Core slicing: closing the gap between leaky confidential VMs and bare-metal cloud

Ziqiao Zhou[*] Yizhou Shan[†] Weidong Cui[*] Xinyang Ge[*‡] Marcus Peinado[*] Andrew Baumann[*§]
[*]*Microsoft Research* [†]*University of California, San Diego*

Abstract

Virtual machines are the basis of resource isolation in today's public clouds, yet the security risks of entrusting that isolation to a cloud provider's hypervisor are substantial. Such concerns have motivated hardware extensions for "confidential VMs" that seek to remove the hypervisor from the trusted computing base by adding a highly-privileged firmware layer that checks hypervisor actions, and supports memory encryption and remote attestation. However, the hypervisor retains control of resource management and observes associated guest actions including nested page table faults and CPU scheduling, and thus confidential VMs remain vulnerable to an ever-changing variety of hypervisor-level side channel attacks. Bare-metal cloud servers avoid such leaks, but remain a niche due to the high cost of dedicated hardware.

We observe that typical cloud VMs run with a static allocation of memory and discrete cores, and increasingly rely on I/O offload, thus negating the apparent need for a hypervisor and the fragile hypervisor/guest isolation boundary. Our design, *core slicing*, enables multiple untrusted guest OSes to run on shared bare-metal hardware. To ensure isolation without the complexity of virtualization, we propose simple hardware extensions that restrict guests to a static *slice* of a machine's cores, memory and virtual I/O devices, and delegate resource allocation to a dedicated management slice. We demonstrate practicality and evaluate performance with prototypes for RISC-V and x86.

1 Introduction

We are in the early stages of a new generation of trusted execution environment (TEE). Motivated by cloud workloads, their main new feature is the ability to run *"confidential VMs"* inside the TEE [13, 17, 53]. Driven by the demand for secure cloud computing in which the user need not trust the cloud provider [78], these TEEs are expected to see much wider adoption than their predecessors [50]. Although confidential VMs offer enhanced functionality, their security model and architecture are largely identical to earlier TEEs such as SGX. The user shares one or more processor cores with a powerful adversary who controls hardware resources. Processor extensions provide the TEE with private memory and a trusted "context switch" to prevent the administrator-adversary from directly breaking the confidentiality and integrity of the TEE.

However, the past decade has produced a broad and rapidly growing spectrum of attacks on this model [22, 25, 41, 44, 62, 65, 67, 68, 72, 82, 90, 98, 104, 110, 112–116], including the complete breakdown of SGX security on several occasions [112, 114, 115]. Most of these can be described as side-channel attacks. As §2.3 will describe, they take advantage of the fact that attacker and victim run on the same core and share a multitude of sometimes obscure microarchitectural components. The same risks [39, 44, 67, 68] exist in confidential VMs. Although today's confidential VM architectures remove privileges from the host hypervisor (e.g., the ability to read plaintext guest memory), it retains a large degree of control over guest execution, such as the ability to arbitrarily interrupt guests, leading inexorably to side-channel attacks.

It is this paper's thesis – backed by evidence from recent work on computer architecture [31, 96, 97] and security [60] – that these failures of TEE hardware are not isolated events of the past. More than a decade of microarchitectural optimizations have taken processor complexity to a level where it is practically impossible to reason about isolation boundaries within a core [60, 96]. This problem is not only the likely ultimate root cause of the known attacks, but it is also bound to result in periodic breakdowns of TEE security for the foreseeable future. Indeed, Intel takes the position that side channels "can't be eliminated" [48, 51, 56] but that it will provide mitigations as new vulnerabilities are found. While similar to the current approach to software vulnerabilities, this expects the users of confidential cloud computing (e.g., financial institutions and governments) to tolerate data leaks whenever a new side channel is discovered.

[†]Yizhou Shan is now at Huawei Cloud.
[‡]Xinyang Ge is now at Databricks.
[§]Andrew Baumann is now at Google.

In this paper, we present a much more robust TEE architecture, *core slicing*, that is realistic for infrastructure-as-a-service (IaaS) workloads. Rather than making confidential guests share cores with an adversarial hypervisor, we give the guest exclusive access to its own CPU cores. This moves the isolation boundary to the much more defensible and robust one between processor cores. We show that this boundary can be enforced with simple (and thus less fragile) hardware.

We observe that, although cloud guests may benefit from a VM-level execution abstraction, cloud providers do not exploit the full complexity enabled by hypervisor-based virtual machines for IaaS workloads. For example, although hypervisors support time-slicing VMs on shared cores, VMs offered by major public cloud providers including Amazon [11] and Azure [81] are sized at core granularity and scheduled on distinct physical cores [7, 76, 77]. Likewise, the memory allocated to guest VMs is static; techniques such as memory ballooning [118] or transparent page sharing [118, 124] are avoided. Cloud providers are also moving to reduce the overhead of I/O virtualization by offloading I/O processing to dedicated hardware [5, 8, 36]. To ensure that resources sold match those available, cloud providers limit oversubscription to only their own (first-party) VMs [28] or disable it entirely [7]. Overall, although today's cloud runs virtual machines, leading public cloud providers do so using an effectively static allocation of cores and memory. The hypervisor is relied upon for isolation, but it does so merely by partitioning platform resources.

By giving the guest exclusive access to CPU cores, core slicing eliminates the potential for the entire class of side channels where the attacker shares per-core microarchitectural resources with its victim. Moreover, we enforce this isolation boundary with a new hardware mechanism that is self-contained and simple enough to permit reasoning about confidentiality and integrity. Fully isolated guest OSes (or, potentially, guest hypervisors) run in their own *slice* of a machine. Each slice consists of a dedicated, static allocation of cores, memory, and directly-assigned I/O devices (e.g., the virtual functions of network and storage controllers); hardware ensures that the cores of a slice are sequestered such that they have no access to memory or I/O devices outside the slice, nor can they interrupt cores of other slices. Because *only* a single guest runs code on any given core, a huge class of microarchitectural side-channel leaks are out of scope, and continued innovation in complex microarchitectural performance optimizations is unhindered, since those cannot impact the TEE isolation boundary. While core slicing eliminates intra-core leakage, it does not prevent *cross-core* side channels such as CrossTalk [91] which will have to be addressed by other means. Nevertheless, we believe that obviating intra-core channels removes by far the largest and most serious part of today's side channel problem in terms of the number and seriousness of known attacks, granularity of sharing, and number of shared components.

Resource allocations are determined by a *slice manager* that runs on a dedicated core (ideally, a separate low-power processor), and is responsible for starting and stopping slices, but is otherwise untrusted by the guest. Guest kernels (or guest hypervisors) are enlightened if necessary to run within a slice, ensuring that they attempt only access to those named resources available to them. For example, a guest cannot assume that physical memory starts at address 0, nor that processors have contiguous IDs; in practice, modern kernels including Linux make no such assumptions, and it suffices to pass boot-time information on the accessible resources.

In the following §2, we elaborate on the security risks of confidential VMs, and explore the way VMs are deployed in the cloud today. Like Keller et al. [58] over 10 years ago, we find that hypervisors add significant needless complexity to the cloud's trusted computing base. However, their system No-Hype [107] did not protect VMs from the (fully trusted) cloud provider and relied on virtualization hardware to enforce isolation. The numerous attacks demonstrated since [31, 62, 97] showed that the security provided by this hardware is fragile [62, 96]. By contrast, core slicing maintains a strict separation between core processor logic that is performance-critical and thus complex, and the hardware that enforces isolation, which is not performance critical and simple. It also permits guests to run their own bare-metal hypervisors.

To grant guests bare-metal access to cores in a shared machine while still securely isolating them from one another raises a key design challenge: *without a more privileged software layer on the core, what can enforce isolation?* The key insight behind our design is that simple hardware techniques used to enable trusted execution features such as secure boot and remote attestation for an entire system [127] can be adapted and applied at the granularity of individual cores to help resolve this dilemma. Specifically, §3 contributes *lockable filter registers* and a *core-local secure reset* mechanism, and describes how they can be used to enable core slicing.

To test the practicality of our design, we build two prototypes. The first (§4) leverages RISC-V physical-memory protection (PMP) registers [93, §3.6] to run multiple isolated Linux slices. The other x86-based prototype (§5) lacks security but enables an evaluation (§6) showing that core slicing offers bare-metal performance without VM overheads, with a substantially smaller TCB, while closing side channels based on caches, page faults [128], and other intra-CPU resources. We also analyze traces from a public cloud to find that we can allocate physically-contiguous slice memory, and confirm that our extensions add minimal hardware cost to an existing design. Finally, §7 outlines a path to applying our design to more mainstream architectures, §8 covers related work, and §9 concludes.

2 Background and motivation

2.1 Hardware-accelerated virtualization

Since VMware first demonstrated the value of virtual machines on commodity platforms [23], hardware vendors have added features to progressively reduce the overhead of virtualization. Although first-generation VM hardware suffered poor performance [1], the gap closed to the point where today's cloud platforms rely fully on hardware support for CPU virtualization, with features such as nested paging and virtual APICs ensuring that many guest VMs now run with low (<5%) CPU overhead compared to bare-metal execution [2]. Unfortunately, this is not true of all VMs; a recent study by Teabe et al. [108] found that up to 30% of CPU time was consumed by virtualization overheads on memory-intensive workloads, even when using nested address translation with huge pages. Other studies reported similar or worse address translation overheads (even with huge pages) [3, 37, 88].

Besides CPU features, modern hardware also supports efficient virtualization of I/O, through advanced IOMMUs [12, 49] and single-root I/O virtualization (SR-IOV) devices [32, 63]. These enable low-overhead virtual I/O by permitting a single physical I/O adapter (such as a network interface or storage controller) to export multiple virtual PCI functions. These are configured by a hypervisor and assigned to guest VMs by installing appropriate IOMMU translations and interrupt mappings. A guest OS thus interacts directly with the device, without the software overhead of traditional I/O virtualization [117]. The hypervisor's role is reduced to configuring the virtual devices and mapping them to the guests, a slower (control path) operation generally performed at startup.

2.2 Confidential VMs

Notwithstanding attempts to reduce the size or attack surface of cloud hypervisors [27, 66, 105, 107, 121, 125], the cloud's trusted computing base is controlled by cloud providers and opaque to its users. Threats such as supply-chain attacks [85] and rogue employees (e.g., cloud administrators and developers) have alternatives to traditional cloud architecture [55, 106, 126]: new architecture extensions such as AMD SEV-SNP [13, 78], Intel TDX [53] and Arm Realms [17] seek to remove the hypervisor entirely from the guest's TCB by extending the approach of earlier user-level TEEs such as Intel SGX [50]. In these designs, guest memory and register context are encrypted by hardware, and resource management actions of the hypervisor, such as mapping of memory to the guest, are checked for consistency with the expected VM state. This prevents, for example, a compromised hypervisor from interfering with a guest's memory layout. Finally, like other trusted computing technologies, these designs include a hardware root of trust with support for remote attestation of the guest VMs, enabling a cloud user to verify that their VM has been correctly launched before trusting it with any secrets.

While the various architectures share many similarities, they differ in the trusted components that check hypervisor operations and enable remote attestation. In AMD's design, these are delegated to firmware on a separate platform security processor, whereas in the Intel and Arm designs these are performed by trusted and attested software running on the CPU itself. Regardless of where it runs, the relevant firmware/software must be trusted by both host and guest, and although it is simpler than a full hypervisor, that is hardly a guarantee of correctness. Notably, AMD's firmware (the only one of the three to have reached production) has already suffered serious vulnerabilities [13, 14, 24, 26].

2.3 Side channels in processor-based TEEs

Because confidential VMs inherit from SGX the key design feature of a privileged attacker who controls resources and shares processor time with the TEE, we expect them to remain vulnerable to many forms of side-channel attacks similar to those that devastated SGX [112, 114, 115].

Several attacks have demonstrated that the processor's address translation mechanism can be used to extract information such as cryptographic keys, text documents or JPEG images from SGX enclaves [101, 111, 128]. In these attacks, the adversary manipulates or simply monitors page tables to observe addresses accessed by the victim. While these attacks were demonstrated for SGX, it is clear that they carry over to TEEs like AMD SEV where the attacker controls nested page tables and handles nested page faults.

Other transmission channels include processor caches [22, 41, 82], branch prediction hardware [65] and interrupt latency [90, 113]. Some of these attacks generalize not only beyond SGX but also beyond TEEs. These channels also form the basis for tools that allow the adversary to single-step instruction-by-instruction through the enclave code [110] and to replay TEE instructions arbitrarily many times without having to rerun the TEE code [104]. Both techniques generalize beyond SGX, as they only require the adversary to control address translation and interrupts, respectively.

Speculative execution attacks have been used to leak information across all x86 isolation boundaries, including virtual machines and SGX [25, 62, 72, 98, 112, 114–116]. For example, Foreshadow [112] results in the disclosure of the entire enclave memory and the processor's SGX attestation key. To mitigate such attacks, confidential VMs rely on the same basic approach as SGX: microarchitectural tweaks and microcode patches to the "context switch" path between TEE and host code. More recent research demonstrates that side channels remain a problem on AMD SEV, including SQUIP attacks [39] via scheduler queues within the *same CPU core*; CipherLeaks [68] via *online encrypted memory analysis*; and attacks via *hypervisor*-observable nested page faults [44, 67]. By contrast, core slicing avoids the shared core resources, the online memory access from other security domains (i.e., slices), and the hypervisor.

2.4 VMs as used in public clouds

We next look at how VMs are deployed in clouds today, focusing on infrastructure-as-a-service platforms, which offer VMs backed by guaranteed resources (CPUs, memory, and in some cases accelerators and I/O bandwidth). We consider Amazon EC2 and Microsoft Azure, as they collectively represent 60% of the worldwide IaaS market [38]. We do not consider non-IaaS workloads such as serverless or micro instances for which core slicing may be a poor fit.

VMs are allocated at core granularity. Despite offering a plethora of different VM sizes [11], all current-generation VMs in Amazon EC2 occupy at least an entire core (i.e., two vCPUs on platforms that support hyperthreading), and Amazon states explicitly that host cores are "pinned" to specific guest vCPUs and are not shared across guests [7]. Microsoft Azure also offers a wide range of VM configurations [81]. Like Amazon, Azure does not oversubscribe customer vCPUs: Cortez et al. [28] note explicitly that their system will "only oversubscribe servers running first-party workloads."

For both providers, *burstable VMs* [10, 80, 119] represent the main special case as far as CPU allocation is concerned. These VM types are optimized for workloads that are mostly idle, with only occasional bursts of CPU activity. Like all cloud VMs, they have a fixed number of vCPUs, but consume on average only a fraction of their vCPU allocation in physical CPU runtime. Thus, of all the VM types offered across EC2 and Azure, burstable VMs are the only type that fundamentally requires the use of a hypervisor to perform time-slicing, in order to account for the VM's actual CPU utilization, and (presumably) to benefit from sharing CPUs across burstable VMs. All other VMs have a guaranteed allocation of physical CPUs, and for these the cloud provider derives no obvious benefit from hypervisor time-slicing.

Virtual I/O is becoming fully offloaded. Cloud vendors have deployed dedicated hardware "cards" that replace software I/O virtualization stacks, exposing virtual devices to guests directly via SR-IOV. For example, Amazon Nitro [5, 7] and Azure AccelNet [36] enable low-overhead networking. EC2 also supports direct access to NVMe storage [8], and Azure supports SR-IOV for InfiniBand and GPUs [57]. It thus seems reasonable to assume that, in the near future, the only I/O devices that are still emulated by host software will be those that are not performance sensitive, such as the virtual serial port or console device used for debugging.

Advanced VM features are not needed. Cloud providers rely on VMs to isolate tenants, but make little to no use of the advanced features enabled by full virtualization. Some features are incompatible with the IaaS model of dedicated resources. For example, a customer paying for a VM with 16 GiB of RAM has no incentive to enable memory ballooning [118] and return unused memory to the hypervisor. Other features, such as transparent shared page detection [118, 124], are disabled because of their significant security risks in a multitenant cloud [86]. Cloud providers may use live migration to update host software [129], but this has significant performance impact and hot patching is often preferred [9, 79]. We will discuss this further in §3.4.

Bare-metal clouds. Although the bulk of IaaS cloud workloads run in virtual machines, there is also a sizable and growing market for bare-metal cloud servers that offer dedicated machines at a premium price. The three primary reasons for a customer to choose a bare-metal instance over a VM are: (a) to avoid the CPU overhead ("virtualization tax") for memory-intensive workloads (described in §2.1), (b) a need for predictable performance without any possible contention from other co-located VMs (or "noisy neighbors"), or (c) security/compliance concerns arising from a shared hypervisor [92]. Customers that need to run their own hypervisor may also choose bare-metal servers to avoid the substantial performance overhead of nested virtualization [20, 70]. Core slicing seeks to offer similar features at flexible granularity and consequently lower cost.

2.5 Summary

We see that the bulk of IaaS VMs deployed in public clouds today run with a fixed allocation of memory and discrete cores, with I/O that is or soon will be fully offloaded, and have little to no use for features of virtual machines except for isolation. At the same time, CPU vendors are adding substantial complexity (not to mention, security risks and performance overhead) to their designs to support confidential VMs. We ask the question: *since the resources assigned to cloud VMs are effectively static slices of a machine, rather than relying on complex software to check the actions of a hypervisor that is not expected to do anything at VM runtime, why not enforce those partitions in hardware?*

3 Design

We describe the design of core slicing, starting with its overall architecture and threat model, before detailing our proposed hardware mechanisms, and how those mechanisms can be used by system software under the control of the host to securely partition hardware among untrusted guests.

Our design goals are as follows:

1. Partition shared hardware at natural boundaries, such

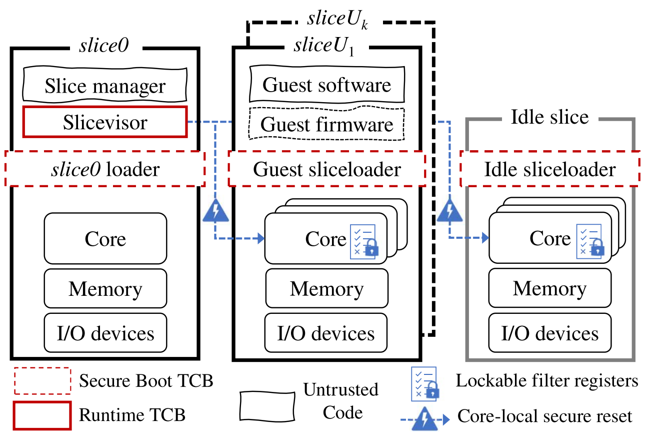


Figure 1: Core slicing system architecture.

as whole cores, while relying only on simple, easy-to-implement hardware mechanisms to ensure isolation.

2. Keep the trusted computing base small and simple, to permit a formally-verified implementation.

3. Support memory encryption and remote attestation features equivalent to confidential VMs.

3.1 Overview and terminology

Rather than VMs, we partition a machine into multiple guest *slices*, as shown in Figure 1 (denoted as *sliceU*). Each slice consists of a distinct set of named hardware resources: cores, memory ranges, and (virtual) I/O devices. Those resources are allocated exclusively to a slice for the duration of its execution. In the case of cores, this means that guest code runs in the highest privilege level (e.g., hypervisor mode), and controls every CPU cycle executed on that core until the slice is terminated. Like VMs, slices may start or stop at any time, and resources that were allocated together in one slice may later be partitioned among distinct slices after the first slice terminates. The key invariant is: *at any given time no two slices may share access to the same resource*. This avoids the need for any hypervisor-level mechanisms to share resources between guests, such as time-slicing VMs on a shared core or demand-paging overcommitted memory.

To allocate resources and manage the lifetime of guest slices, we rely on a distinguished control slice (termed *slice*0) running a *slice manager*. Somewhat like the host domain or root partition of a hypervisor, the slice manager is started at boot, and is responsible for creating and destroying user slices and determining their resource allocations. The slice manager runs on a core dedicated to that purpose, ideally a low-power management processor, potentially even on a separate chip as in Amazon's Nitro system [7]. We do *not* assume that the slice manager shares memory cache-coherently with user cores, nor that it executes the same instruction set. The slice manager software is further divided into a small, privileged portion, the *slicevisor*, that must be trusted by both cloud guests and the host, and a larger, unprivileged portion, that need not be trusted by guests. The unprivileged slice manager cannot interfere in the execution of a guest slice except for terminating it and resetting its cores. The slice manager maintains an *idle slice* to account for any unused resources. Cores in the idle slice do not execute, and merely wait to be assigned to a user slice.

To isolate resources, we rely on a new hardware mechanism, *lockable filter registers*, that restricts access to resources from a given core. Once configured and locked, these registers are read-only until the core is reset via another new mechanism, *core-local secure reset*. We require that it be restricted so that only the slicevisor can initiate a reset. A trusted loader, the *sliceloader*, is the first code to execute after a reset.

3.2 Security properties and threat model

Core slicing offers strong security, eliminating interference between all slices (including the control slice). Specifically:

1. The resources assigned to a slice are static from its creation until its termination.

2. A slice cannot access memory outside the slice, neither from cores nor via DMA.

3. A slice cannot interrupt cores outside the slice.

4. A slice cannot access I/O devices outside the slice.

5. Only *slice*0 may terminate another slice or reset its cores.

The threat model for core slicing is comparable to other forms of trusted computing including confidential VMs and enclaves [29, 35, 64]. The host (cloud provider) and guests (cloud users) are mutually distrusting, with one caveat: a guest relies on the host to provide agreed resources (thus, denial of service is out of scope), but can check at runtime that sufficient resources (e.g., as many hardware cores as expected) are available. The cloud provider trusts the management stack executing in the control slice, which determines both which specific resources (CPU cores, memory, etc.) a guest is permitted to use, and for how long it executes.

Guests must trust only hardware, the slicevisor, and the sliceloader. The slicevisor ensures that resource allocations are disjoint (for example, that none of the memory allocated to a guest slice is ever shared with another slice) and configures hardware protection mechanisms accordingly. The slicevisor is also responsible for attesting guest slices, and forms part of the attestation root of trust. The sliceloader ensures that lockable filter registers are properly configured and locked before transitioning control to guest code on each core of a slice.

Core slicing provides substantially stronger protection from side-channel attacks than confidential VMs. Because slicing partitions a machine at core granularity, the only side channels possible are those that can be observed from another core; notably, side-channel leaks to sibling hardware threads or to a malicious hypervisor are impossible.

Hardware security features including memory encryption and cache partitioning are orthogonal to our design; if present, they allow us to strengthen the defenses against physical and cross-core side-channel attacks respectively.

3.3 Hardware support for core slicing

Recall that our goal is to partition shared hardware at core boundaries while relying only on simple hardware mechanisms to ensure isolation. To keep this hardware simple, a key choice we make is to *expose the underlying physical resources directly to guest software*. Specifically, a guest slice may not have access to all cores or physical memory of a machine, but the identifiers it uses (i.e., the processor IDs and physical addresses) for the resources to which it does have access are always those of the underlying hardware. The only role of our hardware extensions is to restrict the set of accessible resources on a per-core (and thus, per-slice) basis.

This design choice avoids the need for any additional translation layers (as in virtualization) but it does place some requirements on guest software (i.e., OSes or hypervisors). Specifically, guests must be able to run with non-contiguous processor IDs, and cannot assume that physical memory starts at any particular address (such as zero). Luckily, modern OSes already meet this requirement: the initial OS boot image is a position-independent binary that uses a well-specified data structure (either ACPI tables [109] or a devicetree blob [71]) to locate all accessible resources, including memory ranges and additional processors. As long as this boot-time data structure accurately describes the resources accessible to a slice, a correct guest will make no attempt to access other hardware, and it is sufficient to treat any illegal accesses as fatal.

We therefore require a hardware mechanism that can restrict access to named physical resources (physical addresses for memory and I/O, and processor IDs for inter-processor interrupts). However, by design, guest software running in a slice should be able to use the highest privilege level on those cores, which raises a conundrum: how can we configure these restrictions without a slice being able to change them?

Our solution borrows from a pattern seen in hardware support for trusted execution: we introduce *lockable filter registers* that restrict the accessible resources by *all* software (including the most privileged) running on a given core. Once configured and locked, these registers are read-only until a subsequent *core-local secure reset* regains control of the core from the slice. Similar lockable registers (sometimes referred to as "latches") have been used to implement hardware security mechanisms such as secure boot and attestation for an entire system [127]. To our knowledge, we are the first to propose such a technique at the granularity of a single core.

At a hardware level, our proposed secure reset mechanism is just a subset of the existing system-wide reset functionality, exposed separately at core granularity: it stops execution, resets the entire core to a well-defined architectural state (resetting locked registers), and causes the core to begin execution at a fixed address. The unique constraints we place on this mechanism are that (a) only the slicevisor running in the control slice's privileged mode can initiate such a reset, and (b) the address of the jump target that receives control after reset remains inaccessible to any user slice. Here, we host the *sliceloader*, a small piece of trusted code similar to a secure bootloader that will reassign the core by programming and locking its filter registers before transferring control to untrusted user code, or taking it offline as part of the idle slice.

We next describe how hardware filters enable slicing of each distinct resource (memory, interrupts, cache, and I/O).

Memory To prevent a core from accessing any memory outside its slice, we rely on *lockable memory range registers* that restrict physical memory accesses by a core. Although we leave the precise semantics of these registers, such as the number of ranges and any alignment/size constraints, up to hardware designers, we assume that they can be configured in such a way as to restrict access to at least one contiguous range of RAM for a slice, as well as any virtual I/O devices (e.g., network and storage controllers) assigned to the slice, and any other memory-mapped registers (such as a local interrupt controller or timer) that are necessary. The minimum number of range registers is thus platform-specific, but we anticipate that around 10 ranges per core will generally suffice. More ranges will permit the slice manager greater flexibility in memory allocation, especially on multi-socket systems with non-uniform memory, but comes with a small but non-zero cost in additional hardware resources.

Range checks are trivially parallelizable and can be applied before installing a page translation in the TLB, thus having negligible runtime overhead. By contrast, virtualization relies on a nested page table which not only requires additional memory, it also increases page translation overhead and TLB pressure [3, 37, 88, 108]. However, because a slice is restricted to discrete physical ranges, its memory cannot be allocated in arbitrary pages, but instead must occupy a limited number of *contiguous* physical regions. We will evaluate the impact of this constraint on memory fragmentation in §6.4.

Interrupts We require hardware support to prevent a slice from sending inter-processor interrupts (IPIs) to cores outside its slice. Luckily, the address space of processor identifiers is substantially smaller than memory addresses, so we propose to use a *lockable IPI destination mask register* in preference to range checks. This register, which may in reality consist of a number of consecutive model-specific registers (e.g., four

64-bit registers for an 8-bit processor ID), permits a slice to run on any combination of cores. Of course, most workloads will benefit from adjacent cores (and caches).

Cache By design, any state internal to a core including the L1 cache is never shared across slices. However, shared L2 or L3 caches may raise performance interference and security concerns around cache-based side channels. To mitigate these, core slicing can make use of existing hardware support for cache partitioning [84], as long as the relevant configuration registers can also be locked or otherwise restricted.

I/O devices As described in §2.1–2.4, SR-IOV has been deployed by cloud providers to allow VMs to directly access networking, storage, GPUs, and more at no overhead. Just as the virtualization host OS assigns virtual functions to VMs, *slice*0 configures and assigns virtual functions to slices.

I/O devices interact with software in three ways: memory-mapped registers, direct memory access, and interrupts; all three require a suitable access control mechanism. Access to memory-mapped I/O is restricted by the same range checks as regular memory, and we do not discuss it further here. However, we need a way to prevent a slice from initiating DMA transfers to any memory outside its slice. Such restrictions are typically implemented by an IOMMU [12, 49], and typical IOMMU functionality suffices for core slicing, as long as the IOMMU remains under the control of the slicevisor.

One simple approach is to configure the IOMMU to map accessible slice memory 1:1 for each virtual function belonging to a slice. Its main downside is security: all memory assigned to a slice is available for DMA. This does *not* compromise slice isolation, but, like a bare-metal system without an IOMMU, it may allow a buggy device driver to access the wrong guest memory. Rather than reprogramming the IOMMU at runtime to restrict DMA, recent work found that it is more efficient to simply allocate all I/O buffers from a dedicated pool of physical memory that remains mapped in the IOMMU [74]; this is naturally supported by core slicing, and avoids the need for runtime interaction with slicevisor to reprogram a slice's IOMMU translations.

Besides DMA, an I/O device also sends interrupts. The interrupts of virtual functions are mapped and routed to host cores by the IOMMU, and the same mechanisms apply directly to core slicing. Since slice cores are statically assigned and there is no host hypervisor, direct interrupt mapping is substantially simpler than VMs [40].

One unique challenge of SR-IOV is that virtual functions do not implement normal PCI configuration space registers. Rather, a hypervisor typically emulates configuration space accesses for an assigned virtual function. For slices, we could rely either on an enlightened guest to avoid the need for such virtualization (as in our x86 prototype, see §5), or else on a custom PCI "card" [6, 36] to emulate a standard memory-mapped configuration space within its own device window.

3.4 Slice management

We now turn our attention to the *slice*0 software stack responsible for resource allocation, slice lifetime, and a handful of runtime services. This may run on a dedicated host core, a low-power management core, or a separate SoC. We require only that it (a) has hardware privilege separation, (b) is able to trigger secure resets of guest cores, and (c) shares *some* memory, not necessarily cache-coherently, with those cores.

Recall that only the privileged portion of the slice manager, the slicevisor, is trusted by guest slices. The unprivileged slice manager has no access to guest slice memory or cores. The slicevisor implements all security-sensitive aspects of the process of creating, starting, stopping, and deleting a slice. Its primary role is to ensure that no resource is ever accessible to two slices at the same time; this includes checking that memory ranges assigned to slices are disjoint, and ensuring that all cores of an expired slice have been stopped via a secure reset prior to reassigning any resources of that slice.

To keep the slicevisor as simple as possible (and permit its eventual implementation in formally-verified code), it does not directly allocate resources, but merely checks the correctness of resource assignments provided by the unprivileged slice manager. This permits the unprivileged slice manager to implement flexible policies to choose the memory ranges and cores assigned to slices, without the need to trust them.

The only other code that must be trusted by guests is the *sliceloader* which is the first thing to run after a core is reset by the slicevisor. It is responsible for configuring and locking the per-core access filters, and then coordinating with other loaders of the slice to securely boot the guest OS. To do this, it relies on a *slice table* of configuration information maintained by the slicevisor (and inaccessible to untrusted software).

Starting a guest To create a slice, the unprivileged slice manager assigns available cores and memory, and determines the configuration of virtual I/O devices. Next, it invokes the slicevisor to create the slice, which rejects the request if the new slice includes any resource shared with another slice (including *slice*0), other than the idle slice. Otherwise, the slicevisor updates the slice table, programs devices to configure virtual functions, and updates IOMMU translations as described earlier. At this stage, the assigned cores remain idle.

To start the new slice, the slicevisor resets the relevant cores, which causes them to execute the sliceloader and follow a secure boot flow. After reading the relevant configuration from the slice table (recall that at this point the sliceloader is trusted and has unrestricted access to system memory), the loader determines whether the current core belongs to the idle slice or a new guest slice. It then programs and locks the core's access filters for memory ranges and IPIs, before synchronizing with the loaders (if any) for other cores in the slice. The rest of the boot process has no access to memory or cores of other slices. It zero-fills slice memory ranges, before copying the guest's boot image and transferring control.

Terminating a guest Unless they are unresponsive, guest OSes will generally execute a controlled shutdown initiated via an out-of-band signal. To finally terminate a slice, the slice manager invokes the slicevisor which clears the relevant configuration in the slice table. Then, to stop the guest cores (which no longer have access to the slice table), the slicevisor reassigns them to the idle slice and resets them.

Auxiliary services At runtime, the slice manager exposes a simple shared memory device (much like a virtual I/O device) to guest slices, permitting an enlightened guest OS driver to initiate a shutdown or reset of the slice, or access slow-path emulated I/O devices (such as a virtual serial port and console) for which no offload device is warranted.

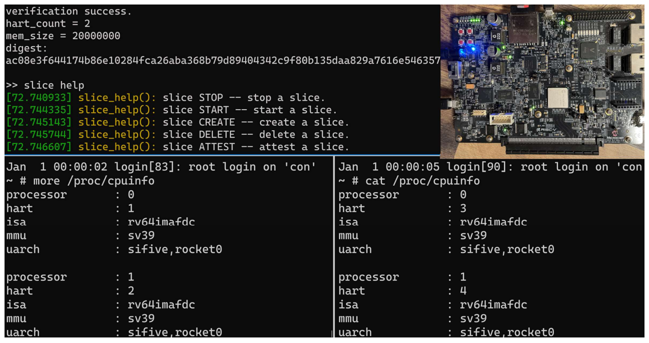
Unsupported features The functionality of a guest slice is similar to bare-metal clouds, and lacks advanced VM features such as live migration. This does not preclude a guest from implementing its own mechanisms (e.g., by running its own hypervisor, or doing so at process level [34]), but it does prevent a cloud provider from transparently migrating guests to implement software upgrades [129]. Since core slicing eschews the use of a hypervisor and runs the entire host stack on a dedicated management core, we do not anticipate that updates will require live migration. In particular, we expect that it will be possible to update the slice manager and slicevisor without any guest interruptions.

3.5 Attestation and memory encryption

We assume that hardware implements a root of trust for secure boot, permitting the initial bootloader and slicevisor to be cryptographically measured and attested. In turn, the slicevisor attests individual slices; this includes a measurement of slice configuration, the sliceloader code, and the guest image. Once booted, a guest can prove to a remote verifier that its slice is configured as expected and that its isolation is enforced by trusted slicevisor and sliceloader implementations.

As described earlier, slice memory is strongly isolated to defend against software attacks. To defend against physical attacks on memory, such as cold-boot and memory-bus attacks [43, 89], core slicing can leverage memory encryption hardware. The details of memory encryption are orthogonal to our design, and we expect to leverage existing platform mechanisms. In particular, because only trusted components (sliceloader, guest code, and I/O devices within DMA regions) can access slice memory, it is irrelevant to slice guests whether memory is encrypted by a random system-wide key (as in Intel SGX and Arm CCA), one of a set of random keys (as in Intel TDX), or a unique key for each guest (as in AMD SEV).

If per-slice memory encryption is nevertheless desired, we assume that the hardware will provide a suitable interface for


```
verification success.
hart_count = 2
mem_size = 20000000
digest:
ac08e3f644174b86e10284fca26aba368b79d89404342c9f80b135daa829a7616e546357

>> slice help
[72.740933] slice_help(): slice STOP -- stop a slice.
[72.744335] slice_help(): slice START -- start a slice.
[72.745143] slice_help(): slice CREATE -- create a slice.
[72.745744] slice_help(): slice DELETE -- delete a slice.
[72.746607] slice_help(): slice ATTEST -- attest a slice.
```

```
Jan  1 00:00:02 login[83]: root login on 'con'
~ # more /proc/cpuinfo
processor       : 0
hart            : 1
isa             : rv64imafdc
mmu             : sv39
uarch           : sifive,rocket0

processor       : 1
hart            : 2
isa             : rv64imafdc
mmu             : sv39
uarch           : sifive,rocket0
```

```
Jan  1 00:00:05 login[90]: root login on 'con
~ # cat /proc/cpuinfo
processor       : 0
hart            : 3
isa             : rv64imafdc
mmu             : sv39
uarch           : sifive,rocket0

processor       : 1
hart            : 4
isa             : rv64imafdc
mmu             : sv39
uarch           : sifive,rocket0
```

Figure 2: RISC-V prototype with the slice manager (upper left) and two Linux guest slices (lower terminals).

the slicevisor to configure a unique slice memory encryption key. In that case, when initializing a new slice, the sliceloader starts out in an unencrypted context before enabling encryption to load and boot the guest. Similarly to other confidential computing architectures [15, 54], once the guest boots, attested I/O devices may access the encrypted memory using the guest's encryption context.

4 RISC-V Prototype

We demonstrate the feasibility of our design with a RISC-V prototype. We chose RISC-V because an existing feature closely approximates the lockable filter registers required by core slicing. Our prototype runs in two environments: on a modified version of QEMU, and on an unmodified Microchip PolarFire Icicle board with a SiFive FU540-C000 SoC shown in Figure 2, the latter with some limitations due to partial support for our requirements. As the guest OS we run Linux.

Common RISC-V CPUs implement three privilege levels: machine (M) mode for firmware, supervisor (S) mode for an OS kernel, and user (U) mode for applications. Our SiFive SoC includes four general-purpose application cores and one less-powerful monitor core that implements a limited instruction set with only M and U modes. The monitor core is ideal for *slice*0, with the slicevisor running in privileged M-mode, and the rest of the slice manager in user mode. The remaining four application cores are available for guest slices in arbitrary combinations (i.e., up to four single-core guests).

Memory In addition to typical address translation mechanisms, RISC-V supports physical memory protection registers [93, §3.6] which can be programmed to restrict access to physical address ranges on a per-core basis. These registers are grouped into 8 or 16 PMP entries. Each PMP entry consists of a configuration register and an address register that specify the access permission (read/write/execute) to a particular region. Once programmed, PMP checks apply to all memory accesses from user and supervisor modes, and

Table 1: PMP permissions by physical region.

Physical address range	*slice*0 M-mode	*slice*0 U-mode	*slice*U_1
sliceU_1 RAM	—	—	RWX
*slice*0 trusted RAM	RWX	—	—
*slice*0 untrusted RAM	RWX	RWX	—
sliceU_1 bus	RW	—	RW
Other *sliceU* bus	RW	—	—
Cache controller	RW	—	—
Reset unit	RW	—	—
Interrupts to *slice*U_1	—	—	RW
Interrupts to *slice*0	RW	—	RW
Physical I/O devices*	RW	—	—
sliceU_1 virtual devices*	—	—	RW

* Not implemented due to hardware limitations.

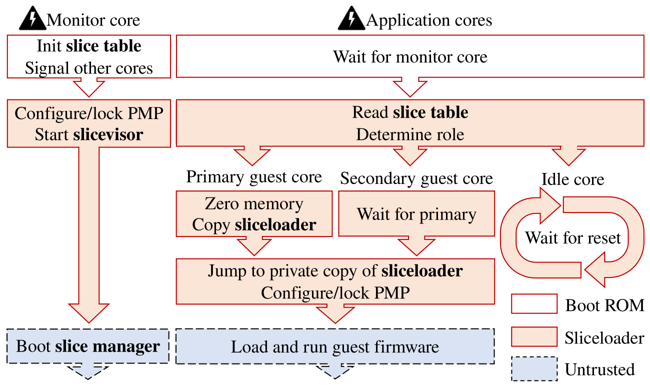


Figure 3: Boot flow for each core after reset.

optionally from machine mode. Finally, as required by core slicing, PMP registers include a *lock* bit that, once set, prevents any subsequent modifications until a reset. Our prototype sliceloader configures and locks the PMP entries for each core in the slice before booting the OS. Thus, code in a slice cannot access physical addresses outside its slice, even from the most privileged machine mode.

Besides memory, other locked PMPs grant access to a slice's hardware resources (described below), and enforce privilege separation between the slice manager and slicevisor on the monitor core. Table 1 summarizes their configuration.

Interrupts Recall that our design calls for lockable mask registers restricting the destinations for inter-processor interrupts. We found that although RISC-V lacks such a feature, the careful use of PMPs permits an equivalent mechanism. To send an IPI on the SiFive SoC, the source core writes to a memory-mapped register of the destination's "core-local interruptor" [102]. Because the register's address is unique for every destination, we can use the source core's PMPs to restrict the addressable (and thus interruptible) destination cores. Access to core-local timers is restricted similarly.

I/O devices Our board includes several I/O controllers, and our prototype grants access to slices using PMPs and routes interrupts accordingly. We were unable to prototype SR-IOV support due to a lack of suitable hardware, such as an IOMMU. The RISC-V community has proposed PMP-like mechanisms to restrict DMA [103], but these are not yet available.

Cache partitioning The SiFive SoC includes a 2 MiB shared L2 cache that supports *way masking*, allowing each cache master to be restricted to a subset of the 16 total ways. Since each core's L1 I- and D-cache act as separate masters for the L2 cache, we can flexibly partition it by enabling distinct way sets for each slice. Our implementation scales the size of a slice's cache partition with the number of cores in that slice (i.e., four ways per core); thus, larger slices enjoy a larger share of the cache. Untrusted code cannot change the cache configuration because the cache control registers are enabled only in the monitor core's M-mode PMP registers.

Core reset Recall that our design relies on a secure core-local reset to re-establish trusted control of a core from a user slice. Unfortunately, our board does not expose per-core reset signals, requiring a full system reset that reboots the entire board to clear any PMP lock bits. We have taken a number of approaches, with different trade-offs, to avoid this limitation.

First, we modified QEMU to implement a new device that exposes per-core reset registers; our prototype slicevisor uses these to reassign cores.

Second, we modified the slicevisor to create slices at boot time using a pre-determined configuration. On our board, such slices cannot be destroyed without a whole-system reset because their PMPs are locked. However, despite the loss of flexibility, this provides a strong security guarantee.

Finally, to test our ability to destroy and create slices on hardware, we implemented an insecure slice manager with an alternative software-based reset mechanism. In this version, PMP lock bits are not set. This provides no meaningful security (a malicious slice could reconfigure PMPs), but permits a cooperative slice to emulate a "secure" reset upon receipt of an IPI by clearing PMPs and jumping to the sliceloader.

Sliceloader and guest firmware As shown in Figure 3, the actions of the sliceloader vary depending on the slice to which the running core is assigned. When booting a guest, the sliceloader copies itself to private slice memory before setting PMPs, because doing so makes the main copy inaccessible.

When Linux boots, it infers the system configuration from a devicetree blob [71] provided by the firmware. This describes the available memory, cores, and devices on a given platform. To allow Linux to boot in a slice, we enlightened the OpenSBI bootloader [94] to run as untrusted code inside a newly-created slice. It constructs a devicetree describing the slice configuration before booting Linux. Given an appropriate devicetree, Linux itself required no modifications to run with arbitrary physical memory ranges or core IDs.

Slice communication To enable communication between the slice manager and guests, we implemented a *slice bus* message transport, using a region of shared memory between each guest slice and *slice*0, with IPIs for signaling.

Attestation To prove that a guest slice runs only the image expected on a trusted platform, we implemented *measured boot* in the sliceloader and *attestation* in the slicevisor. When creating a slice, the sliceloader measures guest code and stores its hash in *slice*0 trusted memory. To attest, slicevisor generates an attestation report including the guest measurement and user-provided data, signed by a slicevisor-held key derived from a hardware root of trust. A remote party can verify the attestation using the device public key and the TCB report.

Limitations Due to hardware limitations, our prototype lacks support for memory encryption. We do not yet implement virtual serial ports, but assign a UART to each slice.

5 x86 Prototype

We also implemented an x86 prototype. This lacks security isolation, but permits us to experiment with SR-IOV devices and compare performance to the state-of-the-art in hardware virtualization. We discuss ideas for actual hardware support on x86 platforms later, in §7.

Since there is no security, we simplified our management stack by building on top of Linux. Specifically, we run Linux on the bootstrap core, using kernel parameters that restrict Linux to a single core and a minimal amount of physical memory. Non-boot cores remain idle, in the *wait-for-startup-IPI* state. A privileged process is later responsible for configuring and booting slices with a user-specified set of cores, range of physical memory, and set of PCI devices.

Booting a slice To boot a slice, we load the guest kernel into the chosen physical memory range (accessed via `/dev/mem`), construct ACPI and E820 tables describing resources available to the slice, and then send a startup IPI to the slice's first core. This runs a tiny (48-instruction) real-mode bootloader that constructs a page table, switches to 64-bit mode, and enters the slice kernel, which then boots as usual, sending further startup IPIs to other cores.

Guest enlightenments Because all the host hardware remains accessible, the Linux guest needed a few modifications. We disabled the CONFIG_DMI and CONFIG_X86_MPPARSE build options to prevent the slice kernel discovering these legacy firmware tables (and hardware they

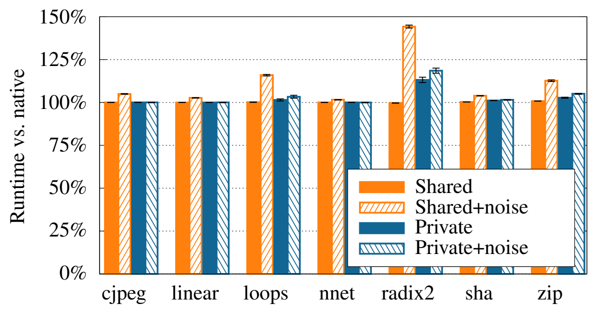


Figure 4: RISC-V CoreMark results (lower is better).

describe) in the system BIOS. We fixed a bug that unconditionally enabled interrupts from the legacy PIC despite ACPI flagging it as absent. Finally, we added 283 lines of code to (a) use an SR-IOV virtual function without a virtual configuration space, and (b) enable only a subset of PCI devices. These enlightenments would be irrelevant to a hardware implementation. In particular, host firmware and devices would be inaccessible to slices, and (as described in §3.3) an I/O device could emulate standard PCI configuration space.

6 Evaluation

This evaluation seeks to answer the following questions:

- What is the performance overhead of core slicing? (§6.1)
- How does the design of core slicing translate into concrete security benefits for guest slices? (§6.2)
- What is core slicing's hardware complexity? (§6.3)
- Does the need for contiguous physical memory lead to slice allocation failures due to fragmentation? (§6.4)

6.1 Performance

Our experiments run on the RISC-V board described in §4 with a 16-way 2 MiB L2 cache and 1 GiB of DRAM, and on an HP Z8 workstation with two Intel Xeon 4214 12-core CPUs (HyperThreading disabled), 64 GiB of RAM, a Mellanox ConnectX-4 25GbE NIC, and a Samsung PM1735 NVMe SSD. Another HP Z8 with two Xeon 4108 CPUs and the same NIC serves as a client.

RISC-V We run CoreMark PRO [33] and focus on two questions: (a) does a slice achieve bare-metal performance, and (b) what is the impact of a "noisy neighbor" slice?

For the native baseline, we enable two application cores and 512 MiB memory. This allows a fair comparison with the slice measurements, in which we launch two guests, *slice*U_1 and *slice*U_2, each with two cores and 512 MiB memory. As shown by the *shared* results, performance in a slice exactly matches

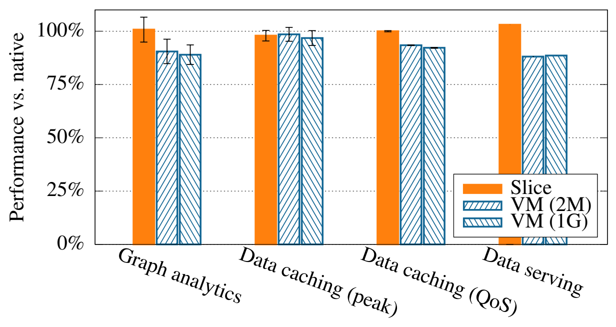


Figure 5: x86 CloudSuite results (higher is better).

bare-metal execution. To determine the impact of cache contention, we run a workload in *slice*U_2 as a noisy neighbor (denoted by 'noise') that repeatedly accesses a 2 MiB array (the same size as the L2 cache). Unsurprisingly, the average runtimes (denoted by *shared+noise*) increase depending on the given workload's cache intensity. Specifically, *radix2* suffers the most (almost 50% overhead) since it operates on a 512 KiB array and suffers frequent cache misses. As demonstrated by *private+noise*, when the cache is partitioned, a noisy neighbor has a substantially smaller impact, with all workloads achieving stable performance, and outperforming the *shared+noise* configuration. The runtimes in *private+noise* increase slightly ($< 5\%$) with noise compared to *private*, due to contention for the memory controller.

x86 We compare slice performance to equivalent VMs using CloudSuite 3.0 [87]. Both slice and VM guests have 8 cores and 16 GiB of RAM (allocated on the same NUMA node), with NIC and NVMe virtual functions for I/O. We run VMs under KVM using full hardware acceleration including virtual APICs, and confirm via performance counters that the only significant source of VM exits is to service and emulate timer interrupts. To minimize memory management overhead we use pre-allocated and locked huge-pages (2 MiB and 1 GiB).

The results in Figure 5 are scaled to a native baseline with the same cores/memory. *Graph analytics*, which uses Apache Spark and GraphX to run PageRank on a large Twitter dataset, runs 10% slower in a VM. *Data caching* models a Twitter cache server with Memcached. Despite sustaining a similar peak throughput, the VMs have higher jitter and thus perform up to 8% worse while meeting the published QoS target of 10 ms p95 latency. This appears to be due to the extra TLB pressure of nested paging: DTLB misses are more than 3× native for 1 GiB pages, and 5× for 2 MiB pages. Finally, *Data serving* runs Apache Cassandra with 10M records of YCSB workload A and incurs a 12% throughput penalty in a VM.

Summary Core slicing achieves bare-metal performance without the overhead of virtualization, which remains significant for memory-intensive workloads, even with huge pages.

Table 2: Size of software TCB.

	Source lines[a]	Executable code[b]
slicevisor	3,609	18 KiB
sliceloader	3,607	11 KiB
Total	4,826[c]	29 KiB

[a] Non-header lines, counted by `cloc` [30]. [b] Size of text section, from binutils `size`. [c] A library common to both is counted once.

Furthermore, hardware cache partitioning (as on RISC-V) not only lessens the impact of a noisy neighbor, it can also eliminate both cache contention and cache-based side channels.

6.2 Security

Size of trusted computing base Although no guarantee of security, a small TCB helps make formal verification tractable. Table 2 reports the executable source and binary sizes of our prototype. These are comparable to the firmware for Arm CCA (4.3 kLOC [69]), although we note that core slicing's TCB eschews runtime interaction with a running guest, and thus its attack surface is drastically simpler.

Side channels By partitioning a machine at core granularity and without a trusted hypervisor, core slicing avoids either the host or another guest running concurrently on a core. This is inherently more secure than either reducing the hypervisor's size [99, 100] or de-privileging it [13, 17, 53]. We also gain a systematic defense from a wide variety of CPU side-channel attacks. Following our threat model (§3.2), we consider two classes of attacker: guest attackers who may run arbitrary code in a guest slice, and the host attacker who controls untrusted code in *slice*0. We describe the extent to which these may compromise a guest's confidentiality.

Cache side-channel attacks are defeated by cache partitioning and memory isolation. Because slices never share memory, a guest attacker cannot steal secrets by analyzing whether a co-located tenant accesses shared memory addresses. Because caches are partitioned, the attacker cannot observe how many cache lines are used by the guest per cache set. *slice*0 is similarly restricted. Thus, given suitable hardware support, core slicing eliminates cache-based side channels. It also defeats all side-channel attacks where the attacker executes on the victim core, including sibling threads [4, 98].

Transient execution attacks can leak secrets through the side effects of speculative memory accesses [62, 72], and can break isolation between hypervisors and VMs. Core slicing relies on lockable filter registers to restrict memory access. Because filters only perform a range comparison (with no memory-bound table walk), they derive no significant benefits from speculation; e.g., current RISC-V CPUs cache PMP range checks in the TLB along with address translations [83].

Cross-core transient execution leaks were recently observed on Intel CPUs [91]; although these remain a threat, we expect them to be drastically simpler for vendors to identify and fix in future CPUs, since few instructions access uncore state. Of note, the demonstrated attack relies on delaying a victim enclave's execution with page faults and exceptions, which is impossible across a slice boundary.

Page-fault- and **page-table-based attacks** are highly exploitable on TEEs where an untrusted hypervisor manages guest memory using a nested page table. In these attacks, a host learns the guest's secret-dependent memory access pattern via page faults, page table access/dirty bits, or even cache contention with hardware table walks. Core slicing prevents this by allocating physical memory directly to guests. In addition, memory encryption can suffer from **ciphertext-only attacks** (e.g., dictionary attacks) with weak encryption algorithms. In our design, lockable filter registers prevent access to guest memory, including ciphertext, even by the host.

Side channels in resources shared by multiple cores remain, including power [61], row-hammer [59], cold-boot [43], and memory bus [89] attacks. Those can be mitigated with additional orthogonal hardware support.

6.3 Hardware complexity

To estimate the hardware cost of supporting core slicing, we extended the default *tiny* configuration of the RISC-V Rocket Chip implementation [18]. To measure the overhead of adding lockable filter registers, we doubled the number of PMPs from 16 to 32 (as might be necessary when existing PMPs are required for other uses), resulting in only a 3% increase in total FPGA resources. We also added per-core resets and a reset device for the monitor core, for 1.7% extra resources. Since these results are for an embedded core, we expect the fraction of resources required on a server-grade CPU to be much smaller.

6.4 Impact of physical contiguity

Unlike VMs, slices require *contiguous* physical memory, and the memory assigned to a slice cannot be changed without terminating and restarting it. Thus, it is possible that the available memory on a node becomes fragmented over time, leading to a situation where sufficient free memory exists to support a new slice, but cannot be used as it is not contiguous. Whether this is a problem in practice depends on both the pattern of memory allocations (i.e., the order of slices created and destroyed) and the policy implemented by the memory allocator (i.e., which region of memory to allocate for any given request). In this section, we report the analysis of VM start/stop events on a public cloud workload, modeling the effects of memory allocation policy and hardware capabilities.

The trace we use is similar to the VM allocation trace of Hadary et al. [42]. It includes all VM start and stop events (in excess of 750k events) for an Azure cluster, and was gathered over two months in mid-2021. Each VM has a type [81] that defines its resource allocation, including its memory size (other resources are irrelevant to our analysis).

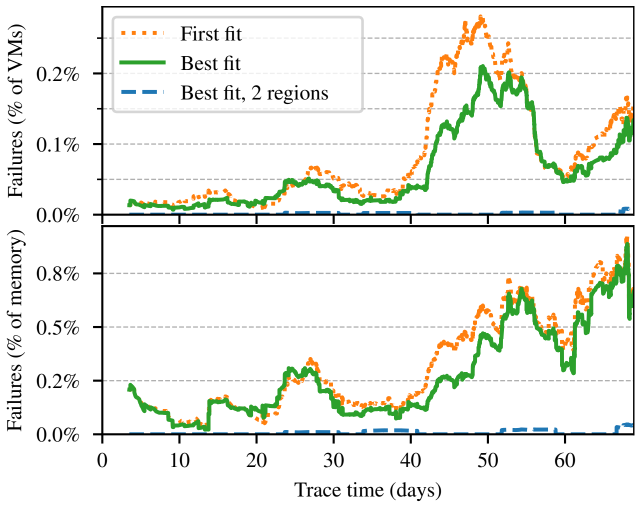


Figure 6: Rate of slice allocation failures due to memory fragmentation over a 7-day moving window.

In this analysis we ask the question: *how often does fragmentation prevent the allocation of a slice on a node when a VM would have succeeded?* Thus, although it may be beneficial for the cloud scheduler to use node-level memory contiguity in its placement decisions (allocating slices on nodes to reduce fragmentation), we leave the mapping of VMs to nodes unchanged and model slices as a drop-in replacement. Since our focus is memory allocation, we model every VM as a slice with physically contiguous memory. In reality, we expect that some types (such as burstable VMs) would continue to run as VMs on the cloud provider's hypervisor either within a slice of a larger machine, or using dedicated machines.

The results of our analysis are shown in Figure 6. We experimented with a variety of memory allocators, and found unsurprisingly that a best-fit policy (which places each new slice in the smallest free region of sufficient size) minimized fragmentation and thus allocation failures. Other policies, including a traditional buddy allocator, performed uniformly worse and are not shown on the figure. Calculating the best allocation will take slightly longer, but given the low rate of slice instantiations relative to traditional memory allocation workloads, this overhead is not expected to be significant.

As shown in the figure, the allocation failure rate for fully contiguous memory is less than 0.3% of VMs in the trace, representing less than 1% of the total requested memory (since larger VMs are more likely to fail allocation). We also modeled the effect of permitting multiple contiguous regions per slice. With hardware support for two memory regions per slice, the memory allocator is able to split large slices across two distinct allocations, and the failure rate drops further to less than 0.1% (only 6 failures across the entire trace). With three regions per slice, there are no failures. Overall, we conclude that restricting slices to contiguous memory does not pose a significant constraint for cloud operators. A different analysis by Teabe et al. [108] reached the same conclusion.

7 Discussion: core slicing beyond RISC-V

For a prototype implementation of core slicing, RISC-V had several advantages; most significantly, the use of physical memory protection registers allowed us to implement the bulk of our design on unmodified hardware. However, our design does not depend on RISC-V, and we ultimately hope to see it adopted by the x86 and Arm architectures that dominate today's cloud. This section discusses some of the challenges in doing so, and offers guidance to hardware designers.

The obvious first step in adapting an existing architecture to support core slicing would be to implement our hardware requirements: lockable filter registers to restrict a core's ability to access memory and send inter-processor interrupts, and a secure core-local reset to regain control of it. Of course, the details matter, and thanks to the long evolution of these architectures, there are many interactions with existing architectural features that must be considered.

Recall that our design goal with core slicing is to give guest software unfettered bare-metal access to a single core (or set of cores). As a first rule of thumb, we propose that *resource restrictions imposed via filter registers should take priority over other core-level architectural features*. This implies that existing features granting privileged access to memory, such as hardware shadow stacks [52] or secure-world memory regions [16] must be constrained by lockable address filters.

Second, *any hardware resources that are shared by more than one core must be restricted*. This includes peripherals, memory and cache controller configurations, and power management registers, among others. In RISC-V systems, such registers are memory mapped and thus restricted by PMPs, but on x86 they are configured via model-specific registers (MSRs) that occupy a distinct address space accessible to privileged software on each core (Arm system registers are similar). The access restriction could be implemented via further filter registers, or as a simpler alternative, access to these resources could be limited to the management core running the sliceloader. For the specific case of x86 MSRs, we expect that the MSR bitmaps found in the VM control block will serve as a useful starting point in determining the appropriate policy. Finally, the x86 legacy I/O address space must be filtered or (for legacy-free guests) blocked outright.

Finally, *the platform must not depend on firmware running on guest cores*. Thus, the system design should avoid the need for platform firmware in x86 system management mode or Arm EL3 on general-purpose cores. The motivation for this requirement is the same as that of core slicing: to avoid relying on intra-core privilege separation due to its demonstrated weakness. In our view, firmware tasks are better delegated to a dedicated management core (along with the slicevisor).

8 Related work

Direct hardware assignment We discussed secure hypervisors and confidential VMs in §2.2.

NoHype's [58, 107] central security goal is to protect a *trusted* cloud provider and its legitimate customers from rogue VMs that try to exploit vulnerabilities in the hypervisor or the associated virtualization stack. NoHype achieves this goal by removing all run-time interfaces that traditional hypervisors expose to traditional VMs. The security goals of core slicing reach significantly further and address threats that have emerged during the decade since NoHype was designed. In addition to protecting the cloud infrastructure from rogue guest VMs, core slicing protects guests from the *untrusted* cloud provider. This task is complicated by an ever-growing array of microarchitectural attacks that can leak information out of VMs. Therefore, core slicing does not allow any cloud provider code to run on a guest's processor cores. In contrast, NoHype requires a highly privileged "temporary hypervisor" on those cores.

Core slicing relies on simple lockable filter registers to confine guests which enjoy bare-metal control over cores, and may run their own hypervisor. NoHype relies on conventional processor privileges; guests thus lack access to virtualization extensions, must be modified to avoid VM exits (notably, they must not execute CPUID, including in user mode), and must ignore spurious interrupts from other guests (both a side channel and a denial-of-service attack).

TrustOSV [120] and Quest-V [123] minimize runtime guest-hypervisor interactions by statically assigning cores and memory. The core of TrustOSV is a *microhypervisor* that uses nested paging to constrain guest memory accesses. Quest-V replaces a global hypervisor by trusted per-core monitors that also run in host/root mode and use nested paging. TrustOSV reduces trust in the cloud provider by attesting the microhypervisor and exposing a limited management interface. In contrast to core slicing's use of I/O offload, TrustOSV exposes a virtual NIC which is also the basis of its storage.

Space partitioning The use of core-granularity spatial partitioning [47, 73] for resource and security isolation has been explored in the context of prior many-core systems including the Tilera TILE64 [122], Intel single-chip cloud computer [75] and M3 [19], and the core idea dates back at least as far as IBM's logical partitioning feature from the 1980s [21]. Core slicing builds on the same mechanism, but is unique in its adoption of the confidential computing threat model, with a clear separation of host and guest trusted computing base. This leads us to the use of a unique mechanism combining per-core secure reset with lockable filter registers to enable attested boot while minimizing the guest TCB. Past designs, including those cited above, permit a guest's accessible resources to be reconfigured at runtime by a privileged management core, requiring substantially more trust by the guest in the host's resource manager.

RISC-V security Several systems use PMP hardware for different goals. Keystone [64] is a framework for trusted execution environments similar to Intel SGX [46]. The OpenSBI bootloader can partition a machine into static PMP-isolated *domains* at boot time [95]. MultiZone [45] isolates software components (e.g., core RTOS and communication stack).

9 Conclusion

VMs are the basis of cloud isolation, but relying on them for confidential computing carries a serious risk from side channels. Core slicing offers an attractive middle ground between bare-metal servers and confidential VMs. By partitioning hardware at natural boundaries (discrete cores and contiguous physical memory ranges), it enables VM-like functionality and bare-metal performance with strong isolation.

Acknowledgments

We thank Luke Marshall for help preparing the VM traces in §6.4. The detailed reviews we received from OSDI'22, ASPLOS'23 and OSDI'23 along with feedback from our shepherd Ed Bugnion helped us greatly improve the paper.

References

[1] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–13, 2006. doi: 10.1145/1168857.1168860.

[2] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa. Firecracker: Lightweight virtualization for serverless applications. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation*, pages 419–434, Feb. 2020. ISBN 978-1-939133-13-7. https://www.usenix.org/conference/nsdi20/presentation/agache.

[3] H. Alam, T. Zhang, M. Erez, and Y. Etsion. Do-It-Yourself virtual memory translation. In *Proceedings of the 44th IEEE International Symposium on Computer Architecture*, pages 457–468, 2017. doi: 10.1145/3079856.3080209.

[4] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. García, and N. Tuveri. Port contention for fun and profit. In *Proceedings of the 40th IEEE Symposium on Security and Privacy*, pages 870–887, 2019. doi: 10.1109/SP.2019.00066.

[5] *Enhanced networking on Linux*. Amazon Web Services, Dec. 2022. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/enhanced-networking.html.

[6] *AWS Nitro System*. Amazon Web Services, Dec. 2022. https://aws.amazon.com/ec2/nitro.

[7] *The Security Design of the AWS Nitro System*. Amazon Web Services, Nov. 2022. https://docs.aws.amazon.com/whitepapers/latest/security-design-of-aws-nitro-system/security-design-of-aws-nitro-system.html.

[8] *Amazon EBS and NVMe on Linux instances*. Amazon Web Services, Dec. 2022. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/nvme-ebs-volumes.html.

[9] *Amazon EC2 Maintenance Help Page*. Amazon Web Services, 2022. https://aws.amazon.com/maintenance-help.

[10] *Amazon EC2: Burstable performance instances*. Amazon Web Services, Dec. 2022. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/burstable-performance-instances.html.

[11] *Amazon EC2: Instance types*. Amazon Web Services, Dec. 2022. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/instance-types.html.

[12] *AMD I/O Virtualization Technology (IOMMU) Specification*. AMD, Dec. 2016. Publication #48882 rev. 3.00 https://developer.amd.com/wordpress/media/2013/12/48882_IOMMU.pdf.

[13] *AMD SEV-SNP: Strengthening VM isolation with integrity protection and more*. AMD, Jan. 2020. https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf.

[14] AMD. AMD server vulnerabilities, Nov. 2021. Security Bulletin ID AMD-SB-1021 https://www.amd.com/en/corporate/product-security/bulletin/amd-sb-1021.

[15] *AMD SEV-TIO: Trusted I/O for Secure Encrypted Virtualization*. AMD, Mar. 2023. https://www.amd.com/content/dam/amd/en/documents/developer/sev-tio-whitepaper.pdf.

[16] *Building a Secure System using TrustZone Technology*. ARM Limited, Apr. 2009. Ref. PRD29-GENC-009492C.

[17] *Arm Realm Management Extension (RME) System Architecture*. Arm Limited, Nov. 2021. Document DEN0129 ver. A.b https://developer.arm.com/documentation/den0129/ab.

[18] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman. The Rocket Chip generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr. 2016. https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html.

[19] N. Asmussen, M. Völp, B. Nöthen, H. Härtig, and G. Fettweis. M3: A hardware/operating-system co-design to tame heterogeneous manycores. In *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 189–203, 2016. doi: 10.1145/2872362.2872371.

[20] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour. The Turtles Project: Design and implementation of nested virtualization. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, Oct. 2010. URL https://www.usenix.org/conference/osdi10/turtles-project-design-and-implementation-nested-virtualization.

[21] T. L. Borden, J. P. Hennessy, and J. W. Rymarczyk. Multiple operating systems on one processor complex. *IBM Systems Journal*, 28(1):104–123, Mar. 1989. ISSN 0018-8670. doi: 10.1147/sj.281.0104.

[22] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi. Software grand exposure: SGX cache attacks are practical. In *Proceedings of the 11th USENIX Workshop on Offensive Technologies*, 2017. doi: 10.5555/3154768.3154779.

[23] E. Bugnion, S. Devine, M. Rosenblum, J. Sugerman, and E. Y. Wang. Bringing virtualization to the x86 architecture with the original VMware Workstation. *ACM Transactions on Computer Systems*, 30(4), Nov. 2012. doi: 10.1145/2382553.2382554.

[24] R. Buhren, C. Werling, and J.-P. Seifert. Insecure until proven updated: Analyzing AMD SEV's remote attestation. In *Proceedings of the 26th ACM Conference on Computer and Communications Security*, pages 1087–1099, 2019. doi: 10.1145/3319535.3354216.

[25] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. SgxPectre: Stealing Intel secrets from SGX enclaves via speculative execution. In *Proceedings of the 2019 IEEE European Symposium on Security and Privacy*, pages 142–157, 2019. doi: 10.1109/MSEC.2019.2963021.

[26] C. Cohen, J. Forshaw, J. Horn, and M. Brand. AMD secure processor for confidential computing security review. Google Project Zero, May 2022. https://googleprojectzero.blogspot.com/2022/05/release-of-technical-report-into-amd.html.

[27] P. Colp, M. Nanavati, J. Zhu, W. Aiello, G. Coker, T. Deegan, P. Loscocco, and A. Warfield. Breaking up is hard to do: Security and functionality in a commodity hypervisor. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, pages 189–202. ACM, 2011. doi: 10.1145/2043556.2043575.

[28] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles*, pages 153–167, Oct. 2017. doi: 10.1145/3132747.3132772.

[29] V. Costan, I. Lebedev, and S. Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *Proceedings of the 25th USENIX Security Symposium*, pages 857–874, Aug. 2016. ISBN 978-1-931971-32-4. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/costan.

[30] A. Danial. cloc: Count lines of code, 2022. https://github.com/AlDanial/cloc.

[31] S. Dinesh, G. Garrett-Grossman, and C. W. Fletcher. SynthCT: Towards portable constant-time code. In *Proceedings of the Annual Network and Distributed System Security Symposium*, Feb. 2022. doi: 10.14722/ndss.2022.24215.

[32] Y. Dong, X. Yang, X. Li, J. Li, K. Tian, and H. Guan. High performance network virtualization with SR-IOV. In *Proceedings of the 16th IEEE International Symposium on High-Performance Computer Architecture*, pages 1–10. IEEE, Jan. 2010. doi: 10.1109/HPCA.2010.5416637.

[33] *CoreMark PRO*. EEMBC, July 2019. v1.1.2743 https://www.eembc.org/coremark-pro.

[34] P. Emelyanov. Checkpoint/restore in userspace (CRIU), 2022. https://criu.org.

[35] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles*, pages 287–305, Oct. 2017. doi: 10.1145/3132747.3132782.

[36] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg. Azure accelerated networking: SmartNICs in the public cloud. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation*, pages 51–66, Apr. 2018. ISBN 978-1-939133-01-4. https://www.usenix.org/conference/nsdi18/presentation/firestone.

[37] J. Gandhi, M. D. Hill, and M. M. Swift. Agile paging: Exceeding the best of nested and shadow paging. In *Proceedings of the 43rd IEEE International Symposium on Computer Architecture*, pages 707–718, 2016. doi: 10.1109/ISCA.2016.67.

[38] *Gartner Says Worldwide IaaS Public Cloud Services Market Grew 40.7% in 2020*. Gartner, June 2021. https://www.gartner.com/en/newsroom/press-releases/2021-06-28-gartner-says-worldwide-iaas-public-cloud-services-market-grew-40-7-percent-in-2020.

[39] S. Gast, J. Juffinger, M. Schwarzl, G. Saileshwar, A. Kogler, S. Franza, M. Kostl, and D. Gruss. SQUIP: Exploiting the scheduler queue contention side channel. In *Proceedings of the 44th IEEE Symposium on Security and Privacy*, pages 468–484, 2023. doi: 10.1109/SP46215.2023.00027.

[40] A. Gordon, N. Amit, N. Har'El, M. Ben-Yehuda, A. Landau, A. Schuster, and D. Tsafrir. ELI: Bare-metal performance for I/O virtualization. In *Proceedings of the 17th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 411–422, 2012. doi: 10.1145/2150976.2151020.

[41] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller. Cache attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security*, pages 1–6, 2017. doi: 10.1145/3065913.3065915.

[42] O. Hadary, L. Marshall, I. Menache, A. Pan, E. E. Greeff, D. Dion, S. Dorminey, S. Joshi, Y. Chen, M. Russinovich, and T. Moscibroda. Protean: VM allocation service at scale. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, pages 845–861, Nov. 2020. ISBN 978-1-939133-19-9. https://www.usenix.org/conference/osdi20/presentation/hadary.

[43] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: Cold boot attacks on encryption keys. In *Proceedings of the 17th USENIX Security Symposium*, pages 45–60, July 2008. https://www.usenix.org/conference/17th-usenix-security-symposium/lest-we-remember-cold-boot-attacks-encryption-keys.

[44] F. Hetzelt and R. Buhren. Security analysis of encrypted virtual machines. *ACM SIGPLAN Notices*, 52 (7):129–142, 2017. doi: 10.1145/3050748.3050763.

[45] *MultiZone Security Reference Manual*. HEX-Five, Sept. 2020. https://github.com/hex-five/multizone-sdk/raw/8c92f55/manual.pdf.

[46] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013. doi: 10.1145/2487726.2488370.

[47] J.-C. Huang, M. Monchiero, Y. Turner, and H.-H. S. Lee. Ally: OS-transparent packet inspection using sequestered cores. In *7th IEEE Symposium on Architectures for Networking and Communications Systems*, pages 1–11, 2011. doi: 10.1109/ANCS.2011.11.

[48] Intel. Resources and response to side channel variants 1, 2, 3, Aug. 2018. https://www.intel.com/content/www/us/en/architecture-and-technology/side-channel-variants-1-2-3.html.

[49] *Intel Virtualization Technology for Directed I/O Architecture Specification*. Intel, Apr. 2021. Order number D51397-013, rev. 3.3 https://www.intel.com/content/www/us/en/develop/download/intel-virtualization-technology-for-directed-io-architecture-specification.html.

[50] *Software Guard Extensions Programming Reference*. Intel Corp., Oct. 2014. Ref. #329298-002 https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf.

[51] *Intel SGX*. Intel Corp., June 2015. Ref. #332680-002 https://www.intel.com/content/dam/develop/external/us/en/documents/332680-002-610985.pdf.

[52] *Control-flow Enforcement Technology Preview*. Intel Corp., June 2016. Ref. #334525-001 https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf.

[53] *Intel Trust Domain CPU Architectural Extensions*. Intel Corp., Sept. 2020. Ref. #343754-001US https://software.intel.com/content/dam/develop/external/us/en/documents/intel-tdx-cpu-architectural-specification.pdf.

[54] *Intel TDX Connect Architecture Specification*. Intel Corp., May 2021. https://www.intel.com/content/www/us/en/content-details/773614/intel-tdx-connect-architecture-specification.html.

[55] S. Jin, J. Ahn, S. Cha, and J. Huh. Architectural support for secure virtualization under a vulnerable hypervisor. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 272–283, 2011. doi: 10.1145/2155620.2155652.

[56] S. Johnson. Intel SGX and side-channels. Intel Developer Zone, Feb. 2018. https://web.archive.org/web/20200228140427/https://software.intel.com/en-us/articles/intel-sgx-and-side-channels.

[57] V. Kanchanahalli. Power your Azure GPU workstations with flexible GPU partitioning. Azure Blog, Mar. 2020. https://azure.microsoft.com/en-us/blog/power-your-azure-gpu-workstations-with-flexible-gpu-partitioning.

[58] E. Keller, J. Szefer, J. Rexford, and R. B. Lee. NoHype: Virtualized cloud infrastructure without the virtualization. In *Proceedings of the 37th IEEE International Symposium on Computer Architecture*, pages 350–361, June 2010. doi: 10.1145/1815961.1816010.

[59] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *Proceedings of the 41st IEEE International Symposium on Computer Architecture*, pages 361–372, 2014. doi: 10.1109/ISCA.2014.6853210.

[60] P. Kocher. Conference presentation of Kocher et al. [62], May 2019. https://youtu.be/zOvBHxMjNls.

[61] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Proceedings of the 19th International Cryptology Conference*, pages 388–397. Springer, Aug. 1999. doi: 10.1007/3-540-48405-1_25.

[62] 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 *Proceedings of the 40th IEEE Symposium on Security and Privacy*, pages 1–19, 2019. doi: 10.1109/SP.2019.00002.

[63] P. Kutch. PCI-SIG SR-IOV primer: An introduction to SR-IOV technology. Intel application note 321211–002, Jan. 2011. https://www.intel.com/content/dam/doc/white-paper/pci-sig-single-root-io-virtualization-support-in-virtualization-technology-for-connectivity-paper.pdf.

[64] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the 15th ACM European Conference on Computer Systems*, pages 1–16, Apr. 2020. doi: 10.1145/3342195.3387532.

[65] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *Proceedings of the 26th USENIX Security Symposium*, pages 557–574, 2017. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/lee-sangho.

[66] M. Li, W. Zang, K. Bai, M. Yu, and P. Liu. MyCloud: Supporting user-configured privacy protection in cloud computing. In *Proceedings of the 29th ACM Annual Computer Security Applications Conference*, pages 59–68, 2013. doi: 10.1145/2523649.2523680.

[67] M. Li, Y. Zhang, Z. Lin, and Y. Solihin. Exploiting unprotected I/O operations in AMD's secure encrypted virtualization. In *Proceedings of the 28th USENIX Security Symposium*, pages 1257–1272, Aug. 2019. ISBN 978-1-939133-06-9. https://www.usenix.org/conference/usenixsecurity19/presentation/li-mengyuan.

[68] M. Li, L. Wilke, J. Wichelmann, T. Eisenbarth, R. Teodorescu, and Y. Zhang. A systematic look at ciphertext side channels on AMD SEV-SNP. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy*, pages 1541–1541, 2022. doi: 10.1109/SP46214.2022.9833768.

[69] X. Li, X. Li, C. Dall, R. Gu, J. Nieh, Y. Sait, and G. Stockwell. Design and verification of the Arm confidential compute architecture. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation*, July 2022. https://www.usenix.org/conference/osdi22/presentation/li.

[70] J. T. Lim and J. Nieh. Optimizing nested virtualization performance using direct virtual hardware. In *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 557–574, 2020. ISBN 9781450371025. doi: 10.1145/3373376.3378467.

[71] *The Devicetree Specification*. Linaro, 0.4-rc1 edition, Nov. 2021. https://www.devicetree.org/specifications.

[72] 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 Security Symposium*, pages 973–990, Aug. 2018. ISBN 978-1-939133-04-5. https://www.usenix.org/conference/usenixsecurity18/presentation/lipp.

[73] R. Liu, K. Klues, S. Bird, S. Hofmeyr, K. Asanović, and J. Kubiatowicz. Tessellation: Space-time partitioning in a manycore client OS. In *Proceedings of the 1st USENIX Workshop on Hot Topics in Parallelism*, Mar. 2009. https://www.usenix.org/legacy/events/hotpar09/tech/full_papers/liu/liu.pdf.

[74] A. Markuze, I. Smolyar, A. Morrison, and D. Tsafrir. DAMN: Overhead-free IOMMU protection for networking. In *Proceedings of the 23rd ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 301–315, Mar. 2018. doi: 10.1145/3173162.3173175.

[75] R. J. Masti, C. Marforio, K. Kostiainen, C. Soriente, and S. Capkun. Logical partitions on many-core platforms. In *Proceedings of the 31st ACM Annual Computer Security Applications Conference*, pages 451–460, 2015. doi: 10.1145/2818000.2818026.

[76] *Hyper-V HyperClear Mitigation for L1 Terminal Fault*. Microsoft, Aug. 2018. https://techcommunity.microsoft.com/t5/virtualization/hyper-v-hyperclear-mitigation-for-l1-terminal-fault/ba-p/382429.

[77] *Managing Hyper-V hypervisor scheduler types: The core scheduler*. Microsoft, Dec. 2021. https://docs.microsoft.com/windows-server/virtualization/hyper-v/manage/manage-hyper-v-scheduler-types#the-core-scheduler.

[78] *About Azure DCasv5/ECasv5-series confidential virtual machines*. Microsoft Azure, Nov. 2021. https://docs.microsoft.com/en-us/azure/confidential-computing/confidential-vm-overview.

[79] *Maintenance for virtual machines in Azure*. Microsoft Azure, Oct. 2021. https://docs.microsoft.com/en-us/azure/virtual-machines/maintenance-and-updates.

[80] *B-series burstable virtual machine sizes*. Microsoft Azure, June 2022. https://docs.microsoft.com/en-us/azure/virtual-machines/sizes-b-series-burstable.

[81] *Azure compute unit*. Microsoft Azure, Apr. 2022. https://docs.microsoft.com/en-us/azure/virtual-machines/acu.

[82] A. Moghimi, G. Irazoqui, and T. Eisenbarth. CacheZoom: How SGX amplifies the power of cache attacks. In *Proceedings of the International Conference on Cryptographic Hardware and Embedded Systems*, pages 69–90. Springer, 2017. https://eprint.iacr.org/2017/618.

[83] L. Nelson and X. Wang. Developing security monitors on RISC-V: Case studies on HiFive Unleashed. Technical Report UW-CSE-2019-11-01, University of Washington, Nov. 2019. URL https://unsat.cs.washington.edu/papers/nelson-hifive-tr.pdf.

[84] K. T. Nguyen. Usage models for cache allocation technology in the Intel Xeon processor E5 v4 family, Feb. 2016. https://www.intel.com/content/www/us/en/developer/articles/technical/cache-allocation-technology-usage-models.html.

[85] U. G. A. Office. Solarwinds cyberattack demands significant federal and private-sector response, Apr. 2021. https://www.gao.gov/blog/solarwinds-cyberattack-demands-significant-federal-and-private-sector-response-infographic.

[86] M. Oliverio, K. Razavi, H. Bos, and C. Giuffrida. Secure page fusion with VUsion. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles*, pages 531–545, 2017. doi: 10.1145/3132747.3132781.

[87] T. Palit, Y. Shen, and M. Ferdman. Demystifying cloud benchmarking. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 122–132, Apr. 2016. doi: 10.1109/ISPASS.2016.7482080.

[88] A. Panwar, R. Achermann, A. Basu, A. Bhattacharjee, K. Gopinath, and J. Gandhi. Fast local page-tables for virtualized NUMA servers with vMitosis. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 194–210, 2021. doi: 10.1145/3445814.3446709.

[89] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard. DRAMA: Exploiting DRAM addressing for Cross-CPU attacks. In *Proceedings of the 25th USENIX Security Symposium*, pages 565–581, 2016. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/pessl.

[90] I. Puddu, M. Schneider, M. Haller, and S. Čapkun. Frontal attack: Leaking Control-Flow in SGX via the CPU frontend. In *Proceedings of the 30th USENIX Security Symposium*, pages 663–680, 2021. https://www.usenix.org/conference/usenixsecurity21/presentation/puddu.

[91] H. Ragab, A. Milburn, K. Razavi, H. Bos, and C. Giuffrida. CrossTalk: Speculative data leaks across cores are real. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy*, May 2021. doi: 10.1109/SP40001.2021.00020.

[92] Research and Markets. Bare metal cloud market by service type, organization size, vertical, and region – global forecast to 2026, Apr. 2021. Report number 5316699 https://www.researchandmarkets.com/reports/5316699/bare-metal-cloud-market-by-service-type-compute.

[93] *The RISC-V Instruction Set Manual, Voume II: Privileged Architecture*. RISC-V International, June 2019. Ver. 20190608-Priv-MSU-Ratified https://riscv.org/risc-v-isa.

[94] *RISC-V Open Source Supervisor Binary Interface (OpenSBI)*. RISC-V International, Jan. 2021. https://github.com/riscv/opensbi.

[95] *OpenSBI Domain Support*. RISC-V OpenSBI, Nov. 2020. https://github.com/riscv/opensbi/blob/c0d2baa/docs/domain_support.md.

[96] J. R. Sanchez Vicarte, P. Shome, N. Nayak, C. Trippel, A. Morrison, D. Kohlbrenner, and C. W. Fletcher. Opening Pandora's Box: A systematic study of new ways microarchitecture can leak private data. In *Proceedings of the 48th IEEE International Symposium on Computer Architecture*, pages 347–360, 2021. doi: 10.1109/ISCA52012.2021.00035.

[97] J. R. Sanchez Vicarte, M. Flanders, R. Paccagnella, G. Garrett-Grossman, A. Morrison, C. W. Fletcher, and D. Kohlbrenner. Augury: Using data memory-dependent prefetchers to leak data at rest. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy*, 2022. doi: 10.1109/SP46214.2022.00089.

[98] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss. ZombieLoad: Cross-privilege-boundary data sampling. In *Proceedings of the 26th ACM Conference on Computer and Communications Security*, pages 753–768, 2019. doi: 10.1145/3319535.3354252.

[99] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, pages 335–350, 2007. doi: 10.1145/1294261.1294294.

[100] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, K. Kono, S. Chiba, Y. Shinjo, and K. Kato. BitVisor: A thin hypervisor for enforcing I/O device security. In *Proceedings of the 5th ACM International Conference on Virtual Execution Environments*, pages 121–130, 2009. doi: 10.1145/1508293.1508311.

[101] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena. Preventing page faults from telling your secrets. In *Proceedings of the 11th ACM Asia Conference on Computer and Communications Security*, pages 317–328, 2016. doi: 10.1145/2897845.2897885.

[102] *SiFive FU540-C000 Manual*. SiFive, 1.0 edition, Apr. 2018. https://static.dev.sifive.com/FU540-C000-v1.0.pdf.

[103] *SiFive WorldGuard White Paper, v1.2*. SiFive, Dec. 2020. https://sifive.cdn.prismic.io/sifive/aa27fffb-cf24-4077-8103-682f26141b69_WorldGuard_White_Paper_v1.2.pdf.

[104] D. Skarlatos, M. Yan, B. Gopireddy, R. Sprabery, J. Torrellas, and C. W. Fletcher. Microscope: Enabling microarchitectural replay attacks. In *Proceedings of the 46th IEEE International Symposium on Computer Architecture*, pages 318–331, 2019. doi: 10.1109/MM.2020.2986204.

[105] U. Steinberg and B. Kauer. NOVA: A microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th ACM European Conference on Computer Systems*, pages 209–222, 2010. doi: 10.1145/1755913.1755935.

[106] J. Szefer and R. B. Lee. Architectural support for hypervisor-secure virtualization. In *Proceedings of the 17th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 437–450, 2012. doi: 10.1145/2150976.2151022.

[107] J. Szefer, E. Keller, R. B. Lee, and J. Rexford. Eliminating the hypervisor attack surface for a more secure cloud. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, pages 401–412, 2011. doi: 10.1145/2046707.2046754.

[108] B. Teabe, P. Yuhala, A. Tchana, F. Hermenier, D. Hagimont, and G. Muller. (No)Compromis: Paging virtualization is not a fatality. In *Proceedings of the 17th ACM International Conference on Virtual Execution Environments*, pages 43–56, 2021. doi: 10.1145/3453933.3454013.

[109] *Advanced Configuration and Power Interface (ACPI) Specification*. UEFI Forum, 6.4 edition, Jan. 2021. https://uefi.org/specifications.

[110] J. Van Bulck, F. Piessens, and R. Strackx. SGX-Step: A practical attack framework for precise enclave execution control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, 2017. ISBN 978-1-4503-5097-6. doi: 10.1145/3152701.3152706.

[111] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *Proceedings of the 26th USENIX Security Symposium*, pages 1041–1056, 2017. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/van-bulck.

[112] J. Van 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 *Proceedings of the 27th USENIX Security Symposium*, pages 991–1008, 2018. ISBN 978-1-939133-04-5. https://www.usenix.org/conference/usenixsecurity18/presentation/bulck.

[113] J. Van Bulck, F. Piessens, and R. Strackx. Nemesis: Studying microarchitectural timing leaks in rudimentary CPU interrupt logic. In *Proceedings of the 25th ACM Conference on Computer and Communications Security*, pages 178–195, 2018. doi: 10.1145/3243734.3243822.

[114] 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 *Proceedings of the 40th IEEE Symposium on Security and Privacy*, pages 88–105, May 2019. doi: 10.1109/SP.2019.00087.

[115] S. van Schaik, A. Kwong, D. Genkin, and Y. Yarom. SGAxe: How SGX fails in practice, 2020. https://sgaxe.com/files/SGAxe.pdf.

[116] S. van Schaik, M. Minkin, A. Kwong, D. Genkin, and Y. Yarom. CacheOut: Leaking data on Intel CPUs via cache evictions. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy*, pages 339–354, 2021. doi: 10.1109/SP40001.2021.00064.

[117] C. Waldspurger and M. Rosenblum. I/O virtualization. *Communications of the ACM*, 55(1):66–73, Jan. 2012. doi: 10.1145/2063176.2063194.

[118] C. A. Waldspurger. Memory resource management in VMware ESX Server. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, Dec. 2002. https://www.usenix.org/legacy/events/osdi02/tech/waldspurger/waldspurger.pdf.

[119] C. Wang, B. Urgaonkar, N. Nasiriani, and G. Kesidis. Using burstable instances in the public cloud: Why, when and how? *Proceedings of ACM on Measurement and Analysis of Computing Systems*, 1(1), June 2017. doi: 10.1145/3084448.

[120] X. Wang, Y. Shi, Y. Dai, Y. Qi, J. Ren, and Y. Xuan. TrustOSV: Building trustworthy executing environment with commodity hardware for a safe cloud. *Journal of Computers*, 9(10):2303–2314, Oct. 2014. ISSN 1796-203X. http://www.jcomputers.us/vol9/jcp0910-07.pdf.

[121] Z. Wang, C. Wu, M. Grace, and X. Jiang. Isolating commodity hosted hypervisors with HyperLock. In *Proceedings of the 7th ACM European Conference on Computer Systems*, pages 127–140, 2012. ISBN 9781450312233. doi: 10.1145/2168836.2168850.

[122] D. Wentzlaff, C. J. Jackson, P. Griffin, and A. Agarwal. Configurable fine-grain protection for multicore processor virtualization. In *Proceedings of the 39th IEEE International Symposium on Computer Architecture*, pages 464–475, 2012. doi: 10.1109/ISCA.2012.6237040.

[123] R. West, Y. Li, E. Missimer, and M. Danish. A virtualized separation kernel for mixed-criticality systems. *ACM Transactions on Computer Systems*, 34(3), June 2016. doi: 10.1145/2935748.

[124] T. Wood, G. Tarasuk-Levin, P. Shenoy, P. Desnoyers, E. Cecchet, and M. D. Corner. Memory buddies: Exploiting page sharing for smart colocation in virtualized data centers. In *Proceedings of the 5th ACM International Conference on Virtual Execution Environments*, pages 31–40, 2009. ISBN 9781605583754. doi: 10.1145/1508293.1508299.

[125] C. Wu, Z. Wang, and X. Jiang. Taming hosted hypervisors with (mostly) deprivileged execution. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium*, Feb. 2013. https://www.ndss-symposium.org/ndss2013/ndss-2013-programme/taming-hosted-hypervisors-mostly-deprivileged-execution.

[126] Y. Xia, Y. Liu, and H. Chen. Architecture support for guest-transparent VM protection from untrusted hypervisor and physical attacks. In *Proceedings of the 19th IEEE International Symposium on High-Performance Computer Architecture*, pages 246–257, 2013. doi: 10.1109/HPCA.2013.6522323.

[127] M. Xu, M. Huber, Z. Sun, P. England, M. Peinado, S. Lee, A. Marochko, D. Mattoon, R. Spiger, and S. Thom. Dominance as a new trusted computing primitive for the internet of things. In *Proceedings of the 40th IEEE Symposium on Security and Privacy*, pages 1415–1430, 2019. doi: 10.1109/SP.2019.00084.

[128] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side-channels for untrusted operating systems. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, pages 640–656, May 2015. doi: 10.1109/SP.2015.45.

[129] X. Zhang, X. Zheng, Z. Wang, Q. Li, J. Fu, Y. Zhang, and Y. Shen. Fast and scalable VMM live upgrade in large cloud infrastructure. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 93–105, 2019. doi: 10.1145/3297858.3304034.