

SHMemora: Protective Key-Value Store on Distributed Shared Memory

Jiajun Luo^{†‡}, Siyu Lin[†], Yunpeng Xu[†], Shengwei Liu[§], Jin Xia[¶], Dong Liu[¶]

Zheng Liu^{‡||}, Huanchen Zhang[†], Teng Ma[‡], Shuwen Deng^{†‡‡**}

[†]Tsinghua University, [‡]Alibaba Group, [§]Cornell University

[¶]Shenzhen Longsys Electronics Co., Ltd., ^{||}Zhejiang University, ^{‡‡}Zhongguancun Laboratory

{luo-jj23, xyp24}@mails.tsinghua.edu.cn, xiaolin0508@outlook.com, sl3597@cornell.edu,

{jin.xia, tomas.liu}@longsys.com, {wenqing.lz, sima.mt}@alibaba-inc.com, {huanchen, shuwend}@tsinghua.edu.cn

Abstract— Modern in-memory key-value stores (IMKVS) are essential for data-intensive applications but increasingly limited by memory capacity. Traditional scale-out over TCP/IP or RDMA incurs high overhead from network latency and data movement. Emerging Compute Express Link (CXL) enables near-local-latency shared memory across hosts, offering a promising path for scaling IMKVS. However, existing CXL-based systems prioritize performance and overlook security, leaving shared memory exposed to unauthorized access. In this work, we present SHMemora, to the best of our knowledge, the first protective multi-host IMKVS built on CXL-based shared memory in coherent Distributed Shared Memory (DSM) environments. SHMemora addresses three key challenges, including strict protection enforcement, fine-grained memory protection, and scalable metadata management. These are solved by SHMemora through Intel MPK-based memory isolation, a capability-based protection, and metadata batching, respectively. We implement SHMemora on real CXL hardware and evaluate its security through formal proofs and penetration tests, and its performance through microbenchmarks and real workloads, demonstrating both strong protection guarantees and practical scalability. SHMemora achieves an average of $3.45\times$ throughput compared to Redis under multiple hosts, incurring an average of 1.68% throughput overhead for the security protection.

I. INTRODUCTION

Modern in-memory key-value stores (IMKVS) play an important role to data-intensive applications such as social networks, e-commerce, and web search [1], [2]. For low latency and high throughput, these systems typically keep the entire working set in DRAM to allow efficient data sharing among entities. However, as emerging workloads drive rapid data growth, the memory capacity requirements of IMKVS often exceed the physical limitations of a single server. Traditional scale-out approaches expand capacity over network using protocols like TCP/IP [3] and RDMA [4]. While effective, these methods incur significant remote access latency and inter-node communication overhead [5]. This highlights the need for more efficient memory scaling architectures.

Compute Express Link (CXL) has recently emerged as a promising alternative, enabling multiple hosts to share a disaggregated memory pool [6]. CXL allows hosts to maintain memory semantics over a globally shared memory space via the PCIe fabric, which is orthogonal to local DDR memory.

Leveraging CXL-based shared memory, IMKVS can transparently expand their addressable memory beyond a single machine, while preserving direct load/store access semantics with latency approaching that of local DDR. This shared memory architecture facilitates the construction of high-performance, memory-rich IMKVS systems that elastically scale across racks without compromising latency.

While emerging CXL-based hardware enables efficient rack-scale shared memory, existing research largely overlooks security concerns. With commercial CXL 2.0 devices now available, systems can already support basic memory sharing across hosts. Future versions of CXL are expected to further improve efficiency by enabling hardware-managed coherent access across hosts. These hardware advancements have spurred a wave of research into CXL-based distributed shared memory (DSM) systems [7]–[11]. This system enables rack-scale IMKVS where threads on any host can directly access any key-value (KV) pair [8]. However, these studies have focused primarily on functionality and performance, largely overlooking the critical aspect of security.

Neglecting the security implications of applying CXL disaggregation to IMKVS can lead to serious vulnerabilities. Without additional safeguards, a rogue thread on any host can read or corrupt any shared IMKVS data, leaking secrets or violating invariants. Standard OS isolation is ineffective because the shared memory is mapped into each host's address space, and currently no hardware privilege (e.g., CHERI-like tags [12]) is used to enforce protection. Regarding the CXL specification, its coarse protection model falls short of modern DSM workloads' thread-level isolation needs [6], [8]. This gap is further exacerbated by the unavailability of commercial CXL 3.0/3.1 hardware. In this case, there remains a critical gap of a fine-grained secure isolation in IMKVS when deployed on CXL-based shared memory systems. Otherwise, there will be data leakage when multiple cloud tenants run their IMKVS instances on a shared CXL memory pool [5].

In this work, we present SHMemora, to the best of our knowledge, **the first protective framework to build multi-host IMKVS over CXL-based shared memory in coherent DSM environments**. SHMemora proposes a symmetric peer-to-peer shared memory model to mitigate the network and data movement overhead inherent in traditional client-server

**Shuwen Deng is the corresponding author.

architectures (e.g., Redis [13], Memcached [14]), and to better align with the low-latency, byte-addressable nature of CXL-based multi-host shared memory.

There are several critical challenges to building such a framework: (C1)¹ *Protection enforcement*. All memory accesses must go through the protection layer, requiring strict bypass prevention. (C2) *Fine-grained memory protection*. CXL's native isolation mechanisms are coarse-grained and host-centric, lacking intra-block and per-thread isolation. (C3) *Scalable data management*. Managing shared memory access across hosts introduces contention on data and metadata.

To address these challenges, we (S1) enforce mandatory protection by isolating all shared memory regions with Intel Memory Protection Key (MPK), ensuring that only trusted APIs can access shared memory through controlled and verifiable paths. We then (S2) introduce a capability-based protection mechanism that provides fine-grained, thread-level permissions over arbitrarily sized memory regions. Furthermore, to enhance scalability, we (S3) partition the data across hosts and apply batching to reduce metadata contention.

Besides, SHMemora introduces two sets of APIs: one for IMKVS operations and another for shared memory access. IMKVS threads across hosts interact through the IMKVS APIs, which internally invoke the shared memory APIs that enforce the protection mechanism.

To evaluate the security of SHMemora, we formalize the system and threat model, and provide proofs of metadata and data protection. We also conduct penetration tests demonstrating resilience against practical attacks, including token forgery, metadata tampering, and out-of-bounds access.

To evaluate the performance of SHMemora, we implement it on real CXL platforms with CXL 2.0-compliant ASIC devices that support memory sharing across multiple hosts. We evaluate the system using a combination of microbenchmarks and macro workloads such as Yahoo! Cloud Serving Benchmark (YCSB) [15] to characterize the performance of SHMemora, focusing on both the performance advantage over traditional client-server models and the lightweight overhead introduced by the protection layer. Compared to Redis with pipelining disabled and without using MSET/MGET operations, SHMemora achieves $3.45\times$ the throughput and reduces latency by a factor of 3.25. The protection mechanism incurs only an average of 1.68% overhead for both throughput and latency compared with the unprotected SHMemora version. Under YCSB workloads, SHMemora incurs an average overhead of 1.71% compared to the unprotected SHMemora.

In summary, we make the following contributions.

- We propose a shared memory model for multi-host IMKVS in the modern DSM era.
- To the best of our knowledge, we are the first to address the fine-grained memory protection challenges in emerging CXL-based DSM.
- We design a lightweight, capability-based protection mechanism that enforces permissions over arbitrarily

sized KV regions with minimal overhead on real CXL shared memory machines.

- We provide formal proofs that SHMemora ensures the protection of metadata and data, and conduct penetration tests to demonstrate its resilience against practical attacks.
- We evaluate the system using microbenchmarks and real-world workloads, demonstrating that the shared memory model outperforms traditional client-server models by $3.45\times$ the throughput under the 12-host configuration, while protection introduces with only 1.68% average throughput overhead.

II. BACKGROUND

A. Distributed Shared Memory

Distributed Shared Memory (DSM) [16]–[26] provides a uniform memory abstraction across distributed nodes, enabling applications to access shared memory regions as if they were local memory. DSM simplifies the development of distributed applications by making remote memory access transparent.

B. Compute Express Link

Compute Express Link (CXL) is an open standard interconnect designed to enhance data-centric applications by enabling high-speed, low-latency communication between CPU, accelerators, memory, and storage. CXL leverages the PCI Express (PCIe) to provide a unified, coherent memory space, allowing for efficient sharing of resources and improved scalability.

CXL devices have three types. Type-1 connects to host DDR memory, Type-2 shares memory with the processor, and Type-3 enables memory expansion and pooling. This work focuses on Type-3 devices which increase DRAM capacity and bandwidth by adding memory without consuming DIMM slots. CXL memory retains a byte-addressable interface, allowing the OS to manage it like standard system memory.

C. CXL-based DSM

CXL has enabled raw memory sharing across hosts since version 2.0, utilizing Multi-Headed Devices (MHD). CXL 3.0 further introduces Global-Fabric-Attached Memory (G-FAM), a highly scalable form of Host-Managed Device Memory (HDM) that supports hardware coherence. This is achieved through the Back-Invalidate (BI) mechanism. When a server writes to shared memory cached by other hosts, the BI mechanism issues invalidation requests to those hosts. After they acknowledge invalidation of their cached copies, the write is committed.

Industrial hardware. Our work is deployed and evaluated on XConn XC50256, the world's first CXL 2.0 switch chip [27]. This CXL switch-based platform has been utilized in numerous cutting-edge research projects and shows promise for extensive commercial application [28]. Although commercial CXL 3.0 hardware is not yet available, our design is forward compatible with upcoming CXL 3.0 devices.

Isolation in CXL specification. CXL specification [29] for G-FAM defines a basic access protection framework, relying on a two-level permission model as shown in Figure 1.

¹Here, C1 denotes Challenge 1, S1 denotes Solution 1, and so forth.

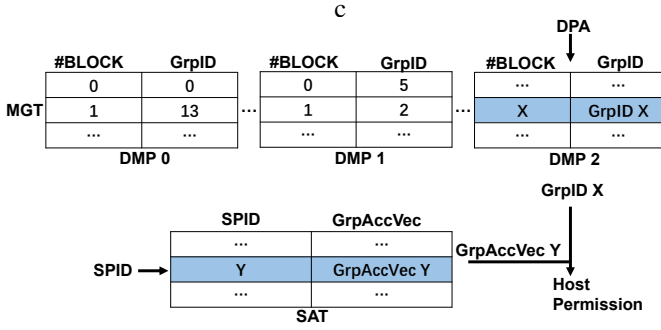


Fig. 1: G-FAM offers protection at host granularity with fixed block size. Each Device Media Partition (DMP), such as DRAM or NVM, maintains a Memory Group Table (MGT) dividing memory into fixed-size blocks with Group IDs (GrpIDs). For a request, the device uses the Device Physical Address (DPA) to identify the DMP, retrieves the GrpID from the MGT, and checks permissions via the Group Access Vector (GrpAccVec) from the SPID Access Tables (SAT). Indexed by the Source Port ID (SPID) of the requesting host, the SAT encodes permissions in the GrpAccVec bit-vector, where each bit grants access to its corresponding GrpID.

However, even for this future design, the protection remains coarse-grained, limited to the host level with a minimum data granularity of 64 [29], and it cannot isolate accesses within individual memory blocks. Moreover, the number of memory ranges a host can access per device is limited [30], further reinforcing the coarse-grained nature of the protection.

Other CXL security features. Beyond isolation, CXL adds security features in versions 3.0–3.2 [29], mainly on the Trusted Security Protocol (TSP). TSP extends Integrity and Data Encryption (IDE) from CXL 2.0 to handle larger flits, complex topologies, multi-fabric attestation, and vendor readiness in CXL 3.0. CXL 3.1 expands TSP to cover direct-attached memory in confidential computing, while CXL 3.2 strengthens TSP for Host-only Coherent Device-Managed Memory with Back Invalidation (HDM-DB) and improves compliance. TSP hardware features secure memory against privileged software and CXL devices.

In contrast, SHMemora works at a higher layer, enforcing isolation of participating entities within the shared memory pool, while TSP protects memory from external threats. The two are orthogonal yet complementary and fully compatible.

III. THREAT MODEL

The IMKVS system can face variant threats from physics-level, system-level, thread-level, or microarchitecture-level attackers in an open-world environment.

Similar to previous works [31], [32], SHMemora considers a widely-adopted threat model of all unprivileged thread-level attackers co-located on the same host (including IMKVS threads), including both malicious threads calling SHMemora service legitimately and any other compromised threads.

Physics-level [33] and system-level [34] attacks lead to destructive consequences and should be mitigated by general

mitigation (e.g., physical device management or system-level firewall), which are out of the scope. So we assume a trusted deployment where the kernel, the IMKVS runtime, and the underlying CXL interconnect and hardware operate correctly.

As the trust model of how the IMKVS is used is application-dependent, sanitizing the call sites and arguments to protected interfaces can be addressed by orthogonal techniques [35]. We focus on enforcing data-layer protection, assuming correct usage of protection interface by trusted code paths.

Microarchitectural side-channel attacks [36]–[38] or I/O-based attacks [39] are not considered. They utilize hardware microarchitectural vulnerabilities and are unavoidable on the software side. Building side-channel-free devices is an orthogonal hardware-side device-specific topic beyond our scope.

IV. SHMEMORA CHALLENGES & DESIGN OVERVIEW

In this section, we analyze key challenges & solution and present the architectural overview of SHMemora.

A. Challenges and Solution Sketches

To build a multi-host IMKVS over CXL-based shared memory in coherent DSM environments, this subsection outlines the key challenges and provides a sketch of the solutions.

Challenge 1: Preventing any bypass of the protection layer. Clients might attempt to bypass protection by directly accessing shared memory. To prevent this, we require a lightweight mechanism that disables direct access by default. In this case, it becomes challenging to enforce the exclusive use of protection APIs and ensure that all memory accesses remain within the bounds of the protection framework.

Solution Sketch 1: Memory Protection Keys (MPK). We leverage MPK [40] to isolate metadata and data in shared memory, denying default raw access. MPK, a hardware feature for efficient intra-process protection, assigns up to 16 keys to pages (via 4 spare PTE bits) and controls per-thread permissions through per-core *PKRU* register, updated with *WRPKRU*. Only trusted library runtime can temporarily raise permissions via *WRPKRU* to perform controlled operations, ensuring all accesses are mediated. Full solution is in Section V-A.

Challenge 2: Fine-grained memory protection. Adopting CXL, although the specification in Section II-C provides baseline access isolation in shared memory, it suffers from significant limitations that prevent its adoption. First, the protection model is host-centric, offering control only at the granularity of entire hosts. It lacks support for thread-level isolation, which is essential in multi-tenant environments where multiple untrusted components may coexist on a single host. Second, the memory grouping is based on fixed block sizes, which hinders fine-grained allocation and leads to internal fragmentation. This is particularly problematic for IMKVS and other data structures with variable-sized entries.

Solution Sketch 2: Capability-based protection. To achieve efficient isolation and access interface, we decide to exploit *capability* for fine-grained protection. *Capability* [12] by concept is the unforgeable tokens of authority granting

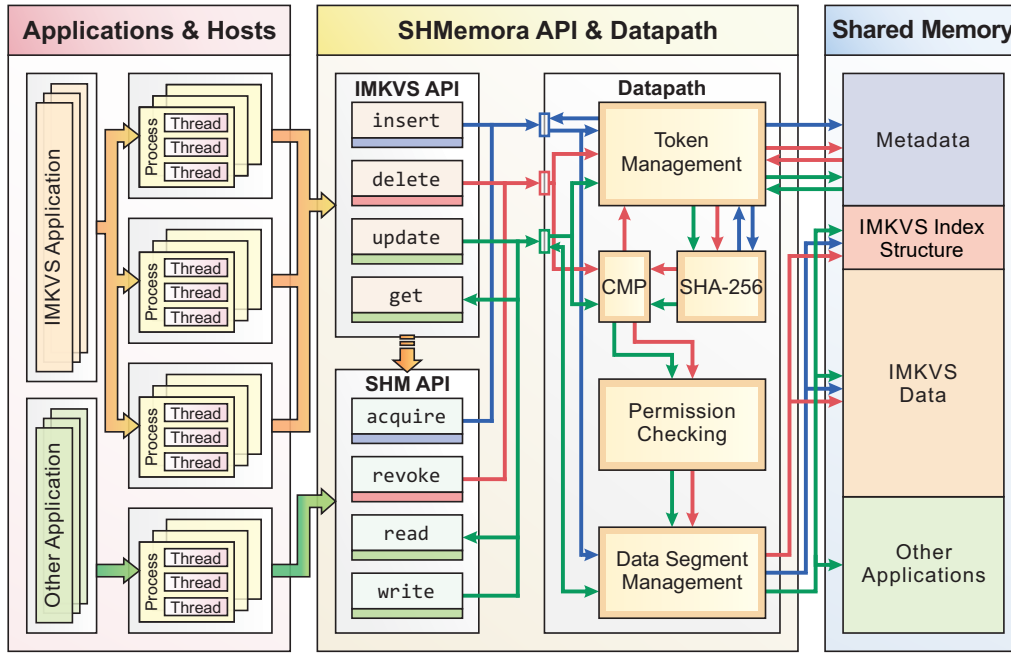


Fig. 2: SHMemora System overview. SHMemora IMKVS and other applications securely access CXL-based shared memory via token-embedded APIs, with token creation (blue), usage (green), and revocation (red).

rights to objects in the system. We provide a capability-based protection mechanism that associates each memory region with a capability token that encodes both the region boundaries and its access permissions. All accesses to shared memory are mediated through these tokens, ensuring that only authorized operations are permitted. Unforgeability of the tokens is guaranteed by a Hash-based Message Authentication Code (HMAC). The full solution is presented in Section V-B.

Challenge 3: Scalable data management across heterogeneous hosts. Managing shared memory across multiple hosts and operating systems is difficult. Heavy concurrency causes contention and bottlenecks, while token growth with increasing keys adds metadata overhead and frequent lookups. This overhead can degrade performance by up to $4.3\times$ due to excessive metadata management [41].

Solution Sketch 3: Partition and batching. To minimize contention in the IMKVS data path, we partition the IMKVS across hosts, enabling each host to operate primarily within its own memory region while still retaining the ability to access other partitions when necessary. We also reserve an elastic memory pool to support dynamic host joining and further partitioning when workloads grow or shift. For metadata management, we apply batching techniques to reduce the frequency of memory requests to the metadata segment, thereby alleviating contention under high-concurrency workloads. The full solution is presented in Sections V-C and VI-C.

B. Architecture Overview

Figure 2 shows SHMemora’s architecture. In a CXL-based shared memory environment, IMKVS applications across hosts access shared memory alongside other applications. SHMem-

ora provides protective APIs embedding capability tokens, which are integrated into *insert*, *get*, *update*, and *delete* operations and internally invoke Shared Memory (SHM) APIs for secure access. Other applications can also use SHM APIs to implement isolation. The blue, red, and green lines indicate token creation, revocation, and usage for read/write. Tokens are authenticated via secure hashing, and all accesses are mediated by permission checks.

V. PROTECTIVE SHARED MEMORY

This section presents our protection mechanism for shared memory, including the runtime library leveraging MPK to prevent unauthorized access, addressing Challenge 1. It also details the design of unforgeable capability tokens for Challenge 2, and partitioned layout solving Challenge 3.

A. Protective Library

To prevent unauthorized access and enforce capability-based protection, SHMemora requires all memory operations to use protective shared memory APIs. The shared memory region is configured as inaccessible by default with MPK. Within the APIs, PKRU permissions are temporarily set via the WRPKRU instruction to allow read/write, and revoked immediately after, ensuring no thread retains access outside controlled execution.

With this design, only 2 of the 16 MPK domains are required: one for default access denial, which prevents out-of-library access, and one for full access within the library. Further fine-grained access control is provided by SHMemora. A software-managed memory key maintained by a trusted kernel could also be used, but hardware MPK offers excellent performance and is widely supported across mainstream Intel,

TABLE I: Shared Memory APIs

API	Return	Description
<i>acquire(offset, size, permission)</i>	Token, Success	Acquire a token that grants access to a memory region. The parameter <i>offset</i> specifies the starting offset in device of the memory region, while <i>size</i> defines its length. The <i>permission</i> parameter indicates the allowed access type for the token (read, write, or both).
<i>revoke(token)</i>	Success	Invalidate a token from the permission table. Returns success if the token is successfully revoked.
<i>read_shm(offset, size, buffer, token)</i>	Success	Read data from a memory region into a buffer using a token. Returns success if the token grants read access to the specified memory region.
<i>write_shm(offset, size, buffer, token)</i>	Success	Write data from a buffer to a memory region using a token. Returns success if the token grants write access to the specified memory region.

AMD [42], and ARM processors (where it is called ARM Memory Domains [43]). Therefore, MPK does not serve as an isolation mechanism in our system. Instead, it functions as a switch, granting access only within the SHM APIs, including non-IMKVS applications shown in Figure 2.

Table I summarizes the core API functions. Before accessing any memory region, users must first obtain a capability token using the *acquire* method, which registers a memory region in the permission table and returns an unforgeable token encoding its boundaries and permissions. All subsequent read and write accesses must be accompanied by the corresponding token. Access is granted only if the token is valid and authenticated. When a token is no longer needed, it can be explicitly invalidated using the *revoke* operation.

B. Capability Tokens

As shown in step ① and ② of Figure 5, when a client requests access to a new memory region, the system generates an unforgeable token and registers it in the protection table along with the region’s metadata and permissions. This is handled by the *acquire* method to ensure unforgeability.

Unforgeability design. Unforgeability is ensured with a Hash-based Message Authentication Code (HMAC). Each token consists of a 64-bit Token ID and a 256-bit MAC, computed using SHA-256 over the Token ID and a secret key. The key is securely embedded in the trusted library and shared across hosts, never exposed to untrusted code. Thus, only the trusted library can generate valid MACs, making it computationally infeasible for untrusted threads to forge or tamper with tokens, preserving authenticity and integrity.

Memory access is controlled by unforgeable tokens, which encode the offset (start address), size, and permissions of the shared region holding KV pairs. All KV operations except *insert* require a valid token. On invocation, the system hashes the token to locate its protection table entry and verifies: (1) token validity, (2) access within bounds, and (3) required permissions. If any check fails, the operation is denied.

C. Shared Memory Layout

As illustrated in Figure 3, the shared memory is divided into four regions. The metadata segment contains a protection table, which is implemented as a hash table, where each entry describes a protected memory region (offset, size, validity, and permissions) along with the token. Entries are indexed by the token’s hashed value. CAS instructions are used to ensure consistency across multiple hosts. The IMKVS occupies two

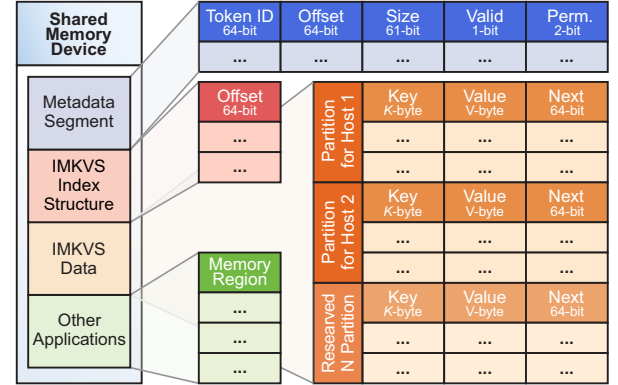


Fig. 3: Shared memory layout across hosts with 4 segments. The metadata segment implements a protection hash table. Each entry includes a Token ID, a validity tag, the offset and size of the memory region, and access permissions (read, write, both, or none). The IMKVS index structure is indexed by keys and stores offsets to corresponding values in the IMKVS data area. The data area is further partitioned by host, with an additional reserved region for elastic memory allocation.

regions: the index structure and data segment. Data segment is further partitioned among participating hosts, with each host being assigned a unique partition. Insert requests in IMKVS are distributed across partitions in a round-robin manner.

The partitioning mechanism is not a strict isolation boundary (guaranteed by capability tokens) but a way to reduce contention during concurrent inserts. Each host is assigned a partition so insertions spread across disjoint regions, avoiding metadata hotspots. After insertion, the partition no longer matters: all lookups, updates, and deletes follow the same semantics regardless of partition. Thus, “local” and “remote” accesses are identical, and partitioning serves only as a concurrency optimization for inserts. Our design therefore makes no distinction between local- and cross-host traffic.

Elastic region. To support dynamic workloads and system growth, SHMemora reserves a dedicated elastic region in IMKVS data segment, demonstrated as *Reserved N Partition* in Figure 3. This reserved space serves two primary purposes. First, when a host exhausts its initially assigned memory partition, it can obtain additional space from the elastic region and continue operating without disruption. This prevents insertion failures and mitigates the need for immediate repartitioning or rebalancing, ensuring seamless operation under uneven

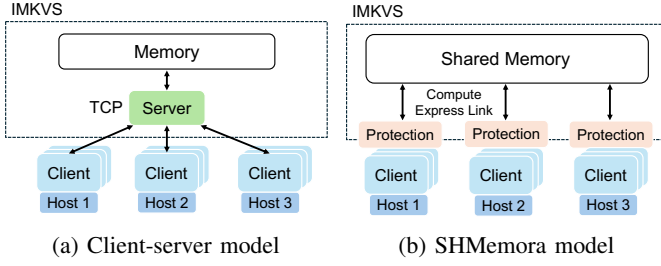


Fig. 4: IMKVS models across multiple hosts. We propose a shared memory model for IMKVS that is well-suited to the properties of CXL-based DSM.

or bursty workloads. Second, the elastic region facilitates dynamic host membership. New host joining the system can naturally obtain a partition from the elastic region based on its unique host ID, enabling fast on-boarding. If the elastic region is used up, subsequent *inserts* are rejected.

Memory reclaim. We introduce a lease-based reclamation mechanism to recover memory after host failures. Each IMKVS entry is assigned a lease, renewed by threads as needed. When it expires, a background task pushes the entry's offset to a free-list stack, which future inserts allocate from first. Since our implementation builds on the IMKVS demo from CXL-SHM [8] with added protection support, its partial failure recovery mechanism can be seamlessly integrated into SHMemora.

VI. SHMEMORA IMKVS

This section describes how SHMemora implements IMKVS and solve scalability addressing Challenge 3.

A. IMKVS Model and Library

We begin by analyzing the IMKVS model for the CXL DSM era. Current scale-out IMKVS adopts a client-server model, exchanging data over TCP (Figure 4a). While effective in clusters, this design is inefficient for multi-host memory sharing: it relies on explicit data movement with high network and copy overhead, and it cannot exploit emerging CXL global shared memory, which provides hardware-coherent access with near-local DDR latency.

To fully utilize CXL, SHMemora introduces a shared memory model (Figure 4b) for IMKVS that replaces the server with a symmetric peer-to-peer design built on hardware-coherent CXL memory. All hosts directly access metadata and KV data in a globally shared region with fine-grained protection, eliminating explicit communication, data copying, and TCP reliance. SHMemora also adds an extra protection layer.

SHMemora supports multiple index structures, including hash tables and B+ trees. Both store offsets relative to the shared memory start address. Offsets are used instead of pointers since pointers are host-specific virtual addresses that differ across machines, making them non-portable.

In hash tables, collisions are resolved with linked lists. Each entry stores a key, value, and next-entry offset. Insertions or lookups hash the key to an index, then traverse the list to find

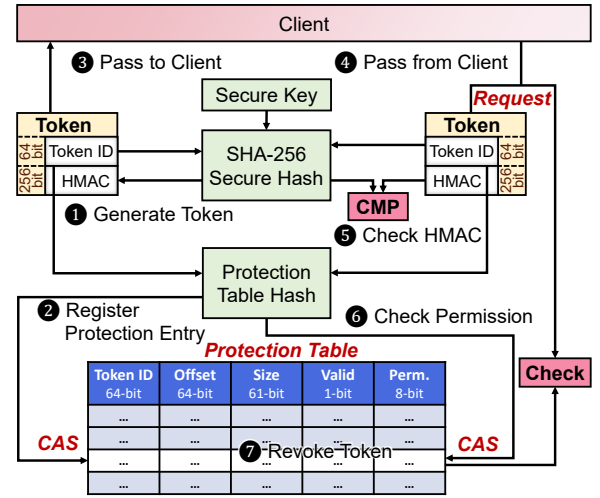


Fig. 5: The protection token and its lifecycle. The token is generated, registered, and passed to client with *insert*; passed from client, and verified by checking MAC and permission with *get* and *update*; revoked with *delete*.

a match. In B+ trees, keys are stored in a balanced hierarchy, enabling both point lookups and efficient range queries.

The protected IMKVS is built on top of the protected shared memory and, therefore, follows to utilize the underlying memory protection APIs. Compared to traditional KV APIs, the protected KV APIs incorporate tokens to enforce protection. Table II summarizes these protected KV APIs. The *insert* method inserts a KV pair into the shared store and invokes the *acquire* method to obtain a token for the KV pair. The user inserting data is considered the owner. The owner is responsible for the token it obtains and may share it with any other user it trusts. Other subsequent KV operations use *read_shm* or *write_shm* with the token to access the corresponding data in shared memory.

B. Token Workflow

Figure 5 shows the token lifecycle in SHMemora. Tokens are created during *insert* when a new block is allocated: ① A Token ID from a partitioned counter is combined with a secret key to compute the MAC. ② The token is registered in the protection table via CAS, using hashed Token ID as index; quadratic probing handles collisions. ③ The token is returned to the client, which may share it with trusted peers. ④ Clients attach the token to requests. ⑤ The system recomputes MAC and verifies with token. ⑥ If valid, metadata and access rights are checked, and request proceeds. ⑦ Tokens are revoked (valid bit cleared) when all associated KV pairs are deleted.

C. Managing Protection Metadata at Scale

For workloads with millions of keys, assigning each key its own protection token is impractical. The metadata segment, which tracks bounds and permissions per token, would grow linearly with the number of keys, causing excessive memory

TABLE II: IMKVS APIs

API	Return	Description
<i>insert(key, value)</i>	Token, Success	Insert a KV pair into the shared store and return a token for future access.
<i>update(key, value, token)</i>	Success	Update a KV pair with a token. Check permission with the token.
<i>get(key, value, token)</i>	Success	Retrieve a KV pair with a token. Check permission with the token.
<i>delete(key, value, token)</i>	Success	Delete a KV pair with a token. Check permission with the token.

Algorithm 1 Batched Token Allocation

```

1: procedure INSERT( $k, v$ )
2:   if #Inserted_keys < THRESHOLD then
3:      $region \leftarrow \text{ALLOCATE}(k, v)$ 
4:      $\tau \leftarrow \text{ACQUIRE}(region, r\&w)$ 
5:     DOINSERT( $k, v, region, \tau$ )
6:   else
7:     if  $region == \text{NULL}$  or  $\text{FULL}$  then
8:        $region \leftarrow \text{ALLOCATEBLOCK}$ 
9:        $\tau \leftarrow \text{ACQUIRE}(region, r\&w)$ 
10:    end if
11:    DOBATCHINSERT( $k, v, region, \tau$ )
12:    // Allocate memory for  $k, v$  within the region
13:    // via buddy allocator
14:  end if
15:  return  $\tau$ 
16: end procedure

```

use and slower lookups. To address this, SHMemora introduces a *batched protection strategy*, grouping multiple KV pairs with identical access semantics into one memory region governed by a shared token, as shown in Algorithm 1.

To balance metadata efficiency and access granularity, SHMemora uses a *threshold-based policy* to determine whether to apply batched or per-key tokenization. This threshold considers the size of the key group. When the number of keys inserted exceeds the threshold, the system switches to using a batched token. This threshold is determined by the number of keys that a batch can accommodate at the saturation point. Based on Figure 9, this value is 80.

For batched tokens, SHMemora employs a *preallocation and amortized model*. When a thread inserts its first KV pair or when all previously allocated blocks are full, the system allocates a new memory block and generates a corresponding token. Within the block, we use a buddy allocator to manage memory therein, minimizing fragmentation. Subsequent inserts by the same thread, as long as they share the same access semantics, reuse the existing token and append new KV pairs to the associated memory block until it is full.

This strategy addresses bursts in the memory footprint of token metadata and reduces the frequency of token acquisitions during inserts. For reads and writes, they are performed through existing tokens. Whether the accessed keys belong to the same or different batches only determines which token is used and does not affect performance.

VII. FORMAL SECURITY PROOF

In this section, we provide formal security proofs on the confidentiality and integrity of data and metadata in SHMemora theoretically. In practice, we will further conduct penetration tests to validate the robustness of the system against various adversarial behaviors in Section VIII-B.

A. Definitions and Preliminaries

We start from formalizing basic components, i.e., functions, threads, and function calls by the following definitions.

DEF. 1. A *function* F is a state machine $F = (\mathcal{S}, \mathcal{P}, \mathcal{T})$, where $\mathcal{S}$ is the set of states, $\mathcal{P}$ is the parameter space, $\mathcal{T} : \mathcal{S} \times \mathcal{P} \rightarrow \mathcal{S} \times \mathcal{P}$ is the transition function of states and parameters. The state and parameter at time t of F are denoted by $F.s|_t$ and $F.p|_t$. A *thread* is a distinguished instance of a function.

DEF. 2. A *combination* of functions F' and F'' with a *function call* $\mathcal{C} = (s'_b, s'_e, \tilde{p}', s''_b, s''_e, \tilde{p}'')$ is a function $F = F' \cup_{\mathcal{C}} F''$, where $F.S = F'.S \cup F''.S$, $F.P = F'.P \times F''.P$.

The function call specifies the entry (i.e., the beginning state s_b) and the exit (i.e., the ending state s_e) as well as the passed parameters $\tilde{p}$ for each sub-functions.

With a slight abuse of notation, $F.T$ is given as:

$$F.T(s, p', p'') = \begin{cases} (s''_b, p', p'' |_{\tilde{p}' \rightarrow \tilde{p}''}) & \text{if } s = s'_e, \\ (s'_b, p' |_{\tilde{p}'' \rightarrow \tilde{p}'}, p'') & \text{if } s = s''_e, \\ (F'.T(s, p'), p'') & \text{if } s \in F'.S - \{s'_e\}, \\ (F''.T(s, p''), p')^{\#} & \text{if } s \in F''.S - \{s''_e\}, \end{cases}$$

where $p' |_{\tilde{p}'' \rightarrow \tilde{p}'}$ denotes replacing sub-field $\tilde{p}'$ of the parameter p' with $\tilde{p}''$, and $(s, p'', p')^{\#}$ stands for a switch of the 2nd and the 3rd parameter, i.e., (s, p', p'') . We denote $F_A \preceq F_B$ if $\exists F'$, s.t. $F_B = F_A \cup F'$.

The combined transition function actually specifies how the exit of each function jumps to the entry of the other function, and how the specified sub-fields of parameters are passed. In states other than the exit, the transition is carried out according to the original transition functions of the sub-functions, and the parameters within the other function remain unaffected.

By permuting the items of $\mathcal{C}$ to $\mathcal{C}^* = (s''_b, s''_e, \tilde{p}'', s'_b, s'_e, \tilde{p}')$, we have $F' \cup_{\mathcal{C}} F'' = F'' \cup_{\mathcal{C}^*} F'$, which indicates that combination by function calls defines a *symmetric* operation.

DEF. 3. An *MPK function* $\mathcal{M}$ is a function satisfying MPK hardware constraints, where $\mathcal{M}.S$ consists of 2 sub-functions: (1) $\mathcal{S}_{perm}$ modifies per-thread access permission parameter $\mathcal{M}.p_{perm}(addr, T)$; (2) $\mathcal{S}_{acc}$ performs memory access at $addr$ when the permission $\mathcal{M}.p_{perm}(addr, T) = 1$.

T stands for the thread of combinations of functions containing $\mathfrak{M}$.

The involvement of the MPK function can break the isolation of parameters among combinations if the address of some MPK accesses $addr$ coincides with some parameter p . We focus on the integrity and confidentiality of the parameters.

DEF. 4. A parameter field p is called *integral* in a thread T if p is free from all MPK access of MPK functions $\mathfrak{M} \preceq T$. The parameter p is called *confidential* to $F \preceq T$ if $p \notin F.P$.

Then we formalize the SHMemora components, and the system under attacker threads with malicious functions.

DEF. 5. An *SHMemora function* $\mathfrak{S}$ is a function satisfying SHMemora design constraints, where $\mathfrak{S}.S$ consists of 2 basic sub-functions:

- (1) $\mathcal{S}_{acq}$ acquires new-generated *Token* at offset $addr$;
- (2) $\mathcal{S}_{acc}$ performs memory access at $addr$ with *Token*.

DEF. 6. An *attacker thread* is defined as $AT = (Mal \cup \mathfrak{M}_1) \cup (\mathfrak{S} \cup \mathfrak{M}_2)$, where Mal is an arbitrary malicious function, $\mathfrak{M}_1$ and $\mathfrak{M}_2$ are MPK functions², $\mathfrak{S}$ is an SHMemora function. The combination of function call satisfies MPK and SHMemora constraints.

B. Assumptions

From the threat model and design constraints of MPK and SHMemora, we have the following assumptions.

ASS. 1. MPK is secure.

- (1) $\mathfrak{M}.p_{perm}$ is hardware-implemented. If $s \notin \mathfrak{M}.S$, $AT.T$ will remain $\mathfrak{M}.p_{perm}$ unchanged;
- (2) Thread isolation on $\mathfrak{M}.p_{perm}$. For 2 threads $T_1 \neq T_2$, T_1 cannot modify contents of $\mathfrak{M}.p_{perm}(addr, T_2)$ via $\mathfrak{M} \preceq T_1$.

ASS. 2. Cryptography is secure. Specifically, HMAC is secure against *key recovery attacks* and *hash collision attacks*.

ASS. 3. Proper initialization. At $t = 0$, all initializations of the attacker thread AT are done properly.

- (1) $AT.s|_{t=0} \in Mal.S$;
- (2) $\mathfrak{M}.p_{perm}(addr, AT)|_{t=0} = 0$ for all $addr$.

ASS. 4. Malicious behavior detection.

- (1) Only trusted library, i.e., $\mathfrak{S}$, can modify $\mathfrak{M}.p_{perm}$. In formal terms, $\mathfrak{M}.p_{perm}$ changes at t , then $AT.s|_t \in \mathfrak{S} \cup \mathfrak{M}_2.S$;
- (2) For any *Token* already registered by other threads, $\mathfrak{S}$ never registers a redundant one or pass the hashed *Token* to Mal .

ASS. 5. SHMemora Design constraints.

- (1) In the attacker thread $AT = (Mal \cup \mathfrak{M}_1) \cup (\mathfrak{S} \cup \mathfrak{M}_2)$, each complete function call of $\mathfrak{S} \cup \mathfrak{M}_2$ takes 1 modification on $\mathfrak{M}.p_{perm}$ before and after performing MPK access, resulting in a total of 2 modifications;
- (2) SHMemora performs address checking before performing MPK access. If $s \in \mathfrak{S}.S_{acc}$, then $addr$ does not point to any metadata, i.e., $*addr \notin \mathfrak{S}.P$;
- (3) SHMemora performs token checking before performing MPK access. If $s \in \mathfrak{S}.S_{acc}$, then the token *Token* registered for $addr$ has been passed in $\mathfrak{S}.P$;

²Since the MPK permission parameter $\mathfrak{M}.p_{perm}$ is global, the parameter space is actually a quotient space identifying the fields $\mathfrak{M}_1.p_{perm}$ and $\mathfrak{M}_2.p_{perm}$ as the same field, though they belong to different MPK function calls.

- (4) The parameter passed in function call from $\mathfrak{S}$ to Mal includes only $Token = \{Tid, HMAC\}$ and status codes, where $HMAC = SHA(Tid, key)$.

C. Formal Security Proof

We will show that the confidentiality and integrity of metadata and data are well protected by giving proof of 1 lemma and 4 propositions.

LEM. 1. Mal cannot acquire MPK access permission directly. In formal terms, if $\mathfrak{M}.p_{perm}(addr, AT)|_t = 1$ for some $t > 0$, it satisfies that $AT.s|_t \in \mathfrak{S} \cup \mathfrak{M}_2.S$.

PROOF. By Ass. 3, we have $\mathfrak{M}.p_{perm}(addr, AT)|_{t=0} = 0$. If the set $W := \{t > 0 | \mathfrak{M}.p_{perm}(addr, AT)|_t = 1 \wedge AT.s|_t \in Mal \cup \mathfrak{M}_1.S\}$ is not empty, W must have a minimal element, denoted as $t_0 > 0$. Then we have a dichotomy, either $\mathfrak{M}.p_{perm}(addr, AT)$ jumps from 0 to 1 at t_0 , or $AT.s$ jumps from $\mathfrak{S} \cup \mathfrak{M}_2.S$ to $Mal \cup \mathfrak{M}_1.S$ at t_0 .

If the former holds, by Ass. 1, it satisfies that $AT.s|_{t_0} \in \mathfrak{M}_1.S$, which contradicts Ass. 4.

Otherwise, if the latter holds, by Ass. 1, there must be an odd number of permission modifications in $\mathfrak{M}_2.S_{perm}$ from 0 to t_0 . By Ass. 5, the number of permission modifications made through complete calls to $\mathfrak{S} \cup \mathfrak{M}_2$ is even. Therefore, there must be an incomplete permission modification from 0 to t_0 . Since $AT.s|_{t=0} \notin \mathfrak{S} \cup \mathfrak{M}_2$, it follows that $AT.s|_{t_0} \in \mathfrak{S} \cup \mathfrak{M}_2$, which contradicts the fact that $t_0 \in W$.

In conclusion, $W = \emptyset$, which proves the lemma.

PROP. 1. All metadata of $\mathfrak{S}$ is integral.

PROOF. It suffices to show that for all $addr$, if $\mathfrak{M}.p_{perm}(addr, AT)|_t = 1$, we have $*addr|_t \notin \mathfrak{S}.P$. By Lem. 1, we have that $AT.s|_t \in \mathfrak{S} \cup \mathfrak{M}_2.S$. So AT is doing a SHMemora function call at time t . When it tries to perform MPK access, i.e., $AT.s|_t \in \mathfrak{S}.S_{acc}$, by Ass. 5., $*addr \notin \mathfrak{S}.P$. Thus all metadata of $\mathfrak{S}$ is integral.

PROP. 2. HMAC Key $key \in \mathfrak{S}.P$ is confidential to Mal .

PROOF. The HMAC Key is a parameter of the function $\mathfrak{S}$. All data that Mal can obtain from $\mathfrak{S}$ falls into two categories: (1) arbitrary data access through the MPK function $\mathfrak{M}_1$, and (2) data passed via function call parameters.

For (1), by Lem. 1, Mal cannot access the *key* and related metadata through $\mathfrak{M}_1$.

For (2), by Ass. 5, the data related to the *key* that Mal can obtain from $\mathfrak{S}$ through function call parameters consists only of a set of *Tokens*, namely a set of $\{Tid_k\}$ and a set of $\{HMAC_k = SHA(Tid_k, key)\}$. By Ass. 2, the hash function is secure against key recovery, so Mal cannot recover *key*.

PROP. 3. Any valid token $Token_0 = (Tid_0, HMAC_0)$ registered by another *victim thread* VT is confidential to Mal .

PROOF. By Lem. 1, Mal cannot obtain $Token_0$ directly through MPK function $\mathfrak{M}_1$. By Ass. 4, $\mathfrak{S}$ refuses to generate a token already registered, so Mal cannot obtain $Token_0$ through $\mathfrak{S} \cup \mathfrak{M}_2$. By Prop. 3, $key \notin Mal.P$. Therefore, by Ass. 2, even if $Tid_0 \in Mal.P$, Mal cannot generate a HMAC collision with $HMAC_0 = (Tid_0, key)$. That means Mal cannot obtain or recover $HMAC_0$ from all possible function calls, i.e., $Token_0$ is confidential to Mal .

TABLE III: Experimental configuration.

	Host 1	Host 2	Host 3
CPU	AMD EPYC Turin	AMD EPYC Turin	Intel Xeon GNR
Sockets	1	1	2
Cores per socket	64	64	96
OS	Ubuntu 24.04	Ubuntu 24.04	Ubuntu 22.04
CXL Switch	XCONN XC50256		
CXL memory device	Longsys ASIC 192G		

PROP. 4. Any address $addr_0$ registered with $Token_0$ by another thread, $*addr_0$ is confidential to AT .

PROOF. On one hand, by Prop. 3, $Token_0 \notin Mal.P$. So $Token_0$ can never be passed from Mal to $\mathcal{S}$. By Ass. 5, $\mathcal{S} \cup \mathcal{M}_2$ never performs MPK access to $addr_0$. On the other hand, by Lem. 1, $Mal \cup \mathcal{M}_1$ cannot perform MPK access to $addr_0$. So $AT = (Mal \cup \mathcal{M}_1) \cup (\mathcal{S} \cup \mathcal{M}_2)$ never performs MPK access to $addr_0$. That means AT cannot acquire any information about $*addr_0$.

To this end, we have demonstrated that all metadata is integral (by Prop. 1), critical metadata is confidential (by Props. 2 and 3), and Token-protected data is complete (by Prop. 4). Practical penetration tests will be further carried out for validation in Section VIII-B.

VIII. EVALUATION

A. Experimental Setup

Hardware Platform. We conduct our evaluation on a real platform equipped with CXL shared memory. The platform uses CXL 2.0-compliant ASIC devices that support memory sharing across up to three hosts. To enforce cache coherence in the absence of hardware-managed coherence, we explicitly insert CLFLUSH instructions to evict stale cache lines, ensuring that clients retrieve the most up-to-date data from the shared memory. We also set up virtual machines to validate that SHMemora functions correctly without CLFLUSH instructions, since in this environment hardware cache coherence is guaranteed by the physical machine.

The evaluation platform has three CXL-capable servers: one Intel Granite Rapids (GNR) and two AMD EPYC Turin, with configurations shown in Table III. In evaluations, Turin servers were used for 1- and 2-host setups, while the 3-host setup included two Turin servers and one GNR. For more nodes, we add NUMA nodes on each server in a round-robin manner with the same order.

B. Security Penetration Test

We construct and execute three classes of attacks that reflect realistic adversarial behaviors under our threat model.

- (1) Token forgery attack against Prop. 2-4 in Section VII: a malicious thread attempts to fabricate a token by constructing arbitrary 256-bit values in an effort to bypass

authorization checks. These forged tokens are supplied to shared memory API calls, aiming to gain unauthorized access to protected memory regions.

- (2) Metadata tampering attack against Prop. 1 in Section VII: here, the adversary attempts to directly overwrite or manipulate the protection table that stores metadata associated with issued tokens such as memory region boundaries and permission flags.
- (3) Out-of-bounds access attack against Prop. 4 in Section VII: in this case, the adversary begins with a valid token issued through the legitimate *acquire* procedure but attempts to perform memory operations beyond the region bounds encoded in the token's associated metadata.

The penetration test followed prior work [31]. Across 1M token-forgery attempts, all forged tokens fail HMAC checks and no unauthorized reads/writes occurred. Similarly, 1M metadata write attempts of varying offsets and sizes trigger segmentation faults, and 1M out-of-bounds attempts (overflows, underflows, misaligned ranges) are uniformly denied.

C. Performance Evaluation

In this section, we compare SHMemora with SHMemora-Unprotected, a version of SHMemora with all security-related features removed, implemented by removing all token-related operations. No *acquire* is required for *insert*, and no permission checks are performed for any other operations.

1) Microbenchmarks

The microbenchmark evaluation focuses on core operations of the IMKVS APIs across varying value sizes (16B to 64KB) and a configuration of 1 to 12 hosts, each with 32 clients. We test the four operations separately 100,000 requests. For each operation, we measure both latency (per-operation completion time) and bandwidth (operations per second) to assess overall performance. To demonstrate the advantages of the shared memory model, we also compare with Redis, a widely used IMKVS with client-server architecture and TCP-based communication. We configure it under the same conditions as our setup, (1KB payload, 100,000 requests, 32 clients, and pipelining disabled). MGET/MSET operations are not used. All other parameters are set as default. In addition, we evaluate the impact of different batching strategies (batch sizes of 1KB to 128KB), assess the performance of bulk insert operation, and quantify the performance overhead introduced by the MAC-based token authentication as well as memory reclamation. Unless otherwise specified, all data sizes used in the experiments are 1KB.

Latency. As shown in Figure 6, SHMemora achieves an average latency of 67 μs . As the number of hosts increases from 1 to 12, the latency of each operation shows a slight upward trend across 4 different operations. This increase is primarily attributed to the growth in communication and coordination overhead introduced by the added hosts. Nevertheless, the additional overhead compared to the SHMemora-Unprotected remains consistent. The overall average overhead is 1.68%, indicating that the system scales efficiently.

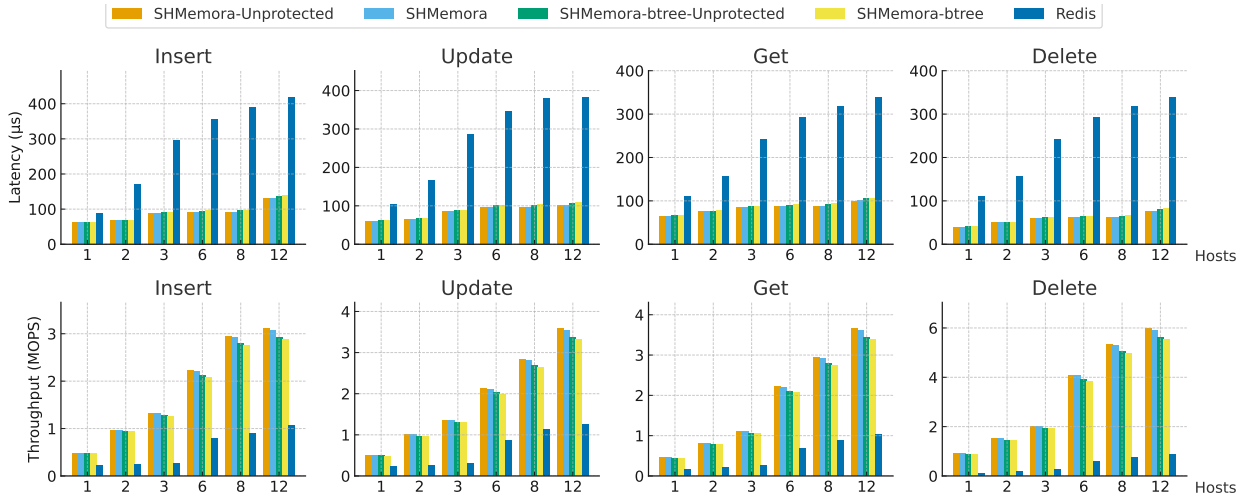


Fig. 6: Latency and throughput comparison for different operations across varying host numbers.

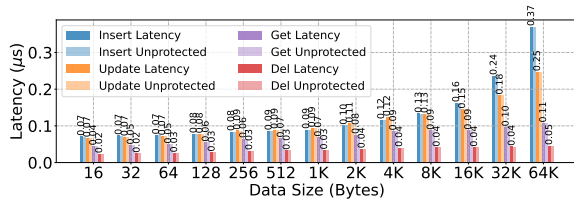


Fig. 7: Latency of different operations across value sizes.

Compared with Redis, SHMemora demonstrates superior latency performance across all operations and host scales. For the 4 operations, Redis exhibits significantly higher latency as the system scales, reaching to $368.20 \mu s$ averagely with 12 hosts, whereas SHMemora maintains $102.84 \mu s$ averagely. This significant performance gap is primarily due to Redis' client-server model, which introduces network communication and data moving overhead.

Throughput. As shown in Figure 6, throughput for all operations increases as the number of hosts grows, and remains high across all configurations. The overhead compared to SHMemora-Unprotected stays low, overall average overhead is 1.68%. With 12 hosts, the system achieves an average throughput of 4.05 million operations per second (MOPS), with delete operations nearing 6 MOPS.

In comparison to Redis, SHMemora delivers significantly higher throughput than Redis across all operations. Specifically, the throughput of SHMemora achieves an average of 2.55 MOPS, while Redis achieves only an average of 0.45 MOPS, representing a $5.67\times$ improvement.

Data size. Figure 7 shows latency either increases or remains stable as data size grows, primarily due to increased data movement and memory access overhead. Compared to SHMemora-Unprotected, our system incurs a slight additional latency at larger sizes, mainly due to shared memory coordi-

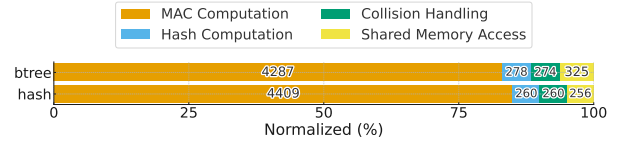


Fig. 8: Overhead breakdown. Note whatever index structure we use, the metadata is managed with hash table.

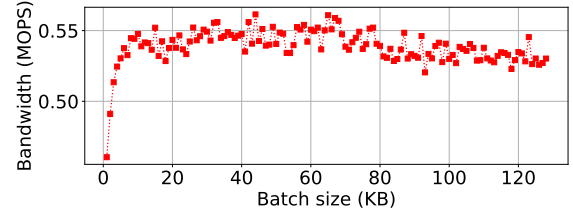


Fig. 9: Throughput of insert operation across batch sizes.

nation, but the overhead remains low, averaging just 0.85%.

Batch size. We explore the use of different batch sizes to evaluate the throughput of the *insert* operation. We evaluate batch sizes ranging from 1KB to 128KB with the buddy allocator. As shown in Figure 9, increasing the batch size generally improves throughput in the initial increments, because larger batches reduce metadata contention and the frequency of token usage. The growth reaches its peak at 80KB. Beyond this point, throughput starts to decrease slightly due to increased contention within a batch.

Bulk insert. We also implement a bulk insert API. During a bulk insert operation, SHMemora preemptively groups all key-value entries with the same access semantics into batches, where the batch size is dynamically determined by the bulk insert size for optimal performance. Specifically, when the total data size is below 80 KB, we set the batch size equal

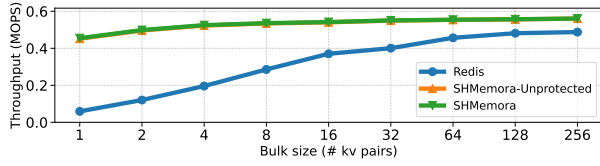


Fig. 10: Throughput of bulk insert compared with Redis enabling pipelining.

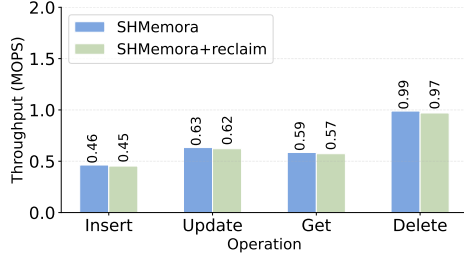


Fig. 11: Throughput with and without memory reclaim mechanism in B+ tree.

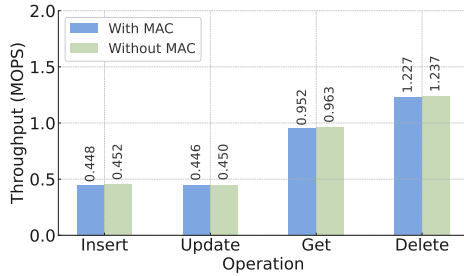


Fig. 12: Throughput of operations with and without MAC in hash table.

to the total data size. If the total data size exceeds 80 KB, the batch size is capped at 80 KB. Figure 10 shows that the throughput of bulk insert increases slightly with bulk size. Although Redis benefits significantly from bulk insert, SHMemora still outperforms it by $1.15\times$ at convergence.

Memory reclaim and MAC. We evaluate the overhead of the reclamation mechanism and MAC across four operations by comparing with a non-reclamation and a non-MAC version. As shown in Figure 11 and Figure 12, the reclamation and MAC-enabled design reduces throughput by just 2.02% and 0.85% across four operations. However, the reclamation is not triggered under normal condition.

2) Macrobenchmarks

We use Yahoo! Cloud Serving Benchmark (YCSB) [15] to characterize the performance of SHMemora. We generate workload traces for YCSB-A, B, C, and D, which represent different read/write access patterns.³ For each workload, we run the benchmark and compare the performance of SHMemora and SHMemora-Unprotected.

³YCSB-A: 50% reads and 50% updates (read-modify-heavy), YCSB-B: 95% reads and 5% updates (read-heavy), YCSB-C: 100% reads (pure read), YCSB-D: 95% reads and 5% inserts (read-mostly with inserts).

Latency. The left subplots of Figure 13 compare the latency of SHMemora and SHMemora-Unprotected across YCSB-A–D with 1–12 hosts. Workload D (insert-heavy) shows the highest latency at $189\mu\text{s}$ from allocation and metadata updates, while get-heavy workloads are much faster, with SHMemora averaging $82.05\mu\text{s}$ under 12 hosts—close to the unprotected version. Overall, SHMemora adds only 1.75% average overhead, excelling on get-dominant patterns due to its optimized path and lightweight checks. Latency scales stably to 12 hosts, with less than 2.52% increase across workloads.

Throughput. In throughput (right subplots of Figure 13), SHMemora maintains high performance across workloads and host counts, especially for get- and update-heavy patterns (YCSB-A to C). For instance, YCSB-A grows from 0.86 MOPS with 1 host to over 3.30 MOPS with 12 hosts, with YCSB-B and C showing similar scalability. By contrast, YCSB-D (insert-dominant) remains much lower due to metadata and memory allocation overhead. Across all cases, SHMemora is slightly below SHMemora-Unprotected, with up to 2.25% degradation in YCSB-A to C.

In summary, we observe that SHMemora incurs an average latency and throughput overhead of 1.75% and 2.25% under YCSB workloads compared to the SHMemora-Unprotected version, which is higher than the overhead of our microbenchmark tests. This discrepancy is primarily due to the differences in workload trace and scale.

To verify that SHMemora is not limited to a peer-to-peer model and can also benefit from centralized coordination, we evaluate a coordinator responsible for maintaining load balance across nodes. The coordinator tracks the load of each node and manages global workloads, dispatching tasks to the node with the lowest load. We tested this setup on the YCSB-D workload, and the average overhead is 0.75%.

In our microbenchmark, each basic operation is tested in isolation with a relatively small data footprint and minimal concurrency, allowing SHMemora to fully exploit its optimized memory path and lightweight memory protection mechanism. In contrast, YCSB workloads reflect more realistic usage patterns, involving large volumes of data and interleaved execution of multiple operations, often across multiple threads and hosts. This not only introduces resource contention, but also increases the complexity of memory management, slightly amplifying the overall overhead. The contention and lack of key locality is the reason of YCSB’s inserts exhibit higher latency.

IX. RELATED WORK

CXL-based shared memory. The feasibility of using CXL to enable shared memory across multiple hosts has been established in recent studies [8]–[11]. CXL-SHM [8] and SWARM [10] investigate robustness issues, including crash consistency and fast replication, in CXL-based shared memory systems. Collectively, these efforts confirm the potential of CXL as a foundation for disaggregated memory systems supporting multiple cooperating hosts.

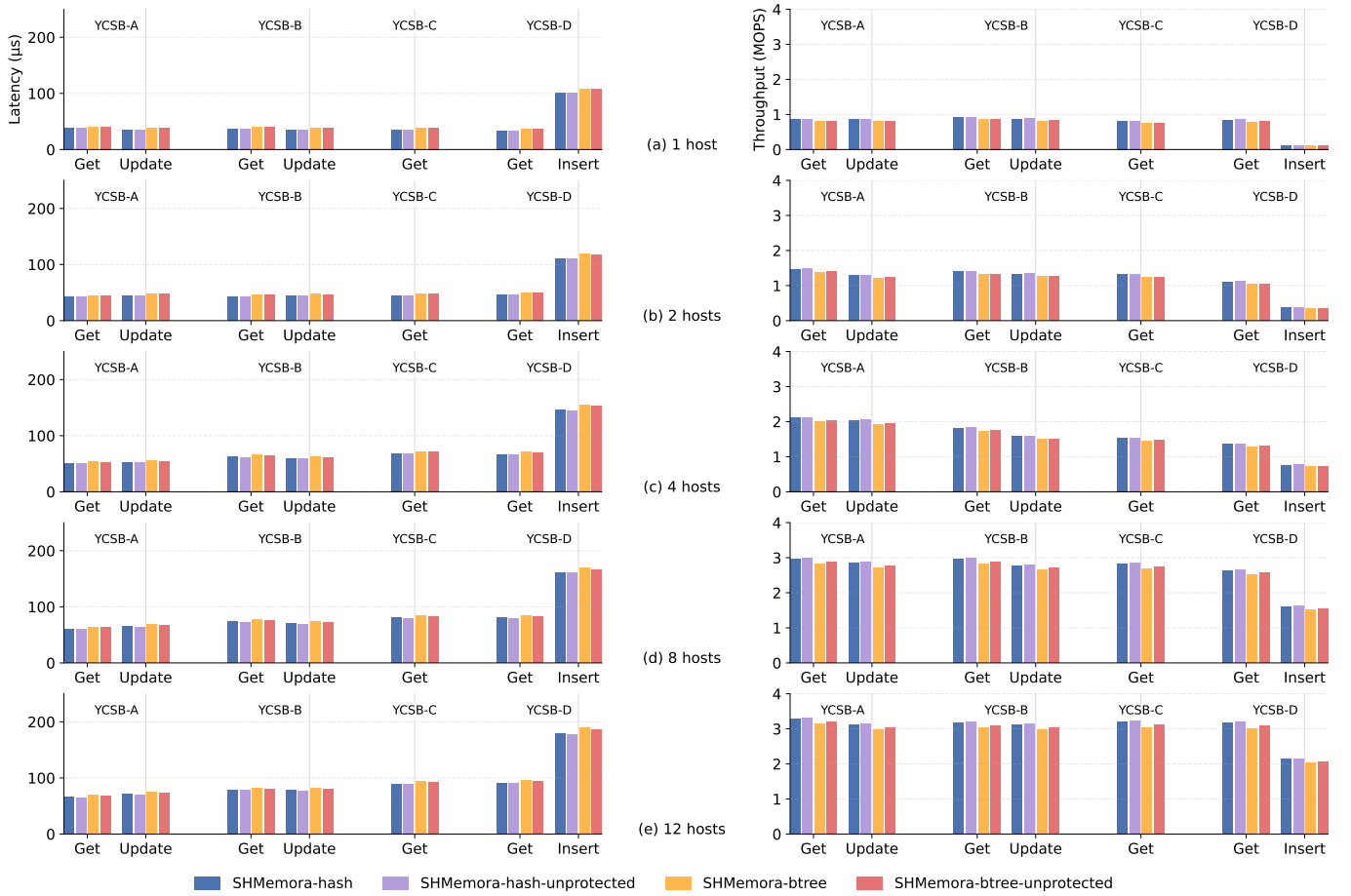


Fig. 13: Latency comparison for YCSB workloads across varying host numbers. Each host has 32 clients.

Other hardware-based shared memory technology. RDMA is widely used for remote memory, relying on specialized verbs but incurring relatively high latency. Patronus [32] leverages the advanced Memory Window verb for protection and proposes optimizations like operation pairing and combining. In contrast, CXL memory offers load/store access with near-DRAM latency but no equivalent advanced verbs. Thus, memory protection in CXL systems must operate directly on memory semantics and ensure no lingering access.

Shared-memory-based IMKVS. Lightning [5] introduces a protection mechanism for shared IMKVS using Intel MPK, enabling isolation between threads. However, it is confined within a single process and it cannot enforce isolation across hosts. CXL-SHM [8] mentions a shared-everything IMKVS but does not incorporate any form of memory isolation. Our work expands these research by enabling secure, shared IMKVS across hosts with fine-grained protection.

Hardware capability. Some existing works leverage hardware capabilities for enforcing memory isolation, exemplified by the prominent CHERI (Capability Hardware Enhanced RISC Instructions) architecture [12]. CHERI extends conventional ISAs with first class capability values that encapsulate pointers along with associated metadata. Commercial hard-

ware support for CHERI remains limited today. Since CXL-based DSM is also in an early prototyping stage, integrating CHERI and CXL in hardware is currently infeasible.

X. CONCLUSION

This paper presents SHMemora, a framework to build IMKVS on CXL-based shared memory. SHMemora introduces a lightweight, capability-based protection mechanism that enforces fine-grained permissions over arbitrarily sized memory regions. We implement a full prototype of SHMemora on real CXL shared memory hardware. The formal security proofs and penetration tests show that SHMemora provides correct protection on metadata and data. Performance evaluation shows that SHMemora only incurs 1.68% overhead on average.

ACKNOWLEDGEMENT

This work was generously supported by the National Key Research and Development Program of China under Grant (2024YFB4405400), NSFC (No. U24A6009, 62572265), Beijing Municipal Science and Technology Project (Nos. Z241100004224028), Beijing Natural Science Foundation (L247013), Alibaba Research Intern Program and BNRist. Furthermore, we gratefully thank the anonymous reviewers for their helpful comments.

This work used ChatGPT to polish grammar and wording based on our original writing. ChatGPT also assisted in developing the Python code used to plot the figures. All original writing, ideas, methodologies, implementation, and the overall framework of this manuscript were conceived and created entirely by the authors without AI assistance.

REFERENCES

- [1] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel, "Finding a needle in haystack: Facebook's photo storage," in *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.
- [2] A. Blankstein, S. Sen, and M. J. Freedman, "Hyperbolic caching: Flexible caching for web applications," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, Jul. 2017, pp. 499–511. [Online]. Available: <https://www.usenix.org/conference/atc17/technical-sessions/presentation/blankstein>
- [3] Redis, "Redis cluster specification," https://redis.io/docs/latest/operate/oss_and_stack/reference/cluster-spec/, 2025.
- [4] Z. Hu, P. Zuo, Y. Chen, C. Wang, J. Hu, and M.-C. Yang, "Aceso: Achieving efficient fault tolerance in memory-disaggregated key-value stores," in *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, ser. SOSP '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 127–143. [Online]. Available: <https://doi.org/10.1145/3694715.3695951>
- [5] D. Zhuo, K. Zhang, Z. Li, S. Zhuang, S. Wang, A. Chen, and I. Stoica, "Rearchitecting in-memory object stores for low latency," *Proc. VLDB Endow.*, vol. 15, no. 3, p. 555–568, Nov. 2021. [Online]. Available: <https://doi.org/10.14778/3494124.3494138>
- [6] "Compute express link™: The breakthrough cpu-to-device interconnect," <https://www.computeexpresslink.org/home>, 2022.
- [7] T. Ma, Z. Liu, C. Wei, J. Huang, Y. Zhuo, H. Li, N. Zhang, Y. Guan, D. Niu, M. Zhang, and T. Ma, "HydraRPC: RPC in the CXL era," in *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. Santa Clara, CA: USENIX Association, Jul. 2024, pp. 387–395. [Online]. Available: <https://www.usenix.org/conference/atc24/presentation/ma>
- [8] M. Zhang, T. Ma, J. Hua, Z. Liu, K. Chen, N. Ding, F. Du, J. Jiang, T. Ma, and Y. Wu, "Partial failure resilient memory management system for (cxl-based) distributed shared memory," in *Proceedings of the 29th Symposium on Operating Systems Principles*, ser. SOSP '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 658–674. [Online]. Available: <https://doi.org/10.1145/3600006.3613135>
- [9] J. Huang, M. Zhang, T. Ma, Z. Liu, S. Lin, K. Chen, J. Jiang, X. Liao, Y. Shan, N. Zhang, M. Lu, T. Ma, H. Gong, and Y. Wu, "Trenv: Transparently share serverless execution environments across different functions and nodes," in *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, ser. SOSP '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 421–437. [Online]. Available: <https://doi.org/10.1145/3694715.3695967>
- [10] A. Murat, C. Burgelin, A. Xygkis, I. Zablatchi, M. K. Aguilera, and R. Guerraoui, "Swarm: Replicating shared disaggregated-memory data in no time," in *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, ser. SOSP '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 24–45. [Online]. Available: <https://doi.org/10.1145/3694715.3695945>
- [11] W. Tang, Y. Han, T. Ai, G. Li, B. Yu, and X. Yang, "Yggdrasil: Reducing network i/o tax with (cxl-based) distributed shared memory," in *Proceedings of the 53rd International Conference on Parallel Processing*, ser. ICPP '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 597–606. [Online]. Available: <https://doi.org/10.1145/3673038.3673138>
- [12] R. N. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera, "Cheri: A hybrid capability-system architecture for scalable software compartmentalization," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 20–37.
- [13] Redis, "Redis," <https://redis.io/>, 2025.
- [14] Memcached, "Memcached," <https://memcached.org>, 2025.
- [15] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 143–154. [Online]. Available: <https://doi.org/10.1145/1807128.1807152>
- [16] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, "Treadmarks: shared memory computing on networks of workstations," *Computer*, vol. 29, no. 2, pp. 18–28, 1996.
- [17] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "FaRM: Fast remote memory," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, Apr. 2014, pp. 401–414. [Online]. Available: <https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevi%C4%87>
- [18] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin, "Latency-Tolerant software distributed shared memory," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. Santa Clara, CA: USENIX Association, Jul. 2015, pp. 291–305. [Online]. Available: <https://www.usenix.org/conference/atc15/technical-session/presentation/nelson>
- [19] Y. Lu, J. Shu, Y. Chen, and T. Li, "Octopus: an RDMA-enabled distributed persistent memory file system," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, Jul. 2017, pp. 773–785. [Online]. Available: <https://www.usenix.org/conference/atc17/technical-sessions/presentation/lu>
- [20] Q. Cai, W. Guo, H. Zhang, D. Agrawal, G. Chen, B. C. Ooi, K.-L. Tan, Y. M. Teo, and S. Wang, "Efficient distributed memory management with rdma and caching," *Proc. VLDB Endow.*, vol. 11, no. 11, p. 1604–1617, Jul. 2018. [Online]. Available: <https://doi.org/10.14778/3236187.3236209>
- [21] C. Buragohain, K. M. Risvik, P. Brett, M. Castro, W. Cho, J. Cowhig, N. Gloy, K. Kalyanaraman, R. Khanna, J. Pao, M. Renzelmann, A. Shamis, T. Tan, and S. Zheng, "A1: A distributed in-memory graph database," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 329–344. [Online]. Available: <https://doi.org/10.1145/3318464.3386135>
- [22] Q. Wang, Y. Lu, E. Xu, J. Li, Y. Chen, and J. Shu, "Concordia: Distributed shared memory with In-Network cache coherence," in *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, Feb. 2021, pp. 277–292. [Online]. Available: <https://www.usenix.org/conference/fast21/presentation/wang>
- [23] L. Logan, A. Kougkas, and X.-H. Sun, "Megammap: Blurring the boundary between memory and storage for data-intensive workloads," ser. SC '24. IEEE Press, 2024. [Online]. Available: <https://doi.org/10.1109/SC41406.2024.00114>
- [24] Y. Zhou, H. M. G. Wassel, S. Liu, J. Gao, J. Mickens, M. Yu, C. Kennelly, P. Turner, D. E. Culler, H. M. Levy, and A. Vahdat, "Carbink: Fault-Tolerant far memory," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 55–71. [Online]. Available: <https://www.usenix.org/conference/osdi22/presentation/zhou-yang>
- [25] Z. Guo, Z. He, and Y. Zhang, "Mira: A program-behavior-guided far memory system," in *Proceedings of the 29th Symposium on Operating Systems Principles*, ser. SOSP '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 692–708. [Online]. Available: <https://doi.org/10.1145/3600006.3613157>
- [26] M. Zhang, Y. Hua, and Z. Yang, "Motor: Enabling Multi-Versioning for distributed transactions on disaggregated memory," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. Santa Clara, CA: USENIX Association, Jul. 2024, pp. 801–819. [Online]. Available: <https://www.usenix.org/conference/osdi24/presentation/zhang-ming>
- [27] XConn Technologies, "World's First CXL 2.0 and PCIe Gen5 Switch IC," <https://www.xconn-tech.com/products>, 2023.
- [28] X. Yang, Y. Zhang, H. Chen, F. Li, G. Fan, Y. Kong, B. Wang, J. Fang, Y. Wang, T. Huang *et al.*, "Unlocking the potential of cxl for disaggregated memory in cloud-native databases," 2025.
- [29] C. S. v3.2, <https://computeexpresslink.org/cxl-specification/>.
- [30] S. W. Stark, A. T. Markettos, and S. W. Moore, "How flexible is cxl's memory protection?" *Commun. ACM*, vol. 66, no. 12, p. 46–51, Nov. 2023. [Online]. Available: <https://doi.org/10.1145/3617580>
- [31] Z. Yuan, S. Hong, R. Chang, Y. Zhou, W. Shen, and K. Ren, "Vdom: Fast and unlimited virtual domains on multiple architectures," in *Proceedings of the 28th ACM International Conference on*

Architectural Support for Programming Languages and Operating Systems, Volume 2, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 905–919. [Online]. Available: <https://doi.org/10.1145/3575693.3575735>

- [32] B. Yan, Y. Lu, Q. Wang, M. Xie, and J. Shu, “Patronus: High-Performance and protective remote memory,” in *21st USENIX Conference on File and Storage Technologies (FAST 23)*. Santa Clara, CA: USENIX Association, Feb. 2023, pp. 315–330. [Online]. Available: <https://www.usenix.org/conference/fast23/presentation/yan>
- [33] S. Skorobogatov, “Physical attacks and tamper resistance,” in *Introduction to Hardware Security and Trust*. Springer, 2011, pp. 143–173.
- [34] J. Xiao, H. Huang, and H. Wang, “Kernel data attack is a realistic security threat,” in *Security and Privacy in Communication Networks: 11th EAI International Conference, SecureComm 2015, Dallas, TX, USA, October 26-29, 2015, Proceedings 11*. Springer, 2015, pp. 135–154.
- [35] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg, “ERIM: Secure, efficient in-process isolation with protection keys (MPK),” in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1221–1238. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity19/presentation/vahldiek-oberwagner>
- [36] F. Boenisch, R. Munz, M. Tiepelt, S. Hanisch, C. Kuhn, and P. Francis, “Side-channel attacks on query-based data anonymization,” in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 1254–1265.
- [37] Z. Wang, M. Taram, D. Moghimi, S. Swanson, D. Tullsen, and J. Zhao, “{NVLeak}:{Off-Chip}{Side-Channel} attacks via {Non-Volatile} memory systems,” in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 6771–6788.
- [38] M. Malik and T. Patel, “Database security-attacks and control methods,” *International Journal of Information*, vol. 6, no. 1/2, pp. 175–183, 2016.
- [39] B. Morgan, E. Alata, V. Nicomette, and M. Kaâniche, “Bypassing iommu protection against i/o attacks,” in *2016 Seventh Latin-American Symposium on Dependable Computing (LADC)*. IEEE, 2016, pp. 145–150.
- [40] I. Corporation, *Intel® 64 and IA-32 Architectures Software Developer’s Manual*. Intel Corporation, 2024.
- [41] K. Wang and F. Chen, “Catalyst: Optimizing cache management for large in-memory key-value systems,” *Proc. VLDB Endow.*, vol. 16, no. 13, p. 4339–4352, Sep. 2023. [Online]. Available: <https://doi.org/10.14778/3625054.3625068>
- [42] S. Park, S. Lee, and T. Kim, “Memory protection keys: Facts, key extension perspectives, and discussions,” *IEEE Security Privacy*, vol. 21, no. 3, pp. 8–15, 2023.
- [43] ARM, <https://developer.arm.com/documentation/ddi0406/b/System-Level-Architecture/Virtual-Memory-System-Architecture/-VM-SA-/Memory-access-control/Domains>.