International Journal of Computational and Experimental Science and ENgineering (IJCESEN)

Vol. 12-No.2 (2026) pp. 631-642
http://www.ijcesen.com

ISSN: 2149-9144

Research Article

Interrupt Security for Virtual Machines and Guest Environments: AMD SEV-SNP Restricted and Alternate Interrupt Injection

Ashish Kalra*

Independent Researcher, USA
* **Corresponding Author Email:** ashis2h@gmail.com- **ORCID:** 0000-0002-5247-9950

Article Info:

DOI: 10.22399/ijcesen.5286
Received : 05 April 2026
Revised : 22 May 2026
Accepted : 25 May 2026

Keywords

AMD SEV-SNP,
Confidential Computing,
Interrupt Injection,
Restricted Injection,
Alternate Injection,
KVM

Abstract:

Encrypting a virtual machine's memory addresses only one dimension of the confidential computing security problem. When the hypervisor retains architectural control over interrupt delivery, a malicious or compromised hypervisor can exploit interrupt injection to undermine the confidentiality and integrity guarantees that confidential virtual machines are designed to provide. Guest operating systems carry deep assumptions about interrupt behavior rooted in bare-metal execution, and violating those assumptions, even for a single instruction cycle, can place the guest kernel in states that neither hardware designers nor operating system developers planned for. AMD Secure Encrypted Virtualization with Secure Nested Paging (SEV-SNP) confronts this problem through two mutually exclusive hardware-enforced mechanisms: Restricted Interrupt Injection and Alternate Interrupt Injection. This article examines the architectural motivation for interrupt security, the design of the HV Doorbell Page (HVDB), the full interrupt delivery flow from the KVM hypervisor through to the Linux guest kernel, interrupt shadow handling, system vector dispatch optimization, preemption issues causing guest hangs at interrupt exit, and nested exception detection. The treatment connects each mechanism to published attacks, including the HECKLER interrupt injection exploit, and addresses the specific difficulties that have so far prevented upstream Linux kernel acceptance of the Restricted Injection implementation.

1. Introduction

1.1 Motivation: Why Interrupt and Exception Protection for Guests?

The promise of confidential computing is that a workload running inside a virtual machine should remain protected even when the underlying infrastructure is compromised. AMD SEV-SNP [5] contributes substantially to this goal through encrypted memory and providing strong guest integrity protection, but encrypted memory alone cannot resolve a subtler vulnerability: the interrupt delivery interface. Guest operating systems are written with specific assumptions about when and how interrupts arrive. On x86 hardware, an operating system expects that maskable interrupts will not be delivered when EFLAGS.IF is cleared, and that low-priority interrupts will not appear when the Task Priority Register (TPR) is elevated. These are not conventions; they are invariants on which kernel code paths depend for correctness. In a non-confidential virtualization environment, trusting the hypervisor to respect these invariants is a reasonable assumption. In a confidential computing environment, it is not. The hypervisor sits outside the trust boundary by definition, which means every capability it retains over the guest is a potential attack vector. The ability to inject arbitrary interrupts is a particularly powerful one, because a single well-timed injection can drive the guest into an unplanned execution state without requiring any memory decryption.

Specific injection scenarios that motivated this work include: virtual interrupt vector 29 (#VC), the guest exception used by SEV-ES guests for VMGEXIT communication, which can be triggered at inopportune moments to create race conditions; virtual interrupts 0 and 14, division-by-zero and page fault injection, which can redirect guest execution or activate unhandled exception paths; and INT 80h, the legacy Linux system call vector, injection of which can redirect execution to attacker-chosen code within the guest. Each

represents a realistic attack that a malicious hypervisor can execute. Software mitigations inside the guest are insufficient because the hypervisor can time injections to arrive before those mitigations take effect.

1.2 Why Secure Interrupt Delivery Is Needed

The HECKLER attack, published in 2024, demonstrates the severity of the threat concretely [4]. HECKLER shows that a malicious hypervisor can inject specifically selected interrupt vectors into an AMD SEV-SNP guest to redirect its execution flow, break cryptographic constant-time guarantees, and in certain configurations achieve a full compromise of the guest's confidential state. The attack requires no memory decryption; it exploits solely the interrupt injection interface, the same interface that has always been present and trusted in non-confidential environments. Beyond HECKLER, fault injection techniques have been applied against AMD SEV at the hardware level to extract architecturally protected secrets [2], and CipherLeaks has demonstrated that even the ciphertext of encrypted guest memory can leak information about cryptographic keys when the attacker can observe memory access patterns or trigger cryptographic operations at chosen moments [3].

Taken together, these results establish that protecting confidential virtual machines requires addressing not only memory encryption but every interface through which the untrusted hypervisor can influence guest execution. The interrupt delivery interface is among the most direct and most exploitable of those interfaces.

1.3 Solution Overview

AMD SEV-SNP addresses the interrupt security problem by giving virtual machines the ability to opt into one of two hardware-enforced interrupt security modes. Restricted Interrupt Injection disables all hypervisor-controlled interrupt and event delivery except for a single new exception vector, #HV (vector 28), which serves as a doorbell. Actual event information is communicated through a shared-memory structure, the HV Doorbell Page (HVDB), that the guest reads and processes on its own terms. Alternate Interrupt Injection preserves the standard interrupt queuing and injection interfaces but transfers control over them from the hypervisor to the guest. The two modes are mutually exclusive; a guest must select one at initialization.

The design rationale for providing two distinct modes reflects the practical diversity of confidential computing deployments. Restricted Injection is the more conservative option: it eliminates nearly all direct hypervisor influence over guest interrupt delivery, replacing the conventional injection path with a para-virtualized HVDB-based channel that the guest fully controls. Alternate Injection is less disruptive to the existing interrupt infrastructure and may be preferable in deployments where the operational overhead of full para-virtualized dispatch is a concern, but it does not provide the same degree of hypervisor isolation.

This article focuses primarily on Restricted Injection, which offers the broadest reduction in hypervisor influence. The sections that follow trace the full implementation: the HVDB structure and doorbell page protocol (Section 2), the interrupt delivery flow from KVM through to the Linux guest kernel (Section 3), interrupt shadow semantics and the safe-halt fix (Section 4), deferred acknowledgement and interrupt re-enabling paths (Section 5), optimized system vector dispatch via the sysvec_table ELF mechanism (Section 6), preemption issues at interrupt exit (Section 7), nested exception detection (Section 8), and the upstream integration challenges that remain open (Section 11).

2. Restricted Interrupt Injection

2.1 Architectural Overview

When Restricted Injection is enabled in the SEV-SNP Virtual Machine Control Block (VMCB), the hardware stops honoring the hypervisor's requests to inject any interrupt or exception other than #HV (vector 28). Non-Maskable Interrupts (NMIs), System Management Interrupts (SMIs), INIT signals, and all Advanced Programmable Interrupt Controller (APIC)-based interrupts are suppressed. The only notification channel the hypervisor retains is the #HV doorbell. Guests running under Restricted Injection are expected to communicate with the hypervisor about events through a software-managed para-virtualization interface, which uses #HV injection as a doorbell to inform the guest that new events have been posted.

Virtual machines ordinarily rely on interrupt delivery through an architectural APIC, even for synthetic messages delivered by the host. The HVDB structure is deliberately aligned with APIC behavior so that existing guest interrupt dispatch and management logic can be reused with minimal modification. The guest is not required to rewrite its interrupt handling stack; it adds the HVDB reading and vector dispatch layer on top of existing infrastructure.The contrast with conventional virtualization interrupt delivery is significant. In a

standard non-Restricted configuration, the hypervisor exercises full control over which interrupt vectors are delivered, when they arrive, and in what order. Under Restricted Injection, that control is removed at the hardware level through the VMCB configuration. The hypervisor cannot bypass this restriction through software alone; the suppression is enforced by the SEV-SNP hardware itself. To operate under Restricted Injection, the Guest OS must have kernel-level support to manage the #HV doorbell, using the Guest-Hypervisor Communication Block (GHCB) to securely communicate with the hypervisor and allow the guest to handle events on its own terms. On the platform side, the feature is enabled within the VM configuration — for example, in QEMU — by creating an isolated guest setting that limits host access to the guest's interrupt delivery path.

2.2 HV Doorbell Page (HVDB)

The HV Doorbell Page is defined by the Guest-Hypervisor Communication Block (GHCB) specification, specifically Section 5 on SNP Restricted Injection, and is registered by the guest during initialization [7]. It contains two key components: the PendingEvent field and the EOI Assist field

2.2.1 PendingEvent Field

The PendingEvent field is a 16-bit value that the hypervisor writes before injecting #HV, and that the guest reads inside the #HV handler to determine what action to take. Table 1 provides the complete bit layout.The NoFurtherSignal bit (bit 15) governs #HV injection for non-maskable events. Per the GHCB specification, when this bit is set the host will not inject another #HV due to a non-maskable event (NMI or #MC); the guest is responsible for clearing it after switching off the IST stack and completing event processing. Bits [7:0] carry the APIC-aligned vector, allowing existing interrupt dispatch logic to be reused.

2.2.2 EOI Assist Field (NoEoiRequired)

The EOI Assist field, designated NoEoiRequired, enables end-of-interrupt processing without requiring a VM exit. When the host can perform an End-of-Interrupt (EOI) without a guest exit, it sets NoEoiRequired to a non-zero value. When the guest wishes to perform an EOI, it should attempt to atomically change NoEoiRequired from non-zero to zero. If successful, no further processing is necessary; if unsuccessful, the guest must request an explicit EOI by performing VMGEXIT to request a WRMSR to the X2APIC EOI MSR. Full description of both fields is provided in the GHCB specification [7].

3. Interrupt Flow: KVM Hypervisor to Guest Linux Kernel

3.1 KVM/Host Side Interrupt Injection Flow

On the KVM host side, the change from conventional interrupt injection is conceptually straightforward but operationally precise. Instead of injecting an interrupt directly via VMCB event injection, KVM writes the event into the guest's HVDB and then injects #HV. A critical guard condition applies that is governed by two fields in the HVDB: the host will not inject another #HV if both NoFurtherSignal is set and PendingEvent.Vector is non-zero. Per the GHCB specification, a new #HV is scheduled only when PendingEvent.Vector was previously zero and NoFurtherSignal was previously zero. KVM must therefore evaluate both conditions before proceeding with doorbell injection, ensuring that non-maskable events such as NMI and #MC can still be signaled via #HV independently of vector interrupt delivery when the NoFurtherSignal bit is clear.

3.2 Guest #HV Exception Flow

Inside the guest, the #HV handler carries more responsibility than a conventional exception handler. It must detect any nested-#HV race condition, switch execution off the Interrupt Stack Table (IST) stack, clear NoFurtherSignal, read the HVDB, dispatch the waiting events, and return cleanly. The ordering of these steps is not arbitrary; each step depends on the previous one having been completed safely. The design ensures that the hypervisor's influence over event delivery is confined entirely to the content of the HVDB, which the guest processes entirely on its own terms.

4. Interrupt Shadows

4.1 Behavior Under Restricted Injection

On bare-metal x86 hardware, executing STI or MOV SS creates a one-instruction interrupt shadow: hardware suppresses interrupt delivery for the subsequent instruction. This safety mechanism underlies patterns such as STI followed immediately by HLT, ensuring that an interrupt cannot arrive between the two instructions and ensure the HLT instruction is guaranteed to execute before any pending interrupt is handled, preventing the CPU from immediately waking up. Under

Restricted Injection, this guarantee does not hold for #HV. Because #HV is an exception rather than a hardware-delivered interrupt, hardware delivers it without regard to interrupt shadows. The STI+HLT atomic idiom is therefore no longer safe as written: a #HV could arrive after STI but before HLT completes, and the guest's idle loop could miss a pending event.

4.2 Safe Halt Handling

Correcting this requires the guest's native_safe_halt() function to first execute STI to re-enable interrupts, then explicitly poll the HVDB for any pending events, and only then issue the HLT instruction. to ensure that guest can properly suspend when no interrupts are pending while also ensuring that a guest will neither miss pending interrupts nor suspend

5. Interrupt Enable and Disable

5.1 Deferred Acknowledgement

One of the more counterintuitive aspects of Restricted Injection is that the guest can receive a #HV at any time, regardless of the current state of EFLAGS.IF. Normally, clearing IF instructs the processor not to deliver maskable interrupts. Under Restricted Injection, that instruction still governs hardware-delivered interrupts, but #HV is an exception and hardware exceptions ignore IF. The guest handles this asymmetry through deferred acknowledgement. When a #HV arrives with IF cleared, the handler observes the pending event but does not immediately dispatch the associated interrupt vector. Instead, dispatch is deferred until IF is re-enabled and the vector's priority level is appropriate for delivery. The #HV handler itself always executes; it is the dispatch of the contained vector that is deferred.

5.2 Interrupt Re-Enabling Code Paths

The deferred dispatch model creates a new obligation for every kernel code path that re-enables interrupts. On bare metal, a STI instruction is sufficient; the processor automatically consults the interrupt controller and delivers any pending interrupt. Under Restricted Injection, the processor has no interrupt controller to consult; polling the HVDB is the guest's responsibility. All call-sites enabling interrupts invoke arch_local_irq_enable() which on x86 invokes native_irq_enable(), hence, this explicit HVDB check is added here, as Listing 4 shows. Similarly, every return-from-exception-path invokes paranoid_exit(), which needs the explicit HVDB checks as Listing 4 shows.
This is a non-trivial change to propagate through the kernel, and it is among the primary reasons upstream integration has been challenging.

6. Optimized System Vector Handling

6.1 System Vector Dispatch via sysvec_table

System vectors, the interrupt vector range that the Linux kernel uses for APIC-delivered messages including the local timer, inter-processor interrupts (IPIs), and APIC error notifications, each have their own Interrupt Descriptor Table (IDT) entry and handler stub under normal operation. Under Restricted Injection, none of them arrive through the IDT. They appear as vector numbers in PendingEvent[7:0] and must be dispatched programmatically from within the #HV handler. A naive implementation would use a switch statement in the #HV handler with one case per system vector. This approach works but is brittle and cumbersome adding any new system vector elsewhere in the kernel would require a corresponding change to the #HV dispatch code. A more maintainable approach is a dispatch table populated at link time.

6.2 Dynamic Table Construction

The sysvec_table is built using a dedicated, named ELF section. Each system vector handler registers itself in this section at build time using the DECLARE_SYSVEC() macro. The linker collects all these entries into a contiguous section bounded by __sysvec_table_start and __sysvec_table_end. During kernel init, this ".sysvec_vectors" section is walked and a static array containing the addresses of all system vectors is created. During #HV exception processing, this array is used to vector into the specific system vector handler. This removes all switch/case and #ifdef handling of system vectors in #HV exception handling. Importantly, there is no change in generic x86 interrupt handling code except modification of the DECLARE_IDTENTRY() assembly-specific macro.

7. Fix Preemption Issues in Interrupt Exit Code Path

When a hardware interrupt returns through the normal ret_from_intr path, the kernel automatically checks whether a higher-priority task became runnable during the interrupt and calls the scheduler if appropriate.

Problem: Any HardIRQ/SoftIRQ processing would have made tasks unblocked waiting for I/O, and those tasks will be rescheduled next. During the IRQ return/exit code path, the TIF_NEED_RESCHED flag is checked and, if set, irqentry_exit_to_user_mode()/exit_to_user_mode_ prepare() will invoke schedule(). This works correctly in the non-Restricted Injection case, but with Restricted Injection active, this causes the thread context executing the #HV exception handler to be explicitly preempted and potentially rescheduled on another (v)CPU. On being rescheduled, the #HV exception handler continues to use the HVDB pointer set up for the (v)CPU it was previously running on, handling pending interrupts for that (v)CPU while issuing EOIs to the (L)APIC of the currently active (v)CPU. Since the (L)APIC on the rescheduled (v)CPU has none of these interrupts in-request/in-service, there is a mismatch on the EOIs issued, causing interrupts to remain pending on the host, awaiting guest EOI. This blocks the host from issuing further interrupts for these devices, causing the guest to wait indefinitely for I/O completion interrupts, resulting in guest hangs. As a side effect, guest per-CPU IRQs may be handled on a different (v)CPU than intended, causing kernel debug logs from within __common_interrupt().

Fix: Modify architecture-specific macros DEFINE_IDTENTRY_IRQ() and DEFINE_IDTENTRY_SYSVEC() to not invoke irqentry_exit() (the interrupt return code path). Instead, they invoke a new wrapper function, irqentry_exit_hv_cond(), which checks whether the current context is executing inside the #HV exception handler and was entered from user mode. If so, it shortcuts the interrupt return path and continues #HV exception handling without performing any user-mode IRQ return processing. Otherwise, it falls through to the standard interrupt return path.

8. Handling Nested #HV Exceptions

8.1 Nesting Model and IST Stack Constraints

#HV is assigned an IST entry, and the IST stack is a fixed-size stack. This creates a dangerous edge case. If a second #HV exception arrives while the first is still executing on that IST stack, the processor will start the second handler at the top of the same IST stack, overwriting the frame that the first handler was going to return to via IRET. At that point, the first handler's execution context is gone and there is no clean recovery path.

8.2 No Further Signal Gate and the Race Window

As per the GHCB specification, the hypervisor will not inject another #HV if PendingEvent.NoFurtherSignal is set, and the guest handler clears this only after switching out of the IST stack and handling the current #HV. But, this does not account for a malicious hypervisor, which can always inject #HV interrupts without conforming to the GHCB specification. This remains an open challenge, as explained in Section 11.2. However, there is a window between the moment #HV is delivered and the moment the handler clears NoFurtherSignal during which a second #HV could arrive. During that window the handler is still executing on the IST stack.

8.3 Nested Exception Detection

The solution is to detect this condition at the very start of the #HV handler, before performing any other work. If the interrupted stack pointer falls within the IST stack bounds, a nested #HV has arrived in the pre-NoFurtherSignal-clear window, and the IRET frame of the previous handler is already corrupted. The only correct response is a kernel panic. Attempting to continue execution from this state would result in unpredictable behavior as the first handler eventually returns through a corrupted frame [6].
After the handler switches off the IST stack and clears NoFurtherSignal, subsequent #HV exceptions are handled safely.

9. Comparison: Restricted Injection vs. Alternate Injection

Table 2 summarizes the key differences between the two AMD SEV-SNP interrupt security modes across the dimensions most relevant to confidential computing deployments.
Restricted Injection eliminates the hypervisor's ability to inject arbitrary vectors but requires the guest to operate a complete software interrupt dispatch stack through the HVDB and sysvec_table. Alternate Injection preserves more of the existing interrupt infrastructure but leaves a wider residual attack surface.
Restricted Injection provides the maximum reduction in hypervisor influence over guest execution but requires the guest to implement HVDB-based dispatch, HVDB-aware safe-halt logic, explicit HVDB polling on every interrupt re-enable code path, and the fixing preemption issues described in Section 7. Alternate Injection is less demanding to implement on the guest side but does

not fully eliminate the hypervisor's ability to influence which vectors are delivered. The choice between the two modes should be made with reference to the specific threat model and the deployment's tolerance for implementation complexity. Restricted Injection offers better performance than Alternate Injection and requires no Virtual Machine Privilege Level (VMPL) transitions; however, it is not currently preferred for Linux guests due to the obstacles in upstreaming, particularly the nested exception detection problem in x86 exception handling. Alternate Injection, by contrast, is intended for injection into a Secure VM Service Module (SVSM), requires VMPL transitions, does not carry the concerns associated with Restricted Injection, is simpler to implement, and can handle nested exceptions properly.

10. Security Analysis

10.1 Threat Model

Both modes operate under the same threat model: the hypervisor is fully malicious. It can inject arbitrary interrupts, modify shared memory accessible to the guest, and time its actions to coincide with critical guest code paths. The guest kernel and its trusted firmware define the Trusted Computing Base (TCB). The hypervisor does not [1][4]. The HECKLER attack is the most directly relevant published exploit within this model [4]. HECKLER demonstrates that a malicious hypervisor can inject carefully selected vectors into a SEV-SNP guest running Linux to redirect execution and break cryptographic constant-time guarantees without decrypting guest memory. Under Restricted Injection, no such direct vector injection is possible; the only injection path the hypervisor retains is the #HV doorbell.

10.2 Residual Attack Surfaces

Restricted Injection does not eliminate every avenue of hypervisor influence. Several residual surfaces remain. First, the hypervisor retains control over the timing of #HV injection, which creates a theoretically exploitable timing side channel, though exploiting it is considerably harder than direct vector injection. Second, the HVDB resides in shared memory that both parties can read/write to; a malicious hypervisor could post unusual or unexpected event combinations to probe edge cases in the guest's dispatch logic, and the guest should validate HVDB content defensively. Third, the hypervisor can withhold #HV entirely, effectively cutting off interrupt delivery, or inject it at a rate sufficient to monopolize guest CPU time in the #HV handler; neither of these enables code execution or data disclosure, but either can degrade or halt the guest. These residual risks are accepted within the SEV-SNP threat model and represent a substantially reduced attack surface compared to unprotected interrupt injection.

11. Implementation Status and Upstream Challenges

11.1 Proof-of-Concept Implementation

Working proof-of-concept implementations of Restricted Injection have been developed across all three layers of the software stack. On the KVM host side, the implementation covers HVDB writes and #HV injection via the VMCB, gated correctly on the dual NoFurtherSignal and PendingEvent.Vector conditions. Guest OVMF firmware has been extended to register the HVDB page via the GHCB protocol and to handle device interrupts under Restricted Injection. The Linux guest kernel implementation includes the #HV IDT entry and IST assignment, HVDB-based dispatch, sysvec_table construction, safe-halt modifications, interrupt re-enable path polling, nested #HV detection, and fixing preemption issues at interrupt exit. An SNP guest has been successfully booted end-to-end using this implementation, confirming that the complete mechanism operates correctly in practice [6] [8][9].

11.2 Issues with Upstream Linux Kernel Integration

Upstream acceptance has been more difficult to achieve. Two problems stand out. First, the nested #HV detection mechanism relies on comparing the interrupted stack pointer against the IST stack bounds. This comparison is not fully reliable in all kernel configurations. Edge cases exist in which this comparison cannot be done reliably, causing a corrupted nested #HV to continue rather than triggering a panic. The crux of the upstream discussion was to enforce this nested #HV handling in microcode. Ideally, #HV would have an internal latch such that a recursive #HV terminates the guest — analogous to double #MC and triple-fault behavior. The proposed hardware model for #HV delivery would: check the internal latch and, if set, terminate the machine; set the latch; and write the IRET frame with a magic bit set. Correspondingly, IRET would check the magic bit and reset the #HV latch. This requires hardware/microcode support and hardware team has rejected this proposal as it is not practical to implement. Upstream maintainers have flagged this correctness issue, and it has not

yet been resolved cleanly [6, 10]. Second, the Linux x86 exception handling infrastructure carries deep assumptions about how exceptions are delivered, nested and returned. Modifying it to accommodate HVDB-based dispatch without introducing regressions in non-SEV-SNP code paths has proven to be a difficult design problem, and the patch series has gone through multiple Request for Comments (RFC) rounds without reaching a consensus on the right approach.

Both issues were discussed publicly at the Linux Plumbers Conference 2022 Confidential Computing Microconference [8] and at KVM Forum 2024 [9]. They remain open. Until the nested exception detection problem is resolved reliably and the broader x86 exception handling integration questions are settled, Restricted Injection support will remain outside the mainline kernel.

Table 1: *PendingEvent Field Bit Layout in the HV Doorbell Page [7]*

Bits	Name	Description
[15]	NoFurtherSignal	When set, indicates the host will not inject another #HV due to a non-maskable event (NMI or #MC) until this bit is cleared by the guest.
[14:10]	Reserved	Reserved for future use.
[9]	Virtual #MC	The hypervisor is presenting a virtual machine check exception to the guest.
[8]	NMI	The hypervisor is presenting a non-maskable interrupt to the guest.
[7:0]	Interrupt Vector	An 8-bit vector number. When non-zero, the hypervisor is presenting an interrupt on that specific vector.

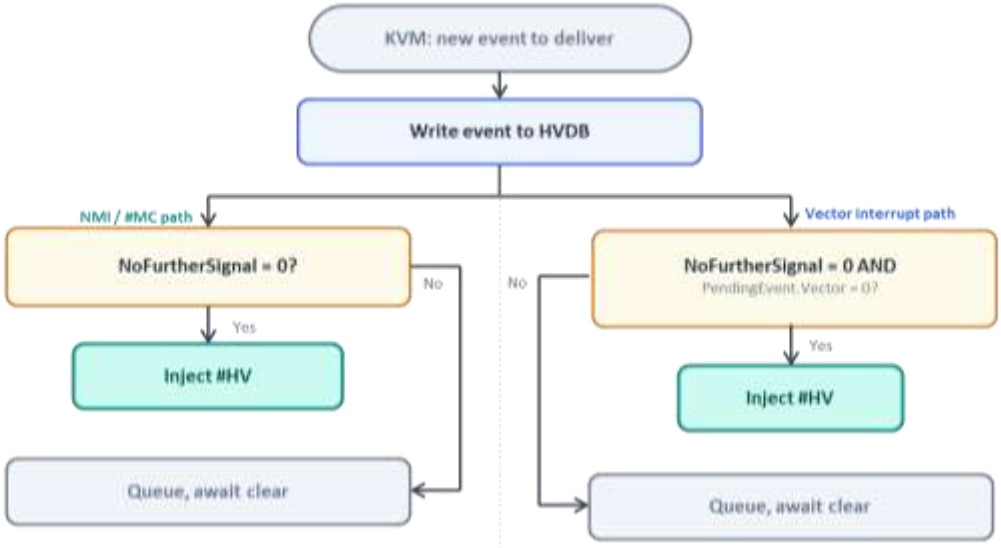

Figure 1: *KVM/Host Side Interrupt Injection Flow*

Listing 1*: KVM Host-Side #HV Injection (C Pseudocode)*

```
inject_pending_event()
      .nmi_pending -> kvm_x86_nmi_allowed() <-> Doorbell.PendingEvent.NMI
                                |
                               > kvm_x86_set_nmi() <-> Doorbell.PendingEvent.NMI = 1
                                      |
                                     > inject_hv()

      .injectable_interrupt -> kvm_x86_interrupt_allowed() <-> Doorbell.PendingEvent.Vector
                                     |
                                    >kvm_x86_set_irq() <-> Doorbell.PendingEvent.Vector = APIC.PendingVector
                                        |
                                       > inject_hv()
```

```
inject_hv()
  {
   If (Doorbell.PendingEvent.NoFurtherSignal == 0) {
    Doorbell.PendingEvent.NoFurtherSignal = 1
    VMCB.EVENTINJ = HV_VECTOR ..
   }
```

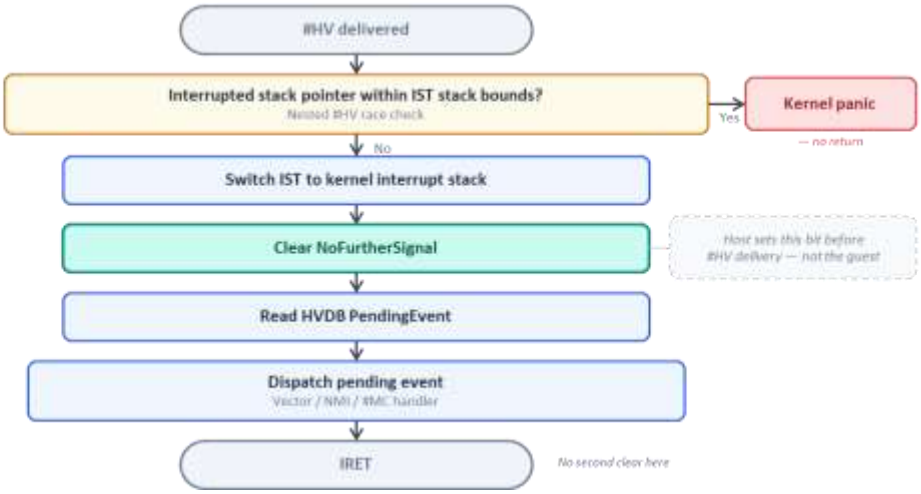

***Figure 2**: Guest #HV Exception Handling Flow*

***Listing 2:** Guest #HV Exception Handler (Linux Kernel C)*

```
hv_raw_handle_exception(regs)
{
 If (EFLAGS.IF == 0) {
  NonMaskableEvents = XCHG(Doorbell.PendingEvent[15..8], 0)
  ; handle NMI or #MC as required
  ; leave handler
 }
 PendingEvent = XCHG(Doorbell.PendingEvent, 0)
  If (PendingEvent.NMI OR PendingEvent.MC) {
  ; handle NMI or #MC as required
  }
  If (PendingEvent.Vector != 0) {
   If (SystemVector) {
    ; dispatch system vectors through sysvec_table[].
    ; sysvec_table[] created dynamically to optimally
    ; dispatch system vector exceptions through a
    ; vector table instead of explicitly calling each
    ; system vector and is placed in a new named
    ; ELF section.
   } Else {
    ; dispatch interrupt vectors via
    ; common_interrupt() handler.
   }
   ; Interrupt exit code path:
   If (user_mode(regs)) {
    ; skip interrupt return/exit code path to avoid
    ; preempting #HV handler.
   } Else {
    ; follow normal interrupt return/exit path
    irqentry_exit(regs, ..);
   }
   ; upon completion of the active interrupt
   NoEoiRequired = XCHG(Doorbell.NoEoiRequired, 0)
   If (NoEoiRequired == 0)
    WRMSR(X2APIC_EOI)
   }
  }
}
```

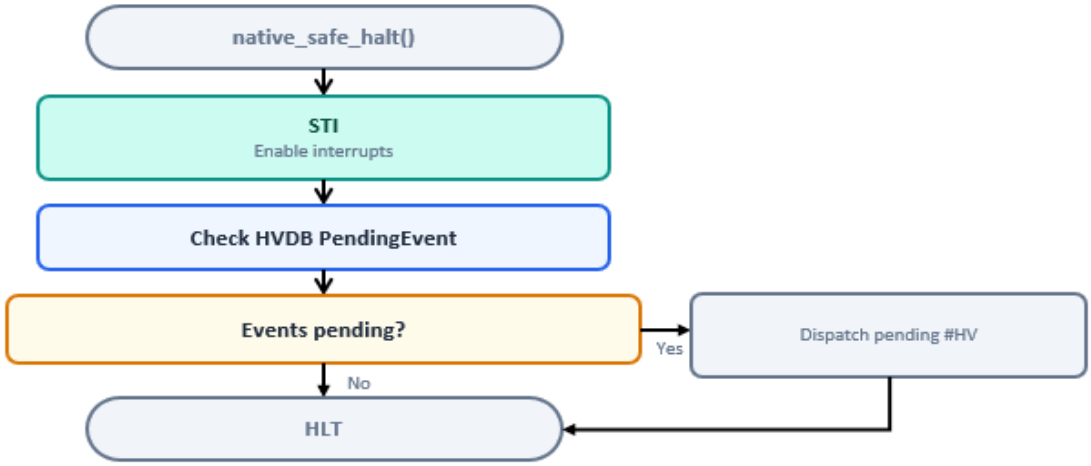

Figure 3: *SNP Restricted Interrupt Injection aware native_safe_halt() Flow*

```
  static inline __cpuidle void native_safe_halt(void)
  {
    asm volatile ("sti : : : "memory");
+   snp_handle_pending_hvdb(..);
    asm volatile ("hlt" : : : "memory");
  }
```

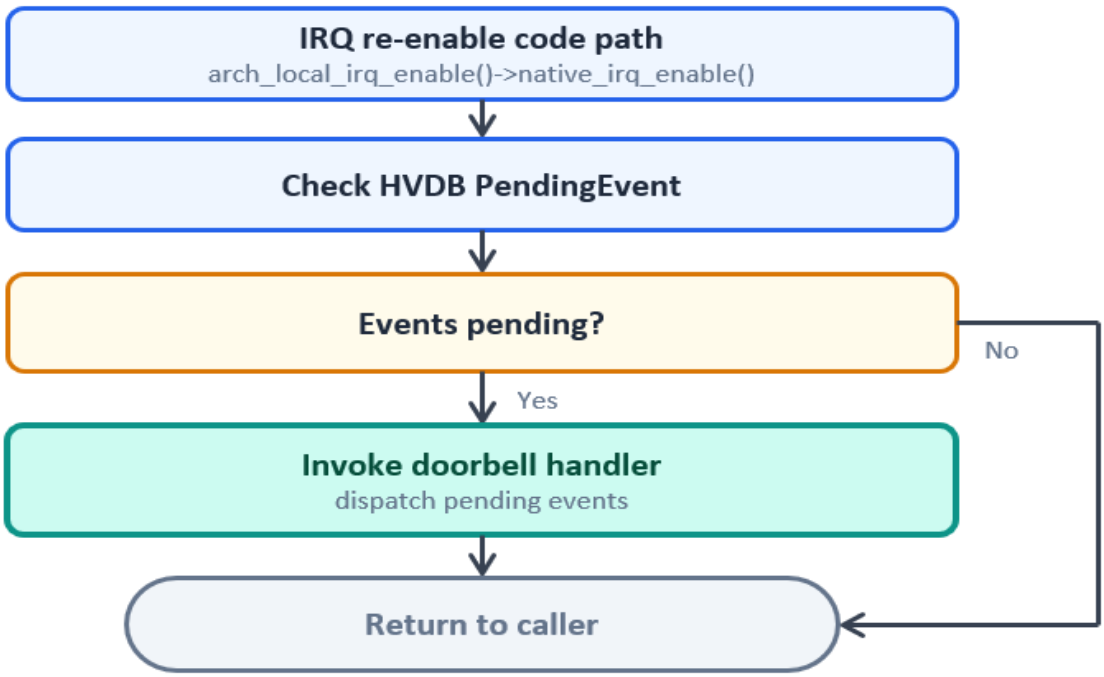

Figure 4: *Interrupt Re-Enable Check Flow*

Listing 4: *Interrupt Re-Enabling Path with Pending #HV Check*

```
static inline native_irq_enable()
  {
    asm volatile("sti" : : : "memory");
+   snp_handle_pending_hvdb(..);
  }
```

```
SYM_CODE_START_LOCAL(paranoid_exit)
...
...
+ /*
+  * If #HV was delivered during execution and interrupts were disabled, check if
+  * it can be handled before the iret (which may re-enable interrupts).
+  */
+ call snp_handle_pending_hvdb
```

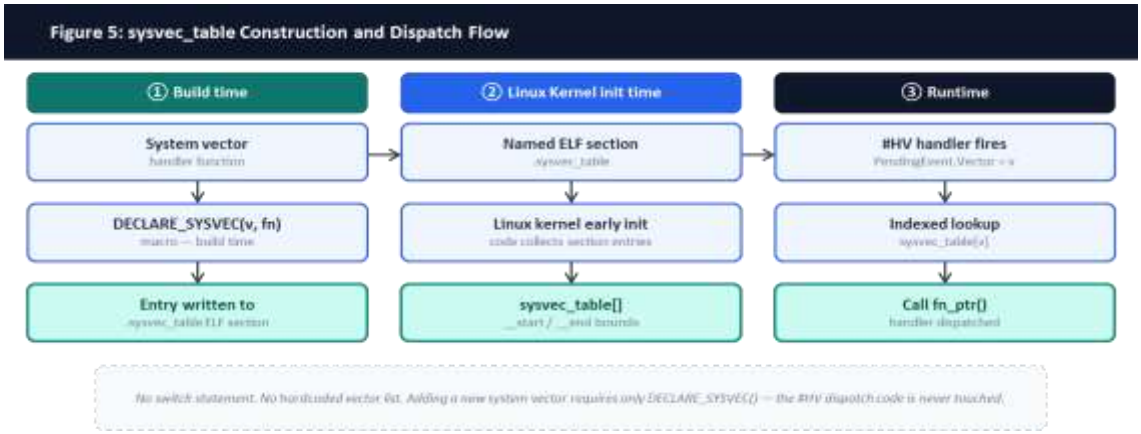

Figure 5: sysvec_table Construction and Dispatch Flow

Listing 5: *sysvec_table ELF Section Declaration and Dispatch*

```
struct __attribute__ ((__packed__)) sysvec_entry {
    unsigned char vector;
    void (*sysvec_func) (struct pt_regs *regs);
};

arch/x86/entry/entry_64.s:

    macro idtentry vector asmsym cfunc
        idtentry_body \cfunc \ has_error_code

        SYM_CODE_END(\asmsym)
    +       .If \vector >= FIRST_SYSTEM_VECTOR && \vector < NR_VECTORS
    +           .section    .system_vectors, "aw"
    +           .byte       \vector
    +           .quad       \cfunc
    +           .previous
    +       .endif
    .endm
```

```
+#ifndef CONFIG_AMD_MEM_ENCRYPT
#define DEFINE_IDTENTRY_IRQ(func)                                       \
    ...
+#else
+#define DEFINE_IDTENTRY_IRQ (func)                                     \
+static void __##func(struct pt_regs *regs, u32 vector);                \
+                                                                       \
+__visible noinstr void func(struct pt_regs *regs,                      \
+                            unsigned long error_code)                  \
+{                                                                      \
+    ...                                                                \
+    run_irq_on_irqstack_cond(_##func, regs, vector);                   \
+    irqentry_exit_hv_cond(regs, state);                                \
+    ...                                                                \
```

```
noinstr void irqentry_exit_hv_cond(..regs, ..state)
{
    ...
    if(user_mode(regs) && hvdb_data->hv_handling_events)
        return;
    else {
        /* follow normal interrupt return / exit path */
        irqentry_exit(regs, state);
    }
}
```

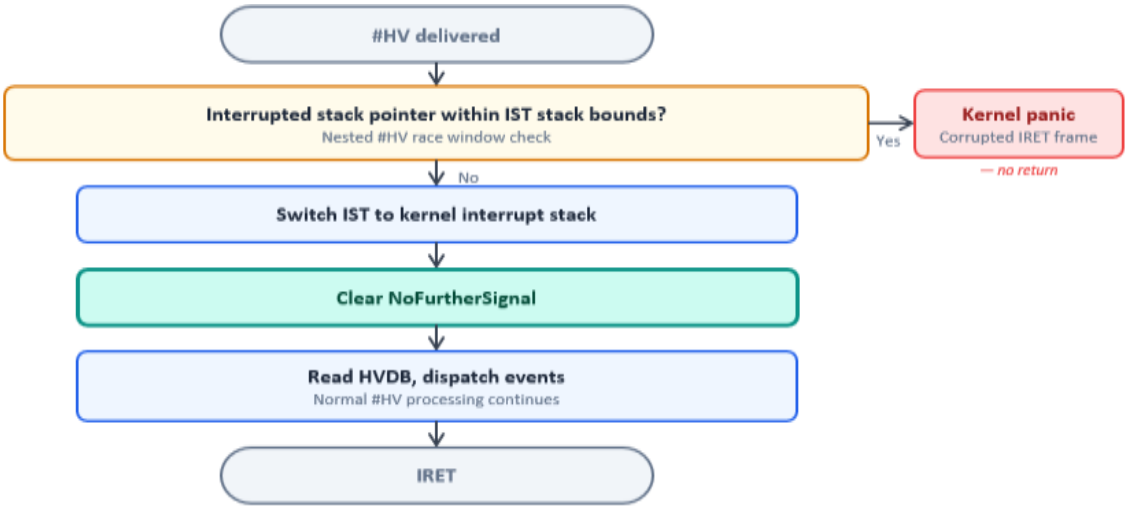

Figure 6: *Nested #HV Exception Detection Flow*

Table 2: *Comparison of AMD SEV-SNP Interrupt Injection Security Modes [4][5]*

Feature	Restricted Injection	Alternate Injection
Hypervisor vector injection	Disabled (all except #HV)	Permitted but guest-controlled
Doorbell mechanism	#HV (vector 28)	Guest-controlled APIC emulation
Para-virtualization required	Yes (HVDB shared memory)	Optional
APIC alignment	Via HVDB structure	Native APIC emulation
Interrupt shadow respect	No (#HV bypasses shadow)	Yes
Guest implementation complexity	Higher (custom dispatch)	Lower
Threat surface reduction	Maximum	Moderate

12. Conclusions

Confidential computing environments expose a fundamental mismatch that encrypted memory alone cannot resolve: the hypervisor is untrusted, yet it retains architectural control over interrupt delivery to the guest. The HECKLER attack establishes that this is not a theoretical concern; it is a practical exploit path that breaks confidential virtual machine guarantees without decrypting guest memory. Addressing it requires a hardware-enforced change to the hypervisor-guest interface, not a software patch within the guest.

AMD SEV-SNP Restricted Interrupt Injection provides that hardware-level response. By blocking all hypervisor-controlled interrupt and exception injection except for the #HV doorbell (vector 28), it reduces the hypervisor's influence over guest execution to a single, tightly scoped notification channel. The HVDB PendingEvent field carries the actual event information — NMI, machine check, or a specific vector — that the guest processes entirely on its own terms after receiving the doorbell.

Implementing this correctly across the full software stack introduces non-trivial challenges. The #HV exception bypasses interrupt shadows, requiring an explicit HVDB poll following STI before HLT in the idle path, and an HVDB check on every interrupt re-enable code path. The sysvec_table ELF section mechanism addresses system vector dispatch cleanly without hard-coding a per-vector list in the exception handling code. Preemption correctness at interrupt exit requires explicit attention because vectors dispatched through the HVDB if preempted can cause guest hangs and lockups due to missing device interrupts. The most dangerous implementation hazard remains the race window between #HV delivery and the guest handler's clearing of NoFurtherSignal, which can lead to IST stack corruption requiring a kernel panic. Resolving the nested detection reliability issue and settling the broader x86 exception handling integration questions are the primary prerequisites for upstream mainline acceptance. As confidential virtual machine deployments continue to grow in production cloud environments, closing the interrupt injection attack surface is a security necessity, not an optional enhancement

Author Statements:

- **Ethical approval:** The conducted research is not related to either human or animal use.
- **Conflict of interest:** The authors declare that they have no known competing financial

interests or personal relationships that could have appeared to influence the work reported in this paper

- **Acknowledgement:** The authors declare that they have nobody or no-company to acknowledge.
- **Author contributions:** The authors declare that they have equal right on this paper.
- **Funding information:** The authors declare that there is no funding to be acknowledged.

- **Data availability statement:** The data that support the findings of this study are available on request from the corresponding author. The data are not publicly available due to privacy or ethical restrictions.
- **Use of AI Tools:** The author(s) declare that no generative AI or AI-assisted technologies were used in the writing process of this manuscript.

References

[1] Masanori Misono, 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/epdf/10.1145/3700418

[2] Robert Buhren, et al., "One Glitch to Rule Them All: Fault Injection Attacks Against AMD's Secure Encrypted Virtualization," arXiv, 2021. Available: https://arxiv.org/pdf/2108.04575

[3] Mengyuan Li, et al., "CipherLeaks: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel," USENIX, 2021. Available: https://www.usenix.org/system/files/sec21-li-mengyuan.pdf

[4] Benedict Schlüter, et al., "HECKLER: Breaking Confidential VMs with Malicious Interrupts," USENIX, 2024. Available: https://ahoi-attacks.github.io/heckler/heckler_usenix24.pdf .

[5] AMD, "AMD Secure Encrypted Virtualization (SEV)." Available: https://www.amd.com/en/developer/sev.html

[6] Tianyu Lan, "[RFC PATCH V3 00/16] x86/hyperv/sev: Add AMD sev-snp enlightened guest support on hyperv," Linux Kernel Mailing List, 2023. Available: https://lkml.org/lkml/2023/1/21/302

[7] AMD, "SEV-ES Guest-Hypervisor Communication Block Standardization," AMD Technical Information Portal, 2025. Available: https://docs.amd.com/v/u/en-US/56421

[8] Ashish Kalra, "Interrupt Security for AMD SEV-SNP," Linux Plumbers Conference 2022, Confidential Computing Microconference, September 2022. Available: https://lpc.events/event/16/contributions/1321/

[9] Melody (Huibo) Wang, "Securing Interrupt Delivery for SEV-SNP Guests," KVM Forum, 2024, Available: https://pretalx.com/kvm-forum-2024/talk/GWF9UC/

[10] https://lore.kernel.org/all/20230606075026.GA905437@hirez.programming.kicks-ass.net/