xMP: Selective Memory Protection for Kernel and User Space

Sergej Proskurin,* Marius Momeu,* Seyedhamed Ghavamnia,† Vasileios P. Kemerlis,‡ and Michalis Polychronakis†
*Technical University of Munich, †Stony Brook University, ‡Brown University
*{proskurin, momeu}@sec.in.tum.de, †{mikepo, sghavamnia}@cs.stonybrook.edu, ‡vpk@cs.brown.edu

***Abstract*—Attackers leverage memory corruption vulnerabilities to establish primitives for *reading* from or *writing* to the address space of a vulnerable process. These primitives form the foundation for code-reuse and data-oriented attacks. While various defenses against the former class of attacks have proven effective, mitigation of the latter remains an open problem. In this paper, we identify various shortcomings of the x86 architecture regarding memory isolation, and leverage virtualization to build an effective defense against data-oriented attacks. Our approach, called *xMP*, provides (in-guest) *selective memory protection* primitives that allow VMs to isolate sensitive data in user or kernel space in disjoint xMP domains. We interface the Xen `altp2m` subsystem with the Linux memory management system, lending VMs the flexibility to define custom policies. Contrary to conventional approaches, xMP takes advantage of virtualization extensions, but after initialization, it does not require any hypervisor intervention. To ensure the integrity of in-kernel management information and pointers to sensitive data within isolated domains, xMP protects pointers with HMACs bound to an immutable context, so that integrity validation succeeds only in the right context. We have applied xMP to protect the page tables and process credentials of the Linux kernel, as well as sensitive data in various user-space applications. Overall, our evaluation shows that xMP introduces minimal overhead for real-world workloads and applications, and offers effective protection against data-oriented attacks.**

I. Introduction

During the past three decades, data-oriented attacks have evolved from a theoretical exercise [1] to a serious threat [2]–[7]. During the same time, we have witnessed a plethora of effective security mechanisms that prompted attackers to investigate new directions and exploit less explored corners of victim systems. Specifically, recent advances in Control Flow Integrity (CFI) [8]–[12], Code Pointer Integrity (CPI) [13], [14], and code diversification [15]–[17] have significantly raised the bar for code-reuse attacks. In fact, CFI schemes have been adopted by Microsoft [18], Google [19], and LLVM [20].

Code-reuse attacks chain short code sequences, dubbed *gadgets*, to hijack an application's control-flow. It suffices to modify one control-flow structure, such as a function pointer or a return address, with the start of a crafted gadget chain, to cause an application to perform arbitrary computation. In contrast, data-oriented attacks completely avoid changes to the control flow. Instead, these attacks aim to modify *non-control data* to cause the application to obey the attacker's intentions [5]–[7]. Typically, an attacker leverages memory corruption vulnerabilities that enable arbitrary *read* or *write* *primitives* to take control over the application's data. Stitching together a chain of *data-oriented gadgets*, which operate only on data, allows an attacker to either disclose sensitive information or escalate privileges, without violating an application's control flow. In this way, data-oriented attacks remain under the radar, despite code-reuse mitigations, and can have disastrous consequences [3]. We anticipate further growth in this direction in the near future, and emphasize the need for practical primitives that eliminate such threats.

Researchers have suggested different strategies to counter data-oriented attacks. Data Flow Integrity (DFI) [21] schemes dynamically track a program's data flow. Similarly, by introducing memory safety to the C and C++ programming languages, it becomes possible to completely eliminate memory corruption vulnerabilities [22]–[25]. While both directions have the potential to thwart data-oriented attacks, they lack practicality due to high performance overhead, or suffer from compatibility issues with legacy code. Instead of enforcing data flow integrity, researchers have started exploring isolation techniques that govern access to sensitive code and data regions [26]–[28]. Still, most approaches are limited to user space, focus on merely protecting a single data structure, or rely on policies enforced by a hypervisor.

In this paper, we leverage virtualization extensions of Intel CPUs to establish *selective memory protection (xMP) primitives* that have the capability of thwarting data-oriented attacks. Instead of enhancing a hypervisor with the knowledge required to enforce memory isolation, we take advantage of Intel's Extended Page Table pointer (EPTP) switching capability to manage different views on guest-physical memory, from inside a VM, without any interaction with the hypervisor. For this, we extended Xen `altp2m` [29], [30] and the Linux memory management system to enable the selective protection of sensitive data in user or kernel space by isolating sensitive data in disjoint *xMP domains* that overcome the limited access permissions of the Memory Management Unit (MMU). A strong attacker with arbitrary *read* and *write* primitives cannot access the xMP-protected data without first having to enter the corresponding xMP domain. Furthermore, we equip in-kernel management information and pointers to sensitive data in xMP domains with authentication codes, whose integrity is bound to a specific context. This allows xMP to protect pointers and hence obstruct data-oriented attacks that target the xMP-protected data.

We use xMP to protect two sensitive kernel data structures that are vital for the system's security, yet are often disregarded by defense mechanisms: *page tables* and *process credentials*. In addition, we demonstrate the generality of xMP by guarding sensitive data in common, security-critical (user-space) libraries and applications. Lastly, in all cases, we evaluate the performance and effectiveness of our xMP primitives.

In summary, we make the following main contributions:

- We extend the Linux kernel to realize *xMP*, an in-guest memory isolation primitive for protecting sensitive data against data-oriented attacks in user and kernel space.
- We present methods for combining Intel's EPTP switching and Xen `altp2m` to control different guest-physical memory views, and isolate data in *disjoint xMP domains*.
- We apply xMP to guard the kernel's page tables and process credentials, as well as sensitive data in user-space applications, with minimal performance overhead.
- We integrate xMP into the Linux namespaces framework, forming the basis for hypervisor-assisted OS-level virtualization protection against data-oriented attacks.

II. Background

A. Memory Protection Keys

Intel's Memory Protection Keys (MPK) technology supplements the general paging mechanism by further restricting memory permissions. In particular, each paging structure entry dedicates four bits that associate virtual memory pages with one of 16 protection domains, which correspond to sets of pages whose access permissions are controlled by the same *protection key* (`PKEY`). User-space processes control the permissions of each `PKEY` through the 32-bit `PKRU` register. Specifically, MPK allows different `PKEY`s to be simultaneously active, and page table entries to be paired with different keys to further restrict access to the associated page.

A benefit of MPK is that it allows user threads to independently and efficiently harden the permissions of large memory regions. For instance, threads can revoke *write* access from entire domains without entering kernel space, walking and adjusting page tables, and invalidating TLBs; instead, threads can just set the *write disable* bit of the corresponding `PKEY` in the `PKRU` register. Another benefit of MPK is that it extends the access control capabilities of page tables, enabling threads to enforce (i) *execute-only* code pages [16], [31], and (ii) *non-readable*, yet *present* data pages [28]. These capabilities provide new primitives for thwarting data-oriented attacks, without sacrificing performance and practicality [9], or resorting to architectural quirks [32] and virtualization [12], [16], [33].

Although Intel announced MPK in 2015 [34], it was integrated only recently, and so far only to the Skylake-SP Xeon family, which is dedicated to high-end servers. Hence, a need for similar isolation features remains on desktop, mobile, and legacy server CPUs. Another issue is that attackers with the ability to arbitrarily corrupt kernel memory can (i) modify the per-thread state (in kernel space) holding the access permissions of protection domains, or (ii) alter protection domain bits in page table entries. This allows adversaries to deactivate restrictions that otherwise are enforced by the MMU. Lastly, the isolation capabilities of MPK are geared towards user-space pages. Sensitive data in kernel space thus remains prone to unauthorized access. In fact, there is no equivalent mechanism for protecting kernel memory from adversaries armed with arbitrary *read* and *write* primitives. Consequently, there is a need for alternative memory protection primitives, the creation of which is the main focus of this work.

B. The Xen `altp2m` Subsystem

Virtual Machine Monitors (VMMs) leverage Second Level Address Translation (SLAT) to isolate physical memory that is reserved for VMs [35]. In addition to in-guest page tables that translate guest-virtual to guest-physical addresses, the supplementary SLAT tables translate guest-physical to host-physical memory. Unauthorized accesses to guest-physical memory, which is either not mapped or lacks privileges in the SLAT table, trap into the VMM [36], [37]. As the VMM exclusively maintains the SLAT tables, it can fully control a VM's *view* on its physical memory [29], [30], [38], [39]. Xen's *physical-to-machine* subsystem (`p2m`) [37], [40] employs SLAT to define the guest's *view* of the physical memory that is perceived by all virtual CPUs (vCPUs). By restricting access to individual page frames, security mechanisms can use `p2m` to enforce memory access policies on the guest's physical memory.

Unfortunately, protecting data through a single *global* view (i) incurs a significant overhead and (ii) is prone to race conditions in multi-vCPU environments. Consider a scenario in which a guest advises the VMM to *read-protect* sensitive data on a specific page. By revoking *read* permissions in the SLAT tables, illegal *read* accesses to the protected page, e.g., due to malicious memory disclosure attempts, would violate the permissions and trap into the VMM. At the same time, for legal guest accesses to the protected page frame, the VMM has to temporarily relax its permissions. Whenever the guest needs to access the sensitive information, it has to instruct the VMM to walk the SLAT tables—an expensive operation. More importantly, temporarily relaxing permissions in the global view creates a *window of opportunity* for other vCPUs to freely access the sensitive data without notifying the VMM.

The Xen *alternate* `p2m` subsystem (`altp2m`) [29], [30] addresses the above issues by maintaining and switching between *different* views, instead of using a single, *global view*. As the views can be assigned to each vCPU individually, permissions in one view can be safely relaxed without affecting the active views of other vCPUs. In fact, instead of relaxing permissions by walking the SLAT tables, `altp2m` allows switching to another, less restrictive view. Both external [29], [30] and internal monitors [26], [41] use `altp2m` to allocate and switch between views. Although `altp2m` introduces a powerful means to rapidly change the guest's memory view, it requires hardware support to establish primitives that can be used by guests for isolating selected memory regions.

C. In-Guest EPT Management

Xen altp2m was introduced to add support for the Intel virtualization extension that allows VMs to switch among Extended Page Tables (EPTs), Intel's implementation of SLAT tables [35]. Specifically, Intel introduced the unprivileged VMFUNC instruction to enable VMs to switch among EPTs without involving the VMM—although altp2m has been implemented for Intel [42] and ARM [30], in-guest switching of altp2m views is available to Intel only. Intel uses the Virtual Machine Control Structure (VMCS) to maintain the host's and the VM's state per vCPU. The VMCS holds an Extended Page Table pointer (EPTP) to locate the root of the EPT. In fact, the VMCS has capacity for up to 512 EPTPs, each representing a different view of the guest's physical memory; using VMFUNC, a guest can choose among 512 EPTs.

To pick up the above scenario, the guest can instruct the system to isolate and relax permissions to selected memory regions, on-demand, using Xen's altp2m EPTP switching. Furthermore, combined with another feature, i.e., the Virtualization Exceptions (#VE), the Xen altp2m allows in-guest agents to take over EPT management tasks. More precisely, the guest can register a dedicated exception handler that is responsible for handling EPT access violations; instead of trapping into the VMM, the guest can intercept EPT violations and try to handle them inside a (guest-resident) #VE handler.

III. Threat Model

We expect the system to be protected from code injection [43] through Data Execution Prevention (DEP) or other proper W^X policy enforcement, and to employ Address Space Layout Randomization (ASLR) both in kernel [44], [45] and user space [15], [46]. Also, we assume that the kernel is protected against return-to-user (ret2usr) [47] attacks through SMEP/SMAP [35], [48], [49]. Other hardening features, such as Kernel Page Table Isolation (KPTI) [50], [51], stack smashing protection [52], and toolchain-based hardening [53], are orthogonal to xMP—we neither require nor preclude the use of such features. Moreover, we anticipate protection against state-of-the-art code-reuse attacks [4], [54]–[56] via either (i) fine-grained CFI [57] (in kernel [58] and user space [59]) coupled with a shadow stack [60], or (ii) fine-grained code diversification [61], [62], and with execute-only memory (available to both kernel [31] and user space [16]).

Assuming the above state-of-the-art protections prevent an attacker from gaining arbitrary code execution, we focus on defending against attacks that leak or modify sensitive data in user or kernel memory [16], [31], by transforming memory corruption vulnerabilities into *arbitrary read* and *write primitives*. Attackers can leverage such primitives to mount data-oriented attacks [7], [63] that (i) disclose sensitive data, such as cryptographic material, or (ii) modify sensitive data structures, such as page tables or process credentials.

IV. Design

To fulfil the need for a practical mechanism for the protection of sensitive data, we identify the following requirements:

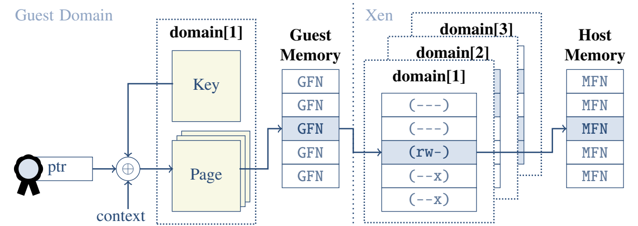

Figure 1. xMP uses different Xen altp2m views, each mapping guest frames to machine frames with different access permissions, to partition memory into isolated xMP domains. In addition, equipping data pointers to protected memory with HMACs establishes context-bound pointer integrity.

❶ *Partitioning* of sensitive kernel and user-space memory regions into individual domains.
❷ *Isolation* of memory domains through fine-grained access control capabilities.
❸ *Context-bound integrity* of pointers to memory domains.

Although the x86 architecture allows for memory partitioning through segmentation or paging ❶, it lacks fine-grained access control capabilities for effective memory isolation ❷ (e.g., there is no notion of *non-readable* pages; only *non-present* pages cannot be read). While previous work isolates user-space memory by leveraging unused, higher-privileged x86 protection rings [64], isolation of kernel memory is primarily achieved by Software-Fault Isolation (SFI) solutions [31]. Even though the page fault handler could be extended to interpret selected *non-present* pages as *non-readable*, switching permissions of memory regions that are shared among threads or processes on different CPUs can introduce race conditions: granting access to isolated domains by relaxing permissions inside the *global* page tables may reveal sensitive memory contents to the remaining CPUs. Besides, each permission switch would require walking the page tables, and thus frequent switching between a large number of protected pages would incur a high run-time overhead. Lastly, the modern x86 architecture lacks any support for immutable pointers. Although ARMv8.3 introduced the Pointer Authentication Code (PAC) [65] extension, there is no similar feature on x86. As such, x86 does not meet requirements ❷ and ❸.

In this work, we fill this gap by introducing *selective memory protection (xMP)* primitives that leverage virtualization to define efficient memory isolation domains—called *xMP domains*—in both kernel and user space, enforce fine-grained memory permissions on selected xMP domains, and protect the integrity of pointers to those domains (Figure 1). In the following, we introduce our xMP primitives and show how they can be used to build practical and effective defenses against data-oriented attacks in both user and kernel space. We base xMP on top of x86 and Xen [40], as it relies on virtualization extensions that are exclusive to the Intel architecture and are already used by Xen. Still, xMP is by no means limited to Xen, as we further discuss in § VIII-D. Furthermore, xMP is both backwards compatible with, and transparent to, non-protected and legacy applications.

A. Memory Partitioning through xMP Domains

To achieve meaningful protection, applications may require multiple *disjoint* memory domains that cannot be accessible at the same time. For instance, an xMP domain that holds the kernel's hardware encryption key must not be accessible upon entering an xMP domain containing the private key of a user-space application. The same applies to multi-threaded applications in which each thread maintains its own session key that must not be accessible by other threads. We employ Xen `altp2m` as a building block for providing disjoint xMP domains (§ II-B). An xMP domain may exist in one of two states, the permissions of which are configured as desired. In the *protected state*, the most restrictive permissions are enforced to prevent data leakage or modification. In the *relaxed state*, the permissions are temporarily loosened to enable legitimate access to the protected data as needed.

The straightforward way of associating an `altp2m` view with each xMP domain is not feasible because only a single `altp2m` view can be active at a given time. Instead, to enforce the access restrictions of all xMP domains in each `altp2m` view, we propagate the permissions of each domain across all available `altp2m` views. Setting up an xMP domain requires at least two `altp2m` views. Regardless of the number of xMP domains, we dedicate one view, the *restricted view*, to unify the memory access restrictions of all xMP domains. We configure this view as the default on every vCPU, as it collectively enforces the restrictions of all xMP domains. We use the second view to *relax* the restrictions of (i.e., *unprotect*) a given xMP domain and to allow legitimate access to its data. We refer to this view as *domain[id]*, with *id* referring to the xMP domain of this view. By entering *domain[id]*, the system switches to the `altp2m` view *id* to bring the xMP domain into its relaxed state—crucially, all other xMP domains remain in their protected state. By switching to the *restricted view*, the system switches *all* domains to their protected state.

To accommodate n xMP domains, we define $n+1$ `altp2m` views. Figure 2 illustrates a multi-domain environment with *domain[n]* as the currently active domain (the page frames of each domain are denoted by the darkest shade). The permissions of *domain[n]* in its relaxed and protected states are `r-x` and `--x`, respectively. The `--x` permissions of *domain[n]*'s protected state are enforced not only by the restricted view, but also by *all other* xMP domains ($\{domain[j] \mid \forall j \in \{1, ..., n\} \wedge j \neq n\}$). This allows us to partition the guest's physical memory into multiple domains and to impose fine-grained memory restrictions on each of them, satisfying ❶.

An alternative approach to using `altp2m` would be to rely on Intel MPK (§ II-A). Although MPK meets requirements ❶ and ❷, unfortunately it is applicable only to user-space applications and cannot withstand abuse by memory corruption vulnerabilities targeting the kernel. Due to the limited capabilities of MPK, and since Intel has only recently started shipping server CPUs with MPK, we opted for a solution that works on both recent and legacy systems, and can protect both user-space and kernel-space memory.

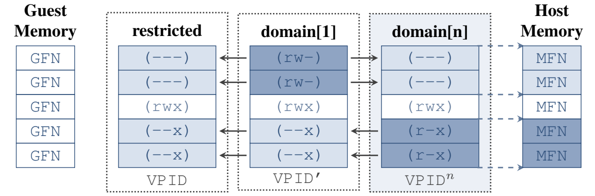

Figure 2. The system configures $n+1$ `altp2m` views to create n disjoint xMP domains. Each $\{domain[i] \mid i \in \{1, ..., n\}\}$ relaxes the permissions of a given memory region (dark shade) and restricts access to memory regions belonging to other xMP domains (light shade).

B. Isolation of xMP Domains

We establish a *memory isolation primitive* that empowers guests to enforce fine-grained permissions on the guest's page frames. To achieve this, we extended the Xen interface to allow utilizing `altp2m` from inside guest VMs. Specifically, we implemented an API for the Linux kernel that allows the use of existing hypercalls (`HVMOP`s) [42] that interact with the Xen `altp2m` subsystem for triggering the VMM to configure page frames with requested access permissions on behalf of the VM. Also, for performance reasons, we extended Xen with a new hypercall: `HVMOP_altp2m_isolate_xmp`. This hypercall places the guest's page frames into xMP domains according to § IV-A. Although this hypercall is not vital for xMP (its functionality can be substituted by a set of existing hypercalls to Xen `altp2m`), it reduces the number of interactions with the hypervisor. Finally, we use `altp2m` in combination with Intel's in-guest *EPTP switching* and the `#VE` feature to allow in-guest agents to take over several EPT management tasks (§ II-C). This setup minimizes VMM interventions and thus improves performance. Consequently, we do not have to outsource logic to the VMM or to an external monitor, as the scheme provides flexibility for defining new memory access policies from inside the guest.

Consider an in-guest application that handles sensitive data, such as passwords, cookies, or cryptographic keys. To protect this data, the application can use the *memory partitioning primitives* that leverage `altp2m` to allocate an xMP domain, e.g, *domain[1]* in Figure 2: in addition to *domain[1]* holding original access permissions to the guest's physical memory, our *memory isolation primitive* removes *read* and *write* permissions from the page frame in the *restricted view* (and remaining *domains*). This way, unauthorized *read* and *write* attempts outside *domain[1]* will violate the restricted access permissions. Instead of trapping into the VMM, any illegal access traps into the in-guest `#VE`-handler, which generates a segmentation fault. Upon legal accesses, instead of instructing the VMM to walk the EPTs to relax permissions, the guest executes the `VMFUNC` instruction to switch to the less-restrictive *domain[1]* and serve the request. As soon as the application completes its request, it will use `VMFUNC` to switch back to the *restricted view* and continue execution.

This scheme combines the best of both worlds: flexibility in defining policies, and fine-grained permissions that are not available to the traditional x86 MMU. Our primitives allow

in-guest applications to revoke *read* and *write* permissions on data pages, without making them *non-present*, and to configure code pages as *execute-only*, hence satisfying requirement ❷.

C. Context-bound Pointer Integrity

For complete protection, we have to ensure the integrity of pointers to sensitive data within xMP domains. Otherwise, by exploiting a memory corruption vulnerability, adversaries could redirect pointers to (i) injected, attacker-controlled objects outside the protected domain, or (ii) existing, high-privileged objects inside the xMP domain.

As x86 lacks support for pointer integrity (in contrast to ARM, in which PAC [65], [66] was recently introduced), we protect pointers to objects in xMP domains in software. We leverage the Linux kernel implementation of SipHash [67] to compute Keyed-Hash Message Authentication Codes (HMACs), which we use to authenticate selected pointers. SipHash is a cryptographically strong family of pseudorandom functions. Contrary to other secure hash functions (including the SHA family), SipHash is optimized for short inputs, such as pointers, and thus achieves higher performance. To reduce the probability of collisions, SipHash uses a 128-bit secret key. The security of SipHash is limited by its key and output size. Yet, with pointer integrity, the attacker has only one chance to guess the correct value; otherwise, the application will crash and the key will be re-generated.

To ensure that pointers cannot be illegally redirected to existing objects, we bind pointers to a specific context that is unique and immutable. The `task_struct` data structure holds thread context information and is unique to each thread on the system. As such, we can bind pointers to sensitive, task-specific data located in an xMP domain to the address of the given thread's `task_struct` instance.

Modern x86 processors use a configurable number of page table levels that define the size of virtual addresses. On a system with four levels of page tables, addresses occupy the first 48 least-significant bits. The remaining 16 bits are sign-extended with a value dependent on the privilege level: they are filled with ones in kernel space and with zeros in user space [68]. This allows us to reuse the unused, sign-extended part of virtual addresses and to truncate the resulting HMAC to 15 bits. At the same time, we can use the most-significant bit 63 of a canonical address to determine its affiliation—if bit 63 is set, the pointer references kernel memory. This allows us to establish pointer integrity and ensure that pointers can be used only in the right context ❸.

Contrary to ARM PAC, instead of storing keys in registers, we maintain one SipHash key per xMP domain in memory. After generating a key for a given xMP domain, we grant the page holding the key *read-only* access permissions inside the particular domain (all other domains cannot access this page). In addition, we configure Xen `altp2m` so that every xMP domain maps the same guest-physical address to a different machine-physical address. Every time the guest kernel enters an xMP domain, it will use the key that is dedicated to this domain (Figure 1). In fact, by reserving one specific memory page for keys, via the kernel's linker script, we allow the kernel to embed key addresses as immediate instruction operands that cannot be controlled by adversaries (i.e., code regions are immutable). Alternatively, to exclude the compiler from managing key material, we can leverage the VMM to establish a trusted path for generating the secret key and provisioning its location to the VM, e.g., through relocation information [62]. In particular, by leveraging the relocation entries related to the kernel image, the VMM can replace placeholders at given kernel instructions with the virtual addresses of (secret) key locations, upon the first access to the respective key. Alternatively, we can provide the hypervisor with all placeholder locations through an offline channel [26].

V. Implementation

We extended the Linux memory management system to establish memory isolation capabilities that allow us to partition ❶ selected memory regions into isolated ❷ xMP domains. During the system boot process, once the kernel has parsed the `e820` *memory map* provided by BIOS/UEFI to lay down a representation of the entire physical memory, it abandons its early memory allocators and hands over control to its core components. These consist of: (i) the (zoned) buddy allocator, which manages physical memory; (ii) the slab allocator, which allocates physically-contiguous memory in the `physmap` region of the kernel space [68], and is typically accessed via `kmalloc`; and (iii) the `vmalloc` allocator, which returns memory in a separate region of kernel space, i.e., the `vmalloc` arena [31], which can be virtually-contiguous but physically-scattered. Both `kmalloc` and `vmalloc` use the buddy allocator to acquire physical memory.

Note that (i) is responsible for managing (contiguous) pages frames, (ii) manages memory in sub-page granularity, and (iii) supports only page-multiple allocations. To provide maximum flexibility, we extend both (i) and (ii) to selectively shift allocated pages into dedicated xMP domains (Figure 3); (iii) is transparently supported by handling (i). This essentially allows us to isolate either arbitrary pages or entire slab caches. By additionally generating context-bound authentication codes for pointers referencing objects residing in the isolated memory, we meet all requirements ❶-❸.

A. Buddy Allocator

The Linux memory allocators use *get-free-page* (`GFP_*`) flags to indicate the conditions, the location in memory (zone), and the way the allocation will be handled [69]. For instance, `GFP_KERNEL`, which is used for most in-kernel allocations, is a collection of fine-granularity flags that indicate the default settings for kernel allocations. To instruct the buddy allocator to allocate a number of pages and to place the allocation into a specific xMP domain, we extend the allocation flags. That is, we can inform the allocator by adding the `__GFP_XMP` flag to any of the system's `GFP` allocation flags. This allows us to assign an arbitrary number of pages in different memory zones with fine-granularity memory access permissions. By additionally encoding an xMP domain index into the allocation

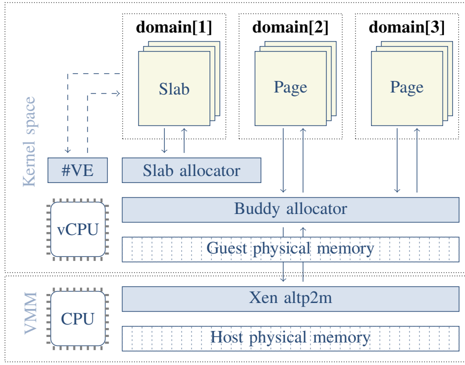

Figure 3. Extensions to the *slab* and *buddy allocator* facilitate shifting allocated pages and slabs into xMP domains enforced by Xen `altp2m`.

flags, the allocator receives sufficient information to inform the Xen `altp2m` subsystem to place the allocation into a particular xMP domain (Figure 3). Currently, we use 8 free bits in the allocation flags to encode the domain index, effectively supporting up to 256 distinct domains—more domains can be supported by redefining `gfp_t` accordingly. This way, we can grant exclusive access permissions to all pages assigned to the target xMP domain, while, at the same time, we can selectively withdraw access permissions to the allocated page from all other domains (§ IV-A). As such, accesses to pages inside the target domain become valid only after switching to the associated guest memory view managed by Xen `altp2m`.

During the assignment of allocated pages to xMP domains, we record the `PG_xmp` flag into the `flags` field of `struct page`, thereby enabling the buddy allocator to reclaim previously xMP-protected pages at a later point in time.

B. Slab Allocator

The slab allocator builds on top of the buddy allocator to subdivide allocated pages into small, sub-page sized objects (Figure 3), to reduce internal fragmentation that would otherwise be introduced by the buddy allocator. More precisely, the slab allocator maintains *slab caches* that are dedicated to frequently used kernel objects of the same size [70]. For instance, the kernel uses a cache for all `struct task_struct` instances. Such caches allow the kernel to allocate and free objects in a very efficient way, without the need for explicitly retrieving and releasing memory for every kernel object allocation. Historically, the Linux kernel has used three slab allocator implementations: `SLOB`, `SLAB`, and `SLUB`, with the latter being the default slab allocator in modern Linux kernels.

Every slab cache groups collections of continuous pages into so-called *slabs*, which are sliced into small-sized objects. Disregarding further slab architecture details, as the allocator manages slabs in dedicated pages, this design allows us to place selected slabs into isolated xMP domains using the underlying buddy allocator. To achieve this, we extend the slab implementation so that we can provide the `___GFP_XMP` flag and xMP domain index on creation of the slab cache. Consequently, every time the slab cache requests further pages for its slabs, it causes the buddy allocator to shift the allocated memory into the specified xMP domain (§ V-A).

C. Switches across Execution Contexts

The Linux kernel is a preemptive, highly-parallel system that must preserve the process-specific or thread-specific state on (i) *context switches* and (ii) *interrupts*. To endure context switches, and also prevent other threads from accessing isolated memory, it is essential to include the index of the thread's (open) xMP domain into its persistent state.[1]

1) Context Switches: In general, operating systems associate processes or threads with a dedicated data structure, the Process Control Block (PCB): a container for the thread's state that is saved and restored upon every context switch. On Linux, the PCB is represented by the `struct task_struct`. We extended `task_struct` with an additional field, namely `xmp_index_kernel`, representing the xMP domain the thread resides in at any point in time. We dedicate this field to store the state of the xMP domain used in kernel space. By default, this field is initialized with the index of the *restricted view* that accumulates the restrictions enforced by every defined xMP domain (§ IV-A). The thread updates its `xmp_index_kernel` only when it enters or exits an xMP domain. This way, the kernel can safely interrupt the thread, preserve its open xMP domain, and schedule a different thread. In fact, we extended the scheduler so that on every context switch it switches to the saved xMP domain of the thread that is to be scheduled next. To counter switching to a potentially corrupted `xmp_index_kernel`, we bind this index to the address of the `task_struct` instance in which it resides. This allows us to verify the integrity and context of the index before entering the xMP domain ❸ (§ IV-C). Since adversaries cannot create valid authentication codes without knowing the respective secret key, they will neither be able to forge the authentication code of the index, nor reuse an existing code that is bound to a different `task_struct`.

2) Hardware Interrupts: Interrupts can pause a thread's execution at arbitrary points. In our current prototype, accesses to memory belonging to any of the xMP domains are restricted in interrupt (IRQ) context. (We plan on investigating primitives for selective memory protection in IRQ contexts in the future.) To achieve this, we extend the prologue of every interrupt handler and cause it to switch to the *restricted view*. This way, we prevent potentially vulnerable interrupt handlers from illegally accessing protected memory. Once the kernel returns control to the interrupted thread, it will cause a memory access violation when accessing the isolated memory. Yet, instead of trapping into the VMM, the thread will trap into the in-guest `#VE` handler (§ II-C). The `#VE` handler, much like a page fault handler, verifies the thread's eligibility and context-bound integrity by authenticating the HMAC of its `xmp_index_kernel`. If the thread's eligibility and the index's integrity is given, the handler enters the corresponding

[1] Threads in user space enter the kernel to handle system calls and (a)synchronous interrupts. Specifically, upon interrupts, the kernel reuses the `task_struct` of the interrupted thread, which must be handled with care.

xMP domain and continues the thread's execution. Otherwise, it causes a segmentation fault and terminates the thread.

3) Software Interrupts: The above extensions introduce a restriction with regard to *nested* xMP domains. Without maintaining the state of nested domains, we require every thread to close its active domain before opening another one; by nesting xMP domains, the state of the active domain will be overwritten and lost. Although we can address this requirement for threads in *process context*, it becomes an issue in *interrupt context*: the former executes (kernel and user space) threads that are tied to different `task_struct` structures, while the latter reuses the `task_struct` of interrupted threads.

In contrast to hardware interrupts that disrupt the system's execution at arbitrary locations, the kernel explicitly schedules software interrupts (`softirq`) [71], e.g., after handling a hardware interrupt or at the end of a system call. As soon as the kernel selects a convenient time slot to schedule a `softirq`, it will temporarily delay the execution of the active process and reuse its context for handling the pending `softirq`.

The Linux kernel configures 10 `softirq` vectors, with one dedicated for the Read-Copy-Update (RCU) mechanism [72]. A key feature of RCU is that every update is split into (i) a *removal* and (ii) a *reclamation* phase. While (i) removes references to data structures in parallel to readers, (ii) releases the memory of removed objects. To free the object's memory, a caller registers a callback that is executed by the dedicated `softirq` at a later point it time. If the callback accesses and frees memory inside an xMP domain, it must first enter the associated domain. Yet, as the callback reuses the `task_struct` instance of an arbitrary thread, it must not update the thread's index to its open xMP domain.

To approach this issue, we leverage the callback-free RCU feature of Linux (`CONFIG_RCU_NOCB_CPU`). Instead of handling RCU callbacks in a `softirq`, the kernel dedicates a thread to handle the work. This simplifies the management of the thread-specific state of open xMP domains, as we can bind it to each task individually: if the thread responsible for executing RCU callbacks needs to enter a specific xMP domain, it can do so without affecting other tasks. As is the case with hardware IRQs, xMP does not allow deferring work that accesses protected memory in `softirq` context.

D. User Space API

We grant user processes the ability to protect selected memory regions by extending the Linux kernel with four new system calls that allow processes to use xMP in user space (Figure 4). Specifically, applications can dynamically allocate and maintain disjoint xMP domains in which sensitive data can remain safe (➊-➋). Furthermore, we ensure that attackers cannot illegally influence a process' active xMP domain state by binding its integrity to the thread's context (➌).

Linux has provided an interface for Intel MPK since kernel v4.9. This interface comprises three system calls, `sys_pkey_{alloc,free,mprotect}`, backed by `libc` wrapper functions for the allocation, freeing, and assignment of user space memory pages to protection keys. Applications

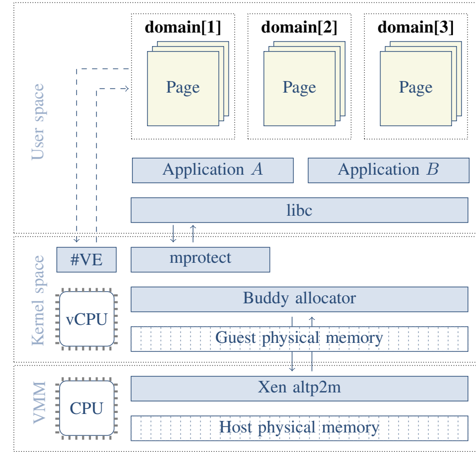

Figure 4. User-space applications interact with the Linux kernel through `mprotect` to configure independent xMP domains.

use the unprivileged `WRPKU` instruction to further manage memory access permissions of the corresponding protection keys (§ II-A). Likewise, we implemented the system calls `sys_xmp_{alloc,free,mprotect}`, which utilize `altp2m HVMOPs` (§ IV-B) for allowing programmers to allocate and maintain different xMP domains in user space. In fact, these system calls implement functionality equivalent to Intel MPK on Linux; they can be used as a transparent alternative on legacy systems without sufficient hardware support (➊-➋). On `sys_xmp_mprotect` invocation, we isolate the target virtual memory area (§ IV-B) and tag it so that we can identify protected memory and release it upon memory reclamation.

Contrary to the MPK implementation of Linux, we do not use the unprivileged `VMFUNC` instruction in user space. Instead, we provide an additional system call, namely `sys_xmp_enter`, which enters a requested, previously allocated xMP domain (either more or less restricted) and updates the state of the currently active xMP domain. We save the respective state inside the `xmp_index_user` field of `mm_struct` that is unique to every thread in user space. Also, we bind this index to the address of `mm_struct` (➌). This enables the kernel to verify the integrity and context of the xMP domain index on context switches—in other words, the kernel has the means to detect unauthorized modifications of this field and immediately terminate the application. Note that, with regard to our threat model, we anticipate orthogonal defenses in user space that severely restrain attackers to data-oriented attacks (§ III). By further removing `VMFUNC` instructions from user space, and mediating their execution via `sys_xmp_enter`, we avoid unnecessary Return-Oriented Programming (ROP) (or similar code-reuse) gadgets, which could be (ab)used to illegally switch to arbitrary xMP domains.

VI. Use Cases

We demonstrate the effectiveness and usefulness of xMP by applying it on: (a) *page tables* and *process credentials*, in the Linux kernel; and (b) sensitive *in-process data* in four security-critical applications and libraries.

A. Protecting Page Tables

With Supervisor Mode Execution Protection (SMEP) [48], the kernel cannot execute code in user space; adversaries have to first inject code into kernel memory to accomplish their goal. Multiple vectors exist that allow attackers to (legitimately) inject code into the kernel. In fact, system calls use the routine `copy_from_user` to copy a user-controlled (and potentially malicious) buffer into kernel memory. While getting code into the kernel is easy, its execution is obstructed by different security mechanisms. For instance, *W⊕X* withdraws *execute* permissions from the memory that contains data copied from user space. In addition, defenses based on *information hiding*, such as Kernel Space Address Layout Randomization (KASLR) [45], further obstruct kernel attacks but are known to be imperfect [51], [73]–[75]. Once adversaries locate the injected code, they can abuse memory corruption vulnerabilities, e.g., in device drivers or the kernel itself, to compromise the system's page tables [76]. This, in turn, opens up the gate for code injection or kernel code manipulation. Consequently, ensuring the integrity of page tables is an essential requirement, which remains unfulfilled by existing kernel hardening techniques [76]–[78].

Our goal is to leverage xMP to prevent adversaries from illegally modifying (i) page table contents and (ii) pointers to page tables. At the same time, xMP has to allow the kernel to update page table structures from authorized locations. With the exception of the initial page tables that are generated during the early kernel boot stage, the kernel uses the buddy allocator to allocate memory for new sets of page tables. Using the buddy allocator, we move every memory page holding a page table structure into a dedicated xMP domain, to which we grant *read-write* access permissions (§ V-A), and limit the access of remaining domains to *read-only*. As the kernel allocates the initial page tables statically, we manually inform Xen `altp2m` to place affected guest-physical page frames into the same domain. Every *write* access from outside the dedicated xMP domain results in an access violation that terminates the process. Thus, we must grant access to the protected paging structures to the kernel components responsible for page fault handling and process creation and termination, by enabling them to *temporarily* enter the xMP domain. This scheme does not disrupt the kernel's functionality and complies with requirements ❶ and ❷.

In addition, we extend the kernel's process and thread creation functionality to protect the integrity of every `pgd` pointer referencing the root of a page table hierarchy. We equip every `pgd` pointer with an HMAC (§ IV-C), and verify its integrity every time the pointer gets written to `CR3` (the control register holding the address of the page table root). This protects the pointer from corruption: as long as adversaries do not know the secret key, they cannot create a valid HMAC. Attackers cannot read the secret key as it remains inaccessible from outside the target domain. Attackers also cannot adjust the pointer to the key, as its address is compiled as an *immediate operand* into kernel instructions, and is thus immutable.

Still, we cannot bind the `pgd` to a specific thread context, as kernel threads inherit the `mm_struct` of interrupted user threads. This, however, does not weaken our protection. From the attackers' perspective, it is impossible to redirect the `pgd` to a different location, as they do not know the key. One attack scenario is to exchange the `pgd` pointer with a different `pgd` that holds a valid authentication code for another existing thread. Yet, this would not allow the attacker to inject a new address space, but just crash the application. Note that while we can choose to bind the `pgd` to the address of the associated `mm_struct`, this would not increase its security. As such, we achieve immutability of the page table pointer (❸).

We highlight that even with KPTI [51], [74] (the Meltdown mitigation feature of Linux that avoids simultaneously mapping user and kernel space), it is possible to authenticate `pgd` pointers. As KPTI employs two consecutive pages, with each mapping the root of page tables to user or kernel space, we always validate both pages by first normalizing the `pgd` to reference the first of the two pages. Lastly, a (Linux) kernel that leverages xMP to protect page tables does so in a transparent manner to user (and legacy) applications.

B. Protecting Process Credentials

Linux kernel credentials describe the properties of various objects that allow the kernel to enforce access control and capability management. This makes them an attractive target of data-oriented privilege escalation attacks.

Similarly to protecting paging structures, our goal is to prevent adversaries from (i) illegally overwriting process credentials in `struct cred` or (ii) redirecting the `cred` pointer in `task_struct` to an injected or existing `struct cred` instance with higher privileges. With the exception of reference counts and keyrings, once initialized and committed, process credentials do not change. Besides, a thread may only modify its own credentials and cannot alter the credentials of other threads. These properties establish inherent characteristics for security policies. In fact, Linux Security Modules (LSM) [79] introduce hooks at security-relevant locations that rely upon the aforementioned invariants. For instance, *SELinux* [80] and *AppArmor* [81] use these hooks to enforce Mandatory Access Control (MAC). Similarly, we combine our kernel memory protection primitives with *LSM* hooks to prevent adversaries from corrupting process credentials.

Linux prepares the slab cache `cred_jar` to maintain `cred` instances. By applying xMP to `cred_jar`, we ensure that adversaries cannot directly overwrite the contents of `cred` instances without first entering its xMP domain (§ V-B). As we check both the integrity and context of the active xMP domain index (`xmp_index_kernel`), adversaries cannot manipulate the system to enter an invalid domain (§ V-C). At the same time, we allow legitimate *write* access to `struct cred` instances, e.g., to maintain the number of subscribers; we guard such code sequences with primitives that enter and leave the xMP domain right before and after updating the data structures. Consequently, we meet requirements ❶ and ❷.

As every cred instance is uniquely assigned to a specific task, we bind the integrity of every cred pointer to the associated task_struct at process creation. We check both the integrity and the assigned context to the task_struct inside relevant LSM hooks. This ensures that every interaction related to access control between user and kernel space via system calls is granted access only to non-modified process credentials. Consequently, we eliminate unauthorized updates to cred instances without affecting normal operation (❸). Again, a kernel that uses xMP to harden process credentials does so in a completely transparent to existing applications.

C. Protecting Sensitive Process Data

An important factor for the deployment of security mechanisms is their applicability and generality. To highlight this property, we apply xMP to guard sensitive data in OpenSSL under Nginx, ssh-agent, mbed TLS, and libsodium. In each case, we minimally adjust the original memory allocation of the sensitive data to place them in individual pages, which are then assigned to xMP domains. Specifically, using the system calls introduced in § V-D, we grant *read-write* access to the xMP domain holding the sensitive data pages, to which remaining domains do not have any access. We further adjust authorized parts of the applications to enter the domain just before *reading* or *writing* the isolated data—any other access from outside the xMP domain crashes the application. In the following, we summarize the slight changes we made to the four applications for protecting sensitive data. Note that an xMP-enabled kernel is backwards compatible with applications that do not make use of our isolation primitives.

OpenSSL (Nginx): OpenSSL uses the BIGNUM data structure to manage prime numbers [82]. We add macros that allocate these structures into a separate xMP domain. Instrumenting the widely-used library OpenSSL allows to protect a wide range of applications. In our case, combining the modified OpenSSL with the Nginx web server (in HTTPS mode) offers protection against memory disclosure attacks, such as Heartbleed [3].

ssh-agent: To avoid repeatedly entering passphrases for encrypted private keys, users can use ssh-agent to keep private keys in memory, and use them for authentication when needed. This makes ssh-agent a target of memory disclosure attacks, aiming to steal the stored private keys. To prevent this, we modify the functions sshbuf_get_(c)string to safely store unencrypted keys in dedicated xMP domains.

mbed TLS: The mbed TLS library manages prime numbers and coefficients of type mbedtls_mpi in the mbedtls_rsa_context [83]. We define the new data structure secure_mbedtls_mpi and use it for the fields D, P, Q, DP, DQ, and QP in the mbedtls_rsa_context. We further adjust the secure_mbedtls_mpi initialization wrapper to isolate the prime numbers in an exclusive domain.

libsodium (minisign): The minimalistic and flexible libsodium library provides basic cryptographic services. By only adjusting the library's allocation functionality in sodium_malloc [84], we enable tools such as minisign to safely store sensitive information in xMP domains.

Table I
PERFORMANCE OVERHEAD OF XMP DOMAINS FOR PAGE TABLES, PROCESS CREDENTIALS, AND BOTH, MEASURED USING LMBENCH V3.0.

	Benchmark	PT	Cred	PT+Cred
Latency	syscall()	0.42%	0.64%	0.64%
	open()/close()	1.52%	75.74%	78.93%
	read()/write()	0.52%	150.84%	149.27%
	select() (10 fds)	2.94%	3.83%	3.83%
	select() (100 fds)	0.01%	0.31%	0.30%
	stat()	-1.22%	52.10%	53.33%
	fstat()	0.00%	107.69%	107.69%
	fork()+execve()	250.04%	9.36%	259.59%
	fork()+exit()	461.20%	7.78%	437.31%
	fork()+/bin/sh	236.75%	8.49%	240.64%
	sigaction()	10.00%	3.30%	10.00%
	Signal delivery	0.00%	2.12%	2.12%
	Protection fault	1.33%	-4.53%	-1.15%
	Page fault	216.21%	-2.58%	216.56%
	Pipe I/O	17.50%	32.87%	73.47%
	UNIX socket I/O	1.16%	1.45%	2.25%
	TCP socket I/O	10.23%	20.71%	37.13%
	UDP socket I/O	13.42%	21.98%	41.48%
Bandwidth	Pipe I/O	7.39%	7.09%	17.49%
	UNIX socket I/O	0.10%	6.61%	13.40%
	TCP socket I/O	6.89%	5.83%	14.53%
	mmap() I/O	1.22%	-0.53%	0.83%
	File I/O	0.00%	2.78%	2.78%

VII. EVALUATION

A. System Setup

Our setup consists of an unprivileged domain DomU running the Linux kernel v4.18 on top of the Xen hypervisor v4.12. In addition, we adjusted the Xen altp2m subsystem so that it is used from inside guest VMs, as described in § IV and § V. The host is equipped with an 8-core 3.6GHz Intel Skylake Core i7-7700 CPU, and 2GB of RAM available to DomU. Although we hardened the unprivileged domain DomU, the setup is not specific to unprivileged domains and can be equally applied to privileged domains, such as Dom0.

B. Performance Evaluation

To evaluate the performance impact of xMP we conducted two rounds of experiments, focusing on the overhead incurred by protecting sensitive data in kernel and user space. All reported results correspond to vanilla Linux vs. xMP-enabled Linux (both running as DomU VMs), and are means over 10 runs. Note that the virtualization overhead of Xen is negligible [39] and is therefore disregarded in our setting.

1) Kernel Memory Isolation: We measured the performance impact of xMP when applied to protect the kernel's *page tables* (PT) and *process credentials* (Cred) (§ VI-A and § VI-B). We used a set of micro (LMbench v3.0) and macro (Phoronix v8.6.0) benchmarks to stress different system components, and measured the overhead of protecting (i) each data structure individually, and (ii) both data structures at the same time (which requires two disjoint xMP domains).

Table I shows the LMbench results, focusing on *latency* and *bandwidth* overhead. This allows us to get some insight on the performance cost at the system software level. Overall, the overhead is low in most cases for both protected page

Table II
PERFORMANCE OVERHEAD OF XMP DOMAINS FOR PAGE TABLES, PROCESS CREDENTIALS, AND BOTH, MEASURED USING PHORONIX V8.6.0.

	Benchmark	PT	Cred	PT+Cred
Stress Tests	AIO-Stress	0.15%	5.87%	5.99%
	Dbench	0.43%	4.74%	3.45%
	IOzone (R)	-4.64%	26.9%	24.2%
	IOzone (W)	0.82%	4.43%	7.71%
	PostMark	0.00%	7.52%	7.52%
	Thr. I/O (Rand. R)	2.92%	7.58%	10.13%
	Thr. I/O (Rand. W)	-5.35%	3.01%	-1.29%
	Thr. I/O (R)	-1.06%	19.54%	20.08%
	Thr. I/O (W)	1.34%	-1.61%	-0.27%
Applications	Apache	6.59%	9.33%	11.14%
	FFmpeg	0.14%	0.43%	0.00%
	GnuPG	-0.66%	-1.31%	-2.13%
	Kernel build	11.54%	1.84%	12.71%
	Kernel extract	2.89%	3.65%	5.91%
	OpenSSL	-0.33%	-0.66%	0.99%
	PostgreSQL	4.12%	0.32%	4.43%
	SQLite	1.10%	-0.93%	-0.57%
	7-Zip	-0.30%	0.26%	0.08%

tables and process credentials. When protecting page tables, we notice that the performance impact is directly related to functionality that requires explicit access to page tables, with outliers related to page faults and process creation (`fork()`). Contrary to page tables, we observe that although the kernel accesses the `struct cred` xMP domain when creating new processes, the overhead is insignificant. On the other hand, the xMP domain guarding process credentials is heavily used during file operations, which require access to `struct cred` for access control. The performance impact of the two xMP domains behaves additively in the combined setup (PT+Cred).

To investigate the cause of the performance drop for the outliers (UNIX socket I/O, `fstat()`, and `read()/write()`), we used the eBPF tracing tools [85]. We applied the `funccount` and `funclatency` tools while executing the outlier test cases to determine the hotspots causing the performance drop by extracting the exact number and latency of kernel function invocations. We confirmed that, in contrast to benchmarks with a lower overhead, the outliers call the instrumented LSM hooks [79] more frequently. In particular, the function `apparmor_file_permission` [81] is invoked by every file-related system call. (This function is related to AppArmor, which is enabled in our `DomU` kernel.) In this function, even before verifying file permissions, we validate the context-bound integrity of a given pointer to the process' credentials. Although this check is not limited to this function, it is performed by every system call in those benchmarks and dominates the number of calls to all other instrumented kernel functions. For every pointer authentication, this function triggers the xMP domain to access the secret key required to authenticate the respective pointer. To measure the time required for this recurring sequence, we used the `funclatency` (eBPF) tool. The added overhead of this sequence ranges between 0.5–1 *µsec*. An additional 0.5–4 *µsec* is required for entering the active xMP domain on

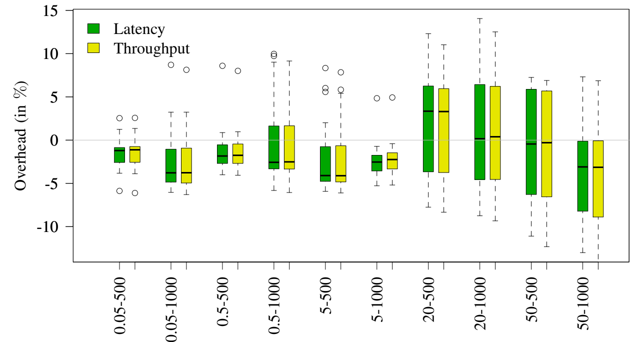

Figure 5. Performance impact of xMP on Nginx with varying file sizes and number of connections (X-axis: [file size (KB)]-[# requests]).

every context switch—including switches between user and kernel space on system calls. Consequently, the context-bound integrity checks affect the performance of light-weight system calls, e.g., `read()` or `write()`, in a more evident way than system calls with higher execution times or even without any file access checks. Having identified the hotspot locations, we can focus on optimizing performance in the future.

Table II presents the results for the set of Phoronix macrobenchmarks used by the Linux kernel developers to track performance regressions. The respective benchmarks are split into *stress tests*, targeting one specific system component, and *real-world applications*. Overall, with only a few exceptions, the results show that xMP incurs low performance overhead, especially for page table protection. Specifically, we observe a striking difference between the *read* (R) and *write* (W) Threaded I/O tests: while the `pwrite()` system call is hardly affected by xMP, there is a noticeable performance drop for `pread()`. Using the eBPF tracing tools, we found that the reason for this difference is that the default benchmark settings *synchronize* `pwrite()` operations. By passing the `O_SYNC` flag to the `open()` system call, `pwrite()` returns only after the data has been written to disk. Thus, compared to `pread()`, which completes after 1–2 *µsec*, `pwrite()` requires 2–8 *msec*, and the added overhead of `apparmor_file_permission` accumulates and does not affect `pwrite()` as much as it affects `pread()`.

2) In-Process Memory Isolation: We evaluated the overhead of in-process memory isolation using our xMP-protected versions of the Nginx and mbed TLS servers (§ VI-C). In both cases, we used the server benchmarking tool `ab` [86] to simulate 20 clients, each sending 500 and 1,000 requests. To compare our results with related work, we run the Nginx benchmarks with the same configuration used by SeCage [26]. The throughput and latency overhead is negligible in most cases (Figure 5). Contrary to SeCage, which incurs up to 40% overhead for connections without KeepAlive headers and additional TLS establishment, xMP does not suffer from similar issues in such challenging configurations, even with small files. The average overhead for latency and throughput is 0.5%. For mbed TLS, we used the `ssl_server` example [87] to

execute an SSL server hosting a 50-byte file. (We chose a small file to not mask the overhead with I/O.) On average, the overhead is 0.42% for latency and 1.14% for throughput.

C. Scalability of xMP Domains

Hardware-based memory isolation features, similar to xMP, support only a small number of domains. For instance, Intel MPK and ARM Domain Access Control (DAC) implement only 16 domains. Nevertheless, we investigate scenarios in which a high number of domains becomes necessary. Modern infrastructures massively deploy OS-level virtualization (i.e., containers), for which Linux namespaces [88] provide an essential building block by establishing different views on selected global system resources. By integrating xMP into Linux namespaces to isolate selected system resources (§ VI), we establish (i) the foundation for hypervisor-assisted OS-level virtualization, and (ii) the means to evaluate the scalability of xMP domains.

To that end, we introduce *xMP namespaces* to isolate process page tables. (Note that xMP namespaces can be extended to isolate arbitrary data structures.) Specifically, we use the `unshare()` system call to move a process into a new xMP namespace, effectively placing the process' page tables into a new xMP domain—the process' descendants inherit their parent's xMP namespace, effectively protecting their page tables as well. Page tables of processes belonging to different namespaces are isolated by different xMP domains, preventing compromised containers from modifying page tables of containers belonging to different xMP namespaces.

To measure the impact of an increasing number of xMP domains, we customized the Phoronix *Hackbench* scheduler stress test. Our adjustments cause the benchmark to place groups of 10 processes each (five senders and five receivers exchanging 50K messages) into separate xMP namespaces. In its standard configuration, Xen supports up to 10 `altp2m` views, with only eight of them being used for xMP domains—`altp2m[0]` is a mirror of the hosts's original view, and must not be changed, and `altp2m[1]` is the restricted view (§ IV-A). We extended Xen so that we can create up to 256 `altp2m` views; recall that this limitation stems from the fact that we encode the xMP domain's index into the `GFP` allocation flags using eight unused bits (§ V-A).

We compare the overhead of an xMP-capable Linux kernel with a vanilla one. Figure 6 shows the scheduling overhead of up to 250 distinct xMP namespaces. (Again, results are means over 10 runs.) Overall, the isolation overhead accumulates linearly with the number of xMP domains—each domain contains the page tables of 10 processes. However, by increasing the number of processes (250 xMP domains correspond to 2.5K processes), the time required to schedule and run each stress test (i.e., 10 processes exchanging 50K messages) amortizes the overhead, which can even drop to about 2%. Further, this experiment presents the ability of our prototype to scale up to 250 distinct isolation domains, an order of magnitude more than what can be achieved by existing schemes, like Intel MPK and ARM DAC (16 domains).

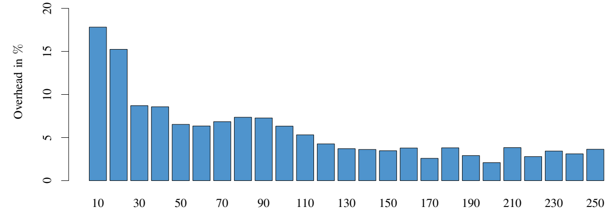

Figure 6. Performance impact of up to 250 xMP domains on the scheduler, measured using the customized Phoronix Hackbench stress test.

Lastly, note that in the experiment above, page tables are assigned to isolated xMP domains during process creation, but are populated while the benchmark is executing, due to copy-on-write and dynamic memory allocations. Therefore, the experiment also captures the management overhead of our prototype when it dynamically propagates changes to the corresponding restricted domain views.

D. Security Evaluation

We evaluated the security of our memory protection primitives using real-world exploits against (i) page tables, (ii) process credentials, and (iii) sensitive data in user space. Despite a strong attacker with arbitrary *read* and *write* primitives to kernel and user memory, by meeting the requirements ❶-❸, our system blocks illegal accesses to sensitive data.

1) Attacking the Kernel: We assume an attacker who aims to elevate their privilege using an arbitrary *read* and *write* primitive in kernel memory. To evaluate this scenario, we used a combination of real-world exploits that achieve the aforementioned capability. We first reconstructed an exploit to bypass KASLR [76]. The `task_struct` of the first process (`init_task`) has a fixed offset to the kernel's base address and is linked to all processes on the system. This provided us with the ability to locate sensitive management information about individual processes, including the root of the page table hierarchy and process credentials. We then abused CVE-2017-16995 (i.e., a sign-extension vulnerability in BPF) to gain an arbitrary read-write primitive to kernel memory.

In the next step, we implemented two different attacks that target (i) the page tables and (ii) the credentials of a given process, respectively. In the first attack, we used the write primitive to modify individual *page table entries* of the target process. This allowed us to grant the *write* permission to (an otherwise *execute-only* mapped) kernel code page with a rarely used system call handler, which is overwritten with shellcode that disables SMEP and SMAP in the `CR4` register. This lends the attacker the power to inject arbitrary code and data into kernel memory. In the second attack, we exchanged the `cred` pointer in the malicious process' `task_struct` with a pointer to an existing `struct cred` instance with higher privileges. In both attacks, we were able to elevate the privileges of the malicious process. By applying xMP to protect page tables and process credentials (§ VI-A and VI-B), we were able to successfully block both attack scenarios.

To systematically evaluate xMP, we consider attacks that can be equally applied to all kernel structures. We generalize the attack vectors against sensitive kernel structures in the following strategies. Under our threat model, attackers can:

- directly modify the data structure(s) of interest;
- redirect a pointer of the targeted data structure to an injected, attacker-controlled instance;
- redirect a pointer of the targeted data structure to an existing instance with higher privileges.

xMP withstands modification attempts of the protected data structures (❶-❷), as only authorized kernel code can enter the associated xMP domains. For instance, when protecting *page tables*, without first hijacking the kernel's execution, the attacker reaches an impasse on how to modify page tables isolated in xMP domains. Injecting code is thus prevented in the first place. Alternatively, the attacker can modify a thread's *pointer* to a sensitive data structure. In this case, the modified value must comply with the added context-bound integrity (❸) that is enforced on every context-switch or right before accessing the sensitive data structure (§ IV-C). Since attackers do not know the secret key, they cannot compute an HMAC that would validate the pointer's integrity. Consequently, attackers cannot redirect the pointer to an *injected* data structure.

To sidestep the secret key, attackers could overwrite the pointer with an *existing* pointer (holding a valid HMAC) to a data structure instance with higher privileges. Yet, as pointers to xMP-protected data are bound to the thread's context (❸), attackers cannot redirect pointers to instances belonging to other threads. Note that attackers would have to overwrite the `pgd` pointer of a *privileged* thread with the `pgd` pointer of an *attacker-controlled* thread, when targeting page tables.

2) Attacking User Applications: We chose Heartbleed [3] as a representative data leakage attack due to its high impact. As a result of the lack of a bounds check of the attacker-controlled `payload_length` field of OpenSSL's `HeartbeatMessage`, the attacker can reveal up to 64KB of linear memory that may hold private keys, passwords, and other sensitive information, without altering the application's control flow. By equipping the vulnerable OpenSSL library with the ability to guard secret material (§ VI-C), we prevented the sensitive regions from leaking. Illegal accesses caused an EPT violation that trapped into the `#VE` handler, which reported the illegal access and terminated the application.

3) Attacking Protection Primitives: Our user-space API does not use the `VMFUNC` instruction, but instead relies on a new system call (§ V-D). Given that `VMFUNC` is an *unprivileged* instruction, an attacker can still use it in an attempt to enter different xMP domains. Even if an attacker introduced a `VMFUNC` instruction in the application's memory to mount a `VMFUNC` faking attack [26], the next context switch would restore the xMP domain's state from `xmp_index_[kernel|user]`, making the kernel immune to illegal domain switches from user space. The attacker could try to use a *write* primitive to modify the kernel's xMP domain state in `xmp_index_[kernel|user]`, forcing the kernel to enter a privileged domain and grant access to sensitive data on the next context switch. Yet, as we bind the integrity of the active xMP domain state to the associated thread's context, any attempt to tamper with it will crash the process.

Further, mediating the execution of `VMFUNC` instructions through the `sys_xmp_enter` system call introduces gadgets that allow switching to previously-allocated xMP domains. Nevertheless, to perform such attacks, the attacker will need to change the application's control flow, something that we assume to be thwarted by orthogonal defenses (§ III).

4) I/O Attacks: Compromised I/O devices or drivers can access memory that holds sensitive data. To address this threat, the VMM should confine device-accessible memory (i) by employing the system's IOMMU (e.g., Intel VT-d [89]) or (ii) by means of SLAT. The former strategy ensures that sensitive memory in one of the xMP domains will not be mapped by the translation tables of the IOMMU; sensitive data structures become inaccessible to devices. In the latter approach, without IOMMU, the guest is likely to use bounce buffers (e.g., in combination with Virtio [90]) or directly access the devices. In both cases, a corrupted device or driver would access guest-virtual addresses, which are regulated by Xen's `altp2m` subsystem. Thus, it becomes impossible to leak or modify protected information, without first having to gain arbitrary code execution capabilities in kernel mode.

VIII. Discussion

A. Limitations

The Linux callback-free RCU feature [91] relocates the processing of RCU callbacks out of the `softirq` context, into a dedicated thread (§ V-C3). This allows RCU callbacks to enter xMP domains without affecting other threads' xMP domain state, as we currently do not provide selective memory protection in IRQ contexts.

Currently, we manually instruct the kernel when to enter a specific xMP domain. Instead, we could automate this step by instructing the compiler to bind annotated data structures to xMP domains. In addition, the compiler could instrument kernel code with calls that enter/leave the xMP domain immediately before/after accessing the annotated data structure.

Also, we do not support nested xMP domains. In fact, we prohibit entering domains, without first closing the active domain; by nesting xMP domains, the state of the opened domain will be overwritten. To address this, the kernel needs to securely keep track of the previously opened xMP domains by maintaining a stack of xMP domain states per thread. Note that this relates to adding xMP support in IRQ contexts.

B. Intel Sub-Page Write Permission

Intel announced the Sub-Page Write-Permission (SPP) feature for EPTs [35] to enforce memory *write* protection on sub-page granularity. Specifically, with SPP, Intel extends the EPT with an additional set of SPP tables that determine whether a 128-byte *sub-page* can be accessed. Selected 4KB guest page frames with restricted *write* permissions in the EPT can be configured to subsequently walk the SPP table to determine whether or not the accessed 128-byte block can be written.

Once this feature is implemented in hardware, it will enrich xMP in terms of performance and granularity. Let us consider the use case of protecting process credentials. Once initialized, the credentials themselves become immutable. However, meta information, such as reference counters, must be updated throughout the lifetime of the `cred` instance. This requires to first enter the xMP domain and relax the permissions to the otherwise *read-only* credentials, before updating the metadata. Using SPP, we can arrange `struct cred` so that all metadata is placed into writable sub-pages, despite the memory access restrictions of the xMP domain.

C. Execute-Only Memory

A corollary of the lack of *non-readable* memory (§ II-A) is that the x86 MMU does not support *execute-only* memory—code pages have to be *readable* as well. This has allowed adversaries to mount Just-In-Time ROP (JIT-ROP) attacks [73], which can bypass code randomization defenses. By reading code pages, an attacker can harvest ROP gadgets and construct a suitable payload on the fly. A defense against JIT-ROP attacks is thus to enforce execute-only memory to prevent the gadget harvesting phase [9], [31], [32], [92]. By defining execute-only xMP domains for code pages, xMP can offer similar protection.

D. Alternative Hypervisors and Architectures

Xen is by no means the only system on which xMP can be integrated. Other hypervisors that implement (or can be extended with [93]) similar functionality to Xen's `altp2m` can be equally used. Similarly, xMP does not dependent on Intel CPUs, as it does neither require hardware-supported EPTP switching nor the in-guest `#VE` feature—maintaining and switching among different views can be done in software. This would also relax Intel's restriction with respect to the maximum number of EPTPs, as the number of views would not be bound to hardware capabilities. For instance, at the risk of sacrificing performance, we could port xMP to Xen `altp2m` on ARM [30], in which `altp2m` does not rely on hardware support. On ARM, xMP would also benefit from PAC [65] for implementing context-bound pointer integrity.

IX. Related Work

While the possibility of non-control data (or data-oriented) attacks has been identified before [1], Chen et al. [2] were the first to demonstrate the viability of data-oriented attacks in real-world scenarios, ultimately rendering them as realistic threats. With FLOWSTITCH [94], Hu et al. introduced a tool that is capable of chaining, or rather stitching together, different data-flows to generate data-oriented attacks on Linux and Windows binaries, despite fine-grained CFI, DEP, and, in some cases, ASLR, in place. Hu et al. [5] further show that data-oriented attacks are in fact Turing-complete. They introduce Data-Oriented Programming (DOP), a technique for systematically generating data-oriented exploits for arbitrary x86-based programs. Similarly, Carlini et al. [4] achieve Turing-complete computation by using a technique they refer to as Control Flow Bending (CFB). In contrast to DOP, CFB is a hybrid approach that relies on the the modification of code pointers. Still, CFB bypasses common CFI mechanisms, by limiting code pointer modifications in a way that the modified control-flows comply with CFI policies. Ispoglou et al. [7] extend the concept of DOP by introducing a new technique they coin as Block-Oriented Programming (BOP). Their framework automatically locates dispatching basic blocks, in binaries that facilitate the chaining of *block-oriented gadgets*, which are then chained together to mount a successful attack.

On the other hand, researchers have started to respond to data-oriented attacks. For instance, DataShield [95] associates annotated data types with security sensitive information. Based on these annotations, DataShield partitions the application's memory into two disjoint regions, and inserts bounds checks that prevent illegal data flows between the sensitive and non-sensitive memory regions. Similar to our work, solutions based on virtualization maintain sensitive information in disjoint memory views [26], [27], [96]. While MemSentry [27] isolates sensitive data, SeCage [26] additionally identifies and places sensitive code into a secret compartment. Both frameworks leverage Intel's *EPTP switching* to switch between the secure compartment and the remaining application code. Yet, in contrast to our work, MemSentry and SeCage are limited to user space. Also, SeCage adds complexity by duplicating and modifying code that would normally be shared (e.g., libraries) between the secret and non-secret compartments.

EPTI [96] implements an alternative to KPTI using memory isolation techniques similar to xMP. PrivWatcher [28] leverages virtualization to ensure the integrity of process credentials. Contrary to our solution, PrivWatcher creates shadow copies of `struct cred` instances, and places them in a *write*-protected region. PT-Rand [76] protects page tables using information hiding. Zabrocki introduces LKRG [97], a runtime guard for the Linux kernel, ensuring the integrity of critical kernel components by shadowing selected data structures or matching their hashes in a database. Although, LKRG does not employ virtualization, the author considers to use a VMM for self-protection. Finally, with PARTS [66], Liljestrand et al. introduce a compiler instrumentation framework to cope with pointer-reuse attacks via the (recently-introduced) ARMv8.3-A pointer authentication features.

X. Conclusion

In this paper we propose novel defenses against data-oriented attacks. Our system, called xMP, leverages Intel's virtualization extensions to set the ground for selective memory isolation primitives, which facilitate the protection of sensitive data structures in both kernel and user space. We further equip pointers to data in isolated memory with authentication codes to thwart illegal pointer redirections. We demonstrate the effectiveness of our scheme by protecting the page tables and process credentials in the Linux kernel, as well as sensitive data in various user applications. We believe that our results demonstrate that xMP is a powerful and practical solution against data-oriented attacks.

Acknowledgment

We thank Christopher Roemheld and Joseph Macaluso for helping us with the Linux kernel extensions and the use cases regarding user applications, respectively. Further, we thank our shepherd, Yuval Yarom, and the anonymous reviewers for their valuable feedback. This work was supported in part by the European Union's Horizon 2020 research and innovation programme, under grant agreement No 830892 (SPARTA), the Office of Naval Research (ONR), through awards N00014-17-1-2891 and N00014-17-1-2788, and the National Science Foundation (NSF), through award CNS-1749895. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the aforementioned supporters.

References

[1] W. D. Young and J. McHugh, "Coding for a Believable Specification to Implementation Mapping," in *IEEE Symposium on Security and Privacy (S&P)*, 1987.

[2] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-Control-Data Attacks Are Realistic Threats," in *USENIX Security Symposium (SEC)*, 2005.

[3] Synopsys, "The Heartbleed Bug," http://heartbleed.com/, 4 2014.

[4] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-Flow Bending: On the Effectiveness of Control-Flow Integrity," in *USENIX Security Symposium (SEC)*, 2015.

[5] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks," in *IEEE Symposium on Security and Privacy (S&P)*, 2016.

[6] B. Sun, C. Xu, and S. Chong, "The Power of Data-Oriented Attacks," *Black Hat, Asia*, 2017.

[7] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer, "Block Oriented Programming: Automating Data-Only Attacks," in *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[8] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-Flow Integrity," in *ACM Conference on Computer and Communications Security (CCS)*, 2005.

[9] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny, "You Can Run but You Can'T Read: Preventing Disclosure Exploits in Executable Code," in *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[10] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, and R. H. Deng, "ROPecker: A Generic and Practical Approach for Defending Against ROP Attack," in *Network and Distributed System Security Symposium (NDSS)*, 2014.

[11] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières, "CCFI: Cryptographically Enforced Control Flow Integrity," in *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[12] J. Werner, G. Baltas, R. Dallara, N. Otterness, K. Z. Snow, F. Monrose, and M. Polychronakis, "No-Execute-After-Read: Preventing Code Disclosure in Commodity Software," in *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2016.

[13] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-Pointer Integrity," in *USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2014.

[14] P. Zieris and J. Horsch, "A Leak-Resilient Dual Stack Scheme for Backward-Edge Control-Flow Integrity," in *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2018.

[15] PaX Team, "Address Space Layout Randomization (ASLR)," https://pax.grsecurity.net/docs/aslr.txt, March 2003.

[16] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz, "Readactor: Practical Code Randomization Resilient to Memory Disclosure," in *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[17] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi, "Timely Rerandomization for Mitigating Memory Disclosures," in *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[18] Microsoft, "Control Flow Guard," https://docs.microsoft.com/en-us/windows/desktop/SecBP/control-flow-guard, October 2018.

[19] Google, "The Chromium Projects," https://www.chromium.org/developers/testing/control-flow-integrity, October 2018.

[20] LLVM, "Control Flow Integrity," http://clang.llvm.org/docs/ControlFlowIntegrity.html, October 2018.

[21] M. Castro, M. Costa, and T. Harris, "Securing Software by Enforcing Data-Flow Integrity," in *USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2006.

[22] G. C. Necula, S. McPeak, and W. Weimer, "CCured: Type-Safe Retrofitting of Legacy Code," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2002.

[23] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, "Cyclone: A Safe Dialect of C," in *USENIX Annual Technical Conference (ATC)*, 2002.

[24] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "SoftBound: Highly Compatible and Complete Spatial Memory Safety for C," *ACM SIGPLAN Notices*, 2009.

[25] ——, "CETS: compiler enforced temporal safety for C," *ACM SIGPLAN Notices*, 2010.

[26] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia, "Thwarting Memory Disclosure with Efficient Hypervisor-Enforced Intra-Domain Isolation," in *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[27] K. Koning, X. Chen, H. Bos, C. Giuffrida, and E. Athanasopoulos, "No Need to Hide: Protecting Safe Regions on Commodity Hardware," in *ACM European Conference on Computer Systems (EuroSys)*, 2017.

[28] Q. Chen, A. M. Azab, G. Ganesh, and P. Ning, "PrivWatcher: Non-bypassable Monitoring and Protection of Process Credentials from Memory Corruption Attacks," in *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2017.

[29] T. K. Lengyel, S. Maresca, B. D. Payne, G. D. Webster, S. Vogl, and A. Kiayias, "Scalability, Fidelity and Stealth in the DRAKVUF Dynamic Malware Analysis System," in *Annual Computer Security Applications Conference (ACSAC)*, 2014.

[30] S. Proskurin, T. Lengyel, M. Momeu, C. Eckert, and A. Zarras, "Hiding in the Shadows: Empowering ARM for Stealthy Virtual Machine Introspection," in *Annual Computer Security Applications Conference (ACSAC)*, 2018.

[31] M. Pomonis, T. Petsios, A. D. Keromytis, M. Polychronakis, and V. P. Kemerlis, "kR^X: Comprehensive Kernel Protection against Just-In-Time Code Reuse," in *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2017.

[32] J. Gionta, W. Enck, and P. Ning, "HideM: Protecting the Contents of Userspace Memory in the Face of Disclosure Vulnerabilities," in *ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2015.

[33] S. Brookes, R. Denz, M. Osterloh, and S. Taylor, "ExOShim: Preventing Memory Disclosure using Execute-Only Kernel Code," in *International Conference on Cyber Warfare and Security (ICCWS)*, 2016.

[34] J. Corbet, "Memory Protection Keys," https://lwn.net/Articles/643797/, May 2015.

[35] Intel Corporation, *Intel 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4*, 2019.

[36] C. A. Waldspurger, "Memory Resource Management in VMware ESX Server," *ACM SIGOPS Operating Systems Review (OSR)*, 2002.

[37] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," *ACM SIGOPS Operating Systems Review (OSR)*, 2003.

[38] Z. Deng, X. Zhang, and D. Xu, "SPIDER: Stealthy Binary Program Instrumentation and Debugging via Hardware Virtualization," in *Annual Computer Security Applications Conference (ACSAC)*, 2013.

[39] S. Proskurin, J. Kirsch, and A. Zarras, "Follow the WhiteRabbit: Towards Consolidation of On-the-Fly Virtualization and Virtual Machine Introspection," in *IFIP International Conference on ICT Systems Security and Privacy Protection (IFIP SEC)*, 2018.

[40] Linux Foundation, "Xen Project," https://www.xenproject.org/, 2018.

[41] B. Shi, L. Cui, B. Li, X. Liu, Z. Hao, and H. Shen, "ShadowMonitor: An Effective In-VM Monitoring Framework with Hardware-Enforced Isolation," in *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2018.

[42] Xen-devel, "Alternate `p2m` design specification," https://lists.xenproject.org/archives/html/xen-devel/2015-06/msg01319.html, September 2015.

[43] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering Code-Injection Attacks with Instruction-Set Randomization," in *ACM Conference on Computer and Communications Security (CCS)*, 2003.
[44] S. Liakh, "NX Protection for Kernel Data," https://lwn.net/Articles/342266/, July 2009.
[45] J. Edge, "Kernel Address Space Layout Randomization," https://lwn.net/Articles/569635/, October 2013.
[46] J. Corbet, "x86 NX Support," https://lwn.net/Articles/87814/, June 2004.
[47] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis, "kGuard: Lightweight Kernel Protection against Return-to-user Attacks," in *USENIX Security Symposium (SEC)*, 2012.
[48] F. Yu, "Enable/Disable Supervisor Mode Execution Protection," https://goo.gl/utKHno, May 2011.
[49] J. Corbet, "Supervisor Mode Access Prevention," https://lwn.net/Articles/517475/, October 2012.
[50] ——, "The Current State of Kernel Page-Table Isolation," https://lwn.net/Articles/741878/, December 2017.
[51] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard, "KASLR is Dead: Long Live KASLR," in *Engineering Secure Software and Systems (ESSoS)*, 2017.
[52] A. van de Ven, "Add `-fstack-protector` Support to the Kernel," https://lwn.net/Articles/193307/, July 2006.
[53] Open Web Application Security Project (OWASP), "C-Based Toolchain Hardening," https://www.owasp.org/index.php/C-Based_Toolchain_Hardening, January 2019.
[54] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications," in *IEEE Symposium on Security and Privacy (S&P)*, 2015.
[55] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, "Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity," in *ACM Conference on Computer and Communications Security (CCS)*, 2015.
[56] E. Göktas, B. Kollenda, P. Koppe, E. Bosman, G. Portokalidis, T. Holz, H. Bos, and C. Giuffrida, "Position-independent Code Reuse: On the Effectiveness of ASLR in the Absence of Information Disclosure," in *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2018.
[57] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-Flow Integrity: Precision, Security, and Performance," *ACM Computing Surveys (CSUR)*, 2017.
[58] X. Ge, N. Talele, M. Payer, and T. Jaeger, "Fine-Grained Control-Flow Integrity for Kernel Software," in *IEEE Symposium on Security and Privacy (S&P)*, 2016.
[59] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM," in *USENIX Security Symposium (SEC)*, 2014.
[60] J. Corbet, "Kernel Support for Control-Flow Enforcement," https://lwn.net/Articles/758245/, June 2018.
[61] Larsen, Per and Homescu, Andrei and Brunthaler, Stefan and Franz, Michael, "SoK: Automated Software Diversity," in *IEEE Symposium on Security and Privacy (S&P)*, 2014.
[62] H. Koo, Y. Chen, L. Lu, V. P. Kemerlis, and M. Polychronakis, "Compiler-Assisted Code Randomization," in *IEEE Symposium on Security and Privacy (S&P)*, 2018.
[63] M. Morton, J. Werner, P. Kintis, K. Snow, M. Antonakakis, M. Polychronakis, and F. Monrose, "Security Risks in Asynchronous Web Servers: When Performance Optimizations Amplify the Impact of Data-Oriented Attacks," in *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2018.
[64] H. Lee, C. Song, and B. B. Kang, "Lord of the x86 Rings: A Portable User Mode Privilege Separation Architecture on x86," in *ACM Conference on Computer and Communications Security (CCS)*, 2018.
[65] Qualcomm, "Pointer Authentication on ARMv8.3," https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf, January 2017.
[66] H. Liljestrand, T. Nyman, K. Wang, C. Chinea Perez, J. Ekberg, and N. Asokan, "PAC it up: Towards Pointer Integrity using ARM Pointer Authentication," in *USENIX Security Symposium (SEC)*, 2019.
[67] J.-P. Aumasson and D. J. Bernstein, "SipHash: a fast short-input PRF," in *International Conference on Cryptology in India (INDOCRYPT)*, 2012.
[68] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis, "ret2dir: Rethinking Kernel Isolation," in *USENIX Security Symposium (SEC)*, 2014.
[69] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*, 3rd ed. O'Reilly Media, 2005, ch. Memory Management, pp. 294–350.
[70] J. Bonwick, "The Slab Allocator: An Object-Caching Kernel Memory Allocator," in *USENIX Summer*, 1994.
[71] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*, 3rd ed. O'Reilly Media, 2005, ch. Interrupts and Exceptions, pp. 131–188.
[72] P. McKenney, "What is RCU, Fundamentally?" https://lwn.net/Articles/262464/, December 2007.
[73] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization," in *IEEE Symposium on Security and Privacy (S&P)*, 2013.
[74] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin *et al.*, "Meltdown: Reading Kernel Memory from User Space," in *USENIX Security Symposium (SEC)*, 2018.
[75] 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 *IEEE Symposium on Security and Privacy (S&P)*, 2019.
[76] L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi, "PT-Rand: Practical Mitigation of Data-only Attacks against Page Tables," in *Network and Distributed System Security Symposium (NDSS)*, 2017.
[77] J. Lee, H. Ham, I. Kim, and J. Song, "POSTER: Page Table Manipulation Attack," in *ACM Conference on Computer and Communications Security (CCS)*, 2015.
[78] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. Adve, "Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation," *ACM SIGPLAN Notices*, 2015.
[79] J. Morris, S. Smalley, and G. Kroah-Hartman, "Linux Security Modules: General Security Support for the Linux Kernel," in *USENIX Security Symposium (SEC)*, 2002.
[80] P. Loscocco and S. Smalley, "Integrating Flexible Support for Security Policies into the Linux Operating System," in *USENIX Annual Technical Conference (ATC)*, 2001.
[81] A. Gruenbacher and S. Arnold, "AppArmor Technical Documentation," http://lkml.iu.edu/hypermail/linux/kernel/0706.1/0805/techdoc.pdf, SUSE Labs / Novell, April 2007.
[82] OpenSSL, "OpenSSL Manpages v1.0.2," https://www.openssl.org/docs/man1.0.2/man3/bn.html, April 2019.
[83] ARM mbed, "mbed TLS v2.16.1 Source Code Documentation," https://tls.mbed.org/api, April 2019.
[84] Libsodium, "Libsodium Documentation," https://libsodium.gitbook.io/doc/memory_management, April 2019.
[85] M. Fleming, "A Thorough Introduction to eBPF," https://lwn.net/Articles/740157/, December 2017.
[86] Apache HTTP Server Project, "ab - Apache HTTP Server Benchmarking Tool," https://httpd.apache.org/docs/2.4/programs/ab.html, April 2019.
[87] ARM mbed, "mbed TLS Sample Programs," https://github.com/ARMmbed/mbedtls/blob/master/programs, April 2019.
[88] M. Kerrisk, "Namespaces in Operation, Part 1: Namespaces Overview," https://lwn.net/Articles/531114/, January 2013.
[89] Intel Corporation, *Intel Virtualization Technology for Directed I/O*, 2019.
[90] R. Russell, "Virtio: Towards a De-Facto Standard for Virtual I/O Devices," in *ACM SIGOPS Operating Systems Review (OSR)*, 2008.
[91] J. Corbet, "Relocating RCU Callbacks," https://lwn.net/Articles/522262/, October 2012.
[92] Y. Chen, D. Zhang, R. Wang, R. Qiao, A. M. Azab, L. Lu, H. Vijayakumar, and W. Shen, "NORAX: Enabling Execute-Only Memory for COTS Binaries on AArch64," in *IEEE Symposium on Security and Privacy (S&P)*, 2017.
[93] R. Quin and B. Kerrigan, "Bareflank Hypervisor," https://bareflank.github.io/hypervisor/, September 2019.
[94] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang, "Automatic Generation of Data-Oriented Exploits," in *USENIX Security Symposium (SEC)*, 2015.
[95] S. A. Carr and M. Payer, "Datashield: Configurable Data Confidentiality and Integrity," in *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2017.
[96] Z. Hua, D. Du, Y. Xia, H. Chen, and B. Zang, "EPTI: Efficient Defence against Meltdown Attack for Unpatched VMs," in *USENIX Annual Technical Conference (ATC)*, 2018.
[97] A. Zabrocki, "Linux Kernel Runtime Guard (LKRG) under the Hood," *CONFidence*, 2018.