The Opportunities and Limitations of Extended Page Table Switching for Fine-Grained Isolation

Vikram Narayanan and Anton Burtsev | University of Utah

Extended Page Table switching with VMFUNC is a hardware isolation mechanism available in Intel CPUs. VMFUNC is attractive for low overhead and the possibility to isolate privileged kernel code. However, many careful design decisions are needed to ensure the security of the isolation boundary.

A steady increase in the number of security attacks (combined with a growing level of attack complexity and automation) triggered a renewed interest in hardware support for isolation. After decades of relatively slow adoption, recent generations of commodity CPUs introduced a range of new hardware isolation mechanisms. Intel Memory Protection Keys (MPKs) and Extended Page Table (EPT) switching with virtual machine (VM) functions (VMFUNC) develop support for memory isolation with overheads gradually approaching the overhead of a function call.[1,2,3,4]

Both Intel and ARM architectures explore hardware support for software fault isolation (SFI). The latest ARM CPUs introduce 16-B-granularity memory isolation with memory tagging extensions (MTEs). MTE can potentially enable low-overhead bounds checks and zero-copy exchange of data between isolated subsystems. Moreover, both ARM and x86 provide support for control-flow integrity (CFI) and stack protection.

While making an appealing promise to enable practical fine-grained isolation, the aforementioned mechanisms require a collection of careful design decisions to enforce the security of the isolation boundary. For example, isolation with MPKs relies on either the binary rewriting of all `wrpkru` instruction instances[1] or dynamic validation with hardware breakpoints[2] (the `wrpkru` instruction updates a register that holds the current tag and hence controls the boundaries of accessible memory).

Understanding the design choices, security requirements, and limitations of modern hardware isolation mechanisms is critical for deploying efficient and secure isolation solutions. Similarly, it is essential for providing informed feedback to hardware engineers responsible for the next generation of hardware isolation primitives in commodity CPUs.

This article takes a deep dive into the design challenges of implementing a secure isolation boundary on top of one of the modern hardware isolation mechanisms: EPT switching with VMFUNC. We base our analysis on our experience of implementing the fine-grained isolation of Linux kernel modules with EPT switching[4] as well as on an exploration of several other VMFUNC-based systems.[3,5,6]

Digital Object Identifier 10.1109/MSEC.2023.3251385
Date of current version: 26 April 2023

We first survey recent hardware isolation mechanisms, trying to highlight their relative advantages and limitations, and then provide a detailed discussion of principles and mechanisms involved in implementing a VMFUNC-based isolation boundary. Like other hardware isolation mechanisms, VMFUNC provides a unique mix of advantages and limitations.

A distinct advantage of VMFUNC is its ability to ensure the security of the isolation boundary (and specifically, the safety of `vmfunc` instructions) through a collection of invariants that control the layout of virtual and physical address spaces as opposed to the binary rewriting and techniques of SFI required to protect other isolation mechanisms, for example, Intel MPK, ARM MTE, and ARM pointer authentication (PAC). On the other hand, VMFUNC requires the execution of a system under the control of the hypervisor that can be restrictive due to the need to support nested virtualization in a virtualized data center. To further highlight the unique properties of VMFUNC-based solutions, we explore how EPT switching can be used for the isolation of a privileged ring 0 kernel code. By executing the system under the control of a hypervisor, a VMFUNC-based solution can control access to a sensitive hardware state at the hypervisor level [that is, control whether an isolated subsystem can access privileged control and model-specific registers (MSRs), input–output (I/O) ports, etc.]. We analyze additional mechanisms that are needed for isolating code that executes with ring 0 privileged, that is, ensuring the safety of cross-subsystem invocations, providing the safe and efficient handling of interrupts, etc.

The Landscape of Modern Hardware Isolation Primitives

In the last decade, commodity CPUs introduced a diverse range of hardware primitives aimed at the support of practical fine-grained memory isolation. While clearly a leap forward, existing hardware mechanisms rely on a complex combination of software techniques required to ensure the integrity of the isolation boundary. No matter which mechanism is used, isolation depends on complex design, engineering, and performance tradeoffs.

Intel MPK

MPKs are an isolation mechanism available on Intel CPUs since Skylake. MPK allows one to enforce isolation within a single address space (Figure 1), that is, a single page table, by tagging individual pages with a 4-b protection key (saved in the unused bits of the page table entry). A special register, `pkru`, holds a bitmap that allows access to a combination of tags (that is, any combination from none to all is possible by setting individual bits in the bitmap). The `pkru` register specifies the access rights for each protection key with 2 b per key (access disable and write disable). Read or write access to a page is allowed only if the value of the `pkru` register matches the tag of the page. Crossing between protection domains is performed by writing a new tag value into the `pkru` register with an unprivileged `wrpkru` instruction.

Limitations. Isolation with MPK requires control over all `wrpkru` instructions throughout the code of the program to prevent unauthorized transitions between address spaces. In the past, control over `wrpkru` was demonstrated with either binary rewriting[1] or dynamic validation of all `wrpkru` instructions with hardware breakpoints.[2] Also, MPK enforces checks only on data accesses but does not limit control-flow transitions.

Moreover, similar to other tag-based solutions (for example, ARM MTE), MPK is limited to only 16 protection domains. *libmpk* provides a software abstraction for overcoming this limitation by virtualizing the keys, but it is expensive as the application programming interface involves several syscalls.[7] Extended protection keys virtualize the number of keys by combining MPK with EPT switching[8] but at the cost of the additional complexity and overheads of the VMFUNC-based approaches that we discuss later.

ARM MTE

Starting with ARMv8.3-A, ARM systems-on-chip (SoCs) introduce support for MTEs that allow

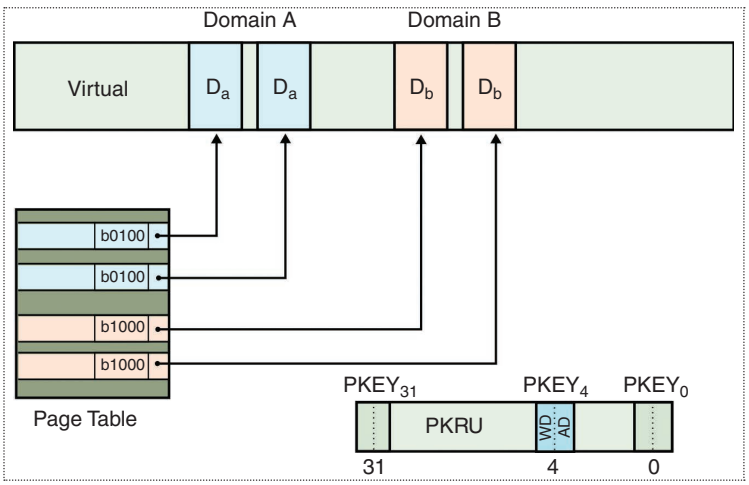

Figure 1. Domain isolation using MPK. Page table entries store a 4-b tag. The `pkru` register controls that tags can be accessed at the moment (that is, in the figure read and write access, to pages with tag 4 is allowed). PKEY: protection key; PKRU: protection-key rights register for user pages; WD: write disable; AD: access disable.

partitioning the address space into 16-B regions that are colored with one of the 16 tags. The hardware maintains a table that stores mapping between addresses and tags, allowing access to the region only if the tag of the pointer (the tag stored in the upper bits of the pointer) matches the tag of the memory region. MTE itself does not directly support isolation—the attacker can change the upper bits of the pointer that contain the tag. To enforce isolation, it is possible to combine MTE with the techniques of SFI, that is, rely on binary rewriting or compile-time instrumentation to enforce a specific tag on every load and store operation.

Limitations. Two overheads impact the performance of MTE-based isolation solutions. First, the enforcement of hardware tags relies on SFI techniques, which, in turn, require the control of not only load and store operations (to restrict access to the memory of the isolated subsystem) but also control-flow enforcement. SFI solutions were demonstrated to achieve an overhead of only a few percent but only with one isolated subsystem.[9] The isolation of multiple subsystems and exchange of objects across the isolation boundaries sharply degrade performance.

Second, similar to Intel MPK, ARM MTE is limited to only 16 isolated subsystems due to the space limitation of unused pointer bits. HAKC combines MTE with ARM PAC to extend the number of possible tags[10] but provides no practical protection as it fails to enforce temporal safety (we will discuss this later).

ARM PAC

Starting with ARMv8.3-A, ARM SoCs support cryptographic PAC. PAC implements the ability to cryptographically sign a pointer and store the signature in the "unused" upper bits of the pointer (Figure 2). The signature is generated from 1) the pointer value, 2) a secret key protected by the operating system (OS), and 3) a 64-b program-defined "signing context" that allows the isolation scheme to restrict the use of a pointer in a custom way, for example, allow using the pointer only if the value of the stack pointer (`sp`) is identical at the moment of signing and authenticating the signature. A signed pointer cannot be used directly but instead has to be authenticated with the same secret key and context. If either the pointer, its signature, or the context is different from the values used during signing, the authentication results in an invalid pointer value that triggers a hardware exception when used. However, this leaves room for an attacker to make repeated guesses at the correct PAC value for this address if the authenticated pointer is not immediately used.

ARMv8.6-A introduced faulting PAC (FPAC), which generates an exception when an authentication failure occurs instead of during use. PAC is a powerful mechanism that can be used to enforce control flow, spatial and temporal[11,12] safety, and the isolation of subsystems.[10,13]

Isolation with PAC requires maintaining metadata about each memory object, that is, the size, type, and liveness of an object that are used to enforce the type, memory, temporal safety, and access rights for a currently executing isolated subsystem. Compiler instrumentation is used to generate instructions that check the memory and temporal safety (the bounds, type, and liveness of an object) on each memory access along with permission to access the object. Metadata can be saved next to the object itself[11] or in a separate memory region.[12] HAKC tries to build a PAC isolation scheme that can avoid metadata lookups altogether by combining PAC with ARM MTE. Specifically, HAKC tags each memory object with an MTE tag that allows it to enforce bounds checks.[10] Unfortunately, in an attempt to avoid metadata lookups, HAKC has no mechanism to check the liveness of objects on the heap. It is therefore prone to a simple attack in which an attacker sprays the heap by allocating a large number of objects that it subsequently deallocates while preserving pointers to these objects. Since HAKC does not perform a liveness check, the attacker can later use saved pointers to access objects that are reallocated on the heap.

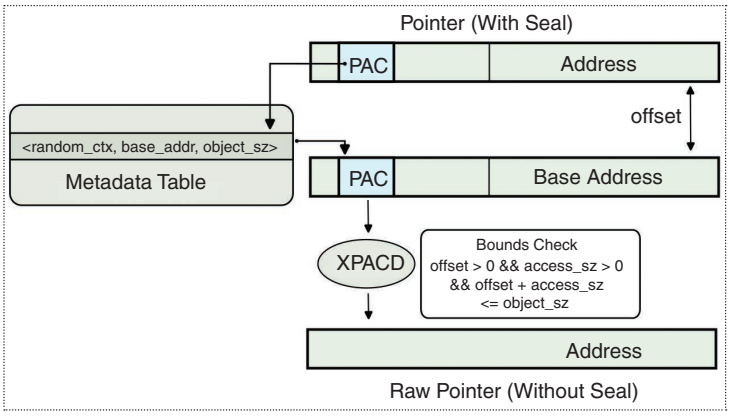

Figure 2. Spatiotemporal memory safety with PAC. PACMem uses the PAC value from a sealed pointer as an index into the metadata table that contains a random context (to unseal the pointer), a base address, and the size of the pointer. If the bounds checks and authentication pass, access through the pointer is allowed. When the object is deallocated, the entry in the metadata table is set to zero.

Limitations. Similar to SFI approaches, PAC suffers from the overheads of executing additional instructions required to look up and validate metadata, check pointer signatures, etc. on each memory access.[10,11,12] For example, the average runtime overhead of PACMem is 68.73%.[12] Additionally, existing PAC-based isolation schemes lack strong security guarantees. For example, PACMem can be

attacked due to the lack of a strong source of randomness as well as a potential of a hash collision.[12]

Intel CET

Intel introduced control-flow enforcement technology (CET), a hardware feature to mitigate return-oriented programming (ROP)-style attacks (ROP, jump-oriented programming, and call-oriented programming) by enforcing coarse-grained CFI. It consists of 1) a shadow stack (`SHSTK`) to protect the return addresses that can be corrupted by buffer-overflow attacks and 2) an indirect branch tracking that protects the forward control flow of the program. `SHSTK` records the return addresses in a hardware-protected stack region along with the regular stack; when the `ret` instruction is executed, the return addresses are compared to generate an exception if there is a mismatch. `SHSTK` provides write-protected pages (using an unused combination of read, write, and dirty bits in the page table) to store return addresses. For error handling, it also provides an instruction to write into the shadow stack (`WRSS`).

Limitations. While by itself CET is not designed to provide the isolation of subsystems, it is possible to utilize a hardware-protected shadow stack memory to provide a memory isolation abstraction.[14] CETIS[14] utilizes the `SHSTK` mechanism and the `WRSS` instruction to protect the data of an isolated subsystem. Though CETIS offers write protection across subsystems, the data in the protected region can be arbitrarily read.

Intel Sub-Page Protection

Intel Sub-Page Protection (SPP) provides a mechanism to control the permissions of a guest physical page at a finer granularity. The EPT page table is complemented with a subpage permission table, a structure similar to a page table, that allows the write accesses to be controlled at a 128-B subpage granularity.

Limitations. SPP controls only write accesses and does not prevent reads. Additionally, similar to VMFUNC, SPP requires that the isolated domains are running in VT-x nonroot context and undergo address translation through EPT. By controlling the layout of data structures shared between the two domains (for example, the kernel and the isolated driver), one could implement SPP protection on top of the existing `VMFUNC` mechanism we described previously to provide read-only and controlled-write accesses to regions of memory in other domains.

Intel VMFUNC

EPT switching with VMFUNC was introduced in the Skylake generation of Intel CPUs. VMFUNC allows a VM guest to change the root of the EPT (Figure 3). The hypervisor configures a table with available EPT root pointers, and the `vmfunc` instruction can select one of the pointers by providing an index. The `vmfunc` is an unprivileged instruction and can be invoked inside the VT-x nonroot context at any privilege level. The `vmfunc` instruction does not change any of the registers (besides advancing the instruction pointer), but the guest physical addresses (GPAs) are translated to host physical addresses (HPAs) through a new EPT. The execution continues with the next instruction, but it is fetched through the new EPT.

The execution of the VMFUNC instruction does not change the value of the `cr3` control register that contains the physical address of the guest page table root. However, switching to a different EPT might change the mapping of GPAs to HPAs and may alter the contents of the guest page table. Since modern CPUs support tagging of the translation lookaside buffer using virtual processor identifiers, the `vmfunc` instruction is relatively fast—the cost of changing the isolation boundary can be as low as 109–147 cycles[2,3,4,15] plus the overhead of performing a cross-domain invocation, that is, saving and restoring general and extended registers, selecting a callee stack, etc. Although Intel patents describe EPT switching as a mechanism to provide the isolation of subsystems within a guest VM, implementing a secure isolation boundary with VMFUNC is challenging and requires multiple careful design decisions. Next, we discuss a collection of principles and mechanisms required to secure an EPT-based isolation boundary.

Challenges of Isolation With EPT Switching

Lightweight EPT switching allows for a conceptually simple isolation approach. Multiple EPTs can map disjoint subsets of machine pages, isolating the address spaces of mistrusting subsystems. To switch between the address spaces, a call-gate page with the `vmfunc` instruction is mapped by a pair of communicating EPTs. Naturally, isolation with EPT relies on the execution of isolated subsystems inside VT-x nonroot context controlled by a hypervisor. This, however, can be transparently achieved for both unprivileged user code (with a minimal hypervisor like Dune[16] that transparently executes user processes as VT-x nonroot contexts) and kernel subsystems (with a late-launch hypervisor that deprivileges the kernel in a manner similar to a rootkit[4]).

Cross-Subsystem Invocations

EPT switching provides a natural mechanism for implementing cross-subsystem invocations without exiting into the OS kernel or hypervisor. During the

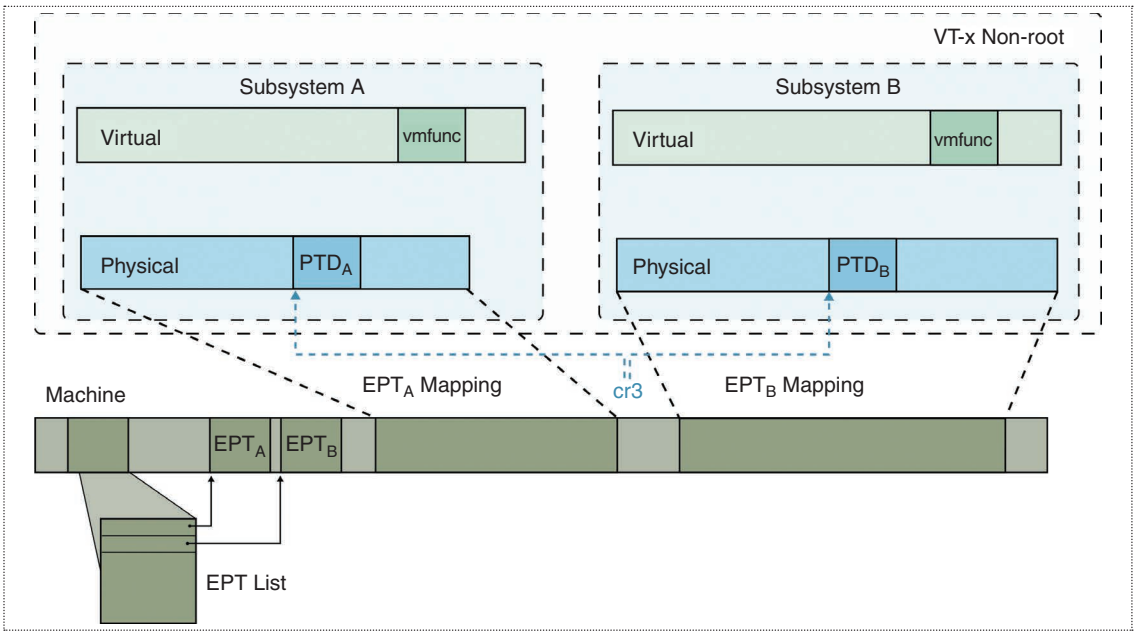

Figure 3. EPT switching with VMFUNC.

invocation, the execution can continue on the same thread (albeit on a new stack) but in a different address space provided by the new EPT mappings. As `vmfunc` advances the program counter to the next instruction, this next instruction must be valid in the address space of the callee. To make sure that execution can continue upon a switch, one can map a special trampoline page in both address spaces at the same virtual address. While the page can contain different code in two address spaces, the callee's entry point must follow the address of the `vmfunc` instruction in the caller's address space.

A typical call-gate trampoline saves the state of the caller on the stack, switches into the address space of the callee with VMFUNC, picks a new stack inside the callee address space, and continues execution by calling a callee dispatch function. Specifically, on the caller side, the trampoline first saves extended registers with the `fxsave` instruction, saves callee saved registers on the stack, and zeroes out all general registers that are not used to pass the arguments and all extended registers. After that, the domain boundary is switched with the `vmfunc` instruction. Inside the callee domain, the trampoline allocates a new stack from a pool of available stacks. If the EPT switching is used inside the kernel, the trampoline saves the values of the segment registers that can be freely changed by the caller and callee in `ring 0`.

Security of the Isolation Boundary

Unlike traditional interrupts and system calls, VMFUNC provides no support for defining an entry point in the callee domain. The next instruction after the `vmfunc` executes with the memory rights of the callee subsystem. The cross-domain invocation mechanism must ensure that the transition is safe, that is, all possible VMFUNC invocations lead to a set of well-defined entry points in the callee (and the caller on the return path) and that both the callee and the caller can securely initialize and restore their state.

Safety of the VMFUNC Instructions

Control-flow attacks inside the isolated subsystem allow an attacker to find executable byte sequences that form valid `vmfunc` instructions. If the virtual address after the VMFUNC instruction is mapped in the address space of another domain, an attacker can escape the isolation boundary. To protect against such an attack, the isolation mechanism should enforce one of the following two invariants (Invs):

Inv 1.a: Virtual address spaces of isolated domains do not overlap.
Inv 1.b: No sequences of executable bytes can form a valid `vmfunc` instruction.

Some solutions, like SkyBridge,[3] rely on scanning and rewriting the executable space of the program to ensure that no byte sequences form valid VMFUNC instructions. In the case of ring 0 isolation, the attack surface for preventing unsafe VMFUNC instructions expands into user applications, which is exceptionally challenging in the face of dynamically loaded and just-in-time

compiled code.[1] Alternatively, Inv 1.a ensures that if an isolated subsystem invokes a self-prepared `vmfunc` instruction anywhere in its address space, the next instruction causes a page fault. The enforcement of Inv 1.a requires control over the virtual address spaces of isolated subsystems.

Secure Saving and Restoring of State

After crossing the isolation boundary, the thread of execution does not trust any of the general-purpose and floating-point registers. In ring 3, segment registers can be trusted (write access to the segment registers has to be disabled in the kernel as well as all OS interfaces that allow updates to segment registers, for example, `arch_prctl` on Linux). If, however, isolated code runs in ring 0, segment registers can be overwritten by untrusted subsystems.

To provide a way of saving and restoring the state of each thread on cross-subsystem invocations, one needs to implement a thread-local store (TLS) that allows saving and restoring state, that is, TLS can store a per-thread pointer to its stack where the thread's state is saved. In ring 3, the TLS can be implemented by using one of the segment registers (`fs` or `gs`). Ring 0 is more challenging. It is possible to implement TLS by relying on the fact that isolated subsystems cannot change their page table hierarchy (see Inv 4 later), which can be updated only by the hypervisor. Specifically, it is possible to utilize two pages mapped at well-known locations inside each subsystem. The first page provides an identifier of the currently executing thread, `thread id`. This page is shared across all isolated subsystems and provides an efficient way to access `thread id`. The part of the trusted computing base (TCB) that performs a context switch updates the current thread identifier, a change that is immediately reflected in all subsystems. The second page, `state page`, is private to each subsystem and allows it to locate TLS regions inside the subsystem. The call-gate code first locates the stack of the current process through its thread identifier and then restores its register state.

cr3 Remapping for EPT Switch

The VMFUNC instruction does not change the root of the page table hierarchy, that is, the value of the `cr3` register. The physical address of the root of the page table should be valid at the same physical address on both sides of the isolation boundary. In many isolation scenarios, it is possible to control the layout of the address space and the address of the page table directory (PTD) inside an isolated subsystem. For example, an in-process isolation solution can rely on a single address space and use the same `cr3` value across all isolated subsystems. In such a case, the isolation subsystem has to enforce the following invariant:

> *Inv 2.a:* The physical address of the root of the guest page table is identical in all isolated domains.

However, in some cases, for example, when isolation is applied to kernel subsystems, the need to provide backward compatibility with the rest of the system prevents one from controlling the physical address of the page table root. Specifically, in a typical kernel, individual processes and kernel threads execute on separate page tables and have different page table roots. When a thread of execution enters the isolated domain in the kernel, one needs to ensure that `cr3` is valid on both sides of the isolation boundary. One way to achieve this is to ensure that for each pair of communicating subsystems, the physical page that contains the PTD is mapped twice by the EPT (Figure 4). The first mapping is the original EPT mapping used by the isolated subsystem. The second mapping is created dynamically when the thread of execution that uses a different PTD address in the caller subsystem enters the callee for the first time. Since the physical address of the PTD is the same in both the caller and the callee, the thread can perform a VMFUNC transition between the two domains.

To make sure that such double mapping is possible (note that the isolation subsystem has no control over the placement of the page table root), it is possible to enforce the following invariant:

> *Inv 2.b:* The physical address spaces of isolated domains and the kernel must not overlap.

This guarantees that the GPA that contains the root of the page table inside an isolated subsystem is not used in any other subsystems, and hence, can be remapped into the host physical page that contains the root of the page table on the other side.

Isolating Privileged Kernel Code

Historically, hardware isolation mechanisms focused on the isolation of user-level code. Isolation of privileged code remains a challenge in the face of a sensitive hardware state that is accessible from ring 0. Historically, the kernel isolation solutions either moved isolated code outside of ring 0 to ring 3 or relied on techniques of language safety or SFI to control access to privileged instructions. VMFUNC, however, provides a unique point in the design space of isolation solutions by executing the system under the control of a hypervisor.[4] Virtualized execution allows mediating access to the sensitive state with only minimal overhead but requires

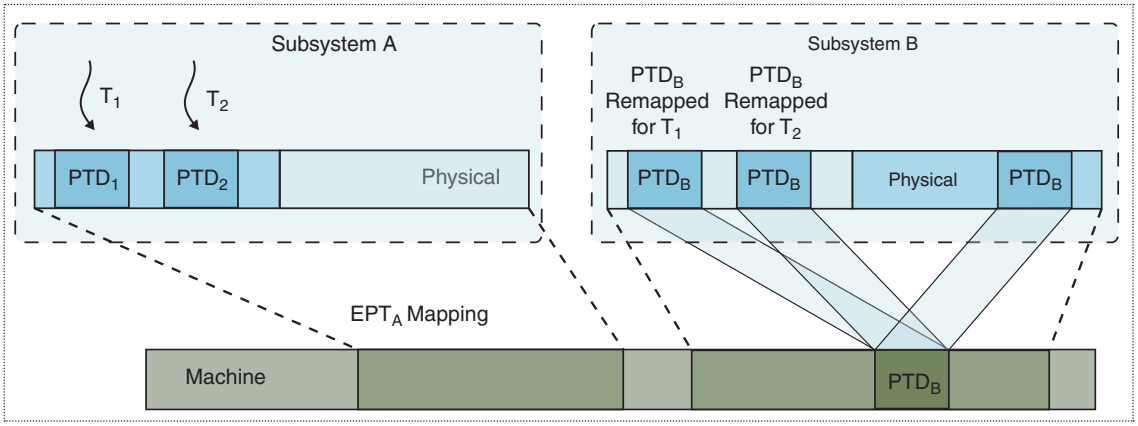

Figure 4. `cr3` remapping. The machine page that contains the root of the page table inside *B* is remapped, so threads T_1 and T_2 perform a VMFUNC transition from A and B.

careful handling of interrupt transitions to avoid severe performance penalties on exits from the VT-x nonroot context.

Protecting the Sensitive State

The efficient isolation of kernel code relies on the assumption that both the core kernel and isolated subsystems execute with ring 0 privileges. This allows one to avoid expensive privilege-level transitions on invocations into isolated subsystems. In turn, an isolated subsystem that runs in ring 0 has access to all sensitive hardware registers, for example, control registers that alter the behavior of the hardware, like the root of the page table stored in `cr3`. To ensure isolation, the isolation boundary should enforce the following invariant:

> *Inv 3:* Access to the sensitive state within the isolated domain is mediated by the hypervisor.

To implement Inv 3, the guest VM has to be configured to exit into the hypervisor on the following instructions that access the sensitive state:

1. stores to the control registers (`cr0`, `cr3`, and `cr4`)
2. stores to the extended control register, `xcr0`
3. reads and writes of MSRs with `rdmsr` and `wrmsr` instructions
4. reads and writes of I/O ports with `in` and `out` instructions
5. access to debug registers
6. loads and stores of descriptor tables, that is, `GDT`, `LDT`, `IDT`, and `TR` registers.

Upon exit, the hypervisor validates if the exit happens from the legitimate use of these instructions in the nonisolated kernel, and if so, emulates the exit-causing instruction. Empirical evaluation demonstrates that VM exits caused by such mediation have little impact on realistic system workloads.[4] Exits due to interrupts, on the other hand, are frequent in I/O-intensive workloads (we discuss a practical approach for addressing this overhead with exitless interrupt handling later).

Locking the Isolated Domain's Address Space

Inv 1.a (virtual address spaces across isolated subsystems do not overlap) becomes challenging in the privileged ring 0 environment—privileged code can modify the page table to create an overlapping mapping. The isolation subsystem has to ensure that isolated subsystems cannot modify the layout of their address space, or specifically:

> *Inv 4:* Isolated domains have read-only access to their page table.

Enforcing this invariant without adding performance overhead is surprisingly hard. A naive approach is to disable updates to the `cr3` register that holds the root of the page table hierarchy along with updates to pages of the page table itself by mapping them read-only in the EPT. This implies that the entire page table hierarchy cannot be modified. Though this enforces Inv 4, it also causes a prohibitive number of exits into the hypervisor when the hardware tries to update the accessed and dirty bits in the page table of the isolated subsystem.

Fortunately, an important observation is that accessed and dirty bits on the page table pages are updated by the CPU via the GPA, that is, hardware walks the EPT to resolve the GPA address of the host physical page of the page table page that contains the accessed and dirty bits. To avoid exits on hardware accesses, it is possible to leave all pages of the page table mapped

with writable permissions in the EPT but enforce read-only access to those pages in the guest page table (Figure 5). This way, when the CPU updates the accessed and dirty bits, the access is allowed by the EPT, but the write access from the guest system results in a page fault when the kernel tries to update it through the guest virtual address.

A natural question is: If a page table is read-only inside an isolated subsystem, how can it grow its address space when allocating pages from a non-isolated kernel? An elegant solution is to create a large virtual address space upfront when the subsystem starts, that is, create a page table that maps GVAs to GPAs but keeps the physical pages unmapped and not backed by host physical pages in the EPT. This way, the subsystem never updates its page table. Instead, additional memory is mapped by an update in the EPT to create a mapping of a guest physical page to a host physical page.

Exitless Interrupt Handling

A natural concern with VMFUNC-based isolation is an increased cost of interrupt delivery. While the VT-x interface can be configured to avoid exits into the hypervisor on interrupt delivery, in most cases, this is unsafe—an isolated subsystem cannot be trusted to handle the interrupt. A naive approach is to trigger an exit into the hypervisor and then reinject the interrupt into the guest kernel.

Fortunately, it is possible to avoid the excessive overhead of interrupt delivery by switching (with VMFUNC) into the core nonisolated kernel instead of exiting into the hypervisor if the interrupt is delivered while one of the isolated subsystems is running.[4] Specifically, it is possible to allow delivery of interrupts directly into the VT-x nonroot guest without exiting into the hypervisor and also allow interrupt delivery into an isolated subsystem but through a protected interrupt descriptor table (IDT) configured by the nonisolated kernel. To enable interrupt delivery into the isolated subsystem, the IDT, global descriptor table, task-state segment (TSS), and interrupt handler trampoline are mapped by both the nonisolated kernel (EPT_K) and the isolated subsystem (EPT_I).

When the system receives an interrupt, the kernel follows a normal interrupt delivery path, that is, the hardware saves the state of the currently executing thread on the stack, locates the interrupt handler through the IDT, and starts executing it. However, the interrupt handler that is controlled by the nonisolated kernel starts with a check of whether the execution is still inside the isolated subsystem, and if so, performs a VMFUNC transition back to the kernel. While conceptually simple, the exitless interrupt delivery with the VMFUNC scheme requires several careful design decisions to maintain the security of the isolation boundary as the isolated domain is running with ring 0 privileges.

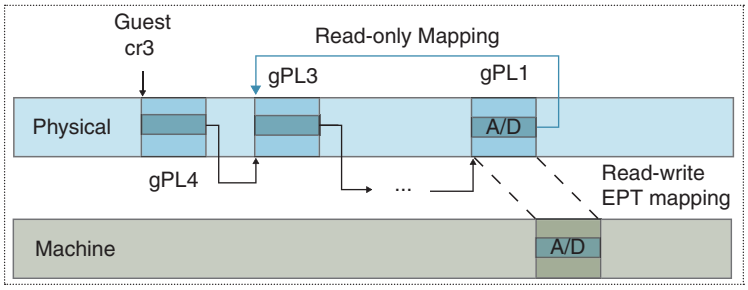

Figure 5. Enforcing read-only access for the pages of the guest page table. The page table entry in gPL1 maps a page of the page table (gPL3) as read-only.

Interrupt Stack Table

Both nonisolated kernel and isolated subsystems execute with ring 0 privileges. Therefore, on interrupt, the CPU does not change the privilege level and continues executing on the same stack. Specifically, the CPU saves the trap frame on the stack pointed by the current kernel stack pointer. This opens an opportunity for a trivial attack. A malicious subsystem configures the stack to point to a valid writable kernel memory and waits for an interrupt. When an interrupt is delivered, the CPU saves the trap frame onto the stack, thereby overwriting the kernel memory.

To prevent such an attack and to make sure that an interrupt is always executed on a valid stack, it is possible to utilize a hardware mechanism, interrupt stack table (IST), which unconditionally switches the stack to a preconfigured address. With IST, one can configure the IDT on a per-handler basis to handle the interrupt either using a traditional interrupt delivery mechanism or unconditionally switching to a preconfigured IST stack even if the privilege level remains unchanged. Each IDT entry has 8 b to specify one of the seven available IST stacks. The Linux kernel already uses the IST mechanism for handling nonmaskable interrupts (NMIs), double-fault, debug, and machine-check exceptions.

To protect the kernel from this attack, one has to configure two additional IST stacks for the execution of synchronous exceptions and asynchronous interrupts (Figure 6). Upon receiving an interrupt, the hardware switches to a preconfigured IST stack from the IST table. First, the IST stack is used to execute a minimal interrupt handler trampoline that is mapped in both nonisolated kernels and all isolated subsystems. The trampoline checks whether the system is running inside the kernel or in one of the isolated subsystems and switches to the kernel's EPT (EPT_K), if needed. Since no register is trusted upon entering

the kernel with VMFUNC, the isolation mechanism securely restores the system's state from the TLS described previously (it is possible to restore the `gs` register used by the kernel to maintain per-CPU data structures and the stack pointer register that points to the kernel stack).

After that, the saved interrupt frame is copied to the regular kernel stack, which is used to continue the execution of the interrupt handler through the normal interrupt-handling path. Note that the kernel can reenable interrupts at this point as the IST stack is no longer used for the current interrupt. On the interrupt-return path, the kernel switches back to the isolated domain if required. If needed, the handler copies the exception frame back to the IST stack (since only the IST stack is mapped inside the isolated subsystem), switches back to the EPT of the isolated subsystem, EPT_I, and returns from the interrupt with the regular `iret` instruction.

The aforementioned interrupt delivery scheme relies on the possibility of disabling subsequent interrupts on all interrupt transitions—this ensures that the IST stack will not be overwritten until the interrupt frame is copied out onto the normal kernel stack. Anytime during the processing of the interrupt, an NMI can be delivered. One has to configure a separate IST stack for the NMI to prevent overwriting the state of the previous interrupt frame on the IST.

To reliably detect whether the interrupt handler is running inside the kernel or an isolated subsystem, it is possible to utilize a special "state" page, `state page`, that is mapped by both the kernel and the isolated subsystem at the same physical address. Inside the kernel, the state page has a flag set to true and false in all isolated subsystems.

Alternative VMFUNC Isolation Schemes

SeCage was the first system to demonstrate the use of EPT switching for isolation.[5] While certain aspects of the implementation are not publicly available, SeCage ensures virtual address space isolation and control over a subset of the guest page table to ensure the safety of the isolation boundary. Skybridge, on the other hand, leverages binary rewriting techniques similar to ERIM to prevent a malicious subsystem from crafting unprotected instances of the `vmfunc` instruction.[3] EPTI leverages VMFUNC to defend against meltdown attacks by isolating the user and kernel address spaces.[6] EPTI develops novel techniques for ensuring read-only access to the page table hierarchy within the guest system that we use in our work on lightweight virtualized domains (LVDs).[4] Concentrating on the performance aspect of VMFUNC isolation, MemSentry leaves most VMFUNC-related attacks out of scope.[15]

VMFUNC implements a unique design spot in the space of hardware isolation mechanisms. Several limitations hinder the immediate adoption of VMFUNC-based isolation solutions. First, isolation with VMFUNC requires several nonstandard invariants that control the layout of both virtual and physical address spaces across isolated subsystems. While the enforcement of these invariants is possible in experimental research prototypes, it is not clear whether such complexity is practical for production systems.

Second, EPT switching has a higher overhead compared to a simpler tag-based MPK. Updating the tag with the `wrpkru` instruction takes only 20–26 cycles, while the execution of `vmfunc` takes 109–147 cycles. Furthermore, EPT switching has no support for zero-copy communication, which is naturally enabled in tag-based solutions like MPK and MTE and isolation schemes based on PAC (passing a pointer to a memory region required reencrypting the pointer for the callee subsystem).

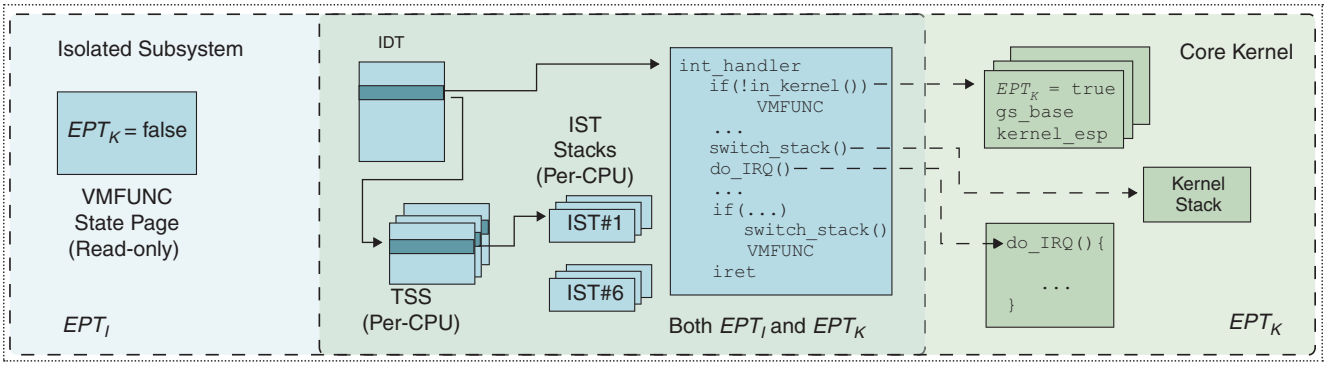

Figure 6. Data structures involved in exitless interrupt delivery (Linux kernel). EPT_k: extended Page table (kernel); IDT: interrupt descriptor table; TSS: task state segment.

Third, another limitation of VMFUNC-based isolation schemes is the requirement to execute the system under the control of a hypervisor. While the overheads of virtualized execution can be addressed,[4] it is not clear if support for nested virtualization required to execute a VMFUNC-isolated system in a modern de facto virtualized cloud environment can be provided efficiently. More important, VMFUNC inherently requires trust in the hypervisor responsible for maintaining EPTs. Unfortunately, this goes against the recent industry trend to remove the hypervisor from the TCB through support for hardware-encrypted VMs like AMD SEV and Intel TDX.

On the other hand, compared to other isolation mechanisms, VMFUNC provides several unique advantages. First, in contrast to a tag-based mechanism like MPK and MTE, VMFUNC relies on the switching of an EPT. An obvious advantage of this approach is the ability to implement isolation without additional mechanisms required to ensure the safety of unprivileged `vmfunc` instructions, that is, without binary rewriting required to ensure the absence of malicious `wrpkru` instructions or enforcing a specific MTE tag.

Second, in contrast to tag-based solutions that are inherently limited to a small number of isolated domains due to the limited number of bits that can hold the current tag (in either unused bits of the page table entries or unused bits of the register), VMFUNC can support a large number of isolated domains. (While the current size of the EPT table is limited to 512 domains, there is no inherent architectural limitation that would prevent increasing this limit.) ■

References

1. A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg, "ERIM: Secure, efficient in-process isolation with protection keys (MPK)," in *Proc. 28th USENIX Secur. Symp.*, 2019, pp. 1221–1238.
2. M. Hedayati et al., "Hodor: Intra-process isolation for high-throughput data plane libraries," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf. (USENIX ATC)*, 2019, pp. 489–504.
3. Z. Mi, D. Li, Z. Yang, X. Wang, and H. Chen, "SkyBridge: Fast and secure inter-process communication for microkernels," in *Proc. 14th ACM Eur. Conf. Comput. Syst. (EuroSys)*, 2019, pp. 1–15, doi: 10.1145/3302424.3303946.
4. V. Narayanan, Y. Huang, G. Tan, T. Jaeger, and A. Burtsev, "Lightweight kernel isolation with virtualization and VM functions," in *Proc. 16th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environ. (VEE)*, 2020, pp. 157–171, doi: 10.1145/3381052.3381328.
5. Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia, "Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, New York, NY, USA: Association for Computing Machinery, 2015, pp. 1607–1619, doi: 10.1145/2810103.2813690.
6. Z. Hua, D. Du, Y. Xia, H. Chen, and B. Zang, "EPTI: Efficient defence against meltdown attack for unpatched VMs," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2018, pp. 255–266.
7. S. Park, S. Lee, W. Xu, H. Moon, and T. Kim, "libmpk: Software abstraction for Intel Memory Protection Keys (Intel MPK)," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2019, pp. 241–254.
8. J. Gu, H. Li, W. Li, Y. Xia, and H. Chen, "EPK: Scalable and efficient memory protection keys," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2022, pp. 609–624.
9. D. Sehr et al., "Adapting software fault isolation to contemporary CPU architectures," in *Proc. 19th USENIX Conf. Secur.*, 2010, pp. 1–11.
10. D. McKee et al., "Preventing kernel hacks with HAKC," in *Proc. Netw. Distrib. Syst. Secur. (NDSS)*, 2022, vol. 22, pp. 1–17, doi: 10.14722/ndss.2022.24026.
11. R. M. Farkhani, M. Ahmadi, and L. Lu, "PTAuth: Temporal memory safety via robust points-to authentication," in *Proc. 30th USENIX Secur. Symp.*, 2021, pp. 1037–1054.
12. Y. Li et al., "PACMem: Enforcing spatial and temporal memory safety via ARM pointer authentication," in *Proc. 29th ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, 2022, pp. 1901–1915, doi: 10.1145/3548606.3560598.
13. S. Yoo, J. Park, S. Kim, Y. Kim, and T. Kim, "In-kernel control-flow integrity on commodity OSes using ARM pointer authentication," in *Proc. 31st USENIX Conf. Secur.*, 2022, pp. 89–106.
14. M. Xie et al., "CETIS: Retrofitting Intel CET for generic and efficient intra-process memory isolation," in *Proc. 29th ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, 2022, pp. 2989–3002, doi: 10.1145/3548606.3559344.
15. K. Koning, X. Chen, H. Bos, C. Giuffrida, and E. Athanasopoulos, "No need to hide: Protecting safe regions on commodity hardware," in *Proc. 12th ACM Eur. Conf. Comput. Syst.*, 2017, pp. 437–452, doi: 10.1145/3064176.3064217.
16. A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis, "Dune: Safe user-level access to privileged CPU features," in *Proc. 10th USENIX Symp. Operating Syst. Des. Implementation (OSDI)*, 2012, pp. 335–348, doi: 10.5555/2387880.2387913.

Vikram Narayanan is a Ph.D. candidate at the School of Computing, University of Utah, Salt Lake City, UT 84108 USA. His research interests include operating systems and security. Narayanan received

a master's in computer science from the University of Saarland. Contact him at vikram@cs.utah.edu.

Anton Burtsev is an assistant professor at the School of Computing, University of Utah, Salt Lake City, UT 84108 USA. His research interests include the design and architecture of operating systems in the age of targeted security attacks, heterogeneous hardware, and data center-scale computing. Burtsev received a Ph.D. from the University of Utah and spent six years as a faculty at the University of California, Irvine before joining the University of Utah. Contact him at aburtsev@cs.utah.edu.