SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

# CVM-IO: Secure High-Performance Storage Stack for Confidential Virtual Machines

Robert Maximilian Schambach

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

# CVM-IO: Secure High-Performance Storage Stack for Confidential Virtual Machines

# CVM-IO: Sicherer Hochleistungs-Speicherstapel für vertrauliche virtuelle Maschinen

| | |
|---|---|
| Author: | Robert Maximilian Schambach |
| Supervisor: | Prof. Pramod Bhatotia |
| Advisor: | Dr. Masanori Misono |
| Submission Date: | 15.02.2024 |

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Garching near Munich, 15.02.2024 Robert Maximilian Schambach

## Acknowledgments

We thank Intel Inc. for the Intel TDX machine access via Intel's academic research program.

# Abstract

*Confidential Virtual Machines* (CVMs) offer protection of data in use, allowing for trustworthy computation on remote untrusted infrastructure. CVMs achieve this protection guarantee by relying on trusted hardware, which excludes host-controlled virtualization stack components from the *Trusted Computing Base* (TCB). These stack components include the host operating system, the hypervisor, and external devices. However, decreasing the base of trust comes with several trade-offs. For one, the CVM's TCB excludes persistent storage devices, disallowing CVMs to offer the same protection guarantees for data at rest as data in use. For another, CVMs disallow DMA into their memory and require expensive context switches when handling IO for micro-architectural security, incurring storage IO performance overheads. Therefore, CVMs are not secure drop-in replacement VMs for storage IO-heavy applications.

This thesis thoroughly investigates CVM storage IO and proposes a novel design for the CVM storage IO stack that provides the same CVM-backed protections to data at rest as data in use while circumventing the CVM architecture imposed performance overheads. Moreover, this thesis thoroughly investigates CVM storage IO. We design a CVM-backed storage backend, which we use to save integrity- and freshness-preserving metadata for a high-performance host storage backend. Further, we sketch a bypass of CVM-architecture imposed storage IO overheads. In particular, CVM-IO encrypts IO data into DMA-able shared memory to avoid an extra IO copy and uses shared poll queues to circumvent `VM_EXIT`s.

With host-side caching disabled, CVMs pose no bandwidth and *IO Operations per Second* (IOPS) storage IO overheads over native VMs, yet cause up to 162% latency overheads. With host-side caching enabled, CVMs cause overheads in bandwidth, IOPS, and latency of up to 189%, 162%, and 198%, respectively. Further, we find the CVM-architecture-induced bounce buffer to produce no storage IO overhead. We partially implement a functioning prototype of CVM-IO.

# Contents

# 1 Introduction

The current dominance of the cloud computing paradigm is indisputable. However, cloud computing users continue to raise security and privacy concerns due to the computing paradigm's execution on an untrusted host. To address these concerns, hardware vendors offer *Confidential Virtual Machine* (CVM)-enabling technologies, such as AMD SEV [KPW16], Intel TDX [Int21], and Arm CCA [Arm]. These CVMs offer protection of data in use and ensure hardware-based isolation between the CVM and the host and hypervisor.

Although CVM-enabling technologies offer near identical compute performance to legacy VMs while granting protection of data in use, CVMs do not extend these properties to storage IO. Specifically, the CVM storage IO stack has two main issues.

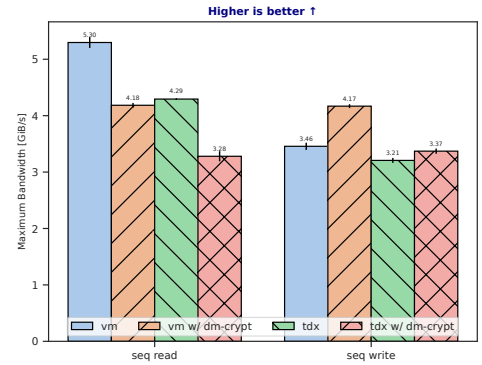
First, in the stack's default configuration, the stack exhibits significant performance overheads compared to native VM storage IO. As we show in Figure 1.1, and specifically the overhead of the Intel TDX CVM read latency compared to the native VM read latency in Figure 1.1c, these overheads are up-to 375%.

The reason for this performance drop is that such confidential virtualization solutions no longer trust the hypervisor and host *Operating System* (OS). Hence, these virtualization stack components cannot access arbitrary guest memory, and the CVM technologies must take security measures when the CPU switches from a secure execution context. Consequently, a guest must copy data into host-shared memory to execute storage IO and perform additional security-ensuring operations, thus adding storage IO performance overhead.

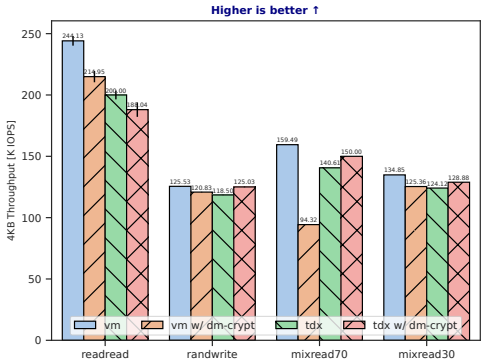
Second, although CVMs protect data in use, CVMs do not protect data leaving the VM. Instead, the CVM-contained software must offer such protection via non-CVM-enabled mechanisms. These mechanisms therefore cannot offer freshness of storage IO data, yet which CVM-machanisms do enable for data in use. As such, the CVM protection guarantees do not extend to storage IO data.

Indeed, the CVMs' inability to transparently offer their protection guarantees to storage IO while inducing significant performance tax may disqualify the use of CVMs for many data-intensive applications, such as databases, or for training large-language models. CVM users expect the CVM-enabling technologies to serve as a trust anchor for protecting data in use and persistently stored data. However, for persistent data, this expectation dangerously does not hold. The base of the existing kernel mechanism's trust for disk protection does not lie in the CVM, thus adhering the disk protection to a weaker threat model than these CVM-enabling technologies.

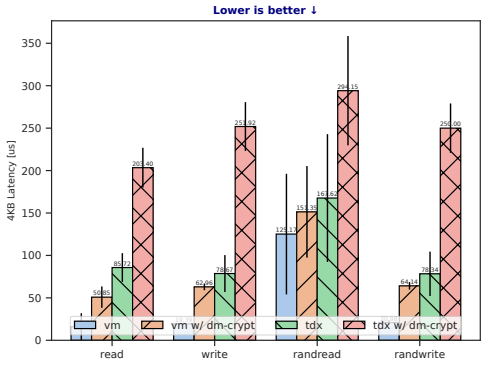
Further, the CVM software-based storage IO performance tax worsens the increasing negative performance impact of software in storage IO. With the ongoing performance improvements of storage devices, the storage IO bottleneck has shifted away from the storage IO hardware to the storage IO software [Xu+15]. Hence, the CVM-imposed storage IO performance tax worsens this




(a) Bandwith comparison.




(b) IOPS comparison.




(c) Latency comparison.

Figure 1.1: Native and confidential VM storage IO performance comparison, with a default configuration. *vm* corresponds to a native VM, and *tdx* to an Intel TDX CVM. *w/ dm-crypt* is the corresponding configuration with the `dm-crypt` protection mechanism enabled. We discuss benchmarking details in Chapter 5.

software bottleneck. As such, potential CVM users must consider using CVMs as a trade-off for security against storage IO performance and thus may decide against using CVMs for storage IO performance-critical applications.

We address these issues with the following research question: *How can we design the CVM storage IO stack to serve as a CVM-secured drop-in storage stack while avoiding CVM-imposed performance loss?*

To answer this question, we propose CVM-IO, which transparently extends the Linux kernel storage stack to set the CVM as the root of trust for the stack-contained protection mechanisms while bridging the CVM architecture imposed performance impediments. CVM-IO thereby provides CVM-backed confidentiality, integrity, freshness, and authenticity to storage IO data. CVM-IO also allows trade-offs in protection guarantees for performance. Further, this storage stack aims to improve storage IO performance compared to a native CVM storage stack.

We need to solve three challenges while designing CVM-IO. First, CVMs do not anchor existing kernel protection mechanisms into the CVM's root of trust. Second, CVM storage IO requires a shared memory bounce buffer, which causes an extra copy, thus taxing storage IO performance. Third, traditional CVM storage IO is interrupt-based, requiring expensive CVM context switches when receiving interrupts upon host-completion of storage IO.

CVM-IO solves the challenges by (1) using CVM sealing functionality to persistently and securely save security primitives, (2) setting a shared memory buffer as the encryption operation's target, and (3) adapting virtualization infrastructure to share shared memory hugepages with a high-performance storage IO backend in which the storage stack places shared polled queues.

We implement our prototype based on the Linux kernel targeting x86_64 with the QEMU/KVM hypervisor [Bel05]. In this thesis, we alter the Linux kernel's *device-mapper* layer to encrypt to and decrypt from host-shared memory, and enable CVM guest-side polling. To implement CVM-IO entirely, we must also modify the guest Linux kernel's *device-mapper* layer to include a CVM-backed storage backend to prevent replay attacks, and enable the guest kernel driver, the hypervisor, as well as the host storage engine to enable polling for CVMs.

We evaluate CVM-IO via the synthetic storage IO benchmarking tool fio [Axba]. We thereby provide an intricately configured storage IO setup and evaluate the overheads of the AMD SEV-SNP and the Intel TDX CVM under various storage IO configurations. Moreover, we investigate the overheads of the CVM-architecture-induced bounce buffer in addition to the overheads of interrupt-based storage IO. In particular, we measure the IO stacks for bandwidth, average latency, and *IO Operations Per Second* (IOPS).

The contributions of this paper are as follows:

1. We analyze the performance overheads of CVM storage IO, particularly the CVM-architecture-induced extra copy, the interrupt-based storage IO overhead, and of the CVM-IO protection mechanism `dm-crypt`.

2. We propose CVM-IO, a transparent storage IO stack for CVMs that provides flexible CVM-grounded IO data protection and avoids CVM storage IO performance overheads.

3. We partially implement our proposal in a guest Linux kernel, the QEMU/KVM hypervisor [Bel05], thereby enabling `dm-crypt`-enabled storage IO which avoids the CVM extra

copy.

# 2 Background and Motivation

We first inspect how far the state-of-the-art native CVM storage IO with existing kernel protection mechanisms achieves our protection goals. We then highlight the shortcomings of state-of-the-art protection mechanisms in achieving these goals and native CVM storage IO's performance overheads.

## 2.1 State-of-the-art CVM Storage IO Data Protection

The protection guarantees we aim to provide to storage IO via CVM-IO correspond to the CVM-provided guarantees. These guarantees comprise confidentiality, integrity, authenticity, and freshness. The motivation behind storage IO protection for the first three guarantees is straightforward. Requiring the in-use confidentiality, integrity, and authenticity protection of data merits the same protection guarantees at rest.

To further motivate the freshness of persistent data, we give as an example an adversarial bank IT administrator who wishes to gain additional spending power by manipulating the persistent storage of a CVM-contained bank transaction database. While the CVM provides the database with sufficient protection, the database itself only provides confidentiality, integrity, and authenticity to its persistently stored data. As the database does not provide freshness, the IT administrator may save the entire persistent storage to a separate storage medium, withdraw all of their money, and restore the persistent storage state to the previously copied state on the external medium. Hence, the set-back state does not reflect the administrator's withdrawal. As the administrator violates no database-applied protection guarantees, we require freshness to mitigate such an attack.

To achieve these protection guarantees, the Linux kernel offers two transparent storage IO block-level components: `dm-crypt` [Broa; Fru05] and `dm-integrity` [BPM18; Brob]. The kernel implements these components in the kernel device-mapper framework, which creates virtual block devices. `dm-crypt` and `dm-integrity` thereby apply protection guarantees on block-level storage IO upon receiving reads or writes [BPM18].

Although we can operate these devices in standalone modes, when we combine the two, the device mapper modules provide authenticated disk encryption. Specifically, the devices provide disk-level encryption and sector-level authentication and integrity guarantees. However, `dm-crypt` and `dm-integrity` do not provide replay protection or, in other words, freshness guarantees. Freshness requires additional hardware support to store independent metadata, which a generic-disk-focused kernel device does not provide [BPM18].

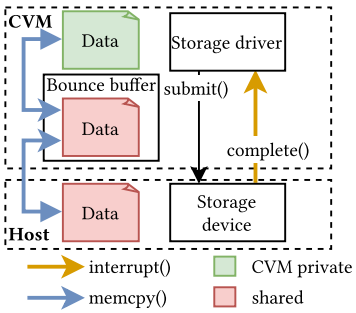



Figure 2.1: Overview of native CVM Storage IO architectural overheads. The CVM must share IO data via a host shared memory bounce buffer, requiring an additional copy. For the CVM to submit IO requests to the host, the CVM must write to shared memory via *Memory Mapped IO* (MMIO). Further, for the host to signal IO completion to the guest, the host must issue interrupts, causing expensive CVM `VM_EXIT`s and `VM_ENTRY`s. (Green marks CVM-private memory, whereas red marks CVM-host-shared memory. Blue marks edges which perform a `memcpy`, while orange marks edges which perform an interrupt and thereby a context switch.)

## 2.2 CVM Storage IO Performance Overheads

As CVMs neither trust the host nor hypervisor, CVM storage IO has inherent architectural overheads. The two main architecture-imposed overheads are the so-called *bounce buffer* and *CVM* `VM_EXIT` tax, which we show in Figure 2.1.

As we show in Figure 2.1, CVM storage IO requires the bounce buffer. In native VM storage IO, the host device reads and writes data directly from or into the guest. However, due to the CVM protection guarantees, reading or writing CVM private memory is prohibited, thus hindering native VM storage IO. Therefore, to perform storage IO, a CVM must provide the bounce buffer to which the host device performs native storage IO. The CVM then copies data into or copies data out of the bounce buffer to perform the IO, thus "bouncing" IO data to or from the host across this buffer.

Several works [Li+23; Intd; YG23] find the extra-copy overhead to have significant performance impact. However, our evaluation found that the copying overhead is not dominant in storage I/O, failing to reproduce these works' claims. Further investigation is needed. Nonetheless, we present the design of zero-copying storage I/O.

Besides the bounce buffer performance overhead, CVMs also have a `VM_EXIT` overhead, which impacts storage IO performance. Upon a `VM_EXIT`, CVMs must protect their register contents for the subsequent context switch to prevent state leakage [Int21; Kap17; Kap20]. These protection measurements incur additional overhead.

These latencies thereby impact the paravirtual storage IO of CVMs as paravirtual storage IO is by default interrupt-based, which we show in Figure 2.1. To submit IO requests, the guest must write to host-device-shared memory via *Memory Mapped IO* (MMIO). To signal IO completion, the host must issue an interrupt to the corresponding component. Handling the interrupt requires

the execution of the guest's interrupt request handler, thus requiring a context switch, and hence a `VM_EXIT` and a subsequent `VM_ENTRY`. As the context switch in both cases requires the CVM technologies to take protective measures, the CVMs introduce additional latency.

We demonstrate the negative impact on storage IO performance of interrupt-based storage IO in Chapter 5 and Figure 5.1. Interrupt-based storage IO can add latency overheads compared to polling-based IO of up to 121% for CVMs.

# 3 Overview

This chapter presents an overview of the proposed system and motivates its design. Our proposed system is applicable to the QEMU/KVM hypervisor [Bel05] with either the AMD SEV-SNP [Kap20] or the Intel TDX [Int21] CVM-enabling technologies.

## 3.1 System Overview

To implement secure CVM storage IO, which removes CVM performance overheads, we design CVM-IO. CVM-IO provides a modified storage stack that grants flexible CVM-rooted storage IO data protection while avoiding CVM-architecture-imposed performance overheads.

As shown in Figure 3.1, we integrate CVM-IO inside the guest's and the host's storage stack. CVM-IO consists of two main components: the protection and zero-copy component, and the polling component (both in orange). While the CVM contains the protection and zero-copy component, the polling component spans the CVM and the host. CVM-IO intercepts in-flight block IO operations and modifies these data structures to enforce protection guarantees. Moreover, CVM-IO alters the behavior of existing storage stack elements to avoid the CVM-architecture imposed overheads.

The protection and zero-copy component performs two functions in one. The component encrypts data to shared memory via *Authenticated Encryption with Additional Data* (AEAD). In particular, the component offers protection guarantees to the IO data via encryption and additional integrity and freshness metadata while removing the extra copy required to store the IO data in shared memory. The other polling component avoids CVM `VM_EXIT`s by enabling polling-based storage IO. To this end, CVM-IO strives for the following design goals:

- *Security*: CVM-IO shall flexibly offer confidentiality, authenticity, integrity, and freshness to all storage IO data.
- *Performance*: CVM-IO shall provide substantial performance improvements over native CVM IO.
- *Transparency*: CVM-contained applications using CVM-IO must be agnostic of the storage stack's existence.

Figure 3.1 demonstrates how the CVM-IO executes storage IO via performing a write. Initially, CVM-IO intercepts an application-issued block IO write request. In Step ①, the protection and zero-copy component modifies the block IO data. In particular, the component uses CVM-IO-modified device-mapper subsystem elements `dm-crypt` and `dm-integrity` to encrypt the IO data to shared memory via AEAD. The device subsystem thereby produces integrity and freshness

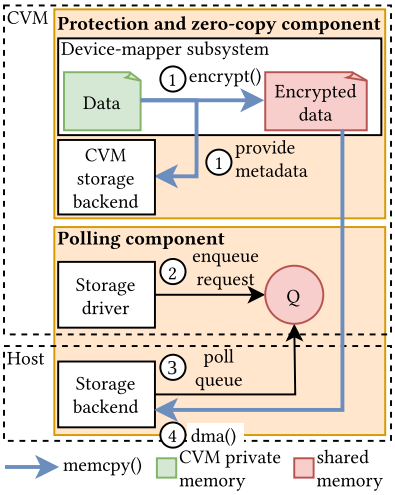



Figure 3.1: Overview of CVM-IO. CVM-IO consists of two main components: the protection and zero-copy component is located in the CVM, while the polling component spans across the CVM and host. This figure displays CVM-IO's handling of a `write()`. A `read()` follows the same path, with the difference of the inversion of the `memcpy` edges' direction. Further, the `read()` decrypts the encrypted data from shared into private memory. We also leave out the IO completion, as the completion is exactly the inversion of Steps ② and ③, taking place after Step ④. (Orange marks the components. Green marks CVM-private memory, whereas red marks CVM-host-shared memory. Blue marks edges which perform a `memcpy`.)

metadata. Moreover, the storage stack avoids the bounce buffer-imposed extra copy (Section 2.2), thus avoiding the first of the two main native CVM storage IO overheads.

CVM-IO further allows the Linux block layer to handle the write request natively. However, CVM-IO modifies the CVM-contained storage driver and host-contained storage backend to use polling-based IO. Hence, after the CVM-IO-altered driver enqueues the IO request (Step ②) into a shared-memory contained request queue, the driver does not issue an interrupt to notify the storage backend of the request. Instead, the backend polls the shared request queue, thus detecting the newly arrived requests (Step ③).

Therefore, the backend's polling avoids an expensive CVM `VM_EXIT`, which is another major CVM storage IO overhead (Section 2.2). Thereupon, the storage backend executes the *Direct Memory Access* (DMA), thus copying the IO data to the disk. The CVM-IO stack then notifies the guest of the IO completion, which consists of the exact inversion of Steps ② and ③, thus completing the IO execution.

## 3.2 Threat Model

As CVM-IO is a CVM storage stack, we consider the Intel TDX and AMD SEV-SNP-informed CVM threat model, while focusing on storage IO-specific attack vectors [Linb]. This threat

model describes a privileged guest-external attacker, which has control over the host OS and the hypervisor and limited physical hardware access. In particular, we assume the adversary to have full control over the guest-used persistent storage medium.

Moreover, we consider all host components untrusted and under the adversary's control, which are not in the CVM's *Trusted Computing Base* (TCB). The TCB thereby comprises the entire CVM and CVM-enabling software and hardware. Such CVM-enabling components include a CVM security manager, the host platform, and the host firmware. Finally, we disregard denial-of-service as well as side-channel attacks.

We explicitly state these goals with regard to CVMs as follows:

- *Confidentiality*: the host may not view the guest IO data.
- *Authenticity*: the guest must detect if the guest itself has written the data it reads.
- *Integrity*: the guest must detect manipulation of stored data.
- *Freshness*: the guest must detect the rollback of previously stored data.

While CVM-IO offers these protection guarantees in the system's most strict configuration, we also enable CVM-IO to relax the protection guarantees for increased storage IO performance. In particular, CVM-IO may disable select protection guarantees entirely. CVM-IO may disable confidentiality, authenticity and integrity, or freshness, or a combination of these protection guarantees. Moreover, CVM-IO allows for relaxed freshness, as strictly upholding this protection guarantee incurs a significant performance loss. For instance, CVM-IO allows for transactional freshness, which persists a user-defined storage IO transaction's written data upon completion.

## 3.3 Design Challenges and Key Ideas

We present three challenges that we address when designing the CVM-IO storage stack. These challenges, thereby, all apply to the final design of this work.

**#1 Enable CVM-backed freshness protection of IO data.** To prevent rollback attacks on written data, CVM-IO must provide mechanisms to ensure freshness of IO data. However, as CVM-IO expects a privileged host adversary (Section 3.2), CVM-IO must root the freshness protection in the CVM. Moreover, CVM-IO must allow for relaxed freshness, as the complete freshness of stored data incurs high performance costs.

To achieve freshness of IO data, we store freshness-providing metadata in a CVM-backed storage backend, which is separate from the host storage backend. Nevertheless, storing additional metadata for every write request in a CVM-backed storage backend incurs high latency until CVM-IO completes the entire IO request.

We address this issue by allowing variable metadata flushes. In particular, we enable freshness metadata transactions. During a transaction, CVM-IO caches metadata storage requests. CVM-IO then commits a transaction by persisting the freshness metadata. Therefore, the storage performance of CVM-IO is not affected by CVM-IO's freshness protection, albeit with more relaxed freshness guarantees.

**#2 Avoid the extra bounce buffer copy.** As we describe in Section 2.2, the CVM memory protection architecture requires an extra copy of IO data to the bounce buffer. The CVM requires this copy to allow the backend device to DMA this data. A design that attempts to bypass this copy must respect the CVM memory protection, which prohibits direct host access to CVM memory.

CVM-IO faces this challenge by reusing a data transfer which the stack requires to enforce the protection guarantees (Section 3.2). As we describe in Section 3.1, we use `dm-crypt`'s encryption step to bypass the bounce buffer. Instead of encrypting to private CVM memory and copying the data to the bounce buffer, we encrypt directly to a buffer in shared memory. Hence, the backend device can directly access the buffer in shared memory, thus removing the need for the extra copy.

**#3 Enable guest- and host side polling in untrusted memory.** To avoid expensive CVM `VM_EXIT`s (Section 2.2), CVM-IO's storage stack uses guest- and host side polling of shared storage request queues. Further, to enable polling with a high-performance storage backend, VMs require a shared memory backend of hugepages. The host must provide these hugepages, and pass the pages to the guest via the hypervisor. However, as VMs do not allocate these pages, CVMs cannot mark them as private or shared in their page tables. As such, CVMs do not support shared hugepages, thus preventing CVMs from using polling-based storage IO with a high performance storage IO backend.

CVM-IO does not address this design challenge. However, the stack will do so in future iterations, and we nevertheless include the polling-based IO in CVM-IO's design.

Nevertheless, we would attempt to map the huge pages into the CVM shared memory to solve this challenge without informing the VM page tables. The VM could then use these designated shared pages to perform polling.

# 4 Design and Implementation

In this chapter, we detail the design and implementation of the CVM-IO storage stack around the design challenges of Section 3.3. We thereby illustrate the design and the implementation side-by-side, as we base both elements on the Linux kernel v6.7.0-rc7 [Lina] targeting x86_64 with QEMU/KVM [Bel05]. We describe the main components comprising CVM-IO: the CVM-backed flexible protection configuration (Section 4.1), the zero-copy bounce buffer bypass (Section 4.2), and finally, the CVM-adapted polling approach (Section 4.3).

For this thesis, we only implement the zero-copy mechanism of Section 4.2 with `dm-crypt` and guest-side polling via VirtIO-blk. We mark and describe the completed implementation via the underlined keyword Implementation. Implementing the complete CVM-backed storage stack protection, specifically integrity, authenticity, freshness, and host-side polling via SPDK, is future work (Chapter 8).

## 4.1 CVM-backed Storage Stack Protection

As we describe in Chapter 3, CVM-IO must provide flexible and advanced CVM-backed protection guarantees to IO data while being transparent to the CVM-contained applications. CVM-IO meets these requirements by patching the kernel-contained authenticated disk encryption `dm-crypt` and `dm-integrity` mechanisms. We give an overview of the protection goals along with the corresponding mechanisms and fulfilling CVM-IO components in Table 4.1.

**Transparency.** CVM-IO fulfills the design goal of transparency (Section 3.1) by using the kernel-contained device-mapper subsystem, thus being invisible to the CVM-contained application. As we show in Figure 4.1, the device-mapper subsystem integrates transparently into the kernel block layer, acting as a virtual block device. The application thereby issues writes via the kernel syscall API (Step ①) to the *Virtual File System* (VFS). The VFS then represents the write request via the kernel `bio` data structure, which the VFS submits to the block layer (Step ③). Finally, the block layer forwards the request to the device-mapper subsystem, which the block layer views as a generic block device. After processing the `bio` and applying the protection guarantees, the device-mapper resubmits the resulting protected `bios` to the block layer for further IO processing [Lina].

Therefore, CVM-IO integrates seamlessly into the overall system. Not only is the CVM-contained application agnostic of the received protection, but the other kernel components also do not need to know of the protection mechanisms' presence, meeting the transparency design goal.

**IO Data Protection.** To meet the design goal of security (Section 3.1), CVM-IO provides flexible CVM-backed protection guarantees (Section 3.2) via the aforementioned device-mapper layer.

| Property | Method | Component |
|---|---|---|
| Confidentiality | Encryption | `dm-crypt` |
| Authenticity | AEAD | `dm-integrity` |
| Integrity | AEAD | `dm-integrity` |
| Freshness | Secure storage of sector:AT mapping | CVM-backed storage |

Table 4.1: Protection properties along with the method to achieve the property and the fulfilling component. CVM-IO's `dm-integrity` employs AES-GCM [Dwo07], which is an *Authenticated Encryption with Additional Data* (AEAD) method. AEAD provides authenticity and integrity on top of confidentiality of data, by providing integrity metadata in the *Associated Data* (AD). The freshness providing *Authentication Tag* (AT) is a hash over the integrity-providing AD, which consists of the sector number and initialization vector.

We thereby present the CVM-IO patch to the device-mapper layer in Figure 4.2, which meets the design challenge of CVM-backed freshness (Section 3.3).

To achieve the protection guarantees, CVM-IO employs the cipher AES [RD01] in the *Galois/-Counter authenticated mode* (AES-GCM) [Dwo07] in the device-mapper layer, which enables AEAD. This cipher receives three inputs: a `bio`-supplied plaintext, a user-supplied password as a key, and an RNG-generated *Initialization Vector* (IV). The cipher then encrypts the plaintext (Step ①) using the key and the IV, producing the ciphertext and an *authentication tag* (AT). This AT is a hash over the key, the plaintext, as well as the *associated data* (AD), which in turn consists of a sector number and the IV.

Further, CVM-IO stores the produced data in separate bios (Step ②), which CVM-IO again stores in separate backends (Step ③). In the high-performance storage backend, CVM-IO stores the ciphertext and the AD. The other CVM-IO-used storage backend is CVM-backed, meaning the CVMs form the root of trust of this backend. CVM-IO thereby stores a pair of the authentication tag (AT) and the corresponding sector number in the CVM-backed storage backend. As such, an adversary cannot manipulate or roll back the sector number and AT pairs.

To decrypt the ciphertext, the cipher requires the user-provided key, the ciphertext, and the AD. CVM-IO then decrypts the ciphertext, thus receiving plaintext data. After decrypting the ciphertext, CVM-IO queries the CVM-backed storage backend via the sector number, thus receiving the associated AT. The storage stack then calculates the hash over the key, the plaintext, and the AD, and compares the hash's equality with the AT. If the hashes match, no adversary has manipulated or replayed the AT or the ciphertext.

To prevent high-performance overheads, CVM-IO allows for relaxing the freshness guarantee. In the strict mode of operation, which strictly upholds the freshness guarantee, CVM-IO flushes the metadata `bio` to the CVM storage backend before completing the write request (Step ③). However, in the relaxed mode, CVM-IO may buffer the metadata `bios` and flush these `bios` either asynchronously or upon a user-defined transaction. However, if the CVM crashes before writing

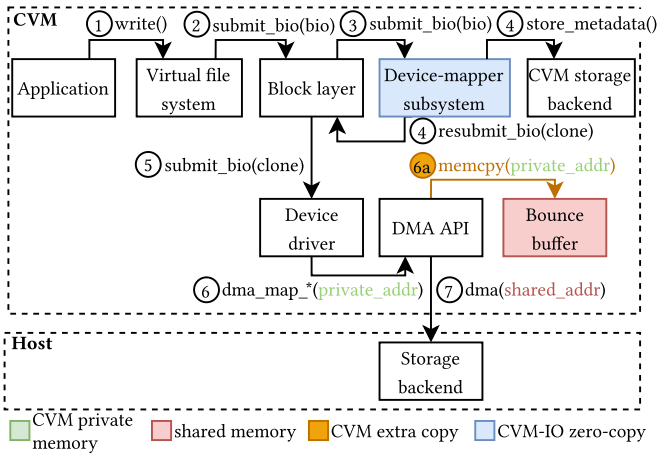



Figure 4.1: Overview of the zero-copy embedding in the VM Linux storage stack. (Orange marks the native CVM storage IO extra bounce buffer copy. Blue marks the zero-copy embedding.)

the metadata bio to the CVM storage backend, the lack of the committed metadata invalidates the corresponding encrypted data writes to the host backend. Thus, these invalidated writes violate the freshness guarantee.

Besides relaxed freshness, CVM-IO offers performance improvements by removing select protection guarantees via removing protection components (Table 4.1). While removing these properties violates the protection goals of CVM-IO (Section 3.2), scenarios exist where the user does not require all guarantees. For instance, a CVM may store votes of a ballot, where the confidentiality of the stored data is less important than persisting all cast ballots fast and with integrity, authenticity, and rollback protection.

Hence, CVM-IO provides the confidentiality guarantee via the AES-GCM-cipher encryption of the storage IO plaintext via a random IV and a user-provided key. Further, the system enables integrity and authenticity via the plaintext-, AD-, and key-derived AT. If an adversary attempts to forge or manipulate data, the input-data-derived hash will not match the AT. Finally, the separate storage of the sector-AT pair in the CVM-backed storage backend provides the freshness guarantee while allowing for performance and security trade-offs by buffering AT metadata writes. These freshness-providing mechanisms solve the Challenge #1 (Section 3.3).

## 4.2 Zero-copy Storage IO

CVM-IO addresses the design goal of performance over native CVM storage IO (Section 3.1) by avoiding the extra bounce buffer copy (Section 3.3). To avoid this copy, CVM-IO repurposes a protection-incurred existing IO data transfer in the kernel. Further, CVM-IO ensures that the premature IO data exposure via bounce buffer bypass does not break the existing IO data protection mechanisms. Finally, the design alters the guest kernel's DMA API to differentiate between CVM-IO and legacy IO data to not break non-CVM-IO IO, which requires the bounce

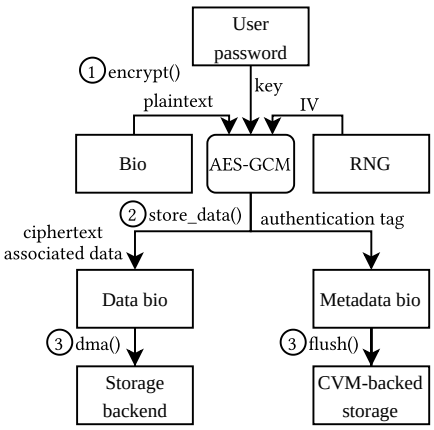



Figure 4.2: Overview of the CVM-IO storage IO data protection mechanism. The protection mechanism uses AES [RD01] in the Galois/Counter authenticated mode [Dwo07]. CVM-IO thereby allows the storage backend to store the ciphertext and associated data, while CVM-IO uses the CVM-backed storage to save the authentication tag with the corresponding sector number. The associated data consists of the *Initialization Vector* (IV) as well as the sector number. Further, the authentication tag is a hash over the ciphertext and the associated data.

buffer's use.

**Repurposing an existing Copy.** As the copy to the bounce buffer is to transfer the IO data to shared memory, we must reuse an existing IO data transfer in the guest kernel. Nevertheless, for native CVM storage IO, the only existing kernel-contained IO data transfer is the bounce buffer copy, as we show in Figure 4.1 without the blue-marked CVM-IO modification. However, we introduce IO data protection via the device-mapper subsystem in Section 4.1. As we show in Figure 4.2, this subsystem transforms the IO data plaintext to ciphertext and adds corresponding metadata. This data transformation causes a data transfer, which CVM-IO uses to avoid the bounce buffer copy.

CVM-IO avoids this copy by allocating the target data `bio`'s pages in shared memory. As such, the DMA API no longer requires the bounce buffer to share the IO data with the host storage backend, as we describe in this section's final paragraph. Hence, CVM-IO saves a copy in comparison to native CVM storage IO.

Implementation: To enable `dm-crypt` to use shared memory, we must modify the system's memory management to allocate pages in shared memory. Internally, `dm-crypt` receives write IO data, allocates a `bio` clone with a corresponding set of pages, and encrypts the IO data into the `bio` using the Linux kernel crypto API. `dm-crypt` thereby allocates the `bio`'s pages from a preallocated kernel `mempool`. This `mempool` allocates and deallocates pages via `dm-crypt`-implemented functions, which `dm-crypt` passes to the `mempool` upon initialization.

CVM-IO then modifies these allocation functions to achieve zero-copy, using platform-independent kernel APIs to convert page types to either private or shared. As such, CVM-IO's implementation

applies to all kernel-supported CVMs. In particular, CVM-IO converts all mempool-allocated functions to shared pages after the pages' allocation. As deallocating shared pages causes a memory management error, CVM-IO also reconverts pages to private memory before deallocation. Therefore, when `dm-crypt` encrypts the IO data, the subsystem encrypts from the incoming `bio` to the shared-memory `bio` clone, thus completing the first step of the zero-copy implementation.

**Considering Protection Vulnerabilities.** Because an adversary has full control over the IO data in shared memory, CVM-IO must protect all IO data before transferring this data to the shared memory. CVM-IO meets this criterion by transferring only the encrypted data and the integrity metadata to the shared memory. Moreover, as we note in Section 4.1, CVM-IO ensures to not use the produced ciphertext to calculate the AT.

As such, an adversary cannot influence the AT's inputs, possibly compromising integrity or authenticity guarantees. Finally, CVM-IO never exposes the AT to shared memory. Hence, the protection of the AT lies fully in the CVM, preventing malicious manipulation of the AT.

**Differentiating between Legacy and CVM-IO IO.** Upon placing the IO data in shared memory, CVM-IO must differentiate between CVM-IO data and legacy IO data to uphold the transparency design goal (Section 3.1). CVM-IO must continue to copy legacy IO data to the bounce buffer to share this data with the host while skipping the bounce buffer for CVM-IO data. Hence, we show in Figure 4.1 how CVM-IO enables transparency in the storage stack. After the previously described Step ④ (Section 4.1), the storage stack submits either a legacy or a CVM-IO-modified `bio` to the target virtual block device (Step ⑤).

The corresponding device driver then calls the DMA API to map the `bio`'s contained page to DMA-able memory (Step ⑥). In native CVM IO, the DMA API performs the copy to the bounce buffer to allow host DMA access to the IO data (Step ⑥a). However, CVM-IO modifies the DMA API by checking if the to-DMA memory is already marked as shared. If so, CVM-IO does not execute the copy, enabling zero-copy storage IO. If not, CVM-IO executes the copy, allowing the execution of legacy IO.

Implementation: We implement the IO separation in the kernel's DMA API. In native CVM storage IO, the DMA API copies all IO data through the bounce buffer. Hence, CVM-IO adds a check to the DMA API to differentiate between private and shared pages. Private pages go through the bounce buffer, while shared pages do not. CVM-IO thereby identifies private and shared pages via the pages' page table entry. In particular, CVM-IO retrieves a bitmask via a CVM-platform independent API to assert if the private bit in the page table entry is set. If so, the page is in private memory; otherwise, the page is not.

## 4.3 CVM-adapted Polling

Again, to fulfill the design goal of performance (Section 3.1), CVM-IO avoids `VM_EXIT`s (Section 2.2) via polling. SPDK [Yan+17] already enables host-side polling for native VMs, and QEMU VirtIO-blk enables guest-side polling for native and confidential VMs. However, to enable polling with SPDK for CVMs, the CVM must share host-allocated hugepages with the host. Therein, the

guest and the host share poll queues. Hence, the CVM must map these hugepages into the guest virtual shared memory.

Again, polling introduces a trade-off between performance and CPU utilization. While introducing polling reduces IO latency, polling requires a spinning thread. However, for CVMs, the latency improvements for polling are more significant than for native VMs due to CVMs' increased CVM `VM_EXIT` costs.

We end the discussion here because we have not implemented host-side polling with SPDK in this thesis. We shall include host- and polling with SPDK in future work.

Implementation: We partially enable polling-based storage IO via polling-based VirtIO-blk. The guest thereby performs guest-side completion polling via shared designated poll queues.

# 5 Evaluation

In this chapter, we thoroughly evaluate CVM storage IO and CVM-IO. We thereby aim to understand CVM storage IO's overheads and, specifically, which factors influence CVM-based storage IO in which manner. Further, we investigate CVM-IO's influence on storage IO performance. Moreover, we perform our evaluation with the two currently available CVM technologies, AMD SEV-SNP and Intel TDX.

**Experimental setup.** ***AMD SEV-SNP specific setup:*** For the AMD SEV-SNP setup, we perform the experiments on a machine equipped with an AMD EPYC 9334 CPU [Adva] running at 2.7 GHz (two sockets, 32 physical cores each with hyperthreading disabled) and 752 GiB of DDR4 memory. Further, the host BIOS is Dell Inc. BIOS 1.5.8 [Del]. The host-equipped NVMe SSD is the Samsung PCIe Gen4-enabled PM1735 1.6TB SSD [Sam]. The attached PCIe link capability and link status are 16 GT/s by 8 x width. We precondition and format the SSD as defined in [JS].

The host OS is NixOS 23.11 (AMD Linux 6.6.0-rc1 `5a170ce` [Advb]), and we use AMD QEMU 8.0.0 `fe4c9e8` [Advd] with AMD OVMF `80318fc` [Advc]. The native VM, the native CVM, as well as the CVM-IO CVM use the CVM-IO-patched Linux 6.7.0-rc7 `48b4fe67` [Sch]. The CVM is thereby the AMD SEV-SNP [Kap20].

***Intel TDX specific setup***: For the Intel TDX CVM, we use a machine with an Intel Xeon Platinum 8480CTDX CPU [Intc] running at 2.0GHz (two sockets, 56 cores each with hyperthreading disabled) and one TiB of DDR4 memory. Moreover, the BIOS is American Megatrends International, LLC. BIOS version 3A15.TEL3P1 [AMI]. The host-attached NVMe SSD is the Intel SSD DC PC3600 800GB [Intb]. We also precondition and format the SSD as defined in [JS].

In this case, the host's as well as the VM's and CVM's OS is Ubuntu 23.10 (Intel Linux 6.5.0-1003-intel-opt [Kobc]), and we use Intel QEMU 8.0.4 `9498d82` [Kobb] and Intel OVMF `b477996` [Koba]. Further, the CVM is the Intel TDX [Int21].

***Generic setup***: The hypervisor sets if a VM executes as a CVM or not. Further, we compile the native- and the CVM-IO CVM with a different kernel configuration parameter set to enable or disable CVM-IO. We assign the guest eight vCPUs and 16 GiB of memory for all experiments. We also employ pinning of physical cores to vCPUs, whereby the pinned cores reside on the same NUMA node as the SSD.

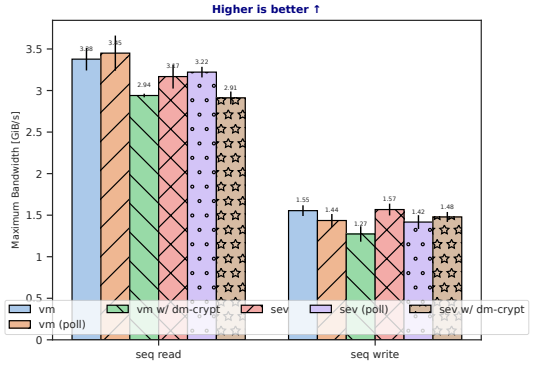
We overview the entire storage stack configuration from guest to host in Table 5.1. In the host- and guest OS, we use `io_uring` to submit and complete IO requests via polling between the user- and kernel space. `io_uring` enables zero-copy between the user- and kernel space and does not require an extra context switch between the two OS layers.QEMU further disables the host-side cache to enable direct IO.

| | Request type | User ↔ kernel | Kernel driver ↔ storage backend |
|---|---|---|---|
| **Guest** | Submission | io_uring | virtio-blk |
| | | polling | MMIO |
| | Completion | io_uring | virtio-blk |
| | | polling | interrupts |
| **Host** | Submission | io_uring | NVMe SSD |
| | | polling | MMIO |
| | Completion | io_uring | NVMe driver |
| | | polling | polling |

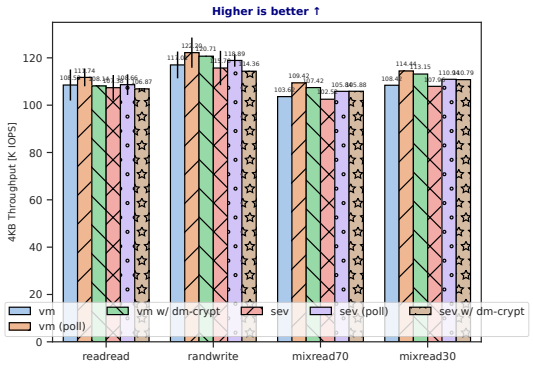
Table 5.1: Overview of the complete storage stack domain-crossing storage request mechanisms. User- and kernel space submit and complete IO requests via `io_uring` [Axbb]. Guest and host execute IO via VirtIO-blk [Rus08]. Finally, host and the SSD perform IO via NVMe [NVM].

| Test metrics | Test cases | Fio configuration (bs, rw, iodepth, numjobs) |
|---|---|---|
| Bandwidth | Read | (128K, read, 128, 1) |
| | Write | (128K, write, 128, 1) |
| IOPS | Randread | (4K, randread, 32, 4) |
| | Mixread | (4K, randread 70%, 32, 4) |
| | Mixwrite | (4K, randwrite 30%, 32, 4) |
| | Randwrite | (4K, randwrite, 32, 4) |
| Average Latency | Randread | (4K, randread, 1, 1) |
| | Randwrite | (4K, randwrite, 1, 1) |
| | Read | (4K, read, 1, 1) |
| | Write | (4K, write, 1, 1) |

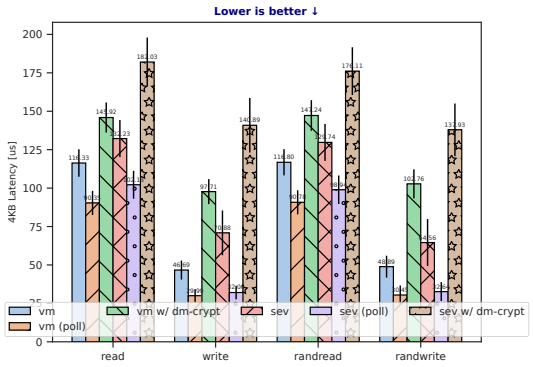
Table 5.2: Fio test cases.




(a) Bandwith comparison.

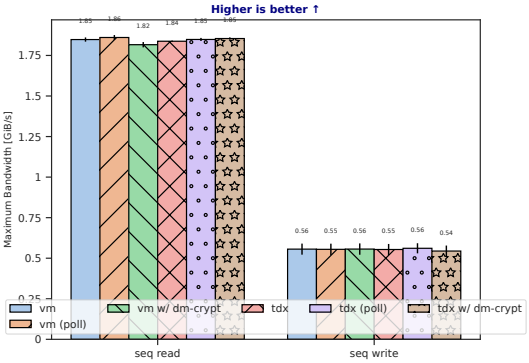



(b) IOPS comparison.

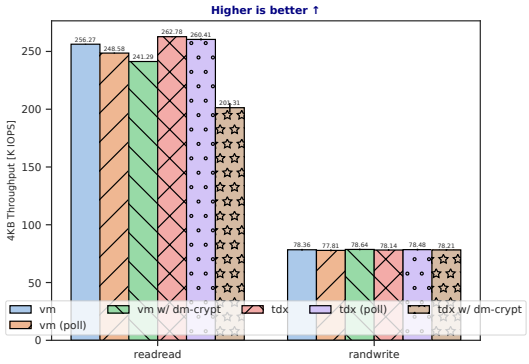



(c) Latency comparison.

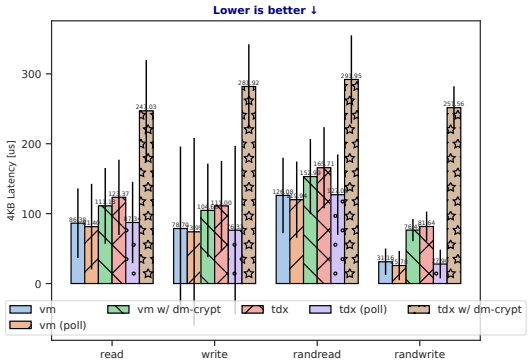
Figure 5.1: Storage IO performance comparison between a native VM and an AMD SEV-SNP CVM on the same platform with various configurations. We list the benchmark configurations in Table 5.2 and describe the experimental setup in Chapter 5. In the plot, *vm* is the native VM with a host/kernel `io_uring` backend without polling enabled, *vm (poll)* is with polling enabled, and *vm w/ dmcrypt* is *vm* with `dm-crypt` enabled in the guest. The *sev* configurations mirror the *vm* configurations, with the difference of the VM being an AMD SEV-SNP CVM.




(a) Bandwith comparison.

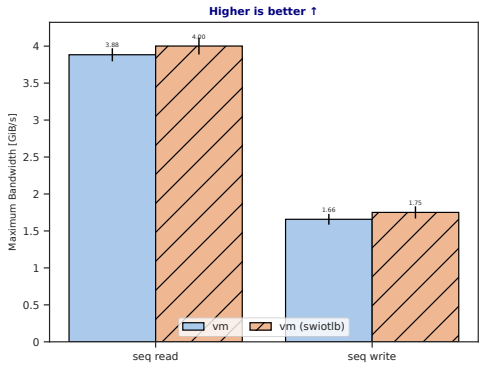



(b) IOPS comparison.

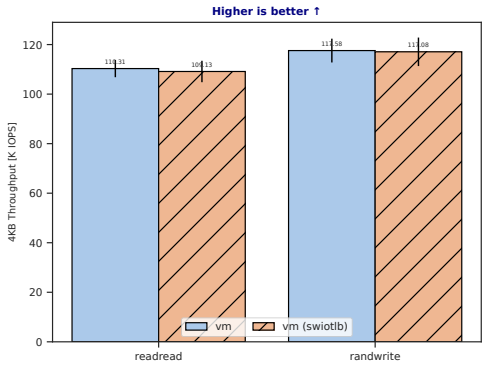



(c) Latency comparison.

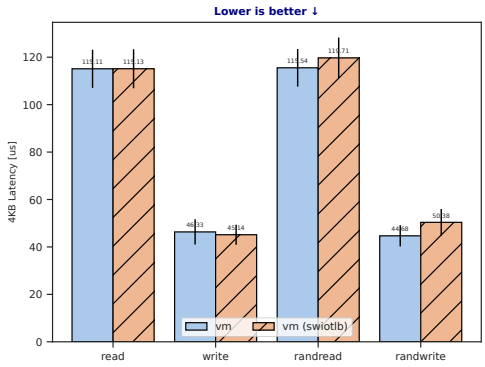
Figure 5.2: Storage IO performance comparison between a native VM and an Intel TDX CVM on the same platform with various configurations. *vm* shares the same configuration as Figure 5.1, with the difference of using the Intel TDX platform of Chapter 5. *tdx* share the same configurations as *vm*, with the difference of running in an Intel TDX CVM.




(a) Bandwith comparison.

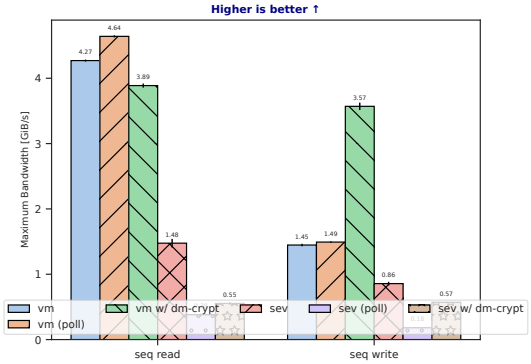



(b) IOPS comparison.

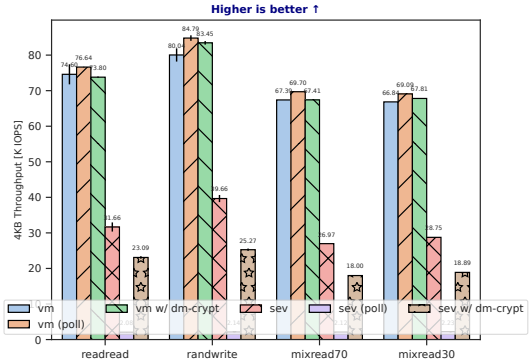



(c) Latency comparison.

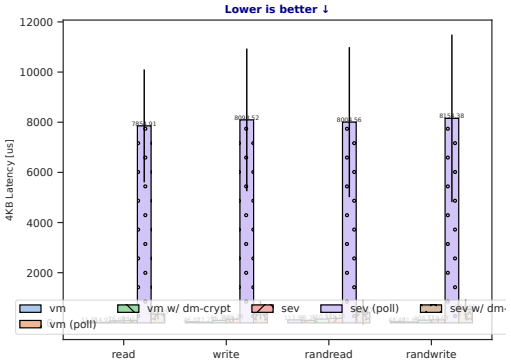
Figure 5.3: Storage IO performance comparison between a native VM and a native VM with the bounce buffer forced on and host cache disabled. We list the benchmark configurations in Table 5.2 and describe the experimental setup in Chapter 5. *vm* is the same configuration as in Figure 5.1, and *vm (swiotlb)* has the same configuration as *vm*, with the difference of the bounce buffer forced on. We execute these benchmarks on the AMD SEV-SNP platform.



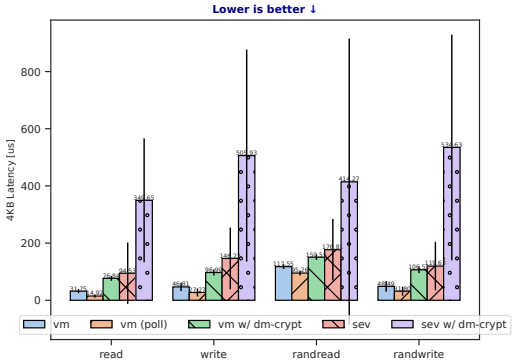

(a) Bandwith comparison.




(b) IOPS comparison.

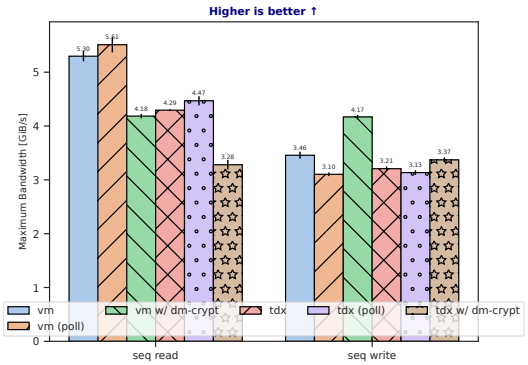



(c) Latency comparison.

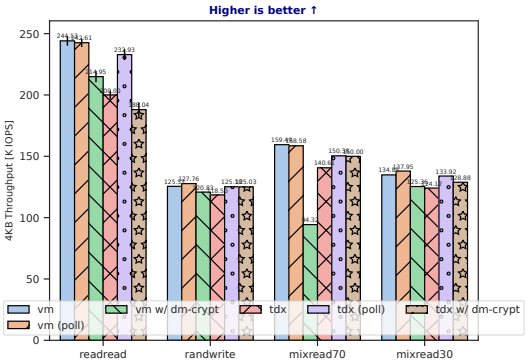



(d) Latency comparison with sev (poll) ommitted.

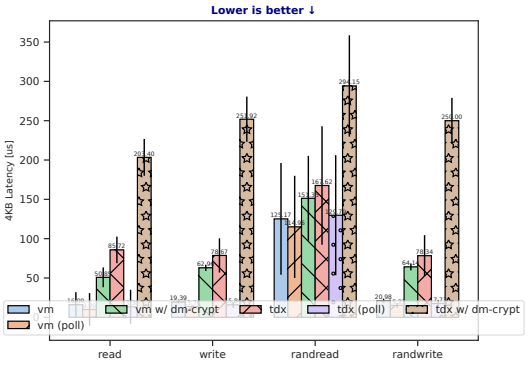
Figure 5.4: Storage IO performance comparison with host-side cache enabled between a native VM and an AMD SEV-SNP CVM on the same platform with various configurations. The configurations of *vm* and *sev* are as in Figure 5.1, with the difference of host-side cache being enabled.




(a) Bandwith comparison.




(b) IOPS comparison.




(c) Latency comparison.

Figure 5.5: Storage IO performance comparison with host-side cache enabled between a native VM and an Intel TDX CVM on the same platform with various configurations. The configurations of *vm* and *tdx* are as in Figure 5.2, with the difference of host-side cache being enabled.

Between the guest kernel driver and the host storage backend, we use VirtIO-blk in its interrupt- and polling-based completion configurations. For the polling configuration, we set four queues, two of which are designated poll queues.

Finally, the host device driver is the kernel NVMe driver, whereas this driver does polling IO with the NVMe SSD. Both the NVMe driver and the NVMe SSD do submission and completion polling.

For the BIOS, we set the parameters as recommended by [JS]: Turbo mode is disabled, hyper-threading is disabled, memory frequency is set to max performance, CPU Power management is set to maximum performance, P states are disabled, C states are disabled, power profile is set to maximum IO performance mode, determinism slider is set to performance, and node interleaving is disabled.

**Base Methodology.** We compare the storage IO of various VMs via the guest-contained synthetic storage IO benchmarking tool fio 3.36 [Axba] (`direct=1, thread=1, ioengine=io_uring, group_reporting=1`). Further, we test for average latency, bandwidth, and *IO Operations Per Second* (IOPS), as we detail in Table 5.2. We run the benchmarks `time_based`, with a `ramptime` of one minute, and a `runtime` of ten minutes, as recommended by [JS].

## 5.1 VM and CVM Storage IO Performance Differences

**RQ1.** *What is the storage IO performance difference between a native VM and a native CVM?*

**Methodology.** For each CVM testbed, we run the fio benchmark in two different configurations. First, we benchmark the native VM as the baseline. Second, we benchmark the CVM to measure the total CVM overhead. We obtain our results by comparing the CVM overhead to the native VM baseline.

**Results.** The result plots in Figure 5.1 and Figure 5.2 show minimal overhead for bandwidth and IOPS, yet significant overhead for average latency for AMD SEV-SNP and Intel TDX, respectively. We mark the baseline with "vm" and the CVM result with "sev" and "tdx" for AMD SEV-SNP and Intel TDX respectively. For AMD SEV-SNP, Figure 5.1c (vm ↔ sev) presents a 14% and 52% overhead for read and write latency of the CVM compared to the native VM, respectively. For randread and randwrite latency, the CVM overhead is 11% and 32%, respectively. For Intel TDX, Figure 5.2 (vm ↔ tdx) shows a 43% and 31% overhead for read and write latency, respectively. Again, for randread and randwrite latency, the CVM overhead here is 31% and even 162%.

**RQ1 takeaway:** CVMs incur little to no overhead regarding bandwidth and IOPS, yet cause high latency overheads.

**RQ2.** *How much of the CVM overhead does the bounce buffer account for?*

**Methodology.** To calculate the bounce buffer overhead, we measure a native VM with the bounce buffer forced on. We then compare it with a native VM, which does not use the bounce buffer. The resulting overhead is that of the bounce buffer, which we then contrast to the performance difference between a CVM and a VM of RQ1.

**Result.** The results in Figure 5.3 (vm ↔ vm (swiotlb)) show little to no overhead. Although we omit the corresponding graphs, enabling host-side caching and disabling asynchronous IO do not change the overhead difference result. As such, the partial implementation of CVM-IO with this storage IO configuration cannot improve the storage IO performance.

**RQ2 takeaway:** The CVM bounce buffer produces little to no overhead. As such, zero-copy storage IO is not significant for storage IO performance optimization.

**RQ3.** *How much of the CVM overhead does interrupt-based IO account for?*

**Methodology.** To capture the overhead of CVM interrupt-based IO, we compare this IO configuration as a baseline with CVM polling-based IO. Moreover, we contrast the CVM interrupt-based overhead with the native VM interrupt-based overhead, which we measure via the same method. In the context of this thesis, we only use guest-side polling, as no mainline paravirtual storage IO engines support guest- and host-side polling.

**Result.** As the plots in Figure 5.1 and Figure 5.2 show, polling does not affect bandwidth or IOPS significantly. Polling does, however, noticeably impact latency. For the native VM case (Figure 5.1 vm ↔ vm (poll)), the latency overhead of interrupt-based IO for read and write benchmarks are 29% and 56%, respectively. For randread and randwrite, the plots show overheads of 29% and 61%, respectively. Further, for CVMs, we find the overheads to be significantly larger. For read and write (Figure 5.1 sev ↔ sev (poll)), the latency overheads are 29% and 121%, whereas for randread and randwrite, we measure 31% and 98%.

Notably, the overhead of the CVM polling configuration compared to the native polling configuration is minimal. For read and write latency ((Figure 5.1 vm (poll) ↔ sev (poll))), the CVM latency overheads are 13% and 7%, and for randread and randwrite, 9% and 7%, respectively.

We attribute the increased latency reduction of polling for CVMs to the high cost of CVM `VM_EXIT`s and context switches. Host-side polling thereby reduces `VM_EXIT`s, removing the CVM-architecture induced `VM_EXIT` overhead.

Notably, using polling-based IO instead of interrupt-based IO comes with a trade-off. While interrupt-based IO is more latency intensive due to expensive interrupt handling and context switches, polling occupies a thread and thus costs CPU resources. Hence, interrupt-based IO may still be the preferable IO configuration for use cases where latency is not critical.

Finally, we receive inconsistent results for polling with host-cache enabled for SEV-SNP (Figure 5.4) and TDX (Figure 5.5). Whereas SEV-SNP, enabling polling with host-cache imposes superb overheads in all evaluation metrics, for TDX the results mirror those of host-cache disabled. Investigating this inconsistency is future work.

**RQ3 takeaway:** CVM polling-based storage IO significantly reduces the latency overhead of interrupt-based storage IO, more so than the latency reduction of native VM polling-based storage IO. Specifically, for native VMs, polling improves latency performance by up to 55%, whereas for CVMs, the latency improvements of polling are up to 121%. Notably, CVM polling-based IO allows CVM storage IO to have latency to native VM polling-based IO.

**RQ4.** *How does host-side storage-IO caching influence the storage-IO performance of CVMs?*

**Methodology.** We set the QEMU blockdevice option `cache.direct=off` to compute the host-side caching storage-IO performance. Importantly, QEMU enables host-side caching per default. We then contrast the native VM to the CVM host-side caching results. Finally, we contrast these results to the baseline results of RQ1.

**Results.** As our results indicate in Figure 5.4 and Figure 5.5 for SEV-SNP and TDX, enabling host-cache causes performance overheads across all benchmarks. For SEV (vm ↔ sev), bandwidth read and write have an overhead of 189% and 69%, respectively. For IOPS, the overheads for randread, randwrite, mixread with 70% reads and mixread with 30% reads are 136%, 102%, 162%, and 132%, respectively. Finally, for latency, for read, write, randread, and randwrite, the overheads are 198%, 107%, 50%, and 147%, respectively.

As QEMU enables host-caching by default, CVM users may assume these significant overheads to state the CVM overhead baseline. Investigating the reason for this overhead is future work.

However, for CVM-IO, we disable host-side caching regardless due to `dm-crypt`. Host-side caching is beneficial if multiple VMs access the same disk data. However, VMs using `dm-crypt` typically have individual keys and individually encrypted data. As such, the VMs do not share any data, thereby undermining the use of host-side caching.

**RQ4 takeaway:** Host-side caching causes significant CVM-architecture-induced overheads in all storage-IO performance metrics.

**RQ5.** *How does the use of the CVM-IO protection mechanism* `dm-crypt` *influence storage IO performance in CVMs?*

**Methodology.** To measure the performance impact of `dm-crypt` in CVMs, we measure a baseline with a native `dm-crypt`-enabled VM per the corresponding CVM platform. We then measure the corresponding `dm-crypt`-enabled CVM. Further, we repeat these measurements with and without host-side caching.

**Result.** Disregarding certain outliers, for direct storage IO without host-side cache `dm-crypt` does not significantly impact IOPS or bandwidth for CVMs (Figure 5.1 and Figure 5.2). Concerning latency, `dm-crypt` demonstrates an overhead compared to the baseline without `dm-crypt`. For the SEV platform (vm ↔ vm w/ `dm-crypt`), a native VM with `dm-crypt` compared to a native VM without `dm-crypt` has a latency overhead of 56%, 109%, 26%, and 110%, for read, write, randread, and randwrite, respectively. In the SEV-SNP case (sev ↔ sev w/ `dm-crypt`), a CVM

with dm-crypt compared to a CVM without dm-crypt exhibits overheads of 38%, 99%, 36%, and 114%, respectively. Finally, the CVM dm-crypt configuration compared to the baseline VM dm-crypt configuration (vm w/ dm-crypt ↔ sev w/ dm-crypt) exhibits overheads of 25%, 44%, 20%, and 34%, for read, write, randread, and randwrite, respectively. For TDX, we see similar results (Figure 5.2c).

Although the CVM overheads have less overhead compared to their corresponding baseline without dm-crypt, the absolute differences between the baseline's and the dm-crypt configurations are minimal. Hence, the native VM dm-crypt overhead is higher, as the same dm-crypt overhead has a proportionally higher impact on the native latency. As such, we see similar overheads between native VM dm-crypt and CVM dm-crypt as native VM's and CVM's without dm-crypt, as we discuss in RQ1.

IOPS and bandwidth overheads are also minimal for the host-cache enabled cases in Figure 5.4 and Figure 5.5. However, latency overheads differ in intensity from the direct case. For the TDX platform (vm ↔ vm (dmcrypt)), for the native case, we observe dm-crypt latency overheads of 216%, 225%, 21%, and 206%, for read, write, randread, and randwrite, respectively. Further, for the TDX CVM, we see dm-crypt latency overheads of 137%, 220%, 81%, and 219%. Finally, when we compare the overhead of native dm-crypt to CVM dm-crypt, we see overheads of 300%, 300%, 94%, and 290%, for read, write, randread, and randwrite, respectively. For SEV, we also observe similar results.

In contrast to the direct dm-crypt benchmarks, the host-cache CVM dm-crypt benchmarks exhibit significantly higher latency overhead. This difference is visible in the relative overhead and the absolute difference of the dm-crypt benchmarks to their corresponding baseline.

**RQ5 takeaway:** dm-crypt incurs minimal bandwidth, IOPS overheads, and significant latency overheads. With the host-side cache disabled, the absolute latency overheads of dm-crypt for native and confidential VMs are roughly equal. For native VMs compared to native VMs with dm-crypt, these overheads are up to 110%, whereas for CVMs compared to CVMs with dm-crypt, the overheads are up to 114%. CVMs with dm-crypt compared to native VMs with dm-crypt exhibit overheads of up to 44%. Nevertheless, with host-side cache enabled, the overheads of dm-crypt for CVMs are significantly higher than for native VMs. In this case, native VM overheads of dm-crypt are up to 220%, whereas for CVMs, the overheads are up to 220%. Finally, with host-side page cache enabled, CVMs with dm-crypt compared to native VMs with dm-crypt show an overhead of up to 300%.

## 5.2 CVM-IO Performance Improvement

**RQ6.** *Does CVM-IO improve storage IO performance?*

**Result.** As Figure 5.3 shows the bounce buffer to impose no storage IO overhead (vm ↔ vm (swiotlb)), the partial implementation of CVM-IO does no improve storage IO performance with this storage IO configuration. This inability to improve performance is due to CVM-IO's design, which targets bridging the bounce buffer. This result is a surprise, as related work for CVM network IO [Li+23] finds the bounce buffer to significantly impact the performance of CVM

network IO. Further, CVM platform manufacturers claim the bounce buffer negatively impacts storage IO performance [Intd]. However, we could not reproduce this claim.

**RQ6 takeaway:** The partial CVM-IO implementation does not improve storage IO performance due to a minimal bounce buffer overhead.

# 6 Related Work

**Improving CVM network IO.** Bifrost [Li+23] analyzes and optimizes the CVM IO overhead for networking. Bifrost thereby identifies similar overheads for network IO, which include the bounce buffer and the `VM_EXIT` overhead. Further, cvm-io and Bifrost apply the protection-anchored zero-copy technique to bridge the bounce buffer. Whereas cvm-io uses `dm-crypt` to write encrypted data into shared memory, Bifrost uses TLS.

However, in several key areas, cvm-io differs from Bifrost. For one, as Bifrost focuses on network IO, Bifrost does not have to face persistent storage-specific attacks, specifically, freshness for persistent storage. In-use protection suffices to ensure freshness guarantees given by TLS, whereas cvm-io requires CVM-backed storage to provide freshness guarantees. Further, the implementation of Bifrost is virtio-net [Rus08] specific, whereas cvm-io's implementation is storage device driver generic. Moreover, Bifrost mitigates `VM_EXIT`s via *Posted Interrupts* (PI), an Intel VT-d [Abr+06] specific feature, allowing its implementation to only use this feature in Intel TDX CVMs. In contrast, cvm-io uses polling to avoid `VM_EXIT`s, which does not require a CPU-specific instruction set extension. As such, cvm-io is available for all CVMs, while Bifrost is not.

**TEE-backed persistant storage protection.** Speicher [Bai+19] presents a secure key-value storage interface, which provides confidentiality and integrity properties in addition to TEE-backed data freshness. As cvm-io, both systems provide persistent storage with TEE-backed protection guarantees, including freshness. Further, both systems asynchronously flush protection-providing metadata via a TEE mechanism, bridging the latencies.

However, Speicher extends a process-based TEE, namely, Intel SGX [Inta], whereas cvm-io extends all currently available VM-based TEEs. Moreover, Speicher provides rollback protection via a trusted counter, while cvm-io uses a sector-authentication tag mapping. Finally, Speicher executes within a process-based TEE and requires explicit calls via Speicher's key-value interface. Contrary to this approach, cvm-io integrates into the Linux kernel of a CVM, thus requiring no explicit calls to use and no code changes to existing applications.

**Improving TEE storage IO performance.** For process-based TEEs, several IO performance-improving works with varying approaches exist [Tha+21; Arn+16; Pri+19]. rkt-io [Tha+21] provides a high-performance direct IO kernel-bypass stack for TEEs, which offers full Linux ABI compatibility. Scone and SGX-LKL are shielded execution frameworks which use asynchronous IO calls, relying on the host OS to handle IO operations via host IO threads.

While cvm-io and the processes-based TEE stacks apply to different TEE types, the stacks face a similar issue: context switching. To perform IO, process-based TEEs must execute a *world switch* when switching from the trusted to the untrusted domain before issuing a host syscall.

CVMs, on the other hand, must submit IO via MMIO and complete IO via host-side interrupts. The CVM must perform `VM_EXIT`s to handle these completion interrupts via the hypervisor. In both cases, the TEE-enabling technologies must execute micro-architectural security-associated sanitizations, adding further overheads.

CVM-IO and RKT-IO employ polling to submit and complete IO requests. Moreover, both use a separate region outside the TEE trust domain for host DMA; RKT-IO uses an explicitly mapped untrusted host region, whereas CVM-IO uses host-shared memory. However, CVM-IO performs paravirtual IO, while RKT-IO does not. As such, CVM-IO's IO must traverse through two storage stacks and use the VM's virtual device driver to communicate with the host virtual device. In particular, CVM-IO polls the shared virtual storage queues, while RKT-IO polls hardware queues. Further, CVM-IO encrypts directly into the shared memory, while RKT-IO performs an explicit copy. Finally, CVM-IO provides integrity and freshness guarantees, while RKT-IO only uses `dm-crypt`, thus only offering confidentiality of storage IO data.

# 7 Conclusion

The lack of advanced protection guarantees and significant overheads for storage IO overheads hinder the real-world use of CVMs. This thesis presents CVM-IO, a novel storage IO stack design tailored to CVMs, which thereby provides strong IO data protection, including freshness, and avoids CVM-architecture-imposed overheads. Further, this thesis provides an in-depth analysis of CVM storage IO performance. We partially implement our prototype based on the Linux kernel, finding its partial implementation cannot provide performance gain due to the lack of CVM bounce buffer overheads. Further, CVMs exhibit only latency overheads with host-cache disabled and high overheads in bandwidth, IOPS, and latency with host-cache enabled. Finally, we find the CVM-forced bounce buffer to cause no overheads. **Artifact: CVM-IO and the reproducible evaluation setup are avilable at `https://github.com/TUM-DSE/CVM_eval`.**

# 8 Future Work

The investigation and improvement of CVM storage IO are novel, and as such, this thesis uncovers many important directions of future work that go beyond this thesis's scope. These directions comprise analysis of CVM storage IO (Section 8.1) and the complete implementation of CVM-IO (Section 8.2).

## 8.1 Analysis of CVM Storage IO

As we note throughout Chapter 5, we observe several CVM storage IO performance variations for which we have no explanation. We list these observations and how we expect to investigate these performance variations.

**Difference in bounce buffer overhead findings.** Contrary to the statements of related work [Li+23; Intd; YG23], this thesis does not find bounce-buffer-induced storage IO performance overheads (Chapter 5, Figure 5.3). This thesis attempted to identify this overhead with host-cache enabled and disabled and with and without asynchronous IO. Future work is to reproduce the findings of related work and investigate under which configuration the bounce buffer impacts IO performance.

**Impact of CVM `VM_EXIT`s.** As the interrupt- and polling-based storage IO evaluation (Chapter 5, Figure 5.1) hints at, CVM `VM_EXIT`s are presumably the main factor in CVM latency overheads in comparison to native VMs. We require performance breakdowns of CVM storage IO request handling for IO submission and IO completion to confirm these claims. Moreover, breakdowns may shed light on other factors that introduce latency overheads in interrupt-based CVM storage IO.

**Host-side page caching influence.** Unexpectedly, host-side caching influences the performance of CVM storage IO strongly (Chapter 5, Figure 5.1, Figure 5.4). In particular, enabling host-side page caching amplifies latency overheads compared to disabled host-side caching. Further, host-side page caching introduces bandwidth and IOPS overheads, which disabled host-side caching does not exhibit. Moreover, host-side caching amplifies overheads of `dm-crypt` and introduces strong overheads for guest-side polling.

Future work is to investigate why host-side caching introduces these overheads. For this investigation, we expect the previously described request performance breakdown to describe which storage stack components introduce this overhead in CVMs.

## 8.2 Implementation of CVM-IO

During this thesis, we could only partially implement the entire design of CVM-IO. However, the complete implementation of CVM-IO offers substantial performance improvements for CVM storage IO and flexible CVM-backed protection guarantees for data at rest. Hence, implementing these features solves concrete challenges that bar the adaption of CVMs for storage IO heavy and security-critical use cases and poses noteworthy research challenges.

**CVM-IO zero-copy performance gain.** The current CVM-IO implementation implements zero-copy, yet does not identify any performance gain, as the thesis finds the bounce buffer to not significantly impact storage IO. If future work reproduces previous results as in Section 8.1, future work is to investigate the performance gain of CVM-IO with the corresponding storage IO configuration.

**Complete host- and guest-side CVM polling with SPDK.** While this thesis uses guest-side polling in Chapter 5, this work cannot use SPDK with CVMs to enable host-side polling and a CVM high-performance storage backend. As we describe in Chapter 4, this inability to use SPDK with CVMs is because of SPDK's requirement to use a shared memory backend between the user-space storage backend process and the CVM. The CVM thereby prohibits this shared memory backend. While this challenge does not hold a research problem, we must resolve this obstacle to enable high-performance storage IO for CVMs.

**CVM-IO complete protection.** In Chapter 3 and Chapter 4, we describe a CVM-backed storage IO design, which enables CVMs to transfer their advanced protection guarantees for data in use to data at rest. Moreover, this design allows asynchronous enforcement of the storage guarantees to trade off storage IO performance for security.

Future work is to finish the design of the CVM-backed storage. This design component poses a research challenge, as this element requires sealing data via CVM-technology instructions. Such a design may also use an external CVM, allowing for remote static or runtime attestation to ensure data freshness, integrity, and authenticity. Moreover, this design may pose important research challenges if we require the distribution of storage protection guarantees across multiple CVMs.

As this thesis only implements confidentiality for CVM storage IO, future work is to complete the design above and the corresponding implementation.

# List of Figures

# List of Tables

# Bibliography


[Abr+06] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Wiegert. "Intel Virtualization Technology for Directed I/O." In: *Intel Technology Journal* 10.3 (2006).

[Adva] Advanced Micro Devices Inc. *AMD EPYC™ 9334*. https://www.amd.com/en/products/cpu/amd-epyc-9334. Accessed: 2024-02-01.

[Advb] Advanced Micro Devices Inc. *AMDESE/linux*. https://github.com/AMDESE/linux/tree/5a170ce1a08259ac57a9074e1e7a170d6b8c0cda. Accessed: 2024-01-29.

[Advc] Advanced Micro Devices Inc. *AMDESE/ovmf at 80318fcdf1bccf5d503197825d62a157efd27c4b*. https://github.com/AMDESE/ovmf. Accessed: 2024-01-29.

[Advd] Advanced Micro Devices Inc. *AMDESE/qemu*. https://github.com/AMDESE/qemu. Accessed: 2024-01-29.

[AMI] AMI. *Custom UEFI/BIOS - Aptio & AMBIOS*. https://www.ami.com/bios-uefi-utilities/. Accessed: 2024-02-01.

[Arm] Arm Ltd. *Arm Confidential Compute Architecture*. https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture. Accessed: 2023-07-18.

[Arn+16] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer. "SCONE: Secure Linux Containers with Intel SGX." In: *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2016.

[Axba] J. Axboe. *axboe/fio: Flexible I/O Tester*. https://github.com/axboe/fio. Accessed: 2024-02-05.

[Axbb] J. Axboe. *axboe/liburing*. https://github.com/axboe/liburing. Accessed: 2024-01-29.

[Bai+19] M. Bailleu, J. Thalheim, P. Bhatotia, C. Fetzer, M. Honda, and K. Vaswani. "SPEICHER: Securing LSM-based Key-Value Stores using Shielded Execution." In: *Proceedings of the 17th USENIX Conference on File and Storage Technologies*. USENIX Association, 2019.

[Bel05] F. Bellard. "QEMU, a Fast and Portable Dynamic Translator." In: *Proceedings of the 2005 USENIX Annual Technical Conference*. USENIX Association, 2005.

[BPM18] M. Broz, M. Patocka, and V. Matyas. “Practical Cryptographic Data Integrity Protection with Full Disk Encryption Extended Version.” In: *Proceedings of the 33rd IFIP TC 11 International Conference*. Springer International Publishing, 2018.

[Broa] M. Broz. *cryptsetup/cryptsetup: DMCrypt*. https://gitlab.com/cryptsetup/cryptsetup/-/wikis/DMCrypt. Accessed: 2023-12-06.

[Brob] M. Broz. *cryptsetup/cryptsetup: DMIntegrity*. https://gitlab.com/cryptsetup/cryptsetup/-/wikis/DMIntegrity. Accessed: 2023-12-06.

[Del] Dell Inc. *Dell Server PowerEdge R6/7625 BIOS*. https://www.dell.com/support/home/en-us/drivers/driversdetails?driverid=1h2v5&oscode=rhel8&productcode=poweredge-r7625. Accessed: 2024-02-01.

[Dwo07] M. Dworkin. *Recommendation for block cipher modes of operation: Galois/Counter Mode (GCM) and GMAC*. Tech. rep. NIST Special Publication (SP) 800-38D. National Institute of Standards and Technology, 2007.

[Fru05] C. Fruhwirth. “New methods in Hard Disk Encryption.” In: Master’s thesis, Vienna University of Technology, 2005.

[Inta] Intel Inc. *Intel Software Guard Extensions*. https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/overview.html. Accessed: 2024-01-26.

[Intb] Intel Inc. *Intel SSD PC 3600*. https://images10.newegg.com/UploadFilesForNewegg/itemintelligence/Intel/ssd_dc_p3600_spec1453010115706.pdf. Accessed: 2024-02-01.

[Intc] Intel Inc. *Intel Xeon Platinum 8480C Processor (105M Cache, 2.00 GHz) Product Specifications*. https://www.intel.com/content/www/us/en/products/sku/231730/intel-xeon-platinum-8480c-processor-105m-cache-2-00-ghz.html. Accessed: 2024-02-01.

[Intd] Intel Inc. *Software Enabling for Intel TDX in Support of TEE-I/O*. https://cdrdv2-public.intel.com/742542/software-enabling-for-tdx-tee-io-fixed.pdf. Accessed: 2022-09.

[Int21] Intel Inc. *Intel Trust Domain Extensions*. https://www.intel.com/content/dam/develop/external/us/en/documents/tdx-whitepaper-v4.pdf. 2021.

[JS] V. V. John Kariuki and R. Sudarikov. *SPDK Best Practices: Performance Benchmarking and Tuning*. https://ci.spdk.io/download/events/2017-summit/08_-_Day_2_-_Kariuki_Verma_and_Sudarikov_-_SPDK_Performance_Testing_and_Tuning_rev5_0.pdf. Accessed: 2024-01-29.

[Kap17] D. Kaplan. “Protecting VM Register State with SEV-ES.” In: Advanced Micro Devices, Inc., 2017.

[Kap20] D. Kaplan. “AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More.” In: Advanced Micro Devices, Inc., 2020.

[Koba] Kobuk Team. *Intel TDX EDK2*. https://code.launchpad.net/~kobuk-team/ubuntu/+source/edk2/+git/edk2/+ref/mantic-tdx/+index. Accessed: 2024-02-01.

[Kobb] Kobuk Team. *kobuk-team/ubuntu/+source/qemu*. https://git.launchpad.net/~kobuk-team/ubuntu/+source/qemu. Accessed: 2024-02-01.

[Kobc] Kobuk Team. *Linux Intel Ubuntu 23.10*. https://ubuntu.pkgs.org/23.10/ubuntu-updates-universe-amd64/linux-image-6.5.0-1003-intel-opt_6.5.0-1003.3_amd64.deb.html. Accessed: 2024-02-01.

[KPW16] D. Kaplan, J. Powell, and T. Woller. "AMD Memory Encryption." In: Advanced Micro Devices, Inc., 2016.

[Li+23] D. Li, Z. Mi, C. Ji, Y. Tan, B. Zang, H. Guan, and H. Chen. "Bifrost: Analysis and Optimization of Network I/O Tax in Confidential Virtual Machines." In: *Proceedings of the 2023 USENIX Annual Technical Conference*. USENIX Association, 2023.

[Lina] Linux kernel developers. *kernel/git/torvalds/linux.git - Linux kernel source tree*. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git. Accessed: 2024-01-23.

[Linb] Linux kernel developers. *linux/Documentation/security/snp-tdx-threat-model.rst at master · torvalds/linux*. https://github.com/torvalds/linux/blob/master/Documentation/security/snp-tdx-threat-model.rst. Accessed: 2023-11-24.

[NVM] NVM Express Consortium. *NVM Express*. https://nvmexpress.org/. Accessed: 2023-09-07.

[Pri+19] C. Priebe, D. Muthukumaran, J. Lind, H. Zhu, S. Cui, V. A. Sartakov, and P. Pietzuch. "SGX-LKL: Securing the Host OS Interface for Trusted Execution." In: arXiv, 2019.

[RD01] V. Rijmen and J. Daemen. "Advanced Encryption Standard." In: *Proceedings of the 2001 Federal Information Processing Standards Publications*. National Institute of Standards and Technology, 2001.

[Rus08] R. Russell. "Virtio: Towards a de-Facto Standard for Virtual I/O Devices." In: *ACM SIGOPS Operating Systems Review* 42.5 (2008). DOI: 10.1145/1400097.1400108.

[Sam] Samsung Semiconductor Global. *PM1735 SSD*. https://semiconductor.samsung.com/ssd/enterprise-ssd/pm1733-pm1735/mzplj1t6hbjr-00007. Accessed: 2024-01-29.

[Sch] R. Schambach. *Robert Schambach/linux: cvm-io*. https://gitlab.lrz.de/robert/linux/-/tree/cvm-io-dev?ref_type=heads. Accessed: 2024-01-29.

[Tha+21] J. Thalheim, H. Unnibhavi, C. Priebe, P. Bhatotia, and P. Pietzuch. "rkt-io: a Direct I/O Stack for Shielded Execution." In: *Proceedings of the 16th European Conference on Computer Systems*. ACM, 2021.

[Xu+15] Q. Xu, H. Siyamwala, M. Ghosh, T. Suri, M. Awasthi, Z. Guz, A. Shayesteh, and V. Balakrishnan. “Performance analysis of NVMe SSDs and their implication on real world databases.” In: *Proceedings of the 8th ACM International Systems and Storage Conference*. ACM, 2015.

[Yan+17] Z. Yang, J. R. Harris, B. Walker, D. Verkamp, C. Liu, C. Chang, G. Cao, J. Stern, V. Verma, and L. E. Paul. “SPDK: A Development Kit to Build High Performance Storage Applications.” In: *Proceedings of the 2017 IEEE International Conference on Cloud Computing Technology and Science*. IEEE Computer Society, 2017.

[YG23] M. Yan and K. Gopalan. “Performance Overheads of Confidential Virtual Machines.” In: *Proceedings of the 31st International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2023.