacing the traditional user/supervisor isolation mechanism and, simultaneously, enabling efficient intra-kernel isolation. We call the resulting mechanism *Protection Keys for Kernelspace* (PKK). To demonstrate its utility and efficiency, we present a system we call IskiOS: a Linux variant featuring execute-only memory (XOM) and the *first-ever race-free shadow stacks for x86-64*.

Experiments with the LMBench kernel microbenchmarks display a geometric mean overhead of about 11% for PKK and no additional overhead for XOM. IskiOS's shadow stacks bring the total to 22%. For full applications, experiments with the system benchmarks of the Phoronix test suite display negligible overhead for PKK and XOM, and less than 5% geometric mean overhead for shadow stacks.

CCS CONCEPTS

• **Security and privacy → Operating systems security**.

KEYWORDS

security, operating systems, protection keys, intra-kernel isolation

ACM Reference Format:
Spyridoula Gravani, Mohammad Hedayati, John Criswell, and Michael L. Scott. 2021. Fast Intra-kernel Isolation and Security with IskiOS. In *24th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '21), October 6–8, 2021, San Sebastian, Spain.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3471621.3471849

1 INTRODUCTION

As the operating system (OS) kernel forms the foundation of the software stack, control-flow hijacking attacks [74] on the OS kernel jeopardize the security of the entire system. To safeguard system integrity, *control-flow integrity (CFI) enforcement* [22, 31, 56, 66] ensures that only paths in an authorized control-flow graph are followed during execution [1]. At the same time, *leakage-resilient diversification* [4, 6, 21, 32, 33, 71] undermines the attacker's knowledge of the code layout in memory, making successful exploitation more difficult. Both techniques must efficiently isolate memory within the OS kernel. CFI must protect the integrity of return addresses on the stack [8, 10, 10, 17, 30, 34]; diversification must prevent attackers from learning the layout of kernel code—e.g., by leveraging buffer overreads [78, 79].

Existing intra-kernel isolation mechanisms generally rely on *software fault isolation* (SFI) [76, 86], adding bounds checks to memory writes [22] and reads [71] to prevent unauthorized access to sensitive data. As the overhead of such checks is proportional to the number of protected regions, SFI-based approaches often group all sensitive information together—typically at the upper end of the virtual address space—so only one bound must be checked [22, 71]. Unfortunately, this grouping reduces design flexibility and diversification entropy. The most efficient SFI implementation on x86-64 [71] relies on Intel's Memory Protection Extensions (MPX), a hardware feature that has recently been deprecated [44].

In this work, we introduce an alternative approach to intra-kernel memory isolation based on novel use of Intel's Protection Keys for Userspace (PKU) [46]. PKU allows each page table entry (PTE) to specify one of 16 protection keys (PKEYs); at run time, unprivileged software can dynamically disable read access or both read & write access to pages on a PKEY-by-PKEY basis. However, as the name implies, PKU applies only to pages whose PTEs are marked as user space (i.e., accessible to application code).

We have developed a new user/kernel isolation mechanism, dubbed *PKK (Protection Keys for Kernelspace)*, which uses PKU hardware to dynamically disable read and/or write access to pages while running in the OS kernel. Our PKK mechanism is both inexpensive and flexible, and supports both SMEP and SMAP (Supervisor-Mode Execution/Access Prevention [46]). Additionally, while it is fully compatible with Kernel Page Table Isolation (KPTI) [39], PKK *does not require* KPTI in order to function, allowing systems that do not need or do not want KPTI to avoid its overhead.[1]

To demonstrate the effectiveness of PKK, we developed *IskiOS*,[2] a system that provides execute-only memory (XOM) for storing code and write-protected shadow stacks for the Linux kernel. Unlike previous approaches [71], IskiOS protects both the code and the

RAID '21, October 6–8, 2021, San Sebastian, Spain

ACM ISBN 978-1-4503-9058-3/21/10.
https://doi.org/10.1145/3471621.3471849

[1] As part of its Trust Domain Extensions (TDX), Intel has recently announced that certain future processors will provide hardware support for Protection Keys in Supervisor mode (PKS). PKS machines will allow 16 protection keys to be used in supervisor mode in addition to the 16 in user mode. Our PKK mechanism provides PKS-like functionality on existing machines.
[2] "IskiOS" means "shadow" in Greek.

shadow stacks efficiently, with no changes to virtual address space layout; IskiOS is thus compatible with existing kernel memory allocators and imposes no limits on the entropy of diversification strategies.

To protect the shadow stack, IskiOS incorporates a novel calling convention that is immune not only to control-flow hijacking attacks in the calling thread but also to attacks that leverage cross-thread races [12, 15]. Along with this calling convention, we introduce a pair of techniques to reduce the frequency of PKU domain switches. The first technique, dubbed *shadow write optimization*, avoids redundant writes of return addresses to the shadow stack. The second performs aggressive function inlining, avoiding calls to many smaller functions altogether.

Our contributions can be summarized as follows:

- We demonstrate that Intel PKU can be used to enable efficient user/kernel isolation, to widen the set of protection modes for kernel memory, and to change protections cheaply at fine temporal granularity.
- We describe a system, IskiOS, that leverages the resulting PKK mechanism to provide execute-only memory (XOM) and protected shadow stacks within the Linux kernel, with arbitrary address space layout. To the best of our knowledge, ours is the *first write-protected, race-free shadow stack for the x86-64.*
- We describe a *shadow write optimization* which, together with aggressive inlining, serves to significantly reduce the remaining overheads of PKK-based shadow stacks.
- Using the LMBench microbenchmark suite [63], we demonstrate a geometric mean overhead, relative to standard Linux, of approximately 11% for PKK and XOM. Adding our protected shadow stacks brings the total to roughly 22%.
- Using workloads drawn from performance regression tracking of the mainline Linux kernel [59]), we confirm that overheads are substantially lower for full-size programs: the average overhead of PKK and XOM becomes negligible, and protected shadow stacks add less than 5% geomean overhead.

Source code for IskiOS is available at https://github.com/URSec/iskios.

2 BACKGROUND

2.1 User/Kernel Isolation on x86

For efficiency, Linux has traditionally mapped kernel code, data, and loadable modules into the address space of every running process. This convention allows system calls, traps, and interrupts to vector into the kernel without flushing entries from the TLB. It also allows the kernel to easily access the active process's user-level memory.

To support this address space sharing, the x86 includes a User/Supervisor (U/S) flag in every page table entry (PTE) [46]. Pages in the upper region of the address space, which map kernel code and data, are marked as supervisor-only: accessing such a page while running in user mode (Ring 3) results in a trap [46].[3]

[3]To mitigate Meltdown [60] attacks, some recent versions of Linux have employed separate page tables for user and supervisor code (i.e., KPTI [39]). The resulting security comes at a significant performance cost [7]; long-term solutions depend on hardware updates that restore the viability of user/kernel address space sharing.

2.2 Protection Keys for Userspace

In its Skylake generation of processors, Intel introduced *Protection Keys for Userspace* (PKU) [46]. A descendant of mechanisms dating back to the IBM 360 [42], PKU is unusual in that it applies only to user-mode pages and allows permissions to be changed by an unprivileged instruction. While PKU is intended mainly as a memory safety enhancement, researchers have used it [9, 40, 50, 83] to provide intra-process isolation for user-space applications.

PKU introduces a 32-bit register called `pkru` and instructions (`rdpkru` and `wrpkru`) to read and write it [46]. Four previously unused bits (bits 62:59) in each PTE are then used to associate the page with one of 16 possible protection domains. The `pkru` uses two bits per key—dubbed access disable (AD) and write disable (WD)—to encode access rights that are restricted in each domain. On any load or store to a user-space page, the processor checks permissions as usual and then drops all rights (if any) associated with the page's protection key in the `pkru` of the currently executing hyperthread. The processor ignores the protection key for instruction fetches and for loads and stores of addresses whose PTE indicates kernel space. Thus, in the expected use case, PKU does not affect accesses to kernel code or data.

To date, intra-address-space isolation in the OS kernel has relied on SFI [76, 86], which adds instrumentation (e.g., bit-masking operations [22] or MPX bounds checks [26, 71]) before stores and/or loads. Enabling protection keys in the kernel would bring two key benefits that are difficult or impossible to attain with existing SFI-based approaches.

First, PKU provides more flexibility in address space layout. To improve performance, solutions using SFI [86] often place pages of the same protection domain in contiguous virtual addresses [11, 22, 26, 71]. This approach requires significant engineering effort (for memory allocators in particular [71]) and reduces the entropy of code-layout randomization schemes. By contrast, protection keys permit pages in different domains to be located anywhere within the virtual address space with no performance or entropy loss.

Second, PKU easily supports up to 16 protection domains [46]. By contrast, the complexity of bit-masking bounds checks increases with the number of protection domains [22], and Intel MPX [46] (which is now deprecated [44]) provides only four bounds registers. With PKU, a single `wrpkru` instruction can change the access rights of an arbitrary subset of the 16 protection domains.

3 PROTECTION KEYS FOR KERNELSPACE

Our goal is to enable what one might call *Protection Keys for Kernelspace (PKK)* while maintaining the traditional guarantees of user/kernel isolation. In a conventional system, however, kernel code and data are placed in pages for which the U/S flag is clear (indicating they can be accessed only in supervisor mode), and the hardware ignores the PKU bits when accessing such pages. Fortunately, we observe that we can use memory protection keys to subsume the functionality of the U/S bit, and then leave that bit set in the PTEs of both user and kernel pages.

There are only two things that the OS kernel can do that user code cannot: execute privileged instructions and access protected OS kernel memory. The execution of privileged instructions is controlled not by the U/S bit in PTEs but by the processor's *execution*

mode (Ring 3 v. Ring 0) [46], which we do not change. The effect of the U/S bit on loads and stores can then be emulated by assigning user and supervisor pages to different protection domains, enabling and disabling access to these domains when crossing from user to kernel space or vice versa, and preventing changes to `pkru` from occurring at other times (see Section 3.1 below). As `pkru` permissions have no impact on instruction fetches, code running in user-mode can jump directly to code within the kernel's code segment. However, such a jump into kernel code is harmless *because the processor is still in user-mode (Ring 3)*: executing a privileged instruction causes a trap (because the execution mode is still unprivileged), and reading or writing kernel memory causes a trap (because access is locked out by the `pkru`). As now, the only way to enter the kernel is to use a system call, trap, or interrupt which will change the processor's mode from user-mode to kernel-mode.

To implement PKK, IskiOS sets the U/S bit in every entry of every page table, with the exception of a few entries used to map trampoline pages that handle system calls and interrupts. This effectively marks all memory as user mode. IskiOS then reserves keys 0–7 for use as *kernel protection keys*. Every page of kernel memory will be assigned one of these eight keys; IskiOS disables both read and write access to pages with these keys when the system is operating in user mode and enables them (selectively, as dictated by the security policy) when executing kernel code. Existing applications that make legitimate use of PKU can continue to obtain keys from the kernel through the `pkey_alloc()` [38] system call, which is modified to return an available key between 8 and 15. As applications may use PKEYs 8–15 to enforce their own security policies, IskiOS saves the value of the `pkru` on kernel entry and restores it on return to user space.

3.1 Controlling Use of `wrpkru`

Listing 1: glibc Modification.

```
1  /* Overwrite the PKRU register with VALUE.  */
2  static inline void pkey_write(unsigned int value)
       {
3    __asm__ volatile(
4        "Lmask: \n\t"
5        "orl  $0x0000ffff, %%eax \n\t"
6        "wrpkru \n\t"
7        "cmpw  $0xffff, %%ax \n\t"
8        "jne Lmask \n\t"
9        :: "a"(value), "c"(0), "d"(0));
10 }
```

Since `wrpkru` is not a privileged instruction [46], application code can modify the `pkru` at will. IskiOS must therefore prevent application code from enabling read or write access to kernel pages. To do so, IskiOS must address two challenges. First, it must allow applications to use the `wrpkru` instruction to enforce their own security policies while preventing them from clearing the WD or AD bits for PKEYs 0–7. Second, given that (unprivileged) kernel instructions are now executable in user mode, IskiOS must prevent applications from compromising security by jumping to `wrpkru` instructions in the kernel.

Restricting Application wrpkrus To address the first challenge, IskiOS borrows techniques from our earlier Hodor project [40]: IskiOS ensures that all application pages with execute permission have write permission disabled; when the kernel maps a page into the virtual address space of an application with execute permissions, it scans the page (including boundaries with the pages before and after it) for any byte sequence that might comprise a `wrpkru` instruction. If it finds any, it places a debug watchpoint [46] on the address. If the address is ever used for an instruction fetch, the watchpoint generates a trap. IskiOS then inspects the value of the intended write to the `pkru` to ensure that the process is not modifying PKEYs 0–7 and, if it is, terminates the process. This convention allows applications to continue using PKEYs 8–15 without restriction. Programs that generate code dynamically can write code to a page and then use `mprotect()` to change the permissions from writable and non-executable to non-writable and executable; IskiOS will scan the page when changing the permissions.

Listing 2: Safe `wrpkru` in Kernel.

```
1  ; wrpkru instance in kernel code
2  wrpkru
3
4  ; ensure execution is in kernel mode
5  movq    %cs, %rcx
6  testb   $3, %cl
7  je      Lskip
8  Ltrap:
9  ud2
10
11 Lskip:
12 ; ensure expected permissions are set in pkru
13 cmpw $EXP_PERM, %ax
14 jne Ltrap
```

Our work on Hodor showed [40] that existing Linux applications almost never contain more than one or two `wrpkru` instructions, confirming that the debug watchpoint approach is practical for existing applications. One might worry that as more applications come to use PKU, the four available watchpoint registers [46] might become a scarce resource. We observe, however, that applications making legitimate use of protection keys can be expected to rely on the routines provided in Linux's C Standard Library (`libc`) to change the `pkru`. To avoid using debug watchpoints in the common case, IskiOS adds lines 4–5 and 7–8 in Listing 1 to the `pkey_write` library function in GNU's `glibc`. These mask the lower 16 bits of the value written to the `pkru` register. With this change, `glibc` requires no watchpoint because it will always ensure that access to kernel pages is disabled.[4]

Protecting Kernel wrpkrus The second challenge reflects the fact that `wrprku` instructions in the kernel typically *will* clear the AD and WD bits of kernel PKEYs. To prevent application code from using these instructions, IskiOS inserts a check after each `wrpkru` in kernel code to ensure that the processor is currently running in Ring 0, as specified by the code segment register (`cs`)—see lines 5–7 in Listing 2. In addition, IskiOS ensures that the expected permissions have been set, thereby avoiding an attack in which overly generous permissions are placed in `eax` before jumping to a `wrpkru`

[4] A signal could occur during `pkey_write`, but IskiOS disables access to PKEYs 0–7 before dispatching signal handlers and when resuming application execution after a signal handler returns.

instruction[5] (lines 13–14 in Listing 2). If any check fails, a production system would terminate the current process; for simplicity, our prototype simply crashes (line 9 in Listing 2).

3.2 SMEP Compatibility

Recent Intel processors provide Supervisor-Mode Execution Prevention (SMEP) [46] to harden the kernel against *ret2usr* [48] attacks. When SMEP is enabled, kernel-mode code cannot fetch instructions from pages whose PTEs are marked as user space; any such access will cause a page fault, allowing the OS to intervene. Since IskiOS configures kernel memory as user memory, we must disable SMEP for kernel code to execute. To prevent the kernel from executing arbitrary user code without SMEP support, IskiOS adopts kGuard's [48] approach and adds control-flow checks on kernel code. These checks, described in more detail in Section 6.2.1, apply to every indirect control transfer and ensure that privileged execution remains within kernel space. If execution attempts to cross to an address in the user portion of the virtual address space, IskiOS can intervene (our prototype simply halts).

3.3 SMAP Compatibility

Supervisor-Mode Access Prevention (SMAP) [46] is a CPU extension that disables supervisor-mode accesses to user pages in an attempt to prevent attacker-controlled pointers from accessing user memory directly, possibly subverting the kernel's control flow [18]. When the operating system needs to access user memory for legitimate purposes (e.g., `copy_to/from_user()` [5]), it can temporarily disable SMAP protection by setting the alignment checking flag bit in the EFLAGS status register [46]. IskiOS configures all linear addresses to be user mode. Consequently, SMAP must be disabled for kernel code to be able to access its own data. As with SMEP, IskiOS clears the SMAP bit at boot time. To replicate SMAP's protections, IskiOS disallows kernel access to pages tagged with keys 8–15 (i.e., user pages) by default. The `copy_to/from_user()` [5] functions temporarily enable access to pages tagged with keys 8–15 when they need to read or write application memory.

3.4 Side Channels

PKK does not introduce any speculative execution side channels on processors following the Skylake generation.[6] Specifically, hardware mitigations against Meltdown [60] guarantee that data will never be loaded, even speculatively, if protection keys do not allow it [45]. However, since PKK makes kernel code executable from user space, user programs may hypothetically be able to infer the location of kernel code fragments by jumping to random locations and observing the side effects on processor and memory state. To minimize the window of vulnerability, IskiOS will terminate and blacklist an application, regardless of installed signal handlers, the first time it traps when executing instructions within the kernel code segment in user-space.

[5]In a future system using PKS hardware, we would need similar checks to prevent the gadget-ization of `wrmsr` instructions, which can be used to disable PKS altogether. Unlike `wrpkru`, which impacts only the protection keys and is therefore relatively rare, `wrmsr` serves many purposes and appears in many parts of the OS kernel.

[6]Skylake processors are susceptible to both Meltdown and Meltdown-PK attacks, with or without PKK.

4 ENHANCING SECURITY WITH PKK

We now describe our threat model, and how we use Intel PKU to (1) make kernel code unreadable (execute-only), thereby preventing buffer overreads [79] from revealing potential gadgets, and (2) provide the kernel with efficient, race-free shadow stacks.

4.1 Threat Model

We assume the attacker aims to execute a computation within the OS kernel with supervisor privileges. The kernel itself is non-malicious but may have exploitable memory safety errors such as buffer overflows [69] and dangling pointers [3]. Our attacker is an unprivileged user: they can execute arbitrary code in user space but cannot direct the OS kernel to load new kernel modules implementing malicious code.[7] We assume the enforcement of the W^X [2, 46, 81] policy that prevents the attacker from injecting code directly into kernel memory. Finally, we assume that the hardware is correctly implemented and that side channels are out of scope.

4.2 Kernel XOM

Code reuse attacks (e.g., ROP [74]) utilize information on where and how code has been placed in memory. To thwart such attacks, code diversification schemes randomize the layout and location of code so that attackers cannot locate useful gadgets and functions using a priori knowledge [21, 49, 70]. Advanced code reuse attacks (e.g., JIT-ROP [78]) exploit buffer overread bugs [79] to inspect the code segment to locate reusable code. To prevent such attacks, IskiOS places all kernel and kernel module code pages in *eXecute Only Memory (XOM)*—memory that can be executed but not read or written. The x86 architecture does not support XOM directly, but IskiOS implements it using PKK.

Specifically, IskiOS reserves one of the 8 OS kernel protection keys for the OS kernel code segment. It configures all page table entries for pages containing kernel code to use this key. It then sets the access disable (`AD`) bit in the `pkru` for this key, disabling read access to pages containing kernel code. Since protection keys affect only data accesses, instruction fetch and execution are permitted [46]. Unlike SFI-based implementations of XOM [6, 71], IskiOS can protect code pages at any virtual address, placing no restrictions on diversification: code pages and data pages can be interspersed and placed wherever desired in the virtual address space.

IskiOS also needs to protect *physmap*, the direct map of physical memory used in kernel space [47]. Physmap facilitates dynamic kernel memory allocation, but it also causes address aliasing—more than one virtual address mapping to the same physical address. To prevent buffer overreads from reading the kernel code segment through its mapping in physmap, IskiOS ensures that both mappings of the kernel code segment (including the one in physmap) use the same reserved kernel protection key.

4.3 Kernel Shadow Stack

Advanced code-reuse attacks commonly modify return addresses on the stack [37, 75]. A *shadow stack* protects against such attacks by keeping return addresses where they cannot easily be modified. While the location of the shadow stack could simply be obscured via

[7]Systems such as SecVisor [77] can prevent the loading of such malicious code if privileged user-space tools cannot be trusted.

randomization [52], such schemes have proven insufficient [17, 35, 91]. For real protection, we need the shadow stack to be *inaccessible*.

We also need it to be *race free*, with no timing windows during which a return address can be modified by an attacker on another core. Previous shadow stack implementations [9, 12, 13, 23, 25] have instrumented function prologues to copy the return address from the original stack to the shadow stack. Such designs leave the return address vulnerable to modification by another thread after the call instruction saves it but before the function prologue copies it. Several previous schemes [12, 13, 25] also exhibit races in function epilogues when they verify the validity of the return address (or copy it from the shadow stack back to the original stack) prior to executing a return instruction: the return address can be corrupted after it is validated (or copied from the shadow stack) but before the called function returns. While the windows of vulnerability in these schemes are small, real-world implementations such as Microsoft's Return Flow Guard (RFG) [12] and Clang's shadow stack for x86-64 [15] were removed quickly after their public release as the inherent races were shown to be exploitable.

IskiOS utilizes PKK, together with call-site modification, to provide what we believe to be *the first write-protected, race-free shadow stack for the x86*. To avoid the need for a separate stack pointer, we utilize a *parallel shadow stack* [23] design in which all shadow stack entries are located at a constant offset from their locations on the original stack. To protect the shadow stack from tampering, we dedicate one of the kernel protection keys to pages used for shadow stacks (and the pages that map to the same frames in physmap). During normal execution, write access to the shadow stack is disabled while read access remains enabled. When IskiOS needs to write a copy of the return address to the shadow stack, it temporarily enables write access to the shadow stack's protection key (PKEY 2) in the `pkru` (this impacts only the current hardware thread), writes the return address to the shadow stack, and then revokes write access. Function return requires no changes to the `pkru` as the shadow stack is always readable.

To provide race-freedom, IskiOS modifies the default calling convention to pass the return address to each function in a register, rather than pushing it onto the stack. We modify the code generator to reserve `%r10` for this purpose. When a caller wishes to make a function call, it first stores `%r10` (which holds the address of the function to which the caller itself should return) to the shadow stack and then loads the return address for the *callee* into `%r10`. On function return, control is redirected to the address saved in `%r10` via a jump-through-register instruction. We could also, of course, compare the main and shadow stack copies of the return address and raise an exception if they differ, but there seems to be no point: when running IskiOS, the main-stack return address is entirely unused, and attackers have no motivation to change it.

After each call, the compiler inserts code to load the return address from the shadow stack back into `%r10`. Since the shadow stack is readable, no `wrpkru` instruction is needed for this reload. By writing to the shadow stack only when making a nested call, IskiOS avoids the expense of a `wrpkru` instruction when a function calls no other function—i.e., when it is a dynamic leaf function. Furthermore, when a function makes more than one nested call, the expense is incurred only once—more on this in Section 5.1.

Interrupt Handling One limitation of our shadow stack design is that it does not leverage the hardware's return address predictor, which is primed by `call` instructions and employed by `ret`. It is tempting to pursue a convention that points the stack pointer into the shadow stack and uses a real `ret` instruction. Unfortunately, such an approach is incompatible with the processor's interrupt dispatch mechanism: a hardware interrupt that occurs when the stack pointer is pointing to the write-protected shadow stack will cause the kernel to crash when it attempts to save the processor state on what the kernel thinks is the kernel stack before executing the interrupt handler.

For two reasons, IskiOS must save and restore the `pkru` on the shadow stack during system call, trap, and interrupt dispatch. First, application code can change that register [46], so IskiOS must preserve the value needed in user code. Second, interrupts can occur while an OS kernel function has enabled write access to the shadow stack; IskiOS must disable write access to the shadow stack prior to executing the interrupt handler and restore the `pkru` when returning from the interrupt.

5 PERFORMANCE OPTIMIZATIONS

Hedayati et al. [40] report and our experiments confirm that the cost of executing a `wrpkru` instruction is roughly 26 cycles. This can lead to overheads ranging from negligible to significant, depending on how protection keys are used. For example, enabling XOM requires a single `pkru` update to drop the access permissions for code—after that, it's free. Protected shadow stacks, on the other hand, require IskiOS to execute two `wrpkru` instructions every time it writes a return address to the shadow stack: one to enable access and one to disable it again, for a cost of about 52 additional cycles in each dynamic call. To avoid this cost whenever possible, we explore two optimizations to our shadow stack.

5.1 Shadow Write Optimization

We observe that IskiOS can avoid writing the return value to the shadow stack if the desired value is already there. In particular, if a function calls 20 other functions, it only has to save its return address when it makes the first of these calls—the rest will be `wrpkru`-free. Likewise, if a recursive function has been optimized using tail-call optimization, its caller will use a `jmp` instruction instead of a `call`. In this case, the return address has already been pushed to the shadow stack; there is no need to write it again.

To leverage this observation, we designed the *Shadow Write Optimization (SWO)*. With SWO, IskiOS adds code to every callsite that first checks whether the word in the shadow stack to which the return address will be written already contains the return address. If it does, the word is not written a second time. Experiments in Section 7 confirm that the reduction in dynamic executions of `wrpkru` leads to a dramatic overall improvement in performance.

5.2 Aggressive Inlining

A more obvious optimization opportunity is to minimize the number of function calls, which in turn eliminates the need to save the return address on the shadow stack. To achieve this, we aggressively increased the inlining threshold option to the compiler until we saw no additional performance improvement. Experiments in

Section 7 confirm that aggressive inlining almost always leads to better performance at the cost of increased code size.

6 IMPLEMENTATION

We implemented IskiOS as a set of patches to the Linux v5.7 kernel and the Clang/LLVM [55] 10.0.0 compiler.

6.1 Kernel Modifications

In our IskiOS prototype, we did not implement the debug watchpoint arming described in Section 3.1. We did, however, use Hodor's code scanner [40] to confirm the absence of `wrpkru` instructions in the binaries used in Section 7.2. We also did not implement the changes to physmap described in Sections 4.2 and 4.3, but we do not expect these to impact performance. Finally, we carefully considered the PKU vulnerabilities described by Connor et al. [16]. We concluded that, while the design of IskiOS is resistant to these attacks, our prototype implementation would require additional engineering to resist certain attacks on memory permissions (e.g., using `process_vm_readv()` [58] to read data from a process's own address space, ignoring the PKU permissions). Fixing the IskiOS prototype to make it complete (and therefore resistant to these attacks) requires that we add simple checks to the relevant memory-related system calls. The changes are straightforward, and we expect their performance impact to be negligible. We built IskiOS as three separate patches to the Linux kernel, adding support for protection keys, execute-only memory, and shadow stacks, respectively. We describe each patch below.

IskiOS-PKK. This patch enables PKU for all of virtual memory. We mark all pages as user mode by setting the U/S bit in every PTE. Since our design associates PKEYs 0–7 with kernel space and PKEYs 8–15 with user space, we change the default protection key for user pages from 0 to 8. By default, the kernel can access only pages with PKEY 0 (and PKEYs 8–15 when it wants to access user memory); user processes can access only pages with PKEY 8.

We also modified Linux to save and restore the `pkru` on kernel entry and exit. As an optimization, we arranged for interrupt dispatch code to elide the `wrpkru` instruction if the `pkru` already holds the correct value; this improves performance if a trap or interrupt occurs while the OS kernel is running. We disable this optimization in the shadow stack patch, as IskiOS must change `pkru` on every interrupt, trap, and system call.

IskiOS-XOM. This patch changes the 4-bit protection key in every PTE that maps a kernel code page to the value 1. Read and write access to pages with PKEY 1 is already disabled in the `pkru`.

IskiOS-SS. This patch (1) doubles (from 16 to 32 KB) the size of every stack in the kernel (i.e., every per-thread stack, per-cpu interrupt stack, etc.) and uses the upper half as a shadow stack, with write access disabled using PKK; (2) modifies kernel entry/exit code to save/restore the `pkru` register to/from the shadow stack, and (3) updates assembly code to conform to the new calling convention (Sec. 4.3). After updating the standard `CALL_NOSPEC` macro, we had to manually modify fewer than 100 additional call sites.

Listing 3: SMEP: Indirect Branch Instrumentation.

```
1  ; check most significant bit of target address
2  bt $63, %TargetReg
3
4  ; if address is in kernel space, skip exception
5  jb L1
6
7  ; raise exception
8  ud2
9
10 L1:
11 ; indirect call/jmp *TargetReg
```

Listing 4: SMEP: Epilogue Instrumentation.

```
1  ; load the return address to %r10
2  movq    [%rsp], %r10
3
4  ; check most significant bit of target address
5  bt $63, %r10
6
7  ; if address is in kernel space, skip exception
8  jb L1
9
10 ; raise exception
11 ud2
12
13 L1:
14 ; adjust %rsp
15 addq 0x8, %rsp
16
17 ; return
18 jmpq *%r10
```

6.2 Compiler Instrumentation

We implemented two separate plugins to the LLVM 10.0.0 compiler [55] to implement our prototype. Both extend the LLVM code generator with *MachineFunction* passes: one adds control-flow assertions on indirect branches and return instructions to enable `SMEP` functionality; the other implements shadow stacks.

6.2.1 SMEP Implementation. As suggested in Section 3.2, Listing 3 shows instrumentation inserted before each indirect branch. Assuming that `%TargetReg` contains the target address, the plugin prepends the indirect `call` or `jmp` (line 11) with a `bt` (line 2) instruction that stores the most significant bit of the target address in the `CF` flag of the status register. A `jb` (line 5) instruction continues regular execution if the address was in kernel space; otherwise, an undefined instruction `ud2/0x0f0b` (line 8) will raise an invalid opcode exception, and the kernel will crash.

Our `SMEP` plugin also instruments function epilogues to ensure that functions do not return to user space. As Listing 4 shows, the plugin loads the target address to `%r10` (line 2), performs the same test as before (lines 5–11), adjusts the `%rsp` (line 15) to emulate a function return, and replaces the `ret` instruction with a `jmp` through `%r10` (line 18).

6.2.2 Shadow Stack Implementation. Our second compiler plugin transforms every callsite to save the return address on the shadow stack and every function epilogue to use the return value on the shadow stack as Section 4.3 describes. To support our new calling convention, we modified the compiler to reserve register `%r10`.

Listing 5: Shadow Stack Callsite Instrumentation.

```
1    | ; skip shadow write if address already present
2    |  cmp  %r10, [%rsp-4*PGSIZE]
3    |  je   Lcall
4
5  ; spill %rax, %rcx, %rdx
6
7  ; flip shadow stack pkru bit to enable write access
8  xor rcx, rcx
9  rdpkru
10 xorl    0x8, %eax
11 wrpkru
12
13 ; ensure execution is in kernel mode
14 movq    %cs, %rcx
15 testb   $3, %cl
16 je      Lskip1
17 Ltrap1:
18 ud2
19
20 Lskip1:
21 ; ensure access to xom has not been enabled
22 cmpb 0xc, %ax
23 jne Ltrap1
24
25 ; copy ret. addr. to shadow stack
26 movq    %r10, [%rsp-4*PGSIZE]
27
28 ; flip shadow stack pkru bit to disable write acces
29 xorl    0x8, %eax
30 wrpkru
31
32 ; ensure execution is in kernel mode
33 movq    %cs, %rcx
34 testb   $3, %cl
35 je      Lskip2
36 Ltrap2:
37 ud2
38
39 Lskip2:
40 ; ensure access to xom has not been enabled
41 cmpb 0xc, %ax
42 jne Ltrap2
43
44 ; restore spilled regs
45
46 Lcall:
47 ; copy new ret. addr. to %r10
48 leaq    Lret(%rip), %r10
49
50 ; actual callsite
51 callq   *%rbx
52 Lret:
53
54 ; restore old ret. addr. to %r10
55 movq    [%rsp-4*PGSIZE], %r10
```

Listing 6: Shadow Stack Epilogue Instrumentation.

```
1 ; adjust %rsp
2 addq 0x8, %rsp
3
4 ; return
5 jmpq *%r10
```

Listing 5 shows IskiOS's callsite instrumentation with and without SWO. When the shadow-write optimization is disabled, our compiler pass enables access to the shadow stack at each callsite by using wrpkru to clear the write-disable (WD) bit for PKEY 3 (lines 8–11). It then copies the return address of the currently executing function to the shadow stack (line 26) and executes another wrpkru (line 30) to disable write access again. Note that since rdpkru and wrpkru force us to zero %ecx and %edx, the compiler must spill (line 5) and restore (line 44) these registers. Before the call instruction (line 51), our compiler loads the callee's return address into %r10 (line 48). (The return address saved to the stack by the call instruction is never used.) After the call site, the compiler restores the return address in %r10 from the shadow stack (line 55). When SWO is enabled, the compiler adds a simple check (lines 2–3) which ensures that the return address is pushed onto the shadow stack only if the value already there differs.

Listing 6 shows IskiOS's instrumentation of function epilogues. The epilogue code adjusts the stack pointer (line 2) to discard the return address on the main stack and performs an indirect jump to the address in %r10 (line 5).

7 EVALUATION

We evaluated the performance overhead that IskiOS incurs for its PKK-based user/kernel isolation mechanism, execute-only memory, and protected shadow stacks. We also evaluated the shadow-write and inlining optimizations described in Section 5. Finally, we evaluated the impact of IskiOS on the size of the kernel code segment. We report numbers for the following configurations:

- **vanilla**: Unmodified Linux kernel v5.7
- **pkk**: IskiOS kernel with protection keys for kernel space
- **pkk+xom**: IskiOS with PKK and execute-only memory
- **pkk+ss**: IskiOS with PKK and unoptimized shadow stacks
- **pkk+ss-swo**: IskiOS with PKK, shadow stacks, and the shadow-write optimization (SWO)
- **pkk+ss-swo-inline**: IskiOS with PKK, shadow stacks with SWO, and an increased inlining threshold
- **pkk+xom+ss-swo-inline**: IskiOS with PKK, XOM, shadow stacks with SWO, and an increased inlining threshold

In all the above configurations, we disabled Kernel Address Space Layout Randomization (KASLR) [27], Kernel Page Table Isolation (KPTI) [39], and hardware SMAP and SMEP. We used the LMBench suite [63] for microbenchmarking and the Phoronix Test Suite (PTS) [64] to measure the performance impact on real-world applications. We performed our experiments on a Fedora 32 system equipped with two 3.00 GHz Intel Xeon Silver 4114 (Skylake) CPUs (2×10 cores, 40 total threads), 16 GB of RAM and a 1 TB Seagate 7200 RPM disk. For all our tests, we loaded the intel_pstate performance scaling driver into the kernel to prevent the processor from reducing frequency (for power saving) during our experiments. For the networking experiments, we ran client and server processes on the same machine. We used the default settings for all benchmarks.

7.1 Microbenchmarks

We used LMBench [63] v.3.0-a9 to measure the latency and bandwidth overheads imposed by IskiOS on basic kernel operations. In particular, our set of microbenchmarks measures the latency of critical I/O system calls (open()/close(), read()/write(), select(), fstat(), stat(), mmap()/munmap()), execution mode switches (null system call), and inter-process context switches. We

also measured the impact on process creation followed by `exit()`, `execve()`, and `/bin/sh`, as well as the latency of signal installation (via `sigaction()`) and delivery, protection faults, and page faults. Finally, we measured the latency overhead on pipe and socket I/O (TCP, UDP, and Unix domain sockets) and the bandwidth degradation on pipes, sockets, and file and memory-mapped I/O. We report the arithmetic mean of 10 runs for each microbenchmark.

7.1.1 IskiOS's Performance. Figures 1 and 2 present our results for latency and bandwidth microbenchmarks, respectively (absolute performance numbers and standard deviations appear in Tables 4 and 5 in Appendix A). First we discuss IskiOS's effect on the latency of basic kernel operations. Figure 1a shows that baseline support for PKK incurs low overhead for most microbenchmarks relative to `vanilla` Linux: roughly 11% geometric mean and a maximum of ∼25% for the `write` system call. PKK incurs higher overheads on lightweight system call tests. For example, the `null` test calls `getppid()`, a system call that does minimal work inside the kernel, in a loop to estimate the round-trip latency of entering and exiting the kernel. The `read` test reads a single byte from `/dev/zero`, and the `write` test writes one byte to `/dev/null`, estimating a lower bound on the latency of any user/kernel interaction. PKK adds less than ten instructions to the OS kernel entry/exit path and two instructions for every indirect branch in kernel code. As expected, the marginal impact on kernel operations (except for extremely small services as described above) is negligible.

IskiOS's XOM incurs virtually no overhead (when accounting for standard error) compared to PKK, and an average (geomean) of ∼12% for latency microbenchmarks compared to `vanilla` Linux. The `pkk+ss-swo-inline` kernel (fully optimized shadow stack, with SWO and aggressive inlining) incurs ∼21% overhead compared to `vanilla` and 10% compared to PKK. Finally, the `pkk+xom+ss-swo-inline` kernel, with both XOM and a fully-optimized shadow stack, incurs ∼22% overhead compared to vanilla Linux.

Figure 2a shows the effect of IskiOS on the bandwidth of various inter-process communication mechanisms. IskiOS imposes the most overhead—roughly 16% over `vanilla`—when moving data over a TCP/IP socket with both XOM and a fully-optimized shadow stack enabled. The complexity of the underlying TCP/IP protocol combined with the speed of network communication to localhost explains this result: TCP performs congestion, flow, and transmission-error control, and the network stack performs many function calls both within and between layers. These calls increase the number of instructions executed (including the relatively expensive `wrpkru`) when the shadow stack is enabled, and there is no latency from a real network to hide the additional overhead.

7.1.2 PKK Overhead Breakdown. To assess the contribution to PKK overhead of our software-based SMEP support, we disabled the LLVM SMEP pass and reevaluated PKK's performance on the LMBench microbenchmarks. Due to space, we summarize the results; Appendix A contains full details for interested readers. Our results show that our SMEP instrumentation causes most of PKK's overhead: disabling SMEP reduces the geometric mean PKK overhead from 11% to 5% on the LMBench latency benchmarks and from 2% to 1% on the LMBench bandwidth benchmarks. We believe this overhead is due to the extra instructions added to every indirect branch and the loss of utilizing the processor's return predictor.

7.1.3 Shadow Stack Optimizations. Figures 1b and 2b assess the importance of our shadow-write and inlining optimizations for the IskiOS shadow stack. The unoptimized shadow stack implementation (`pkk+ss`) adds an `rdpkru` and two `wrpkru` instructions to every callsite in a function, incurring a maximum overhead of nearly 520% for the `select100tcp` latency microbenchmark. Across all latency microbenchmarks, the unoptimized shadow stack incurs a geometric mean overhead of ∼217% compared to our baseline. These results are consistent with the recent work of Burow et. al [9], who evaluate various ISA features, including Intel PKU, for protecting a user-space shadow stack. In particular, Burow et. al report a maximum overhead of ∼420% on the `SPEC CPU2006` [41] benchmark suite when using Intel PKU to write-protect the shadow stack.

When shadow-write optimization (SWO) is enabled, however (i.e., in the `pkk+ss-swo` kernel), overheads decrease dramatically, to a maximum of ∼97% and a geometric mean of ∼37%. When both SWO and the inlining optimization are enabled (`pkk+ss-swo-inline`), the overheads drop to a maximum of ∼43% and a geomean of ∼21%. While differences in control flow and instruction mix make it difficult to directly compare user and kernel workloads, our results nonetheless demonstrate the viability of PKU for intra-address-space isolation, even for mechanisms like shadow stacks, which require frequent domain switching. In particular, it should be possible to significantly improve the performance of user-space shadow stacks [9] using SWO and (perhaps) more aggressive inlining.

7.2 Macrobenchmarks

To assess the overhead of IskiOS on real-world programs, we used the Phoronix Test Suite [64] v9.8.0 (Nesodden). Phoronix is an open-source automated benchmarking suite with over 450 different test profiles grouped into categories such as disk, network, processor, graphics, and system.

We chose the set of tests used to track the performance of the mainline Linux kernel [59]. We omitted stress tests, which resemble the LMBench microbenchmarks of Section 7.1.1. Our chosen tests cover (a) web servers (`Apache` and `NGNIX`); (b) the build toolchains for Apache (`Build Apache`) and PHP 7 (`Build PHP`); (c) encryption software (`Cryptsetup-sha215`, `Cryptsetup-whirlpool`, and `OpenSSL`); (d) the PHP interpreter (`PHPBench`) and LuaJIT compiler (`LuaJIT`); (e) the *PostgreSQL* object-relational database search (`PGBench`) and a protein sequence database searched via Hidden Markov Models (`HMMer`); (f) web and mail server workloads (`Ebizzy` and `PostMark`); (g) filesystem and disk I/O workloads (`FS-Mark` and `CompileBench`); (h) video encoding (`SVT-AV1` and `SVT-HEVC`); (i) audio encoding (`FLAC` and `MP3`); (j) file compression (`Parallel BZIP2`); (k) game engines for chess (`Crafty`, `Stockfish`) and Connect-4 (`Fhourstones`); (l) computation-heavy prime number generation (`Primesieve`), Poisson equation solving (`Himeno`), and matrix multiplication (`PolyBench-C`); (m) system boot-up performance (`Systemd Kernel`); (n) 3D computer graphics raytracing (`C-Ray`) and rendering (`TTSIOD 3D Renderer`); and (o) version control (`Git`).

Phoronix keeps running a benchmark until the relative standard deviation (RSD) falls below a specific threshold (3.5% by default) or a maximum number of runs (15 in our experiments) is exhausted. Table 1 presents IskiOS's overhead on each benchmark. The second column shows the metric used by each benchmark and the result

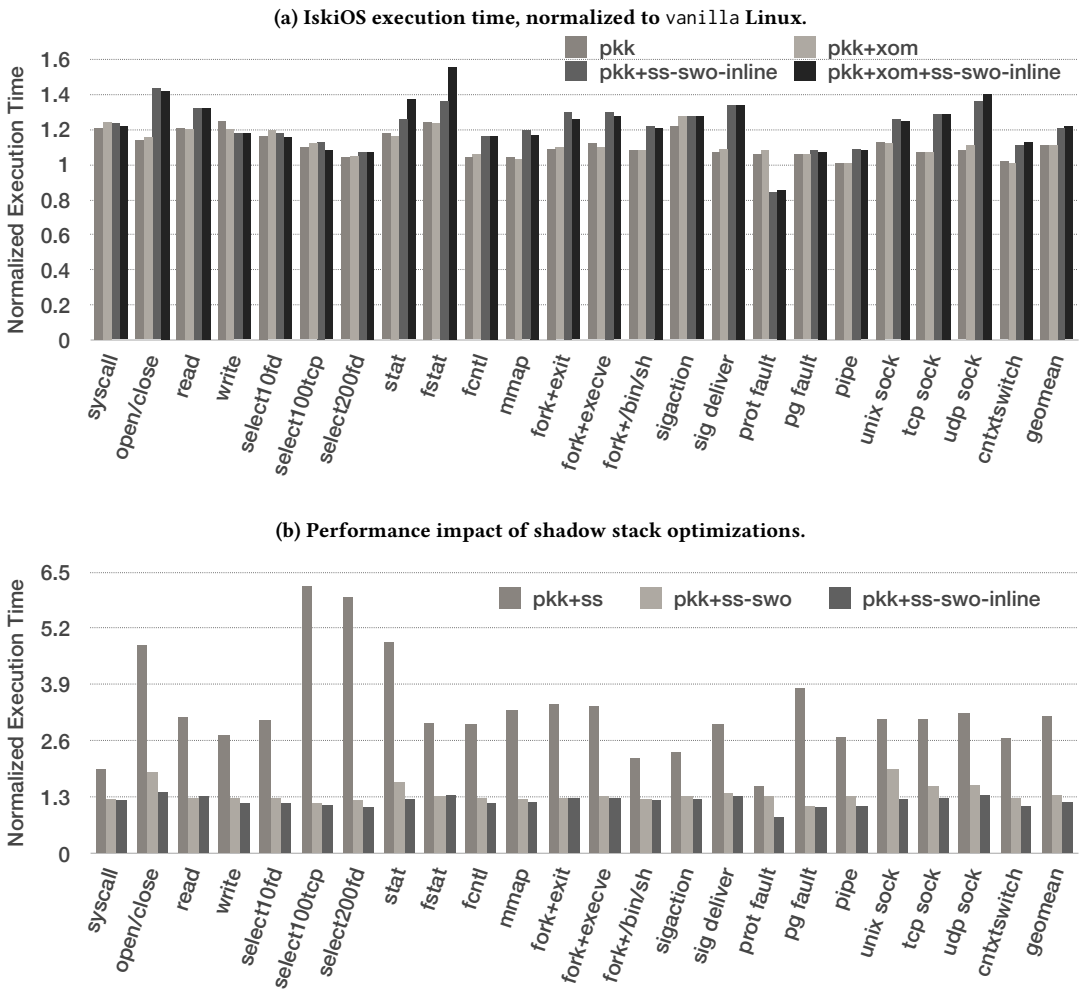

Figure 1: Evaluation of IskiOS's performance on latency microbenchmarks of the LMBench suite.

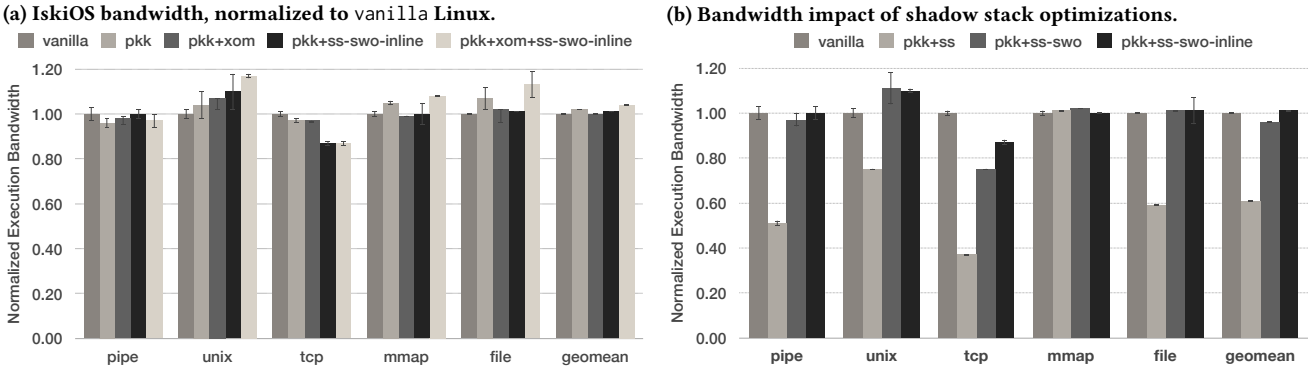

Figure 2: Evaluation of IskiOS's performance on bandwidth microbenchmarks of the LMBench suite. The error bars represent the normalized standard deviation (SD) of 10 runs for each benchmark.

Table 1: IskiOS run-time overhead (% over `vanilla` Linux) on applications from the Phoronix Test Suite. The final row reports average change in the reported metric (geometric mean).

Benchmark	vanilla		pkk		pkk+xom		pkk+ss-swo-inline		pkk+xom+ss-swo-inline	
			Mean ± RSD	Overhead	Mean ± RSD	Overhead	Mean ± RSD	Overhead	Mean ± RSD	Overhead
Apache	24626 ± 2.20%	*req/s*	20531 ± 1.30%	16.63%	21294 ± 1.81%	13.53%	16048 ± 0.95%	34.83%	15141 ± 2.40%	38.51%
Build Apache	33.24 ± 1.39%	*sec*	33.38 ± 0.42%	0.43%	33.58 ± 0.22%	1.04%	33.82 ± 0.56%	1.75%	34.08 ± 0.45%	2.54%
Build PHP	71.00 ± 0.47%	*sec*	71.95 ± 1.68%	1.34%	71.83 ± 0.56%	1.17%	72.34 ± 0.27%	1.90%	72.41 ± 0.47%	1.99%
C-Ray	148.30 ± 46.61%	*sec*	117.72 ± 28.45%	−20.62%	114.85 ± 43.71%	−22.56%	155.53 ± 22.89%	4.88%	181.50 ± 27.93%	22.39%
CompileBench	296.84 ± 6.59%	*MB/s*	247.43 ± 3.26%	16.65%	258.08 ± 1.23%	13.06%	219.27 ± 4.19%	26.13%	215.28 ± 3.43%	27.48%
Crafty	5038443 ± 0.25%	*nodes/s*	5016383 ± 0.17%	0.44%	5031947 ± 0.06%	0.13%	5042075 ± 0.04%	−0.07%	5035883 ± 0.09%	0.05%
Cryptsetup-sha512	1016062 ± 0.00%	*iter/s*	1010211 ± 0.58%	0.58%	1014096 ± 0.00%	0.19%	1012801 ± 0.40%	0.32%	1008968 ± 1.05%	0.70%
Cryptsetup-whirlpool	426949 ± 0.43%	*iter/s*	424871 ± 0.32%	0.49%	425561 ± 0.32%	0.33%	426020 ± 0.19%	0.22%	426944 ± 0.00%	0.00%
Ebizzy	524560 ± 3.49%	*records/s*	515582 ± 0.61%	1.71%	531721 ± 4.19%	−1.37%	537828 ± 4.07%	−2.53%	514557 ± 2.08%	1.91%
Encode FLAC	18.69 ± 3.30%	*sec*	18.42 ± 1.08%	−1.44%	18.93 ± 2.27%	1.28%	19.24 ± 3.01%	2.98%	18.64 ± 2.44%	−0.23%
Encode MP3	16.95 ± 0.23%	*sec*	16.95 ± 0.27%	0.02%	16.96 ± 0.28%	0.07%	16.95 ± 0.21%	0.02%	16.95 ± 0.31%	0.05%
FS-Mark	24.7 ± 0.84%	*files/s*	24.7 ± 3.28%	0.00%	24.8 ± 0.23%	−0.40%	24.3 ± 0.68%	1.62%	22.5 ± 0.26%	8.91%
Fhourstones	8816.50 ± 0.02%	*Kpos/s*	8812.30 ± 0.28%	0.05%	8797.00 ± 0.33%	0.22%	8834.60 ± 0.04%	−0.21%	8807.50 ± 0.08%	0.10%
Git	88.74 ± 0.35%	*sec*	88.87 ± 0.27%	0.15%	89.03 ± 0.20%	0.33%	89.37 ± 0.44%	0.72%	89.94 ± 0.63%	1.35%
HMMer	189.69 ± 0.19%	*sec*	190.35 ± 0.04%	0.35%	190.41 ± 0.13%	0.38%	193.30 ± 0.14%	1.90%	193.43 ± 0.15%	1.97%
Himeno	2113.80 ± 0.08%	*Mflops*	2099.09 ± 0.60%	0.70%	2112.68 ± 0.16%	0.05%	2112.08 ± 0.06%	0.08%	2112.72 ± 0.22%	0.05%
LuaJIT	912.22 ± 0.50%	*Mflops*	912.02 ± 0.34%	0.02%	904.81 ± 0.11%	0.81%	915.87 ± 0.32%	−0.40%	915.56 ± 0.10%	−0.37%
NGINX	26850 ± 1.96%	*req/s*	22981 ± 0.52%	14.41%	22646 ± 1.53%	15.66%	17901 ± 5.89%	33.33%	16878 ± 0.20%	37.14%
OpenSSL	3788 ± 0.23%	*signs/s*	3693 ± 0.45%	2.42%	3792 ± 0.19%	−0.11%	3792 ± 0.13%	−0.13%	3791 ± 0.27%	−0.08%
PGBench	681 ± 6.64%	*trans/s*	695 ± 6.90%	−2.06%	679 ± 8.76%	0.29%	649 ± 1.68%	4.70%	634 ± 6.59%	6.90%
PHPBench	405719 ± 0.13%	*score*	405729 ± 0.10%	0.01%	405094 ± 0.25%	0.15%	406372 ± 0.15%	−0.16%	405861 ± 0.10%	−0.03%
Parallel BZIP2	4.36 ± 16.34%	*sec*	4.36 ± 16.35%	0.00%	4.37 ± 16.66%	0.18%	4.24 ± 18.27%	−2.75%	4.30 ± 17.66%	−1.38%
PolyBench-C	5.03 ± 4.12%	*sec*	4.91 ± 4.50%	−2.25%	4.91 ± 4.88%	−2.29%	4.82 ± 2.46%	−4.16%	4.83 ± 2.18%	−3.90%
PostMark	4548 ± 3.10%	*trans/s*	4121 ± 0.95%	9.39%	4237 ± 0.00%	6.84%	3607 ± 2.18%	20.69%	3641 ± 0.84%	19.94%
Primesieve	15.79 ± 0.87%	*sec*	16.03 ± 0.65%	1.53%	15.86 ± 0.20%	0.46%	15.80 ± 0.44%	0.08%	15.76 ± 0.26%	−0.16%
SVT-AV1	2.95 ± 1.87%	*frames/s*	2.96 ± 0.87%	−0.24%	2.92 ± 1.42%	1.22%	2.95 ± 1.53%	−0.10%	2.99 ± 1.99%	−1.22%
SVT-HEVC	57.45 ± 10.37%	*frames/s*	57.05 ± 10.66%	0.70%	57.04 ± 10.47%	0.71%	56.65 ± 10.30%	1.39%	56.70 ± 10.23%	1.31%
Stockfish	27.87M ± 3.24%	*nodes/s*	28.17M ± 1.26%	−1.08%	27.11M ± 2.43%	2.74%	27.59M ± 2.64%	1.01%	28.12M ± 3.03%	−0.91%
Systemd Kernel	2975 ± 0.00%	*msec*	2985 ± 0.00%	0.34%	3065 ± 0.00%	3.03%	3252 ± 0.00%	9.31%	3220 ± 0.00%	8.24%
TTSIOD 3D Renderer	319.34 ± 1.60%	*frames/s*	383.37 ± 7.65%	−20.05%	377.98 ± 10.90%	−18.36%	361.43 ± 8.35%	−13.18%	352.00 ± 9.29%	−10.23%
Geometric Mean Overhead				0.39%		0.36%		3.67%		4.87%

Table 2: Macrobenchmarks with highest overhead for `PKK`; top four system calls for each, with percent of CPU time spent running in the kernel.

Benchmark	Syscall	%CPU	Syscall	%CPU	Syscall	%CPU	Syscall	%CPU
CompileBench	`write()`	60.57%	`openat()`	15.49%	`fstat`	10.77%	`close()`	5.86%
Apache - server	`futex()`	91.00%	`read()`	4.11%	`epoll_wait()`	1.87%	`accept4()`	0.45%
Apache - client	`epoll_ctl()`	23.29%	`connect()`	18.58%	`write()`	15.05%	`read()`	13.16%
NGINX - server	`close()`	14.61%	`recvfrom()`	12.91%	`writev()`	11.79%	`sendfile()`	10.75%
NGINX - client	`epoll_ctl()`	29.65%	`connect()`	18.95%	`write()`	13.47%	`read()`	10.70%
PostMark	`write()`	52.27%	`read()`	36.67%	`unlink()`	6.05%	`openat()`	2.34%

on the `vanilla` kernel (i.e., our baseline). Columns 3–6 show the percentage overhead of each configuration over the baseline kernel. The final row shows the geometric mean, across all applications, of the change in reported metric $((\prod_i (overhead_i + 100))^{1/n} - 100)$. As different applications use different metrics, this average should be taken as merely suggestive; nonetheless, overhead of less than 5% when running on the full version of IskiOS, with XOM and a race-free protected shadow stack, strikes us as a very strong result.

For `PKK` alone, we observe a maximum overhead of ~17% for `CompileBench` and `Apache`, followed by ~14% for `NGINX` and ~9% for `PostMark`. Table 2 presents the top four system calls with respect to system time—i.e., CPU time in the kernel—for each of these four macrobenchmarks, as reported by `strace`. `CompileBench` tests file system performance by simulating kernel builds (creating, compiling, patching, stating, and reading kernel trees). We examined the initial creation of 1000 1 MB files. Table 2 shows that ~61% of system time is spent on `write()`, followed by `openat()`, `fstat()`, and `close()`. As Section 7.1 explains, IskiOS's `PKK` affects these calls the most, as their total execution time is small. Since `CompileBench` uses extremely small services frequently, the overhead of `PKK` is higher. `PostMark` simulates the behavior of mail servers and thus of small-file testing (file sizes of 5–512 KB). Table 2 shows that `PostMark` spends most of its system time (~52%) in `write()`, followed by `read()`, `unlink()`, and `openat()`. As in `CompileBench`, frequent use of such short kernel calls leads to higher overheads.

For `NGINX` and `Apache`, we used `strace` to examine both client and server system calls. Table 2 shows that the `Apache` server spends 91% of its system time in `futex()`, followed by `read()`, `epoll_wait()`, and `accept4()`. The `NGINX` server spends roughly the same amount of system time among `close()`, `recvfrom()`, `writev()`, and `sendfile()`. Table 2 also shows that clients in both cases spend most of their system time in `epoll_ctl()`, `connect()`, `write()`, and `read()`. This is expected, as the two benchmarks share the `Apache Bench` (`ab`) client. Calls to `futex()` are used for synchronization and are typically very brief. The `connect()` system call is used to initiate a connection over a socket and `accept4()` to

Table 3: IskiOS Code Size Overheads

Kernel	Code Size	Overhead
vanilla	28.01 *MB*	–
pkk	30.2 *MB*	7.82%
pkk+xom	30.2 *MB*	7.82%
pkk+ss	71.5 *MB*	155.27%
pkk+ss-swo	73.75 *MB*	163.30%
pkk+ss-swo-inline	135.7 *MB*	384.47%
pkk+xom+ss	71.5 *MB*	155.27%
pkk+xom+ss-swo	73.75 *MB*	163.30%
pkk+xom+ss-swo-inline	135.7 *MB*	384.47%

accept such connections. The `recvfrom()` and `sendfile()` system calls are used to receive and send data over a socket, respectively. As Section 7.1 explains, PKK incurs significant overhead on such services, especially when running TCP (as both NGINX and Apache do). The `epoll_ctl()` and `epoll_wait()` system calls are used to synchronize communication over a set of file descriptors, and much like `select()` (discussed in Sec. 7.1), they suffer high overhead when the number of descriptors is high.

Within the bounds of experimental error, IskiOS's XOM implementation incurs no overhead beyond that of PKK. The fully-optimized shadow stack implementation (`pkk+ss-swo-inline`), as expected, increases the overheads for macrobenchmarks, such as web servers and file-system workloads, that spend significant time in the kernel. The maximum overhead across all benchmarks and configurations is 38.51%; it occurs for the Apache web server when both security defenses are enabled (`pkk+xom+ss-swo-inline`). NGINX is similar (maximum overhead of 37.14%).

7.3 Code Size Overhead

To see IskiOS's impact on code size, we measured the `.text` section in the final binary for each configuration (including all the loaded modules). Results appear in Table 3. IskiOS's PKK instrumentation has relatively little impact—7.82%—as it only adds a few lines of code in the kernel entry/exit path and two instructions for every indirect branch.

As expected, IskiOS's XOM does not increase the code size relative to PKK. In contrast, SFI-based XOM approaches, which instrument every load instruction, have higher memory overheads: existing implementations of execute-only memory for user-level applications e.g., LR^2 [6] and uXOM [53], report code-size increases of 10% to 50%. Unfortunately, kR^X [71], the only SFI-based approach for XOM in the Linux kernel, does not report code-size overheads.

IskiOS's unoptimized shadow stack implementation incurs a 155% increase in code segment size. This is expected, as our shadow stack compiler pass adds 25 instructions to each callsite (Listing 5) and one instruction to each function epilogue (Listing 6). The shadow-write optimization (SWO) adds two more instructions per callsite (Listing 5), for another ~8% overhead compared to the unoptimized shadow stack and 163% compared to our baseline. XOM again adds nothing more to the code size of the shadow stack implementation. Finally, increasing Clang's inlining threshold to 8000 adds roughly 385% overhead compared to the baseline, which is also the cumulative overhead when both security defenses are enabled. While this increase is high, the absolute size of the kernel along with the loaded modules in IskiOS is less than 136 MB, which we consider entirely reasonable given the added security and the fact that modern desktop and server machines come with 8 GB or more of physical memory.

8 RELATED WORK

We discuss three areas of related work: intra-address-space memory isolation, execute-only memory defenses, and methods to protect the integrity of return addresses.

8.1 Intra-address-space Isolation

Address-based Isolation Several research efforts [11, 22, 62, 71, 89] employ *software fault isolation* (SFI) [76, 86] to protect sensitive data from untrusted code. Typically, these approaches add a run-time check on each memory access to ensure that the target address is not within the protected region. Additionally, they enforce some form of *control-flow integrity* (CFI) [1] to ensure that the SFI instrumentation is not bypassed. As the protection overhead is proportional to the complexity of the code performing the checks, solutions using SFI typically 1) protect only a single memory region, and 2) place pages belonging to the same protection domain contiguously within the virtual address space [11, 22, 71, 89]. Unfortunately, this approach restricts the number of protected domains and requires significant engineering effort (memory allocator modifications in particular [71]). In contrast, our PKK supports up to 8 distinct protection domains and permits pages in different domains to be located anywhere within the virtual address space without incurring additional performance loss.

Domain-based Isolation Other approaches [36, 40, 67, 80, 83, 87, 88] use hardware isolation primitives to create *protection domains* in which various executable components are granted asymmetric access to sensitive data in memory. Hodor [40], ERIM [83], and MonGuard [87] use PKU to provide efficient intra-process isolation for user-space programs. Like IskiOS, these systems use memory protection keys. However, IskiOS provides intra-kernel isolation and, consequently, solves several challenges such as retrofitting PKU for protecting kernel memory while maintaining SMAP and SMEP. Sung et al. [80] leverage Intel PKU to provide intra-address-space isolation for the RustyHermit [54] unikernel. A RustyHermit [54] unikernel instance consists of a single application, statically linked against a minimal library OS written in Rust [54]; application and kernel share the same *user address space* (i.e., pages with U/S bit set), and they both execute in *kernel mode*. Sung et al.'s [80] proposed mechanism places the application in a different protection domain from the kernel, and it isolates unsafe Rust code blocks that handle low-level hardware operations from the rest of the kernel. As all code uses user-space memory pages, the use of Intel PKU is straightforward. Since the goal is to increase fault tolerance within a single domain of trust, Sung et al. [80] do not prevent reconfiguration of the pkru register. Even if they did, their system would not need to differentiate between `wrpkru` instructions in user and kernel space. In contrast, IskiOS retrofits Linux's existing user/kernel isolation mechanism to use protection keys. In doing so, IskiOS takes additional measures to ensure that user-space application code cannot misuse the `wrpkru` instruction to break IskiOS's isolation mechanism. UnderBridge [36], developed concurrently to our work, restructures microkernel OSes by moving user-space system servers to kernel space; it uses PKU in conjunction with

Kernel Page Table Isolation (KPTI) [39] to isolate system servers running in kernel mode. While compatible with KPTI, our PKK mechanism does not require it, and therefore PKK does not incur KPTI's additional overhead [7].

Other x86 hardware primitives can be used for intra-address-space isolation. Several works [40, 51, 61, 65, 67, 72] use Intel's `VMFUNC` [46] to switch between preset *Extended Page Tables (EPTs)* that correspond to different protection domains. Among them, SkyBridge [65] uses `VMFUNC` to improve the latency of IPCs in a microkernel setting, LVDs [67] use `VMFUNC` to isolate device drivers from the core Linux kernel, and xMP [72] uses the same mechanism to protect sensitive user and kernel data from data-only attacks. `VMFUNC` requires that these systems execute under a hypervisor, which incurs non-trivial virtualization overheads on normal execution. Conversely, IskiOS uses no hypervisor, keeping its overheads during normal execution low. Additionally, as `VMFUNC` is 3–5× slower than `wrpkru` [40, 83, 88], its use for defenses requiring frequent domain switching (e.g., shadow stacks) is prohibitive [83, 88].

SEIMI [88] uses *SMAP* [46] to protect sensitive data in user-space programs. It enables SMAP and maps all pages with the supervisor bit except for those that store sensitive user data. SEIMI then executes *all* (trusted and untrusted) user code in privileged mode and allows trusted components to temporarily disable SMAP to access the protected memory. To prevent user code from executing privileged instructions, SEIMI places the system under the control of a hypervisor and forces all privileged instructions to cause a VM exit. Both IskiOS and SEIMI make unconventional use of x86 hardware to efficiently isolate memory. Unlike SEIMI, IskiOS does not require a hypervisor to protect its isolation mechanism and security defenses. In addition, SEIMI's use of SMAP no longer protects the kernel from inadvertently accessing user memory. In contrast, IskiOS's PKK mechanism still allows user programs to use PKU (though it does halve the number of user-level keys available).

8.2 eXecute-Only Memory

Defenses in user space prevent code pages from being read [4, 6, 21, 33, 53] and use code-pointer hiding techniques [6, 21] to protect against indirect memory disclosure attacks. Inspired by their user-space counterparts, recent approaches in the OS kernel [32, 71] combine execute-only memory with kernel diversification to prevent gadgets from being leaked. KHide [32] uses a hypervisor to prevent read accesses to kernel code. IskiOS does not require more privileged software for its execution, keeping its trusted computing base small and avoiding unnecessary virtualization overheads. kR^X [71] instruments all read instructions with run-time checks to ensure that they never read the code segment. In its fully-optimized software-based implementation, kR^X performs roughly the same as IskiOS; however, kR^X must place all code in a contiguous region, weakening diversification schemes such as KASLR [27]. To make up for the entropy loss, kR^X re-arranges code in the protected region using function permutation and basic block reordering [71]. In contrast, IskiOS does not break the memory layout, provides more flexibility (i.e., supports more than one protected area in memory), and preserves the protection guarantees of existing randomization schemes. A faster implementation of kR^X exists but relies on Intel MPX which is now deprecated [44].

8.3 Return Address Protection

Stack canaries [20, 28, 85] and systems that encrypt return addresses [19, 43, 57, 68, 71, 82] can detect return address corruption on the stack but are susceptible to memory disclosure [19], brute-force [57], signing-key forgery [90], and substitution attacks [43, 57]. LLVM Safestack [14] stores return addresses on a second separate stack which it stores in a randomized location. Other defenses create a separate (shadow) stack in memory and, on each function call, store a copy of the return address to the shadow stack [9, 13, 23, 84]. Shadow stacks may or may not be write-protected. Systems deploying the latter (e.g., Shadesmar [9]) place the shadow stack at a random location in memory like the LLVM Safestack [14]. While these solutions make attacks more difficult—the base address needs to be guessed correctly and, in the case of shadow stacks, two return addresses must be corrupted in memory—they are still susceptible to information leaks and memory corruption attacks [9, 14, 29, 91].

Write-protected shadow stacks (e.g., Read-Only RAD [13]), if implemented correctly to be thread-safe, offer the strongest protection for return addresses. Prior works [9, 88] examine the performance overheads of various hardware mechanisms to preserve the integrity of a user-space shadow stack. Burow et al. [9] report a 12.12% geomean overhead over the SPEC CPU2006 [41] benchmark suite when using Intel MPX [46] and a 61.18% geomean overhead when using Intel PKU. Wang et al. [88] report a 14.57% geomean overhead over the same set of benchmarks for the MPX-based implementation of a shadow stack, and a 21.08% geomean overhead for the PKU-based shadow stack. Unfortunately, Intel MPX is deprecated and will be unavailable in future processors [44]. SEIMI's `SMAP`-based solution incurs 12.49% geomean overhead but, as noted above, requires hardware virtualization, adding significant overheads to the entire system—68.37% and 33.56% geomean overhead on process- and filesystem-related kernel operations, respectively; it also no longer supports `SMAP`'s original purpose. In contrast, IskiOS uses hardware that is available in all processors following the Intel Skylake generation, does not require any virtualization support, keeping the overheads of kernel operations low, maintains half of the protection keys for userspace protection, and incurs only 22% geomean overhead on the LMBench latency microbenchmarks while providing robust race-free protection to kernel shadow stacks.

Microsoft's *Hardware-enforced Stack Protection* [73] is the first security mechanism to use the long-awaited Intel *Control-Flow Enforcement Technology (CET)* for return addresses. On each function call, CET stores the return address on both the program stack and a hardware-enforced read-only shadow stack. On return, it compares the two addresses and raises an exception in case of a mismatch. Unfortunately, at the time of this writing, CET is available only on certain *Tiger Lake* mobile and embedded processors.

9 DISCUSSION AND FUTURE WORK

Side Channels As Section 3.4 discusses, PKK introduces a side channel through which user code may infer locations in the kernel code segment. An alternative design for PKK would be to only set the U/S bit on kernel data pages and let kernel code reside in supervisor-mode pages. Such a design would offer equivalent security guarantees and would avoid the additional overheads of

our SMEP instrumentation, but it would not support kernel XOM. Note that enabling shadow stack support in our current prototype narrows the window of side channel vulnerability by introducing vetted `wrpkru` instructions at each function call site.

Intel PKS When available, we could leverage Intel's supervisor protection keys [46] and eliminate the need for SMEP emulation. Such a system would still require additional code after `wrmsr` instructions to prevent code reuse attacks from disabling protection (as IskiOS does for `wrpkru` instructions). Our shadow stack optimizations should be compatible with PKS.

Register Allocation Given better integration with local register allocation, IskiOS could often avoid spilling the three registers used to instrument call sites. Inter-procedural register allocation could similarly avoid spilling the return address to the shadow stack in many cases, further reducing the frequency of `wrpkru` instructions.

Fine-grained Intra-kernel Isolation Additional future work might use PKK to protect sensitive kernel data regions—e.g., process control blocks (PCBs), credential structures, and interrupt vector tables—against unauthorized accesses. We also hope to investigate the use of PKK to better isolate kernel components. Previous work in this area [24] relies on expensive serializing instructions and can only protect data integrity; PKK offers a potentially faster mechanism that can ensure both integrity and confidentiality.

10 CONCLUSIONS

IskiOS is, to the best of our knowledge, the first system to implement race-free write-protected shadow stacks and flexible (non-entropy-compromising) execute-only memory for the OS kernel. Shadow stacks protect return addresses from corruption, and execute-only memory enables state-of-the-art leakage-resilient diversification schemes by hiding code from buffer overread attacks. Unlike previous work, IskiOS imposes no restrictions on virtual address space layout, allowing the OS to place kernel stacks and code pages at arbitrary locations. IskiOS achieves these benefits through a novel use of Intel's PKU for protection inside the OS kernel. Our "PKK"-based implementation of XOM (with software-emulated SMEP) incurs roughly 12% geomean overhead on the LMBench microbenchmarks, relative to the vanilla Linux kernel and virtually no performance overhead on most real-world applications (less than 16%, worst case, on the Phoronix programs used for Linux regression analysis). IskiOS's protected shadow stacks incur about 22% geomean overhead on LMBench and less than 5% (geomean) across the chosen Phoronix applications (less than 39%, worst case).

ACKNOWLEDGMENTS

The authors thank the anonymous referees and our shepherd, Erik van der Kouwe, for their helpful feedback and suggestions. This work was supported in part by NSF grants CNS-1618213, CNS-1900803, and CNS-1955498, by a Google Faculty Research award, and by ONR Award N00014-17-1-2996.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-Flow Integrity Principles, Implementations, and Applications. *ACM Trans. on Information Systems Security (TISSEC)* 13, Article 4 (November 2009). Issue 1.
[2] Advanced Micro Devices. 2017. AMD64 Architecture Programmer's Manual.
[3] Jonathan Afek and Adi Sharabani. 2007. Dangling Pointer: Smashing the Pointer for Fun and Profit. In *Black Hat USA*.
[4] Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. 2014. You Can Run but You Can't Read: Preventing Disclosure Exploits in Executable Code. In *21st ACM SIGSAC Conf. on Computer and Communications Security (CCS)*. Scottsdale, AR. https://doi.org/10.1145/2660267.2660378
[5] D. P. Bovet and Marco Cesati. 2003. *Understanding the LINUX Kernel* (2^{nd} ed.). O'Reilly, Sebastopol, CA.
[6] Kjell Braden, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2016. Leakage-resilient layout randomization for mobile devices. In *23rd Annual Network and Distributed System Security Symp. (NDSS)*. San Diego, CA. http://www.eurecom.fr/publication/4797
[7] Brendan Bregg. 2018. KPTI/KAISER Meltdown Initial Performance Regressions. http://www.brendangregg.com/blog/2018-02-09/kpti-kaiser-meltdown-performance.html [Online; accessed 10-November-2020].
[8] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. 2017. Control-Flow Integrity: Precision, Security, and Performance. *ACM Computing Surveys (CSUR)* 50, 1, Article 16 (April 2017). https://doi.org/10.1145/3054924
[9] N. Burow, X. Zhang, and M. Payer. 2019. SoK: Shining Light on Shadow Stacks. In *40th IEEE Symp. on Security and Privacy (S&P)*. Los Alamitos, CA, 1239–1253. https://doi.org/10.1109/SP.2019.00076
[10] Nicolas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. 2015. Control-flow Bending: On the Effectiveness of Control-flow Integrity. In *24th USENIX Security Symp. (SEC)*. Washington, D.C. http://dl.acm.org/citation.cfm?id=2831143.2831154
[11] Scott A. Carr and Mathias Payer. 2017. DataShield: Configurable Data Confidentiality and Integrity. In *12th ACM Asia Conf. on Computer & Communications Security (ASIA CCS)*. Abu Dhabi, United Arab Emirates. https://doi.org/10.1145/3052973.3052983
[12] Microsoft Security Response Center. 2018. The Evolution of CFI Attacks and Defenses. https://github.com/microsoft/MSRC-Security-Research/tree/master/presentations/2018_02_OffensiveCon [Online; accessed 03-December-2020].
[13] Tzi-Cker Chiueh and Fu-Hau Hsu. 2001. RAD: A Compile-time Solution to Buffer Overflow Attacks. In *21st Intl. Conf. on Distributed Computing Systems (ICDCS)*. Phoenix, AZ. https://doi.org/10.1109/ICDSC.2001.918971
[14] Clang Documentation. [n.d.]. SafeStack. https://clang.llvm.org/docs/SafeStack.html [Online; accessed 18-June-2021].
[15] Clang Documentation. [n.d.]. ShadowCallStack LLVM Pass. https://clang.llvm.org/docs/ShadowCallStack.html [Online; accessed 03-December-2020].
[16] R. Joseph Connor, Tyler McDaniel, Jared M. Smith, and Max Schuchard. 2020. PKU Pitfalls: Attacks on PKU-based Memory Isolation Systems. In *29th USENIX Security Symp. (SEC)*, Srdjan Capkun and Franziska Roesner (Eds.). 1409–1426. https://www.usenix.org/Conf./usenixsecurity20/presentation/connor
[17] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. 2015. Losing Control: On the Effectiveness of Control-Flow Integrity Under Stack Attacks. In *22nd ACM SIGSAC Conf. on Computer and Communications Security (CCS)*. Denver, CO. https://doi.org/10.1145/2810103.2813671
[18] Jonathan Corbet. 2009. Fun with NULL pointers. https://lwn.net/Articles/342330/ [Online; accessed 03-December-2020].
[19] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. 2003. Pointguard™: Protecting Pointers From Buffer Overflow Vulnerabilities. In *12th USENIX Security Symp. (SEC)*. Washington, DC.
[20] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. 1998. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-overflow Attacks. In *7th USENIX Security Symp. (SEC)*. San Antonio, TX. http://dl.acm.org/citation.cfm?id=1267549.1267554
[21] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. 2015. Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *36th IEEE Symp. on Security and Privacy (S&P)*. San Jose, CA. https://doi.org/10.1109/SP.2015.52
[22] John Criswell, Nathan Dautenhahn, and Vikram Adve. 2014. KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels. In *35th IEEE Symp. on Security and Privacy (S&P)*. San Jose, CA. https://doi.org/10.1109/SP.2014.26

[23] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. 2015. The Performance Cost of Shadow Stacks and Stack Canaries. In *10th ACM Asia Conf. on Computer & Communications Security (ASIACCS)*. Singapore, Republic of Singapore. https://doi.org/10.1145/2714576.2714635

[24] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. 2015. Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation. In *20th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Istanbul, Turkey. https://doi.org/10.1145/2694344.2694386

[25] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. 2011. ROPdefender: A Detection Tool to Defend Against Return-oriented Programming Attacks. In *6th ACM Asia Conf. on Computer & Communications Security (ASIACCS)*. Hong Kong, China. https://doi.org/10.1145/1966913.1966920

[26] Xiaowan Dong, Zhuojia Shen, John Criswell, Alan L. Cox, and Sandhya Dwarkadas. 2018. Shielding Software From Privileged Side-Channel Attacks. In *27th USENIX Security Symp. (SEC)*. Baltimore, MD. https://www.usenix.org/Conf./usenixsecurity18/presentation/dong

[27] Jake Edge. 2013. Kernel Address Space Layout Randomization. https://lwn.net/Articles/569635 [Online; accessed 11-March-2019].

[28] Hiroaki Etoh. 2004. Gcc Extension for Protecting Applications from Stack-smashing Attacks.

[29] Isaac Evans, Sam Fingeret, Julian Gonzalez, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. 2015. Missing the Point(Er): On the Effectiveness of Code Pointer Integrity. In *36th IEEE Symp. on Security and Privacy (S&P)*. San Jose, CA, 781–796. https://doi.org/10.1109/SP.2015.53

[30] Reza Mirzazade Farkhani, Saman Jafari, Sajjad Arshad, William Robertson, Engin Kirda, and Hamed Okhravi. 2018. On the Effectiveness of Type-based Control Flow Integrity. In *34th Annual Computer Security Applications Conf. (ACSAC)*. San Juan, PR. https://doi.org/10.1145/3274694.3274739

[31] X. Ge, N. Talele, M. Payer, and T. Jaeger. 2016. Fine-Grained Control-Flow Integrity for Kernel Software. In *1st IEEE European Symp. on Security and Privacy (EuroS&P)*. Saarbrücken, Germany. https://doi.org/10.1109/EuroSP.2016.24

[32] J. Gionta, W. Enck, and P. Larsen. 2016. Preventing Kernel Code-reuse Attacks through Disclosure Resistant Code Diversification. In *IEEE Conf. on Communications and Network Security (CNS)*. Philadelphia, PA, 189–197. https://doi.org/10.1109/CNS.2016.7860485

[33] Jason Gionta, William Enck, and Peng Ning. 2015. HideM: Protecting the Contents of Userspace Memory in the Face of Disclosure Vulnerabilities. In *5th ACM Conf. on Data and Application Security and Privacy (CODASPY)*. San Antonio, TX. https://doi.org/10.1145/2699026.2699107

[34] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. 2014. Out of Control: Overcoming Control-Flow Integrity. In *35th IEEE Symp. on Security and Privacy (S&P)*. San Jose, CA. https://doi.org/10.1109/SP.2014.43

[35] Enes Göktaş, Robert Gawlik, Benjamin Kollenda, Elias Athanasopoulos, Georgios Portokalidis, Cristiano Giuffrida, and Herbert Bos. 2016. Undermining Information Hiding (and What to Do about It). In *25th USENIX Security Symp. (SEC)*. Austin, TX, 105–119. https://www.usenix.org/Conf./usenixsecurity16/technical-sessions/presentation/goktas

[36] Jinyu Gu, Xinyue Wu, Wentai Li, Nian Liu, Zeyu Mi, Yubin Xia, and Haibo Chen. 2020. Harmonizing Performance and Isolation in Microkernels with Efficient Intra-kernel Isolation and Communication. In *31st USENIX Annual Technical Conf. (ATC)*. Virtual Conf., 401–417. https://www.usenix.org/Conf./atc20/presentation/gu

[37] E. Göktas, B. Kollenda, P. Koppe, E. Bosman, G. Portokalidis, T. Holz, H. Bos, and C. Giuffrida. 2018. Position-Independent Code Reuse: On the Effectiveness of ASLR in the Absence of Information Disclosure. In *3rd IEEE European Symp. on Security and Privacy (Euro S&P)*. London, UK, 227–242. https://doi.org/10.1109/EuroSP.2018.00024

[38] Dave Hansen. [n.d.]. System Calls for Memory Protection Keys. https://lwn.net/Articles/695972/ [Online; accessed 11-March-2019].

[39] Dave Hansen. 2017. KPTI Patch. https://lkml.org/lkml/2017/12/18/1523 [Online; accessed 03-December-2020].

[40] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. 2019. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *30th USENIX Annual Technical Conf. (ATC)*. Renton, WA, 489–504. https://www.usenix.org/Conf./atc19/presentation/hedayati-hodor

[41] John L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *ACM SIGARCH Computer Architecture News* 34, 4 (September 2006). https://doi.org/10.1145/1186736.1186737

[42] IBM Corp. 1964. *IBM System/360 Principles of Operation*. IBM Press.

[43] Qualcomm Technologies, Inc. 2017. Pointer Authentication on ARMv8.3: Design and Analysis of the New Software Security Instructions. https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf [Online; accessed 15-November-2019].

[44] Intel Corp. 2013. Introduction to Intel® Memory Protection Extensions. https://software.intel.com/content/www/us/en/develop/articles/introduction-to-intel-memory-protection-extensions.html [Online; accessed 10-November-2020].

[45] Intel Corp. 2018. Intel Security Features and Technologies Related to Transient Execution Attacks. https://software.intel.com/content/www/us/en/develop/articles/software-security-guidance/best-practices/related-intel-security-features-technologies.html [Online; accessed 17-June-2021].

[46] Intel Corp. 2021. Intel 64 and IA-32 Architectures Software Developer's Manual. 325384-074US.

[47] Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis. 2014. Ret2Dir: Rethinking Kernel Isolation. In *23rd USENIX Conf. on Security Symp. (SEC'14)*. USENIX Association, San Diego, CA. http://dl.acm.org/citation.cfm?id=2671225.2671286

[48] Vasileios P. Kemerlis, Georgios Portokalidis, and Angelos D. Keromytis. 2012. kGuard: Lightweight Kernel Protection Against Return-to-user Attacks. In *21st USENIX Security Symp. (SEC)*. Bellevue, WA.

[49] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. 2006. Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software. In *22nd Annual Computer Security Applications Conf. (ACSAC)*. Miami Beach, FL. https://doi.org/10.1109/ACSAC.2006.9

[50] Chris Kjellqvist, Mohammad Hedayati, and Michael L. Scott. 2020. Safe, Fast Sharing of Memcached as a Protected Library. In *49th Intl. Conf. on Parallel Processing (ICPP)*. Edmonton, AB, Canada, Article 6, 8 pages. https://doi.org/10.1145/3404397.3404443

[51] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. 2017. No Need to Hide: Protecting Safe Regions on Commodity Hardware. In *12th European Conf. on Computer Systems (EuroSys)*. Belgrade, Serbia.

[52] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2014. Code-Pointer Integrity. In *11th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*. Broomfield, CO, 147–163. https://www.usenix.org/Conf./osdi14/technical-sessions/presentation/kuznetsov

[53] Donghyun Kwon, Jangseop Shin, Giyeol Kim, Byoungyoung Lee, Yeongpil Cho, and Yunheung Paek. 2019. uXOM: Efficient eXecute-Only Memory on ARM Cortex-M. In *28th USENIX Security Symp. (SEC)*. Santa Clara, CA, 231–247. https://www.usenix.org/Conf./usenixsecurity19/presentation/kwon

[54] Stefan Lankes, Jens Breitbart, and Simon Pickartz. 2019. Exploring Rust for Unikernel Development. In *10th Workshop on Programming Languages and Operating Systems (PLOS)*. Huntsville, ON, Canada. https://doi.org/10.1145/3365137.3365395

[55] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Intl. Symp. on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO)*. Palo Alto, CA. http://dl.acm.org/citation.cfm?id=977395.977673

[56] J. Li, X. Tong, F. Zhang, and J. Ma. 2018. Fine-CFI: Fine-Grained Control-Flow Integrity for Operating System Kernels. *IEEE Trans. on Information Forensics and Security (TIFS)* 13, 6 (June 2018), 1535–1550. https://doi.org/10.1109/TIFS.2018.2797932

[57] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan. 2019. PAC It Up: Towards Pointer Integrity Using ARM Pointer Authentication. In *28th USENIX Security Symp. (SEC)*. Santa Clara, CA. http://dl.acm.org/citation.cfm?id=3361338.3361352

[58] Linux Manual Pages. [n.d.]. process_vm_readv(2). https://man7.org/linux/man-pages/man2/process_vm_readv.2.html [Online; accessed 23-June-2021].

[59] LinuxBenchmarking. [n.d.]. Daily Mainline Linux Kernel Tests. http://www.linuxbenchmarking.com/?daily-mainline-linux-kernel-tests [Online; accessed 03-December-2020].

[60] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symp. (SEC)*. Baltimore, MD. https://www.usenix.org/Conf./usenixsecurity18/presentation/lipp

[61] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. 2015. Thwarting Memory Disclosure with Efficient Hypervisor-enforced Intra-domain Isolation. In *22nd ACM SIGSAC Conf. on Computer and Communications Security (CCS)*. Denver, CO.

[62] Stephen Mccamant and Greg Morrisett. 2005. *Efficient, Verifiable Binary Sandboxing for a CISC Architecture*. Technical Report MIT LCS TR #988. MIT Compter Science and Artificial Intelligence Lab.

[63] Larry McVoy and Carl Staelin. 1996. lmbench: Portable Tools for Performance Analysis. In *7th USENIX Annual Technical Conf. (ATC)*. San Diego, CA. http://dl.acm.org/citation.cfm?id=1268299.1268322

[64] Phoronix Media. [n.d.]. Phoronix Test Suite. https://www.phoronix-test-suite.com [Online; accessed 11-March-2019].

[65] Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, and Haibo Chen. 2019. SkyBridge: Fast and Secure Inter-process Communication for Microkernels. In *14th European Conf. on Computer Systems (EuroSys)*. Dresden, Germany, Article 9. https://doi.org/10.1145/3302424.3303946

[66] João Moreira, Sandro Rigo, Michalis Polychronakis, and Vasileios P. Kemerlis. 2017. DROP THE ROP: Fine Grained Control-Flow Integrity for The Linux Kernel. In *Black Hat Asia*.

[67] Vikram Narayanan, Yongzhe Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. 2020. Lightweight Kernel Isolation with Virtualization and VM Functions. In *16th ACM SIGPLAN/SIGOPS Intl. Conf. on Virtual Execution Environments (VEE)*. Lausanne, Switzerland, 157–171. https://doi.org/10.1145/3381052.3381328
[68] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. 2010. G-Free: Defeating Return-oriented Programming Through Gadget-less Binaries. In *26th Annual Computer Security Applications Conf. (ACSAC)*. Austin, TX. https://doi.org/10.1145/1920261.1920269
[69] Aleph One. 1996. Smashing the Stack for Fun and Profit. *Phrack* 7 (November 1996). Issue 49. http://www.phrack.org/issues/49/14.html [Online; accessed 03-December-2020].
[70] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. 2012. Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization. In *33rd IEEE Symp. on Security and Privacy (S&P)*. San Francisco, CA. https://doi.org/10.1109/SP.2012.41
[71] Marios Pomonis, Theofilos Petsios, Angelos D. Keromytis, Michalis Polychronakis, and Vasileios P. Kemerlis. 2019. Kernel Protection Against Just-In-Time Code Reuse. *ACM Trans. on Privacy and Security (TOPS)* 22, 1, Article 5 (January 2019). https://doi.org/10.1145/3277592
[72] S. Proskurin, M. Momeu, S. Ghavamnia, V. P. Kemerlis, and M. Polychronakis. 2020. xMP: Selective Memory Protection for Kernel and User Space. In *41st IEEE Symp. on Security and Privacy (S&P)*. https://doi.org/10.1109/SP40000.2020.00041
[73] Hari Pulapaka. 2020. Understanding Hardware-enforced Stack Protection. https://techcommunity.microsoft.com/t5/windows-kernel-internals/understanding-hardware-enforced-stack-protection/ba-p/1247815 [Online; accessed 16-June-2021].
[74] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. 2012. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Trans. on Information Systems Security (TISSEC)* 15, 1, Article 2 (March 2012).
[75] Robert Rudd, Richard W. Skowyra, David Bigelow, Veer Dedhia, Thomas Hobson, Stephen Crane, Christopher Liebchen, Per Larsen, Lucas Davi, Michael Franz, Ahmad-Reza Sadeghi, and Hamed Okhravi. 2017. Address Oblivious Code Reuse: On the Effectiveness of Leakage Resilient Diversity. In *24th Annual Network and Distributed System Security Symp. (NDSS)*. San Diego, CA.
[76] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. 2010. Adapting Software Fault Isolation to Contemporary CPU Architectures. In *19th USENIX Security Symp. (SEC)*. Washington, DC. http://dl.acm.org/citation.cfm?id=1929820.1929822
[77] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. 2007. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *21st ACM SIGOPS Symp. on Operating Systems Principles (SOSP)*. Stevenson, WA. https://doi.org/10.1145/1294261.1294294
[78] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2013. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *34th IEEE Symp. on Security and Privacy (S&P)*. San Francisco, CA. https://doi.org/10.1109/SP.2013.45
[79] Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund, and Thomas Walter. 2009. Breaking the Memory Secrecy Assumption. In *2nd European Workshop on System Security (EUROSEC)*. Nuremburg, Germany. https://doi.org/10.1145/1519144.1519145
[80] Mincheol Sung, Pierre Olivier, Stefan Lankes, and Binoy Ravindran. 2020. Intra-Unikernel Isolation with Intel Memory Protection Keys. In *16th ACM SIGPLAN/SIGOPS Intl. Conf. on Virtual Execution Environments (VEE)*. Lausanne, Switzerland, 143–156. https://doi.org/10.1145/3381052.3381326
[81] The PaX Team. [n.d.]. NOEXEC. https://pax.grsecurity.net/docs/noexec.txt [Online; accessed 03-December-2020].
[82] The PaX Team. 2015. RAP: RIP ROP. https://pax.grsecurity.net/docs/PaXTeam-H2HC15-RAP-RIP-ROP.pdf [Online; accessed 03-December-2020].
[83] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *28th USENIX Security Symp. (SEC)*. Santa Clara, CA, 1221–1238. https://www.usenix.org/Conf./usenixsecurity19/presentation/vahldiek-oberwagner
[84] Vendicator. 2000. Stack Shield: A 'Stack-smashing' Technique Protection Tool for Linux. http://www.angelfire.com/sk/stackshield/info.html [Online; accessed 11-March-2019].
[85] Perry Wagle and Crispin Cowan. 2003. Stackguard: Simple Stack Smash Protection for GCC. In *GCC Developers Summit*. 243–255.
[86] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient Software-Based Fault Isolation. In *14th ACM SIGOPS Symp. on Operating Systems Principles (SOSP)*. Asheville, NC. https://doi.org/10.1145/168619.168635
[87] Xiaoguang Wang, SengMing Yeoh, Pierre Olivier, and Binoy Ravindran. 2020. Secure and Efficient In-Process Monitor (and Library) Protection with Intel MPK. In *13th European Workshop on Systems Security (EuroSec)*. Heraklion, Greece, 7–12. https://doi.org/10.1145/3380786.3391398
[88] Zhe Wang, Chenggang Wu, Mengyao Xie, Yinqian Zhang, Kangjie Lu, Xiaofeng Zhang, Yuanming Lai, Yan Kang, and Min Yang. 2020. SEIMI: Efficient and Secure SMAP-Enabled Intra-process Memory Isolation. In *41st IEEE Symp. on Security and Privacy (S&P)*. San Francisco, CA. https://doi.org/10.1109/SP40000.2020.00087
[89] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2010. Native Client: a sandbox for portable, untrusted x86 native code. *Commun. ACM* 53, 1 (January 2010).
[90] Brandon Azad, Project Zero. 2019. Examining Pointer Authentication on the iPhone XS. https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html [Online; accessed 15-November-2019].
[91] Philipp Zieris and Julian Horsch. 2018. A Leak-Resilient Dual Stack Scheme for Backward-Edge Control-Flow Integrity. In *13th ACM Asia Conf. on Computer & Communications Security (ASIACCS)*. Incheon, Republic of Korea. https://doi.org/10.1145/3196494.3196531

A DETAILS OF LMBENCH EXPERIMENTS

This appendix includes detailed results from our experiments for interested readers.

Tables 4 and 5 show results for our LMBench experiments. In each, the second column shows the arithmetic mean and standard deviation (SD) for ten runs of each latency and bandwidth microbenchmark on the unmodified Linux kernel (i.e., our baseline). Similarly, columns 3–6 show the arithmetic mean and standard deviation, as well as the percentage overhead, of each configuration over the baseline kernel. The last row of each table shows the geometric mean (in reported metric $((\prod_i(overhead_i + 100))^{1/n} - 100)$) overhead across latency and bandwidth microbenchmarks, respectively, for each IskiOS configuration.

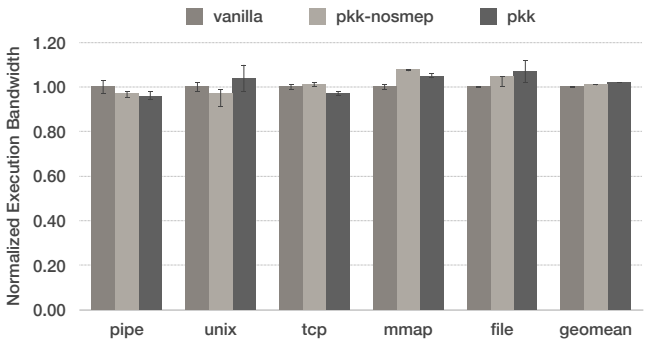

Figure 3: Breakdown of `PKK` overhead (over `vanilla` Linux) on LMBench bandwidth microbenchmarks. Columns labeled as `pkk-nosmep` represent IskiOS's `PKK` implementation with the `SMEP` LLVM pass disabled.

Figures 3 and 4 show the detailed overhead of using PKK with and without our SMEP instrumentation on the LMBench bandwidth and latency experiments, respectively.

Table 4: IskiOS run-time overhead (% over `vanilla` Linux) on latency microbenchmarks of the LMBench suite.

Benchmark	vanilla		pkk		pkk+xom		pkk+ss-swo-inline		pkk+xom+ss-swo-inline	
			Mean ± SD	Overhead	Mean ± SD	Overhead	Mean ± SD	Overhead	Mean ± SD	Overhead
syscall()	0.24 ± 0.00	μs	0.29 ± 0.00	20.88%	0.29 ± 0.00	24.05%	0.29 ± 0.01	22.98%	0.29 ± 0.00	21.92%
open()/close()	1.30 ± 0.00	μs	1.48 ± 0.00	14.12%	1.48 ± 0.01	14.40%	1.85 ± 0.01	42.61%	1.84 ± 0.01	41.67%
read()	0.29 ± 0.00	μs	0.35 ± 0.00	20.80%	0.35 ± 0.00	19.80%	0.39 ± 0.00	31.99%	0.39 ± 0.00	31.87%
write()	0.27 ± 0.00	μs	0.33 ± 0.00	24.66%	0.32 ± 0.00	19.84%	0.32 ± 0.01	18.21%	0.31 ± 0.00	17.98%
select(10 fds)	0.39 ± 0.00	μs	0.46 ± 0.00	17.26%	0.47 ± 0.00	20.15%	0.47 ± 0.00	18.60%	0.45 ± 0.00	15.42%
select(100 TCP fds)	3.34 ± 0.04	μs	3.65 ± 0.04	9.12%	3.72 ± 0.15	11.21%	3.77 ± 0.21	12.64%	3.60 ± 0.15	7.65%
select(200 fds)	3.07 ± 0.11	μs	3.16 ± 0.00	2.97%	3.17 ± 0.00	3.36%	3.26 ± 0.03	6.24%	3.23 ± 0.01	5.45%
stat()	0.65 ± 0.01	μs	0.78 ± 0.00	20.11%	0.76 ± 0.00	18.01%	0.83 ± 0.01	28.52%	0.90 ± 0.00	39.75%
fstat()	0.29 ± 0.00	μs	0.36 ± 0.00	23.83%	0.36 ± 0.00	23.41%	0.39 ± 0.00	35.96%	0.45 ± 0.00	54.81%
fcntl()	1.93 ± 0.03	μs	2.00 ± 0.03	4.01%	2.05 ± 0.03	6.37%	2.24 ± 0.02	16.57%	2.24 ± 0.02	16.27%
mmap()/munmap()	71.90 ± 0.88	μs	76.60 ± 1.17	6.54%	76.20 ± 0.63	5.98%	88.00 ± 1.76	22.39%	86.90 ± 1.52	20.86%
fork()+exit()	328.10 ± 12.70	μs	352.94 ± 5.62	7.57%	355.22 ± 5.67	8.27%	418.47 ± 7.28	27.54%	406.75 ± 8.94	23.97%
fork()+execve()	335.15 ± 4.63	μs	369.95 ± 5.40	10.38%	363.33 ± 5.73	8.41%	430.35 ± 7.39	28.41%	420.13 ± 7.18	25.36%
fork()+/bin/sh	2744.75 ± 8.81	μs	2953.10 ± 3.04	7.59%	2967.65 ± 8.68	8.12%	3356.80 ± 8.86	22.30%	3308.80 ± 13.49	20.55%
sigaction()	0.28 ± 0.00	μs	0.34 ± 0.00	21.35%	0.36 ± 0.00	27.11%	0.36 ± 0.00	26.90%	0.36 ± 0.00	26.94%
Signal Delivery	1.05 ± 0.00	μs	1.13 ± 0.01	7.30%	1.15 ± 0.00	8.81%	1.41 ± 0.01	34.13%	1.41 ± 0.00	33.68%
Protection Fault	0.71 ± 0.00	μs	0.76 ± 0.01	6.48%	0.77 ± 0.01	8.59%	0.60 ± 0.00	−15.75%	0.61 ± 0.01	−14.10%
Page Fault	0.22 ± 0.00	μs	0.24 ± 0.00	6.01%	0.23 ± 0.00	5.28%	0.24 ± 0.01	8.14%	0.24 ± 0.00	6.82%
Pipe I/O	6.76 ± 0.10	μs	6.96 ± 0.06	2.92%	6.96 ± 0.09	2.99%	7.48 ± 0.11	10.64%	7.44 ± 0.06	10.11%
Unix Socket I/O	6.56 ± 0.01	μs	7.42 ± 0.04	13.14%	7.31 ± 0.04	11.58%	8.24 ± 0.05	25.75%	8.15 ± 0.05	24.35%
TCP Socket I/O	13.08 ± 0.18	μs	14.02 ± 0.12	7.16%	14.07 ± 0.05	7.55%	16.87 ± 0.07	28.94%	16.93 ± 0.14	29.42%
UDP Socket I/O	10.39 ± 0.08	μs	11.25 ± 0.11	8.29%	11.52 ± 0.12	10.91%	14.16 ± 0.07	36.30%	14.48 ± 0.43	39.40%
Context switch	2.58 ± 0.06	μs	2.58 ± 0.04	0.04%	2.57 ± 0.05	−0.35%	2.82 ± 0.06	9.18%	2.86 ± 0.03	10.77%
Geometric Mean Overhead				11.18%		11.67%		21.01%		22.33%

Table 5: IskiOS run-time overhead (% over `vanilla` Linux) on bandwidth microbenchmarks of the LMBench suite.

Benchmark	vanilla		pkk		pkk+xom		pkk+ss-swo-inline		pkk+xom+ss-swo-inline	
			Mean ± SD	Overhead	Mean ± SD	Overhead	Mean ± SD	Overhead	Mean ± SD	Overhead
Pipe I/O	2879.34 ± 89.88	MB/s	2760.65 ± 64.33	4.12%	2819.08 ± 73.25	2.09%	2892.75 ± 55.22	−0.47%	2800.93 ± 73.45	2.72%
Unix Socket I/O	5903.98 ± 109.51	MB/s	6145.74 ± 346.57	−4.09%	6323.30 ± 269.05	−7.10%	6518.00 ± 460.98	−10.40%	6901.41 ± 61.81	−16.89%
TCP Socket I/O	5033.57 ± 28.51	MB/s	4889.97 ± 44.86	2.85%	4874.17 ± 58.14	3.17%	4379.98 ± 29.15	12.98%	4354.83 ± 56.52	13.48%
File I/O	8155.58 ± 32.22	MB/s	8704.69 ± 411.54	−6.73%	8287.26 ± 529.07	−1.61%	8247.62 ± 30.08	−1.13%	9240.31 ± 473.58	−13.30%
MMapped File I/O	18769.69 ± 115.36	MB/s	19761.46 ± 270.87	−5.28%	18532.37 ± 86.70	1.26%	18739.26 ± 930.41	0.16%	20225.81 ± 38.04	−7.76%
Geometric Mean Overhead				−1.93%		−0.51%		−0.05%		−4.98%

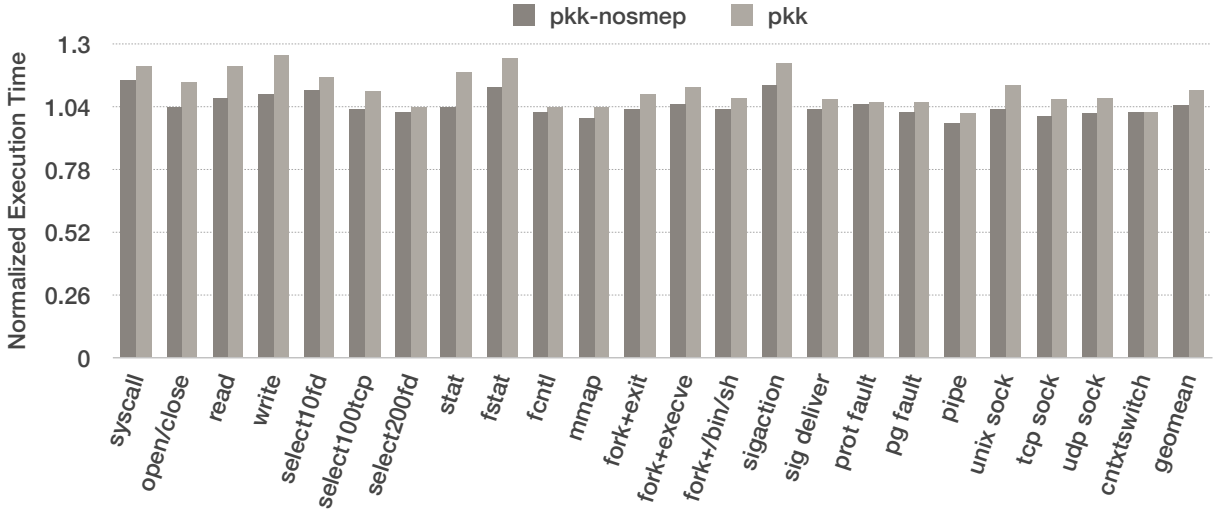

Figure 4: Breakdown of IskiOS's `PKK` overhead (over `vanilla` Linux) on LMBench latency microbenchmarks. Columns labeled as `pkk-nosmep` represent IskiOS's `PKK` implementation with the `SMEP` LLVM pass disabled.