Artifact Evaluated
NDSS
Available
Functional
Reproduced

CVM-STORE: A Secure Storage System for Confidential Virtual Machines

Thore Sommer, Masanori Misono, Dimitrios Stavrakakis, Pramod Bhatotia
Technical University of Munich, Germany
{thore.sommer, masanori.misono, dimitrios.stavrakakis, pramod.bhatotia}@tum.de

***Abstract*—Modern stateful cloud workloads require high-performance persistent storage when running on untrusted cloud infrastructure. Examples are AI inference and training pipelines that load multi-gigabyte model weights, checkpoints, and large datasets that must not be silently rolled back, and other workloads such as container runtimes that provision immutable image layers and assets. Confidential Virtual Machines (CVMs) promise to protect these workloads through hardware-enforced memory encryption and isolation without requiring application changes. However, they protect only data in use; persistent storage lies outside the CVM trust boundary, leaving all stateful data exposed to the cloud provider.**

Extending the trust boundary to persistent storage presents inherent challenges. High-bandwidth workloads require near-raw NVMe throughput, yet the default kernel CVM I/O stacks are inadequate, and userspace I/O libraries are not designed with CVM use cases in mind. Access patterns and storage semantics vary between workloads, necessitating access-pattern-aware security abstractions rather than uniform block-level encryption. In order to mitigate against rollback, freshness is needed, but CVMs lack hardware primitives for this purpose, requiring alternative solutions.

We present CVM-STORE, a storage system that co-designs a high-throughput userspace I/O engine with CIF (confidentiality, integrity, freshness) data protection. The key insight is that neither authenticated encryption nor I/O is a bottleneck by itself, but their naive combination is. CVM-STORE exploits this by interleaving them, eliminating redundant copies and OS/hypervisor mediation. CIF guarantees are provided as composable primitives that adapt protection to workload access patterns, from which we derive three concrete authenticated abstractions: a key-value store, a shared log, and a filesystem. An asynchronous trusted counter service provides freshness guarantees with epoch-based batching, which in turn enables rollback protection without dedicated hardware support. CVM-STORE delivers full CIF guarantees within ≈1% (write) and 1–19% (read) of raw NVMe bandwidth for large objects.

I. INTRODUCTION

A broad range of sensitive workloads are increasingly deployed in public clouds and require performant persistent storage. These include container runtimes that provision large, immutable image layers and datasets [68], [128], [42], checkpointing and audit-logging services [77], [24], and HPC, AI training and inference pipelines [116], [80], [90], [127], spanning sensitive domains such as healthcare record management and financial transaction processing. In all cases, the cloud provider controls the entire workload lifecycle and may try to undermine the security guarantees such workloads depend on [86], [71].

Recently, Confidential Virtual Machines (CVMs) have become the dominant model for confidential computing [41] in the cloud. CVMs protect entire VMs with hardware-enforced memory encryption and isolation, enabling unmodified applications to run securely on untrusted infrastructure. Major cloud providers [52], [78], [56] have deployed CVMs, offered by various vendors (e.g., AMD SEV-SNP [20], Intel TDX [59], Arm CCA [25]), at scale, making them the de facto foundation for running security-sensitive workloads.

However, CVMs protect only data *in use*, leaving persistent storage outside their trust boundary. NVMe devices, storage controllers, and the I/O stack remain under full control of the untrusted cloud provider. Thus, a malicious operator can read, corrupt, or replay persistent data, exposing every workload that requires storage outside CVM memory and relies on local storage such as NVMe SSDs. Importantly, fully closing this threat requires not only cryptographic protection of stored data, but also a mechanism to bind that data to a verified, trusted identity. Therefore, designing such secure storage systems for CVMs is necessary but non-trivial.

Simply layering existing confidential-storage techniques [66], [79] onto the CVM I/O path compounds the problem: CVMs encrypt their guest memory by default, preventing devices from performing direct memory access (DMA) to it. All I/O must instead pass through *bounce buffers* [98], unencrypted shared memory regions where data is copied to and copied back, along with multiple context switches, further degrading I/O performance compared to conventional VMs [66], [89]. Block-level encryption then adds another layer on top. Neither cost is inherent. Userspace I/O frameworks [125], [48] bypass the kernel to optimize I/O operations, and modern AEAD ciphers can saturate NVMe bandwidth on a single core. However, userspace I/O frameworks assume a trusted device model, which does not hold for CVM deployments: a malicious NVMe controller can read from DMA buffers, corrupt control structures, or selectively drop and reorder I/O operations, rendering these frameworks incompatible with the CVM threat model in their current form. This mismatch between a fast I/O path built for trusted devices and the CVMs' untrusted-device reality makes securing CVM storage fundamentally challenging (§ III).

Further, protecting persistent data requires ensuring three properties: confidentiality, integrity, and freshness

Network and Distributed System Security (NDSS) Symposium 2027
22–26 March 2027, Seoul, Republic of Korea
ISBN 978-1-970672-09-1
https://dx.doi.org/10.14722/ndss.2027.230799
www.ndss-symposium.org

(CIF). Existing solutions fall short, as generic block-level encryption [13], [36] protects only data uniformly across all access patterns, performing redundant cryptographic operations, and application-level schemes [28], [103] bind their guarantees to a single workload. Both remain vulnerable to rollback attacks where adversaries replay cryptographically valid but stale data after crashes or migrations. Although counter-based freshness mechanisms exist [38], [24], they rely on hardware primitives (e.g., Intel SGX monotonic counters) or external distributed systems [74] that suffer from limited write endurance [28], [103] or extreme latency, rendering them unsuitable for high-throughput CVM storage workloads.

As a solution, we present CVM-STORE, a storage system tailored for CVM deployments that delivers end-to-end CIF guarantees. Our key insight is that CVM storage security and I/O performance are *not* fundamentally at odds. Neither encryption nor userspace I/O is the bottleneck in isolation, only their naive composition is. CVM-STORE's novelty lies in four principles that make this combination work under the CVM threat model, rather than in any single building block: *(i)* hardening the userspace I/O trust boundary instead of assuming a trusted device, *(ii)* interleaving cryptographic operations with in-flight DMA instead of serializing them, *(iii)* aligning integrity structures with each abstraction's access pattern instead of applying uniform block-level protection, and *(iv)* decoupling freshness into an external, epoch-batched service behind a minimal interface. Precisely, CVM-STORE combines a hardened, kernel-bypass I/O engine, access-pattern-aware CIF primitives instantiated across a key-value store, shared log, and filesystem, an asynchronous trusted counter service that externalizes monotonic state via epoch-based batching, and an attestation and sealing pipeline that binds provisioned keys and counter state to the attested CVM identity.

We implement CVM-STORE for AMD SEV-SNP, contributing upstream fixes to SPDK DMA handling and the SEV-SNP attestation pipeline. Our evaluation demonstrates that CVM-STORE with full CIF protections stays within ≈1% (write) and 1–19% (read) of raw NVMe bandwidth on large objects from a single core (up to 6.6 GiB/s and 13.3 GiB/s), outperforming unencrypted CVM kernel I/O. End-to-end attestation completes in ≈188 ms, and the trusted counter interval (1–281 ms) allows applications to tune the performance-freshness trade-off according to their needs.

Overall, this paper makes four key contributions:

- A *hardened kernel-bypass I/O engine* that minimizes exposed state through CVM-aware DMA allocations and hardens the NVMe driver against untrusted infrastructure.
- *Composable CIF primitives* that interleave AEAD encryption with in-flight DMA, hiding the CIF cost behind I/O.
- Per use-case *integrity structures* matched to each access pattern, demonstrated through a KVS, shared log, and filesystem.
- A *pluggable trusted counter service* behind a minimal interface with configurable epoch granularity to bound the rollback window, anchored by an attestation and sealing pipeline binding keys and counter state to the attested CVM identity.

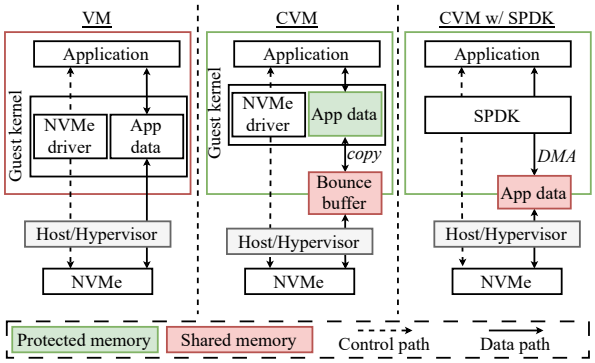


Fig. 1: I/O mechanism comparison in normal VMs, CVMs with kernel I/O, and a strawman design of SPDK in CVMs.

II. BACKGROUND

A. Confidential Virtual Machines (CVMs)

CVMs [20], [59], [25] enable confidential computing in untrusted clouds through hardware-enforced isolation and memory encryption for entire VMs [79]. During launch, the hypervisor loads guest code, data, and state, but thereafter cannot access or modify guest memory or state, and the guest firmware configures the private memory and performs necessary measurements. Further, CVMs support generating attestation evidence to verify their initial state and platform integrity.

Sealing and provisioning. To enable secure data persistence and provisioning, sealing mechanisms bind encrypted data to specific evidence. Sealing is implemented by generating an ephemeral asymmetric key and including its public-key hash in the user-controllable portion of the attestation evidence. After verifying the evidence and public-key hash, the sealing party encrypts sensitive data (e.g., encryption keys) so only the CVM with the sealing key can decrypt it.

B. CVM I/O Architecture

In CVMs, the hypervisor cannot directly access encrypted memory; thus, CVMs rely on an *unencrypted* shared buffer (*bounce buffer* [98]) for I/O communication. The typical I/O process involves [66]: *(i)* the guest kernel encrypts and copies data to a bounce buffer, *(ii)* the hypervisor or device (i.e., DMA) accesses data from this shared region, and *(iii)* for received data, the guest copies and decrypts it from the bounce buffer.

Userspace I/O frameworks. To reduce I/O overhead, kernel-bypass frameworks (e.g., SPDK [125], DPDK [48]) map device registers directly into userspace, so applications interact with devices without syscalls and poll for completion instead of receiving interrupts. In virtualized environments, userspace I/O frameworks can be combined with device passthrough mechanisms (VFIO [111], iommufd [54]). In CVMs, userspace I/O still requires bounce buffers for DMA, and direct device access exposes the CVM to potentially malicious device behavior (Figure 1). While recent work has explored userspace I/O in CVMs [65], [124], [66], [67], these efforts focus on the network I/O path and do not explore combining the I/O layer with CIF properties, and there is no publicly available userspace I/O framework for modern CVMs.

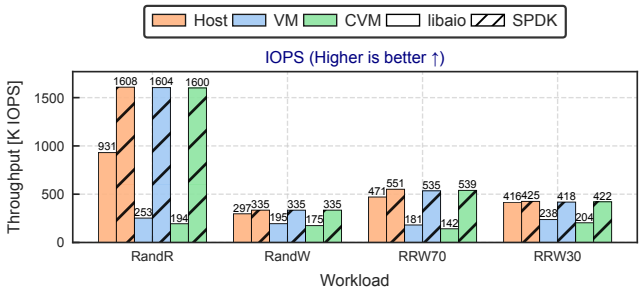


Fig. 2: Difference in throughput between host, VM, and CVM using the kernel libaio and the userspace SPDK driver.

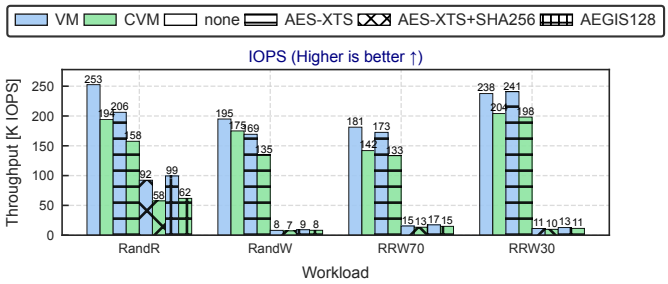


Fig. 3: Disk encryption performance: LUKS overhead with different cipher suites vs. baseline kernel driver inside a VM.

Trusted I/O devices. The recent TEE Device Interface Security Protocol (TDISP) [85], [108] extends the CVM trust boundary to include I/O devices. It provides device attestation and encrypts PCIe transactions, enabling devices to directly access encrypted memory. TDISP requires hardware support and is not yet widely commercially available. Furthermore, it does not ensure security properties for data at rest; it still depends on software or support from the storage medium (e.g., self-encrypting drives [114] or in-storage computing [61]).

C. Modern Cloud Storage Devices

Modern, high-performance NVMe SSDs communicate with the host via a queue-based interface comprising submission queues (SQs) and completion queues (CQs), enabling parallel processing and reduced latency compared to legacy storage protocols [82]. The host submits I/O commands to SQs. The controller fetches them, performs the requested DMA transfers, and posts completion entries to CQs.

III. Motivation and Key Ideas

Protecting persistent storage under the CVM threat model is essential for modern stateful cloud workloads. However, although AEAD ciphers, Merkle trees, and userspace I/O are each well-established individually, naively composing them under the CVM trust model leaves significant performance on the table. Below, we identify three storage requirements arising from this intersection (experimental setup in § IX-A).

Requirement #1: Secure, performant storage I/O. Container runtimes load multi-GB image layers before pod start, AI inference loads tens-of-GB model weights on every cold start, and ledger services require continuous append-only logging. However, standard CVM I/O stacks fail to meet such demands. To this end, we measure I/O throughput using fio [3] across virtualization architectures with varying access patterns (Figure 2). We observe that CVMs achieve only up to 90% of VM write throughput and 77% of read throughput via the kernel I/O stack. This is mainly because devices cannot perform DMA into protected CVM memory, making bounce buffers mandatory for I/O operations. As a result, each operation incurs at least two extra memory copies. Additional context switches across the trust boundary further increase the incurred overhead [79], [66].

Userspace I/O [125] eliminates kernel overhead but does not resolve the bounce buffer under the CVM threat model: devices cannot DMA into protected CVM memory, and an attacker can snoop data in DMA buffers or manipulate NVMe control data structures, requiring their drivers to be hardened.

Application	Stored data	Pattern	Volume
Confidential AI [116]	Model weights	Write-once	GB–TB
Cloud apps [68], [128]	Image layers	Write-once	MB–GB
Forensic media [123], [95], [100]	Video, imaging	Write-once	GB–PB
Object caches [69], [81]	Cached objects	Keyed R/W	KB–GB
Event logs [77]	Audit trails	Append-only	KB–MB

TABLE I: Motivating large-object CVM-Store workloads.

Scheme	Throughput (GB/s) [VM/CVM]	CPU util. (%)
No encryption	13.41 / 6.56	26.30 / 43.78
LUKS (AES-XTS)	10.33 / 4.10	38.43 / 46.15
LUKS (AES-XTS+sha256)	5.94 / 4.11	73.51 / 68.24
LUKS (AEGIS-128)	8.14 / 5.07	46.38 / 52.95
dm-verity	3.42 / 2.53	54.89 / 50.23

TABLE II: Block-level read encryption/integrity performance.

CVM-Store integrates protection into a hardened kernel-bypass I/O engine. CVM-aware DMA allocations keep sensitive driver state out of shared memory, while in-path encryption/decryption subsumes unnecessary bounce-buffer copies. On top of that, completion validation and bounds checks harden the device interface against untrusted infrastructure.

Requirement #2: Workload-tailored abstractions. CVMs provide no built-in data-at-rest protection, and application-level encryption is error-prone and often incomplete. These large-object workloads exhibit fundamentally different access patterns that collapse into three classes, as summarized in Table I. For instance, container image layers, model weights, and forensic media are large and write-once, with sequential writes followed by sequential or random reads [68], [128], [116], [123], [95], [100]. Audit trails and event logs are append-only and rarely read [77], [24], whereas object caches store variable-size blobs retrieved by key [69], [119].

Generic block-level encryption [110] ignores access-pattern structure, incurring unnecessary overheads. For write-once files, dynamic Merkle tree updates on every write are redundant; a static tree computed at write time suffices. For append-only logs, Merkle tree integrity poorly matches the access pattern; instead, counter-chained MACs provide integrity and ordering more efficiently. Sometimes, a simple KV store is sufficient to securely persist application data, making the full block-level storage overhead unnecessary. Figure 3 and Table II quantify this cost: authenticated block encryption with AES-XTS+sha256 reduces throughput by 37–56%, while AEGIS-128 achieves up to 8.14 GB/s.

Algorithm	Throughput (GB/s) [Kernel/OpenSSL]		
	Host	VM	CVM
AES-256-GCM	4.52 / 4.38	4.47 / 4.31	4.43 / 4.22
AES-256-XTS	6.15 / 7.97	6.19 / 7.83	5.98 / 7.77
ChaCha20	1.83 / 3.20	1.79 / 3.14	1.73 / 3.13
ChaCha20-Poly1305	1.18 / 2.24	1.12 / 2.21	1.09 / 2.20
SHA3-256	0.14 / 0.30	0.13 / 0.29	0.13 / 0.29
SHA-256	1.03 / 1.04	1.02 / 1.03	0.98 / 1.01
AEGIS-128	4.19 / –	4.38 / –	3.90 / –
AEGIS-128L	– / 9.47	– / 9.33	– / 9.25
AEGIS-128X2	– / 20.07	– / 19.76	– / 19.59
AEGIS-128X4	– / 27.06	– / 26.70	– / 26.38

TABLE III: Encryption throughput on 4k block size.

Table III shows the encryption throughput of several algorithms at a 4K block size. We measure each algorithm using both the Linux kernel's crypto implementation and OpenSSL in three environments: on the host, in a VM, and in a CVM. AEGIS-128 is available only in the kernel, while AEGIS-128{L, X2, and X4} are available only in OpenSSL. Modern AEAD ciphers (i.e., AEGIS-128X4) are $\sim 6\times$ faster than traditional AES-256-GCM, so CVM-STORE builds on AEGIS-128X4.

Beyond this, CIF enforcement must be lifted to the abstraction level to exploit per-pattern structure. Co-designing each storage API with the I/O engine reduces cryptographic overhead while preserving full CIF guarantees. CVM-STORE provides composable CIF primitives that enable access-pattern-aware protection, with pattern-specific abstractions layered on top of a hardened I/O engine. We demonstrate this through three instantiations: a KV store, a shared log, and a filesystem, each composing these primitives into workload-specific CIF enforcement.

Requirement #3: Fresh, attested stateful data. Confidentiality and integrity alone do not protect against rollback attacks. A stale snapshot retains valid authentication tags: reverting an audit log violates non-repudiation [24], [38], and rolling back model weights can reintroduce known vulnerabilities (e.g., model backdoors). However, generic rollback defenses often require replication and synchronization, while host-level freshness mechanisms contradict the CVM threat model.

Precisely, CVMs lack a local hardware freshness primitive, such as SGX monotonic counters. A prior work augments CVMs with SGX monotonic counters [72], but this incurs 60–250 ms latencies and wears out within days [28]. Counter state must therefore be externalized to a trusted party, incurring a round trip on each freshness update.

Furthermore, external counters alone are insufficient: the trusted party must also verify *which* CVM instance may receive storage keys, initialize counters, and resume persistent state. Without this binding, an attacker can replay sealed state into a modified instance or bind a valid counter to the wrong identity. Attestation is therefore not an auxiliary feature but a foundational design requirement: storage keys are released only after the requester's identity is verified, and provisioning seals those keys and the counter state to that attested identity.

CVM-STORE externalizes monotonic counter state and binds initialization, recovery, and key provisioning to its attestation flow. Per-abstraction counter chains embed values in persistent structures for read-time rollback detection.

Solution	Hardware	Scope	C	I	F	Perf.
Unprotected CVM I/O paths						
Bounce buffer [98]	CVM	Kernel	✗	✗	✗	Medium
SPDK [125][f]	CVM	Userspace	✗	✗	✗	High†
Block-level protection						
Kernel FDE [33], [110]	CVM	Kernel	✓	✓	✗	Medium
MlsDisk [122][c]	SGX, CVM	Kernel	✓	✓	✓	Low
Rollbaccine [38][c,g]	CVM	Kernel	✓	✓	✓	Medium
Shielded runtimes						
Haven [30]	SGX	Userspace	✓	✓	✗	Low
SCONE [26]	SGX	Userspace	✓[b]	✓[b]	✗	Medium
rkt-io [109][f]	SGX	Userspace	✓	✗	✗	High†
Application-specific secure storage						
ShieldStore [63][a]	SGX	KV store	✓	✓	✗	Medium
Speicher [28]	SGX	KV store	✓	✓	✓	Low
Pesos [64][d]	SGX	Objects	✓	✓	✓	Low
EnclaveDB [88][a]	SGX	SQL DB	✓	✓	✓	Medium
NeXUS [47][e]	SGX	Files	✓	✓	✗	Low
Freshness and state-continuity services						
Memoir [84]	TPM	Module state	–	–	✓	Low
ROTE [74][c]	SGX	Counters (repl.)	–	–	✓	Medium
ADAM-CS [73][c]	SGX, TPM	Counters (async)	–	–	✓	Medium
Nimble [24][c]	SGX, CVM	Ledgers	–	–	✓	Medium
CVM-STORE[c]	CVM	KV/Log/FS	✓	✓	✓	High†,‡

[a] whole dataset must fit in memory; [b] read-only files, writes are ephemeral; [c] freshness needs an external trusted store or replicas; [d] freshness needs trusted storage hardware; [e] per-file versions only, hierarchy open to forking attacks; [f] driver not hardened against a malicious device; [g] rollback prevention via replication. † kernel-bypass userspace I/O path; ‡ on MB-scale objects and larger; – out of scope.

TABLE IV: TEE secure storage approaches, grouped by type.

Epoch-based batching sets the performance/freshness trade-off: per-write synchronization bounds the rollback window to one write. In contrast, batching lets a crash revert the store only to the last committed epoch.

Summary and problem statement. The three requirements highlighted above are interdependent. Secure high-throughput I/O requires integrating CIF protection directly into the I/O engine. Workload-tailored abstractions require lifting CIF enforcement to the abstraction level, where per-pattern properties can be exploited. Fresh, attested state requires embedding counter values in the persistent structures provided by those abstractions. Addressing any one requirement in isolation leaves the others unsolved. Unfortunately, existing approaches fail to simultaneously satisfy all three (Table IV): Existing block-level systems protect every block uniformly and ignore access patterns; shielded runtimes mainly focus on their volatile states and do not provide integrity or freshness to persistent data; application-specific designs bind their guarantees to a single workload and to enclave-only primitives (e.g., SGX monotonic counters); and freshness services add rollback protection at 1–76 ms per state advance but provide no data path. This leads to our core research question: *how can we co-design an attested CVM I/O engine and CIF storage abstractions to simultaneously achieve high-throughput secure I/O, access-pattern-aware protection, and freshness?*

IV. OVERVIEW

A. System Architecture

CVM-STORE targets CVM deployments where applications demand high-performance storage access, and the underlying devices and storage stack are exposed to an untrusted cloud

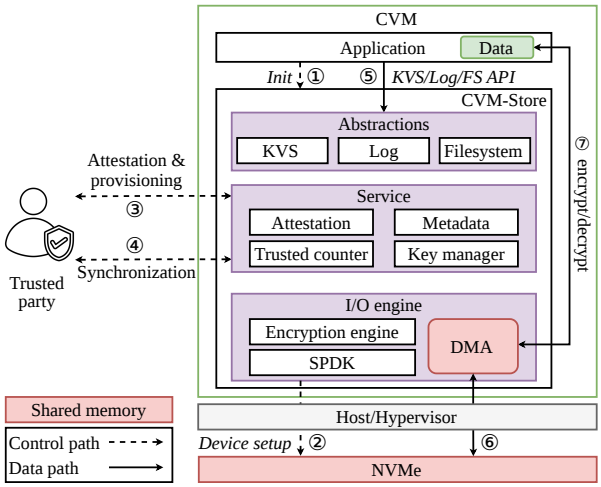


Fig. 4: Overview of CVM-STORE's system architecture.

provider. Figure 4 depicts CVM-STORE's architecture, highlighting its key components, trusted entities, and workflow.

CVM-STORE's I/O engine provides general CIF building blocks: authenticated encryption, trusted counters, and integrity structures. From these, CVM-STORE exposes three access-pattern-aware storage abstractions to support common cloud workloads: First, the *key-value store* manages large immutable values, read back in full and replaced as a whole, with minimal overhead for CIF protections. Second, the *shared log* serves append-only workloads, such as audit logs, where entries are continuously appended, periodically truncated, and rarely read except during recovery. Third, the *filesystem* covers write-once bulk data, spanning input provisioning (i.e., container images, ML model weights, system libraries, signed configuration) and output archives (i.e., model checkpoints, VM disk image snapshots, container layer tarballs). It supports random reads, trading flexibility for more CIF overhead via a Merkle tree. Unlike generic block-level protections, each abstraction exploits its specific access pattern to minimize cryptographic overhead while preserving full CIF guarantees.

At the core, CVM-STORE implements an *I/O engine* built on SPDK Blobstore [102] for high-performance userspace storage access. It integrates an *encryption engine* that applies modern AEAD ciphers directly in the I/O path, optimizing bounce buffer copying with encryption and integrity checks. Further, the I/O engine hardens SPDK's device-facing boundary against untrusted NVMe controllers (Figure 1).

The CVM-STORE *Service* manages all control-path operations. It handles metadata for storage abstractions, ensures crash consistency via copy-on-write updates, and enforces CIF properties by integrating the I/O engine's cryptographic primitives with an asynchronous *Trusted counter* mechanism. The trusted counter maintains monotonic values synchronized with an external trusted party to prevent rollback attacks. Beyond this, the *Service* provides remote attestation and sealing/unsealing primitives for secure provisioning.

Use case	API	Description
KVS	*get(key)*	Retrieve value of KV pair
	put(key, value)	Store/update value for KV pair
	exists(key)	Check the existence of a key
	delete(key)	Delete KV pair
Shared log	*create(name)*	Create named shared log
	append(data)	Append data to log end
	truncate(pos)	Truncate log to position
	read(pos)	Read data at position
Filesystem	*create(name)*	Create named file
	open(name, op)	Open file for read/write
	close(fd)	Close file descriptor
	read(fd, pos, length)	Read data at position/length
	write(fd, data)	Write data to file
Provision	*load(blob)*	Load keys & counter from blob
	store()	Export keys & counter as blob
	evidence(nonce)	Get evidence of CVM

TABLE V: CVM-STORE application and provisioning APIs.

B. System Deployment and Workflow

CVM-STORE's deployment involves the *application user*, and the *trusted parties* responsible for attestation, provisioning, and counter synchronization. CVM-STORE's cryptographic material includes the user-provided symmetric keys for blob encryption and the CVM-generated sealing key used to bind the provisioned configuration to the attested system identity.

Figure 4 illustrates the system workflow. During *initialization* (①), the CVM boots, and the application selects CVM-STORE as its storage backend. CVM-STORE's I/O engine then performs the *device setup* (②), configuring the NVMe via SPDK and allocating untrusted DMA buffers for data transfers.

Next, the user uses the CVM-STORE Service to run the *attestation and provisioning* protocol (③) (§ V-D). The Service collects CVM attestation evidence, which the user verifies through an external verifier, and, upon success, the user provisions symmetric encryption keys and CVM-STORE configuration sealed to the attested CVM. The Service unseals and applies this configuration. Then, it establishes *trusted counter synchronization* (④) with an external trusted party, retrieving or initializing monotonic counter values and periodically synchronizing them according to CVM-STORE's configuration.

At runtime, applications interact with CVM-STORE via its storage abstractions (⑤, Table V). For each operation, CVM-STORE transparently enforces CIF guarantees. The CVM-STORE Service manages the blobs and metadata, while the I/O engine issues NVMe commands and performs DMA transfers using its untrusted DMA buffers (⑥). On writes, CVM-STORE encrypts data and adds integrity metadata and trusted counter values before writing it to the DMA buffer, ensuring that only CIF-protected data is written to storage (⑦). On reads, CVM-STORE decrypts the fetched data and performs CIF verification (⑦) by validating trusted counter values and cryptographic authentication tags before returning it to the application.

C. Programming Model and APIs

Table V lists CVM-STORE's APIs organized by storage abstraction. CVM-STORE transparently provides CIF guarantees to the application through each abstraction: confidentiality

and integrity are guaranteed by its authenticated encryption scheme, and freshness is ensured by trusted counter validation.

The *KVS API* stores and retrieves immutable values by key, with key management backed by persistent storage. The *Shared Log API* supports append-only workloads with efficient truncation and sequential access. The *Filesystem API* provides write-once files with sequential and random reads. The *Provisioning API* generates attestation evidence and seals cryptographic material and system state to the attested CVM identity. All APIs treat objects as a whole, without in-place mutations. Application developers should pick the API that best matches their access pattern and adopt it instead of POSIX, embedding the CVM-STORE library into the application.

D. Threat Model and Security Goal

Threat model. CVM-STORE adopts the standard confidential computing threat model [37], [93], [18]. The TCB comprises the CVM firmware and hardware, and the software running in the CVM. The rest of the system software and infrastructure, including the hypervisor, host OS, and storage devices, is untrusted. We consider attackers who have full control over the host environment and can inspect, modify, or replay any data outside the CVM boundaries, including storage, DMA buffers, and I/O traffic. We assume that the confidential computing hardware functions correctly and that cryptographic primitives are secure. We further assume all CVM components can be attested and that CVM-STORE exclusively uses attestation primitives to prevent multi-protocol attacks [99]. External services used for attestation and counter synchronization are likewise assumed to be trusted and correctly implemented. Side-channel [19], [58], [120], [21], [44], [57], [117], denial-of-service, and physical attacks, including physical RAM attacks [39], [76], are considered out of scope.

Security goal. Under this threat model, CVM-STORE aims to preserve the confidentiality and integrity of all data written through its interfaces, even across system restarts and crashes. Additionally, CVM-STORE offers two guarantees: *(i) freshness*: CVM-STORE never exposes stale or rolled-back state, and *(ii) durability*: unsynchronized operations may be lost after an adversarial restart, but never reappear as stale data. CVM-STORE therefore admits *data loss* only up to the last synchronization point, whose frequency is a configurable trade-off between consistency and performance. CVM-STORE does not provide metadata (e.g., file size, I/O pattern) confidentiality, but ensures its integrity and freshness.

V. CVM-STORE DESIGN

CVM-STORE consists of five building blocks: the I/O engine (§ V-A), the trusted counter (§ V-B), the CVM-STORE service (§ V-C), and the attestation (§ V-D) and sealing (§ V-E) mechanisms, which form the basis of the CIF-protected abstractions (§ VI). The implementation details are presented in § VIII.

A. I/O Engine

CVM-STORE's I/O engine (Figure 5 (a)) builds on SPDK [125] and provides high-performance userspace NVMe access from within the CVM. It ensures confidentiality and integrity through authenticated data encryption. On writes, CVM-STORE transparently handles the encryption from application memory to an unprotected shared memory ring buffer before NVMe DMA, and follows the same process in reverse on reads.

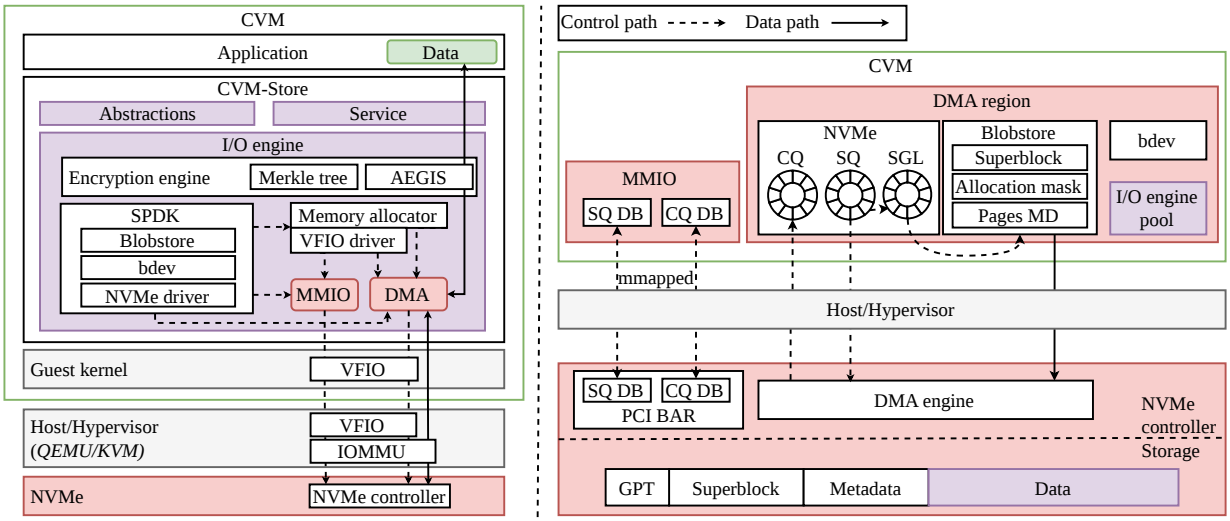
SPDK Blobstore. SPDK Blobstore [102] is a userspace, power-fail-safe data allocator designed for storage services without implementing a POSIX filesystem. It organizes NVMe device space into a four-level hierarchy (logical blocks, pages, clusters of 1 MiB by default, and blobs), where a blob is an ordered list of clusters forming a variable-sized, contiguous device allocation. Blobstore manages all logical-to-physical block mappings and provides allocation, read, and write primitives via the SPDK NVMe driver. CVM-STORE adopts blobs as its fundamental storage unit, delegating physical block management to Blobstore and focusing its own logic on CIF-protected blob layout.

On the control path, the I/O engine uses Blobstore to manage storage operations, and combines it with confidentiality and integrity checks. It organizes each blob on disk into three sections: unprotected metadata (nonces, MACs, file size), optional encrypted metadata, and encrypted data chunks.

Encryption and integrity protection. CVM-STORE's I/O engine interleaves encryption and integrity protection in the data path. Combining encryption with data movement between DMA-shared memory and application memory eliminates the need for additional copies. On writes, the I/O engine submits a ring-buffer slot only after it is filled with ciphertext, then encrypts later chunks while earlier DMA transfers are in flight. On reads, it posts multiple DMA reads into ring-buffer slots and decrypts completed slots in order, while further reads are in flight.

Its crypto subsystem provides *authenticated encryption* for confidentiality and integrity, and *Merkle tree* verification for tamper detection. It is built to be cipher-agnostic and by default employs the modern AEGIS-128X4 cipher [121], [45], [12], which is nearly 6× faster than the common AES-256-GCM (further details in Table III). For the Merkle tree, CVM-STORE uses MACs as leaves rather than separate hashing, removing an additional cryptographic operation [35]. Caching validated Merkle tree nodes further reduces verification overhead. By interleaving CIF protections, using modern ciphers, and eliminating copies in the I/O engine, CVM-STORE saturates a modern NVMe device on a single core for large data-intensive workloads.

Hardening. In CVMs, the NVMe controller is untrusted and may read DMA buffers, corrupt control structures, or forge completion entries. Therefore, CVM-STORE must harden SPDK itself, which by default assumes a trusted controller and co-locates driver state with device-accessible memory. We reviewed the SPDK codebase for its interactions with external devices and identify the following issues. Each `nvme_tracker` embeds the PRP list, SGL descriptors, and metadata SGL in DMA memory, exposing request pointers, callback pointers and arguments, and queue links to a malicious host or NVMe controller. CVM-STORE hardens this by splitting the `nvme_trackers` into private driver state and a separate per-tracker DMA buffer containing only the NVMe-required PRP/SGL descriptors. The device receives



(a) I/O engine components

(b) I/O engine memory and storage layout

Fig. 5: Inner components of the I/O engine and the view of storage and shared memory region between CVM and host.

the physical address of the descriptor buffer, while the tracker array remains outside DMA-accessible memory. CVM-STORE further verifies that the controller-supplied completion identifier (CID) falls within the allocated tracker range before using it to index the array. Invalid CIDs fail the queue, preventing a forged completion from turning an untrusted DMA field into an out-of-bounds access to SPDK's internal tracker state.

Device passthrough. CVM-STORE uses a pass-through NVMe device to achieve near-raw bandwidth (this assigns a device to a single guest and SR-IOV can relax this restriction [85]). For this, the I/O engine employs VFIO [111] to bind the NVMe to CVM-STORE's CVM. CVM memory protection alone ensures that devices cannot write to arbitrary guest memory without IOMMU. Therefore, CVM-STORE securely uses VFIO without an IOMMU in the CVM. The VFIO PCI BAR space is mapped directly into guest memory (Figure 5 (b)), which the guest then maps to userspace. This direct mapping eliminates the need for expensive MMIO hypercalls for device access.

B. Trusted Counter

Persistent storage remains vulnerable to rollback attacks [84], [38], [24], [74]. As CVMs lack monotonic counters, they require external synchronization with a trusted party for freshness (e.g., a local SGX sidecar [72] or a remote service). CVM-STORE treats this trusted party as a pluggable service, implemented as a library exposing a trusted counter interface. It distinguishes *durability*, persisting metadata to disk, from freshness, synchronizing the covering epoch with the trusted party.

CVM-STORE groups operations into *epochs*, the granularity at which it maintains durability and freshness. An uncommitted epoch defines the vulnerability window, configured by the *epoch length* and *synchronization mode*: the host can revert to the last *committed* one. In *blocking* mode, further mutations are held until the in-flight epoch is durable and committed, so a crash loses at most one epoch; the *non-blocking* mode lets them proceed while it synchronizes, trading the risk of losing up to two epochs for lower latency. The extended version of this paper [101] details the counter interface, the required service properties, and the durability and freshness modes.

C. CVM-STORE Services

The CVM-STORE Service manages all control-path operations (i.e., metadata, crash consistency, CIF enforcement), handles the data lifecycle, and reclaims unused resources.

Metadata management. The CVM-STORE Service maintains Blobstore metadata for low-level storage management and application-specific metadata for the storage abstractions. SPDK's Blobstore handles blob allocation and its metadata on the physical disk. Application-specific metadata is persisted via the I/O engine with explicit synchronization control to balance consistency with performance. To ensure crash consistency, the Service employs a copy-on-write, writing modified metadata to new locations before swapping between them, ensuring atomic updates. Cleanup of stale blobs is deferred till a successful metadata synchronization to prevent inconsistencies after crashes. CVM-STORE omits journaling-based recovery to maintain simplicity and minimize overhead.

CIF enforcement. The CVM-STORE Service ensures the CIF properties through two mechanisms: the I/O engine's cryptographic primitives enforce confidentiality and integrity, while the trusted counter mechanism ensures freshness. Trusted counters are embedded in persistent data, binding each storage operation to a counter state, ensuring no stale data is replayed after crashes or restarts. Counter increments are batched with storage operations to minimize synchronization overhead. At startup, the Service validates all CIF properties, verifying counter values against the external trusted party and authenticating all metadata structures, before granting application access.

Attestation and provisioning service. The CVM-STORE Service serves as the central authority for attestation evidence collection and provisioning, coordinating with external verifiers and trusted counter services. It implements sealing/unsealing primitives that enable the secure provisioning of secrets and

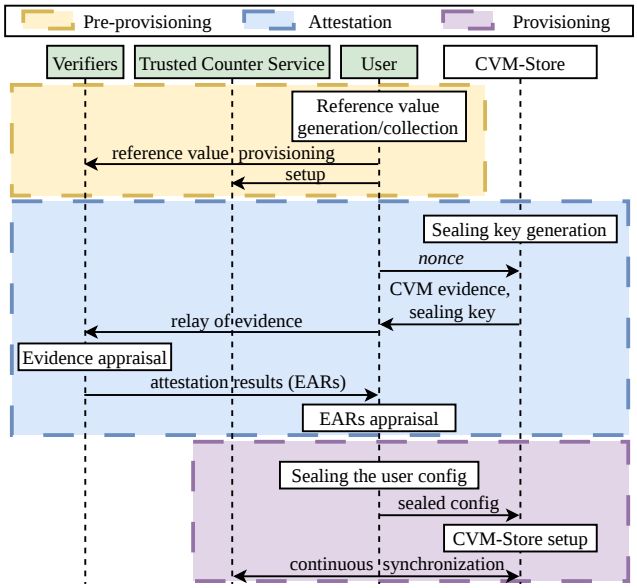


Fig. 6: Attestation and provisioning flow in CVM-STORE.

configuration, allowing CVM-STORE to bind persistent state to CVM instances via attestation-derived sealing keys.

D. Attestation and Provisioning

CVM-STORE demonstrates a complete attestation and provisioning pipeline (Figure 6). Before deployment, the user collects platform endorsements and reference values [23], provides them to a verifier, and sets up the trusted counter service.

Attestation. On CVM startup, CVM-STORE generates a sealing key pair, and the Service embeds a user-supplied nonce and the public key in the CVM attestation report. The user has the evidence appraised by the verifier (an EAT Attestation Result [49]) and checks that it names the expected public sealing key.

Provisioning. The user seals the CVM-STORE configuration to the sealing key; the Service unseals it, initializes CVM-STORE, and connects to the trusted counter service. The configuration carries the symmetric on-disk key and the freshness-service parameters. The sealing key is per boot, so a rebooted or migrated instance is re-attested and re-provisioned by the user, and the counter service rejects cloned or restored copies.

E. Data Sealing and Unsealing

CVM-STORE provides four operations for secure data management. User-to-CVM-STORE sealing encrypts data with CVM-bound keys, ensuring only the attested CVM instance can access provisioned material. CVM-STORE-to-user sealing encrypts internal state using the configuration-provisioned symmetric key, enabling the user to export and later restore system state. The corresponding unseal operations perform decryption and system setup or allow data processing as opaque blobs. For data provisioning, the trusted party runs a local CVM-STORE with the target symmetric key and simulated counters. This allows the data to be written without any trusted counter round trips. The resulting counter value is then provisioned into the freshness service so that the receiving CVM can validate the data freshness and consistency.

VI. CVM-STORE ABSTRACTIONS

To support diverse cloud workloads with high-performance I/O while remaining easy for applications to adopt, CVM-STORE provides three storage abstractions: KVS, shared log, and filesystem. All three build on CVM-STORE's I/O engine (§ V-A) and Service (§ V-C), and share copy-on-write and deferred cleanup policies for crash consistency and counter-based rollback protection. However, rather than applying uniform block-level protection, each abstraction co-designs its integrity metadata with its access pattern, enabling workload-specific CIF enforcement and overlap of encryption with I/O.

A. Key-Value Store (KVS)

CVM-STORE's KVS (Figure 7 (a)) stores one KV pair per blob and models updates as object replacement, targeting large values that are read sequentially.

Data organization. CVM-STORE's KVS maintains a key map in memory along with CIF-protected on-disk blobs. The map stores keys, trusted counter values, and blob identifiers, enabling existence checks and freshness validation without keeping values in memory. This design trades memory proportional to the number of keys for fast lookup and rollback checks, targeting workloads with MB-to-GB objects rather than high-cardinality small records. The metadata blob stores a multiset hash [40] over all key-counter pairs and the latest trusted counter value. Each KV blob has an encrypted metadata section (trusted counter value, key), and a data section for the value.

Crash consistency. CVM-STORE stores the KVS hash with the trusted counter value in the encrypted KVS metadata blob, updated atomically. Blobs are removed only after synchronization, preventing dangling references. On startup, CVM-STORE reads the metadata blob to retrieve the trusted counter value and hash, then reconstructs the key mapping by walking all on-disk blobs, discarding those with counters exceeding the metadata counter and selecting the higher valid counter for duplicate keys. Integrity is verified by recomputing the hash over all reconstructed key-counter pairs and comparing it against the stored value.

Operations. The *Put* operation inserts or updates a KV pair. CVM-STORE allocates a blob, obtains and increments the trusted counter, encrypts the metadata (trusted counter, key) and data section (value), writes them to the blob, and places the metadata in the in-memory map. Then, if requested, it atomically synchronizes the metadata blob.

The *Get* operation retrieves a value given a key. CVM-STORE looks up the key in the in-memory map to obtain the blob identifier and expected trusted counter, reads and decrypts the blob, and validates the counter and the key in the encrypted metadata against the looked-up key before returning the value.

For the *Delete* operation, CVM-STORE looks up the key in the map, marks the blob as deleted, removes the key from the in-memory map, and defers blob deallocation to the next metadata synchronization.

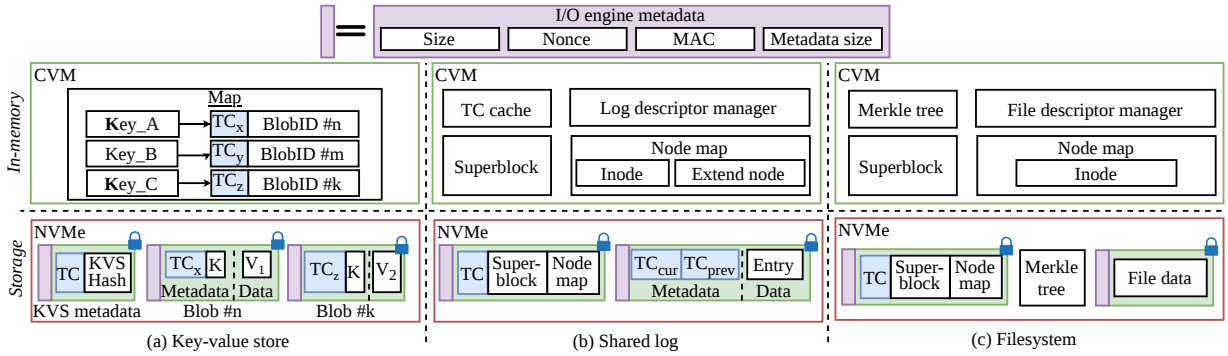


Fig. 7: CVM-STORE's storage interface in-memory metadata and persistent data structure organization.

B. Shared Log

CVM-STORE's shared log (Figure 7 (b)) is suitable for append-heavy workloads, such as audit logs, where entries are appended, truncated, and rarely read outside recovery.

Data organization. A *super-block* contains a map of *inodes* and *extend nodes*. The *inode* stores the log name, entry count, flags, first/last-entry counters and direct references to the first and last extend nodes to avoid full list traversal on append and retrieval. The *extend nodes* form a doubly linked list of fixed-size segment-range arrays. Each *log entry blob* stores encrypted metadata (the current and previous entry's counters) and encrypted log data; the MAC covers both, preventing tampering with the counter chain or the entry data. CVM-STORE caches expected counter values in memory to validate entries without full log traversal.

Crash consistency. CVM-STORE updates the super metadata block atomically by rewriting only dirty nodes into the cold blob before swapping it with the hot one. Embedding the trusted counter value into the metadata guarantees freshness. On first read/open of a log, it traverses all entries and builds the counter cache, verifying for each entry that the stored previous counter matches the predecessor's counter value, with the authenticated encryption MAC protecting the binding between counters and entry data. Finally, CVM-STORE confirms that the inode's recorded first and last counter values match those encountered during traversal.

Operations. The *Append* operation adds new data at the end of the log. CVM-STORE first marks the inode with an *append-in-progress* flag to prevent concurrent modifications, then allocates a new blob and obtains the last log entry's counter and the current counter from the CVM-STORE service. If the blob extends the last node's last range or fits a free slot, it is recorded there; otherwise, CVM-STORE allocates a new extend node, links it from the inode and the previous node, and records the blob as its first range. It then encrypts the metadata and data sections, and updates the trusted counter cache before writing to the blob. The inode metadata is updated to reflect the new entry, including incrementing the entry count and updating the last entry's trusted counter value. Finally, the append-in-progress flag is cleared, and the trusted counter is incremented.

The *Retrieve* operation reads a log entry by an entry number. CVM-STORE uses the log descriptor to identify the inode, returns an error if an append is in progress, and checks the requested number against the inode's entry count. It then traverses extend nodes backward from the inode's last extend node reference to locate the target, retrieves the blob identifier, reads and decrypts the blob, and validates the counter in the blob's metadata against the trusted counter cache. For the first and last entries, CVM-STORE additionally validates counters against the inode's boundary values. Only after CIF verification is the decrypted entry returned to the application.

The *Truncate* operation removes old log entries. An application specifies how many entries to retain; the oldest entries are removed first to maintain ordering. CVM-STORE traverses the extend node linked list from the beginning, marking the blobs for deletion and any node whose reference count is zero for deallocation. To keep subsequent consistency and freshness checks, it updates the entry count and the first entry's counter in the inode to reflect the truncation. It also updates the cached trusted counter mapping to remove metadata for truncated entries. Under CVM-STORE's deferred cleanup policy, marked blobs are freed only after metadata synchronization.

C. Filesystem

CVM-STORE's filesystem abstraction (Figure 7 (c)) exposes file-oriented APIs (Table V) for write-once, read-many workloads such as input provisioning (e.g., container images) and output archives (e.g., snapshotting). CVM-STORE therefore adopts the design choice of treating files as immutable, name-addressed objects in a flat namespace, computing a *static* Merkle tree once at write time instead of the dynamic authenticated structures required by mutable data.

Data organization. CVM-STORE's filesystem reuses the shared log's inode and node map, without extend nodes, but adapts the integrity mechanism to support Merkle tree verification for random access to file chunks. Each file consists of an encrypted blob for chunked file data and an *unencrypted* blob for its Merkle tree, computed over all chunks in the file's data blob. The inode contains the Merkle tree root hash and direct references to the encrypted file data and the Merkle tree blob, enabling efficient integrity checks without exposing file contents. For newly created files, the inode is initialized with a file size of zero and no blob references.

Category	Attack Vector	Mitigation
General	Loading compromised instance	Attestation
	CVM memory access	CVM protection
	Storage data snooping	AEAD encryption
	Storage data tampering	AEAD authentication
	Device descriptor manipulation	Bounds checking on DMA
	Rollback of storage state	Trusted counter
	Cross-instance key reuse	Attestation-bound key provisioning
	State inconsistency exploitation	Copy-on-write & CIF revalidation on startup
KVS	KV value rollback	Per-blob counter checks
	KV unauthorized deletion	Hash over key–counter pairs
Log	Log entry splicing	Counter-chained MACs
	Log truncation (front/back)	Boundary counter validation
	Log entry reordering	Monotone counter chain
FS	Chunk/block replacement	Merkle tree leaf MAC
	File replacement	Root hash in trusted-counter-protected metadata

TABLE VI: Attack vectors and CVM-STORE's mitigations.

Crash consistency. The superblock is updated with the same mechanism as the shared log. If a blob is no longer referenced in the metadata on disk, then it gets reclaimed (§ V-C).

Operations. The *Write* operation stores data to an *empty* file. CVM-STORE marks the inode as *write-in-progress* to prevent concurrent accesses, and chunks the data according to a configurable policy. It encrypts each chunk using a nonce derived from a global file nonce and the chunk number. Because files are write-once, the global file nonce is generated once at creation and never reused, guaranteeing nonce uniqueness across all file chunks. As chunks are written, CVM-STORE incrementally builds the Merkle tree using MAC-based leaf hashes and stores the tree in a separate blob. The root hash and Merkle tree blob identifier are stored in the inode. Upon completion, the file becomes read-only, and the inode is marked as available.

The *Read* operation retrieves data from a file at a specified position and length. CVM-STORE uses the file descriptor to retrieve the inode, checks that no write is in progress, and verifies the requested range against the inode's stored size. On the first read, it loads and caches the Merkle tree from disk. It then identifies the chunks for the requested range, reads and decrypts them directly into an application-provided destination buffer, and verifies them against the cached Merkle tree, skipping already-verified nodes. Only after successful CIF verification does CVM-STORE return control to the application.

VII. SECURITY ANALYSIS

We analyze CVM-STORE against attack vectors from an untrusted host under our considered threat model (§ IV-D). Table VI summarizes these attacks and CVM-STORE's mitigations.

A. General Attack Vectors

Loading compromised instance. An attacker may try to bypass CVM-STORE by loading a compromised instance; attestation detects this through invalid measurements.

CVM memory access. CVM protection blocks unauthorized reads or writes to private memory.

Storage data snooping. An attacker may observe storage or DMA-shared memory. The I/O engine encrypts data inside the CVM before it enters DMA-shared memory. Thus, DMA buffers and the storage device see only ciphertext or control structures (§ V-A). Keys are provisioned through attestation and never leave the protected CVM memory (§ V-D).

Storage data tampering. An attacker may alter an encrypted CVM-STORE blob while preserving the storage layout. Each blob written through the I/O engine carries an AEAD authentication tag (§ V-A), allowing CVM-STORE to detect tampering on a subsequent read operation. For KVS and Shared Log, ciphertext, metadata, or tag changes lead to a failed decryption; filesystem cross-chunk replacement is handled by Merkle-tree checks, as explained in § VII-D.

Manipulating device descriptors. An attacker may manipulate device descriptors in the shared memory, injecting adversarial metadata (e.g., invalid descriptor offset) [55]. CVM-STORE mitigates this by limiting DMA-shared state and performing bounds-checking when reading DMA-received fields (§ V-A).

Rollback of storage state. An attacker may replay an older disk state with valid authentication tags. CVM-STORE mitigates this by embedding trusted counters in persisted data and synchronizing them with an external trusted party (§ V-B). These values are validated before returning data (§ V-C). The asynchronous synchronization interval bounds freshness, as discussed in § VII-E.

Cross-instance key reuse. An attacker may replay sealed configuration or storage keys into another CVM. CVM-STORE binds provisioned keys and configuration to the public sealing key in attested evidence in such a way that a different or compromised CVM cannot unseal them (§ V-D).

State inconsistency exploitation. An attacker may crash a CVM to induce an inconsistent CVM-STORE state. CVM-STORE's copy-on-write metadata updates, deferred cleanup, and restart revalidation ensure each synchronized state is a valid snapshot, effectively eliminating inconsistent state (§ V-C).

B. Attacks on KVS

CVM-STORE persists all key–counter mappings on disk via a multiset hash in the CIF-protected metadata blob (§ VI-A), enabling the detection of the following attacks.

Value rollback. An attacker may replace a value with an older, valid blob for the same key. On every GET, CVM-STORE compares the blob's key and counter with the in-memory key map to prevent stale valid blobs from replacing current values (§ VI-A).

KV unauthorized deletion. An attacker may remove a valid blob while leaving others intact. Startup reconstruction is then a different set of key–counter pairs, so the hash set recomputed over them no longer matches the stored one, catching the deletion before exposing the key space. Legitimate deletions atomically update the metadata hash and defer physical deallocation until the next synchronization.

C. Attacks on Shared Log

The Shared Log's potential attack surface consists of rearrangements that leave individual entries valid. Its security invariant is an AEAD-protected counter chain, along with

inode boundary counters for the first and last entries. Any rearrangement becomes visible during reads or at log-open validation (§ VI-B).

Log entry splicing. An attacker may replace an entry in the middle of the log with another valid entry. However, CVM-STORE maintains the trusted counter so that its values are unique across all log entries, including those from different logs. Successfully replacing an entry in the middle of the log would require mutually consistent counter values and MACs; specifically, the replacement must match its own counter, its predecessor counter, and the successor's stored predecessor counter. Without access to the encryption key, the adversary cannot forge such an entry or modify neighboring entries accordingly.

Log truncation (front/back). An attacker may remove valid entries from either log end. The log inode stores boundary counters and entry count, validated at log open and boundary-entry retrieval. Therefore, a truncation that changes boundary counters or entry count would fail validation (§ VI-B).

Log entry reordering. An attacker may swap valid entries. The monotonic counter chain commits each entry to its predecessor's counter, so any swap breaks at least one predecessor relationship and exposes tampering to CVM-STORE.

D. Attacks on Filesystem

The Filesystem attack surface consists of block-level and whole-file replacement. Our filesystem invariant is that immutable file chunks are verified against a static Merkle tree whose root is stored in CIF-protected filesystem metadata (§ VI-C).

Chunk/block replacement. An attacker may replace, reorder, or replay individual chunks within a file. At write time, CVM-STORE commits the encrypted chunks to the file's Merkle root; on read, it releases plaintext only after the requested chunks get verified against this root (§ VI-C). These operations ensure that replacing, reordering, or replaying a chunk invalidates the corresponding leaf and propagates to a root mismatch before plaintext is returned.

File replacement. Although an old data blob and a matching Merkle-tree blob can verify internally when replayed together, CVM-STORE detects the file replacement as read operations check chunks against the current file root hash recorded in trusted-counter-protected filesystem metadata. Replaying the metadata, including the superblock state, is rejected during startup because its trusted counter value is detected as stale.

E. Limitations

Metadata exposure. As CVM-STORE encrypts at the abstraction rather than the block level [126], [53], [34], metadata such as file sizes, blob counts, and I/O access patterns remain visible to the untrusted host. Integrating oblivious execution [17] could hide the access pattern with a performance overhead trade-off.

Trusted counter service dependency. CVM-STORE relies on an external trusted counter service for freshness. A configuration-controlled synchronization interval trades synchronization overhead for a larger rollback window: data written since the last synchronization point may be lost after an adversarial restart.

Side-channel / physical attacks. Side-channel and physical attacks are out of CVM-STORE's threat model (§ IV-D). CVM-STORE relies on guest-side hardening and defenses [5], and physical data center security [8].

Denial-of-service. Similar to the CVM threat model [20], CVM-STORE does not prevent an adversarial host, hypervisor, or storage device from dropping requests, delaying completions, exhausting resources, or refusing to provide the NVMe device.

VIII. IMPLEMENTATION

We implement CVM-STORE targeting AMD SEV-SNP, but our design is vendor-agnostic and applicable to other CVMs such as Intel TDX and Arm CCA. CVM-STORE is implemented as a userspace library with SPDK, totaling ≈20,000 lines of code (LoC) for the core. Further, it provides example clients, modifications to the Linux guest kernel, and extended SPDK [125]/DPDK [48] components to support encrypted memory management, inspired by prior work [65], [124], [66], [67], [109].

Userspace I/O handling. We re-enable userspace MMIO handling in SEV-SNP CVMs, which is disabled to prevent hypervisor coordination attacks [1], which attestation of the entire usersapce mitigates. To enable userspace DMA memory allocation, we extend mmap with the *VM_CVM_SHARED* flag, which decrypts pages on fault for DMA access and re-encrypts them on unmapping.

DPDK integration. SPDK utilizes DPDK [48] for environment abstraction and memory management. Since CVMs protect memory by default, DPDK's allocator must distinguish between regular and DMA-capable memory. We modify the Environment Abstraction Layer (EAL) memory allocation APIs to propagate memory type and maintain separate *memzones*, *memseg* lists, and heaps per type (≈1,500 LoC), and export APIs to manage distinct pools for encrypted and decrypted memory. This approach reuses existing DPDK allocation infrastructure while enabling memory-type awareness, avoiding the alternative of external allocation which would forgo integration with DPDK's allocation logic.

SPDK integration. SPDK abstracts device handling and memory management via an environment layer. In its abstraction layer, SPDK distinguishes between normal and DMA memory allocations, requiring changes in fewer than 50 LoC to integrate with CVM-STORE's encrypted memory management. During development, we identified upstream SPDK code that did not properly use the allocation APIs and upstreamed fixes.

CVM-STORE *library.* The I/O engine uses SPDK's Blobstore [102] on top of NVMe *bdev*, getting its DMA buffers through its own `iobuf` modules using SPDK's pools, which are allocated in CVM shared memory. Authenticated encryption uses AEGIS-128X4 from libaegis [11], and filesystem integrity is build on top of a Merkle tree from cryptsetup [13]. During development, we further identified that DMA buffers without 4K alignment slow down NVMe

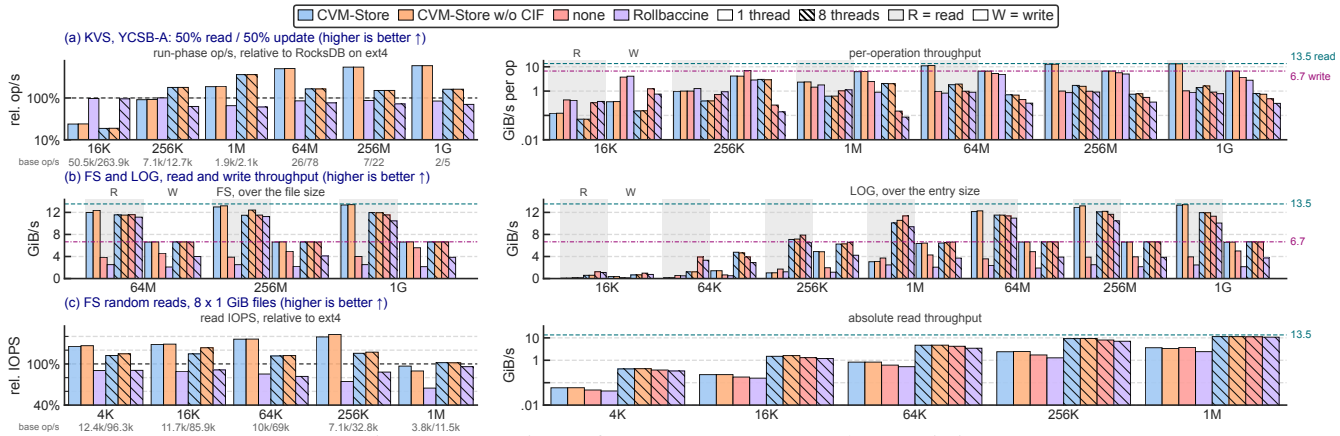


Fig. 8: Comparison of CVM-STORE's I/O characteristics.

transfers, and CVMs without AES-NI and AVX512 enabled suffer a ≈50% reduction in AES performance [2].

Trusted counter. CVM-STORE implements trusted counters as a pluggable library that supports different latency and synchronization trade-offs and can be replaced with a trusted third-party module.

Attestation. CVM-STORE's attestation follows the IETF RATS background check model [32]. We provision SEV-SNP-specific reference values generated by *go-gen-ref* [92] along with AMD trust anchors [20] into the verifier. The CVM generates a key pair and includes the key hash with a verifier nonce in the attestation evidence. CVM-STORE uses ratsd [15] to collect CVM evidence via configfs-tsm [6] in RATS CMW format with an Entity Attestation Token (EAT) [31], [70], and forwards it to a relying party for verification. This party uses Veraison [7], [16] to validate the evidence and receives an EAT Attestation Result (EAR) [49], seals the CVM-STORE configuration against the attested public key, and transmits it to the CVM. Deployment required backporting Linux kernel 7.0 patches [4] for certificate chain provisioning, patching QEMU, and extending sev-snp-measure-go [10] to support OVMF measurements; all applicable changes are sent upstream.

IX. EVALUATION

We evaluate CVM-STORE across six dimensions: I/O characteristics of its storage interfaces (§ IX-B), end-to-end performance (§ IX-C), read/write amplification (§ IX-D), ablation study (§ IX-E), attestation and sealing (§ IX-F), and freshness and durability costs (§ IX-G).

A. Experimental Setup

Testbed. We conduct our experiments on two AMD SEV-SNP servers with 9334/9654 CPUs, and a 3.8 TB KIOXIA CM7-R NVMe. The host runs NixOS 25.11 with Linux 6.18, including patches for NVMe speed and backported SEV-SNP KVM certificate syscall [104], [97], [96]. (C)VMs run Linux 6.17 with our mmap patch, 64 GB RAM, 64 vCPUs (32 vCPUs/256 GB for checkpointing), mirroring the host's NUMA topology for NVMe locality.

Variants. We compare CVM-STORE against ext4 and Rollbaccine [38], a state-of-the-art anti-rollback block layer for TEEs. We match their durability settings per experiment.

B. I/O Analysis

To isolate each interface's raw performance, we use synthetic workloads against ext4 with direct I/O and Rollbaccine at $f=0$ (no replication, `O_DIRECT`). CVM-STORE runs non-blocking, synchronizing its epoch only at shutdown, so neither pays per-operation synchronization. We use small entry/file sizes of 16 KB to 1 MB (128 GB in total) and large ones of 64 MB, 256 MB, and 1 GB (256 GB in total), and run with 1 and 8 threads for single- and multi-core behavior. Dashed lines mark the raw NVMe bandwidth measured with fio (13.5 GiB/s read, 6.7 GiB/s write). The *Filesystem* reads entire files sequentially, if not specified otherwise. The *KVS* runs the update-heavy YCSB workload A [43] (50/50 read/update) against RocksDB with its write-ahead log disabled and BlobDB for values ≥4 MB. The *Shared Log* sequentially appends and retrieves data, simulating high-volume writes and recovery.

CVM-STORE's *KVS* (Figure 8 (a)) trades small-object throughput for large-object bandwidth. It reaches 4.7–4.9 GiB/s at 256 MB and 1 GB (per request, 94–98% of the device's read and 99% of its write bandwidth), 5.5–7.0× higher than RocksDB, which stalls below 0.9 GiB/s on both ext4 and Rollbaccine, and overtakes it from 1 MB upward (256 KB with 8 threads). Below that, CVM-STORE is slower, as RocksDB is optimized for small objects. On the *Filesystem* (Figure 8 (c)), a single CVM-STORE thread leads both baselines on small random reads: at 256 KB it reads 2.4 GiB/s, 1.4× higher than ext4 (1.7 GiB/s) and 1.9× higher than Rollbaccine (1.3 GiB/s), converging with ext4 by 1 MB. Eight threads lift CVM-STORE to 11.4 GiB/s at 1 MB, still ahead of ext4 (11.2 GiB/s) and Rollbaccine (10.7 GiB/s). For large sequential file reads (64 MB–1 GB) (Figure 8 (b), left), CVM-STORE splits each file into pipelined chunks. A single thread sustains 13.0–13.3 GiB/s from 256 MB, 96–99% of the device, against 30% for ext4 and 19% for Rollbaccine. On the *Shared Log* (Figure 8 (b), right), a single CVM-STORE thread leads both baselines on appends at every entry size, 2.3–2.8× at 16 KB (397 MiB/s), while a fixed per-entry retrieval cost keeps small reads 2.2–3.5× below ext4 (80 MiB/s at 16 KB). From 1 MB and above, CVM-STORE's appends reach 95–99% of the device's write bandwidth, 3× Rollbaccine (2.1 GiB/s), and sequential replay 97–98% of its read bandwidth after the counter-cache build.

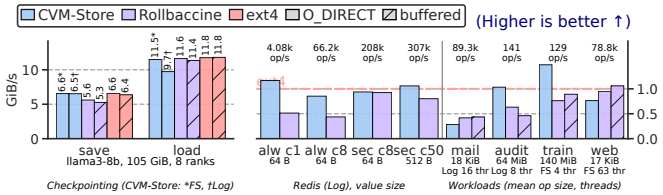


Fig. 9: End-to-end evaluation results.

Takeaway 1: From 64 MB, one CVM-STORE core with CIF is within ≈1% (write) and 1–19% (read) of raw NVMe bandwidth, reaching 13.3 GiB/s read and 6.6 GiB/s write; single-threaded ext4 and Rollbaccine reach 18–30% on reads and need eight threads to compensate bounce buffer and CIF overhead.

C. End-to-end Performance

We evaluate the CVM-STORE's end-to-end application performance with the following benchmarks: ML training checkpointing (105 GB Llama3-8B model) from MLPerf-Storage [29], Redis with AOF (Append Only File) persistence [14], and a Filebench [107] inspired client that replays representative file-system operations. We compare CVM-STORE in blocking mode, closing an epoch per fsync or barrier, with ext4 and Rollbaccine at $f=1$ with local metadata replication, so both are durable per sync. Figure 9 presents the results.

In the checkpointing workload, CVM-STORE's *Filesystem* and *Shared Log* reach 6.53–6.55 GB/s on saves, matching ext4 and outperforming Rollbaccine (5.61 GB/s) by 1.17×. Loading matches both baselines within 2.3% via the *Filesystem* (11.5 GB/s). The *Shared Log* trails by 15% (9.74 GB/s) because the retrieval of checkpoint entries is partially serialized on reads, not achieving full bandwidth.

On Redis, CVM-STORE outperforms Rollbaccine by 1.95–2.27× under per-operation durability, due to faster per-operation sync. Under once-per-second batching, it overtakes Rollbaccine by 1.32× at 50 clients (325 kops/s). Against ext4, it is either on par or trailing by 6–15%. CVM-STORE is integrated with Redis on its own userspace SPDK core with I/O batching, while the others go through the kernel's I/O stack.

Across four Filebench-style workloads, CVM-STORE's performance matches the expected characteristics from § IX-B. The *train*, modeled after 3D U-Net, and the fsync-barriered *audit* transfer megabytes per operation. In *train*, CVM-STORE reads 9.28 GiB/s (1.48× ext4, 1.94× Rollbaccine), while in *audit* it sustains 6.63 GiB/s (1.03× ext4, 1.63× Rollbaccine). At the small-object and high-thread scale, which CVM-STORE's design deprioritizes, it trails ext4: *web* reaches 0.76× and *mail* is its worst case at 0.29×. In these workloads, small operations cannot amortize the per-operation cost caused by partial serialization for metadata and SPDK Blobstore handling.

Takeaway 2: On data-heavy workloads, CVM-STORE's I/O characteristics carry through to full applications: it matches or outperforms both baselines on its targeted large-object workloads, trailing only on small-object and high thread count workloads.

Workload (R/W)	SPDK	MS	IO Meta	MT	Total (‰)
KVS 64 MB	80/70	0/0.2	20/53	0/0	0.10/0.13
KVS 4 GB	1.3/1.6	0/0	0.2/3.0	0/0	0.0015/0.0044
Log 64 MB	30/70	0/90	20/21	0/0	0.05/0.18
Log 4 GB	0/1.5	0/8.0	0.2/0.3	0/0	0.0002/0.0097
FS 64 MB	60/190	0/100	7.6/8.3	20/20	0.09/0.32
FS 4 GB	0/3.7	0/8.2	0.1/0.2	20/20	0.02/0.03

TABLE VII: Breakdown of read/write amplification. SPDK blob meta (SPDK), metadata sync (MS), I/O engine meta (IO Meta), and Merkle tree (MT) in 10^{-4}%; Total in ‰.

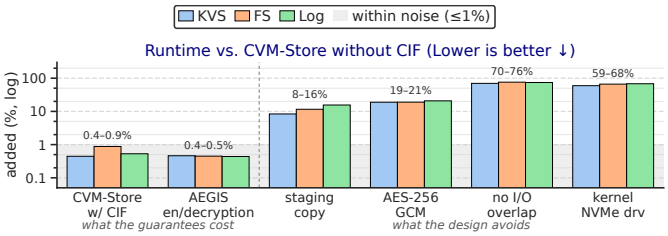


Fig. 10: Ablation of CVM-STORE's design decisions.

D. Read/Write Amplification

To measure CVM-STORE's read/write amplification with 100 ms synchronization epoch, we trace API-level operations and compare application-visible I/O against actual bytes written through the SPDK bdev layer, capturing overhead from SPDK metadata, CVM-STORE metadata, and CIF structures. We do not measure physical write amplification on the NVMe, as it is NVMe- and internal-state-specific.

Table VII presents the results. SPDK allocates a single 4 KiB metadata page per blob, and CVM-STORE adds one further block for AEAD MACs, nonces, and sizes. The rest is for global metadata and block size padding. KVS and shared log store an additional encrypted metadata block for their freshness guarantees, while the filesystem stores a Merkle tree, one leaf per chunk (<0.002%, logarithmic depth). SPDK's fragmentation-free allocation keeps amplification across workloads minimal.

Takeaway 3: CVM-STORE's abstractions incur minimal read/write amplification for their metadata and operations.

E. Ablation Study

To justify CVM-STORE's individual design decisions, we run an ablation study with a synthetic workload that reads and writes 256 GB data. We compare the following configurations: CVM-STORE with CIF, CVM-STORE without CIF, and CVM-STORE variants in which one mechanism is disabled. Figure 10 reports the runtime normalized to CVM-STORE without CIF.

First, full CIF protection ("CVM-STORE w/ CIF") adds 0.4–0.9% of runtime. Merkle tree, metadata, and trusted counter operations each contribute less than 0.3% of runtime in traces of the same runs. "AEGIS en/decryption" solely shows the encryption cost: the crypto pipeline runs in parallel with I/O and spans 51–56% of total runtime, of which only 0.4–0.5% is visible end-to-end, as the remainder overlaps with in-flight DMA.

The remaining configurations isolate the contribution of each mechanism. Simulating only the additional copy through a staging buffer ("staging copy") adds 8–16%. Substituting AES-256-GCM for AEGIS-128X4 ("AES-256-GCM") adds

	Evidence collection	Evidence appraisal	Attestation appraisal	Seal/ Unseal	Total
CVM	26.57	—	—	24.65	188.35 ms
Trusted party	—	65.95	0.76	20.73	87.43 ms

TABLE VIII: Breakdown of attestation and provisioning time.

Data Size	KVS (s) S / M / C	Filesystem (s) S / M / C	Shared log (s) S / M / C
100 MB	0.03 / 0.12 / –	0.06 / 0.25 / 0.13	0.03 / 0.05 / 0.08
1 GB	0.16 / 0.32 / –	0.16 / 0.49 / 0.14	0.15 / 0.23 / 0.07
10 GB	1.5 / 1.3 / –	1.5 / 1.3 / 0.15	1.5 / 0.9 / 0.08
100 GB	15.5 / 7.9 / –	15.1 / 8.1 / 0.14	15.0 / 8.8 / 0.08
1000 GB	150.1 / 75.2 / –	150.2 / 76.4 / 0.15	150.2 / 76.1 / 0.09

TABLE IX: Time (in seconds) to seal (S) new data to CVM-STORE, to unseal & load abstraction metadata (M), and to check consistency (C).

19–21%, making the I/O cipher-bound instead of device-bound. Serializing encryption and I/O instead of pipelining them ("no I/O overlap") inflates runtime by 70–76%. Replacing the hardened kernel-bypass data path with the guest's NVMe driver ("kernel NVMe drv") incurs a 59–68% overhead, since every request then traverses the kernel and a bounce buffer.

Takeaway 4: CVM-STORE provides CIF guarantees at 0.4–0.9% runtime overhead by interleaving cryptographic operations with in-flight DMA on a hardened kernel-bypass data path. Naively implementing any one of these parts instead could cost up to 76% runtime overhead.

F. Attestation and Sealing

We evaluate CVM-STORE's attestation and sealing mechanisms using a Veraison [7] verifier with experimental SEV-SNP support, which returns a standardized EAT Attestation Result (EAR) [49]. CVM-STORE collects CVM attestation evidence, the trusted party submits them to the verifier, and successful appraisal releases secrets (1 MiB). For provisioning, CVM-STORE runs externally to seal data before providing the symmetric key and trusted counter values to the CVM. For recovery, data is split across 1024 entries/files. The filesystem consistency check validates the superblock metadata; the shared log checks its counter chain on log opening; and the KVS folds its check into loading, rebuilding its in-memory key lookup by reading all entries. We measure CVM-side and trusted-party operations separately and report the average of three runs over a local network. This one-time startup cost is 188.35 ms on the CVM and 87.43 ms on the trusted party (Table VIII). Sealing and unsealing then scale linearly with the data, from 0.03–0.06 s at 100 MB to 150.2 s at 1 TB (Table IX). Recovery, in contrast, is asymmetric: the filesystem and shared log validate metadata only (0.07–0.15 s), while the KVS combines the check as part of loading, reading every entry on rebuild, which scales with the data size.

Takeaway 5: CVM-STORE incurs a one-time overhead (188 ms) for its attestation process and enables external data sealing for subsequent use, with standardized attestation results supporting future multi-party provisioning scenarios.

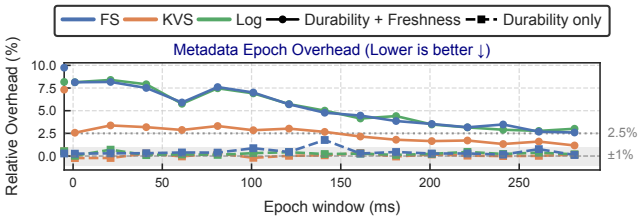


Fig. 11: Blocking-mode epoch synchronization relative overhead with and without freshness (5 ms RTT).

G. Trusted Counter

To evaluate trusted counter synchronization overhead in blocking mode, we inject a synthetic 5 ms delay, modeling network Round Trip Time (RTT) to a remote service [24] using the same workloads as in the ablation study. We vary the epoch length from 1 ms to 281 ms in 20 ms increments, and compare durability alone against durability coupled with freshness.

Figure 11 shows the relative overhead, compared to only durability at shutdown. The markers on the y-axis show the worst case of synchronizing every operation. Durability alone persists metadata without contacting the trusted party, keeping its overhead within ±1% across most workloads and epoch lengths. Coupling durability with freshness adds a round trip to the trusted party at every epoch boundary. At the worst case, this overhead is at 7.3–9.7% for all three workloads. The filesystem and shared log stay near this worst case at a 1 ms epoch (≈8%), close to 2.5% at the widest epochs. KVS overhead is lower (1–3% over all epochs), as the workload mixes reads and writes and some epochs do not entail synchronization.

Takeaway 6: In CVM-STORE, applications can trade freshness for performance by widening the epoch, taking filesystem and shared log overhead from 8–10% at per-operation synchronization down to close to 2.5% at the widest epochs.

X. RELATED WORK

Secure storage for VMs and TEEs. We discuss related work on secure storage systems for TEEs in §III. Additionally, TEE-compatible designs target secure storage in single-node [122], [36], [113], [17], [46], [118] and distributed settings [27], [71], [86], as well as secure databases and key-value stores [87], [83], [115], [62], [28], [63], [75], [88], [105]. Generic kernel-level mechanisms (dm-crypt, dm-integrity, dm-verity) provide block-level security without CVM-specific optimizations. Building on these mechanisms, the Confidential Data Hub [42] provides secure storage for confidential containers, using LUKS [13] for block device encryption or Gocryptfs [94] for object storage. However, these backends rely on the kernel I/O stack, and neither provides full integrity or freshness guarantees. In contrast, CVM-STORE provides optimized storage abstractions that support diverse access patterns within CVMs while maintaining strong CIF guarantees.

Userspace I/O in confidential computing. Userspace I/O frameworks (e.g., DPDK [48], SPDK [125]) reduce context switching overhead by directly interacting with devices [109], [48]. SPDK has been combined with SGX [109], but not adapted for CVMs, due to limited device passthrough and

DMA memory support in CVM userspace. In the network domain, Bifrost [66] leverages the fact that application-level protocols (e.g., TLS) already provide end-to-end confidentiality, thereby avoiding unnecessary encryption at the I/O layer, and other approaches demonstrate that careful software optimization can achieve near-optimal I/O performance [67]. CVM-STORE builds upon these insights to design a secure userspace I/O engine tailored for storage workloads in CVMs.

Trusted counters. CVMs do not natively support trusted counters. AMD provides SecureTSC to protect time-source values from hypervisor manipulation, but it introduces new attack vectors [60], [22] and cannot be used as a replacement. Therefore, remote trusted services have been proposed [24], [73], [74] to restore the trusted counter functionality and build replication on top. CVM-STORE addresses this by exposing a pluggable trusted counter interface through which it synchronizes with an external trusted party to provide its freshness guarantees.

Attestation. While CVM attestation performance has already been studied [79], supply chain challenges remain for endorsements and reference values [51]. RATS [32] provides standardized frameworks used by Trustee KBS [9] and Veraison [7]. CVM-STORE contributed to Veraison to attest evidence from the CVM before provisioning cryptographic materials.

Authenticated data structures. Recent work proposes skewed Merkle trees with shorter paths for frequently accessed addresses [106] and Bonsai Merkle Forests that split integrity trees to reduce overhead [50]. Additional optimizations adapt the tree layout at runtime [91], [112]. CVM-STORE only requires a simple Merkle tree for filesystem integrity verification, while avoiding the complexity of dynamic restructuring through its immutable file design.

XI. CONCLUSION

CVMs protect data in use, but leave a *storage gap*: no prior CVM storage system provides CIF while matching modern NVMe performance, forcing stateful workloads to trade security for performance. CVM-STORE closes this gap by hardening the userspace I/O trust boundary, interleaving cryptographic operations with in-flight DMA, aligning integrity structures with each abstraction's access pattern, and decoupling freshness into an externally epoch-batched counter service. With full CIF, CVM-STORE stays within ≈1% (write) and 1–19% (read) of raw NVMe bandwidth on large objects, outperforming unencrypted CVM kernel I/O and rollback-protected storage in terms of runtime overhead, while applications can tune the freshness cost through the epoch length configuration.

Artifact and appendix. CVM-STORE's source code is available at https://github.com/TUM-DSE/CVM-Store-NDSS27. The Appendix details the SPDK hardening, and the extended version of this paper [101] additionally covers the trusted counter details and the instance lifecycle.

ACKNOWLEDGMENT

This work was supported in part by an ERC Starting Grant (ID: 101077577) and the Chips Joint Undertaking (JU), European Union (EU) HORIZON-JU-IA, under grant agreement No. 101140087 (SMARTY). We also thank the Veraison and SPDK open-source projects, whose work CVM-STORE builds upon, for their input and help contributing fixes upstream.

ETHICS CONSIDERATIONS

This paper does not introduce any new vulnerability and does not involve human subjects research. Our work aims to strengthen security in confidential cloud computing, and we therefore believe this work raises no ethical concerns.

REFERENCES

[1] CVE-2023-46813. https://www.cve.org/CVERecord?id=CVE-2023-46813. Last accessed: October 2025.
[2] Enable AES and AVX instructions for SEV-SNP CVMs · Issue #12382 · kata-containers/kata-containers. https://github.com/kata-containers/kata-containers/issues/12382. Last accessed: February 2026.
[3] Fio - Flexible I/O tester rev. 3.38 — fio 3.38-38-g3ee5 documentation. https://fio.readthedocs.io/en/latest/fio_doc.html. Last accessed: July 2025.
[4] [GIT PULL] KVM changes for Linux 7.0 - Paolo Bonzini. https://lore.kernel.org/all/20260213162059.22230-1-pbonzini@redhat.com/. Last accessed: February 2026.
[5] Intel® Trust Domain Extension Guest Kernel Hardening Documentation — Intel® Trust Domain Extension. https://intel.github.io/ccc-linux-guest-hardening-docs/index.html. Last accessed: August 2026.
[6] Kernel.org/doc/Documentation/ABI/testing/configfs-tsm. https://www.kernel.org/doc/Documentation/ABI/testing/configfs-tsm. Last accessed: February 2026.
[7] Veraison. https://github.com/veraison. Last accessed: July 2025.
[8] More Information on Encrypted Memory Frameworks. https://www.intel.com/content/www/us/en/developer/articles/news/more-information-encrypted-memory-frameworks.html, 2025. Last accessed: August 2026.
[9] Trustee. Confidential Containers, July 2025. Last accessed: July 2025.
[10] Virtee/sev-snp-measure-go. VirTEE, Nov. 2025. Last accessed: February 2026.
[11] Aegis-aead/libaegis. The AEGIS authenticated encryption algorithms, Feb. 2026. Last accessed: February 2026.
[12] Aegis-aead/openssl. The AEGIS authenticated encryption algorithms, Jan. 2026. Last accessed: March 2026.
[13] cryptsetup / cryptsetup · GitLab. https://gitlab.com/cryptsetup/cryptsetup, Feb. 2026. Last accessed: February 2026.
[14] Redis persistence. https://redis.io/docs/latest/operate/oss_and_stack/management/persistence/, 2026. Last accessed: August 2026.
[15] Veraison/ratsd. Veraison, Jan. 2026. Last accessed: February 2026.
[16] Veraison/services. Veraison, Feb. 2026. Last accessed: February 2026.
[17] A. Ahmad, K. Kim, M. I. Sarfaraz, and B. Lee. OBLIVIATE: A Data Oblivious Filesystem for Intel SGX. In *Proceedings 2018 Network and Distributed System Security Symposium*, San Diego, CA, 2018. Internet Society. doi:10.14722/ndss.2018.23284.
[18] E. Aktas, C. Cohen, J. Eads, J. Forshaw, and F. Wilhelm. Intel Trust Domain Extensions (TDX) Security Review April 2023, 2023. Last accessed: March, 2025. URL: https://services.google.com/fh/files/misc/intel_tdx_-_full_report_041423.pdf.
[19] A. K. M. M. Alam and K. Chen. Making Your Program Oblivious: A Comparative Study for Side-channel-Safe Confidential Computing. In *Proceedings of the IEEE 16th International Conference on Cloud Computing*. Institute of Electrical and Electronics Engineers, 2023. doi:10.1109/CLOUD60044.2023.00040.
[20] AMD. AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More. Technical report, AMD, Jan. 2020.
[21] AMD. TECHNICAL GUIDANCE FOR MITIGATING EFFECTS OF CIPHERTEXT VISIBILITY UNDER AMD SEV REVISION 5.10.22, 2022. Last accessed: July, 2024. URL: https://www.amd.com/system/files/documents/221404394-a_security_wp_final.pdf.
[22] AMD. AMD Security Bulletin AMD-SB-3007. https://www.amd.com/en/resources/product-security/bulletin/amd-sb-3007.html, Feb. 2024. Last accessed: July 2025.
[23] AMD. Versioned Chip Endorsement Key (VCEK) Certificate and KDS Interface Specification, Jan. 2025.

[24] S. Angel, A. Basu, W. Cui, T. Jaeger, S. Lau, S. Setty, and S. Singanamalla. Nimble: Rollback Protection for Confidential Cloud Services. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 193–208, 2023.

[25] ARM. Arm Confidential Compute Architecture, [n. d.]. Last accessed: March, 2025. URL: https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture.

[26] 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 *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 689–703, 2016.

[27] M. Bailleu, D. Giantsidi, V. Gavrielatos, D. L. Quoc, V. Nagarajan, and P. Bhatotia. Avocado: A Secure In-Memory Distributed Storage System. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 65–79, 2021.

[28] M. Bailleu, J. Thalheim, P. Bhatotia, C. Fetzer, M. Honda, and K. Vaswani. SPEICHER: Securing LSM-based Key-Value Stores using Shielded Execution. *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 173–190, Feb. 2019.

[29] O. Balmau. Characterizing I/O in Machine Learning with MLPerf Storage. *ACM SIGMOD Record*, 51(3):47–48, Nov. 2022. doi:10.1145/3572751.3572765.

[30] A. Baumann, M. Peinado, and G. Hunt. Shielding Applications from an Untrusted Cloud with Haven. *ACM Transactions on Computer Systems (TOCS)*, 33(3):8:1–8:26, Aug. 2015. doi:10.1145/2799647.

[31] H. Birkholz, N. Smith, T. Fossati, and H. Tschofenig. Remote ATtestation procedureS (RATS) Conceptual Message Wrapper (CMW). Request for Comments RFC 9999, Internet Engineering Task Force, July 2026. doi:10.17487/RFC9999.

[32] H. Birkholz, D. Thaler, M. Richardson, N. Smith, and W. Pan. Remote ATtestation procedureS (RATS) Architecture. Request for Comments RFC 9334, Internet Engineering Task Force, Jan. 2023. doi:10.17487/RFC9334.

[33] BitDefender. What is Full Disk Encryption (FDE) & How It Works. https://www.bitdefender.com/en-us/business/infozone/what-is-full-disk-encryption-fde, 2026. Last accessed: February, 2026.

[34] M. Brož. *Authenticated and Resilient Disk Encryption*. Doctoral Thesis, 2018.

[35] Q. Burke, R. Sheatsley, R. King, O. Hines, M. Swift, and P. McDaniel. On Scalable Integrity Checking for Secure Cloud Disks. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pages 391–405, 2025.

[36] A. Chakraborti, B. Jain, J. Kasiak, T. Zhang, D. Porter, and R. Sion. Dm-x: Protecting Volume-level Integrity for Cloud Volumes and Local Block Devices. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, pages 1–7, Mumbai India, Sept. 2017. ACM. doi:10.1145/3124680.3124732.

[37] P.-C. Cheng, W. Ozga, E. Valdez, S. Ahmed, Z. Gu, H. Jamjoom, H. Franke, and J. Bottomley. Intel TDX Demystified: A Top-Down Approach. *ACM Comput. Surv.*, 56(9), 2024. doi:10.1145/3652597.

[38] D. Chu, A. Balasubramanian, D. Bao, N. Crooks, H. Howard, L. E. Katahanas, and S. Ponnapalli. Rollbaccine : Herd Immunity against Storage Rollback Attacks in TEEs [Technical Report], May 2025. arXiv:2505.04014, doi:10.48550/arXiv.2505.04014.

[39] J. Chuang, A. Seto, N. Berrios, S. van Schaik, C. Garman, and D. Genkin. TEE.fail: Breaking Trusted Execution Environments via DDR5 Memory Bus Interposition. In *47th IEEE Symposium on Security and Privacy (IEEE S&P '26)*. IEEE Computer Society, 2026.

[40] D. Clarke, S. Devadas, M. Van Dijk, B. Gassend, and G. E. Suh. Incremental Multiset Hash Functions and Their Application to Memory Integrity Checking. In G. Goos, J. Hartmanis, J. Van Leeuwen, and C.-S. Laih, editors, *Advances in Cryptology - ASIACRYPT 2003*, volume 2894, pages 188–207. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003. doi:10.1007/978-3-540-40061-5_12.

[41] Confidential Computing Consortium. Confidential Computing: Hardware-Based Trusted Execution for Applications and Data: November 2022, V1.3, 2022. Last accessed: September, 2025. URL: https://confidentialcomputing.io/wp-content/uploads/sites/10/2023/03/CCC_outreach_whitepaper_updated_November_2022.pdf.

[42] Confidential Containers. Confidential Containers - Secure Storage Documentation. https://github.com/confidential-containers/guest-components/blob/main/confidential-data-hub/docs/SECURE_STORAGE.md. Last accessed: July 2025.

[43] B. F. Cooper. YCSB. https://github.com/brianfrankcooper/YCSB, 2025. Last accessed: May, 2025.

[44] S. Deng, M. Li, Y. Tang, S. Wang, S. Yan, and Y. Zhang. CipherH: Automated Detection of Ciphertext Side-channel Vulnerabilities in Cryptographic Implementations. In *Proceedings of the 32nd USENIX Security Symposium*. USENIX Association, 2023.

[45] F. Denis. Adding more parallelism to the AEGIS authenticated encryption algorithms, 2023.

[46] B. Dickens, III, H. S. Gunawi, A. J. Feldman, and H. Hoffmann. StrongBox: Confidentiality, Integrity, and Performance using Stream Ciphers for Full Drive Encryption. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 708–721, Williamsburg VA USA, Mar. 2018. ACM. doi:10.1145/3173162.3173183.

[47] J. B. Djoko, J. Lange, and A. J. Lee. NeXUS: Practical and Secure Access Control on Untrusted Storage Platforms using Client-Side SGX. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 401–413, June 2019. doi:10.1109/DSN.2019.00049.

[48] DPDK Project. DPDK. https://www.dpdk.org/, 2026. Last accessed: February, 2026.

[49] T. Fossati, E. Voit, S. Trofimov, and H. Birkholz. EAT Attestation Results. Internet Draft draft-ietf-rats-ear-00, Internet Engineering Task Force, Apr. 2025. Work in Progress.

[50] A. Freij, H. Zhou, and Y. Solihin. Bonsai Merkle Forests: Efficiently Achieving Crash Consistency in Secure Persistent Memory. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, pages 1227–1240, New York, NY, USA, Oct. 2021. Association for Computing Machinery. doi:10.1145/3466752.3480067.

[51] D. Glaze. Meaningful Attestations from Supply Chain to the World by Dionna Glaze, Apr. 2025.

[52] Google. Google Cloud Confidential Computing, 2019. Last accessed: March, 2025. URL: https://cloud.google.com/confidential-computing/.

[53] T. Groß, M. Ahmadova, and T. Müller. Analyzing Android's File-Based Encryption: Information Leakage through Unencrypted Metadata. In *Proceedings of the 14th International Conference on Availability, Reliability and Security*, pages 1–7, Canterbury CA United Kingdom, Aug. 2019. ACM. doi:10.1145/3339252.3340340.

[54] J. Gunthorpe and K. Tian. IOMMUFD. https://docs.kernel.org/userspace-api/iommufd.html, 2026. Last accessed: February, 2026.

[55] F. Hetzelt, M. Radev, R. Buhren, M. Morbitzer, and J.-P. Seifert. VIA: Analyzing Device Interfaces of Protected Virtual Machines. In *Proceedings of the 37th Annual Computer Security Applications Conference*, ACSAC '21, pages 273–284, New York, NY, USA, Dec. 2021. Association for Computing Machinery. doi:10.1145/3485832.3488011.

[56] IBM. Confidential computing on IBM Cloud, 2019. Last accessed: March, 2025. URL: https://www.ibm.com/cloud/confidential-computing.

[57] Intel. Host Firmware Speculative Execution Side Channel Mitigation, 2018. Last accessed: July, 2024. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/host-firmware-speculative-side-channel-mitigation.html.

[58] Intel. Guidelines for Mitigating Timing Side Channels Against Cryptographic Implementations, 2022. Last accessed: July, 2024. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/secure-coding/mitigate-timing-side-channel-crypto-implementation.html.

[59] Intel. Intel® Trust Domain Extensions (Intel TDX), [n. d.]. Last accessed: March, 2025. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html.

[60] J. Juffinger, S. R. Neela, and D. Gruss. Not So Secure TSC. In M. Fischlin and V. Moonsamy, editors, *Applied Cryptography and Network Security*, pages 3–23, Cham, 2025. Springer Nature Switzerland. doi:10.1007/978-3-031-95764-2_1.

[61] L. Kang, Y. Xue, W. Jia, X. Wang, J. Kim, C. Youn, M. J. Kang, H. J. Lim, B. Jacob, and J. Huang. IceClave: A Trusted Execution Environment for In-Storage Computing. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, pages 199–211, New York, NY, USA, Oct. 2021. Association for Computing Machinery. doi:10.1145/3466752.3480109.

[62] I. Kim, J. H. Kim, M. Chung, H. Moon, and S. H. Noh. A Log-Structured Merge Tree-aware Message Authentication Scheme

for Persistent Key-Value Stores. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 363–380, 2022.

[63] T. Kim, J. Park, J. Woo, S. Jeon, and J. Huh. ShieldStore: Shielded In-memory Key-value Storage with SGX. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 1–15, New York, NY, USA, Mar. 2019. Association for Computing Machinery. doi:10.1145/3302424.3303951.

[64] R. Krahn, B. Trach, A. Vahldiek-Oberwagner, T. Knauth, P. Bhatotia, and C. Fetzer. Pesos: Policy enhanced secure object store. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 1–17, New York, NY, USA, Apr. 2018. Association for Computing Machinery. doi:10.1145/3190508.3190518.

[65] W. Lam. Boosting the Network Performance of Confidential VM Using Userspace Stack, Sept. 2024.

[66] 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 *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 1–15, 2023.

[67] M. Li, S. Srivastava, and M. Yan. Bridge the Future: High-Performance Networks in Confidential VMs without Trusted I/O devices, Mar. 2024. arXiv:2403.03360, doi:10.48550/arXiv.2403.03360.

[68] C. Lin, S. Nadi, and H. Khazaei. A Large-scale Data Set and an Empirical Study of Docker Images Hosted on Docker Hub. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 371–381, Sept. 2020. doi:10.1109/ICSME46990.2020.00043.

[69] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, page 133, 2016.

[70] L. Lundblade, G. Mandyam, J. O'Donoghue, and C. Wallace. The Entity Attestation Token (EAT). Request for Comments RFC 9711, Internet Engineering Task Force, Apr. 2025. doi:10.17487/RFC9711.

[71] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud Storage with Minimal Trust. *ACM Transactions on Computer Systems*, 29(4):1–38, Dec. 2011. doi:10.1145/2063509.2063512.

[72] M. Mahhouk. MoeMahhouk/gramine-sealing-key-provider, Apr. 2025. Last accessed: July 2025.

[73] A. Martin, C. Lian, F. Gregor, R. Krahn, V. Schiavoni, P. Felber, and C. Fetzer. ADAM-CS: Advanced Asynchronous Monotonic Counter Service. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 426–437, June 2021. doi:10.1109/DSN48987.2021.00053.

[74] S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun. ROTE: Rollback Protection for Trusted Execution. In *26th USENIX Security Symposium (USENIX Security 17)*, page 1289, 2017.

[75] I. Messadi, S. Neumann, N. Weichbrodt, L. Almstedt, M. Mahhouk, and R. Kapitza. Precursor: A fast, client-centric and trusted key-value store using RDMA and Intel SGX. In *Proceedings of the 22nd International Middleware Conference*, pages 1–13, Québec City, Canada, Nov. 2021. ACM. doi:10.1145/3464298.3476129.

[76] J. D. Meulemeester, D. Oswald, I. Verbauwhede, and J. Van Bulck. Battering RAM: Low-Cost Interposer Attacks on Confidential Computing via Dynamic Memory Aliasing. In *47th IEEE Symposium on Security and Privacy (IEEE S&P '26)*, May 2026.

[77] N. Michael, J. Mink, J. Liu, S. Gaur, W. U. Hassan, and A. Bates. On the Forensic Validity of Approximated Audit Logs. In *Annual Computer Security Applications Conference*, pages 189–202, Austin USA, Dec. 2020. ACM. doi:10.1145/3427228.3427272.

[78] Microsoft. Azure Confidential Computing, 2019. Last accessed: March, 2025. URL: https://azure.microsoft.com/en-us/solutions/confidential-compute.

[79] M. Misono, D. Stavrakakis, N. Santos, and P. Bhatotia. Confidential VMs Explained: An Empirical Analysis of AMD SEV-SNP and Intel TDX. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 8(3):1–42, Dec. 2024. doi:10.1145/3700418.

[80] A. Mohan, R. Walkup, B. Karacali, M.-h. Chen, A. Kayi, L. Schour, S. Salaria, S. Wen, I.-h. Chung, A. Alim, C. Evangelinos, L. Luo, M. Dombrowa, L. Schares, A. Sydney, P. Maniotis, S. Koteshwara, B. Tang, J. Belog, R. Odaira, V. Tarasov, E. Gampel, D. Thorstensen, T. Gershon, and S. Seelam. Vela: A Virtualized LLM Training System with GPU Direct RoCE. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 1348–1364, Rotterdam Netherlands, Mar. 2025. ACM. doi:10.1145/3676641.3716280.

[81] L.-D. Nguyen and V. Leis. Why Files If You Have a DBMS? In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pages 3878–3892, Utrecht, Netherlands, May 2024. IEEE. doi:10.1109/ICDE60146.2024.00297.

[82] NVM Express®. NVM Express® Base Specification. https://nvmexpress.org/specification/nvm-express-base-specification/, 2026. Last accessed: February, 2026.

[83] A. Papadimitriou, R. Bhagwan, N. Chandran, R. Ramjee, A. Haeberlen, H. Singh, A. Modi, and S. Badrinarayanan. Big Data Analytics over Encrypted Datasets with Seabed. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 587–602, 2016.

[84] B. Parno, J. R. Lorch, J. R. Douceur, J. Mickens, and J. M. McCune. Memoir: Practical state continuity for protected modules. In *2011 IEEE Symposium on Security and Privacy*, pages 379–394, 2011. doi:10.1109/SP.2011.38.

[85] PCI-SIG. PCI Express® Base Specification Revision 6.3, Dec. 2024.

[86] R. A. Popa, J. R. Lorch, D. Molnar, H. J. Wang, and L. Zhuang. Enabling Security in Cloud Storage SLAs with CloudProof. In *2011 USENIX Annual Technical Conference (USENIX ATC 11)*, 2011.

[87] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 85–100, Cascais Portugal, Oct. 2011. ACM. doi:10.1145/2043556.2043566.

[88] C. Priebe, K. Vaswani, and M. Costa. EnclaveDB: A Secure Database Using SGX. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 264–278, San Francisco, CA, May 2018. IEEE. doi:10.1109/sp.2018.00025.

[89] L. Qiu, R. Taft, A. Shraer, and G. Kollios. The Price of Privacy: A Performance Study of Confidential Virtual Machines for Database Systems. In *Proceedings of the 20th International Workshop on Data Management on New Hardware*. Association for Computing Machinery, 2024. URL: https://doi.org/10.1145/3662010.3663440.

[90] D. L. Quoc, F. Gregor, S. Arnautov, R. Kunkel, P. Bhatotia, and C. Fetzer. secureTF: A Secure TensorFlow Framework. In *Proceedings of the 21st International Middleware Conference*, Middleware '20, pages 44–59, New York, NY, USA, Dec. 2020. Association for Computing Machinery. doi:10.1145/3423211.3425687.

[91] J. Rakshit and K. Mohanram. ASSURE: Authentication Scheme for SecURE energy efficient non-volatile memories. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2017. doi:10.1145/3061639.3062205.

[92] J. Raman. Jraman567/go-gen-ref, Mar. 2025. Last accessed: February 2026.

[93] E. Reshetova and C. Bilbao. Confidential Computing in Linux for x86 virtualization, 2023. Last accessed: March, 2025. URL: https://docs.kernel.org/security/snp-tdx-threat-model.html.

[94] rfjakob. gocryptfs: an encrypted overlay filesystem written in Go. https://github.com/rfjakob/gocryptfs. Last accessed: February, 2026.

[95] D. Rodríguez-Silva, L. Adkinson-Orellana, F. González-Castaño, I. Armiño-Franco, and D. González-Martínez. Video Surveillance Based on Cloud Storage. In *2012 IEEE Fifth International Conference on Cloud Computing*, pages 991–992, June 2012. doi:10.1109/CLOUD.2012.44.

[96] M. Roth. KVM: Introduce KVM_EXIT_SNP_REQ_CERTS for SNP certificate-fetching, Jan. 2026. Last accessed: February 2026.

[97] M. Roth. KVM: SEV: Add KVM_SEV_SNP_ENABLE_REQ_CERTS command, Jan. 2026. Last accessed: February 2026.

[98] M. Rybczyńska. Bounce Buffers for Untrusted Devices [LWN.net], 2019. Last accessed: March, 2025. URL: https://lwn.net/Articles/786558/.

[99] A. Segall. Using the TPM: Machine Authentication and Attestation, Oct. 2012.

[100] L. A. B. Silva, C. Costa, and J. L. Oliveira. A PACS archive architecture supported on cloud services. *International Journal of Computer Assisted Radiology and Surgery*, 7(3):349–358, May 2012. doi:10.1007/s11548-011-0625-x.

[101] T. Sommer, M. Misono, D. Stavrakakis, and P. Bhatotia. CVM-Store: A Secure Storage System for Confidential Virtual Machines, Oct. 2026. arXiv:2610.34560, doi:10.48550/arXiv.2610.34560.

[102] SPDK community. Blobstore Programmer's Guide. https://spdk.io/doc/blob.html, 2026. Last accessed: February, 2026.

[103] D. Stavrakakis, D. Giantsidi, M. Bailleu, P. Sändig, S. Issa, and P. Bhatotia. Anchor: A library for building secure persistent memory systems. *Proc. ACM Manag. Data*, 1(4), Dec. 2023. doi:10.1145/3626718.

[104] J. Sun. [PATCH v4 0/2] PCI: Fix the issue of failed speed limit lifting, Jan. 2025. Last accessed: February 2026.

[105] Y. Sun, S. Wang, H. Li, and F. Li. Building enclave-native storage engines for practical encrypted databases. *Proceedings of the VLDB Endowment*, 14(6):1019–1032, Feb. 2021. doi:10.14778/3447689.3447705.

[106] J. Szefer and S. Biedermann. Towards fast hardware memory integrity checking with skewed Merkle trees. In *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '14, pages 1–8, New York, NY, USA, June 2014. Association for Computing Machinery. doi:10.1145/2611765.2611774.

[107] V. Tarasov, E. Zadok, and S. Shepler. Filebench: A Flexible Framework for File System Benchmarking. *login Usenix Mag.*, 2016.

[108] L. Ternullo. IDE and TDISP: An Overview of PCIe® Technology Security Features. https://pcisig.com/blog/ide-and-tdisp-overview-pcie%C2%AE-technology-security-features, 2026. Last accessed: February, 2026.

[109] 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 Sixteenth European Conference on Computer Systems*, EuroSys '21, pages 490–506, New York, NY, USA, Apr. 2021. Association for Computing Machinery. doi:10.1145/3447786.3456255.

[110] The kernel development community. dm-crypt. https://docs.kernel.org/admin-guide/device-mapper/dm-crypt.html, 2026. Last accessed: February, 2026.

[111] The kernel development community. VFIO - "Virtual Function I/O". https://docs.kernel.org/driver-api/vfio.html, 2026. Last accessed: February, 2026.

[112] S. Thomas, K. Workneh, J. McCarty, J. Izraelevitz, T. Lehman, and R. I. Bahar. A Midsummer Night's Tree: Efficient and High Performance Secure SCM. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, volume 3 of *ASPLOS '24*, pages 22–37, New York, NY, USA, Apr. 2024. Association for Computing Machinery. doi:10.1145/3620666.3651354.

[113] H. Tian, X. Yu, S. Song, Q. Chen, Z. Zhang, S. Wang, W. Liu, E. Xu, S. Yan, and Y. Zhang. AtomicDisk: A Secure Virtual Disk for TEEs against Eviction Attacks. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pages 449–459, 2025.

[114] Trusted Computing Group. TCG Storage Security Subsystem Class (SSC): Opal Version 2.30. Technical report, Trusted Computing Group, Jan. 2025.

[115] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich. Processing analytical queries over encrypted data. *Proceedings of the VLDB Endowment*, 6(5):289–300, Mar. 2013. doi:10.14778/2535573.2488336.

[116] G. Wang, O. Ruwase, B. Xie, and Y. He. FastPersist: Accelerating Model Checkpointing in Deep Learning, June 2024. arXiv:2406.13768, doi:10.48550/arXiv.2406.13768.

[117] W. Wang, M. Li, Y. Zhang, and Z. Lin. PwrLeak: Exploiting Power Reporting Interface for Side-Channel Attacks on AMD SEV. In *Proceedings of the 20th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer-Verlag, 2023.

[118] C. Weinhold. jVPFS: Adding Robustness to a Secure Stacked File System with Untrusted Local Storage Components. In *2011 USENIX Annual Technical Conference (USENIX ATC 11)*, 2011.

[119] B. Welch and G. Noer. Optimizing a hybrid SSD/HDD HPC storage system based on file size distributions. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12, Long Beach, CA, USA, May 2013. IEEE. doi:10.1109/MSST.2013.6558449.

[120] J. Wichelmann, A. Pätschke, L. Wilke, and T. Eisenbarth. Cipherfix: Mitigating Ciphertext Side-Channel Attacks in Software. In *Proceedings of the 32nd USENIX Security Symposium*. USENIX Association, 2023.

[121] H. Wu and B. Preneel. AEGIS: A Fast Authenticated Encryption Algorithm. In *Revised Selected Papers on Selected Areas in Cryptography – SAC 2013 - Volume 8282*, pages 185–201, Berlin, Heidelberg, Aug. 2013. Springer-Verlag. doi:10.1007/978-3-662-43414-7_10.

[122] E. Xu, X. Yu, L. Yin, X. Luo, S. Song, Q. Chen, S. Yan, J. Shu, H. Tian, and Y. Zhang. MlsDisk: Trusted Block Storage for TEEs Based on Layered Secure Logging. In *24th USENIX Conference on File and Storage Technologies (FAST 26)*, pages 443–459, 2026.

[123] T. Xu, L. M. Botelho, and F. X. Lin. VStore: A Data Store for Analytics on Large Videos. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 1–17, New York, NY, USA, Mar. 2019. Association for Computing Machinery. doi:10.1145/3302424.3303971.

[124] Z. Yang. Fast and Secure: DPDK Meets Confidential Computing, Sept. 2023.

[125] 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 *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 154–161. IEEE, 2017. doi:10.1109/CloudCom.2017.14.

[126] E. Zadok, I. Badulescu, and A. Shender. Cryptfs: A Stackable Vnode Level Encryption File System. 1998. technical report. doi:10.7916/D82N5935.

[127] C. Zhang, J. Xia, B. Yang, H. Puyang, W. Wang, R. Chen, I. E. Akkus, P. Aditya, and F. Yan. Citadel: Protecting Data Privacy and Model Confidentiality for Collaborative Learning. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, pages 546–561, New York, NY, USA, Nov. 2021. Association for Computing Machinery. doi:10.1145/3472883.3486998.

[128] N. Zhao, V. Tarasov, H. Albahar, A. Anwar, L. Rupprecht, D. Skourtis, A. S. Warke, M. Mohamed, and A. R. Butt. Large-Scale Analysis of the Docker Hub Dataset. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–10, Albuquerque, NM, USA, Sept. 2019. IEEE. doi:10.1109/CLUSTER.2019.8891000.

APPENDIX A
SPDK HARDENING & DMA ALLOCATIONS

We list the key SPDK structures that require DMA-accessible memory in Table X. They cover NVMe queue management, blobstore metadata, and I/O buffer pools. We identified these allocations by tracing SPDK memory allocations with GDB and manually cross-referencing the source code.

To harden the NVMe driver, we added bounds checks to the completion queues and split the `nvme_tracker` struct so that only the required data is allocated in DMA memory. CVM-STORE uses SPDK's `iobuf` sized accordingly and allocated in DMA shared memory with its own module.

Structure	Frequency and description
`pqpair->cmd` (`spdk_nvme_cmd[]`)	[Per queue pair] NVMe submission queue
`pqpair->cpl` (`spdk_nvme_cpl[]`)	[Per queue pair] NVMe completion queue
`pqpair->dma_bufs` (`nvme_dma_prp_sgl_buffer[]`)	[Per queue pair] PRP/SGL scatter-gather descriptors (4 KB, 4 KB-aligned), split from `nvme_tracker`, which is now private
`spdk_bs_super_block`	[On load/sync] Blobstore metadata (version, cluster size)
`spdk_bs_md_mask`	[Periodic] Used-pages/blobids/clusters bitmasks (458 KB ea.)
`spdk_blob_md_page`	[Per opened blob] Blob extent maps and cluster metadata (4 KB)
`spdk_nvme_ctrlr_data` / `spdk_nvme_ns_data`	[Per identify command] Identify controller/namespace payload (4 KB)
`small_pool_base` / `large_pool_base`	[Once (init)] SPDK's modules with default names (unused), CVM-STORE's DMA buffers (resized based on configuration, e.g., 180 MiB single-threaded w/ 1 MiB chunks)
`g_bdev_mgr.zero_buffer`	[Once (init)] Zero buffer for write-zero ops (1 MB)

TABLE X: Selection of SPDK main DMA memory allocations. All structures need to be allocated in DMA memory.

APPENDIX B
ARTIFACT APPENDIX

This artifact contains the implementation and the scripts to reproduce the experiments, figures, and tables of the paper. It consists of the CVM-STORE library and its example clients, the patched guest kernel, the CVM-enabled DPDK/SPDK forks, the applications ported to CVM-STORE (Redis, MLPerf Storage), and the benchmarking framework. The framework builds the guest images, boots AMD SEV-SNP guests with NVMe passthrough, and automates the measurements: the fio and crypto motivation experiments, the CVM-STORE evaluation against ext4 and Rollbaccine (I/O characteristics, ablation, amplification, epochs, seal/unseal recovery), and the end-to-end MLPerf Storage, Redis and attestation benchmarks.

A. Description & Requirements

1) How to access: The artifact is publicly available on GitHub at https://github.com/TUM-DSE/CVM-Store-NDSS27 (DOI: 10.5281/zenodo.23703217). This repository is the evaluation harness and pins every other component (CVM-STORE, DPDK/SPDK forks, guest kernel patches, MLPerf Storage and Redis forks, and the orchestration framework) as Nix flake inputs, so all components are built reproducibly.

2) Hardware dependencies:

- An AMD SEV-SNP enabled server (EPYC Zen 4 and up, we mainly used EPYC 9334)
- At least 64 physical cores and 320 GB of DDR5 RAM.
- A dedicated KIOXIA CM7-R PCIe Gen 5 NVMe SSD or similar with 3.8 TB or more. It is erased by the experiments and must not hold any data. A slower or Gen 4 NVMe can demonstrate the functionality, not the absolute numbers. NVMe NUMA and guest configuration should match.
- Up to about 400 GB of free disk space for traces and build artifacts.

3) Software dependencies:

- A Linux host with SEV-SNP support (7.2 Linux kernel upstream, or older 6.18 with custom patches), IOMMU and `vfio-pci` included. Hugepages should be allocatable on the host.
- Nix with flakes and ccache enabled. Everything else is built via Nix directly.
- Docker on the host, for the MLPerf Storage container in the guest image and Veraison.
- Root access, for booting VMs and binding the NVMe to VFIO.

4) Benchmarks: All workloads are included in the artifact repository or generated reproducibly.

B. Artifact Installation & Configuration

1) Install Nix with flakes on the host and clone `CVM-Store-NDSS27`, then enter the shell with `nix develop`.
2) Add the host to `config.toml` with the correct NVMe.
3) Build OVMF, QEMU, and the guest image with MLPerf: `oma-arvio build.all` and `oma-arvio accelstore.build-guest --mlperf`
4) Set up Veraison for attestation: `oma-arvio veraison.setup`, then check with `oma-arvio veraison.status` that every service is up.
5) Smoke test that boots a SEV-SNP guest with NVMe: `oma-arvio-root vm.start --type snp --size medium -u`

C. Major Claims

The artifact supports the following claims of the paper. C1 covers the motivation, C2 to C7 correspond to the evaluation's Takeaways 1 to 6, and each names the experiment and figure or table that it relates to.

- (C1) CVM kernel I/O loses at least 10–23% of VM throughput, authenticated block-level encryption (AES-XTS+sha256) a further 37–56% of read bandwidth, while AEGIS exceeds NVMe bandwidth on one core. (E1), Figures 2–3, Tables II and III.
- (C2) For large objects, one core with full CIF is within ≈1% (write) and 1–19% (read) of raw NVMe bandwidth, while single-threaded ext4 and Rollbaccine reach 18–30% on reads (Takeaway 1). (E2), Figure 8.
- (C3) These characteristics carry through to applications: CVM-STORE matches or outperforms both baselines on large-object workloads and trails only on small-object or high-thread-count ones (Takeaway 2). (E3), Figure 9.
- (C4) Read/write amplification of the three abstractions stays below 1‰ (Takeaway 3). (E4), Table VII.
- (C5) Full CIF costs ≈1% runtime, while naively implementing any one part costs up to 76% (Takeaway 4). (E4), Figure 10.
- (C6) Attestation is a one-time overhead; sealing scales linearly with data (Takeaway 5). (E5), Tables VIII–IX.
- (C7) Widening the epoch takes filesystem and shared-log overhead from 8–10% down to ≈2.5% (Takeaway 6). (E6), Figure 11.

D. Evaluation

Every figure and table of the paper is tied to one `paper.<name>` task that runs its experiment, exports results into `bench-result/parsed/`, and figures into `bench-result/plots/`. All runs are automated, with minimal human intervention required. `oma-arvio-root paper.all --quick` runs a representative subset of them in about a day. Without `--quick`, the full set may take several weeks, mainly because of the fio runs, which require the NVMe to reach steady state. The `README.md` in the artifact repository lists the exact differences between the quick subset and the paper, and a runtime breakdown of the tasks. Quick runs use less data, so the absolute numbers may differ from those in the paper. However, the ratios between the arms and the shapes of the curves are expected to be similar on matching hardware. The following table shows the experiments and tasks that are part of it.

Task `paper.*`		Output	Paper
Motivation			
E1	`crypto`*	`crypto-summary.csv`	Tab. III
	`fio-iops`	`fio-iops_iops.pdf`	Fig. 2
	`fio-stacks`	`fio-luks_iops.pdf`, `fio-bandwidth.csv`	Fig. 3, Tab. II
I/O analysis of the three abstractions			
E2	`synthetic`*	`synthetic.pdf`	Fig. 8
End-to-end applications			
E3	`apps`*	`apps.pdf`	Fig. 9
Ablation and amplification study			
E4	`ablation`*	`ablation.pdf`	Fig. 10
	`amplification`*	`amplification.csv`	Tab. VII
Attestation, sealing and recovery			
E5	`attestation`*	`attestation-summary.csv`	Tab. VIII
	`seal-unseal`*	`seal-unseal-summary.csv`	Tab. IX
Trusted counter epochs			
E6	`epoch`*	`epoch-overhead.pdf`	Fig. 11

* included in `paper.all --quick`

Prerequisites:

- All: make sure the build steps are done and no one else is using the NVMe.
- E1: the two fio tasks precondition the whole SSD (only run if time permits).
- E3: requires MLPerf in the guest image. If the runs fail, rebuild the guest image.
- E5: `paper.attestation` requires Veraison running on the host. Check with `oma-arvio veraison.status`.

E. Customization

The `README.md` and `docs/benchmarks.md` of the artifact repository describe how to run every benchmark family manually, including how to customize the VM size, the storage backends, crypto variants, durability and freshness settings, and workloads.