Principled Microarchitectural Isolation on Cloud CPUs

Stavros Volos
svolos@microsoft.com
Azure Research, Microsoft
Cambridge, UK

Cédric Fournet
fournet@microsoft.com
Azure Research, Microsoft
Cambridge, UK

Jana Hofmann
t-jhofmann@microsoft.com
Azure Research, Microsoft
Cambridge, UK

Boris Köpf
boris.koepf@microsoft.com
Azure Research, Microsoft
Cambridge, UK

Oleksii Oleksenko
oleksii.oleksenko@microsoft.com
Azure Research, Microsoft
Cambridge, UK

ABSTRACT

We present Marghera, a system design that prevents cross-VM microarchitectural side-channel attacks in the cloud. Marghera is based on isolation contracts which, for a given CPU, describe partitions of physical threads and memory that prevent information leakage through shared microarchitectural resources.

We develop isolation contracts for the AMD EPYC 7543P, a modern cloud CPU. To this end, we first identify how microarchitectural resources are shared between its physical threads, including caches, cache-coherence directories, and DRAM banks. We then develop coloring schemes—that comprehensively partition these resources—using previously unknown, reverse-engineered indexing functions.

We implement Marghera in Microsoft Hyper-V and evaluate it using cloud benchmarks. Our results show that our approach effectively eliminates side-channels caused by shared microarchitectural resources with small performance overheads.

CCS CONCEPTS

• **Security and privacy → Virtualization and security**; **Side-channel analysis and countermeasures**; **Hardware reverse engineering**.

KEYWORDS

Cloud; isolation; microarchitecture; side-channels

ACM Reference Format:
Stavros Volos, Cédric Fournet, Jana Hofmann, Boris Köpf, and Oleksii Oleksenko. 2024. Principled Microarchitectural Isolation on Cloud CPUs. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3690183

1 INTRODUCTION

Cloud computing aims to provide scalable and cost-effective resources to a wide range of tenants. This involves sharing resources while isolating tenants from one another and from the cloud provider to ensure their security. For example, modern CPUs pack 100s of physical threads sharing terrabytes of DRAM, which can be flexibly assigned to many independent VMs.

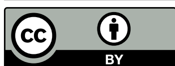
Cloud security relies on architectural isolation between trust domains, enforced by the hardware and the hypervisor via access control and permissions. Hence, a VM cannot read or write private memory pages assigned to another VM. Some CPUs also support hardware-isolated VMs [27], further hardening architectural isolation via memory encryption and access control to protect tenants against privileged attackers [25].

Unfortunately, the implicit sharing of microarchitectural resources, such caches, cache-coherence directories, and DRAM banks, between trust domains creates a large, complex, and poorly documented attack surface that can be exploited to bypass architectural isolation and leak information between VMs. Practical attacks range from classical side-channel attacks (on caches [23, 35, 54, 67, 70], directories [32, 65], DRAM [42]) to more advanced attacks that use microarchitectural resources to leak data obtained during transient execution [31, 34, 45]. As striking examples, security researchers have demonstrated how to exploit eviction sets in shared L3 caches to stream a high-quality video between two independent co-located VMs in AWS [39] or mount side-channel attacks across co-located containers in Google Cloud Run [69, 70]; and how to read the host memory [34]. Attacks are possible even if trust domains are isolated via confidential computing hardware [3, 5, 17, 33, 55, 56, 60], highlighting the need for cross-domain microarchitectural isolation.

Prior works seek to provide microarchitectural isolation via spatial and temporal partitioning of shared resources [15, 16, 40, 47, 71]. For example, the shared L3 cache can be spatially partitioned using memory coloring or hardware partitioning; and cores can be time-sliced across trust domains by scrubbing their microarchitectural state upon transitions. However, they isolate one specific resource at a time, and target resource allocation at the core granularity.

Modern CPUs pose novel challenges and opportunities for providing comprehensive spatial isolation. On one hand, there are varying granularities at which resources are shared. For instance, AMD CPUs employ a chiplet-based architecture, where the L3 cache is private to the chiplet, but shared between the chiplet's cores. On the other hand, the number and complexity of microarchitectural structures is increasing. Only a few of these can be partitioned in hardware; and existing CPU topologies (available to hypervisors) do not capture all of them, making it hard to partition them.

Approach. We present Marghera, a system design for principled and practical system-level protection against microarchitectural side-channel attacks on modern cloud CPUs. Marghera provides


This work is licensed under a Creative Commons Attribution International 4.0 License.

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3690183

exclusive-resource VMs (XVMs) by enforcing the architectural abstraction of private physical threads and private memory, against attackers that may try to circumvent architectural isolation by observing shared microarchitectural resources.

Marghera builds on *resource isolation contracts*, an abstraction of the way a CPU shares microarchitectural structures between its physical threads. In particular, a resource isolation contract describes how physical threads and physical memory pages can be assigned to trust domains to prevent information leakage between trust domains due to shared microarchitectural resources. We show how to compute isolation contracts for a given CPU: for each assignment of physical threads to trust domains, we (a) partition each microarchitectural resource that is shared between trust domains (b) *without* partitioning resources that are private to a trust domain. We achieve this using hardware mechanisms if available, and using multi-resource memory coloring if not. For the latter, we compute memory coloring schemes that respect (a) and (b) given arbitrary resources described by their *indexing functions*.

We show how to enforce isolation contracts in type-1 hypervisor-based platforms. First, we extend the *scheduler* to assign physical threads and L3 cache resources to arbitrarily shaped compute partitions, e.g., such that it never runs two trust domains on the same chiplet at the same time. Second, we extend the *memory manager* to allocate physical pages using the coloring scheme defined by the contract, to prevent allocation of the same color to two trust domains at the same time. Finally, for temporal isolation we rely on scrubbing and flushing when the VM is created and destroyed. This is acceptable because today's cloud systems are not oversubscribed and their cores are not time-sliced [71].

Implementation. We implement Marghera in Microsoft Hyper-V on a modern cloud chiplet-based CPU, AMD EPYC 7543P, that supports confidential VMs. We develop a microbenchmark suite to characterize leakage between cores and between chiplets via shared microarchitecture. We reverse engineer the indexing functions of all physically-indexed caches (L2, L3), all cache-coherence directories (chiplet and cross-chiplet), and DRAM channels. Reverse engineering the cross-chiplet directory poses particular challenges as its size, associativity, and granularity were previously unknown. We use these indexing functions to compute an isolation contract.

Evaluation Highlights. Using a collection of microbenchmarks and cloud benchmarks, we demonstrate that:

- flushing has minor performance impact;
- allocating resources at chiplet and channel granularity (i.e., coupling a chiplet with one of the local DRAM channels) incurs $< 1.8\%$ overhead while eliminating all identified side-channels;
- allocating resources at chiplet granularity incurs small overheads, up to 3.8% and 7.7% when coloring with 2MB and 4KB pages, while eliminating cache and directory side-channels;
- cache partitioning (required when multiple XVMs share a chiplet) introduces an additional overhead of 1.4–4.3%.

Contributions. We make the following contributions.

- We propose isolation contracts as a framework to constrain resource allocation for microarchitectural isolation, based on novel specifications for coloring multiple resources.
- We propose a hypervisor design that enforces contracts to provide isolation of VMs, without changing their applications.

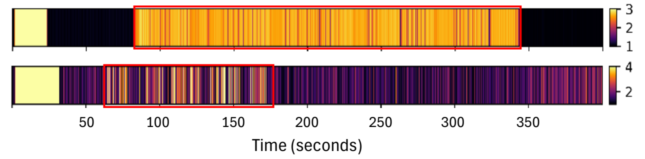


Figure 1: Heatmap visualizing leakage via the L3 cache (up) and cross-chiplet directory (below) on AMD CPUs. Black represents the smallest latency and brighter colors represent higher latencies. The Target first runs an L3 thrasher (denoted by light yellow regions) and then a Graph Analytics benchmark, denoted by the box-enclosed regions. We use two observers, the first on the same chiplet probes the L3 cache; the second on a different chiplet probes its L1-D cache to observe contention within the cross-chiplet directory. Observers measure the time to access each memory line within their probe sets. (For more details, see §6.1.)

- We reverse engineer the indexing functions of caches, directories, and DRAM channels of AMD EPYC3 CPUs, and use them to develop tight isolation contracts.
- We implement and evaluate our approach in Microsoft Hyper-V, showing its effectiveness and practicality in eliminating side-channels arising from resource sharing.

Responsible Disclosure. We shared our findings with AMD, for whose CPUs information leakage through the L3 cache and cache-coherence directories had not been demonstrated before. AMD has confirmed our findings and published a security notice [1].

2 BACKGROUND

This section outlines contemporary processor and memory architectures, and resource management in public clouds.

2.1 CPU and Memory Basics

Modern CPUs employ numerous cores, a multi-level cache hierarchy, and DRAM interfaces. Each core supports multiple physical threads (e.g., two in Intel, AMD), employs private caches (e.g., L1, L2), and a memory-management unit that maps physical memory to virtual memory via a set of page tables. Today's CPUs support variable page sizes: 4KB (small), 2MB (huge), and 1GB (giga). Finally, CPUs employ a shared L3 cache, which may be shared among the CPU's cores (Intel) or among the chiplet's cores (AMD). The L2 and L3 caches are physically-indexed.

CPUs enforce coherence across private caches in a directory that tracks memory lines kept in each private cache. For instance, L2 caches in Intel and AMD CPUs are kept coherent using a CPU [53] and chiplet [12] directory, respectively. In addition, in platforms with multiple L3 caches, these are kept coherent via an additional directory, such as the in-memory ccNUMA directory in Intel CPUs [37] and the cross-chiplet directory in AMD CPUs [51].

The L3 cache implements a cache allocation technology (CAT) allowing software to run each core with a different Class-of-Service (CoS) defined as a subset of L3 cache ways. Partitioning the cache across trust domains requires that cores in different trust domains are assigned CoS with disjoint ways.

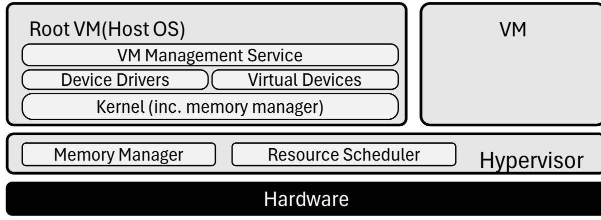


Figure 2: System software: a type-1 hypervisor managing resources, and a root VM running the host operating system.

Memory controllers serve DRAM accesses by controlling and managing memory channels. Each channel hosts several DRAM modules, each DRAM module houses multiple DRAM chips, each DRAM chip houses multiple banks, and each bank comprises multiple memory arrays organized in rows and columns. Upon a DRAM access, a set of DRAM chips, called a rank, is activated together, and the row is read into the bank's row buffer. Subsequent accesses to the same row are faster, as they are served by the row buffer [42, 58]. The row buffer must be written back to the memory array before a different row in the same bank can be accessed.

The microarchitecture discussed above affects the memory access latency observed by the core, enabling one trust domain (that shares resources with another) to use timing to observe usage patterns of shared resources. Such effects are the core of memory-based side-channel attacks. Figure 1 shows an Observer exploiting timing to observe the execution of a Target application via the L3 and cross-chiplet directory side-channels on AMD CPUs.

2.2 Resource Management in Public Clouds

In type-1 hypervisor-enabled systems (e.g., Xen, Microsoft Hyper-V), illustrated in Figure 2, the system software comprises a hypervisor and a host operating system (OS). The hypervisor boots across all physical threads and utilizes hardware virtualization to run virtual machines (VM). The host OS runs inside the root VM. While the host OS houses services for management of guest VMs and (virtual) device drivers, the hypervisor manages hardware resources. The host OS calls the hypervisor when a new VM is created and is assigned resources. The host OS allocates memory for guest VMs, but the hypervisor manages second-stage page tables (mapping guest physical memory to system physical memory), handles interrupts, and schedules virtual processors on physical threads.

The hypervisor implements two useful features for assigning compute resources to VMs: *CPU groups* for guest VMs, and *root configuration* for the root VM. Memory assignment is complicated by the fact that all physical memory is initially mapped to the root VM's address space. Thus, assignment of a private memory page to a guest VM requires the page to be unmapped by the hypervisor.

While most memory is exclusively assigned to a VM, a VM shares some memory with the root VM, required for implementing ring buffers for guest I/O. These pages are mapped to both VMs. Similarly, the hypervisor and VMs utilize shared memory for hypercalls.

The host OS's memory manager supports multiple page sizes by maintaining a free list for each page size. The memory manager may support page coloring by dividing each list across colors.

3 RESOURCE ISOLATION CONTRACTS

In this section, we develop *resource isolation contracts*, an abstraction that reflects the way a CPU shares microarchitectural (uarch) structures between its physical threads. These contracts provide the hypervisor with the information required to allocate architectural resources in a way that eliminates information flow between trust domains, both architecturally and microarchitecturally (see §4). In this paper, we represent isolation contracts in mathematical language, and we infer them for a given CPU by reverse-engineering (see §5). In the future, side-channel conscious CPU vendors may choose to release them as part of the CPU specification, and represent them in a machine-readable domain-specific language.

3.1 Partitioning Resources

We model isolation contracts using (mathematical) partitions. A *partition* $P = \{C_0, \ldots, C_{p-1}\}$ of a set S is a collection of pairwise disjoint sets $C_i \subseteq S$ whose union equals S. We call each set C_i a *class*, or a *color*. A partition P is *finer* than a partition Q, written $P \sqsubseteq Q$, if each class of P is included in a class of Q.

We consider partitions on two kinds of architectural resources: *Compute*, represented as physical threads $T = \{t_0, \ldots, t_{n-1}\}$, and *Memory*, represented as physical addresses $M = \{0, \ldots, 2^m - 1\}$.

Example 1. *The partition $P_{Core} = \{\{t_0, t_1\}, \{t_2, t_3\}, \ldots\}$ groups the set of physical threads T into pairs, corresponding to physical cores. The partitions $P_{4K} = \{\{0, \ldots, 2^{12}-1\}, \ldots\}$ and $P_{2M} = \{\{0, \ldots, 2^{21}-1\}, \ldots\}$ divide physical memory into small and huge pages, respectively, where $P_{4K} \sqsubseteq P_{2M}$.*

We can combine partitions of disjoint sets as the disjoint set union $\uplus$ of their classes. Hence, $P_{Core} \uplus P_{4K}$ defines a partition of all resources in T and M into cores and small pages.

3.2 Allocating Resources for uarch Isolation

We describe the exclusive allocation of resources to trust domains $d \in D$ as a partition P_D of $T \uplus M$, where each trust domain is assigned a class $C_d \in P_D$. We then use the partition refinement relation $P \sqsubseteq P_D$ to express constraints on resource allocation.

Example 2. *$P_{Core} \uplus P_{4K} \sqsubseteq P_D$ captures the requirement that compute in P_D must be allocated at core granularity (i.e., never sharing cores between trust domains) and memory at 4KB page granularity (i.e., never sharing pages between trust domains).*

We call a partition P_D of resources into trust domains *isolating* if the use of these resources by a trust domain does not leak any information to any other trust domain. However, not every partition is isolating, as microarchitectural resources (e.g., caches) can still be shared across trust domains, resulting in observable side-channels.

We call allocation constraints that ensure that all P_D are isolating, as *resource isolation contracts*:

Definition 1 (Resource Isolation Contract). *A partition $P_T \uplus P_M$ is a* resource isolation contract *if every P_D with $P_T \uplus P_M \sqsubseteq P_D$ is isolating.*

To define an isolation contract for a given CPU, we first identify shared microarchitectural resources. For a given partition P_T of compute resources, a microarchitectural resource is *private* when its implementation is local to each class of P_T and *shared* otherwise.

Example 3. *For partition P_{Core}, on-core resources, such as buffers, TLBs, predictors, L1/L2 caches are private, whereas off-core resources, such as L3, cache-coherence directories, and DRAM are shared.*

With cache allocation technology, the L3 cache can be turned into a private resource for any allocation of cores to trust domains. For most microarchitectural resources (e.g., directories, DRAM banks) such hardware mechanisms are not available.

Instead, we leverage the fact that any partition of M implicitly also partitions all microarchitectural resources that are indexed from memory, which is known as *coloring*.

Example 4. *Consider an indexing function $f_{Dir}(x) = x_6, x_7, \ldots, x_{16}$ mapping addresses to 2^{11} directory sets. The set $P_{Dir} = \{f^{-1}(i) \mid i \in \mathbb{F}_2^{11}\}$ of preimages of f_{Dir} forms a partition of M. Enforcing $P_{Dir} \sqsubseteq P_M$ ensures that no two trust domains share a directory set, which eliminates directory side-channel attacks.*

For comprehensive side-channel protection, it is not sufficient to color individual resources in isolation. Thus, we identify global requirements on P_M such that $P_T \uplus P_M$ defines an isolation contract, and we show how they can be fulfilled by multi-resource coloring.

3.3 Multi-resource Memory Coloring

The memory partition P_M of an isolation contract $P_T \uplus P_M$ should meet the following criteria:

(1) P_M complies with all architectural constraints, e.g., supports page-based allocation: $P_{4K} \sqsubseteq P_M$, and ideally also $P_{2M} \sqsubseteq P_M$. (See §3.3.1.)
(2) P_M partitions *all shared* microarchitectural resources f we have identified to achieve isolation. (See §3.3.1.)
(3) P_M does *not* partition any *private* microarchitectural resources, as this would degrade performance. (See §3.3.2.)

3.3.1 Coloring for Partitioning Multiple Resources. To satisfy requirements (1) and (2) we color different resources simultaneously, which technically amounts to computing the *join* of two partitions.

Definition 2. *Given two partitions P and Q, their* join $P \sqcup Q$ *is the finest partition that is coarser than both P and Q.*

For simple indexing functions, we can compute the join of their partitions by taking the *intersection* of their common bits, as this is the strongest constraint that is still satisfied by both.

Example 5. *P_{4K} is represented by the preimages of the function $f_{4K}(x) = x_{12}, x_{13}, \ldots$ which is the projection to the page frame number. The join $P_{Dir} \sqcup P_{4K}$ is given by the common bits of f_{4K} and f_{Dir} from Example 4, i.e., $x_{12}, \ldots, x_{16}$, which amounts to $2^5 = 32$ colors.*

We can compute a joint memory coloring scheme for an arbitrary number of microarchitectural resources, albeit at the cost of reducing the number of colors.

Example 6. *For additionally coloring DRAM (e.g., on Haswell [42]), we further intersect with bit x_{15}, which selects the rank. $P_{Dir} \sqcup P_{4K} \sqcup P_{DRAM}$ is represented by a single bit x_{15} and supports two colors.*

3.3.2 Coloring Without Partitioning Private Resources. When assigning a color to a trust domain, we may inadvertently partition its *private* resources, leading to performance loss due to underutilization. To satisfy requirement (3) we must ensure that, when restricting memory to a color C, each set of a private resource (represented by a preimage B of its indexing function) is still used in identical proportions. The following definition formalizes this requirement for memory partitions.

Definition 3. *Two partitions P and Q of a set are* orthogonal, *written $P \perp Q$, if for every color $C \in P$ and every $B \in Q$ we have*

$$\frac{|B \cap C|}{|C|} = \frac{|B|}{|M|}$$

As an example, partitions defined by linear functions that operate on disjoint address bits are orthogonal. For other indexing functions, we can ensure that their partitions are orthogonal by removing from the coloring function all bits that are shared with the private resource, as this would inadvertently partition the resource.

Example 7. *Consider a private L2 cache given by the indexing function $f_{L2}(x) = x_6, \ldots, x_{14}$ and the coloring $P_{Dir} \sqcup P_{4K}$ from Example 5. Achieving orthogonality $P_{L2} \perp (P_{Dir} \sqcup P_{4K})$ requires dropping x_{12}, x_{13}, x_{14} and leaves bits x_{15}, x_{16} for coloring, totalling four colors.*

3.3.3 Discussion. For the indexing functions of our target AMD CPU (see §5.3), taking the intersection of common bits is sufficient for computing joins and removing shared bits is sufficient for achieving orthogonality. Concurrent work [21] explores the theory and algorithms for computing joins and achieving orthogonality of arbitrary linear (and partially linear) functions, which makes our approach applicable to a broader set of current and future CPUs.

4 MARGHERA

Marghera provides the abstraction of exclusive-resource VMs (XVMs) by adapting a type-1 hypervisor architecture against attackers that can observe shared microarchitectural resources. In this section, we first present our threat model, then present our design assuming the host is trusted, including the hypervisor and the root VM, and finally we show how to extend it if the root VM is untrusted.

4.1 Threat Model

We consider an attacker that runs on the same CPU as the victim VM and can control any other VM, possibly including the root VM. We assume this attacker can monitor the fine-grained usage of shared resources, such as caches (e.g., [22, 35]), directories (e.g., [32, 65]) and DRAM banks (e.g., [42]) by observing *conflicts* in these structures. Conversely, we assume the hypervisor is not adversarial and can be trusted to enforce isolation contracts.

Coarse-grained attacks that observe *overall congestion* on shared memory bus (e.g., [63]) or the on-chip interconnects (e.g., [41, 59]) are out of scope—some are mitigated by our approach, see §5.3.

Side-channels attacks due to sharing storage, networking devices, or other PCIe devices, as well as physical side-channel (e.g., power, EM radiation analysis) and fault attacks are out of scope.

Side-channel attacks due to usage patterns of intentionally *shared memory* between VMs (e.g., ring buffers used for communication between the guest and root VMs for storage/network I/O) are out of scope. Finally, cross-domain side-channel attacks that exploit leakage via global hypervisor state or shared colors between the hypervisor and victim VM are out of scope.

4.2 Core Design

Marghera is designed around two high-level requirements. The first is to provide cross-domain microarchitectural isolation for XVMs that share a server CPU. The second is to allow for running XVMs and conventional VMs on the same host. Whereas XVMs are protected from any other co-located VMs, conventional VMs can still freely pool their assigned resources, which can be time-shared.

4.2.1 Overview. Marghera implements XVMs by enforcing two complementary properties:

(1) **Spatial Isolation (by Partitioning)**. An XVM is assigned resources whose microarchitectural state cannot be observed or altered by other running VMs.

(2) **Temporal Isolation (by Scrubbing and Flushing)**. An XVM is assigned resources whose state does not depend on previous VMs and cannot be observed by future VMs.

Partitioning. The system software enforces the *memory partition* (P_M) and *compute partition* (P_C) specified by a fixed, global CPU resource isolation contract (§3). Each XVM is exclusively assigned a subset of the colors of the memory partition during creation, and a compute partition class during initial scheduling. This compute partition class is assigned to the VM for its entire lifetime based on the observation that the host is not over-subscribed; while not essential, this amortizes expensive microarchitectural flushing operations.

Scrubbing and Flushing. While system software always zeroes memory and registers between successive assignments, comprehensive and efficient microarchitectural flushing requires hardware support. The hypervisor scheduler flushes the cache hierarchy (and as a consequence any entries in the directories) both during the initial assignment of resource to the XVM and, eventually, after destroying the XVM and before re-cycling its resources.

4.2.2 Memory Management. The memory coloring specified in the isolation contract (P_M) is enforced in two stages: by assigning free colors to VMs at creation time, then by allocating physical pages using only the colors assigned to the VM. Both color assignment and memory allocation are managed by the host OS memory manager, whereas the hypervisor may also monitor the exclusive coloring invariant, as discussed in §4.3.

The memory manager assigns colors to VMs by setting a bitmap with enough colors to match the VM memory size. (XVMs use exclusive bitmaps, whereas conventional VMs can pool their colors in the same bitmap.) The memory manager uses this bitmap to allocate free pages within the assigned colors.

The memory manager assigns colors and allocates pages to meet two goals. First, during color assignment, it *minimizes memory fragmentation* by favouring the assignment and allocation of larger page (e.g., 2MB) of contiguous colors. Colors are assigned at a finer granularity only when the VM memory size is *smaller than* or *not multiple of* the memory span of a huge page. Second, during page allocation, it *maximizes the use of partitioned and non-partitioned resources* by iterating over the color bitmap so that page allocations are uniformly spread over all assigned colors. (This complements the contract design goal that each color is uniformly spread across non-partitioned resources, see Definition 3.)

Efficient memory allocation requires that the memory manager maintains multiple free page lists, for each size and color. Allocating a page (if the free list is empty) requires demoting a chain of larger pages so that page(s) comprising demoted large page(s) are added to the corresponding free lists. In case a larger page (e.g., 2MB) demotion is required, a free large page that includes a page of the requested color is selected using an inverted coloring function that returns a page offset given a page frame number and a color. Multiple offsets in a page may yield a requested color, in which case the memory manager randomly choses one of them.

4.2.3 Compute Resource Management. The hypervisor's resource scheduler enforces the invariant that an XVM is exclusively assigned a compute partition class. To maximize utility, the resource scheduler supports conventional VMs, which are allowed to share their assigned resources. The following invariants are enforced:

(1) A VM is either an XVM or a conventional VM throughout its lifetime;

(2) An XVM is exclusively assigned a compute partition class throughout its lifetime;

(3) A compute partition class assigned to a running XVM cannot be unassigned during its lifetime; and

(4) Microarchitectural resources are flushed between successive assignments.

We implement compute partition classes by extending *CPU Groups*. Using CPU groups, each VM can be bound to a group, and each group has a *ThreadAffinity* property that specifies the physical threads available for scheduling the virtual processors of bound VMs. We add two new properties for cache partitioning and XVMs: *CacheWayAffinity* specifies the cache ways available to the groups's physical threads; and *ResourceExclusivity* specifies that the resources of the group are exclusive to a VM.

To enforce Invariants (1) and (2), we require that the group properties be immutable while a VM is bound to the group; and that, when *ResourceExclusivity* is set, at most one VM is bound to the group—making it an XVM for its lifetime.

A VM can be assigned to a resource-exclusive group only when none of the group's threads and L3 cache ways are already assigned to other running VMs, and, if successful, this exclusive assignment blocks their assignment to other running VMs. Note that the exclusivity of L3 cache ways is enforced only when the implemented contract P_m does not partition the L3 cache.

While the exclusive resource invariant is enforced at the time of group-VM binding, not all VMs are bound to a group; and hence they may still run on any physical thread. To overcome this limitation, the scheduler maintains *availability* metadata that track physical threads (*AvailableThreads*) and L3 cache ways (*AvailableL3CacheWays*) that have not been exclusively assigned to an alive resource-exclusive group, i.e., there is a running VM bound to the group. This metadata are updated accordingly whenever a resource-exclusive group is assigned an XVM or released from an XVM binding. Before completing an XVM binding (but after updating availability metadata), the scheduler drains the physical threads and migrates the preempted virtual processors to available physical threads based on the updated *AvailableThreads*. Once draining is complete, it flushes their microarchitectural state.

Enforcing Invariant (3) requires the hypervisor to not unbind a running XVM. Coupled with immutability of the group's properties, it is guaranteed that none of their resources can be unassigned.

Enforcing Invariant (4) requires the group's microarchitectural resources to be flushed before an XVM binding, and before reclaiming any resources. Once an XVM is destroyed, its binding is no longer active; its resources are made available to non-bound VMs.

4.3 Removing the Host OS from the TCB

We now discuss how to extend our design to remove the host OS from the TCB. While the host OS remains responsible for assigning resources to guest VMs, the hypervisor must enforce isolation and exclusivity invariants, and refuse to create or schedule a VM that would break the isolation contract. That is, the hypervisor must additionally enforce microarchitectural isolation of XVMs from the root VM and monitor the exclusive coloring invariant.

4.3.1 Isolating XVMs from Root VM. The *root configuration* feature restricts physical threads on which the root VM is scheduled. This feature must provide a configuration mode, in which the root VM memory size is specified. (E.g., Xen already supports such mode.)

In this mode, the hypervisor maps to the root VM address space as many memory colors as needed to fullfil its memory size requirements. Similarly, the memory manager of the root VM (which supports two memory compartments) uses the root compartment for its own allocations, initialized from the memory assigned at boot time, and uses the VM reserve compartment for guest VM allocations, initialized from the remaining memory.

The hypervisor also maps to the root VM one designated color from which the root VM allocates memory to be shared with VMs.

4.3.2 Invariant Enforcement. The hypervisor ensures that XVMs do not share memory colors or compute resources with root VM.

Memory Partitions. The hypervisor monitors the exclusive coloring invariant during color assignment to VMs and during mapping of a physical page to a VM's address space.

The hypervisor tracks VM ownership of memory colors and whether they are exclusive or not. This metadata needs to be updated whenever the root VM's memory manager assigns colors to a VM. The hypervisor must reject re-assignment of exclusive colors.

During memory allocation, the root VM calls the hypervisor to map a range of memory pages to the VM's address space. The hypervisor accepts mappings based on whether the page is exclusive or shared. The hypervisor must reject any request to map to a VM address space any exclusive page within a color assigned to an XVM, and to map to an XVM address space any exclusive page within an unassigned color or a color already assigned to other (X)VMs. The hypervisor must accept requests to map a shared page to a VM's address space only if the page lies within the designated non-exclusive color owned by the root VM.

Compute Partitions. During creation of a CPU group, the hypervisor must enforce the invariant that none of the threads assigned to the root VM are ever assigned to a resource-exclusive group. The scheduler (§4.2.3) implicitly restricts the L3 cache ways of the root VM, as soon as an XVM is allocated on physical threads that belong to the same L3 cache domain as the root VM's physical threads.

5 IMPLEMENTATION

We implement Marghera in Microsoft Hyper-V on a modern cloud CPU, the AMD EPYC 7543P. We describe its architecture, investigate leakage via its microarchitectural structures, and finally compute isolation contracts.[1] Table 3 summarizes all its indexing functions when memory interleaving is disabled and each memory channel is populated with a 64GB dual-rank DIMM.[2]

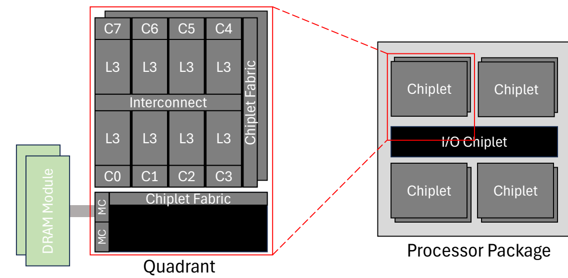


Figure 3: Chiplet-based AMD CPU architecture [12, 51].

5.1 Hardware Architecture

Figure 3 illustrates the chiplet-based AMD CPU architecture. The processor package features chiplets connected via chiplet fabric. The *I/O chiplet* has four quadrants, each connected to two *compute chiplets*, and interfaces to DRAM and I/O subsystems. The quadrants are interconnected, enabling cross-chiplet communication.

Each chiplet houses up to eight cores sharing a 16-way 32MB sliced L3 cache. Each core runs two physical threads and employs private L1-I/L1-D (8-way 32KB each) and L2 (8-way 512KB) caches. The L2 cache is inclusive, guaranteeing that L1 cache lines also reside in the L2 cache. The L3 cache acts as a victim cache to L2, meaning that lines in an L2 cache may not reside in the L3 cache. To maintain cache coherence, a sliced *chiplet directory* duplicates the tags for all L2 cache lines [12, 50]. The L3 cache implements CAT (§2) allowing software to run each core with a different Class-of-Service (CoS), defined as a subset of the 16 ways.

Memory accesses that miss in the chiplet memory hierarchy are served either by DRAM or by a remote chiplet. This decision is driven by a *cross-chiplet directory* (physically located in the I/O chiplet [51]) that maintains cache coherence between all chiplets by tracking the tags of their cache lines.

5.2 Characterizing Microarchitectural Leaks

As a basis for defining isolation contracts (see §3) we now infer how the AMD CPU shares and indexes microarchitectural resources.

We develop Algorithm 1 for inferring arbitrary linear indexing functions of set-associative structures. It goes beyond prior work [38] in that it only requires checking if two addresses are *congruent* (i.e., evicted by the same set) without any knowledge about the indexed structure, including the number of sets.

5.2.1 L3 Cache. We infer the L3 cache set indexing function, and validate that chiplet- and CAT-based partitioning both ensure L3 cache isolation for private memory. We first describe how to construct minimal L3 eviction sets, which are needed for both tasks.

[1] In our extended (arXiv) version, we discuss the feasibility of our approach on Intel and ARM CPUs, and provide additional details on the reverse-engineering process.
[2] We have also reverse engineered other memory configurations, where we varied the interleaving type and number of populated memory channels. While the cross-chiplet directory and DRAM channel indexing functions vary, coloring is still possible.

Algorithm 1: Inference of linear indexing functions.

(1) Pick a basis for the input space, i.e., addresses that are linearly independent.
(2) Group basis addresses into congruent groups.
(3) Find which groups represent a basis for the output space and how the other groups depend on them.
(4) Assign vectors from output space to groups while respecting dependencies; solve the resulting linear equation system.

Finding L3 Eviction Sets. Generating a minimal eviction set for a given *victim cache line* (see, e.g. [35, 57]) requires (a) a *cache allocation primitive* for allocating a cache line and (b) a *candidate set* that, when traversed and allocated in the cache, evicts the victim cache line. This set is then minimized by applying standard algorithms.

In inclusive cache hierarchies, a load instruction on the target address serves as cache allocation primitive. The situation is more complex on contemporary CPUs, where L3 cache lines are allocated only by L2 cache line evictions. We create an *L3 cache allocation primitive* by first accessing the target address and then traversing its L2 eviction set, to ensure the target address is evicted into the L3 cache. To compute L2 eviction sets, we leverage that the L2 is inclusive to L1 and use a standard algorithm, which does not require any knowledge about the L2 indexing function [57].

We populate the *L3 candidate set* from a set of L2-congruent addresses. (This choice is based on the observation that L2 and L3 cache set indexing in Intel CPUs have common bits.) This allows us to start with a small candidate set (e.g., 1024 addresses to account for multiple L3 slices), which improves efficiency in our target CPU.

Inferring L2/L3 Indexing Functions. We apply Algorithm 1 to an input basis, generating eviction sets targeting the cache sets identified by the addresses of the input basis. The algorithm yields 11 and 16 congruent groups for L2 and L3. One group is mapped to the zero group and the remaining groups are independent and represent the output basis; the dimension of L2 (L3) output basis is 10 (15) as they employ 1K (32K) cache sets.

We also identify the bits of the L3 indexing function that correspond to the 3-bit slice identifier. We leverage hardware counters to identify precisely the L3 cache slice in which each cache line of the input basis is allocated. Addresses of the input basis, whose a_8, a_7, a_6 bits are set to zero, are mapped to the 0th slice; if one of a_8, a_7, a_6 bits is set to one, they are mapped to the $4^{th}, 2^{nd}, 1^{st}$ slices, respectively.

Finally, we solve the linear system for each resource. The resulting functions are depicted in Table 3.

Validating L3 Isolation. We perform a test using two threads, a *Target* and an *Observer* that uses an L3 eviction set for one of the Target addresses. We find that when scheduled (i) on the same chiplet without CAT, the Observer can reliably evict the Target address from L3; and (ii) on different chiplets, or on different cores with CAT, the Observer can no longer evict the Target address.

5.2.2 Chiplet Coherence Directory. We hypothesize that the chiplet directory does not lead to cross-core interference. This is because it is sliced similarly to the L3 cache and it duplicates the tags of all L2 cache lines for the slice's address space [50]. It employs a multi-bank structure (one bank per physical core) and each bank mirrors the L2 for the slice.

	Interference	No Interference
L2 Miss Ratio	5.406% ± 0.153%	5.402% ± 0.159%

Table 1: Chiplet directory interference. L2 miss ratios (along with 95% confidence intervals) observed by one core with or without interference from another core.

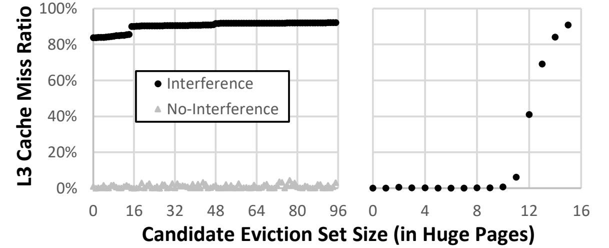


Figure 4: Cross-chiplet directory interference. On the left, we obtain a minimal eviction set by repeatedly removing a page at a time while maintaining maximal interference (L3 cache miss ratio). On the right, we confirm that the obtained eviction set is minimal by removing a page at a time.

Validating Cross-core Isolation. We validate this hypothesis by relying on the fact that directories are inclusive of caches for which they provide coherence [65]. Thus, directory allocations of one trust domain that evict directory entries of a second trust domain, result in evictions in the private caches of the second trust domain.

Our test involves an *Observer* thread on a core and a *Target* thread on each of the remaining cores of the chiplet. It proceeds in multiple rounds, each consisting of 3 steps: (1) the Observer primes its L2 cache; (2) In *odd* rounds, the Targets prime their L2 caches; (3) the Observer probes its L2 cache. Each L2 prime set is constructed from a pool of huge (2MB) pages with fixed $a_{28}, \ldots, a_{21}$ bits, including only the lower quarter (512KB) of each page. This ensures that each core will target the same directory sets, maximizing contention. (We carefully allocate memory for metadata to minimize intra-core contention between the prime set and metadata. Remaining contention is due to the thread's instruction footprint.)

If Observer's L2 lines and any of Target's L2 lines were mapped to the same directory set, the Observer probe would observe statistically different L2 cache miss ratio across odd and even rounds. Our results (shown in Table 1) indicate that one core cannot interfere with another's core execution via the chiplet directory, confirming that chiplet directories provide cross-core isolation.

5.2.3 Cross-Chiplet Coherence Directory. We show that this directory leads to interference, and then infer its indexing function.

Violating Cross-core Isolation. We develop a cross-chiplet interference test that runs an *Observer* thread on one chiplet and one *Target* thread on each of the other six chiplets. The test proceeds in multiple rounds: (1) the Observer primes one L3 cache way (i.e., 32K cache sets); (2) in *odd* rounds, each Target primes its entire L3 cache

(i.e., 16 L3 cache ways); (3) the Observer probes its L3 cache way. Each L3 prime set is constructed from a pool of huge (2MB) pages with some huge PFN (hPFN) bits fixed to obtain a good pool of candidates. In our experiments, we fix $a_{38}, a_{37}, a_{36}, a_{31}, a_{27}, \ldots, a_{21}$, obtaining 1 Observer and 96 Target pages, and confirm interference; Figure 4 (left) shows that Observer measures 90-92% of L3 miss ratio in odd rounds compared to 0-2% in even rounds. This confirms that directory evictions are observable across chiplets.

Inferring the Associativity. We compute a minimal eviction set as follows: we repeatedly run the test above while removing one page at a time from the Target's candidate set; we add the page back to the candidate set if the Observer's L3 miss ratio drops lower than 90%. Figure 4 (left) shows that we can remove all pages except for 15. Once the iteration over the initial candidate set completes, we confirm that the reduced candidate set is a minimal eviction set: we remove one of its pages at a time and measure the Observer's L3 miss ratio. Figure 4 (right) shows it drops to 0% for 10 huge pages. The sigmoid shape is a side-effect of the replacement policy.

Inferring the Indexing Function. Due to large aggregated L3 capacity, we expect that some output bits of the indexing function are defined only on hPFN bits. We therefore split the address space into hPFN bits ($a_{38}, \ldots, a_{21}$) and huge page offset bits ($a_{20}, \ldots, a_6$) and apply Algorithm 1 on the hPFN subspace first.

hPFN Bits. To apply Algorithm 1 to the hPFN subspace, we need to construct minimal eviction sets for an entire huge page. (This avoids identifying hPFN bits that are combined with huge page offset bits.) Similar to the cross-chiplet directory interference test, we form a candidate set that is congruent to the target huge page, and reduce it to 16 pages based on our associativity findings.

In step (1) of the algorithm, we choose an input basis that resembles the standard basis, i.e., each huge page has only 1-2 hPFN bits set while a_{33} is set to 1. This choice yields eight groups: one is mapped to value 0; the rest represents a basis of the output space. The output bits inferred by our algorithm are given in XD in Table 3. We observe that the address (if larger than 4 GB) is shifted by 2 GB before indexing the directory, corroborating recent findings that DRAM is accessed using addresses shifted by an offset due to some system addresses below 4 GB being mapped to PCI space [24].

In non-interleaved mode, a_{38}, a_{37}, a_{36} are included in the channel indexing function (see also §5.2.4), implying that the cross-chiplet directory is distributed across the eight memory controllers.

Remaining Bits. Rather than running Algorithm 1 for the remaining bits, we guess that they are simple functions (i.e., singletons or a combination of one huge page offset bit and hPFN bits) and adapt the cross-chiplet interference test to infer one output bit at a time.

We construct Observer and Target prime sets so that they are mapped to distinct colors. A color is a singleton from the huge page offset bits or an XOR of a huge page offset bit and a hPFN bit not used yet ($a_{38}, \ldots, a_{32}, a_{30}, \ldots, a_{28}, a_{25}$). The prime set includes entire pages to avoid identifying bits that are combined with small page offset bits. XD in Table 3 shows the findings for $a_{20}, \ldots, a_{11}$.

This process indicates that bits $a_{10}, \ldots, a_6$ are not used in the indexing function. We use a simple test to confirm that we did not overlook any non-simple output bits. We choose a value for $a_{38}, \ldots, a_{11}$ where each bit of $a_{38}, \ldots, a_{32}$ is set to zero. We then construct an eviction set that targets the zero offset ($a_{10}, \ldots, a_6$ set to 0). We flip one bit at a time (offsets 1, 2, 4, 8, 16) and check if this yields a congruent address (confirming that no bit of $a_{10}, \ldots, a_6$ is part of the indexing function). Thus, it is likely that the directory is maintaining coherence at coarse granularity [26].

Core / HPFN	0	4	8	12	16	20	24	28
0x01000	310	316	327	329	351	357	362	372
0x09000	315	310	328	329	351	357	363	371
0x11000	329	329	310	317	362	370	351	358
0x19000	329	330	316	310	362	370	351	358
0x21000	351	351	362	362	311	320	330	332
0x29000	351	351	362	363	313	319	329	330
0x31000	363	362	351	351	328	329	310	320
0x39000	363	362	350	351	329	333	319	320

Table 2: DRAM access latency observed by different cores (on different chiplets) when accessing the first memory line within a huge page. Due to the I/O chiplet being divided in quadrants, we observe four classes of latency: local / no-interconnect (white); x-axis (light grey); y-axis (dark grey); and xy-plane (black).

Discussion: Private vs. Shared. Above, each prime set is private to a chiplet, raising the question whether memory lines shared across chiplets are tracked by a different structure [26].

We refine the interference test so that each Observer and Target spans two chiplets. Each prime set is accessed by the corresponding pair of chiplets before the next prime set is accessed, ensuring that the cache lines of each prime set are marked as shared in the directory. When using XD to select the color of each prime set, we observe interference when prime sets are mapped to the same color and no interference when prime sets are mapped to different colors. This result implies that private and shared memory lines are tracked by the same structure.

5.2.4 DRAM. Identifying the physical-to-DRAM address mapping function requires a test to identify whether two input basis vectors are congruent on a bank. We could not reliably uncover statistically different timings between first and second accesses to a row. This may be due to AMD's row management policy that supports timeout-based, predictive row closing based on DRAM bank history [8]. (We also ran DARE [24], a tool for reverse engineering AMD's DRAM indexing functions, but without success.)

Instead, we exploit the variable I/O chiplet interconnect latency to infer how channels are indexed. Table 2 illustrates the latencies observed by the eight chiplets for various input basis vectors. The latencies vary depending on how the memory request is routed through the I/O-chiplet interconnect. We make the observation that flipping a_{38}, a_{37} also flips the chiplet that observes the lowest (i.e., local) DRAM access latency. Thus, in non-interleaved mode, a_{38}, a_{37} index the quadrant while a_{36} (involved in the cross-chiplet directory indexing function) indexes the quadrant's channels.

5.3 Resource Isolation Contracts

We use the experimental results of §5.2 to instantiate the isolation contracts of §3. Due to partial reverse-engineering of DRAM functions, only the chiplet scheduling contract prevents DRAM side-channels. For completeness, we include hypothetical contracts that would prevent DRAM side-channels based on bank coloring.

M	4K/2M/1G	L3/L2	XD	DRAM(<2GB)	DRAM(>4GB)	XDC	XDDC	XDDC'
a6		L2 a6						
a7		a7						
a8		a8						
a9		a9⊕a21						
a10		a10⊕a22						
a11		a11⊕a23	a11⊕a28			a11⊕a28		
a12	a12	a12⊕a24	a12⊕a29			a12⊕a29		
a13	a13	a13⊕a25	a13⊕a30			a13⊕a30		
a14	a14	a14⊕a26	a14			a14		
a15	a15	a15⊕a27	a15			a15		
a16	a16	a16	a16					
a17	a17	a17	a17					
a18	a18	a18	a18⊕a25			a18⊕a25		
a19	a19	a19	a19					
a20	a20	a20	a20					
a21	2M a21		a21			a21		
a22	a22		a22⊕a26			a22⊕a26		a22⊕a26
a23	a23		a23⊕a27			a23⊕a27		a23⊕a27
a24	a24	used above	a24⊕a31⊕ (> 4GB ? 1 : 0)			a24⊕a31⊕ (> 4GB ? 1 : 0)		
a25	a25							
a26	a26							used above
a27	a27							
a28	a28		used above			used above		
a29	a29							
a30	1G a30							
a31	a31							
a32	a32							
a33	a33				used above			
a34	a34							
a35	a35							
a36	a36			a36	a36⊕¬(a35∨…∨a31)			
a37	a37		As DRAM	a37	a37⊕¬(a36∨…∨a31)		As DRAM	As DRAM
a38	a38			a38	a38⊕¬(a37∨…∨a31)			

Table 3: Gray areas represent functions for: architectural pages (4K/2M/1G); indexing caches (L3/L2); indexing the cross-chiplet directory (XD); indexing the DRAM channel (DRAM); coloring the cross-chiplet directory *without* coloring DRAM/L3/L2 (XDC); and coloring the cross-chiplet directory and DRAM channels *without* coloring L3/L2 (XDDC). L2 indexing and 2M, 1G page frame indexing are annotated using a box within L3 and 4K. a_8, a_7, a_6 select the L3 cache and chiplet directory slice. $DRAM$ also selects the cross-chiplet directory slice. Non-linearity is due to addresses (that are larger than 4GB) being shifted by 2 GB, i.e., a_{31} is flipped, also requiring that a higher order bit (a_n) is flipped if lower order bits ($a_{n-1}, \dots, a_{31}$) are all zero. $XDDC'$ is a *hypothetical* function that refines $XDDC$ to color DRAM banks, assuming they are partially indexed by $XD's$ a_{23}, a_{22}.

Chiplet Scheduling. We first define three contracts that allocate physical threads at chiplet granularity ($P_{Chiplet} = \{\{t_0, \dots, t_7\}, \{t_8, \dots, t_{15}\}, \dots\}$). Only the third contract prevents DRAM side-channels as each chiplet is allocated an entire DRAM channel.

XDC in Table 3 defines the coloring partitions P_{XDC} computed as the finest partition that is both coarser than the index function of the cross-chiplet directory and orthogonal to the indexing functions of DRAM channels and the L3 cache. (We experimentally validated this by counting the number of L3 cache colors to which a memory is mapped to.) Likewise, XDDC in Table 3 defines the coloring partitions P_{XDDC} computed as the finest partition that is both coarser than the index functions of the cross-chiplet directory slice and DRAM channels, and orthogonal to the L3 cache indexing function. We obtain three contracts:

(1) $P_{Chiplet} \uplus (P_{4K} \sqcup P_{XDC})$ supports 4KB pages, 512 colors;
(2) $P_{Chiplet} \uplus (P_{2M} \sqcup P_{XDC})$ supports 2MB pages, 16 colors;
(3) $P_{Chiplet} \uplus (P_{1G} \sqcup P_{XDDC})$ supports 1GB pages, 8 colors.

Core Scheduling. We now define two contracts that allocate cores at a sub-chiplet granularity (P_{Core}). We partition both the L3 cache and the cross-chiplet directory, but not DRAM. In doing so, we independently use cache allocation technology (CAT) to partition the L3 cache and use coloring to partition the cross-chiplet directory. This allows for allocating memory sizes and cache sizes independently.

For CAT, we define numerous CoSes that partition the L3 cache ways in four-way clusters: (i) 4 CoS, each of them receives a quarter of the ways; (ii) 4 CoS, each of them receives a half of the ways; (iii) 6 CoS, each of them receives three quarters of the ways; and (iv) 1 CoS that includes all ways. A CoS is assigned to a trust domain based on its number of cores. For instance, a single-core trust domain will be assigned a CoS from the first class. Concurrent trust domains need to be assigned CoSes that contain disjoint ways.

For coloring, we omit the partition P_{XDDC} as it does not provide sufficient colors, and use the partition P_{XDC} for two contracts:

(1) $P_{Core} \uplus (P_{4K} \sqcup P_{XDC})$ supports 4KB pages, 512 colors.
(2) $P_{Core} \uplus (P_{2M} \sqcup P_{XDC})$ supports 2MB pages, 16 colors.

Discussion. Finally, we present two hypothetical contracts that would prevent DRAM side-channels with sufficient colors. The CPU designer could use XD's a_{22}, a_{23} to index a quadrant of banks. (See XDDC′ in Table 3). These patterns resemble those of Intel [42], but differ from those of AMD [24] which utilize page offset bits.

(1) $P_{Chiplet} \uplus (P_{2M} \sqcup P_{XDDC'})$ supports 2MB pages, 32 colors.
(2) $P_{Core} \uplus (P_{2M} \sqcup P_{XDDC'})$ supports 2MB pages, 32 colors.

5.4 Marghera Prototype

We now describe our implementation of Marghera in Microsoft Hyper-V. Our baseline hypervisor protects against cross-domain transient-execution and private-core microarchitectural attacks by employing core scheduling and virtual-processor address space isolation [2, 64]. That is, during a VM-to-VM switch, the hypervisor flushes L1-D and microarchitectural buffers; and during a VM intercept, the hypervisor can access only state that is global or private to the intercepted virtual processor.

Our extensions span various layers of the system software stack, including the VM worker process, virtualization infrastructure driver (VID), the Windows NTOS kernel, and the hypervisor. We have extended CpuGroups and the host compute system service (HCS) to provide an interface for managing resource-exclusive groups.

Our prototype implements the design in §4.2 but not yet the extensions in §4.3, meaning that the root VM needs to be trusted.

5.4.1 Memory Management in Root VM. Each VM is managed by a VM worker process, and each VM is backed by a VID partition. During VM creation, the VM worker process interacts with VID to instantiate multiple memory blocks, which back the VM's RAM space. For each memory block, VID utilizes the NTOS memory manager page allocation interface to reserve memory.

The baseline hypervisor maps the entire host memory to the root VM's address space; the entire memory is part of the root memory compartment, from which physical memory is allocated to the VMs. We have implemented color assignment, colored page allocation, and color revocation as follows.

For color assignment, we extended the NTOS memory manager with a function that assigns colors (as described in §4.2.2) by taking memory size requirement as an input and returning the color bitmap. The memory manager enforces the exclusive color assignment invariant by tracking assigned colors in a *reserved color bitmap* within the metadata of the root memory compartment.

The color assignment function is exposed to VID. The *assigned color bitmap* is stored in the VID partition metadata, so it can be used by VID during memory reservation for memory blocks. The function is exposed (via VID) to the VM worker process, and called by the latter before memory reservation is initiated.

For page allocation, we extended the NTOS memory manager with a new function that allocates memory pages by taking the color bitmap as an input. As described in §4.2.2, the memory manager uses one free list for each page size, and each free list is divided across the colors supported by the page size. The required coloring functions are implemented using bitwise operations on the physical page frame number. Finally, during memory reservation for a memory block, VID uses the assigned color bitmap to parametrize its page allocation requests to the NTOS memory manager.

For color revocation, we extended the NTOS memory manager with a function that takes as input a color bitmap and unsets the reserved color bitmap. This function is called by VID once the partition's memory blocks have been destroyed and their backing physical memory has been free'd.

5.4.2 Hypervisor Scheduler. Each VM is backed by a process with as many software threads as the number of virtual CPUs. CPU groups are backed by scheduler groups, which are managed by a scheduler group manager. Processes get bound to (unbound from) a scheduler group via the managers's process-group (un)binding interface during process initialization (destruction). Physical cores are backed by core schedulers while physical threads are backed by logical processor dispatchers. In core scheduling, a process' threads are organized in pairs, and each pair is scheduled on a core.

Our extensions span all these components. The scheduler group and core scheduler managers jointly enforce the resource exclusivity invariant for processes that may or may not be bound to a CPU group. The logical processor dispatcher restricts the L3 cache ways into which a physical thread can allocate cache lines.

We extended the core scheduler manager to maintain the set of physical cores and L3 cache ways available to non-bound processes. This availability metadata are updated via new functions for resource reservation and resource revocation.

Reservation and revocation of L3 cache ways is straightforward inasmuch as the scheduler manager directly unsets and sets the L3 cache ways in the availability metadata for each L3 domain. Reservation and revocation of physical cores is more complicated inasmuch as it entails flushing of resources.

During reservation of a physical core, and once the core is unset in the availability metadata, the core scheduler manager drains the newly reserved physical core. The draining is implemented as a non-blocking operation, which takes place in the next scheduling interval. Once a physical core is drained, it is marked as drained, and all of the drained threads are migrated to other physical cores based on the new set of available physical cores. Finally, when a physical core has been marked as drained, the core scheduler executes `wbinvd` to invalidate all caches associated with the core,[3] implicitly invalidating all corresponding directory entries.

During revocation of a physical core, the core scheduler manager executes `wbinvd` on the core,[4] and then sets the core as available in the availability metadata.

We extended the scheduler group manager to maintain the set of physical cores and L3 cache ways that have been assigned to alive resource-exclusive scheduler groups and referenced by alive conventional scheduler groups. We extended the existing process-group (un)binding interface to allow for (un)binding of processes while enforcing the resource exclusivity invariant. During (un)binding of a process to (from) a resource-exclusive group, the scheduler group manager calls the core scheduler manager to reserve (revoke) assigned resources. Successful (un)bindings (un)set the process' resource-exclusive flag, stored in the process' metadata.

For processes that are bound to groups, we rely on existing logic in the core scheduler manager that uses the group's ThreadAffinity to restrict the set of physical cores on which the process's threads can be scheduled. For processes that are not bound to groups, we extended the core scheduler manager to use the physical cores set in the availability metadata as the restriction set.

We extended the logical processor dispatcher to restrict the L3 cache ways into which a physical thread can allocate cache lines. The dispatcher updates the physical thread's model-specific registers that correspond to the assignment of a Class-of-Service and the mask of the assigned Class-of-Service. For processes bound to

[3] In our target CPU, executing `wbinvd` on one physical core invalidates its L1/L2 caches and L3 cache, but does not guarantee invalidation of L1/L2 caches of other cores.
[4] We do not yet invalidate TLBs, which may result in some leakage to future VMs.

a group, the mask includes the L3 cache ways within the group's L3CacheWayAffinity. For processes not bound to a group, the mask includes the L3 cache ways set in the availability metadata.

Finally, we extended the hypervisor's interface used for managing groups (called by HCS during a CpuGroups operation) to reject requests that would unbind a resource-exclusive process.

6 EVALUATION

We analyze the effectiveness and the performance overheads of Marghera defenses against microarchitectural leakage. Our testbed hosts an AMD EPYC 7543P processor and 512 GB of DRAM. The processor consists of eight chiplets, and each chiplet houses four physical cores and 32 MB of L3 cache. The first chiplet runs the root VM, while the rest are reserved for guest VMs.

We evaluate compute partitions using chiplet ($P_{Chiplet}$) and core scheduling (P_{Core}), and memory partitions using the contracts $P_{4K} \sqcup P_{XDC}$ and $P_{2M} \sqcup P_{XDC}$. In chiplet scheduling, we also evaluate a memory partition using the contract $P_{1G} \sqcup P_{XDDC}$. In core scheduling, we size the L3 cache partition of each VM based on the number of virtual CPUs, i.e., two ways per virtual CPU.

6.1 Validation of Marghera's Defenses

To demonstrate the effectiveness of Marghera's microarchitectural isolation, we measure cross-VM interference by running a collection of Observer and Target micro-benchmarks.

As detailed below, the Target emits a signal by thrashing its private memory, while the Observer measures timings to access its own private memory. Figures 5–7 visualize the time series of Observer's measurements as heatmaps, where each vertical line represents a measurement. The darkest color of the spectrum corresponds to the smallest measurement (*min*) and the brightest color corresponds to a measurement equal to $N * min$, where N is indicated by the scale on the right. For each benchmark, we use the same scale with and without defenses. We also use performance counters to measure the Observer's DRAM access rate, as the number of requests served by DRAM divided by the number of instructions.

While our microbenchmarks fall short of an end-to-end security evaluation against VMs running applications, the Observers are representative of state-of-the-art probes used in side-channel attacks, such as workload fingerprinting [48], whereas the Targets are designed to emit the strongest signals.

Our implementation does not prevent indirect leakage via shared colors between VM and system software (root VM and hypervisor), which explains the small residual leakage visible in Figures 5, 6.

Intra-chiplet. Figure 5 plots the heatmap when the Target thrashes the L3 cache using six threads to access a large array, while the Observer primes the L3 cache and measures the time to access all its cache lines. It confirms that cross-VM information leakage is eliminated either by scheduling them on separate chiplets or on the same chiplet but with disjoint compute and cache partitions.

We also measure the Observer's DRAM access rate. In the baseline case, it increases from 0.032 to 110 per thousand instructions across the black and orange regions. In the other cases, the increase is within our (95%) confidence interval.

Cross-chiplet. Figure 6 plots the heatmap when a Target thrashes the cross-chiplet directory using threads on six chiplets to access a large array, while the Observer probes its L1-D on a separate chiplet. It confirms that cross-chiplet information leakage is eliminated by assigning disjoint P_{XDC} colors (0 to Observer, another to Target).

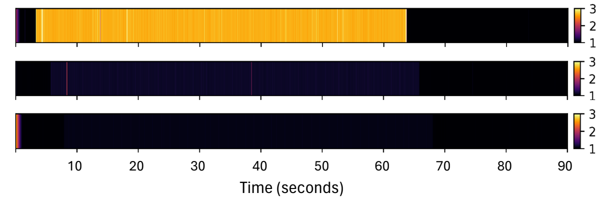


Figure 5: L3 cache information leakage between an Observer and a Target that thrashes the L3 cache. The Observer, Target run in different VMs on distinct cores in three configurations: within the same chiplet (top); within the same chiplet with L3 partitioning (middle); on separate chiplets (bottom).

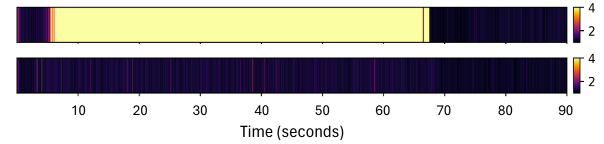


Figure 6: Cross-chiplet information leakage between an Observer and a Target that thrashes the cross-chiplet directory. The Observer and Target run in different VMs on separate chiplets, using separate colors (bottom) or not (top).

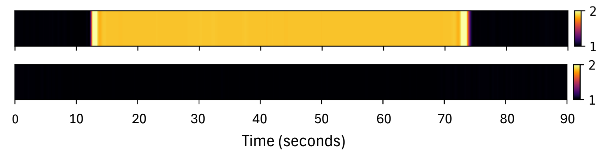


Figure 7: DRAM information leakage between an Observer and a Target that thrashes memory bus bandwidth. The Observer and Target run in different VMs on separate chiplets, using separate colors based on XDC (top) or XDDC (bottom).

We also measure the Observer's DRAM access rate. In the baseline case, it increases from 0.45 to 60 per million instructions. With Marghera, the DRAM access rate remains nearly constant.

DRAM. Figure 7 plots the heatmap when a Target thrashes its L3 caches using threads on six chiplets to access a large array, resulting in high memory bus contention and higher memory latencies. The Observer (on a separate chiplet) traverses randomly (at a cache line granularity) a large array that does not fit in the L3 cache, and hence each array access results in a DRAM access. It confirms that DRAM-level information leakage is eliminated by assigning disjoint channels (P_{XDDC} colors) to the Target and Observer.

6.2 Performance Analysis

Marghera introduces three sources of overheads: microarchitectural flushing, additional page table walks, and resource partitioning. As Marghera flushes microarchitectural state only during VM

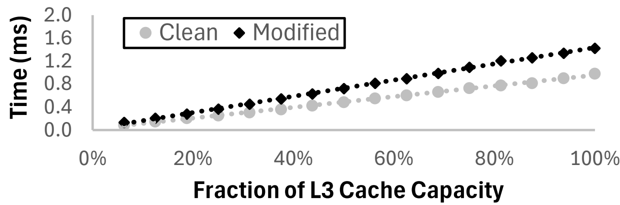


Figure 8: Time spent on flushing the cache hierarchy after accessing a variable fraction of the L3 cache.

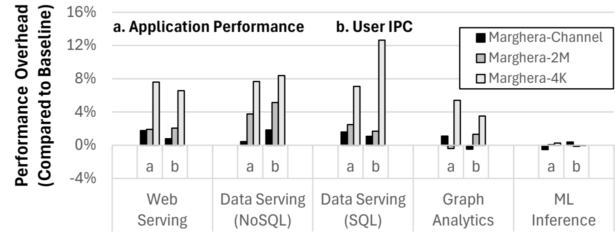


Figure 9: Impact of page coloring in Chiplet Scheduling.

creation/destruction, performance of workloads running in stable state is impacted only by the latter two. We use a mix of micro-benchmarks and cloud benchmarks to measure these overheads.

Cloud Benchmarks. We consider an ML Inference workload and cloud workloads [13] including: Web Serving, Data Serving (NoSQL), Data Serving (SQL), and Graph Analytics. Unless stated otherwise, the benchmarks are configured as follows.

For ML Inference, we infer the Inception-v3 TensorFlow model on the Nvidia Triton inference server and measure its throughput with a 95% response latency below 300ms. For Web Serving, an Nginx server runs the Elgg social networking engine and connects to a chiplet-sized memcached server and a two-chiplet-sized database server that is populated with 100K users, which yields a 2.5GB database. We measure the throughput of a mix of operations while maintaining quality of service. For Data Serving (NoSQL), we populate the Cassandra NoSQL data store with 10M records (10GB) and measure the throughput of a workload of 50% read and 50% update operations with a 95% response latency below 10ms. For Data Serving (SQL), we populate the Postgress database with 100 warehouses (10 GB) and measure the throughput of the TPC-C workload with a 95% response latency below 300ms. For Graph Analytics, we run the PageRank algorithm on Spark GraphX on a 1.2GB Twitter dataset and measure its execution time.

All workloads run for for three minutes except for Graph Analytics, which runs to completion. We repeat each experiment five times; resulting 95% confidence intervals are within 1% of the mean.

Metrics of Interest. Besides application performance metrics, we use microarchitectural metrics to understand the performance overheads. We use performance counters to measure DRAM access rates, TLB hit rates, and User IPC. User IPC is measured as the number of instructions committed by the application divided by the number of total cycles (include both application and OS); it has been shown to be proportional to application performance [13].

6.2.1 Overhead due to Flushing. First, we examine the performance implications of flushing the cache hierarchy.

Latency. We quantify the time spent on completing the flushing operation. Figure 8 plots the time spent on executing `wbinvd` for two micro-benchmarks. The first micro-benchmark (*Clean*) loads a varying fraction of the L3 cache. The second micro-benchmark (*Modified*) writes to a varying fraction of the L3 cache.

Flushing *Clean* incurs lower latency than flushing *Modified* because flushing *Clean* involves only invalidating tags in caches and directories whereas flushing *Modified* additionally involves writing back modified cache lines to memory. Overall, the time is linear to the number of cache lines, ranging from 0.085 to 1.425ms. This latency is small even when compared to fast VM boot times, e.g., 125ms for Firecracker microVMs, with no performance impact in scenarios where resources are not time-sliced.

Interference. The `wbinvd` instruction invalidates the entire L3 cache, potentially degrading performance of other VMs running on the same chiplet. Given the large VM lifetimes (at least minutes [9]), we expect creation and destruction of VMs to happen at the granularity of at least a few minutes. We over-estimate the performance impact on a VM by creating and destroying another VM on the same chiplet every five seconds. Each VM is assigned two physical cores, eight L3 cache ways, and distinct memory colors.

We measure the VM's User IPC and DRAM access rate for a period during which we create and destroy a VM every five seconds compared to a period during which the VM runs without any interference. For Graph Analytics, the period spans the whole execution, while for the remaining workloads we use a three-minute period For all cloud workloads, we find that the DRAM access rates are impacted marginally (up to 5% higher). The impact on User IPC is small, around 1% in the worst case of Data Serving (NoSQL). This is because the benchmarks quickly warm up their caches and enjoy their temporal locality until the next flushing event.

6.2.2 Overhead due to Partitioning. We now analyze the performance impact of page coloring and L3 cache partitioning.

Page Coloring. We analyze three configurations that implement the chiplet scheduling contract with (i) channel allocation, $P_{Chiplet} \uplus (P_{1G} \sqcup P_{XDDC})$, labelled as Marghera-Channel; (i) 2MB pages, $P_{Chiplet} \uplus (P_{2M} \sqcup P_{XDC})$, labelled as Marghera-2M; and (iii) 4KB pages, $P_{Chiplet} \uplus (P_{4K} \sqcup P_{XDC})$, labelled as Marghera-4K. Each workload runs within a chiplet-sized VM with 8 GB of memory. (This is the maximum size we could allocate with 4KB pages.)

Figure 9 plots the overhead of Marghera over the baseline system. Marghera-Channel and Marghera-2M introduce small overheads, up to 1.8% and 3.8%, respectively. Marghera-4K reduces performance by up to 7.7% due to increased TLB pressure.

We confirm the sources of overhead using performance counters to examine the impact of Marghera's defenses on DRAM access rates and TLB miss rates. Marghera-Channel exhibits similar metrics as the baseline. For Marghera-2M and Marghera-4K, the DRAM access rates are affected the most in Data Serving SQL (15%), Data Serving NoSQL (16%), Web Serving (35%) and the least in ML Inference (6%), and Graph Analytics (<1%). While Marghera-2M exhibits similar TLB hit rates as the baseline system, Marghera-4K is impacted by the use of 4KB pages. Its TLB hit rates are affected the

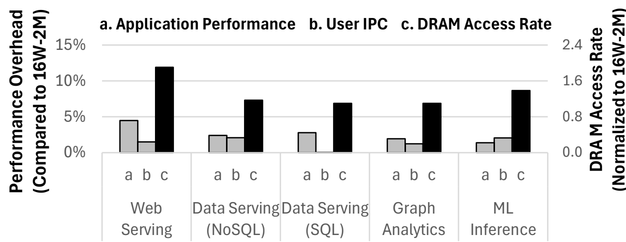


Figure 10: Impact of L3 cache partitioning in Core Scheduling.

most in the Data Serving and Web Serving workloads (highest TLB pressure, > $2x$ higher instruction L2 TLB miss rates) and the least in ML Inference and Graph Analytics (lowest TLB pressure).

The microarchitectural impact is reflected into User IPC. In Data Serving (SQL) workload, we observe that Marghera-4K exhibits higher User IPC overheads than application-level overheads (12.7% vs 7.1%). A possible explanation is that application-level performance exhibited high variance while User IPC was quite stable.

L3 Cache Partitioning. We analyze two configurations that implement the core scheduling $P_{Core} \uplus (P_{2M} \sqcup P_{XDC})$ with 8 and 16 L3 cache ways, labelled as Marghera-8W-2M and Marghera-16W-2M. Each workload runs within a two-core VM with 16 GB of DRAM; both Data Serving workloads use 15 GB databases.

Figure 10 presents the performance and DRAM access rates of Marghera-8W-2M compared to Marghera-16W-2M. While ML Inference and Web Serving encounter the largest increase in DRAM access rate, the performance loss is small as their L3 cache hit ratios are still high (75–80%). Overall, we find that the performance impact of cache partitioning is small (1.4–4.3%) corroborating the small sensitivity of cloud benchmarks to L3 cache capacity [13].

7 DISCUSSION

Adoption Challenges. In this paper, we inferred an isolation contract for a single CPU by reverse-engineering, which does not scale to the diverse hardware fleet of cloud providers. In the future, we expect CPU vendors to develop isolation contracts themselves and expose them via their hardware interface for consumption by the hypervisor's hardware abstraction layer (HAL).

The isolation contracts developed by different CPU vendors must provide sufficient information so they can be transformed (by the hypervisor's HAL) into a universal configuration that parametrizes generic resource management.

L3 Cache Partitioning. Our findings showed that cache allocation technology is sufficient to provide cache-level isolation for private memory lines. In our target CPU, the isolation contract employs hardware partitioning to divide each of the eight L3 caches (each one is shared across the chiplet's cores) into 16 classes, totalling in 128 classes—enough to run as many VMs as the number of cores.

CPUs may employ a shared L3 cache across a number of cores that is larger than the L3 cache associativity. In such cases, isolation contracts must provide a coloring scheme that partitions the L3 cache jointly with other shared resources. For instance, on Intel CPUs, where the directory and L3 cache are indexed identically [65], coloring the directory would also color the L3 cache.

Resource Underutilization. Page coloring may lead to memory stranding for arbitrary VM memory sizes. Cloud VMs are not arbitrary sized; their size is based on the number of virtual CPUs and the ratio of host memory to the total number of physical threads, e.g., Amazon EC2 and Microsoft Azure VMs use 2 GB, 4 GB, or 8 GB per virtual CPU. The cloud provider could minimize memory stranding by sizing VM memory in multiples of a color's memory span, e.g., 1 GB on our evaluated system (512 GB, 512 colors).

Another reason page coloring may lead to memory underutilization is that platforms—that isolate XVMs from the root VM—must designate one shared P_{4K} color for allocating shared pages between root VM and guest VMs. On our evaluated system (with 512 colors), this accounts for 0.2% of the system's memory.

Finally, microarchitectural resources assigned to a VM may exceed its working sets, reducing the resource's effective capacity. Our evaluation showed that the associated performance loss is small.

Resource Oversubscription. Our design allows for resource oversubscription only for conventional VMs inasmuch as non-resource-exclusive groups can share resources between them and/or have multiple live VMs bound to them. Flushing their L3 cache and directory sets is not needed, minimizing context switching overheads.

Oversubscription of XVM resources could be enabled by allowing multiple XVMs to share a resource-exclusive group while enforcing the invariant that only one XVM runs at a time. The hypervisor would need to support (i) an atomic XVM context switch, during which it flushes the group's microarchitectural resources; and (ii) time-slicing of the group at a granularity coarser than that of typical scheduling intervals to amortize the cost of flushing.

Minimizing Leakage via Hypervisor. The platform could minimize cross-VM leakage via the hypervisor context by restricting the colors for the hypervisor context: (i) the bootloader would need to map the hypervisor binary to the reserved colors; and (ii) the hypervisor memory is allocated only from these colors while dedicating one color for allocating memory intended for sharing with guest VMs, e.g., for ring buffers for hypercalls. The hypervisor could allocate second-stage page tables within the VM's assigned colors.

Leakage. While we leave an analysis of application-level leakage due to *intended* sharing of memory for future work, we expect,

(1) via its use of shared colors, an XVM may leak to other VMs when it performs I/O and hypercalls and some information as it performs I/O with other VMs. Such attacks may be mitigated by enlightening XVM drivers to obliviously access I/O buffers.

(2) by observing the shared hypervisor context, an attacker VM could detect execution of the intercept handler on other cores. This may leak the frequency and type of XVM intercepts, but not VM data as long as the hypervisor is secret-free [64].

8 RELATED WORK

Here we focus on work that was not discussed in the paper body.

Spatial partitioning has been employed for various purposes (besides security isolation of trust domains [15, 47]) including: performance isolation for caches [46] using hardware support or coloring [4, 28, 62], DRAM banks [68], or both [52]; and RowHammer mitigation by coloring DRAM subarrays [36]. Our work introduces isolation contracts, which provide multi-resource partitioning of

shared resources without partitioning private resources. Unlike prior work which proposes platform-specific hierarchical (L3 and DRAM bank) coloring [52], we capture all security/performance constraints in one coloring scheme, which can then be enforced in a platform-agnostic way by the memory manager.

More recently, Quarantine provides an approach to mitigate transient execution attacks by eliminating sharing of microarchitectural resources [20]. Their goals are similar to those of the secret-free hypervisor architecture [64], already implemented by our baseline hypervisor [2]. Quarantine dedicates a set of physical cores (to the hypervisor) for handling hypervisor calls and VM intercepts received by a privilege stub running on each of the remaining cores. For resources shared across physical cores, Quarantine partitions only the L3 cache; thus, it could be extended to leverage isolation contracts to provide holistic microarchitectural domain isolation.

Avoiding spatial partitioning requires hardware support to ensure that microarchitectural components do not leak information.

• Prior work proposed locking L3 cache lines (and pages) to mitigate cache-based side-channels [30]. Such approach is limited to one resource, and also requires changes to guest VMs and their applications to use locked pages for their sensitive data.

• Prior work proposed hardware-level mitigations to tackle cache- and directory-based side-channels, including secure sparse directories [66], secure shared caches that employ randomized cache replacement policies [61], or encrypted cache indexing [43, 44] to break the link between evicted cache lines and accessed memory addresses. Our work can leverage such resources by treating them as private, i.e., they do not need to be partitioned.

Besides attacks included in our threat model, there are additional side-channel attack vectors due to sharing the CPU's IOMMU and I/O devices (e.g., NICs and GPUs) across trust domains [10, 11, 29]. We leave secure resource sharing of devices for future work.

Leakage models have been used for capturing and detecting (classic and speculative) side-channels in applications [7, 18]. Formalizing and reasoning about side-channels at the system level is still a more open problem [19]; Sison et al. proposed a formalization of "time protection" for operating systems [49]; it is an avenue for future work to build similar models on top of isolation contracts.

9 CONCLUSION

We presented Marghera, a system design that prevents cross-VM microarchitectural side-channel attacks by partitioning resources based on isolation contracts. We showed how these contracts can be developed to form a basis for developing hypervisor-enforced allocation strategies for physical threads and memory that eliminate cross-VM information leakage. We hope that our results will inspire CPU vendors to develop such contracts, enabling cloud providers to provide comprehensive VM microarchitectural isolation.

An interesting avenue for future work would be to provide side-channel protection in confidential computing platforms, where the host OS and hypervisor are untrusted. Software-based confidential computing platforms [6, 14] are ideal for providing microarchitectural isolation. In such platforms, host-driven resource management is decoupled from isolation enforcement, which is implemented in a security monitor. The latter could be extended to microarchitecturally isolate VMs from untrusted hosts and VMs.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable comments and suggestions, and colleagues at Microsoft, especially Adrien Ghosn and Kapil Vaswani, for their insightful discussions. Istvan Haller provided feedback during the early stages of this work.

REFERENCES

[1] AMD. 2024. Microarchitectural cache side-channel attacks. https://www.amd.com/en/resources/product-security/bulletin/amd-sb-7025.html.
[2] Azure. 2019. Hyper-V HyperClear. https://techcommunity.microsoft.com/t5/virtualization/hyper-v-hyperclear-mitigation-for-l1-terminal-fault/ba-p/382429.
[3] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software Grand Exposure: SGX cache attacks are practical. In *USENIX Conference on Offensive Technologies*.
[4] Edouard Bugnion, Jennifer M. Anderson, Todd C. Mowry, Mendel Rosenblum, and Monica S. Lam. 1996. Compiler-directed page coloring for multiprocessors. *ACM SIGPLAN Notices* 31, 9 (Sept. 1996), 244–255.
[5] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *USENIX Security Symposium*.
[6] Charly Cases, Adrien Ghosn, Neelu S. Kalani, Yuchen Qian, Marios Kogias, Mathias Payer, and Edouard Bugnion. 2023. Creating trust by abolishing hierarchies. In *Workshop on Hot Topics in Operating Systems*.
[7] Sunjay Cauligi, Craig Disselkoen, Daniel Moghimi, Gilles Barthe, and Deian Stefan. 2022. SoK: Practical Foundations for Software Spectre Defenses. In *IEEE Symposium on Security and Privacy*.
[8] Pat Conway, Nathan Kalyanasundharam, Gregg Donley, Kevin Lepak, and Bill Hughes. 2010. Cache hierarchy and memory subsystem of the AMD Opteron processor. *IEEE Micro* 30, 2 (March-April 2010), 16–29.
[9] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource Central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *ACM SIGOPS Symposium on Operating Systems Principles*.
[10] Sankha Baran Dutta, Hoda Naghibijouybari, Nael Abu-Ghazaleh, Andres Marquez, and Kevin Barker. 2021. Leaky Buddies: Cross-component covert channels on integrated CPU-GPU systems. In *Annual International Symposium on Computer Architecture*.
[11] Sankha Baran Dutta, Hoda Naghibijouybari, Arjun Gupta, Nael Abu-Ghazaleh, Andres Marquez, and Kevin Barker. 2023. Spy in the GPU-Box: Covert and side channel attacks on multi-GPU systems. In *Annual International Symposium on Computer Architecture*.
[12] Mark Evers, Leslie Barnes, and Mike Clark. 2022. The AMD next-generation "Zen 3" core. *IEEE Micro* 42, 3 (May-June 2022), 7–12.
[13] Michael Ferdman, Adileh Almutaz, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, and Anastassia Ailamaki. 2012. Clearing the Clouds: A study of emerging scale-out workloads on modern hardware. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.
[14] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. 2017. Komodo: Using verification to disentangle secure-enclave hardware from software. In *ACM SIGOPS Symposium on Operating Systems Principles*.
[15] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. 2019. Time protection: The missing OS abstraction. In *European Conference on Computer Systems*.
[16] Michael Godfrey and Mohammad Zulkernine. 2014. Preventing cache-based side-channel attacks in a cloud environment. *IEEE Transactions on Cloud Computing* 2, 4 (Oct.-Dec. 2014), 395–408.
[17] Johannes Gotzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Mller. 2017. Cache attacks on Intel SGX. In *European Workshop on Systems Security*.
[18] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. 2021. Hardware-software contracts for secure speculation. In *IEEE Symposium on Security and Privacy*.
[19] Gernot Heiser, Gerwin Klein, and Toby Murray. 2019. Can we prove time protection?. In *Workshop on Hot Topics in Operating Systems*.
[20] Mathé Hertogh, Manuel Wiesinger, Sebastian Österlund, Marius Muench, Nadav Amit, Herbert Bos, and Cristiano Giuffrida. 2023. Quarantine: Mitigating transient execution attacks with physical domain isolation. In *International Symposium on Research in Attacks, Intrusions, and Defenses*.
[21] Jana Hofmann, Cédric Fournet, Boris Köpf, and Stavros Volos. 2024. Gaussian Elimination of Side-channels: Linear algebra for memory coloring. In *ACM Conference on Computer and Communications Security*.
[22] Mehmet Sinan Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2016. Cache attacks enable bulk key recovery on the cloud. In *International Conference on Cryptographic Hardware and Embedded Systems*.

[23] G. Irazoqui, T. Eisenbarth, and B. Sunar. 2015. S$A: A shared cache attack that works across cores and defies VM sandboxing – and its application to AES. In *IEEE Symposium on Security and Privacy*.
[24] Patrick Jattke, Max Wipfli, Flavien Solt, Michele Marazzi, Matej Bolcskei, and Kaveh Razavi. 2024. ZenHammer: Rowhammer attacks on AMD Zen-based platforms. In *USENIX Security Symposium*.
[25] Matthew A. Johnson, Stavros Volos, Ken Gordon, Sean T. Allen, Christoph M. Wintersteiger, Sylvan Clebsch, John Starks, and Manuel Costa. 2024. Confidential Container Groups. *ACM Queue* 22, 2 (March-April 2024), 57–86.
[26] Vydhyanathan Kalyanasundharam, Kevin M. Lepak, Amit P. Apte, Ganesh Balakrishnan, Eric C. Morton, Elizabeth M. Cooper, and Ravindra N. Bhargava. 2021. Region based directory scheme to adapt to large cache sizes. Patent No. US11119926B2, Filed Dec. 18th, 2017, Issued Sep. 14th, 2021.
[27] David Kaplan. 2023. Hardware VM isolation in the cloud. *Commun. ACM* 67, 1 (Dec. 2023), 54–59.
[28] Hyoseung Kim and Ragunathan Rajkumar. 2016. Real-time cache management for multi-core virtualization. In *International Conference on Embedded Software*.
[29] T. Kim, H. Park, S. Lee, S. Shin, J. Hur, and Y. Shin. 2023. DEVIOUS: Device-driven side-channel attacks on the IOMMU. In *IEEE Symposium on Security and Privacy*.
[30] Taesoo Kim, Marcus Peinado, and Glria Mainar-Ruiz. 2012. STEALTHMEM: System-level protection against cache-based side-channel attacks in the cloud. In *USENIX Security Symposium*.
[31] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting speculative execution. In *IEEE Symposium on Security and Privacy*.
[32] Zili Kou, Sharad Sinha, Wenjian He, and Wei Zhang. 2022. Attack directories on ARM big.LITTLE processors. In *International Conference on Computer-Aided Design*.
[33] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. 2021. CIPHERLEAKS: Breaking constant-time cryptography on AMD SEV via the ciphertext side channel. In *USENIX Security Symposium*.
[34] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading kernel memory from user space. In *USENIX Security Symposium*.
[35] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy*.
[36] Kevin Loughlin, Jonah Rosenblum, Stefan Saroiu, Alec Wolman, Dimitrios Skarlatos, and Baris Kasikci. 2023. Siloz: Leveraging DRAM isolation domains to prevent inter-VM Rowhammer. In *ACM SIGOPS Symposium on Operating Systems Principles*.
[37] Kevin Loughlin, Stefan Saroiu, Alec Wolman, Yatin A. Manerkar, and Baris Kasikci. 2022. MOESI-Prime: Preventing coherence-induced hammering in commodity workloads. In *Annual International Symposium on Computer Architecture*.
[38] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. 2015. Reverse engineering Intel last-level cache complex addressing using performance counters. In *International Symposium on Research in Attacks, Intrusions, and Defenses*.
[39] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. 2017. Hello from the Other Side: SSH over robust cache covert channels in the cloud. In *Symposium on Network and Distributed System Security*.
[40] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. 2018. Varys: Protecting SGX eEnclaves from practical side-channel attacks. In *USENIX Annual Technical Conference*.
[41] Riccardo Paccagnella, Licheng Luo, and Christopher W. Fletcher. 2021. Lord of the Ring(s): Side channel attacks on the CPU on-chip ring interconnect are practical. In *USENIX Security Symposium*.
[42] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. 2016. DRAMA: Exploiting DRAM addressing for cross-CPU attacks. In *USENIX Security Symposium*.
[43] Moinuddin K. Qureshi. 2018. CEASER: Mitigating conflict-based cache attacks via encrypted-address and remapping. In *IEEE/ACM International Symposium on Microarchitecture*.
[44] Moinuddin K. Qureshi. 2019. New attacks and defense for encrypted-address cache. In *Annual International Symposium on Computer Architecture*.
[45] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-privilege-boundary data sampling. In *ACM Conference on Computer and Communications Security*.
[46] Mohammad Shahrad, Sameh Elnikety, and Ricardo Bianchini. 2021. Provisioning differentiated last-level cache allocations to VMs in public clouds. In *ACM Symposium on Cloud Computing*.
[47] Jicheng Shi, Xiang Song, Haibo Chen, and Binyu Zang. 2011. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *International Conference on Dependable Systems and Networks Workshops*.
[48] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. 2019. Robust website fingerprinting through the cache occupancy channel. In *USENIX Security Symposium*.
[49] Robert Sison Sison, Scott Buckley, Toby Murray, Gerwin Klein, and Gernot Heiser. 2023. Formalising the prevention of microarchitectural timing channels by operating systems. In *International Symposium on Formal Methods*.
[50] Sriram Srinivasan and William L. Walker. 2018. Shadow tag memory to monitor state of cachelines at different cache level. Patent No. US10073776B2, Filed June 23rd, 2016, Issued Sep. 11th, 2018.
[51] David Suggs, Mahehs Subramony, and Dan Bouvier. 2020. The AMD "Zen 2" processor. *IEEE Micro* 40, 2 (March-April 2020), 45–52.
[52] Noriaki Suzuki, Hyoseung Kim, Dionisio De Niz, Bjorn Andersson, Lutz Wrage, Mark Klein, and Ragunathan Rajkumar. 2013. Coordinated bank and cache coloring for temporal protection of memory accesses. In *IEEE International Conference on Computational Science and Engineering*.
[53] Simon M. Tam, Harry Muljono, Min Huang, Sitaraman Iyer, Kalapi Royneogi, Nagmohan Satti, Rizwan Qureshi, Wei Chen, Tom Wang, Hubert Hsieh, Sujal Vora, and Eddie Wang. 2018. SkyLake-SP: A 14nm 28-Core Xeon processor. In *IEEE International Solid-State Circuits Conference*.
[54] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi. 2003. Cryptanalysis of DES implemented on computers with cache. In *Annual Conference on Cryptographic Hardware and Embedded Systems*.
[55] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, Frank Piessens, and Ku Leuven. 2020. LVI: Hijacking transient execution through microarchitectural load value injection. In *IEEE Symposium on Security and Privacy*.
[56] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. Telling Your Secrets Without Page Faults: Stealthy page table-based attacks on enclaved execution. In *USENIX Security Symposium*.
[57] Pepe Vila, Boris Köpf, and José F. Morales. 2019. Theory and practice of finding eviction sets. In *IEEE Symposium on Security and Privacy*.
[58] Stavros Volos, Javier Picorel, Babak Falsafi, and Boris Grot. 2014. BuMP: Bulk memory prediction and streaming. In *IEEE/ACM International Symposium on Microarchitecture*.
[59] Junpeng Wan, Yanxiang Bi, Zhe Zhou, and Zhou Li. 2022. MeshUp: Stateless cache side-channel attack on CPU mesh. In *IEEE Symposium on Security and Privacy*.
[60] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. 2017. Leaky Cauldron on the Dark Land: Understanding memory side-channel hazards in SGX. In *ACM Conference on Computer and Communications Security*.
[61] Zhenghong Wang and Ruby B. Lee. 2007. New cache designs for thwarting software cache-based side channel attacks. In *International Symposium on Computer Architecture*.
[62] Bryan C. Ward, Jonathan L. Herman, Christopher J. Kenna, and James H. Anderson. 2013. Making shared caches more predictable on multicore platforms. In *Euromicro Conference on Real-Time Systems*.
[63] Zhenyu Wu, Zhang Xu, and Haining Wang. 2012. Whispers in the Hyper-space: High-bandwidth and reliable covert channel attacks inside the cloud. In *USENIX Security Symposium*.
[64] Hongyan Xia, David Zhang, Wei Liu, Istvan Haller, Bruce Sherwin, and David Chisnall. 2022. A Secret-Free Hypervisor: Rethinking isolation in the age of speculative vulnerabilities. In *IEEE Symposium on Security and Privacy*.
[65] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. 2019. Attack Directories, Not Caches: Side channel attacks in a non-inclusive world. In *IEEE Symposium on Security and Privacy*.
[66] Mengjia Yan, Jen-Yang Wen, Christopher W. Fletcher, and Josep Torrellas. 2019. SecDir: A secure directory to defeat directory side-channel attacks. In *Annual International Symposium on Computer Architecture*.
[67] Yuval Yarom and Katrina Falkner. 2014. Flush+Reload: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security Symposium*.
[68] Heechul Yun, Renato Mancuso, Zheng-Pei Wu, and Rodolfo Pellizzoni. 2014. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *IEEE Real-Time and Embedded Technology and Applications Symposium*.
[69] Zirui Neil Zhao, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. 2024. Everywhere All at Once: Co-location attacks on public cloud FaaS. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.
[70] Zirui Neil Zhao, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. 2024. Last-level cache side-channel attacks are feasible in the modern public cloud. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.
[71] Ziqiao Zhou, Yizhou Shan, Weidong Cui, Xinyang Ge, Marcus Peinado, and Andrew Baumann. 2023. Core Slicing: Closing the gap between leaky confidential VMs and bare-metal cloud. In *USENIX Operating Systems Design and Implementation*.