RUHR
UNIVERSITÄT
BOCHUM

RUB

RUHR-UNIVERSITÄT BOCHUM

Tracing the Footsteps of the WeSee Attack on AMD SEV

Jana Ilic

Bachelor Thesis – April 8, 2025.
Chair for Information Security.

1st Supervisor: Prof. Dr. Ghassan Karame
2nd Supervisor: M.Sc. Annika Wilde

InfSec

HG

Abstract

Confidential computing is essential in today's cloud environments, safeguarding sensitive data with technologies like AMD SEV, which ensure data confidentiality and integrity. Nevertheless, certain attacks exploit weaknesses related to interrupt handlings to bypass the security measures. This thesis examines two such hypervisor-based attacks, WeSee and Heckler, that undermine security guarantees by maliciously manipulating interrupts. This thesis investigates the distinct anomalies in system behaviour these attacks leave on their victims and explores how monitoring system events can help detect them. By employing two detection methods based on frequency analysis and sequential pattern analysis, we gain insights into the practical impact of these attacks and conclude that detection is achievable. Moreover, our findings determine that the proposed detection approach remains effective even against an adaptive adversary.

Contents

1 Introduction

With the increasing adoption of cloud computing, organizations and individuals rely on cloud providers to host sensitive workloads. However, this introduces a fundamental security challenge: how can a virtual machine (VM) be protected from a potentially untrusted cloud provider or malicious hypervisor? This concern has led to the rise of confidential computing, which aims to protect data in use, preventing unauthorized access even from privileged system components like the hypervisor.

To address this issue, Trusted Execution Environments (TEEs) were developed. TEEs initially focused on securing individual applications, but as security demands evolved, modern TEEs now extend protection to entire virtual machines. One of the implementations of VM-based TEEs is AMD Secure Encrypted Virtualization (SEV)[DK21] and its improved extensions, SEV-Encrypted State (SEV-ES)[Kap17] and SEV-Secure Nested Paging (SEV-SNP)[AMD20]. These technologies encrypt VM memory and provide integrity protections, preventing an untrusted hypervisor from tampering with or inspecting VM operations.

However, recent research has shown that even modern hardware-assisted TEEs are not completely resistant to attack[Zha24, Men21, Ste25, Buh21]. The Ahoi attacks[SSK+24, SSBS24, SBSS24] are one such example, demonstrating that a malicious hypervisor can still infer sensitive information from a VM protected by TEEs. These attack exploits encrypted memory access patterns, allowing an adversary to trace program execution and leak secret data. They leverage interrupt-based manipulation to subvert security guarantees, showing that control over system events can still lead to information leaks. These attacks highlight a crucial security question: Are there effective methods to detect such hypervisor-based attacks on SEV-SNP? While previous research has focused on breaking AMD SEV's security, less attention has been given to potential detection mechanisms.

This thesis addresses this gap by recreating two of the Ahoi attacks, the WeSee and Heckler attack and observes whether such attacks leave detectable traces in system performance metrics. More precisely, we aim to answer the following research questions:

- Can Ahoi attacks [SSK+24, SSBS24, SBSS24] be systematically detected through performance monitoring?

- Do the performance anomalies caused by the attack, such as interrupt manipulations or memory access patterns, generate identifiable footprints?
- What are the limitations of detection and how resilient is it against an adaptive adversary?

We aim to answer these questions by monitoring system events such as system calls and interrupts to determine a pattern in their occurrence caused by the attack.

2 Background

This chapter introduces topics crucial for understanding this thesis. Namely, we introduce the AMD SEV technology, the Ahoi attacks[SSK+24, SSBS24, SBSS24] and the performance monitoring tools used in the following chapters.

2.1 AMD Secure Encrypted Virtualization

AMD Secure Encrypted Virtualization (SEV) is a hardware-based technology designed to enable trusted computation in cloud environments. SEV limits a hypervisor's ability to inspect or modify a guest virtual machine's (VM) memory by providing encryption and isolation mechanisms at the hardware level. It addresses the issues of an untrusted hypervisor and co-tenants, and while it significantly enhances confidentiality, it does not provide complete protection against all types of attacks. Every SEV VM is assigned a 128-bit AES encryption key, which is securely managed by the AMD-SP, an isolated microcontroller embedded in the CPU. The primary objective of SEV is to ensure that the contents of a VM remain confidential even if an attacker gains control of the hypervisor. Data is encrypted when written to DRAM and decrypted when read, ensuring that a compromised hypervisor cannot extract plaintext data from a running VM.

Significant vulnerabilities remaining unresolved in AMD SEV stem from the fact that general-purpose registers are not encrypted during context switches triggered by VM exits, meaning that a malicious hypervisor can force a VM exit to gain access to protected data. Additionally, the VM Control Block (VMCB), which controls the guest's execution and state, remains accessible and modifiable by the hypervisor. And lastly the lack of memory integrity protections, which remain unsolved even in the next extension[HB17]. **SEV-ES[Kap17]**: The second extension of AMD's Secure Encrypted Virtualization (SEV), known as SEV-ES (Encrypted State), significantly strengthens guest protection by expanding security beyond memory encryption to include CPU register state.

In the original SEV model, only the guest memory was encrypted, while the general-purpose registers and control structures like the VM Control Block (VMCB) remained visible to the hypervisor. This allowed a malicious hypervisor to extract

sensitive data during VMEXITs, manipulate control flow by altering register values, or tamper with the guest's execution context.

SEV-ES addresses these vulnerabilities by encrypting the guest register state and adding protections around the VMCB. As a result, the hypervisor can no longer read or modify register contents directly. This encryption is applied automatically during VMEXITs, ensuring that no plaintext register state is exposed to the hypervisor.

To accommodate this new level of isolation, SEV-ES introduces a secure communication mechanism between the guest and hypervisor, since standard hypercalls and shared state mechanisms are no longer viable. This is accomplished through #VC (VMM Communication) exceptions.

These exceptions are designed to minimize the amount of information the hypervisor can access, restricting its access to only the specific register required to complete the operation that triggered the exception. When a VM attempts to execute an instruction that requires hypervisor assistance (e.g. I/O operations or MMIO access), it triggers a #VC exception. This exception is then delivered to the VM's kernel along with an exit_reason attribute, which details the cause of the exception. In response, the VM invokes an exception handler, called vc_handler, which responds based on the exit_reason and executes the appropriate set of operations to handle the exception effectively. The communication with the hypervisor is regulated through a shared memory region known as the Guest Hypervisor Communication Block (GHCB) [AMD25].

AMD SEV-ES has achieved the goal of confidentiality but still lacks integrity. Hence, it is open to integrity attacks including replay attacks, data corruption, memory aliasing, and remapping, all of which can undermine the system's security despite the encryption of the guest's memory and execution state.

SEV-SNP[AMD20]: The latest extension of AMD SEV, known as SEV-SNP (Secure Nested Paging), addresses the most critical vulnerabilities of the previous extension; the lack of memory integrity protection [Du,17]. To address the attacks on integrity[WMA+19, WWME20], AMD SEV-SNP presents two key innovations:

- Reverse Map Table (RMP): RMP is a single, shared data structure that tracks the ownership and security attributes of every physical memory page across the entire system. Each entry in the RMP assigns a specific owner, either a virtual machine, the hypervisor or the AMD-SP. The hardware uses the information stored in the table to perform lookups and enforce the integrity of pages, ensuring that private memory pages can only be read or modified by their rightful owner while only permitting carefully controlled write operations system-wide. RMP introduced three addresses for every page; the Guest Virtual Adress (GVA), Guest Physical Address (GPA) and System Physical Address (SPA). Entries in the table are indexed with the SPA, meaning that for every check

the RMP performs a translation of pages from GVA to GPA and lastly to SPA. The introduction of three different page addresses is a way to prevent memory aliasing attacks, which involve a malicious hypervisor mapping two different guest pages to one physical memory page.

- Page Validation: Page Validation introduces the use of a validation bit. Each entry in RMP includes a validation bit that starts in an "invalid" state (set to zero). This means that, by default, pages are not considered validated and are off-limits for use. Before a page can be employed by the guest, the guest must explicitly validate it. This is done by executing the special CPU instruction called PVALIDATE. When the guest runs PVALIDATE on a page, the hardware changes the validation bit from 0 to 1, marking the page as validated. This process ensures that only pages that the guest intends to use are activated for operations, reducing the risk that a stale or uninitialized page could be exploited.

Despite the protections introduced by AMD SEV and its subsequent iterations, the hypervisor retains indirect mechanisms to influence the execution of guest code [WWRE24, MHHW18, Ste25]. Even though the contents of guest memory are encrypted, the hypervisor can still monitor memory access patterns, identifying which pages are actively used and when they are executed. This profiling can be leveraged to distinguish between different execution phases of an application, allowing an attacker to infer sensitive information without directly decrypting memory. Additionally, the hypervisor can enforce memory access restrictions, such as marking pages as non-executable (NX) or read-only. By modifying the page table entries, the hypervisor can selectively prevent certain code from executing. This is crucial in the attacks that seek to manipulate execution flow without violating encryption protections and is the cornerstone of the Ahoi attacks.

2.2 The Ahoi Attacks

The Ahoi attacks represent a group of attacks on confidential virtual machines (CVM). The attack scenario is based on a malicious hypervisor injecting interrupts. The CMs are dependent on interrupts as a way of communicating with the hypervisor and do not treat these as malicious. Every interrupt has a corresponding interrupt handler, and the Ahoi attacks choose the interrupts based specifically on their handlers. They focus on interrupts, whose handling allows for a manipulation of registers' contents.
In addition to interrupts, the second necessity of these attacks is determining the right time and place for the interrupts. If injected randomly, the attacker has no control over the inputs or outputs and cannot gain any new knowledge. Hence, the attacker needs to follow along with the execution and determine at which point is the interrupt useful to him.

The Heckler attack has been successful for both AMD SEV-SNP and Intel TDX, while WeSee has only worked on AMD SEV-SNP. Thus, this thesis is limited to VMs running on AMD SEV-SNP.

2.2.1 Heckler

Heckler[SSK+24] is the first of the Ahoi attack to successfully target AMD SEV-SNP and Intel TDX. It demonstrates how the security guarantees of these architectures can be circumvented through carefully crafted interrupt injections. Two core components make the attack possible: the choice of the interrupt and the precise timing and placement of the injection.

A key element of Heckler's versatility is the use of interrupt 0x80, a legacy software interrupt used to invoke system calls on Linux systems. This interrupt is particularly powerful because it can trigger the execution of any system call, with the specific call determined by the value of the `eax` register at the time of injection. By injecting 0x80 at just the right moment, when the victim's execution context contains a target syscall number in eax, the attacker can gain control of the execution flow and of the content in the register. Besides the 0x80 interrupt, Heckler presents another possible interrupt, the 0x0. This interrupt triggers the floating-point exception signal (SIGFPE), which is used to report a fatal arithmetic error.In the case of 0x0 there is no predefined handler, so the attacker can manually add a handler and determine its return value. Again, this allows the attacker to overwrite the register holding the return value and as such, corrupt the continued execution.

As mentioned previously, the second essential part of the attack is determining the right timing and the target application. For this, Heckler relies on a complex system of observing page faults. Starting from the boot, the attacker collects page fault traces to build the set of boot traces $\mathbb{S}_{\text{boot}}$ and of the idle execution traces $\mathbb{S}_{\text{vm}}$. As shown in Figure 2.1, the attacker then proceeds determining the set of user traces $\mathbb{S}_{\text{user}}$. All the pages belonging to the idle execution traces, but not to the boot process, are part of the user's page traces. To isolate the application traces$\mathbb{S}_{\text{app}}$, the attacker repeatedly executes the application and derives a subset containing only the pages called by the application from the user traces. This knowledge is the basis for determining the right place to inject an interrupt.

In the practical case, the authors divided Heckler into two attacks: the sudo attack and the SSH attack, both of which target the circumvention of password-based authentication mechanisms. The sudo attack specifically exploits privileged access management (PAM), the security framework responsible for enforcing authentication and authorization policies, including those of privilege escalation. This is why the attack uses a tool for profiling exactly two pages: `pam_sm_authenticate` and `_unix_blankpasswd`, where `pam_sm_authenticate` is used by PAM to authenticate

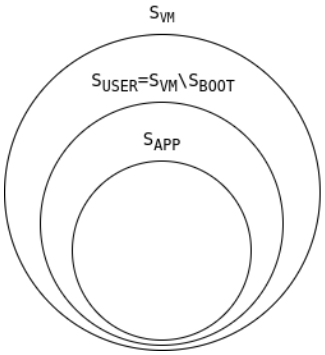

Figure 2.1: Page traces (sub)sets used for deriving the target application traces

the user and calls the function `_unix_blankpasswd` to check the correctness of the entered password. Analog to this, the SSH attack is looking for a pattern and addresses of pages `sshd_auth_password` and `sshd_mm_answer_authpassword`.

Before the injection, the attacker gains knowledge of the page addresses relevant to the target application and performs a series of unsuccessful logins to confirm the exact sequence of accessed pages. Initially, both pages involved in the login i.e. password checking process are non-executable, which causes page faults as the VM attempts to execute them; allowing the attacker to observe and trace the execution flow. Once all the necessary information is collected, the attacker allows the first page to execute and waits for it to transition execution to the second page. At that point, the first page is made non-executable again.

The attacker then monitors the second page closely, waiting for the moment when it writes its return value into the `eax` register and causes a page fault at the address of the first page. Knowing that the password-checking function on the second page will always return zero on a failed login, the attacker injects an interrupt just before further execution continues. With that, the `eax` register is overwritten, replacing the zero with the return value of system call 0 (e.g., -4), effectively faking a successful authentication result. This part of the attack is depicted in Figure 2.2, with two examples functions; authenticate and check password.

2.2.2 WeSee

The WeSee[SSBS24] attack follows a similar principle, but rather than abusing the 0x80 interrupt, it focuses on AMD's #VC exceptions. By using #VC exceptions instead of the 0x80 interrupt, the attacker can now not only change a register value to a fixed value but also arbitrarily read and write to a register. It compromises the confidentiality and integrity of a VM and manages to gain access to its private

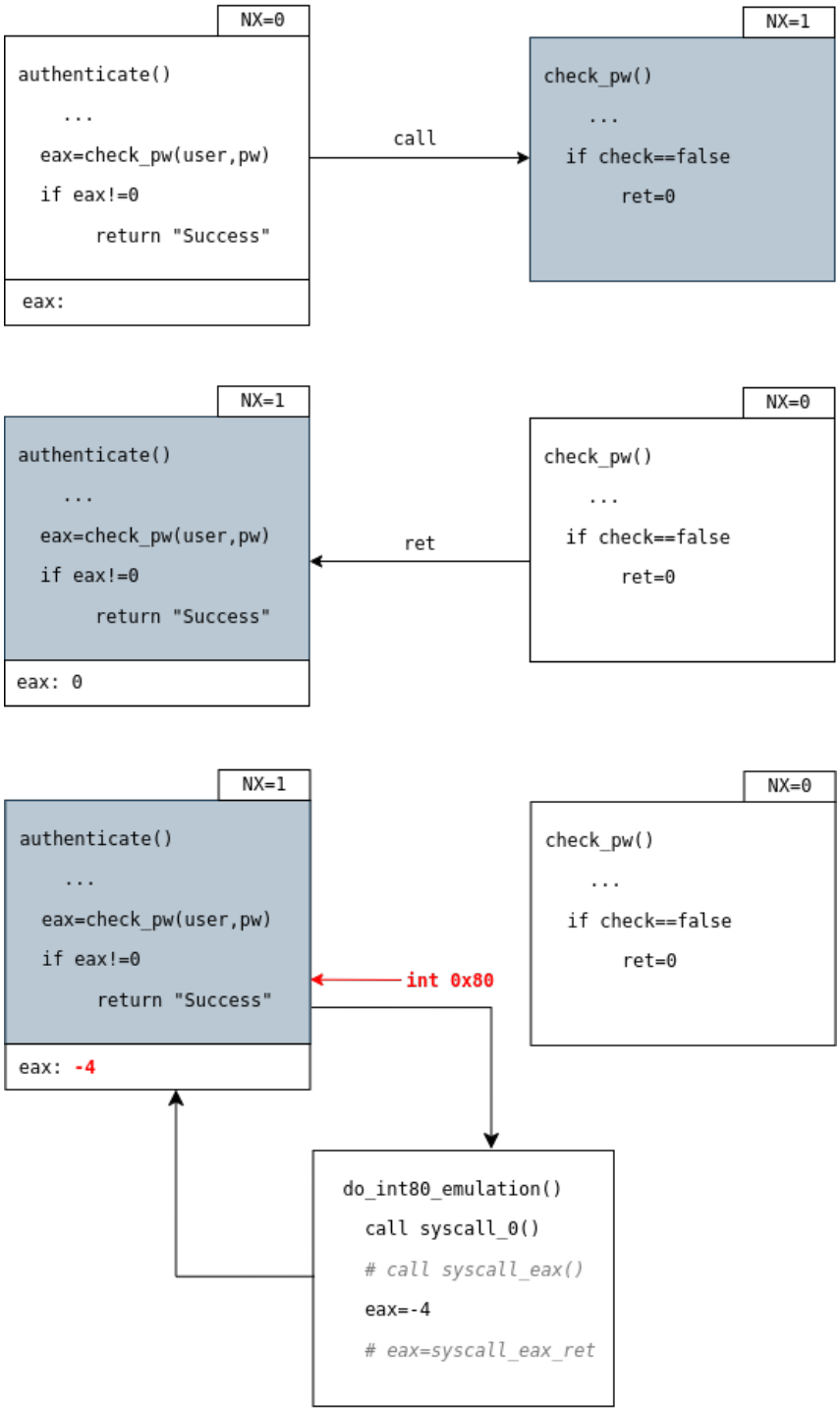

Figure 2.2: Execution flow of the Heckler attack on authentication systems

information. The carefully crafted #VC exceptions are injected into the victim VM's CPU during execution by a malicious hypervisor.

Another key distinction between the two attacks lies in their approach to identifying the optimal injection point. WeSee prioritizes the manipulation of instruction execution rather than focusing on profiling memory pages. Instead of profiling pages to determine the ideal location for interrupt injection, WeSee is focused on manipulating the execution flow at an instruction level. The attack is built out of primitives for skipping instructions, reading and writing to `rax` and kernel memory. They are constructed solely of #VC interrupts, using only three exit reasons:

- `vmm_call`: The vmm_call instruction is used to let the VM interact directly with the hypervisor. In the case of WeSee attack, this instruction allows the hypervisor to both read from and write to the `rax` register. Considering `rax` is often used to hold return values, counters, or results of calculations, being able to access it means the hypervisor can observe sensitive values or even modify them during runtime. This can change the behaviour of the VM's programs or leak private data without the VM considering the injected #VC interrupts as suspicious.

- `mmio_read`: This operation is triggered when the VM executes a memory read using a `mov` instruction e.g. `mov rbx,[rax]`. In this case, the hypervisor can write any value into the destination register (`rbx` in the example). This gives the hypervisor a way to insert manipulated data into the VM's program. In the WeSee attack, it is used as a part of the primitive enabling writing into kernel memory.

- `mmio_write`: Similarly to the previous one, this operation is only triggered on `mov` instructions (`mov [rax],rbx`), where in this case, the VM prints out the contents of the register (`rbx`). Analog to mmio_read, the attacker builds a primitive for reading from kernel memory with this instruction.

Thanks to the primitives, WeSee is a powerful attack with a wide range of capabilities. An example of how primitives are constructed is shown in Figure 2.3. S,R_{rax} and W_{rax} can all be constructed from a single #VC interrupt with the exit_reason set to vmmcall. This allows the attacker to either read and write to `rax` or leave it unmodified to only skip the instruction.

The practical impact of the WeSee attack has been demonstrated in three scenarios: leaking kernel TLS keys, disabling the firewall, and obtaining a root shell. Each of these exploits leverages the previously discussed attack primitives. Since these primitives significantly simplify kernel memory modification, an attacker can easily alter the code of specific functions or entirely replace them with their malicious instructions.

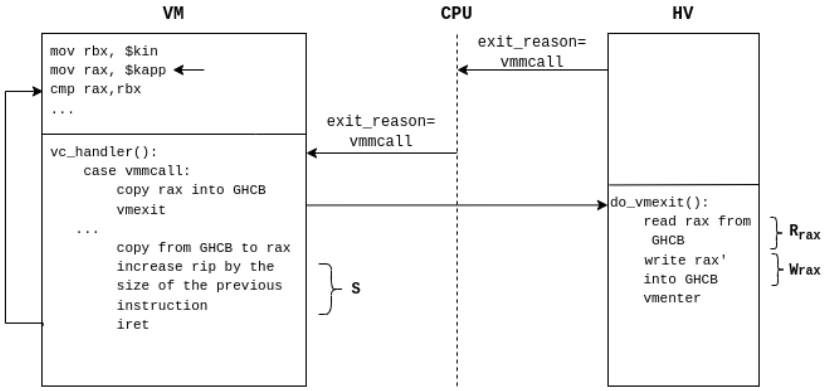

Figure 2.3: Skip, read and write to `rax` primitives

These cases prove that corrupting the VM's registers and arbitrarily injecting code is achievable by using only malicious interrupts and primitives created from them.

2.3 Tools

2.3.1 eBPF

Extended Berkeley Packet Filter (eBPF)[eBP25] offers a versatile framework that integrates into the Linux operating system. Beyond running secure programs inside the kernel without the need to rewrite kernel code, eBPF has evolved to support an increasingly real-time monitoring, debugging, and security-related tasks.

The core concept of eBPF is its use of hooks within the kernel. These hooks, points in the kernel's execution flow, allow eBPF programs to intercept events ranging from system calls to network events. Through kernel probes (kprobes) and user probes (uprobes), users can collect fine-grained details from both kernel-level and user-space applications. This allows us to observe system behavior in real time and has a wide range of potential uses, such as system tracing and performance profiling.

Another noteworthy aspect of eBPF is its mechanism for transferring data from the kernel to user space. Data collection is performed by the eBPF program running within the kernel, which then deposits information into a shared buffer i.e. (perf) ring buffer, illustrated in . This allows the data exchange between user and kernel space and enables the further processing of the data in user space applications. The significance of this buffer is discussed further in Chapter 5.

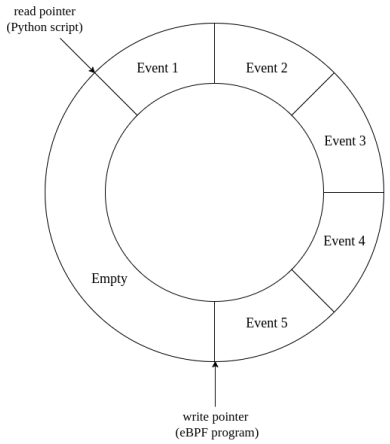

Figure 2.4: Efficient exchange of large amounts of data with ring buffers

2.3.2 Phoronix Test Suite

Phoronix Test Suite (PTS) is an open-source benchmarking and testing platform for assessing system performance. It contains a wide range of benchmarks, with which we can simulate typical cloud workloads such as web servers, the execution of AI tasks, or mimic database queries.

While the initial idea was to use these benchmarks to add noise to the system and see if it would be possible to detect the attacks in the presence of such noise, it does not have an impact on the logging system running inside the VM as the events processed in other machines are not reflected in our VM. The same applies the other way around.

Therefore, we use these benchmarks for measuring the performance overhead of the logging mechanism. We will establish the baseline performance, seeing how the VM would perform without it, and then we will measure the performance again with the logging system enabled. Results are presented in section 4.4.

3 Experimental Setup

This chapter details the setup used for recreating the attacks and the data collection process.

First, we set up the host and the VM according to the instructions provided[Ben24b, Ben24a] and reproduce the case studies from Heckler[SSK+24]. We then implement real-time monitoring and collect different system events occurring on the victim's system.

3.1 System Setup

We ran the experiments on an AMD EPYC 7313P 16-Core Processor. The host operating system is Ubuntu 24.04.1 LTS. For the Heckler attack, we used the provided Linux kernel version 5.19.0-rc6-heckler-host-snp-host-c36ef8e0c812, which has been modified specifically for this attack to allow for profiling of the VM. Another important part of kernel modifications is that it's compiled with the CONFIG_IA32_EMULATION flag enabled. This flag is essential for the interrupts from the x32 architecture such as 0x80 and 0x0 to be processed and emulated in the x86 architecture.

The experimental setup consists of a virtual machine used as the victim and a server running the hypervisor, which hosts the VM. The SEV-SNP enabled virtual machine is running on Ubuntu 23.10, as specified by the authors[Ben24b]. We initially tried running Heckler on a VM with Debian 16, used for WeSee[SSBS24], but this proved unsuccessful.

3.2 Reproducing Attacks

3.2.1 Heckler

To reproduce the Heckler attacks, we begin by following the setup instructions provided by the authors[Ben24b]. This involved booting the kernel and configuring the virtual machine. Both available attacks, sudo and SSH, required some modifications before they could be executed successfully. Also, they both rely on heckler bindings script, where functions needed for the attacks, such as tracking all pages, injecting,

etc. are defined. Each attack is structured into a set of helper scripts and a main Python file that performs the actual exploit. The attacks are designed to work on powered-off virtual machines, which are started as part of the attack process. As a result, any VM that is currently running will be terminated when the attack is launched.

sudo The sudo attack involves both host-side and VM-side preparation. The VM-side helpers are logically only possible if the attacker has already successfully circumvented the SSH authentication by first executing the SSH attack.

The host-side helpers include a script for converting the virtual address to physical and a boot filter to monitor the boot process and the log messages occurring during it.

Within the VM, the existing compiled pam_unix.so library, which is responsible for performing sudo checks, must be replaced with an unmodified version of the PAM shared library from Ubuntu 23.10. The reason for this is that not every PAM library is vulnerable to the attack and the attack's profiling of PAM has already been fine-tuned to known libraries. After accessing the VM via SSH, the attacker runs the profile_pam script. This script causes repeated jumps between two PAM-related functions, pam_sm_authenticate and _unix_blankpasswd, simulating their behaviour during real authentication. This simulation helps the attack build the set of application page traces. The issue we faced with the sudo attack was that no possible candidate pages ("gadgets") for the injection could be found. After examining the attack's script and analysing the number of occurrences of the found candidates, we concluded that the issue lies in textttmin_hits threshold. In the original code, this value is set to 10,000, meaning that any page profiled fewer than 10,000 times was excluded as a candidate for the injection. However, on our system, the number of hits varied between fewer than 100 and a maximum of around 5,000. To prevent the attack from stalling at this condition, we lowered the `min_hits` threshold to 50.

SSH With the SSH attack, we encountered several issues with the original code that uses named pipes for synchronization. If neither a reader nor a writer exists, the process becomes blocked. In addition, the pipes were in the wrong order, which means e.g. if a process is waiting to read from a pipe that isn't firstly being written into, it will block indefinitely. This causes a deadlock because the process is stuck waiting on input that never comes, halting the further execution and stalling the attack.

And lastly, in some cases, communication failed due to insufficient permissions when attempting to open the pipes. We have changed the pipes to open in a non-blocking mode, added error handling through a try-except block to prevent stalls and adjusted the timeout handling. We also include timestamps during the attacks for easier correlation with the events in the log and changed the SSH port from the originally defined 2222 to 2220. The reason for this is that port 2222 was already in use by

other users on the host, so to avoid conflicts and port collisions, we moved the attack to port 2220.

An unresolved issue remains the unpredictability of the attack's outcome. It often injects the interrupt either at the wrong time or address, resulting in the VM freezing, or it marks the attack as successful without actually bypassing the authentication. This has minimal impact on our logs, as the interrupt had already been injected and it is reflected on them.

3.2.2 WeSee

For the WeSee attack, we again relied on the official setup instructions provided by the authors [Ben24a]. Similar to Heckler, this attack requires a custom, modified kernel to enable its functionality. However, after rebooting our system with the supplied kernel, we encountered a critical issue: any attempt to launch a virtual machine would immediately crash the kernel. Resolving this issue would require kernel debugging, which lies outside the scope of this thesis. As a result, we chose to shift our focus to the Heckler attack instead.

3.3 Monitoring

The primary objective of monitoring the VM and logging events is to identify distinct variations and patterns in the occurrence of page faults, interrupts, and syscalls. Hence, the data collection consists of collecting the logs of events happening with and without the attack active, from both inside the VM and in the host.

The motivation behind monitoring page faults is the fact that the attacks rely on detecting a specific pattern of page faults. The attacker sets all pages as non-executable at the beginning of the attack, which generates page faults when a VM attempts to execute them. By monitoring these page faults, the attacker can essentially follow the execution of the code on a page level and therefore determine the right time to inject an exception. The goals of these attacks revolve primarily around bypassing authentication or leaking sensitive information, so targeted pages usually contain functions responsible for authentication checks.
The core idea is to find a page fault pattern of type Page_A, Page_B, Page_A; where Page_A executes the authentication and relies on the output of a function on Page_B. By injecting an interrupt after Page_B's execution but before Page_A continues, the attacker can successfully modify the register containing the return value and the authentication on Page_A continues with a corrupted register value.

An example of how the attack uses this sequence in practice is shown in Figure 3.1. After profiling the VM and the target application, three possible candidate pages ("gadgets") were found. The first two candidates are an obvious choice, since the

number of hits (occurrences) differentiates by 1, meaning they follow the pattern of [Candidate$_2$, Candidate$_1$, Candidate$_2$]. After repeated sudo login attempts, the attack produces the expected page fault pattern, further proving this is the right injection place. Lastly, the attacker injects the interrupt, which is followed by a root shell opening inside the VM. Hence, the rise of page faults, especially from processes

Figure 3.1: Excerpt from the Heckler sudo attack

concerned with authentication can be an indicator of the attacker profiling these pages to find the right candidates for injection.
The reason for collecting system calls is the fact that the int 0x80 interrupt is emulated to a system call and only as such shown in logs inside the VM. An increase in system calls can be another sign of an ongoing attack. Further discussion on this interrupt and its use in the attack is discussed in 4.1.

For this, we use a Python script leveraging eBPF that collects system events like page faults, interrupts and system calls, in real-time. As explained, eBPF relies on probes, functions that attach to specific points in the kernel to capture system events. In this case, we use tracepoint probes, which are predefined hooks within the kernel. First, we define the probe and attach it to the event we want to collect (page fault, interrupt, system call), meaning the probe will be triggered each time an event of that type occurs. In the structure, we define the data we want to collect about the event, such as timestamps, process IDs, or error codes.

Once the event data is captured, the probe submits it to user space using a perf ring buffer. A user-space application, such as a Python script, can then retrieve and handle the event data for logging.

An example of how page fault events are collected and logged is shown in Listing 3.1.

```
struct page_fault_event_t {
u64 ts;                 // Timestamp
u32 pid;                // Process ID
```

```
    u64 address;            // Fault address
    u32 error_code;         // Error code
    char comm[TASK_COMM_LEN]; // Process name
};

// Tracepoint for page faults
TRACEPOINT_PROBE(exceptions, page_fault_user) {
    struct page_fault_event_t event = {};
    u32 pid = bpf_get_current_pid_tgid() >> 32;

    if (!filter_pid(pid))
        return 0;

    event.ts = bpf_ktime_get_ns();
    event.pid = pid;
    event.address = args->address;
    event.error_code = args->error_code;
    bpf_get_current_comm(&event.comm, sizeof(event.comm));

    page_fault_events.perf_submit(args, &event, sizeof(event));
    return 0;
}

...

def print_page_fault_event(cpu, data, size):

    global shutdown_in_progress
    if shutdown_in_progress:
        return

    event = b["page_fault_events"].event(data)
    timestamp = get_timestamp(event.ts)
    error_code_str = "W" if event.error_code & 2 else "R" # 2 means write access
    outfile.write(f"{timestamp},PAGE_FAULT,{event.pid},{event.comm.decode('utf-8', 'replace')
        },address=0x{event.address:x} type={error_code_str}\n")
    outfile.flush()
```

Listing 3.1: Excerpt from the process_monitor script

An excerpt from a log generated by our monitoring script is shown in Listing 3.3. Each log entry provides detailed information, including a timestamp, event type, process ID, the name of the process (command) that triggered the event, and additional event-specific details. For soft IRQs, the log specifies the IRQ type, such as TIMER, SCHED, or RCU. In the case of page faults, it captures the faulting memory address and whether the fault was caused by a read or write operation. For system calls, we log the syscall number and, when available, the return value. The event types described with ENTRY and EXIT should occur in pairs and a discrepancy in the number of entries and exits can be another useful of irregular behaviour.

```
2025-03-24 16:43:54.683924,SOFTIRQ,0,swapper/0,TIMER,1.69 $\micro$s
2025-03-24 16:43:57.544353,PAGE_FAULT,98,systemd-journal,address=0x7aad90321998 type=W
2025-04-07 19:27:37.526627,SYSCALL_ENTRY,344,python3,syscall_nr=3
2025-04-07 19:27:37.526635,SYSCALL_EXIT,344,python3,syscall_nr=3,ret=0
2025-04-07 19:27:37.533066,SOFTIRQ_ENTRY,344,python3,TIMER
2025-04-07 19:27:37.533081,SOFTIRQ_EXIT,344,python3,TIMER,ret=0
```

4 Evaluation

This chapter provides information on the process of evaluating the collected data and proposing a detection tool. We also expand on the 0x80 interrupt and how we expect it to appear in logs. By collecting both the logs during the attack and without the attack being executed, we can observe a change in the system's behaviour and discuss how monitoring can be used for detection.

We evaluated the data using two approaches: by analyzing the frequency of events and by examining the patterns in which they occurred. For frequency analysis, we analyzed the total number of events and also specific system calls occurring during baseline and attack. For the pattern analysis, we look for

4.1 Interrupt 0x80 Analysis

One of the central parts of the Heckler attack is the external injection of int 0x80 interrupts into the guest VM [SSK+24]. Despite being deprecated in modern x86 Linux systems, the int 0x80 mechanism remains a valid method for invoking system calls, and as such, can still be exploited by an attacker to inject malicious execution without raising obvious alarms.

Understanding how this interrupt behaves in system logs is crucial for recognising it and designing an effective detection mechanism. To analyze its footprint, we constructed a minimal assembly program that explicitly calls int 0x80, mimicking the style of injection used in the Heckler attack. This allowed us to observe how the interrupt manifests in logs under controlled conditions. We saw that in this particular example, the execution of int 0x80 is not reflected as an interruption, but it is translated into a series of syscalls with different numbers.

The interrupt is processed as a caller of system calls. A system call corresponding to the value stored in `eax` is called, while `ebx, ecx, edx` store the arguments. Upon exiting the system call, the result is stored again in the `eax` register[Man06].

```
section .data
    message db 'Hello, int 0x80 syscall!', 10 ; Message to print, 10 is newline
    message_len equ $ - message             ; Calculate message length

section .text
    global _start

_start:
```

```
    ; write syscall using int 0x80
    ; syscall number for write is 4 on x86
    mov eax, 4         ; syscall number (write)
    mov ebx, 1         ; file descriptor (stdout)
    mov ecx, message   ; pointer to message
    mov edx, message_len ; message length
    int 0x80           ; invoke syscall

    ; exit syscall
    mov eax, 1         ; syscall number (exit)
    xor ebx, ebx       ; exit status 0
    int 0x80           ; invoke syscall
```

When the attacker precisely times the injection, they inject the 0x80 interrupt while the eax register holds a value of zero. As a result, the system attempts to execute syscall number 0, corresponding to the `restart_syscall` system call. This call is designed to be used exclusively within the kernel, not by user-space applications, causing it to fail with an `EINTR` error. As illustrated in Listing 4.2, error values are returned as negative numbers, meaning -4 (representing `EINTR`) is stored in the eax register.

```
    /**
 *  sys_restart_syscall - restart a system call
 */
SYSCALL_DEFINE0(restart_syscall)
{
  struct restart_block *restart = &current->restart_block;
  return restart->fn(restart);
}

long do_no_restart_syscall(struct restart_block *param)
{
  return -EINTR;
}
```

Listing 4.1: Definition of syscall 0 in Linux[Tor25]

During the attack, we also observe unsuccessful interrupt injections that trigger a kernel panic. Such an event typically occurs when the interrupt is injected at an inappropriate time or location, or when the necessary system call context is absent. In these scenarios, the kernel lacks the correct state to properly handle the injected 0x80 interrupt, which causes it to enter an unrecoverable state, triggering a panic that halts execution across all virtual CPUs. As a result, if an attacker misjudges the proper injection point or timing, the error will manifest as a fatal system crash, which is also recorded in the VM's logs.

4.2 Frequency Based Approach

The first way of analysing the footprints left by the attacks focuses on detecting significant increases in event occurrences. For that, we collected 10 logs from both

the attack present and not present. For every log, we followed the same procedure; booted the VM, which automatically started the monitoring script, attempting to log in three times with an incorrect password and finally logged in successfully before stopping the monitoring. For an easier analysis, we split the events from their further details. We are interested primarily in the events which show the presence of an attack, such as soft IRQs (entries and exits), and system call entries, exits and return values. These low-level kernel events provide insights into how processes interact with the system and how those interactions deviate under malicious behaviour.

We first calculate the occurrences of all main events (without additional event details) and compare the numbers of events between the malicious and benign execution cases, as shown in Figures 4.1 for SSH and 4.2 for sudo attack.

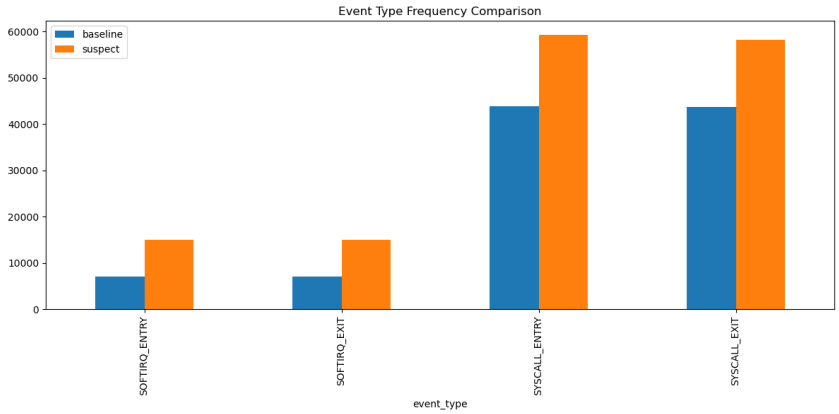

Figure 4.1: Event frequency comparison for SSH attack

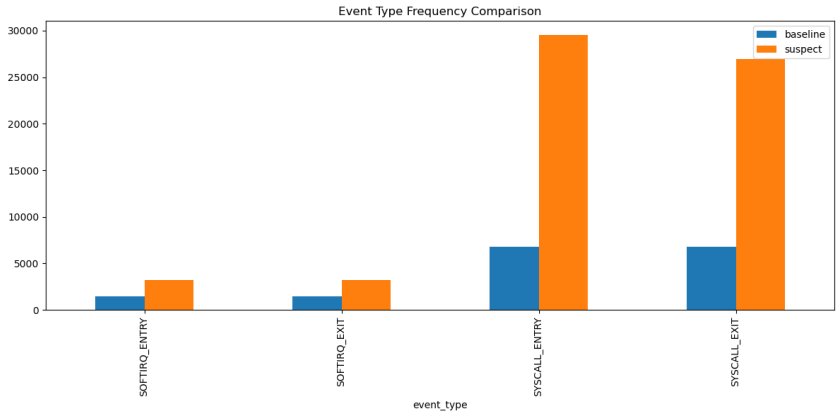

Figure 4.2: Event frequency comparison for sudo attack

As evident from Figures 4.1 and 4.2, the attack scenarios consistently result in an increased number of system call entries and exits, along with a moderate rise in soft IRQ activity. This suggests that the attack induces heavier or unusual kernel interaction, possibly through syscall injection or triggering additional system events.

This motivates a further analysis of which specific system calls and return values of syscalls are most commonly occurring. The figure 4.3 shows the most frequent syscalls in both of the cases.

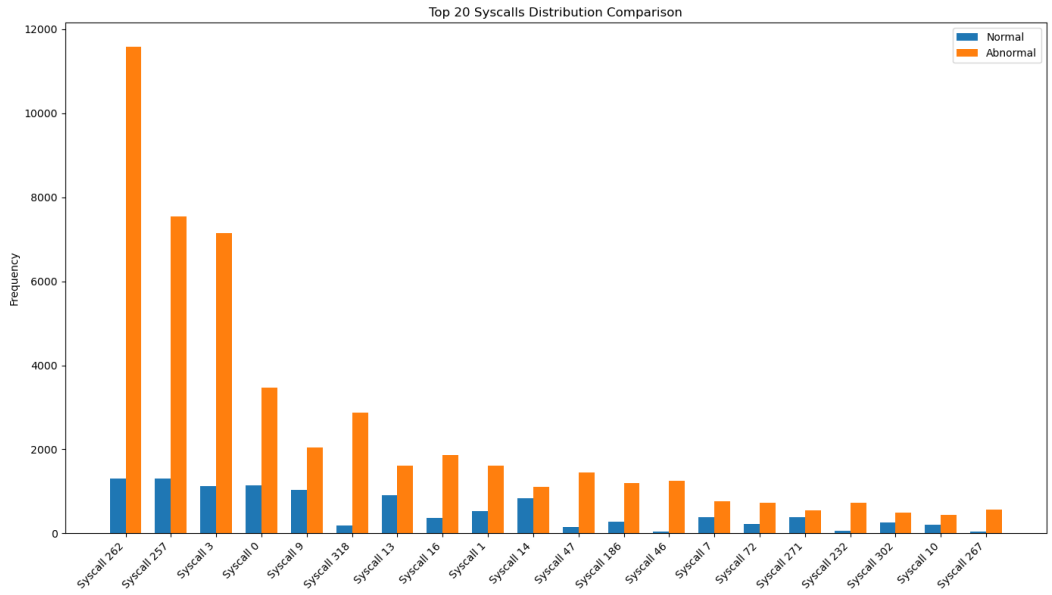

Figure 4.3: Most commonly occurring syscalls and their number of occurrences

Table 4.1 compares the frequency of these return codes across baseline and suspect logs, where we can notice:

- a rise in common errors such as -2 and -25
- certain error values occurring only during the attack, including -4, which aligns with the attack's expected behaviour in case of a successfully injected interrupt
- high-value errors such as -512 or -516, which are also suspicious, considering that legitimate syscall errors fall within a well-defined range of lower values

This frequency analysis is the preparation for further evaluation, in which we not just occurrences of rare and unusual events, but also the sequence in which the events occur.

Return Code (ret)	Baseline Count	Suspect Count
-2.0	296	1910
-22.0	32	59
-25.0	26	662
-11.0	14	134
-1.0	11	38
-10.0	6	7
-61.0	5	68
-3.0	2	8
-17.0	1	19
-13.0	0	5
-512.0	0	4
-4.0	0	2
-107.0	0	2
-111.0	0	2
-39.0	0	1
-97.0	0	1
-110.0	0	1
-104.0	0	1
-516.0	0	1

Table 4.1: Comparison of syscall error return codes between baseline and attack log

4.3 Sequence Based Approach

In addition to analysing the frequency of system calls, we perform a sequence-based evaluation to understand not just how often certain calls occur, but also in what order. This helps us detect more subtle changes in system behaviour that might indicate an attack.

The frequency-based evaluation cannot give us enough data to base detection on, because the number of events that occur can stay similar during the attack, but the order in which they happen and to which process they are related to, can give us enough information to classify certain patterns as suspicious.

Note that we collect system-wide events, as the victim cannot know in advance which syscall it is going to receive, which interruption or which process might send it. Further, the logs have multiple dimensions, i.e. many different events are collected and they have text values as well as numerical ones. For each of the 2 attacks, the traces contain the same events and follow the same pattern, i.e. a user normally connecting to the VM via ssh or executing the sudo command.

The sequence-based approach consists of many steps, such as:

1. Parsing the logs and extracting the important information (timestamp, event type, PID, command and additional information)
2. Using sliding window analysis, we split the data of both attack and normal logs into time-based windows. This allows for a better overview, considering that a huge number of events occur in a very short time and it is therefore easy to oversee important events.
3. For each window, we repeat the process of counting the frequency of each event type, extracting system call numbers and capturing sequences of system calls per process ID, as shown in Figure 4.4. We also plot a summary of the total events detected per window in Figure 4.6 for the normal events and Figure 4.7 for the ones containing attack traces.
4. In this next step, we convert the syscall sequences into n-grams. N-grams are sequences of n successive items or in this particular case, of n successive syscalls. By using this, we can turn syscall sequences into patterns used for comparison. By analyzing sequences of syscalls rather than individual calls, it is possible to detect behavioural patterns that aren't visible when looking at isolated events
5. Using windows and n-grams, we compute the cosine similarity between the baseline windows and attack windows, as well as between the syscall patterns, which can be seen in Fig. 4.5.
6. We depict a scatter plot that shows which specific syscalls appear in each time window, making it easy to spot Syscalls that only appear in abnormal traces, Windows where new or unusual syscalls suddenly appear, and changes in the kind of syscalls used over time

All the results presented in the Figures in this section are collected during 60 seconds for both the attack and normal traces. There are 4 injections during the attack and 4 logging attempts in the normal ones. There are 131597 log entries in the normal log file and 291875 in the abnormal one.

As Figure 4.5 shows, there are some windows where the events are clearly different (i.e. the cosine similarity is low, as values close to 1 show they are similar). It is expected to have similar traces in windows where the machine is idle and just executing its background tasks.

OT_INT events seen in Figures 4.6 and 4.7 reflect events related to KVM (the hypervisor in our experiments). The big four peaks of such events in the 4.7 correspond to the instances where the attack is injecting the syscalls. Therefore, they are a good indicator of when to start paying more attention to the syscall events and the sequence of those.

Figure 4.8 shows that there are some windows where multiple syscall events are received simultaneously. We note that KVM related events are also syscalls and

```
=== ABNORMAL LOG SUMMARY ===

Window 1 — 2025-04-04 20:21:36.650682 to 2025-04-04 20:21:37.150682
  Event Types:
    SOFTIRQ_ENTRY: 15
    SOFTIRQ_EXIT: 15
    SYSCALL_EXIT: 8
    SYSCALL_ENTRY: 8
  Unique Syscalls: [0, 61, 77, 100, 186, 232, 270, 286]
  Syscall Sequences:
    PID 98: 232 → 0 → 77 → 186 → 286
    PID 356: 270 → 61 → 100

Window 2 — 2025-04-04 20:21:37.150682 to 2025-04-04 20:21:37.650682
  Event Types:
    SOFTIRQ_ENTRY: 11
    SOFTIRQ_EXIT: 11
    SYSCALL_EXIT: 1
    SYSCALL_ENTRY: 1
  Unique Syscalls: [7]
  Syscall Sequences:
    PID 319: 7
```

Figure 4.4: Example of the summary of the analysis for 2 windows for the traces referring to the SSH attack

have different numbers than normal syscalls in the system. The mapping between syscall numbers and the syscall itself can be found in a Linux system by executing `ausyscall -dump`. For example, the first numbers correspond to 0-read, 1-write, 2-open, 3-close.

4.4 Monitoring Overhead

Our monitoring script collects a vast amount of data in a very short period, generating logs that can contain up to thousands of events. We aim to assess how this intensive logging affects overall VM performance, given that monitoring is a central component of a real-time detection system. The Table 4.2 highlights the performance impact of enabling logging on various benchmarks, serving as a clear indicator of monitoring overhead. Pronounced effects on performance are observed in system call operations like syscall/basic and syscall/getpid, where the overhead exceeds 700%. These numbers show the trade-off made between comprehensive system monitoring and performance, emphasizing the need for careful tuning and selective logging. Conversely, benchmarks like Phoronix Test Suite's pts/redis-1.4.0 demonstrate lower overhead, suggesting that the impact of logging can vary significantly depending on the workload and the nature of operations being monitored.

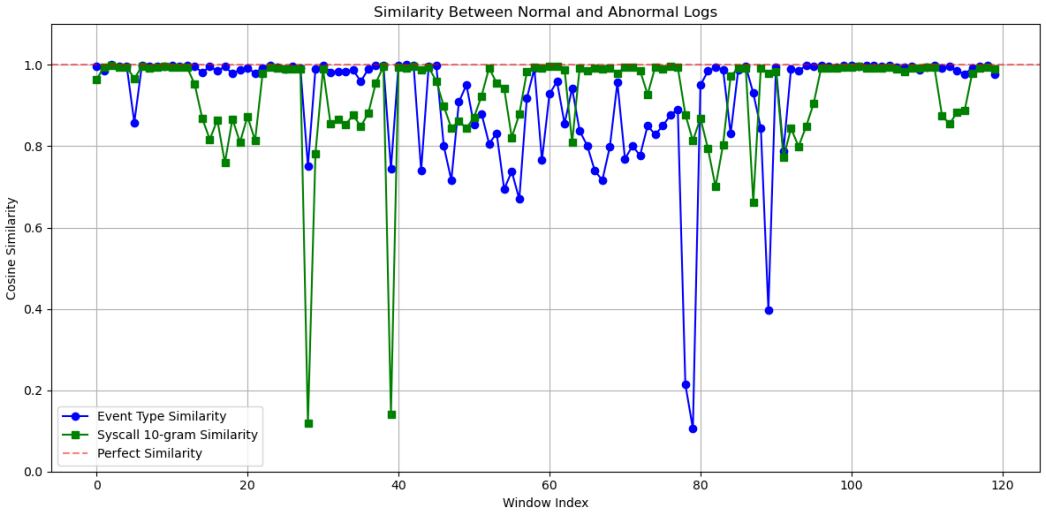

Figure 4.5: Cosine similarities of event frequency and of syscalls 10-grams

	Benchmark	log off	log on	overhead
perf	sched/messaging	0.605 s	1.17 s	93%
	sched/pipe	28.686 s	31.302 s	9%
	syscall/basic	3.011 s	25.106 s	734%
	syscall/getpid	2.942 s	24.600 s	736%
Phoronix Test Suite	pts/redis-1.4.0	2360 s	2030 s	16%

Table 4.2: Benchmark results of the monitoring overhead

4.5 Detection

The main goal of this thesis is to determine whether developing a detection tool against these attacks is possible. Based on the evaluation of the collected data, we have found that the attack leaves identifiable traces within the VM.
Our analysis reveals that system calls observed during the attack, but absent in normal logs, serve as reliable indicators of malicious activity. These unique syscall patterns can form the foundation of an effective detection tool. The findings presented in the Evaluation section lay the groundwork for a potential detection mechanism capable of continuously monitoring and processing a variety of system events, looking for an anomalous pattern.

We deduce that the detection method effective for Heckler should similarly work for WeSee. As demonstrated in Figure 4.9, WeSee also depends on observing page faults and on a high number of injected #VC interrupts. Given that #VC interrupts correspond to interrupt number 29, the logic is analogous to detecting 0x80 interrupts;

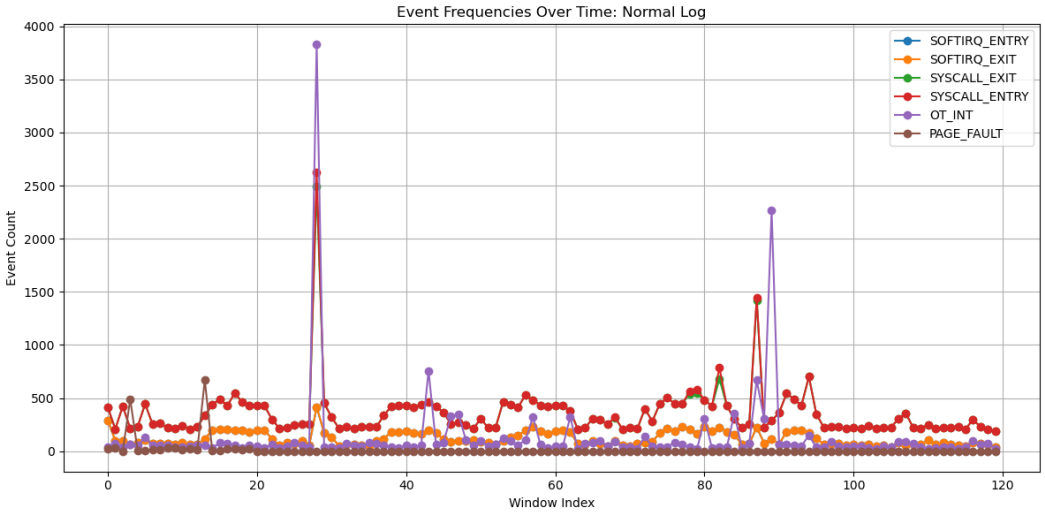

Figure 4.6: Summary of the events found per window during the normal execution of the ssh requests

if we can detect that, then detecting interrupt number 29 follows the same principle. And while the Heckler attack relies on one correctly injected interrupt, WeSee must inject over 200 #VC interrupts for success. Based on this, we can argument that WeSee creates a greater and more obvious anomaly in system's behaviour than Heckler; therefore any detection system fine-tuned enough to detect Heckler should have no issues detecting WeSee as well.

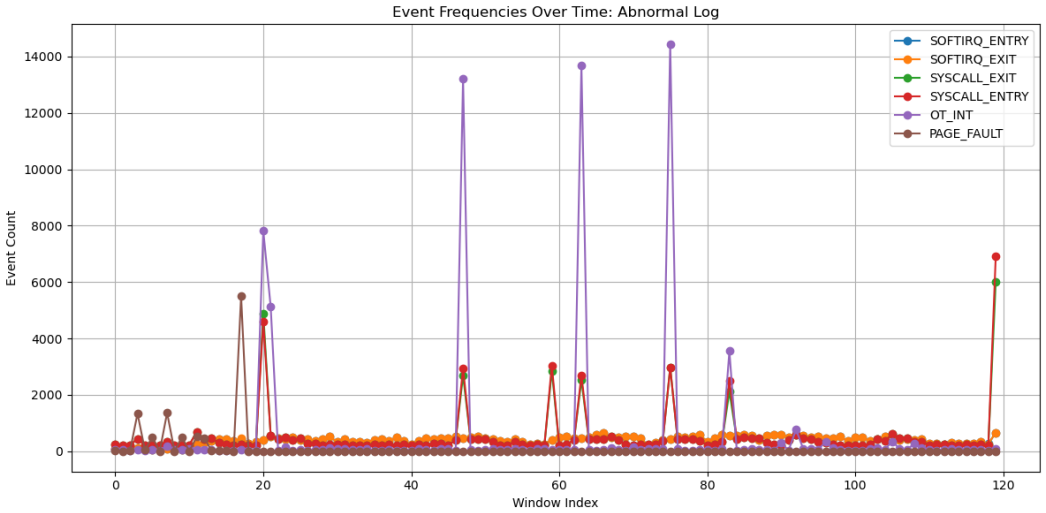

Figure 4.7: Summary of the events found per window during the injection of interruptions of the ssh requests

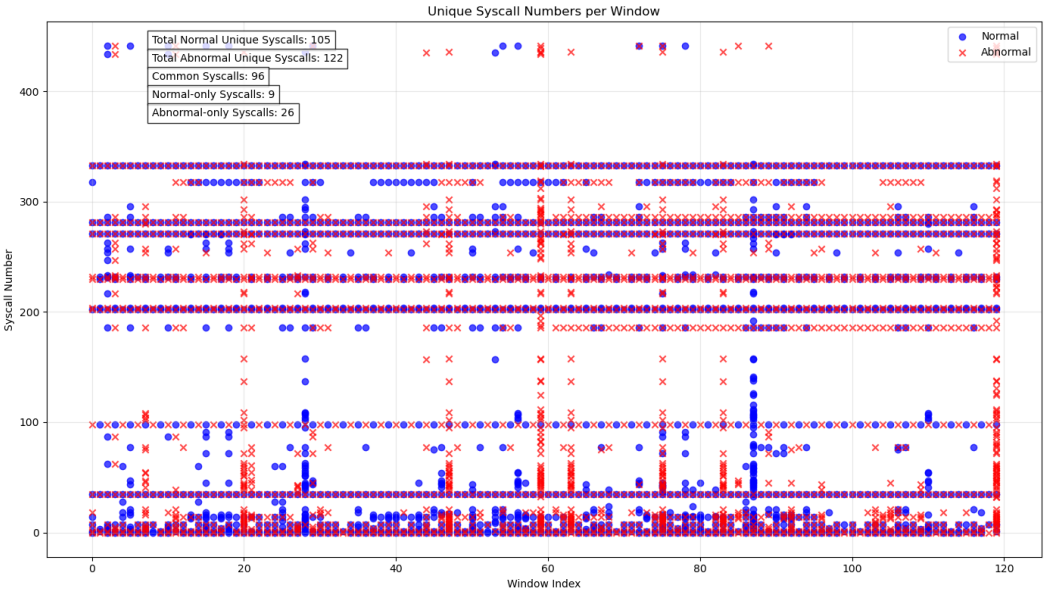

Figure 4.8: Unique syscalls by number found in each window

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

Figure 4.9: Event occurrences across case studies[SSBS24]

5 Adaptive Adversaries

This chapter evaluates the feasibility of our proposed detection method against adaptive adversaries. In the context of security, an adaptive adversary is an attacker who continuously changes strategies to evade detection. In the case of Heckler and WeSee attacks, an adaptive adversary could attempt to minimize or disguise their impact on system performance to bypass detection.

Based on the evaluation, the focus of detection is identifying the anomalies occurring only during the attacks. Even a single injected interrupt leaves distinct traces, making it a clear indicator of malicious activity. Therefore, attempts by the attacker to time or delay the injections offer the attacker no advantage, as a single injection suffices for detection.

One way an adaptive adversary could evade detection is by hiding (masking) their traces. Our detection method depends on an eBPF-based monitoring system that uses a perf event buffer to facilitate communication between the kernel and a user-space application[eBP25]. This buffer is continuously written into by the kernel, which collects events, and read by the script that processes and stores them. We assume an adversary familiar with our detection approach and the inner workings of eBPF.

The buffer is designed to prevent overflows i.e. data loss by stopping the kernel from writing new events when it becomes full until the application firstly clears older entries. While the buffer is full and waiting on the user space application, any new events collected are dropped. An adaptive adversary could exploit this by generating a large number of events before the actual attack, intentionally filling the buffer. As a result, subsequent events, including those produced during the attack, would not be logged, effectively allowing the attack to go unnoticed by our detection system.

Event loss in eBPF is a known issue, particularly when the rate of generated events exceeds the buffer's capacity. However, we can mitigate the impact of this by increasing the buffer size and using the lost field. This field, which can be included in the headers of buffer entries, tracks the number of events that were dropped while the buffer was full. Once space in the buffer becomes available again, the kernel inserts "record lost" messages, indicating how many events were lost during this time.

6 Concluding Remarks

In this section, we summarize the results of this thesis and propose ideas for future work.

6.1 Summary

This bachelor's thesis analyzes the impact of hypervisor-based attacks using malicious interrupts. We focus specifically on Heckler and WeSee attacks, which target confidential virtual machines (VMs) protected by AMD SEV. The primary goal of this thesis is to address the frequently overlooked task of detection.

In order to demonstrate if detection mechanisms can mitigate these attacks, we first recreated the Heckler attack. We then monitored the system in real time to collect extensive logs, capturing data during both normal use and under attack conditions. During the monitoring phase, we focused on collecting and identifying potentially suspicious system events using eBPF programs.

An analysis of these logs revealed that the overall number of events significantly increased during an attack, that certain events not observed in normal traces emerged, and that the sequence of executed events during the attack displayed distinctive patterns. These observations strongly suggest that detection is achievable.

A key finding of this thesis is that victims can detect these attacks by monitoring their kernel events. Specifically, the injected events associated with the attack manifest in the logging system as anomalies; events that typically cannot be observed during ssh and sudo authentication. Ultimately, we showed that reliable detection seems feasible even against adaptive adversaries, and our findings successfully answered the research questions posed by this thesis.

6.2 Future Work

For future work, we leave the development of detection tool, based on the findings of this thesis. Another possible future work is testing this detection logic against other interrupt numbers and deducing whether this sort of detection can universally prevent all injection-based attacks.

The future work regarding our proposed detection would be to fix the mentioned vulnerability, which could be exploited by an adaptive adversary. Finding a way to resolve and prevent eBPF's lost events would improve the quality of detection.

Bibliography

[AMD20] AMD. AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More, 2020. 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, as of April 8, 2025.

[AMD25] AMD. SEV-ES Guest-Hypervisor Communication Block Standardization, 2025. https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/specifications/56421.pdf, as of April 8, 2025.

[Ben24a] Benedict Schlüter and Supraja Sridhara and Andrin Bertschi and Shweta Shinde. WeSee Project, 2024. https://github.com/ahoi-attacks/WeSee/blob/main/README.md, as of April 8, 2025.

[Ben24b] Benedict Schlüter and Supraja Sridhara and Mark Kuhne and Andrin Bertschi and Shweta Shinde. Heckler Project, 2024. https://github.com/ahoi-attacks/heckler/blob/master/README.md, as of April 8, 2025.

[Buh21] Buhren, Robert and Jacob, Hans-Niklas and Krachenfels, Thilo and Seifert, Jean-Pierre. One Glitch to Rule Them All: Fault Injection Attacks Against AMD's Secure Encrypted Virtualization. 2021.

[DK21] Tom Woller David Kaplan, Jeremy Powell. AMD MEMORY ENCRYPTION, 2021. https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/memory-encryption-white-paper.pdf, as of April 8, 2025.

[Du,17] Du, Zhao-Hui and Ying, Zhiwei and Ma, Zhenke and Mai, Yufei and Wang, Phoebe and Liu, Jesse and Fang, Jesse. Secure Encrypted Virtualization is Unsecure. 2017.

[eBP25] eBPF.io. What is ebpf?, 2025. https://ebpf.io/what-is-ebpf/, as of April 8, 2025.

[HB17] Felicitas Hetzelt and Robert Buhren. Security Analysis of Encrypted Virtual Machines, 2017.

[Kap17] David Kaplan. PROTECTING VM REGISTER STATE WITH SEV-ES, 2017. https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/Protecting-VM-Register-State-with-SEV-ES.pdf, as of April 8, 2025.

[Man06] Manu Garg. Sysenter Based System Call Mechanism in Linux 2.6, 2006. https://articles.manugarg.com/systemcallinlinux2_6.html, as of April 8, 2025.

[Men21] Mengyuan Li and Yinqian Zhang and Huibo Wang and Kang Li and Yueqiang Cheng. CIPHERLEAKS: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel. In *USENIX Security Symposium*, 2021.

[MHHW18] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. Severed: Subverting amd's virtual machine encryption. In *ACM SIGOPS Symposium on Operating Systems Principles*, 2018.

[SBSS24] Supraja Sridhara, Andrin Bertschi, Benedict Schlüter, and Shweta Shinde. SIGY: Breaking Intel SGX Enclaves with Malicious Exceptions & Signals, 2024. https://arxiv.org/abs/2404.13998, as of April 8, 2025.

[SSBS24] Benedict Schlüter, Supraja Sridhara, Andrin Bertschi, and Shweta Shinde. WeSee: Using Malicious #VC Interrupts to Break AMD SEV-SNP. In *IEEE Symposium on Security and Privacy*, 2024.

[SSK+24] Benedict Schlüter, Supraja Sridhara, Mark Kuhne, Andrin Bertschi, and Shweta Shinde. Heckler: Breaking Confidential VMs with Malicious Interrupts. In *USENIX Security Symposium*, 2024.

[Ste25] Stefan Gast, Hannes Weissteiner, Robin Leander Schroder and Daniel Gruss. CounterSEVeillance: Performance-Counter Attacks on AMD SEV-SNP. In *Symposium on Network and Distributed System Security (NDSS)*, 2025.

[Tor25] Linus Torvalds. Linux kernel source code, 2025. https://github.com/torvalds/linux.git, as of April 8, 2025.

[WMA+19] Jan Werner, Joshua Mason, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. The SEVerESt Of Them All: Inference Attacks Against Secure Virtual Enclaves. In *ACM Conference on Computer and Communications Security (CCS)*, 2019.

[WWME20] Luca Wilke, Jan Wichelmann, Mathias Morbitzer, and Thomas Eisenbarth. SEVurity: No Security Without Integrity : Breaking Integrity-Free Memory Encryption with Minimal Assumptions. In *IEEE Symposium on Security and Privacy*, 2020.

[WWRE24] Luca Christopher Wilke, Jan Wichelmann, Anja Rabich, and Thomas Eisenbarth. SEV-Step: A Single-Stepping Framework for AMD-SEV. In *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024.

[Zha24] Zhang, Ruiyi and Gerlach, Lukas and Weber, Daniel and Hetterich, Lorenz and Lü, Youheng and Kogler, Andreas and Schwarz, Michael. CacheWarp: software-based fault injection using selective state reset. In *USENIX Security Symposium*, 2024.