ARTIFACT EVALUATED
USENIX
AVAILABLE

Memclave: Secure In-Memory Enclave for Untrusted Hosts

Amit Choudhari*
CISPA Helmholtz Center for Information Security

Fabian van Rissenbeck*
TU Dortmund

Christian Rossow
CISPA Helmholtz Center for Information Security

Abstract

Cloud platforms run data-intensive workloads in multi-tenant settings, where frequent CPU–memory traffic can leak access patterns via cache side channels. Processing-in-Memory (PIM) devices such as UPMEM move computation into DRAM, sharply reducing data movement and shrinking the CPU cache footprint. However, commercial PIM architectures expose a host-programmed control plane and host-shared module memory, leaving device-resident code and data vulnerable to a compromised host. Existing secure-PIM proposals either add encryption/access-control hardware or rely on heavyweight host-side cryptographic protocols, complicating practical deployment.

We present Memclave, a *software-only* framework that brings code integrity and data confidentiality to commodity PIM without hardware changes. A TPM-attested hypervisor permanently isolates the PIM's control plane from host access at boot. On each in-memory core, a trusted loader authenticates the user kernel and establishes a per-session protected data path. Memclave preserves the programming model and kernel code: host applications replace a small set of data-movement calls with secure drop-ins, keeping the trusted computing base small and porting effort low. We implement Memclave on off-the-shelf UPMEM DIMMs and evaluate it across the PrIM benchmark suite, covering heterogeneous memory-access, compute, and synchronization patterns. After a one-time ∼100 ms authenticated load, in-memory kernel time remains close to the PIM baseline: Multilayer Perceptron (MLP) stays within 1.5× at practical sizes, and Breadth–First Search (BFS) is 1.1× on some graphs with modest rise as number of frontier levels increase.

1 Introduction

Modern cloud environments increasingly support demanding workloads that require fast, secure, energy-efficient processing of large data volumes [66, 68, 70], such as machine learning inference and large-scale data analytics. Traditional CPU-centric architectures struggle to find a balance in performance and security. Frequent data transfers between CPUs and memory creates significant performance bottlenecks, reducing throughput and energy efficiency [62]. Moreover, in multi-tenant environments, shared CPU caches expose sensitive memory access patterns, leaving systems vulnerable to cache-based side-channel attacks [20, 31, 36, 52]. Existing defenses such as Oblivious RAM (ORAM) eliminate these side-channels but incur prohibitive performance overhead, making them impractical for large-scale deployment [32, 52, 65]. This raises a need for solutions that simultaneously address performance bottlenecks and cache-based security vulnerabilities.

Processing-in-Memory (PIM) architectures offer a promising solution to memory intensive workloads by embedding lightweight, low-power processors directly within DRAM modules [24, 30, 38, 42, 62, 75, 79]. This DRAM Processing Unit (DPU) architecture reduces data transfers, enhancing energy efficiency and throughput. Additionally, because PIM executes computations within DRAM and does not interact with shared CPU caches, PIM inherently mitigates cache-based side-channel risks.

However, current commercial PIM implementations still lack critical security primitives. In particular, the host OS retains full access to the Control Interface (CI) registers. These registers enable the host to load, configure, and dispatch PIM compute programs, called kernels, and access shared DRAM regions. This unrestricted CI access allows a malicious host to reprogram/instrument PIM functions at will, inject unauthorized or tampered kernels, reroute data flows, and read or corrupt in-compute state of PIM. When combined with full read/write rights over shared memory, these flaws allow adversaries to break all isolation guarantees, compromising computational integrity and data confidentiality [60]. Commercial PIM solutions thus do not fulfill the vital security guarantees that security-aware cloud tenants typically require.

Research work such as SE-PIM [27] and PIM-Enclave [26] attempts to address these vulnerabilities by proposing security aware frameworks, but suffers from limitations that

[1]These authors contributed equally to this work.

prevent easy deployment on existing commercial PIM hardware. SE-PIM relies on specialized hardware enhancements within memory modules and assumes trusted CPU enclaves, which require tightly integrated hardware packages, complicating practical adoption. Similarly, PIM-Enclave proposes integrating dedicated AES-enabled DMA engines and specific memory access-control logic directly into the DRAM, necessitating significant hardware modifications, unavailable on any currently available PIM modules. Importantly, neither SE-PIM nor PIM-Enclave have a real hardware implementation, lacking the practicality of the solution.

Recent software-only work by Ghinani et al. [29] and SecNDP [80] keep PIM untrusted and push cryptography to the CPU: linear kernels are handled via arithmetic secret sharing, while non-linear steps leverage Yao's garbled circuits. This design achieves confidentiality and verifiable correctness only for linear operations, leaving non-linear correctness on untrusted PIM outside the verification envelope. Their systems also introduce additional CPU-PIM communication and share-switching costs (including measurable switching overheads) and rely on precomputation to avoid CPU bottlenecks. In addition, these design approaches do not harden the control plane (e.g., on-PIM CFI or OS-level CI mediation), relying on cryptographic protection around an untrusted PIM module.

Existing VM-based TEEs such as Intel TDX, AMD SEV-SNP primarily protect a VM's conventional memory (DRAM) from a malicious host and hypervisor [4, 17]. However, their protection boundaries do not inherently extend to cover peripheral devices or accelerator control planes [58]. Confidential VM I/O rely on shared buffers, or an additional platform support. Recent standards in the TEE-IO such as TDISP, SEV-TIO and TDX Connect aim to securely extend the TEE boundary over the PCIe/CXL bus [9, 39, 67]. Despite these advancements, current commodity PIM devices are DDR-attached memory modules and do not implement PCIe/CXL device protocols. Hence, a confidential VM cannot use these standards to protect the PIM control plane.

In this paper, we introduce Memclave, a practical, software-only security framework to enable secure computation on current off-the-shelf commercial PIMs. Our approach follows the "trust but verify" principle, where tenants can perform remote attestation of the trusted hypervisor and trusted loader. During system boot, Memclave launches a trusted hypervisor to permanently remove direct OS access to CI registers and introduces a runtime PIM lockdown mechanism, enabling the DPU to securely execute computations in a confidential computing mode. Additionally, we provide system kernels for lightweight encryption and key-exchange directly within the DPU, ensuring secure data transfers with on-DPU cryptographic capabilities. Finally, we implement a runtime *loader* directly within the DPUs to verify the integrity of these kernel, preventing execution of malicious instructions.

Our proposal to use an attestable hypervisor to mediate all privileged commands is a well-explored method. This design can be adopted by cloud providers who already anchor tenant isolation in hypervisors (e.g., GCE's KVM, Azure's Hyper-V, Windows' VBS, AWS' Nitro, Cyberus' Xen). Thin, open hypervisors like Xen (1MB TCB), pKVM or jailhouse (9kLoC) [43, 55, 73] can be leveraged to further minimize the Trusted Computing Base (TCB) even if the provider's software plane is adversarial, potentially even using formally-verified microkernel-based designs (e.g., seL4 [45]) that guarantee TCB security. Our core contributions lie in sandboxing and key confinement, necessary even if device-level CI protection is introduced.

We implement Memclave[1] on commodity UPMEM with a TPM-attested *hypervisor* and a tiny in-PIM *loader*, requiring no hardware changes and only minor changes to host-side data-transfer code. On one rank (64 DPUs $\times$ 16 tasklets/DPU), security costs are dominated by a one-time $\sim$100 ms authenticated kernel load; steady-state kernel time remains close to the PIM baseline. Across end-to-end workloads, Memclave isolates the control plane with low overhead and for protected transfers, overhead scales with transferred bytes. In our testbed, Multilayer Perceptron (MLP) stays within 1.5$\times$ of the PIM baseline at realistic sizes, and Breadth–First Search (BFS) is 1.1$\times$ on some graphs, with modest increase as number of frontier levels grows.

Our key contributions are as follows:

- We derive generic design goals and requirements for software-only in-memory enclaves on commodity PIM, and assess prior approaches against them, highlighting unmet guarantees and deployability gaps.
- We design and implement Memclave, a software-only framework that adds confidentiality and integrity to commercial PIMs with no hardware changes, anchored in a TPM-backed chain of trust and remote attestation.
- We present a minimal on-DPU trusted loader that authenticates and loads segmented user code, enforces control-flow integrity to block code-reuse attacks, and establishes a secure channel for code/data transfers.
- We evaluate and demonstrate the effectiveness of Memclave on commercially available PIM hardware (UPMEM), validating security enhancements and practical deployability with modest performance overhead.

2 Background

2.1 Processing in-Memory (PIM)

PIM architectures integrate computational logic within or adjacent to memory devices, thereby reducing the costly data movement between memory and CPU that causes a

[1] Available here: https://github.com/fabianvanrissenbeck/memclave

performance bottleneck in conventional von Neumann systems [8, 15, 59, 61, 71, 72]. In this work, we focus on processing-near-memory (PnM), which integrates discrete processing cores alongside the memory as separate chips on the same DIMM module [15, 18, 42]. This design provides much higher internal bandwidth and lower latency than off-chip busses but typically forgoes cache coherence. Instead, it uses explicit DMA engines and memory-mapped control registers for host–PIM communication and exposes programmer-managed scratchpads rather than hardware caches. Consequently, PIM accelerates [33] dense and sparse linear algebra, database scans and aggregations, data analytics [37], graph traversals [8, 63], neural-network inference [18, 47], bioinformatics [25], and image-processing workloads.

Figure 1 illustrates the PIM architecture. A *kernel* is a program executed on the PIM device. Each compute unit, referred to as DPU, provides Local Instruction Memory (LIM) for kernel code and Local Scratchpad Memory (LSM) for temporary data; External Memory (EM) denotes the large DRAM region on the module used for bulk data and for exchanging buffers with the host. The Control Interface (CI) refers to the memory-mapped registers through which software boots kernels, triggers transfers, and manages device state.

2.2 UPMEM PIM Architecture

UPMEM integrates fully programmable DRAM DPUs on commodity DDR4 DIMMs; each module combines many DPUs with DRAM banks to deliver high parallelism and bandwidth while retaining DDR4 compatibility. Each DPU is a lightweight 32-bit RISC with IRAM (Instruction RAM), WRAM (Working RAM), and MRAM (Main RAM) [76], corresponding to LIM, LSM, and EM in Figure 1. DMA engines move data among these memories [33, 38]. Host–device communication uses a memory-mapped Control Interface (CI) exposing command registers for DMA, kernel boot, and synchronization. There is no direct inter-DPU interconnect; coordination occurs via the host. In a typical run, the host loads the compiled DPU kernel into IRAM via the CI, stages inputs to MRAM, triggers execution, and retrieves results back from MRAM on termination. DPU kernels are small self-contained programs that consist of text and data sections representing the initial state of IRAM and WRAM. DPUs have 24 independent hardware threads and follow a barrel style architecture. Threads are controlled via DPU instructions `boot`, `resume` and `stop` or via CI commands from the host. Thread state is tracked in a DPU's internal memory called RUN, which can be modified via thread control instructions and `clr_run`. While the memories are shared between threads, each thread has 24 private registers. DPUs primarily operate on WRAM, data is moved between WRAM and MRAM via the DMA instructions `ldma` and `sdma`. IRAM cannot be read, but written to via the `ldmai` DMA instruction.

2.3 Security Guarantees of PIM

Commercial PIM architectures, despite their performance benefits, lack basic security guarantees required by cloud tenants (confidentiality and integrity of code/data) and omit built-in security primitives [60, 61]. Unlike host CPUs, PIM cores do not provide on-die cryptographic engines, hardware isolation, secure key storage, or remote attestation, leaving them exposed to malicious CI commands from a compromised OS. For example, a compromised OS can dump a victim DPU's LSM data or load tampered code into its LIM via the CI and DMA. This risk is acute in cloud settings where the operator controls both hypervisor and OS.

2.4 Hardware Root-of-Trust

A secure computing environment relies on a trusted initial component, a *hardware root-of-trust*, typically a discrete Trusted Platform Module (TPM) [2, 5] or vendor security extension (Intel SGX, AMD SEV, ARM TrustZone) [1, 4, 10, 21]. These roots provide secure key storage, measured boot, and remote attestation, enabling a chain of trust.

A critical building block is secure boot, where each startup component is cryptographically measured, and each measurement extends a hash into platform configuration registers (PCRs) maintained by the root-of-trust. PCRs form a tamper-evident record of the boot state, enabling detection of unauthorized modifications [2, 69]. Secure key storage provided by these roots underpins sealed storage and key exchange/attestation. Sealed storage binds keys to specific PCR states, ensuring access only under verified conditions [2, 6].

Remote attestation builds upon this foundation by enabling a remote verifier (a client) to securely validate the integrity and identity of a target platform. The verifier issues a challenge, and the root-of-trust returns a signed quote over selected PCRs that proves the measured software stack [3, 14].

3 Threat Model

We target cloud settings where a tenant submits code/data to DPUs while the cloud provider and the *guest* software stack is untrusted. In such a multi-tenant cloud setting, Memclave provides trusted in-memory computation by leveraging CPU-side roots (TPM) to create an in-memory enclave inside the PIM. This enclave enables tenants to perform authenticated loading of code and sensitive data into PIM. We use *host* for the CPU machine that physically hosts the PIM module and *guest* for all untrusted software above a trusted hypervisor (guest OS, drivers, management software and any nested hypervisors). We distinguish (i) CI registers (host-visible PIM control plane), (ii) EM (host-visible PIM DRAM region), and (iii) LIM and LSM (per-DPU local memories) used during execution (as shown in Section 2.1 and Figure 1).

Trusted Computing Base (TCB). The TCB consists of the TPM-backed measured-boot chain, a TPM-measured hypervisor that exclusively owns the CI and mediates all privileged device commands, and the on-DPU loader placed in the execute-only LIM (mechanisms in Section 5). A tenant initiates a session and provisions secrets only after verifying a fresh TPM quote for the expected measurements.

Adversaries. We assume the cloud provider and the entire *guest* software stack above the trusted hypervisor are fully adversarial. The tenants are mutually untrusted, as deployed kernels may be buggy or malicious and may attempt to violate confinement, escalate into privileged loader routines, or persist across sessions to affect other tenants. We assume the DPU hardware itself is benign (no hardware trojans). Since commodity PIM modules lack a device root of trust, our attestation does not prove silicon genuineness; counterfeit PIM devices or malicious hardware are out of scope.

Residual State and Sessions. We treat any pre-session contents of a DPU's execution context in LIM and residual state in EM/LSM (e.g., stale kernels or leftover buffers) as adversarial. To support secure deployment of the on-DPU loader at boot, we require that the hypervisor can bring DPUs to a clean starting state (e.g., by re-intializing).

Attacker Capabilities. With no CI lockdown mechanism on commodity PIM, an untrusted guest software stack can access PIM control registers and issue commands to install/replace DPU code and steer execution. After the loader grants execution to a tenant kernel on PIM, we assume the kernel may attempt (i) unauthorized entry into loader code paths, (ii) corruption of loader key/control state, or (iii) TOCTOU attack using multithreading to bypass loader checks and inject malicious code; Memclave aims to prevent such escalation and cross-tenant leakage. We exclude DoS and side-channels. We exclude invasive physical attacks (e.g., probing/injecting commands on the DRAM bus) and similar board-level fault attacks. Computation inside PIM reduces exposure to shared CPU cache side channels for in-PIM portions of the workload.

4 Goals and Requirements

The goal of Memclave is to ensure that the DPU executes only authenticated code and that all sensitive data remains confidential, even if the guest OS is entirely untrusted. In addition, Memclave must integrate seamlessly into existing DPU platforms without requiring any hardware modifications or intrusive software changes. We therefore derive the following security requirements (SR) and auxiliary requirements (AR):
SR1. Code Integrity. Only authenticated code may execute in LIM; reject unsigned or tampered binary at load time.
SR2. Data Confidentiality. Sensitive code and data remain confidential at rest (in EM) and in transit (host–PIM); plaintext exists only inside LSM/LIM during execution.
SR3. Control-Flow Integrity. Prevent arbitrary code execution on the DPU such that attackers cannot corrupt or launch

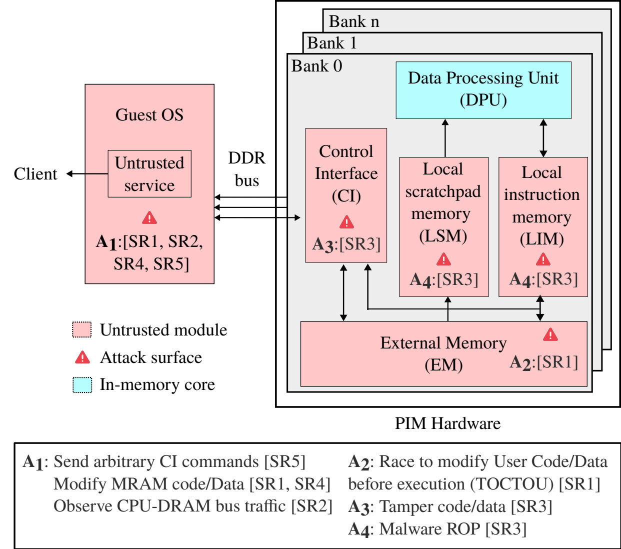

Figure 1: PIM overview and threat surface. Threat labels A_i denote adversary actions; $[SRj]$ show security requirements.

unauthorized code or jump outside allowed regions.
SR4. Replay and Rollback Protection. Detect and reject replay of staged code, data, or session state across sessions.
SR5. Isolation from Guest and OS. Untrusted guest software (guest OS and other guest components) must not be able to read or write CI registers, trigger DPU execution, or derive plaintext from guest–PIM traffic; all interactions with the PIM hardware must occur only through an authenticated interface.
AR1. Multi-DPU Scalability. Memclave must support partitioning and loading of trusted sections across multiple DPUs in parallel, enabling large-scale in-memory computation.
AR2. Modular, Staged Loading. Client programs larger than LIM can be split and streamed into LIM in stages, and the system must handle multiple sequential sessions without requiring a full DPU reset.
AR3. Deployability. Integration into existing DPU platforms and cloud stacks must require no hardware modifications and only minimal software changes, ensuring rapid adoption.
AR4. Lightweight TCB and Audibility. The on-device TCB should remain minimal so it can be easily audited and updated, and should incur modest performance and memory overhead.
AR5. Multi-Client Support. Multiple clients can share PIM hardware concurrently without cross-tenant leakage or interference; ensuring isolated data and control-plane operations.

4.1 Design Tradeoffs and Shortcomings

Meeting these requirements (SR, AR) demands understanding approaches and limits. Section 4.1 summarizes requirement coverage. We briefly survey prominent alternatives before introducing our hypervisor-assisted, software-centric design.
Hardware-Modified PIM Architectures. SE-PIM [27] and

Scheme	Security Requirements (SR)					Auxiliary Requirements (AR)				
	SR1 Code Integrity	**SR2** Data Confidentiality	**SR3** Control–Flow Integrity	**SR4** Anti Replay	**SR5** Guest/OS Isolation	**AR1** Multi-DPU Scaling	**AR2** Staged Loading	**AR3** Deployability (no HW mods)	**AR4** Light TCB & Audit.	**AR5** Multi-Client
UPMEM (baseline)	✗	✗	✗	✗	✗	✓	△	✓	–	✗
SE-PIM/PIM-Enclave Ⓢ	✓	✓	✓	✓	✓	△	△	✗	△	△
MPC-on-PIM	✗	✓	✗	△	✗	△	✗	✓	✗	△
Memclave	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 1: Indicates the SR and AR per scheme. ✓=met, △=partial/assumption-heavy, ✗=not met and (Ⓢ)=simulation only.

PIM-Enclave [26] add dedicated secure execution units, per-bank access controls, and encryption engines to PIM hardware. These solutions meet SR1–SR5. However, extensive hardware changes raise costs and limit compatibility with commercial platforms (~~AR3~~), scaling across many DPUs (~~AR1~~) or supporting staged loading (~~AR2~~); often requiring a redesign.

CPU-Resident Trusted Execution Environments. Host TEEs (e.g., SGX [21], TrustZone [1,10]) protect CPU-resident code/data but do not satisfy ~~SR1~~/~~SR3~~/~~SR4~~ for on-PIM execution and cannot enforce ~~SR2~~ for data in EM. They also cannot directly protect CI registers or DMA (~~SR5~~). Routing between CPU enclaves and DPUs adds complexity and enlarges the TCB (~~AR4~~), and architectural constraints complicate multi-DPU scaling (~~AR1~~) and staged loading (~~AR2~~).

Encryption-Based PIM Approaches. PUF-based designs [49], ChaoPIM [51], and P^3M [50] secure data with specialized cryptographic hardware. They provide SR2 but depend on non-commodity features (~~AR3~~) and do not protect against unauthorized DMA/CI accesses (~~SR1~~, ~~SR3~~, ~~SR5~~) or support staged loading (~~AR2~~) and multi-DPU scaling (~~AR1~~).

MPC-Based PIM Approaches. Cryptography-driven, device-untrusted approaches [29, 80] keep PIM hardware untrusted. In MPC-on-PIM [29], linear steps use arithmetic secret sharing (with linear checks) and non-linear steps use Yao's garbled circuits; in SecNDP [80], computation proceeds on encrypted data with integrity verification limited to linear operations at the device. These attain confidentiality (SR2), but provide no on-PIM code authentication or fine-grained control-flow guarantees (~~SR1~~, ~~SR3~~) and leave the device control plane unhardened (~~SR5~~); replay/rollback is protocol-dependent (SR4: partial). They require no PIM hardware changes (AR3) but enlarge the host-side TCB (~~AR4~~), and re-expressing programs as MPC routines weakens AR2 and throttles multi-DPU scaling due to CPU–PIM interaction (AR1). Device-side checks cover only *linear* operations, non-linear integrity is not enforced on PIM hardware.

Hypervisor-based Isolation. Using a hypervisor (or a small security monitor) to remove the guest's direct access to the device's control plane, mediate privileged commands and DMA configuration is commonly used in production virtualization stacks and protected-hypervisor designs [7, 12, 35, 56, 57]. This enforces SR5 by mediating the control plane from an untrusted guest OS and supports SR1 (partial) if the mediator authenticates guest-to-device code loads. However, hypervisor-only mediation is insufficient for commodity PIM, where the DPU directly reads/writes EM during execution, and the hypervisor is not on the data path of in-PIM memory accesses (~~SR2~~). Furthermore, it cannot robustly enforce CFI against malicious kernels executing on the DPU (~~SR3~~) nor protect against replay attacks across sessions (~~SR4~~) while preserving multi-tenant reuse without requiring full device reset (~~AR5~~).

Hypervisor-Assisted In-Memory Enclave (Our Approach). We leverage a TPM-attested hypervisor to lock CI and DMA paths at boot. Unlike hardware- or encryption-centric designs, this requires no hardware changes, enabling deployment on commercial DPUs (AR3). Our framework also introduces a minimal attested loader in LIM (see Section 5.1); the hypervisor mediates DMA/CI access (SR5), the loader enforces code authentication (SR1) and control-flow restrictions (SR3), and together they provide channel and at-rest confidentiality for staged code/data (SR2) with replay protection (SR4). The design supports staged loading (AR2) and multi-DPU scaling (AR1) while keeping a minimal, auditable TCB (AR4). The loader enforces strict separation of tenants, ensuring Multi-Client support (AR5).

5 Memclave's Design

5.1 Architecture

Memclave's core concept is isolating PIM control plane and on-device execution blocks from untrusted software. To achieve this, we introduce two trusted entities verified during secure boot: the *hypervisor* and the *loader*. The *hypervisor* primarily functions as a gatekeeper, preventing unauthorized communication with the PIM, whereas the *loader* provides a secure runtime environment for executing authenticated code blocks and establishing a protected communication channel with the client. Together, these measures block code injection, message interception, and control/data tampering during execution.

Figure 2 illustrates the end-to-end architecture of Memclave. PIM ranks are active devices that directly process data and are controlled via CI registers. However, these registers lack protection and can be easily manipulated by an untrusted OS. To prevent unauthorized access, the *hypervisor* exposes

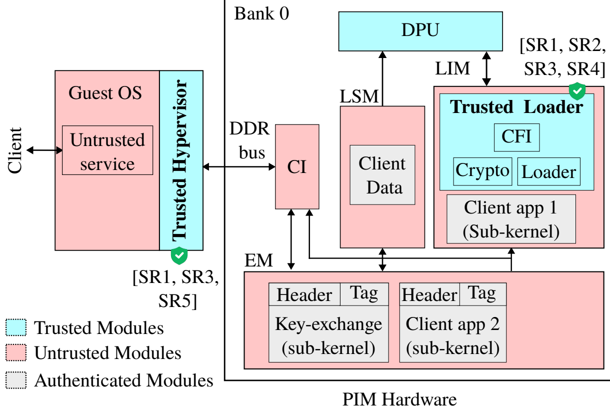

Figure 2: High-level architecture of Memclave. [$SR\,j$] indicates security requirements that are fullfilled by the corresponding trusted block.

only a virtualized interface, removing direct register accesses by the untrusted OS. Additionally, since data still has to flow through the untrusted host, concerns about the integrity and confidentiality of client code and data arise. To address these concerns, the *loader* includes a lightweight cryptographic library to authenticate code/data and to establish a secure channel with the client.

Memclave establishes a verifiable chain of trust from boot. The *hypervisor*, verified via measured boot (e.g., TPM), deploys a First Stage Loader (FSL) onto the DPU. The FSL generates a system key that remains confined within the PIM, wraps the initial client program, and then erases itself from LIM as it hands control to the *loader*. The *loader* becomes the persistent runtime for subsequent authenticated execution.

To support flexible and multi-stage computations under the loader, Memclave introduces *subkernels*, client programs authenticated before execution that can be chained to overcome the limited size of LIM.

Establishing secure execution within PIM raises several challenges including preventing unauthorized CI access, authenticating and executing client code within constrained LIM, and protecting code/data without persistent secure key storage. We next detail how Memclave addresses these with a minimal TCB and a structured subkernel execution model.

5.2 Key Terminologies and Concepts

5.2.1 Key Hierarchy

To enforce a layered authentication model, Memclave defines three cryptographic keys: the System Key (K_{Sys}), the Loader DH Key Pair (K_{DH}^{Priv}, K_{DH}^{Pub}), and the Session Key (K_{Sess}). These keys differ in generation time, lifespan, storage location, ownership, and impact of compromise, as described next.

System Key (K_{Sys}). During system initialization, the FSL generates the *System Key*, K_{Sys}, which remains valid until the next power cycle. Under K_{Sys}, all system subkernels, such as the Key-Exchange and messaging subkernels, are encrypted and authenticated. Any leakage of K_{Sys} would allow an attacker to forge system subkernels. Hence, this key is confined to the DPU's internal registers, accessible only to FSL and the subsequent *loader*; it is never written to EM or exposed to any client subkernel or to the untrusted OS.

Loader DH Key Pair (K_{DH}^{Priv}, K_{DH}^{Pub}). At initialization, FSL generates a DH key pair. The private half, K_{DH}^{Priv}, is encrypted and authenticated using the system key K_{Sys} and stored as part of a special key-exchange subkernel in EM while not in use. The public half, K_{DH}^{Pub}, is placed in EM so the client can retrieve it for the key exchange. The pair is used to derive K_{Sess} for the client. Only the *loader* can access K_{DH}^{Priv} using K_{Sys}, ensuring it never leaves the DPU.

Session Key (K_{Sess}). Following the DH exchange, both client and *loader* compute the *Session Key*, K_{Sess}, which is stored exclusively in the LSM. This key is valid while the client session is active. Under K_{Sess}, all application subkernels and their associated data are encrypted and authenticated using ChaCha20-Poly1305. Ownership and lifecycle management of K_{Sess} rest solely with the client. While a leaked session key would enable an attacker to tamper with application workloads, it does not permit the forging of system subkernels or affect future sessions.

5.2.2 Subkernel Taxonomy

Memclave classifies subkernels into two categories based on their origin, purpose, and authentication key:

System Subkernels are created by the FSL and authenticated/encrypted under the *System Key* (K_{Sys}). They provide essential services for establishing the secure runtime. The *Key-Exchange Subkernel* performs a DH handshake to derive K_{Sess}, and the *Messaging Subkernel* implements mechanisms to communicate with the guest. These subkernels ease the access control and cryptographic workload for a client kernel.

Application Subkernels are supplied by the remote client and authenticated/encrypted under K_{Sess}. They encapsulate domain-specific logic, e.g. machine-learning subkernels.

All subkernels are authenticated (SR1) and encrypted (SR2), failure in authentication results in a fault. This separation allows Memclave to enforce a minimal trusted computing base while supporting flexible, client-controlled workloads.

5.2.3 First Stage Loader (FSL)

The First Stage Loader (FSL) is a minimal, non-persistent module responsible to bootstrap the secure runtime (*loader*). It is loaded into LIM by the *hypervisor* immediately after secure-boot verification, and before booting the guest VM.

FSL performs three key tasks before it yields control to *loader*, replacing itself in LSM. First, it generates the system key K_{Sys}. Second, it creates the loader DH key

pair ($K_{DH}^{Priv}, K_{DH}^{Pub}$), embedding the private half into the Key-Exchange Subkernel for session establishment. Third, using K_{Sys}, it encrypts/authenticates the initial system subkernels. K_{Sys} is placed in the registers of a dedicated key storage thread, before yielding control to the *loader*.

5.3 Memclave Hypervisor

The PIM CI exposes a set of memory-mapped I/O (MMIO) registers through which the host can manage and control DPU execution. These registers are used to launch tasklets, reset threads, configure DMA operations, and access scratch memory. However, they are typically mapped into the host's physical address space without any hardware-enforced access control. A compromised or malicious OS can directly tamper with CI registers, injecting commands, disrupting execution, or leaking sensitive memory regions. Existing TEEs provide no enclave-level isolation for device MMIO. SGX enclaves cannot perform I/O and rely on untrusted OCALLs and IOMMU-mediated DMA [21]; TrustZone's device access control is coarse-grained and SoC-specific rather than per-process, leaving accelerator MMIO outside app-level TEEs [64].

To protect access to PIM's control plane (SR5), Memclave introduces a *hypervisor* as the first software root of trust. It enforces early isolation over the CI registers and mediates secure deployment of the *loader* and system subkernels.

5.3.1 Chain of Trust

Memclave employs TPM-backed remote attestation to ensure only a verified *hypervisor* runs on the system. At boot, the TPM measures the *hypervisor* into Platform Configuration Register (PCR). At runtime, a client can request attestation; the *hypervisor* returns a TPM quote over relevant PCRs, proving integrity and freshness via a client-supplied nonce.

Memclave also measures the *loader* and system subkernels into distinct PCRs; the verifier enforces a policy over the full chain (firmware→bootloader→*hypervisor*→*loader*). Any additional pre-hypervisor layer (e.g., an underlying hypervisor) changes the PCR composite and is therefore rejected by policy. This makes tampering with runtime components remotely detectable. These measurements form a TPM-anchored chain of trust from the *hypervisor* to runtime components.

TPMs operate below the OS and resist tampering even with a fully compromised kernel. Client-supplied nonces in quotes prevent replay. The design follows measured-boot frameworks (e.g., Intel TXT, AMD SKINIT) adapted to PIM-based isolation and attestation. Attestation is on demand: PCR persist until extended, so the *hypervisor* can quote for multiple clients long after boot, enabling secure provisioning without reboot or static trust assumptions [22, 53, 54].

5.3.2 Control-Plane Isolation

Once verified, the *hypervisor* immediately unmaps all CI registers from the guests's address space. This blocks all guest OS components—including the kernel—from issuing unauthorized PIM control commands. This unmapping is performed atomically during *hypervisor* initialization, eliminating any window of exposure before the OS boots.

We separate (i) the *Control Interface* (CI), which are host-visible MMIO registers used to control DPUs and program access from (ii) *External Memory* (EM). In Memclave, the guest *never* receives CI access: the *hypervisor* removes the CI MMIO region from guest mappings and acts as a proxy, issuing a small, whitelisted set of CI operations only to set up bounded DMA to/from EM. Access to EM is exclusively owned by either guest or DPU. A software "ready" line is used to transfer access. Exclusivity is enforced by the *hypervisor*, which is in control of a physical EM MUX present on PIM.

By unmapping the CIs, the *hypervisor* enforces code integrity (SR1) and data confidentiality (SR2), and isolates PIM from guest side accesses (SR5).

5.4 Memclave Loader

After the *hypervisor* initializes and secures the control plane (CI and EM), it enforces secure runtime execution within the constrained DPU environment. However, supporting multiple clients (AR5) opens up the risk of adversarial clients that attempt to leak other clients' sensitive data. To address this, Memclave introduces the *loader*, an isolated and immutable component that resides entirely in execute-only LIM. The *loader* manages authenticated subkernel loading and ensures control flow integrity within critical points of itself (SR1, SR3). Its primary responsibilities include: 1) Authenticated subkernel loading. 2) Strict separation of different clients and separation of clients and *loader*. 3) Storage of the system key K_{Sys}. The following subsections detail these responsibilities.

5.4.1 Secure Initialization and Key Management

Key provisioning during runtime initialization presents significant risks, including key leakage. Memclave mitigates these risks by embedding K_{Sys} in a dedicated key storage thread's registers. The key itself never leaves them. Other threads can request the encryption/decryption of data from the thread. This request is only granted after ensuring that the DPU is in a specific state, so that client code cannot abuse the key storage thread as an encryption/decryption oracle.

During bootup, the FSL generates and securely stores the system key for the *loader* to take over (cf. Section 5.2.3). The *loader* then launches the Messaging Subkernel, waiting for the client's initial connection request. Upon receiving this request via EM buffers, the *loader* invokes the Key-Exchange Subkernel, passing the client's public key via EM. The derived

session key is stored in LSM. User subkernels are authenticated and encrypted using the session key and may additionally use it to perform custom cryptographic operations. On session teardown, the loader removes all user data, including the session key, unless requested otherwise.

5.4.2 Trampoline Enforcement & Controlled Execution

Concurrent threads executing within a subkernel can violate execution integrity by racing ahead of the *loader*'s checks, tampering with LSM or modifying state before isolation barriers are enforced. For example, one thread may manipulate the return addresses pushed to the stack by a thread that is currently attempting to fetch a new subkernel, skipping the check for privileged instructions.

On every entry, the *loader* checks hardware thread-status registers and verifies that only one dedicated thread is active. If any other thread is concurrently running the loader faults immediately and halts execution.

To prevent control-flow hijacking (SR3), subkernels are not permitted to invoke arbitrary loader logic. All control transfers into the loader begin from a defined trampoline entry point, which performs the isolation check and sanitizes execution context. Upon subkernel completion, control is returned explicitly to the trampoline. This structured entry and exit model upholds strict control-flow integrity.

5.4.3 Authenticated Subkernel Loading

Each subkernel carries a 64-byte header (size fields, IV, and TAG) authenticated using ChaCha20-Poly1305 AEAD. Upon invocation, the *loader* decrypts and verifies the header and payload using either the current session key or the system key in case of system subkernels (SR1, SR2). The *loader* is solely responsible for loads into LIM and bans any instruction that facilitates directly or indirectly gaining control of LIM. Hence, before loading the subkernel into LIM, the *loader* explicitly checks for privileged opcodes and registers, such as thread control instructions (`clr_run`, `boot`, `resume`) and LIM writes (`ldmai`). Banning them eliminates the possibility of replacing the *loader* or executing code capable of leaking the system key. If validation fails, a fault is raised. If the subkernel validation passes, the *loader* fills LIM and LSM with the code and data carried in the subkernel.

5.5 Subkernel Management

The limited size of execute-only memory (LIM) constrains the amount of code securely executable within a single DPU invocation. Monolithic programs risk exceeding LIM capacity, while loading arbitrary code blocks risks injection of malicious code and unauthorized instructions, or breaches of data confidentiality. To securely accommodate larger client workloads (AR2), Memclave adopts a modular approach, dividing functionality into self-contained subkernels.

5.5.1 Subkernel Chaining

On termination, subkernels return to the entrypoint of the trusted loader. By default, this causes all data to be wiped and the messaging subkernel to be loaded. Subkernels may choose to persist certain pieces of data and can signal the *loader*, that a specific subkernel should be executed next. The subkernels are identified via their authentication tag and their address in memory. This design allows securely accommodating programs that are larger than LIM, such as our key exchange implementation, further discussed in Section 6.2.1.

5.5.2 Subkernel Lifecycle: Load, Execute, Cleanup

The *loader* manages each subkernel through three stages. In the *Load* stage, it streams in the subkernels code and data in fixed-size chunks into LIM and LSM, verifies the tag to detect tampering and ensure authenticity, and performs checks for size, alignment, and privileged opcodes. In the *Execute* stage, the loader spawns 15 additional threads and transfers control to the subkernel's entrypoint. Subkernels can reuse cryptographic primitives present in the loader, reducing their potential memory footprint. The *loader* removes user data in the *Cleanup* stage, after the Subkernel terminates without specifying a follow-up subkernel.

5.6 Key Exchange Mechanism

Each DPU uses its static DH pair $(K_{DH}^{Priv}, K_{DH}^{Pub})$, which is embedded into a key exchange subkernel, to derive shared keys between a client and itself. The key exchange subkernel that contains the key is encrypted and authenticated using the system key K_{Sys}. We guarantee replay and rollback protection (SR4), by including a monotonically increasing 128-bit per-DPU counter in the final key derivation step. This binds the exchange to the current counter value, making it impossible to force the DPU to recompute older keys. The counter's value is shared with the client before sharing its public key, allowing them to derive the shared key before the first roundtrip.

6 Implementation

We demonstrate Memclave by implementing it on UPMEM's PIM platform (Version 1a @ 350 MHz). UPMEM's PIM introduces some interesting challenges: It lacks a concept of privilege levels, provides no memory isolation, and exposes a powerful control interface to the host CPU. This section details how we realize Memclave on this platform and what platform-specific constraints and costs arise in practice.

6.1 Subkernel Loader

We introduce two software-defined privilege levels, system and user mode. A thread runs in system mode when (i) its

thread ID is 0 and (ii) `r20`/`r21` satisfy the loader's invariant (i.e., are not set to t_{invalid}). We define t_{invalid} outside the valid ranges of both EM and LIM address spaces. The trusted loader has a single entry and exit point for entering and exiting system mode. To avoid privilege escalation, the *loader* performs system mode checks at each critical loading stage. User code is not permitted to access system mode registers. The *loader* scans subkernels that write to these registers, before loading them into LIM.

6.1.1 Secure Key Storage

Security guarantees (SR1, SR2) rely on the confidentiality of the system key. The private loader DH key is sealed using the system key, therefore leaking the system key would also leak the private DH key and in turn, the session key. Each DPU thread has a complete set of its own registers, which is fully isolated from other threads. We dedicate a specific DPU thread for key storage. This thread holds the system key K_{Sys} in 8 of its 24 hardware registers. On request, the thread performs the ChaCha20 block function on an arbitrary IV and counter, and writes the resulting 64 byte matrix into LSM. The full computation is performed in the thread's registers, the key itself never leaves them.

It is practically infeasible to recover the key based on the ChaCha20 block function's outputs. All higher-level cryptographic operations, such as the AEAD construction used to authenticate and decrypt subkernels, are based on this primitive. The mechanism to request execution of the block function requires that the caller is already in system mode. User code cannot abuse the mechanism to create malicious system subkernels nor can it decrypt the loader DH pair.

6.1.2 Enforcement of Control-Flow-Integrity

To perform privilege escalation, a malicious thread can try to jump into any component of the *loader*, violating SR3. Hence, each component of the *loader* that performs some critical function, such as code loading, decryption or authentication has to be protected against jumps from user code. We use registers `r20` and `r21`, in combination with the thread ID (id 0) as an indicator of safe execution path. A system mode assertion follows each critical section within the *loader*. A thread that enters *loader* code at any point other than at the expected entry-point will therefore fault.

On UPMEM, thread state can only be checked sequentially. This leaves room for a TOCTOU attack against the thread state verification, where multiple cooperating malicious threads boot each other up between checks. To prevent this, we prohibit thread control instructions in subkernels.

6.1.3 Protected Code Loading

The CFI protections described above are insufficient for protecting against direct jumps to the `ldmai` instruction. `ldmai` instructs the DPU's DMA engine to copy data from EM into LIM and is the only way to load code on UPMEM's DPUs. Source and target address are passed via registers. A malicious subkernel may set up the target address, such that the execution of `ldmai` removes a system mode assertions following it, bypassing the protection mechanism.

As shown in Figure 3, we therefore treat `ldmai` as an additional *system-mode assertion* by enforcing a register invariant on its operands. All loader `ldmai` sites use `r20` as the destination and `r21` as the source. Upon leaving system mode, the loader sets `r20` and `r21` to t_{invalid}. Consequently, any direct jump (ROP C in Figure 3) to `ldmai` while not in system mode triggers a DMA fault (since `r20`/`r21` hold invalid pointers), preventing further execution. Similarly, any jump (ROP A in Figure 3) to force decryption of data or the extraction of the system key results in a security fault.

6.1.4 Replay Protection

We provide a monotonically increasing 128-bit counter within the *loader* for replay protection purposes (SR4). It is accessed by system subkernels, such as the key exchange subkernel. Each access increments it. This counter remains present for the lifetime of the DPU and is implemented via a dedicated thread, similar to the key storage mechanism. When the counter is requested, the dedicated thread boots up and increments the counter, which is split across four of its registers. The thread faults if the counter overflows.

6.2 Subkernel Development

Subkernels for Memclave are implemented using UPMEM's kernel construction toolchain (a LLVM patch), preventing the allocation of privileged registers `r20` and `r21` when compiling subkernels. Memclave provides a runtime library that ensures correct linkage given the *loader* positioning in LIM as well as provides bindings to core *loader* functionality. Memclave's guest-side library takes care of encrypting and tagging a subkernel with the session key just before being loaded by the DPUs. By introducing only these two modifications to the UPMEM kernel construction workflow, Memclave achieves a reasonably high compatibility with existing kernel code.

6.2.1 Key Exchange Subkernels

Due to the limited amount of LIM, it is not possible to load a full cryptographic library such as mbedtls at once. Even a trimmed down version implementing a complete key exchange exceeded the bounds of LIM on UPMEM. We therefore split the implementation of the key exchange mechanism over multiple subkernels and use subkernel chaining to enforce a strict execution order. The first key exchange stage gathers public values from the client and shares the DPU's monotonic counter. The second stage calculates the shared

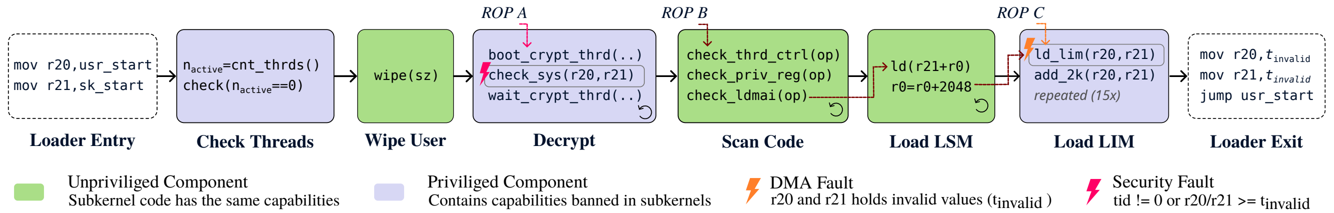

Figure 3: Loader control-flow integrity. Privileged loader stages are reachable only in system-mode; any ROP-style jump into sensitive gadgets triggers a fault. In particular, unauthorized jumps (ROP B, ROP C) to `ldmai` fault due to invalid memory accesses (DMA fault), while jumps (ROP A) to key/crypto-related gadgets raise a security fault.

DH secret over the 2048-bit MODP group defined in RFC 3526. The final stage derives the session key from the monotonic counter and the shared secret. Both the second and third subkernels are linked against mbedtls and use its implementation of DHKE and SHA2 respectively.

6.3 Hypervisor and Guest-Side Development

We implement the *hypervisor* as two cooperating services. One service handles setting up the guest OS, implemented as an extension to QEMU with KVM as its backend [13]. It passes each physical PIM rank through to the guest system, leaving the CI memory region out and replacing it with a virtual one, yielding a significantly reduced amount of control to the guest (SR5). The second service, referred to as *CI-switch*, handles PIM's control plane. Data received on the virtual control interface is trapped and passed to the *CI-switch* via a UNIX socket and processed there.

To synchronize EM access between guest and DPU, we introduce a virtual DPU ready line. The DPU raises this ready line, causing it to halt computation, allowing EM access to the guest. To revoke EM access and resume computation on the DPU, the guest in turn lowers the ready line. The *CI-switch* facilitates this synchronization mechanism and allow guest to query the status of a PIM rank.

We provide a kernel driver and a userspace library to interact with PIM from within the guest system. We provide a porting guide in the appendix.

7 Evaluation

7.1 Evaluation Goals

We evaluate Memclave on real UPMEM hardware using (i) microbenchmarking of core mechanisms, (ii) the Processing-In-Memory Benchmarks (PrIM) benchmarking suite, and (iii) end-to-end programs that span compute-heavy and irregular, memory-bound access patterns. Furthermore, we correlate the runtime overhead to Memclave's mechanisms and empirically validate the design against our SR/AR requirements.

Parameter	UPMEM	Memclave
LIM size (KiB)	24	18△
Usable EM capacity (MiB)	64	63△
EM→LSM (stream) (MiB/s)	140.76	140.52✓
EM↔EM copy (MiB/s)	90.89	90.77✓
Virtual DPU Ready Line (ms)	n/a	4-7ms

Table 2: Static capability (hardware/SDK–defined; invariant across datasets). Parity badges: ✓= indistinguishable from UPMEM (95% CI overlap), △= differs. *Terminology:* LIM/LSM/EM ≡ UPMEM IRAM/WRAM/MRAM.

7.2 Experimental Setup

We run on a dual-socket Intel Xeon Silver 4216 platform (Linux `5.15.0-142-generic`). The system hosts 20 UPMEM DIMMs, yielding 2,560 DPUs; we log the SDK-reported DPU frequency of 350MHz per rank at runtime. We use the UPMEM SDK `upmem-2025.1.0`; guest code is built with `-O3`. Unless stated, all kernels and datasets are from the PrIM benchmark suite [34], which we port to Memclave with minimal glue code; the baselines run the unmodified PrIM suite. Each result is the mean of 5 runs (one warm-up), timed with `CLOCK_MONOTONIC_RAW`; end-to-end breakdowns use fences and, when available, in-DPU timestamps. All baselines run binaries with same algorithm and datasets.

Baselines. CPU-only (*CPU*) provides a point of reference and an upper bound on what the CPU can achieve without offloading. Plain UPMEM PIM (*PIM-insecure*) quantifies the raw timings of PIM and serves as the baseline for overhead normalization. *Memclave* measures our framework where the subkernel is fully encrypted and authenticated for both code and data sections, and the guest has no direct CI access. However, data sent externally post subkernel loading is not included (~~SR2~~). We cite *SE-PIM*, *PIM-Enclave* and *MPC-on-PIM* in a design context only, comparing their capabilities qualitatively with *Memclave*; we do not reimplement or compare them quantitatively.

Metric	Baseline insec.	Memclave auth	Memclave enc
SK load (ms)	8.14	107.98	119.88
Auth; Enc; Scan	n/a	105.52; 0; 0.80	107.53; 9.89; 0.80
SK unload (ms)	0	1.65	1.66
Key exchange (s)	n/a	13.58	13.60
Enc(EM↔LSM) (MiB/s)	n/a	15.43	15.43

Table 3: Dynamic, per-DPU costs. "auth" disables code/model sealing; sealed I/O microbenchmarks are independent of that toggle. Test subkernel size: 6 KiB Text, 10 KiB Data; Buffer size for decryption: 16 MiB

Operation	UPMEM	Memclave	
Workers	8	8	12
Broadcast	12654.11	12554.88	9656.06
Scatter	4686.45	4785.20	6853.39
Gather	2442.19	1816.51	2598.52

Table 4: Guest ↔ EM throughput in MiB/s; 16 MiB per DPU; Measurements per Rank

7.3 Microbenchmarks

We first perform microbenchmarks such that application-specific experimental results can be traced to specific mechanisms. We use two variants of Memclave, *Memclave-auth* and *Memclave-enc*. *Memclave-auth* does not encrypt the subkernel, reducing the cost when confidentiality is not needed, e.g. for open source models. *Memclave-enc* uses a fully encrypted subkernel, including both code and data sections. We compare *Memclave-auth* and *Memclave-enc* against *PIM-insecure*. All measurements are per-DPU; parity is judged via 95% confidence-interval overlap over five runs.

Microbenchmarks: To rule out the core-DPU compute/memory as confounders, we create microbenchmarks that contain DPU centric operations without the overhead of kernel loading or guest interactions. The results align with this design: EM↔LSM streaming is 140.76 vs. 140.52 MiB/s (✓), and EM↔EM copy is *90.89 MiB/s* vs. *90.77 MiB/s* (✓); see Section 7.2. The only static differences are small capacity reserves (LIM 24 → 18 KiB, EM 64 → 63 MiB), which do not affect steady-state bandwidths. *hypervisor* incurs the overhead for lowering the DPU ready line replacing previously accessible CI functionality. Next we measure the dynamic overheads in Memclave mechanisms (e.g., staging, sealing).

Overhead of Security Primitives: Memclave confines security enforcement to *staging* and *I/O* boundaries, leaving steady-state compute/memory paths identical to UPMEM. Its measurable costs thus concentrate in (i) subkernel staging, (ii) per-session key setup, and (iii) sealed EM→LSM transfers.

Subkernel staging. We time a representative subkernel (SK) load, reporting its authenticated load, decryption (for Memclave-enc), and opcode scan; teardown measures register and LSM reset. As observed in Section 7.2, loads on *Memclave-enc* take almost 14× longer compared to *PIM-insecure*. Subkernel authentication contributes for a significant portion of this overhead. Poly1305 heavily relies on modular multiplication, which UPMEM's primitive DPUs cannot efficiently perform with the lack of native 32×32 bit integer multiplication instructions. As the subkernel load is a one-time overhead, size-dependent cost amortizes over long-running kernels but is visible for short or many tiny stages.

Session key exchange. We perform one key exchange during the session setup per-DPU using the three chained key exchange subkernels. The latency for the full session establishment is at 13.6s per DPU. The major contributor to the overall runtime is mbedtls's `mpi_exp_mod` function, which heavily relies on integer multiplication.

Encrypted DMA. With a fixed 64 B chunk over a 16 MiB payload, the sealed path exhibits 15.43 MiB/s per DPU. Full DMA encryption causes some workloads on the fundamentally compute bound DPUs [33] to appear memory bound, as throughput drops ten-fold from 140.52 MiB/s to 15.43 MiB/s.

Memory Transfer Overheads: Transfers from the guest's memory to EM are subject to extra costs due to virtualization overheads and re-implementing transfer routines. The transfers are categorized as: (1) Broadcasts, which transfer the same data to all DPUs of a rank. (2) Scatters, transferring unique data to each DPU of a rank. (3) Gathers, to fetch data from each DPU of a rank. UPMEM's transfer routines limit the maximum worker count to 8, Memclave imposes no limits. Improvements beyond 12 workers are negligible. As shown in table 4, Memclave suffers from a drop in throughput speeds for gather operations given an identical worker count. Scatter and gather operations improve with 12 worker threads, increasing CPU cost, while broadcast performance worsens.

7.4 PrIM benchmarks

We evaluate Memclave on the PrIM benchmark suite, a common set of PIM workloads with heterogeneous memory-access, operation, and communication patterns. We port all 16 benchmarks, exhibiting Memclave's compatibility across a wide range of subkernels. We evaluate Memclave against the *CPU* and *PIM-insecure* (UPMEM baselines) for (i) subkernel execution time (ii) the corresponding guest→DPU and (iii) DPU→guest transfer times.

Figure 4 (a) shows that Memclave's subkernel execution largely remains close to the *PIM-insecure* baseline for compute-intensive subkernels (e.g., MLP), while the relative gap grows for very short-running subkernels, where fixed costs dominate or that require frequent guest coordination. Across PrIM, Memclave's overhead is a combination of workload-dependent costs and a static one-time cost per benchmark run. The one-time overhead of ∼100 ms includes

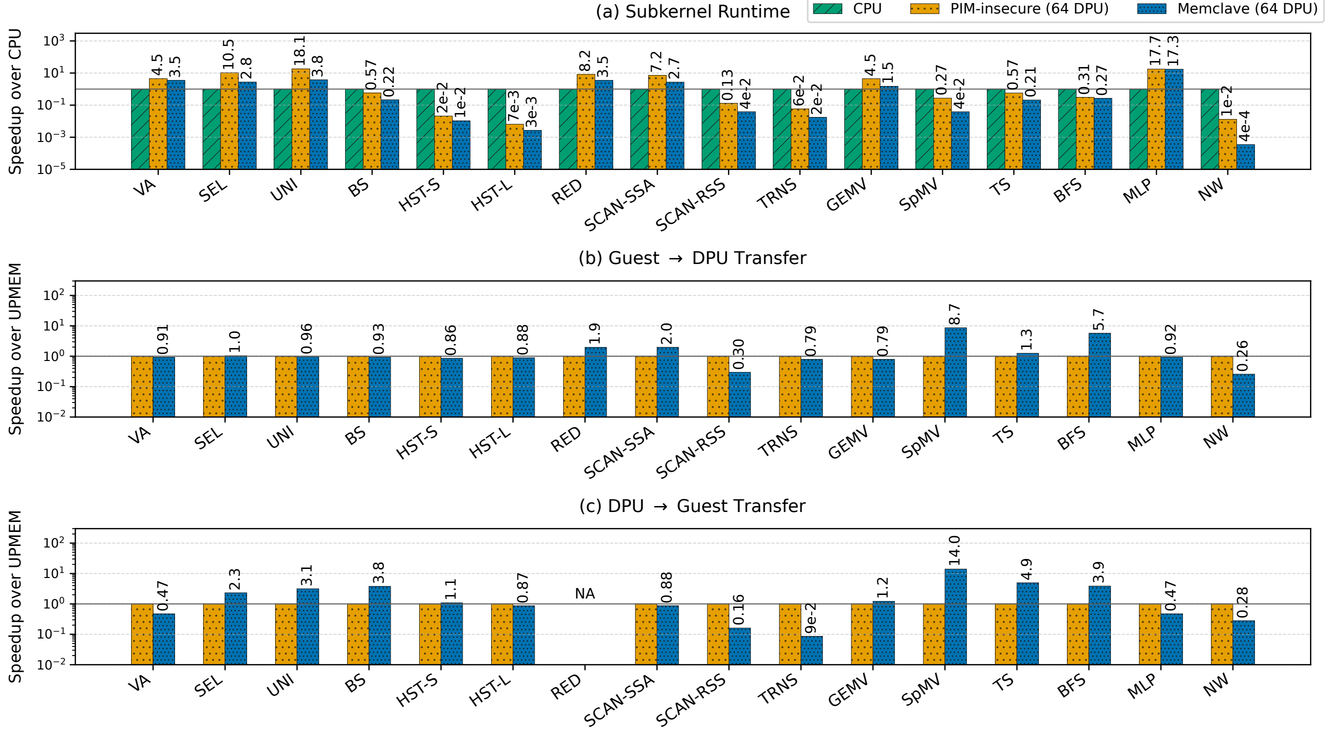

Figure 4: PrIM benchmark breakdown. (a) Subkernel runtime speedup over *CPU* for *CPU*, *PIM-insecure*, and *Memclave*. (b) Guest→DPU transfer time: *Memclave* speedup over *PIM-insecure* (UPMEM baseline). (c) DPU→guest transfer time: *Memclave* speedup over *PIM-insecure*.

subkernel authentication, integrity checks, and detection of privileged instructions/registers; this cost is amortized for compute-intensive subkernels.

Figure 4 (b) and (c) shows that transfer behavior is strongly workload-, pattern-, and direction-dependent: several workloads are near parity with *PIM-insecure*, and some even exceed it (due to more worker threads), while others incur substantial slowdowns, often on gather-heavy DPU→guest paths (e.g., TRNS/SCAN-RSS) or fine-grained transfer patterns.

7.5 MLP Inference

The Multilayer Perceptron (MLP) is a deep learning model composed of multiple fully connected layers, widely used for tasks such as classification, regression, and feature extraction. We select MLP inference as a regular, compute-dense workload whose inner loop is a dense GEMV followed by a simple nonlinearity. Its predictable access pattern and high arithmetic intensity make it well suited to study how Memclave overhead amortizes as compute grows.

Workload and Execution Mapping We use a standard row-partitioned MLP. For each layer, the guest broadcasts the current input vector to all DPUs; each DPU holds a disjoint block of rows in EM, loads into LSM to compute, and writes its output slice back to EM. The guest then gathers per-DPU slices, applies ReLU, and rebroadcasts the result for next layer. The subkernel is loaded once at startup and reused across layers, preventing per-layer authenticated-loading overhead.

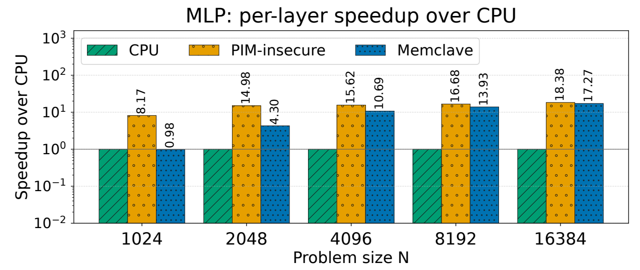

Figure 5: MLP inference. Per-layer DPU runtime speedup over *CPU* versus layer width N (shown for *PIM-insecure* and Memclave; *CPU* baseline is 1).

Datasets and metrics. We sweep dense, square layer widths $N \in \{1k, 2k, 4k, 8k, 16k\}$. Inputs and weights are application-generated, and the topology remains same across runs. We measure per-layer speedup over *CPU* as reference baseline and compare *PIM-insecure* against *Memclave*.

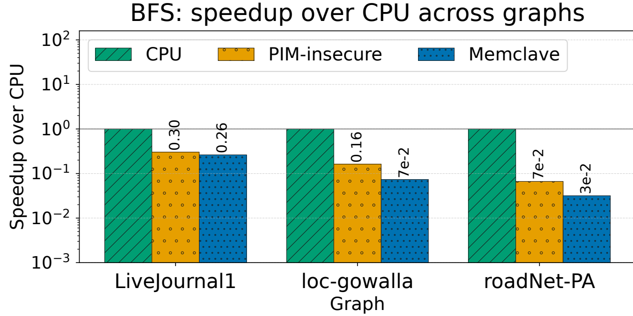

Figure 6: BFS. DPU runtime speedup over *CPU* across graphs (*CPU* baseline is 1), comparing *PIM-insecure* and *Memclave*.

Results. Figure 5 shows per-layer speedup over *CPU* as we increase the layer width N. For small widths, Memclave underutilizes the DPUs and the runtime is dominated by fixed guest-mediation overhead, so the benefit of acceleration is limited. As N increases, the GEMV compute dominates, and Memclave converges toward the *PIM-insecure* baseline, amortizing the overhead. For realistic widths ($N \geq 4$k), Memclave incurs $\leq 1.5\times$ overhead over *PIM-insecure* per layer at practical widths.

7.6 Breadth–First Search (BFS)

PIM benefits from linear workloads with large inputs where parallel computation is possible. Hence, MLP performs better on PIM than on a CPU. However, we do not expect a client to run only linear, PIM favorable tasks. To assess whether Memclave is UPMEM compatible beyond PIM-friendly kernels, we evaluate BFS, which stresses data-dependent control flow and repeated global coordination. Note that current MPC-based security solutions are incompatible with non-linear workloads if one wants to maintain result authenticity [29,80].

Workload and Execution Mapping We use a level-synchronous BFS. At each level, the guest provides a (global) frontier bitmap; each DPU range-partitions the vertex set and scans only its local vertices that are in the frontier. DPU threads traverse adjacency lists from the graph stored in EM, update a global *visited* bitmap, and set bits in a global *next-frontier* bitmap in EM (synchronized via a mutex to avoid races). The guest then gathers the next frontier and repeats; because UPMEM lacks DPU–DPU messaging, cross-partition coordination is guest-mediated. Under Memclave, the traversal logic on the DPU is unchanged; differences mainly arise from guest mediation and per-level guest–DPU coordination.

Datasets and metric. We evaluate three real graphs with distinct frontier shapes: *loc-gowalla* (moderate diameter and width with 196,591 nodes, 950,327 edges, 10 levels) [19], *LiveJournal1* (shallow diameter, wide frontiers with 4,847,571 nodes, 68,993,773 edges, 15 levels) [11,48], and *roadNet-PA* (long diameter, thin frontiers with 1,088,092 nodes, 1,541,898 edges, 542 levels) [23,48].

Results Figure 6 shows speedup over *CPU*. In contrast to the compute-intensive MLP, BFS is dominated by irregular, data-dependent memory accesses and repeated guest-mediated coordination across levels, so the *CPU* baseline remains fastest for all three graphs. Relative to *PIM-insecure*, Memclave adds $1.1\times$ overhead on LiveJournal1, where wide frontiers increase useful work per level and amortize guest-mediation costs. In contrast, for loc-gowalla and roadNet-PA, where the work per level is smaller and coordination is more prominent (and for roadNet-PA, repeated over many levels), Memclave 's additional guest-mediation overhead slows down the runtime relative to *PIM-insecure*.

7.7 Implementation Footprint

TCB size (code & memory). On-device, the *loader* occupies a fixed 8 KiB in LIM. We reserve 1 MiB of EM for system subkernels (key exchange, messaging), leaving 63 MiB usable. The DPU-resident TCB is $\sim$ 0.9k SLOC; the hypervisor mediation that locks the CI is $\sim$ 1.1k SLOC.

Porting effort & API intrusiveness. DPU kernels are compatible (tasklet loops, mutexes, DMA sizes). On the guest, changes are replacing the direct DPU communication calls with Memclave's drop-in secure variants. In practice this amounts to tens of lines of code per application.

7.8 Cross-System Capability & Comparisons

Section 7.8 contrasts *deployability* and *capabilities*. Hardware-enhancement designs show low–moderate overhead, but are demonstrated only using simulations. MPC-on-PIM keeps the device untrusted and achieves LA confidentiality, but shifts large parts of the protocol to the host while leaving the on-PIM control plane unauthenticated. In contrast, Memclave attests a LIM-resident loader, isolates PIM and provides on-dpu crypto capabilities, without reducing the worktype coverage.

7.9 Security Analysis

We illustrate how the protections summarized in Table 6 materialize against three high-impact threats.

Guest snoops/tampers with subkernel, inputs, or outputs (touches SR1/SR4). A privileged guest may attempt to read or modify subkernel code or data in transit, or alter outputs. *Defence.* Our framework counters this by ensuring that code and data loaded as a subkernel is kept authentic (SR1) and confidential. It gives client code a key that can be used to encrypt/authenticate outputs via libraries such as mbedtls,

Framework	Impl.	SW only	Crypto.	Hardening	TCB	Overhead	PrIM Compatibility
UPMEM	HW	—	✗	✗	—	1× (baseline)	✓
SE-PIM / PIM-Enclave	Ⓢ	✗	AES-GCM	Bank lock + AEAD	Medium	Low–Mod	△†
MPC-on-PIM	HW	✓	SS + GC	Enc compute	Large	High	◐
Memclave	HW	✓	CC20-P1305	CI lock + EncDMA	Hyp + LIM	Low–Mod	✓

Table 5: Cross-framework capability, mechanisms, and deployment. Legend: ✓ Supported, △ Porting required; no paradigm change, ✗ Unsupported, ◐ Major paradigm change and porting effort, Ⓢ Sim-only, †No explicit evaluation, HW = real hardware.

Claim (SR)	Mechanisms	Enforcement (who @ when)	Overhead
SR1 [Code integrity]	Authenticated Subkernels; staged load	Loader @ load-time	99.85 ms
SR2 [Data confidentiality]	Encrypted channel; per-session keys; sealed outputs	Hyp @ execution; DPU crypto @ emit-time	10.126 ms / two way transfer
SR3 [Control-flow & privilege safety]	Trampoline CFI; restricted ABI; forbidden-opcode filter	Loader @ call-time	2.9 ms
SR4 [Anti-replay/rollback]	Counter-bound key derivation	Trusted thread @ call time	≈ 0
SR5 [CI register lockdown]	Early CI unmap	Hypervisor @ pre-boot	≈ 0 runtime

Table 6: Security guarantees summary. Enforcement shows who applies the check and when it runs.

ensuring that outputs cannot be altered. Our frameworks key exchange mechanism ensures, that an adversary cannot replay or rollback prior sessions. (SR4)

User subkernel attempts ROP-style escalation to privileged control (SR3). A malicious subkernel may craft an indirect control transfer to bypass the entry path and jump into privileged loader routines. *Defence.* Trampoline-based CFI restricts indirect targets to whitelisted stubs, the restricted ABI validates call entry, and a forbidden-opcode filter blocks privileged operations, collectively preventing escalation (SR3).

Post-boot CI register abuse (SR5). After boot, the guest may attempt to read or write CI registers to introspect or steer DPU execution. *Defence.* The hypervisor unmaps CI and mediates access before any guest driver initializes, leaving no window in which CI is exposed (SR5). Enforcement occurs at pre-boot, before guest initialization.

8 Discussion

8.1 VM Overhead

Memclave runs in a KVM/QEMU VM on a Linux host [13, 44], which introduces mediation costs orthogonal to PIM. Memclave's DPU ready line induces VM exits and traverses the guest→KVM→QEMU→KVM→guest path; EM↔ guest transfers suffer from longer page-table walks and, in the worst-case, may trigger two page-fault handlers in one access. Prior work quantifies that reducing user-space mediation (or moving backends in-kernel) would shrink overheads [16]. Our reported in-VM overheads are therefore a conservative upper bound, an in-kernel backend is expected to reduce the DPU ready line cost. Additionally, the stock UPMEM stack exploits CI-assisted routines to stage code to LIM and move data to LSM; our security policies disable CI fast paths, further widening the gap to the insecure baseline.

8.2 Generality and applicability

Beyond UPMEM. Memclave's mechanism focuses on four abstractions: a gateable command path to the device, a small execute-only region for an authenticated loader, a bulk DMA-addressable data region, and a means to enforce guest-exclusive vs. device-exclusive windows over that region. These abstractions recur in commercial PIM families. Samsung's HBM-PIM integrates programmable logic within the HBM stack and is demonstrated on an unmodified commercial host without source changes [47]. It includes a logic-layer processing unit and local SRAM with a host-visible command path [41]. We therefore treat Memclave's control-plane mediation and small authenticated routine as a software mapping onto those exposed control/compute primitives. Likewise, AxDIMM provides a DIMM-compatible near-memory acceleration path via the standard DDR interface and is recognized by the OS as normal memory, enabling host-controlled bulk, in-place processing [46]. In both cases, the control and data interfaces satisfy Memclave's minimal requirements.

Beyond PIM. At a high level, accelerators (GPUs, NPUs, FPGAs, and PIM) share a common security requirement for the *control plane*. The control plane configures and/or dispatches work via MMIO registers and queues, which is controlled by privileged host software. To secure end-

to-end, accelerator TEEs either (i) add device-side isolation/attestation primitives (e.g., GPU/accelerator TEEs and hardware-enforced access control) [28, 77, 78], (ii) introduce a trusted mediator outside the untrusted OS [40, 81], or (iii) extend CPU TEE invariants to I/O interconnect [74]. These solutions rely on a trusted hardware anchor on device, the host CPU, or trusted I/O interconnect support.

In contrast, Memclave provides a security framework for PIM in the absence of device side roots of trust, yielding four building blocks. (i) gated control plane: a privileged mediator must have exclusive ownership of the device's control plane. On UPMEM PIM, this is implemented by removing CI access. On GPUs, the analogous surface is the driver-controlled submission path using memory buffers (command queues) and PCIe MMIO registers to notify the GPU command processor. (ii) privileged runtime: an immutable, measured runtime that runs only authenticated workloads. On UPMEM PIM, loader gains privilege leveraging execute-only LIM. On GPUs, TEE designs achieve a similar role with device-side trusted firmware that enforces secure execution. (iii) Confinement: to prevent privilege escalation, the runtime must sandbox untrusted workloads. On UPMEM PIM, confinement is enforced by single-entry/single-exit loader and restricting a small set of instructions (thread control/ldmai) and registers (r20/r21). On GPUs, analogous confinement can be enforced at command submission via command processor validation and address space isolation (GPU virtual memory/page tables). (iv) key storage: secrets must be restricted to privileged runtime only. On UPMEM PIM, system keys are stored in hardware thread registers. Crypto operations run in this thread so the key never leaves registers. On GPUs, keys can be rooted in device-unique secrets fused into eFuses and managed by trusted firmware so they never leave the device.

9 Conclusion

Memclave shows that a *software-only* design can provide integrity and confidentiality to commodity PIM today by leveraging a TPM–attested *hypervisor* and a lightweight in-DPU *loader*. We evaluate Memclave on real hardware (UPMEM DIMMs). Across workloads, steady-state PIM compute remains close to the *PIM-insecure* baseline, (MLP stays within 1.5× at practical widths), while end-to-end overhead is dominated by authenticated loading and guest mediation for short-running workloads. For irregular BFS, the overhead is modest on some graphs (1.1× on LiveJournal1) and increases with the number of frontier levels. Memclave offers a near drop-in guest API while keeping the on-device trusted base compact.

Acknowledgements

We thank the reviewers for their valuable feedback that significantly improved the paper. This work was supported in part by the German Research Foundation (DFG) priority program 'Disruptive Memory Technologies' (SPP 2377).

Ethical Considerations

PIM is an emerging architecture that adds a compute module near memory to reduce data movement. This improves efficiency for data-intensive workloads and hence becomes attractive for multi-tenant cloud deployments. However, today's commodity PIM devices expose a host-controlled control plane and lack built-in security primitives for isolation and attestation. This motivates our study to investigate UPMEM PIM and to design a software-only framework that provides confidentiality and integrity on currently available hardware, without requiring vendor hardware changes.

Stakeholders and benefits. Primary stakeholders are (i) cloud tenants with data-intensive workloads on PIM, (ii) cloud providers who manage these platforms, and (iii) PIM vendors. Memclave is a defensive mechanism against a compromised guest OS or co-tenant which can read or tamper with tenant sensitive information on PIM. PIM has applications for data-intensive workloads such as graph analytics and ML inference. The commodity PIM lacks security primitives, hence there is need for confidentiality and integrity for tenants without needing new hardware changes. The security community benefits from a concrete software only design and implementation of PIM enclave, and also helps future architectures.

Potential misuse. Same as any TEE systems, a malicious tenants could use such framework to hide abusive workloads from benign cloud operators, and make forensic analysis harder. Our artifacts consists of software defense framework and do not include attack gadgets.

Experimentation and Mitigations. All experiments use public, non-sensitive PIM benchmarks (e.g., MLP, BFS from PRIM) on our internally shared hardware, with no real tenants. The benchmarks emulate realistic PIM workloads without using any proprietary/user data. We found and disclosed no vendor vulnerabilities. If Memclave were deployed in practice, it would be best to combine them with abuse-detection techniques (e.g., rate limiting, logging at trust boundaries) to balance confidentiality with platform safety.

Open Science

All our code artifacts *hypervisor*, *loader*, FSL, guest library, benchmarks, and build/evaluation scripts are available on Zenodo: doi.org/10.5281/zenodo.17986462, enabling full reproduction of figures/tables on commodity UPMEM hardware.

References

[1] Arm security technology: Building a secure system using trustzone technology. White Paper PRD29-GENC-

009492C, ARM Limited, 2009.

[2] Trusted Platform Module Library, Part 1: Architecture. Specification Family 2.0, Level 00, Revision 01.38, Trusted Computing Group, September 2016.

[3] TCG Trusted Attestation Protocol (TAP) Information Model for TPM Families 1.2 and 2.0 and DICE Family 1.0. Specification (Public Review Draft) Version 1.0, Revision 0.29A, Trusted Computing Group, January 2019.

[4] AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More. White paper, Advanced Micro Devices, Inc., January 2020.

[5] TCG PC Client Platform TPM Profile Specification for TPM 2.0. Specification Version 1.05, Revision 14, Trusted Computing Group, September 2020.

[6] Tpm 2.0 keys for device identity and attestation. Specification Version 1.00, Revision 12, Trusted Computing Group, October 2021.

[7] The security design of the AWS nitro system. Whitepaper, Amazon Web Services, Inc., November 2022.

[8] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 105–117, 2015.

[9] AMD. AMD SEV-TIO: Trusted I/O for Secure Encrypted Virtualization. *AMD Whitepaper*, March 2023.

[10] ARM Limited. ARM Security Technology: Building a Secure System using TrustZone Technology. Online documentation (latest).

[11] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. Group formation in large social networks: Membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '06)*, pages 44–54. ACM, 2006.

[12] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization.

[13] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[14] Henk Birkholz, Dave Thaler, Michael Richardson, Ned Smith, and Wei Pan. Remote ATtestation procedureS (RATS) Architecture. RFC 9334, January 2023.

[15] Amirali Boroumand, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur, Daehyun Kim, Aki Kuusela, Allan Knies, Parthasarathy Ranganathan, and Onur Mutlu. Google workloads for consumer devices: Mitigating data movement bottlenecks. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, page 316–331. Association for Computing Machinery.

[16] Jiahao Chen, Dingji Li, Zeyu Mi, Yuxuan Liu, Binyu Zang, Haibing Guan, and Haibo Chen. Security and performance in the delegated user-level virtualization. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 209–226, Boston, MA, 2023.

[17] Pau-Chen Cheng, Wojciech Ozga, Enriquillo Valdez, Salman Ahmed, Zhongshu Gu, Hani Jamjoom, Hubertus Franke, and James Bottomley. Intel tdx demystified: A top-down approach, 2023.

[18] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. Prime: a novel processing-in-memory architecture for neural network computation in reram-based main memory. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, page 27–39. IEEE Press, 2016.

[19] Eunsong Cho, Seth A. Myers, and Jure Leskovec. Friendship and mobility: User movement in location-based social networks. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '11)*, pages 1082–1090. ACM, 2011.

[20] Amit Choudhari, Shorya Kumar, and Christian Rossow. Nicraft: Malicious nic firmware-based cache side-channel attack. page 64–83, Berlin, Heidelberg, 2025. Springer-Verlag.

[21] Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptol. ePrint Arch.*, 2016:86, 2016.

[22] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 857–874, Austin, TX, 2016. USENIX Association.

[23] Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 2009.

[24] Fabrice Devaux. The true processing in memory accelerator. In *2019 IEEE Hot Chips 31 Symposium (HCS)*, pages 1–24, 2019.

[25] Safaa Diab, Amir Nassereldine, Mohammed Alser, Juan Gómez Luna, Onur Mutlu, and Izzat El Hajj. A framework for high-throughput sequence alignment using real processing-in-memory systems. *Bioinformatics*, 39(5):btad155, 03 2023.

[26] Kha Dinh Duy and Hojoon Lee. Pim-enclave: Bringing confidential computation inside memory. *CoRR*, abs/2111.03307, 2021.

[27] Kha Dinh Duy and Hojoon Lee. SE-PIM: in-memory acceleration of data-intensive confidential computing. *IEEE Trans. Cloud Comput.*, 2023.

[28] Erhu Feng, Dahu Feng, Dong Du, Yubin Xia, and Haibo Chen. snpu: Trusted execution environments on integrated npus. In *51st ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2024, Buenos Aires, Argentina, June 29 - July 3, 2024*, pages 708–723. IEEE, 2024.

[29] Sahar Ghoflsaz Ghinani, Jingyao Zhang, and Elaheh Sadredini. Enabling low-cost secure computing on untrusted in-memory architectures, 2025.

[30] S. Ghose, A. Boroumand, J. S. Kim, J. Gómez-Luna, and O. Mutlu. Processing-in-memory: A workload-driven perspective. *IBM Journal of Research and Development*, 63(6):3:1–3:19, 2019.

[31] Michael Misiu Godfrey and Mohammad Zulkernine. Preventing cache-based side-channel attacks in a cloud environment. *IEEE Trans. Cloud Comput.*, 2014.

[32] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, May 1996.

[33] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F. Oliveira, and Onur Mutlu. Benchmarking a new paradigm: An experimental analysis of a real processing-in-memory architecture. *CoRR*, abs/2105.03814, 2021.

[34] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F. Oliveira, and Onur Mutlu. Benchmarking a new paradigm: Experimental analysis and characterization of a real processing-in-memory system. *IEEE Access*, 10:52565–52608, 2022.

[35] Google Cloud. Compute engine overview, 2025.

[36] Karan Grover, Shruti Tople, Shweta Shinde, Ranjita Bhagwan, and Ramachandran Ramjee. Privado: Practical and secure dnn inference with enclaves, 2019.

[37] Nastaran Hajinazar, Geraldo F. Oliveira, Sven Gregorio, João Dinis Ferreira, Nika Mansouri Ghiasi, Minesh Patel, Mohammed Alser, Saugata Ghose, Juan Gómez-Luna, and Onur Mutlu. Simdram: a framework for bit-serial simd processing using dram. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 329–345, New York, NY, USA, 2021. Association for Computing Machinery.

[38] Bongjoon Hyun, Taehun Kim, Dongjae Lee, and Minsoo Rhu. Pathfinding future pim architectures by demystifying a commercial pim technology. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 263–279, 2024.

[39] Intel. Intel TDX Connect Architecture Specification. Technical report, Intel Corporation, 2023. Document Number: 354629.

[40] Insu Jang, Adrian Tang, Taehoon Kim, Simha Sethumadhavan, and Jaehyuk Huh. Heterogeneous isolated execution for commodity gpus. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, pages 455–468. ACM, 2019.

[41] Jin Hyun Kim. Aquabolt-XL: Samsung HBM2-PIM with in-memory processing for machine learning accelerators. Hot Chips 33 (HC33) slides, 2021. https://www.hc33.hotchips.org/assets/program/conference/day1/20210813_HC33_Aquabolt-XL_PIM_Jin_Kim_slide.pdf.

[42] Jin Hyun Kim, Shin-Haeng Kang, Sukhan Lee, Hyeonsu Kim, Yuhwan Ro, Seungwon Lee, David Wang, Jihyun Choi, Jinin So, YeonGon Cho, JoonHo Song, Jeonghyeon Cho, Kyomin Sohn, and Nam Sung Kim. Aquabolt-xl hbm2-pim, lpddr5-pim with in-memory processing, and axdimm with acceleration buffer. *IEEE Micro*, 2022.

[43] Jan Kiszka. Hard partitioning for linux: The jailhouse hypervisor. Linux Foundation / LinuxCon NA Slides, August 2015.

[44] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, Ottawa, Canada, 2007.

[45] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, and Kai Engelhardt. sel4: Formal verification of an os kernel. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[46] Donghun Lee, Minseon Ahn, Jungmin Kim, Oliver Rebholz, Jinin So, Jong-Geon Lee, Jeonghyeon Cho, Vishnu Charan Thummala, Ravi Shankar JV, Sachin Suresh Upadhya, Mohammed Ibrahim Khan, and Jin Hyun Kim. Improving in-memory database operations with acceleration dimm (AxDIMM). In *Data Management on New Hardware (DaMoN)*, 2022.

[47] Sukhan Lee, Shin-haeng Kang, Jaehoon Lee, Hyeonsu Kim, Eojin Lee, Seungwoo Seo, Hosang Yoon, Seungwon Lee, Kyounghwan Lim, Hyunsung Shin, Jinhyun Kim, O Seongil, Anand Iyer, David Wang, Kyomin Sohn, and Nam Sung Kim. Hardware architecture and software stack for pim based on commercial dram technology : Industrial product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 43–56, 2021.

[48] Jure Leskovec and Andrej Krevl. Stanford large network dataset collection (snap). https://snap.stanford.edu/data/, 2014.

[49] Wen Li, Ying Wang, Huawei Li, and Xiaowei Li. Leveraging memory pufs and pim-based encryption to secure edge deep learning systems. In *2019 IEEE 37th VLSI Test Symposium (VTS)*, 2019.

[50] Wen Li, Ying Wang, Huawei Li, and Xiaowei Li. P3m: a pim-based neural network model protection scheme for deep learning accelerator. ASPDAC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[51] Ning Lin, Xiaoming Chen, Chunwei Xia, Jing Ye, and Xiaowei Li. Chaopim: A pim-based protection framework for dnn accelerators using chaotic encryption. In *2021 IEEE 30th Asian Test Symposium (ATS)*, 2021.

[52] Zhibo Liu, Yuanyuan Yuan, Yanzuo Chen, Sihang Hu, Tianxiang Li, and Shuai Wang. Deepcache: Revisiting cache side-channel attacks in deep neural networks executables. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, CCS '24, page 4495–4508, New York, NY, USA, 2024. Association for Computing Machinery.

[53] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. Trustvisor: Efficient TCB reduction and attestation. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 143–158. IEEE, 2010.

[54] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys '08)*, pages 315–328, Glasgow, Scotland, UK, 2008. ACM.

[55] Kayvan Memarian, Ben Simner, David Kaloper-Meršinjak, Thibaut Pérami, and Peter Sewell. Ghost in the android shell: Pragmatic test-oracle specification of a production hypervisor. In *Proceedings of the 30th ACM Symposium on Operating Systems Principles (SOSP '25)*. Association for Computing Machinery, 2025.

[56] Microsoft. Hyper-v overview, August 2025.

[57] Microsoft. Virtualization-based security (VBS), February 2025.

[58] Masanori Misono, Dimitrios Stavrakakis, Nuno Santos, and Pramod Bhatotia. Confidential vms explained: An empirical analysis of AMD SEV-SNP and intel TDX. In *Abstracts of the 2025 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS 2025, Stony Brook, NY, USA, June 9-13, 2025*. ACM, 2025.

[59] Onur Mutlu. Processing data where it makes sense in modern computing systems: Enabling in-memory computation. In *2018 7th Mediterranean Conference on Embedded Computing (MECO)*, pages 8–9, 2018.

[60] Onur Mutlu. Pim security seminar: Challenges and techniques. November 2023.

[61] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. A modern primer on processing in memory. *CoRR*, abs/2012.03112, 2020.

[62] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. *A Modern Primer on Processing in Memory*, pages 171–243. Springer Nature Singapore, Singapore, 2023.

[63] Lifeng Nai, Ramyad Hadidi, Jaewoong Sim, Hyojong Kim, Pranith Kumar, and Hyesoon Kim. Graphpim: Enabling instruction-level pim offloading in graph computing frameworks. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 457–468, 2017.

[64] Bernard Ngabonziza, Daniel Martin, Anna Bailey, Haehyun Cho, and Sarah Martin. Trustzone explained: Architectural features and use cases. In *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*, pages 445–451, 2016.

[65] Hyunyoung Oh, Adil Ahmad, Seonghyun Park, Byoungyoung Lee, and Yunheung Paek. Trustore: Side-channel resistant storage for sgx using intel hybrid cpu-fpga. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 1903–1918, New York, NY, USA, 2020. Association for Computing Machinery.

[66] Anjaly Parayil, Jue Zhang, Xiaoting Qin, Íñigo Goiri, Lexiang Huang, Timothy Zhu, and Chetan Bansal. Towards cloud efficiency with large-scale workload characterization, 2024.

[67] PCI-SIG. TEE Device Interface Security Protocol (TDISP) Specification. Technical report, PCI-SIG, 2024. Version 1.0.

[68] Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-chin Wang, and K. K. Ramakrishnan. Spright: extracting the server from serverless computing! high-performance ebpf-based event-driven, shared-memory processing. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 780–794, New York, NY, USA, 2022. Association for Computing Machinery.

[69] Andrew Regenscheid and Karen Scarfone. Bios integrity measurement guidelines. NIST Special Publication 800-155 (Initial Public Draft), National Institute of Standards and Technology, Gaithersburg, MD, December 2011.

[70] Naresh Kumar Sehgal, Pramod Chandra P. Bhatt, and John M. Acken. *Cloud Workload Characterization*. Springer International Publishing, 2020.

[71] Vivek Seshadri, Yoongu Kim, Chris Fallin, Donghyuk Lee, Rachata Ausavarungnirun, Gennady Pekhimenko, Yixin Luo, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. Rowclone: fast and energy-efficient in-dram bulk data copy and initialization. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, page 185–197, New York, NY, USA, 2013. Association for Computing Machinery.

[72] Vivek Seshadri, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie Kim, Michael A. Kozuch, Onur Mutlu, Phillip B. Gibbons, and Todd C. Mowry. Ambit: in-memory accelerator for bulk bitwise operations using commodity dram technology. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, page 273–287, New York, NY, USA, 2017. Association for Computing Machinery.

[73] Siemens. Jailhouse: Linux-based partitioning hypervisor. GitHub repository.

[74] Supraja Sridhara, Andrin Bertschi, Benedict Schlüter, Mark Kuhne, Fabio Aliberti, and Shweta Shinde. ACAI: protecting accelerator execution with arm confidential computing architecture. In *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. USENIX Association, 2024.

[75] UPMEM. Processing In-Memory: Ultra-Efficient Acceleration for Data-Intensive Applications. Technical report, UPMEM, 2024.

[76] UPMEM SAS. *User Manual — UPMEM DPU SDK 2025.1.0 Documentation*, 2025. Version 2025.1.0, accessed 2025-08-17.

[77] Kapil Vaswani, Stavros Volos, Cédric Fournet, Antonio Nino Diaz, Ken Gordon, Balaji Vembu, Sam Webster, David Chisnall, Saurabh Kulkarni, Graham Cunningham, Richard Osborne, and Dan Wilkinson. Confidential computing within an AI accelerator. In *Proceedings of the 2023 USENIX Annual Technical Conference, USENIX ATC 2023, Boston, MA, USA, July 10-12, 2023*, pages 501–518. USENIX Association, 2023.

[78] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. Graviton: Trusted execution environments on gpus. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 681–696. USENIX Association, 2018.

[79] Zhao Wang, Yijin Guan, Guangyu Sun, Dimin Niu, Yuhao Wang, Hongzhong Zheng, and Yinhe Han. Gnn-pim: A processing-in-memory architecture for graph neural networks. In Dezun Dong, Xiaoli Gong, Cunlu Li, Dongsheng Li, and Junjie Wu, editors, *Advanced Computer Architecture*, pages 73–86, Singapore, 2020. Springer Singapore.

[80] Wenjie Xiong, Liu Ke, Dimitrije Jankov, Michael Kounavis, Xiaochen Wang, Eric Northup, Jie Amy Yang, Bilge Acun, Carole-Jean Wu, Ping Tak Peter Tang, G. Edward Suh, Xuan Zhang, and Hsien-Hsin S. Lee. SecNDP: Secure near-data processing with untrusted memory. Cryptology ePrint Archive, Paper 2021/1642, 2021.

[81] Jianping Zhu, Rui Hou, XiaoFeng Wang, Wenhao Wang, Jiangfeng Cao, Boyan Zhao, Zhongpu Wang, Yuhui Zhang, Jiameng Ying, Lixin Zhang, and Dan Meng. Enabling rack-scale confidential computing using heterogeneous trusted execution environment. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1450–1465. IEEE, 2020.

Porting UPMEM Applications to Memclave

Guest API mapping. Table 7 shows one-to-one mapping from common UPMEM Guest calls to their Memclave equivalents. It serves as a quick porting checklist for allocation, kernel loading, launch/wait, and Guest $\leftrightarrow$ MRAM data movement (per-lane transfer/gather and broadcast). Entries cover Guest-side APIs only.

UPMEM Guest API	Memclave Guest API	Porting notes
`dpu_alloc(N)`	`vud_rank_alloc(N)`	Allocate N ranks (64 DPUs/rank on UPMEM).
`dpu_load(set, binary)`	`vud_ime_load(rank, binary)`	Load the same subkernel image on all DPUs of a rank.
`dpu_launch(set)`	`vud_ime_launch(&r)` `vud_ime_wait()`	Same launch/wait semantics.
`dpu_prepare_xfer` `+ dpu_push_xfer(TO_DPU)`	per-rank: `vud_transfer_to(sz, buf, sym)`	Same buffer size & address for each DPU of a rank; Unique data per DPU; word sized units (8 B) only
`dpu_push_xfer(FROM_DPU)`	per-rank: `vud_gather_from(sz, sym, buf)`	MRAM → guest counterpart to `vud_transfer_to`
`dpu_broadcast_to(set,...)`	per-rank: `vud_broadcast_to(sz, buf, sym)`	Broadcast by writing the same 8 B words at the same MRAM offset/size on every DPU.

Table 7: Minimal Guest-side mapping for allocation, loading, launch, and data movement.

UPMEM DPU feature / instruction	Memclave policy	Security rationale	Porting consequence / alternative
Writes to special registers `r20`, `r21`	Privileged (treated as reserved)	Used for system/user-mode assertion.	Small Clang patch to reject/rename defs; no runtime perf impact.
Thread control: `boot`, `resume`, `clr_run`	Privileged	Avoid TOCTOU abuse (tasklet run-state manipulation).	Replace UPMEM barrier with mutex/spin barrier; minor slowdown
Dynamic code loading: `ldmai`	Privileged	Prevents loading arbitrary code that could replace the *loader* at runtime.	Self-modifying code unsupported. Use subkernel chaining (pre-linked stages).
Guest-writable LIM/LSM: `__host` variables	Disallow (pass via MRAM block)	Direct writes into LIM/LSM enable data/code injection into trusted memory.	Replace with EM address; device reads with `mram_read` at start. Slightly higher latency vs direct LSM.
Any direct Guest control over LSM/LIM via CI registers	Privileged	Revoke control into privileged memories.	Stage Guest instructions in EM; subkernel to read them.

Table 8: DPU-side guide: privileged/limited features and secure alternatives needed to preserve Memclave's SR.

DPU-side constraints & mapping. Table 8 shows the DPU-side portability: which UPMEM instructions/features are disallowed under Memclave, why they are restricted, and the alternatives needed to ensure SR.