WeSee: Using Malicious #VC Interrupts to Break AMD SEV-SNP

Benedict Schlüter Supraja Sridhara Andrin Bertschi Shweta Shinde
ETH Zurich

***Abstract*—AMD SEV-SNP offers VM-level trusted execution environments (TEEs) to protect the confidentiality and integrity for sensitive cloud workloads from untrusted hypervisor controlled by the cloud provider. AMD introduced a new exception, #VC, to facilitate the communication between the VM and the untrusted hypervisor. We present WeSee attack, where the hypervisor injects malicious #VC into a victim VM's CPU to compromise the security guarantees of AMD SEV-SNP. Specifically, WeSee injects interrupt number 29, which delivers a #VC exception to the VM who then executes the corresponding handler that performs data and register copies between the VM and the hypervisor. WeSee shows that using well-crafted #VC injections, the attacker can induce arbitrary behavior in the VM. Our case-studies demonstrate that WeSee can leak sensitive VM information (kTLS keys for NGINX), corrupt kernel data (firewall rules), and inject arbitrary code (launch a root shell from the kernel space).**

1. Introduction

Hardware-based trusted execution environments (TEEs) make it possible to execute sensitive computation on an untrusted cloud while providing confidentiality and integrity guarantees. Hardware platforms and vendors, including Intel, AMD, Arm, and IBM have rolled out or announced support for VM-level TEEs [1], [2], [3], [4]. AMD Secure Nested Paging (SEV-SNP) provides both confidentiality and memory integrity of VM execution [1]. It is in production on major cloud service providers, including Azure, Google Cloud, and AWS [5], [6], [7] and has been applied to security-sensitive workloads [8], [9], [10], [11], [12].

Since the cloud provider controls the hypervisor on a cloud platform, TEEs such as AMD SEV-SNP deem this privileged software to be untrusted. In the CVM setting, the hypervisor is still responsible for configuration and management of resources, including interrupts. This change in the trust model strengthens the security guarantees by enforcing that the hypervisor can no longer access the VM's memory or registers in plain text, thus protecting the VM. However, this breaks several traditional systems abstractions that are essential to execute VMs. For example, the VM needs services such as CPUID and hypercalls from the hypervisor, which is rendered impossible if the hypervisor cannot access the VM's state in unencrypted form.

To address this challenge, AMD SEV-SNP introduces new interfaces between the untrusted hypervisor and the trusted VM. This allows the VM to re-enable the essential functionality, while being able to control the use of the interface and perform sanitization and correctness checks. For example, the VM can selectively exchange data from its own memory to a shared memory region with the hypervisor. Further, to maintain performance and compatibility, AMD SEV introduces a new exception called *VMM Communication Exception (`#VC`)* [13]. The CPU raises this exception when the VM needs to communicate with the hypervisor. This enables the VM's exception handler to perform the communication via the shared memory region transparently without changing the guest kernel or applications. Whenever the VM executes an instruction that require hypervisor intervention (e.g., `cpuid`, `rdtsc`, memory mapped I/O), the SEV-SNP hardware raises a `#VC` exception.

Our attack, called WeSee, abuses the `#VC` exception to break the security guarantees of AMD SEV-SNP. Our first observation is that the hypervisor can inject a *malicious `#VC`* into a CPU that is executing a SEV-SNP VM at any time. Specifically, the hypervisor has the ability to inject external interrupts to the CPUs, including `#VC` which is yet another exception. Our second observation is that SEV-SNP invokes the `#VC` exception handler in the VM without checking the authenticity of the root cause. Specifically, the VC handler does not check if the VM indeed executed an instruction that would legitimately cause the CPU to generate a `#VC` exception. Our third observation is that the VC handler performs sensitive operations of copying data between the VM and the hypervisor to emulate the semantics of the instruction that generated the `#VC`. The handler is programmed to be bug-free and has checks to defend against Iago attacks, i.e., it clears all registers and performs checks on the data values provided by the hypervisor before it uses them as per AMD specifications [14]. However, it is not programmed to defend against `#VC` that is maliciously injected by the hypervisor. Worse yet, each malicious `#VC` injection tricks the handler into emulating an instruction that either writes attacker-controlled data to the VM or leaks sensitive VM data to the hypervisor.

WeSee shows that with each malicious `#VC` injection from the hypervisor, the attacker can induce a basic primitive operation on the victim guest VM. For example, by faking a `#VC` for MMIO read, the attacker can achieve an arbitrary memory write—the hypervisor can write any value of its choice to any location in the victim VM. To achieve each basic primitive, we address several challenges such as ensuring that the victim VM does not crash (e.g., due to existing sanitization checks) and identifying particular

0. This is the author's version of the IEEE S&P 2024 paper.

execution points in the victim's execution to inject malicious #VC. We demonstrate 4 main primitives namely: skipping instruction execution, leaking registers, corrupting registers, and arbitrary read/write to VM memory.

Finally, WeSee shows that the attacker can inject several malicious #VC to cascade the effect of the above basic primitives (e.g., memory write followed by memory read). We demonstrate several nuances that allow WeSee to (a) inject consecutive #VCs before the VM resumes execution; (b) inject nested #VCs while the VM is executing the handler itself; and (c) combine consecutive and nested interrupts. Put together, this allows WeSee to bring about highly expressive attacks such as arbitrary code injection and execution.

We demonstrate the expressiveness of WeSee with three end-to-end case studies. We leak kernel TLS session keys for NGINX with the arbitrary read. We use arbitrary write and code injection primitives to disable firewall rules and open a root shell. Orchestrating these case studies requires addressing challenges such as identifying suitable points of execution in the victim VM. Prior works have shown that AMD SEV-SNP is vulnerable to side channels that can be leveraged to achieve single-stepping primitives. However, WeSee does not require such high-resolution information about the victim VM. Instead, we purely rely on the page fault sequences of the victim VM to perform our end-to-end attacks. We discuss potential software and hardware-based defenses to thwart WeSee and argue for robust hardware mechanisms to limit the hypervisor's capabilities.

Heckler [15] and WeSee show that the hypervisor can abuse the notification mechanisms, existing and new respectively, to break CVM guarantees. These works points to a family of attacks called *Ahoi attacks*,[1] where the attacker sends malicious notifications, both in time and in value, to trick the victim. Prior works that abuse timer interrupts and page faults can also be classified as Ahoi attacks, because they generate fake interrupts that allows the attacker to observe side-effects (e.g., cache and timing). However, WeSee and Heckler generate interrupts that lead to explicit effect handler execution which directly update the global state (registers and memory) of the victim.

In summary, we make the following novel contributions:

- WeSee abuses the #VC exceptions to break AMD SEV-SNP.
- WeSee injects multiple well-crafted #VC exceptions into the victim VM to induce arbitrary reads, writes, and code injection.
- We demonstrate three case studies for WeSee: leaking kTLS keys for NGINX, bypassing the firewall, and obtaining a root shell.

We responsibly disclosed our findings to AMD on 26 October 2023 and the cloud providers on 5 February 2024. WeSee was assigned CVE-2024-25742.
WeSee tooling and PoC exploits are open-source at:
https://ahoi-attacks.github.io/wesee

1. Ahoi is a signal word to call a ship or boat. It is also an anagram of Iago [16] with edit distance of one.

2. Overview

AMD SEV-SNP disallows hypervisors from accessing guest VM registers and memory, thus necessitating #VC.

2.1. Background

AMD virtualization extensions (AMD-V), introduced in 2006, provide hardware support for the hypervisor to create and launch guest VMs. Since the VMs cannot directly access certain system resources (e.g., `rdtsc`), AMD-V allows the hypervisor to set up intercepts on particular instructions to manage and facilitate execution in the VMs.
Instruction interception for virtualization. Several operations in the guest VM—reads/writes to hypervisor-controlled Model Specific Registers (MSRs) and memory-mapped devices, accessing `rdtsc`, generic calls to the hypervisor—require co-operation with the hypervisor. When a VM executes such an instruction, the CPU intercepts it and automatically triggers a `vmexit` that is handled by the hypervisor. Therefore, the CPU's instruction intercept mechanism facilitates calls to the hypervisor when these operations are performed in the guest VM. This mechanism is fast and transparent to the guest VM as it does not require the involvement of the guest kernel to support it. Further, the CPU maintains a fixed set of instructions for which it triggers a `vmexit`, that the hypervisor can handle [14]. For example, a guest VM uses the `vmmcall` instruction to explicitly communicate with the hypervisor. First, the guest VM sets up registers and memory that contain values to indicate the reason to the hypervisor to process the VM's request. Then, the guest VM executes the instruction (e.g., `vmmcall`) that causes a `vmexit` to the hypervisor. Due to the `vmexit`, the hypervisor's handler is invoked where it can directly read the registers and memory from the guest VM and process the request. Since the hypervisor is trusted in AMD-V, this mechanism allows the VMs to execute unchanged while the hypervisor handles the special instructions.
AMD SEV-SNP. AMD SEV-ES and its successor SEV-SNP enable the creation and isolation of confidential VMs in various cloud deployments [10]. In particular, AMD SEV-SNP provides a hardware-based trusted execution environment that protects the memory and registers of the VMs and renders them inaccessible to the hypervisor. This protection breaks some existing abstractions (e.g., instruction intercepts set by hypervisor). However, the isolated VM still needs to communicate with the hypervisor to perform different operations. To address this gap, SEV defines a strict protocol to share data between the VM and the hypervisor using an unprotected shared memory region called the Guest Hypervisor Communication Block (GHCB) [17].

2.2. Implications of Instruction Interception

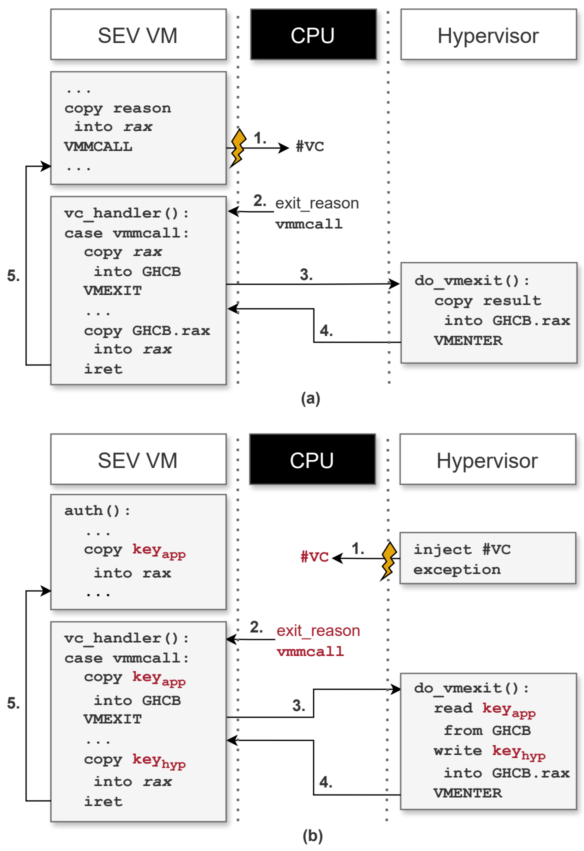
AMD adds support for secure instruction interception for VMs.[2] We analyze the need for such interception and

2. VM is a shorthand notation for SEV-SNP VM unless stated otherwise.

the security implications of this support in AMD SEV-SNP.

Need for #VC exception. When AMD SEV-SNP is enabled, the hypervisor cannot directly access the VM registers and memory. Therefore, to enable instruction intercepts, VM's data needs to be copied into the GHCB. However, this copy to the GHCB is not performed automatically when the CPU intercepts an instruction. There are several approaches to solving this issue: In the first approach, the user application in the VM is aware that it is executing in a VM and it explicitly uses the GHCB APIs by calling the guest kernel to copy data to the GHCB before exiting to the hypervisor. This approach requires invasive changes to the application code and breaks compatibility. In the second approach, the guest kernel intercepts the instructions and performs the GHCB API calls. This solution is cumbersome as the guest kernel has no mechanism to determine which instructions in the applications to intercept (e.g., `mov` instructions executed for MMIO). Therefore, this approach either needs instrumentation of the application and the kernel or invasive changes to the application. There is a third approach: if the VM executes a specific set of instructions, the CPU raises a special exception that is delivered to the VM's kernel. The VM can register a handler for this exception where it can assess the reason for the exception and perform data copies to the GHCB, such that the VM can communicate with the hypervisor by performing a `vmexit` in the exception handler. The hypervisor can then, in its own handler, access the GHCB, perform the relevant operations, and save the results in the GHCB. When the hypervisor returns control to the VM, the guest VM kernel handler uses the results from the GHCB and returns them to the user. If the hardware supports a new exception, this is the best approach. It is transparent to the user while the guest kernel can perform instruction-specific data copies as well as enforce interface sanitization. Therefore, SEV introduces a new exception called *VMM Communication Exception (VC)* with interrupt number 29, to facilitate VM and hypervisor communication. The AMD SEV hardware has added support to intercept a certain set of instructions from a VM. When such an instruction is executed in the guest VM (user or kernel-space), the CPU generates a `#VC` exception and sets the `exit_reason` register to indicate the instruction that caused the exception. The rest of the execution flow is software-based via the VC handler in the guest and the `vmexit` handler in the hypervisor where they use the GHCB as a shared memory region according to AMD's specification. Fig. 1(a) depicts this and shows the pseudo-code and execution flow for `#VC` handling.

Example: Supporting `vmmcall` with #VC. Consider an application in the VM that executes a `vmmcall` to request data from the hypervisor, as shown in Fig. 1(a). When the VM executes a `vmmcall`, i.e., instruction that should be intercepted by the hypervisor, the CPU triggers a `#VC` exception (Step 1). Then the CPU sets a hardware register `exit_reason` with the instruction (`vmmcall`) and raises a `#VC`. As in the non-confidential case, the application in the VM still sets up the `rax` register with the reason for the `vmmcall` for the hypervisor to process the request.



```
...
copy reason
 into rax
VMMCALL
...
```

```
vc_handler():
case vmmcall:
  copy rax
   into GHCB
  VMEXIT
  ...
  copy GHCB.rax
   into rax
  iret
```

```
do_vmexit():
  copy result
   into GHCB.rax
  VMENTER
```

```
auth():
  ...
  copy key_app
   into rax
  ...
```

```
inject #VC
exception
```

```
vc_handler():
case vmmcall:
  copy key_app
   into GHCB
  VMEXIT
  ...
  copy key_hyp
   into rax
  iret
```

```
do_vmexit():
  read key_app
   from GHCB
  write key_hyp
   into GHCB.rax
  VMENTER
```

Figure 1. (a) Benign execution of `#VC` in SEV VM. 1. application sets up `rax` and does `vmmcall` 2. CPU intercepts the instruction and raises `#VC` with `exit_reason` set 3. VC handler copies values (`rax`) into GHCB and exits to hypervisor 4. Hypervisor returns back to the VC handler which copies values (`rax`) from GHCB 5. Return to application that caused `#VC`. (b) Attack execution with WeSee.

When the `#VC` occurs, the execution returns back to the VC handler in the VM (Step 2). The VC handler is responsible for copying only the data required to process the `vmmcall` into the GHCB based on the `exit_reason` register. For the `vmmcall`, the handler only copies `rax` into the GHCB. Therefore, it ensures that the `#VC` exception does not leak information to the hypervisor. Finally, the handler does a `vmexit` to return control to the hypervisor to process the VM's request (Step 3). Before resuming the VM execution, the hypervisor writes the result of the `vmmcall` in `rax` of the GHCB and returns execution back to the VC handler (Step 4). Then, the VC handler in the VM only copies the data requested based on the `exit_reason` register into the application context. For the `vmmcall`, it only copies the `rax` register back into the application. Finally, the application that caused the `#VC` can resume execution (Step 5). This process ensures that the hypervisor intercepts function correctly without compromising the VM's security.

Interrupt delivery to SEV-SNP VMs. The hypervisor manages the delivery of interrupts for the VMs. It can inject physical (e.g., timer interrupts) and virtual interrupts (e.g., virtio interrupts) to the VMs using the Interrupt Controller. x86-64 treats interrupt vector numbers 0-31 as exceptions [14], [18]. Therefore, the hypervisor can use the interrupt controller to also inject the newly introduced `#VC` at any time to any cores that are executing the VM.

2.3. WeSee Attack

An attacker with the ability to inject interrupts into the VM (e.g., malicious hypervisor) can trigger a `#VC` at any time during the VM execution. Further, the hypervisor can write any value to the `exit_reason` register. More importantly, this interrupt causes the guest to execute the VC handler that examines the `exit_reason`, copies data from the VM to GHCB, and then does a `vmexit` to the hypervisor who can look at the data in the GHCB. Furthermore, the VC handler also takes in the data provided by the hypervisor and copies it from the GHCB into VM memory. In other words, the hypervisor can misuse `#VC` to execute the VM's handler at any point during the VM's execution and compromise it. Using this insight, we present WeSee attack that exploits the `#VC` to induce register and memory copy operations between the malicious hypervisor and the victim VM.

Example. Consider a victim application executing in a VM. It authenticates a remote user by comparing an input key (k_{in}) with a secret key (k_{app}). Lst. 1 shows this simple logic and Fig. 1(b) shows WeSee attack on Lst. 1.

```
1 mov rbx, $kin
2 mov rax, $kapp
3 cmp rax, rbx
4 jne deny
5 auth: ...
6 jmp fin
7 deny: ...
```

Listing 1. Example application

The malicious hypervisor does not know the value of k_{app} and therefore is not authenticated to use the application. With WeSee, the hypervisor can successfully authenticate itself, thus breaking the SEV-SNP guarantees. The application copies the value of k_{app} into `rax` (Line 2 in Lst. 1) before comparing it with k_{in} (Line 3). The hypervisor maliciously injects the `#VC` exception after Line 2. Fig. 1(b) Step 1 shows that the hypervisor can set the `exit_reason` as `vmmcall`. This triggers the VC handler in the VM, which copies the k_{app} from `rax` into the GHCB and exits to the hypervisor (Step 2, 3). After this point k_{app} is leaked to the hypervisor allowing it to authenticate successfully using this input. Alternatively, with this attack, the hypervisor can also write any value of its choosing (k_{hyp}) into the GHCB's `rax` to authenticate successfully. Specifically, it writes its own key which it sent to the application (k_{in}) to the GHCB before returning to the VM's VC handler (Step 4). The VC handler then copies the value of `rax` to the application and returns to it (Step 5). This changes the value of `rax` in the application leading to Line 3 computing k_{in} == k_{hyp} which will always be true as both the values are controlled by the hypervisor. Therefore, the hypervisor authenticates successfully and the `auth` block is executed (Line 5). Our example shows how a malicious hypervisor can use WeSee to compromise the execution integrity as well as data confidentiality and integrity of the VM, thus breaking AMD SEV-SNP.

3. WeSee Overview

The VC handler executes in a VM that the hypervisor cannot modify or tamper with. To assess the potential of WeSee, we manually analyze the Linux kernel v6.7-rc4 that implements the VC handler. Our analysis shows that there are constraints on: (a) when the hypervisor can trigger certain `#VC` (e.g., MMIO can only be triggered on a `mov` instruction); and (b) what operations the handler performs based on `exit_reason` (e.g., `vmmcall` can only read and write `rax`). Therefore, to mount WeSee attacks, the hypervisor has to ensure that it achieves its desirable effects (e.g., change the value of `rax` to `0xdeadbeef`) without resulting in a VM crash. More importantly, the `#VC` handler performs several other operations (e.g., masking registers before copying to GHCB) and checks (e.g., checking the operands of the instruction that caused the `#VC`) that either hinder the attacker from achieving its desired effect by causing check failures or result in excess execution (i.e., undesirable effects). Thus, we have to carefully craft the `exit_reason` at a well-chosen execution point in the VM such that we precisely induce the desirable effects of `#VC` handling while avoiding any checks and undesirable effects. WeSee aims to leverage the hypervisors' ability to inject multiple `#VC`s to cascade the desired effects of the handler to bring about expressive attacks. For example, use one `#VC` to change the `rax` value and then use another `#VC` to change `rcx` value. While injecting multiple `#VC`s is straightforward, doing so requires knowing when to inject the next `#VC` while ensuring that the handler does not crash (e.g., due to nested exceptions beyond a certain depth).

3.1. Analysis of #VC Handler

The VM's VC handler copies different registers and memory to and from the GHCB, based on the intercepted instruction indicated by the `exit_reason` register. We analyzed the VC handler in the latest version of the Linux kernel (v6.7-rc4 as of this writing). It is implemented as per the GHCB protocol specified by AMD SEV-SNP developer manual [14] and handles 19 instruction intercept events. Out of these, we found that 10 events lead to register and memory copies (see Tab. 1). For example, in case of `rdpmc` the handler sends values via register `rcx` to the hypervisor, and copies values from the hypervisor into registers `rax` and `rdx`. For the remaining 9 events, the VC handler does not send or receive any data from the hypervisor. Of the 10 events, in WeSee, we only use the 3 events shown in Tab. 1 and show that they are sufficient to build strong attack

TABLE 1. INTERCEPT EVENTS WITH #VC HANDLER IMPLEMENTATION IN SNP LINUX GUEST.
*: Register is masked before/after exchanged with the hypervisor, #reg: Register depends on the register used in the assembly instruction, GPA: Guest Physical Address, `vmgexit`: Guest exits to the hypervisor.

Event	Description	Reg. copied to Hyp	Reg. copied from Hyp	Sample Instr.	`vmgexit`	Used in WESEE
NPF MMIO Read	Memory mapped I/O read	-	#reg	mov rbx, [rax]	✓	✓
NPF MMIO Write	Memory mapped I/O write	#reg	-	mov [rax], rbx	✓	✓
VMMCALL	Call to VM Monitor	rax, cs*	rax	vmmcall	✓	✓
RDTSC/ RDTSCP	Read Time Stamp Counter	-	rax, rdx, rcx	rdtsc	✓	✗
RDPMC	Read Perf. Monitor Counter	rcx	rax, rdx	rdpmc	✓	✗
RDMSR	Read from MSR	rcx	rax, rdx	rdmsr	✓	✗
WRMSR	Write to MSR	rcx, rax, rdx	-	wrmsr	✓	✗
CPUID	CPU Identification	rax*, rcx*, xcr0*	rax, rbx, rcx, rdx	cpuid	✓	✗
IOIO_PROT	IO Ports (IN, OUT, INS, OUTS)	rax*	rax*	in eax, 0	✓	✗
DR7 write	Debug Control Reg. write	#reg*	-	mov, dr7, rax	✓	✗
RD7 read	Debug Control Reg. read	-	-	mov rax, dr7	✗	✗
INVD	Invalidate Internal Caches	-	-	invd	✗	✗
WBINVD	Write Back and Invalidate Cache	-	-	wbindv	✓	✗
MONITOR/ MONITORX	Set Up Monitor Address	-	-	monitor	✗	✗
MWAIT/ MWAITX	Monitor Wait	-	-	mwait	✗	✗
AC	Alignment Check	-	-	mov [0x1001], rax	✗	✗

primitives such as arbitrary register reads/writes, memory reads/writes, code injection.

Chaining multiple #VC. AMD SEV-SNP allows the hypervisor to inject consecutive `#VC`s to the same CPU. Consider a case where the VM is executing a user program. The hypervisor injects a first `#VC` and waits till the VM's kernel executes the corresponding handler. When the VM's kernel returns from the handler, the hypervisor can inject a second `#VC` right before the user program resumes on the CPU. This execution flow is feasible because each `#VC` handler sets up its own stack and tears it down before returning. Thus, the hypervisor can inject two consecutive `#VC`s and use them to perform two different changes to the victim VM. For example, Fig. 2 (a) shows how the hypervisor can change `rax` and then `rcx` by chaining two `#VC`s. The hypervisor can also inject `#VC`s in a staggered fashion, allowing the application to execute a few instructions between the first and the second `#VC`s. In our analysis, we did not find hardware or software limitations on the number of `#VC`s that a hypervisor can inject consecutively or with staggering.

Nesting #VC in non-critical section. The hypervisor can inject a `#VC` while the guest kernel is executing a `#VC` handler, thus causing nested interrupts. This is functionally safe because each `#VC` sets up its own stack. More importantly, from an attack perspective recall that a `#VC` handler changes the state (register or memory) of the code that was executing when the `#VC` was injected. In case of a nested interrupt, the effects of the second `#VC` handler change the state of the first `#VC` handler. As shown in Fig. 2(b), if the second handler effects a change in `rax`, this influences the execution of the first `#VC` handler that uses the modified `rax`. In our experiments, we were able to nest to at least a depth of 3. We report that the hypervisor can also inject consecutive nested interrupts to change various parts of the `#VC` handler (e.g., we inject 2 consecutive nested interrupts of depth 1 in Fig. 2(b)). There is practically no limit to how many consecutive nested `#VC`s the hypervisor can inject.

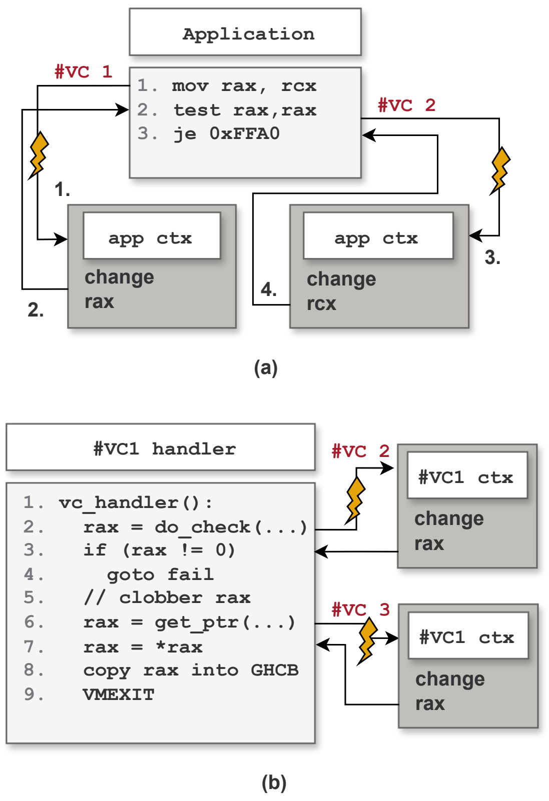
Nesting #VC in critical section. The `#VC` handler implementation has one critical section. In particular, the hypervisor and the guest VM communicate via two shared buffers as part of the GHCB. For correctness, in the case that the hypervisor is benign, the handler synchronizes accesses to these buffers to avoid race conditions with the VM. This creates a critical section in the handler. In typical interrupt handlers, it is standard to disable interrupts when executing a critical section. In the case of the `#VC` handler, to our surprise, we find that the hypervisor can inject nested `#VC`s even in the critical section execution. In our experiments, we were able to achieve a nesting depth of 1 in the critical section for Linux kernel implementation. We investigated if this was an implementation bug or an intentional design choice. Our analysis shows that this is a necessary functionality to handle a particular case where both the operands of a `mov` operations are memory addresses. Interested readers can refer to Appx. A for details.

In summary, there are several cases in the VC handler that we can leverage to change or read registers and VM memory. The ability to chain and nest `#VC`s allows us to achieve a cascading effect akin to return-oriented-programming (ROP).

3.2. Challenges & decisions

If the hypervisor can corrupt memory and registers of the VM, then we can build powerful attacks. Next, we outline the challenges in achieving our goal, especially when the hypervisor has limited access to the VM. We identify the best ways to use `#VC` to mount our attacks and provide rationale for our choices of what to exploit.

Challenge 1: Targeted #VC injection. To perform a meaningful attack using `#VC`, we first need to identify instructions in target programs that would result in a meaningful effect as a result of the `#VC`. In our example from § 2.3, changing the value of `rax` on Line 3 in Lst. 1 leads to the attacker successfully authenticating the target program. Once we have identified the target instruction, we should time the `#VC` such that it is injected just before our target instruction is executed. There are two nuances that we need to address.



```
1. mov rax, rcx
2. test rax,rax
3. je 0xFFA0
```

```
1. vc_handler():
2.   rax = do_check(...)
3.   if (rax != 0)
4.     goto fail
5.   // clobber rax
6.   rax = get_ptr(...)
7.   rax = *rax
8.   copy rax into GHCB
9.   VMEXIT
```

Figure 2. (a) VC chaining. VC1 changes `rax`, VC2 changes `rcx`. (b) 1-level VC nesting.

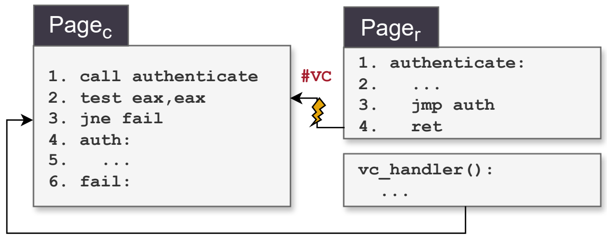
First, when we trick the VC handler into accessing certain pages, it might cause stage-1 or stage-2 page faults. If a stage-1 page fault occurs during VC handling it will crash the VM. Therefore, WESEE has to ensure that this never happens. In contrast, stage-2 page faults are not an issue because the hypervisor can always ensure that the pages with the addresses of interest are paged in when the VC handler executes. Second, some pages in the kernel have limited permissions (e.g., `.text` section is not writable). If our attack attempts to write to these pages, it will cause a page fault and crash the VM. As we will show in § 4.5, WESEE gets around this limitation by changing the page permissions before it triggers an access to such a page, avoiding a crash.

Challenge 2: Constructing primitives using handler effects. The VC handler performs different checks and induces different side-effects depending on the `exit_reason`. For example, the `vmmcall` handler checks the return value of the `perform_VMEXIT` function. This function looks up the `exit_reason` and then performs a series of checks. If the checks pass, it returns OK. For `vmmcall`, the VC handler checks that the hypervisor has written a value into `rax` of the incoming GHCB. Therefore, this reason is controlled by the hypervisor making it easy to get around the check in Line 3 in Fig. 2 (b). However, there might be other checks in the VC handler that access protected memory which cannot be influenced by the hypervisor. For example, the handler looks up the last instruction that was executed and reads the value of the register used as an operand to that instruction. The last instruction and the value of its operand are stored in the application context that is protected and cannot be controlled by the hypervisor. Such checks could crash the handler, thwarting our attack. Therefore, to build WESEE primitives we should carefully choose and chain the handler effects to avoid such crashes.

Insight 1: Using only two `exit_reasons`. While there are more expressive effects of the VC handler (e.g., `rdpmc`, `cpuid`), we find that the handling of intercepts for `vmmcall` and MMIO is sufficient to build powerful attack primitives. Further, handling these intercepts does not have many side-effects (e.g., changes to memory, changes to registers) and checks that would otherwise corrupt execution. So, they can easily be used to build WESEE primitives.

Insight 2: Limiting to kernel memory. The hypervisor can raise `#VC` while executing in both the user and kernel space. Therefore, the hypervisor can use `#VC` to leak or tamper both user and kernel space registers and memory. However, using `#VC` to attack user-space applications is more challenging than compromising the kernel execution. First, since the VC handler is executed in the kernel space, the memory accessed from the handler should be mapped in the kernel. User-space application memory is not mapped as-is in the kernel and therefore the VC handler cannot use the userspace virtual address to access it. The attacker needs to either perform the address translation using the process's page tables or single-step the victim process's lifecycle to track the virtual to guest physical address (GPA) mapping, such that it can supply the GPA. Second, determining when to inject the `#VC` exception for user-space applications is not straightforward. For example, in Lst. 1 the hypervisor should inject the `#VC` exception after Line 3. Determining when this instruction is executed in user-space requires a single-stepping primitive at instruction granularity. In contrast, targeting the kernel space does not incur such challenges. In the kernel space, the pages of the `.text` segment are mapped contiguously during boot and therefore we can easily compute the address and page of the instruction we want to trigger the `#VC` on. Therefore, to determine when to inject the `#VC` exception while executing in the kernel only requires using page faults to profile the pages that are being executed. Since the hypervisor controls the stage-2 page tables of the VMs, profiling with page faults is easy. Thus, we focus on building our attack primitives and case studies using kernel space code.

Insight 3: Target instructions executed after page fault. In SEV, the hypervisor manages the stage-2 page tables for VMs and handles page-faults. We can use this mechanism to induce page faults by marking pages as non-executable in the VM to observe the pages that are executed. We use this page tracing technique to time the `#VC` injection. First, we limit our attack primitives to target instructions that are executed when jumping or returning from another page (e.g., a call instruction that returns execution from another page, the target of a `jmp` instruction). Therefore, we can use the



```
1. call authenticate
2. test eax,eax
3. jne fail
4. auth:
5.   ...
6. fail:
```

```
1. authenticate:
2.   ...
3.   jmp auth
4.   ret
```

```
vc_handler():
  ...
```

Figure 3. Timing the #VC injection. 1. Hypervisor observes page fault sequence [Page$_c$, Page$_r$, Page$_c$] and injects #VC when Line 4 returns. 2. This triggers the execution of the VC handler, which when done returns back to the application on Page$_c$

page tracing technique to detect when the execution returns to our target page and inject a #VC. For example, consider a target program that performs authentication by calling an `authenticate` function and then tests its return value (see Fig. 3). The hypervisor can use #VC with `vmmcall` to change the value of `rax` as explained in § 2.3. Therefore, to corrupt the return value of the `authenticate` function we should target our #VC injection to when the function returns i.e., on the line after the `call` instruction. To identify when the `authenticate` function returns, the hypervisor marks pages Page$_c$ containing the target instruction and Page$_r$ containing a return to the target instruction as not executable. Then, when the hypervisor observes a page-fault trace of the pattern [Page$_c$, Page$_r$, Page$_c$], it knows that this is the return from the `authenticate` function. Therefore, before resuming execution of the target program by marking Page$_c$ as executable again, the hypervisor injects the #VC. While this choice limits the instructions that WeSee can target, we show that it is sufficient to build powerful primitives and mount attacks on VMs.

ASLR. The kernel uses different Address Space Layout Randomization (ASLR) to thwart attacks that rely on deterministic addresses. To use our page-tracing technique and build attack primitives that target specific addresses we need to break these address randomization techniques. We use insights from previous works that have explained techniques to defeat the kernel's physical ASLR. This does not require #VC or other WeSee primitives. We use WeSee primitives to defeat virtual address space ASLR (see § 5.2).

3.3. Threat model

We assume an untrusted hypervisor creates and launches AMD SEV-SNP VMs. The trusted hardware generates an attestation report. AMD SEV-SNP's trusted hardware ensures that all memory and registers of the VM are protected and encrypted. Further, on context switches the trusted hardware saves and restores the context of VMs. It also ensures that all register states are cleared before resuming execution of other untrusted code. We assume that all trusted software in the VM, including the VC handler, and hardware are free from bugs. To communicate with the hypervisor, the VM sets up shared-memory (GHCB) according to AMD specification. We assume that the VM uses the GHCB strictly in accordance with the AMD specification and is bug free. The hypervisor is still responsible for interrupt delivery and memory management for the VMs according to AMD specifications, and can observe page faults. We assume that all cryptographic algorithms used by AMD SEV-SNP are secure. We do not assume any other architectural, microarchitectural, power, or voltage side-channels.

4. WeSee Primitives

We build basic primitives either by using one #VC's effects or by cascading #VCs. Our main challenge is to use the #VC to induce the desired effect while avoiding the #VC's undesired checks and effects. For example, using `exit_reason` as `vmmcall` and writing to `rax` requires passing certain checks as explained in § 3.2. First, we build a primitive to skip instructions using the #VC handler's effect (§ 4.1). Next, we observe that the #VC handler leaks `rax` and writes to `rax` when `exit_reason` is set to `vmmcall`. With these effects we build primitives that read and write to `rax` (§ 4.2). Further, we use these primitives to induce desired effects (e.g., change `rax` to an attacker controlled value) and avoid undesirable effects (e.g., skip checks that would otherwise jump to a `fail` block) while building more powerful primitives. Further, we build primitives to read and write to kernel memory using MMIO read and write handling effects in the #VC handler (§ 4.3 and § 4.4). Finally, we show how WeSee uses these primitives to inject arbitrary code into the kernel (§ 4.5).

4.1. Skipping Instructions

We build a basic primitive S that uses #VC to skip over an arbitrary instruction during the VM's execution. Such a primitive can be used to bypass checks and negate undesired effects in a target program (e.g., to skip over an instruction that jumps to a `fail` block).

Skip one instruction. During normal operation, the CPU does not move the instruction pointer past the instruction that caused #VC, to give the application an opportunity to retry the instruction if it fails. Therefore, in Fig. 1(a) the return from the VC handler would result in the `vmmcall` instruction executing again on Step 5. Therefore, the VC handler always has an effect which advances the instruction pointer. Depending on the type of the #VC, the VC handler could have other effects (e.g., updates to registers, writes to memory). An attacker can induce the effect in the VC handler that increments `rip` and use #VC to build a skip primitive (S) that skips arbitrary instructions.

WeSee should ensure that the skip primitive does not have any other undesired effects (e.g., changes to register values) that can corrupt the target application's execution. Therefore, we use a #VC with `exit_reason` set to `vmmcall` to build this primitive. Handling the `vmmcall` has only one other effect besides incrementing `rip`; it leaks the values of `rax` to the hypervisor and writes a value from the hypervisor to `rax` as shown in Fig. 4 (Lines 3, 6). Because the `vmmcall` handling writes the value from the hypervisor to `rax`, the attacker can always ensure that this effect does not corrupt any state in the application. Specifically, when the VC handler exits to the hypervisor, the hypervisor first reads the value of `rax` from the GHCB. Then, it copies this value back to the GHCB to be read by the VC handler for `vmmcall`. This ensures that the value in `rax` remains unchanged. Next, the VC handler only increments the `rip` if the call from the hypervisor was successful i.e., for `vmmcall` the hypervisor wrote a value into the `rax` field of the GHCB (Line 11 in Fig. 4). Therefore, for `vmmcall` this is fully controlled by the hypervisor. Specifically, by copying the value of `rax` into the GHCB, the hypervisor also ensures that the function on Line 4 always returns OK. The VC handler does not increment the `rip` by a fixed number of bytes. Instead, it looks up the last instruction that was executed from the program context. Then, it computes the size of that instruction and the number of bytes to increment (Line 10) ensuring that a full instruction is always skipped.

VC chaining: Skip n instructions. The hypervisor can chain the skip primitive to skip any number of consecutive instructions. For this, it pauses the execution of the target program by marking the page that the VC handler returns to as non-executable. This ensures that every time the VC handler returns to the target program, a page fault is generated. The hypervisor then uses the skip primitive to skip 1 instruction on each page fault of the program. The VC handler looks up the current instruction that `rip` points to (Line 10 in Fig. 4), which ensures that each subsequent instruction is skipped correctly irrespective of its length.

4.2. Read & Write `rax`.

Using malicious `#VC`s, we build WESEE primitives to read and write to `rax`. Return values from function calls are stored in `rax` which makes it particularly interesting. Therefore, leaking or tampering the value of `rax` can be used to construct powerful attacks.

Read & Write `rax` and skip 1 instruction. As explained in § 2.3, a malicious hypervisor can use the `#VC` to read and write the `rax` register at any point during the instruction execution. Further, as discussed in § 4.1, the VC handler has an effect of incrementing the `rip`. We build attack primitives $R_{rax}S$ and $W_{rax}S$ that read and write to `rax` respectively and then skip 1 instruction. To build these two primitives, we simply inject one `#VC` with `exit_reason` as `vmmcall`. This will leak the value of the target program's `rax` (Line 3 in Fig. 4) and copy a value from the hypervisor into the `rax` of the target program (Line 6 Fig. 4) and skip one instruction. Even though we define $W_{rax}S$ as a separate primitive, the VC handler always leaks the value of `rax` (Line 3 in Fig. 4) and therefore $W_{rax}S$ implicitly always reads `rax` as well.

Read & Write `rax` without skipping. The effect of incrementing the `rip` in $R_{rax}S$ and $W_{rax}S$ is sometimes not desirable while building attacks where we want to execute the instruction that we injected the `#VC` on. To negate the `rip` increment effect we build 2 more primitives R_{rax} and W_{rax}

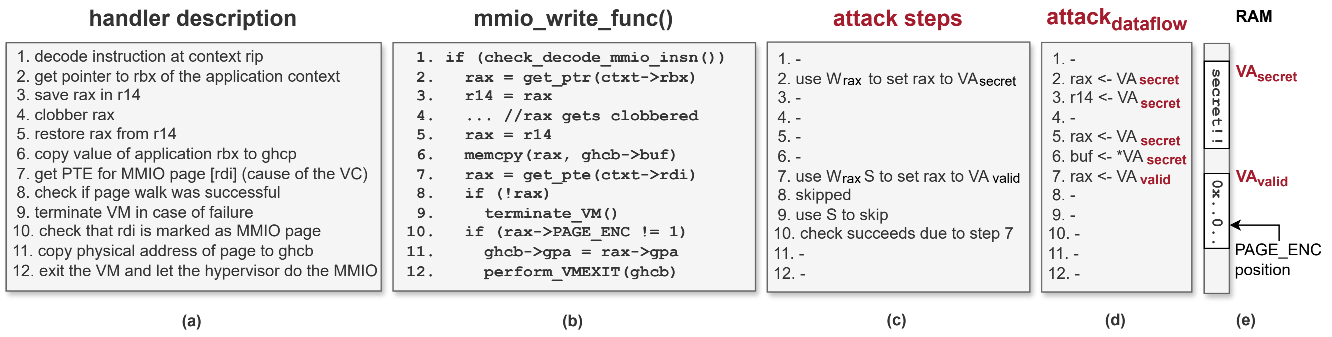
```
 1. switch(exit_reason):
 2.   case vmmcall: desired side-effect to leak rax
 3.     ghcb_set_rax(ghcb, ctxt->regs->ax) //write rax to ghcb
 4.     ret = perform_VMEXIT(ghcb)
 5.     if (ret == OK) desired side-effect change rax
 6.       ctxt->regs->ax = ghcb->rax; //read rax from ghcb
 7. ... //function call
 8. switch (ret) //ret is return value of exit_reason handling
 9.   case OK:                                      |  15
10.     n = compute_size(ctxt->rip)                 |  instructions
11.     ctxt->rip = ctxt->rip + n                   |  skip using
12.   case ...: ...                                 |  15 S
13.   case RETRY:                                   v
14.     //no rip increment
15. return ret; //instruction return (iret)
```

Figure 4. Psuedo-code of VC handler with effects.

that read and write to `rax` respectively without skipping an instruction. Negating the undesired `rip` increment effect is not straightforward. Notice that the `rip` is not incremented if the return value on Line 4 in Fig. 4 is RETRY (Line 13). However, the hypervisor cannot force this condition on Line 4 for `vmmcall` handling. Instead, to negate the instruction skipping effect, we construct a mechanism that skips all instructions (15 instructions) starting from Line 8 to Line 13. This ensures that the VC handler executes the RETRY case even though the return value on Line 4 was OK. With this, the desired effect on Line 3 for R_{rax} and on Line 6 for W_{rax} is preserved and the undesired effect on Line 11 is negated. To skip these instructions, we use a single-level of nesting using the skip primitive (S). We chain the skip primitive 15 times in succession. Therefore, $(R_{rax}S).15S \equiv R_{rax}$ and $(W_{rax}S).15S \equiv W_{rax}$. This ensures that the `rip` is not incremented and the VC handler simply returns back to the target program. To decide when to inject the first S primitive in the chain, we notice that the VC handler performs a call to a function (Line 7) before our target instruction on Line 8. Therefore, we use our page-trace mechanism to inject the first S on the return of this function call.

4.3. Reading kernel memory

The attacker can use `#VC`s to read values from any kernel memory (R_{mem}). To successfully read any kernel memory of the attacker's choosing, the attacker needs 2 capabilities: (a) control a pointer to kernel memory, and (b) dereference the attacker-controlled pointer and copy the value to the GHCB. We observe that the attacker can use the VC handler's MMIO write case to gain these capabilities. During normal operation, the CPU invokes the `#VC` for MMIO write when a `mov` instruction copies a register value to a memory-mapped region (e.g., `mov [rdi], rbx` where `[rdi]` points to a memory-mapped region). To enable the hypervisor to perform the actual MMIO write, the handler copies the value of the target program's register into the GHCB as shown in Fig. 5 (Line 6). To perform the memory copy, the handler first fetches a pointer to the program's context on stack (Line 2) and then de-references it (i.e., `memcpy` dereferences `rax`). Our main intuition is that we can use the primitives that write to `rax` to gain control


handler description

1. decode instruction at context rip
2. get pointer to rbx of the application context
3. save rax in r14
4. clobber rax
5. restore rax from r14
6. copy value of application rbx to ghcp
7. get PTE for MMIO page [rdi] (cause of the VC)
8. check if page walk was successful
9. terminate VM in case of failure
10. check that rdi is marked as MMIO page
11. copy physical address of page to ghcb
12. exit the VM and let the hypervisor do the MMIO

(a)

mmio_write_func()

```
 1. if (check_decode_mmio_insn())
 2.   rax = get_ptr(ctxt->rbx)
 3.   r14 = rax
 4.   ... //rax gets clobbered
 5.   rax = r14
 6.   memcpy(rax, ghcb->buf)
 7.   rax = get_pte(ctxt->rdi)
 8.   if (!rax)
 9.     terminate_VM()
10.   if (rax->PAGE_ENC != 1)
11.     ghcb->gpa = rax->gpa
12.     perform_VMEXIT(ghcb)
```

(b)

attack steps

1. -
2. use W_{rax} to set rax to VA_{secret}
3. -
4. -
5. -
6. -
7. use W_{rax} S to set rax to VA_{valid}
8. skipped
9. use S to skip
10. check succeeds due to step 7
11. -
12. -

(c)

attack$_{dataflow}$

1. -
2. rax <- VA_{secret}
3. r14 <- VA_{secret}
4. -
5. rax <- VA_{secret}
6. buf <- *VA_{secret}
7. rax <- VA_{valid}
8. -
9. -
10. -
11. -
12. -

(d)

(e)

Figure 5. Read memory primitive. (a) Handler description for `#VC` caused by instruction `mov [rdi], rbx`. (b) Pseudo-code of MMIO write handling. (c) Attack steps. (d) Attack dataflow. (e) Memory used for attack.

of the pointer in the VC handler before it is dereferenced. When the pointer is dereferenced, the VC handling for MMIO writes will copy a value from the attacker-controlled pointer to the GHCB. This satisfies the 2 capabilities the attacker needs to successfully read kernel memory. Next, we explain the benign MMIO write case in the VC handler shown in in Fig. 5 (a) and (b).

Benign execution of VC handler for MMIO write. To ensure that the hypervisor can perform the MMIO write operation correctly, the VC handler's MMIO write case copies the value of the register to write and the guest physical address of the MMIO region to the GHCB. To determine the register and the region of memory, the handler first decodes the last instruction that was executed in the program's context which caused the `#VC` (Line 1 in Fig. 5). The handler has these checks because it expects the instruction that caused the `#VC` to be a `mov` instruction that performs a write to memory. The code-snippet in Lst. 2 shows this check (Line 3). Further, once the instruction is correctly decoded, it also sets the operands of the instruction (Line 6) in the VC handler's stack.

```
1 check_decode_mmio_insn():
2   type = decode(ctxt -> insn)
3   if (type != MMIO_WRITE)
4     goto fail
5   // sets the instruction operands
6   get_insn_operands()
```

Listing 2. Pseudo-code for check and decode mmio instructions during MMIO write handling

After determining the operands, the VC handler needs to copy the value of the register to the GHCB. When an exception occurs the hardware pushes the context of the program that caused the exception on the exception handling stack. Therefore, to get the value of the register, the VC handler fetches a pointer to the register in the program's context and uses `memcpy` to copy the value in the register to the GHCB's buffer (Line 6 in Fig. 5(b)). Then, to determine the GPA of the memory-mapped region, the handler performs a page-walk to get the page table entry (Line 7 in Fig. 5(b)). It checks that the page table entry is valid (Line 8) and that the memory-mapped region is in plaintext and in hypervisor-accessible memory (Line 10). It finally exits the VM to go to the hypervisor. In summary, handling the MMIO write has 2 side-effects : (a) dereference a pointer to memory and copy its value to the GHCB, and (b) copy a valid GPA of an unencrypted memory region to the GHCB.

The first effect that dereferences the pointer in memory and copies the value to the GHCB is desirable and can be used to gain the 2 capabilities that the attacker needs. However, for R_{mem} the checks and effects that result from copying a valid GPA to the GHCB need to be negated.

Building the read-memory primitive (R_{mem}). To build the read-memory primitive, we start with a `#VC` with `exit_reason` set to `mmio` which triggers the VC handler for MMIO write. We need to ensure that the 2 capabilities that the attacker needs to read kernel memory are satisfied. The attacker should control a pointer while handling MMIO, and this pointer should be dereferenced and copied to the GHCB. We can achieve this by writing an attacker-controlled pointer address into `rax` when the function on Line 2 in Fig. 5 returns. The attacker-controlled pointer in `rax` is then dereferenced by the `memcpy` on Line 6 which copies the value (8 bytes) to the GHCB buffer. The `rax` is first saved to `r14` (Line 3) and then restored (Line 5) before performing the `memcpy`. Therefore, it is important for our primitive that the write to `rax` on Line 2 does not skip the next instruction on Line 3. Therefore, we use the W_{rax} primitive to change the value of `rax` when the `get_ptr` function returns (Line 2). With this we have achieved the desired effect of controlling a pointer to memory, and correctly copying it to the GHCB.

However, handling the MMIO write has another undesired effect. It performs a page-walk (Line 7) and checks the page table entry (Line 8, 10) before performing a `vmexit`. If the page-walk or the checks fail, the handler returns an error and does not perform the `vmexit` to the hypervisor. Therefore, in our read-memory primitive, we need to ensure that the side-effects and checks are negated and the VC handler sucessfully exits to the hypervisor. We observe that the result of the page walk on Line 7 is saved into `rax`. We can use our primitive that writes to `rax` again to change the value of `rax` such that it passes the checks on Lines 8 and 10. Here, we opt to nest the primitive that writes to `rax` and skips 1 instruction ($W_{rax}S$). This implies that the instruction that performs the check on Line 8 is skipped. However, to fully bypass the check we should ensure that one more instruction on Line 9 is skipped using the skip primitive (S). With this, we have successfully bypassed the first check. The second check (Line 10) ensures that the memory-mapped region is in unencrypted memory by checking the `PAGE_ENC` bit of the page table entry. This is a simple memory dereference at an offset from the base address in `rax`. Therefore, we identify a region of memory in the kernel such that the value at this offset is always 0. We set it as the address in `rax` before `get_pte` returns on Line 7. This will guarantee that the check on Line 10 will pass. The `rax` is dereferenced once more on Line 11 to write the GPA to the GHCB buffer. However, for our R_{mem} this value is not important. Therefore, MMIO write handling will gracefully exit to the hypervisor.

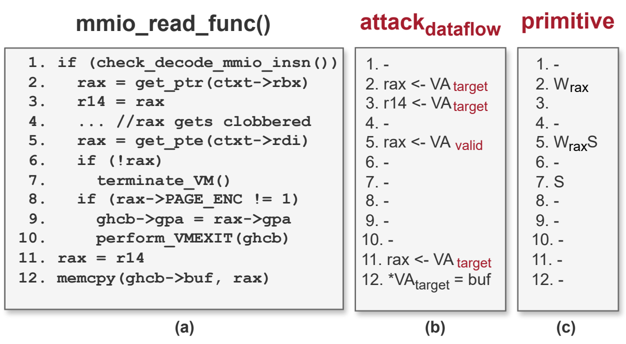
Fig. 5 only shows the pseudo-code of the VC handler for MMIO write. For simplicity, we show the checks on Line 10 to be inlined after the check in Lines 8 and 9. In the Linux implementation, the check represented on Line 10 is in a different function. Therefore, we cannot use a chain of skip primitives to skip all instructions from Lines 9–12 to perform the `vmexit`. Instead, we detect the return from the functions using our page fault tracing method and then inject subsequent `#VC`s as described above.

Target instruction for R_{mem}. We identify a target `mov` instruction that writes register values to memory in the scheduling subsystem of the kernel. This instruction is on the hot path of the kernel's execution and is frequently executed. Furthermore, this instruction is executed after a `jmp` from another page. Therefore, we use the page fault profiling as explained in § 3.2 to identify the page fault sequence and time the `#VC` with `exit_reason` as `mmio` to trigger the read-memory primitive (R_{mem}).

Reading n bytes of kernel memory. R_{mem} only reads 8 bytes of memory from the kernel. To read more memory, we pause the execution by marking the page of our target instruction as non-executable. Then on each page fault of our target instruction, we use R_{mem}. Using this, we can chain R_{mem} to read kernel memory in 8 byte chunks.

4.4. Writing to kernel memory

We can use `#VC` to build a primitive that writes arbitrary data to arbitrary addresses in the kernel memory (W_{mem}). For this, the attacker needs 2 capabilities, similar to the read primitive: (a) control a pointer to kernel memory, and (b) the attacker-supplied value in the GHCB should be copied to the attacker-controlled pointer. We observe that the hypervisor can achieve these capabilities using the MMIO read case. During normal operation, this handler is triggered on a `mov` instruction that reads a value from memory into a register. Therefore, similar to the MMIO write case, the read case has 2 effects: (a) it fetches a pointer to the program context and then copies the value from GHCB into the register, and (b)



```
 1. if (check_decode_mmio_insn())
 2.   rax = get_ptr(ctxt->rbx)
 3.   r14 = rax
 4.   ... //rax gets clobbered
 5.   rax = get_pte(ctxt->rdi)
 6.   if (!rax)
 7.     terminate_VM()
 8.   if (rax->PAGE_ENC != 1)
 9.     ghcb->gpa = rax->gpa
10.     perform_VMEXIT(ghcb)
11. rax = r14
12. memcpy(ghcb->buf, rax)
```

1. -
2. rax <- VA_{target}
3. r14 <- VA_{target}
4. -
5. rax <- VA_{valid}
6. -
7. -
8. -
9. -
10. -
11. rax <- VA_{target}
12. $*VA_{target}$ = buf

1. -
2. W_{rax}
3.
4. -
5. $W_{rax}S$
6. -
7. S
8. -
9. -
10. -
11. -
12. -

Figure 6. Write memory primitive. (a) Pseudo-code of MMIO read handling. (b) Attack dataflow. (c)WESEE primitives used for attack

it copies a valid GPA to the GHCB. This allows the benign hypervisor to read data from the memory-mapped region, send it to the VC handler which then copies the data into the register.

To achieve the attacker's objectives of writing to an arbitrary address in the kernel, WESEE first uses the W_{rax} primitive to get hold of an attacker-controlled pointer (Line 2 in Fig. 6). This pointer is then dereferenced (on Line 12) when the handler copies attacker supplied data from the GHCB into the attacker-controlled pointer address. Finally, as with R_{mem}, WESEE chains $W_{rax}S$ and S to negate the undesirable checks and side effects on Lines 6 and 8. With this, the value from the hypervisor (8 bytes) in the GHCB is copied into a kernel memory location that is controlled by the hypervisor.

Target instruction for W_{mem}. We identify a target `mov` instruction in the kernel's scheduling subsystem, that writes a value from memory to a register and is also the first instruction executed after a return from another page. Therefore, we time the injection of W_{mem} using our page trace mechanism. Further, to write more than 8 bytes to kernel memory, we pause the execution of the target instruction by marking the page as non-executable. Then we chain W_{mem} to write to kernel memory in 8 byte chunks.

4.5. Arbitrary code injection

Using WESEE primitives explained so far, we can perform arbitrary code injection in the kernel. To inject code to be executed in the kernel, we need to write to the `.text` section. By default, the kernel sets up its page tables such that the `.text` section is executable but not writable. To get around this constraint, we first use our R_{mem} and W_{mem} primitives to change the permissions in the kernel page tables. For this, we first read the value of the kernel's CR3 using the read memory primitive. This value indicates the base address address of the kernel's page tables. We identify the virtual address of the target page where we want to inject our code in the kernel's `.text` section. We then use R_{mem} to perform a page walk starting from the base address to this page. At each level of the page table, we use W_{mem} to change the permission of the page to be writable. Once

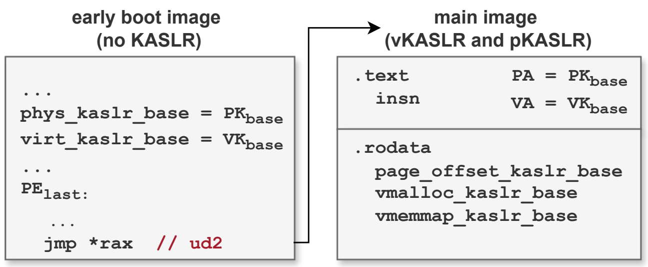


Figure 7. Kernel boot images and KASLR. The red instruction is inserted by the hypervisor for offline profiling.

all the permissions are changed, our target page is writable. Finally, we can write shell code using a chain of W_{mem} to the target page. It remains to show how we execute our injected code in the VM. This depends on where the code is injected as shown in our case studies (§ 6).

5. Bypassing ASLRs

Address Space Layout Randomization (ASLR) is a simple way to stop attacks that rely on deterministic addresses. Linux uses ASLR for both physical and virtual addresses.

5.1. Physical kernel ASLR (pKASLR)

The physical ASLR protection in the kernel (pKASLR) randomizes the physical address where the kernel is loaded for every kernel boot. Previous works explain techniques to break pKASLR of SEV-ES VMs using page faults observed by the hypervisor [19], [20]. We use insights from these works to build an attack that compromises the pKASLR of SEV-SNP VMs. To compromise pKASLR we do not need any WESEE primitives but just the page fault tracing technique in the hypervisor.

Overview. The pKASLR base is the address of the first page of the main kernel image (PK$_{base}$). The kernel memory is allocated contiguously from this base address. So, the address of every page in kernel memory is a fixed offset from PK$_{base}$. Therefore, to compromise pKASLR we simply need to determine the value of PK$_{base}$. To do this, we analyze the Linux kernel's boot process. We find that before the main kernel image is loaded, an early boot image sets up PK$_{base}$ by choosing a random address [20]. Then, it loads the main kernel image at this random base address and jumps to it (see Fig. 7). Crucially, this early boot image which sets up the base address is not subject to any ASLR protection. Therefore, for every execution of the Linux kernel, the physical addresses of the early boot image remains constant. We can use this insight to find PK$_{base}$ by profiling the kernel boot. Specifically, if we determine the address of the last page that the early boot image executes (PE$_{last}$), then the very next page that is executed will always be PK$_{base}$ (see Fig. 7).

Attack steps. To find PE$_{last}$, we modify the early boot image and replace the `jmp *rax` in Fig. 7 with a dummy instruction `ud2`. This step can be performed by the hypervisor offline without the victim's knowledge. We boot this modified kernel image and induce page-faults for each page executed during kernel boot. This enables us to gather a page trace of all the pages executed during the kernel boot and their corresponding physical addresses. Using this page trace we determine the physical address of the last page (PE$_{last}$) that is executed by the early boot image. Because the early boot image is not subject to ASLR, PE$_{last}$ is constant across all boots of the Linux kernel.

Next, we boot the unmodified victim kernel image that contains the jump from the early boot image to the main kernel image (see Fig. 7). Now, the page executed right after PE$_{last}$ is the physical address of the main kernel image (PK$_{base}$) which we want to de-randomize. Therefore, we can deterministically leak the value of PK$_{base}$ compromising pKASLR.

5.2. Virtual Kernel ASLRs with WESEE Primitives

Like physical ASLR, the Linux kernel randomizes the addresses of its virtual address space using virtual ASLR (vKASLR). Once we have compromised pKASLR, we can compute the physical address of any function in the kernel. Our main insight to compromise virtual ASLR (vKASLR) in the kernel is to find an instruction during the kernel's execution that loads the virtual address of a kernel's function into `rax`. At this point, we can use our R_{rax} primitive, to leak the virtual address of the function. We observe that the kernel sets up its virtual address space such that the kernel memory is contiguous in the virtual address space. Therefore, we can compute the base of the virtual kernel ASLR (VK$_{base}$) as a fixed offset from the function's virtual address that we leak.

We analyze the kernel and find an instruction in the `secondary_startup_64` routine that loads the virtual address of the function `x86_64_start_kernel` into `rax` (Line 2 in the code snippet below).

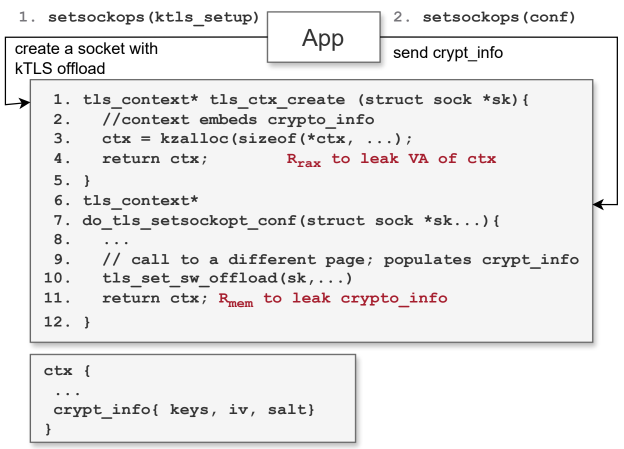
```
1 xor  ebp, ebp
2 mov  rax, [rip + offset]
3 push rax
4 ret
```

We inject W_{rax} right after `ret` on Line 4 to leak the virtual address of `x86_64_start_kernel`. Using this, we compute the value of VK$_{base}$ as a fixed offset from the `x86_64_start_kernel` function.

The Linux kernel randomizes regions for the identity map (`page_offset`), `vmalloc`, and `vmemmap` separately [21]. The randomized base address for these regions is stored in the `.rodata` section that is only subject to vKASLR. Therefore, once we have compromised vKASLR, we can compute the virtual addresses of all the base addresses and use R_{mem} to leak their values.

6. Case studies

We present 3 case studies that use WESEE primitives. Our case studies do not need profiling beyond page faults.



```
 1. tls_context* tls_ctx_create (struct sock *sk){
 2.   //context embeds crypto_info
 3.   ctx = kzalloc(sizeof(*ctx, ...);
 4.   return ctx;          R_rax to leak VA of ctx
 5. }
 6. tls_context*
 7. do_tls_setsockopt_conf(struct sock *sk...){
 8.   ...
 9.   // call to a different page; populates crypt_info
10.   tls_set_sw_offload(sk,...)
11.   return ctx; R_mem to leak crypto_info
12. }
```

```
ctx {
 ...
 crypt_info{ keys, iv, salt}
}
```

Figure 8. kTLS offload in the Linux kernel.

6.1. Leaking keys from Kernel TLS

The Linux kernel allows userspace applications to offload TLS computation to the kernel. This boosts performance by reducing the number of data copies from user to kernel space. To use the kernel TLS offload, the userspace uses the system call `setsockopt` to indicate to the kernel to hook on packets sent through a particular socket (Step 1 in Fig. 8). Then, the userspace application performs the TLS handshake to setup symmetric session keys. Once the symmetric session keys are setup, the application transfers the keys along with all the information required by the kernel (session key, IV, salt, and sequence number) to encrypt and decrypt the session packets. The application sends the `struct crypto_info` using the `setsockopt` system call (Step 2) to send this data to the kernel. We aim to leak the `crypto_info` using R_{mem} to read this struct such that we can encrypt and decrypt any session packet compromising the TLS session.

To perform an end-to-end attack, we need to find the virtual address of the `crypto_info` struct for the target socket. The userspace first creates the socket and establishes the kernel TLS using `setsockopt` (Step 1 Fig. 8). On receiving this system call, the kernel allocates kernel memory for the socket context using `kzalloc` which returns a pointer to the memory in `rax` (Line 2). The context structure contains the `crypto_info` struct for the socket as shown in Fig. 8. Therefore, we leak the virtual address of the socket's context using the read `rax` primitive (R_{rax}) and compute the virtual address of `crypto_info` as VA$_{ci}$. At this point, the userspace application has not yet transferred the keys and other sensitive information to the kernel. Therefore, we have to wait for the userspace application to transfer the session keys and other information to the kernel using a second `setsockopt`. When the userspace invokes this system call with the values (Step 2 in Fig. 8) the kernel copies the information with the session keys to VA$_{ci}$ (Line 6). After this, we can leak the `crypto_info` using R_{mem}. To trigger R_{mem}, we observe that after storing the values into `crypto_info` the system call handling in the kernel calls another function (Line 7) which is on a different page. So, we use the page-fault technique to inject R_{mem} on Line 11 and read out the `crypto_info` struct. To verify that this information is sufficient to compromise the TLS session, we record all packets for the session and check that we can decrypt them correctly using the leaked information.

6.2. Disabling Firewall

The `iptables` utility allows system administrators to setup packet filter rules for incoming and outgoing traffic to create firewalls using the Linux kernel. While the `iptables` rules are configured in the userspace, the actual packet filtering is performed in the kernel. The kernel invokes the `nf_hook_slow` shown in Lst. 3 for the filtering.

```
endbr64
movzwl eax, [rdx]
cmp    ecx, eax          ; ... do packet filtering
```

Listing 3. Exiting implementation of `nf_hook_slow`

We aim to prevent the kernel from filtering the packets and compromise its firewall protections. To perform this attack, we can use WeSee's code injection from § 4.5 and replace the code of the `nf_hook_slow` function. Specifically, we inject shell code showin in Lst. 4 at the virtual address of the `nf_hook_slow` function.

```
1 endbr64
2 mov $1, rax
3 ret
```

Listing 4. Code to inject in place of `nf_hook_slow`

Our code simply ensures the function returns success (1) without performing any filtering. We have to always inject 8 bytes at a time because we use the W_{mem} primitive. Therefore, while it would have been sufficient to inject Lines 2 and 3 to achieve our objective, we inject Line 1 to satisfy the alignment of 8 bytes. We inject all 3 lines of shell code as 16 bytes using 2 W_{mem}. To verify that our code injection works as expected and prevents the kernel from performing the packet filtering, we setup a victim VM with `iptable` rules that drop all incoming and outgoing packets in the VM. This implies that the firewall blocks all network traffic to the VM. Then, we launch our attack, inject the shell code, and send ICMP packets to the VM from the hypervisor and observe that we receive responses from the SEV VM. This shows that our code injection was successful, and any firewall protection setup using `iptable` rules are bypassed in the VM.

6.3. Gaining a Root Shell in the SEV-SNP VM

The kernel exposes an API called `call_usermodehelper` to kernel modules. This API allows kernel modules to start a user space application from the kernel context. This API takes as arguments an executable to run in the user space along with arguments and environment variables for the executable. When invoked by a kernel module, it starts a process with the executable in user space with root privileges. For example, executing the command below from `call_usermodehelper` spawns a root shell in user space on the VM. This shell takes as input all network data from port 8001. Therefore, if executed, this command allows all unauthorized remote adversaries root access to the VM.

```
/bin/bash -c rm /tmp/t; mknod /tmp/t p;
/bin/sh 0</tmp/t \| nc -ln 8001 1>/tmp/t
```

To attack a victim VM, we aim to maliciously trick it into executing `call_usermodehelper` with this command as the argument. To do this, we generate shell code that executes this command using the `call_usermodehelper` API. We choose to inject this shell code in the Linux kernel's function that receives and handles ICMP packets (`icmp_rcv`). To inject the shell code we compute the virtual address of the `icmp_rcv` function and use the WeSee code-injection from § 4.5 which overwrites the function's code. Now, we send an ICMP packet to trigger our shell code in the SEV VM. When the shell code calls the `call_usermodehelper` API, it creates a new userspace process with root privileges that provides a root shell that listens on port 8001. Then, to interact with the spawned shell we connect to the VM from the hypervisor using `netcat`. There is no authentication or firewall, so our attack connection succeeds and we get a root shell using WeSee.

At this point, we can not only leak all private keys on the file system (e.g., SSH private keys) but also install keys of our choice, manipulate userspace programs, and corrupt stored data [22], [23], [24], [25]. While WeSee does not attack user space by choice (§ 3.2), it achieves the same goals by compromising the kernel space.

7. Implementation & Evaluation

Experimental Setup. We perform our experiments on an AMD EPYC 9124 16-core 3.7 GHz processor with 192GiB RAM with SEV-SNP Gen 4 with microcode version `0x0a10113e`. We boot Linux kernel v6.5.0 [26] with Ubuntu 20.04.6 LTS as guest using QEMU v8.0.0. On the host we use the same kernel with Ubuntu 23.10 as disk-image. For the rest of this section, we report measurements averaged over 3 experiments.

Injecting #VC. To inject the `#VC` with `exit_reason` for WeSee we modified the KVM subsytem of the host kernel with 17 LoC. We inject multiple `#VC` with `exit_reason` set to `vmmcall` and measure the time. One round-trip from the hypervisor takes 3.19 μseconds on average.

WeSee **primitives.** We first build our page tracing mechanism by exposing an API in the host kernel to interact with its stage-2 page table management system with 338 LoC, similar to SEV-STEP [23]. Our changes allow us to mark pages as not executable and disable huge pages in the stage-2 page tables. Then, we implement a function in the host Linux kernel that uses the page-fault trace to detect specific page-fault patterns and exits from the VM and determines when to use WeSee primitives. WeSee needs to inject 1, 34, and 34 `#VC`s for S, R_{mem}, and W_{mem} primitives respectively. Further, WeSee can perform on average 216450 instruction skips/s, 9.25 memory reads/s, and 8.95 memory writes/s.

TABLE 2. Case studies.

Case study	no. of R_{mem}	no. of W_{mem}	no. of `#VC`	no. of page faults
kTLS	8	0	288	2115
firewalls	4	5	238	1474
root shell	4	52	2891	7796

ASLR. To break physical KASLR we profile the boot stage and record on average 568960 page faults for the early boot image with the last page address as `0x7a753000`. To compromise vKASLR we use one `#VC` for one R_{rax}.

kTLS. To leak session keys from kernel TLS we use the WeSee primitives to implement a host kernel module (+220 LoC) and modify the host kernel (+428 LoC). We compile NGINX v1.25.3 with kTLS support in OpenSSL v3.2.0 and host a website in the SEV VM. We then access the website from a remote machine on the same network. Client and server use a standard TLSv1.3 connection with a standard and secure cipher suite TLS_AES_256_GCM_SHA384 for communication. We capture the packets for this connection using Wireshark. We write a utility to manually decrypt the captured packets in nodejs using `crypto_info` that we leak from the SEV VM as explained in § 6.1. From our experiments, we see that WeSee needs 5.97 seconds to recover the full `crypto_info` from when the client offloads the connection to kTLS to when we leak the session key. Further, WeSee uses 1 R_{rax} and 8 R_{mem} to leak `crypto_info`.

Firewall. We manually craft 16 bytes of shell code that we inject to the VM. Recall that the page we target with our code injection is in the `.text` section and does not have write permissions set in the page table. To edit the page permissions, we perform 3-level page walks and use W_{mem} to change the permissions which takes on average 2.2 seconds. For both our case studies which perform code-injection, our target address is always in a 2MiB page. Therefore, we only do a 3-level page walk to the page middle directory (pmd) even though the kernel uses 4-level paging. Then, we inject our shell code to the target page which takes 0.36 seconds on average. We setup the VM with firewall rules that block all network traffic. Finally, to test our shell code we use the `ping` utility in the host to send ICMP packets to the VM and observe that we receive responses indicating that our attack was successful.

Root shell. To get a root shell on the SEV VM, we use GCC and objdump to generate 392 bytes of shell code. After identifying the virtual address of the `icmp_rcv` function, we perform a 3-level page walk that takes 3.1 seconds on average. Injecting this shell code takes 8.1 seconds on average. This shell code is significantly larger than what we needed to disable the firewall and therefore takes longer to inject. To trigger our shell code, the execution should jump to the `icmp_rcv` function. To do this, we use the `ping` utility to send an ICMP packet to the VM. Then we use a `netcat` client to connect to and use the root shell.

Accuracy. To determine when to inject `#VC`, we find functions that are on different pages (see § 3.2). This technique ensures that our injections always work and the `#VC` handler is executed as expected. However, this function alignment can change with different kernel versions. For the other versions, an attacker would either need to find different target functions or use single-stepping to target the `#VC` [15], [23]. In this case, the accuracy of the attack depends on the newly identified functions or the single-stepping framework.

8. Impact Beyond Linux

Besides Linux, we analyze four open source `#VC` implementations in different projects (Enarx [27], Oak [28], Coconut SVSM [29], and Mushroom [30]) and two closed source OSes with AMD SEV-SNP support from Microsoft.

8.1. Open Source Implementations

All four open source projects implement at most 3 out of the total 19 `exit_reasons` listed in Tab. 1. Thus their attack surface is smaller than the Linux `#VC` handler.

Coconut SVSM is the official implementation recommended by AMD that is supposed to be executed in VMPL0. As of 4th April 2024, it only supports 3 `exit_reasons`: `TRAP`, `CPUID`, and `IOIO`. While these implementations have side effects on the execution state, Coconut SVSM decodes the instruction that causes the `#VC` and faults if it is not a valid instruction that can potentially raise a `#VC` (e.g., `cpuid`). Thus, the attacker can only inject a `#VC` when the victim is executing an instruction to ensure that the instruction decoding succeeds. Since the instruction decoding and the `error_code` are not linked, one can launch WeSee attacks. In our experiments, we could inject a `CPUID` exception in a location where the processor would normally raise an `IOIO` exception. This way we were able to corrupt the register state of the application.

Enarx only implements the `CPUID` exception. The handler decodes the `#VC` generating instruction and compares it to the `cpuid` opcode. If these do not match, it terminates the VM. Thus, we conclude that the Enarx is not vulnerable.

Oak only handles `CPUID` exception, similar to Enarx. However, Oak does not decode the `#VC` generating instruction and will always execute the handler and increment `rip` by 2. The attacker can use this to selectively skip instructions or jump in the middle of an instruction. Since Oak is not officially supported on QEMU and is mainly used with Google's internal hypervisor we were not able to test our exploit. However, we privately reported this to Oak maintainers on 15th March 2024, they acknowledged our attack and patched the handler [31].

Mushroom implements a custom VMPL0 and VMPL3 kernel. It supports restricted injection and fetches the `error_code` for the `#VC` directly from the guests VMPL3 VMCB. Since the hypervisor cannot control the `error_code`, Mushroom is not vulnerable to WeSee as long as the VMPL3 kernel uses alternate injection mode (explained in § 9.2).

8.2. Closed Source Implementations

We boot two Windows VMs using images with SNP support: (a) Windows Server 2022 Datacenter evaluation ISO on VirtualBox in a local setup; and (b) Windows Server 2019 Datacenter VM on Azure machine with SNP support. We extract the kernel and system files on both these setups. The kernel includes symbols, but no reference to SNP-specific terms such as "sev", "ghcb", "snp". Perhaps the SNP functionality is in a loadable module. However, C:/Windows contains thousands of files. Our file search for SEV terms returns zero hits. Instead of static code search, an alternative is to dynamically observe the execution from the hypervisor. KVM lacks support to boot Windows in SNP mode. HyperV supports SNP but is closed-source, so we cannot easily change it to inject `#VC`. In summary, analyzing Windows internals proves exceptionally challenging and we were unable to check whether Windows implements the `#VC` handler or supports restricted and alternate injection modes.

9. Potential Mitigations

We propose defenses to detect WeSee and stop the hypervisor from injection external `#VC`s.

9.1. Software-based Defenses

The defense can augment the VC handler. First, it checks the instruction that caused the `#VC`. Then, it can determine if the `#VC` was raised due to a valid instruction that can be intercepted, before processing it. For example, consider that the hypervisor triggers a `#VC` with `exit_reason` as `vmmcall` on a `cmp` instruction for the application in Lst. 1. The above defense will prevent the `#VC` handler from executing the `vmmcall` handling logic and corrupting `rax`. Instead, the handler will decode the instruction as `cmp`, see that it is not a legitimate instruction that should be intercepted, and discard the `#VC`. Further, the handler must use the instruction it decodes rather than the `exit_reason` register that can be controlled by the hypervisor.

The decoding approach is sufficient for most intercepted instructions because they have distinct opcodes (e.g., `vmmcall`, `rdtsc`). However, some instruction intercepts (e.g., MMIO, reads and writes to debug control registers) are triggered on a `mov` instruction. For these cases, instruction decoding and checking against the valid list of instructions is insufficient. The VC handler must also check that the `mov` instruction has the correct operands (e.g., memory-mapped regions, debug register `dr7`), thus complicating the logic. For example, the VC handler has to perform page-walks to see if the address accessed by the `mov` instruction was in the memory-mapped region. To complicate matters further, instruction intercepts for alignment checks could be triggered by any instruction (e.g., accesses to unaligned memory). So the VC handler will have to check, for every instruction, if it accessed unaligned memory.

When the exception handler performs the above checks, it has to mask all interrupts, which incurs overheads. More importantly, ensuring that the checks are complete is challenging and error-prone. In fact, as explained in § 4.3, the VC handler already checks and decodes the instruction for MMIO handling. However, in light of WeSee, the current decoding logic is (a) incorrect: it treats simple register accesses as accesses to memory; (b) incomplete: it does not check that the address accessed by the `mov` is actually memory-mapped, just that the access was to an unencrypted region. Therefore, this logic is insufficient or incorrect to securely determine if the `#VC` was caused by an MMIO operation. In summary, a purely software-based defense does not guarantee security and can be potentially bypassed.

Linux Patch for WeSee. AMD implemented this software patch as a quick fix to prevent arbitrary code execution [32]. We worked with the AMD team and improved the initial version of the patch to also cover the early boot image. Even with this patch, an attacker can still leak 8 bytes of the VM. The guest copies the register content to the GHCB and validates the legitimacy of the VC next. The hypervisor can infer the copied 8 bytes even if the guest terminates itself due to an illegitimate `#VC`. This serves as a case in point for the incompleteness of software defenses.

9.2. Hardware-based Defenses

There are two main approaches to defend against WeSee that both require hardware support to either protect the malicious hypervisor from tampering `exit_reason` from or injecting `#VC`.

Protecting `exit_reason`. One key requirement for WeSee is the ability to set the `exit_reason`. An obvious solution is to protect the `exit_reason` register, such that the hypervisor can no longer write to it. Then, the current VC handler can simply trust the values in the `exit_reason` and handle them without worrying about the hypervisor corrupting the value. This would eliminate the need for complex decoding and checks proposed in § 9.1. However, this might break functionality and needs microcode changes.

Blocking external `#VC` injection. There are three ways to block the hypervisor from externally injecting `#VC` into a victim VM's CPU. First, since there is no valid use-case where a hypervisor needs to inject a `#VC`, the hardware/microcode can directly block all external `#VC` injections into SEV VMs. Second, AMD SEV supports restricted and alternate injection mode. If the guest OS uses the Secure VM Service Module (SVSM) feature, it can selectively accept/drop external interrupts [1]. However, current open-source projects do not fully implement support for these modes. Third, AMD announced Secure Advanced Virtual Interrupt Controller (sAVIC) on 14 Nov 2023, where the SEV guest OS can mask interrupt vector to selectively allow/drop external interrupts. Due to lack of documentation and software, it is unclear if this will mitigate WeSee.

Immediate Hardware-defense Adoption. All SEV SNP-capable processors support restricted and alternate injection modes which can be used to prevent WeSee. Currently, the software support to enable these modes are not available in the Linux kernel. Efforts by AMD to upstream the secure software support have been hindered with known gaps and problems which are not straightforward to fix. For example, interrupts can be injected irrespective of the guest's `RFLAGS.IF` register state [33]. With these problems, introducing new microcode to block external `#VC` injection as a hot-fix is the most straightforward solution. However, for this AMD has to deem WeSee as a hardware vulnerability. To the best of our knowledge, AMD considers WeSee as a software bug and is working on a combination of restricted and alternate injection to fully mitigate WeSee.

10. Related Work

Attacks and vulnerabilities in SEV VMs. Google found multiple vulnerabilities in SEV-SNP by analyzing the security co-processor on AMD CPUs [34]. Prior works break SEV by targeting the cryptographic algorithms [35], performing page-remapping in the hypervisor [36], and using side channels [37], [38], [39], [40], [41]. CacheWarp reverts modified cache lines to corrupt memory writes [24]. Crossline uses hypervisor controlled address space identifiers (ASIDs) to compromise SEV VMs before they crash [42]. Code execution has been demonstrated by exploiting weak memory protection and performing Iago attacks on hypervisor interfaces [20], [43]. WeSee uses insights, such as page tracing and breaking physical ASLR, from these prior works. All the above attacks target SEV or SEV-ES; except for CipherLeaks [40] and CacheWarp [24] which need fine-grained information about the VM to break SEV-SNP. WeSee also breaks AMD SEV-SNP but only needs page fault information.

Untrusted interfaces. Attacks on Intel SGX exploit system call and other interfaces [16], [44], [45]. Several defenses build secure interfaces to counter these attacks [46], [47], [48], [49]. For malicious timer interrupts, AEX-Notify proposes a framework for Intel SGX that thwarts attacks that abuse asynchronous exits from the enclaves (e.g., timer interrupts for single-stepping) [50]. TrustZone introduces the concept of secure interrupts to prevent untrusted entities from sending malicious interrupts to protected applications [51]. In their SEV-ES analysis, Radev et al. and Hetzelt et al. point that the hypervisor can perform Iago attacks when `#VC` is used by the VMs to communicate with the hypervisor (e.g., bad rdtsc) [43], [52]. Since they do not consider malicious `#VC`s they conclude that the handler does not leak any sensitive information. WeSee observes that the hypervisor can trigger `#VC` at any point and uses benign `#VC` handlers to compromise the VMs.

Attacker's Capability to Profile Victim VM. SGX-step uses several known attacks, including APIC timer-interrupts, to build a single-stepping primitive for Intel SGX [53]. On

SEV, the hypervisor can observe page-faults of the guest OS [36]. Cachewarp builds a single-stepping framework for SEV-ES using timer interrupts, information from encrypted register states, and cache timing analysis [24]. SEV-STEP demonstrates a single-stepping framework for SEV-SNP VMs using page-faults, timer interrupts and eviction set-based cache attacks [23]. WESEE only uses page-faults observed by the hypervisor. However, if single-stepping techniques are available WESEE can mount stronger attacks (e.g., on user space instead of kernel space) [15].

Gap when supporting unmodified legacy applications. Several prior works run unmodified applications on Intel SGX and Arm TrustZone using abstraction layers (e.g., library OSes) [54], [55], [56], [57], [58], [59]. VM abstractions (e.g., AMD SEV, Intel TDX, Arm CCA) reduce invasive porting changes to legacy applications. Running unmodified legacy VMs requires hardware and software support in the kernel (e.g., hypervisor controlled mmio) [60]. Prior works do not analyze the effects of introducing new interfaces (e.g., `#VC`) coupled with existing hypervisor capabilities (e.g., ability to inject external interrupts). WESEE is a concrete attack in this direction that requires close scrutiny.

Kernel Defenses & Hardening. Existing kernel defenses against memory leakage and corruption [61], [62], and code-reuse [63] (e.g., ebpf, system call filtering, seccomp, control-flow-integrity) are orthogonal to WESEE. We demonstrate WESEE on the recommended AMD SEV-SNP kernel, up-to-date patches with all hardening enabled, standard compilers and default configurations recommended by AMD. The only defenses WESEE has to subvert were kernel ASLR and write protection of executable pages. Stronger defenses, if enabled, may make exploiting WESEE harder but may not stop it completely. Instead, it calls for systematic and rigorous kernel hardening for TEEs [64], [65].

Impact Beyond AMD SEV-SNP. Interrupt number 20 corresponds to the Virtualization Exception (`#VE`) on TDX. The handler is similar to AMD's `#VC` implementation. In theory, `#VE` is also vulnerable to WESEE attack—Linux handler does not decode to check if the VM indeed raised a `#VE`. However, we were unable to exploit the interface for two reasons. Intel TDX prohibits the injection of interrupt number 20 into the guest. Thus we cannot trigger the handler. Secondly, the hypercall used to obtain the register state is served by the trusted TD-module and the untrusted host has no direct way of controlling the arguments. On ARM CCA we were not able to identify a handler with a similar functionality. Thus we conclude that WESEE does not apply to Intel TDX and ARM CCA.

11. Conclusion

WESEE leverages the untrusted hypervisor's ability to inject malicious `#VC` interrupts into AMD SEV-SNP VMs. WESEE triggers the exception handler in the victim VM with well-crafted and well-timed `#VC`s to induce register and memory read/writes as well as arbitrary code injection into the VM memory. Our three case-studies show that WESEE compromises confidentiality and integrity of a victim VM.

Acknowledgement

We thank our shepherd, the anonymous reviewers, Mark Kuhne, and Mélisande Zonta-Roudes for their constructive feedback for improving the paper. Thanks to AMD and Borislav Petkov for the mitigation discussions and for developing the Linux patches.

References

[1] AMD, "AMD SEV-SNP Strengthening VM Isolation with Integrity protection and more," (2023). Accessed: Apr 4, 2024. [Online]. Available: https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf.

[2] ARM, "Arm Confidential Compute Architecture (ARM-CCA)," Accessed: Apr 4, 2024. [Online]. Available: https://www.arm.com/why-arm/architecture/security-features/arm-confidential-compute-architecture.

[3] IBM, "Confidential computing on IBM Cloud," Accessed: Apr 4, 2024. [Online]. Available: https://www.ibm.com/cloud/confidential-computing.

[4] Intel, "Intel Trust Domain Extensions (Intel TDX)," Accessed: Apr 4, 2024. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html.

[5] Microsoft Azure, "Azure confidential VMs using SEV-SNP (DCasv5/ECasv5) are now generally available," (2022). Accessed: Apr 4, 2024. [Online]. Available: https://azure.microsoft.com/en-us/updates/azureconfidentialvm/.

[6] Google, "Oh SNP! VMs get even more confidential," (2023). Accessed: Apr 4, 2024. [Online]. Available: https://cloud.google.com/blog/products/identity-security/rsa-snp-vm-more-confidential.

[7] AWS, "AWS: AMD SEV-SNP," Accessed: Apr 4, 2024. [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/sev-snp.html.

[8] Microsoft, "Microsoft moves 25 Billion in credit card transactions to Azure confidential computing," (2023). Accessed: Apr 4, 2024. [Online]. Available: https://techcommunity.microsoft.com/t5/azure-confidential-computing/announcing-microsoft-moves-25-billion-in-credit-card/ba-p/3981180.

[9] ——, "NLP Inferencing on Confidential Azure Container Instance," (2023). Accessed: Apr 4, 2024. [Online]. Available: https://techcommunity.microsoft.com/t5/azure-confidential-computing/nlp-inferencing-on-confidential-azure-container-instance/ba-p/3827628.

[10] AMD, "AMD Shares The Technical Details of Technology Powering Innovative Confidential Computing Leadership Cloud Offerings," (2023). Accessed: Apr 4, 2024. [Online]. Available: https://www.amd.com/en/newsroom/press-releases/2023-8-30-amd-shares-the-technical-details-of-technology-pow.html.

[11] Oracle, "Product News: Protect data in use with OCI Confidential Computing," (2023). Accessed: Apr 4, 2024. [Online]. https://blogs.oracle.com/cloud-infrastructure/post/protect-data-in-use-with-confidential-computing.

[12] Confidential Computing Consortium & Opaque, "Confidential Computing Summit 2023," Accessed: Apr 4, 2024. [Online]. Available: https://confidentialcomputingsummit.com/.

[13] AMD, "Protecting VM Register State with SEV-ES," (2017). Accessed: Apr 4, 2024. [Online]. Available: https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/Protecting-VM-Register-State-with-SEV-ES.pdf.

[14] ——, "AMD64 Architecture Programmer's Manual Volumes 1–5," (2024). Accessed: Apr 4, 2024. [Online]. Available: https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/40332.pdf.

[15] B. Schlüter, S. Sridhara, M. Kuhne, A. Bertschi, and S. Shinde, “Heckler: Breaking Confidential VMs with Malicious Interrupts,” in *USENIX Security*, 2024.

[16] S. Checkoway and H. Shacham, “Iago attacks: why the system call api is a bad untrusted rpc interface,” *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1, pp. 253–264, 2013.

[17] AMD, “SEV-ES Guest-Hypervisor Communication Block Standardization,” (2023). Accessed: Apr 4, 2024. [Online]. Available: https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/specifications/56421.pdf.

[18] Intel, “Intel® 64 and IA-32 Architectures Software Developer’s Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4,” (2024). Accessed: Apr 4, 2024. [Online]. Available: https://cdrdv2.intel.com/v1/dl/getContent/671200.

[19] M. Morbitzer, S. Proskurin, M. Radev, M. Dorfhuber, and E. Q. Salas, “SEVerity: Code Injection Attacks against Encrypted Virtual Machines,” in *IEEE Security and Privacy Workshops (SPW)*, 2021.

[20] L. Wilke, J. Wichelmann, M. Morbitzer, and T. Eisenbarth, “SEVurity: No Security Without Integrity : Breaking Integrity-Free Memory Encryption with Minimal Assumptions,” in *IEEE S&P)*, 2020.

[21] J. Koschel, C. Giuffrida, H. Bos, and K. Razavi, “TagBleed: Breaking KASLR on the Isolated Kernel Address Space using Tagged TLBs,” in *EuroS&P*, 2020.

[22] M. Morbitzer, M. Huber, and J. Horsch, “Extracting Secrets from Encrypted Virtual Machines,” in *CODASPY*, 2019.

[23] L. Wilke, J. Wichelmann, A. Rabich, and T. Eisenbarth, “SEV-Step A Single-Stepping Framework for AMD-SEV,” *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023.

[24] R. Zhang, L. Gerlach, D. Weber, L. Hetterich, Y. Lü, A. Kogler, and M. Schwarz, “CacheWarp: Software-based Fault Injection using Selective State Reset,” in *USENIX Security*, 2024.

[25] J. Werner, J. Mason, M. Antonakakis, M. Polychronakis, and F. Monrose, “The SEVerESt Of Them All: Inference Attacks Against Secure Virtual Enclaves,” in *ACM AsiaCCS*, 2019.

[26] AMD, “AMD SEV snp-host-latest tree ad9c0bf475,” Accessed: Apr 4, 2024. [Online]. Available: https://github.com/AMDESE/linux/tree/87146075f0d55c346ae7dbb902f8abc312e71004.

[27] Enarx, “Enarx,” (2023). Accessed: Apr 4, 2024. [Online]. Available: https://github.com/enarx/enarx.

[28] Project Oak, “Project Oak,” Accessed: Apr 4, 2024. [Online]. Available: https://github.com/project-oak/oak.

[29] Coconut SVSM, “Coconut SVSM,” (2023). Accessed: Apr 4, 2024. [Online]. Available: https://github.com/coconut-svsm/svsm.

[30] T. Dohrmann, “Mushroom,” Accessed: Apr 4, 2024. [Online]. Available: https://github.com/Freax13/mushroom.

[31] Project Oak, “Oak Pull Request: Ensure CPUID triggered the VC exception,” (2024). Accessed: Apr 4, 2024. [Online]. Available: https://github.com/project-oak/oak/pull/4974.

[32] B. Petkov, “x86/sev: Harden #VC instruction emulation somewhat,” (2024). Accessed: Apr 4, 2024. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=e3ef461af35a8c74f2f4ce6616491ddb355a208f.

[33] T. Lan, “x86/hyperv/sev: Add AMD sev-snp enlightened guest support on hyperv,” (2023). Accessed: Apr 4, 2024. [Online]. Available: https://lore.kernel.org/all/20230515165917.1306922-1-ltykernel@gmail.com/.

[34] Google Project Zero and Google Cloud Security, “AMD Secure Processor for Confidential Computing Security Review,” (2022). Accessed: Apr 4, 2024. [Online]. Available: https://storage.googleapis.com/gweb-uniblog-publish-prod/documents/AMD_GPZ-Technical_Report_FINAL_05_2022.pdf.

[35] M. Li, Y. Zhang, Z. Lin, and Y. Solihin, “Exploiting Unprotected I/O Operations in AMD’s Secure Encrypted Virtualization,” in *USENIX Security*, 2019.

[36] M. Morbitzer, M. Huber, J. Horsch, and S. Wessel, “SEVered: Subverting AMD’s Virtual Machine Encryption,” in *EuroSec*, 2018.

[37] W. Wang, M. Li, Y. Zhang, and Z. Lin, “PwrLeak: Exploiting Power Reporting Interface for Side-Channel Attacks on AMD SEV,” in *DIMVA*, 2023.

[38] R. Buhren, C. Werling, and J.-P. Seifert, “Insecure Until Proven Updated: Analyzing AMD SEV’s Remote Attestation,” in *ACM CCS*, 2019.

[39] L. Wilke, J. Wichelmann, F. Sieck, and T. Eisenbarth, “undeSErVed trust: Exploiting Permutation-Agnostic Remote Attestation,” in *2021 IEEE Security and Privacy Workshops (SPW)*, 2021.

[40] M. Li, Y. Zhang, H. Wang, K. Li, and Y. Cheng, “CIPHERLEAKS: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel,” in *USENIX Security*, 2021.

[41] M. Li, L. Wilke, J. Wichelmann, T. Eisenbarth, R. Teodorescu, and Y. Zhang, “A Systematic Look at Ciphertext Side Channels on AMD SEV-SNP,” in *IEEE S&P*, 2022.

[42] M. Li, Y. Zhang, and Z. Lin, “CrossLine: Breaking ”Security-by-Crash” Based Memory Isolation in AMD SEV,” in *ACM CCS*, 2021.

[43] M. Radev and M. Morbitzer, “Exploiting interfaces of secure encrypted virtual machines,” in *ROOTS*, 2020.

[44] M. R. Khandaker, Y. Cheng, Z. Wang, and T. Wei, “COIN Attacks: On Insecurity of Enclave Untrusted Interfaces in SGX,” in *ASPLOS*, 2020.

[45] J. Van Bulck, D. Oswald, E. Marin, A. Aldoseri, F. D. Garcia, and F. Piessens, “A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes,” in *ACM CCS*, 2019.

[46] Y. Chen, J. Li, G. Xu, Y. Zhou, Z. Wang, C. Wang, and K. Ren, “SGXLock: Towards efficiently establishing mutual distrust between host application and enclave for SGX,” in *USENIX Security*, 2022.

[47] M. Busi, J. Noorman, J. Van Bulck, L. Galletta, P. Degano, J. T. Mühlberg, and F. Piessens, “Securing Interruptible Enclaved Execution on Small Microprocessors,” *ACM Trans. Program. Lang. Syst.*, 2021.

[48] C. T. Cortiñas, M. Vassena, and A. Russo, “Securing Asynchronous Exceptions,” in *IEEE CSF*, 2020.

[49] R. de Clercq, F. Piessens, D. Schellekens, and I. Verbauwhede, “Secure Interrupts on Low-End Microcontrollers,” in *IEEE ASAP*, 2014.

[50] S. Constable, J. V. Bulck, X. Cheng, Y. Xiao, C. Xing, I. Alexandrovich, T. Kim, F. Piessens, M. Vij, and M. Silberstein, “AEX-Notify: Thwarting Precise Single-Stepping Attacks through Interrupt Awareness for Intel SGX Enclaves,” in *USENIX Security*, 2023.

[51] ARM, “Learn the Architecture: TrustZone for AArch64,” (2021). Accessed: Apr 4, 2024. [Online]. Available: https://developer.arm.com/architectures/learn-the-architecture/trustzone-for-aarch64/trustzone-in-the-processor.

[52] F. Hetzelt and R. Buhren, “Security Analysis of Encrypted Virtual Machines,” *ACM SIGPLAN Notices*, 2017.

[53] J. Van Bulck, F. Piessens, and R. Strackx, “SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control,” in *SysTEX*, 2017.

[54] L. Guan, P. Liu, X. Xing, X. Ge, S. Zhang, M. Yu, and T. Jaeger, “TrustShadow: Secure Execution of Unmodified Applications with ARM TrustZone,” in *MobiSys*, 2017.

[55] A. Baumann, M. Peinado, and G. Hunt, “Shielding Applications from an Untrusted Cloud with Haven,” in *OSDI*, 2014.

[56] C. che Tsai, D. E. Porter, and M. Vij, “Graphene-SGX: A practical library OS for unmodified applications on SGX,” in *USENIX ATC*, 2017.

[57] Y. Shen, H. Tian, Y. Chen, K. Chen, R. Wang, Y. Xu, Y. Xia, and S. Yan, "Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX," in *ASPLOS*, 2020.

[58] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer, "SCONE: Secure linux containers with intel SGX," in *OSDI*, 2016.

[59] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data," 2016.

[60] X. Ge, H.-C. Kuo, and W. Cui, "Hecate: Lifting and shifting on-premises workloads to an untrusted cloud," in *ACM CCS*, 2022.

[61] Linux, "The Kernel Address Sanitizer (KASAN)," Accessed: Apr 4, 2024. [Online]. Available: https://www.kernel.org/doc/html/v6.5/dev-tools/kasan.html.

[62] ——, "Kernel Self-Protection," Accessed: Apr 4, 2024. [Online]. Available: https://www.kernel.org/doc/html/v6.5/security/self-protection.html?highlight=kaslr.

[63] S. Tolvanen, "Linux Clang CFI," (2021). Accessed: Apr 4, 2024. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=cf68fffb66d60d96209446bfc4a15291dc5a5d41.

[64] Intel, "Intel TDX Guest Kernel Hardening Documentation," Accessed: Apr 4, 2024. [Online]. Available: https://intel.github.io/ccc-linux-guest-hardening-docs/index.html.

[65] M. Rybczyńska, "Hardening virtio," (2021). Accessed: Apr 4, 2024. [Online]. Available: https://lwn.net/Articles/865216/.

Appendix A.
Analysis of Nesting in Critical Section

x86 ISA supports instructions that operate on multiple memory operands. The Linux kernel frequently uses one instruction class that dereferences two memory addresses. For instance, the movsb instruction is used for memory copy operations in Lst. 5.

```
SYM_TYPED_FUNC_START(__memcpy)
  mov rax, rdi
  mov rcx, rdx
  rep movsb
  ret
SYM_FUNC_END(__memcpy)
```

Listing 5. x86 __memcpy in the Linux kernel

movsb copies the content from the memory referenced by rsi to the memory referenced by rdi. If either of the registers point to a memory-mapped region, MMIO access induced by this instruction triggers the VC handler. However, when the CPU passes the exit_reason to the VC exception handler, it does not indicate if the source, destination, or both operands caused the exception. Thus, it is up to the handler to detect the faulting memory access. As a solution to this problem, the VC handler first reads from the source and then writes the result to the destination. It effectively splits one optimized movsb instruction into two mov instructions. When the handler executes the mov instructions, it expects to raise a nested #VC, due to either one or both instructions. But it is sure that the instruction causing the nested exception dereferences only one memory address. Since the read and write exceptions happen sequentially, Linux must only support a nesting depth of one #VC. However, the VC handler itself has a critical section (seeLst. 6). In the critical section, the VC handler has exclusive access to the GHCB page used for communication with the hypervisor.

```
ghcb = __sev_get_ghcb(&state);
...
result = vc_handle_exitcode(ghcb, error_code);
__sev_put_ghcb(&state);
```

Listing 6. VC handler entering and leaving the critical section

To support one level #VC nesting in the critical section Linux introduces a second backup GHCB page. This is needed in our example because the MMIO #VC originating from a movsb instruction holds a lock to the first GHCB page. Attempting to handle the read and write accesses sequentially within the first #VC causes the nested #VCs to acquire and release the lock of another GHCB page. As of now the guest kernel only supports two GHCB pages. This effectively limits the nesting capabilities of #VCs in the critical section to one. If a second nested exception occurs while the previous exceptions are in the critical section, the guest will panic. While it is theoretically possible to support the case of a second nested exception in the critical section, there is no benign use-case as of now. However, nesting limits outside of the critical section are only bounded by the stack size available for the VC handler.