Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries

Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, and Michael L. Scott, ***University of Rochester;*** **Kai Shen and Mike Marty,** ***Google***

https://www.usenix.org/conference/atc19/presentation/hedayati-hodor

This paper is included in the Proceedings of the 2019 USENIX Annual Technical Conference.

July 10–12, 2019 • Renton, WA, USA

ISBN 978-1-939133-03-8

Open access to the Proceedings of the 2019 USENIX Annual Technical Conference is sponsored by USENIX.

Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries

Mohammad Hedayati
University of Rochester

Spyridoula Gravani
University of Rochester

Ethan Johnson
University of Rochester

John Criswell
University of Rochester

Michael L. Scott
University of Rochester

Kai Shen
Google

Mike Marty
Google

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 safety and provides resource multiplexing, 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 distrusting, 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 data plane libraries. We propose *protected libraries* as a new OS abstraction which provides separate user-level protection domains for different services (e.g., network and in-memory database), with performance approaching that of unprotected kernel bypass. We also show how this new feature can be utilized to enable sharing of data plane libraries across distrusting applications. Our proposed solution uses Intel's memory protection keys (PKU) in a safe way to change the permissions associated with subsets of a single address space. In addition, it uses hardware watchpoints to delay asynchronous event delivery and to guarantee independent failure of applications sharing a protected library.

We show that our approach can efficiently protect high-throughput in-memory databases and user-space network stacks. Our implementation allows up to 2.3 million library entrances per second per core, outperforming both kernel-level protection and two alternative implementations that use system calls and Intel's VMFUNC switching of user-level address spaces, respectively.

1 Introduction

A principal task of an operating system (OS) 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 [21] and mTCP [24] move packet processing and transport layer processing into user space; SPDK [22] does the same for direct access to fast storage devices like Optane SSDs [19]. Accelerators like Google's TPU [25] 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 [4, 39, 40]. 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 [23]). Unfortunately, device-level resource isolation is not always available and typically lacks the flexibility to implement OS-level resource management policies.

The anticipated widespread availability of byte-addressable non-volatile memory (NVM) DIMMs [45] brings similar challenges. If NVM is mapped into a process's address space so that it can be accessed directly with application load/store instructions, a memory safety error within the process could corrupt data structures on the NVM [37]. Relying on OS kernel mechanisms e.g., a file system interface, to protect access to NVM would throw away the performance potential of direct loads and stores to persistent memory.

One can, of course, implement protection domains within an address space using a trusted compiler with static [17] or dynamic [52] checking. The static approach 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 [44].

What we desire is a mechanism that allows services traditionally implemented in the kernel to be encapsulated as *protected libraries* in user space. Such a mechanism should be compatible with existing applications (i.e., via re-linking), provide fast transitions into and out of protected library routines, impose little or no cost on ordinary code, accommodate multiple protected code and data segments in a single application, and support independent failure to allow a protected library to be shared across distrusting applications. Toward that end, this paper proposes *Hodor*, a mechanism for low-overhead intra-process isolation. Hodor leverages the existence of user libraries to define protection domains for services previously offered by the kernel (e.g., file systems, network stack, device drivers, etc.). Relying on library boundaries, Hodor offers practical intra-process isolation without requiring any significant effort on the part of the application programmer. It allows multiple mutually distrusting libraries to be loaded into the same address space, providing each library (and the main application) with a different "view" of code and data, and protecting each from failures in the others. (When a failure occurs, library calls in non-erroneous protection domains are permitted to complete before the process terminates.) Hodor employs the standard function call/return interface but interposes a *trampoline* on each call to change the view of the address space to that of the library being entered.

Hodor can be used to provide instances of a protected library in multiple applications with access 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 propose a concrete implementation of Hodor for recent Intel processors that is based on Intel's memory protection keys, called *Protection Keys for Userspace (PKU)* [20]. We introduce a novel method that uses hardware watchpoints (i.e., debug registers [20]) to efficiently monitor program execution and ensure the safety of our approach without relying on a trusted compiler, changes to application source code, or expensive dynamic binary translation.

We also describe two alternative implementations of Hodor's isolation: 1) using a system call to switch between page tables, and 2) using Intel's Extended Page Table (EPT) switching with VM function (VMFUNC) instructions [20]. We compare our PKU-based protection with each of the alternative solutions and demonstrate that the PKU approach offers better performance. While none of the implementations is fast enough to be used for fine-grained intra-process isolation (e.g., for shadow stacks [7] or code-pointer integrity [50]), our results show that both PKU- and VMFUNC-based approaches are able to support on the order of two million calls per second per core into a protected library.

In summary, our contributions are as follows:

- We introduce Hodor, a mechanism that provides a new OS abstraction to isolate fast data-plane libraries from both the calling application and each other.
- We propose a concrete implementation of Hodor for current Intel processors based on PKU. We present a novel method that combines binary inspection and hardware watchpoints to prevent bypassing of the PKU-based protection and safely isolate libraries linked in arbitrary x86 applications.
- We quantify the performance benefits of Hodor on real-world applications with respect to both unprotected kernel bypass and isolation based on kernel-mediated page table switching and EPT switching via VMFUNC.
- We present two proof-of-concept examples of protected libraries that share state between library instances in separate applications (with independent failure modes), and discuss challenges that must be addressed in such designs.

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, threading libraries, operating system kernel) that must be modified to ensure isolation. Sec. 3 then describes our candidate implementations. We evaluate the performance of these implementations in Sec. 4 using microbenchmarks, the Silo in-memory database [48], the DPDK data-plane library package [21], and the Redis [42] NoSQL server. Sec. 5 discusses related work. Sec. 6 summarizes our conclusions.

2 Protected Libraries

Hodor's *protected library* mechanism partitions an application into multiple domains of executable code. Each domain is granted access to some parts of the address space and denied access 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 standard calling conventions, mediated by *trampoline* routines that switch to the appropriate address space view, switch stacks, set up arguments to maintain calling conventions, and possibly scrub any remaining registers to avoid information leaks. Trampolines also switch back to the caller's domain when a library call returns.

2.1 Threat Model

With Hodor, an untrusted application uses protected libraries to access protected resources. A resource might comprise ordinary memory, non-volatile memory, or a memory-mapped device. By default, an application shares its entire memory space with each protected library, but the library shares only the *trampoline* code needed for cross-domain calls. In addition, an application can optionally be modified to share only buffers with the library.

Figure 1 shows an example with user-space network and storage libraries. The storage library has default access to the

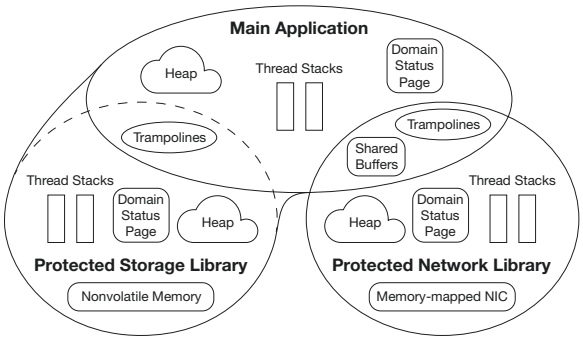

Figure 1: Protected Library Architecture. The example application has, by default, shared its entire memory space with the storage library. It has opted to share only certain buffers with the network library.

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. When a protected library opts to share state with instances in other applications (e.g., to track resource usage and enforce fairness), the library is responsible for synchronization. We accommodate independent failures by arranging to complete the execution of any library call whose process incurs a fault in a different domain (more on this in Sec. 2.5).

As replacements for traditional kernel services, protected libraries are assumed to be written with care. Among other things, they should employ caution when dealing with potentially unsafe arguments (e.g., using methods like copy-in/copy-out) just as kernel code would. They should also ensure that transitions into other domains (e.g., invocation of a callback or a third-party library function) happen in a safe context. It would be incorrect, for example, to acquire a lock and then call an outside function, since it might terminate before returning back to the library. In our implementation, protected libraries are statically linked with all their dependencies to ensure that transitions into and out of the library conform to its API. This does not prevent use of shared libraries by the rest of the application. A more flexible implementation could dynamically link a separate instance of shared libraries in each protection domain that requires them.

A protected library is trusted to enforce security and management policies for its protected resources but is otherwise untrusted. The hardware and the operating system are part of the Trusted Computing Base (TCB), and are assumed to be 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. While side-channel attacks, and in particular those targeting transient execution [8, 27, 30], are out-of-scope, we explain the ramifications of our implementations on related transient execution attacks.

We assume that an 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 identifies the context in which the syscall was made (i.e., the value of `pkru` register for the PKU-based implementation and the instruction pointer for page-table switching implementations) and grants requests to change the mappings or permissions of protected library space only when made in an appropriate context.

2.3 Local Variables and Protected Stack

Within its protected memory, in addition to code, global variables, and heap, each protected library also maintains per-thread private stacks on which to store return addresses and local variables. When an application creates a new thread, we must create a new stack for each of the domains in the application. We embed 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 arrange for the target function to execute in the library's protection domain, where it can access its protected code and data. In particular, the trampoline must switch the stack pointer to the local stack of the calling thread. Sec. 3 describes the design of our trampolines for each intra-process isolation mech-

anism in more detail. If a protected library invokes a callback function within the application, it will also use trampoline code to switch back to the application's domain and 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 [31] or by not supporting mutually distrusting domains [50]. 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 (`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.

Like stacks, 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 maintains a list (readable, but not writable in user space) of currently running thread IDs for all CPU cores. The fast `gettid` performs a vDSO `getcpu` lookup and uses 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

Hodor employs a trusted loader, running as root (to allow it to open I/O device files) 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 function, 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. The initialization function also allocates the first stack, initializes the heap, and calls 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 the location of each protected library. This allows the kernel to enforce restrictions on system calls that configure the virtual address space (as per Sec. 2.2). The trusted loader then loads the application code and all other pre-loaded dynamic libraries. If inspection is required (as in the PKU-based version of Hodor that we introduce in Sec. 3.3), the loader performs it now; the kernel arranges for similar inspection on any additional libraries that are loaded on demand and on any other pages for which 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 transfers control to the application's `main` routine.

2.5 Asynchronous Events and Termination

To support unmodified applications, Hodor must address asynchronous event delivery via signals—in particular, the possibility that the kernel might invoke a signal handler in a thread that was executing trampoline or protected library code. Such a handler might then gain access to protected library state.

In a similar vein, termination of a process while a thread is executing protected library code could leave data structures (possibly shared with library instances in other applications) in an inconsistent state or cause deadlock (by failing to release a lock). To preclude protection violations in signal handlers and to accommodate independent failures of processes whose libraries share state, we modify the kernel so that it never delivers a signal or terminates the process while threads are still running protected library or trampoline code. Instead, it places a *hardware watchpoint* (using registers `DR0`–`DR3` on an Intel machine) in the "boundary trampoline" used to exit protected libraries (line 37 of Listing 1), and delays signal delivery or termination until the watchpoint has been triggered.

As noted in Sec. 2.1, we assume that protected libraries are written with care. In particular, we assume that their operations take modest, bounded time. If a protected library does not return in a timely fashion after we install the hardware watchpoint on the trampoline (detected by expiration of a timer initiated when we arm the watchpoint), we assume that the library is defective and terminate the application (having given any other, non-defective libraries time to finish execution). We perform a similar summary cleanup if a fatal error (e.g., a `SIGSEGV`) is caused *by* library code.

Our design does not permit signal handlers to be registered for execution by a protected domain. None of the privileged library use cases we evaluated need signal handling. If they did, the kernel's signal handler API could, in principle, 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 Sec. 2.4), it could confirm whether a registration request was made from trusted library code and allow or deny the request accordingly.

3 Fast Memory Isolation

This section presents three implementations of memory isolation for Hodor. The most straightforward implementation, described in Sec. 3.1, relies on separate page tables for each domain, and uses system calls to change the page table root pointer. Hodor-VMFUNC, described in Sec. 3.2, also uses per-domain page tables, and switches between them using Intel's VM Function (VMFUNC) mechanism. Both the syscall-based system and Hodor-VMFUNC rely on context identifier tags (Intel's PCID and EP4TA, respectively) to avoid flushing the translation lookaside buffer (TLB) when changing domains.

Hodor-PKU, our preferred solution (described in Sec. 3.3), uses memory protection keys to provide different access rights in the same page table. Since the access rights for each domain can be modified by user code, we need to prevent an application from bypassing our PKU-based isolation mechanism. We present a novel method in Sec. 3.3.1 that combines binary inspection and the use of hardware monitors for efficient run-time monitoring to ensure the safety of Hodor-PKU.

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, Hodor-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. Executable pages appear only in the tables of their corresponding domains. Protected pages rely on protection bits in the page tables of each domain to prevent undesired accesses. Unprotected application pages and trampoline pages appear in all page tables. 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, are 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 top-level page for each table, but many of the lower-level pages are physically shared. This approach simplifies entry manipulation and minimizes memory footprint.

3.2 Hodor-VMFUNC

Beginning with its Nehalem generation of processors, Intel has provided *extended page tables (EPT)* 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. This mechanism allows a guest to pre-register a set of second-level page tables and provides a (non-privileged) instruction to switch amongst them. Several systems (e.g., SeCage [32] and MemSentry [28]) have used VMFUNC to isolate a sensitive region within an application, but, they require source-code analysis and non-trivial modifications to existing applications.

Hodor-VMFUNC isolates a memory region 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 the approach of Sec. 3.1, which 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 inevitable (absent major architectural changes e.g., CODOMs [51] and CHERI [55], which themselves impose new overheads), using VMFUNC avoids the need for a system call when switching domains. As Sec. 4 shows, 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 Hodor-VMFUNC. Line 2 saves the stack pointer of the source domain to the source domain's status page. As Sec. 2 describes, this step allows the trampoline to restore the stack when returning from the protected library; it also supports callback functions (argument copying and register scrubbing code is omitted for brevity). 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 14 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 19); it then resets the extended page table to the source domain (lines 25–27). Finally, it loads the source domain stack pointer from the source's status page into `rsp` (line 32) and resumes execution (line 37).

Listing 1: Hodor Trampoline: VMFUNC (left), 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 $TGT_IDX, %ecx             |  movl $TGT_PERM, %eax
9  vmfunc                          |  wrpkru
10                                 |  cmpl $TGT_PERM, %eax
11                                 |  jne 1b        ; error
12
13 ; Switch to target domain stack
14 movq target_stack, %rsp
15
16 ; target_domain_func()
17
18 ; Save target domain stack pointer
19 movq %rsp, target_stack
20
21 ; Disable target domain view &
22 ; Enable source domain view
23                                 | 2:
24                                 |  xorl %ecx,%ecx
25 xorl %eax, %eax                 |  xorl %edx,%edx
26 movl $SRC_IDX, %ecx             |  movl $SRC_PERM, %eax
27 vmfunc                          |  wrpkru
28                                 |  cmpl $SRC_PERM, %eax
29                                 |  jne 2b        ; error
30
31 ; Switch back to source domain stack
32 movq source_stack, %rsp
33 jmp BOUNDARY_TRAMP
34
35 ; Boundary Trampoline
36 BOUNDARY_TRAMP:
37 ret
```

As a starting code base, our Hodor-VMFUNC implementation uses the Dune system of Belay et al. [3], with the application running in ring 3 of VMX non-root (VM guest) mode. 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. (That said, system calls are uncommon except during initialization in applications that use kernel-bypass data-plane libraries.) TLB refill costs increase as well, due to 2-level translation.

Ideally, we should like 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 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 not see additional overhead for system calls.

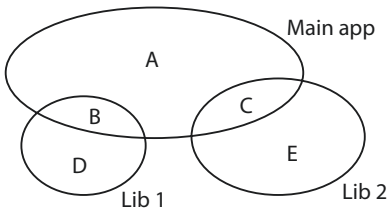

Figure 2: Address space regions in Hodor-PKU.

3.3 Hodor-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.

PKU [20] 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 serialize the pipeline, and can execute very quickly.

Hodor-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.

Listing 1 (including the parts to the right of the vertical lines) shows the trampoline code for Hodor-PKU. Lines 6 and 7 set `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 28) 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 19), the trampoline resets the `pkru` register to the restrictions of the source domain (lines 24–27), and returns as in Hodor-VMFUNC.

3.3.1 Safety of Hodor-PKU

Since the processor allows user-mode code to execute the `WRPKRU` instruction, we must prevent a malicious application from using the instruction to attain access to a protected library's memory. One could think of employing static binary rewriting [50] to replace implicit occurrences of `WRPKRU` with equivalent alternatives. Unfortunately, such rewriting (including definitive determination of instruction alignment) is undecidable in the general case [41, 54], and seems inapplicable to any program that mixes read-only data into the text segment. *Dynamic* binary rewriting [6, 33] might be a viable alternative, but would likely incur prohibitive overhead (up to 2.5× for Intel Pin and 5× for DynamoRIO [33]). To address the problem, Hodor-PKU uses a trusted loader to identify all text-segment occurrences of the `WRPKRU` opcode outside of trampolines, and uses *hardware watchpoints* (debug registers [20]) to vet their execution at run time.

Binary Inspection The `WRPKRU` instruction can occur explicitly (intended by the programmer) or implicitly (unintended occurrence), as a sequence of bytes within an instruction or across the boundary between instructions. Implicit instances pose a significant threat: an adversary that seeks to bypass Hodor-PKU may attempt to corrupt control data and jump to a point in the program that happens to encode the `WRPKRU` instruction. By setting the contents of `ecx`, `edx`, and `eax` appropriately before subverting execution, the attacker could set the `pkru` register to any desired value, rendering the isolation useless. To address this issue, the trusted loader scans the application code and makes a list of any untrusted instances, explicit or implicit. It passes this list to the kernel, which in turn places the addresses of the potentially problematic opcodes in the debug registers. A hardware watchpoint will be triggered when any of these instructions is about to be executed, allowing the kernel to vet the instruction and let the execution proceed only if deemed safe.

Our current implementation inspects program text whenever a library is loaded and whenever execute permission for a page is enabled during execution. Once a page is marked as executable, Hodor-PKU prevents further write accesses to the page. Hodor-PKU could easily be extended to support JIT compilation by marking the faulting pages as writable but not executable, allowing JIT code to be emitted. On future attempts to execute the added code, a page fault would occur, and Hodor-PKU would reinspect the page and continue as in the current implementation.

Runtime Vetting Since the debug registers are limited in number (four—`DR0` through `DR3`—on Intel processors [20]), we can rely on hardware to vet only a handful of `WRPKRU` instances at a time on each thread. Hodor therefore uses hardware watchpoints as an *LRU cache* for all the required watchpoints. Specifically, Hodor initially marks all executable pages containing `WRPKRU` instances as non-executable. Upon first execution, resulting in a page fault, Hodor reclaims a sufficient number of hardware watchpoints, marks the pages they formerly watched as non-executable, and uses the debug registers for the new page. If all `WRPKRU` instances in the page are monitored by a hardware watchpoint, Hodor marks the page as executable. In the extremely rare case of more `WRPKRU` instructions in a single page than the number of debug registers, we resort to single-step execution [20] for that page. We use per-thread page tables (only the root page must be unique for each thread; most lower-level pages can be shared between threads) so that the set of hardware watchpoints can be different in different threads. When watchpoints have been inserted at all appropriate locations, we rewire the page tables leading to the page containing the watchpoint for the current thread and mark it as executable.

Protection Overhead Under normal circumstances, no implicit `WRPKRU` will be executed. Moreover, the processor triggers a watchpoint only when a debug register points to the first byte of the executed instruction [20], so spurious traps will never occur when correctly aligned execution runs past an implicit instance.

Experiments confirm that there is no measurable overhead for this approach as long as the number of "hot" watchpoints in each thread is smaller than the number of hardware watchpoints. To assess how often this might occur, we inspected all packages in the Fedora 29 distribution for occurrences of `WRPKRU`. Across 58,273 `rpm` packages, containing about 108K executable binaries, we found only 111 binaries with a single instance of `WRPKRU`, 8 with two, 2 with three, none with four, and only 2 (less than 0.02%) with five or more. Most of the occurrences were implicit—typically caused by an instruction with a byte pattern ending in `0f` followed by `add %ebp,%edi`, which has a byte pattern of `01 ef`. These occurrences could easily be eliminated by modifying the compiler to insert a `nop` before the culprit add instructions. While such a change would not guarantee that implicit instances never occur (due to inline assembly and code generated at run time), it would almost certainly eliminate any practical performance impact.

4 Evaluation

We have evaluated Hodor using microbenchmarks and three real-world applications in which we isolated a high-throughput data-plane library or in-memory database from the rest of the application. We also constructed two proof-of-concept demonstrations of safe memory sharing among instances of a protected library in otherwise distrusting applications. We ran the 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. We ran the network experiments 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 were connected back-to-back through dual-port Mellanox ConnectX3-Pro 40 Gbps Host Channel Adapters (HCAs) to isolate their connection. All servers ran Fedora Linux 4.15 with our modifications (except for baseline experiments, which used an unmodified kernel). All machines had hyper-threading and Turbo Boost enabled.

We emulated the overhead of PKU on Haswell machines in a manner similar to previous work [28, 50]. We verified the overhead of the 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 Hodor—kernel bypass with no intra-process isolation.
- `ptsw`: isolation via syscall-initiated page table switching, as described in Sec. 3.1.
- `ptsw-pti`: same as `ptsw`, except with kernel page-table isolation enabled.
- `vmfunc`: Hodor-VMFUNC, as described in Sec. 3.2.
- `pku`: Hodor-PKU, as described in Sec. 3.3.

Unless otherwise noted (shown in legends with `-pti`), experiments were conducted with kernel page-table isolation [13] disabled. 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. The source code for Hodor is available at http://github.com/hedayati/hodor.

4.1 Microbenchmarks

We used microbenchmarks to measure the overhead of relevant instructions and basic operations as well as the latency of different implementations of Hodor on the Skylake machine, which supports PKU. We also implemented a no-op system call and a no-op VM call and measure their latencies. We used `rdtscp` with proper serialization [38] 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 Hodor; the latency of `VMFUNC` and `WRPKRU` impacts Hodor-VMFUNC and Hodor-PKU, respectively. 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 with virtualization, as used in Hodor-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) [31] 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`.

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

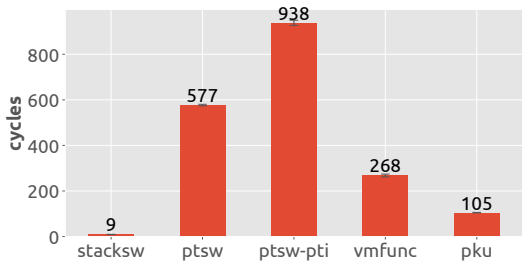

Figure 3: Transition Microbenchmarks.

Figure 3 compares the transition time from one domain to another and back again using different isolation implementations. Additionally, we measured the cost of switching stacks without providing isolation as it contributes a small amount to all implementations of Hodor. 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 enabled; Figure 3 denotes this as `ptsw-pti`. KPTI has no impact on Hodor-VMFUNC and Hodor-PKU.

Among the implementations of isolation, Hodor-PKU has the lowest transition cost, followed by Hodor-VMFUNC. This matches the results in Table 1: changing the `pkru` register costs much less than using `vmfunc`. System calls dominate the cost of the implementations based on `ptsw`. Relative to `ptsw`, kernel page table isolation in `ptsw-pti` incurs a penalty of 62%. Stack switching itself has an almost negligible impact. As noted in Sec. 3.3, there is no measurable overhead to using debug registers to vet instances of `WRPKRU`, so long as there are no more than four watchpoints in each thread.

4.2 Silo

Silo [48] is a scalable in-memory database. It 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 [10] or TPC-C [47]) in a loop. We configured the main Silo library as a separate domain whose pages are protected from the benchmark driver. Even in the context of a single

application, Hodor ensures that the database can be accessed only by library code—never, for example, as the result of a memory access bug in the main application. This protection may be helpful even in the course of a single execution. If the database were kept in nonvolatile memory and retained across program runs, it might be considered essential. The benchmark calls (trampolines of) library routines to perform one domain transition per transaction. All data and metadata reside in memory, and the workload is CPU intensive.

Figure 4 (i) shows the overhead of isolation for the YCSB [10] and TPC-C [47] 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 [10] is a key-value benchmark with tiny transactions. We first filled the database with 1 million records and then ran a workload with an 80/20 read/write mix. The unmodified Silo reaches 2.27 million transactions per second on each core. Hodor incurs 44%, 54%, 27%, and 9.85% overhead in the PT-Switch, PT-Switch with KPTI, VMFUNC, and PKU implementations, respectively.

TPC-C [47] is a relational database benchmark with significantly larger transactions [10]. As a result, the maximum number of transactions per second is reduced to around 600,000 per core on unmodified Silo. With a lower rate of library transitions, the overhead of Hodor drops to 3%, 4.66%, 13.6%, and 1.5% for the PT-Switch, PT-Switch with KPTI, VMFUNC, and PKU implementations, respectively. While Hodor-VMFUNC incurs the largest overhead in this experiment, we discovered that 12% of that overhead is due to running inside a VM—apparently due to frequent use of the `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) [21] 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. We evaluate Hodor 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 packets from one port of the HCA and forwards them directly to another port without accessing packet contents. We connected two hosts back-to-back for endless forwarding of packets in an isolated network. We used Hodor to separate the packet-forwarding logic from the DPDK library.

Figure 4 (ii) shows the effect of Hodor 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 Hodor overhead, we configured 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 configuration can forward more than 720,000 packets per second, and the overhead of Hodor is less than 25% with VMFUNC and 7% with PKU. As we increase the batch size (Fig. 4 (ii-a) vs. (ii-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 CPUs ((b) vs. (c)), the performances of all approaches improve 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 [42] 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 [46] on top of the DPDK packet processing framework and driver to provide connections to Redis clients. We use Hodor 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 [10] on a remote client to benchmark the server configuration. Both the YCSB client and the Redis server are running on the Haswell 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 impact of Hodor, we first loaded the Redis server with 1 million records each of length 1200 bytes. We then ran a YCSB workload [10] with a 95%/5% read-/write mix and measured how many transactions per second the Redis server supported. Figure 4 (iii) shows the results as measured and reported by the YCSB client. The unmodified server can support 220,000 transactions per second. The PT-Switch, PT-Switch with KPTI, VMFUNC, and PKU implementations of Hodor reduce the throughput of Redis by 12%, 35%, 5%, and 2.78%, respectively.

4.5 Discussion

The preceding subsections reveal significant performance differences among the three implementations of Hodor: syscall-based page table switching, Hodor-VMFUNC, and Hodor-

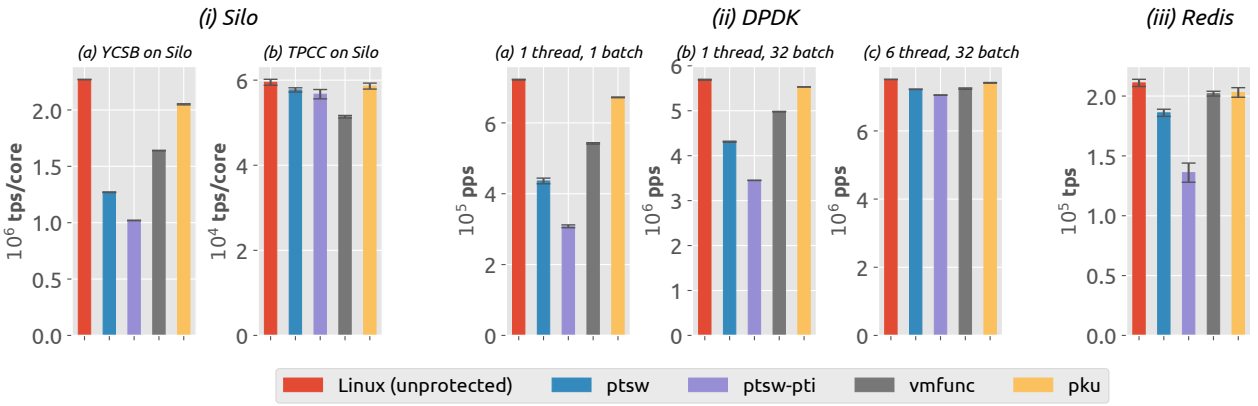

Figure 4: Hodor Overhead: (i) Silo Benchmarks, (ii) DPDK Raw Packet Forwarding Benchmarks, and (iii) Redis Benchmark.

PKU. These differences must be considered together with the issues of generality and confidentiality when applying Hodor in a particular environment.

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

One limitation of Hodor-PKU is that current Intel hardware 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 mutually distrusting protected libraries in any given application—fewer if they wish to make direct calls to one another. Hodor-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 Hodor-VMFUNC. As discussed in Sec. 3.2, VMFUNC is only available in a virtualized mode. Hodor-VMFUNC, like MemSentry [28], uses Dune [3] to run applications inside a virtual machine. This restriction imposes a significant cost on system calls in the application, which now incur the latency of a VM exit; we see the impact of this latency in Figure 4 (i-b). While we don't expect frequent system calls in a data-plane library, an alternative design [32, 35] avoids VM exits on system calls by running the kernel, in addition to user programs, inside the virtual machine. Such a design has a system-wide impact on performance, since the entire software stack is virtualized, not just the intended application. Ideally, we should 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.

Absent direct access, a malicious program may attempt to steal information from protected libraries through a side channel. While such attacks are out of scope, we note that Hodor-PKU is inherently vulnerable to Meltdown-PK [8], an attack that defeats the purpose of PKU itself: the protection key bits are part of the TLB access permissions, which the processor may check late in the pipeline [8, 26]. While all Skylake processors are susceptible to this attack, the vulnerability has been fixed in more recent microarchitectures [18].

4.6 Cross-Application Sharing

Because they are protected from their calling applications, protected libraries in Hodor can, at least in principle, safely share state among library instances in separate applications. As proof-of-concept demonstrations, we have used Hodor to implement sharing in Silo and resource management in DPDK. The implementations highlight general issues that must be considered by the library designer.

Sharing in Silo: We wrote a library that uses Silo [48] internally to implement a TPCC [47]-like database. Our library provides two different views of the same database: one that can do `NewOrder`, `Payment`, and `OrderStatus` transactions, and another that can do `Delivery` and `StockLevel` transactions. Both interfaces use the same set of tables, which Silo maintains in physically shared pages. Silo guarantees consistency of the database and serializability of the transactions, while Hodor guarantees that the only way for an application to modify the database is to use the provided interface. When sharing the database between two separate applications, our library can control which interface is available to each application. More significantly, by preventing access of any kind when running outside the library, Hodor can ensure that stray memory references in a buggy application (e.g., due to out-of-bound array indexing or uninitialized pointer dereference) never compromise database invariants.

Resource management in shared DPDK: The DPDK Environment Abstraction Layer (EAL) [21] has recently added multi-process support so that mutually trusting processes can share DPDK huge-pages, memory buffers, and queues. A group of DPDK processes can then work together in a simple

transparent manner to perform packet processing or similar tasks. Using Hodor, we extended this mechanism to allow *distrusting* processes to share a single NIC. We wrote a simple library that exports several DPDK APIs (`rte_eth_rx_burst`, `rte_eth_tx_burst`, etc.). Internally, it uses shared memory to record the rate at which each application sends packets, to implement proportional share. We link this library, via Hodor, into two distrusting applications. The protected library in each application then measures its own traffic, updates shared statistics (under control of appropriate synchronization), and periodically adjusts the rate that will be allowed in the next time period.

While we were able to port the two libraries to use Hodor for sharing in just a few hundred lines of code, the experience highlighted several issues that need to be considered when tying together library instances in separate applications. As a rule of thumb, developers should think of protected libraries as extensions to the operating system. Safety-critical arguments passed by applications should be copied into library space before applying sanity checks (to avoid modification by other application threads). Libraries should also treat shared regions as potential attack vectors and should employ conventional defenses (e.g. retpolines to mitigate Spectre-type attacks [8] when relevant). Significantly, Hodor does not prevent a buggy application from invoking library routines in the "wrong" sequence, or with the "wrong" arguments. It does, however, prevent an application from undermining any invariant that is carefully maintained by those routines. It is the responsibility of the protected library to provide appropriate synchronization, scalability, and fault tolerance. The latter may be simplified by using nonblocking data structures, or by depending on Hodor to execute through to the end of library routines in the event of process failure.

5 Related Work

Hodor connects to three areas of related work: fast I/O systems that move device and resource management into user space, methods to isolate software components sharing the same virtual address space, and systems that impose security policies on operating system kernels and hypervisors.

5.1 Fast I/O Systems

Existing kernel-bypass systems do not protect libraries from untrusted applications. Arrakis [39] uses a library OS without isolation in the same address space as the application and relies on device-level SR-IOV [23] support. Device-level resource isolation policies are often rigid—e.g., limited to simple partitioning. Hodor protected libraries enable more powerful protection policies like proportional bandwidth sharing and even safe, concurrent accesses to the same data. IX [4] and ZygOS [40], both of which build on Dune [3], use virtualization to run their kernel-bypass stack in ring 0 of VMX non-root mode. While this design already isolates networking logic from the applications, it is limited to only a single trusted domain and does not support multiple distrusting data-plane libraries within the same application, as Hodor does.

Kernel-based high-throughput software stacks like MegaPipe [14] and StackMap [57] depend on aggressive batching to limit the frequency and cost of protection domain switching. Aggressive I/O batching, however, requires asynchronous programming models that are generally hard to employ and not always supported by library APIs. In Sec. 4, for example, we were unable to batch over the Silo database API [48] or F-Stack's send calls [46].

5.2 Intra-Process Isolation

There has been much previous work on intra-process isolation. The method with least overhead is to write code in a type-safe language. Work in single address space operating systems such as Singularity [17] and Verve [56] shows that application and kernel code can execute safely within the same virtual address space. The disadvantage of such systems is their incompatibility with much existing code. Hodor, in contrast, supports existing fast I/O applications.

For type-unsafe languages, approaches such as SFI [52] and XFI [49] 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 is within bounds or transforms out-of-bounds pointers to in-bounds pointers. SFI [52] incurs an average overhead of 17.6% for read-write protection and 4.3% overhead when only instrumenting writes. Hodor 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 [51] and CHERI [55] augment instructions with capabilities. Segmentation also provides intra-process isolation [43] 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 OS kernel can isolate untrusted components. Segmentation is supported in 32-bit but not 64-bit x86 systems [20]. ARM memory domains [2] are similar to Intel PKU [20] but available only on 32-bit processors, and memory domain permissions can be modified only in supervisor mode. Our work focuses on hardware support available in 64-bit x86 systems.

ERIM [50], developed concurrently to our work, uses protection keys like Hodor to provide an isolated domain within a single virtual address space. We believe ERIM's use of static binary rewriting to eliminate occurrences of `WRPKRU` in the application binary is insufficient to guarantee the safety of protected domains: static binary rewriting is undecidable for arbitrary x86 code [41, 54]. *Dynamic* binary rewriting (not considered in ERIM) would incur prohibitive costs, negating the performance gain of PKU.

Several OS abstractions are similar to our work. Wedge [5]

provides privilege separation and isolation among its sthreads. Each sthread is a lightweight process that inherits only a subset of the memory mappings and file descriptors of its parent, as specified in a security policy. Shreds [9] use ARM memory domains [2] to divide execution within a user-space thread. Each shred is a thread fragment with a private memory pool in which to store secret data and sensitive code. Light-weight contexts (lwCs) [31] 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. Each of these systems requires a system call to change domains, while Hodor does not. Hodor can also be linked to unmodified applications.

MemSentry [28] is a memory isolation framework that provides compiler support for multiple hardware features, including EPT-switching VMFUNC and PKU, to create a safe region within a process. It analyzes and instruments applications with code which, like Hodor's trampolines, enables and disables access to the protected domain using the desired isolation mechanism. SeCage [32] 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 and MemSentry, Hodor relies on existing explicit library boundaries, alleviating the need for compiler analysis to extract components. SeCage places the entire OS and its applications in a virtual machine, while Hodor-VMFUNC and MemSentry leverage Dune's [3] process-level virtualization to expose the VMFUNC EPT-switching mechanism to individual applications. Executing the intended application in non-root mode affects the performance of that application only. SeCage [32] compensates for the system-wide performance impact of the virtualization layer with its additional protections against a malicious OS.

SkyBridge [35] uses VMFUNC to improve the latency of IPCs in a micro-kernel setting. Unlike Hodor, SkyBridge does not enforce a single way to cross protection boundaries (Hodor ensures that *only* trampolines are mapped in both source and target EPTs), which introduces the possibility for malicious VMFUNCs. To address this, SkyBridge uses static binary rewriting (inspired by ERIM [50]), which as discussed earlier, is undecidable for an arbitrary x86 binary [41, 54]. Finally, EPTI [16] uses VMFUNC to provide isolation between kernel and user-space page tables to mitigate Meltdown [8].

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 [36]. CrossOver [29] proposes a cross-world interaction mechanism that provides communication 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 OS and Hypervisor Security

Hodor builds on previous work on security enforcement in OS kernels and hypervisors. The design of the Hodor-PKU trampoline is inspired by the Nested Kernel [12] trampoline code. Both Hodor-PKU and Nested Kernel must check that the inputs to domain switching instructions are correct because neither system enforces control flow integrity [1]. Finally, Hodor's restrictions on `mmap` to enforce code segment integrity are similar to protections in Secure Virtual Architecture [11], HyperSafe [53], and Nested Kernel [12].

6 Conclusions

We have introduced Hodor, an in-process isolation system for protection and sharing of fast data-plane libraries. Our proposed solution uses Intel's memory protection keys (PKU) to isolate components within a single address space. We also presented two alternative implementations based on separate user-level address spaces—one uses system calls for page-table switching, the other Intel's VMFUNC switching of extended page tables. Additionally, Hodor uses asynchronous event delivery and a novel application of hardware watchpoints to ensure that when multiple processes share a protected library, failure in one will not affect the others.

Our evaluation with microbenchmarks, Silo, DPDK, and Redis confirm that Hodor can provide full isolation of protected libraries while approaching unprotected kernel bypass performance. Hodor-PKU, in particular, provides 90–98% of kernel-bypass throughput in all of our experiments.

Hodor could benefit from a VMFUNC-like instruction that switches 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 Hodor implementation based on software fault isolation [52] and to explore hardware-supported implementations for additional processor architectures (e.g., ARM and Power).

Acknowledgment

We thank our shepherd, Adam Belay, and the anonymous reviewers for their helpful feedback. 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. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow Integrity Principles, Implementations, and Applications. *ACM Trans. on Information Systems Security*, 13:4:1–4:40, Nov. 2009.

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

[3] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. 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, Oct. 2012.

[4] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. 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, Oct. 2014.

[5] A. Bittau, P. Marchenko, M. Handley, and B. Karp. Wedge: Splitting Applications into Reduced-privilege Compartments. In *5th USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, pages 309–322, San Francisco, CA, Apr. 2008.

[6] D. L. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 2004. AAI0807735.

[7] N. Burow, X. Zhang, and M. Payer. Shining Light On Shadow Stacks. *arXiv e-prints*, abs/1811.03165, Nov. 2018.

[8] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. *arXiv e-prints*, abs/1811.05441, Nov. 2018.

[9] 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.

[10] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *1st ACM Symp. on Cloud Computing (SoCC)*, pages 143–154, Indianapolis, IN, June 2010.

[11] J. Criswell, N. Geoffray, and V. Adve. Memory Safety for Low-level Software/Hardware Interactions. In *18th USENIX Security Symp. (SEC)*, pages 83–100, Montreal, PQ, Canada, Aug. 2009.

[12] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. 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, Mar. 2015.

[13] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard. KASLR is Dead: Long Live KASLR. In *Intl. Symp. on Engineering Secure Software and Systems (ESSoS)*, pages 161–176, Bonn, Germany, July 2017.

[14] S. Han, S. Marshall, B.-G. Chun, and S. 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, Oct. 2012.

[15] T. C.-H. Hsu, K. Hoffman, P. Eugster, and M. Payer. Enforcing Least Privilege Memory Views for Multithreaded Applications. In *ACM SIGSAC Conf. on Computer and Communications Security (CCS)*, pages 393–405, Vienna, Austria, Oct. 2016.

[16] Z. Hua, D. Du, Y. Xia, H. Chen, and B. Zang. EPTI: Efficient defence against meltdown attack for unpatched vms. In *USENIX Annual Technical Conf. (ATC)*, pages 255–266, Boston, MA, 2018. USENIX Association.

[17] G. C. Hunt, J. R. Larus, M. Abadi, M. Aiken, P. Barham, M. Fähndrich, C. H. O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An Overview of the Singularity Project. Technical Report MSR-TR-2005-135, Microsoft Research, Oct. 2005.

[18] Intel Corp. Engineering New Protections Into Hardware. http://www.intel.com/content/www/us/en/architecture-and-technology/engineering-new-protections-into-hardware.html.

[19] 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.

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

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

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

[23] 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.

[24] E. Y. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. 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, Apr. 2014.

[25] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark,

J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. 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.

[26] V. Kiriansky and C. Waldspurger. Speculative Buffer Overflows: Attacks and Defenses. *ArXiv e-prints*, abs/1807.03757, July 2018.

[27] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symp. on Security and Privacy (SP)*, May 2019.

[28] K. Koning, X. Chen, H. Bos, C. Giuffrida, and E. 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, Apr. 2017.

[29] W. Li, Y. Xia, H. Chen, B. Zang, and H. 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.

[30] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symp. (SEC)*, pages 973–990, Baltimore, MD, Aug. 2018.

[31] J. Litton, A. Vahldiek-Oberwagner, E. Elnikety, D. Garg, B. Bhattacharjee, and P. 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, Nov. 2016.

[32] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. 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, Oct. 2015.

[33] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proc. of the 2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 190–200, Chicago, IL, June 2005.

[34] K. Menychtas, K. Shen, and M. 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, Mar. 2014.

[35] Z. Mi, D. Li, Z. Yang, X. Wang, and H. Chen. SkyBridge: Fast and Secure Inter-process Communication for Microkernels. In *14th EuroSys Conf. (EuroSys)*, pages 9:1–9:15, Dresden, Germany, Mar. 2019.

[36] J. Nakajima. Xen as High-performance NFV Platform, Aug. 2018. http://events.static.linuxfound.org/sites/events/files/slides/XenAsHighPerformanceNFVPlatform.pdf.

[37] F. Nawab, J. Izraelevitz, T. Kelly, C. B. Morrey III, D. R. Chakrabarti, and M. L. Scott. Dalí: A Periodically Persistent Hash Map. In *31st Intl. Symp. on Distributed Computing (DISC)*, pages 37:1–37:16, Vienna, Austria, Sept. 2017.

[38] G. 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, Sept. 2010.

[39] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. 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, Oct. 2014.

[40] G. Prekas, M. Kogias, and E. Bugnion. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *26th Symp. on Operating Systems Principles (SOSP)*, pages 325–341, Shanghai, China, Oct. 2017.

[41] G. Ramalingam. The Undecidability of Aliasing. *ACM Trans. on Programming Languages and Systems (TOPLAS)*, 16(5):1467–1471, Sept. 1994.

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

[43] J. H. Saltzer and M. D. Schroeder. The Protection of Information in Computer Systems. *Proc. of the IEEE*, 63(9):1278–1308, Sept. 1975.

[44] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen. Adapting Software Fault Isolation to Contemporary CPU Architectures. In

19th USENIX Security Symp. (SEC), pages 1:1–1:11, Washington, DC, Aug. 2010.

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

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

[47] Transaction Processing Council. TPC-C Benchmark, 2018. http://www.tpc.org/tpcc.

[48] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy Transactions in Multicore In-memory Databases. In *24th ACM Symp. on Operating Systems Principles (SOSP)*, pages 18–32, Farmington, PA, Nov. 2013.

[49] Úlfar Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. 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, Nov. 2006.

[50] A. Vahldiek-Oberwagner, E. Elnikety, D. Garg, and P. Druschel. ERIM: Secure and Efficient In-process Isolation with Memory Protection Keys. *arXiv e-prints*, abs/1801.06822, Nov. 2018.

[51] L. Vilanova, M. Ben-Yehuda, N. Navarro, Y. Etsion, and M. Valero. CODOMs: Protecting Software with Code-centric Memory Domains. In *41st Intl. Symp. on Computer Architecuture (ISCA)*, pages 469–480, Minneapolis, MN, June 2014.

[52] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient Software-based Fault Isolation. In *14th ACM Symp. on Operating Systems Principles (SOSP)*, pages 203–216, Asheville, NC, Dec. 1993.

[53] Z. Wang and X. 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.

[54] R. Wartell, Y. Zhou, K. W. Hamlen, M. Kantarcioglu, and B. Thuraisingham. Differentiating Code from Data in x86 Binaries. In *2011 European Conf. on Machine Learning and Knowledge Discovery in Databases (ECML PKDD)*, pages III 522–536, Athens, Greece, Sept. 2011.

[55] 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.

[56] J. Yang and C. 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.

[57] K. Yasukata, M. Honda, D. Santry, and L. 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.