00SEVen – Re-enabling Virtual Machine Forensics: Introspecting Confidential VMs Using Privileged in-VM Agents

Fabian Schwarz [*]
CISPA Helmholtz Center for Information Security
fabian.fsblack@gmail.com

Christian Rossow
CISPA Helmholtz Center for Information Security
rossow@cispa.de

Abstract

The security guarantees of confidential VMs (e.g., AMD's SEV) are a double-edged sword: Their protection against undesired VM inspection by malicious or compromised cloud operators inherently renders existing VM introspection (VMI) services infeasible. However, considering that these VMs particularly target sensitive workloads (e.g., finance), their customers demand secure forensic capabilities.

In this paper, we enable VM owners to remotely inspect their *confidential* VMs without weakening the VMs' protection against the cloud platform. In contrast to naïve in-VM memory aggregation tools, our approach (dubbed 00SEVen) is *isolated* from strong in-VM attackers and thus resistant against kernel-level attacks, and it provides VMI features *beyond* memory access. 00SEVen leverages the recent intra-VM privilege domains of AMD SEV-SNP—called VMPLs—and extends the QEMU/KVM hypervisor to provide VMPL-aware network I/O and VMI-assisting hypercalls. That way, we can serve VM owners with a *protected in-VM* forensic agent. The agent provides VM owners with *attested remote* memory and VM register introspection, secure pausing of the analysis target, and page access traps and function traps, all isolated from the cloud platform (incl. hypervisor) and in-VM rootkits.

1 Introduction

The security of virtual machines (VMs) is a crucial factor that can determine if customers are willing and regulatorily permitted to offload processes to a cloud platform. In order to increase trust into cloud platforms, researchers and vendors have developed several VM security and monitoring solutions rooted in the privileged VM manager, i.e., the hypervisor. These solutions range from kernel code integrity schemes and virtualization-based enclaves to forensic VM introspection (VMI) services [9, 35, 54, 64]. An impactful recent virtualization extension has introduced so-called *confidential VMs*. This security extension protects the confidentiality and integrity of a VM's memory and registers against the cloud platform, including the hypervisor and peripherals. Confidential VMs aim at enabling cloud adoption for highly sensitive customers that need to satisfy high security regulations and might distrust third party cloud providers, e.g., the finance and health sector. AMD SEV is the pioneering solution, which has so far been extended multiple times (SEV-ES, SEV-SNP) [10] and is widely available at cloud platforms [8, 30, 31]. Due to the high demand, other major CPU vendors provide their own confidential VM designs, e.g., Intel TDX and Arm CCA [16, 34].

However, while confidential VMs provide attestable hardware protection against the cloud platform, they are still prone to runtime compromises. Remote attackers can exploit vulnerable network services within the VM or perform software supply chain attacks to gain control over a confidential VM. For sensitive workloads, it is therefore crucial to monitor the VM for indicators of compromise. Hypervisor-based VMI is a well-explored forensic method to analyze a VM's memory and execution state, allowing to scan for such indicators, e.g., traces of rootkits [35]. Unfortunately, confidential VMs are inherently in conflict with VM security schemes that root their trust in the cloud hypervisor, including VMI. Furthermore, the protection of confidential VMs blocks major access methods required by these schemes—rendering them unfeasible.

Our goal therefore is to analyze these limiting factors and then propose a new design for secure *remote* introspection of *confidential* VMs. That way, we fill an important gap by reenabling VM owners, i.e., the cloud customers, to inspect the runtime states of their VMs (memory, registers) *without* sacrificing the security and confidentiality guarantees against a potentially malicious hypervisor (e.g., cloud vendor). At the same time, the approach offers reliable and instant VM introspection even if the VM is compromised (e.g., kernel-level rootkits) unless the in-VM attacker colludes with the cloud provider on delaying or preventing the analysis. We aim to enable VM owners to, e.g., scan for attacks or perform post-mortem digital forensics without leaking any inspection data to the hypervisor or in-VM attackers. This goal aligns with recent research aiming to re-enable other hypervisor-based

[*]We thank Saarland University for supporting Fabian Schwarz.

security schemes for confidential VMs (e.g., Hecate [28]).

To the best of our knowledge, we are the first to focus on the challenges of re-enabling secure remote VMI for confidential VMs. Previous work on inspecting confidential VMs is limited to an attacker perspective. Approaches like SEVered [46] have exploited the missing integrity protection of early SEV versions to leak memory and register content of a VM [45,47]. However, recent confidential VMs, e.g., SEV-SNP, feature integrity protection that fixes the root causes of these attacks. Existing out-of-VM forensic systems are blocked by SEV's memory protection [25,55,62]. VM owners currently must fall back on deploying forensic tools *inside* the confidential VMs, e.g., LeechAgent [26] or GPR [29]. However, these tools lack VMI features, e.g., VM pausing and traps. Even worse, they are *not* isolated from privileged in-VM malware (e.g., rootkits) providing system-level attackers full control over them and thus rendering them *insecure*.

In this paper, we present *00SEVen*, a design for secure remote introspection of confidential AMD VMs, even under a strong in-VM attacker. Our design introduces a *privileged in-VM agent* that exposes introspection capabilities via the network to the VM owner, e.g., a cloud customer. The VM owner can securely connect an analysis client (e.g., based on LibVMI [42]) to our agent to start a remote introspection session of the VM. By deploying our agent *inside* the confidential VM, the agent can access the *private* VM memory without being blocked by the memory protection while still being isolated from the untrusted hypervisor. Furthermore, our agent is protected against in-VM system-level attackers, offers hardware-based attestation of our VMI infrastructure, and supports VMI features *beyond* pure memory forensics, e.g., register access, VM pausing, and memory access traps.

However, these goals are not trivial to achieve for confidential VMs: The cloud hypervisor is *untrusted*, i.e., we can *not* rely on it to protect the agent against the in-VM OS. Furthermore, the hypervisor is still involved in important VM tasks, e.g., the scheduling of virtual CPUs, memory setup, and device I/O (e.g., networking). We therefore cannot simply apply existing isolation and introspection techniques [55,62]. Instead, 00SEVen builds on hardware-based in-VM isolation mechanisms and adds new secure VMI-assisting hypercalls. Our implementation targets confidential SEV-SNP VMs [10], which are widely available in server CPUs and at cloud platforms (in contrast to, e.g., Intel TDX) and provide primitives for intra-VM isolation [8,30,31]. 00SEVen leverages SEV-SNP's intra-VM privilege domains, called VM privilege levels (VMPLs) [10], to protect our in-VM VMI modules (incl. agent) and grant them full VM memory and register access. Our modules run in a bare-metal environment *independent* of the untrusted in-VM OS. 00SEVen deprivileges the in-VM OS (incl. user space services) by placing it in a less privileged VMPL with memory restrictions that protect our VMI against in-VM attackers [11]. 00SEVen ensures that VMI results are shared *only* with the authenticated VM owner. The VM owner can securely send VMI requests to the in-VM agent using a new *attested* end-to-end encrypted network channel. We enhance the QEMU/KVM hypervisor to support binding a virtual channel device and its I/O directly to our agent's privileged VMPL to operate the channel independent of the in-VM OS. Finally, we extend the VM-to-hypervisor interface of SEV with new VMI-assisting hypercalls used by our agent to securely offload sub-tasks to the hypervisor, e.g., to pause the untrusted in-VM OS during a consistent analysis.

We implemented an open-source prototype[1] of 00SEVen with support for the common LibVMI client library. That way, 00SEVen becomes compatible with all analysis scripts and tools building on LibVMI, e.g., Volatility's VMI plugin [56]. Our prototype includes the bare-metal in-VM VMI agent, our extensions to the Linux KVM/QEMU hypervisor, and an extended version of LibVMI usable by the remote clients. Our evaluation shows a reasonable performance and effectiveness based on practical analysis tasks and real-world rootkits.

In summary, we make the following contributions:

- We analyze the incompatibilities of *confidential VMs* with existing VMI techniques and derive the resulting challenges imposed on confidential VMI designs.
- We design 00SEVen, a remote VMI for confidential SEV-SNP VMs. 00SEVen's in-VM agent enables clients to control and inspect their VMs *while preserving* security.
- We implement 00SEVen for KVM/QEMU, including its in-VM agent, hypervisor extensions, and remote client/s.
- We analyze the security of our implementation and evaluate our LibVMI-compatible prototype (open-source) based on macro-benchmarks and real-world rootkits.[1]

2 Background: AMD SEV-SNP and SVSM

We now provide information on AMD SEV-SNP VMs (our VMI targets) and AMD's SVSM (used by our prototype).

2.1 Confidential VMs using AMD SEV-SNP

AMD SEV-SNP is an ISA extension that enables a new type of VMs, so-called confidential VMs (also: TEE VMs), whose memory and registers are protected against out-of-VM attackers, e.g., malicious hypervisors [10]. SEV-SNP generates and uses a unique crypto key in hardware for each confidential VM to encrypt a VM's private memory pages when storing them in system RAM. All VM code and page tables, as well as data pages marked for encryption in the in-VM page tables (via the "C-bit"), are treated as *private* pages, i.e., are encrypted by SEV-SNP. Access to private pages succeeds only from *within* the respective VM, thus protecting their confidentiality [13]. SEV-SNP protects the integrity of private pages and their address mappings using a reverse map table, which

[1]00SEVen prototype at: `https://github.com/sev-vmi/00seven`

keeps track of a VM's pages and thus enables denying out-of-VM writes to private pages. Moreover, SEV-SNP protects a VM's execution contexts by relocating the VM's save areas (VMSAs)—storing a VM's CPU registers on a VM exit (context switch)—into private memory. If the hypervisor requires register access, the in-VM OS must explicitly copy values to dedicated *shared*, i.e., unencrypted, memory [15]. Finally, SEV-SNP provides attestation support, enabling remote hosts to verify the protection and initial state (code, data) of a VM.

VMPLs: AMD SEV-SNP recently introduced VMPLs for providing intra-VM protection domains [10]. VMPLs are orthogonal to a CPU's kernel and user modes. They allow for multiple separate execution contexts per VM CPU (vCPU), with VMPL-specific memory access permissions to the private VM memory. SEV-SNP provides four hierarchically-ordered VMPLs, called VMPL0 to VMPL3, with VMPL3 being the least privileged. VMPL0 can exclusively access the full VM memory, adjust the memory views of all other VMPLs, and register ("validate") private pages to the VM, including VMSA pages. By default, the VM starts execution in VMPL0, while the others are unused. For each vCPU, the hypervisor requires one VMSA per used VMPL, i.e., up to four separate register sets per vCPU. A vCPU cannot switch between VMSAs (incl. VMPLs) by itself. Instead, before entering a VM, the hypervisor selects a vCPU's next VMSA to be scheduled, i.e., its register set and VMPL. The hypervisor *cannot* tamper with the protected VMSAs and their associated VMPLs, and the boot-time VMSAs are attested by SEV-SNP.

2.2 AMD Secure VM Service Module (SVSM)

AMD's SVSM provides a bare-metal VMPL0 execution environment in SEV-SNP VMs to host privileged services [11]. The QEMU/KVM hypervisor partitions the physical VM memory into a large VMPL1 region and a VMPL0-exclusive region at the top of the guest memory. The hypervisor maps the SVSM into the VMPL0 region but the regular BIOS/UEFI into the VMPL1 region, i.e., the VM OS is deprivileged inside VMPL1. While the VM OS can still execute all privileged kernel instructions in VMPL1, it can no longer execute VMPL0-exclusive ones: `pvalidate` for registering private pages to the SEV-SNP VM, and `rmpadjust` if used to mark pages for storing a vCPU's VMSA [14]. Therefore, SVSM manages the VMSA and vCPU setup, and exposes a hypercall-based service interface to VMPL1 that enables the VM OS boot code to request memory page registration, proxied via the SVSM. The VMSA and page setup steps occur only during boot up (except for hot-plug events), not restricting or slowing down the VM OS at runtime [10].

3 Setting: Confidential VM In(tro)spection

As shown in Figure 1, we envision an organization that wants to *securely* offload services to a third party cloud platform,

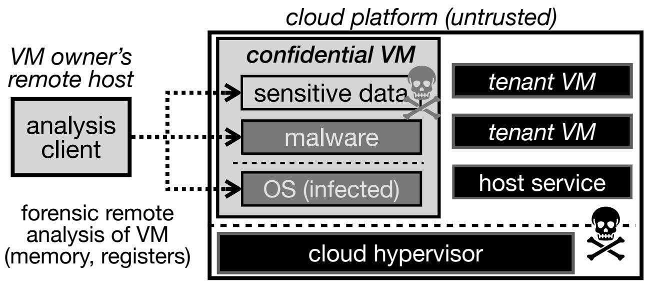

Figure 1: A client deploys a confidential VM at an untrusted cloud platform *(dark gray)*. As the VM might become compromised *(mid gray)*, the client wants to perform a remote analysis of the protected VM to scan for attack traces. *(light gray: trusted, dark: untrusted cloud, mid: untrusted in-VM)*

e.g., to benefit from the cloud's resource scalability and availability. These services operate on *highly sensitive* data, e.g., health or financial data, customer data, or IP assets (intellectual property). The client ("*VM owner*") therefore decides to deploy these services in confidential VMs, in our case based on AMD SEV-SNP. However, as the VM might become compromised *at runtime*, the VM owner wants to perform a remote forensic analysis to search for attack traces. Memory forensics typically covers the aggregation of the target's memory, the identification of data structures, and finally the analysis tasks [18, 27, 43]. The VM owner plans to augment forensics with *VM introspection (VMI)*, which leverages the hypervisor's control over the VMs to expand memory forensics with additional techniques, including on-demand pausing of the VM, CPU register inspection, and event-based analysis [35, 42, 55]. Such runtime attack detection in confidential VMs implies a new threat model, as discussed next.

3.1 Threat Model

Our threat model combines aspects of the model of remote VMI for non-confidential VMs and that of confidential VMs. We refer to the former as "classical VMI". We trust the VM owner that deploys confidential SEV-SNP VMs in a third party cloud and wants to remotely inspect them using a trusted client system *(light gray, Figure 1)*. We regard the network path between clients and VMs as untrusted. In contrast to the classical VMI model, the VM owner does *not* fully trust the cloud platform. We therefore follow the model of confidential VMs in that we assume all system-level software of the cloud platform (incl. the hypervisor) and co-located VMs to be untrusted, e.g., compromised by an attacker *(dark gray)*, and trust the cloud's CPU(s) and confidential VM implementation *(light gray)* to be secure and free of exploitable vulnerabilities, i.e., in our case, the SEV-SNP extension and attestation [10].

We follow a stronger model than confidential VMs regarding the VM security in that we do *not* regard the *whole* confidential VM as trusted. Instead, similar to the classical VMI model, we assume a potential in-VM attacker (*mid gray*) that has compromised the VM OS (e.g., malware, rootkits) that the VM owner wants to remotely detect or analyze [27,35]. An attacker might have gained control over the VM for instance by exploiting in-VM network services or software supply chain attacks against the VM's package managers.

We assume the in-VM attacker to *be distinct from* the cloud operator and *not* in control of the cloud platform (Figure 1). That is, we explicitly exclude collusion attacks of the in-VM attacker *(mid gray)* and the untrusted cloud platform *(dark gray)* as they could otherwise give running VMs an opportunity to remove attack traces prior to inspection. In § 6.1, we will discuss the security impact of collusion attacks and the limiting factors of a full mitigation using SEV-SNP in detail.

Finally, we exclude all side-channel and hardware attacks that go beyond the guarantees of SEV-SNP and refer to orthogonal research on these topics, e.g., Cipherfix [57]. Similarly, we regard the *prevention* of denial of service (DoS) attacks issued by the cloud platform as out of scope, because current confidential VMs cannot prevent them either [10]. However, our VMI design will explore if we can detect or limit DoS attacks by the hypervisor against our VMI operations. Furthermore, we will discuss in-VM attackers trying to detect and delay (or even DoS) analysis attempts to hide from them.

3.2 Design Goals and Requirements

To guide our secure remote introspection of confidential VMs (00SEVen), we define *eight* major design requirements (*R1–R8*) that capture the functional and threat model-specific security demands for a practical solution. In addition, we define two desirable extra goals (*E1+E2*) going slightly *beyond* our threat model achievable with the current hardware support.

R1 Remote Memory and Register Access: 00SEVen must provide the VM owner full remote access to the VM's memory and virtual CPU registers (vCPU).

R2 Consistent Analysis: 00SEVen must support secure pausing of the VM for a consistent memory and register analysis. Cloud attackers must not be able to resume VM execution without an explicit approval by 00SEVen.

R3 Event Traps: 00SEVen must enable event-based analysis by supporting secure traps on VM read/write accesses to monitored memory pages or calls to kernel functions.

R4 Isolation from In-VM OS-level Attackers: 00SEVen's VMI components and analysis results must be protected against in-VM OS-level attackers to enable secure analysis of user malware and kernel rootkits.

R5 Isolation from Cloud Attackers: 00SEVen's VMI components and analysis results must be protected against the cloud platform (incl. the hypervisor) in line with the threat model of confidential VMs (here: SEV-SNP).

R6 Secure Communication Channel: The network communication between 00SEVen's in-VM VMI components and the VM owner must be protected against passive (sniffing), active (tampering), and impersonation attacks by in-VM, cloud platform, and network attackers.

R7 Small TCB: 00SEVen should keep the TCB and attack surface small to minimize the risk of a compromise.

R8 Small Overhead on VM Workload: 00SEVen should minimize the extra overhead imposed on the confidential VM's workload while no introspection is active.

(E1 Detect Analysis DoS:) 00SEVen should enable the VM owner to detect DoS attempts (e.g., scheduling-based) by cloud attackers (cf. *R5*) against VMI operations.

(E2 Hide Analysis from In-VM Attackers:) 00SEVen should support hiding incoming remote analysis requests from in-VM attackers, e.g., to prevent attackers from hiding attack traces just in time (cf. *R4*).

00SEVen provides the foundation for secure remote VMI of *confidential* VMs, using the example of AMD SEV-SNP. We encourage future work to build on top of 00SEVen in order to securely explore further VMI features and optimization techniques [21, 24, 35, 62] or transfer our concepts to other platforms, as discussed in § 8.2 for Intel TDX and Arm CCA.

3.3 (Un)Applicability of Existing VMI

Existing hypervisor-based VMI systems are inherently in conflict with our threat model. These systems rely on a *trusted* hypervisor to control the target VMs and securely access their memory or register content for the analysis [42]. However, in our setting, the VM owner assumes the cloud provider to be *untrusted* (cf. Figure 1 and *R5*). In fact, the hardware protection of SEV-SNP (cf. § 2.1) renders *any out-of-VM* approach unfeasible. SEV-SNP blocks any out-of-VM inspection attempts, including memory inspection [55, 62], injection of forensic code [32], and vCPU register inspections.

Existing *in-VM* VMI approaches suffer from functional and security issues. While in-VM forensic tools (cf. § 1) can successfully access the private VM memory, they are *unprotected* against in-VM OS-level attackers. Therefore, in-VM attackers can easily tamper with the tools and their results, violating *R4* and *R6* (§ 3.2). In addition, they lack several VMI features, e.g., secure VM pausing for a consistent analysis (*R2*) and VM traps (*R3*). Unfortunately, it is non-trivial to protect and securely extend existing in-VM tools within our threat model. In SEV-SNP (and other confidential VMs), several important resources required by existing VMI systems are still handled by the untrusted hypervisor, which prevents a direct transfer of out-of-VM techniques [32, 55, 62]. The hypervisor controls the scheduling of vCPUs and the second-level memory mappings (incl. permissions) via nested page tables (NPTs). Furthermore, the untrusted hypervisor is still in the sole control of vCPU event interception [13]. Therefore, in-VM tools can neither use NPTs to isolate their memory space from in-VM attackers [32,55,62] (*R5*), nor pause vCPUs for a consistent introspection (*R2*), nor directly monitor page accesses or intercept VM executions (*R3*) [28].

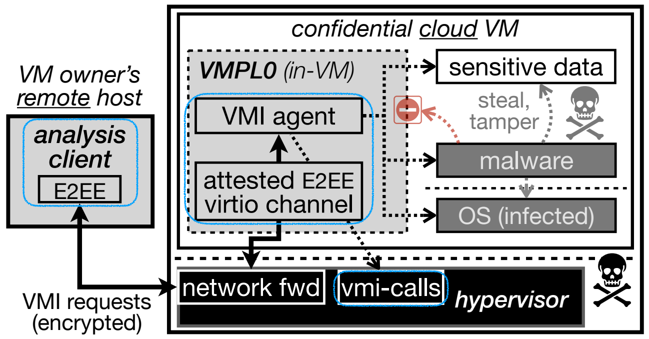

Figure 2: 00SEVen's design: secure in-VM agents enable remote VMI of confidential VMs. VM owners query the agents via attested end-to-end encrypted (E2EE) network channels. *(light gray: trusted, dark: untrusted cloud, mid: untr. in-VM)*

4 Design of 00SEVen

Next, we present the design of 00SEVen, our remote VMI solution. In § 5, we focus on its implementation details.

4.1 Design Overview

As shown in Figure 2, 00SEVen combines a secure *in-VM* agent with VMI-specific hypervisor extensions. Together they provide the VM owner with remote analysis capabilities (*R1*), securely overcoming the limitations imposed by SEV-SNP. Our new VMI agent forms 00SEVen's in-VM TCB, while the rest of the VM and the cloud platform stay untrusted. In order to start a remote VMI session, the VM owner executes a 00SEVen-compatible forensic client application on a trusted system. The client connects via a network (reverse) proxy running at the cloud platform to our agent inside the SEV-SNP VM and uses SEV's hardware-assured remote attestation to authenticate it [12]. The VM owner then uses the client to issue VMI requests to our in-VM agent as required for the forensic analysis. The agent performs the requested VMI operations on behalf of the VM owner, e.g., accessing memory or register content (described in § 4.2), and returns the results. The workflow of 00SEVen will be familiar to users of the common LibVMI [42] framework. However, in contrast to LibVMI, which interacts with a local hypervisor, our client library *remotely* communicates with 00SEVen's securely attested *in-VM agent*, not with the untrusted cloud hypervisor.

In-VM Agent Our in-VM agent forms the core of 00SEVen's VMI. The agent is responsible for processing the introspection commands of the analyst by implementing the respective VMI operations (cf. § 4.2). While SEV-SNP protects the in-VM agent against out-of-VM attackers (incl. the hypervisor), it is crucial for a secure VMI that the agent is also protected against in-VM attackers, e.g., a compromised VM OS (*R4–R6*). Therefore, we leverage SEV-SNP's VM privilege levels (VMPLs) for in-VM isolation [10] (cf. § 2.1). We deploy the modules of our VMI agent inside VMPL0, which forms a special in-VM management domain. VMPL0 provides our agent with full VM memory access (*R1*) and the capability to define per-VMPL memory permissions (read, write, execute) that restrict access of less privileged VMPLs to a memory subrange. In addition, VMPL0 securely manages the VMSA pages and therefore enables our agent full access to all registers of each vCPU (*R1*). 00SEVen deprivileges the untrusted VM OS and user services inside a less privileged VMPL (cf. § 2.2). That way, at boot time, our agent can define VMPL memory permissions for the less privileged VMPLs that isolate our agent and the vCPU registers, i.e., VMSA pages, from in-VM attackers (*R4*). For ease of discussion and without loosing generality, we assume the VM OS and services to be relocated only into VMPL1, ignoring the even less privileged VMPL2 or VMPL3. The VM OS running in VMPL1 can still execute every privileged CPU instruction except those restricted to VMPL0 (see § 2.2). Our agent executes in a bare-metal environment without an OS kernel, *independent of* the potentially compromised VM OS (*R8*). By refusing the use of a full-blown OS inside VMPL0, (in contrast to, e.g., Hecate [28]), we keep 00SEVen's TCB and attack surface significantly smaller (*R7*).

Hypervisor Integration We extend the untrusted hypervisor to assist 00SEVen's in-VM agent with scheduling, remote communication, and VM control primitives. Together with new in-VM security checks, that way, our agent can securely enable remote VMI despite the untrusted hypervisor's VM control. During regular workloads, we want the hypervisor to execute the VM OS in VMPL1 without additional overhead by our VMI system (*R8*). On remote VMI requests by the VM owner, we expect the hypervisor to schedule our VMPL0 agent to perform VMI operations. However, existing hypervisors (e.g., KVM) do not yet distinguish between different VMPLs when scheduling vCPUs or delivering I/O events. We therefore extend the hypervisor with *VMPL-aware* scheduling and I/O operations. That way, we can bind a virtual I/O device for our VMI remote channel exclusively to VMPL0, i.e., our agent's domain. The channel device demands scheduling of VMPL0 from the hypervisor on VMI requests and permits channel I/O only by our agent. All other I/O devices (e.g., disk, NIC) stay associated with the VM OS in VMPL1, keeping their performance unaffected by our agent (*R8*). When finishing the requested VMI operations, our agent hyper-calls into the hypervisor to re-schedule VMPL1 execution. In § 4.2, we will present 00SEVen's hypervisor interfaces for VMI control primitives: VM pausing and trapping.

Secure Client-to-Agent Communication The VMI channel between 00SEVen's in-VM agent and the VM owner's remote client requires additional protection. In the current SEV-SNP design, all hardware device I/O must pass through the untrusted hypervisor. Therefore, we must rely on a packet forwarding service at the hypervisor-level to forward our VMI channel messages via the network to the remote client (cf. Figure 2), affecting their security (*R6*). In order to protect our VMI messages, the remote client and the in-VM agent use an end-to-end encrypted (E2EE) connection (*R6*). Otherwise, attackers could tamper with the messages to hide attack traces or leak private VM memory by exploiting inspection requests. 00SEVen's connection endpoint is isolated from out-of-VM (cloud, network) and in-VM attackers by placing the protocol stacks directly inside VMPL0—letting packets leave VMPL0 only in E2EE form. As the agent operates only on the (E2EE) application-layer messages rather than full network packets, we preserve a small in-VM TCB (*R7*). Being located in VMPL0, the VMI channel operates independent of the untrusted VM OS in VMPL1 (*R8*)—not passing any packets through the VMPL1 network stack. To prevent impersonation attacks, we combine certificates with SEV-SNP's remote attestation for mutual authentication (see § 5.3).

4.2 VMI Work Flow

We now describe 00SEVen's VMI (*R1*) and its hypervisor extensions for secure VM pausing (*R2*) or event traps (*R3*).

4.2.1 Modus Operandi

00SEVen supports multiple forms of remote analysis triggers and modes. A typical use case is the scanning for attack traces or active malware by the VM owner as part of an incident response process, e.g., triggered by an intrusion detection system or as part of a periodic security check. Beyond manually or periodically triggered analysis, 00SEVen re-enables more advanced use cases by supporting event-based triggers based on page access monitoring or code execution traps (cf. § 4.2.5). That way, 00SEVen can notify the client-side analysis script, e.g., if an in-VM attacker tries to tamper with a memory page of a sensitive service (*R3*). 00SEVen's main analysis mode then enables the remote analyst (VM owner) to perform a *consistent* analysis of the VM by providing secure pausing of the untrusted VMPL1 services, i.e., the VM OS and user processes (cf. § 4.2.4). In contrast to existing forensic tools that require downloading a full VM memory dump for a client-side offline analysis, 00SEVen enables interactive and selective remote memory and register introspection of the VM state and thus better scalability by avoiding gigabyte-size memory dump transfers. Depending on the analysis results, the VM owner can then quarantine or resume the VM.

4.2.2 Remote VMI Interface

00SEVen's remote interface enables flexible analysis tasks by the VM owner. 00SEVen's in-VM agent implements fundamental operations required for VMI and exposes them via

```
 1  connect_to_agent(target_vm)
 2  pause_vmpl(OS) // consistent, like LibVMI
 3  tentry = ksym_va("init_task") + tasks_off
 4  while (true) { // scan the list
 5      proc = tentry - tasks_off
 6      exec = read_str_va(proc + comm_off)
 7      if (exec == "malware") { ... }
 8      ...
 9  }
10  resume_vmpl(OS) // VMPL1 OS
11  disconnect_from_agent()
```

Figure 3: Simplified excerpt of a client script scanning the remote VM's process list for malware. *(_va: VM virtual addr.)*

an RPC-like interface to the remote client. These operations form the basis for high-level VMI tasks by providing memory and register access (cf. § 4.2.3). Furthermore, the agent exposes secure VM control primitives for pausing (cf. § 4.2.4) or trapping (cf. § 4.2.5) the VM. That way, analysts have full control of the VMI and can implement flexible analysis scripts tailored to their use cases. Our operations are similar to the features of the common LibVMI framework [42], which makes it possible to adopt many existing LibVMI client analysis scripts and tools built on it, e.g., Volatility's VMI plugin [56]. In our prototype, the remote analysis client is based on LibVMI. We extend LibVMI with a new 00SEVen driver, i.e., API backend, which sets up the attested E2EE connection and sends VMI operation and VM control requests to our agent, instead of interfacing with a local hypervisor.

Figure 3 shows a simplified client script for remotely inspecting the process list using 00SEVen. In the preamble (lines 1+2), the VM owner's client connects to 00SEVen's in-VM agent and requests secure pausing of the VM OS for a consistent analysis (cf. § 4.2.4). Afterwards, the client resolves symbols to get the address of the process list of the Linux-based VM OS (line 3) and issues multiple memory read requests to 00SEVen's agent to scan the list (lines 4–9). The agent receives the requests via the E2EE channel and performs the respective memory accesses inside the VM. As we will discuss in § 4.2.3, the client might perform some steps locally to speed up the analysis process, e.g., by caching page table information [21, 42]. Finally, in the epilogue (lines 10+11), the client resumes the VM execution if no malware has been found and disconnects from the agent.

4.2.3 VMI Operations

We now outline the basic VMI operations 00SEVen supports.

Physical Memory Access 00SEVen's basic memory access operation takes a physical memory address of the VM (PA_{vm}) as input. Our agent maps the PA_{vm} and then uses the resulting virtual address (VA_{vmpl0}) to access the page, returning the requested content to the remote analyst. In contrast to out-of-VM VMI, there is no need to explicitly translate the VM address to a host-level virtual address, as it will be automatically handled by the hardware on the in-VM access. Before each access, the agent must check that the requested physical address range does not overlap with the VMPL0-exclusive memory region containing our agent's code and data (§ 5.1). Otherwise, in-VM attackers might maliciously modify kernel pointers or remap page table entries to let them point into the VMPL0 region to cause a corruption of 00SEVen on incautious VMI write requests. Furthermore, the agent must ensure that the PA_{vm} is correctly mapped as private (encrypted) or shared page as registered in SEV-SNP (§ 2.1). This information must be encoded as a bit ("C-bit") in each PA_{vm} but might be unavailable or untrusted when the PA_{vm} is taken from VMPL1's memory or page tables. As VMPL0 registers private pages in SEV-SNP (cf. § 2.1), the agent can keep track of each page bit to correctly map them, requiring only 1 MiB for a VM with 32 GiB RAM and 4 KiB page size. Optionally, the agent can pre-map all pages linearly ($VA_{vmpl0} = PA_{vm} + \text{offset}$) for a direct VA_{vmpl0} lookup.

Virtual Memory Access For accessing virtual kernel or process addresses of the VMPL1-located VM OS, 00SEVen requires an address translation step. Our VMPL0 agent uses page tables (PTs) separate from those of the untrusted VM OS. That way, in-VM attackers cannot tamper with our agent's address space, e.g., by remapping PT entries, thus preserving 00SEVen's isolation guarantees. Consequently, for accessing a virtual address of the VM OS (VA_{vmpl1}), 00SEVen must translate the VA_{vmpl1} to a PA_{vm} before the agent can access it via a VA_{vmpl0}, as described before. The translation requires a page table walk through the respective PTs of the VM OS [43]. Depending on the virtual address space of the VA_{vmpl1}, the physical address (PA_{vm}) of the respective root PT (directory table base) can be located using different existing methods [43]: for kernel VAs based on the Linux 'init_top_pgt' kernel symbol, for process VAs inside the OS process list (Figure 3), or for the current address spaces in the `CR3` registers accessible in the vCPUs' VMSAs (§ 3.3). We refer to the forensic literature for more details, e.g., [43]. With our remote client being based on LibVMI (§ 4.2.2), the client performs kernel symbol translations locally, e.g., based on the symbol table of the compiled VM Linux kernel [51]. For walking the VM OS page tables, our remote client issues the respective physical memory and `CR3` read requests to our agent as required for the $VA_{vmpl1} \Rightarrow PA_{vm}$ translation. After translation, 00SEVen's agent can then map and access the target virtual address via the resolved PA_{vm}, before returning the result to the client.

Note that while the separation of VMPL0 and VMPL1 PTs requires an additional address translation step, it is important to preserve the isolation of 00SEVen's VMI agent from in-VM attackers. In Appendix A, we argue why existing VMI techniques that directly share PTs between the VMI agent and the target VM OS—eliminating the translation steps—are *not secure* for confidential SEV VMs. Furthermore, we explain how 00SEVen mitigates the translation overhead by adopting LibVMI's client-side caching strategies, and outline future ideas for offloading address translation steps to the agent.

Virtual CPU Register Access Our agent securely manages the register save states (VMSAs) of the vCPUs. The VMPL0 setup code allocates and registers one VMSA page per VMPL for each vCPU [15]. When a vCPU of an SEV-SNP VM is yielded, the CPU stores the general purpose, control, and virtualization registers of the vCPU in the respective private VMSA page (cf. § 3.3). Therefore, our agent can directly inspect the register state of paused vCPUs (§ 4.2.4). To prevent in-VM attackers from tampering with vCPU registers, our agent protects VMSA pages using VMPL permissions.

4.2.4 Secure Pausing for a Consistent Analysis

00SEVen supports secure pausing of in-VM attackers for a consistent memory and register introspection (*R2*). That is, 00SEVen stops in-VM attackers from tampering with the VM memory and registers during the analysis, e.g., to hide attack traces via page table manipulation. But 00SEVen must not fully stop vCPU execution because the in-VM agent must still perform the analysis. Instead, only the execution of the untrusted VM OS and user space services should be paused, i.e., the VMPL1 domain. Furthermore, in our threat model, 00SEVen cannot trust the hypervisor to keep VMPL1 paused throughout the analysis. Therefore, we temporarily disable virtualization support in the `EFER` CPU control registers of all VMPL1 VMSAs to prevent their execution while performing the analysis in VMPL0. First, we extend the hypervisor with two new hypercalls from VMPL0 to the hypervisor—one to request pausing of all VMPL1 contexts, yielding them if active, and one for resuming them. Second, on a pause request, we let our VMPL0 agent iterate all VMPL1 VMSAs, i.e., saved register states (§ 3.3), and atomically unset the virtualization-enable CPU register bit `EFER.SVME`. If the register updates succeed, VMPL1 has been paused by the hypervisor, and we have *locked* VMPL1 execution, i.e., all attempts by the hypervisor to resume it will be blocked by the CPU. If the hypervisor maliciously ignores a pause request, e.g., to keep colluding in-VM rootkits scheduled (cf. § 6.1), the CPU register updates will fail as the VMSAs are not paused [11], causing the agent to notice and retry. The remote analyst will detect the attack, as the analysis will not proceed (*E1*). On a resume request, the agent re-enables all VMPL1 VMSAs by setting their `EFER.SVME` and requests their scheduling by the hypervisor. As shown in Figure 3 (lines 2 and 10), we expose pause and resume APIs to the remote analyst.

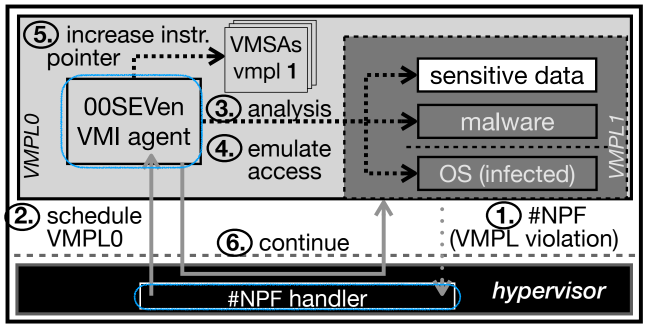

Figure 4: 00SEVen's VMPL0 agent combines VMPL permissions with instruction emulation for secure page monitoring. *(light gray: trusted, dark: untrusted cloud, mid: untr. in-VM)*

4.2.5 Event-based VMI

00SEVen provides support for event-based VMI (*R3*): SEV-enabled memory access traps and kernel function traps.

Page R/W-Monitoring 00SEVen enables the remote analyst to request the monitoring of read and/or write accesses to private VM pages. That way, the analyst can for instance write-monitor a function pointer table in the VM OS, e.g., the system call table, to detect malicious tampering attempts by a rootkit. Existing VMI implements page monitoring using NPT permissions which we cannot trust [58]. Instead, to monitor a page, our agent securely modifies the VMPL1 memory permissions for that page to non-readable or non-writable using SEV's `RMPADJUST` CPU instruction, restricting access of the VM OS. On an access trap, the agent informs the analyst and waits for VMI requests while keeping the VM OS paused.

Figure 4 shows 00SEVen's control flow on an access trap. VMPL1's access is blocked by the VMPL permission and results in a nested page fault (NPF) at the untrusted hypervisor [13] (*step 1*). We extend the hypervisor's #NPF-handler to forward such VMPL violations to our agent by scheduling VMPL0 (*step 2*). The agent inspects the #NPF details (given in the vCPU's VMSA [13]) and, if a violation is associated with a monitored page, securely pauses the vCPU's VMPL1, notifies the remote analyst, and waits for analysis requests (*step 3*). After the analysis, our agent must proceed execution of VMPL1. Existing hypervisor-based VMI grants VMPL1 *temporary* page access and uses hardware single-stepping to securely perform the trapped access [58]. However, only the untrusted hypervisor can intercept the single-stepping exception, and the hypervisor could simply resume VMPL1 execution without informing our agent, resulting in relaxed VMPL permissions, i.e., a disabled trap. Therefore, instead, our agent reads VMPL1's instruction pointer register (RIP) to decode and emulate the violating memory access, i.e., performing it on behalf of VMPL1 (*step 4*) [38]. That way, the VMPL restrictions are never relaxed such that we do not risk disabled monitoring traps. During emulation, our agent translates the VMPL1 memory address using the vCPU's VMPL1 PTs, checks that it does not overlap with VMPL0 memory, and then maps it temporarily into VMPL0. We must extract the page offset of the target address from the instruction itself, as SEV masks the offset in the #NPF details for security reasons [41, 59]. Finally, after access emulation in VMPL0, the agent updates the VMPL1 registers in the VMSA, including the RIP to step over the emulated instruction (*step 5*), clears the #NPF to prevent a fault replay [28], and calls into the hypervisor to resume execution (*step 6*). As VMPL0 has exclusive control over the VMSAs and VMPL permissions, the monitoring is protected against the untrusted hypervisor and in-VM attackers (details in Appendix B).

Kernel-Function Traps Conceptually, 00SEVen also supports execution traps for the VMPL1 OS kernel. That way, an analysis is triggered on execution of a certain OS function, e.g., a system call [60]. This is realized by injecting VMPL0 trampolines at the beginning of VMPL1 kernel functions. For more technical background, we refer to Appendix C.

5 Implementation

We now describe details of our 00SEVen implementation for the QEMU/KVM hypervisor and a VM with Linux OS.

5.1 Agent Integration and Startup

We currently implement 00SEVen's in-VM agent and its modules as an extension to AMD's SVSM infrastructure [11] (see § 2.2). The SVSM provides a bare-metal VMPL0 environment written in Rust, which handles the vCPU (i.e., VMSA) and memory setup that is specific to SEV-SNP. QEMU/KVM maps 00SEVen together with the SVSM into a VMPL0-exclusive memory region, separated from the VM OS. 00SEVen's agent builds on SVSM's memory and VMSA management facility to provide analysts with our new secure VMI and communication channel infrastructure, isolated from in-VM attackers. On VM startup, the SVSM sets up the vCPUs (i.e., VMSAs) and the VMPL memory protection of the VMPL0 region, starts 00SEVen's VMI agent, and transfers control to VMPL1 for the Linux boot process. Figure 5 shows our implementation, excluding SVSM's modules.

During startup, 00SEVen's agent prepares the network communication with the remote analyst. The agent first initializes the dedicated virtual channel device with the hypervisor and then sets up a raw server socket that asynchronously listens for a remote connection by the VM analyst. On a successful connection, the agent starts a VMI session controlled by the remote analyst's operation requests, as described in § 4.2.

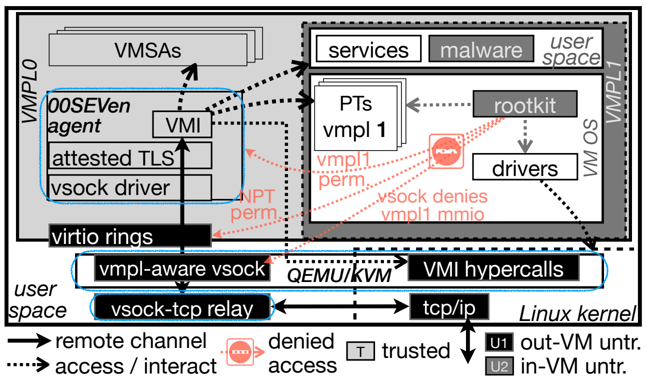

Figure 5: 00SEVen's implementation: in-VM agent, network relay, and VMPL-aware hypervisor extensions *(blue)*. *(light gray: trusted, dark: untrusted cloud, mid: untr. in-VM)*

5.2 Channel Device and Scheduling

We extend QEMU/KVM's VM setup process to prepare 00SEVen's VMPL-aware remote channel (§ 4.1). We adapted QEMU to allocate a dedicated virtual MMIO (memory-mapped I/O) page next to the VMPL0 memory region and associate a virtual MMIO-based I/O bus with it (virtio-mmio [49]). We bind the bus to VMPL0 (cf. next) and attach a virtual socket device (vsock) [49] to it as our remote channel device. A vsock device enables hypervisor services—in 00SEVen's case, a network relay—to connect to a socket-like interface in the VM and exchange messages without requiring complex network stacks (*R7*). We ported the required virtio drivers [5] into VMPL0 (Figure 5) and implemented SEV-SNP support for them in line with SEV's shared memory-based GHCB interface [15] (Guest-Hypervisor Communication Block), such that our agent can interact with the channel device via read and write GHCB requests to the MMIO page.

VMPL-aware MMIO We extend QEMU/KVM to support VMPL-aware virtual MMIO devices. On each virtual MMIO access by a VM, QEMU virtio-mmio devices can now inspect the accessor's VMPL and deny the access based on that. 00SEVen uses this mechanism to bind its remote channel device to VMPL0 (i.e., our VMI agent) by making its bus permit MMIO operations only by VMPL0, preventing any MMIO access by VMPL1 attackers. On a VM exit of a vCPU, our extension augments KVM's `kvm_run` structure ($\widehat{=}$ interface to QEMU) with the current VMPL of that vCPU. Furthermore, we extend QEMU's MMIO device callbacks to propagate the VMPL as a new access attribute to the MMIO target device.

VMPL-aware Scheduling As described in § 4.1, we extend QEMU/KVM with VMPL-aware scheduling requests. That way, 00SEVen's channel device can demand explicit scheduling of VMPL0 on VMI requests by the remote analyst, such that the agent can read the channel and process the requests (§ 4.2). On a scheduling request, QEMU/KVM yields the vCPU, lets it switch from its VMPL1 register set (VMSA) to its VMPL0 set, and resumes the vCPU. Note that by default, a vsock device would instead inject an interrupt (IRQ) into the VM on a new message. However, QEMU/KVM does not yet support VMPL-aware IRQ delivery. Furthermore, in SVSM, VMPL0 executes with masked IRQs in SEV-SNP's restricted injection mode, which prevents hypervisors from arbitrarily injecting interrupts into VMPL0, decreasing 00SEVen's attack surface [10]. In Appendix D, we provide technical details on IRQ issues we had to overcome when scheduling VMPL0.

5.3 Attested Remote Communication

The remote communication of 00SEVen's agent with the VM analyst is built on top of the vsock channel. 00SEVen uses the channel to pass their messages via shared virtio rings between VMPL0 and a network relay service at the hypervisor which forwards them via TCP/IP through the network, as shown in Figure 5. Technically, the agent and client exchange VMI requests and results (§ 4.2) using two separate transport layers: VSOCK for the agent–relay interconnect and TCP/IP for the relay–client one. To protect the messages, we integrate a TLS server endpoint into our agent and a TLS client endpoint into the remote analysis client. That way, the communication is end-to-end protected against in-VM and out-of-VM attackers even though it passes through the untrusted hypervisor.

Network Relay The relay is a Linux user space service that manages a VSOCK client and a TCP server socket (e.g., socat). The relay waits for an incoming remote client TCP/IP connection by the VM analyst. Upon receiving a connection, the relay connects to our in-VM agent via the vsock channel device and then starts forwarding packet payloads, i.e., TLS messages, between the agent and remote client.

Authentication and Attestation 00SEVen's TLS channel combines mutual certificate-based authentication with AMD SEV's remote attestation [12]. We use a client TLS certificate that is pinned by 00SEVen's agent to verify the VM owner's remote client, e.g., shipped to the agent inside an encrypted VM disk image. To enable authentication of the agent, we bind the TLS connection to the hardware-assured attestation report of SEV. The attestation measurement covers the VM's load-time state including 00SEVen's SVSM image with all agent modules. In addition, the SEV hardware adds the VMPL of the report-generating VM component to the attestation report. Therefore, the VM owner can remotely verify that it is in fact communicating with the VMPL0-protected agent of the owner's VM, not an attacker-controlled VM or an imposter agent in VMPL1. To bind the TLS connection to the attestation, the agent adds the hash of a fresh TLS server public key to the attestation report and sends the report via TLS to the client [37]. The client verifies the binding by checking if the keys in the agent's TLS certificate and the report match.

5.4 VMI-assisting Hypercalls

00SEVen adds new hypercalls to QEMU/KVM that securely support the agent's VM control primitives (cf. § 4.1). The hypercalls are commands without arguments: pausing or resuming the VMPL1 contexts (§ 4.2.4), switching back to VMPL1 after an r/w-trap (§ 4.2.5) or to VMPL0 on a function trap (Appendix C). The calls are implemented as new GHCB requests, i.e., extend SEV's facility for guest-to-hypervisor communication [15]. The VMPL switches change a vCPU's active VMSA (§ 4.1). On a VMPL1 pause request, we prevent the vCPU/s running the VMI agent from switching back to VMPL1 and let QEMU pause (later resume) all other vCPUs.

6 Security Analysis

We now analyze the security of 00SEVen against two non-colluding adversaries: in-VM and out-of-VM attackers (see § 3.1). The in-VM attackers aim to evade detection, e.g., by compromising the agent. The out-of-VM attackers aim to gain access to the private memory and register values of the VMs or their VMI results, e.g., by tampering with VMI operations. Considering these two attackers, 00SEVen aims to protects the VMI operations and their requests and enables the detection, prevention, or analysis of in-VM attacks as follows.

00SEVen's VMPL0-located agent forms its in-VM TCB. The agent's security is built on top of SEV-SNP's hardware-enforced memory and register protection [10]. Inside VMPL0, the agent is protected against out-of-VM cloud attackers and can leverage VMPL permissions to block access attempts by in-VM VMPL1 attackers. The client uses SEV's remote attestation to verify the protection and initial state of the VM, including the VMPL0 and VMPL1 code and data. That way, the client can detect manipulated VMI agents or VMPL1 boot-up code, preventing, e.g., attempts to integrate backdoors or map a second, hidden VM OS to an unknown address.

As SEV VMs rely on support by the hypervisor, the same holds for 00SEVen. Beyond scheduling and memory setup, 00SEVen offloads new VM control tasks to the hypervisor via new hypercalls (e.g., pausing, VMPL switch). However, the hypercalls expose only a minimal attack surface, and 00SEVen is designed to actively prevent malicious hypervisor behavior on these tasks (e.g., VMPL1-locking on pausing) or remotely observe it as anomalous VMI freezes. 00SEVen's pausing of VMPL1 can therefore securely enable consistent memory forensics, i.e., in-VM attackers cannot manipulate or remap any data during the analysis, thus preventing attempts to hide attack traces. On memory introspection, the range and C-bit checks of the VMPL0 agent (cf. § 4.2.3) rule out attacks that map pages or pointers to VMPL0 or unprotected memory.

The network forwarding of 00SEVen's remote channel is also offloaded to the hypervisor, but the communication is TLS-protected and authenticated using certificates and SEV's remote attestation. Therefore, neither out-of-VM nor in-VM attackers can tamper with or leak VMI operation requests or results. The channel's interface to the untrusted hypervisor is based on MMIO-based virtio [49] such that it exposes a minimal attack surface—in contrast to PCIe-based device I/O.

00SEVen currently relies on cooperation by the untrusted hypervisor to protect MMIO and shared pages against access by in-VM attackers, because SEV-SNP's VMPL permissions are enforced only for *private* VM pages [13]. 00SEVen's QEMU/KVM extension for VMPL-aware MMIO blocks MMIO accesses by VMPL1 to the channel device to prevent reconfiguration attacks. Optionally, the device configuration could be write-protected after device setup. 00SEVen's agent requires shared pages for the channel's virtio rings and for GHCB buffers used to perform hypercalls and MMIO requests (§ 5.2). In-VM attackers might tamper with these pages to change GHCB requests or perform DoS attacks against the remote channel. In addition, they might try to detect if a new analysis is pending by observing virtio ring changes (*E2*). That way, attackers could try to hide attack traces just before an analysis in order to evade it. Optionally, 00SEVen could extend the hypervisor with per-VMPL nested PT views to entirely block VMPL1 access to shared pages, or adopt SEV's vTOM feature [28] at the cost of extra complexity and device I/O overhead. However, note that 00SEVen's channel is TLS-protected and 00SEVen detects misbehaving hypercalls, limiting attacks against shared pages. Furthermore, in-VM attackers cannot trap page accesses by VMPL0 or the hypervisor but rather need to continuously scan all shared pages for changes to time an attack—increasing their risk of detection. Finally, as soon as VMPL1 has been paused for an analysis, VMPL1 can no longer interfere with any buffers (*R2*).

6.1 Beyond 00SEVen: Collusion Attacks

00SEVen's design excludes collusion attacks between the untrusted cloud platform (incl. hypervisor) and in-VM attackers (cf. § 3.1). The biggest risk of collusion attacks are attempts to delay the VMI until all traces of an in-VM attacker have been erased. As the untrusted hypervisor is in control of the vCPU and VMPL scheduling, the hypervisor can delay scheduling of our agent and warn in-VM attackers of a pending VMI request message. That way, the in-VM attackers gain time to finish their attack and delete their attack traces to prevent detection or analysis. While 00SEVen's secure pausing locks the VMPL1 contexts, such that the hypervisor cannot resume their execution until the VMI has finished, there is an exploitable time window between the pausing request and the locking of the VMPL1 contexts (cf. § 4.2.4). The root cause of this is SEV-SNP's reliance on the hypervisor to switch into VMPL0 and yield the VMPL1 contexts before they can be locked. The hypervisor's platform control also makes it hard to prevent all communication between the hypervisor and VMPL1. Even if some direct channels could be blocked, e.g., shared pages, there are several ways to create other (covert) channels not controllable by VMPL0, e.g., based on timed scheduling or VM exit events.

However, the cloud and in-VM attackers must be careful to not risk detection of collusion. For instance, while small analysis delays might be hidden in the network jitter of the remote channel, too many or long delays could be detected by the client. If the hypervisor ignores a secure pausing request, causing the agent to loop in the locking process (§ 4.2.4), the remote client will eventually detect the analysis DoS. Furthermore, hypervisor-VMPL1 interactions might leave new memory traces detectable via VMI. So even if collusion attacks are possible to DoS or delay an analysis, they increase the risk of detection. In addition, memory traps can still prevent malicious VMPL1 read/write accesses to critical regions, e.g., the syscall table. In § 8.1, we suggest SEV changes that further harden 00SEVen against collusion.

7 Evaluation

We now evaluate the analysis performance of our 00SEVen prototype, its effectiveness for detecting or preventing existing rootkits, and its VMPL0 memory and CPU overhead.

00SEVen's open-source prototype (§ 1, footnote 1) consists of our in-VM agent that extends the SVSM, extensions to QEMU/KVM, and extended LibVMI library for analysis clients. It supports remote VMI operations, secure pausing, and page monitoring traps. As SVSM currently supports only Linux VMs, we focus on VMI of Linux—even though conceptually, support for other OSes is possible. The prototype does not yet support function traps and the optional shared buffer isolation.

As the evaluation testbed, we use a Dell PowerEdge R6515 server as our cloud platform running Ubuntu 22.04 with our modified 5.14 kernel on a 2.85 GHz AMD 7443P CPU. The Dell server hosts our 00SEVen prototype including an SEV-SNP Ubuntu VM serving as the VMI target. As the LibVMI remote analysis client, we use a Debian 12 server with a 3.2 GHz AMD 74F3 CPU that shares a LAN with the VMI target. For comparison, we measure three additional setups: LibVMI with KVMi, and 00SEVen with a local (same-host) LibVMI—with TLS and without (TCP-only). Our baseline is LibVMI with the standard KVMi backend (v12) that we measure *locally* (same-host) on the Dell server targeting a non-SEV VM—KVMi lacks remote VMI support. By evaluating 00SEVen with local (TLS / TCP-only) and remote (TLS) clients, we can distinguish sheer network and TLS overhead.

7.1 (Remote) Analysis Performance

We now evaluate the VMI performance of our four setups. To foster a better comparison, we incorporate all compatible

Table 1: VMI policies and their targets, adopted from [44].

P1. process list	P6. process memory map
P2. escalated privileges	P7. keyboard sniffers
P3. VFS hooks	P8. module list
P4. TTY keyloggers	P9. TCP4 "netstat-ops"
P5. syscall table hooks	P10. open files

policies of Liu et al. [44], which resemble techniques used by rootkits. As shown in Table 1, this also includes two policies focused on protecting user input against key loggers (P4,P7), a scenario which is less relevant in a full remote setup but underline 00SEVen's generality. We measured the initialization and analysis time of each setup for 50 iterations per policy, cleaning LibVMI's caches before every run. 00SEVen's local and remote setups showed negligible one-shot initialization times <2 s, while KVMi's default config takes ≤ 10 s. Figure 6 compares the LibVMI analysis times of the four setups. The analysis time captures all VMI queries to KVMi or 00SEVen's agent for the paused target VM. During the analysis, 00SEVen schedules only its agent, avoiding overhead by VMPL switches. The median analysis times are 68–151 ms for KVMi, 65–148 ms for 00SEVen with local TCP-only client, 65–157 ms with local, and 69–204 ms with remote TLS client. That is, compared to KVMi, 00SEVen faces reasonable average median overheads of $+1.91$ % (TCP-only), $+6.85$ % (local TLS), and $+20.0$ % (remote TLS).

00SEVen's relative overhead increases with the number of queries (cf. x-labels, Figure 6). Each VMI read inherently requires a page to be copied out of SEV-protected memory and sent via the VMI channel to the LibVMI client. The effect on the local setups is significantly smaller, showing that the biggest overhead can be attributed to the per-query network overhead. For small query numbers (≤ 34, incl. initialization), 00SEVen was even slightly faster than KVMi. In addition, the local results show that our TLS support, currently missing CPU acceleration and zero-copy, adds noticeable extra overhead. Therefore, optimizing the TLS code and decreasing the number of remote messages by using more caching and offloading strategies (cf. Appendix A) could further improve 00SEVen's performance. For instance, the P5 overhead is small because LibVMI's client-side page cache makes it require only 17 VMI queries to 00SEVen's agent (incl. initialization) for iterating the *hundreds* of co-located syscalls.

7.1.1 Microbenchmark Results

To gain additional insights on the VMI performance, we perform two microbenchmarks. First, we evaluate a single physical page read. We measured the total time on the client-side for 00SEVen's local (loopback) TCP-only and TLS settings, and determined how much each of the agent's sub-operations contributes to the results (using `rdtsc` with fences). On average, a single physical page read took 0.100 ms ($\sim$279.4 k

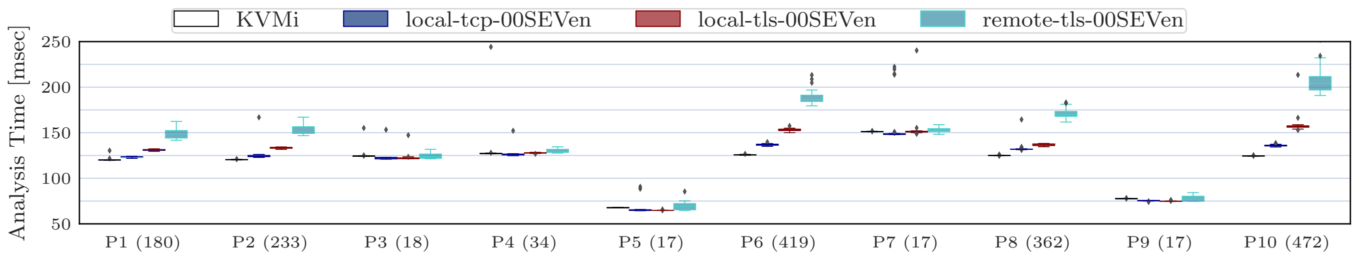

Figure 6: Analysis times for a local KVMi and three 00SEVen setups: local-TCP, local-/remote-TLS (number of VMI queries in brackets at the x-axis labels, notably *including* the pause/resume/initialization requests that are not part of the time measurements).

ticks) with the TCP and 0.160 ms (∼453 k ticks) with the TLS channel, showing the high impact of TLS. The runtime is dominated by message passing and scheduling overhead; Appendix E contains a more detailed breakdown.

Second, we measure the page translation (VM OS virtual address to physical) overhead. The page translation requires a page table (PT) walk, each step of which requires a high-level interaction between the 00SEVen client and agent (plus a CR3 lookup if not already cached). When disabling caching (except for the CR3 register) and TLS, this benchmark took 0.24ms on average on our system with 2MB pages, i.e., with three uncached PT level lookups and CR3 being cached.

7.2 Intrusion and Malware Detection

00SEVen is not limited to inspecting rootkits but can detect any attack leaving traces in memory. For instance, a common trace of an intrusion or data exfiltration attempt are anomalous network connections, e.g., to known malicious command-and-control servers. Therefore, cloud platforms can monitor the network traffic of VMs for suspicious connections and inform a customer on a potential incident. 00SEVen enables customers to securely identify the malicious process to which the suspicious connection belongs, even if the VM is already compromised. To this end, we implemented a policy that iterates the list of open file descriptors of each active process, and looks for IP sockets with the given suspicious source or destination IP to report the corresponding process name. That is, we iterate over all active processes starting from the init task (similar to P1 and Figure 3), and for each process, we iterate over all file descriptors referenced by the task struct (similar to P10), parsing their type-specific information to find sockets that are connected to the suspicious IP in question. 00SEVen's respective median introspection times for a single suspicious connection are 102.2, 116.2, and 122.3 ms (TCP-only, local-TLS, remote-TLS), with 30 VMI queries in total.

On a successful intrusion, attackers often install malware for additional exploitation steps. First, malware then tries to stop security services to prevent detection, e.g., the Linux ClamAV [1] anti-virus scanner daemon. 00SEVen allows to detect such malware intrusions by periodically scanning the kernel's process list to verify if the AV services are still running, e.g., using P1 of § 7.1. If an AV service has unexpectedly stopped, the remote analyst can use 00SEVen to keep the OS securely paused and directly start to further investigate the active attack, e.g., scanning processes for malware signatures.

7.3 Rootkit Detection and Active Trapping

We now evaluate 00SEVen's capability of detecting or preventing rootkits. We adapted four open-source Linux rootkits (three also used in [44]) to our VM's v5 kernel: Sutekh, Spy, Diamorphine, and Randkit [2–4, 6]. *Sutekh* hooks the `umask` and `execve` syscalls by overwriting their function pointers in the syscall table to enable userspace processes to gain root privileges. *Spy* is a keylogger that registers itself as a keyboard listener to log all entered keystrokes. *Diamorphine* hooks the `kill`, `getdent`, and `getdent64` syscalls to enable processes to request service via signals, e.g., hiding of files or threads. *Randkit* hooks the `getrandom` syscall and the read methods of the `u/random` device files to inject an insecure random data generation (all zeros, or xor128), rendering, e.g., all derived crypto seeds and keys vulnerable. For each rootkit, we implemented policies (inspired by Table 1) that detect an infection and evaluated their analysis performance. Our policies check if the syscall table entries point to valid (in-kernel) functions to detect their hooks (*Sutekh, Diamorphine, Randkit*), check for privileged shells (*Sutekh*), and check for registered keyboard notifiers (*Spy*). Sutekh required a median analysis time of 256, 265, and 282 ms (tcp-only, local, remote), Spy of 213, 217, and 217 ms, Diamorphine of 129, 131, and 132 ms, and Randkit of 115, 129, and 131 ms. That is, the analysis times show reasonable performance, allowing to periodically scan the VMs for common rootkit infections.

Instead of detecting rootkits post-mortem, 00SEVen can also *actively prevent* the infection process using page access traps (§ 4.2.5). This is an advantage over approaches like RDMI [44] that have no trap support. We implemented event-based policies for all three rootkits that successfully leverage page write-traps that trigger when the rootkits are trying to tamper with the syscall table or keyboard notifier list. That way, the remote client can directly pause the VM and perform an analysis of the stopped exploitation chain. However, page traps add non-negligible overhead. They should be used preferably to monitor suspicious accesses to critical pages, e.g., write attempts to the syscall table. We measured the trap-and-resume overhead of a single write (`ADD`) showing median overheads of 13.0 µs for KVMi's traps (single stepping-based), 738.5 µs and 761.8 µs for 00SEVen's emulation-based traps in the local setups, and in the remote one ∼1–3 ms if the VM is not yielded, otherwise ∼45 ms (includes network overhead). 00SEVen's overhead is caused by the VMPL switches, TLS and network overhead, and missing single stepping. Suppressing VMPL switches accelerates the local setups to 85.7 µs and 93.3 µs, showing that 00SEVen benefits from reduced switching overhead, e.g., by adding a dedicated agent vCPU or improving AMD SEV (cf. § 8.1). To further decrease 00SEVen's overhead, LibVMI's trap handlers could be partially offloaded into the agent's userspace to avoid communication overhead, and the emulation could be optimized.

7.4 In-VM Requirements and Overhead

00SEVen is designed with a small idle load and TCB (*R7+8*) *without* a full-fledged OS kernel. Our in-VM TCB consists of only ∼13.3 kLOC (≈12.5 k in Rust), including the SVSM (≈6.5), our VMI agent (∼5.1), and virtio drivers (∼1.7). By default, SVSM reserves only 256 MiB of a VM's RAM for VMPL0, i.e., 6.25 % for a 4 GiB VM. Our actual memory requirement is even smaller (in the order of 10 MiB) but depends on the number of vCPUs and the virtio ring size. 00SEVen's agent does *not* impose performance overhead on VM workloads while no analysis is active (*R8*). By default, only the VM OS in VMPL1 is getting scheduled. VMPL0 is scheduled only if either VMPL1 calls into a SVSM memory service—which is typically done only during boot—or if the VM owner sends a remote request to 00SEVen's agent. In the ”idle state”, 00SEVen's VMPL0 components cause no overhead, in contrast to other recent (non-VMI) SEV designs, e.g., Hecate [28], which must actively virtualize scheduling and device I/O for VMPL1 (cf. § 9).

8 Discussion

We now propose SEV extensions further improving VMI and explain 00SEVen's portability to other confidential platforms.

8.1 Improving AMD SEV for Secure VMI

00SEVen would benefit from new optimizations and features for AMD SEV. 00SEVen's agent would benefit from VMPL0 support for directly yielding and locking lower-privileged VMPLs and intercepting VM exit events *without* relying on the hypervisor. That way, 00SEVen's secure pausing feature would not depend on support by the hypervisor, and 00SEVen could directly trap writes to control registers (e.g., to `CR3` to trap page table switches) or use single-stepping for easier page access monitoring (§ 4.2 and Appendix C). 00SEVen would also benefit from VMPL permissions for shared memory pages to prevent in-VM attackers from tampering with its virtio and hypercall buffers *without* relying on the hypervisor (§ 6). Finally, as already observed by Ge et al. [28], VMPL switches through the hypervisor cause non-negligible overhead. Hardware support for directly switching VMPLs without hypervisor intervention would improve the performance of 00SEVen, especially its page/function traps.

8.2 Other Confidential (VM) Platforms

00SEVen's current implementation is tailored to AMD SEV-SNP VMs. However, 00SEVen's concepts generalize to other confidential VM platforms, all of which are incompatible with existing VMI techniques by default (§ 3.3). In the following, we discuss how the concepts of 00SEVen can be implemented in Intel TDX [34] and Arm CCA [16].

Intel TDX Intel's confidential VMs are called trusted domains (TDs) and provide similar protection guarantees as SEV VMs using per-TD crypto keys. Intel has announced support for TD partitioning in future Intel TDX version 1.5, which enables up to four nested VM environments inside a single TD—comparable to VMPLs. TD partitioning provides the foundation for the isolation of 00SEVen's in-VM agent and the deprivileging of the untrusted VM OS. Intel's TDs also have a state-save area comparable to AMD's VMSAs, which are a key component for register introspection.

Arm CCA Arm CCA introduces realms that provide secure execution environments, e.g., for confidential VMs. In contrast to SEV and TDX, the isolation between realms is not based on VM-unique crypto keys but on nested PTs (stage 2 PTs). These NPTs are managed by the new trusted realm management monitor (RMM), which acts as intermediate between the realms and untrusted host hypervisor. The RMM executes with hypervisor-like privileges (EL2), sharing some similarities with the TDX monitor. As the RMM manages NPTs for all realms, it should be possible to redesign existing NPT-based isolation concepts to protect an in-VM agent or create a co-located VMI VM [55, 62]. That way, a special per-realm domain for integrating concepts of 00SEVen's agent could be created. Some of 00SEVen's monitoring approaches could be adapted to benefit from RMM's trusted NPT management.

9 Related Work

Memory Forensic, VMI, and Kernel Monitors Most related to 00SEVen is work on non-confidential VMI and memory forensic, e.g., out-of-VM [42, 62] and in-VM [55] VMI. In Table 2, we provide a comparison of 00SEVen with existing VMI approaches. 00SEVen takes inspiration from these designs to fill the gap of enabling VMI techniques securely

Table 2: Comparison of VMI designs for non-confidential VMs with 00SEVen. (HV ≙ hypervisor, NPT ≙ nested PT)

	Agent			conf.
Name	Place	Isolation	TCB	VMs
LibVMI	HV	virt./HV	HV	✗
SIM [55]	in-VM	NPTs	in-VM/HV	✗
ImEE [62]	2nd VM	NPTs	co-VM/HV	✗
00SEVen	in-VM	VMPLs	VMPL0/SEV	✔

for SEV-SNP VMs. To the best of our knowledge, 00SEVen is the first solution enabling secure remote VMI for confidential VMs. LibVMI [42] is a common framework for non-confidential VMI providing memory and register access, address and symbol translation, as well as event-based traps. We designed 00SEVen to provide similar features and based our remote client on it to enable reuse of existing analysis scripts and tools (cf. § 4.2.2). Zhao et. al [62] use the hypervisor to provide fast out-of-VM VMI using a special co-located VM (ImEE) that shares the untrusted page tables of the target VM securely using NPT permissions. SIM [55] provides an in-VM VMI agent for non-confidential VMs that is protected by the hypervisor and uses special call gates to switch into the agent without a VM exit. In contrast, 00SEVen focuses on remote VMI for SEV-SNP VMs, uses VMPLs to protect its in-VM agent, and designs VMI techniques *not* trusting the hypervisor. Bridging the semantic gap [35] is a fundamental issue of VMI, rendering all these techniques relevant to 00SEVen. Katana and LogicMem [24, 52] enable automatic symbol and data structure extraction, which is an orthogonal feature useful for 00SEVen. Similarly, Oliveri et. al [50] proposed OS-agnostic memory forensic, not requiring prior knowledge on the target, i.e., the VM OS. VMIfresh [21] improves LibVMI's caching (Appendix A) using active monitoring of PT changes to prevent stale entries. 00SEVen could adopt this approach using its r/w-page traps.

Other related work includes remote memory aggregation and forensics for trusted execution environments (TEEs). PCIleech [25] and RDMI [44] enable memory access via devices with (remote) direct memory access. While they provide fast access, they are vulnerable to redirection attacks by a malicious OS or hypervisor [17, 36] and are not tailored to VMI, lacking respective features, e.g, VM pausing and event traps. While there is no work on VMI for SEV VMs, Smile [63] provides secure live memory inspection for Intel SGX enclaves. Similar to 00SEVen, Smile had to securely overcome the hardware-based memory protection of enclaves. In contrast to 00SEVen, Smile relies on a semi-trusted out-of-enclave agent in the system management mode and faces different design challenges. Furthermore, 00SEVen provides additional features, e.g., secure in-VM pausing and event traps. Guerra et. al [33] add VMI modules into Arm TrustZone to inspect the non-secure system but not the Arm TEE itself.

Further related work includes kernel security monitors. Nested Kernel [22] redesigns FreeBSD to provide an isolated in-kernel monitor which deprivileges the kernel by interposing on all page table changes (e.g., via x86-64's write-protect bit) and enforcing kernel code integrity and data protection. In-VM OSes might adopt the monitor for extra security while 00SEVen watches the monitor and other OS resources. Future work could explore if parts of 00SEVen's in-VMI agent could be offloaded into an in-kernel monitor. SVA [20] provides compiler-based VMs that instrument the VM OS to intercept its operations outside the VM and enforce security policies, e.g., memory safety. Future work could explore compiler-based methods inside confidential VMs to assist 00SEVen.

SEV Research There is several orthogonal research on confidential SEV VMs. Most related is Hecate [28], which supports legacy OSes inside SEV-SNP VMs by tailoring a single-VM capable nested hypervisor to SEV-SNP. In addition, Hecate drafts support for in-VM kernel code integrity and network filter policy enforcement. Similar to Hecate, 00SEVen redesigns non-confidential hypervisor techniques securely for SEV-SNP VMs. However, 00SEVen's focus is on secure remote VMI and a small TCB with negligible runtime overhead while no analysis is active. Narayanan et. al [48] integrate a virtual TPM as a new orthogonal SVSM service into VMPL0, which could augment the attestation of 00SEVen's remote channel. Veil [7] is parallel work that provides an alternative VMPL0 service framework, similar to SVSM. Veil provides flexible VMPL management and a set of secure services, e.g., an append-only audit log writable by the VMPL1 OS. However, Veil does not support VMI functionalities like 00SEVen. A future prototype of 00SEVen could replace AMD's SVSM with Veil, and integrate 00SEVen's VMI agent and VSOCK channel. Offensive work on SEV VMs explores their weaknesses (e.g., cipher side-channels) and proposes countermeasures orthogonal to 00SEVen [19, 23, 39–41, 57].

10 Conclusion

00SEVen re-enables an essential security technique for confidential SEV VMs: secure remote VMI. 00SEVen introduces new concepts to redesign existing non-confidential VMI techniques for SEV-SNP VMs. By leveraging the recent virtual machine privilege levels of SEV-SNP, 00SEVen realizes an in-VM VMI agent that is hardware-isolated from out-of-VM and in-VM attackers. 00SEVen's agent provides the VM owner with secure remote inspection capabilities of the private VM memory and registers, as well as secure pausing and trapping mechanisms for consistent and event-based analysis. Using 00SEVen, highly sensitive customers, e.g., of the finance and health sector, can securely offload their workloads to the cloud while retaining full introspection access for periodic security scans, incident response, or attack detection and analysis.

References

[1] ClamAV. https://wiki.ubuntuusers.de/ClamAV/ .

[2] Diamorphine. https://github.com/m0nad/Diamorphine .

[3] randkit. https://github.com/vrasneur/randkit .

[4] Sutekh. https://github.com/PinkP4nther/Sutekh .

[5] VirtIO-drivers-rs. https://github.com/rcore-os/virtio-drivers .

[6] Spy, 2021. https://github.com/jarun/spy .

[7] A. Ahmad, B. Ou, C. Liu, X. Zhang, and P. Fonseca. Veil: A Protected Services Framework for Confidential Virtual Machines. ACM ASPLOS '23, 2024.

[8] Amazon Web Services. Amazon EC2 now supports AMD SEV-SNP, 2023. https://aws.amazon.com/about-aws/whats-new/2023/04/amazon-ec2-amd-sev-snp/.

[9] Amazon Web Services. Nitro Enclaves, 2023. https://aws.amazon.com/ec2/nitro/nitro-enclaves/.

[10] AMD Inc. SEV-SNP: Strengthening VM Isolation with Integrity Protection and More. Technical report, January 2020.

[11] AMD Inc. Secure VM Service Module for SEV-SNP Guests. Technical report, August 2022. Revision: 0.50.

[12] AMD Inc. SEV Secure Nested Paging Firmware ABI Specification. Technical report, November 2022.

[13] AMD Inc. AMD64 Architecture Programmer's Manual Volume 2: System Programming. Technical report, January 2023.

[14] AMD Inc. AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions. Technical report, June 2023.

[15] AMD Inc. SEV-ES Guest-Hypervisor Communication Block Standardization. Technical report, January 2023.

[16] ARM Limited. ARM Confidential Compute Architecture, 2023. https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture.

[17] A. Atamli, G. Petracca, and J. Crowcroft. IO-Trust: An out-of-Band Trusted Memory Acquisition for Intrusion Detection and Forensics Investigations in Cloud IOMMU Based Systems. In *ARES*, 2019.

[18] T. Barabosch, N. Bergmann, A. Dombeck, and E. Padilla. Quincy: Detecting Host-Based Code Injection Attacks in Memory Dumps. In *DIMVA*, 2017.

[19] R. Buhren, H.-N. Jacob, T. Krachenfels, and J.-P. Seifert. One Glitch to Rule Them All: Fault Injection Attacks Against AMD's Secure Encrypted Virtualization. In *ACM CCS*, 2021.

[20] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve. Secure virtual architecture: a safe execution environment for commodity operating systems. In *ACM SOSP*, 2007.

[21] T. Dangl, S. Sentanoe, and H. P. Reiser. VMIFresh: Efficient and Fresh Caches for Virtual Machine Introspection. In *ARES*, 2022.

[22] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. Adve. Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation. In *ACM ASPLOS*, 2015.

[23] S. Deng, M. Li, Y. Tang, S. Wang, S. Yan, and Y. Zhang. CipherH: Automated Detection of Ciphertext Side-channel Vulnerabilities in Cryptographic Implementations. In *USENIX Security*, 2023.

[24] F. Franzen, T. Holl, M. Andreas, J. Kirsch, and J. Grossklags. Katana: Robust, Automated, Binary-Only Forensic Analysis of Linux Memory Snapshots. In *RAID*, 2022.

[25] U. Frisk. PCILeech. https://github.com/ufrisk/pcileech.

[26] U. Frisk. The LeechCore Physical Memory Acquisition Library. https://github.com/ufrisk/LeechCore.

[27] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *NDSS*, 2003.

[28] X. Ge, H.-C. Kuo, and W. Cui. Hecate: Lifting and Shifting On-Premises Workloads to an Untrusted Cloud. In *ACM CCS*, 2022.

[29] Google. GRR Rapid Reponse. https://github.com/google/grr.

[30] Google Cloud. Confidential Computing concepts, 2023. https://cloud.google.com/compute/confidential-vm/docs/about-cvm.

[31] A. Gowda, M. Withrow, and H. Bontha. Kata confidential containers with Azure Kubernetes Service, 2023. https://techcommunity.microsoft.com/t5/azure-confidential-computing/aligning-with-kata-confidential-containers-to-achieve-zero-trust/ba-p/3797876 .

[32] Z. Gu, Z. Deng, D. Xu, and X. Jiang. Process Implanting: A New Active Introspection Framework for Virtualization. In *IEEE Symposium on Reliable Distributed Systems*, 2011.

[33] M. Guerra, B. Taubmann, H. P. Reiser, S. Yalew, and M. Correia. Introspection for ARM TrustZone with the ITZ Library. In *IEEE International Conference on Software Quality, Reliability and Security (QRS)*, 2018.

[34] Intel Corporation. Intel Trust Domain Extensions (Intel TDX), 2023. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html.

[35] B. Jain, M. B. Baig, D. Zhang, D. E. Porter, and R. Sion. SoK: Introspections on Trust and the Semantic Gap. In *IEEE S&P*, 2014.

[36] D. Jang, H. Lee, M. Kim, D. Kim, D. Kim, and B. B. Kang. ATRA: Address Translation Redirection Attack against Hardware-Based External Monitors. In *ACM CCS*, 2014.

[37] T. Knauth, M. Steiner, S. Chakrabarti, L. Lei, C. Xing, and M. Vij. Integrating Remote Attestation with Transport Layer Security. *CoRR*, abs/1801.05863, 2018.

[38] M. Lentz, R. Sen, P. Druschel, and B. Bhattacharjee. SeCloak: ARM Trustzone-Based Mobile Peripheral Control. In *ACM MobiSys*, 2018.

[39] 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.

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

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

[42] LibVMI Project. LibVMI: Simplified Virtual Machine Introspection. https://github.com/libvmi/libvmi .

[43] M. H. Ligh, A. Case, J. Levy, and A. Walters. *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory*. Wiley Publishing, 1st edition, 2014.

[44] H. Liu, J. Xing, Y. Huang, D. Zhuo, S. Devadas, and A. Chen. Remote Direct Memory Introspection. In *USENIX Security*, 2023.

[45] M. Morbitzer, M. Huber, and J. Horsch. Extracting Secrets from Encrypted Virtual Machines. In *ACM CODASPY*, 2019.

[46] M. Morbitzer, M. Huber, J. Horsch, and S. Wessel. SEVered: Subverting AMD's Virtual Machine Encryption. In *EuroSec*, 2018.

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

[48] V. Narayanan, C. Carvalho, A. Ruocco, G. Almási, J. Bottomley, M. Ye, T. Feldman-Fitzthum, D. Buono, H. Franke, and A. Burtsev. Remote attestation of SEV-SNP confidential VMs using e-vTPMs, 2023. https://arxiv.org/pdf/2303.16463.pdf.

[49] OASIS Open. Virtual I/O Device (VIRTIO) Version 1.1. OASIS Committee, 2019.

[50] A. Oliveri, M. Dell'Amico, and D. Balzarotti. An OS-agnostic Approach to Memory Forensics. In *NDSS*, 2023.

[51] B. D. Payne. An Introduction to Virtual Machine Introspection Using LibVMI (slides). In *Malware Memory Forensics Workshop (MMF)*, 2014.

[52] Z. Qi, Y. Qu, and H. Yin. LogicMEM: Automatic Profile Generation for Binary-Only Memory Forensics via Logic Inference. In *NDSS*, 2022.

[53] S. Rostedt. ftrace, 2008. https://www.kernel.org/doc/Documentation/trace/ftrace.txt .

[54] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *ACM SOSP*, 2007.

[55] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi. Secure In-VM Monitoring Using Hardware Virtualization. In *ACM CCS*, 2009.

[56] Volatility Foundation. Volatility. https://github.com/volatilityfoundation/volatility/.

[57] J. Wichelmann, A. Pätschke, L. Wilke, and T. Eisenbarth. Cipherfix: Mitigating Ciphertext Side-Channel Attacks in Software. In *USENIX Security*, 2023.

[58] F. Wilhelm. Tracing Privileged Memory Accesses to Discover Software Vulnerabilities. Master thesis, Karlsruhe Institute of Technology, Germany, 2015.

[59] 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.

[60] C. Willems, R. Hund, and T. Holz. CXPInspector: Hypervisor-Based, Hardware-Assisted System Monitoring. Technical report, Ruhr-Universitat Bochum, 2012.

[61] B. Wójcik. Windows Hot Patching Mechanism Explained, 2020. https://dev.to/bartosz/windows-hot-patching-mechanism-explained-2m1f.

[62] S. Zhao, X. Ding, W. Xu, and D. Gu. Seeing Through The Same Lens: Introspecting Guest Address Space At Native Speed. In *USENIX Security*, 2017.

[63] L. Zhou, X. Ding, and F. Zhang. Smile: Secure Memory Introspection for Live Enclave. In *IEEE S&P*, 2022.

[64] Z. Zhou, M. Yu, and V. D. Gligor. Dancing with Giants: Wimpy Kernels for On-Demand Isolated I/O. In *IEEE S&P*, 2014.

A Caching and Offloading Strategies

We can accelerate 00SEVen's memory access operations (§ 4.2.3) by adopting client- or agent-side optimizations.

Client-side Caching As 00SEVen's current remote client is based on an extended version of LibVMI [42] (cf. § 4.2.2), we can benefit from LibVMI's performance-enhancing client-side caching [51]. We adopt LibVMI's page-level data caching to avoid additional network overhead for follow-up accesses to the same physical VM page, e.g., when reading multiple fields of a structure or page table entries [42, 51]. That is, the client requests whole page reads (4 KiB) of the agent and caches $\{PA_{vm} \rightarrow \text{page buffer}\}$ mappings, where PA_{vm} is the physical page address, to enable client-local follow-up accesses to the same memory page. That way, we avoid extra communication overhead. Furthermore, we accelerate the address translation process ($VA_{vmpl1} \rightarrow PA_{vm}$) required when accessing virtual addresses of the VMPL1 VM OS (cf. § 4.2.3). LibVMI supports multiple related client-side caches, which maintain $\{\text{kernel symbol} \rightarrow PA_{vm}\}$[2], $\{VA_{vmpl1} \rightarrow PA_{vm}\}$, and $\{\text{process ID} \rightarrow PA_{vm}(\text{root PT})\}$ mapping entries that decrease the number of required VMPL1 page table walks and remote memory access requests.

Address Translation Offloading to Agent Beside client-side caching, future extensions of 00SEVen's in-VM agent could accelerate virtual address access. We could offload the VMPL1 page table walk to the agent to decrease the communication overhead between agent and remote client by avoiding VMPL1 page table (PT) transfers. Furthermore, we could then adjust LibVMI's page-level data caching from $\{PA_{vm} \rightarrow \text{page buffer}\}$ to $\{VA_{vmpl1} \rightarrow \text{page buffer}\}$, such that the remote client does not need to calculate and send physical addresses (PA_{vm}) to the agent anymore.

Alternatively, an extension could explore how to eliminate the translation steps from VA_{vmpl1} to VA_{vmpl0}. By securely using parts of the untrusted VM OS (VMPL1) PTs directly from within our agent, we could omit the additional translation ($VA_{vmpl1} \rightarrow PA_{vm}$) and mapping ($PA_{vm} \rightarrow VA_{vmpl0}$) steps. However, existing VMI techniques following such an approach [55, 62] are not applicable to 00SEVen. In fact, they rely on a *trusted* hypervisor managing nested PT entries, permissions, or trapping PT changes of VMPL1. Otherwise, attackers might manipulate or relocate PTs, e.g., to hide attack traces or corrupt the VMI agent. Trusting the hypervisor violates the threat model of SEV-SNP and 00SEVen, rendering these designs insecure for confidential VMI. Moreover, SEV-SNP currently lacks support for VMPL0-controlled nested PTs and intercepting VMPL1's PT switches (cf. § 8.1). We therefore leave a design open for future work.

[2]actually $\{\text{ksym} \rightarrow VA_{vmpl1}\}$, but PA_{vm} is statically derived for Linux [43]

B Security Details on Page R/W-Monitoring

In § 4.2.5, we described how 00SEVen leverages VMPL permissions to securely trap read and/or write accesses by VMPL1 to private VM pages. That way, the remote analyst can perform event-based VMI triggered by the monitoring traps. Even though the hypervisor is untrusted, 00SEVen enforces a scheduling of its in-VM agent on r/w-traps. Only VMPL0 has the privileges required to resolve the VMPL1 access violation for resuming the trapped instruction. The `RMPADJUST` instruction for modifying VMPL permissions is only usable from within the SEV-SNP VM and can only adjust permissions of less privileged VMPLs. Therefore, neither the hypervisor nor the VM OS running in VMPL1 can tamper with the VMPL1 permissions. Furthermore, only our VMPL0 agent can access the vCPU registers of VMPL1 in the respective VMSA to perform an instruction decoding and emulation. Therefore, the hypervisor must schedule 00SEVen's agent to successfully resume the trapped instruction. If the hypervisor tries to directly re-schedule the VM OS (VMPL1) on the access violation, the nested page fault (NPF) will be re-raised. Attempts to modify the page's SEV-SNP attributes, e.g., making a private page shared or trying to directly modify the VMPL1 permissions, will either result in page data corruption or in the page becoming invalid for the VM, requiring a re-registration ("validation" [10]) only resolvable by VMPL0. The only option left is that if the in-VM attacker and hypervisor collude (excluded from our threat model, cf. § 3.1 and § 6.1), they might skip the failed memory access entirely.

C Kernel Function Traps

In § 4.2.5, we introduced 00SEVen's support for event-based VMI triggers by describing the page read/write-access traps. Conceptually, 00SEVen also supports trapping the execution of VMPL1 kernel code. That way, an analysis can be triggered on execution of a certain VM OS function, e.g., a system call [60]. While we could adapt our page read/write-monitoring idea to trap page execution by marking pages as non-executable in the VMPL1 permissions, this approach would introduce significant emulation complexity. On a read/write-trap, 00SEVen's VMPL0 agent must emulate a single memory-accessing VMPL1 instruction to resume VMPL1. However, on a code execution trap, the agent would need to emulate all instructions located at the monitored memory page, including all kinds of control flow instructions, e.g., function calls. Alternatively, we could fall back on hardware single-stepping, but as discussed in § 4.2.5, we then could not guarantee that our agent can re-enable the trap afterwards, because single-stepping requires the untrusted hypervisor.

Instead, 00SEVen can inject VMPL0 trampolines at the beginning of VMPL1 kernel functions using compiler-assistance, similar to how ftrace is implemented in Linux [53] or hot patching in Windows [61]. The injected code serves as pseudo-breakpoints that, if enabled by our agent, call into VMPL0 for analysis. We inline a loop that performs a hypervisor call (`VMGEXIT` instruction), telling the hypervisor to schedule our VMPL0 agent, and a check of the return register. The trampoline uses the statically-known GHCB MSR interface [15] to pass the request number to the hypervisor. After the analysis, our agent sets the return register inside the VMPL1 VMSA of the trapped vCPU in order to confirm a successful trap handling to the trampoline. Otherwise, the trampoline loop retries the call to prevent the hypervisor from ignoring our scheduling request, similar to how AMD SVSM handles service calls [11]. That way, we can reliably inject code execution traps. While we cannot hide our injected code, we can reliably set the VMPL1 permissions as non-writable to prevent tampering by in-VM attackers, because all code pages are treated as private VM pages in SEV-SNP and are therefore affected by VMPL permissions (cf. § 2.1). The injected code is disabled by default by a prepended jump instruction skipping the call loop. Our agent can replace the jump with a `NOP` in memory when the remote analyst requests to enable a function trap. However, note that the context switches between VMPL1 and VMPL0 through the hypervisor cause non-negligible overhead, limiting frequent execution tracing.

D Interrupts on VMPL Scheduling

In § 5.2, we explained that 00SEVen's remote channel device schedules the VMPL0-agent on new request messages. Similarly, 00SEVen schedules VMPL0 when forwarding nested page faults to the agent on read/write-monitoring traps (§ 4.2.5). However, one additional concern must be addressed: When scheduling VMPL0, there must be no event injections by the hypervisor into the VM, e.g., device interrupt requests (IRQ) or non-maskable interrupts (NMI). Otherwise, as 00SEVen's VMPL0 executes in SEV-SNP's restricted injection mode to be protected against malicious event injections by the untrusted hypervisor [10], an event injection attempt would result in a hardware error raised by the CPU when trying to resume VMPL0 execution. Therefore, we implement the following additional steps in QEMU/KVM: On a VMPL0 scheduling request, we temporarily disable NMIs for the vCPU in the hypervisor. Furthermore, we check for a pending IRQ injection for the VM OS by other devices. If so, we stash the pending (VMPL1) IRQ in a new field inside KVM's vCPU data structure. On the back-schedule to VMPL1, KVM then restores the stashed IRQ to deliver it to the VM OS and re-enables NMIs for the vCPU. That way, we prevent NMIs (e.g., caused by Linux's CPU stall detection) and pending (device) IRQs from causing errors or getting dropped (potentially causing a vCPU hang). Note that VMPL0 currently executes with IRQs turned off, i.e., there will be no new IRQ injection attempts during VMPL0 execution beyond the stashed one.

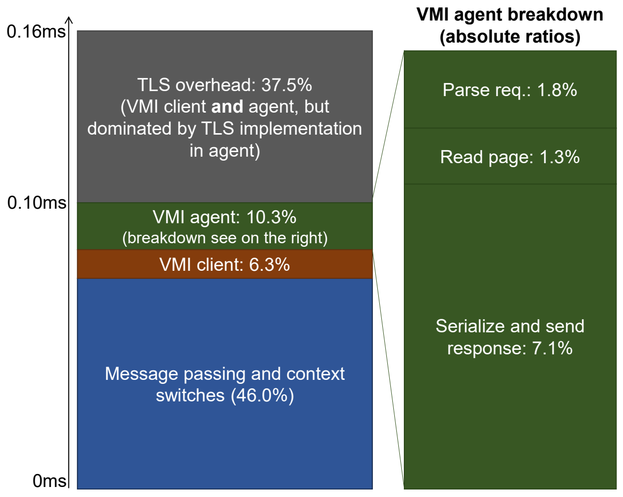

Figure 7: Breakdown of page access microbenchmark costs.

E Microbenchmark Breakdown

Figure 7 provides a detailed breakdown of our page read microbenchmark (§ 7.1.1). The TLS overhead contributes to 37.5 % of the overall runtime. Due to its slow TLS implementation, the agent overall consumes 41 % of the overall overhead, with the most dominant sub-operations being the encrypt-send (86.3 % of the agent), unpack-decrypt (5.3 %), and packet de-/serialize (3.2 %/2.1 %) operations, rather than the actual page access. Ignoring the TLS parts, the agent contributes only 10.3 %, out of which most cycles are spent on parsing the VMI request and preparing the VMI response; the actual page read takes 1.3 % on average. The client contributes 6.3 %, mainly for issuing VMI requests and parsing responses. The remaining parts of the total overhead (46.0 %) are caused by the message channel through the VSOCK device, QEMU process, and `socat` packet forwarding service and the corresponding context switches. This reveals an optimization potential, e.g., by replacing `socat` with a QEMU-integrated proxy service to reduce process context switches.