

Cache Coherency and NUMA Balancing Performance Pathologies in AMD SEV-SNP Confidential Virtual Machines: A Real-World Kernel-Level Case Study

Ashish Kalra

Independent Researcher, USA

Abstract

AMD Secure Nested Paging (SNP) provides hardware-enforced memory encryption for confidential virtual machines (VMs), but its security mechanisms can interact unexpectedly with the Linux kernel's memory management subsystems. This article presents a kernel-level case study of a performance regression in SNP guests caused by a security fix for cache incoherency in AMD Secure Encrypted Virtualization (SEV). The fix invoked system-wide cache flushes via `wbinvd_on_all_cpus()` through MMU invalidation notifiers, which triggered a pathological interaction with the NUMA autobalancing subsystem on large hosts. On a 500-CPU test system, SNP guests with 128 virtual CPUs and ≥ 10 GB of memory exhibited kernel soft lockups and Read-Copy-Update (RCU) stalls. These were traced to spurious MMU notifier events fired by the NUMA autobalancer on pinned, non-migratable guest memory pages. This article identifies the root cause through kernel stack trace forensics, evaluates upstream remediation alternatives, and examines the definitive fix: removing host virtual address mappings for private encrypted guest memory via the `guest_memfd` abstraction. The findings highlight the importance of co-designing security mitigations with performance-critical kernel subsystems.

Keywords: AMD SEV-SNP Cache Coherency, Confidential Virtual Machine Performance Regression, Linux KVM MMU Notifier Mechanism, NUMA Autobalancing Pinned Memory Pathology, `Guest_memfd` Confidential Computing Architecture

1. Introduction

Confidential computing has emerged as a foundational pillar of cloud security, enabling tenant workloads to execute within hardware-enforced Trusted Execution Environments (TEEs) that are isolated from the hypervisor, the host operating system, and co-located tenants [1]. AMD's SEV technology and its successive extensions, SEV-Encrypted State (SEV-ES) and SEV-SNP (Secure Nested Paging), are among the most widely deployed confidential virtual machine (VM) implementations in production cloud infrastructure [2]. SNP technology relies on hardware memory encryption and reverse mapping table (RMP) enforcement to provide strong isolation guarantees and integrity of guest memory [3]. The isolation guarantees provided by TEE-based architectures, however, come with non-trivial implications for the host kernel's memory management subsystems. Prior work has demonstrated that memory isolation mechanisms for hardware, when not carefully co-designed with kernel software stacks, can introduce both correctness hazards and performance pathologies.

The intersection of TEE-level memory isolation with the Linux kernel's MMU notifier framework [4] and NUMA-aware page migration logic introduces subtle hazards that are not immediately apparent from either subsystem's design in isolation. Prior work has examined performance overheads in TEE-based systems from related angles: throughput penalties from EPC paging in SGX enclaves have been documented [13], and memory initialization costs and access latency scaling in SEV-SNP guests under varying memory

configurations have been empirically characterized [14]. However, neither study addresses the specific interaction between host-side NUMA autobalancing and MMU notifier hot paths in SNP confidential VMs, the precise fault mode analyzed here. This case study therefore fills a gap in the literature on emergent, cross-subsystem performance pathologies in confidential computing deployments. This article presents a rigorous case study of one such hazard: a performance regression introduced by a security patch for SEV cache incoherency [5], which inadvertently caused severe guest-visible soft lockups and RCU stalls due to a pathological interaction with NUMA autobalancing on a 500-physical-CPU host system [6]. The analysis proceeds from vulnerability characterization through kernel stack trace forensics, upstream design deliberation, and ultimately to the architectural resolution facilitated by the *guest_memfd* kernel subsystem. The case study illustrates a broader principle: security mitigations in layered systems software must be co-designed with performance-critical subsystems to avoid emergent regressions that cannot be anticipated from component-level reasoning alone.

2. Background: Cache Incoherency in AMD SEV and the Security Patch

2.1 Cache Coherency Gap at the VM Boundary in AMD SEV

AMD SEV provides memory confidentiality by encrypting guest physical memory with a guest-unique key, such that the hypervisor cannot directly observe guest memory contents [2]. However, a fundamental architectural limitation exists at the VM boundary: cache coherency is not enforced across the host-guest boundary in base SEV and SEV-ES deployments [3]. Confidential cache lines generated by a guest executing with SEV protections enabled are tagged with the guest's C-bit and remain in the processor cache hierarchy in their encrypted form. Crucially, these cache lines are not automatically invalidated or written back to main memory when the guest releases the corresponding physical page. As a consequence, if a memory page containing dirty confidential cache lines is released by the guest VM and subsequently reallocated to another consumer, whether another VM, a kernel subsystem, or a user-space process, the stale confidential cache lines may be evicted to the backing physical memory at a later, unpredictable time, corrupting the data of the new owner [3]. This is a fundamental correctness hazard, not merely a performance concern.

The Linux KVM hypervisor had historically taken a simplifying assumption to sidestep this issue: all confidential memory for SEV and SEV-ES VMs was treated as implicitly pinned for the lifetime of the guest. Under this assumption, a released page would never be reallocated before VM termination, making the eviction of stale cache lines harmless. This assumption, however, is architecturally unsound for unpinned memory allocations. A malicious or buggy user-space VMM could exploit this gap by creating a VM that continuously allocates and releases unpinned confidential memory pages while the VM is actively running, thereby inducing cache line corruption in subsequently allocated kernel or user-space memory and crashing the host kernel [5].

2.2 The Security Patch and Its Unintended Consequence

To address this cache incoherency hazard, an upstream Linux kernel fix was introduced that hooks two MMU notifier events, `mmu_notifier_invalidate_range_start` and `mmu_notifier_release`, and performs an explicit cache flush when guest memory is reclaimed [5]. The MMU notifier framework in the Linux kernel provides a mechanism by which subsystems that maintain shadow page tables or other derived mappings can be notified when the kernel modifies the primary page tables [4]. KVM uses this framework to maintain consistency between guest and host page table states.

The remediation specifically invokes `wbinvd_on_all_cpus()` within the newly introduced `sev_guest_memory_reclaimed()` callback, which is reached via `kvm_arch_guest_memory_reclaimed()`

from `kvm_mmu_notifier_invalidate_range_start()` [5]. The `WBINVD` instruction writes back and invalidates the entire cache hierarchy on the issuing logical processor; when issued across all CPUs via Inter-Processor Interrupts (IPIs), it guarantees that no confidential cache lines remain in any processor's cache. The full dispatch chain from the MMU notifier event to the cache flush instruction is shown in Figure 1.

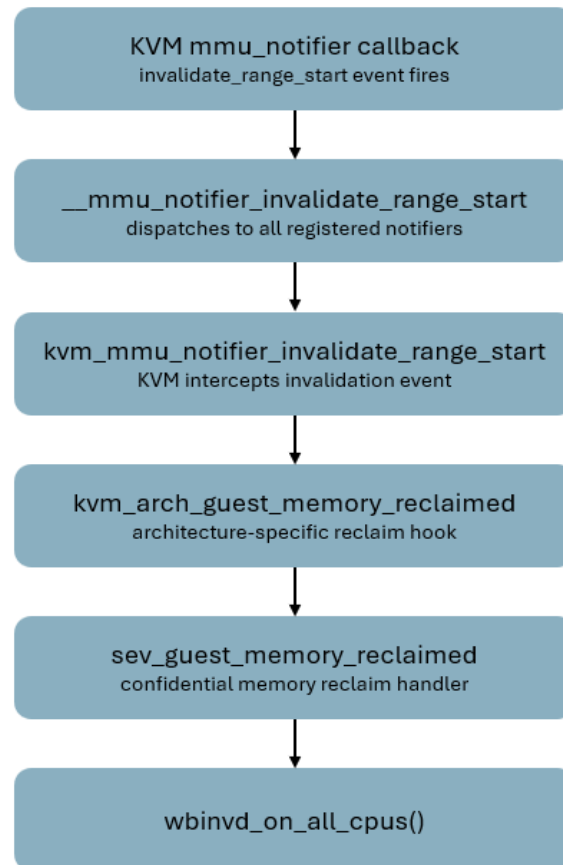


Fig. 1: MMU notifier dispatch chain terminating in full-system cache invalidation [4, 5]

While the proposed approach is a correct and conservative solution to the cache incoherency problem, its performance implications at scale were not adequately characterized prior to deployment. When applied to systems running SNP guests, this remedy triggered a severe regression: guests configured with 128 vCPUs and ≥ 10 GB of guest memory began exhibiting kernel soft lockups and RCU stalls. Soft lockup detectors in the Linux kernel fire when a CPU executes in kernel context beyond a configurable threshold without voluntarily yielding [9]. RCU stalls occur when grace periods do not complete within expected bounds. This typically happens when a CPU is spinning or blocked in a long-running kernel code path, delaying RCU quiescent states [10]. Both of these symptoms point towards a system that is experiencing severe latency pressure, with one or more CPUs taking unexpectedly long periods of time in the kernel, thus causing a delay in re-entry of the guest as well as a lack of execution time for the guest vCPUs.

3. Performance Analysis Methodology

The methodology followed for the analysis was kernel stack trace forensics, which utilizes the Linux kernel's Magic SysRq (System Request) facility, which provides several keys to dump call stack traces and process states directly to the system console or kernel log. This is particularly useful for debugging hung systems. The SNP guest appeared hung, with vCPU threads unresponsive, prompting the use of this facility to dump the call stacks of host VMM processes and determine why the guest was not progressing. Kernel stack traces were triggered by:

```
□ echo t > /proc/sysrq-trigger
```

□ This command causes the kernel to emit the call stack of every running and blocked thread to the kernel log. The analysis concentrated on the call stacks of host VMM processes, given that vCPU execution transpires within the framework of the user-space virtual machine monitor's host threads. The test environment comprised a host system with 500 physical CPUs running Linux kernel *5.19.0-rc5-next-20220706*, with SNP guests configured at 128 vCPUs and guest memory exceeding 10 GB [6]. This configuration represents a realistic high-density cloud deployment scenario.

4. Root Cause Analysis

4.1 Call Stack Evidence: NUMA Autobalancing Triggering MMU Notifiers

Analysis of the captured kernel stack traces revealed a critical and unexpected code path. Two representative vCPU thread stacks showed the following call sequences reaching *wbinvd_on_all_cpus()*:

Stack trace #1 (via hugepage migration fault):

```
□ sev_guest_memory_reclaimed()
  kvm_arch_guest_memory_reclaimed()
    kvm_mmu_notifier_invalidate_range_start()
      __mmu_notifier_invalidate_range_start()
        try_to_migrate_one()
          rmap_walk_anon()
            try_to_migrate()
              migrate_pages()
                migrate_misplaced_page()
                  do_huge_pmd_numa_page()
                    __handle_mm_fault()
                      handle_mm_fault()
```

□

Stack trace #2 (via NUMA protection change):

```
□ sev_guest_memory_reclaimed()
  kvm_arch_guest_memory_reclaimed()
    kvm_mmu_notifier_invalidate_range_start()
      __mmu_notifier_invalidate_range_start()
        change_protection()
          change_prot_numa()
            task_numa_work()
              task_work_run()
                xfer_to_guest_mode_handle_work()
```

Both traces converge on the same fundamental trigger: the Linux NUMA autobalancing subsystem. The call to `do_huge_pmd_numa_page` → `migrate_misplaced_page` → `migrate_pages` in the first trace is architecturally anomalous. SNP guest memory is locked and pinned by design; it is not eligible for migration. Yet the NUMA autobalancer was still initiating the migration code path against this memory, which drove the MMU notifier chain and ultimately invoked `wbinvd_on_all_cpus()` across all 500 physical CPUs on the test system.

4.2 The NUMA Autobalancer's Deficiency with Pinned Memory

The Linux kernel's NUMA autobalancing mechanism periodically scans address spaces and clears the present bit in PTEs for pages deemed suboptimal for the accessing thread [11, 12]. This PTE zapping is performed speculatively, without verifying migration feasibility. The rationale is that the cost of the subsequent page fault, which re-evaluates migration feasibility, is considered acceptable relative to the memory locality gains achieved when migration succeeds.

For normal, unpinned anonymous memory, this approach is sound. For locked and pinned SNP guest memory, however, it creates a pathological interaction. When the NUMA autobalancer zaps PTEs, covering guest physical memory and treating it as any other anonymous mapping, it triggers `change_prot_numa()` or `try_to_migrate_one()`, which in turn fires the MMU notifier chain. The `kvm_mmu_notifier_invalidate_range_start()` callback is invoked, which reaches `sev_guest_memory_reclaimed()` and calls `wbinvd_on_all_cpus()`. When the subsequent page fault reveals that the page cannot be migrated, the original PTE is reinstalled. This reinstallation triggers another round of MMU notifier events and another call to `wbinvd_on_all_cpus()`. The entire sequence produces no beneficial outcome: no page migration occurs, yet the full cost of a system-wide cache flush is incurred on every iteration. This pathological feedback cycle is illustrated in Figure 2.

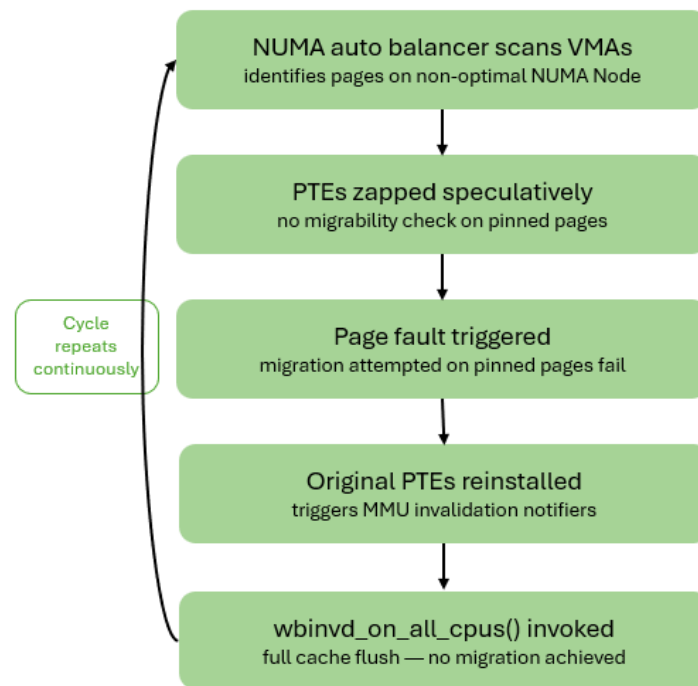


Fig. 2: Pathological NUMA autobalancing cycle on pinned confidential guest memory

4.3 Lock Contention and Guest Re-entry Latency

The performance pathology is compounded by a secondary effect: *mm->mmap_lock* contention. The code paths through *try_to_migrate_one()* and *change_protection()* hold *mm->mmap_lock* for their duration. Other vCPU threads that encounter Nested Page Faults (#NPF) and require *get_user_pages_unlocked()* must acquire this lock in read mode. These threads block on *rwsem_down_read_slowpath()*, as evidenced by additional captured stack traces showing multiple VMM threads in state *D* (uninterruptible sleep).

On a 500-physical-CPU host, *wbinvd_on_all_cpus()* incurs latency proportional to the IPI broadcast round-trip time across all sockets and NUMA nodes. This latency is substantial on the order of hundreds of microseconds to milliseconds depending on system topology and directly delays guest re-entry. When multiple vCPU threads are simultaneously blocked waiting for *mmap_lock* release while the lock holder is blocked in *wbinvd_on_all_cpus()*, the cumulative delay readily exceeds the soft lockup detection threshold of the guest kernel [9], and RCU grace periods fail to complete within expected bounds [10].

5. Upstream Design Discussion and Rejected Alternatives

5.1 Hardware-Assisted Cache Coherency: Capabilities and Limitations

Processor generations that introduced hardware support for cache coherency between encrypted and unencrypted mappings on the same physical page include a platform capability referred to as *SME_COHERENCE* in the platform feature flags [3]. On platforms advertising this capability, the processor guarantees coherency among all CPU cores regardless of whether the memory encryption C-bit is set, effectively eliminating the need for software-initiated cache flushes in the CPU-to-CPU case for SNP hosts. However, *SME_COHERENCE* provides coherency guarantees only within the CPU coherency domain. It explicitly does not extend to Direct Memory Access (DMA) devices. On processor generations that advertise *SME_COHERENCE*, confidential cache lines may still be inconsistently observed by DMA-capable peripherals, such that a DMA read from a page recently used as encrypted guest memory may encounter stale cache content from the prior confidential mapping, resulting in data corruption and potential kernel panics. This limitation renders *SME_COHERENCE* an incomplete solution for general-purpose confidential computing deployments that involve I/O-intensive workloads.

5.2 Alternative Mitigations Considered and Rejected

During upstream discussions [6], several specific improvements to the MMU notifier cache flush path were suggested and tested. A comparative summary of all remediation approaches and their trade-offs is presented in Table 1.

Approach	Mechanism	Platforms covered	Residual gaps	Verdict
clflush / clflushopt substitution	Replace wbinvd with per-line flush for ranges ≤ 2 MB	SME_COHERENCE platforms only	No benefit on non- SME_COHERENT platforms; DMA coherency unresolved	Rejected
Flag-based notifier filtering	Suppress cache flush for change_protection notifier path	All platforms	Treats the symptom, not cause; other triggering paths remain active	Rejected

10.48047/jocaaa.2026.35.05.15

MM subsystem patch	Exempt pinned guest RAM pages from NUMA autobalancing enrollment	All platforms	KSM and “numad” migrate_pages() syscall paths remain unaddressed	Partial fix
SNP + guest_memfd integration	Remove HVA-backed mappings for private encrypted memory; MMU notifier events no longer reach private pages	All SNP platforms	None structurally eliminates the triggering precondition	Accepted upstream

Table 1: Comparison of remediation approaches evaluated during upstream kernel discussion [6]

The first proposal was to substitute *clflush/clflushopt* for *wbinvd_on_all_cpus()* for invalidation ranges up to 2 MB, replacing the full cache invalidation with cache line-level flush instructions. On SME_COHERENT platforms, *clflush/clflushopt* correctly flush guest-tagged cache entries. However, on non-SME_COHERENT platforms, all processor generations lacking SME_COHERENCE support for *clflush/clflushopt* are insufficient: guest-tagged entries require either the *VM_PAGE_FLUSH MSR* or a full *WBINVD* instruction for correct invalidation [3]. This approach therefore provides no optimization on the very platforms where performance is most critical. The second proposal introduced conditional logic *mmu_notifier_invalidate_range_start()* to suppress cache flush invocations based on flags passed to the notifier, specifically to skip the flush for the *change_protection* invocation path associated with NUMA balancing. Although technically feasible, this approach was described as a temporary solution that tackles a symptom instead of the underlying issue. It leaves the fundamental design deficiency intact, the NUMA autobalancer operates on memory it has no business touching and introduces fragility if new triggering paths emerge [6].

The upstream consensus was that the correct architectural approach would be to prevent the NUMA autobalancer from considering SEV/SEV-ES and SNP pinned guest memory as a migration candidate in the first place. Table 2 below summarizes the causal chain driving the performance pathology, to aid synthesis across the alternative approaches evaluated.

Stage	Event
Trigger	NUMA autobalancer speculatively zaps PTEs on pinned SNP guest pages
MMU notifier fires	kvm_mmu_notifier_invalidate_range_start() invoked
Cache flush	wbinvd_on_all_cpus() broadcast IPI across all 500 CPUs
Migration fails	Page is pinned, PTE reinstalled, cycle repeats
Guest impact	vCPU starvation → soft lockups and RCU stalls inside guest

Table 2: Cause-Effect Chain for the SNP MMU Notifier Performance Pathology

A Linux kernel Memory Management (MM) subsystem patch was developed to this end, preventing pinned guest RAM pages from being enrolled in the NUMA autobalancing machinery. However, this patch does not cover all triggering paths: Kernel Same-page Merging (KSM) independently triggers MMU invalidation notifiers during page deduplication, and the “numad” user-space daemon-triggered page migration invokes the *migrate_pages()* system call directly, which constitutes an explicit kernel migration request that bypasses the in-kernel NUMA balancer's filtering logic entirely. The patch is therefore necessary but not sufficient.

6. Workarounds and Their Limitations

6.1 Disabling NUMA Load Balancing and “numad”

As an immediate operational workaround, NUMA autobalancing was disabled on the host by appending *numa_balancing=disable* to the kernel command line, and the “numad” user-space daemon was stopped. With both the in-kernel NUMA balancer and the user-space NUMA management daemon inactive, the anomalous *migrate_pages()* invocations on pinned SNP guest memory cease and the associated MMU notifier events are not generated. Under this configuration, guest soft lockups and RCU stalls are not observed.

However, NUMA autobalancing is a critical performance mechanism for memory-intensive workloads on multi-socket NUMA compute systems [11]. Disabling it results in memory allocations that are not optimally placed relative to the accessing vCPUs' physical CPU affinity, increasing average memory access latency and reducing memory bandwidth for all workloads on the host. For cloud service providers and Linux distribution maintainers, such an approach represents an unacceptable trade-off: the confidentiality guarantee of SNP guests is preserved, but at the cost of global host performance degradation affecting all tenants.

6.2 Targeted MM Subsystem Patch

A more surgical workaround, a patch to the Linux kernel's Memory Management subsystem that exempts pinned guest RAM pages from NUMA autobalancing consideration, addresses the primary trigger path without globally disabling NUMA balancing. This patch prevents the NUMA scanner from inserting NUMA fault PTEs into the address space regions corresponding to pinned guest memory, thereby preventing *do_huge_pmd_numa_page()* and *change_prot_numa()* from being invoked on pinned pages. This patch greatly reduces the occurrence of pathological *wbinvd_on_all_cpus()* calls but does not address KSM-related paths or explicit calls to *migrate_pages()* from “numad”. This is a viable intermediate mitigation for environments that cannot yet move to a kernel version that contains the definitive fix that will be presented in the next section.

7. Final Resolution: SNP + guest_memfd Integration

7.1 Architectural Rationale with guest_memfd

The definitive resolution to the SNP MMU notifier performance pathology is grounded in a fundamental architectural reconsideration of how SNP private memory is mapped in the host address space. The *guest_memfd* subsystem, merged upstream as part of ongoing KVM confidential computing infrastructure development [8], provides a dedicated file-descriptor-based interface for managing guest private memory, continuing a line of work in which secure execution environments are architected to minimize the trusted software footprint and reduce exposure to host-level subsystems. Under this model, SNP private/encrypted memory pages are backed by *guest_memfd* file-backed memory rather than anonymous host virtual memory

(HVA) mappings. Critically, this private memory is not accessible via any host process virtual address mapping.

The MMU notifier framework, including `kvm_mmu_notifier_invalidate_range_start()`, is fundamentally an HVA-based notification mechanism: it fires when the host kernel modifies PTEs in a process's virtual address space [4]. With SNP private memory unmapped from any host process VMA (Virtual Memory Area), the NUMA autobalancer has no PTE to zap and no migration to attempt on private guest pages. HVA-based MMU notifier events reaching `sev_guest_memory_reclaimed()` under SNP+`guest_memfd`, therefore, pertain exclusively to shared (unencrypted) guest memory pages, those explicitly mapped into the host address space for legitimate I/O or Host-VM communication purposes. For shared pages, no WBINVD-based cache invalidation is required because shared pages carry no confidential encryption tag. The architectural contrast between the pre- and *post-guest_memfd* memory mapping topology is illustrated in Table 3.

Aspect	Before <code>guest_memfd</code> HVA-backed private memory	After <code>guest_memfd</code> File-descriptor-backed private memory
Private memory backing	Anonymous host virtual memory (HVA); guest encrypted pages reside in host process VMAs	<code>guest_memfd</code> file-backed memory; no host VMA created for private pages
Host address space visibility	Private pages are addressable via host user-space virtual addresses	Private pages are unreachable via any host process virtual address mapping
NUMA autobalancer exposure	Balancer scans host VMAs, zaps PTEs for private guest pages, triggering spurious migration attempts	No host VMA to scan; NUMA balancer has no visibility into private guest pages
MMU notifier scope	Notifier events fire for both private (encrypted) and shared (unencrypted) guest pages	Notifier events fire for shared pages only; private pages generate no HVA-based notifications
<code>wbinvd_on_all_cpus()</code> behavior	Invoked on every MMU invalidation event, including spurious NUMA-triggered ones on pinned pages	Bypassed entirely for SNP guests; shared pages require no cache flush on reclaim
Guest soft lockup risk	High on large CPU-count hosts, repeated inter-processor interrupts (IPIs) on all CPUs and full cache flushes delay guest re-entry beyond lockup threshold	Eliminated-structural precondition for spurious flushes no longer exists

Table 3: Architectural comparison of SNP guest private memory management before and after `guest_memfd` integration. The removal of HVA-backed mappings eliminates the NUMA autobalancer's ability to trigger MMU notifier events on private encrypted pages.

This architectural observation resolves the performance pathology at its root: the structural precondition for erroneous cache flush invocations, the existence of HVA-backed mappings for SNP private memory, is eliminated by design.

7.2 The Upstream Kernel Fix

The upstream correction, which was merged into the Linux/KVM tree [7], modifies *sev_guest_memory_reclaimed()* to return immediately for SNP guests, bypassing *wbinvd_on_all_cpus()* entirely:

```

□ void sev_guest_memory_reclaimed(struct kvm *kvm)
{
-  if (!sev_guest(kvm))
+  /*
+   * With SNP+gmem, private/encrypted memory is unreachable via the
+   * hva-based mmu notifiers, so these events are only actually
+   * pertaining to shared pages where there is no need to perform
+   * the WBINVD to flush associated caches.
+   */
+  if (!sev_guest(kvm) || sev_snp_guest(kvm))
      return;

```

wbinvd_on_all_cpus();

□ The guard condition *sev_snp_guest(kvm)* causes an early return for all SNP guests, irrespective of NUMA balancing activity, KSM, or “*numad*” behavior on the host. With this fix applied on the same 500-physical-CPU host with 128-vCPU SNP guests, soft lockup and RCU stall symptoms are fully eliminated without any modification to NUMA balancing policy or host performance configuration.

Conclusion

This case study demonstrates a class of emergent performance pathology that arises when a correctness-motivated security patch in one kernel subsystem interacts adversarially with a performance-critical mechanism in a second, unrelated subsystem. The causal chain documented here is precise: a cache incoherency fix introduced a full system-wide cache flush operation into the MMU notifier hot path; the NUMA autobalancing subsystem routinely triggered that hot path by speculatively zapping page table entries for pinned confidential guest memory without verifying migration feasibility; and on a high-CPU-count host, the accumulated cache flush latency and inter-processor interrupts broadcast across 500 physical processors was sufficient to starve guest vCPUs of execution time, manifesting as soft lockups and RCU stalls visible inside the guest. No single component was independently defective. The pathology was entirely emergent from their interaction at scale.

The resolution, enabled by the *guest_memfd* abstraction, is instructive beyond the immediate improvement. By eliminating host virtual address mappings for private encrypted guest memory, the architecture removes the structural precondition that allowed a host memory management subsystem to operate on memory it had no legitimate reason to touch. This reflects a broader design principle: security isolation boundaries must be expressed in the kernel's memory management abstractions, not enforced solely at the hardware layer. When software subsystems can observe and manipulate memory regions whose semantics they do not understand, correctness patches in one layer will inevitably produce performance regressions in another.

10.48047/jocaaa.2026.35.05.15

Architectural clarity, making private memory genuinely unreachable rather than merely protected, eliminates an entire category of such interactions by construction.

For practitioners operating confidential VM infrastructure at scale, three lessons follow directly. Security patches that introduce CPU-count-proportional operations into kernel hot paths require performance validation at production-scale topology before upstreaming, not merely correctness verification on small test systems. NUMA autobalancing is a non-obvious performance variable for confidential VM deployments and must be included in regression test matrices alongside vCPU count and guest memory size. Finally, the interaction surface between confidential computing extensions and general-purpose kernel subsystems, memory management, and page migration, same-page merging, remains an active source of correctness and performance risk, warranting continued co-design discipline across kernel subsystem boundaries.

Future Work: Several directions merit further investigation. First, the interaction patterns documented here for AMD SEV-SNP warrant systematic study on Intel TDX, which employs a different memory isolation model but shares the Linux KVM MMU notifier stack. Second, the guest_memfd abstraction is still maturing; its behavior under high-density multi-tenant deployments with heterogeneous workload mixes, particularly I/O-intensive and KSM-eligible workloads, has not been characterized at scale. Third, the MM subsystem patch that exempts pinned guest RAM from NUMA auto-balancing may interact with future kernel features such as memory tiering and CXL-backed memory pools in non-obvious ways, warranting proactive co-design analysis before those features reach production confidential computing deployments.

References

- [1] Confidential Computing Consortium, "A Technical Analysis of Confidential Computing," 2023. Available: https://confidentialcomputing.io/wp-content/uploads/sites/10/2023/03/CCC-A-Technical-Analysis-of-Confidential-Computing-v1.3_unlocked.pdf
- [2] David Kaplan et al., "AMD Memory Encryption," 2021. Available: <https://docs.amd.com/v/u/en-US/memory-encryption-white-paper>
- [3] AMD Inc., "AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More," AMD White Paper, 2020. Available: <https://docs.amd.com/v/u/en-US/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more>
- [4] Jonathan Corbet, "Memory Management Notifier," Linux Weekly News (LWN.net), 2008. Available: <https://lwn.net/Articles/266320/>
- [5] Ovidiu Panait, "KVM: x86: Add cache flush operations to mmu_notifier for SEV," Linux KVM Mailing List Archive, 2022. Available: <https://lore.kernel.org/kvm/YzJFvWPb1syXcVQm@google.com/T/#mb79712b3d141cabb166b5049>
- [6] Ashish Kalra, "KVM: SEV: Performance Issue with mmu_notifier and NUMA Balancing," Linux KVM Mailing List Archive, 2022. Available: <https://lore.kernel.org/kvm/20220927000729.498292-1-Ashish.Kalra@amd.com/>
- [7] Michael Roth, "KVM: x86: SNP: Skip wbinvd for SNP+gmem guests in sev_guest_memory_reclaimed," Linux KVM Mailing List Archive, 2024. Available: <https://lore.kernel.org/kvm/20240510211024.556136-16-michael.roth@amd.com/>
- [8] Paolo Bonzini et al., "KVM: guest_memfd: A New Memory Backend for Confidential VMs," Linux Weekly News (LWN.net), 2023. Available: <https://lwn.net/Articles/950433/>

10.48047/jocaaa.2026.35.05.15

- [9] Daniel P. Bovet and Marco Cesati, "Understanding the Linux Kernel," O'Reilly, 2000. Available: <https://scis.uohyd.ac.in/~atulcs/COOS/Linux-HOWTOs/linux%20kernel/Understanding%20The%20Linux%20Kernel.pdf>
- [10] Paul E. McKenney and John D. Slingwine, "Read-Copy Update: Using Execution History to Solve Concurrency Problems," Proc. Ottawa Linux Symposium (OLS), 2002. Available: <http://www.rdrop.com/~paulmck/scalability/paper/rclockpdcspdf.pdf>
- [11] Mohammad Dashti et al., "Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems," Proc. ACM ASPLOS, 2013. Available: <https://people.ece.ubc.ca/sasha/papers/asplos284-dashti.pdf>
- [12] Jonathan Corbet, "AutoNUMA: the other approach to NUMA scheduling," Linux Weekly News, 2012. Available: <https://lwn.net/Articles/488709/>
- [13] S. Arnautov et al., "SCONE: Secure Linux Containers with Intel SGX," USENIX OSDI, 2016. Available: <https://www.usenix.org/system/files/conference/osdi16/osdi16-arnautov.pdf>
- [14] Moritz Schneider et al., "Confidential VMs Explained: An Empirical Analysis of AMD SEV-SNP and Intel TDX," Proceedings of the ACM on Measurement and Analysis of Computing Systems, 2024. Available: <https://dl.acm.org/doi/10.1145/3700418>