HECKLER: Breaking Confidential VMs with Malicious Interrupts

Benedict Schlüter Supraja Sridhara Mark Kuhne Andrin Bertschi Shweta Shinde
ETH Zurich

Abstract

Hardware-based Trusted execution environments (TEEs) offer an isolation granularity of virtual machine abstraction. They provide confidential VMs (CVMs) that host security-sensitive code and data. AMD SEV-SNP and Intel TDX enable CVMs and are now available on popular cloud platforms. The untrusted hypervisor in these settings is in control of several resource management and configuration tasks, including interrupts. We present HECKLER, a new attack wherein the hypervisor injects malicious non-timer interrupts to break the confidentiality and integrity of CVMs. Our insight is to use the interrupt handlers that have global effects, such that we can manipulate a CVM's register states to change the data and control flow. With AMD SEV-SNP and Intel TDX, we demonstrate HECKLER on OpenSSH and sudo to bypass authentication. On AMD SEV-SNP we break execution integrity of C, Java, and Julia applications that perform statistical and text analysis. We explain the gaps in current defenses and outline guidelines for future defenses.

1 Introduction

Hardware-based trusted execution environments (TEEs) flip the conventional trust mode. They designate the cloud service provider and privileged software such as the hypervisor as untrusted entities. Recent TEEs lean towards a virtual machine abstraction for isolation granularity to provide *confidential VMs (CVMs)* that host security-sensitive code and data. AMD Secure Encrypted Virtualization-Secure Nested Paging (SEV-SNP) and Intel Trust Domain Extensions (TDX) are the two main extensions offered currently from hardware providers [2, 35], while Arm Confidential Computing Architecture (CCA) is anticipated to be in production in the future [7]. CVMs have received wide-scale adoption as cloud confidential computing hosted by major cloud providers such as Google Cloud, Microsoft Azure, Alibaba Cloud, and IBM Cloud [1, 9, 28, 29, 33].

[0]This is the author's version of the USENIX Security 2024 paper.

Hardware isolation and memory encryption in TEEs ensure the confidentiality and integrity of CVMs. However, despite being untrusted, the privileged software components such as the hypervisor remain responsible for resource allocation and virtualization management. As a result, it's crucial to reconsider how these untrusted components interact with the CVMs. We examine one such class of interfaces, namely the interrupt management that is under the hypervisor's control.

In this paper we present HECKLER, a new software-based attack that breaks the confidentiality and integrity of CVMs on AMD SEV-SNP and Intel TDX. HECKLER leverages the untrusted hypervisor's ability to inject controlled interrupts into the victim CVM at points of its choice. Since the CVMs run a full-fledged trusted operating system, it has valid handlers for several interrupts. Thus, unbeknownst to itself, the victim CVM starts executing the interrupt handlers corresponding to the interrupt injected by the hypervisor. However, unlike timer interrupts that are widely used for side-channel attacks [54–56, 62] because of their effects on cache and micro-architectural states, the CVM has handlers change registers and global state thus impacting the subsequent execution. Thus by simply injecting interrupts, the hypervisor is able to change the victim VM's data and control flow.

HECKLER is part of a larger family of attacks where an privileged attacker sends malicious notifications to the victim running in a TEE. We coin the term *Ahoi* to refer to this class of attacks.[1] Previous studies that exploit timer interrupts and page faults fall within the category of Ahoi attacks—they produce malicious interrupts to allow the attacker to monitor side-effects like cache and timing. Unlike these prior instances of Ahoi, HECKLER generates interrupts that go beyond side-effects; it targets explicit effect handler execution that directly modifies registers i.e., the CVM's global state.

Findings. We analyze the hypervisor's interrupt injection behavior on AMD SEV-SNP and Intel TDX. We find that both of them forward some, if not all, interrupts to the victim CVMs. Notably, both of them allow the attacker to inject int 0x80

[1]Ahoi is a signal word to call a ship or boat. It is also an anagram of Iago [17] with edit distance of one.

on cores executing CVMs. As an effect, the CVM executes the corresponding handler on behalf of a user-space process (e.g., statistical analysis, user authentication, daemons) that is currently executing on the core. Worse yet, as per the semantics of int 0x80, the handler treats the current register state set up by the process as syscall number (`rax`) and input args (`rbx`, `rcx`, `rdx`) for the system call. The guest kernel in the CVM, completely unaware that the hypervisor and not the process invoked this handler, executes the system call and returns the result of the system call back to the process by updating its `rax`. HECKLER abuses this behavior to operate as a gadget that changes the victim programs' `rax`. Further, AMD SEV-SNP allows the attacker to inject other interrupts such as int 0x0 and many more. Some of these interrupts are presented as signals to the user program. We find that the application-specific handler for these signals can have global side effects. For example, scientific calculations have handlers to convert the operands of faulting instructions (e.g., the denominator in a divz is set to a NaN) to capture specific notions (e.g., ∞, -∞). HECKLER changes this behavior into a gadget to convert particular program variables (e.g., to NaN) and continue execution. Lastly, we can chain gadgets by injecting multiple interrupts at selective locations of victim's execution to change more than one data and control flow.

Orchestrating HECKLER. End-to-end exploits built with HECKLER require injecting interrupts at targeted execution points in the victim programs to induce effects brought on by our gadgets. Specifically, we need to know the exact core on which the user program executes inside the CVM, the guest physical address of the point of gadget injection, and the moment when the program reaches the point of interest in its execution. For AMD SEV-SNP, we use several heuristics particular to our target programs based on the information we can glean about its execution (e.g., page faults). We maximize this by leveraging auxiliary information leaked by observable behavior despite encryption of CVM state (e.g., order of page accesses, execution in shared libraries) [45, 60].

Implications. We use the HECKLER gadgets to alter the data and control flow of five case-studies to break confidentiality and integrity of CVMs. First, on AMD SEV-SNP and Intel TDX, we bypass the authentication in OpenSSH and sudo, thus allowing the hypervisor to gain complete root access to the CVM. Next, we break execution integrity of AMD SEV-SNP by altering the results of statistical and text analysis in C, Java, and Julia. Lastly, we discuss the effectiveness of existing defenses offered by AMD SEV-SNP and show that they are insufficient. We develop kernel-patches for Intel TDX to stopgap the effects of our int 0x80 gadget.

Contributions. We make the following novel contributions:

- Novel Attack. We introduce HECKLER, a new attack wherein a hypervisor injects malicious interrupts to trigger handlers that change the data and control flow of victim CVMs.
- Gadgets & Chaining. We identify several crucial gadgets in prevalent services and workloads typically hosted in cloud-based CVMs. We invoke and chain these gadgets using custom orchestration techniques.
- Proof-of-concept Exploits. We show that our AMD SEV-SNP and Intel TDX exploits can bypass OpenSSH and sudo; our AMD SEV-SNP exploits can break statistical and text analysis for AMD SEV-SNP. This demonstrates that HECKLER breaks the integrity and confidentiality guarantees offered by these state-of-the-art TEEs.

Disclosure. We informed Intel and AMD about int 0x80 on 27 and 28 September 2023 respectively. We updated AMD on 14 October 2023 about our findings for other interrupts and our analysis of their defenses. HECKLER is tracked under two CVEs: CVE-2024-25744 for int 0x80 was mitigated with a kernel patch for SEV-SNP and TDX [51]. CVE-2024-25743 for other interrupts remains unmitigated for AMD on 6 March 2024 at the time of the writing.
HECKLER tooling and PoC exploits are open-source at: `https://ahoi-attacks.github.io/heckler`

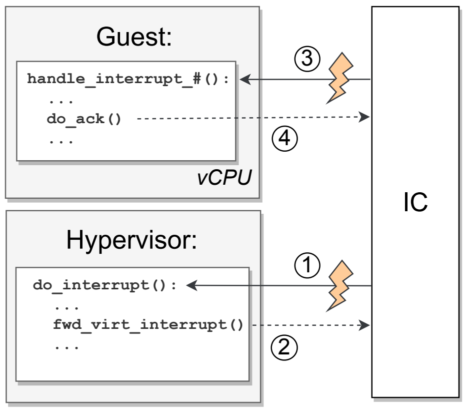
2 Overview

Hardware-based trusted execution environments provide an abstraction to execute code and data, such that its confidentiality and integrity is preserved even in the presence of privileged software. AMD Secure Encrypted Virtualization-Secure Nested Paging (AMD SEV-SNP), AMD Secure Encrypted Virtualization-Encrypted State (SEV-ES), and Intel Trust Domain Extensions (Intel TDX) provide a VM-level abstraction called confidential VMs (CVMs). For these TEE abstractions, the untrusted privileged hypervisor provisions the execution resources (e.g., CPU and memory) for VMs. The hardware ensures execution and memory isolation such that the untrusted software cannot compromise the CVM.

Notably, the untrusted hypervisor provides virtualization abstractions such as interrupt routing to CVMs. Thus, the attacker can abuse this interface to inject non-genuine (e.g., wrong interrupt number) and unexpected interrupts (e.g., at the wrong instruction), i.e., *malicious interrupts* into the target. Physical timers, the most widely-studied interrupt, have been shown to break the confidentiality of TEEs by amplifying side-channel attacks [54]. However, other interrupts have received little to no attention, because they are assumed to never explicitly affect the victim's execution beyond side-effects that can be gleaned via side-channels.

2.1 Interrupt Delivery to CVMs

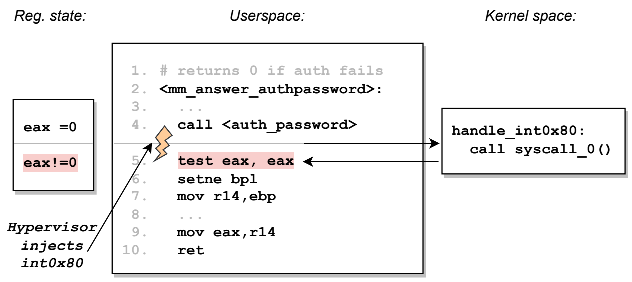
The guest OS executing inside the CVMs relies on interrupts for its operation (e.g., the Linux kernel requires timer interrupts for scheduling). Therefore, similar to traditional virtualization in non-confidential execution, the hypervisor has to virtualize the interrupt management and delivery to the



```
handle_interrupt_#():
  ...
  do_ack()
  ...
```

```
do_interrupt():
  ...
  fwd_virt_interrupt()
  ...
```

Figure 1: Virtualized interrupt for CVMs. Solid arrows (①, ③): asserted interrupt lines; dotted arrows (②, ④): memory-mapped write. The interrupt controller (IC) delivers a physical interrupt to the hypervisor ①. The hypervisor writes to a memory-mapped region of memory ② that emulates a virtual Interrupt Controller (vIC) for the vCPU to forward the virtual interrupt ③. The OS writes to a memory-mapped register in the vIC to acknowledge the interrupt ④.



```
1.  # returns 0 if auth fails
2.  <mm_answer_authpassword>:
3.    ...
4.    call <auth_password>
5.    test eax, eax
6.    setne bpl
7.    mov r14,ebp
8.    ...
9.    mov eax,r14
10.   ret
```

```
handle_int0x80:
  call syscall_0()
```

Figure 2: Inject int 0x80 for OpenSSH authentication. `mm_answer_authpassword` is invoked during ssh authentication. It returns 0 when authentication fails. A malicious int 0x80 triggers a call to the syscall 0 handler which sets `eax` to a non-zero value when `auth_password` returns, resulting in a successful authentication.

CVMs. To do so, the hypervisor hooks on all physical interrupts in the interrupt controller. Fig. 1 shows this mechanism at a high-level. For every interrupt, the hypervisor determines which VM the interrupt should be routed to, based on the CPU-to-vCPU mapping it maintains. Then, the hypervisor forwards the virtual interrupt to the vCPU. The guest OS of the CVM services the virtual interrupt. Finally, the guest OS acknowledges the interrupt in the interrupt service routine (ISR). The SEV and TDX hardware implementations and hardened guest Linux images (called enlightened guest OS) attempt to limit the interfaces that a CVM exposes to the untrusted hypervisor. However, our analysis shows that the hypervisor is still able to inject certain or all types of interrupts (see Sec. 3.2 for results).

2.2 HECKLER Attack

The hypervisor can arbitrarily inject interrupts to the CVMs. Such interrupts cause the guest OS to execute its interrupt service routines (ISRs) which can have side-effects that an attacker can exploit. For example, Linux uses interrupt number 0x80 for legacy 32-bit system calls on x86. Asserting interrupt 0x80 triggers the corresponding ISR. The ISR reads register `eax` and executes the system call. Further, it stores the result of the system call in the `eax` register. Note that this system call interface only updates the `eax` register. All other registers are restored by the kernel before returning to the user-space. Therefore, a malicious hypervisor can inject interrupt 0x80 and change the value stored in `eax`.

Attacking OpenSSH. We consider the OpenSSH application executing in user-space that runs a server process sshd. A CVM may host this process to allow trusted users to login and manage the confidential services. The SSH authentication routine in sshd invokes the `mm_answer_authpassword` function to check the user's credentials. If authentication fails, the function returns 0. The disassembly of this function shows that the return value of `auth_password` is stored in the `eax` register (see Lines 5-10 in Fig. 2). Further, the caller of `mm_answer_authpassword` checks if the return value is non-zero, and if so, allows the user to login. Consider the case where the attacker is trying to log into the CVM. Since it does not have the correct user credentials, the return value of `auth_password` and consequently `mm_answer_authpassword` will always be 0. However, if the attacker can change `eax` from zero to a non-zero value, then the caller of `mm_answer_authpassword` will let the attacker login, despite using wrong credentials. From a malicious hypervisor's perspective, if it injects an int 0x80 right after the return of `auth_password`, it can indeed change the value of `eax` before it is used by `mm_answer_authpassword`. Then, `mm_answer_authpassword` returns a non-zero value to the caller. The only thing that remains is to trigger int 0x80 such that it returns some other non-zero value in `eax`. If we take a closer look at the point of interrupt injection, `eax` is set to 0 by the function `auth_password`. If a malicious hypervisor injects an int 0x80 at this point, it triggers the execution of the handler on behalf of the sshd process. This results in executing system call number 0. In the Linux kernel, this corresponds to the restart system call which should always be invoked from within the kernel. Since we invoke it from the user-space, the kernel returns an `EINTR` error (−4, i.e., a non-zero value) in `eax`. In summary, the hypervisor uses the interrupt injection primitive to gain access to the CVM.

3 Malicious Interrupts

HECKLER leverages the effects of interrupt handlers on user-level applications, such that the attacker can alter their benign behavior to do its bidding. Apart from the int 0x80 handler we used in our motivating example, we systematically analyze other interrupts and their potential use in HECKLER.

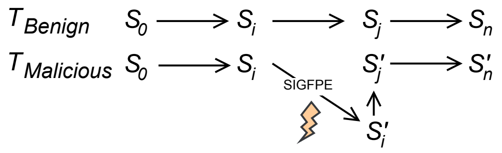
Threat Model. We operate in the standard threat model of confidential VMs provided by Intel TDX and AMD SEV-SNP. The untrusted hypervisor loads the CVM image in memory and controls the initial configurations. Remote attestation measures the CVM's initial memory before initiating the boot up. The software executing inside the CVM (guest OS, user applications, trusted modules for TEEs) is included in the TCB. As for configurations, the specifications for TDX and SEV-SNP outline certain initial state that the hypervisor has to setup (e.g., number of vCPUs, supported hardware features, memory size). The hardware checks this and only enters the CVM if the setup is correct. The hardware zeroes out certain values (e.g., certain general-purpose registers) before exiting the CVM. During execution, SEV-SNP and TDX encrypt and integrity protect the VM pages. Further, they protect register state and check some control and communications pages (e.g., Virtual Machine Control Block) that are shared with the hypervisor. The hypervisor is still expected to manage the CVMs by allocating physical pages and scheduling vCPUs. This includes injecting interrupts through different interfaces such that the CVM can continue to perform its tasks (e.g., virtio updates) and to notify the CVM about critical interrupts (e.g., virtual timers). We note that the specific protections of state shared between the hypervisor and the CVM vary for AMD SEV-ES, AMD SEV-SNP, and Intel TDX.

Scope. It has been shown that attacking AMD SEV-SNP is more challenging than attacking AMD SEV-ES [2]. This is mainly because SEV-ES does not provide integrity protection [59]. We leave attacks on AMD SEV-ES out of scope for this paper and instead focus on AMD SEV-SNP, with the expectation that if the attacks work on SEV-SNP, they will work on SEV-ES as well.

3.1 Trace-based Reasoning

Our goal is to identify interrupt handlers that, when executed at arbitrary points during a victim program execution, induce changes that impact the application. To capture this systematically, we introduce the notion of traces as defined below.

Trace. Consider a given program P and an input I that produces output O. Then program trace $T_P\ (I,O)$ is a sequence of states $S_1,\ldots,S_n$, where S_i is the program state that captures registers and virtual memory at time t_i. We capture explicit inputs as well as environment variables in I, and our state captures the register states and virtual memory of the process. Note that for a given P, I, O, its trace $T_P(I,O)$ is always deterministic.



$$\begin{tikzcd}
T_{Benign} & S_0 \arrow[r] & S_i \arrow[r] & S_j \arrow[r] & S_n \\
T_{Malicious} & S_0 \arrow[r] & S_i \arrow[dr, "\text{SIGFPE}"] & S'_j \arrow[r] & S'_n \\
 & & & S'_i \arrow[u] &
\end{tikzcd}$$

Figure 3: T_{Benign} and $T_{Malicious}$ represent traces for benign and malicious execution of P under input I. This leads to traces $S_0, S_i, S_j, \ldots, S_n$ and $S_0, S_i, S'_i, S'_j, \ldots, S'_n$ to produce outputs O and O' respectively. The attacker injects int 0x0 when P is in state S_i. This induces a state $S'_i : S_i[mem|mem[a] \mapsto 1]$, where the memory that holds variable a (i.e., $mem[a]$) is set to 1.

Explicit Effect Handlers. If a program P incurs a fault, interrupt, exception, or signal during its benign execution, then the system executes a handler either in the guest kernel or user space via an application-registered handler. The trace T captures it gracefully. For kernel handlers, they do not affect the program and hence are not accounted for in the trace. If the program executes a handler to terminate the program, that is captured by the state with the last state being program exit. More importantly, handlers that update the program state and continue execution are also captured by the notion of states. For example, consider a program with a custom floating point error handler that rounds off the value to the nearest integer, say 1. When the program executing on input I is in state S_i, it receives a SIGFPE for an operation on variable a that overflows. The program executes the handler that converts the problematic variable from a to a', thus changing the state from S_a to $S_{a'}$. We refer to such handlers, that effect a state change, as *explicit effect handlers*. But, if the program receives a timer interrupt then the program states stay unaffected.

Inducing Malicious State Transitions. The attacker has the capability to inject arbitrary interrupts into the CVM to invoke the corresponding handlers. For example, consider a benign execution of program P. At time t_i, it is in state S_i and changes to S_j at t_{i+1}. However, under a malicious execution, at time t_i, the attacker sends an int 0x0 to the VM's vCPU that is executing P who receives a SIGFPE. P's handler will execute at t_{i+1}, thus inducing a malicious state transition from S_i to S'_j. If we consider our above described handler that sets variable a to 1 on SIGFPE, the attacker has successfully managed to achieve a state transition from S_i to $S_{j'}$ where $mem[a] \mapsto 1$. Worse yet, since the handler resumes execution of the program, the attacker can time the interrupt such that the subsequent program logic uses the modified state variables, a in our example, thus leading to a different data or control flow and trace (see Fig. 3).

3.2 Detected Explicit Effect Handlers

We first analyze the hypervisor's ability to inject interrupts into the CVM, both on Intel TDX and AMD SEV-SNP. For this, we conduct a simple test on AMD SEV-SNP and Intel

TDX machines (see Sec. 8 for CPU and software details). We enumerate the interrupts from 0-255, the valid range of interrupts that a VM can receive. We inject them in our victim application executing inside the CVM via the hypervisor-provided interface. Then, we use 2 main observations regarding the x86 architecture to detect explicit event handlers for interrupts: (a) it has an explicit instruction that uses the interrupt number 128 (i.e., `int 0x80`) to perform legacy system calls, and (b) the Linux kernel maps interrupts to signals that are delivered to user-space applications. First, we test if int 0x80 is delivered to the CVM on both AMD SEV-SNP and Intel TDX machines when injected from the hypervisor. We see that the Linux kernel's int 0x80 handler always returns the result of the legacy system call in the `eax` register. Further, the different system call handlers conditionally read `ebx`, `ecx`, `edx`, `esi`, and `edi` registers.

Next, to detect if interrupts from the hypervisor are delivered as signals to the user application, we write a C application that registers handlers for all signals and waits in a busy loop. With this setup, we inject all interrupts to the CVM. For a given interrupt, if the CVM has a valid handler registered we can observe its impact, if any, on the application. We see that, for most interrupts, the Linux kernel uses a default handler that acknowledges the interrupt in the kernel and has no explicit effect on the application. Next, we summarize our specific findings for interrupts that impacted the applications.

SEV. Our experiments show that all interrupts were delivered to the CVM and handled by the guest Linux kernel. We observe that int 0x80 is delivered to the CVM and always noticeably impacts the user application. Further, the guest Linux kernel delivers 11 interrupts as a signal to the user-space application. Therefore, these 12 interrupts have explicit effects on the application.

TDX. All interrupts below 31 were dropped by the hardware and never even delivered to the guest VM. The only interrupt that was selectively allowed in this range was an NMI. For interrupts above 31 that reached the guest VM, only int 0x80 noticeably impacted the application.

4 HECKLER Gadgets

Next, we detail particular explicit effect handlers we detected and their exact effects. We refer to handler code as a HECKLER *gadget*, inspired by memory corruption attacks [32, 49].

4.1 Syscalls from Userspace

Linux uses int 0x80 for legacy system calls as shown in Fig. 4. Asserting int 0x80 triggers the corresponding ISR in the kernel space of the CVM. The ISR reads register `eax` and executes the corresponding system call. Further, it stores the result of the system call in the `eax` register. Therefore, a malicious hypervisor can inject int 0x80 and arbitrarily change the value stored in `eax` at any time (see Sec. 2.2). Further, based on the value in `eax` an attacker can use this interface to execute arbitrary system calls to attack the victim CVM (e.g., change page permissions, copy memory).[2]

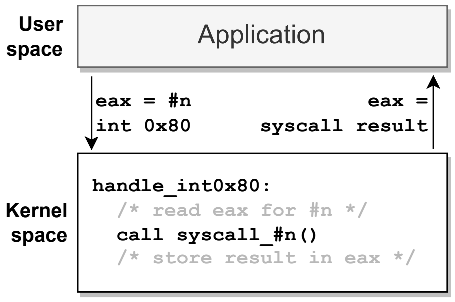


Figure 4: For int 0x80, the Linux kernel executes a system call corresponding to the number (#n) stored in `eax` by the application. When returning to the application, the kernel stores the result of the system call in the `eax` register.

Example. Consider an application that stores a secret on the stack (`ebp`-4) and accesses shared memory in the non-secure region (e.g., for communication with a non-secure VM). An attacker can use the int 0x80 to leak this secret by triggering the write system call. The Linux kernel executes the write system call in the int 0x80 handler (see Fig. 4) when `eax` is set to 4. Then, with the right parameters, such a call writes the secret to the hypervisor accessible shared memory. Specifically, the write system call takes 3 parameters; (fd) a file descriptor to write to in `ebx`, (buf) the address to read from in `ecx`, and (count) the number of bytes to read in `edx`. Therefore, we need an application that has a gadget as shown in the code snippet below:

```
1 %% Example: Leak secret
2 mov eax , 4             % write syscall number
3 mov ebx ...             % move shared memory fd
4 mov ecx, [ebp - 4]      % buf
5 mov edx, 8              % count
6 ...
```

Now, if the hypervisor injects int 0x80 on line 6, the kernel in the CVM will execute the `write` system call and leak the secret in `ecx` to the shared memory region in `ebx`. Note that, this program never executes the `int 0x80` instruction. So, the attacker's int 0x80 injection introduces a new state $S_{a'}$, where a' captures the result of executing the int 0x80 handler.

Scope of Syscalls & Registers. The attacker has a choice of invoking all syscalls by injecting int 0x80. As shown in the `write` syscall example, the attacker needs to have precise arguments in general purpose registers: `eax` should hold the correct syscall number and `ebx`, `ecx`, and `edx` should hold the correct syscall arguments. Then, depending on register states, an attacker can change `eax` and memory (arguments passed by reference) with syscalls. Identifying code locations

[2]int 0x80 instruction can be executed in 64 and 32-bit binaries.

in applications that satisfy this requirement, if not impossible, is challenging. To reduce the search space, we limit our analysis to syscalls that only depend on `eax`. We analyze 328 syscalls and find 40 syscalls only take `eax` as an argument and return `eax` i.e., independent of other registers (e.g., `getpid`, `getmask`, and other getter functions). `sigreturn` uses the current user stack to restore the process stack and can be used for code reuse attacks. Similarly, `setsid` creates a new session and process group and can be used to modify the value of `eax`. Next, we assess which of these syscall invocations are of interest to an attacker. It is unlikely that at an interesting point during a program's execution `eax` will hold the value of one of these syscalls. `eax` usually stores the return value of functions, so it often contains pointers and error values. While we cannot meaningfully change pointer values by invoking syscalls, we observe that we can change returned error codes as shown in Sec. 2.2. However, it raises the question: is such a primitive too weak to bring about any malicious effects?

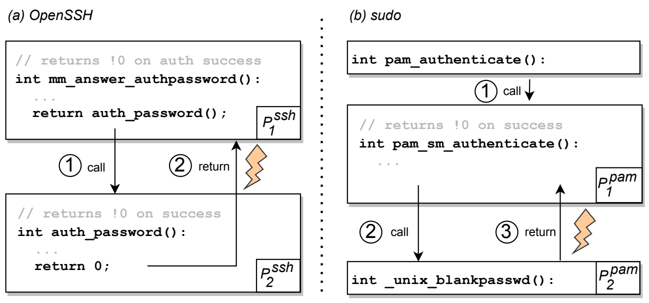
Altering `eax` to non-zero value. Often guard conditions check for non-zero values, which if maliciously altered, can induce data and control flow changes, as shown in Rowhammer [31] and non-control-data attacks [19]. Thus, we make the conscious choice to restrict ourselves to only use the int 0x80 gadget with `eax` equal to zero (e.g., change the return value from 0 to -4). Our case studies in Sec. 5.1 show that this is a powerful primitive in itself.

4.2 Signals to Userspace

x86 architecture maps floating point exceptions (e.g., divide by zero, overflow) to interrupts. When these interrupts occur, the Linux kernel handles them and raises a signal (SIGFPE) to the user-space application. Applications can register user-space handlers for these signals which are executed when the kernel raises the signal. We surveyed open-source applications that register explicit effect handlers for these signals.

int 0, 9, and 16: Floating Point Exceptions (FPEs). We found that of all the signals that the kernel raises because of interrupts, SIGFPE is the most interesting. Handlers for SIGFPE perform operations like setting variables to certain values (e.g., set the denominator to a non-zero value to handle a divide-by-zero), or skipping some operations (e.g., ignore faulting data that cause overflows). Therefore, a malicious hypervisor can change the control and data flow of applications by triggering interrupts that raise SIGFPE.

```
1  /* Example: SIGFPE handling */
2  double arr[] = {...}
3  double weights[] = {...}
4  double avg = 0
5  void handler() { /* compute non-weighted avg */ }
6  int compute_weighted() {
7    register(SIGFPE, handler)
8    avg = ...       /* compute weighted avg */
9    ...
10   return avg
11 }
```



```
// returns !0 on auth success
int mm_answer_authpassword():
  ...
  return auth_password();
```

```
// returns !0 on success
int auth_password():
  ...
  return 0;
```

```
int pam_authenticate():
```

```
// returns !0 on success
int pam_sm_authenticate():
  ...
```

```
int _unix_blankpasswd():
```

Figure 5: (a) P_1^{ssh} and P_2^{ssh}: gadget pages in the OpenSSH binary. (b) sudo P_1^{sudo} and P_2^{sudo}: gadget pages in the pam shared library used by the sudo binary.

For example, in the code snippet above, the application registers a SIGFPE handler on line 7. If the computation on line 8 causes a SIGFPE, the handler is executed. Then, the execution continues on line 5. An attacker can inject the divide-by-zero interrupt on line 9. This forces the application to always execute the handler changing its execution. As a result, the function always computes a non-weighted average compromising its integrity. Therefore, by injecting int 0x0 an attacker can introduce a new state $S_{a'}$ in the program's execution state (see Sec. 3.1).

Note that, unlike the attack using int 0x80 gadget which always invokes a syscall, the gadgets for FPE rely on application-specific handlers in user-space. Further, if the application does not register a handler, the kernel uses a default handler that terminates the process.

Other Signals. HECKLER can inject interrupts that generate SIGTRAP (1), SIGILL (6), SIGSEGV (4, 5, 10), and SIGBUS (11, 12, 17, 29) signals to userspace applications. However, we did not find applications that registered explicit effect handlers for these four signals. In the absence of handlers, POSIX standard states that userspace application must be terminated. Thus, these four signals are uninteresting for HECKLER.

Chaining Interrupts. A malicious hypervisor can chain multiple gadgets by injecting interrupts at different points during an application's execution. For example, consider an application that performs multiple authentication checks and registers a SIGFPE handler. To successfully authenticate, the attacker should compromise the data flow on lines 7 and 9. First, the attacker uses int 0x80 to bypass the check on line 7. Then, after line 8, the attacker triggers SIGFPE to change the value of `n` to 0. This changes the execution on line 9 passing the second check.

```
1  /* Example: Chaining interrupts */
2  int n = 1
3  void handler() { n = 0 }
4  int auth() { return 0 }
5  void grant_access() {
```

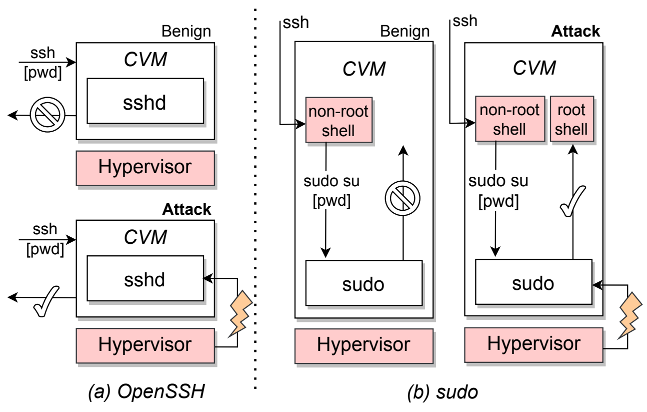
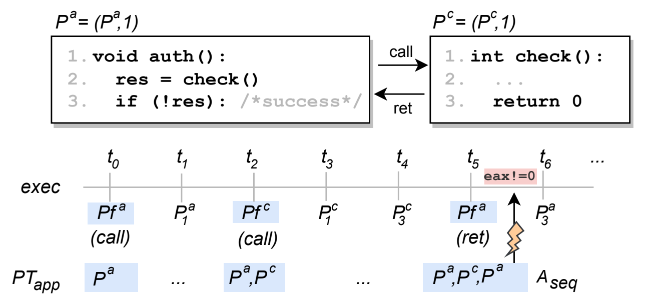


Figure 6: Red: attacker-controlled, lightning: int 0x80 injection, (a): Attack on OpenSSH, a malicious hypervisor successfully authenticates ssh on CVM with wrong pwd. (b): Attack on sudo, a malicious hypervisor with non-root shell on CVM escalates privilege to root shell.

```
register(SIGFPE, handler)
if (!auth()) { ... } /* deny access       */
n = second_auth()    /* !0 if auth fails */
if (!n) { ... }      /* auth. success    */
}
```



```
void auth():
  res = check()
  if (!res): /*success*/
```

```
int check():
  ...
  return 0
```

Figure 7: Attacker bypasses authentication check by injecting interrupt at time t_5 when detecting A_{seq}. Superscript for P: page id, subscript for P: line number in page, Pf^a: page fault in page with id a. For every page fault (blue), the GPA of the page is added to the PT_{app}.

5 Case Studies

We choose open-source applications to demonstrate the feasibility and impact of HECKLER. Then, we identify gadgets that allow a malicious hypervisor to mount HECKLER.

5.1 int 0x80

OpenSSH. It allows authenticated users to obtain a secure shell, use subsystems (e.g., sftp) for file transfers, and execute commands on remote servers. In our threat model, bypassing OpenSSH's authentication imparts the attackers with powerful capabilities to compromise the execution of a CVM. To this end, we demonstrate an attack on an OpenSSH server on the CVM using int 0x80 as shown in Fig. 6(a). We assume a malicious hypervisor that does not have the correct root password to authenticate a secure-shell on the CVM. As shown in Fig. 5(a), we identify a gadget where changing the return value to a non-zero number leads to successful authentication. Specifically, our attack sets the return value of `auth_password` to a non-zero value using int 0x80.

Sudo. Using sudo, an authorized non-root user can escalate privileges to a root user. We demonstrate an attack on sudo where an adversary with access to a non-root shell on the CVM can gain root access (see Fig. 6(b)). Specifically, the malicious hypervisor uses int 0x80 to bypass sudo's authentication mechanisms. By default, sudo is configured to use Privileged Access Management (PAM). With PAM enabled, sudo invokes a PAM module to authenticate the user. We identify a gadget in the PAM module with the `pam_sm_authenticate` function as shown in Fig. 5(b). This function first checks if the user has a blank password by calling the `_unix_blankpasswd` function. If this check succeeds, the PAM module does not prompt the user for a password. Instead, it considers the user to be correctly authenticated and returns to sudo. Therefore, we can use int 0x80 to change the return value of `_unix_blankpasswd` to a non-zero value leading to successful authentication. Applications that use the same PAM library to authenticate a user (e.g., doas [24]) are also susceptible to HECKLER in principle.

Chaining OpenSSH and Sudo. OpenSSH can be configured to prevent login as the root user. Similarly, sudo can be configured (using the sudoers file) to limit the users who can execute it. With this setup, our attack using OpenSSH can only get a non-root shell and our attack using sudo is not possible. However, we can chain the two attacks to get past these issues. Specifically, we attack OpenSSH to get a non-root shell of a user in the sudoers list. This ensures that the non-root user can execute sudo. Then, we use the attack on sudo to escalate the non-root shell to root privilege as explained above. Note that, to successfully chain the attacks, the malicious hypervisor injects int 0x80 two times.

5.2 Applications with SIGFPE

We first surveyed language support for signal handlers and then looked for existing applications that register SIGFPE handlers with explicit effects.

Java Statistical Analysis Tool. In Java, the runtime (Java virtual machine or JVM) registers a handler for SIGFPE in the user-space. When it receives SIGFPE from the kernel, the JVM translates it to a language-level ArithmeticException. The ArithmeticException is then caught and handled in the application. We analyze open-source Java applications that catch the ArithmeticException. We find an interesting gadget in the Java Statistical Analysis Tool (JSAT) [47]: a function that is used to add new data to a distribution that recalculates the mean and covariance as shown below.

```
1 /* Example: Disrupt Java with SIGFPE */
2 try {
3   Vec newMean = ...;          /* new mean */
4   Matrix covariance = ...;  /* new covariance */
5   this.mean = newMean;
6   setCovariance(covariance);
7 } catch(ArithmeticException ex)
8 { this.mean = origMean; }
```

During normal execution, if the function catches an ArithmeticException it uses the original mean, effectively ignoring the faulting data. On line 3, a malicious hypervisor can inject an interrupt that raises SIGFPE (e.g., int 0x0 for divide-by-zero) and consequently the ArithmeticException to the application. This will ensure that the function always ignores any new data added. This gadget is used to add new data to a multivariate normal distribution. Therefore, our attack can be used to bias the distribution to never accept new data.

TextAnalysis.jl in Julia. Like Java, the Julia runtime forwards signals for SIGFPE to a language-level DivideError. We find an interesting gadget in an established Julia package for text analysis (TextAnalysis.jl) [20]: an evaluation function to calculate a performance metric based on precision and recall scores (F-Score). If the function catches a DivideError, it reports the worst performance, indicating that a pair of text (e.g. machine and human-produced) are not similar.

```
# Example: Disrupt Julia with SIGFPE
function fmeasure_lcs(RLCS, PLCS, beta=1)
  try
    return ((1+beta^2) * RLCS * PLCS) /
           (RLCS + (beta^2) * PLCS)
  catch ex
    if ex isa DivideError
      return 0
    ...
```

We leverage this by maliciously raising SIGFPE and consequently DivideError to report the worst performance.

Hand-coded Multi-layer Perceptron (MLP) in C. We take an MLP implementation written in C [46] that uses `tanh` from the math library as an activation function as shown in the code snippet below. We manually add a SIGFPE handler, that recovers from overflows by setting the return value to 1 as shown in the code snippet below.

```
1 /* Example: Disrupt MLP with SIGFPE */
2 void tan_h_classify(...) {
3   output[0] = 1            /* bias term */
4   for (i = 0; i < n; i++)
5     if (sigsetjmp(buf, 1)) /* on SIGFPE */
6       output[i+1] = 1
7     else                   /* no overflow */
8       output[i+1] = tanh(input[i])
9 }
```

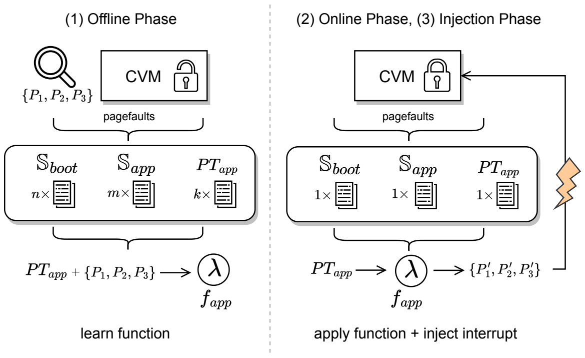

(1) Offline Phase
(2) Online Phase, (3) Injection Phase
$\{P_1, P_2, P_3\}$
CVM
pagefaults
$\mathbb{S}_{boot}$
$\mathbb{S}_{app}$
PT_{app}
$n\times$
$m\times$
$k\times$
$1\times$
$PT_{app} + \{P_1, P_2, P_3\} \longrightarrow \lambda$
f_{app}
$PT_{app} \longrightarrow \lambda \longrightarrow \{P'_1, P'_2, P'_3\}$
learn function
apply function + inject interrupt

Figure 8: Overview of profiling. During offline phase (1) we learn a function (f_{app}) that maps pagefault patterns to HECKLER gadgets. We repeatedly create $\mathbb{S}_{\text{boot}}$, $\mathbb{S}_{\text{app}}$ and PT_{app}. During online phase (2), we apply the function to monitor when the CVM reaches a point of interest in its execution, in the injection phase (3) we inject the interrupt. $\{P_1, P_2, P_3\}$: physical addresses of HECKLER gadget pages in PT_{app}, $\{P'_1, P'_2, P'_3\}$: predicted HECKLER gadget pages in PT_{app}.

We then maliciously invoke the handler to bias the model trained by the MLP. Specifically, on every call to the `tanh` function, we inject the interrupt to trigger SIGFPE (line 8 in the code snippet below). This ensures that the `tanh` function always returns 1. This allows us to bias the final confusion matrix for our test data set.

6 When & Where to Inject Interrupts?

For our attacks to succeed, it is crucial that we inject the interrupts at specific points during the application's execution. For example, to attack OpenSSH (see Sec. 5) we should inject the interrupt before the `mm_answer_authpassword` uses the value returned by `auth_password` as shown in Fig. 5(a). If we inject the interrupt at other points during the application's execution, the injection might not have the desired side-effect (e.g., changing `eax` before it is used), or crash the application. Next, if the CVM has multiple VM cores, we should ensure that our interrupt injection is targeted to the right core that executes the application logic with our gadget.

Overview. For SEV-SNP, the main challenge for a successful attack is identifying the physical pages of the functions of interest (i.e., `mm_answer_authpassword` and `auth_password` for OpenSSH). By marking the stage-2 page tables as non-executable we can trace the transition from `auth_password` to `mm_answer_authpassword`. This is possible because our two target functions are on two different physical pages. If this is not the case, i.e., both the functions are on the same page, we will have to resort to single-stepping this part of the execution [60]. However, for our builds of the target libraries, the functions are indeed on different pages. Therefore, once we observe a page fault on `auth_password` followed by a page fault on `mm_answer_authpassword`, we inject int 0x80. Specifically, every stage-2 page fault causes a VMEXIT transparent to the CVM. This allows HECKLER to inject an interrupt when the VM resumes, right before it executes the next guest instruction. In summary, the VM uses the attacker altered state on resumption from the page fault.

Attack Phases. HECKLER attack requires three phases: (a) an offline analysis to learn a function (f_{app}) that maps page fault patterns to HECKLER gadgets; (b) an online analysis to monitor when the CVM reaches a point of interest in its execution; and (c) injecting the interrupt (see Fig. 8). In the offline phase, we assume that the malicious hypervisor can create and run CVMs identical to the victim CVM multiple times to profile the behavior of the victim applications [63]. In this phase, the attacker controls both the malicious hypervisor and the CVM. In the online phase, when the attacker injects the interrupt, the attacker only controls the malicious hypervisor but can observe the CVM. Next, we detail how HECKLER uses the different phases to learn the f_{app} function.

Page Traces. To time and target our interrupts to the right core, we rely on the fact that our gadgets sequentially execute functions on different pages as shown in Figs. 5 and 7. Let us assume that the hypervisor can capture all pages with the cores they were used on during a CVM's execution (e.g., using page faults). Specifically, the hypervisor captures a list ($PT_{\texttt{vm}}$) of tuples with the guest physical addresses (GPAs) of pages and their corresponding cores $[(p^{\texttt{id}}, \texttt{core})]$. Using $PT_{\texttt{vm}}$, the hypervisor creates application-specific PT_{app} shown in Fig. 7 with all pages executed by the app in user-space.

The code snippets in Fig. 7 are analogous to the gadgets we detail for our case studies in Sec. 5. Here, the `auth` function on page P^{a} calls `check` on P^{c} and uses its return value. Therefore, the application's page trace (PT_{app}) always contains the sequence $A_{\texttt{seq}} = [P^{\text{a}}, P^{\text{c}}, P^{\text{a}}]$. To time the interrupt and target the right core, the hypervisor observes the application's access to these pages and waits to detect the sequence of pages. When the hypervisor detects the sequence $A_{\texttt{seq}}$ it injects the interrupt (e.g., int 0x80 to change the return value of `check`) before execution resumes on line 3 on P^{a} (P^{a}_{3} in Fig. 7). Note that PT_{app} is sufficient to target the interrupt to the right core as it contains information about the core on which the page was accessed by the application.

Application Trace (PT_{app}). To capture $PT_{\texttt{vm}}$, we assume that the hypervisor can induce page faults for all page accesses in the CVM. Creating PT_{app} from $PT_{\texttt{vm}}$ is not straightforward. First, the GPAs for the application's pages are different for every execution. Next, $PT_{\texttt{vm}}$ contains pages used by the kernel and all user-space applications. Further, the order in which the pages are accessed in the CVM is affected by the scheduling decisions in the Linux kernel. Given these challenges, we detail a method to reliably create PT_{app} and identify $A_{\texttt{seq}}$.

Capturing every page access for a CVM's execution ($PT_{\texttt{vm}}$) is expensive (many page faults for the same page) and generates an intractable trace. Instead, it is sufficient to start with a set of pages executed when the victim application executes on CVM ($\mathbb{S}_{\texttt{vm}}$). Note that, this only requires 1 page fault per page that is executed on the CVM. $\mathbb{S}_{\texttt{vm}}$ contains some pages executed by the kernel that need to be removed while creating PT_{app}. To identify the kernel's pages, we capture the set of pages accessed during kernel boot to form $\mathbb{S}_{\texttt{boot}}$. By removing all pages in $\mathbb{S}_{\texttt{boot}}$ from $\mathbb{S}_{\texttt{vm}}$ we get $\mathbb{S}_{\texttt{user}}$ i.e., $\mathbb{S}_{\texttt{user}} = \mathbb{S}_{\texttt{vm}} \setminus \mathbb{S}_{\texttt{boot}}$. Now, $\mathbb{S}_{\texttt{user}}$ contains all user-space pages executed in the CVM. To eliminate pages that do not belong to our victim application (e.g., OpenSSH, sudo) we execute the application multiple times (n) and compute $\mathbb{S}_{\texttt{user}_\texttt{i}}$ for every iteration (i). The set intersection of all $\mathbb{S}_{\texttt{user}_\texttt{i}}$ gives us $\mathbb{S}_{\text{app}}$ i.e., $\mathbb{S}_{\text{app}} = \bigcap_{i=1}^{n} \mathbb{S}_{\texttt{user}_\texttt{i}}$. By increasing the value of n, we can ensure that $\mathbb{S}_{\text{app}}$ only contains pages executed by our application.

Once we have correctly identified the application's pages, we can capture the pages in $\mathbb{S}_{\text{app}}$ every time they are executed to form PT_{app}. The guest physical addresses of the application's pages change when the VM is rebooted. Therefore, to reliably find our gadget pages (P^{a} and P^{c}) we should account for the changing GPAs. To capture this, we collect PT_{app} over multiple VM boots. Then, we analyze all PT_{app} to find a function f_{app} to get the gadget pages P^{a} and P^{c} in Fig. 7. Finally, we can use the gadget pages to identify $A_{\texttt{seq}}$ to correctly time and target the interrupt injection.

7 Implementation for AMD SEV-SNP

We describe our method to identify the guest physical address of the page that houses the gadgets of our interest.

7.1 Generating Page Traces

To generate the page trace for the application (PT_{app}), we need to induce page faults every time a page in the CVM is executed. In SEV-SNP the hypervisor can force page faults in the CVM [45, 60]. SEV-Step implements a mechanism that can be configured to induce page faults on all pages, or only on 1 page. We use the former configuration to create the unordered sets described in Sec. 6. Specifically, before booting the VM, we mark all pages as not-executable by setting the `nx` bit. Every time a page fault occurs, we note the page's GPA and core. Before the CVM resumes execution, KVM clears the `nx` bit. This ensures that only 1 page fault is triggered per page. To create the ordered list (PT_{app}) we use the mechanism from SEV-Step to mark single pages as not-executable. We start by setting all pages in $\mathbb{S}_{\text{app}}$ as not executable. Then, on every page fault, we note the GPA and core. Next, we set the `nx` bit of the page that generated the previous page fault. This mechanism ensures that every access to the application's pages generates a fault.

To implement this mechanism, we use the modified KVM from SEV-Step which exposes ioctls to the user-space [60]. These ioctls allow user-space applications to register and wait for events (e.g., page faults). We create CPython (409 LoC) and Python programs (2291 LoC) to interface with KVM to register and handle events for page faults.

Optimization. If we enable page faults for all application pages, the size of PT_{app} grows. We know that our gadget pages will only be accessed a few times during the application's execution. Therefore, we define an upper limit on the number of occurrences of a particular page in our tracing. This reduces PT_{app} size and optimizes the application execution time.

7.2 Boot Set ($\mathbb{S}_{\texttt{boot}}$) and Application Set ($\mathbb{S}_{\texttt{app}}$)

In both the offline and online phases of the attack, to create the application page trace (PT_{app}), we first need to form the boot set and application sets for each case study.

Boot set. We use the boot set to eliminate all pages executed by the kernel from PT_{app}. To create this set, we mark all pages as not-executable before booting the CVM. We capture all pages that generate page-faults while the Linux kernel boots on the CVM and add them to the boot set. We stop the capture once the CVM boot completes. This ensures that only kernel pages are captured in the boot set. Next, we explain how we create the application set for our end-to-end case studies.

OpenSSH. For password authentication, OpenSSH prompts the user for a password. If the authentication fails, it prompts the user again. The code gadgets we are interested in (see Sec. 5) are executed between these successive prompts. Therefore, to form the application set for OpenSSH, it is sufficient to capture the pages that are executed in this password prompt window. To do this, we implement a Go program as an ssh client with 70 LoC. For fine-grained control over the password authentication process, we modify Go's `crypto/ssh` standard library. We execute the ssh client from the untrusted host multiple times and capture the pages that are executed to form $\mathbb{S}_{\texttt{user}_\texttt{i}}$ and subsequently $\mathbb{S}_{\texttt{app}}$ as explained in Sec. 6.

Sudo. It uses PAM to perform password authentication by calling the `pam_unix` shared library which has our code gadget from Sec. 5. The Linux kernel executes shared libraries from the same physical addresses. Thus, for all executions of the shared library, the GPAs remain constant. We use this fact to create our application set for the sudo binary. Specifically, we write a C program to repeatedly access the pages with our code gadget i.e., P_1^{pam} and P_2^{pam} of the `pam_unix` library shown in Fig. 5(b) as shown below.

```c
/* Profiling shared libraries */
char* lib = "/usr/lib64/security/pam_unix.so";
unsigned long gad1, gad2; char* a; int fd;
fd = open(lib, O_RDONLY);
a  = mmap(0, 0x4000000, (PROT_READ | PROT_EXEC),
          MAP_SHARED, fd, 0);
gad1 = a + 0xCAFEBABE; /* ret gadget 1 */
gad2 = a + 0xCAFED00D; /* ret gadget 2 */
while (1) {
  asm volatile("mfence":: :"memory");
  asm volatile("push %0" : : "r" (&&jmp1));
  asm volatile("jmp *%0" : : "r" (gad1));
jmp1:
  asm volatile("mfence":: :"memory");
  asm volatile("push %0" : : "r" (&&jmp2));
  asm volatile("jmp *%0" : : "r" (gad2));
jmp2:
}
```

We execute this C program several times on the CVM and capture the pages that are executed to create $\mathbb{S}_{\texttt{app}}$. Note that, the addresses for `pam_unix` are fixed, so we do not need to execute sudo application during this phase to form $\mathbb{S}_{\texttt{app}}$.

MLP. We use an open-source implementation of MLP written in C [46] and add a SIGFPE handler to its tanh activation function implementation. Every call to this activation function results in multiple calls to the `tanh` function in the math shared library as shown in Sec. 5.2. We implement an interface in the CVM to allow users to start and stop the training of the MLP. The training process takes a long time. Therefore, capturing all pages executed during the training results in a very large unusable set. So, we capture the pages multiple times during the training in small windows of 1 second. To form $\mathbb{S}_{\texttt{user}_\texttt{i}}$ we compute an intersection over all pages from the windows to create $\mathbb{S}_{\texttt{app}}$ (see Sec. 6).

7.3 Finding a Function

In the offline phase, we use the application page trace (PT_{app}) to define a function (f_{app}) to predict the physical addresses of our gadget pages. Next, we explain our how to create f_{app} for each of the end-to-end case studies.

OpenSSH. We analyze the page traces (PT_{app}) from multiple CVM boots. We first create 2 sets with potential candidates for gadget pages (P_1^{ssh} and P_2^{ssh}). Using the page traces across multiple CVM boots we profile the OpenSSH behavior during password authentication and define a frequency interval $[9,11]$. We define all pages that appear in PT_{app} with frequencies in this interval as candidate pages for P_1^{ssh}. Similarly, we define a frequency interval $[5,7]$ to find the candidate pages for P_2^{ssh}. Note that, for these VM boots, the attacker also controls the CVM. During the attack, we first form the candidate sets using the values for the frequency intervals we define above. Then, to further eliminate pages from the candidate sets and form page tuples (P_1^{ssh}, P_2^{ssh}), we use the fact that the gadget pages must appear in a particular sequence ($A_{\texttt{seq}}$) in all page traces.

Sudo. Unlike OpenSSH identifying the gadget pages is straightforward for sudo. First, in this setting the attacker already controls a non-root shell on the CVM. Then, our gadget pages lie in the `pam_unix` shared library whose GPAs do not change across multiple runs. The C program's loop uses the virtual addresses of the gadget pages to repeatedly access them (see Sec. 7.2). To determine the GPAs of these gadget pages our function (f_{app}) just picks the 2 pages that occur the most number of times in PT_{app}. Then, it uses the order of accesses to determine the GPAs for the tuple (P_1^{pam},P_2^{pam}).

MLP. We identify 3 gadget pages for MLP: P_1^{mlp} the page that contains the calling function of tanh, P_2^{mlp} the tanh shared library, and P_3^{mlp} a page in the shared library executed by the tanh function. While the first page is backed by different GPAs for each application execution, the second and third page in the shared library remain constant. On investigating the application trace PT_{app}, we identify a sequence of length 9 with the gadget pages that occur with high frequency. We use this to define the function (f_{app}) the finds candidates for tuples of gadget pages (P_1^{mlp}, P_2^{mlp}, P_3^{mlp}).

Effect of Imperfect Page Analysis. Our AMD SEV-SNP analysis is intentionally specific to our observations per application. It is not designed for other gadgets that may not conform to such behaviour, and depending on the gadget, may need instruction single-stepping [60, 63]. Injecting int 0x80 on the wrong page either has no observable effect or crashes OpenSSH which is restarted by the daemon.

Remark on Intel TDX. We need a primitive to know when the A_{seq} occurs during the application's execution. Since our goal is not to build single-stepping and analysis techniques demonstrated for AMD SEV [60], we do not investigate using page faults, cache side-channels, or timer interrupts, to achieve this primitive. We had limited access to the TDX machine to fully experiment. To make the best use of our limited access and to demonstrate our attack, we use a busy loop in functions `mm_answer_authpassword` and `pam_sm_authenticate` for OpenSSH and sudo respectively. Future works can address this using advances in TDX-step [36].

8 Proof-of-concept Exploits

To demonstrate HECKLER on SEV-SNP and Intel TDX, we use the latest production systems and setups recommended by AMD and Intel respectively.

SEV-SNP. We demonstrate our attacks on an EPYC 9124 with Zen 4 SEV-SNP enabled workstation with 16 cores and 192 GB RAM. We boot the host Linux kernel with patches from SEV-Step that introduce the page-fault interfaces in KVM [60]. This kernel also contains the patches for KVM to launch and manage SEV-SNP VMs. Further, we use the same QEMU version 6.1.50 and SEV-SNP VM Linux kernel v5.19.0 to perform our experiments.

TDX. We had early access to TDX in September 2023. We confirm our attacks on a pre-production Intel Xeon Platinum processor with TDX support with 112 cores and 256GiB of RAM. We follow the official Intel documentation and boot a patched Linux kernel v5.19.17 on both the guest and the host. Further, we use modified QEMU v7.0.50 provided by Intel to create TDX VMs. In March 2024, we tested int 0x80 injection on a production Intel Xeon Gold 6526Y processor with TDX support and confirmed that it is vulnerable to HECKLER.

8.1 Injecting Interrupts

While both SEV-SNP and TDX allow the hypervisor to inject interrupts to the CVMs, the method to inject the interrupt is different for each of them. Below, we outline the mechanisms we use to inject interrupts for HECKLER.

SEV-SNP. AMD virtual machine extensions expose various interfaces that a hypervisor can use to inject interrupts into a VM. In our implementation, we use the event injection interface to inject int 0x80 and int 0x0 (see Appx. A for other interfaces). For this, we use the event injection field (`VMCB.EventInj`) that is accessible to the hypervisor in the Virtual Machine Control Block (VMCB) of the SEV VM. We implement a kernel module with 150 LoC, which interfaces with KVM to write the interrupt number to be injected in the respective VMCB field. When the SEV VM resumes execution, this method ensures that the interrupt is always raised before the next instruction is executed [4]. This makes our injection deterministic, thus ensuring HECKLER does not need to time the interrupt injection between a window of a few CPU cycles as already explained in Sec. 6. The KVM implementation expects acknowledgments from the guest kernel in the VM for most external interrupts. During normal operation, for all external interrupts, the guest Linux kernel writes the acknowledgments to a register in this virtual APIC page. We observe that the int 0x80 handler in the guest Linux kernel does not acknowledge the interrupt because it does not expect these interrupts to be injected externally. Without such acknowledgment, KVM will not inject certain interrupts which can lead to unexpected behavior (e.g., frozen terminal because of tty interrupts). To remedy this, we perform the virtual APIC page register write from the host.

TDX. We implement a kernel module in 150 LoC to inject interrupts into the TDX VM. Our host module uses kernel hooks to call a function in KVM that is used to deliver int 0x80 interrupts to TDX VMs. Unlike SEV-SNP, TDX does not expose the Virtual Machine Control Structure (VMCS) or the virtual APIC pages to the untrusted hypervisor. Instead, it expects the hypervisor to write into a Posted Interrupt Request (PIR) buffer. This buffer is used by hardware to inject interrupts into TDX VMs through the virtual APIC [34]. We inject two interrupts into two different cores of the CVM with this mechanism, one to gain login into the TDX VM with OpenSSH and another to get root access with sudo. During these two injects, the guest kernel does not acknowledge the interrupts. While this does not stop our attacks, it does leave the APIC with an elevated Task-Priority-Register (TPR), blocking all lower-priority interrupts on the affected vCPU. This may break CVM functionality that is noticeable by the user. To evade such detection, we implement a guest kernel module (`kern_ack`) that resets the APIC state. We inject this kernel module into the TDX VM as the last part of our attack

Table 1: Cardinality of the sets ($\mathbb{S}_{\texttt{boot}}$, $\mathbb{S}_{\texttt{user}}$, $\mathbb{S}_{\text{app}}$) and traces (PT_{app}) to find gadget pages. VMb: VM boot, max captures: maximum number of times we capture a page in PT_{app} where 0 indicates that we always capture.

	VMb	traces per VMb	$\lvert\mathbb{S}_{\texttt{boot}}\rvert$	$\lvert\mathbb{S}_{\texttt{user}}\rvert$	$\lvert\mathbb{S}_{\text{app}}\rvert$	$\lvert PT_{\text{app}}\rvert$	max captures
Openssh	392	10	82433	666	236	22440	200
Sudo	9	1	82431	259	6	199013	0
MLP	6	20	82378	718	255	32832	200

after gaining root access.

8.2 OpenSSH

We do our attack on an OpenSSH binary v9.4.P1+ with PAM disabled. We run an ssh client on the same host as the CVM.

SEV-SNP. In the offline phase, we profile the behavior of OpenSSH over 392 VM boots. For every VM boot, we collect 10 user sets ($\mathbb{S}_{\texttt{user}}$). Using these, we create the application set ($\mathbb{S}_{\text{app}}$) and page traces (PT_{app}) of sizes shown in Tab. 1.

In the online phase, to profile and attack the application, we set up the VM to generate page faults during boot and during application execution. With the page fault mechanism enabled, we observe an overhead of 11.41 seconds to boot as compared to 10.01 seconds without the page faults (+14%). Creating the application set in the password prompt window takes 32.9 ms to execute compared to 14.2 ms without page faults (+131%). Creating a page trace PT_{app} for the password prompt window takes 773.9 ms to execute (+5332%). In Tab. 5, we summarize page fault overheads for all the case studies. For OpenSSH and MLP, we cap the number of times each page is captured to 200 (see Sec. 7.1). Tab. 2 shows the number of times our gadget pages appear on average in PT_{app}. From our profiling, we report that on average the size of our candidate set for $P_1^{\texttt{ssh}}$ is 4.81, and $P_2^{\texttt{ssh}}$ is 8.52 before considering the attack sequence. Finally, when we account for the sequence ($A_{\texttt{seq}}$) in PT_{app}, on average we get 2.24 ($P_1^{\texttt{ssh}}$, $P_2^{\texttt{ssh}}$) tuples. With this, we get an average probability of success of 44.71% with 1 interrupt injection.

TDX. As explained in Sec. 7, we implement busy loops in our gadget page with the function `mm_answer_authpassword`. This eliminates the need to time our interrupt injection. We use our kernel module in the host to inject int 0x80. The interrupt-delivering function takes 1835 cycles for every injection. Further, once the attack succeeds, we insert a kernel module in the TDX VM to reset the APIC. The reset takes about 3092 cycles on average. With this setup, we report that our attack always succeeds.

Table 2: Number of times (in % and absolute) the gadget pages for the different applications appear in the application's page trace (PT_{app}). Page trace size ($\lvert PT_{\text{app}}\rvert$) as detailed in Tab. 1. The gadget page P_3 is not applicable to OpenSSH and sudo as they only have 2 gadget pages.

	OpenSSH		Sudo		MLP	
	%	*abs.*	%	*abs.*	%	*abs.*
P_1	0.044	9.8	25.3	50348.6	0.6	200
P_2	0.026	5.9	24.6	49051.0	0.6	200
P_3	-	-	-	-	0.4	133

Table 3: Overheads for boot trace, application set ($\mathbb{S}_{\text{app}}$), and page trace (PT_{app}) w.r.t. execution without page faults in %.

App	boot	$\mathbb{S}_{\text{app}}$	PT_{app}
OpenSSH	14	131	5332
Sudo	37	3	602
MLP	38	3	81

8.3 Sudo

We use an unmodified sudo binary in the Ubuntu 23.10 distribution with default configurations.

SEV-SNP. We perform our offline profiling over 9 VM boots and create the application set ($\mathbb{S}_{\text{app}}$). To create an application trace (PT_{app}), we execute the loop that repeatedly accesses the shared library pages as explained in Sec. 7.2. With this, we see that our gadget pages are in $\mathbb{S}_{\text{app}}$ and up to 49.9% of the final trace (PT_{app}) as shown in Tab. 2. For the attack, we execute `sudo su` from the non-root shell on the CVM. Our looping technique to access the pages of the `pam_unix` shared library ensures that we reliably find the GPAs of the gadget pages and our attack always succeeds with 1 injection.

TDX. To perform the sudo attack, we implement a busy-loop in the `pam_sm_authenticate` function that waits for int 0x80. Therefore, our attack always succeeds and we escalate to a root shell on the TDX VM. To acknowledge the interrupt, we insert the kernel module as with the OpenSSH attack.

Chaining OpenSSH and Sudo. We chain our attacks on OpenSSH and sudo to get around the problems discussed in Sec. 5.1 by injecting int 0x80 two times.

8.4 FPE

We use three different applications to demonstrate HECKLER with interrupts that raise SIGFPE.

MLP. To profile the MLP application offline, we record all pages that are executed in one second windows. We capture the pages over 6 VM boots and collect 10 user sets ($\mathbb{S}_{\texttt{user}}$) per boot. We observe average application set ($\mathbb{S}_{\text{app}}$) sizes of 255 pages. We create 20 traces (PT_{app}) per VM boot. Our gadget pages P_1, P_2, P_3 occur 1.6% in PT_{app}. Using our function from Sec. 7.3 on average we find 20.5 tuples for the gadget pages. We see an average probability of success of 41.6%.

JSAT and TextAnalysis.jl. Our method in Sec. 7 requires more engineering to Java and Julia applications with runtimes (e.g., OpenJDK and Julia Runtime). As opposed to ahead-of-time compiled programs, finding the gadget pages for interpreted programs requires profiling the dynamic behavior of the runtime's code cache and hot paths. For simplicity, we run our programs with a busy loop in the gadget function instead of profiling it. We run the JVM in interpreter mode where SIGFPE is translated to a language-level ArithmeticException.

For JSAT, we run the `LVQLLC` test to create a multivariate normal distribution from the JSAT repository [47]. With our attack, we need to inject 240 interrupts while the application executes to change all return values of our gadget function (Sec. 5.2). Similarly, for TextAnalysis.jl, we run the `Evaluation Metrics` test suite from the TextAnalysis.jl repository [20] and need to inject 2 interrupts.

8.5 End-to-End Attack Cost

HECKLER is performed in 3 different phases as shown in Fig. 8. To understand the end-to-end cost of our attack, we explain the overheads for each of these phases.

Offline Phase. During the offline phase, we get multiple traces as summarized in Tab. 1. In this case, the overheads of tracing slowdown the function generation described in Sec. 8.1. While this can be further optimized, we did not put efforts in such optimizations since this is a preparatory step before the victim runs its VM.

Online Phase. HECKLER also enables page fault tracing in the online phase, i.e. when the victim starts interacting with the VM. Tab. 5 shows a timing analysis to generate boot trace, application set ($\mathbb{S}_{\text{app}}$), and page trace (PT_{app}) during online phase, when compared to the execution of the CVM without page fault tracing. HECKLER causes some slowdown but it does not impact the victim's usability or result in detection. This is because we can potentially perform the tracing and the injection after attestation, but during the CVM provisioning which can take several minutes even in a benign setting. Thus, HECKLER attack happens before the user gets access to the VM, so it will not notice the lag. In cases where this is not possible, we can further cap the number of page faults for any given page (max capture in Tab. 1) to reduce the lag, as well as repeat the set intersection for $\mathbb{S}_{\text{app}}$ to decrease the size of the resulting trace (PT_{app}).

Interrupt Injection. As already layed out in Sec. 6 and 8.1, we use SEV's event injection interface (`VMCB.EventInj`) to inject interrupts. This method ensures that the hardware raises the interrupt to the guest kernel before the VM executes subsequent instructions.

9 Ineffectiveness of Current Defenses on AMD

AMD SEV-SNP outlines two optional modes called Restricted and Alternate injections. They are designed to restrict the hypervisor's interrupt and exception interface to the CVM. We explain the changes brought by these modes and then analyze their effectiveness against HECKLER.

AMD SEV-SNP Restricted Injection. The hypervisor sets bit 3 in the `SEV_FEATURES` register per vCPU of the CVM to enable or disable this mode. When disabled, the hypervisor continues to use the legacy interfaces to inject *all* interrupts. When enabled, the hypervisor is still able to partially use the legacy interface (see Fig. 9(b)). Specifically, it can inject *only* #HV interrupt—a new interrupt with number 28 introduced for this mode. Further, the hypervisor cannot use the virtual interrupt queuing. Instead, the hypervisor and the CVM setup a shared memory region to house the event queue. The hypervisor uses the #HV as a doorbell to inform the CVM about a new interrupt in the queue. The #HV handler in the CVM then accesses the queue, retrieves the actual interrupt number (e.g., int 0x80) and then handles the queued-up interrupt.

AMD SEV-SNP Alternate Injection. The restricted mode described above introduces a new interface for the hypervisor. More importantly, it breaks compatibility with existing guest OS implementations, requires enlightening the guest OS, and hinders lift-and-shift. To limit this effect, the alternate injection mode offers the traditional interrupt interface, but with a caveat. First, one of the vCPUs in the CVM runs at a special privileged level called VMPL0 while the rest of the vCPUs execute at non-privileged levels VMPL1-VMPL3. Second, all the vCPUs that execute the guest OS run in VMPL1-3 and enable alternate mode. With this combination, they continue to see a traditional interrupt interface both for configuring and receiving interrupts. Third, the vCPU that executes in VMPL0 acts as a trusted bridge between VMPL1-3 CPUs and the hypervisor. It also performs security and virtualization tasks within the CVM. Since this is a new piece of code that is introduced, it can very well be in charge of presenting legacy interrupt interfaces for the CVM. This is why, it runs in restricted mode, creates a shared page, handles #HV, converts them to virtual interrupts, and delivers them to the guest OS. Fig. 9(b) shows the setup where both the modes are enabled on CVM cores. Note that both of these modes change the delivery mechanism and interfaces that the hypervisor needs to use to deliver the interrupts to the guest OS, it does not fundamentally introduce any filtering or dropping rules.

Hardware Availability. Our machine supports both of these modes in the hardware and we were able to test that the newly introduced MSRs are operational.

Impact on HECKLER. The main goal of these new modes is to allow the CVMs to continue with their assumed behavior about the interrupt interface provided by the hypervisor for compatibility. The AMD documentation alludes that this mode can address potential misbehavior by the hypervisor

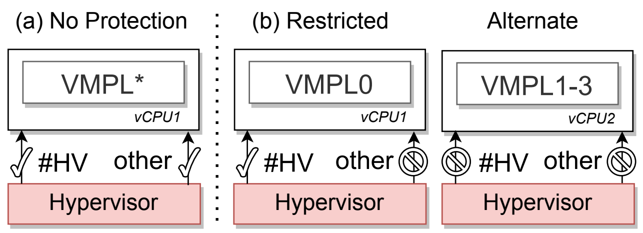


Figure 9: (a): Without any defense enabled, the hypervisor can inject all interrupts into the SEV VM. (b) Restricted mode: enabled on vCPU that runs VMPL0 and the hypervisor can only inject #HV. Alternate mode: enabled on all non-VMPL0 cores and the hypervisor cannot inject any interrupts.

that breaks the OS assumptions (e.g., inject interrupts while TPR is elevated). But, it does not discuss any mandatory security checks or filtering rules. The pseudo-code provided by AMD does not do any security checks. More importantly, the software support for restricted mode does not perform any checks or filters [5]. Thus, even with restricted mode and #HV, HECKLER attacks are possible. The main reason is that the new mode changes the delivery mechanism but does not stop or filter the delivery of interrupts. As for alternate mode, the current hypervisor and guest OS implementations do not support alternate mode. When implemented, it remains to see if it filters any interrupts, even though such filtering is not specified by AMD.

10 Potential Defenses

Given that existing mechanisms for interrupt security are insufficient, we develop software methods (where possible) and propose hardware mechanisms to mitigate HECKLER.

10.1 Software Mitigations

The main ingredient for HECKLER is the ability of the hypervisor to *externally* inject malicious interrupts into a vCPU executing the CVM.

Detecting External Interrupts. One seemingly straightforward fix is to address the symptom of external interrupts in software. For example, the guest kernel can be patched to detect and selectively allow external interrupts. Interrupts such as int 0x80, should perhaps never arise externally and can be dropped. However, we did find use-cases where this is a desired behavior [10, 12], after all, it is part of the x86-64 ISA standard. To disable external delivery of int 0x80 in the guest, the kernel's handler can check if the instruction came from the user-space or from an external source by examining the previously executed instruction referenced by the RIP on the context stack or by checking the APIC page. For other interrupts such as int 0x0, determining if it is a genuine or a malicious interrupt is unclear because it would require analyzing and interpreting the executed user space code.

Disabling Interrupt Handlers. Another approach is to disable vulnerable interrupts by not registering handlers for them in the guest OS. This works for int 0x80 if the kernel is recompiled without the configuration flag `CONFIG_IA32_EMULATION`, which disables IA32 emulation. However, again, this does not generalize beyond int 0x80; and even then may break compatibility with legacy code that relies on int 0x80 behavior. We survey 5 flavors of GCP- and Azure-recommended CVM images (Redhat, Fedora, CentOS, Ubuntu), standalone Debian-rolling, and ArchLinux. All of them have kernels with 32-bit support compiled in at the time of writing. It is required to ensure maximal compatibility and guarantee legacy support. Linux 6.6. onwards it is possible to dynamically disable `CONFIG_IA32_EMULATION` at boot time.

TDX Implementation. For Intel TDX, we implemented both software-based defenses. First, we compiled the Linux kernel with the `CONFIG_IA32_EMULATION` flag disabled in the configuration. Second, detecting if an int 0x80 came externally required a patch of 14 LoC where we checked the APIC page bit. Since this is the only way to inject external interrupts on TDX, this patch was sufficient. When running a user application in the guest OS that did a genuine int 0x80, servicing it on our patched kernel resulted in an overhead of 460 cycles when compared to a vanilla kernel. We tested HECKLER on both these patched versions on Intel TDX and confirmed that the attack does not go through.

We co-operated with Intel and Linux kernel developers to apply the second approach that detects external interrupts to protect TDX VMs against HECKLER. By default, TDX VMs execute with the IA32 emulation enabled and a patch to the guest kernel checks the APIC page bit to stop external injections of int 0x80 [26].

SEV-SNP Implementation. We attempted to implement the defense of detecting interrupts by examining the APIC page. On AMD, the hypervisor can inject external interrupts asynchronously via the APIC page (same as TDX). Similar to TDX, we implemented the virtual APIC check for AMD SEV-SNP with 14 LoC. We observed an overhead of 10182 cycles compared to the original unpatched execution of a binary that genuinely performs int 0x80. However, this is insufficient on AMD because the hypervisor can also inject interrupts via the VMCB registers which are handled when the VM resumes execution (Sec. 8.1). To stop this attack surface we used the defense strategy of detecting external interrupts by examining the last instruction that the user-space application executed. This requires examining the memory referenced by the `rip` on the saved context stack to check if the program executed an `int` instruction with `0x80` as a parameter. This requires disassembling the `rip` in reverse for 2-bytes (since int 0x80 results in a 2-byte opcode), where we inevitably run into classic problems stemming from variable length instructions. Determining if the user-code indeed performed int 0x80 or some other stream of instructions and parameters that result in the same opcodes is undecidable. Thus our patches provide incomplete protection.

To defend against HECKLER, the Linux kernel introduced a patch that disables IA32 emulation by default for SEV VMs [51]. While this software patch stops HECKLER's int 0x80 attacks, it is ineffective against attacks from interrupts (e.g., int 0x0) which are converted to signals. Detecting external injections of these interrupts using the `rip` is not feasible. To decide if an interrupt is legitimate, the guest kernel would need to parse the whole instruction (opcode and all arguments), and in some cases emulate the instruction. For example, to check if the application legitimately caused an overflow resulting in an int 0x10, the guest kernel would need to emulate the full arithmetic operation to reliably determine overflow conditions. Protecting against these interrupts would require hardware-based filtering techniques in Sec. 10.2.

Using Restricted & Alternate Injection. We attempted to leverage the restricted and alternate mode to implement a software defense that adds the missing checks at least for int 0x0 and int 0x80. However, due to lack of software support for these modes in the hypervisor and the guest OS, we were unable to prototype these checks. One can implement standalone restricted injection directly in the host Linux kernel. However, there are no open-source implementations that we can test. Further, an initial patchset proposed by Microsoft received strong pushback by the Linux community [38]. The main criticism for rejecting the patches was that a nested #HV might corrupt the stack and hardware cannot protect against this race condition. One can also implement restricted and alternate mode in combination, which necessitates nested virtualization to take advantage of VMPLs. Prior works that implement such nested virtualization for AMD SEV report high performance cost—throughput drops between 57% and 85% for MySQL, memcached, and Nginx [25]. We anticipate further slowdown for interrupt filtering since, for each interrupt injection the host has to schedule the vCPU running in VMPL0 followed by the vCPU in a higher VMPL running the nested guest Linux OS.

10.2 Hardware-based Selective Filtering

Instead of relying on kernel patches that may break compatibility, hardware-level filtering offers a cleaner defense. One extreme solution is to filter all external interrupts for the CVM, but this breaks critical functionality such as timers. Instead, we propose selective filtering of interrupts that typically have explicit effect handlers.

TDX. Intel already blocks interrupts 0-31 from APIC by default. If the hypervisor needs to inject necessary interrupts between 0-31 (e.g., NMI), it needs to use the TDX interface. The trust domains module (TD module), which is in the TCB, provides this interface and determines whether to forward it to the CVM. As we reported in Sec. 3.2, none of the interrupts between 0-31 with explicit effect handlers are forwarded by the TD module. We recommend that TDX should treat int 0x80 the same as 0-31 and filter it. This will break legacy code that may externally inject int 0x80 [10, 12].

SEV-SNP. We recommend that SEV should employ similar filtering of all externally injected interrupts that may have explicit effect handlers. Doing such filtering in microcode can provide comprehensive protection against HECKLER. While the same effect can perhaps be achieved with the restricted and alternate modes, we have two reservations. This requires correctly patching several codebases for hypervisors, guest OSes, and VMPL0 implementations. Since we were not able to test the complete and functional implementations of these modes, it is unclear if they are completely robust against hypervisors. Specifically, one needs to ensure that the hypervisor has no way to: (i) inject these interrupts via the APIC or the synchronous interface; (ii) disable the restricted and alternate modes at any point during the CVM's execution; (iii) re-enter the handlers to exploit race-conditions or break atomicity and nested interrupt assumptions [30]. The upcoming secure AVIC proposal from AMD is a good candidate to achieve hardware-level filtering, where the CVM can specify a hardware interrupt filter without software intervention [57].

11 Related Work

Previous works attack SEV's memory protection to inject arbitrary code to the CVM. [42, 59] CrossLine attacks use hypervisor-controlled address space identifiers (ASIDs) to compromise SEV VMs just before they crash [41]. Further, there have been numerous exploits that compromise SEV VMs using side-channels [40, 43, 45, 58]. Buhren et al. [14] compromise SEV's remote attestation mechanisms to extract platform keys and perform arbitrary code injection in SEV VMs. Zhang et al. architecturally revert modified cache lines to break SEV [63]. Buhren et al. mount fault injection attacks against SEV-SNP VMs by extracting endorsement keys using voltage glitching [13]. SEV-ES has been shown to offer much weaker security than SEV-SNP [2]. However, HECKLER breaks SEV-SNP guarantees without relying on any micro-architectural, architectural, power, or glitching side-channels. Google performed a security review of Intel TDX and SEV SNP and reported several issues [27, 30]. Notably, on TDX they found a vulnerability that allowed untrusted firmware to induce software exceptions during the early boot stages. Using this, they gain control over the instruction pointer during trusted firmware execution, thus achieving arbitrary code execution. To the best of our knowledge, HECKLER is the first attack on TDX from untrusted hypervisor. Further, we do not control the instruction pointer, instead we re-use the handlers in the trusted software (guest OS and user applications). AMD emphasizes that the hypervisor must respect RFLAGS.IF to preserve guest kernel functionality [2], but HECKLER does not violate this flag. Future works can explore the combination of HECKLER with this mechanism to exploit the kernel [39].

Tooling. SEV-step and SGX-step use timer interrupts to build single-stepping primitives for SEV-SNP VMs and SGX enclaves respectively [54, 60, 63]. HECKLER does not require the full-fledged suite of primitives offered by these tools and they do not apply out-of-box for our attack. However, when we build our tooling, we re-use valuable insights and implementation details from these tools.

Lift and Shift. Porting legacy applications to TEE platforms with zero developer efforts is referred to as lift-and-shift. Porting applications to Intel SGX entails maintaining compatibility [8, 11, 16, 50] and performance [8, 50]. CVMs, due to their VM abstraction, reduce the overheads of porting legacy applications. However, using AMD SEV-SNP and Intel TDX still requires enlightening the guest OS to ensure that legacy code written with the assumption of a trusted hypervisor is protected in the TEE threat model. Further, the untrusted hypervisor also needs to support the creation of CVMs for different TEE backends. To this end, Intel, AMD, and several hypervisor solutions such as KVM and Hyper-V are working towards patching the hypervisors and guest OSes. Other approaches introduce a trusted manager inside the CVM that acts as a bridge between the hypervisor and the guest OS, removing the need to patch existing guest OSes. Recent works have shown that one can leverage AMD SEV-SNP's VMPL modes to achieve this goal [25]. All of these works emphasize and aim to protect against the threats of untrusted privileged software. However, their reasoning about malicious interrupts, especially for CVMs, is either missing or incomplete.

Interface Security. Previous works that attack Intel SGX enclaves show the importance of correctly securing untrusted interfaces (e.g., system calls) [17, 37]. Several works exploit interfaces of various TEEs to leak secret keys and enable remote code reuse [18, 44, 52, 53]. In a similar vein, HECKLER abuses the interrupt interface controlled by the untrusted hypervisor but for CVMs which offer a different abstraction.

Physical vs. Virtual Interrupts. Physical interrupts, including timers and page faults, are transparent—the victim application/CVM does not recognize it was interrupted and resumed. This allows the attacker to observe side-effects of said interruption [54, 60]. Defenses such as AEX-Notify make the victim aware of physical interrupts, such that it can take preventive actions [21]. HECKLER observes that virtual interrupts are not transparent to the CVM, they do not cause a VM exit but instead the CVM actively reacts to them as if they were benign interrupts. One effect of such unexpected virtual interrupts is that the victim VM crashes (e.g., invalid opcode in kernel mode) or resumes execution (e.g., timers). This can perhaps be used to amplify side-channels, as is the case with physical interrupts. More importantly, HECKLER shows that certain virtual interrupts, when injected at the right time and location, have explicit effects that alter the register state of the victim CVM. HECKLER is the first work that abuses the virtual interrupt injection interface to alter the guest state to break the execution integrity of CVMs. WeSee [48] is our follow-up work on HECKLER. It expands our analysis but focuses on one particular interrupt vector 29, VMM communication exception (#VC), which was introduced in AMD SEV-SNP. Refer to Appendix E for further details.

Interrupt Protection. Wojtczuk and Rutkowska showed that in a mutually untrusted co-tenant VM setting, attackers can use rogue devices to perform interrupt injection attacks [61]. Next, we discuss prior works that focus on TEE settings. Isolated computation on low-end micro-controllers can be made resistant to interrupt/exception attacks (e.g., timer interrupts for side-channels) with programming mechanisms [15, 22, 23]. TrustZone's secure interrupts can isolate interrupts of the secure-world from the untrusted normal world [6]. AEX-Notify makes SGX enclaves aware of timer interrupt [21] using an ISA extension. Specifically, enclaves can register interrupt handlers to thwart single-stepping attacks stemming from timer interrupts.

Arm CCA. Unlike x86, Arm uses different interrupt architecture and nomenclature. The Arm defines 4 classes of exceptions (synchronous exception, IRQ, FIQ, and SError). We study the Arm CCA support for creating CVMs and report that it only allows injection of IRQs and FIQs into Arm CCA CVMs. The rest are filtered by the trusted Realm Management Monitor (RMM). We tested all the IRQs and FIQs with RMM v0.3.0 and did not observe explicit effect handlers. Arm does not have a concept of a syscall interrupt like x86.

12 Conclusion

HECKLER presents a new attack on Intel TDX and AMD SEV-SNP that offer VM abstractions. It uses the untrusted hypervisor's interrupt management and delivery interface to inject malicious interrupts into CVMs. HECKLER's gadgets use the explicit and global effects of the interrupt handlers to change the data and control flow of victim programs. By injecting particular malicious interrupts at the right time in the right core, HECKLER breaks the integrity and subsequently confidentiality of CVM. Our case-studies show the severity of HECKLER and highlight the need for robust defenses.

Acknowledgement

We thank our shepherd, the anonymous reviewers, and Mélisande Zonta-Roudes for their constructive feedback for improving the paper. Thanks to Intel, AMD, and Linux for the mitigation discussions and for developing the patches. We thank Benny Fuhry and Mona Vij from Intel for granting us early-access to TDX pre-production machines.

References

[1] Alibaba. Build a TDX confidential computing environment , 2024.

[2] AMD. AMD SEV-SNP: Strengthening VM Isolation with Integrity protection and more, 2020.

[3] AMD. AMD SEV-SNP Host Tree, 2023.

[4] AMD. AMD64 Architecture Programmer's Manual Volumes 1–5, Rev. 4.07, 2023.

[5] AMD. Linux SVSM (Secure VM Service Module), accessed 2023-10-15.

[6] ARM. Learn the Architecture: TrustZone for AArch64, v. 1.1, 2021.

[7] ARM. Arm Confidential Compute Architecture (ARM-CCA), accessed 2023-10-15.

[8] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure Linux Containers with Intel SGX. In *USENIX OSDI*, 2016.

[9] Microsoft Azure. Azure Confidential VM options, 2024.

[10] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *SOSP*, 2003.

[11] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *USENIX OSDI*, 2014.

[12] Frederic Beck and Olivier Festor. Syscall Interception in Xen Hypervisor. 2009.

[13] Robert Buhren, Hans-Niklas Jacob, Thilo Krachenfels, and Jean-Pierre Seifert. One Glitch to Rule Them All: Fault Injection Attacks Against AMD's Secure Encrypted Virtualization. In *ACM CCS*, 2021.

[14] Robert Buhren, Christian Werling, and Jean-Pierre Seifert. Insecure Until Proven Updated: Analyzing AMD SEV's Remote Attestation. In *ACM CCS*, 2019.

[15] Matteo Busi, Job Noorman, Jo Van Bulck, Letterio Galletta, Pierpaolo Degano, Jan Tobias Mühlberg, and Frank Piessens. Securing interruptible enclaved execution on small microprocessors. *ACM TOPLAS*, 2021.

[16] Chia che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *USENIX ATC*, 2017.

[17] Stephen Checkoway and Hovav Shacham. Iago attacks: why the system call API is a bad untrusted RPC interface. In *ASPLOS 13*.

[18] Sanchuan Chen, Zhiqiang Lin, and Yinqian Zhang. Controlled Data Races in Enclaves: Attacks and Detection. USENIX Security, 2023.

[19] Shuo Chen, Jun Xu, and Emre C. Sezer. Non-Control-Data Attacks Are Realistic Threats. In *USENIX Security*, 2005.

[20] Julia Community. TextAnalysis.jl, Julia package for Text Analysis, accessed 2023-10-15.

[21] Scott Constable, Jo Van Bulck, Xiang Cheng, Yuan Xiao, Cedric Xing, Ilya Alexandrovich, Taesoo Kim, Frank Piessens, Mona Vij, and Mark Silberstein. AEX-Notify: Thwarting Precise Single-Stepping Attacks through Interrupt Awareness for Intel SGX Enclaves. In *USENIX Security*, 2023.

[22] Carlos Tomé Cortiñas, Marco Vassena, and Alejandro Russo. Securing Asynchronous Exceptions. In *IEEE CSF*, 2020.

[23] Ruan de Clercq, Frank Piessens, Dries Schellekens, and Ingrid Verbauwhede. Secure interrupts on low-end microcontrollers. In *IEEE ASAP*, 2014.

[24] Debian. Doas: minimalist replacement for the more popular sudo, accessed 2023-10-15.

[25] Xinyang Ge, Hsuan-Chi Kuo, and Weidong Cui. Hecate: Lifting and Shifting On-Premises Workloads to an Untrusted Cloud. In *ACM CCS*, 2022.

[26] Thomas Gleixner. x86/entry: Do not allow external 0x80 interrupts, accessed 2023-12-10.

[27] Google. AMD Secure Processor for Confidential Computing, 2022.

[28] Google. Confidential VMs on Intel CPUs: Your new intelligent defense, 2023.

[29] Google. Oh SNP! VMs get even more confidential, 2023.

[30] Google. Intel Trust Domain Extensions (TDX) Security Review, 2023.

[31] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another Flip in the Wall of Rowhammer Defenses. In *IEEE SP*, 2018.

[32] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks. In *IEEE S&P*, 2016.

[33] IBM. Confidential computing for total privacy assurance, accessed 2023-10-15.

[34] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4, 2023.

[35] Intel. Intel Trust Domain Extensions (Intel TDX), accessed 2023-10-15.

[36] Intel. Intel Trust Domain Extension Research and Assurance, accessed 2023-10-15.

[37] Mustakimur Rahman Khandaker, Yueqiang Cheng, Zhi Wang, and Tao Wei. COIN Attacks: On Insecurity of Enclave Untrusted Interfaces in SGX. In *ASPLOS*, 2020.

[38] Tianyu Lan. x86/sev: Add Check of #HV event in path, accessed 2023-10-15.

[39] Yoochan Lee, Changwoo Min, and Byoungyoung Lee. ExpRace: Exploiting Kernel Races through Raising Interrupts. In *USENIX Security*, 2021.

[40] Mengyuan Li, Luca Wilke, Jan Wichelmann, Thomas Eisenbarth, Radu Teodorescu, and Yinqian Zhang. A Systematic Look at Ciphertext Side Channels on AMD SEV-SNP. In *IEEE S&P*, 2022.

[41] Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. CrossLine: Breaking "Security-by-Crash" Based Memory Isolation in AMD SEV. In *ACM CCS*, 2021.

[42] Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin. Exploiting Unprotected I/O Operations in AMD's Secure Encrypted Virtualization. In *USENIX Security*, 2019.

[43] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. CIPHERLEAKS: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel. In *USENIX Security*, 2021.

[44] Aravind Machiry, Eric Gustafson, Chad Spensky, Christopher Salls, Nick Stephens, Ruoyu Wang, Antonio Bianchi, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. BOOMERANG: Exploiting the Semantic Gap in Trusted Execution Environments. In *NDSS*, 2017.

[45] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. SEVered: Subverting AMD's Virtual Machine Encryption. In *EuroSec*, 2018.

[46] Manohar Mukku. Implementation of Multi Layer Perceptron in C, 2021.

[47] Edward Raff. Java Statistical Analysis Tool, a Java library for Machine Learning, 2017.

[48] Benedict Schlüter, Supraja Sridhara, Andrin Bertschi, and Shweta Shinde. WeSee: Using Malicious #VC Interrupts to Break AMD SEV-SNP. In *IEE S&P*, 2024.

[49] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *ACM CCS*, 2007.

[50] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX. In *ASPLOS*, 2020.

[51] Kirill Shutemov. x86/coco: Disable 32-bit emulation by default on TDX and SEV, accessed 2023-12-10.

[52] Darius Suciu, Stephen McLaughlin, Laurent Simon, and Radu Sion. Horizontal Privilege Escalation in Trusted Applications. In *USENIX Security*, 2020.

[53] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D. Garcia, and Frank Piessens. A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes. In *ACM CCS*, 2019.

[54] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In *SysTEX*, 2017.

[55] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic. In *ACM CCS*, 2018.

[56] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *USENIX Security*, 2017.

[57] Kishon Vijay Abraham I, Suravee Suthikulpanit, and AMD. Secure AVIC: Securing Interrupt Injection from a 'malicious' Hypervisor. In *LPC*, 2023.

[58] Jan Werner, Joshua Mason, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. The SEVerESt Of Them All: Inference Attacks Against Secure Virtual Enclaves. In *ACM AsiaCCS*, 2019.

[59] Luca Wilke, Jan Wichelmann, Mathias Morbitzer, and Thomas Eisenbarth. SEVurity: No Security Without Integrity : Breaking Integrity-Free Memory Encryption with Minimal Assumptions. In *IEEE S&P*, 2020.

[60] Luca Wilke, Jan Wichelmann, Anja Rabich, and Thomas Eisenbarth. SEV-Step: A Single-Stepping Framework for AMD-SEV, 2023.

[61] Rafal Wojtczuk and Joanna Rutkowska. Following the White Rabbit: Software attacks against Intel (R) VT-d technology, 2011.

[62] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *IEEE S&P*, 2015.

[63] Ruiyi Zhang, Lukas Gerlach, Daniel Weber, Lorenz Hetterich, Youheng Lü, Andreas Kogler, and Michael Schwarz. CacheWarp: Software-based Fault Injection using Selective State Reset. In *USENIX Security*, 2024.

A Interrupt Injection Flow

```
<mm_anwser_auth_password>
...
call auth_password
test eax,eax
...
```

Listing 1: OpenSSH mm_anwser_auth_password function

1. Guest executes `ret` in `auth_password`
2. CPU fetches page of `mm_answer_authpassword` to continue execution at line 3 in `mm_answer_authpassword`.
3. CPU throws a stage-2 page fault, since the page which contains `mm_answer_authpassword` is marked as non-executable.
4. CPU exits VM mode and transfers control back to the hypervisor.
5. Hypervisor clears NX (non-executable) bit of `mm_answer_authpassword` stage-2 entry
6. Hypervisor modifies `VMCB.EventInj` field to inject interrupt into VM
7. Hypervisor executes `VMRUN`
8. `VMRUN` evaluates `VMCB.EventInj` field and signals an Interrupt before guest execution is resumed.
9. Guest Kernel handles interrupt (int0x80)
10. Guest continues execution on line 3 with corrupted state

Technically this corresponds to a window of single instruction from the victim CVM point of view. However, our page fault mechanism exits the VM. Then on the hypervisor, the attacker can take as much time as it wants to set the interrupt vector and resume the VM. On resumption, subsequent steps happen out of the box.

Table 4: SEV-SNP interfaces for interrupt injection to VM [4]

AMD VM interface	Injection Point	Effect
VMCB Event Injection	synchronous	raise interrupt before the first guest instruction is executed
virtual APIC	asynchronous	raise interrupt whenever the AVIC registers a change in the IRR register
VMCB V_IRQ	synchronous	VMRUN loads the intr. information into the respective on-chip registers
Physical IRQ Intercept	asynchronous	Physical interrupts are interpreted as virtual interrupts and do no exit the VM

B Interrupt Injection Interfaces

AMD has four different interrupt injection interfaces implemented in their virtualization extensions as summarized in Tab. 4. As described in the main text we exclusively used `VMCB.EventInj` in our proof-of-concept implementation. It allows to serve interrupts when a CVM is resumed with the `VMRUN` instruction. Next, we have the virtual APIC (AVIC Advanced Virtual Interrupt Controller in AMDs nomenclature). This is used for asynchronous interrupt injection, i.e., the vCPU does not need to exit and enter again to receive an interrupt. We did not use the `V_IRQ` but the documentation indicates it does the same as the `VMCB.EventInj`. Most interestingly is the last interface, the Physical IRQ interception bit. We can unset this bit on `VMRUN` and cause all physical interrupts to be received by the CVM rather than causing a `VMEXIT`. However, this bit is ignored when Restricted / Alternate Injection is enabled. Thus this interface cannot be used to circumvent those modes. TDX has a similar interface but is not susceptible. The interrupt interception bit is controlled by the TD module that is part of the TCB.

C Busy Wait Loop

The code busy loops until `authenticated` becomes equal to one. This is used in OpenSSH and sudo for the TDX proof-of-concept exploits.

```
// assume authenticated == 0
__asm__ __volatile__("1:;\n"
          "nop\n"
          "test %%eax, %%eax\n"
          "je 1b \n"
          :"+a"(authenticated):);
return authenticated;
```

D Tracing Overhead

Table 5 shows the time taken to generate boot trace, application set ($\mathbb{S}_{\text{app}}$), and page trace (PT_{app}), when compared to the execution of the CVM without page fault tracing. During the offline phase, we get multiple traces as summarized in Table 1. In this case, the overheads of tracing slowdown the function generation described in Section 8.1. While this can be further optimized, we did not put efforts in such optimizations since this is a preparatory step before the victim runs its VM.

HECKLER also enables page fault tracing in the online phase, i.e. when the victim starts interacting with the VM. HECKLER causes some slowdown but it does not impact the victim's usability or result in detection. This is because for OpenSSH and sudo, we can potentially perform the tracing and the injection after attestation, but during the CVM provisioning which can take several minutes even in a benign setting. Thus, HECKLER attack happens before the user gets access to the VM, so it will not notice the lag. In cases where this is not possible (e.g. MLP), we can further cap the number of page faults for any given page (max capture in Tab. 1) to reduce the lag.

E Interrupt Classification

Custom Interrupt Injections. To find interrupts of interest, we analyze the impact of malicious injections on a guest VM kernel and userspace application. For this, we survey all 256 interrupt vectors that are supported by current AMD SEV-SNP-enabled CPUs. To get started, we examine the legacy interrupt injection functions in KVM that are used by the hypervisor to inject benign interrupts. On the host kernel, we hook these KVM functions to build a kernel module interface. Using a custom kernel module, we can now selectively inject interrupts into VMs as explained in Sec. 8.1.

```
jmp_buf jumper;
void handler(int signum){
    printf("in sig handler\n");
    longjmp(jumper, 1);
}
int main()
{
    volatile int i, j;
    struct sigaction act;
    struct sigaction oldact;
    memset(&act, 0, sizeof(act));
    act.sa_handler = handler;
    act.sa_flags = SA_NODEFER | SA_NOMASK;
    for (i = 0; i <= 32; i++){
        if (sigaction(i, &act, NULL)){
            printf("Sig. install err\n");}}
    while (1){
        int x;
        asm("label2:");
        printf("Reached outside of loop\n");
        x = setjmp(jumper);
        if (x == 0){
            loop1:
            while (1){
                asm("cmpl $0, % eax\n\r"
                    "jne label2");
            }
        } else {
           goto loop1;
        }
    }
    return 0;
}
```

Listing 2: User-space application with general signal handler

Setup. Similar to the end-to-end attacks in Sec. 8, we use a workstation with an EPYC 9124 CPU with 16 cores and 192 GB RAM. We boot a Linux kernel v6.5.0-rc2 from AMD [3] as the host operating system, which is modified to support our injection hooks. To start the VMs and perform our experiments, we use QEMU version 8.0.0. For the SEV-SNP VM we use the same Linux kernel as for the host, but without the patches. To test if the guest kernel generates a signal based on the injected interrupt, we implement a userspace application that is capable of catching these signals. The application (sources in Lst. 2) registers dummy handlers for signals 0-31 with the kernel and busy-waits for their delivery. Furthermore, it checks continuously if registers, such as `eax`, were modified, for example by the kernel. To ensure that we are actually injecting the the interrupt while our application is executing, we run and pin it on each available vCPU (8 in our case) using `taskset`. This lowers the risk of the interrupt being delivered during kernel execution, by preventing it from idling.

Evaluation. To classify the interrupts, we first manually analyzed the interrupt handler of the guest kernel for the anticipated behavior. This gives us the ability to compare the actual behavior of the OS to the injected interrupt with an expected normal execution flow. To the best of our abilities, we classify the predictions into the following categories using manual analysis:

1. **Signal to usermode**: The kernel is expected to generate and send a signal to the user space application when the interrupt arrives while the core is executing in user space.

2. **Specific kernel handler** The kernel has reserved handlers that are executed in the kernel when an interrupt of this type is delivered. However, it does not send any signal to user space.

3. **Default kernel handler** The kernel has not registered any special handling for this type of interrupt. Instead, a basic handler is executed that logs the delivery and resumes the execution.

4. **Kernel Panic** When an interrupt of this type is delivered, the kernel reacts with an unrecoverable panic and halts the execution on all vCPUs.

5. **Syscall** We expect the kernel to handle a system call originated by the user space application.

6. **Unclear** The expected behavior of this interrupt handler is inconclusive from our manual analysis.

Table 5: Overheads to generate boot trace, application set ($\mathbb{S}_{\text{app}}$), and page trace (PT_{app}) w.r.t. execution without page faults.

App	boot			$\mathbb{S}_{\text{app}}$			PT_{app}		
	trace off	trace on	+%	trace off	trace on	+%	trace off	trace on	+%
OpenSSH	10.01s	11.41s	14%	14.2ms	32.9ms	131%	14.2ms	773.9ms	5332%
Sudo	10.01s	13.68s	37%	1.00s	1.03s	3%	1.00s	7.02s	602%
MLP	8.71s	12.01s	38%	1.08s	1.11s	3%	1.08	1.95s	81%

Table 6: Summary of observations for injecting interrupts into a guest VM on AMD SEV-SNP (per vector).

Int. (Dec)	Int. (Hex)	Description	Expected behavior in guest VM	Observed behavior
0	00	Divide by 0	Signal to usermode	Signal to usermode (SIGFPE)
1	01	Debug	Signal to usermode	Signal to usermode (SIGTRAP)
2	02	NMI	Specific kernel handler	VM_EXIT fail
3	03	Breakpoint	Specific kernel handler	VM_EXIT fail
4	04	Overflow	Signal to usermode	VM_EXIT fail
5	05	Bound Range Exceeded	Signal to usermode	VM_EXIT fail
6	06	Invalid Opcode	Signal to usermode	Signal to usermode (SIGILL)
7	07	Device not available	Specific kernel handler	App crash
8	08	Double Fault	Kernel Panic	Kernel panic
9	09	Co-Processor Segment overrun	Signal to usermode	VM_EXIT fail
10-13	0A-0D	TSS/Segment/Protection Faults	Unclear	App crash
14	0E	Page Fault	Specific kernel handler	Kernel panic
15	0F	Spurious Interrupt	Unclear	VM_EXIT fail
16	10	x87 Floating Point Exception	Unclear	No effect
17	11	Alignment Check	Specific kernel handler	App crash
18	12	Machine Check	Specific kernel handler	No effect
19	13	SIMD Floating Point Exception	Unclear	No effect
20	14	Virtualization Exception	Specific kernel handler	VM_EXIT fail
21	15	Control Protection Exception	Specific kernel handler	App crash
22-28	16-1C	Undefined	Unclear	VM_EXIT fail
29	1D	VMM Communication Exception	Specific kernel handler	VM_EXIT fail
30	1E	Undefined	Unclear	Kernel panic
31	1F	Undefined	Unclear	VM_EXIT fail
32	20	IRET Exception	Specific kernel handler	No effect
33	21	Undefined	Default kernel handler	Default handler executed
34	22	Undefined	Default kernel handler	No effect
35-47	23-2F	Undefined	Default kernel handler	Default handler executed
48	30	ISA IRQ	Specific kernel handler	No effect
49	31	ISA IRQ	Specific kernel handler	Default handler executed
50	32	ISA IRQ	Specific kernel handler	System unresponsive
51-63	33-3F	ISA IRQ	Specific kernel handler	Default handler executed
64-127	40-7F	Undefined	Default kernel handler	Default handler executed
128	80	Syscall	Syscall	App control flow changed
129-235	81-EB	Undefined	Default kernel handler	Default handler executed
236-243	EC-F3	Local Timer and Hypervisor Int.	Specific kernel handler	No effect
244	F4	Deferred Error	Specific kernel handler	Specific handler executed
245-247	F5-F7	IRQ Work + x86 IPI Interrupts	Specific kernel handler	No effect
248	F8	Reboot Interrupt	Specific kernel handler	System unresponsive
249-250	F9-FA	Threshold + Thermal APIC Int.	Specific kernel handler	Specific handler executed
251-254	FB-FE	Function Call, Resched., Error Int.	Specific kernel handler	No effect
255	FF	Spurious APIC Interrupt	Specific kernel handler	System unresponsive

We classify the observable behavior into the following categories:

1. **Signal to user space (TYPE)** We observed that a TYPE signal was received in the user space application
2. **Specific handler executed** The guest VM executed a dedicated interrupt handler for this vector in the kernel.
3. **Default handler executed** The guest VM executed the basic placeholder interrupt handler without any implications.
4. **No effect** The invocation of the interrupt did not have an observable effect on the guest VM.
5. **System unresponsive** The status of the guest VM is inconclusive; no information exchange with the guest Kernel is possible after the interrupt injection.
6. **App crash** The execution of our user space application was terminated, but the OS was able to continue operating normally.
7. **App control flow changed** We were able to see an impact on the control flow of the executed user space application.
8. **VM_EXIT fail** After injecting the interrupt, the hypervisor was not able to continue the execution of the VM as the CPU refused to enter the guest successfully.

We present a summary of our findings in Tab. 6. We used the `svm_deliver_interrupt` function within the Linux kernel to inject all the interrupts. This API does not allow us to set an error code nor does it always set the right hardware flags for all interrupts (i.e., NMIs are supposed to be injected using a different API). Thus, for all interrupts where we report *VM_EXIT fail*, one can still inject these interrupts in the VM by directly modifying the respective fields in the `VMCB`. Furthermore, some interrupts might panic the kernel only under certain circumstances and have different effects if the interrupt is raised at the right time (e.g., int3 on a debug instruction). As a concrete example, we consider interrupt 29 which was flagged as "VM_EXIT fail" in our analysis. On further manual investigation, we identified that by directly manipulating `VMCB.EventIn`, we can inject the interrupt into the VM and cause a termination. By further modifying the `error_code` we can circumvent the crash and cause different global state change effects when the CVM resumes execution. We investigate the impact of this behavior in detail in WeSee [48]. For the scope of HECKLER, we do not do an extensive analysis of: (a) all potential hardware features and interfaces to inject interrupts and (b) all injection points during the victim execution. We leave this analysis as well as effects of combining (a) and (b) to detect other instances of Ahoi attacks to future work.