AMD Confidential Computing Technologies Evaluation Report

Roberto Castellotti
TU Munich

Abstract

We report a preliminary performance evaluation of AMD SEV (Secure Environment Virtualization) technologies. We are running virtual machines using QEMU/KVM as hypervisor on a host we control. Details about our environment can be found in section 3.

Confidential Computing technologies may be predominant in the future, as more and more customers with sensitive computing workloads move their code from on-premise hardware to public cloud vendors. An important aspect of successfully migrating workload will be identifying new performance bottlenecks. After thoroughly explaining how the technologies introduced by AMD work we run some measurements to assess the impact these have on microbenchmarks and traditional workloads such as compilation of popular open-source projects.

We find that *Confidential Computing* technologies cause some slowdowns for memory operations, while the performance degradation in CPU intensive workloads is generally negligible.

1 Introduction

Confidential Computing is a topic that started to become relevant in the last years. In the last two decades the way software is released to production changed radically: the majority of software deployed nowadays is hosted by cloud providers (Google Cloud Platform, Amazon Web Services, Microsoft Azure, etc.), while some code is still run on-premise for privacy/safety reasons. Confidential Computing aims to provide a safe environment for developers to run highly sensitive code on hosts they don't fully control.

Cloud providers' customers want to be sure no one can access their disks, memory or CPU registers, neither other customers running virtual machines on the same hardware, nor whoever is controlling the hypervisor, be it the cloud vendor or, in worst case scenarios, malign actors who compromised the physical machines. Check Appendix A to see a simple demo attack where a process running on the host accesses memory from the Virtual Machine. Encryption at rest, designed to prevent the attacker from accessing the unencrypted data by ensuring the data is encrypted when on disk [2], has been around for a long time, and is currently supported by all major providers [18], [16], [2] but, almost by definition, it leaves a few components used in daily computing unencrypted, namely RAM and CPU registers. To tackle this issue major chip producers started to develop technologies to enable *Confidential computing*, like *AMD Secure Encrypted Virtualization (SEV)* [6], *Intel Trusted Domain Extensions (TDX)* [11] and *Arm Confidential Compute Architecture (CCA) [14]*. In this article we will focus on summarizing the AMD technologies before running some measurements to evaluate the impact of *Confidential Computing* technologies.

2 Background

2.1 AMD Secure Memory Encryption (SME)

AMD Secure Memory Encryption (SME) is the basic building block for the more sophisticated technologies we'll cover later, so it is imperative we understand how it works. In a machine with *SME* enabled memory operations are performed via dedicated hardware, in fact, on an entirely different chip on the same die as the CPU. AMD EPYC™ processors introduced two hardware security components:

- a *AES-128 hardware encryption engine* embedded in the memory controller, a chip whose job is to make sure data transferred to and from the main memory is encrypted during write operations and decrypted during read operations. See Figure 1 to better understand how data is managed when *SME* is used. This memory controller is inside the EPYC SOC, so that memory lines leaving the SOC are always encrypted

- the *AMD Secure Processor*, a small processor providing cryptographic functionality to perform secure key
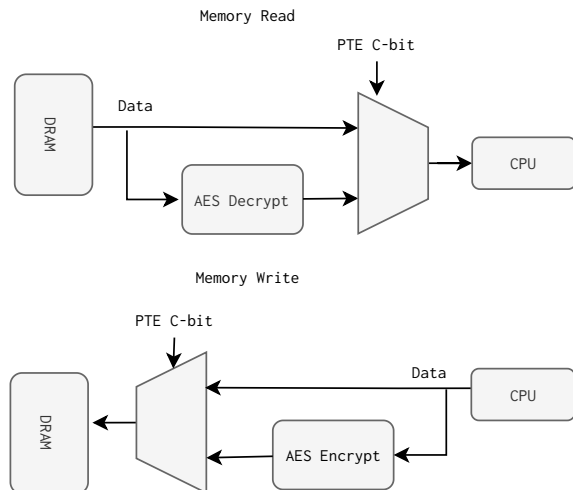
Figure 1: Memory Encryption Behavior, from [6]

Figure 2: SEV security layers, from [6]

generation and key management

The AES engine needs a key to work with, which is generated securely by the *AMD Secure-Processor (SMD-SP)*, a 32 bit micro-controller not accessible by software running on the main CPU. Furthermore *SME* does not require software running on main CPU to participate in Key Management making the enclave even more secure by reducing the attack surface.

The *C-bit* is a bit present in every memory page and indicates whether the current page is to be encrypted. It can be retrieved together with some additional information by running the cpuid command to inspect leaf 0x8000001F, as specified by AMD Reference Manual [9]:

```
$ cpuid -1 -l 0x8000001F
CPU:
AMD Secure Encryption (0x8000001f):
    SME: secure memory encryption support   = true
    SEV: secure encrypted virtualize support = true
    VM page flush MSR support               = true
    SEV-ES: SEV encrypted state support     = true
    SEV-SNP: SEV secure nested paging       = true
    VMPL: VM permission levels              = true
    Secure TSC supported                    = true
    virtual TSC_AUX supported               = false
    hardware cache coher across enc domains = true
    SEV guest exec only from 64-bit host    = true
    restricted injection                    = true
    alternate injection                     = true
    full debug state swap for SEV-ES guests = true
    disallowing IBS use by host             = true
    VTE: SEV virtual transparent encryption = true
    VMSA register protection                = true
    encryption bit position in PTE          = 0x33 (51)
    physical address space width reduction  = 0x5 (5)
    number of VM permission levels          = 0x4 (4)
```
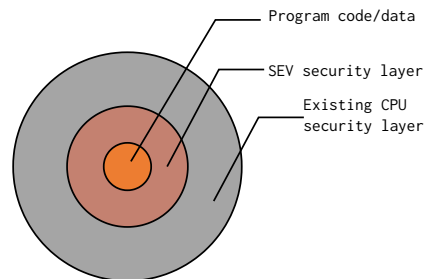
```
number of SEV-enabled guests supported  = 0x1fd (509)
minimum SEV guest ASID                  = 0x80 (128)
```

We can see SME, SEV, SEV-ES and SEV-SNP are enabled, and that the *C-bit* on our machine is 51.

SME is a very powerful mechanism to provide memory encryption, but it requires support from the Operating System/hypervisor, while *Transparent SME (TSME)* is a solution to encrypt every memory page regardless of the *C-bit*. As the name suggests this technology provides encryption without further modification to OS/HV, which is crucial because it means that neither Operating System developers nor hypervisor developers have to implement and maintain additional code and older operating systems can be run in a protected way with TSME.

2.2 AMD Secure Encrypted Virtualization (SEV)

AMD SEV is an attempt to make virtual machines more secure to use by encrypting data to and from a virtual machine, and it enables a new security model protecting user processes from higher privileged resources, such as hypervisors or some privileged code running on the physical computer hosting the virtual machines. This makes running code in a secure environment without needing to trust the hypervisor possible.

When SEV is enabled virtual machines tag their data with a *VM ASID* (an unique identifier for that specific machine), which is used inside the SOC to isolate it from other VMs and the hypervisor, when data leaves the chip we have no such problem because it is encrypted using the previously exchanged AES-128 bit key. The aforementioned expedients provide strong cryptography isolation between VMs run by the same hypervisor and between VMs and the hypervisor itself. SEV guests can choose which pages to encrypt setting the C-bit as mentioned before for SME. Only memory pages explicitly meant for outside communications with the hypervisor are considered shared and thus not encrypted. This and other information are discussed more in depth in the introductory white-paper [6].

2.3 AMD Secure Encrypted Virtualization-Encrypted State (SEV-ES)

Up until now we only discussed encryption for memory, but a crucial portion of the system we want to protect are CPU registers. *AMD SEV-ES* encrypts all CPU register contents when a VM stops running.

Protecting CPU register may be a daunting task because sometimes an hypervisor may need to access VM CPU's register to provide services such as device emulation. These accesses must be protected, and the *ES* technology allows the guest VM to decide which registers are encrypted, in the same way it can choose which memory pages are to be encrypted via the C-bit.

SEV-ES introduces a single atomic hardware instruction, VMRUN, when this instruction is executed for a guest the CPU loads all registers, when the VM stops running (VMEXIT), register's state is saved automatically to back to memory. These instructions are atomic because we need to be sure no one can sneak into this process and alter it, *ES* guarantees in this way it is impossible to leak memory.

Whenever hardware saves registers it encrypts them with the very same AES-128 key we mentioned before, furthermore the CPU computes an integrity-check value and saves it into a memory region not accessible by the CPU, on the next VMRUN instruction this will be checked to ensure nobody tried to tamper register's state. For further information about external communication consult the white-paper [12] and AMD reference manual chapter 15 [9].

Similarly to *AMD-SEV AMD-ES* is completely transparent to application code, guest VM developers and hypervisors developers are the only ones who need to implement these specific features.

2.4 AMD Secure Encrypted Virtualization-Secure Nested Paging (SEV-SNP)

After the introduction of *AMD-SEV* and *AMD-ES* AMD introduced a newer generation of SEV called *Secure Nested Paging (SEV-SNP)*, which builds on top of the technologies we have seen and extends them further to implement strong memory integrity protection to prevent hypervisor based attacks, such as *replay attacks*, *memory remapping*, *data corruption* and *memory aliasing*. Let's now explain what these attacks are before we cover how *SEV-SNP* mitigates them.

- a *replay attack* happens when a malicious actor captures the state at a certain moment and modifies memory successfully with those values

- *data corruption* attacks happen when the attacker, knowing they are not able to read memory, decides to corrupt the memory to trick the machine into unpredicted and possibly dangerous behavior

- a *memory aliasing* attack happens when an external actor may map a memory page to multiple physical pages he controls

- a *memory remapping* attacks happens whenever the intruder maps a page to a different physical page

These attacks are a problem because a running program has no notion of memory integrity and it could end up in a state that was not originally considered by the developers and this may lead to security issues.

The basic principle of *SEV-SNP* integrity is that if a VM is able to read a private (encrypted) page of memory, it must always read the value it last wrote. What this means is the VM should be able to throw an exception if the memory a process is trying to access was tampered by external actors.

2.4.1 Threat Model

In this computing model we consider:

- *AMD System-On-Chip (SOC) hardware*, *AMD Secure Processor (AMD-SP)* and the **VM** as fully trusted, to this extent the VM should enable Full Disk Encryption (FDE) at rest, such as LUKS, major cloud providers have been supporting FDE for long time.

- **BIOS** on the host system, the **hypervisor**, **device drivers** and **other VMs** as untrusted, this means the threat model assumes they are malicious and they may conspire to compromise the security of our Confidential Virtual Machine.

The way *SEV-SNP* ensures protection against the attacks we mentioned before is by introducing a new data structure, a *Reverse Map Table (RMP)* that tracks ownership of memory pages. Using the *RMP* it is possible to enforce that only the owner of a certain memory page can alter it.

A page can be owned by the VM, the hypervisor or by the AMD Secure Processor. The *RMP* is used in conjunction with standard x86 page tables mechanisms to enforce memory restrictions and page access rights. Introducing *RMP* check for write operations on memory mitigates replay, remapping and data corruption attacks.

To prevent memory remapping a technique called *Page Validation* is introduced. Inside each *RMP* entry there is the *Validated* bit. Pages assigned to guests that have no *Validated* bit set are not usable by the hypervisor, the guest can only use the page after setting the validated bit through a PVALIDATE instruction. The VM will make sure that it is not possible to validate a SPA (System Physical Address) corresponding to a GPA (Guest Physical Address) more than once.

More details are discussed in the introductory white-paper [7]

Having introduced the main building blocks of AMD's effort to popularize *Confidential Computing*, we will run some

benchmarks to measure if and how these technologies impact performances.

3 Environment

We are running our experiments in QEMU/KVM virtual machines, with 16GB of RAM and 16 vCPUs each, the table 1 contains more details about our hardware and software versions. The code to setup the environment and launch VMs can be found at https://github.com/rcastellotti/gr.

QEMU is a generic open source machine emulator and virtualizer [3], we use QEMU together with KVM, the Kernel Virtual Machine to virtualize our machines.

OVMF is a project [17] aiming to enable UEFI support for virtual machines, based on EDK 2. We will use OVMF to generate the executable firmware and the non-volatile variable store. It is imperative to create a vm-specific copy of OVMF_vars.fd because the variable store should be private for every virtual machine. UEFI support is mandatory to run a SEV-SNP machine.

4 Benchmarks

We expect the aforementioned encryption techniques used in SEV/SEV-ES/SEV-SNP to cause a degradation in performance, especially for memory-intensive workloads. Additionally we want to investigate whether this technology slows down CPU intensive workloads and disk operations. To quantify the performance impact we have to trade off in order to have a more secure virtual environment we are running three distinct kinds of benchmarks: Compilation, Memory Benchmarks and I/O benchmarks. We are using rcastellotti/tinyben [4], an experimental benchmarking tool aimed to replace the popular Phoronix Test Suite [13] benchmarking suite to run three main categories of benchmarks:

Compilation Benchmarks We are using compilation as an all-around benchmark because it is a "real world" benchmark. We are compiling some popular open-source projects like Godot Game Engine, Linux (defconfig), the entire LLVM Project (using ninja) and measuring how long does it take to complete the compilation process. From Figure 3 we can see there is a barely measurable slowdown in completion times, , on average a slowdown of a factor of 0.96.

Memory Benchmarks To benchmark memory we are using ssvb/tinymembench [19], a tool to measure memory throughput and latency. We are mainly interested in bandwidth for the MEMCPY and MEMSET operations. In

Figure 4 we can see bandwidth for MEMSET is barely impacted, while the slowdown in MEMSET is a little more evident, in a factor of approximately 0.92 on average. We suspect it might be related to lookups in the *Reverse Map Table*, but we are not excluding the bottleneck might be related to something entirely different.

I/O Benchmarks To measure I/O performance we are running two different benchmarks, first we are performing 2500 SQLite insertions in a table, then we are running redis-benchmark, a tool included in Redis. We are interested in Requests per Second and latency for the main operations (namely SET and GET). In Figures 5,6 and 7 we see the results. Redis benchmarks highlight a certain performance degradation, in terms of requests per second this approximately translates in a slowdown of factor 0.89 for GET operations and 0,91 for GET operations, this is reflected in the minimum latency visualization.

As we can see the overhead introduced by the usage of *Confidential Computing* technology across the three categories of benchmarks is generally mild, as discussed before, in almost every benchmark, with the exception for LZ4-compression, where we can see a slowdown in factor or 0.8 approximately. Further investigation tweaking the amount of resources available to each machine is necessary to correctly esteem the exact performance degradation values.

One reason why CPU-intensive workloads are little to no penalized by SEV-SNP is the processor might not be doing much more when executing that kind of workload on a confidential machine. Instructions that really slow down the execution are VMEXIT and VMRUN, since the compilation of a certain program should not include a considerable amount of those our workloads are barely impacted.
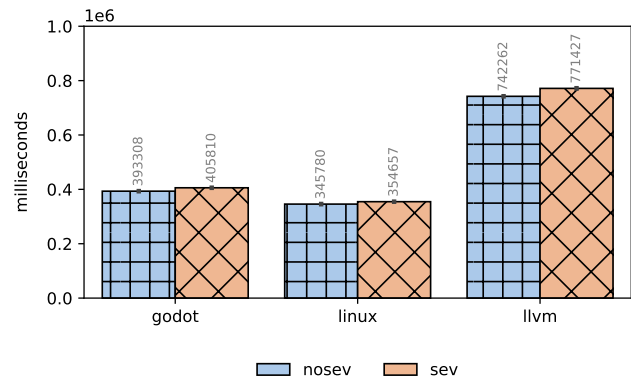
Figure 3: Compilation benchmarks, in milliseconds (Lower is Better)

To investigate storage related operations we ran some additional benchmarks with different storage virtualization techniques, more specifically we tested *virtio-scsi*, *virtio-blk*

| Host CPU	`AMD EPYC 7713P 64-Cores`
Host Memory	`HMAA8GR7AJR4N-XN (Hynix) 3200MHz 64 GB × 8 (512GB)`
Host Kernel	`6.3.0-rc2 #1-NixOS SMP PREEMPT_DYNAMIC (NixOS 23.05) commit:` fea9b78
QEMU	`8.0.0 (AMD) (patched) commit:` a248931
OVMF	`Stable 202211 (patched) commit:` 6598f62
Guest vCPUs	`16`
Guest Memory	`16GB`
Guest Kernel	`5.19.0-41-generic #42-Ubuntu SMP PREEMPT_DYNAMIC (Ubuntu 22.10 )`

Table 1: Experiment Environment
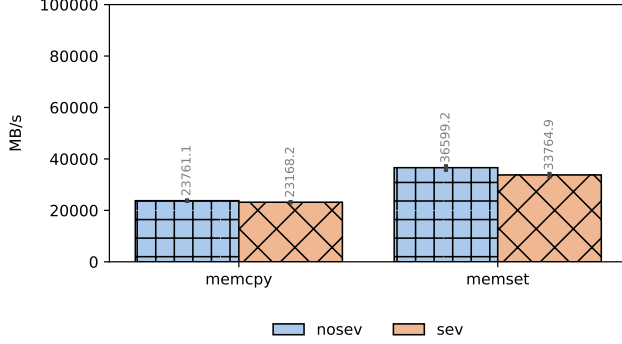
Figure 4: tinymembench bandwidth benchmark, MEMSET and MEMCPY (Higher is Better)
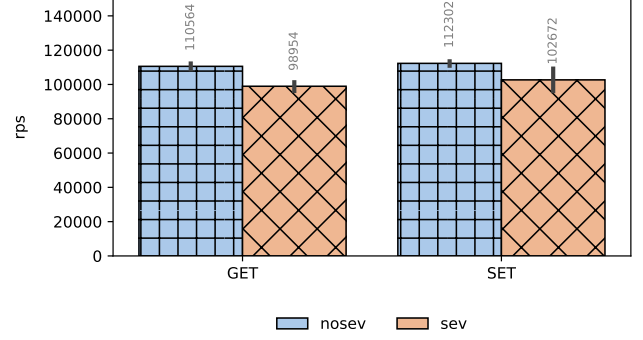

Figure 6: Redis-benchmark, SET and GET operations Requests Per Second (Higher is Better)
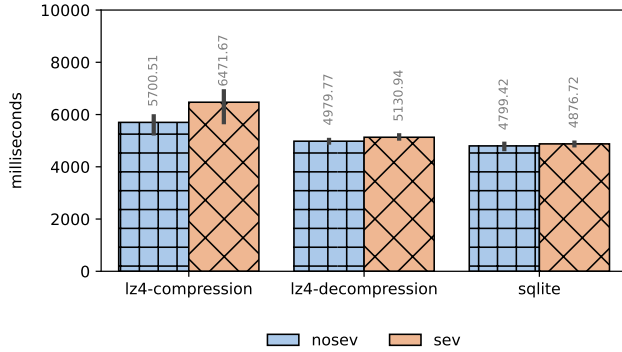

Figure 5: Miscellaneous benchmarks, LZ4 compression and decompression and SQLite insertions time to completion (Lower is Better)
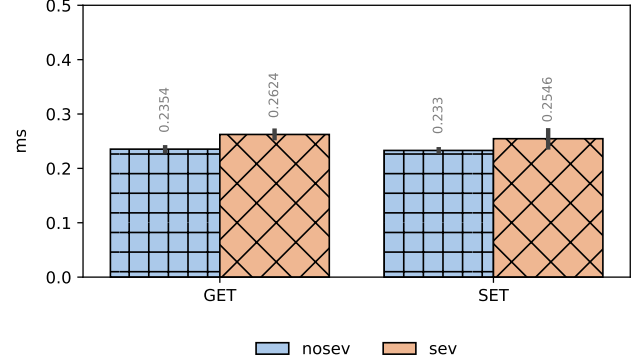

Figure 7: Redis-benchmark, SET and GET operations minimum latency(milliseconds) (Lower is Better)

and *nvme* technologies. Virtio is the main platform for IO virtualization in KVM, providing a common framework for hypervisors to do IO virtualization.

nvme is the user-space NVMe driver that enables virtual machines to interact with NVMe devices.

virtio-blk devices are very simple block devices, the front-end driver reads and writes by appending commands to the virtualization queue so that the back-end driver can process them on the host.

virtio-scsi aims to overcome some limitations introduced by *virtio-blk*, support more devices per guest (one PCI device per disk is not a limiting factor anymore) and supports technologies like multiqueueing while keeping the performance of *virtio-blk*, additionally *virtio-scsi* provides a pass-through technology to present physical storage devices directly to guests.[15]

We perform some measurements using the FIO [1] benchmarking tool with a setup similar to the one used in [20] to evaluate: bandwidth, average latency and IOPS. We set the `-direct=1` flag to use unbuffered I/O. Results are reported in 8, 9, 10. We also report the values in tables to better analyze the relative performance variations.

The FIO benchmarks highlight some interesting patterns: bandwidth is severely impacted, especially in read work-

| benchmark	test cases	FIO Configuration (bs, rw, iodepth, numjobs)
bandwidth	read	(128K, read, 128, 1)
bandwidth	write	(128K, write, 128, 1)
IOPS	randread	(4K, randread, 32, 4)
IOPS	mixread	(4K, randread 70%, 32, 4)
IOPS	mixwrite	(4K, randwrite 30%, 32, 4)
IOPS	randwrite	(4K, randwrite, 32, 4)
average latency	randwrite	(4K, randread, 1, 1)
average latency	randwrite	(4K, randwrite, 1, 1)
average latency	read	(4K, read, 1, 1)
average latency	write	(4K, write, 1, 1)

Table 2: FIO benchmarks, from [20]
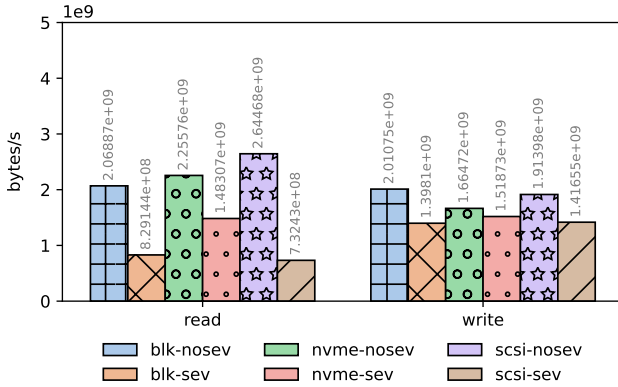
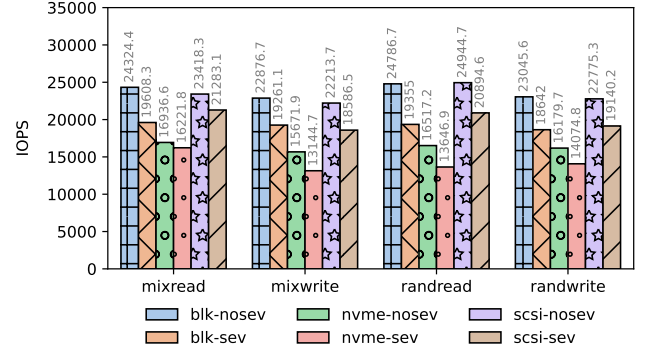Figure 8: Storage Bandwidth Benchmarks (Higher is Better)

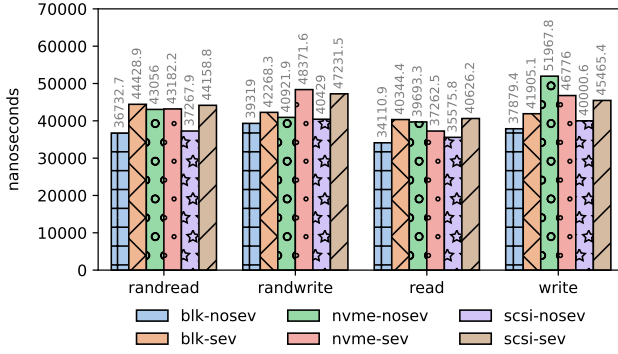Figure 10: IOPS benchmarks (Higher Is Better)

Figure 9: Average Latency Benchmarks (Lower is Better)

mode	storage	nosev	sev	result
read	blk	2068866712	829144265	0.400772
read	nvme	2255760134	1483068817	0.657459
read	scsi	2644684295	732429620	0.276944
write	blk	2010752479	1398101333	0.695312
write	nvme	1664716006	1518729595	0.912306
write	scsi	1913978295	1416545941	0.740106

Table 3: FIO bandwidth benchmarks results, measured in KB/s, ratio is the ratio between **sev** and **nosev**

mode	storage	nosev	sev	result
mixread	blk	24324.394544	19608.347670	0.806119
mixread	nvme	16936.555110	16221.782178	0.957797
mixread	scsi	23418.259782	21283.104652	0.908825
mixwrite	blk	22876.690811	19261.131521	0.841954
mixwrite	nvme	15671.907694	13144.662288	0.838740
mixwrite	scsi	22213.710702	18586.500284	0.836713
randread	blk	24786.686838	19354.991140	0.780862
randread	nvme	16517.169681	13646.936332	0.826227
randread	scsi	24944.714055	20894.627770	0.837637
randwrite	blk	23045.626374	18642.013938	0.808918
randwrite	nvme	16179.730897	14074.845638	0.869906
randwrite	scsi	22775.325804	19140.186916	0.840391

Table 4: FIO IOPS benchmarks results, ratio is the ratio between **sev** and **nosev**

mode	storage	nosev	sev	ratio
randread	blk	36732.676746	44428.919037	1.209520
randread	nvme	43056.001194	43182.237293	1.002932
randread	scsi	37267.866497	44158.765942	1.184902
randwrite	blk	39318.967918	42268.305611	1.075011
randwrite	nvme	40921.898750	48371.563309	1.182046
randwrite	scsi	40428.971912	47231.466854	1.168258
read	blk	34110.888638	40344.405910	1.182743
read	nvme	39693.322182	37262.537930	0.938761
read	scsi	35575.824402	40626.232517	1.141962
write	blk	37879.406013	41905.050934	1.106275
write	nvme	51967.809612	46775.977871	0.900095
write	scsi	40000.623734	45465.357731	1.136616

Table 5: FIO average latency benchmarks, measured in nanoseconds, ratio is the ratio between **sev** and **nosev**

loads, further investigation is needed to better pinpoint the bottleneck causing this. The number of Input/Output Operations per second is always lower on the SEV machine, as expected, however the difference is very contained, being on average a factor of 0.8 lower in the confidential machines. Average latency is almost always worse on SEV machines, in certain cases it is marginally lower, but we believe this is due to some artifact and is not relevant.

5   Final Remarks and Conclusion

Overall, from a first evaluation, it seems SEV/SEV-ES/SEV-SNP technologies do not cause a noticeable degradation in performance. The only notable exception is read bandwidth across all different storage virtualization techniques, further evaluation may be needed to better pinpoint what causes this decrease in performance. All the code to run virtual machines and benchmarks is available at https://github.com/rcastellotti/gr

6   Future Work

Further performance evaluation for SEV/SEV-ES/SEV-SNP confidential virtual machines will include:

- repeating micro-benchmarks and FIO benchmarks tweaking machines configuration (vCPUs, RAM, storage virtualization)

- analyzing IO performance when using SEV Trusted Input Output [8]

- analyzing whether having multiple SEV machines on the same physical host incurs in a large overhead

- analyze performance overhead caused by VMRUN and VMEXIT when SEV-ES is enabled.

- investigate unikernels supporting SEV like microsoft/monza

- exploring Confidential Containers [5] project to identify potential bottlenecks

- understand AMD Secure VM Service Module [10], a module to offload sensitive operations onto a privileged guest and see if this leads to a performance improvement

References

[1] Jens Axboe. *Flexible I/O tester rev. 3.35*. URL: https://fio.readthedocs.io/en/latest/fio_doc.html (visited on 07/07/2023).

[2] Microsoft Azure. *Azure Data Encryption at rest*. URL: https://learn.microsoft.com/en-us/azure/security/fundamentals/encryption-atrest (visited on 07/07/2023).

[3] Fabrice Bellard. "QEMU, a Fast and Portable Dynamic Translator". In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC '05. Anaheim, CA: USENIX Association, 2005, p. 41.

[4] Roberto Castellotti. *a <1000 loc benchmark suite in python*. URL: https://github.com/rcastellotti/tinyben (visited on 07/07/2023).

[5] Confidential Containers. *Confidential Containers*. URL: https://github.com/confidential-containers (visited on 07/21/2023).

[6] Tom Woller David Kaplan Jeremy Powell. *AMD Memory Encryption*. 2021. URL: https://www.amd.com/system/files/TechDocs/memory-encryption-white-paper.pdf (visited on 07/07/2023).

[7] Advanced Micro Devices Inc. *AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More*. 2020. URL: https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf (visited on 07/07/2023).

[8] Advanced Micro Devices Inc. *AMD SEV-TIO: Trusted I/O for Secure Encrypted Virtualization*. 2023. URL: https://www.amd.com/system/files/documents/sev-tio-whitepaper.pdf (visited on 07/07/2023).

[9] Advanced Micro Devices Inc. *AMD64 Architecture Programmer's Manual Volumes 1–5*. 2023. URL: https://www.amd.com/system/files/TechDocs/40332.pdf (visited on 07/07/2023).

[10] Advanced Micro Devices Inc. *Secure VM Service Module forSEV-SNP Guestsn*. 2022. URL: `https://www.amd.com/system/files/TechDocs/58019-svsm-draft-specification.pdf` (visited on 07/21/2023).

[11] Intel®. *Intel® Trust Domain Extensions*. URL: `https://cdrdv2.intel.com/v1/dl/getContent/690419` (visited on 07/07/2023).

[12] David Kaplan. *Protecting VM Register State with SEV-ES*. 2017. URL: `https://www.amd.com/system/files/TechDocs/Protecting%20VM%20Register%20State%20with%20SEV-ES.pdf` (visited on 07/07/2023).

[13] Michael Larabel. *The Phoronix Test Suite open-source, cross-platform automated testing/benchmarking software*. URL: `https://github.com/phoronix-test-suite/phoronix-test-suite` (visited on 07/07/2023).

[14] Arm Limited. *Introducing Arm Confidential Compute Architecture*. URL: `https://developer.arm.com/documentation/den0125/latest` (visited on 07/07/2023).

[15] ovirt.org. *Virtio-SCSI*. URL: `https://www.ovirt.org/develop/release-management/features/storage/virtio-scsi.html` (visited on 07/31/2023).

[16] Google Cloud Platform. *Default encryption at rest*. URL: `https://cloud.google.com/docs/security/encryption/default-encryption` (visited on 07/07/2023).

[17] Tianocore's EDK II project. *Open Virtual Machine Firmware (OVMF)*. URL: `http://www.linux-kvm.org/downloads/lersek/ovmf-whitepaper-c770f8c.txt` (visited on 07/07/2023).

[18] Amazon Web Services. *Encryption of Data at Rest*. URL: `https://docs.aws.amazon.com/whitepapers/latest/efs-encrypted-file-systems/encryption-of-data-at-rest.html` (visited on 07/07/2023).

[19] Siarhei Siamashka. *Simple benchmark for memory throughput and latency*. URL: `https://github.com/ssvb/tinymembench` (visited on 07/07/2023).

[20] Shuai Xue et al. "Spool: Reliable Virtualized NVMe Storage Pool in Public Cloud Infrastructure". In: *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC'20. USA: USENIX Association, 2020. ISBN: 978-1-939133-14-4.

A Appendix

A.1 A simple attack to demo Confidential Computing

Let's demo a very simple attack, first of all we start two machines, *sev* and *nosev*, the former has SEV-SNP enabled, as we can check:

```
sev:~$ sudo dmesg | grep SEV
Memory Encryption Features active: AMD SEV SEV-ES SEV-SNP
SEV: Using SNP CPUID table, 31 entries present.
SEV: SNP guest platform device initialized.
```

We will write something into a file and cat it in order to load the data in memory

```
sev:~$ echo "hi from SEV!" > sev.txt
sev:~$ cat sev.txt
hi from SEV!
```

```
nosev:~$ echo "hi from NOSEV!" > nosev.txt
nosev:~$ cat nosev.txt
hi from NOSEV!
```

Now we can dump the memory for the processes using `gcore`

```
h:~$ sudo gcore -o mem-dump <SEV_PID>
h:~$ grep -rnw mem-dump.<SEV_PID> -e "hi from SEV!"
h:~$
```

```
h:~$ sudo gcore -o mem-dump <NOSEV_PID>
h:~$ grep -rnw mem-dump.<NOSEV_PID> -e "hi from NOSEV!"
grep: mem-dump.<NOSEV_PID>: binary file matches
```

From the host machine we are able to see nosev's machine memory while this is not possible with SEV enabled.