xMP: Selective Memory Protection for Kernel and User Space

Anonymous Submission – #165

***Abstract*—Attackers leverage memory corruption vulnerabilities to establish primitives for *reading* from or *writing* to the address space of the 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. We implement xMP, which consists of (in-guest) *selective memory protection* primitives that equip VMs with the ability to isolate sensitive data in user or kernel space into disjoint protection 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 protection 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 fortify 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

Memory-safety attacks have only become better over time [1]: during the past three decades, data-oriented attacks [2] have evolved from a theoretical exercise [3] to serious threats [4]–[9]. Therefore, the everlasting race between attackers and defenders continues. In the past, we have witnessed a plethora of effective security mechanisms that urged attackers to investigate new directions and exploit less explored corners of various systems. Specifically, recent advances in Control-Flow Integrity (CFI) [10]–[14], Code-Pointer Integrity (CPI) [15], [16], and code diversification [17]–[19] have significantly raised the bar for code-reuse attacks. In fact, CFI mechanisms have been adopted by Microsoft [20], Google [21], and LLVM [22], forcing attackers to explore the uncharted world of data-oriented exploitation.

Code-reuse attacks chain short code sequences, dubbed *gadgets*, to hijack an application's control-flow. It is sufficient to overwrite one control-flow structure, such as a function pointer, or a return address on the stack, with the start of a crafted gadget chain, to cause a target application to perform arbitrary computation. In contrast, data-oriented attacks completely avoid changes to the control flow. Instead, this class of attacks aims to modify *non-control data* to cause the application to obey the attacker's intentions [7]–[9]. 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 on data only, 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 the presence of code-reuse mitigation techniques, and can have disastrous consequences [5]. We anticipate further growth in this direction, in the near future, and emphasize the need for practical primitives that eliminate such threats in advance.

Researchers have suggested different strategies to counter data-oriented attacks. For instance, Data-Flow Integrity (DFI) [23] schemes dynamically track a program's data flow. Similarly, by introducing memory safety to the C/C++ programming language, it becomes possible to completely eliminate memory corruption vulnerabilities [24]–[27]. 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. Therefore, instead of enforcing the integrity of the data flow, researchers have started exploring isolation techniques that govern access to sensitive code and data regions [28]–[30]. 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 modern virtualization extensions (available on 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 semantic 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 underlying hypervisor. For this, we extend Xen `altp2m` [31], [32] and the Linux memory management system to enable the selective protection of sensitive data in user or kernel space by isolating it in disjoint protection domains that are not subject to the limited access permissions of the x86 Memory Management Unit (MMU).[1] A powerful attacker with arbitrary *read* and *write* memory primitives cannot access the fortified data without first having to enter the corresponding protection domain. Furthermore, we equip in-kernel management information and pointers to sensitive data in protection domains with authentication codes, whose

[1] In this paper, we refer to both x86 and x86-64 as the x86 architecture.

integrity is bound to a specific context. This allows xMP to protect pointers from illegal modifications and hence obstruct data-oriented attacks that target the fortified data.

We use xMP to fortify two sensitive kernel data structures that are vital for the system's security, yet 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 contributions:

- We present techniques for combining Intel's EPTP switching and Xen `altp2m` to control different guest physical memory views, and isolate data into *disjoint protection domains*.
- We extend the Linux kernel to implement *xMP*, an in-guest selective memory isolation framework for the protection of sensitive data against data-oriented attacks in both user and kernel space.
- We apply xMP to guard *page tables* and *process credentials*, as well as sensitive data in user-space applications, with minimal performance overhead.

II. Background

Both user and kernel space maintain sensitive data whose integrity must be protected by all means. Memory corruption vulnerabilities facilitate arbitrary *read* and *write* primitives, which are key to privilege escalation and data disclosure via code-reuse and data-oriented attacks. Effective protection against the former kind of attacks is primarily provided by CFI [33], CPI [15], [16], and code diversification [34], [35]. Yet, there is no practical defense against data-oriented attacks.

In this section, we introduce Memory Protection Keys [36], Intel's extension to the x86 Instruction Set Architecture (ISA) for fine-grained memory isolation, which can challenge data-oriented attacks in user space. Furthermore, we present the Xen `altp2m` subsystem [31], [32] that leverages Intel's virtualization extensions and sets the ground for selective memory protection primitives that equip software developers with the ability to obstruct data-oriented attacks targeting both user and kernel space.

A. Memory Protection Keys

In 2015, Intel announced an extension to the x86 ISA, called Intel Memory Protection Keys (MPK), for establishing a means for hardening access to user space pages (i.e., memory pages with bit `U/S = 1` in the respective page table entries) [36], [37]. Intel MPK supplements the general paging mechanism by further restricting memory access permissions. In particular, each paging structure entry dedicates four bits that associate virtual memory pages with one of a total of 16 memory protection domains. A protection domain can be regarded as a set of pages whose access permissions are controlled by the same *protection key* (`PKEY`). User-space processes control access permissions of each `PKEY` through the 32-bit `PKRU` register. Specifically, MPK allows different memory protection keys to be simultaneously active, and page table entries to be paired with different keys that further restrict the memory permissions of the associated page. For each `PKEY`, the thread-local `PKRU` register holds two bits (*write disable* and *access disable*) that define the access permissions of the corresponding protection domain. Data accesses to protection domains are thus restricted by both the protection key and page table access permissions. Intel MPK allows threads to individually partition memory that belongs to their address space into 16 (at most) protection domains, and to freely constrain access to individual domains without affecting domains defined by other threads.

One of the main benefits of Intel MPK is that it allows user threads to independently and efficiently harden the permissions of potentially large memory regions. For instance, a thread can revoke *write* access from an entire protection domain, without switching to kernel space, walking and adjusting page table entries, and invalidating TLB entries to enforce the changes; instead, the thread achieves the desired behavior by only setting the *write disable* bit of the corresponding `PKEY` in the `PKRU` register. Another benefit of this technology is that it extends, or rather refines, access control capabilities of page tables on x86. Through Intel MPK, a thread can enforce (i) *execute-only* code pages [18], [38] or (ii) *non-readable*, yet *present* data pages [30] by setting the *access disable* bit of the associated protection key. Since the MMU on x86 lacks the ability to enforce such policies via page tables, mapped code and data pages can become subject to code-reuse [39] and data-oriented attacks [4], [5], [7], [40], [41] that result from memory disclosures. The introduced capabilities provide new primitives for thwarting data-oriented attacks, without sacrificing performance and practicality [11], or resorting to architectural quirks [42] and virtualization [14], [18], [43].

Although Intel announced MPK back in 2015 [37], the feature has been introduced only recently, and to just one CPU class dedicated to servers (Skylake-SP Xeon). Hence, the need for a similar isolation feature on desktop, mobile, and legacy server CPUs remains. Another issue is that attackers with the ability to arbitrarily corrupt kernel memory can (i) modify the per-thread state (in kernel space) holding access permissions of individual 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 Intel 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

Modern Virtual Machine Monitors (VMMs) leverage Second Level Address Translation (SLAT) to isolate physical memory that is reserved for VMs [36]. 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 entries, trap into the VMM [44], [45]. This lends VMMs strong memory isolation properties. As SLAT tables are solely maintained by the VMM, it has the privilege to fully control the VM's *view* on its physical memory [31], [32], [46], [47].

The *physical to machine* subsystem (`p2m`) of the Xen hypervisor [45], [48] employs SLAT to define the guest's *view* of the physical memory that is perceived by all virtual CPUs (vCPUs); modern virtualization solutions employ only one set of SLAT tables, as the static view on the physical memory changes only in rare situations. Still, by restricting access to individual page frames, security mechanisms can use Xen `p2m` to enforce memory access policies on the guest's physical memory. This allows protecting in-guest kernel memory (which may hold sensitive information) from illegal accesses.

Yet, 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 information on a specific page. By revoking *read* permissions in the SLAT tables, illegal *read* accesses to the protected page, e.g., originating through malicious memory disclosure attempts, would violate memory permissions and thus trap into the VMM. At the same time, for legal guest accesses to the protected region, the VMM has to temporarily relax permissions of the particular page frame; every time the guest requires access to the sensitive information, it has to instruct the VMM to walk the SLAT tables—an expensive operation. More importantly, by temporarily relaxing permissions in the global view, the VMM creates a window in which other vCPUs can freely access the sensitive data without notifying the VMM.

The Xen *alternate p2m* subsystem (`altp2m`) [31], [32] addresses the above issues. Instead of using a single, *global view*, Xen `altp2m` provides the necessary means to maintain and switch among *different views*. As the views can be assigned to each vCPU individually, access permissions in one view can be safely relaxed without affecting the active views of other vCPUs. In fact, instead of walking the SLAT tables to relax memory permissions, Xen `altp2m` allows switching to another, less restrictive view. Both external [31], [32] and internal monitors [28], [49] use the `altp2m` interface to allocate, switch, and destroy `altp2m` views. Although `altp2m` introduces a powerful means to rapidly change the guest's memory view, it requires additional hardware support to establish primitives that can be used by guests for isolating selected memory regions.

C. In-Guest EPT Management

The intention behind the Xen `altp2m` subsystem has been to add support for the Intel virtualization extension that allows VMs to switch among Extended Page Tables (EPTs).[2] Specifically, Intel introduced the unprivileged `VMFUNC` instruction to enable a VM to switch among different predefined EPTs without involving the VMM—although Xen `altp2m` has been implemented for Intel and ARM [32], in-guest switching of different `altp2m` views is available to Intel only. Intel uses a hardware-defined data structure, Virtual Machine Control Structure (VMCS), to maintain the host's and the VM's state per vCPU. Further, the VMCS defines 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; by using the `VMFUNC`, a guest can choose among 512 different 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.

[2]EPTs refer to Intel's implementation of SLAT tables [36].

III. Threat Model

We expect the system to be protected from code injection [50] through Data Execution Prevention (DEP) or other proper W^X policy enforcement, and to employ Address Space Layout Randomization (ASLR) both in kernel [51], [52] and user space [17], [53]. Also, we assume that the kernel is protected against return-to-user (ret2usr) [54] attacks by means of SMEP/SMAP [36], [55], [56]; other hardening features, such as Kernel Page Table Isolation (KPTI) [57], [58], stack smashing protection [59], and toolchain-based hardening [60], 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 [6], [61]–[63] via either (i) fine-grained CFI [33] (in kernel [64] and user space [65]), coupled with a shadow stack [66], or (ii) fine-grained code diversification [34], [35], and with execute-only memory (available to both kernel [38] and user space [18]).

Assuming the above state-of-the-art protections restrict an attacker's capability to achieve arbitrary code execution, we focus on defending against attacks that leak or modify sensitive data in user or kernel memory [18], [38], by transforming memory corruption vulnerabilities into *arbitrary read* and *write primitives*. Attackers can leverage such primitives to mount data-oriented attacks [9], [41] that (i) disclose sensitive information, such as an application's cryptographic material, or (ii) modify sensitive data structures, such as page tables or process credentials.

IV. Design

Memory corruption vulnerabilities form the foundation for data-oriented attacks. As such, there is a need for practical primitives that can protect sensitive data from such attacks. To that end, we identify the following requirements:

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

Although the x86 architecture allows for memory partitioning through segmentation or paging ❶, it lacks the fine-grained access control capabilities that are required 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 [67], isolation of kernel memory is primarily achieved by Software-Fault Isolation (SFI) solutions [38]. 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 protection domains by relaxing permissions inside the *global* page tables may reveal sensitive memory contents to the remaining CPUs. Besides, each memory access 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) [68] extension, there is no similar feature on x86. As such, the x86 architecture does not meet requirements ❷ and ❸.

In this work, we fill this gap by introducing *selected memory protection (xMP)* primitives that leverage virtualization to define efficient memory protection domains in both kernel and user space, enforce fine-grained memory permissions on selected protection domains, and protect the integrity of pointers to those domains (Figure 1). In contrast to common virtualization-based hardening frameworks that resort to additional logic introduced into the VMM to enforce the envisaged properties, xMP does not require any interaction with the VMM at run-time. In the following, we introduce xMP primitives and show how they can be used to build practical and effective defenses against real-world data-oriented attacks in both user and kernel space. We base xMP on top of x86 and Xen [48], as it relies on virtualization extensions that are exclusive to the Intel architecture and are already used by Xen.

A. Memory Partitioning through Protection Domains

To achieve meaningful protection, applications may require multiple disjoint *memory protection domains* that cannot be accessible at the same time. For instance, a protection domain that holds the kernel's hardware encryption key must not be accessible upon entering a protection 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 multiple disjoint protection domains (§ II-B). A protection 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 relaxed to enable legitimate access to the protected data as needed by applications or the kernel.

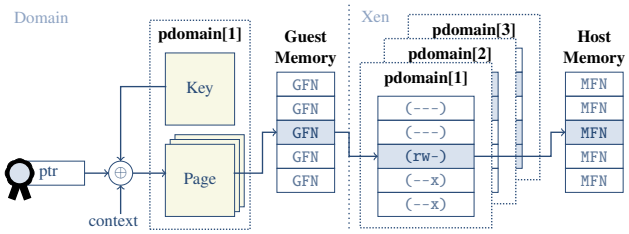

Figure 1. xMP uses different Xen `altp2m` views, each mapping guest frames to machine frames with different access permissions, to partition memory into isolated protection domains. By additionally equipping data pointers to protected memory with HMACs, we establish context-bound pointer integrity.

The straightforward approach of associating an `altp2m` view with each protection 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 protection domains in each `altp2m` view, we propagate the permissions of each domain across all available `altp2m` views. Setting up a protection domain requires at least two `altp2m` views. Regardless of the number of protection domains, we dedicate one view, the *restricted view*, to unify the memory access restrictions of all protection domains. We configure this view as the default on every vCPU, as it collectively enforces the restrictions of all protection domains. We use the second view to *relax* the restrictions of (i.e., *unprotect*) a given protection domain and to allow legitimate access to its data. We refer to this view as *pdomain[id]*, with *id* referring to the protection domain of this view. By entering *pdomain[id]*, the system switches to the `altp2m` view *id* to bring the protection domain into its relaxed state—crucially, all other protection domains remain in their protected state. By switching to the *restricted view*, the system switches *all* domains to their protected state.

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

An alternative approach to using `altp2m` would be to leverage Intel MPK (§ II-A). Although MPK meets requirements ❶ and ❷, unfortunately it is applicable only to user-space applications, and it cannot withstand abuse by memory corruption vulnerabilities targeting the kernel. Furthermore, even when focusing only on user-space protection using MPK,

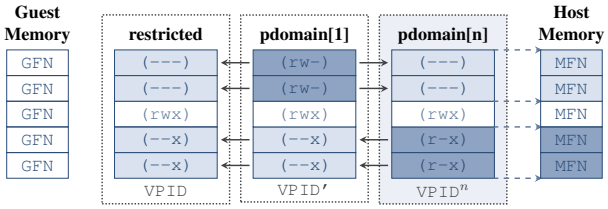

Figure 2. The system configures $n + 1$ `altp2m` views to create n disjoint protection domains. Each $\{pdomain[i] \mid i \in \{1, ..., n\}\}$ relaxes the permissions of a given memory region (dark blue) and restricts access to memory regions belonging to other protection domains (light blue).

attackers can still disclose sensitive data in multi-threaded environments. Given that threads share the same page tables, a controlled memory corruption vulnerability in a malicious thread can gain access to protected data as soon as another benign thread relaxes the permissions of a protection domain. 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 is applicable for the protection of both user- and kernel-space memory.

B. Isolation of Protection 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 extend the `altp2m` interface in Linux to release the full potential of `altp2m` for use from inside guest VMs. Specifically, we introduce hypercalls that trigger the VMM to configure page frames with requested access permissions on the VM's behalf. We further 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 reduces the number of 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 a scenario in which an in-guest application handles sensitive data, such as passwords, cookies, or cryptographic keys. To protect this data, the application can use the previously defined *memory partitioning primitives* that leverage `altp2m` to allocate a protection domain, e.g, *pdomain[1]* in Figure 2: in addition to *pdomain[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 *pdomains*). This way, unauthorized *read* and *write* attempts outside *pdomain[1]* will violate the restricted access permissions. Instead of trapping into the VMM, any illegal access traps into the in-guest `#VE`-handler, which in turn 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 *pdomain[1]* and carry out the request. As soon as the application completes its request, it will execute `VMFUNC` again 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. This primitive allows 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 protection 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 protection domain.

As x86 lacks support for pointer integrity (in contrast to ARM, in which PAC [68], [69] was recently introduced), we protect pointers to objects in protection domains in software. We leverage the Linux kernel implementation of SipHash [70] 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 a protection 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 [71]. 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 protection domain in memory. After generating a key for a given protection domain, we grant the page holding the key *read-only* access permissions inside the particular domain (all other domains cannot access

the page in question). In addition, we configure Xen `altp2m` so that every protection domain maps the same guest-physical address to a different machine-physical address. Every time the guest kernel enters a protection 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 the address inside its code region as an immediate operand that cannot be controlled by adversaries (without additional means, code regions are immutable).

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 protection 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 [71], 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 [38], 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 protection 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_`$\star$) flags to indicate the conditions, the location in memory (zone), and the way the allocation will be handled [72]. 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 not only allocate a number of pages in a particular memory zone, but also to place the allocation into a specific protection domain, we extend the allocation flags. That is, we can inform the allocator by adding the `___GFP_PDOMAIN` 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 a protection domain index into the allocation flags, the allocator receives sufficient information to inform the underlying Xen `altp2m` subsystem to place the allocation into a particular protection domain (Figure 3). This way, we

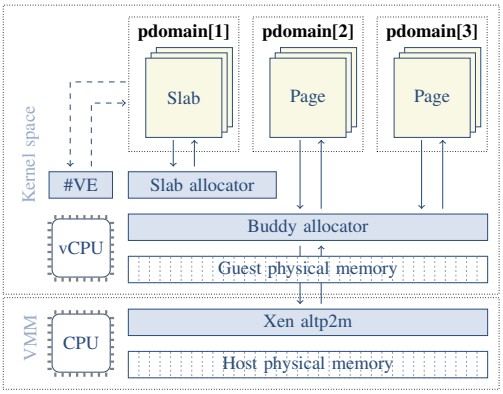

Figure 3. Extensions to the *slab* and *buddy allocator* facilitate shifting allocated pages and slabs into protection domains enforced by Xen `altp2m`.

can grant exclusive access permissions to all pages assigned to the target protection domain. 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 assignment of allocated memory pages to protection domains, we record the `___GFP_PDOMAIN` flag into the `flags` field of `struct page`, thereby enabling the buddy allocator to reclaim fortified 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 [73]. E.g., 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 protection domains using the underlying buddy allocator. To achieve this, we extend the slab implementation so that we can provide the `___GFP_PDOMAIN` flag and protection 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 protection domain (§ V-A).

C. Switches across Execution Contexts

The Linux kernel is a preemptive, highly-parallel system that must preserve the process- 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) protection domain into its persistent state.[3]

1) Context Switches: Generally, *Operating Systems* (OSes) 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 extend `task_struct` with an additional field, namely `pdomain_index_kernel`, representing the protection domain the thread resides in at any point in time. We dedicate this field to store the state of the protection 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 protection domain (§ IV-A). A process with an active *restricted view* will not be able to access the memory protected by any other protection domain. The thread updates its `pdomain_index_kernel` only when it enters or exits a protection domain. This way, the kernel can safely interrupt the thread, preserve its currently open protection domain, and schedule a different thread. In fact, we extend the scheduler so that on every context switch it switches to the saved protection domain of the thread that is to be scheduled next. To counter switching to a potentially corrupted `pdomain_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 the context of the index before entering the protection domain ❸ (§ IV-C). Since adversaries cannot create valid authentication codes without knowing the secret key, they will neither be able to forge the authentication code of the index, nor reuse an existing authentication code that is bound to a different `task_struct`.

2) Hardware Interrupts: Despite the per thread management of active protection domains, interrupts can pause a thread's execution at arbitrary points. We ensure that access to memory in any of the protection domains is restricted in the interrupt context. 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. This strategy causes temporarily interrupted threads to enter the restricted protection domain. 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 `pdomain_index_kernel`. If the thread's eligibility and the index's integrity is given, the handler enters the corresponding protection domain and continues the thread's execution. Otherwise, the `#VE` handler causes a segmentation fault and terminates the thread.

[3]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.

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

In contrast to hardware interrupts that disrupt the system's execution at arbitrary locations, the kernel explicitly schedules software interrupts (`softirq`) [74], 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 whichever process is currently active and re-use its context for handling the pending `softirq`. As such, the kernel handles `softirq` events at seemingly arbitrary times and thus adds irregular delays to the execution of the processes. Generally, work outsourced into a `softirq` cannot access the state of a certain thread. Without considerable adjustments of the `softirq` mechanism, there is no way to associate a `softirq` with a specific thread on behalf of which it is executed. This is because the `softirq` executes in the context of an arbitrarily selected process.

The Linux kernel configures ten `softirq` vectors, with one dedicated for the Read-Copy-Update (RCU) mechanism [75]. 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 a protection 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 protection 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 protection domains, as we can bind it to each task individually: if the thread responsible for executing RCU callbacks needs to enter a specific protection domain, it can do so without affecting other tasks.

D. User Space API

To counter unauthorized manipulation and information disclosure attacks, we grant user processes the ability to protect selected memory regions; we extend 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 protection domains, in which sensitive data can remain safe (❶-❷). Further, we ensure that attackers cannot illegally influence the process' active protection domain state by binding its integrity to the thread's context (❸).

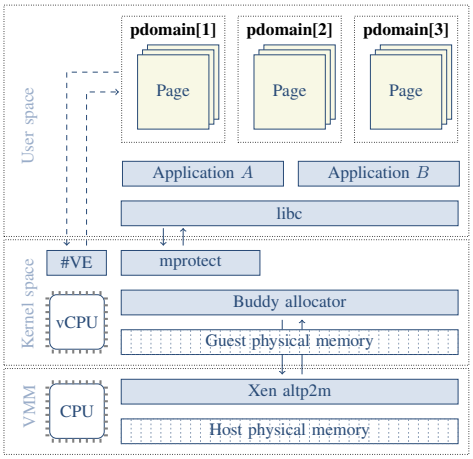

Figure 4. User-space applications interact with the Linux kernel through `mprotect` to configure independent protection domains.

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 use the unprivileged `WRPKU` instruction to further manage the memory access permissions of the corresponding protection keys (§ II-A). Likewise, we implement the system calls `sys_pdomain_{alloc, free, mprotect}` that allow programmers to dynamically allocate and maintain different protection domains in user space. In fact, these system calls implement functionality equivalent to Intel MPK on Linux. As such, they can be used as a transparent alternative on legacy systems without sufficient hardware support (❶-❷). On `sys_pdomain_mprotect` invocation, we reuse the bits of the `struct vm_area_struct` reserved for Intel MPK, and in particular, we tag the target virtual memory area with the index of the protection domain. This allows us to identify protected memory regions and release them from the associated protection domain 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_pdomain_enter`, which activates a previously allocated protection domain. By calling this system call, the kernel switches to the requested protection domain and updates the state of the currently active protection domain. We save the respective state inside the `pdomain_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` (❸). This way, we enable the kernel to verify the integrity and context of the protection domain index on every context switch—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_pdomain_enter`, we avoid unnecessary Return-Oriented Programming (ROP) (or similar code-reuse) gadgets, which could be otherwise abused by adversaries to illegally switch to arbitrary protection domains.

VI. Use Cases

In this section, we demonstrate the effectiveness and usefulness of xMP for protecting sensitive data in kernel and user space. Specifically, we apply xMP to fortify *page tables* and *process credentials* in the Linux kernel, as well as sensitive in-process data in four security-critical applications and libraries.

A. Protecting Page Tables

With Supervisor Mode Execution Protection (SMEP) [55], 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, $W \oplus X$ withdraws *execute* permissions from memory, holding the contents copied from user space. In addition, defenses based on *information hiding*, such as Kernel Space Address Layout Randomization (KASLR) [52], further obstruct kernel attacks but are known to be imperfect [39], [58], [76], [77]. 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 [78]. 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 through existing kernel hardening techniques [78]–[80].

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 protection 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 protection domain results in an access violation that terminates the process. Thus, we must grant access to the fortified paging structures to kernel components responsible for process creation and termination, and page fault handing by enabling them to *temporarily* enter the protection 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 value* into the kernel's code, 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 chose 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 [58], [76] (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.

B. Protecting Process Credentials

Linux kernel credentials describe the properties of various objects, that allow the kernel to enforce access control and capability management. Consequently, they are often targeted by 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) [81] introduce hooks at security-relevant locations that rely upon the aforementioned invariants. For instance, *SELinux* [82] and *AppArmor* [83] 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 `struct cred` instances. By applying xMP to `cred_jar`, we ensure that adversaries cannot directly overwrite the contents of `cred` instances without first entering the associated protection domain (§ V-B). As we check both the integrity and context of the active protection domain index (`pdomain_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 protection 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 additionally bind the integrity of every `cred` pointer to the associated `task_struct` during process creation. In addition, we check both the integrity and the assigned context to the specific `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, meeting requirement ❸.

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 selective memory protection 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, in order to place them in individual pages, which are then assigned to dedicated protection domains. Specifically, using the system calls introduced in § V-D, we assign the memory pages to a protection domain that grants *read-write* access to the sensitive data, 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 protection domain crashes the application. In the following, we summarize the slight changes we made to the four applications for protecting sensitive data.

OpenSSL (Nginx): OpenSSL manages prime numbers in memory via the `BIGNUM` data structure [84]. We add macros that allocate these structures into a separate protection domain. Instrumenting a widely-used library like OpenSSL enables the protection of a wide range of applications. In our case, the instrumented OpenSSL library, in combination with the Nginx web server, in HTTPS mode, offers protection against memory disclosure attacks, such as Heartbleed [5].

`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 protection domains.

mbed TLS: The mbed TLS library manages prime numbers and coefficients of type `mbedtls_mpi` in the `mbedtls_rsa_context` [85]. 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 fortify 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` [86], we enable tools such as `minisign` to safely store sensitive information in protection domains.

VII. Evaluation

A. System Setup

Our setup consists of an unprivileged domain `DomU` running the Linux kernel v4.18, atop the Xen hypervisor v4.12. In addition, we adjusted the Xen `altp2m` subsystem, so as to be used from inside guest VMs, as described in § 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 have 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

Performance is a critical aspect of modern OSes. New exploit mitigation technologies are unlikely to be employed in practice if they incur a significant run-time overhead [1]. To evaluate the performance impact of xMP, we have conducted two rounds of experiments, focusing on the overhead incurred by protecting sensitive data in kernel and user space. All reported results are means over three runs.

1) Kernel Memory Isolation: We measured the performance impact of xMP when applied to fortify 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 protection domains).

Table I shows the LMbench micro-benchmarks 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 quite low in most cases for both protected page 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()`). In contrast to page tables, we observe that although the kernel needs to access the `struct cred` protection domain when creating new processes, the overhead is insignificant. On the other hand, the protection 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 protection 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 [87]. 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 smaller overhead, the outliers call the instrumented LSM hooks [81] more frequently. In particular, the function `apparmor_file_permission` [83] 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 particular function, it is performed by every system call in those benchmarks and dominates the number of calls to all other fortified kernel functions. For every pointer authentication, this function triggers the protection 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 protection domain on 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 exact locations responsible for the performance overhead, we can focus on optimizing performance as part of our future work.

Table II presents the results for the set of Phoronix macro-benchmarks used by the Linux kernel developers to track

Table I
Performance overhead (OHD) of protection domains for *page tables*, *process credentials*, and both measured by 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 %

Table II
PERFORMANCE OVERHEAD (OHD) OF PROTECTION DOMAINS FOR *page tables*, *process credentials*, AND BOTH MEASURED BY 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 %

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 memory. 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 measured the overhead of xMP when applied to Nginx using the fortified OpenSSL library, and a server based on mbed TLS (§ VI-C). In both cases, we used the server benchmarking tool `ab` [88] 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 [28]. As shown in Figure 5, the throughput and latency overhead is negligible in most cases. 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 both latency and throughput is 0.5%. For mbed TLS, we used the `ssl_server` example [89] to execute an SSL server hosting a 50-byte file. (Note that we explicitly chose a small file so as not to mask the overhead with I/O.) Our results show an average overhead of 0.42% for latency and 1.14% for throughput.

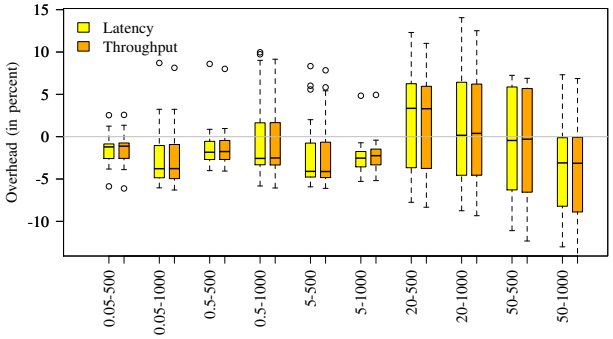

Figure 5. Latency and throughput overhead of *Nginx* using protected *OpenSSL* with varying file sizes and connections ([File Size (KB)]-[Req #]).

C. 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 [78]. 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 or (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 fortify page tables and process credentials (§ VI-A and VI-B), we were able to successfully block both attack scenarios.

To systematically evaluate xMP, we have to consider attacks that can be equally applied to all kernel structures. Therefore, disregarding which data structure is under attack, we generalize the attack vectors against sensitive kernel structures in the

following strategies. Under our threat model, attackers can:

- directly modify entries in the data structure of interest;
- redirect a pointer to the targeted data structure to an injected, attacker-controlled instance;
- redirect a pointer to 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 protection 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 protection domains. The injection of 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, the attacker can overwrite the pointer with an *existing* pointer value—holding a valid HMAC—to an instance of the data structure with higher privileges. Yet, as pointers to fortified data structures are bound to the thread's context (❸), the attacker cannot redirect pointers to instances belonging to other threads. The only remaining option to bypass context-bound pointer integrity is this to find an HMAC collision. Note that this step must be reversed when targeting page tables, i.e., the attacker must overwrite the `pgd` pointer of a *privileged* thread with the pointer to the address space of an *attacker-controlled* thread.

2) Attacking User Applications: User applications offer a broad range of different attack scenarios related to sensitive data modification or leakage. We have chosen Heartbleed [5] as a representative data leakage attack due to its high impact. Due to the lack of 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 legitimate control flow.

By equipping the vulnerable OpenSSL library with the ability to guard secret key material (§ VI-C), we prevented the sensitive regions from leaking. Illegal accesses induced an EPT violation that trapped into the `#VE` handler, in which we reported the illegal access and terminated the application.

3) Attacking Protection Primitives: Although 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 protection domains. Even if an attacker introduces a `VMFUNC` instruction in the application's memory to mount a `VMFUNC` faking attack [28], the next context switch would restore the protection domain's state from `pdomain_index_[kernel|user]`, making the kernel immune to illegal protection domain switches from user space. The attacker can attempt to use a *write* primitive to modify the kernel's protection domain state in `pdomain_index_[kernel|user]`, forcing the kernel to enter a privileged domain and grant access to sensitive data on the next context switch. However, as we bind the integrity of the active protection domain state to the associated thread's context, any attempt to tamper with it will crash the process.

On the other hand, a `VMFUNC` faking attack would allow a user-space application to create a small time window, in which the attacker could successfully switch the currently active protection domain in user space; the protection domain will be restored on the next context switch. Further, mediating the execution of `VMFUNC` instructions through the dedicated `sys_pdomain_enter` system call introduces gadgets that allow switching to protection domains previously allocated by the application. Nevertheless, to perform such attacks, the attacker will need to change the application's control flow, which will be detected by orthogonal CFI defenses (§ III).

VIII. Discussion

In this section we discuss the limitations of our xMP implementation and review extensions and alternative scenarios.

A. Limitations

The Linux callback-free RCU feature [90] relocates the processing of RCU callbacks out of the software interrupt context into a dedicated thread (§ V-C3). This allows RCU callbacks to enter protection domains without affecting other threads' protection domain state. Yet, RCU callbacks reside in a linked list of `rcu_head` instances that are embedded in the (protected) data structures. As such, the RCU subsystem tries to access the `rcu_head` instance in protected memory, e.g., every time another deferred callback is about to be linked to it. To avoid this, we disable RCU callbacks for data structures to be protected. Alternatively, we could adjust data structures to be protected so that the corresponding `rcu_head` fields are not embedded into targeted data structures.

Besides, in our implementation, we manually instruct the kernel when to enter a specific protection domain. We can automate this step by providing a compiler pass. We could instruct the compiler to bind annotated data structures to protection domains. Further, the compiler could instrument calls that enter and leave the protection domain immediately before and after accessing the annotated data structure.

Also, the current implementation does not foresee *nested* protection domains. In fact, we prohibit entering protection domains, without first closing the active domain; by nesting protection domains, the state of the previously opened domain will be overwritten. To address this limitation, the kernel needs to keep track of the previously opened protection domains by maintaining a stack of protection domain states per thread.

B. Intel Sub-Page Write Permission

Intel announced the Sub-Page Write-Permission (SPP) feature for EPTs [36] to enforce memory *write* protection on sub-page granularity. Specifically, with SPP, Intel extends the EPT

by 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 complexity. 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 `struct cred` instance. As such, the system has to first enter the protection domain to relax permissions to otherwise *read-only* credentials, before it can update the maintenance information. By means of Intel SPP, we can arrange the `struct cred` data structure in such a way that all meta information is placed into writable sub-pages, despite the memory access restrictions of the protection domain. Also, as discussed above, we have solved the RCU related limitation by deactivating RCU callbacks for protected in-kernel data-structures. With Intel SPP, we will be able to place such pointers to writable sub-pages, without the need for deactivating callbacks for the protected data structure.

C. Execute-Only Memory

Similarly to the lack of *non-readable* memory (§ II-A), the x86 MMU does not support *execute-only* memory either; every code page has to be *executable* and *readable*. This forced adversaries to come-up with advanced attacks targeting the control flow. For instance, Just-In-Time ROP (JIT-ROP) [39] enables code-reuse attacks, despite the existence of fine-grained ASLR. By reading code pages, an attacker can harvest code-reuse gadgets, on-the-fly, to construct a suitable payload. As a response, defenders [11], [38], [42], [91] enforce execute-only memory to obstruct such attacks. Likewise, by defining execute-only protection domains to hold code pages, xMP can counter *JIT-ROP* similarly to Readactor [18].

IX. Related Work

While the possibility of non-control data, or data-oriented, attacks has been identified before [3], Chen et al. [4] were the first to demonstrate the viability of data-oriented attacks in real-world scenarios, ultimately rendering them as realistic threats. With FlowStitch [40], 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. [7] 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. [6] 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 also permits 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. [9] 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*. Specifically, they present a BOP compiler that translates exploits, written in a high-level language, to *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 [92] associates annotated data types with security sensitive information. Based on these annotations, DataShield partitions the application's memory into two disjoint regions. Further, DataShield 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 [28], [29], [93]. While MemSentry [29] isolates sensitive data, SeCage [28] 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 between the secret and non-secret compartments. EPTI [93] implements an alternative to KPTI by using memory isolation techniques similar to xMP. PrivWatcher [30] leverages virtualization to ensure the integrity of process credentials. Contrary to our solution, PrivWatcher creates shadow copies of `struct cred` instances, and maintains them in a *write*-protected region. Lastly, PT-Rand [78] protects page tables without employing virtualization. Instead, the authors avoid potential modifications of paging structures by randomizing their location. Finally, with PARTS [69], Liljestrand et al. introduce a compiler instrumentation framework to cope with pointer-reuse attacks by leveraging the (recently-introduced) ARMv8.3-A pointer authentication features.

X. Conclusion

In this paper, we proposed novel techniques against data-oriented attacks. Specifically, we leveraged Intel's virtualization extensions to set the ground for xMP primitives that facilitate the protection of sensitive data structures in kernel and user space. These primitives extend the Linux memory management capabilities to empower software developers with the ability to shift sensitive data structures into disjoint and isolated protection domains, despite limitations of the x86 MMU. We further equip pointers to memory inside protection domains with authentication codes to thwart illegal pointer redirection. We applied xMP to fortify the Linux kernel page tables and process credentials, as well as sensitive data in user space. The protection domains withstood various attacks, despite a strong attacker. In conclusion, we believe that our primitives introduce a powerful means that brings us one step closer towards winning the fight against data-oriented attacks.

REFERENCES

[1] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal War in Memory," in *IEEE Symposium on Security and Privacy (S&P)*, 2013.

[2] L. Cheng, H. Liljestrand, T. Nyman, Y. T. Lee, D. Yao, T. Jaeger, and N. Asokan, "Exploitation Techniques and Defenses for Data-Oriented Attacks," *arXiv preprint arXiv:1902.08359v2*, 2019.

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

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

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

[6] 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*, 2015.

[7] 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.

[8] B. Sun, C. Xu, and S. Chong, "The Power of Data-Oriented Attacks: Bypassing Memory Mitigation Using Data-Only Exploitation Technique, Part I," *Black Hat, Asia*, 2017.

[9] 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.

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

[11] 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.

[12] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, and R. H. Deng, "ROPecker: A generic and practical approach for defending against ROP attack," in *ISOC Network and Distributed System Security Symposium (NDSS)*, 2014.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

[20] Microsoft, "Control Flow Guard," https://docs.microsoft.com/en-us/windows/desktop/SecBP/control-flow-guard, 10 2018.

[21] Google, "The Chromium Projects," https://www.chromium.org/developers/testing/control-flow-integrity, 10 2018.

[22] LLVM, "Control Flow Integrity," http://clang.llvm.org/docs/ControlFlowIntegrity.html, 10 2018.

[23] 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.

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

[25] 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*, 2002.

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

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

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

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

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

[39] 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.

[40] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang, "Automatic Generation of Data-Oriented Exploits," in *USENIX Security Symposium*, 2015.

[41] 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.

[42] 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.

[43] S. Brookes, R. Denz, M. Osterloh, and S. Taylor, "Exoshim: Preventing memory disclosure using execute-only kernel code," in *International Conference on Cyber Warfareand Security (ICCWS)*, 2016.

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

[45] 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," 2003.

[46] 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.

[47] 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.

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

[49] 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.

[50] 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.

[51] S. Liakh, "NX Protection for Kernel Data," https://lwn.net/Articles/342266/, July 2009.

[52] J. Edge, "Kernel Address Space Layout Randomization," https://lwn.net/Articles/569635/, October 2013.

[53] J. Corbet, "x86 NX Support," https://lwn.net/Articles/87814/, June 2004.

[54] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis, "kGuard: Lightweight Kernel Protection against Return-to-user Attacks," in *USENIX Security Symposium*, 2012.

[55] F. Yu, "Enable/Disable Supervisor Mode Execution Protection," https://goo.gl/utKHno, May 2011.

[56] J. Corbet, "Supervisor Mode Access Prevention," https://lwn.net/Articles/517475/, October 2012.

[57] ——, "The Current State of Kernel Page-Table Isolation," https://lwn.net/Articles/741878/, December 2017.

[58] 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.

[59] A. van de Ven, "Add `-fstack-protector` Support to the Kernel," https://lwn.net/Articles/193307/, July 2006.

[60] Open Web Application Security Project (OWASP), "C-Based Toolchain Hardening," https://www.owasp.org/index.php/C-Based_Toolchain_Hardening, January 2019.

[61] 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.

[62] 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.

[63] 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.

[64] 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.

[65] 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*, 2014.

[66] J. Corbet, "Kernel Support for Control-Flow Enforcement," https://lwn.net/Articles/758245/, June 2018.

[67] 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.

[68] Qualcomm, "Pointer Authentication on ARMv8.3," https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf, January 2017.

[69] 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*, 2019.

[70] J.-P. Aumasson and D. J. Bernstein, "SipHash: a fast short-input PRF," in *International Conference on Cryptology in India*, 2012.

[71] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis, "ret2dir: Rethinking Kernel Isolation," in *USENIX Security Symposium*, 2014.

[72] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*, 3^{rd} ed. O'Reilly Media, 2005, ch. Memory Management, pp. 294–350.

[73] J. Bonwick, "The Slab Allocator: An Object-Caching Kernel Memory Allocator," in *Proc. of USENIX Summer*, 1994.

[74] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*, 3^{rd} ed. O'Reilly Media, 2005, ch. Interrupts and Exceptions, pp. 131–188.

[75] P. McKenney, "What is RCU, Fundamentally?" https://lwn.net/Articles/262464/, December 2007.

[76] 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*, 2018.

[77] 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.

[78] L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi, "PT-Rand: Practical Mitigation of Data-only Attacks against Page Tables," in *ISOC Network and Distributed System Security Symposium (NDSS)*, 2017.

[79] J. Lee, H. Ham, I. Kim, and J. Song, "POSTER: Page Table Manipulation Attack," in *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[80] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. Adve, "Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation," in *ACM SIGPLAN Notices*, 2015.

[81] J. Morris, S. Smalley, and G. Kroah-Hartman, "Linux Security Modules: General Security Support for the Linux Kernel," in *USENIX Security Symposium*, 2002.

[82] P. Loscocco and S. Smalley, "Integrating Flexible Support for Security Policies into the Linux Operating System," in *USENIX Annual Technical Conference*, 2001.

[83] A. Gruenbacher and S. Arnold, "AppArmor Technical Documentation," 2007.

[84] OpenSSL, "OpenSSL Manpages v1.0.2," https://www.openssl.org/docs/man1.0.2/man3/bn.html, April 2019.

[85] ARM mbed, "mbed TLS v2.16.1 Source Code Documentation," https://tls.mbed.org/api, April 2019.

[86] Libsodium, "Libsodium Documentation," https://libsodium.gitbook.io/doc/memory_management, April 2019.

[87] M. Fleming, "A Thorough Introduction to eBPF," https://lwn.net/Articles/740157/, December 2017.

[88] Apache HTTP Server Project, "ab - Apache HTTP Server Benchmarking Tool," https://httpd.apache.org/docs/2.4/programs/ab.html, April 2019.

[89] ARM mbed, "mbed TLS Sample Programs," https://github.com/ARMmbed/mbedtls/blob/master/programs, April 2019.

[90] J. Corbet, "Relocating RCU Callbacks," https://lwn.net/Articles/522262/, October 2012.

[91] 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.

[92] S. A. Carr and M. Payer, "Datashield: Configurable Data Confidentiality and Integrity," in *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2017.

[93] 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*, 2018.