Janus: Intra-Process Isolation for High-Throughput Data Plane Libraries*

Technical Report #1004

Mohammad Hedayati,[1] Spyridoula Gravani,[1] Ethan Johnson,[1] John Criswell,[1]
Michael L. Scott,[1] Kai Shen[2] and Mike Marty[2]

[1]Department of Computer Science, University of Rochester
[2]Google Inc.
{hedayati,sgravani,ejohns48,criswell,scott}@cs.rochester.edu
{kshen,mikemarty}@google.com

August 2018

Abstract

As network, I/O, accelerator, and NVM devices capable of a million operations per second make their way into data centers, the software stack managing such devices has been shifting from implementations within the operating system kernel to more specialized kernel-bypass approaches. While the in-kernel approach guarantees both safety and fairness, it imposes too much overhead on microsecond-scale tasks. Kernel-bypass approaches improve throughput substantially but sacrifice safety and complicate resource management: if applications are mutually untrusting, then either each application must have exclusive access to its own device or else the device itself must implement resource management.

This paper shows how to attain both safety and performance via intra-process isolation for multiple data plane libraries. Our protected libraries *provide separate user-level protection domains for different services (e.g., network and in-memory database), with performance approaching that of unprotected kernel bypass. We explore two concrete implementations of protected libraries. The first uses Intel's VM functions (VMFUNC) mechanism to switch between pre-established address spaces without entering the kernel. The second uses the mechanism of memory protection keys (Intel's PKU) to change the permissions associated with subsets of a single address space.*

We show that these approaches can efficiently protect high-throughput in-memory databases and user-space network stacks. The PKU approach incurs less than 10% protection cost at up to 2.3 million library switches per second per core. The VMFUNC approach is more expensive, but, it eliminates the need to inspect program binaries and offers stronger protection against speculative side-channel data leaks. Both approaches significantly outperform kernel-level protection or system call switching of user-level address spaces.

*This work was supported in part by NSF grants CNS-1319417, CCF-1717712, CCF-1422649, CNS-1618213 and CNS-1629770, and by a Google Faculty Research award.

1. Introduction

A principal task of an operating system is to multiplex hardware resources, making them accessible to multiple user-level applications, and to arbitrate use of those resources to satisfy system-wide performance and fairness goals. User/kernel isolation enables the OS to enforce its resource management decisions in the face of untrusted and potentially malicious applications. In recent years, however, developers have begun to move I/O management into user space for the sake of higher performance, specialization, and rapid development. This strategy is often referred to as *kernel-bypass* I/O. DPDK [20] and mTCP [25] move packet processing and transport layer processing into user space; SPDK [21] does the same for direct access to fast storage devices like Optane SSDs [17]. Accelerators like Google's TPU [26] and Nvidia's GPUs [34] also rely on kernel-bypass software stacks for low-latency hardware access and rapid evolution of drivers.

The trend toward kernel bypass has enabled significant improvements in device throughput and latency [3,38,39]. These gains, however, have typically come at the cost of granting an application exclusive access to a device, trusting other users of the device, or relying on the existence of a hardware-level arbitrator that virtualizes or partitions the device (e.g., SR-IOV [22]). Unfortunately, device-level resource isolation is not always available or it lacks the policy flexibility of OS-level resource containers.

The looming widespread availability of byte-addressable non-volatile memory (NVM) DIMMs [42] brings new challenges towards coordinating the use of shared data without mediation by OS-level file system reads and writes [36]. Virtual memory alone provides insufficient protection from memory safety errors, and relying on kernel-implementation mechanisms throws away the performance potential of loads and stores directly to persistent memory.

One can, of course, implement protection domains within an address space using a trusted compiler with static [16] or dynamic [51] checking. The static approach, however, requires a type-safe language, and is thus incompatible with many existing applications. The dynamic approach incurs overhead that is significant even in the simplest cases (e.g., when checking pointers against a single boundary address), and rises steeply for more complex address space layouts.

What we desire is a mechanism that is compatible with existing applications (i.e., by re-linking I/O libraries), provides fast transitions into and out of protected library routines, imposes little or no cost on ordinary code, and can accommodate domains with relatively complicated address space layouts (multiple protected code and data segments). Toward that end, this paper proposes *Janus*, a mechanism for low-overhead intra-process isolation. Janus relies on library boundaries to define protection domains. It allows multiple mutually distrusting libraries to load into the same address space and provides different "views" of the address space (both code and data) to the main application and to each library. Janus employs the standard function call/return interface, but interposes a *trampoline* routine on each library call, allowing an application to safely change its view of the address space to acquire access to a protected resource—but only when executing target code that has such rights.

Janus enables instances of a protected library in multiple applications to coordinate accesses to shared resources. Instances of a network library, for example, might provide fast, user-level access to a NIC while enforcing rate-limiting policies that require coordination among otherwise uncoordinated and mutually distrusting applications.

We describe two concrete implementations of Janus for Intel processors, one based on the Extended Page Table (EPT) switching VM function (VMFUNC) mechanism, the other based on memory protection keys, which Intel calls PKU (Protection Keys for Userspace). Both implementations support mutually distrusting protected libraries, allowing an application to use several high-performance libraries safely and concurrently. The PKU approach offers better performance, but requires trusted inspection of the full program binary and, in some cases, a modest amount of binary rewriting. The VMFUNC approach, while slower, avoids these limitations and offers stronger protection against speculative side-channel data leaks. Both implementations require modification to threading and signal handling, but neither requires a trusted compiler or significant changes to application source code.

In summary,

- We introduce a mechanism, Janus, to isolate fast data-plane libraries from both the calling application and each other.
- We present two concrete implementations of Janus for current Intel processors—one based on VMFUNC, the other on a novel use of memory protection keys.
- We quantify the performance benefits of Janus on real-world applications with respect to both unprotected kernel bypass and isolation based on a kernel-mediated page table switch.
- We explore the implications of our implementations for speculative side-channel data leaks.

The following section describes in more detail the problem addressed by protected libraries, including the threats against which we protect, the assumptions we make about library code, the capabilities we provide to libraries, and the system components (signal interface, thread package, operating system kernel) that must be modified to ensure isolation. Section 3 then describes our candidate implementations. We evaluate the performance of these implementations in Section 4 using microbenchmarks, the Silo in-memory database [45], the DPDK data-plane library package [20], and the Redis [40] NoSQL server. Section 5 discusses related work. Section 6 summarizes our conclusions.

2. Protected Libraries

Our goal is to develop a *protected library* mechanism that partitions an application into multiple domains of executable code, where each domain may see a different view of the process's address space. That is, each domain can be granted access rights to some parts of the address space and denied rights to other parts. Each domain has private stacks and possibly a private heap, but also shares access to some pages, allowing efficient communication with other domains. Domain transitions follow the calling conventions of the architecture. Behind the scenes, however, each transition must move between domains. We therefore invoke the entry points of protected libraries using *trampoline* routines, which switch to the domain's address space view, switch stacks, and set up arguments to maintain calling conventions. Trampolines also ensure that returning from a protected library switches back to the caller's domain.

2.1. Threat Model

In Janus, an untrusted application uses protected libraries to access protected resources. A resource might comprise ordinary memory, non-volatile memory, or a memory-mapped device. An unmodified application, by default, shares its entire memory space with a (trusted) protected library, but the library shares only the *trampoline* code needed for cross-domain calls. If it wishes, a (slightly modified) application can choose to share only protected buffers with the library. Figure 1 shows an example with a user-space network library and a userspace storage library. The storage library has default access to the application; the network library has been given access only to shared buffers. We assume that applications may be multi-threaded and that library entry points may accept pointers to callback functions. Consequently, protected libraries must be multi-thread safe and re-entrant.

In the terminology of Saltzer and Schroeder [41], each protected library provides a *protected subsystem* that encapsulates a protected resource. A protected library is trusted to enforce security and resource management policies for the protected resources it manages but is otherwise untrusted. We further assume that the hardware and the operating system kernel are within the Trusted Computing Base (TCB) (the part of the system that must be implemented correctly for security policies

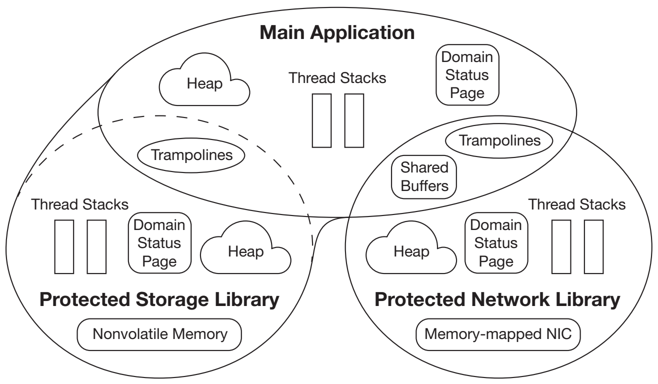

Figure 1: Protected Library Architecture. The application has, by default, shared its entire memory space with the storage library. It has opted to share only certain buffers with the network library.

to be properly enforced) and are correctly implemented. The application and other untrusted libraries are outside the TCB: they are not trusted to read or write protected library memory.

In this work we are primarily interested in preserving the integrity and confidentiality of protected memory and devices from direct memory reads and writes. Ideally, we may also wish to prevent leaks of protected data via speculative side-channel attacks [28] launched by other domains.

The attacker controls the application and untrusted libraries and can add arbitrary native code to the application and to untrusted libraries. We assume that the attacker cannot gain direct access to the data within a protected library's memory via the library's own API. We must consider the possibility, however, that the attacker may attempt to:

- Gain access to protected memory by changing virtual to physical mappings using system calls like `mmap`.
- Modify, from a compromised thread, local variables or return addresses in the stack of a thread that is running in the library.
- Subvert the loading mechanism so that a different library has access to protected memory.
- Install malicious signal handlers and then arrange for a signal to be delivered while the library is running.

We consider these issues in turn in the following subsections.

2.2. Virtual Address Space Integrity

In a standard Linux system, a process can change the page permissions of its own memory with the `mprotect` system call, and change the mappings between virtual and physical addresses with the `mmap` system call. For any given protected library L, we must prevent address space changes, when requested by code outside of L, from making L's code or data accessible to the application or to another library.

This is the easiest vulnerability to address. We assume that the static and dynamic loaders are part of the trusted computing base. When asked to load a protected library, they inform the operating system of the virtual addresses used by the protected library's code and data. On any subsequent call to `mprotect`, `mmap`, etc., the kernel checks to see what portion of the address space is being modified and what portion contains the syscall instruction. It denies requests to change the mappings or permissions of protected library space unless the request is made from the code of the library itself.

2.3. Local Variables and Protected Stack

To protect the confidentiality and integrity of its data, each protected library owns memory that other protected libraries and the application cannot read, write, or execute. Within that memory, the protected library owns its code, global variables, and heap. It also contains a private stack to store return addresses and local variables. Since we support multithreaded applications, each protected library must contain a separate stack for each thread. Thus, when an application creates a new thread, we must create a new stack for each of the domains in the application. We envision embedding this logic within the threading library (e.g., `pthreads`) so that application developers do not need to explicitly modify any application code.

When an application calls a function within a protected library, trampoline code accessible to the application must switch the protection domain so that the target function executes in the library's protection domain and can access its protected code and data. The trampoline must then switch the stack pointer to the stack for the appropriate library and thread. Section 3 describes the design of our trampolines for each intra-process isolation mechanism in more detail.

If the protected library invokes a callback function within the application, it will also use trampoline code to switch back to the application's domain and back to the application's stack.

Switching stacks can be challenging when the source or target distrusts the other. Previous work addressed this issue either by going through a trusted domain like the kernel to save and restore the stacks [32] or by not supporting mutually distrusting domains [49]. While we could employ a trusted *trampoline domain*, such an implementation would double the overhead of transitions by changing the view first to the trampoline domain and then to the target domain. We address this challenge by first saving the state of the source domain (i.e., `rsp`, `fs`, etc.) in a *domain status page* accessible only in the source domain, then switching the address space view to the target domain, and finally restoring the state of the target domain from a domain status page accessible only to the target. Note that domain status pages are per-thread entities.

Unfortunately, in the absence of trust, we need to access domain status pages without relying on registers such as `fs` (used for thread local storage). We can address this issue by arranging for the kernel to support a fast (vDSO-style) `gettid` call to acquire the current thread ID. The kernel will maintain a list (readable, but not writable in user space) of currently running thread IDs for all CPU cores. The fast `gettid` will perform a vDSO `getcpu` lookup and use the result to find the thread ID. This enables trampoline code to access thread-local storage without relying on fs or performing a system call.

2.4. Program Loading

Janus employs a trusted loader, running as root, to start up any application that uses one or more protected libraries. The trusted loader first maps all protected libraries into the virtual address space using the mmap system call. It then calls an initialization function within each protected library. In this initialization code, a library can open and map the device files it needs so that it has direct read/write access to a device's memory-mapped I/O registers or to a region of persistent or shared memory. This initialization code will also allocate the first stack, initialize the heap, and call constructor functions (e.g., for C++ global variables) for the protected library.

Once all protected libraries are initialized, the trusted loader uses a system call to inform the kernel of where it has loaded each protected library. This allows the kernel to enforce restrictions on system calls that configure the virtual address space (as Section 2.2 describes). The trusted loader then loads the application code and all other pre-loaded dynamic libraries. If inspection or modification of this code is required (as in the PKU-based version of Janus that we introduce in Sec. 3.3), the loader performs it now; the kernel arranges to perform similar inspection and modification on any additional libraries that are loaded on demand and on any other pages for which the execute permission is enabled during execution. Finally, the trusted loader drops root privileges using the setresuid system call, runs the constructor functions of the application, and then transfers control to the application's main routine.

2.5. Signal Handlers

To support unmodified applications, Janus must address asynchronous event delivery via signals [5]. Signal handlers pose two threats. First, as signals can arrive at any time, the kernel may execute a signal handler while trampoline or protected library code is executing. This might allow malicious code to register one of its functions as a signal handler and then arrange for a signal to be sent to itself while execution is in some library's protection domain. We therefore modify the kernel's signal-handling code so that it switches to the application's domain before executing signal handlers and then back to the previously executing domain when signal handlers return.

In conjunction with the change of domain, the kernel arranges for the handler to execute on the appropriate stack for the thread and domain. It keeps track of these stacks in domain status pages (mentioned in Section 2.3), tracking not only the mapping from $\langle$thread, domain$\rangle$ pairs to stacks, but also the order in which to return in the case of nested signals (whose handlers, of course, may make calls into protected libraries).

A second threat stems from the convention of saving the state of the interrupted program on the stack on which the handler executes. Naively, this convention might allow an (unprotected) handler to examine register values (including the program counter) from a protected library, and to modify application data and control flow by changing their saved values [4]. In a similar vein, a malicious application might call sigreturn with corrupted state on the user-space stack to resume execution in the middle of protected library code [4].

To address this threat, we modify the kernel's signal-handling code to save interrupted program state in kernel memory instead of on the user-space stack (this approach has also been used in previous work [8, 9]). State is saved in stack order to ensure correct restoration in the presence of nested signals. Dummy state saved on the user-space stack provides backwards compatibility with existing code. This design breaks applications that intentionally modify state saved on the stack during signal handler dispatch, but we believe such applications to be rare.

Our design does not permit signal handlers to register for execution in a protected domain. We deemed such functionality as low priority because none of the privileged library use cases and examples we evaluated need signal handling. Nonetheless, if it were required, the kernel's signal handler registration API could be extended to allow a protected library to request that a handler should execute in the library's domain. Since the kernel knows the locations of all protected library code segments in memory (see Section 2.4), it could confirm whether the request was made from trusted library code and allow or deny the registration accordingly.

3. Fast Memory Isolation

In this section we present two implementations of memory isolation for Janus. The first implementation, Janus-VMFUNC, is described in Section 3.2. It relies on separate page tables for each domain, and switches between them using Intel's VM Function (VMFUNC) mechanism, in a manner reminiscent of the CrossOver [30] and SeCage [33] systems. The second implementation, Janus-PKU, is described in Section 3.3. It uses memory protection keys to provide different access rights in the same page table. In the process of writing this paper, we discovered that the ERIM project [49] has concurrently explored a similar use of PKU. We discuss related work in more detail in Section 5.

We also describe (in Section 3.1) a more straightforward implementation that uses system calls to change the page table root pointer. Both the syscall-based system and Janus-VMFUNC rely on context identifier tags (Intel's PCID and EP4TA, respectively) to avoid the need to flush the translation lookaside buffer (TLB) when changing domains.

In an attempt to capture intuition, we speak of the domains of an application as having different "views" of a single address space. That is, conceptually, the application has a single set of virtual-to-physical mappings within which we adjust permissions on individual pages. In actuality, Janus-VMFUNC and the syscall-based system use separate page table root pointers for separate views.

3.1. Page Table Switching via Syscalls

A straightforward way to implement protected libraries is to employ a separate page table for each domain and to use a system call to change page tables. Protected pages appear only in the tables of the corresponding domains; unprotected application pages and trampoline pages appear in all. A new system call serves to change the page table root pointer (register `CR3` on Intel machines), if and only if the requesting syscall instruction lies in the appropriate (previously registered) trampoline.

Assuming kernel page table isolation (KPTI) [13], every system call changes `CR3` on entry to the kernel. Our new syscall simply arranges (after appropriate checks) to restore the target domain's root pointer, rather than that of the calling domain, when returning to user space. This limits the overhead to only slightly more than that of a no-op system call. There may also be a rise in TLB pressure for certain applications, given that some pages will appear in the TLB more than once, with separate context tags. On hardware that has such tags, however, there is no need to flush the TLB as part of a domain switch. As a separate issue, syscalls like `munmap`, together with the TLB shootdown mechanism, must be modified to remove a mapping under all applicable context tags.

In this approach, each domain of an application has a separate page table root pointer. Fortunately, the content of the tables is largely overlapping (generic heap, vDSO, kernel translations, etc.). We use a separate first-level page for each table, but most of the lower level pages are physically shared. This approach simplifies entry manipulation and minimizes memory footprint.

3.2. Janus-VMFUNC

Beginning with its Nehalem generation of processors, Intel has provided *extended page tables* for virtualized environments. The traditional page table of a guest OS translates from "guest virtual" to "guest physical" addresses; the extended (second-level) page table translates from guest physical to (host) physical addresses. In the subsequent Haswell generation, Intel introduced a VM Function (VMFUNC) mechanism for fast invocation of hypervisor functions in a paravirtualized guest. The mechanism allows a guest to pre-register a set of second-level page tables and provides a (non-privileged) instruction to switch among them.

Janus-VMFUNC leverages this mechanism by setting up a degenerate traditional page table that implements the identity function (with all types of access allowed) and employing a separate extended page table—analogous to the ordinary page tables of the syscall-based system—for each protection domain. An application can then switch among views with no kernel involvement. Compared to a kernel-based approach (as in Section 3.1) that uses trusted kernel code to check that a domain switch is permissible, a new challenge in this approach is that we must fold the permission check into the VMFUNC instruction itself. We do so by placing the trampolines of a given library in their own page(s) and making those the only pages that are executable in the domains of both the main application and the library. A VMFUNC instruction that attempts to switch to the library's domain but lies anywhere other than an appropriate trampoline page will find the next instruction non-executable, resulting in a fault.

While the serialization overhead of an address-space-changing instruction appears to be inevitable (absent major architectural changes—e.g., CODOMs [50] and CHERI [53], which themselves impose new overheads), VMFUNC allows us to avoid the need for a system call when switching domains. As we shall see in Section 4, this cuts the cost of a switch by more than 50%.

Listing 1 (including the parts to the left of the vertical lines) shows the trampoline code for Janus-VMFUNC. Line 2 saves the stack pointer of the source domain to the source domain's status page. As Section 2 describes, this step allows the trampoline to restore the stack when returning from the protected library; it also supports callback functions. Line 7 sets `eax` to zero, indicating that an extended page table switch is desired. Line 8 sets `ecx` to the index (in a pre-approved table) of the domain to which to switch; line 9 effects the switch itself. Line 15 loads the stack pointer of the target domain from the target domain's status page into `rsp`; this is possible since `VMFUNC` has just enabled access to the private data of the target domain. At this point, the trampoline transfers control to a function in the target domain. Once the function returns, the trampoline saves the stack pointer of the target domain in its status page (line 20); it then resets the extended page table to the source domain (lines 26–28). Finally, it loads the source domain stack pointer from the source's status page into `rsp` (line 34) and resumes execution (line 35).

As a starting code base, our Janus-VMFUNC implementation uses the Dune system of Belay et al. [2], with the application running in ring 3 of VMX non-root. It is notable that running in virtualized (VMX) mode, with 2-level address translation, imposes additional overheads that are, in principle, unneeded. Most system calls, which must be handled by the operating system, incur the cost of a VM exit that is significantly more expensive than a (nonvirtualized) syscall. TLB refill costs increase as well, due to 2-level translation.

We ideally want a hardware mechanism that allows a non-privileged instruction to switch among pre-approved page table root pointers without the need for virtualization. In the meantime, optimizations are available to mitigate the cost. First, we use huge pages to reduce the first-level (identity-function) page tables from four levels to two, eliminating half the extra cost of a VMX TLB fill. Second, it should be possible (not yet implemented) to mix the use of virtualized and non-virtualized threads within a single application. Threads running in VMX mode will experience faster protected library calls but slower system calls; those running natively will have to use syscall-based page-table switching for library calls, but will see native costs for system calls and signals.

Listing 1: Janus Trampoline: VMFUNC (left) and PKU (right).

```
 1  ; Save source domain stack pointer
 2    movq %rsp, source_stack
 3
 4  ; Enable target domain view
 5                                 | 1:
 6                                 |   xorl %ecx,%ecx
 7    xorl %eax, %eax              |   xorl %edx,%edx
 8    movl $TARGET_IDX, %ecx       |   movl $TARGET_PERM, %eax
 9    vmfunc                       |   wrpkru
10                                 |   lfence
11                                 |   cmpl $TARGET_PERM, %eax
12                                 |   jne 1b         ; error
13
14  ; Switch to target domain stack
15    movq target_stack, %rsp
16
17  ; target_domain_func()
18
19  ; Save target domain stack pointer
20    movq %rsp, target_stack
21
22  ; Disable target domain view &
23  ; Enable source domain view
24                                 | 2:
25                                 |   xorl %ecx,%ecx
26    xorl %eax, %eax              |   xorl %edx,%edx
27    movl $SOURCE_IDX, %ecx       |   movl $SOURCE_PERM, %eax
28    vmfunc                       |   wrpkru
29                                 |   lfence
30                                 |   cmpl $SOURCE_PERM, %eax
31                                 |   jne 2b         ; error
32
33  ; Switch back to source domain stack
34    movq source_stack, %rsp
35    ret
```

3.3. Janus-PKU

In its Skylake generation of processors, Intel introduced a mechanism it calls *memory protection keys for userspace* (PKU). (Similar mechanisms have appeared in previous architectures from several other vendors.) While PKU is intended mainly as a memory safety enhancement (e.g., as a means of reducing vulnerability to stray-pointer bugs), we have realized that it can, with care, be used for protected libraries as well. Researchers at the Max Planck Institute have come to a similar conclusion concurrently [49].

PKU [19] employs previously unused bits in each page table entry to assign a four-bit protection key to every page, allowing that page to be associated with one of 16 potential sets of access restrictions. A new 32-bit `pkru` register, writable in user space, then specifies which rights (read and/or write) should be restricted for each of the 16 key values. On every user-mode data access, the processor checks access rights in the TLB or page table as usual, then drops any rights that are found to be restricted for the PTE's key value. Since protection keys have no impact on instruction fetches (executability) and make no changes to page tables or TLB entries, the `WRPKRU` instruction, which changes the `pkru` register, does not have to

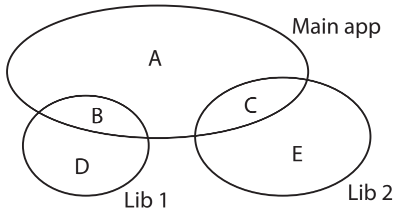

Figure 2: Address space regions in Janus-PKU.

serialize the pipeline, and can execute very quickly.

Janus-PKU is based on protection keys. If we think of a protection domain as comprising a subset of the application's address space and we plot those subsets as a Venn diagram, we can assign a protection key to each separate region of the diagram and associate with each domain a `pkru` value that disables access rights for regions outside its subset of the address space. In Figure 2, the main application would disable access to regions D, and E; library 1 would disable access to regions A, C, and E; library 2 would disable access to regions A, B, and D. Note that these conventions assign disjoint views to libraries 1 and 2.

Listing 1 (including the parts to the right of the vertical lines) shows the trampoline code for Janus-PKU. Lines 6 and 7 set registers `ecx` and `edx` to zero; this is a required precondition of the `WRPKRU` instruction. Line 8 initializes `eax` with the appropriate set of restrictions for the domain to which the trampoline is transitioning; line 9 sets the `pkru` register to the content of `eax`. The latter change simultaneously disables the view of the source domain and enables the view of the target domain. The subsequent comparison (line 30) verifies that the expected permissions have been set, thereby avoiding an attack in which a domain puts overly generous permissions into `eax` and then jumps on top of the `WRPKRU` instruction. Once the target function has returned and we have saved the stack pointer of the target domain (line 20), the trampoline resets the `pkru` register to the restrictions of the source domain (lines 25–28), and returns as in Janus-VMFUNC.

Since the processor allows user-mode code to execute the `WRPKRU` instruction, we have to worry about the possibility that a malicious application may execute a `WRPKRU` to attain access to a protected library's memory. To prevent this, we currently restrict the application from using the instruction anywhere other than trampoline routines. The trusted loader can scan the application code to ensure that it does not contain any `WRPKRU` instructions. It is possible, of course, that some sequence of bytes within an instruction, or spanning a sequence of instructions, might contain the bits encoding `WRPKRU`. In this case, we must edit the binary to replace the instructions containing these bytes with a different but functionally equivalent sequence. Vahldiek et al. confirm by exhaustive analysis that this is always possible [49]. They also note that one could, in principle, allow applications to use protection keys for other purposes, provided that each `WRPKRU` was followed by an instruction sequence to halt the program if memory of any protected library had been made accessible.

3.4. Spectre Attacks

A malicious program can attempt to use speculative execution attacks [27, 28, 31], commonly known as Spectre attacks, to steal information from protected libraries. The United States Computer Emergency Response Team (US-CERT) identifies the following four types of speculation attacks [47, 48]:

1. **Bounds Check Bypass.** A Spectre bounds check bypass causes the processor to speculatively execute instructions that read or write data outside the bounds of a memory object even if the victim program has proper bounds checking code [27, 28]. The attack may speculatively load secret data into a processor register which is then leaked to the attacker via a side channel [18, 28], or it may speculatively forward the result of a store to a younger, speculative load, enabling more sophisticated attacks such as speculative Return Oriented Programming (ROP) [27].
2. **Branch Target Injection.** Branch target buffers allow the processor to fetch and execute instructions speculatively while the target of an indirect branch is being computed. An attacker may train the branch target buffer to jump to a desired location and speculatively execute code that leaks confidential victim data [23, 28].
3. **Rogue Data Cache Load.** Meltdown attacks [31] leverage the fact that a processor checks page table permissions lazily right before an instruction retires. Consequently, a load can read data from any memory location into a CPU register and then leak that data via a side-channel before the processor determines that the page table permissions disallow the load and squash the instruction. This opens a window for attackers to speculatively read privileged memory through user-mode accesses.

   Meltdown [31] impacts user/kernel isolation and therefore is out of scope for Janus. However, a similar attack, *Read-Only Protection Bypass* [27], allows an attacker to bypass the Read/Write permission checks on a page table entry, speculatively storing to read-only data and thwarting isolation between domains. Similarly, we can envision a *PKU Protection Bypass* attack. Suppose that protection domain A attempts to load from an address x belonging to domain B. While the TLB entry that maps x has a protection key that makes x inaccessible to domain A, the processor performs the page permission check late in the pipeline [27, 31]. Domain A may thus speculatively load x into a processor register and leak it via the cache before the processor checks the protection key information in the TLB entry.
4. **Speculative Store Bypass.** Memory disambiguation predictors allow the processor to load data speculatively from an address to which a preceding store has yet to complete. An attacker can exploit this feature to speculatively load stale data into a processor register and leak it via side channels [27].

While our threat model (Section 2.1) excludes speculative side-channel attacks, our trampoline code should not introduce new Spectre vulnerabilities; doing so would hamper defenses against Spectre attacks that protected libraries may employ.

Both the Janus-VMFUNC and Janus-PKU trampolines are immune to bounds check bypass attacks and branch target injection attacks. Neither trampoline has a conditional or an indirect branch that an attacker can control by poisoning the branch predictors or branch target buffers. Likewise, neither trampoline computes an address to read or write based on untrusted input from the source or target domain. The Janus-VMFUNC trampoline is resistant to rogue data cache loads because TLB entries for data in mutually distrusting domains are cached with different tags in the TLB. In contrast, Janus-PKU is vulnerable to both the *Read-Only Protection Bypass* and *PKU Protection Bypass* attacks because the read/write permissions and protection key bits are part of the TLB access permissions which the processor may check late in the processor pipeline. Fortunately, hardware updates in future Intel processors [24] should mitigate these attacks against Janus-PKU. Finally, both the Janus-VMFUNC and Janus-PKU trampolines prevent *Speculative Store Bypass* attacks. The VMFUNC instruction is a serializing instruction [19], so no loads in the target domain can commence until the VMFUNC instruction has retired. The Janus-PKU trampoline executes an `LFENCE` instruction after switching the view between domains (listing 1, lines 10 and 29). The `LFENCE` enforces an ordering on instructions; no instruction following an `LFENCE` can execute, even speculatively, until all instructions preceding it have completed locally [24]. As a result, an attacker that launches a *Speculative Store Bypass* attack in one domain cannot corrupt addresses exclusively available to a different domain.

The different memory isolation mechanisms also affect the level of risk posed by Spectre attacks to libraries protected by Janus. As Section 4.5 explains, users may want to make different performance tradeoffs if they want stronger protections against Spectre attacks. Neither Janus-VMFUNC nor Janus-PKU mitigate bounds check bypass, branch target injection, and speculative store bypass attacks against protected library code. Libraries protected by Janus-VMFUNC are resilient to rogue data cache load attacks since TLB entries for data in mutually distrusting domains are cached with different tags in the TLB. Libraries protected by Janus-PKU are vulnerable to *Read-Only Protection Bypass* and *PKU Protection Bypass* attacks just like the Janus-PKU trampoline but, again, we expect future Intel processors to mitigate such attacks [24].

4. Evaluation

We evaluate Janus using microbenchmarks and three real-world applications in which we isolate a high-throughput data-plane library or in-memory database from the rest of the application. We run our microbenchmarks and in-memory database experiments on a Dell PowerEdge R640 server with two Intel Xeon Silver 4114 (Skylake) 2.20 GHz CPUs with 10 cores each and 16 GB of main memory. Our network experiments are conducted on Dell PowerEdge R640 servers equipped with two Intel Xeon E5-2630 v3 (Haswell) 2.40 GHz CPUs with 8 cores each and 64 GB of main memory. These machines are connected back-to-back through dual-port Mellanox ConnectX3-Pro 40 Gbps Host Channel Adapters (HCAs) to isolate their connection. All servers are running Fedora Linux using a 4.15 kernel with our modifications (except for baseline experiments, which use an unmodified kernel). All machines have hyper-threading and TurboBoost enabled.

We emulate the overhead of PKU on Haswell machines in a manner similar to other recent work [29, 49]. We verified the overhead of our emulation by comparing it with the PKU transition cost on the Skylake machine.

Graphs in this section are labeled as follows:

- `unprotected`: baseline system without Janus—kernel bypass with no intra-process isolation.
- `ptsw`: isolation via syscall-initiated page table switching, as described in Section 3.1.
- `vmfunc`: Janus-VMFUNC, as described in Section 3.2.
- `pku`: Janus-PKU, as described in Section 3.3.

Unless otherwise noted (and shown in legends with `-nopti`) experiments were conducted with kernel page-table isolation enabled for Meltdown mitigation [13]. We ran all experiments 10 times and report the arithmetic mean. We indicate 95% confidence intervals in all cases, but these are often so narrow as to be illegible in the bar graphs.

4.1. Benchmarks

We use microbenchmarks to measure the overhead of relevant instructions and basic operations as well as the latency of different implementations of Janus on the Skylake machine, which supports PKU. We also implement a no-op system call and a no-op VM call and measure their latencies. We use `rdtscp` with proper serialization [37] to measure the overhead of 1 million executions (again, computing the arithmetic mean across 10 runs).

Table 1 shows the calculated overhead of a single instance of each operation. The latency of writing to the `cr3` register impacts the syscall-based version of Janus; the latency of `VMFUNC` and `WRPKRU` impact the two preferred implementations. The cost of entering and leaving the kernel also impacts the syscall-based version; this cost itself depends on whether KPTI [13] is enabled. System calls in the virtualized environment, necessary for Janus-VMFUNC, would experience the overhead of VM calls.

For reference (and to put the overheads in perspective with respect to approaches like light-weight contexts (lwC) [32] which use processes to isolate domains), we also measured the cost of a context switch caused by a semaphore and of a user-space context switch using POSIX `getcontext` and `setcontext`.

Figure 3 compares the transition time from one domain to another (and back again) using our different implementations.

Table 1: Latency of Basic Operations

Instruction or Operation	Cycles*
write to `cr3` with `CR3_NOFLUSH`	186 ± 9
`vmfunc`	109 ± 1
`wrpkru`	26 ± 2
no-op system call w/ KPTI	433 ± 12
no-op system call w/o KPTI	96 ± 2
no-op VM call	1694 ± 131
user-space context switch	748 ± 8
process context switch using semaphore	4426 ± 41

* ± half the width of the 95% confidence interval

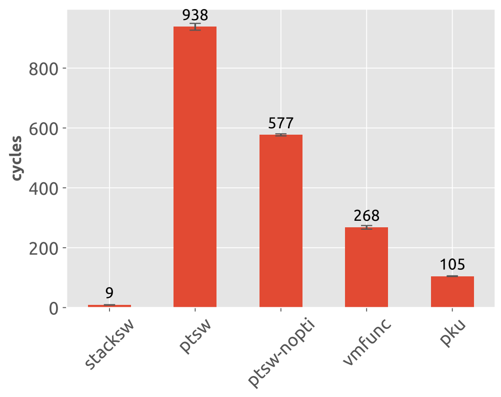

Figure 3: Transition Microbenchmarks.

Additionally, we measured the cost of switching stacks without providing isolation as it contributes a small amount to all implementations of Janus. To do this, we removed the code in Listing 1 that changes domain and calls the protected library function. Figure 3 denotes the average stack switch time as `stacksw`. We also measured the cost of page table switching with kernel page-table isolation disabled; Figure 3 denotes this as `ptsw-nopti`.

Among the implementations of isolation, Janus-PKU has the lowest transition cost, followed by Janus-VMFUNC. This matches the results in Table 1: changing the `pkru` register costs much less than using `vmfunc`. The two required system calls dominate the cost of the page table switching implementations. Relative to `ptsw-nopti`, kernel page table isolation incurs a penalty of 62%. Stack switching itself has an almost negligible impact.

4.2. Silo

Silo [45] is a scalable in-memory database which uses optimistic concurrency control and periodically updated epochs to provide the same guarantees as a serializable database without the scalability bottlenecks. It is implemented as a library linked to the benchmark. Each benchmark thread issues transactions (of YCSB [7] or TPC-C [44] workloads) in a loop. In the experiments here, we identify the main Silo library as a separate domain whose pages are protected from the benchmark driver. The benchmark calls (trampolines of) library

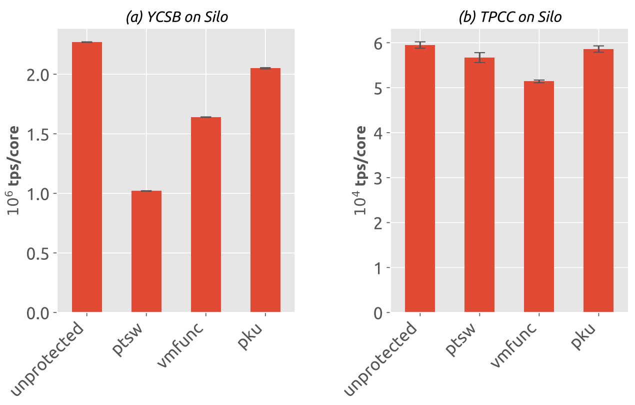

Figure 4: Silo Benchmarks.

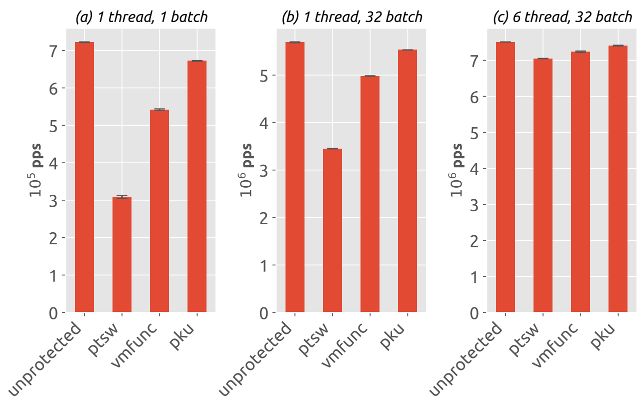

Figure 5: DPDK Raw Packet Forwarding Benchmarks.

routines to perform transactions (one domain transition per transaction). All data and metadata reside in memory, and the workload is CPU intensive.

Figure 4 shows the overhead of isolation for the YCSB [7] and TPC-C [44] workloads on the Skylake machine. Both use the synchronous database API in Silo, precluding batching and necessitating a very high switching rate. Both workloads were run with 20 threads.

YCSB [7] is a key-value benchmark with tiny transactions. We first fill the database with 1 million records and then we run a workload with an 80/20 read/write mix. The unmodified Silo reaches 2.27 million transactions per second on each core. Janus incurs 54%, 27%, and 9.85% overhead in the PT-Switch, VMFUNC, and PKU implementations, respectively.

TPC-C [44] is a relational database benchmark with significantly larger transactions compared to YCSB [7]. As a result, the maximum number of transaction per second is reduced to around 600,000 per core on unmodified Silo. With a lower rate of library transitions, the overhead of Janus drops to 4.66%, 13.6%, and 1.5% for the PT-Switch, VMFUNC, and PKU implementations, respectively. While Janus-VMFUNC incurs the largest overhead in this experiment, we discovered that 12% of that overhead is due to running inside a VM. We suspect that is caused by frequent use of `nanosleep` system call in the benchmark's epoch-based garbage collector. While we have not attempted to modify applications to remove system calls (or to replace them with equivalent functionalities that don't cause VM exits), we believe that such a change would be straightforward in this case.

4.3. DPDK TestPMD

Intel's Data Plane Development Kit (DPDK) [20] is a set of data-plane libraries that implement kernel bypass, polling drivers, and a fast packet processing framework. Packet processing applications can link against one or more of the DPDK libraries and use them to access network devices directly. In this section, we evaluate the performance of Janus with a packet-forwarding application, `testpmd`, distributed for performance testing as a part of DPDK. Running on the Haswell machines with dual-port Mellanox ConnectX-3 HCAs, this benchmark receives raw 1500-byte packets from one port of the HCA and forwards them directly to another port without accessing packet contents. We connect two hosts back-to-back to create an endless forwarding of packets in an isolated network. We separate the packet-forwarding logic from the DPDK library using Janus.

Figure 5 shows the effect of Janus on `testpmd` throughput with different thread counts and batching degrees (packets per library call / domain transition). We report throughput in packets forwarded per second as measured by `testpmd`. As a worst-case scenario for Janus overhead, we configure the benchmark to use only a single thread and to forward packets one-by-one without batching. (Such a configuration would not be common in practice.) The unmodified DPDK in this setting can forward more than 720,000 packets per second, and the overhead of Janus is less than 25% with VMFUNC and 7% with PKU. As we increase the batch size (Fig. 5(a) v. (b)), the number of processed packets per transition increases and the overhead of switching becomes a smaller part of overall run time. As we provide more threads and therefore more CPU ((b) v. (c)), performance of all approaches improves but the gaps decrease since the abundance of CPU resources makes the network line rate the new throughput limiter.

4.4. Redis on DPDK

Redis [40] is a NoSQL store that serves read requests from an in-memory data-structure. Redis can also store data on persistent secondary storage using snapshots; we disabled this functionality in our experiments to avoid the overhead of system calls. The Redis server uses TCP to receive requests from clients. In our set-up, we use a user-space network stack called F-Stack [43] on top of the DPDK packet processing framework and driver to provide connections to Redis clients. We use Janus to isolate the network and packet processing stack from the Redis data store logic—i.e., both F-Stack and DPDK run within the same protection domain. We run YCSB [7] on a remote client to benchmark the server configuration. Both the YCSB client and the Redis server are running on the Haswell

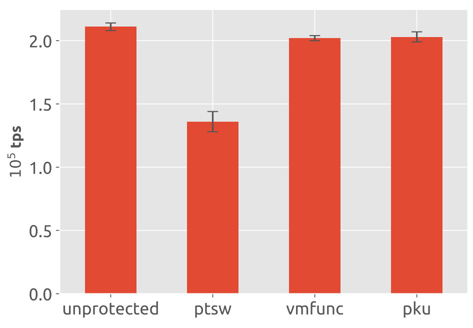

Figure 6: Redis Benchmark.

machines, connected back-to-back via Mellanox ConnectX-3 HCAs. The server here is the bottleneck: Redis is single threaded; it runs a loop that waits for request arrival using an `epoll`-like call to F-Stack, receives and processes the requests, and then sends results back with F-Stack's equivalent of the `send` system call. As a result, there are at least two domain transitions per transaction.

To measure the effect of Janus on Redis, we first loaded the Redis server with 1 million records each of length 1200 bytes. We then ran a YCSB workload [7] with a 95%/5% read/write proportion and measured how many transactions per second the Redis server supported. Figure 6 shows the results as measured and reported by YCSB client. The unmodified server can support 220,000 transactions per second. The PT-Switch, VMFUNC, and PKU implementations of Janus reduce the throughput of Redis by 35%, 5%, and 2.78%, respectively.

4.5. Discussion

The preceding subsections reveal significant performance differences among our three implementations of Janus: syscall-based page table switching, Janus-VMFUNC, and Janus-PKU. These differences must be considered together with the issues of generality and confidentiality when applying Janus in a particular environment.

The overhead of Janus-PKU is very low, even at millions of domain switches per second (Fig. 4(a)). Janus-VMFUNC has higher overhead but is still significantly faster than page table switching in most cases. With a lower number of transitions per second (Figs. 4(b), 6), possibly effected via batching or multiple worker threads (Fig. 5(b)–(c)), the advantage of PKU over VMFUNC diminishes substantially. In any case, both Janus-VMFUNC and Janus-PKU remain considerably faster than syscall-based page table switching in most cases (Figs. 5(a)–(b), 6).

A significant limitation of Janus-PKU is its assumption (and verification) that the application contains no `WRPKRU` instructions that would make protected memory accessible outside the corresponding library. Current Intel hardware, moreover, supports only 16 distinct memory keys. The need for a separate key for each of the regions of the "protection domain Venn diagram" (e.g., Fig. 2) thus limits us to no more than 7 protected libraries in any given application—fewer if they wish to make direct calls to one another.

Janus-VMFUNC has no similar limitations on generality. There are 512 distinct function codes on current Intel machines, and a VM that uses some of these for its own purposes is still compatible with Janus-VMFUNC. As discussed in Section 3.2, however, Janus-VMFUNC does require that the application run inside a virtual machine. This restriction imposes significant cost on system calls elsewhere in the application, which (unless they are moved to a thread outside the VM) now incur the latency of a VM exit; we see the impact of this latency in Figure 4(b). Ideally, we would like to see support on future hardware for a VMFUNC-like mechanism that allows a non-privileged instruction to switch among pre-approved page table root pointers without the need for virtualization.

For users eager to mitigate speculative side-channel attacks, Janus-PKU is problematic on current hardware as the trampoline fails to mitigate Meltdown-style attacks [27, 31] against protected library code. Janus-VMFUNC does mitigate such attacks, providing stronger confidentiality.

5. Related Work

There are three areas of related work: fast I/O systems that move device and resource management into user space, methods of isolating software components sharing the same virtual address space, and systems that impose security policies on operating system kernels and hypervisors from which we draw inspiration for parts of Janus's design.

5.1. Fast I/O Systems

Existing kernel-bypassing solutions do not protect libraries from untrusted applications. Arrakis [38] uses a library OS without isolation in the same address space as the application and relies on device-level SR-IOV [22] support. Device-level resource isolation policies are often rigid, e.g., limited to simple partitioning. Janus protected libraries enable more powerful protection policies like proportional bandwidth sharing and even safe, concurrent accesses to the same data. IX [3] and ZygOS [39], both of which build on top of Dune [2], use virtualization to run their kernel-bypass stack in ring 0 of VMX non-root mode. While this design can be extended to provide Janus-like isolation, it is limited to only a single trusted domain and does not support multiple distrusting data-plane libraries within the same application, as Janus does.

Kernel-based high-throughput software stacks like MegaPipe [14] and StackMap [55] depend on aggressive batching to limit the protection domain switching frequency and costs. However, large I/O batching requires asynchronous programming models that are generally hard to employ and not always supported by library APIs. For example, in last section's experiments, we were unable to batch over Silo database API [45] and F-Stack [43] send calls.

5.2. Intra-Process Isolation

There has been much previous work on intra-process isolation mechanisms. The method with least overhead is to write code in type-safe languages. Work in single address space operating systems such as Singularity [16] and Verve [54] shows that application and kernel code can execute safely within the same virtual address space. The disadvantage of such systems is that they require applications to be rewritten in type-safe languages. Janus, in contrast, supports existing fast I/O applications.

For type-unsafe code, approaches such as SFI [51] and XFI [46] employ either source- or binary-level instrumentation to guarantee that code cannot read or write outside of designated sections of the virtual address space. Load and store instrumentation either checks that the accessed address in within bounds or transforms out-of-bounds pointers to in-bounds pointers. SFI [51] incurs an average overhead of 17.6% for read-write protection and 4.3% overhead when only instrumenting writes. Janus works without sophisticated binary rewriting techniques and incurs less overhead than SFI by leveraging newer hardware support.

Hardware mechanisms can isolate code running within the same virtual address space. CODOMs [50] and CHERI [53] augment instructions with capabilities. Segmentation also provides intra-process isolation [41] by requiring code to possess a *descriptor* to address a particular section of memory. By restricting which descriptors are accessible to various code components, the operating system kernel can isolate untrusted components. Segmentation is supported in 32-bit but not 64-bit x86 systems [19]. ARM memory domains [12] are similar to Intel PKU [19] but are only available for 32-bit ARM processors, and memory domain permissions can only be modified in supervisor mode. Our work focuses on hardware support available in 64-bit x86 systems.

ERIM [49], developed concurrently to our work, uses protection keys like Janus to provide an isolated domain within a single virtual address space. Janus supports multi-threaded applications with multiple mutually distrusting libraries, which ERIM currently does not. Janus-PKU also addresses Speculative Store Bypass attacks [27] while ERIM does not, and we analyze the dangers of other speculation side-channel attacks on both Janus-VMFUNC and Janus-PKU.

Several operating system abstractions are similar to our work. *Shreds* [6] use ARM memory domains [12] to divide execution within a user-space thread. Each shred is a thread fragment with a private memory pool for storing secret data and sensitive code. Light-weight contexts (*lwCs*) [32] isolate units within an address space. Each *lwC* has its own virtual memory mappings, file descriptors, access rights and execution state. *Secure Memory Views (SMV)* [15] use per-thread page tables to enforce isolation while allowing sharing between threads. SMV does not support multiple domains within a thread. Shreds [6], lwCs [32], and SMVs [15] require a system call for domain transitions while Janus does not. Janus also works with unmodified applications.

MemSentry [29] uses EPT-switching VMFUNC, MPK and Intel MPX [19] to provide a single protected domain within a process. In contrast, Janus supports multiple distrusting components. *SeCage* [33] uses static and dynamic compiler analysis to decompose a monolithic program into different domains and uses EPT-switching VMFUNC to prevent memory disclosure attacks even when running on a compromised OS. Unlike SeCage, Janus relies on existing explicit library boundaries, alleviating the need for compiler analysis to extract components. Janus also does not execute the operating system kernel within a virtual machine; only the intended application executes within a virtual machine while all other applications run natively. Unlike SeCage [33], Janus does not protect applications from a malicious OS.

VMFUNC has been used for communication between components isolated at coarse granularity. High-throughput network function virtualization has used VMFUNC and EPT-switching to provide efficient communication between VMs hosting different network functions [35]. *CrossOver* [30] proposes a cross-world interaction mechanism that provides communications between VMs as well as different address spaces and privilege levels in or between VMs. It uses EPT-switching VMFUNCs to approximate the cost of cross-world interaction and suggests architectural changes to VMFUNC to allow such calls. While CrossOver can theoretically be used for intra-process isolation, the paper focuses on providing cross-world calls as a generic communication mechanism.

5.3. Operating System and Hypervisor Security

Janus utilizes lessons learned from previous work on enforcing security on OS kernels and hypervisors. Our mitigations for vulnerabilities in signal handler dispatch are based on KCoFI [8] which, like our system, avoids saving interrupted program state on the user-space stack to prevent corruption by signal handler code. Similarly, the design of the Janus-PKU trampoline is inspired by the Nested Kernel [11] trampoline code. Both Janus-PKU and Nested Kernel must check that the inputs to domain switching instructions are correct because neither system enforces control flow integrity [1]. Finally, Janus's restrictions on `mmap` to enforce code segment integrity are similar to code segment integrity protections in Secure Virtual Architecture [10], HyperSafe [52], and Nested Kernel [11].

6. Conclusions

We have introduced Janus, an in-process isolation system for fast data-plane libraries. We presented two concrete implementations for current Intel processors—one based on VMFUNC, the other on a novel use of memory protection keys. Experiments with microbenchmarks, the Silo in-memory database, the Data Plane Development Kit (DPDK), and the Redis NoSQL store confirm that Janus can provide full isolation of protected libraries while approaching the performance of unprotected kernel bypass. Janus-PKU, in particular, provides 90–98% of kernel-bypass throughput in all of our experiments.

Careful study of Janus trampoline code suggests only a single vulnerability to Spectre data leaks in the trampoline in Janus-PKU, and this is expected to be fixed in future Intel processors. Our study also shows that Janus-VMFUNC mitigates rogue data cache load attacks against protected library code while Janus-PKU cannot, but again, we expect future Intel processors to fix the issue.

Ideally, Janus would use a VMFUNC-like instruction that switched among pre-approved page table root pointers without requiring virtualization. We encourage hardware designers to consider such an extension. We would also welcome a variant of PKU with a larger number of keys and with coverage of execute rights. In future work, we hope to evaluate the cost of a Janus implementation based on software fault isolation [51] and to explore hardware-supported implementations for additional processor architectures (ARM and Power in particular).

Acknowledgement

Our thanks to Joe Izraelevitz, who encouraged us to explore this topic and provided valuable initial ideas.

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Trans. on Information Systems Security*, 13:4:1–4:40, November 2009.

[2] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe User-level Access to Privileged CPU Features. In *10th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 335–348, Hollywood, CA, October 2012.

[3] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *11th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 49–65, Broomfield, CO, October 2014.

[4] E. Bosman and H. Bos. Framing Signals—A Return to Portable Shellcode. In *35th IEEE Symp. on Security and Privacy (SP)*, pages 243–258, San Jose, CA, May 2014.

[5] D. P. Bovet and Marco Cesati. *Understanding the LINUX Kernel*. O'Reilly, Sebastopol, CA, 2nd edition, 2003.

[6] Y. Chen, S. Reymondjohnson, Z. Sun, and L. Lu. Shreds: Fine-Grained Execution Units with Private Memory. In *37th IEEE Symp. on Security and Privacy (SP)*, pages 56–71, Oakland, CA, May 2016.

[7] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *1st ACM Symp. on Cloud Computing (SoCC)*, pages 143–154, Indianapolis, IN, June 2010.

[8] John Criswell, Nathan Dautenhahn, and Vikram Adve. KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels. In *35th IEEE Symp. on Security and Privacy (SP)*, pages 292–307, San Jose, CA, May 2014.

[9] John Criswell, Nathan Dautenhahn, and Vikram Adve. Virtual Ghost: Protecting Applications from Hostile Operating Systems. In *19th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 81–96, Salt Lake City, UT, March 2014.

[10] John Criswell, Nicolas Geoffray, and Vikram Adve. Memory Safety for Low-Level Software/Hardware Interactions. In *18th USENIX Security Symp.*, pages 83–100, Montreal, PQ, Canada, August 2009.

[11] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. 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)*, pages 191–206, Istanbul, Turkey, March 2015.

[12] ARM Memory Domains. `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0211k/Babjdffh.html`.

[13] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. KASLR is Dead: Long Live KASLR. In Eric Bodden, Mathias Payer, and Elias Athanasopoulos, editors, *Intl. Symp. on Engineering Secure Software and Systems*, pages 161–176, Bonn, Germany, July 2017.

[14] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. MegaPipe: A New Programming Interface for Scalable Network I/O. In *10th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 135–148, Hollywood, CA, October 2012.

[15] Terry Ching-Hsiang Hsu, Kevin Hoffman, Patrick Eugster, and Mathias Payer. Enforcing Least Privilege Memory Views for Multithreaded Applications. In *ACM SIGSAC Conf. on Computer and Communications Security (CCS)*, pages 393–405, Vienna, Austria, October 2016.

[16] Galen C. Hunt, James R. Larus, Martín Abadi, Mark Aiken, Paul Barham, Manuel Fähndrich, Chris Hawblitzel Orion Hodson, Steven Levi, Nick Murphy, Bjarne Steensgaard, David Tarditi, Ted Wobber, and Brian Zill. An Overview of the Singularity Project. Technical Report MSR-TR-2005-135, Microsoft Research, October 2005.

[17] Intel Corp. Intel Optane SSD DC P4800X Series. `http://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/data-center-ssds/optane-dc-p4800x-series/p4800x-750gb-aic.html`.

[18] Intel Corp. Analysing Potential Bounds Check Bypass Vulnerabilities. `http://software.intel.com/sites/default/files/managed/4e/a1/337879-analyzing-potential-bounds-Check-bypass-vulnerabilities.pdf?source=techstories.org`, July 2018.

[19] Intel Corp. *Intel 64 and IA-32 Architectures Software Developer's Manual*, May 2018. 325462-067US.

[20] Intel Corp. Intel DPDK: Data Plane Development Kit, 2018. `http://www.dpdk.org`.

[21] Intel Corp. Intel SPDK: Storage Performance Development Kit, 2018. `http://www.spdk.io`.

[22] Intel Corp. PCI-SIG SR-IOV Primer: An Introduction to SR-IOV Technology, 2018. `http://www.intel.sg/content/dam/doc/application-note/pci-sig-sr-iov-primer-sr-iov-technology-paper.pdf`.

[23] Intel Corp. Retpoline: A Branch Target Injection Mitigation. `http://software.intel.com/sites/default/files/managed/1d/46/Retpoline-A-Branch-Target-Injection-Mitigation.pdf`, June 2018.

[24] Intel Corp. Speculative Execution Side Channel Mitigations. `http://software.intel.com/sites/default/files/managed/c5/63/336996-Speculative-Execution-Side-Channel-Mitigations.pdf`, May 2018.

[25] Eun Young Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems. In *11th USENIX Conf. on Networked Systems Design and Implementation (NSDI)*, pages 489–502, Seattle, WA, April 2014.

[26] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *44th Intl. Symp. on Computer Architecture (ISCA)*, pages 1–12, Toronto, ON, Canada, June 2017.

[27] V. Kiriansky and C. Waldspurger. Speculative Buffer Overflows: Attacks and Defenses. *ArXiv e-prints*, July 2018.

[28] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. *CoRR*, abs/1801.01203, 2018.

[29] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. No Need to Hide: Protecting Safe Regions on Commodity Hardware. In *12th ACM SIGOPS European Conf. on Computer Systems (EuroSys)*, pages 437–452, Belgrade, Serbia, April 2017.

[30] Wenhao Li, Yubin Xia, Haibo Chen, Binyu Zang, and Haibing Guan. Reducing World Switches in Virtualized Environment with Flexible Cross-world Calls. In *42nd Intl. Symp. on Computer Architecture (ISCA)*, pages 375–387, Portland, Oregon, June 2015.

[31] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *CoRR*, abs/1801.01207, 2018.

[32] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. Light-weight Contexts: An OS Abstraction for Safety and Performance. In *12th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 49–64, Savannah, GA, November 2016.

[33] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. Thwarting Memory Disclosure with Efficient Hypervisor-enforced Intra-domain Isolation. In *22nd ACM SIGSAC Conf. on Computer and Communications Security (CCS)*, pages 1607–1619, Denver, CO, October 2015.

[34] Konstantinos Menychtas, Kai Shen, and Michael L. Scott. Disengaged Scheduling for Fair, Protected Access to Fast Computational Accelerators. In *19th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 301–316, Salt Lake City, UT, March 2014.

[35] Jun Nakajima. Xen as High-Performance NFV Platform, August 2018. `http://events.static.linuxfound.org/sites/events/files/slides/XenAsHighPerformanceNFVPlatform.pdf`.

[36] Faisal Nawab, Joseph Izraelevitz, Terrence Kelly, Charles B. Morrey III, Dhruva R. Chakrabarti, and Michael L. Scott. Dalí: A Periodically Persistent Hash Map. In *31st Intl. Symp. on Distributed Computing (DISC)*, pages 37:1–37:16, Vienna, Austria, September 2017.

[37] Gabriele Paoloni. How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures. `http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf`.

[38] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System is the Control Plane. In *11th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 1–16, Broomfield, CO, October 2014.

[39] George Prekas, Marios Kogias, and Edouard Bugnion. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *26th Symp. on Operating Systems Principles (SOSP)*, pages 325–341, Shanghai, China, October 2017.

[40] Redis Labs. Redis, 2018. `http://www.redis.io`.

[41] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proc. of the IEEE*, 63(9):1278–1308, September 1975.

[42] Lisa Spelman. Reimagining the Data Center Memory and Storage Hierarchy, May 2018. `newsroom.intel.com/editorials/re-architecting-data-center-memory-storage-hierarchy/`.

[43] Tencent Corp. F-Stack, 2018. `http://www.f-stack.org`.

[44] Transaction Processing Countil. TPC-C Benchmark, 2018. `http://www.tpc.org/tpcc`.

[45] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy Transactions in Multicore In-memory Databases. In *24th ACM Symp. on Operating Systems Principles (SOSP)*, pages 18–32, Farmington, PA, November 2013.

[46] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software Guards for System Address Spaces. In *7th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 75–88, Seattle, WA, November 2006.

[47] US-CERT. Alert (TA18-004A). `http://www.us-cert.gov/ncas/alerts/TA18-004A`.

[48] US-CERT. Alert (TA18-141A). `http://www.us-cert.gov/ncas/alerts/TA18-141A`.

[49] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, and Peter Druschel. ERIM: Secure and Efficient In-process Isolation with Memory Protection Keys. *CoRR*, abs/1801.06822, 2018.

[50] Lluïs Vilanova, Muli Ben-Yehuda, Nacho Navarro, Yoav Etsion, and Mateo Valero. CODOMs: Protecting Software with Code-centric Memory Domains. In *41st Intl. Symp. on Computer Architecuture (ISCA)*, pages 469–480, Minneapolis, MN, June 2014.

[51] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-based Fault Isolation. In *14th ACM Symp. on Operating Systems Principles (SOSP)*, pages 203–216, Asheville, NC, December 1993.

[52] Zhi Wang and Xuxian Jiang. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *31st IEEE Symp. on Security and Privacy (SP)*, pages 380–395, Oakland, CA, May 2010.

[53] R. N. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *36th IEEE Symp. on Security and Privacy (SP)*, pages 20–37, San Jose, CA, May 2015.

[54] Jean Yang and Chris Hawblitzel. Safe to the Last Instruction: Automated Verification of a Type-Safe Operating System. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 99–110, Toronto, ON, Canada, June 2010.

[55] Kenichi Yasukata, Michio Honda, Douglas Santry, and Lars Eggert. StackMap: Low-latency Networking with the OS Stack and Dedicated NICs. In *USENIX Annual Technical Conf. (ATC)*, pages 43–56, Denver, CO, June 2016.