A Secret-Free Hypervisor: Rethinking Isolation in the Age of Speculative Vulnerabilities

Hongyan Xia[†*] David Zhang[†] Wei Liu[†‡] Istvan Haller[†] Bruce Sherwin[†] David Chisnall[†]
[†]Microsoft

***Abstract*—In recent years, the epidemic of speculative side channels significantly increases the difficulty in enforcing domain isolation boundaries in a virtualized cloud environment. Although mitigations exist, the approach taken by the industry is neither a long-term nor a scalable solution, as we target each vulnerability with specific mitigations that add up to substantial performance penalties. We propose a different approach to secret isolation: guaranteeing that the hypervisor is Secret-Free (SF).**

A Secret-Free design partitions memory into secrets and non-secrets and reconstructs hypervisor isolation. It enforces that all domains have a minimal and secret-free view of the address space. In contrast to state-of-the-art, a Secret-Free hypervisor does not identify secrets to be hidden, but instead identifies non-secrets that can be shared, and only grants access necessary for the current operation, an allow-list approach. SF designs function with existing hardware and do not exhibit noticeable performance penalties in production workloads versus the unmitigated baseline, and outperform state-of-the-art techniques by allowing speculative execution where secrets are invisible. We implement SF in Xen (a Type-I hypervisor) to demonstrate that the design applies well to a commercial hypervisor. Evaluation shows performance comparable to baseline and up to 37% improvement in certain hypervisor paths compared with Xen default mitigations.

Further, we demonstrate Secret-Free is a generic kernel isolation infrastructure for a variety of systems, not limited to Type-I hypervisors. We apply the same model in Hyper-V (Type-I), bhyve (Type-II) and FreeBSD (UNIX kernel) to evaluate its applicability and effectiveness. The successful implementations on these systems prove the generality of SF, and reveal the specific adaptations and optimizations required for each type of kernel.

***Index Terms*—security-in-depth, speculative vulnerabilities, hypervisor security, secret-free.**

I. Introduction

In the conventional model of kernel and user space separation, exploiting user space server applications has always been an attractive target. These applications often run with administrative privileges and handle data from multiple parties, which are prone to attacks from malicious clients [1]–[5]. However, sophisticated static analysis tools, dynamic code instrumentation, new programming languages, randomization and sandboxing techniques in recent years have significantly increased the complexity of mounting such attacks [6]–[9]. Instead, security researchers have turned to kernel vulnerabilities. The ever increasing code base of OS kernels exerts tremendous amount of pressure on code reviews and security auditing. For example, the size of the Linux kernel has grown from 6.6 MLOC back in 2.6.11 to 27.8 MLOC in 2020 [10]. Analysing all the code for security vulnerabilities is astronomically difficult, if not outright impossible. The growing attack surface inevitably leads to an increasing number of disclosed kernel vulnerabilities, ranging from heap overflows, use-after-free bugs, undefined behaviour to race conditions and insufficient privilege checks [11]–[14].

In recent years, speculative vulnerabilities such as *Spectre* [15] and *Meltdown* [16] further complicate kernel and VM isolation. Enforcing architectural boundaries is no longer sufficient in the presence of speculative side channels. To prevent leaking sensitive content over the speculative paths on affected hardware, mitigations are deployed including Kernel Page Table Isolation (KPTI) [17], Indirect Branch Prediction Barriers (IBPB) and Indirect Branch Restricted Speculation (IBRS) [18], fences, cache flushes, core scheduling [19], retpolines [20], and so on. As these mitigations introduce undesirable performance degradation especially in system-call-heavy workloads, hardware modifications have been proposed to limit the effect of speculation [21]–[23]. Although these CPU and cache changes are able to block or restrict speculation at the silicon-level, it remains to be seen how well they integrate into high-performance server farms and how long such integration may take before the next vulnerability is discovered.

As existing software mitigations can be inefficient and hardware modifications take time to reach commercialization, we explore new designs of the hypervisor such that each domain always has a minimal surface and visibility. In the wake of recent speculative execution vulnerabilities, we no longer assume that architectural boundaries are sufficient and must include speculative side channels in the design. In this paper, we present the Secret-Free (SF) hypervisor as a systematic defense-in-depth solution. We categorize system state as either secrets or non-secrets and restrict all domains to the minimum, having no access to secrets. In contrast to many existing mitigations, we operate under an allow-list approach, treating states as secrets by default. An address space contains secrets of its own and explicitly identified non-secret data. Unlike KPTI or XPTI (Xen Page Table Isolation), we do not assume full access in the hypervisor. The hypervisor is restricted to the current guest domain upon entry, and only creates temporary mappings for secret access when necessary.

Reaching the SF guarantees requires several key design components, including direct-map teardown, per-domain and per-vCPU private memory, ephemeral mappings, efficient map cache as well as the isolation of vCPU state, guest register

[*]Part of the work is done while at Amazon Web Services.
[‡]Part of the work is done while at Citrix/Xen Project.

frames and hypervisor stacks. Together, they guarantee the isolation of guest and hypervisor secrets while not introducing noticeable performance impact from strict domain isolation. Overall, the design and implementation of a secret-free hypervisor do not depend on hardware changes and can be deployed at scale on existing cloud platforms with negligible overhead, and is a generic structure that can be extrapolated to multiple types of kernels.

The contributions of this paper are summarized below:

- We propose a design that isolates all domains (including the hypervisor) from secrets. Instead of identifying secrets that need to be hidden, the SF design maintains a minimal address space and identifies non-secrets that can be exposed.
- We extend the isolation to guest registers, vCPU state and hypervisor stacks using private mappings.
- We implement optimizations to minimize the overhead from the isolation of secrets. We invent an efficient caching mechanism for ephemeral mappings, proving that a full address space is unnecessary for efficiency.
- We implement and evaluate the secret-free design on the open-source Type-I Xen hypervisor. With specific adaptations and optimizations to Xen, we achieve negligible overhead in real-world workloads and superior performance compared with default mitigations. We evaluate its security and demonstrate that it is impervious to several categories of architectural and speculative attacks.
- We implement the Secret-Free design in Hyper-V, bhyve and the FreeBSD kernel. The implementations show secret freedom is a generic technique that can be retrofitted into a variety of systems including traditional UNIX kernels, Type-I, and Type-II hypervisors. We analyse the applicability and effectiveness of the technique on these systems and reveal necessary adaptations for further performance improvement and security hardening.

II. Background

In this section, we provide the background for kernel privilege separation and address-space layout. We explain why such a structure is susceptible to a broad category of architectural and speculative side channel attacks.

A. Address space layout of modern kernels and VMMs

Assuming no Page Table Isolation (PTI), most kernels have a similar address space layout under virtual memory protection. User space guest address range (typically mapped at low addresses) maps user accessible memory which is per-process whereas the kernel / hypervisor address range (at high addresses) is mapped into all page tables. As is shown in Fig. 1, the full address space of a user process includes the global kernel mappings guarded by permission bits in the Page Table Entry (PTE), which are architecturally accessible only under kernel privileges. Such a global mapping accelerates system calls, interrupts and exception handling by requiring only a privilege level change but not a page table swap,

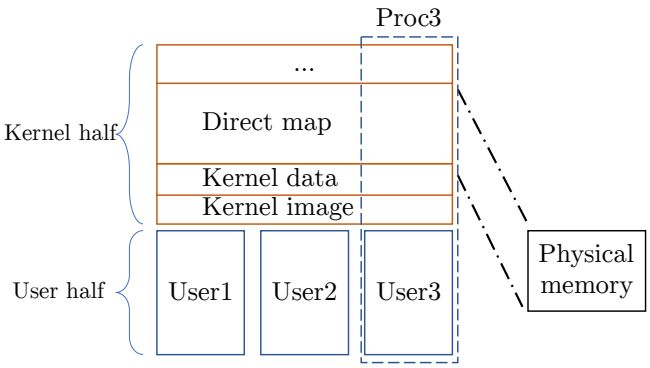

Fig. 1. Kernel/user split. The dashed square indicates the full address space of Process 3.

reducing the performance cost from the accompanying CPU pipeline and TLB flushes.

This layout applies to hypervisors as well. The second-level paging or shadow page tables will present a different set of memory to each VM but the full hypervisor address range is mapped into all address spaces, ensuring fast hypercalls and VM exit handling. The kernel and user address split is so ubiquitous that we have not found a different design in the kernels or hypervisors (Xen, Linux/KVM, FreeBSD/bhyve) studied in this paper.

B. The direct map

By taking advantage of the abundance of virtual address space, modern 64-bit kernels implement a common facility for performance and ease of management: the direct map. This is a large contiguous virtual address range in kernel that is 1:1 mapped to the entire physical memory (Fig. 1). Variations exist that deviate from a flat 1:1 mapping on different platforms. The Xen hypervisor permits "compression" by removing large holes on the direct map, useful for accommodating devices with RAM starting at high addresses or in multiple disjoint ranges. Windows NT maintains multiple paged and non-paged pools. These pools are physical pages mapped to virtual addresses, which are either required to remain in memory (non-paged pool) or can be removed from the memory because a copy is already stored on the disk (paged pool). Combined, these pools map a large fraction of the physical memory into the kernel address space of every process.

The direct map leverages the address space under 64-bit ISAs and is permanently mapped in all page tables, enabling the kernel or hypervisor to efficiently access all (or most) memory at any time. The access is fast because such large contiguous mappings are typically handled by superpages to reduce TLB pressure and page-table size.

C. Exploiting kernel privileges

Kernel code is susceptible to classes of vulnerabilities introduced by programming errors and can be exploited to perform data leakage, data corruption, code injection and remote execution. The lack of mutual isolation and the monolithic

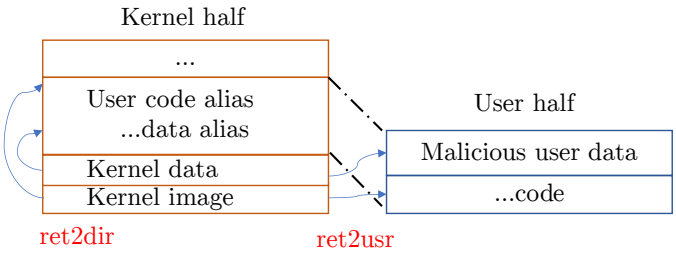

Fig. 2. ret2usr and ret2dir

nature open up possibilities to inject code and data directly from lower privilege levels, as demonstrated by *ret2usr* and *ret2dir* in Fig. 2.

In both attacks, the attacker requires no code or data injection into kernel memory. The payloads are prepared directly in user space before redirecting kernel control flow to malicious code or to a stack pivoting gadget [24], [25].

In response, hardware features such as Supervisor Mode Access Prevention (SMAP), Supervisor Mode Execution Prevention (SMEP) [18] and Privileged Execute-Never (PXN) [26] were introduced to thwart unintended redirection of kernel access to user memory. Unfortunately, they are ineffective against *ret2dir*. The direct map is valid kernel memory, thus it is impossible to distinguish between intended and unintended direct map access with the aforementioned hardware features. The authors of *ret2dir* proposed eXclusive Page Frame Ownership (XPFO) as a mitigation that unmaps memory from the direct map on page allocation to user space. However, the cost of TLB shootdowns for direct map maintenance prohibits its use in scenarios where processes are created and destroyed frequently and does not scale as the core count increases.

D. Speculative execution attacks of VMs

The recent epidemic of speculative execution attacks have shown that VMs are not excluded from the breach of traditional architectural protection boundaries. Spectre, Meltdown and many other speculative vulnerabilities have disclosed multiple variants that are applicable to virtualization [15], [16], [27]–[29].

In addition, both Spectre [15] and L1TF [27] have demonstrated that it is insufficient to restrict speculative side channels in the VM context, as the hypervisor can be manipulated into fetching data into the cache. PTI, for example, recovers the full kernel map upon entry and is not effective against branch predictor mistraining. Worse, the presence of the direct map allows a speculatively manipulated hypervisor to access arbitrary physical pages. IBRS, IBPB, and STIBP [18] are provided to guard against mistraining. These measures either flush potential malicious branch predictor states or isolate between privilege levels and sibling threads. They do not address the fundamental problem that a kernel/hypervisor exposes a giant surface that can be exploited once another side channel is found. As a result, several implementations have emerged to limit the hypervisor itself, preventing speculation from accessing secrets via its vast address space [30], [31].

Although necessary, the increasing number of security hardening and speculation defenses that accumulated over the years has resulted in more than 100% time overhead in several core kernel operations [32]. Outside micro-benchmarks, the study also revealed a degradation of more than 30% in real-world workloads. A systematic mechanism that defends against a class of vulnerabilities is desirable, rather than countering each attack in different ways, which may not compose well with other defenses, and slowly degrade performance over time as more vulnerabilities are found.

E. Type-I, Type-II hypervisors and Xen

Hypervisors are categorized as either Type-I or Type-II. Type-I hypervisors (for example Xen, Hyper-V, and VMWare ESXi) sit directly atop the hardware and manage guest operating systems, often requiring at least a privileged domain to support driver back-ends and to issue commands to manage unprivileged domains. Type-II hypervisors (for example KVM or bhyve) exist atop or within an operating system, abstracting guest domains as user processes and using the host OS drivers for hardware access.

Xen is a Type-I hypervisor. A privileged domain, dom0, is started first to manage other guest domains (domUs). Dom0 has drivers for hardware, and provides virtual disks and network access for domUs. Dom0 will run the back-end driver for paravirtualized devices that are made available to other domains (for example, network interfaces and disks), which multiplexes and forwards to the hardware requests from the front-end driver in each DomU [33].

Xen's fully paravirtualized (PV) mode, inherited from the Nemesis operating system, is a unique feature that predates x86 virtualization hardware extensions. PV mode exposes a series of hypercalls for MMU management, I/O drivers, timers and interrupts, enabling the guest kernel and user space to function in lower privilege rings without virtualization extensions. Most hypervisors provide paravirtualized device drivers, even if they rely on hardware features for CPU and memory virtualization.

III. Threat Model

We assume that an attacker resides within an unprivileged VM. In regard to speculative execution attacks, we assume the attacker does not co-exist in the same VM with the victim, or in the context of OS kernels, the same process. We categorize speculative vulnerabilities into three different classes:

a) Permission: Speculation violating permissions. These attacks are performed directly in the lower privileged context and target mappings that are available in the page table but are restricted from usage within the current domain. A typical attack of this category is Meltdown.

b) Coercion: Triggering a speculative control-flow transition in the higher-privileged context to execute a disclosure gadget. The side-effects of the gadget executed under speculation can still persist in the micro-architectural state

and can be converted to architectural state from the lower privileged context to extract secrets. Examples of this class include Spectre-V1 and V2.

c) Micro-architectural: Attacks that purely gather residual signals from shared μarch structures (for example store buffers, load ports, L1D cache). These attacks are performed directly in the lower privileged domain after it is transitioned to. The sharing of μarch can be either spatial (hyper-threading) or temporal (hypervisor entry/exits and context switches). L1TF and MDS belong to this category.

We consider all three categories of speculative attacks to be in scope. The secret-free design mitigates permission and coercion attacks fully, by eliminating secrets from both guest and hypervisor contexts. Existing mitigations for these attack classes are therefore unnecessary. The design does not directly address speculative side channels from the μarch category, but this work composes with existing mitigations such as Core Scheduling. We demonstrate that the defense against the μarch class is significantly simpler to build on top of a secret-free hypervisor than on the previous state of the art.

Randomization-based mitigations are out of scope. We do not rely on any form of ASLR for secret isolation, as existing literature has demonstrated that the limited entropy and possible surfaces of pointer leaking often limit its effectiveness. An attacker is permitted to reveal the memory location of secrets, provided that secret contents remain inaccessible.

IV. Design goal

a) Definition of secrets: We partition data into secrets and non-secrets. The former includes all guest register state and memory as well as their copies (e.g., guest register spills when entering the hypervisor, `copy_from_guest()` or `copy_to_guest()` during hypercalls). In the case of Type-I hypervisors, even though the privileged guest domain (dom0) is not directly occupied by customers, the implicit copying between driver back-ends and domU front-ends means that dom0 state must be treated as secret. We do not consider guest state to be secret to its owner. An exploit that reveals register or memory contents belonging to the attacker does not reveal secrets by our definition.

b) Secrets by default: State-of-the-art techniques audit each vulnerability and identify vulnerable surfaces for isolation, expanding the deny-list. The secret-free design considers all data as secrets by default. We shall explore a different approach by constructing a minimal surface for all domains (including the hypervisor) and identifying non-secrets that are permitted in the address space.

c) Secret-Free: The secret-free design shall not allow secrets to be visible to any domains other than the owner, neither architecturally nor in any form of the speculative side channels discussed in Section III. A secret-free view defines both guest and hypervisor space. The hypervisor entered under a domain's context (hypercalls, exceptions or interrupts) cannot contain secrets of other domains. KPTI, for example, violates this guarantee because it recovers the full address space and maps secrets to other domains on kernel entry.

d) Performance: Overall, the overhead needs to be low and must not impact real-world application performance. A new hypervisor design that exhibits high overhead is unable to provide sufficient value against existing mitigations. Many specific mitigations are no longer required because a secret-free design is a systematic defence against several categories of vulnerability. Our evaluation demonstrates that the secret-free design replaces said mitigations and shows competitive or improved performance.

V. A secret-free hypervisor

We construct the minimal and secret-free view of all domains with the following components.

A. Tearing down the direct map

As discussed in Section II-B on page 2, the direct map is a huge and permanent window to physical memory. Such a surface contradicts the secret-free principle and has the following drawbacks:

1) We observe that this is a major attack surface for a range of architectural and speculative attacks [15], [16], [25]. It is a convenient surface for malicious parties as any attacker-controlled out-of-bound vulnerability quickly escalates to full memory access.
2) It consumes a significant chunk of the address space, limiting the degree of randomization in kernel. With sufficient amount of RAM, the entropy of the kernel ASLR can be as low as 7 bits, requiring only 128 probes [16].
3) It becomes a dependency for future kernel development. For example, as the code heavily depends on the direct map window for memory access, both FreeBSD and Xen HVM on amd64 require a sufficiently large virtual address range to cover all physical memory, without which high physical memory would be inaccessible.
4) There are no guard zone around allocations. Out-of-bound accesses can easily corrupt data on adjacent pages. As a result, Linux has moved to *virtually mapped* kernel stacks as the default to trap and handle overflows [34].

We introduce mapping APIs to **all** hypervisor memory allocations so that dereferences are made only after explicit mapping requests. The direct map is removed because the code no longer assumes an implicit mapping before accessing memory. The API differs depending on whether the allocation contains secrets. Non-secrets create and destroy globally visible mappings on allocation and deallocation, allowing fast access from all contexts during their lifetime. These are typically hypervisor-internal data structures unrelated to guest secrets, including the hypervisor image, host ISA descriptors (for example, x86 host GDT, IDT, TSS), generic scheduler state and so on. Access to secret memory will be isolated via the mechanisms described in the following sections.

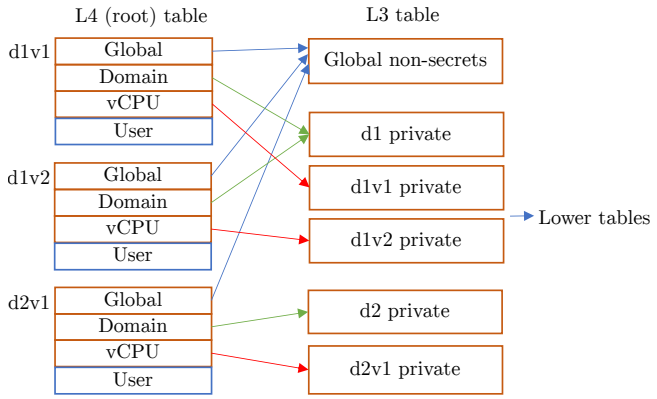

Fig. 3. L3 table sharing for global, domain-private and vCPU-private L4 entries. d*nvm* denotes the vCPU from domain *n* with vCPU ID *m*. For 2-stage address translation, the user region may live in a separate table instead of sharing address space with the hypervisor.

B. Domain- and vCPU-private mappings

We do not permit secrets to be mapped globally. In addition to the user half and the globally mapped hypervisor half, we further construct two regions: domain-private and vCPU-private mapping areas. A vCPU is scheduled on zero or one physical CPUs (pCPU) at any given time and so *vCPU-private implies pCPU-private*. The regions can be mapped to the abundant space made available by removing the direct map. In modern ISAs with hierarchical page tables, shared domain-private mappings among vCPUs and shared global non-secret mappings can be cheaply implemented via sharing lower level tables. In 4-level paging, all L4 (root) page table entries can be categorized as global, domain-private or vCPU-private, shown in Fig. 3. Private regions are intended for long-lived secret data structures belonging to the current domain or vCPU, such as hypervisor stacks and vCPU register frames, whose lifetimes track their corresponding vCPUs. Short-lived objects will be handled by ephemeral mappings later.

A hypervisor may consider guest memory as long-lived and may map it in its entirety in the domain-private region to simplify `copy_from_guest()` and `copy_to_guest()` operations. We make the design choice to avoid permanently mapping all guest memory in hypervisor space, including in domain- and vCPU-private regions. We have learned from *ret2dir* that aliasing user memory in kernel enables gadget injection at lower privilege levels. Thus, our secret-free design creates ephemeral mappings to access guest memory even if the hypervisor is entered under that guest's context for defense-in-depth.

C. Hypervisor stacks, vCPU state and register frames

As we define guest registers and their copies to be secrets, any spills of register state must be isolated. Upon entry, the hypervisor switches to its stack and a copy of the vCPU state (including the guest register frame) is spilled onto it, potentially leaking sensitive data to speculative side channels. A secret-free hypervisor no longer allows such private states to be globally mapped.

We leverage the vCPU-private region and create hypervisor stacks for each vCPU with private mappings. This ensures that guest register spills and hypervisor handling of guest data on the stack are invisible to other domains and even to other vCPUs of the same domain, assisting the guest OS to further isolate its processes. When implementing vCPU state isolation, one must be aware of any implicit spills by the architecture (such as the x86 VMCS page) and should not allow them to be visible. We find two common patterns that do not compose with private stacks which must be addressed via global bounce buffers.

a) Spatial bouncing: Hypervisor code may pass stack allocations as function call arguments. The callee may not have a mapping to the caller's vCPU-private stack. Such a scenario is common. For example, the callees of an Inter-Processor Interrupt (IPI) are from different contexts and no longer share mappings after stack isolation. We address this using global bounce buffers which are allocated per physical core and are globally mapped. Arguments are fetched from the buffer instead of the caller's stack.

Global bouncing must enforce the secret-free principle by not storing any secrets in the buffer. When secrets are passed, ephemeral mappings must be used instead. In practice, this is not a concern as hypervisor IPIs typically pass function pointers and non-secret hypervisor data, which are globally mapped as the hypervisor image and non-secret pool anyway.

b) Temporal bouncing: Bouncing may be needed even within the same host CPU, especially when enforcing secret freedom on per-physical-CPU hypervisor stacks. Xen x86_64, for example, allocates a hypervisor stack for each host core. Context switching to another vCPU reuses the current per-pCPU stack. This causes two problems. First, isolating the stack mapping but still sharing the underlying memory does not correctly enforce secret freedom. Second, after isolating the underlying pages, the new context is unable to read the current stack frame or perform function returns as the new stack is empty.

For a per-pCPU stack hypervisor, we first isolate physical memory by implementing per-vCPU stacks. Then, we ensure the next context does not rely on previous stack frames. The previous context writes shared variables to the per-pCPU temporal bounce buffers, allowing the context switch to complete on a new empty stack. Similar to spatial buffers, care must be taken to ensure no secrets are bounced, otherwise ephemeral mappings must be used instead.

Overhead from global bounce buffers: Although these specific patterns need to be modified, avoiding hypervisor stack allocations as function call arguments and not sharing the stack on context switch boundaries are further hardening of the hypervisor. The run-time overhead of spatial bouncing is negligible. For temporal bouncing during context switch, the overhead is higher because using two separate stacks and the global buffer increases cache and TLB misses.

Per-vCPU stacks increase memory consumption as well. For commercial cloud platforms, the maximum supported number of host pCPUs and total vCPUs are 512 and 2048 for Hyper-V [35], 768 and 4096 in ESX [36] and 288 host CPUs (1152 vCPUs assuming an overcommit factor of 4) for XenServer [37]. Assuming 256 host cores with a total of 4096 vCPUs, introducing 4KiB per-pCPU bounce buffers and 16KiB per-vCPU stacks consumes 65MiB of memory, which we do not believe is a substantial pressure on the host and is only a pessimistic case on a per-pCPU stack hypervisor.

D. Ephemeral mappings

A secret-free hypervisor context sees only the hypervisor image, the vCPU-private stack, register state of the current vCPU, and internal secret-less bookkeeping structures. The hypervisor must create ephemeral mappings for short-lived objects or other accesses whose mappings are not present in the minimal address space. These include walking and modifying page tables, `copy_from_guest()` and `copy_to_guest()` during hypercalls, background scrubbing of free heap memory and so forth. This is a stark contrast to existing hypervisor and OS kernel designs, as we never switch to the full page table but only grant temporary access necessary for the hypervisor to complete the current operation. Mapping and unmapping for a temporary access in the global map area is costly, because the IPI and TLB operations quickly multiply as the core count increases. For example, XPFO suffers 27–31% performance degradation from IPIs even on a 4-core desktop after optimizations [25]. Broadcasting is feasible when hardware acceleration exists. AMD Milan [38] and Arm MP Extensions [26] allow for TLB invalidation on all CPUs. We do not rely on hardware TLB broadcasting because such an architectural assist is not yet ubiquitous across ISAs or across different generations of CPUs.

When accessing guest memory during `copy_from_guest()`, for example, another hypervisor context is unlikely to simultaneously copy and mutate data at the same page. Based on this observation, we introduce a per-vCPU ephemeral mapping infrastructure. The hypervisor uses local APIs for temporary access. Ephemeral mappings are created and destroyed in the local ephemeral address range visible only to the current vCPU, avoiding scalability issues from broadcasting page table maintenance operations. Care must be taken to guarantee the private ephemeral window does not outlive the underlying pages. For example, pages ballooned out by a guest may be allocated to other domains. The hypervisor must ensure the vCPU ephemeral mappings are flushed during ballooning, or take references to prevent the underlying memory from changing ownership while the mappings are alive.

A brief attack window exists if the hypervisor ephemerally maps secrets of domain B under the context of domain A. In reality, this is not a concern because a sensible hypervisor implementation will not map live pages from B under an unprivileged A. If A is privileged, ephemeral mappings of foreign memory under its context can be triggered via privileged hypercalls. However, a privileged domain such as dom0 is already able to architecturally access guests' memory (for example, via device emulation, driver back-ends and debugging hypercalls) and a speculative side channel is uninteresting. De-privileging dom0 is outside the scope of this paper.

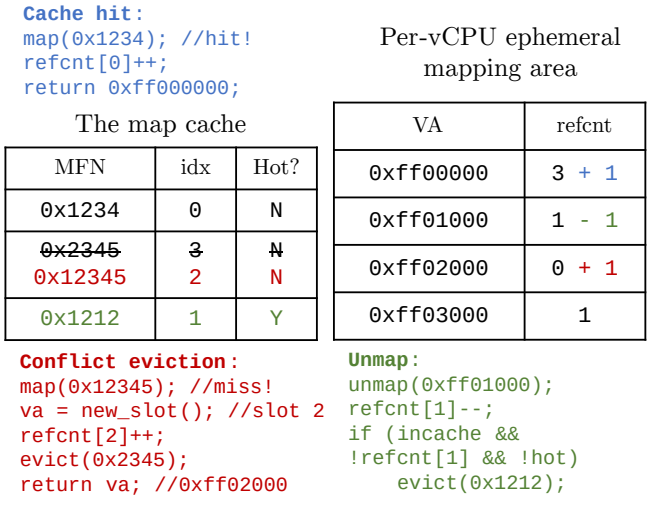

Fig. 4. Structure of a direct-map map cache. Note that in the unmap case, a hot entry will not be evicted even after all references are dropped. MFN `0x1234`, `0x12345`, `0x1212` are hashed to cache slot 0, 1, 2 respectively. The index field of a cache entry points to the ephemeral mapping slot backing the cache entry, used to look up the allocated virtual address and to adjust the reference count.

Self-mapping page tables: Creating ephemeral mappings requires modifying the page table of the current hypervisor context. We lose the ability to walk page tables after the removal of the direct map because there is no longer a convenient direct map alias to access an arbitrary physical address. We use page table self-map to overcome this limitation to locate and modify the PTEs of ephemeral mappings, which is a common technique in kernel code for PTE modification without manual page table walking (see Appendix for details). Note that self-map can only be used for a virtual address of the current installed page table, meaning it cannot act as a generic page table walker starting from an arbitrary root. A generic walker needs to be implemented on top of the ephemeral mapping infrastructure to map arbitrary physical addresses when the direct map is absent.

E. The map cache

The cost of manipulating mappings locally is still substantially more expensive than bitwise operations to access the direct map. The x86 `invlpg` instruction for TLB invalidation, for example, is a serialising operation that flushes the pipeline [18]. We introduce a map cache to allow for efficient ephemeral memory access. Ephemeral mappings often observe certain level of spatial and temporal locality: guest buffers passed in a hypercall are likely to be reused; consecutive mappings are either at adjacent guest physical memory, or at least share the same top levels of page directory pages.

The structure of the map cache is shown in Fig. 4. When the hypervisor requests an ephemeral mapping to a Machine Frame Number (e.g. a 4KiB-page at physical address `0x1234000` has an MFN of `0x1234`), it computes the hash slot based on the MFN and fetches the cached entry. If the entry already contains a mapping to the same MFN, the cache immediately returns the virtual address by looking up the associated ephemeral mapping slot backing the cache entry. If not, the entry is evicted from the cache and a new mapping is inserted. The old entry will then be replaced with the new MFN and its allocated ephemeral slot. We associate a reference count to each ephemeral entry so that the mapping can be replaced only when all owners have dropped the reference via unmap calls.

We consider several optimizations. To exploit temporal locality, we promote an entry to become *hot* when the same mapping has been requested repeatedly, preventing it from immediate eviction even when all references are dropped. To increase map cache performance, we explore batch invalidation, superpage caches and set associativity to reduce the cost of local TLB flushes, large region mappings and collision cache misses respectively. These caching optimizations must not expose additional side channels. For example, no ephemeral mapping to other domains can be cached or promoted as *hot* entries under the context of an unprivileged domU. This guarantees that cached entries and map cache contention are only caused by the domain itself, thus an attacker is unable to reveal secrets by probing the timing of ephemeral mappings or by speculatively accessing the cached entries.

We explore the design space of several parameters of the cache (elaborated in the Evaluation section) to achieve a high hit rate to amortize ephemeral mapping costs. With an optimal set of parameters, we are able to achieve a hit rate of 80-90% in our implementation, greatly reducing the cost of ephemeral access from the hypervisor. A high-performance map cache proves that the hypervisor address space can be minimized. Switching to a full address space like KPTI or XPTI with a direct map is unnecessary for efficiency.

F. μarch isolation

We do not propose new defenses against pure μarch sniffing attacks, but we do not exclude this category from the threat model for the secret-free hypervisor. As the overhead of secret freedom is small, we are able to compose it well with other known mitigations, including core scheduling and μarch buffer flushing on context switch, to mitigate all categories in the threat model. Note that this is also necessary for attacks that combine coercion and μarch sharing. An attacker may mistrain a sibling vCPU thread from another domain to fill the shared L1 cache with secrets before launching L1TF. Without μarch isolation, secret freedom would be unable to prevent this attack combination.

G. Putting it all together

Fig. 5 shows the address space of a vCPU. It separates the address space into multiple tiers of secret levels. At the global non-secret level, only the hypervisor image and non-secret data are visible. The next tier, domain secrets, is shared with all vCPUs of the same domain. Next, the hypervisor stack, register frames and other vCPU secrets appear at the vCPU-private level. Ephemeral mappings and the map cache reside in vCPU-private range, hidden from other vCPUs and domains while the hypervisor is temporarily accessing memory. These provide a minimal address space that is secret-free.

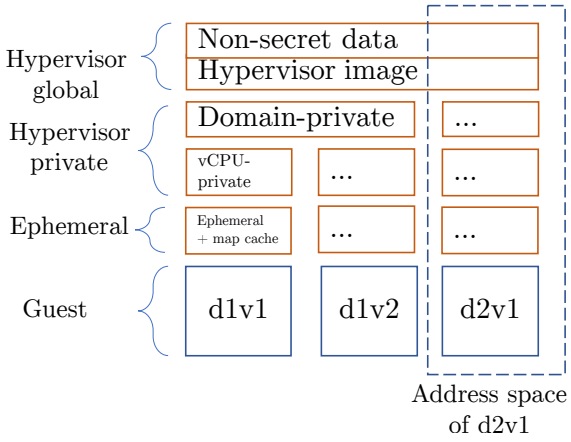

Fig. 5. Secret-Free address space separation.

There are two major differences with state-of-the-art techniques. First, the minimal address space is maintained at all times. We never expose the full hypervisor space that contains secrets belonging to other domains, unlike PTI techniques. The hypervisor has the same restricted address space as the guest, and only creates ephemeral mappings when necessary. Second, we adopt an allow-list approach by identifying and promoting non-secrets. Data is accessed via ephemeral mappings by default. Long-lived objects are added to the vCPU-private level first. We promote memory to be visible within a domain or globally only when it both does not violate secret freedom and is performance critical. Our approach does not identify secrets that need to be hidden. Instead, it identifies performance-critical non-secrets that should be shared.

H. Secret-freedom as a generic design principle

The recent epidemic of speculative vulnerabilities motivates the secret-free hypervisor design as we would like to introduce a comprehensive framework for isolating customer secrets in a multi-party cloud environment. However, we believe the components introduced in this section are not unique to any specific hypervisor and can be easily extrapolated to a variety of implementations. For OS kernels, a secret-free design applies as well because the abstraction of kernel, user space, processes and threads are analogous to hypervisor, guest domain, VMs and vCPUs.

To demonstrate the generality of the secret-free principle, we implement and evaluate the design on multiple systems including Xen (Type-I), Hyper-V (Type-I), bhyve (Type-II) and FreeBSD (UNIX kernel). We are able to apply common secret-

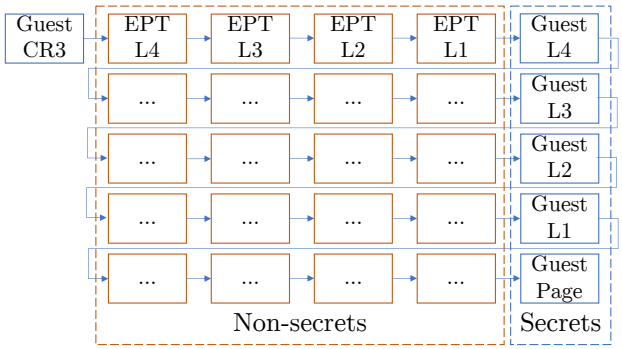

Fig. 6. A 2-stage page table walk. 20 ephemeral mappings are avoided by promoting EPT to global non-secrets.

free design components to all systems and only introduce minor changes to each. In this paper, we first elaborate on the Xen hypervisor for detailed evaluation and analysis while later demonstrating other implementations for comparison, showing secret-free as a generic mechanism as well as focusing on necessary adaptations for each type of kernel.

VI. Implementation: A Secret-Free Xen

We implemented the secret-free hypervisor in Xen 4.14.0 on x86_64 architecture. We made several necessary adaptations to the Xen codebase in addition to the aforementioned secret-free components.

1) Xen requires a direct map under 4GiB during early boot. We implemented a lightweight mapping mechanism by reserving 5 `fixmap` entries (for up to 5-level paging) to bootstrap Xen and to set up initial address spaces, which is superseded by the per-vCPU ephemeral mapping infrastructure once bootstrapping is done.
2) We replaced Xen's `domain_page()` mapping API with per-vCPU ephemeral mappings and revealed three bugs that resulted from unbalanced map/unmap calls, off-by-one-page unmap, and unmapping global memory. The bugs did not manifest when the API was backed by the direct map, as unmap calls on the direct map were compiled into no-ops.
3) For Xen HVM guests with EPT [1], we further identified EPT pages as performance-critical non-secrets that should be globally mapped for efficient 2-stage page table walks.

In 3), walking 2 stages of page tables in the hypervisor requires up to 25 ephemeral mappings, shown in Fig. 6. This creates high and unpredictable latencies depending on the map cache hit rate. The performance of 2-stage walks is critical for Xen because memory arguments in Xen hypercall ABI are passed as Guest Virtual Addresses (GVA), which need to be manually walked by the hypervisor before copying memory. This is a legacy of Xen's original fully paravirtualized mode, where pseudo-physical addresses (guest physical addresses) were an ephemeral concept. We promoted EPT to the global non-secret pool to avoid 20 ephemeral mappings. Note that we cannot treat the 5 guest page table pages as non-secrets as they are guest memory and may contain secrets at any point. The amount of code changed is shown in TABLE I.

[1]We use EPT (Extended Page Table) as a generic term for the second stage page table. On AMD systems such pages are named NPT (Nested Page Table).

Items	LoC
Introducing private and ephemeral APIs	821
Removing other dependencies and direct map teardown	411
Domain/vCPU-private region for stacks, vCPU state, etc.	729
Per-vCPU ephemeral mapping infrastructure	284
Bootstrapping for Secret-Free	115
Bug fixing, misc.	55
Total	**2415**

TABLE I
Lines of code for Secret-Free Xen

We have sent the first patch series for new APIs and direct map teardown to Xen upstream for review. At present, 40 out of 54 patches have been merged into the latest main branch. We reported the bugs revealed by Secret-Free to Xen upstream and our fixes have been merged.

VII. Evaluation of Secret-Free Xen

A. Experimental setup

We evaluate our secret-free Xen implementation on an AMD system, featuring a 12-core (24-thread) Ryzen 5900X CPU, 32GB of DDR4 3200MT/s RAM running Ubuntu 18.04 as dom0. We approximate a common cloud configuration using a guest with 8 vCPUs and 16GB RAM, matching an Azure A8v2 or AWS c4.2xlarge instance.

We evaluate performance with a range of benchmark suites. We show CPU and memory benchmarks using the industry standard SPEC-CPU2017 suite. To reveal the worst case scenarios and give insights on how the critical path is affected, we run micro-benchmarks to show hypercall latency, context switch speed, IPI latency and MMIO performance. We then analyse disk and network I/O to investigate cross-domain communication between domU front-end and dom0 back-end PV drivers. Lastly, we run real-world workloads representative of a wide range of cloud applications including databases, HTTP servers, decompression, kernel builds, scientific computing, etc.

We build several Xen configurations with different mitigation options:

Baseline: Xen without any compiled-in speculative mitigation facilities and with boot-time speculative defenses disabled. The baseline is susceptible to all speculative execution attacks that the underlying hardware is vulnerable to.

Default: Xen with compiled-in mitigation support and with default boot-time mitigations by detecting the hardware. On the 5900X CPU, this enables IBPB, `lfence` for indirect branches, conditional branch hardening and core scheduling.

XPTI: force enabling Xen Page Table Isolation for Meltdown mitigation in addition to default Xen parameters. PV only.

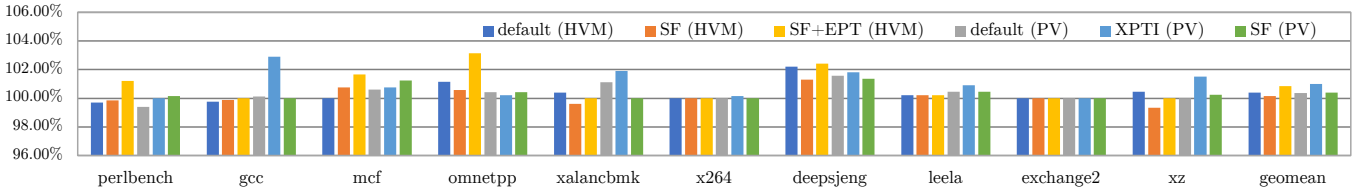

Fig. 7. SPEC-CPU 2017. Normalized execution time

HVM (ns)	hypercall reg	overhead	hypercall mem	overhead	context switch	overhead	IPI	overhead
Baseline	303.97		776.15		1832.95		2555.25	
Default	348.2	14.55%	1066.60	37.42%	2362.04	28.87%	2577.50	0.87%
SF	309.86	1.94%	780.26	0.53%	2392.78	30.54%	2602.62	1.85%
SF+EPT	312.19	2.70%	995.50	28.26%	2420.05	32.03%	2696.26	5.52%
PV (ns)								
Baseline	39.60		83.81		1214.49		2584.74	
Default	83.49	110.83%	131.77	57.22%	1660.24	36.70%	2686.06	3.92%
XPTI	306.40	673.74%	366.12	336.85%	1861.93	53.31%	2703.18	4.58%
SF	39.87	0.68%	84.35	0.64%	1740.84	43.34%	2592.96	0.32%

TABLE II
MICRO-BENCHMARKS

Frequency (MHz)	Baseline	Default	SF	SF+EPT
HVM HPET	6.24	4.70	6.04	5.40
Slowdown	-	24.68%	3.26%	13.50%

TABLE III
HPET BENCHMARK. MAX FREQUENCY FROM GETTIMEOFDAY().

Secret-Free: Xen with the secret-free implementation. Address space isolation and branch hardening options are disabled. However, mitigations for μarch sharing (core scheduling and μarch flushing on full domain switch) are kept.

SF+EPT: Secret-free with EPT as guest secrets. Although EPT can be treated as non-secrets by our definition, this setup stresses 2-stage page table walks and allows us to test the ephemeral map cache.

Note that our setup (Ryzen 5900X) is not susceptible to Meltdown, but we choose to add this data point to study the impact of XPTI on Xen PV guests. Xen PV is still supported by multiple cloud vendors on older generation hardware which is vulnerable to Meltdown, thus XPTI must still be enabled for isolation on these platforms.

B. SPEC-CPU 2017

Fig. 7 shows the performance impact of different configurations of mitigations when running SPEC-CPU 2017. Compared with the unmitigated baseline, the slowdown is under 3% for all benchmarks.

Such results are within our expectation as SPEC-CPU benchmarks are CPU- and memory-intensive. These workloads remain largely unaffected by hypervisor mitigations because they require few transitions into the hypervisor. We observe that the geomeans of different setups deviate from the baseline by less than 1.01%. XPTI and SF+EPT have the largest drop at 1.01% and 0.86% due to the increased latency in hypercalls whereas all other configurations are below 0.5%, showing little performance impact.

C. Micro-benchmarks

We show four micro-benchmarks: hypercall register, hypercall memory, context switch and IPI performance. Register hypercalls contain input and output arguments purely in registers, whereas memory hypercalls require mapping and copying guest memory. For context switch, we launch two guest vCPUs from two domains and schedule both on the same host core. The IPI benchmark measures the synchronous operation of a global IPI sequence, with its arguments placed on the caller's stack in the baseline configuration. These micro-benchmarks are selected specifically to stress the paths hardened by the SF implementation.

TABLE II shows the overhead of the secret-free design relative to existing Xen mitigations. We see that the secret-free version has shown near-baseline performance in several categories. The overhead of both register and memory hypercalls is negligible. If EPT is treated as guest memory secrets, HVM memory hypercalls see a degradation of 28.26% due to 25 total ephemeral maps from 2-stage page table walks. This explains our desire to promote EPT to global non-secrets. For default Xen mitigations, the overhead comes from restricted speculations and explicit branch predictor barriers which more than double the latency of a PV register hypercall, while the percentage is not as significant in HVM since the majority of the cost is VMENTER and VMEXIT. On the other hand, XPTI shows detrimental penalties for PV guests with more than 6× hypercall latencies due to two `cr3` swaps, bringing its hypervisor round trip cost at a comparable level with a full VMENTER and VMEXIT.

The secret-free implementation shows noticeable degradation in full context switches. The overhead comes primarily

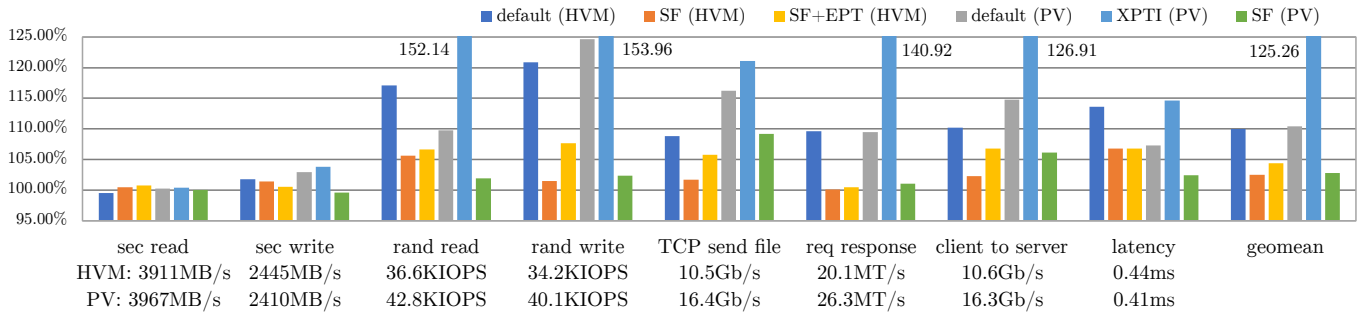

Fig. 8. Disk and network I/O benchmarks. Sequential and random disk accesses are tested by FIO. Netperf measures TCP performance and latency. The numbers on the X axis indicate baseline performance. Apart from TCP latency, the Y axis represents normalized execution time of fixed-size workloads.

from three sources. First, as secret-freedom alone does not address the problem of guests sharing micro-architectural state, we still flush μarch buffers during a full context switch to 1) prevent the next guest from sniffing secrets and to 2) isolate the next guest against malicious states like mistrained branch predictors. This cost only occurs when switching to a vCPU of a different domain. Second, moving to per-vCPU stacks means a context switch also switches to a new hypervisor stack and mapping, resulting in increased cache and TLB misses. Last, using global buffers to grant visibility to the next stack reduces data locality. Existing Xen mitigations show slightly lower overhead due to the cache and TLB locality of the per-pCPU stack. XPTI, however, exhibits more than 50% additional context switch latency relative to baseline.

The IPI performance remains largely unchanged and the cost is dominated by the IPI itself rather than speculative mitigations. Sending the IPI synchronously shows high latency and variance. In comparison, the overhead of the secret-free hypervisor using global bounce buffers for IPI arguments is insignificant. The latency suggests why maintaining globally mapped memory (like direct map maintenance in XPFO) is costly unless hardware support is present: an IPI alone is more expensive than a full context switch for the caller and interrupts all callees.

HVM MMIO: We further micro-benchmark a more realistic setup by measuring the maximum number of `gettimeofday()` calls that can be issued per second from guest user space after switching the guest Linux kernel clock source to the High Precision Event Timer (HPET, mapped as an MMIO device). Emulated MMIO, like memory hypercalls, requires 2-stage walks to fetch the Guest Physical Address (GPA) from Guest Virtual Address (GVA), although Xen walks to Host Physical Address (HPA) to take a page reference. We choose HPET as it represents the performance of other MMIO emulations, tests the map cache with a mix of page table walking and guest kernel activity, and is used in guest OSes as a timer source.

The maximal achieved number of `gettimeofday()` calls per second (denoted as Frequency and measured in MHz) is shown in TABLE III. With 2MB superpages used by default for HPET MMIO region, SF and SF+EPT map 4 and 19 pages respectively. Compared with baseline, SF only reduces the `gettimeofday()` frequency by 3.26% whereas default Xen mitigations degrade performance by 24.68%, which means the secret-free design offers significantly improved performance in device emulation.

D. I/O performance

Fig. 8 shows the overhead of disk and network I/O throughput. Sequential read and write saturate the backing storage, but random synchronous 4K I/O is bottlenecked by communication between the front- and back-end drivers. In these benchmarks, we see all secret-free configurations showing competitive performance close to baseline with the biggest drop being SF+EPT at 7.67%. In contrast, Xen mitigations degrade random synchronous I/O by more than 15%. XPTI again shows severe performance impact with its latency increased by 50%, resulting in 36% less IOPS.

We see an impact in TCP bandwidth by up to 2.26% in SF which increases to 6.8% when EPT is treated as domain secrets. XPTI is expectedly the worst performer, showing up to 41% overhead in bandwith and 15% increase in latency. Overall, SF demonstrates a combined geomean of all disk and network I/O overhead at 2.63%, almost one quarter of the 10.2% from enabling Xen default mitigations (excluding XPTI).

E. Application benchmarks

We further collect the results from a variety of real-world benchmarks, including Apache httpd, Linux kernel compilation, Numpy, LLVM compilation, LevelDB, Nginx, PostgreSQL (pgbench), SQLite and code repository decompression. The normalized execution time is shown in Fig. 9. Except for XPTI, the geomean of the overhead in each setup falls below 2.5%, which is insignificant compared with the stress tests in previous sections. This is expected as real-world workloads are a mix of CPU, memory and I/O activity.

For HVM, our LevelDB benchmark heavily tests random database mutation. The overhead tracks our I/O benchmarks well, with Xen default showing up to 11% increase in random write transaction latency. Averaging all read and write sub-benchmarks, default shows a LevelDB overhead

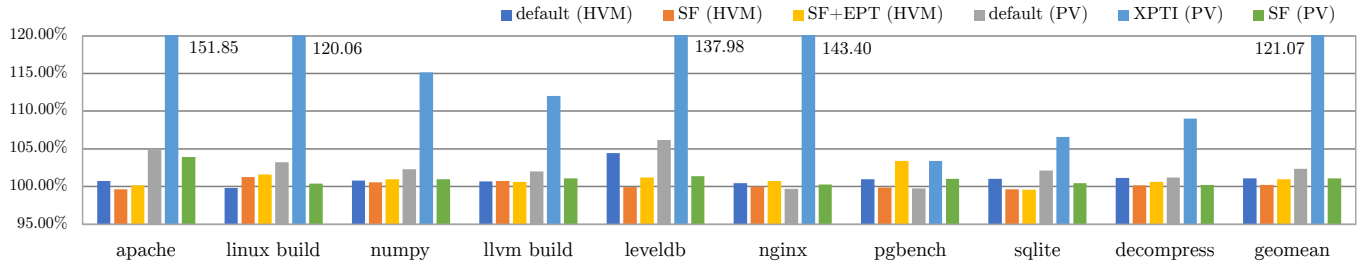

Fig. 9. Real-world application benchmarks. Normalized execution time.

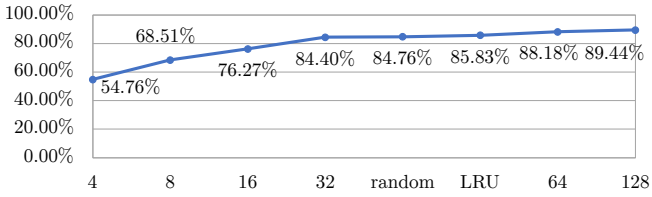

Fig. 10. Map cache hit rate versus size. Random and LRU denote 32-entry 2-way set associative caches with different eviction policies.

of 4.42%. The higher overhead from Xen default continues for other benchmarks with random disk I/O patterns, namely pgbench (0.97%), SQLite (1.00%) and code repo decompression (1.12%). On the other hand, the large variations of network performance across setups in Section VII-D do not manifest as their maximal bandwidths have all reached 10Gbps, meaning server benchmarks are still likely execution-bound than network-I/O-bound. As a result, no significant overhead is seen in HTTP server workloads. SF continues to show no noticeable degradation across all HVM benchmarks and has a overall geomean at only 0.2% versus the default at 1.11%.

XPTI is generally unusable with server workloads, seeing 50% overhead in HTTP servers and in write-intensive LevelDB benchmark. PV default mitigations without XPTI have a noticeably higher overhead due to doubled hypercall latency, although generally under 5%. For PV, the secret-free version consistently outperforms the default configuration, at less than half of the overall overhead. More importantly, our implementation mitigates Meltdown at a significantly lower cost than XPTI, because page table swaps are avoided on hypervisor entry and exit.

F. Map cache performance

We explore several map cache parameters including size, set associativity and block size (superpage cache entries). The test application is pgbench.

We notice that the dominant factor is the cache size (number of entries) as shown in Fig. 10. Even though the miss rate continues to decline as the size grows, in practice an oversized map cache begins to hurt guest performance due to CPU cache contention. We choose 32 entries as the optimal parameter for a balance between high hit rate and best overall application performance.

We further attempt to improve on the 32-entry cache by implementing set associativity (denoted by *random* and *LRU* in Fig. 10), which has a consistent but minor impact on hit rate. A randomly evicted 2-way set-associative cache shows negligible advantage over the direct-map one, whereas the Least-Recently-Used eviction policy sees a more noticeable improvement by 1.43%. Nevertheless, software implementation of set associativity requires branch instructions and is unable to exploit parallel lookups like hardware CPU caches, thus the minor increase in hit rate and negligible increase in application performance do not justify the complexity.

The block size can be grown by adding superpage (2MB) cache entries for large ephemeral mappings. However, we fail to observe any performance improvement as almost no ephemeral mapping request ever exceeds the 4K page size. Eventually, we choose a direct-mapped 32-entry map cache with only 4K mappings for the best overall performance and a simple but fast critical path, which is the configuration used throughout the evaluation section.

G. Security

We evaluate the security of our secret-free Xen hypervisor using proof-of-concept speculative execution exploits from the three categories (permission, coercion and μarch). The attacks are launched from an unprivileged domU. The victim contains a secret message buffer in a separate VM. We also evaluate against *ret2dir* by injecting a stack overflow vulnerability to Xen.

For Meltdown, we launch a malicious PV domU [2] to dump the victim buffer via the Xen direct map. For Spectre, the attacker attempts to mistrain the branch predictor, redirecting hypervisor speculative execution to access the secret message. For L1TF, the malicious domain sniffs for any secrets left in the L1 cache from the sibling thread. Redirecting code execution to the direct map is not effective as Xen marks it as non-executable. To exploit the *ret2dir* vulnerability, we replicate the attack for non-executable direct map in [25] using a stack-pivoting gadget, so that the hypervisor stack pointer can be pivoted to malicious frames on the direct map. Each

[2]Xen HVM guests with hardware assist run on EPT and cannot exploit Meltdown directly.

	Spectre-v1	Spectre-v2	Meltdown	L1TF	ret2dir
Baseline	✗	✗	✗	✗	✗
Default	✓	✓	✓*	✓	✗
SF	✓*	✓*	✓*	✓	✓

* Speculative execution is still permitted. However, no secrets can be fetched as only mappings to non-secrets exist.

TABLE IV
VULNERABILITY MATRIX

attack is run on baseline (un-mitigated), default mitigations and Secret-Free Xen (w/ μarch isolation).

As shown in TABLE IV, the secret-free hypervisor successfully thwarts any attack that attempts to circumvent permission or to mistrain the hypervisor for secret access. With μarch isolation, we further guard against L1TF. While stack-pivoting using the direct map synonym succeeds in baseline and in default for *retdir*, it fails under the secret-free Xen because the mapping is absent, triggering a hypervisor page fault on the first access.

From the evaluation, we are able to confirm that our Xen implementation correctly enforces the secret-free principle. The secret-free design, when compared to other mitigations, provides defence-in-depth. This guards against future attacks and undiscovered microarchitectural vulnerabilities of the same categories. Finding another means of information disclosure (either executed speculatively or architecturally) is not useful, unless an attack is successfully mounted first to bring secrets into the address space.

VIII. HYPER-V (TYPE-I)

We explore a secret-free Hyper-V as an independent implementation. We choose to focus on Xen for evaluation because it is open-source, which allows us to elaborate on code and implementation details. The secret-free Hyper-V has been deployed at scale to Azure, showing the applicability of secret freedom as a generic isolation framework for Type-I hypervisors.

The direct map has been removed to avoid the implicit mapping of secrets. We then leverage vCPU-private area to construct separate secret virtual address spaces for each guest thread. Hypervisor stacks, register frames and other vCPU state are now placed in private mappings and are only visible to the current context. Due to the isolation of secrets, the hypervisor requests ephemeral mappings to access guest memory or cross-domain secrets. Here, we find a major difference with secret-free Xen on map cache pressure. The hypercall ABI of Hyper-V takes GPA instead of GVA for arguments in guest memory, which avoids a full manual 2-stage page table walk. We introduce the same optimization as Xen by identifying EPT pages as performance-critical non-secrets and promoting them to the global non-secret pool. As only one temporary mapping is required to perform a GPA to HPA walk for hypercalls, we choose not to implement the map cache to avoid contending with the guest for CPU data caches.

Our platform exposes nested virtualization capabilities on certain types of instances. A customer may launch L2 guests belonging to multiple security domains in the L1 hypervisor. To ensure a strong isolation boundary for different domains within a guest VM, we further implement state scrubbing. When the hypervisor must copy guest secrets to complete an operation, it overwrites buffers with zeroes prior to exiting the L0 context. This guarantees that secrets from L1 guest hypervisor or L2 guest virtual processor state are not resident in the cache when switching between security domains in the L1 guest VM. We minimize the overhead by carefully tracking the memory which needs to be scrubbed.

The secret-free design in Hyper-V makes several existing mitigations redundant. Although core scheduling is required for μarch isolation, we no longer ask the scheduler to coordinate sibling entry and exit because secrets can no longer be fetched even in hypervisor context. Similarly, exiting the hypervisor no longer needs to flush the L1D cache. We also remove unnecessary guards against poisonous guest branch predictor state on hypervisor entry as it cannot cause any caching of secrets.

Together, we see an overall overhead of the SF implementation at only 1% (with secret data scrubbing). The implementation has been proven in production and we have not received any reports from customers on performance degradation.

IX. BHYVE (TYPE-II)

We apply Secret-Free to bhyve, a Type-II hypervisor from the FreeBSD OS. In this work, one challenge prevents us from declaring our bhyve implementation secret-free. Type-II hypervisors reuse host device drivers to feed PV devices with data. Unfortunately, data buffers in host drivers are allocated and cached as kernel structures, globally mapped into all processes. I/O buffers are copied to and from guest memory, which violate Secret-Free by our definition.

We may implement such buffers via thread- or process-private mappings visible only to the driver daemons, but the challenge remains. Unlike Type-I hypervisors, we find it tremendously difficult to introduce private or ephemeral APIs to all I/O buffers due to components being closely-intertwined in a monolithic kernel, which requires a substantial amount of code rewrite. Thus, we think a secret-free design would be less intrusive for Type-I hypervisors or micro-kernels. However, even before restructuring the kernel for back-end isolation, a mostly secret-free design still exposes a much reduced attack surface and provides the same guarantees as Type-I implementations when guest I/O is hidden (for example, by using passthrough devices or enabling disk encryption, which are common on cloud platforms).

X. FREEBSD (UNIX KERNEL)

The bhyve implementation applies the secret-free changes to the underlying FreeBSD kernel. The same challenge for bhyve remains, which is the difficulty in rewriting the API for all globally mapped I/O buffers and buffer caches in a monolithic kernel. Further, we identify one type of workloads that needs optimization.

Slowdown	Process creation	FreeBSD kernel build
SF	38.44%	3.93%
Optimization	8.16%	0.85%

TABLE V
PROCESS CREATION OVERHEAD.

Processes and threads are significantly more dynamic than VMs and can be created and destroyed at high frequencies, to the point where it determines the application performance. Such a high rate overwhelms the map cache due to the huge number of page table copies during `fork()`. Unfortunately, mapping page table pages globally like Xen EPT optimization is unhelpful. It reduces the run-time cost when accessing the pages but increases the overhead during process destruction, because IPI broadcasts are issued for TLB invalidation for changes in the global non-secret mapping range. We have observed a staggering slowdown of 38% in process creation micro-benchmark and 4% in FreeBSD kernel compilation (TABLE V), evaluated under baremetal FreeBSD on the same hardware setup as Xen.

We implement a proof-of-concept optimization by reserving part of the kernel memory for non-secrets. A direct map range is reintroduced, but only covers the non-secret pool. The pool is globally and permanently mapped by the kernel during early boot for page table pages. TABLE V shows that the optimization brings kernel build overhead below 1%. However, future work is needed to dynamically adjust the size of the pool (and the non-secret direct map) when the system is under memory pressure because the ratio of the reservation cannot be optimally pre-determined. We have one extreme where the system may contain lots of copy-on-write pages (high page table to user memory ratio) and another extreme where memory is all populated with superpage mappings (low ratio).

XI. RELATED WORK

A. eXclusive Page Frame Ownership

XPFO targets the Linux direct map to reduce the kernel attack surface. Memory allocated to user processes will be excluded from the direct map, guaranteeing unique ownership of the mapping. No kernel synonyms can be used for a user space gadget.

This approach suffers from the overhead of global map maintenance, because pages need to be scrubbed before being added back to the direct map and TLB shootdowns need to be broadcast when pages are allocated to user. In contrast, a secret-free design adopts an allow-list rather than a deny-list approach. It has a minimal secret-free address space at all times and its isolation is not limited to the direct map. Memory is only temporarily mapped in while the kernel accesses it, and the per-thread mapping infrastructure avoids TLB shootdown and their associated scalability issues.

B. KPTI and XPTI

KPTI and XPTI address the isolation failure in CPU speculation paths, where speculative instructions are not restricted by current privilege level. A partial page table is created for user processes that removes most of the kernel address range. System calls (or hypercalls) first jump to trampoline code to restore the full kernel mapping, and switch back to the partial table when exiting kernel space.

Our work demonstrates that a full kernel map in most (if not all) situations is unnecessary, as the cost of two page table swaps is intrusively expensive. A secret-free design either handles secret access in private mappings, or uses ephemeral mappings for guest memory and for temporary access outside the current address space. Our evaluation has shown that the kernel map does not have to be restored in its entirely for the kernel to function efficiently.

C. Linux Kernel Address Space Isolation

As more kernel isolation requirements emerge to defend against speculative vulnerabilities, Linux proposes a generic Kernel Address Space Isolation framework (KASI) [31]. For KVM isolation, it handles VMEXITs in a separate address space that exposes only the per-VM structures and Linux code and data, similar to the secret-free address space.

However, its secret-free address space is only limited to simple VMEXIT handling. Any access outside the minimal address space switches to the full kernel and halts sibling thread to ensure the kernel cannot be mistrained into bringing secrets to the shared L1 data cache. At the moment, the performance of the framework is unknown.

D. Corevisor

Corevisor partitions the monolithic hypervisor into a trusted core and a large untrusted hostvisor. Several kernel components including vCPU scheduling and the memory allocator are deprivileged into the hostvisor for a minimal TCB. For PV I/O devices, the back-end drivers reside within the hostvisor instead of the privileged core [39].

We welcome such separation of the hypervisor components. SF has seen tremendous obstacles in introducing private APIs to a Type-II hypervisor atop a monolithic kernel, because every kernel module and device driver may potentially need source-code modification. We anticipate that a corevisor design, with clear separation among its components (especially the ones interacting with guest secrets), can be easily retrofitted with secret-free abstractions for defense against a broad category of architectural and speculative attacks.

XII. CONCLUSION

We have presented the secret-free hypervisor, a design atop existing hardware for a minimal and secret-free address space. We have shown that this technique is a defence-in-depth approach against several categories of vulnerabilities. The optimizations bring the performance to a similar level with an unmitigated baseline. Compared with state-of-the-art mitigations, we improves performance by permitting speculations in situations where secrets are invisible. Our implementations suggest that the Secret-Free approach is applicable to multiple platforms and different types of kernels, as a performant

alternative to existing mitigations and as a comprehensive framework against potential future attacks.

References

[1] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey, and J. A. Halderman, "The Matter of Heartbleed," in *Proceedings of the 2014 Conference on Internet Measurement Conference*, ser. IMC '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 475–488. [Online]. Available: https://doi.org/10.1145/2663716.2663755

[2] A. C. Mary, "Shellshock Attack on Linux Systems – Bash," in *International Research Journal of Engineering andTechnology*, vol. 2. Tamilnadu, India: IRJET, 2015, pp. 1322–1325.

[3] B. Bierbaumer, J. Kirsch, T. Kittel, A. Francillon, and A. Zarras, "Smashing the Stack Protector for Fun and Profit," in *SEC*, 2018.

[4] "CVE-2021-33910," last visited on 2021-10-12. [Online]. Available: https://access.redhat.com/security/cve/cve-2021-33910

[5] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning, "On the Expressiveness of Return-into-Libc Attacks," in *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection*, ser. RAID'11. Berlin, Heidelberg: Springer-Verlag, 2011, p. 121–141.

[6] E. Zannoni, "Improving Application Security with Undefined-BehaviorSanitizer (UBSan) and GCC," May 2021.

[7] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the Effectiveness of Address-Space Randomization," in *Proceedings of the 11th ACM Conference on Computer and Communications Security*, ser. CCS '04. New York, NY, USA: Association for Computing Machinery, 2004, p. 298–307. [Online]. Available: https://doi.org/10.1145/1030083.1030124

[8] N. Nethercote and J. Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," *SIGPLAN Not.*, vol. 42, no. 6, p. 89–100, Jun. 2007. [Online]. Available: https://doi.org/10.1145/1273442.1250746

[9] B. Anderson, L. Bergstrom, D. Herman, J. Matthews, K. McAllister, M. Goregaokar, J. Moffitt, and S. Sapin, "Experience Report: Developing the Servo Web Browser Engine using Rust," 2015.

[10] T. Anderson, "Linux in 2020: 27.8 million lines of code in the kernel, 1.3 million in systemd," Jan 2020. [Online]. Available: https://www.theregister.com/2020/01/06/linux_2020_kernel_systemd_code

[11] "Vulnerability details : CVE-2017-11176," Jul 2017. [Online]. Available: https://www.cvedetails.com/cve/CVE-2017-11176/

[12] "Vulnerability details : CVE-2021-43267," Nov 2021. [Online]. Available: https://www.cvedetails.com/cve/CVE-2021-43267/

[13] "Vulnerability details : CVE-2021-43057," Oct 2021. [Online]. Available: https://www.cvedetails.com/cve/CVE-2021-43057/

[14] "Vulnerability details : CVE-2021-41073," Sept 2021. [Online]. Available: https://www.cvedetails.com/cve/CVE-2021-41073/

[15] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," 2018.

[16] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading Kernel Memory from User Space," in *Proceedings of the 27th USENIX Conference on Security Symposium*, ser. SEC'18. USA: USENIX Association, 2018, p. 973–990.

[17] "Page Table Isolation (PTI)." [Online]. Available: https://www.kernel.org/doc/html/latest/x86/pti.html

[18] 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*, June 2021.

[19] L. Kurth, "What's new in xen 4.13," Dec 2019. [Online]. Available: https://xenproject.org/2019/12/18/whats-new-in-xen-4-13/

[20] N. Nethercote and J. Seward, "Retpoline: A Branch Target Injection Mitigation," no. 3, Jun. 2018.

[21] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, "InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 428–441.

[22] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, 2019, pp. 1–6.

[23] E. M. Koruyeh, S. H. A. Shirazi, K. N. Khasawneh, C. Song, and N. B. Abu-Ghazaleh, "SPECCFI: Mitigating Spectre Attacks using CFI Informed Speculation," *CoRR*, vol. abs/1906.01345, 2019. [Online]. Available: http://arxiv.org/abs/1906.01345

[24] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis, "kGuard: Lightweight Kernel Protection against Return-to-User Attacks," in *21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA: USENIX Association, Aug. 2012, pp. 459–474. [Online]. Available: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/kemerlis

[25] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis, "ret2dir: Rethinking Kernel Isolation," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 957–972. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/kemerlis

[26] D. Seal, *ARM Architecture Reference Manual*, 2nd ed. USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[27] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, p. 991–1008. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/bulck

[28] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "RIDL: Rogue In-flight Data Load," in *S&P*, May 2019.

[29] H. Ragab, A. Milburn, K. Razavi, H. Bos, and C. Giuffrida, *CrossTalk: Speculative Data Leaks Across Cores Are Real*. United States: Institute of Electrical and Electronics Engineers Inc., Jun. 2021, pp. 1–16.

[30] "Hyper-V hyperclear mitigation for L1 Terminal Fault," Mar 2019. [Online]. Available: https://techcommunity.microsoft.com/t5/virtualization/hyper-v-hyperclear-mitigation-for-l1-terminal-fault/ba-p/382429

[31] G. Marsden, "Improve Security with Address Space Isolation (ASI)," Jul 2019. [Online]. Available: https://blogs.oracle.com/linux/post/improve-security-with-address-space-isolation-asi

[32] X. J. Ren, K. Rodrigues, L. Chen, C. Vega, M. Stumm, and D. Yuan, "An Analysis of Performance Evolution of Linux's Core Operations," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 554–569. [Online]. Available: https://doi.org/10.1145/3341301.3359640

[33] "Dom0," Mar 2015. [Online]. Available: https://wiki.xenproject.org/wiki/Dom0
[34] J. Corbet, "Virtually mapped kernel stacks," Jun 2016. [Online]. Available: https://lwn.net/Articles/692208/
[35] B. Armstrong, "Plan for Hyper-V scalability in Windows Server," Nov 2021. [Online]. Available: https://docs.microsoft.com/en-us/windows-server/virtualization/hyper-v/plan/plan-hyper-v-scalability-in-windows-server
[36] "VMware vSphere ESX and vCenter Configuration Maximums." [Online]. Available: https://www.virten.net/vmware/vmware-vsphere-esx-and-vcenter-configuration-maximums/
[37] "Configuration limits," Dec 2020. [Online]. Available: https://docs.citrix.com/en-us/xenserver/7-1/system-requirements/configuration-limits.html
[38] AMD, *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, March 2021.
[39] S.-W. Li, J. S. Koh, and J. Nieh, "Protecting Cloud Virtual Machines from Hypervisor and Host Operating System Exploits," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1357–1374. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/li-shih-wei

APPENDIX

We first list examples of how SF blocks various categories of speculative vulnerabilities and how it compares to state-of-the-art mitigations. Then, we demonstrate how the self-map technique allows accessing Page Table Entries (PTEs) without manual page table walking.

A. Guest-to-hypervisor attacks

This category reveals secrets by entering a speculatively-manipulated hypervisor context.

1) Spectre: Consider the following hypervisor code gadget.

```
if (x < array1_size)
  y = array2[array1[x] * 256];
```

An attacker is able to mis-train the hypervisor branch prediction to take the branch even if `x` is out of bounds. If `x` is controlled by the attacker, the speculative overflow may fetch any memory visible in the hypervisor space into the cache. The direct map is often abused by such attacks to access arbitrary physical memory.

Existing mitigations aim to prevent out-of-bound memory accesses by restricting speculation. Common measures include CPU fences, or array masks that guarantee in-bound indices.

```
unsigned long array_index_mask_nospec(
    unsigned long index,
    unsigned long size) {
    return ~(long)
    (index | (size - 1UL - index))
        >> (BITS_PER_LONG - 1);
}
```

The above function generates a $\sim$0 mask when index $<$ size, 0 otherwise. The mask sanitizes the user-controlled index to be in bounds even in speculation.

Instead of restricting the source of speculation, SF restricts the target, i.e., the hypervisor space itself. The array index is permitted to be speculatively out-of-bounds, but it is unable to dereference any secrets as no secrets to other domains exist in an SF hypervisor. This holds for Spectre-V2 as well. SF does not block indirect branches via retpolines or hardware branch predictor flushes, but instead guarantees that no secrets are visible to a gadget.

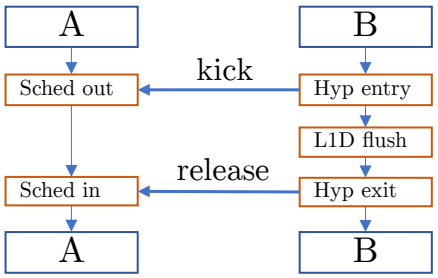

Fig. 11. Core Scheduler under a non-SF hypervisor

2) L1TF: A basic implementation of Core Scheduling prevents vCPUs of different domains from being scheduled on sibling hyperthreads. However, this is insufficient under the presence of guest-to-hypervisor attacks. Assuming both vCPU A and B are from the same guest VM scheduled on sibling threads, vCPU B could enter the hypervisor while vCPU A is still in guest mode, actively mis-training the shared branch predictor or sniffing secrets in L1D cache. Fully mitigating against this attack requires coordinating sibling entry and exits shown in Fig. 11. The hypervisor kicks the sibling thread A out of guest execution until returning to guest mode on B. Before returning to B, the hypervisor flushes the L1D cache to ensure no secrets remain for possible L1TF attacks from either A or B.

Core Scheduling is still required under an SF hypervisor. However, this guest-to-hypervisor attack surface is eliminated as the hypervisor space contains no secrets outside the VM which A and B belong to. Entering the hypervisor from one sibling no longer requires kicking the other out of guest execution, and returning to guest no longer needs an L1D flush. This shows that mitigating against attacks in the μarch category is significantly simpler than state-of-the-art thanks to an SF hypervisor.

B. Guest-to-guest attacks

This category reveals secrets directly from guest mode without entering the hypervisor.

1) Meltdown: Meltdown circumvents page table permissions in the speculative path to reveal secrets from other guests. Current mitigations on affected hardware use PTI to remove hypervisor page table entries on guest entry, and restores the full hypervisor table on hypervisor entry. In this regard, SF addresses Meltdown similarly to PTI by eliminating secrets to other domains when running in guest mode. However, SF stays on the secret-free address space on hypervisor entry, which avoids the prohibitively expensive page table swaps.

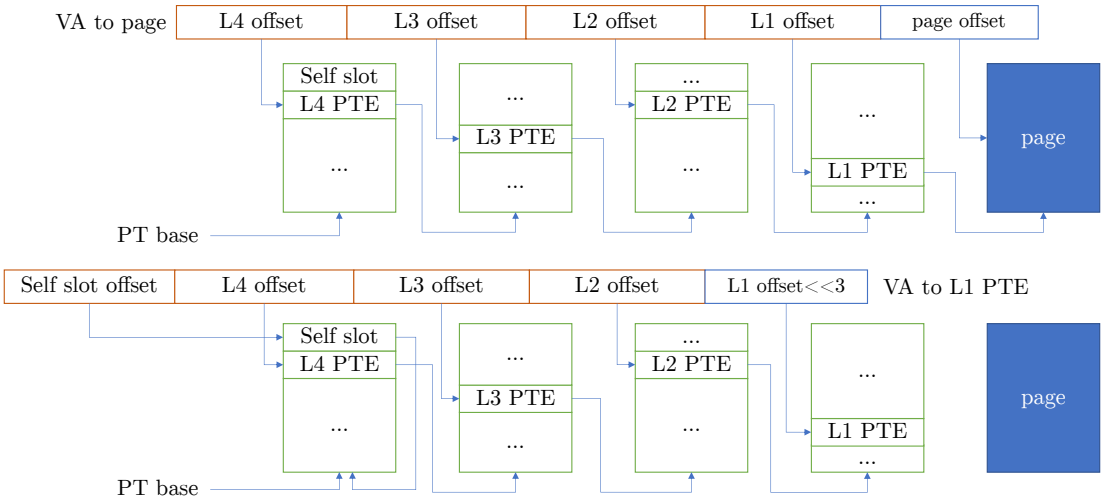

Fig. 12. Accessing a PTE using self mapping. *VA to page* indicates a pointer to memory, whereas *VA to L1 PTE* is a bitwise transformation that points to the L1 PTE. This functions only if a top level PTE pointing to itself is reserved.

2) Guest-to-guest Spectre and L1TF: **This category is not mitigated by an SF address space.** Two vCPUs from different domains may be scheduled on the same host CPU, sharing μarch structures including branch predictors and caches. An attacker may manipulate μarch state to affect the speculation path of the sibling victim, allowing secrets to be placed on covert channels. It is impossible to prevent this direct guest-to-guest attack without isolating the underlying hardware. Therefore, this work reuses several physical isolation techniques, mostly Core Scheduling, to prohibit different domains from sharing physical CPUs to mitigate attacks in this category. As stated in the guest-to-hypervisor L1TF example, physical isolation is less complex under the SF design, since its implementation no longer needs to guard against leakage from the hypervisor.

C. Page table self mapping

Hypervisors and OS kernels often perform manual page table walks to locate and manipulate page table entries for modifications of an address space. This can be done efficiently when the direct map is present because there exists a direct map alias of any arbitrary physical address, allowing us to quickly access the $L(n-1)$ page table page after decoding the physical address pointing to it in the PTE of the Ln level. However, kernel code often employs self mapping to avoid manual page table walking entirely, by reserving a top level PTE pointing to the top level table itself. Then, bitwise transformations allow direct access to PTEs.

An example is given in Fig. 12. Here, we assume four-level paging under a virtual address space of 48 bits with 4KiB pages. A pointer is comprised of four 9-bit page table page offsets and a final 12-bit page offset. By reserving a PTE in the top level and pointing it to the top level itself, a fixed transformation

```
uintptr_t L1_PTE_from_VA(uintptr_t va) {
    uintptr_t ret = va >> 12 << 3;
    // Insert SELF_OFFSET into L4 bits.
    ret[47:39] = L4_SELF_OFFSET;
    return ret;
}
```

returns a pointer to the L1 PTE (denoted by *VA to L1 PTE* in Fig. 12) of a given VA (denoted by *VA to page*). This works by asking the hardware page table walker to walk L4 twice, thus the final access will terminate at the L1 PTE inside the page table page instead of the actual physical memory. Note that this can be changed to access the L2 PTE as well, by shifting the given VA further right and insert `SELF_OFFSET` to L3 offset bits, effectively asking the hardware walker to walk the L4 table three times before terminating at the L2 PTE.

Self-map is used to enable ephemeral mapping infrastructure. When a new mapping request to a physical address comes, an ephemeral VA is allocated and its L1 PTE is located via `L1_PTE_from_VA()`. The PTE is modified to point to the requested physical address and the VA is now usable. The motivation of using this technique for ephemeral mappings under a secret-free hypervisor is not just efficiency. It is also simply because the direct map is no longer present and we cannot manipulate PTEs in the ephemeral mapping region by a normal manual page table walk.