Secure Monitoring of Confidential VMs with Isolated Agents

Tomoharu Nono
Kyushu Institute of Technology
Iizuka, Fukuoka, Japan
nono@ksl.ci.kyutech.ac.jp

Kenichi Kourai
Kyushu Institute of Technology
Iizuka, Fukuoka, Japan
kourai@csn.kyutech.ac.jp

Abstract

To prevent insiders from eavesdropping on sensitive information in virtual machines (VMs), recent clouds provide confidential VMs, whose memory is transparently encrypted. Since even confidential VMs cannot protect data from intruders inside them, it is still necessary to use intrusion detection systems (IDS). IDS offloading is used to run host-based IDS outside VMs and prevent IDS from being disabled by intruders. However, offloaded IDS cannot monitor information in the memory of confidential VMs due to memory encryption. This paper proposes SEVmonitor for enabling IDS offloading by running agents inside confidential VMs. Offloaded IDS running in another confidential VM securely obtains memory data from the agent in the target VM. To enhance the security of the agent, SEVmonitor confines a target system in an isolated execution environment created in the target VM and runs the agent outside it. It supports two types of isolated execution environments, a container and an inner VM, to take various tradeoffs. We have implemented SEVmonitor using KVM, Linux, BitVisor, and Xen, and examined monitoring and system performance.

CCS Concepts

• **Security and privacy → Virtualization and security**.

Keywords

virtual machine, trusted execution environment, AMD SEV, nested virtualization, VM introspection, intrusion detection system, agent

ACM Reference Format:
Tomoharu Nono and Kenichi Kourai. 2025. Secure Monitoring of Confidential VMs with Isolated Agents. In *2025 IEEE/ACM 18th International Conference on Utility and Cloud Computing (UCC '25), December 1–4, 2025, Nantes, France.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3773274.3774273

1 Introduction

Infrastructure-as-a-Service (IaaS) clouds are widely used and provide virtual machines (VMs) to users. As users deal with sensitive information in VMs, eavesdropping on VMs by cloud insiders becomes a significant risk. To protect VMs from insiders, recent IaaS clouds provide confidential VMs, e.g., in Amazon Web Services [2], Google Cloud [16], and Microsoft Azure [28]. The memory of confidential VMs is transparently encrypted by trusted execution environments (TEEs), e.g., AMD SEV [1] and Intel TDX [20], provided by processors. It is decrypted only when VMs access memory. Using confidential VMs, even hypervisors that manage VMs cannot access data in the memory of the VMs.

Due to such characteristics of confidential VMs, protection by memory encryption is not effective if attackers intrude into VMs. To prevent intruders from eavesdropping on sensitive information inside VMs, it is still necessary to monitor the systems in the VMs using intrusion detection systems (IDS). A technique called IDS offloading with VM introspection (VMI) [12] is used so that IDS is not disabled by intruders. This technique runs IDS outside VMs and monitors the systems in the VMs by analyzing the memory of the VMs. However, IDS offloading cannot be used for confidential VMs because offloaded IDS cannot access the encrypted memory of the VMs. If the systems in the VMs naively expose information to offloaded IDS, sensitive information could leak to cloud insiders via compromised IDS.

This paper proposes *SEVmonitor* to enable secure IDS offloading for confidential VMs. SEVmonitor runs an agent inside a user's confidential VM. The user's offloaded IDS obtains the memory data of the VM from the agent and monitors the system in the VM by analyzing the obtained data. In addition, SEVmonitor runs the user's IDS in another confidential VM and prevents information leakage via the IDS. The IDS and the agent communicate with each other using a virtual network or shared memory established between the VMs. To protect this communication channel from cloud insiders, SEVmonitor encrypts communication data independently of confidential VMs because the memory of each confidential VM is encrypted by a different key.

To enhance the security of the agent, SEVmonitor confines the target system to an isolated execution environment created inside the target VM. Then, it runs the agent outside that execution environment. Since isolated execution environments have tradeoffs in terms of security, performance, and functionality, SEVmonitor supports two execution environments: a container and an inner VM. When it confines the target system to a container, it locates the agent in the kernel of the guest operating system (OS) running in the target VM. This method almost does not affect system performance, while the isolation by a container is weaker than that by a VM. When using an inner VM created inside a confidential VM [13, 32, 40], SEVmonitor locates the agent in the guest hypervisor running in the target VM. This method can isolate the agent more strongly, while system performance degrades.

To reduce the overhead of nested virtualization [7] used for creating an inner VM, SEVmonitor uses BitVisor [39] and Xen [6]. BitVisor is specialized to run a single VM and virtualizes only the minimum devices. Xen's Domain 0 uses lightweight para-virtualization

UCC '25, Nantes, France

ACM ISBN 979-8-4007-2285-1/2025/12
https://doi.org/10.1145/3773274.3774273

and directly accesses devices in the passthrough mode. Users can select one of the two to take tradeoffs between monitoring and system performance. We have implemented SEVmonitor using confidential VMs enabled by AMD SEV, which run on KVM [22]. Our experiments showed that SEVmonitor enabled offloaded IDS to obtain information necessary for the proc filesystem used in Linux. We also clarified various tradeoffs in terms of performance.

The organization of this paper is as follows. Section 2 describes the issues of monitoring confidential VMs. Section 3 proposes SEVmonitor to enable IDS offloading with secure agents inside confidential VMs, and Section 4 explains its implementation. Section 5 analyzes the security of SEVmonitor, and Section 6 shows experimental results. Section 7 describes related work, and Section 8 concludes this paper.

2 Monitoring Confidential VMs

It is reported that insiders such as malicious administrators might exist inside clouds [9, 34, 41]. If insiders eavesdrop on the memory of VMs, sensitive information such as customer data could leak from VMs. To address this issue, recent IaaS clouds provide confidential VMs [2, 16, 28], which are protected by TEEs, to users. As an example of TEEs, AMD EPYC processors provide Secure Encrypted Virtualization (SEV) [1]. SEV encrypts the memory of VMs transparently. It uses a different encryption key for each VM in the encryption engine embedded into the memory controller. Since the encryption keys are securely managed by the AMD secure processor, even insiders who have privileges for managing VMs cannot eavesdrop on the memory of VMs.

Even though the memory of confidential VMs is encrypted, it is not protected if attackers intrude into the VMs. This is because the memory encryption is effective only against attacks from the outside of the VMs. Since the memory is transparently decrypted for access inside the VMs, intruders could easily eavesdrop on sensitive information in memory. From the perspective of processors, they cannot be distinguished from legitimate users in VMs. Even if they cannot directly access the memory containing sensitive information inside the VMs, they could steal sensitive information by exploiting the vulnerabilities of software such as the guest OS or running malware.

Therefore, it is still necessary to detect attacks inside VMs using IDS. In particular, host-based IDS monitors the system states in VMs and finds the symptoms of intrusion. Since this type of IDS inherently runs inside VMs, it could be disabled by intruders. To protect IDS from intruders, a technique called IDS offloading with VMI [12] has been proposed. This technique securely runs IDS outside VMs and enables it to monitor the systems inside VMs. Offloaded IDS detects attacks by analyzing OS data obtained from the memory of VMs. As such, intruders cannot disable offloaded IDS, while offloaded IDS can detect intruders.

However, there are two issues with applying IDS offloading to confidential VMs. First, offloaded IDS cannot analyze OS data in the memory of VMs because the memory is encrypted. From the perspective of processors, offloaded IDS cannot be distinguished from cloud insiders. Therefore, IDS has no means of decrypting the memory of VMs. Second, sensitive information could leak via offloaded IDS if the systems in the VMs naively expose information to IDS. To monitor the system states, IDS obtains OS data, which could contain sensitive information, from VMs. If cloud insiders succeed in compromising IDS, they could obtain the sensitive information that is kept inside IDS.

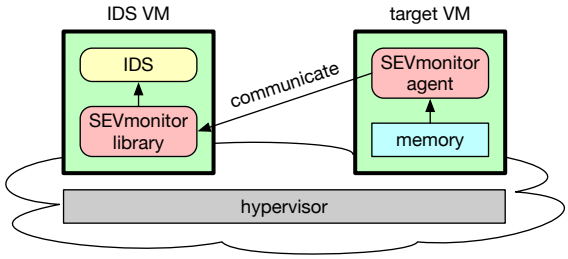


Figure 1: The system architecture of SEVmonitor.

3 SEVmonitor

This paper proposes SEVmonitor to enable secure IDS offloading for confidential VMs.

3.1 System Architecture

Fig. 1 illustrates the system architecture of SEVmonitor. SEVmonitor securely runs its agent inside a user's confidential VM. This *SEVmonitor agent* is a tiny software installed in the target VM. It is used to obtain memory data on behalf of offloaded IDS because the agent running inside the confidential VM can access the memory of the VM. Offloaded IDS can obtain memory data via the agent even from a confidential VM. Then, it can analyze the obtained memory data and monitor the OS data contained in the memory data. In addition, SEVmonitor securely offloads the user's IDS to another confidential VM called an *IDS VM*. Since the memory of the IDS VM is encrypted by a processor-based TEE, even cloud insiders cannot eavesdrop on sensitive information that the IDS obtains from the target VM.

The threat model of SEVmonitor is as follows. SEVmonitor trusts processors providing TEEs and assumes that they are free of vulnerabilities. It trusts the agents running in the target confidential VMs. It also trusts the systems running in the target confidential VMs at boot time, e.g., by using remote attestation. However, we assume that the systems in the target VMs have vulnerabilities and could be compromised by external attackers or cloud insiders. The agents running in the target VMs could be attacked by intruders as well, but SEVmonitor isolates them, as mentioned in Section 3.2. SEVmonitor also trusts the IDS VMs and assumes that the systems in the IDS VMs are not compromised. Since the IDS VM runs only IDS, it is easier to protect the IDS VMs than the target VMs. SEVmonitor does not assume that denial-of-service (DoS) attacks against IDS because TEEs basically cannot protect VMs from DoS attacks.

To enable IDS to obtain memory data transparently via the agent, SEVmonitor provides the *SEVmonitor library* to IDS running in the IDS VM. IDS communicates with the agent running in the target VM via this library. This communication is performed using a virtual network or shared memory established between the IDS and target VMs. There are tradeoffs between these two communication methods in terms of security and performance. The virtual network

Table 1: The tradeoffs between two isolation methods.

	container	inner VM
agent security		✓
agent performance	✓	
system performance	✓	
system functionality		✓

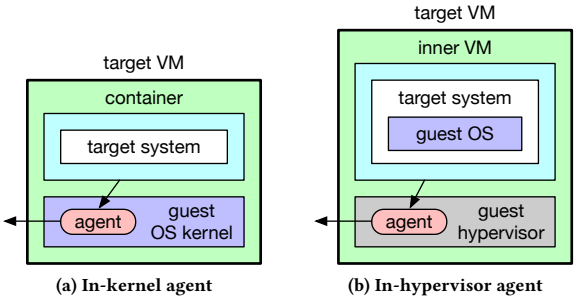
is more secure because access to its packets is restricted more rigidly than data in shared memory. In contrast, the communication overhead is smaller in shared memory. When IDS requires memory data of the target VM, it invokes the library and sends the address of the memory data to the agent. The agent obtains the memory data corresponding to the received address and returns it to the library. The library caches the received memory data and returns the requested data to the IDS.

SEVmonitor protects the address and the memory data between the IDS and target VMs by its own encryption and integrity checking. Since the virtual network and shared memory are managed by the hypervisor, cloud insiders could eavesdrop on and tamper with data in the virtual network and shared memory. The cryptographic key used by SEVmonitor is stored only in the two VMs. To securely share the key, the agent generates a new key, encrypts it using the public key of the IDS, and sends it to the IDS. Even cloud insiders cannot obtain the key or decrypt data in the virtual network or shared memory. Since attackers who do not have the generated key cannot send requests to the agent, the memory data in the confidential VMs is still protected against external attackers. Note that the shared memory cannot be encrypted by a TEE. Since the memory of each confidential VM is encrypted by a different key, it cannot be shared if it is encrypted by either of the two keys used for the IDS and target VMs.

3.2 Isolation of SEVmonitor Agents

To protect the agent running in the target VM, SEVmonitor confines the target system in an isolated execution environment created in the target VM and securely locates the agent outside it. This is mandatory to prevent the agent from being disabled by intruders in the target VM and continue to run inside the target VM after intrusion. There are various tradeoffs between isolated execution environments, e.g., security, performance, and functionality, as shown in Table 1. Currently, SEVmonitor supports two types of isolation methods.

One method is to confine the target system in a container created by the guest OS inside the target VM, as illustrated in Fig. 2(a). In this method, SEVmonitor locates the agent in the guest OS kernel, which is outside the container. The container prevents intruders from attacking the outside. Even if intruders could escape the container, the kernel still protects the agent from the intruders. This isolation method almost does not degrade the performance of the target system because a container is a lightweight virtualization provided by the guest OS. In addition, the in-kernel agent can obtain the memory data of the target system efficiently because it can access the requested memory of the guest OS directly without address translation. However, isolation by a container is weaker than that achieved by a VM. This method assumes that the guest



(a) In-kernel agent (b) In-hypervisor agent

Figure 2: Two types of isolation methods.

OS is trusted, but the in-kernel agent could be disabled if the guest OS is compromised by intruders. Since the container is usually not given high privileges, the functionality of the target system in the container is limited. This limitation could be relaxed by using a container with high privileges, but that is more dangerous.

The other method is to confine the target system in an inner VM created inside the target VM, as illustrated in Fig. 2(b). Such nested virtualization for confidential VMs is achieved in Hecate [13], OpenHCL [32], and Nested SEV [40]. In this method, SEVmonitor locates the agent in the guest hypervisor, which is outside the inner VM. Since a VM provides stronger isolation than a container, the inner VM can protect the in-hypervisor agent from intruders more securely. In addition, the target system can freely customize the guest OS in the inner VM, so that the functionality of the target system is not limited by the isolation. However, the overhead of nested virtualization used for creating the inner VM is large. In particular, the I/O performance of the target system degrades. Since the guest hypervisor is isolated from the inner VM, the in-hypervisor agent needs address translation and indirect access to the memory of the target system. This can lead to low monitoring performance.

Due to these agent locations, SEVmonitor does not run IDS itself inside the target VM. It is not desirable to run IDS inside the guest OS kernel or the guest hypervisor because vulnerabilities in IDS lead to the compromise of the entire system of the target VM. In addition, it is difficult to run complex IDS *inside* the kernel or the hypervisor because they provide special execution environments, which are different from those provided to regular IDS. SEVmonitor can run IDS *on top of* the guest OS in the IDS VM.

4 Implementation

We have implemented SEVmonitor using confidential VMs with AMD SEV [1] on top of KVM [22]. The in-kernel agent supports Linux 5.4, and the in-hypervisor agent supports BitVisor [39] and Xen 4.16 [6].

4.1 In-kernel Agent

The agent runs in the guest OS kernel when SEVmonitor confines the target system in a container inside the target VM. We have developed a Linux kernel module that runs the agent with a kernel thread.

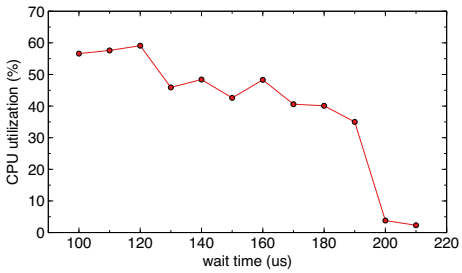


Figure 3: CPU utilization by a busy loop with a wait time.

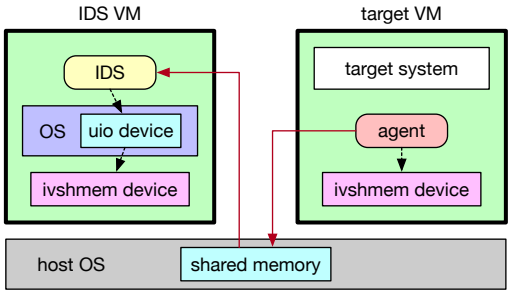


Figure 4: Communication between VMs using shared memory.

4.1.1 Communication with a Virtual Network. The agent communicates with IDS using TCP/IP via the virtual network between the IDS and target VMs. When IDS starts, the SEVmonitor library establishes a TCP connection to the agent. Whenever IDS requires the memory data of the target VM, the library intercepts the access, as described in Section 4.3, and sends its virtual address to the agent. The agent obtains the data of the 4-KB memory page corresponding to the address and returns it to the library. The library stores the received data in the cache and provides it to the IDS. Since the agent can access the entire memory of the target VM with virtual addresses, it is not necessary to translate virtual addresses into physical ones, unlike traditional IDS offloading. The data sent and received by the agent is encrypted with AES.

To enable the agent to receive management commands such as agent termination, the agent uses the non-blocking mode to wait for requests from the SEVmonitor library. It executes the `kernel_accept` function repeatedly until it receives a connection request. After it establishes the connection to the SEVmonitor library, it executes the `kernel_recvmsg` function repeatedly to receive data requests. It also checks the reception of management commands between executions. To prevent the agent from occupying one CPU with a busy loop in the kernel, the agent waits for several hundred microseconds between function invocations. According to our experimental results on CPU utilization in Fig. 3, the wait time is configured to 200 μs.

4.1.2 Communication with Shared Memory. To reduce the overhead of communication with a virtual network, the agent can use shared memory between the VMs for fast communication. Using ivshmem [44] provided by QEMU, SEVmonitor runs the ivshmem-server on the host OS and maps the same part of the host memory onto both the IDS and target VMs, as illustrated in Fig. 4. Since it provides a virtual PCI device to the VMs, they access the shared memory using the ivshmem-uio driver [26].

In the target VM, the guest OS remaps the memory of the PCI device onto the kernel memory address space. As a result, the agent can directly access the shared memory. Upon the memory remapping, the guest OS sets the C-bit of the corresponding page table entries (PTEs) to zero. SEV uses the C-bit to control the memory encryption and encrypts a memory page if the corresponding C-bit is set to one. Therefore, the agent can access the shared memory without encryption by SEV and share it with the IDS VM.

In the IDS VM, on the other hand, SEVmonitor provides the PCI device used for shared memory as a uio device to the IDS process. Uio is the interface for accessing the device memory from the user space in Linux. The SEVmonitor library linked to IDS accesses the shared memory via the uio device. However, it cannot directly access the shared memory because the existing ivshmem-uio driver supports only the `read` and `write` system calls. These system calls need to copy data between a process and the kernel.

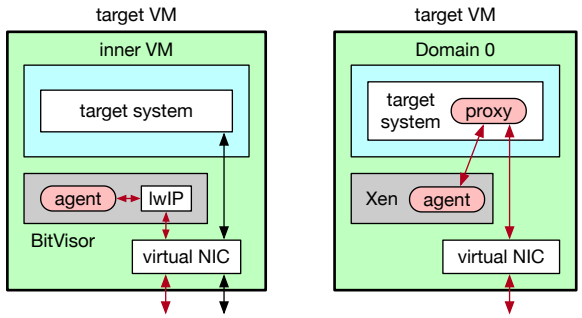
To enable zero-copy access, SEVmonitor provides the modified ivshmem-uio driver that supports the `mmap` system call. The driver enables the SEVmonitor library to map the memory of the uio device onto its memory address space and directly provide the shared memory to IDS. When the library issues the `mmap` system call, the driver remaps the memory of the PCI device onto the process memory address space. However, the Linux kernel does not set the C-bit of the PTEs to zero when remapping device memory onto the process memory address space. Therefore, the modified driver sets the C-bit to zero for the memory regions where the device memory is mapped.

SEVmonitor achieves synchronization between IDS and the agent using polling because the interrupt mechanism provided by ivshmem does not work. The agent waits for the SEVmonitor library to write a request to the shared memory by checking the request using a busy wait with 200-μs sleep. After the library writes a request, it waits for the agent to write a response to the shared memory using a busy wait. The request consists of the virtual address of OS data, while the response consists of memory data. To prevent insiders from eavesdropping on the shared memory that is not encrypted by SEV, the library and the agent encrypt the request and the response by themselves, respectively.

4.2 In-hypervisor Agent

The agent runs in the guest hypervisor when SEVmonitor confines the target system in an inner VM inside the target VM. We have developed the agents for two types of hypervisors.

4.2.1 Agent for BitVisor. To reduce the overhead of nested virtualization, SEVmonitor uses BitVisor in the target VM and runs the target system in the inner VM created by BitVisor. Since BitVisor is optimized to run only one VM, it is more lightweight than general-purpose hypervisors running multiple VMs, e.g., KVM. BitVisor adopts the para-passthrough architecture, which virtualizes I/O only when the hypervisor needs to interpose network and storage



(a) Agent for BitVisor (b) Agent for Xen

Figure 5: Two types of in-hypervisor agents.

access. SEVmonitor can prevent the degradation of I/O performance because it does not need to virtualize any I/O devices.

When the in-hypervisor agent communicates with IDS using a virtual network, it uses lwIP [11], which is a lightweight TCP/IP stack provided in BitVisor. To enable the agent to use the virtual network of the target VM, BitVisor shares the virtual NIC between the guest hypervisor and the inner VM and assigns different IP addresses, as illustrated in Fig. 5(a). To enable this, the target VM uses the virtual e1000e device, which is supported by the BitVisor hypervisor. Since BitVisor does not support network sockets in lwIP, the agent uses its Raw API. Unlike the socket API, when it receives a request from the SEVmonitor library, the registered callback is invoked. Then, it obtains the requested memory data and returns it to the library.

Since the agent cannot directly access the memory of the inner VM using the virtual address received from the library, it translates the guest virtual address used in the inner VM into the host physical address that can be used in the guest hypervisor. First, it finds the PTE corresponding to the guest virtual address using the page tables in the memory of the inner VM and obtains the guest physical address stored in the PTE. Then, it translates the guest physical address into the host physical address using the nested page tables (NPT) for the inner VM, which is stored in the guest hypervisor. Finally, it obtains the memory data of the inner VM using the host physical address.

When the agent communicates with IDS using shared memory, it leverages the shared memory established between the IDS and inner VMs, not between the IDS and target VMs. This is because the device driver for the shared memory is not provided for BitVisor. The ivshmem-uio driver in the inner VM obtains the guest physical address of the shared memory and its size and passes them to the guest hypervisor using a new hypercall. The agent translates the received guest physical address into a host physical one and can access the shared memory. We assume that the target system in the inner VM is not compromised at its boot time. Since the agent in BitVisor cannot wait for requests from the SEVmonitor library using polling, unlike the in-kernel agent, it sets an interval timer and periodically checks requests in the shared memory.

4.2.2 Agent for Xen. As another way of reducing the overhead of nested virtualization, SEVmonitor can use Xen in the target VM and run the target system in a para-virtualized VM. A para-virtualized VM enables the efficient execution of the guest OS by modifying it. In particular, SEVmonitor uses Domain 0 as a para-virtualized VM. Domain 0 is a management VM that is booted together with the guest hypervisor and virtualizes only CPUs and memory. Since it virtualizes no I/O devices, the degradation of I/O performance is suppressed in the inner VM. Although Domain 0 has privileges for managing the entire virtualized system, the running guest hypervisor is protected from Domain 0. SEVmonitor can detect the replacement of the hypervisor binary in Domain 0 using remote attestation, which checks the legitimacy of the guest hypervisor.

Table 2: The tradeoffs between BitVisor and Xen's Domain 0.

	BitVisor	Xen
monitoring performance	✓	
system performance		✓

When the in-hypervisor agent communicates with IDS using a virtual network, it uses a proxy running in Domain 0, as illustrated in Fig. 5(b). This is because the hypervisor in Xen does not provide network functions, unlike BitVisor. When the proxy receives a request from IDS, it invokes the guest hypervisor using a new hypercall to obtain the requested memory data of Domain 0. Note that the proxy does not directly obtain memory data for security, although it can do so. The request is encrypted by the SEVmonitor library and can be decrypted only by the guest hypervisor. Similarly, the obtained memory data is encrypted by the guest hypervisor and can be decrypted only by the SEVmonitor library. Therefore, memory data does not leak to intruders via the proxy in Domain 0.

Table 2 shows the tradeoffs between BitVisor and Xen's Domain 0. Since the in-hypervisor agent in Xen needs the proxy to communicate with IDS, its monitoring performance is worse than that of BitVisor. However, the performance of a para-virtualized Domain 0 is better than that of a fully virtualized VM in BitVisor. In addition, Domain 0 can directly use the virtio-net device provided by the target VM in the passthrough mode because it is unnecessary to support that device in the Xen hypervisor. The virtio-net device is specialized for VMs and achieves better performance than the virtual e1000e device, which is used in BitVisor.

The address translation for a para-virtualized VM is different from that for a fully virtualized one. To obtain the memory data of Domain 0 using the guest virtual address received from IDS, the guest hypervisor translates it into a pseudo physical frame number (PFN) using the page tables in the memory of Domain 0. Then, it translates the PFN into a machine frame number (MFN) using the physical-to-machine (P2M) table. Finally, it maps the memory page corresponding to the MFN and accesses the memory of the inner VM.

When the agent communicates with IDS using shared memory, it leverages the shared memory established between the IDS VM and Domain 0. This mechanism is similar to that for BitVisor. The ivshmem-uio driver in Domain 0 obtains the PFN of the first page of the shared memory and its size and passes them to the guest hypervisor. The in-hypervisor agent translates the PFN into an MFN and maps the corresponding memory page.

4.3 SEVmonitor Library

To make it easier to analyze OS data and monitor the target system, SEVmonitor uses the LLView framework [33]. LLView enables developers to write IDS programs using the source code of the Linux kernel. It compiles the developed IDS programs using LLVM [43] and converts the `load` instructions in the emitted intermediate representation (bitcode). It embeds code for invoking the SEVmonitor library before the `load` instructions into the bitcode. The SEVmonitor library communicates with the agent in the target VM and obtains memory data. The obtained memory data is stored in the cache and read by the `load` instructions of the IDS.

5 Security Analysis

First, let us consider that external attackers or cloud insiders intrude into a target VM. Such intruders cannot access or disable the agent unless they compromise the guest OS for the in-kernel agent or the guest hypervisor for the in-hypervisor agent. SEVmonitor assumes that the guest OS is not compromised when using the in-kernel agent and that the guest hypervisor is not compromised when using the in-hypervisor agent. Also, intruders cannot access or disable IDS running in another confidential VM.

In addition, intruders cannot access communication data between the agent and IDS when SEVmonitor uses a virtual network. The virtual network is accessible only to the guest OS running the in-kernel agent and to the guest hypervisor running the in-hypervisor agent. If Xen is used to run the in-hypervisor agent, intruders could access the data handled by the proxy running on top of the guest OS, but that data is protected by the guest hypervisor. Even when SEVmonitor uses the shared memory, intruders cannot access data in the shared memory if the in-kernel agent is used. In this case, the shared memory is accessible only to the guest OS. If the in-hypervisor agent is used, intruders could access the shared memory because the shared memory is established using the guest OS, which is assumed to be compromised in this case. However, the data is protected by the guest hypervisor.

Second, cloud insiders could directly attack the agent and IDS without intruding into the VMs. However, they cannot access the agent or IDS running in confidential VMs from the outside. They can stop or terminate the target VM with the agent, but this attack is easily detected by the user of the VM because all the services provided by the target VM stop. Cloud insiders can also stop or terminate the IDS VM, but this attack is also easily detected by the user.

6 Experiments

We conducted several experiments to show the effectiveness of SEVmonitor. First, we confirmed that offloaded IDS could obtain various OS data in a VM using SEVmonitor. Then, we examined the time needed to obtain the data. We always enabled SEV for an IDS VM, but we applied SEV to a target VM only when using the in-kernel agent. We have implemented SEV support for an inner VM created by BitVisor and Xen in our previous work [40], but we did not use that support in these experiments because we have not integrated it with SEVmonitor yet.

We used a server with an AMD EPYC 7262 processor, 128 GB of memory, and 10 Gigabit Ethernet. We ran Linux 5.11 as the host OS and QEMU-KVM 6.2.0 as virtualization software. For target and IDS VMs, we assigned two virtual CPUs and 2 GB of memory. In a target VM, we ran a Docker container [10] on top of Linux 5.11 when using the in-kernel agent. For the in-hypervisor agent, we ran Linux 5.4 when using BitVisor and Linux 5.11 when using Xen 4.16. As a remote client, we used a PC with an Intel Core i7-10700 processor, 16 GB of memory, and 10 Gigabit Ethernet and ran Linux 5.13.

Table 3: The nine pseudo files of the proc filesystem generated by the IDS.

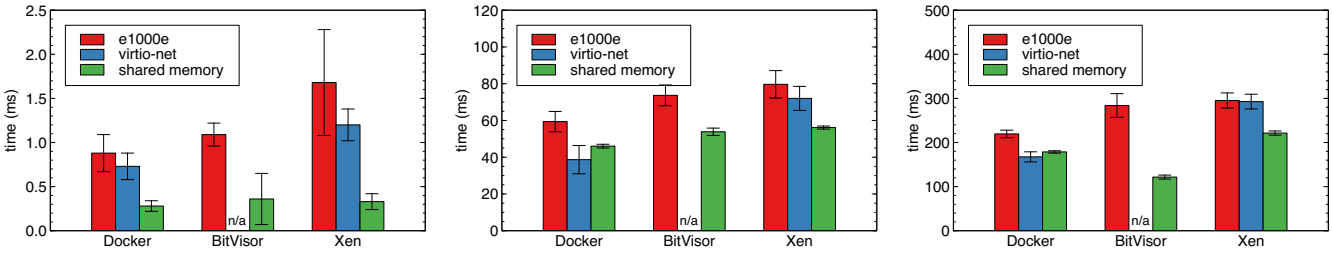
pseudo file	contents
stat	kernel and system statistics
meminfo	statistics about memory usage
uptime	the uptime of the system
tty/drivers	the list of tty drivers
sys/kernel/osrelease	the Linux kernel version
sys/kernel/pid_max	the maximum process ID
[pid]/stat	status information on the process
[pid]/status	human-readable [pid]/stat
[pid]/auxv	information passed to the process

6.1 System Monitoring

First, we have developed an IDS that obtains the version of the guest OS running in the target system. This IDS obtained the string stored in the linux_banner variable in the Linux kernel. When we ran this IDS in the IDS VM, SEVmonitor sent one request to the agent in the target VM and received memory data of 4 KB. As a result, the IDS could obtain the OS version when we located the agent in either the guest OS kernel or the guest hypervisor inside the target VM.

Next, we have developed an IDS that obtains information on the processes running in the target system. This IDS traversed the process list from the init_task variable in the Linux kernel and obtained the IDs and names of the processes. When we confined the target system to a container or an inner VM using BitVisor, SEVmonitor sent 119 requests to the agent and received memory data of 476 KB. As a result, the IDS could obtain information on 119 processes. When we confined the target system to an inner VM using Xen, SEVmonitor sent 127 requests because Domain 0 ran 127 processes including those related to Xen and the proxy used by SEVmonitor.

Finally, we have developed an IDS that obtains the information provided by the proc filesystem in the target system. The proc filesystem is often used to monitor the system states. The IDS generated nine pseudo files of the proc filesystem, whose details are shown in Table 3. When we confined the target system in a container, SEVmonitor sent 717 requests and received memory data of 2.8 MB. We confirmed that the contents of the generated pseudo files were basically the same as those in the target system. Similarly, we confirmed that the IDS worked correctly when we used an inner VM.



(a) OS version (b) Process list (c) Proc filesystem

Figure 6: Monitoring performance.

6.2 Monitoring Performance

We measured the time needed to obtain system information in the target VM via the agent. Fig. 6(a) shows the performance when we obtained the OS version. When we used the virtual network, using virtio-net was faster than using e1000e. The times were 17% and 29% shorter in Docker-based and Xen-based SEVmonitor, respectively. We could not measure the time of using virtio-net in BitVisor-based SEVmonitor because the BitVisor hypervisor does not support virtio-net. Compared with Docker-based SEVmonitor, the Xen-based one was slower because the IDS communicated with the agent via the proxy running as a process. In contrast, using shared memory was much faster than using the virtual network. The monitoring performance was improved by 2.6x, 3.6x, and 3.0x for Docker-based, BitVisor-based, and Xen-based SEVmonitor, respectively. The performance difference was small between the three types of SEVmonitor when using shared memory.

Fig. 6(b) shows the time for obtaining the process list in the target VM via the agent. Like the experiment of the OS version, using virtio-net was faster than using e1000e, but the trend was different. The time was 54% faster in Docker-based SEVmonitor, while that was only 11% faster in the Xen-based one. This is probably because the virtio-net driver in Linux is more efficient than the e1000e driver when many packets are sent and received. Using shared memory was faster in BitVisor-based and Xen-based SEVmonitor, but that was 19% slower than virtio-net in the Docker-based one. One possible reason is that the virtio-net driver in Linux can handle packets more efficiently than communication using shared memory and polling.

Fig. 6(c) shows the time for generating nine pseudo files in the proc filesystem. The trend was similar to that of obtaining the process list. In Xen-based SEVmonitor, the difference between e1000e and virtio-net became smaller and almost the same. Using shared memory in BitVisor-based SEVmonitor became much faster and the fastest among the three types of SEVmonitor. The reason is that the agent in the BitVisor hypervisor uses a timer instead of a busy loop with a wait time. The timer woke up the agent more quickly. Fig. 7 shows the generation time of each pseudo file in Docker-based SEVmonitor. Since the number of processes was 119, it took longer to generate per-process pseudo files in total.

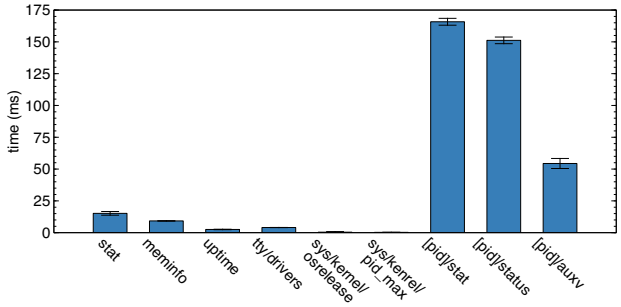


Figure 7: The generation time of each pseudo file.

6.3 System Performance

To examine the overhead of isolating the agent from the target system, we first measured the network performance of the target system. We ran the iperf server in the target system and the iperf client at a remote host. As a baseline, we measured the performance in Linux without the isolation of a container or an inner VM. To examine whether BitVisor and Xen could reduce the overhead of nested virtualization, we also used an inner VM created by KVM inside the target VM. KVM provides VMs whose devices are fully virtualized. We used virtio-net for the inner VM on KVM, while we used e1000e in the target VMs for all the configurations.

Fig. 8 shows the throughput of TCP/IP. When we isolated the target system using a Docker container, the performance degradation was only 3.8%. In contrast, that was 24% when we isolated the target system by an inner VM using KVM. BitVisor and Xen improved the performance by 19% and 16%, respectively. Compared with no isolation, the performance degradation was 8.9% for BitVisor and 11% for Xen.

Next, we measured the performance of the nginx Web server [31] running in the target VM. We sent requests using the ApacheBench benchmark [42] on the same host. Fig. 9 shows the throughput of the Web server. When we isolated the target system in a Docker container, the throughput improved unexpectedly. This is probably due to measurement errors. When we isolated the target system in an inner VM, the throughput degraded due to nested virtualization. When we used a VM created by KVM, the performance degradation was 32%. In contrast, that was only 15% and 19% in BitVisor-based

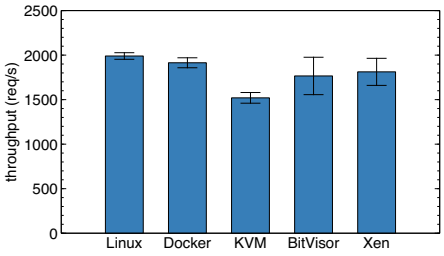


Figure 8: The network performance of the target system.

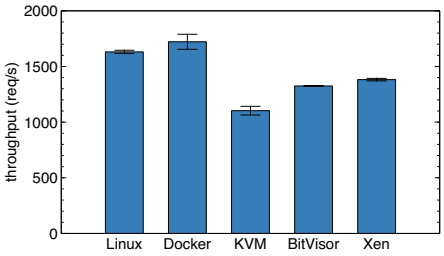
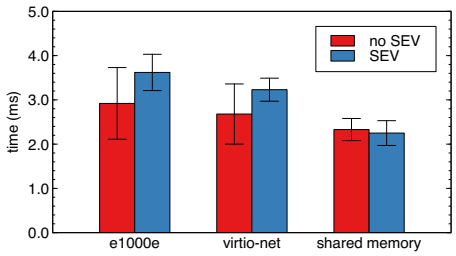
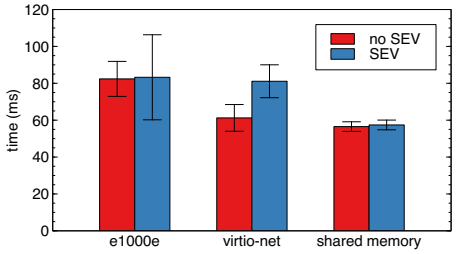


Figure 9: The Web performance in the target system.



(a) OS version



(b) Process list

Figure 10: The overhead of SEV.

and Xen-based SEVmonitor, respectively. This means that a VM created by BitVisor and Domain 0 in Xen could reduce the isolation overhead.

6.4 Overhead of Enabling SEV

We examined the impact of SEV on system monitoring in SEVmonitor. In this experiment, we ran the in-kernel agent and compared monitoring performance for the target VM and the IDS VM with and without SEV enabled. We measured the time needed to obtain information on the OS version and the process list from the target VM. As shown in Fig. 10, the virtual network basically degraded the monitoring performance by SEV. When we obtained the OS version, the overhead was 19% for e1000e and 17% for virtio-net. This was mainly caused by data copies from and to unencrypted DMA bounce buffers, which were accessed by the device emulator to emulate virtual NICs for SEV-enabled VMs. When we obtained the process list, the overhead was 25% for virtio-net but only 1% for e1000e. It is under investigation why the overhead of SEV was so small in this configuration.

On the other hand, the overhead of SEV was negligible when we used shared memory instead of the virtual network to communicate with the agent. The time of obtaining the process list was 1.6% longer, while that of obtaining the OS version was 3.4% shorter. This is because shared memory does not need extra encryption or decryption by enabling SEV.

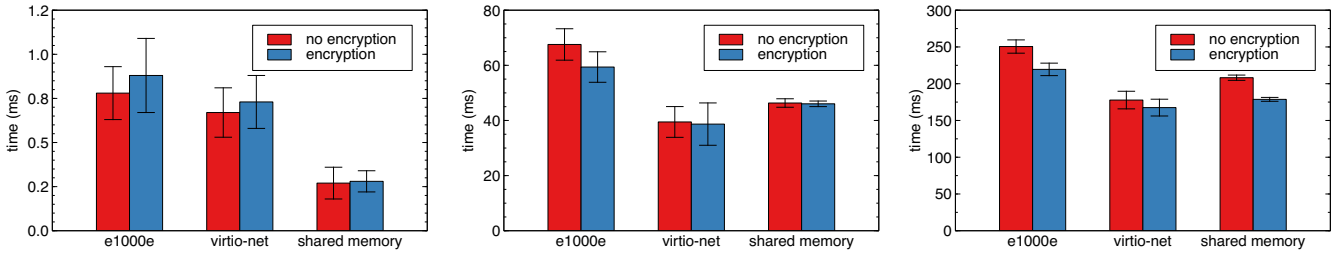
6.5 Overhead of Data Encryption

We examined the overhead of encrypting data between the SEVmonitor library and the agent. We compared the time needed to obtain information with and without the encryption of communication data in Docker-based SEVmonitor. Fig. 11 shows the results. For the OS version, the overheads of data encryption were 13% and 9% when using e1000e and virtio-net, respectively. In contrast, the overhead was only 3.7% for shared memory. This is because the agent needs only data encryption as an extra task when it copies data from the memory of the target system to the shared memory. When using the virtual network, the agent has to copy data with encryption from the memory of the target system to a local buffer as an extra task. For the process list and the proc filesystem, on the other hand, the monitoring time always became shorter due to data encryption. The reason is under investigation.

This trend was a bit different in Xen-based SEVmonitor, as shown in Fig. 12. For the OS version, the overhead of data encryption was 65% when using e1000e. This is probably due to the large variance. In contrast, using virtio-net with data encryption was slightly faster than without data encryption. For the process list and the proc filesystem, the monitoring performance was degraded by data encryption when using e1000e, unlike Docket-based SEVmonitor.

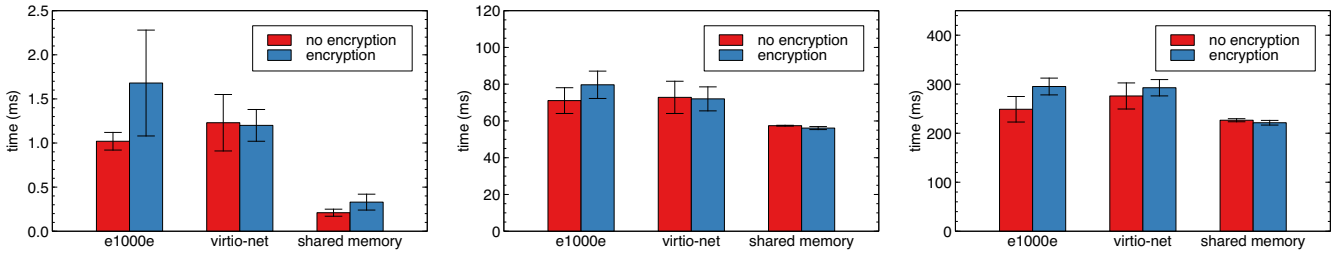
7 Related Work

00SEVen [36] is yet another method to monitor confidential VMs. It securely runs an agent inside a confidential VM using VM privilege levels (VMPLs) provided by AMD SEV-SNP. VMPLs enable a VM to divide its address space into four levels. The agent runs in the highest privilege called VMPL0, while the target system runs in a lower privilege. The agent can directly access the memory of the target system and send the memory data to remote IDS. However, 00SEVen heavily relies on the hypervisor to switch VMPLs and run the agent. In contrast, SEVmonitor can schedule the agent *inside*



(a) OS version (b) Process list (c) Proc filesystem

Figure 11: The overhead of data encryption in Docker-based SEVmonitor.



(a) OS version (b) Process list (c) Proc filesystem

Figure 12: The overhead of data encryption in Xen-based SEVmonitor.

a confidential VM. It is difficult for the hypervisor to stop only the agent without stopping the entire confidential VM. In addition, 00SEVen cannot be applied to confidential VMs with the other TEEs such as Arm CCA [3] because it strongly relies on the mechanism in SEV-SNP.

There are several systems that run an agent in the enclave provided by Intel SGX [27], which is one of the TEEs. The SGX enclave is a protection domain created inside a process and encrypts its memory by processors. For example, an agent is used to save and restore the internal state of an enclave to and from the external memory in VM migration [17] and container migration [30]. It is necessary because the migration mechanism outside the enclave cannot access the memory of the enclave. However, the agent is not protected from the rest of the enclave. Therefore, it cannot be executed securely if an untrusted service runs inside the enclave.

To protect an agent in an enclave, Ryoan [19] constructs a sandbox inside an enclave using Google NaCl [15] and runs an agent outside the sandbox. The agent verifies the code executed in the sandbox and performs runtime checks. Similarly, AccTEE [14] creates a sandbox using WebAssembly [18] and securely runs a service inside the sandbox. The agent consistently records the resource usage by the service in the enclave. However, it is difficult to apply NaCl and WebAssembly to the entire system including the OS in a confidential VM.

SEVmonitor protects IDS using a confidential VM, but the systems protecting IDS with SGX have been proposed. S-NFV [38] locates the internal states of network-based IDS, specifically Snort [8], and the code handling those states in an enclave. IDS securely uses the internal states by invoking the code via the provided API. SEC-IDS [24] runs the entire Snort in an enclave using the Graphene-SGX library OS [45]. It uses DPDK [25] to obtain network packets in the enclave. These systems could be used to securely run IDS instead of the confidential IDS VM in SEVmonitor.

SGmonitor [29] runs host-based IDS in an enclave. The IDS obtains the memory data of the target VM via the hypervisor and analyzes OS data to monitor the target system. SCwatcher [21] enables the existing host-based IDS by providing the standard OS interface with SCONE [4] and Occlum [37]. It also provides the proc filesystem that returns information on the target system in the enclave. These systems need to trust the underlying hypervisor to securely obtain the memory data of the target VM. Instead of the hypervisor provided by clouds, SEVmonitor trusts the agent that securely runs in the target confidential VM.

Before TEEs are provided by processors, the system management mode (SMM) in x86 is used to run IDS securely. HyperGuard [35] checks the integrity of the hypervisor in BIOS. HyperSentry [5] securely inserts an agent in the hypervisor using SMM to monitor the target system. HyperCheck [46] transfers the memory data of the target system to a remote host using SMM and monitors it. SSdetector [23] combines SMM with SGX. It runs IDS in an SGX enclave and securely obtains the memory data of the target system using SMM. However, the execution in SMM is slow and needs to stop the entire system. The code executed in SMM has to be implemented in BIOS.

8 Conclusion

This paper proposed SEVmonitor to enable secure IDS offloading for confidential VMs. SEVmonitor confines the target system in an isolated execution environment, e.g., a container or an inner VM, created in the target VM and securely runs an agent outside the execution environment. IDS securely runs in another confidential VM and communicates with the agent to obtain memory data using the virtual network or shared memory. Our experiments showed that IDS could obtain OS data from a confidential VM efficiently.

One of our future work is to improve monitoring performance. The communication overhead is imposed whenever IDS obtains memory data in the current implementation, but it can be reduced by obtaining necessary data in batches. Another direction is to support other methods for isolating agents. For example, SEVmonitor can locate an agent in BIOS running in a confidential VM. Such an agent is more secure than the in-kernel agent, while system performance is better than using an inner VM.

Acknowledgments

This work was partially supported by JST, CREST Grant Number JPMJCR21M4, Japan.

References

[1] Advanced Micro Devices, Inc. 2020. Secure Encrypted Virtualization API Version 0.24.
[2] Amazon Web Services, Inc. 2024. AMD SEV-SNP for Amazon EC2 Instances. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/sev-snp.html.
[3] Arm Limited. 2023. Arm Confidential Computing Architecture. https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture.
[4] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *Proc. USENIX Symp. Operating Systems Design and Implementation*. 689–703.
[5] A. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. Skalsky. 2010. HyperSentry: Enabling Stealthy In-context Measurement of Hypervisor Integrity. In *Proc. ACM Conf. Computer and Communications Security*. 38–49.
[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. 2003. Xen and the Art of Virtualization. In *Proc. ACM Symp. Operating Systems Principles*. 164–177.
[7] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour. 2010. The Turtles Project: Design and Implementation of Nested Virtualization. In *Proc. 9th USENIX Symp. Operating Systems Design and Implementation*. 423–436.
[8] Cisco Systems. [n. d.]. Snort – Network Intrusion Detection & Prevention System. https://www.snort.org/.
[9] CyberArk Software. 2009. Global IT Security Service.
[10] Docker, Inc. [n. d.]. Docker: Accelerated Container Application Development. https://www.docker.com/.
[11] A. Dunkels. [n. d.]. lwIP – A Lightweight TCP/IP Stack. https://savannah.nongnu.org/projects/lwip/.
[12] T. Garfinkel and M. Rosenblum. 2003. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proc. Network and Distributed Systems Security Symp.* 191–206.
[13] X. Ge, H. Kuo, and W. Cui. 2022. Hecate: Lifting and Shifting On-premises Workloads to an Untrusted Cloud. In *Proc. ACM SIGSAC Conf. Computer and Communications Security*. 1231–1242.
[14] D. Goltzsche, M. Nieke, T. Knauth, and R. Kapitza. 2019. AccTEE: A WebAssembly-based Two-way Sandbox for Trusted Resource Accounting. In *Proc. Int. Middleware Conf.* 123–135.
[15] Google, Inc. 2016. Native Client. https://developer.chrome.com/docs/native-client/.
[16] Google LLC. 2025. Confidential VM Documentation. https://cloud.google.com/confidential-computing/confidential-vm/docs.
[17] J. Gu, Z. Hua, Y. Xia, H. Chen, B. Zang, H. Guan, and J. Li. 2017. Secure Live Migration of SGX Enclaves on Untrusted Cloud. In *Proc. Int. Conf. on Dependable Systems and Networks*. 225–236.
[18] A. Haas, A. Rossberg, D. Schuff, B. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. 2017. Bringing the Web Up to Speed with WebAssembly. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*. 185–200.
[19] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel. 2016. A Distributed Sandbox for Untrusted Computation on Secret Data. In *Proc. USENIX Symp. Operating Systems Design and Implementation*. 533–549.
[20] Intel Corporation. 2023. *Intel Trust Domain Extension*. Technical Report. Intel Corporation.
[21] T. Kawamura and K. Kourai. 2022. Secure Offloading of User-level IDS with VM-compatible OS Emulation Layers for Intel SGX. In *Proc. IEEE Int. Conf. Cloud Computing*. 157–166.
[22] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. 2007. kvm: the Linux Virtual Machine Monitor. In *Proc. Ottawa Linux Symp.* 225–230.
[23] Y. Koga and K. Kourai. 2023. SSdetector: Secure and Manageable Host-based IDS with SGX and SMM. In *Proc. Int. Conf. Trust, Security and Privacy in Computing and Communications*. 539–548.
[24] D. Kuvaiskii, S. Chakrabarti, and M. Vij. 2018. Snort Intrusion Detection System with Intel Software Guard Extension (Intel SGX). In *arXiv:1802.00508*.
[25] Linux Foundation. [n. d.]. Data Plane Development Kit (DPDK). https://www.dpdk.org/.
[26] C. Macdonell. [n. d.]. ivshmem-uio. https://github.com/shawnanastasio/ivshmem-uio.
[27] F. McKeen, I. Alexandrovich, A. Berenzon, C. Rozas, H. Shafi, V. Shanbhogue, and U. Savagaonkar. 2013. Innovative Instructions and Software Model for Isolated Execution. In *Proc. Int. Workshop on Hardware and Architectural Support for Security and Privacy*.
[28] Microsoft Corporation. 2024. About Azure confidential VMs. https://learn.microsoft.com/en-us/azure/confidential-computing/confidential-vm-overview.
[29] T. Nakano and K. Kourai. 2021. Secure Offloading of Intrusion Detection Systems from VMs with Intel SGX. In *Proc. IEEE Int. Conf. Cloud Computing*. 297–303.
[30] K. Nakashima and K. Kourai. 2021. MigSGX: A Migration Mechanism for Containers Including SGX Applications. In *Proc. Int. Conf. Utility and Cloud Computing*.
[31] NGINX, Inc. [n. d.]. NGINX: High Performance Load Balancer, Web Server, & Reverse Proxy. https://www.nginx.com/.
[32] C. Oo. 2024. OpenHCL: A Linux based paravisor for Confidential VMs. Linux Plumbers Conference 2024.
[33] Y. Ozaki, S. Kanamoto, H. Yamamoto, and K. Kourai. 2019. Detecting System Failures with GPUs and LLVM. In *Proc. ACM SIGOPS Asia-Pacific Workshop on Systems*. 47–53.
[34] PwC. 2014. US Cybercrime: Rising Risks, Reduced Readiness.
[35] J. Rutkowska and R. Wojtczuk. 2008. Preventing and Detecting Xen Hypervisor Subversions. Black Hat USA.
[36] F. Schwarz and C. Rossow. 2024. 00SEVen – Re-enabling Virtual Machine Forensics: Introspecting Confidential VMs Using Privileged in-VM Agents. In *Proc. USENIX Security Symposium*. 1651–1668.
[37] Y. Shen, H. Tian, Y. Chen, K. Chen, R. Wang, Y. Xu, Y. Xia, and S. Yan. 2020. Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX. In *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems*. 955–970.
[38] M. Shih, M. Kumar, T. Kim, and A. Gavrilovska. 2016. S-NFV: Securing NFV States by Using SGX. In *Proc. ACM Int. Workshop on Security in Software Defined Networks & Network Function Virtualization*. 45–48.
[39] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, K. Kono, S. Chiba, Y. Shinjo, and K. Kato. 2009. BitVisor: A Thin Hypervisor for Enforcing I/O Device Security. In *Proc. Int. Conf. Virtual Execution Environments*. 121–130.
[40] K. Takiguchi and K. Kourai. 2024. Protecting Nested VMs with AMD SEV. Poster at ACM SIGOPS Asia-Pacific Workshop on Systems.
[41] TechSpot News. 2010. Google Fired Employees for Breaching User Privacy. http://www.techspot.com/news/40280-google-fired-employees-for-breaching-user-privacy.html.
[42] The Apache Software Foundation. [n. d.]. Apache HTTP Server Benchmarking Tool. https://httpd.apache.org/.
[43] The LLVM Foundation. [n. d.]. The LLVM Compiler Infrastructure. https://llvm.org/.
[44] The QEMU Project Developers. [n. d.]. Inter-VM Shared Memory Device. https://qemu-project.gitlab.io/qemu/system/ivshmem.html.
[45] C. Tsai, D. Porter, and M. Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *Proc. USENIX Annual Technical Conf.* 645–658.
[46] J. Wang, A. Stavrou, and A. Ghosh. 2010. HyperCheck: A Hardware-assisted Integrity Monitor. In *Proc. Int. Symp. Recent Advances in Intrusion Detection*. 158–177.