PIM-ORAM: Towards Oblivious RAM Primitives in Commodity Processing-In-Memory

Byeongsu Woo[1], Kha Dinh Duy[2], Youngkwang Han[3†],
Brent Byunghoon Kang[1*], Hojoon Lee[2*]
[1]KAIST, [2]Sungkyunkwan University, [3]New York University Abu Dhabi
{wbs79, brentkang}@kaist.ac.kr, {khadinh, hojoon.lee}@skku.edu, yh6412@nyu.edu

***Abstract*—Oblivious RAM (ORAM) is theoretically proven to render memory access patterns of a computation completely uniform, mitigating memory side-channel attacks. However, it is accompanied by orders of magnitude slower memory access latency and, thus, is often impractical in many circumstances. On the other hand, Processing-In-Memory (PIM) has been advancing as a solution to accelerate memory-intensive workloads and mitigate the memory wall problem. In this paper, we explore the new direction of in-DRAM oblivious RAM with a design named PIM-ORAM. We retrofit the currently available commodity PIM hardware to provide future direction for secure computation on PIM, and design PIM-ORAM. Our design proposes *split-data* ORAM, a parallelizable in-memory ORAM scheme that takes full advantage of the parallel computing power of the PIM while retaining the original security guarantee of ORAM and dealing with the constraints existing in the commodity PIM. We evaluate PIM-ORAM using the PIM-enabled testbed cloud to provide more realistic numerical values. The evaluation shows that PIM-ORAM alleviates the increase of memory bus usage and ORAM access latency when the ORAM capacity increases.**

***Index Terms*—Processing-in-Memory, ORAM, Oblivious Computing**

1. Introduction

Hardware-based trusted execution technologies [1], [2], [3] have been extensively researched and deployed [4], [5] to preserve the confidentiality of sensitive data. However, the memory side-channel has allowed the adversary to extract the secrets guarded by confidential computation [6], [7], [8], [9]. *Oblivious RAM (ORAM)* [10], [11], [12], [13], [14] has been discussed as a fundamental solution against the memory side-channel attacks based on its ability to render memory access trace completely uniform. However, the theoretically proven obliviousness comes with a cost; adapting ORAM requires the target program to endure several magnitudes of slower memory performance [15], [16], [17], [18], [19]. A reason for this is that ORAM constantly shuffles data by adding a large volume of additional memory accesses to each *effective* data access.

Thus, while ORAM is a promising solution for providing side-channel resilience to cloud workloads, ORAM algorithms are particularly unfit for today's cloud, which is already overloaded with memory-intensive workloads. The cloud today is suffering from the so-called *memory wall* problem [20], [21], [22], where the memory bandwidth becomes a bottleneck in multiple memory-intensive computations carried out by powerful processors and accelerators. When applied to workloads such as large-scale data processing and machine learning, ORAM algorithms exacerbate the memory wall problem by further increasing memory bandwidth demands.

Meanwhile, *Processing-In-Memory (PIM)* has been advancing as a solution to the memory wall problem. PIM provides computing *inside* memory, eliminating unnecessary data movement between the memory and computing devices. Previous works have shown performance advantages of PIM in memory-intensive tasks and system-wide memory bandwidth pressure reduction effects [23], [24], [25], [26], [27]. Moreover, the major DRAM manufacturers released prototypes of PIM-enabled memory devices [28], [29], allowing us to peek into the memory devices in the future cloud. PIM opens up a broad design space for cloud computation models better adapted for large data computation.

In this paper, we explore a new direction in reducing ORAM overheads and alleviating the memory wall problem. We propose **PIM-ORAM**, an in-DRAM oblivious RAM design that drastically reduces system bus traffic caused by ORAM operations, harvesting massively parallel computation power of commodity DRAM PIM memory devices [23], [24], [30], [31], [32].

The design of PIM-ORAM tackle two non-trivial challenges in incorporating ORAM into commodity PIM devices. First, its design must satisfy hardware constraints inherent to all memory devices that follow the current DRAM standards, to be widely adopted in today's cloud infrastructure. Notably, we found that DRAM-based real PIM devices do not support direct communication among the PIM-equipped memory banks; this constraint obstructs an intuitive approach like constructing a single ORAM for an entire memory module. Second, reconciling the security of ORAM and the security model of PIM devices poses

† The author was affiliated with KAIST at the time of the research and is currently affiliated with New York University Abu Dhabi.
** Corresponding authors*

2576-9103/25/$31.00 ©2025 IEEE
DOI 10.1109/ACSAC67867.2025.00083

challenges that PIM-ORAM must address. Specifically, in ORAM's original attack model, the core ORAM components, such as the position map and stash, are stored within the client's private storage, assumed to be trusted. Now, the components must be separated from the client and relocated into PIM memory, as the in-memory processors manage ORAM storages.

PIM-ORAM is meticulously designed to simultaneously satisfy the inherent hardware constraints of current DRAM-based PIM memory and carefully drawn security requirements. Drawing on the parallel architecture of PIM devices, PIM-ORAM implements a distributed *split-data* ORAM scheme. In this scheme, each memory bank maintains an independent ORAM structure that hosts a partial piece of the data block. On each ORAM access, a *Split-Join* process is employed to securely distribute/aggregate partial data between different memory banks. The *Split-Join* process activates all banks simultaneously and makes the execution flow of access operations identical to retain obliviousness in the main memory bus pattern. This way, each PIM-ORAM bank operates independently, overcoming the lack of bank-to-bank communication, while parallelization is also maximized for improved throughput.

We also explore how future PIM architectures might better support secure and oblivious computation without significant changes to existing hardware designs. For instance, current PIM platforms have limited capabilities in efficiently performing cryptographic operations, making secure computation infeasible. To address this, we propose equipping PIM with an AES-enabled DMA engine to offload cryptographic operations. Based on the exploration, we also introduce a lightweight integrity verification mechanism leveraging the built-in AES-GCM authentication tag, avoiding the need for additional expensive hash functions or extra hardware assumptions. We verify the functional correctness of suggested enhancements by using a simulator, while evaluating the PIM-ORAM's performance by using real hardware PIM. We made our simulator and implementation for real hardware available to the public to contribute to future efforts[1].

PIM-ORAM is implemented and evaluated on UPMEM PIM-enabled DRAM modules [30], the only commodity PIM device available at the time of writing. Our implementation overcame the inefficiencies of the provided SDK through several low-level optimizations. We conduct a comprehensive evaluation that encompasses microbenchmarks showing PIM-ORAM's bus traffic reduction effect and access latency improvement, as well as experiments with real-world memory-intensive workloads, such as machine learning and the Redis in-memory database. Our evaluations indicate that PIM-ORAM can reduce the main bus usage by up to 67.13% while achieving up to 3.19× reduced access latency compared to a CPU-only ORAM implementation.

In all, we summarize the contributions of PIM-ORAM as follows:

- We propose first ORAM primitive for massively parallel confidential computation by leveraging the currently available commodity PIM hardware.
- We carefully lay out the hardware constraints and security requirements for the PIM-based ORAM scheme and propose PIM-ORAM's design to resolve them.
- We evaluate the implementation on the real hardware PIM test-bed data center to provide real-world performance evaluation.

[1]https://github.com/Mysigyeong/PIM-ORAM-artifact

2. Background

2.1. Oblivious RAM

Oblivious RAM (ORAM) algorithms make each data access sequence indistinguishable by adding dummy accesses and relocating the stored data. Previous works have introduced variations of ORAM optimized for different scenarios [12], [13], [14], [33]. Among them, we adopted Path-ORAM [11], one of the most adopted ORAM algorithms, for its simplicity and effectiveness.

Path-ORAM. Path-ORAM is a tree-based ORAM utilizing a tree data structure as its data storage, referred to as the *ORAM tree*. A *path* denotes a route from the root node to a leaf node of the ORAM tree. Each node of ORAM tree contains a *bucket*, which consists of a configurable number of data blocks, and each data block has its own identifier, *block ID*. The ORAM tree is usually located in abundant but untrusted storage; thus, its contents must be encrypted. Path-ORAM utilizes two core components: the *stash*, which is a cache storing fetched data blocks from the ORAM tree temporarily, and the *position map*, which stores mapping information between block ID and path. In the rest of the paper, we will call these core components *ORAM Control Components (OCCs)*. The OCCs should be located in a Path-ORAM client's secure storage because they contain critical data: the raw data and metadata of Path-ORAM.

Path-ORAM access. Path-ORAM [11] obfuscates access patterns and hides the access type (*read* vs. *write*) by maintaining the following identical processes regardless of the target data block and the access type. An ORAM access contains the following steps: (1) The position map is referenced to locate the target data block's path, (2) The target data block's path value in the position map is updated with a new random path value, (3) *All* data blocks on this path are retrieved from the ORAM tree and temporarily stored in the stash, (4) The requested data block is retrieved from the stash, and the target data block is updated when the requested operation is *write*, (5) The previously fetched path is then refilled using the blocks from the stash, where blocks are strategically placed from the leaf up to the root.

Recursive-ORAM. Since the size of the position map increases linearly as the capacity of ORAM tree increases, ORAM client's secure memory space could easily become exhausted. A recursive ORAM scheme alleviates the client-side secure memory consumption by exporting the position map as another ORAM tree [34].

Integrity guarantee. ORAM storage integrity can be guaranteed by embedding the Merkle tree within the ORAM

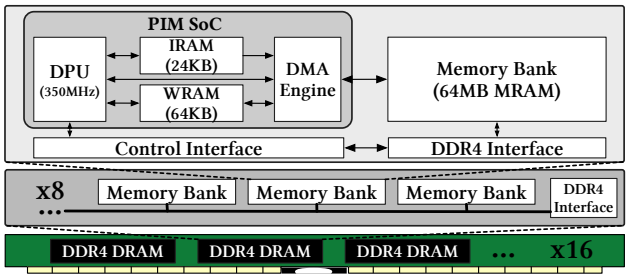


Figure 1: The structure of UPMEM PIM-enabled DIMM.

tree structure [11]. ORAM tree nodes store their child nodes' checksums, computed by hashing the child's data and its stored checksums with SHA-256, and the root node's checksum should be stored in secure storage. During a path access, integrity is verified from root to leaf by comparing the checksum in the parent with the freshly computed value from its child. Since all nodes along a path are updated atomically, this mechanism prevents replay attacks by ensuring that stale nodes cannot be reinserted into the tree.

2.2. Processing-In-Memory

Many research works and industry efforts have proposed new *Processing-In-Memory (PIM)* architectures and applications of PIM [28], [35], [36], [37], [38], [39], [40], [41], [42], [43], [44], [45], and the endeavors are still ongoing.

DRAM-based PIM. PIM-ORAM's design targets PIM-capable memory device that is based on the current DRAM standards. PIM-ORAM's target, UPMEM's PIM datacenter server, represents a generalized DRAM-based PIM that can be constructed for the current cloud infrastructure. The design space of PIMs is quite broad; some works propose more experimental designs that involve more assumptions on the memory device, and, therefore, they are evaluated with simulation. These works often target future memory devices such as HMC [46], [47], [48] or enjoy the freedom of redefining the DRAM structure and host-side memory controller [46], [47], [48], [49]. On the other hand, more practical designs [30], [50], [51] seek to propose designs within the constraints that arise due to the limitations of today's DRAM technology.

PIM-ORAM's design targets real hardware with only a minimal assumption on hardware modifications. In the process, PIM-ORAM's design faces constraints that pose non-trivial challenges. The design elements of PIM-ORAM that satisfy the constraints towards practical hardware-based ORAM design for the current DRAM-based PIM are one of the key contributions of this work.

UPMEM PIM architecture. PIM-ORAM is based on the DDR4 DRAM-based UPMEM hardware, the only currently available commodity PIM to our knowledge. Figure 1 shows the structure of the UPMEM chip in the UPMEM PIM [30]. The PIM SoC contains a processor, *DRAM Processing Unit (DPU)*, that executes code stored in *Instruction RAM (IRAM)* using a scratchpad memory, *Working RAM (WRAM)*. Understandably, the capacity of IRAM and WRAM are limited. The IRAM is a 24KB SRAM, whereas the WRAM is a 64KB SRAM. The PIM SoC also has a DMA engine that offers very low latency access to the memory bank, *Main RAM (MRAM)*. Each DDR4 DRAM module contains a 64MB DRAM as MRAM. The host system communicates with PIM through the control interface and DDR4 interface. DIMMs of UPMEM are organized into ranks, with 64 DPUs per rank and 128 DPUs per 8GB memory DIMM.

2.3. Threat model

We assume a strong adversary model that is conventional in in-cloud confidential computing. Such an adversary is characterized by full control over system software from firmware to the operating system kernel. Another powerful attack vector included in this model is the possibility of physical attacks; the adversary can snoop the main memory bus as shown feasible in a previous work [52], and also physically extract contents of memory devices [53]. However, we do trust the host CPU and the PIM core package.

PIM-ORAM requires host-side software that drives PIM-ORAM and facilitates the movement of sensitive data in and out of PIM-ORAM. We assume that this host-side software is protected with *Trusted Execution Environment (TEE)* within the host (e.g., Intel SGX [1]). However, the scope of this paper focuses on the security model of PIM-ORAM and regards side-channel attacks on TEEs supported in the host out of scope.

Within the PIM-ORAM-equipped DDR4 DRAM memory module, we trust the PIM SoC, including the DPU, the DMA engine, and its on-chip memory (IRAM and WRAM) to be resistant to physical attacks. However, the memory banks (MRAMs) are untrusted in PIM-ORAM's attack model. This is because untrusted software in the host can aptly extract the contents of memory banks through direct access or DMA if the operating system memory access control fails to contain them. The memory banks in PIM-ORAM-equipped DRAM module have conventional hardware composition and are therefore susceptible to the well-known physical attacks on DRAMs (e.g., cold-boot attacks [53]). The difficulty of performing physical attacks on on-chip memory and security models that trust on-chip memory while distrusting off-chip memory has been discussed by many works [54], [55], [56], [57], and PIM-ORAM also embraces a similar model. Additionally, the memory bus between the host and PIM-ORAM DRAM module is also untrusted. Hence, maintaining obliviousness with the presence of a malicious observer who monitors bus traffic is also a security objective of PIM-ORAM.

3. Motivation and challenges

The memory-bound operations involved in ORAM algorithms are quintessentially the type of workload that can be effectively accelerated by PIM, judging by the previous

applications that had seen a significant performance boost with PIMs [23], [24], [25], [26], [27].

Limitations of previous explorations. The previous simulation-based endeavors towards side-channel resilient memory devices [18], [47], [58] assumed freedom of hardware modifications or overly ideal PIM device capacity, where neither is the case for the real-world commodity device [30]. Many works [18], [47] also assumed powerful server-grade processors with multi-level caches as PIM's processing units (e.g., 1.6∼2.5GHz). On the other hand, the UPMEM PIM DPU's capacity is limited, presumably due to the tight area constraints [30]. The DPU runs at a low (350MHz) clock speed without any caches. The working memory (i.e., WRAM) allowed for each PIM DPU is also limited to 64KB. Moreover, many works [18], [47], [58] assumed a *modified* host-side memory controller and also arbitrarily modified internal data paths in the PIM device. However, many such assumptions are not to be expected with a commodity PIM device. The UPMEM PIM memory device must retain its compatibility with the current DRAM standards and the memory controllers of the server processor architectures.

Challenges. Retrofitting and implementing Path-ORAM for commodity PIM devices requires meticulous design decisions. First, PIM-ORAM's design must define and overcome the hardware constraints in the PIM device towards achieving hardware-accelerated ORAM (§3.1). In addition, PIM-ORAM design also establishes a unique security model that ensures secure ORAM operations inside memory (§3.2). This is because, unlike conventional ORAM implementation, OCCs are separated from the client in PIM-ORAM; thereby, PIM-ORAM must securely handle the communication between the client and OCCs.

3.1. Hardware constraints

Devising solutions to the inherent hardware constraints of DRAM-based PIMs is a unique contribution of PIM-ORAM. PIM-ORAM's design must overcome the limitations to construct a practical ORAM in commodity DRAM-based PIM [30]. The limitations are either due to the inherent design of the DRAM architecture (**HC1**) or real hardware capacity that is far from idealistic settings in the simulation-based prototypes (**HC2**).

HC1. Lack of bank-to-bank communication channel. Bank-to-bank communication is inherently absent in today's DRAM interface and protocols. This is a well-known limitation faced by PIM designs for the current DRAM technology [23], [50], [59], [60], [61] that is unfortunately also present in the UPMEM hardware. In research prototypes that are evaluated through simulation, a significant redesign to the DRAM architecture can be considered to fundamentally resolve the issue [50], [51], [59], [60], [61], [62], [63]. However, such deviations from the well-established DRAM architecture standard in real hardware are not likely in the foreseeable future. For this reason, we choose to overcome the limitation with PIM-ORAM design rather than introducing a design assumption that deviates from the real hardware.

HC2. Limited PIM DPU scratchpad memory capacity. PIM has limited scratchpad memory due to the restrictions of the memory hardware. For example, each UPMEM PIM DPU must perform computation using a scratchpad memory (WRAM) of 64KB. As we will explain, the limited size of WRAM introduces a design challenge. This is because the ORAM management data structures that map the ORAM tree itself in MRAM must be maintained within the WRAM according to our threat model. These management data structures grow proportionally to the ORAM tree's storage capacity, and we found that the capacity of WRAM is insufficient to saturate the MRAM with ORAM data.

3.2. Security requirements

PIM-ORAM is designed specifically to address the following security requirements to be feasible with the commodity PIM device. Satisfying **SR1** allows PIM-ORAM to retain the security of ORAM itself amid decomposing ORAM components into the PIM architecture. With **SR2**, the host's interaction with DRAM is rendered oblivious against the powerful adversary assumed in PIM-ORAM.

SR1. Protecting ORAM functionality. In the traditional ORAM use case, a client who uses Path-ORAM backend storage has OCCs in their private, secure storage. This strategy guarantees that the block ID of the requested data block and the plaintext of the data are not exposed to untrusted parts. However, the client and OCCs have to be separated when the ORAM functionality is offloaded to the PIM. This separation makes the client and OCCs communicate via the untrusted memory bus. The PIM-ORAM design, therefore, must locate OCCs to the trusted part of PIM, never allow the contents of the OCCs to be exposed in the untrusted data storage or data-flow path at all times. Furthermore, PIM-ORAM must establish a secure channel between the host and the OCCs to prevent the data from being exposed to the untrusted memory bus.

SR2. Retaining the obliviousness guarantee. PIM-ORAM seeks to inherit the obliviousness guarantees of the original ORAM algorithms. These algorithms normally hide the access pattern and the type of access operation, i.e., read vs. write. However, unlike the original ORAM algorithms, OCCs are separated from the client in PIM-ORAM, and the obliviousness of the additional communication channel between the client and the OCCs must be considered. Thus, PIM-ORAM must render the communication channel uniform to hide the access pattern and the type of access operation. Moreover, the plaintext-ciphertext correlation must be removed from the communication channel to deal with the case of repeatedly reading the same data.

4. PIM confidential compute model

Unfortunately, the UPMEM PIM prototype currently lacks device-side confidential computing support. Nevertheless, the design requirements for trusted accelerators are

already well-defined by the existing works [64], [65], standardization efforts [66], [67], and NVIDIA's implementation with H100 [68].

To explore the potential of PIM as an accelerator in the confidential compute landscape, we add simulated components. With these simulated components, we employ a two-track evaluation strategy. We modified the UPMEM's simulator to allow functional correctness simulation of confidential computing, as these modules are commonplace hardware IPs with well-known performance implications. Then, we use real hardware while accounting for the overhead of confidential computing support through simulated delays. This way, PIM-ORAM closely approximates a possible future implementation of a confidential computing capable PIM device.

Simulated Components. We simulate two components: an attestation module and an AES-capable DMA engine. These components are essential for a PIM device to meet the requirements imposed by the *TEE Device Interface Security Protocol (TDISP)* [66] that is supported by Intel TDX [69] and AMD SEV [2]. We used the hardware extensions adopted in NVIDIA's H100 [68] for confidential compute acceleration as a reference to model the required additional hardware components.

- **Attestation module** The attestation module serves as the foundation for all operations in confidential computing, establishing a secure channel between the host CPU TEE instance and the PIM processor, and loading binary code securely onto the PIM. A private key for attestation should be prepared in the module in advance; thereby, a TEE instance can attest the PIM by verifying the information signed with the module's private key. Then, both parties establish a secure channel by using the Diffie-Hellman key exchange scheme. Moreover, the source of randomness needed for the cryptographic and ORAM process is generated during the attestation process by exchanging a random nonce.
- **AES-capable DMA engine** We introduce an AES-accelerated (AES-GCM-128) DMA engine [70] into our design to overcome the limited computation power of the DPU. Each PIM bank equips the AES-capable DMA engine, which encrypts or decrypts data transferred through the DMA engine. The engine allows the DPU to either encrypt or decrypt for DMA operations. In our design, the DPU encrypts all data transfers from the trusted WRAM to untrusted MRAM and decrypts in the opposite direction.

5. PIM-ORAM Design

This section explains the design of PIM-ORAM that satisfies the design requirements we have explained thus far. PIM-ORAM's components and their operations are illustrated in Figure 2. We first provide an overview of PIM-ORAM's components (§5.1) and explain PIM-ORAM operations (§5.2). PIM-ORAM employs a distributed split-data ORAM scheme (§5.3) and introduces a decoy traffic generation scheme (§5.4) to fully utilize the parallel computation power of PIM while retaining the oblivious guarantee. In addition, PIM-ORAM implements recursive ORAM (§5.5) to overcome the limited capacities and introduces a lightweight integrity verification scheme to protect ORAM trees while avoiding additional hardware assumptions (§5.6).

5.1. PIM-ORAM component overview

PIM-ORAM *Memory Controller.* *PIM-ORAM Memory Controller (POMC)* is a host-side software-level memory controller. It exposes PIM-ORAM API to applications, manages data communication between the host and the PIM, and synchronizes the execution of DPUs. The POMC consists of two components: the in-TEE component protects the symmetric key, the counter for the secure channel, and security-sensitive logic such as preparing messages for communication with the PIM. The non-TEE component is in charge of I/O operations between the CPU and PIM on the data already encrypted inside the TEE.

Command Buffer. The *Command Buffer (CMD Buffer)* is an encrypted MMIO channel for the host to communicate with PIM-ORAM. The CMD Buffer mapping is also a single *fixed-address* communication between the host and ORAM. This means that the attacker learns of no information from the bus addresses in the bus packets exchanged between the host and the PIM-ORAM. The CMD Buffer accepts a *command* and an associated *data* from POMC. The *command* has ORAM control commands such as `INITIALIZE()`, `READ(block ID)`, and `WRITE(block ID, data)`. The commands allow the host to initialize and interact with the ORAM-guarded data structures.

ORAM Trees. PIM-ORAM employs a recursive ORAM to overcome the limited capacity of secure WRAM. There are two recursively-structured ORAM trees: *Level 1 ORAM tree (ORAMTree$_{L1}$)* and *Level 2 ORAM tree (ORAMTree$_{L2}$)*. PIM-ORAM maintains the two trees in the spacious but untrusted, MRAM. ORAMTree$_{\text{L1}}$ stores the encrypted position map entries into the ORAMTree$_{\text{L2}}$, and ORAMTree$_{\text{L2}}$ stores the encrypted host data.

ORAM Control Components (OCCs). PIM-ORAM stores sensitive OCCs in the WRAM such that only the DPU is granted access. In the classical server-client ORAM models, the server with large storage capacity stores the encrypted ORAM trees, and the client or a secure proxy keeps the position map and the stashes that store data blocks retrieved from the server [11], [13]. Similarly, while the DPU maintains the trees in MRAM, the position map and the stashes are to be strictly kept in the secure WRAM. In case of PIM-ORAM's recursive ORAM scheme, (1) *Position map for ORAMTree$_{L1}$ (Posmap$_{L1}$)*, (2) *Stash for ORAMTree$_{L1}$ (Stash$_{L1}$)*, and (3) *Stash for ORAMTree$_{L2}$ (Stash$_{L2}$)* are to be maintained in WRAM. The Posmap$_{\text{L1}}$ is used to retrieve a path from the ORAMTree$_{\text{L1}}$ into the Stash$_{\text{L1}}$, where the path value in the ORAMTree$_{\text{L2}}$ can be collected. After having retrieved the data blocks in the path from ORAMTree$_{\text{L2}}$ into Stash$_{\text{L2}}$, the data can finally be obtained. PIM-ORAM's cryptographic secrets are also placed

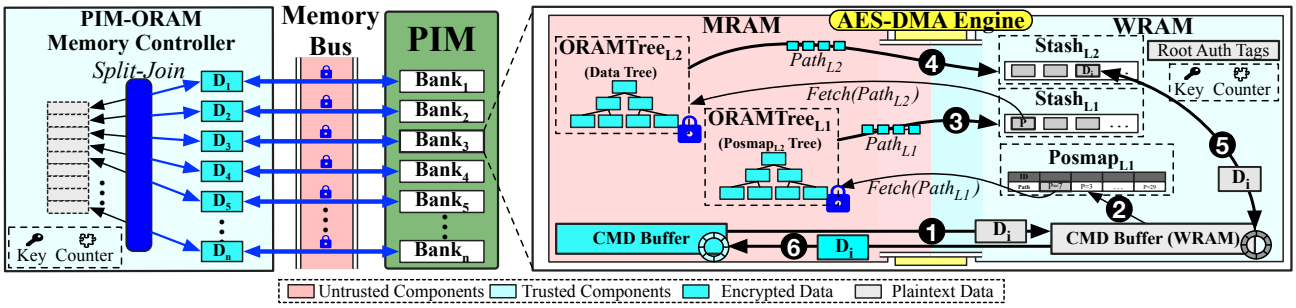


Figure 2: The ORAM access process of PIM-ORAM's *split-data* distributed ORAM, recursively structured in each bank. Each PIM bank manages its independent ORAM storage in parallel.

inside WRAM, which include the secure channel encryption key, the encryption counter, and the authentication tag of the root ORAM node for integrity checking (§5.6).

5.2. PIM-ORAM operations

Initialization. POMC is initialized in the TEE of the host. The host allocates a set of DPUs and establishes secure channels between the POMC and the allocated DPUs by communicating with the attestation modules of the DPU banks. After secure channels are established between POMC and the DPUs, the ORAM program is loaded on the DPUs securely. The host sends the initialization command to the DPUs, and each DPU initializes its OCCs, and ORAM trees in parallel. After initialization, the host can command the DPUs to store or load data in ORAMTree_{L2}.

ORAM access: Pre-in-memory operations. The PIM-ORAM access operation begins as the host-side POMC issues a command to PIM-ORAM. This request includes a command (either `READ` or `WRITE`), a block ID, and associated data, e.g., `pim_cmd(WRITE,blk_id,data)`. Dummy data is used when the command is `READ`. POMC splits the data (i.e., $D \rightarrow D_1, D_2 \ldots D_n$) and encapsulates them into commands to be delivered to each bank through the secure CMD Buffer channel. Then, the banks process the requested operation *concurrently*.

ORAM access: In-memory operations. The encrypted request is sent from POMC to the CMD Buffer of each PIM bank. ❶ The received encrypted content in the CMD Buffer is copied into *Command Buffer in WRAM (CMD* $\textit{Buffer}_{WRAM}$*)* through the DMA engine. The AES-capable DMA engine decrypts the content of the CMD Buffer during the DMA data transfer. As PIM-ORAM employs a recursive ORAM scheme, the path value of the given block ID ($Path_{L2}$) should be retrieved from the ORAMTree_{L1} first. ❷ To retrieve $Path_{L2}$, the block ID of the ORAMTree_{L1} (ID_{L1}) is obtained from the given block ID, and the associated path value ($Path_{L1}$) is retrieved from the Posmap_{L1}. ❸ The data blocks in $Path_{L1}$ are fetched from the ORAMTree_{L1} and stored in the Stash_{L1}. Likewise, ❹ after $Path_{L2}$ is retrieved from the Stash_{L1}, the data blocks in $Path_{L2}$ are fetched from the ORAMTree_{L2} and stored in the Stash_{L2}. ❺ The target data block is retrieved from the Stash_{L2}. If the command is `READ`, the target data block is copied into CMD Buffer_{WRAM} to be sent to the host. Otherwise, if the command is `WRITE`, the target data block is updated with the given split data D_i, and dummy data is written to CMD Buffer_{WRAM}. Lastly, ❻ CMD Buffer_{WRAM} is encrypted and copied to the CMD Buffer in MRAM. After the in-memory operations are finished, the result in the CMD Buffer is transferred to the host.

During the ORAM tree accesses, $Path_{L1}$ and $Path_{L2}$ are updated with new random path values before flushing the stashes. The accessed data blocks are relocated to one of the new nodes in the new paths at every ORAM access; thus, the access pattern of ORAM trees is fully obfuscated. Moreover, as the AES-capable DMA engine encrypts and decrypts all data moving between MRAM and WRAM, re-encryption for the same data is conducted, and the correlation between ciphertext and plaintext is removed.

ORAM access: Post-in-memory operations. POMC retrieves the results in parallel from the CMD Buffer and joins the received responses into a single coherent data. If the request is `READ`, POMC transfers the joined data to the user; otherwise, the data is discarded.

5.3. Distributed split-data ORAM scheme

PIM-ORAM distributes ORAM trees to banks to remove direct communication among banks. However, a naive distribution scheme can break the obliviousness guarantee. Imagine a naive distributed ORAM design where data is stored across 16 PIM banks. Upon receiving a request for data, only one bank that stores the associated data is activated. This per-bank ORAM does not uphold the obliviousness guarantee of the original Path-ORAM [11] because it induces the correlation between the data and the bank. The adversary observing the changes in the ORAM tree can detect updates in exactly one of the 16 banks.

PIM-ORAM tackles this challenge with the distributed *split-data* ORAM scheme specifically devised for the DRAM-based PIM device. The scheme retains the distributed ORAM trees among the banks from the aforementioned naive design; each bank independently maintains an ORAM tree for the data it stores. However, PIM-ORAM's scheme splits the data itself into multiple subblocks (i.e., D

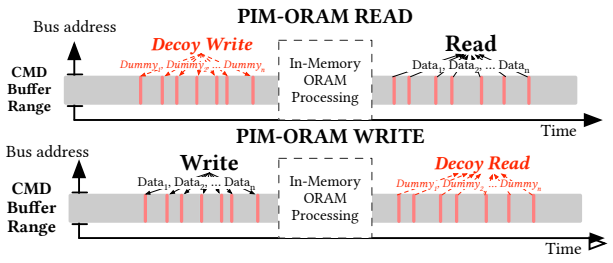


Figure 3: *Split-Join* process generates decoy traffic to render PIM-ORAM operations (`READ` and `WRITE`) indistinguishable from the attacker.

= D_1, D_2 ... D_n), then distributes the blocks to n banks. Hence, for each PIM-ORAM access operation, all banks involved are activated simultaneously, and data fetching and ORAM tree shuffling process are carried out in parallel. As a result, the *Split-data* ORAM scheme activates all banks simultaneously to retain obliviousness in the main memory bus pattern. This way, each bank operates independently, overcoming the lack of bank-to-bank communication, while parallelization is also maximized for improved throughput.

5.4. Decoy traffic for read and write obliviousness

PIM-ORAM applies a decoy traffic generation scheme to ensure read/write pattern obliviousness as shown in Figure 3. *Split-Join* process considers not only the application of *split-data* ORAM scheme but also the retention of obliviousness; *Split-Join* process must render `READ` and `WRITE` indistinguishable. If obliviousness is not considered, the command execution pattern will differ as PIM-ORAM processes the different commands. For example, the data content is only transferred from the host to the PIM before the ORAM process in PIM when `WRITE` is performed. The data content is only transferred from the PIM to the host after the ORAM process in PIM when `READ` is performed.

To avoid the above problem, a decoy traffic generation is used to make the execution flow of POMC and the data transfer pattern of the memory bus consistent. To generate the decoy traffic, we made the execution flow of *Split-Join* process consistent regardless of the requested command. If the requested command is `READ`, dummy data is split and sent to the PIM. Likewise, dummy data is retrieved from the PIM and joined, if the requested command is `WRITE`.

5.5. Recursive ORAM scheme for PIM

Another design challenge arises as we implement the aforementioned distributed ORAM scheme for each DPU is the limited WRAM capacity. Considering the area constraint in SoC embedding in DRAMs, we expect that this limitation will ensue in the near future. We employ a 2-level recursive ORAM scheme specifically designed for PIM-ORAM to handle the limited capacity. A recursive ORAM scheme reduces the required memory footprint for the position map while enduring the additional ORAM tree access as a trade-off. Our design choice to adopt a recursive ORAM proved to be a profitable one. Our target PIM has highly limited WRAM (64KB) that is considered secure and only visible to PIM DPU in our model, and memory access latency *inside* a PIM bank is very low.

Without the recursive scheme, a PIM bank's WRAM can only support a position map with a capacity of around 10,000 ORAM data blocks. For most practical data block sizes, the position map capacity is insufficient for fully saturating the 64MB MRAM with ORAM data blocks. With the recursive scheme, the size of the position map is significantly reduced, analogous to how the modern OS's multi-level page tables maintain their relatively small footprint. PIM-ORAM's in-WRAM position map ($\text{Posmap}_{\text{L1}}$) can support up to 100,000 data blocks with one additional ORAM access to the newly populated ORAM tree that stores the position map ($\text{ORAMTree}_{\text{L1}}$).

5.6. Lightweight integrity verification

The data inside the ORAM trees must be protected from malicious writes. However, if an integrity scheme similar to Path-ORAM (§2.1) were to be employed, it would either incur significant overheads through software-based SHA-256 hashing or require additional hardware acceleration.

We propose a lightweight integrity verification mechanism that avoids these problems by utilizing the authentication features that are built into AES-GCM cipher. Instead of additionally introducing SHA256 functionality and storing hash values in the tree, each node stores its child nodes' AES-GCM's authentication tag. On each ORAM access, the integrity of the tree nodes is verified at the same time these data blocks are being fetched through the AES-capable DMA engine. To achieve this, we program the authentication tag obtained from the parent node into the DMA engine, before engaging in the DMA transfer. During the stash eviction, we first leverage the engine to write the encrypted ORAM node into the DRAM. Then, the authentication tag of the encrypted block is stored within its parent node. The authentication tag of the root node is securely stored within WRAM, protecting the integrity of the entire ORAM tree.

6. Implementation

PIM-ORAM's implementation consists of the PIM-side binary (i.e., PIM kernel) and the host-side POMC, which are implemented in C/C++ using a combination of the UPMEM SDK [71] and the Intel SGX SDK for Linux [72].

PIM-ORAM *memory controller.* The POMC comprises an untrusted component that interfaces with the PIM device through the UPMEM SDK and a trusted library linked with the enclave application. The trusted POMC library provides three API interfaces to the enclave application: `oram_init()`, `oram_read(block_id, data)`, and `oram_write(block_id, data)`.

oram_init issues INITIALIZE command to initialize PIM-side components with predefined configurations. Particularly, the configuration is fixed with 2-level recursive ORAM and 100,000 128-Byte data blocks per DPU, defined based on UPMEM's memory bank capacity.

oram_read and oram_write issue ORAM accesses through the commands READ and WRITE, as described in §5.2. The trusted POMC library performs the oblivious *Split-Join* process by sending distributed requests to the DPUs and gathering the data from DPUs to a single buffer before returning to the user.

PIM kernel. The PIM kernel, the program to be executed inside PIM, is built using UPMEM's custom clang compiler and contains the functional components illustrated in Figure 2. POMC maps the CMD Buffer from PIM's memory to the user application's memory to interact with the CMD Buffer through MMIO. On execution, the PIM kernel handles requests by reading commands from CMD Buffer and invoking the corresponding handler.

Cryptographic process. For in-enclave data encryption using AES-GCM, we used the cryptographic library for Intel SGX [73]. The cryptographic process in PIM is performed by the AES-capable DMA engine in the PIM confidential compute model of PIM-ORAM. We simulated the overhead introduced by the AES-capable DMA engine because current PIM hardware does not provide such functionality. The simulation method is described in §8.1.

6.1. Optimizations

We now explain the optimization applied to the UPMEM SDK, to the PIM-side ORAM operations, and the host-side *Split-Join* process to reduce the latencies of PIM-ORAM operations.

Burst traffic mode. We implemented the so-called *burst traffic mode* for UPMEM SDK's MRAM management structure. The current structure is optimized to yield the maximum throughput for one-time large data transfers. The UPMEM SDK manages per-rank worker threads to manage the communication between the host and PIM rank. Each worker thread also spawns up to four MRAM threads to efficiently handle large amounts of data transfer. Our burst mode simplifies this layered thread structure to prioritize latency over maximum throughput; the MRAM threads are eliminated, and the worker thread now performs MRAM data transfers directly. The simplified thread-switching structure brought a 30.49% decrease in the communication latency between the host and PIM and allowed us to substantially reduce the access latency of PIM-ORAM in real-world applications.

Parallelizing PIM-side ORAM operations. We further optimized the PIM-side ORAM operations by parallelizing independent ORAM operations and utilizing DPU's hardware threads [71]. For instance, the eviction operation conducted on $\text{ORAMTree}_{\text{L1}}$ (after Path_{L1} retrieval) is independent from the subsequent Path_{L2} retrieval from $\text{ORAMTree}_{\text{L2}}$. Hence, we parallelized such independent operations to run on separate hardware threads to yield a PIM-side ORAM operation improvement of 30.76%.

Parallelizing Split-Join process. The *Split-Join* process in POMC can also be parallelized because there is no dependence among the data pieces. We introduced threading to reduce the cryptographic overhead by parallelizing each independent cryptographic process. The number of threads in POMC is determined empirically, and the specific numbers are described in §8.

7. Security analysis

PIM-ORAM must satisfy the following requirements: it must guarantee the confidentiality and integrity of data and must not reveal the access pattern.

7.1. Confidentiality

Let's define that D is data transferred from or to the host, and M is metadata for management of ORAM algorithm. The content of CMD Buffer belongs to D. The metadata in the ORAM tree and the content of OCCs belong to M.

Claim 1. No plaintext of D is revealed to the untrusted part.

We already assumed that the host leverages its TEE to preserve the confidentiality of D on the host side. When D gets through the memory bus, the content of D has already been encrypted in POMC. The encrypted D is only decrypted when transferred to the WRAM in the PIM.

Claim 2. No plaintext of M is revealed to the untrusted part.

The OCCs reside only in the trusted WRAM. The ORAM tree blocks are always encrypted except that they are fetched to the stashes in WRAM.

7.2. Obliviousness and integrity

Let A_B be an adversary who snoops the memory bus between the host and PIM, and A_M is an adversary who monitors MRAM. A_M can monitor the components in MRAM: $\text{ORAMTree}_{\text{L1}}$, $\text{ORAMTree}_{\text{L2}}$, and CMD Buffer.

Claim 3. A_B cannot learn of the access pattern by monitoring traffic on the memory bus.

A_B can only observe the consistent access pattern because all DPUs have the same memory layout, and their CMD Buffer's location is fixed. A_B also cannot distinguish WRITE and READ operations because of the decoy traffic generation scheme. In addition, A_B cannot leverage the correlation of the plaintext and the ciphertext because re-encryption is conducted in every data transfer.

Claim 4. A_M cannot learn of the access pattern by monitoring the CMD Buffer.

A_M cannot trace the content of the CMD Buffer because the DMA engine re-encrypts the data before inserting it into the $\text{ORAMTree}_{\text{L2}}$. Likewise, retrieved data from the $\text{ORAMTree}_{\text{L2}}$ is also re-encrypted before being copied to the CMD Buffer. A_M also cannot distinguish WRITE and

READ because CMD Buffer is constantly read and written regardless of the type of operation (❶ and ❻ in Figure 2).

Claim 5. A_M cannot learn of the access pattern by monitoring the ORAM trees.

A_M cannot extract information as already discussed in §7.1. Moreover, as mentioned in Claim 4, A_M cannot find the content of the CMD Buffer in the $\text{ORAMTree}_{\text{L2}}$ because all data are re-encrypted by the AES-capable DMA engine.

Claim 6. A_B cannot break the integrity of the CPU-PIM communication channel.

The secure communication channel between the host and the PIM is managed with AES-GCM to prevent the data on the memory bus from being corrupted or replayed.

Claim 7. A_M cannot break the integrity of the ORAM trees.

As already discussed in §5.6, all nodes' authentication tags are encrypted in their parent nodes, and the root node's authentication tag is stored in WRAM. Thereby, A_M cannot modify nodes and their authentication tags.

8. Evaluation

We evaluate PIM-ORAM's efficacy and practicality through microbenchmarks and real-world workload experiments. PIM-ORAM is compared with *CPU-ORAM*, a conventional Path-ORAM implementation that only utilizes the host processor's computing power.

8.1. Methodology

We evaluate PIM-ORAM implementation in the UPMEM test-bed data center that offers server-grade machines with UPMEM PIM-equipped DRAM modules. The server features 2× Intel Xeon Silver 4110 CPU (8 cores) @ 2.10GHz and 128GB of DRAM. The DRAM modules provide 40 UPMEM ranks with 2560 DPUs, whose clock speed is 350MHz. According to PIM-ORAM's threat model, PIM-ORAM's host-side component POMC is placed inside an SGX enclave in the host. In the case of CPU-ORAM, the entire ORAM implementation is inside an enclave to handle the same threat model. We functionally verified the additional hardware modules by implementing PIM-ORAM simulator, as explained in §4.

Shared Setting for CPU-ORAM & PIM-ORAM									
Data SZ (GB)	0.1	0.2	0.4	0.8	1.5	3.1	6.1	12.2	24.4
Block SZ (KB)	1	2	4	8	16	32	64	128	256
PIM-ORAM-Specific Parameters									
# DPUs Involved	8	16	32	64	128	256	512	1024	2048
# POMC Threads	1	1	2	2	4	4	8	8	8

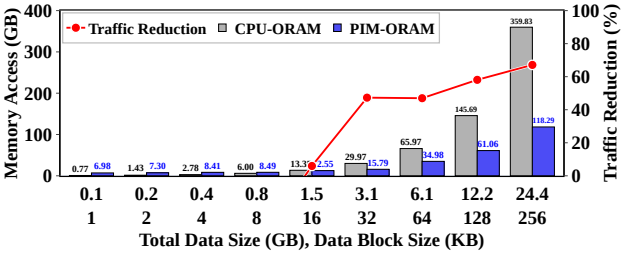
TABLE 1: Experiment parameters of CPU-ORAM and PIM-ORAM. The logical data block size of PIM-ORAM matches the data block size of CPU-ORAM.

Evaluation parameters. Table 1 shows experiment parameters for PIM-ORAM and CPU-ORAM that we used throughout the experiments. The total data size hosted by ORAM varies in 9 steps (0.1GB to 24.4GB) for both PIM-ORAM and CPU-ORAM. Note that both PIM-ORAM and CPU-ORAM use 2-level recursive ORAM configured with 100,000 blocks and a bucket capacity of 4. Each DPU manages its independent $\text{ORAMTree}_{\text{L2}}$ with a 128Bytes data block size, and the total hosted data size is scaled by increasing the number of involved banks and hence the DPU number. For example, PIM-ORAM has logically corresponding tree topology with the ORAM tree with 1KB block size, when PIM-ORAM allocates 8 DPUs (8 independent ORAM trees with 128Bytes block size). Thus, to match the total data size and logical tree topology of PIM-ORAM, CPU-ORAM adjusts the total data size by adjusting its data block size accordingly.

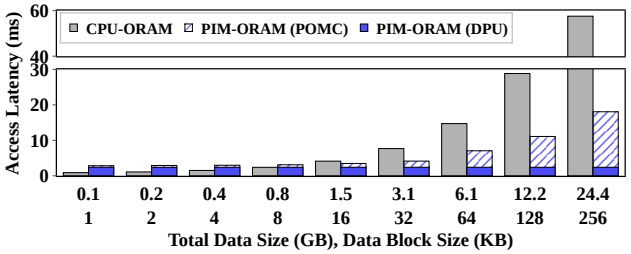
Overhead of AES-capable DMA. To reflect the additional runtime overhead induced by our PIM confidential compute model on the real PIM hardware, we simulated the overhead of the AES-capable DMA engine. We inserted simulated overhead for cryptographic operations between MRAM and WRAM for each DMA operation. DMA engines with AES accelerators are, in fact, common and simplistic hardware IPs whose overhead from AES operations is predictable and directly proportional to the total amount of data transferred. We estimated the cryptographic delay by studying previous works that also adapted simulated cryptographic acceleration [47], [74], [75], [76]. We adopted a delay of 22.35% from the previous work that calculated the overhead of an AES-capable DMA engine in a cycle-accurate simulation [76].

CPU-ORAM. We implemented CPU-ORAM, a conventional Path-ORAM implementation that runs inside an SGX enclave and runs entirely on the host processor. CPU-ORAM shares the same ORAM implementation with PIM-ORAM for the fair comparison. Although ZeroTrace stands out as a representative ORAM implementation for SGX, its implementation [77] resulted in crashes when given more than 3.1GB of data. Hence, we concluded that the implementation is inapt for our evaluation that must show large data computation capabilities. We provide the incomplete microbenchmark results from successful iterations of ZeroTrace, which shows 19.51× slower performance than PIM-ORAM in §A. Thus, we used CPU-ORAM, which can effectively handle the given evaluation parameters, to facilitate more comprehensive evaluations.

Intel SGX simulation. We opted for SGX simulation mode due to the lack of TEE support in the UPMEM data center. Note, it gives an advantage to CPU-ORAM because the simulation mode performs better than the hardware mode; the SGX simulation mode is mainly intended for correctness testing of SGX applications and does not perform memory encryption [78]. Nevertheless, as our experiments will illustrate, PIM-ORAM still exhibits performance advantages. We additionally provide the performance comparison result between the simulation mode and the hardware mode on the local SGX-enabled machine in §B.



(a) Accumulated system bus traffic usage. *Traffic reduction* illustrates the reduction ratio of PIM-ORAM's system bus traffic usage compared to the CPU-ORAM.



(b) Average access latency. *DPU* and *POMC* illustrate the access latency breakdown of host-side (CPU) POMC execution vs. PIM-side DPU execution.

Figure 4: The microbenchmarks performing 10,000 ORAM accesses for each ORAM storage capacity configuration.

8.2. System bus usage and access latency

We conducted a set of microbenchmarks to show the bus traffic usage and access latency of PIM-ORAM and CPU-ORAM. The microbenchmark measures (1) the pressure on the main bus (i.e., observed memory throughput) during 10,000 ORAM accesses and (2) the access latency of the two implementations during 10,000 ORAM accesses.

Main memory bus pressure comparison. The pressure on the system bus during the evaluation for PIM-ORAM and CPU-ORAM is shown in Figure 4a. For accurate measurement of the system main bus usage, we opted for Intel's high precision *Performance Counter Monitor (PCM)* [79] to measure the total accumulated main memory usage during the microbenchmarks.

The total accumulated memory bandwidth usage ranged from 0.77GB to 359.83GB in the case of CPU-ORAM. The increase in data block size for CPU-ORAM has a drastic effect on bandwidth usage. The result is rather straightforward, considering that the known estimated bandwidth consumption for Path-ORAM is $\approx 2 \times \log_2 \frac{N_{Block}}{\text{BucketSize}} \times$ DataBlockSize $\times$ BucketSize, where N_{Block} denotes the total number of blocks in the tree (set to 100,000 in the experiments).

On the other hand, PIM-ORAM's bandwidth scales much more efficiently as the data block size increases. PIM-ORAM's bus usage is higher than that of CPU-ORAM when the total data is less than 1.5 GB (block size = 16KB). However, as the total data size increases PIM-ORAM's bus usage experiences only small increments, and the advantage over CPU-ORAM is shown by a large margin towards the 24.4GB total data size. This is because PIM-ORAM's bandwidth use is composed of CPU-PIM communication for command issuance, the actual transfer of the data to be written, and decoy traffic pattern generation (§5.4). These bus transactions form the *lower-bound* of PIM-ORAM's latency, which is higher than that of the CPU in very small block sizes. The lower-bound, however, is amortized as the data block size increases, and the increments of bus usage for PIM-ORAM almost entirely come from the need to perform a one-time transfer of larger data blocks.

At a closer look, 60 data blocks of ORAMTree_{L2} and 40 data blocks of ORAMTree_{L1} are transferred through the memory bus in one ORAM access of CPU-ORAM, whereas only one data block per ORAM access is transferred in PIM-ORAM. As a result, PIM-ORAM reduced the number of data blocks transferred through the system memory bus and reduced system bus traffic usage up to *67.13%*.

Access latency comparison. Figure 4b shows the average time of 10,000 accesses in the ORAM programs as the data block size is varied. We profiled the composition of PIM-ORAM's access through `perfcounter` functions inserted into the UPMEM SDK to measure time consumed in the host-side execution and in the DPU (subgraphs **PIM-ORAM (POMC)** and **PIM-ORAM (DPU)** in the figure).

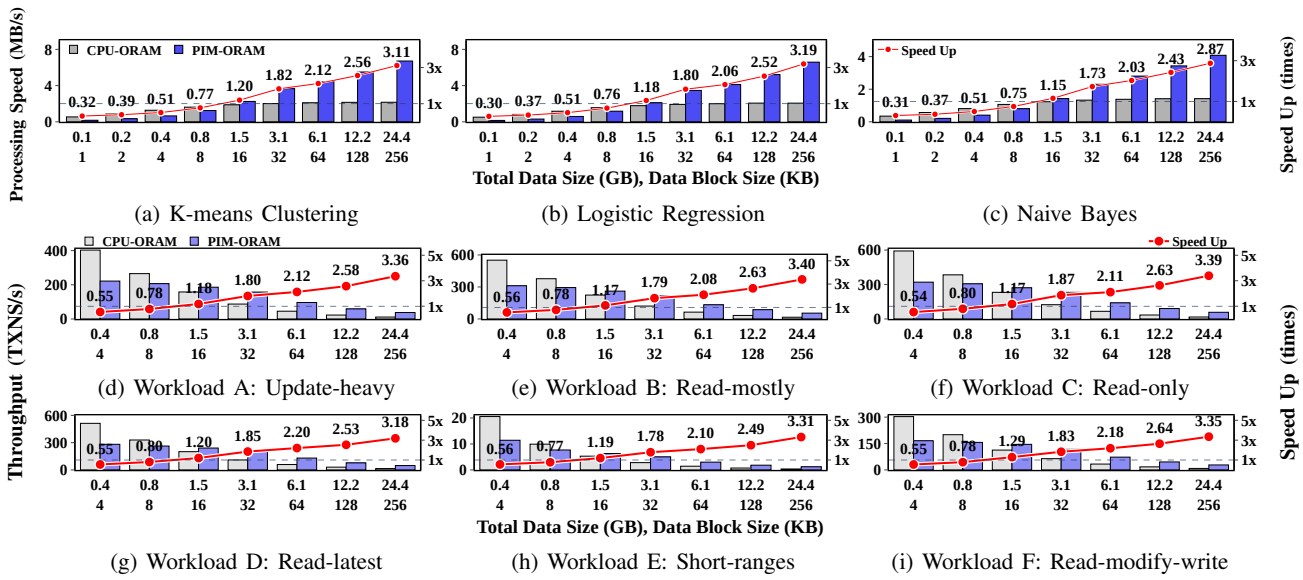
The average ORAM latency in PIM-ORAM ranges from 2.84ms (block size = 1KB) to 18.06ms (block size = 256KB), whereas the latency of CPU-ORAM ranges from 0.88ms to 57.55ms. PIM-ORAM outperforms CPU-ORAM at a block size of 16KB, and the performance gap widens to 3.19× at the block size 256KB.

The composition of PIM-ORAM latency shown through subgraphs allows us to understand the characteristics of PIM-ORAM's access latency composition. The POMC execution time gradually increases, likely due to the increased load on the *Split-Join* process as the data block size is increased. On the other hand, the execution time of DPU remains more or less the same (2.41ms) for all data block size configurations. This is because each DPU manages its own ORAM tree in parallel in PIM-ORAM's design, and the size of ORAM trees in the DPUs are consistent regardless of the number of the allocated DPUs and the PIM banks.

8.3. Real-world workloads

We evaluated PIM-ORAM's efficacy and practicality in memory-intensive real-world applications, which include machine learning algorithms (K-means clustering, Logistic Regression, and Naive Bayes), and the Redis [80] in-memory database. We compiled two versions of the applications, one with PIM-ORAM and one with CPU-ORAM, for comparison.

Benchmark method: Machine learning. We implement in-house benchmark tools for benchmarking the machine learning workloads with ORAM-backed data storage. We



(a) K-means Clustering (b) Logistic Regression (c) Naive Bayes

(d) Workload A: Update-heavy (e) Workload B: Read-mostly (f) Workload C: Read-only

(g) Workload D: Read-latest (h) Workload E: Short-ranges (i) Workload F: Read-modify-write

Figure 5: PIM-ORAM vs. CPU-ORAM performance in machine learning workloads and YCSB benchmark's workload set.

manually replace storage-accessing code lines with ORAM access APIs, and measure the data processing speed during training. We match the training data set according to the capacity of ORAM to keep the proportion of ORAM usage constant. For instance, the training set size is 1MB when the ORAM capacity is 0.1GB and 2MB when the ORAM capacity is 0.2GB.

Benchmark method: Redis. We identified the Redis server database structure that stores the client's data and the low-level functions in Redis that manage the database contents, such as the database managing functions in `dict.c`, and Redis' internal data structure managing functions in `t_hash.c`. Then, we ported the identified functions to use the ORAM interface, allowing the hash table to be allocated in the ORAM trees and accessed exclusively through the ORAM access functions. ORAM-ported Redis uses the key's hash value as block ID, and the associated data block contains the internal data structures that define the corresponding entry and manage hash collisions.

We benchmarked Redis with the *Yahoo! Cloud Serving Benchmark (YCSB)* [81] framework, which is a widely used benchmark for mimicking various workloads imposed on the cloud key-value stores. We executed the workloads with varied numbers of DPU to observe the change in throughput as the capacity of the ORAM storage changes and to check whether the performance relationship between CPU-ORAM and PIM-ORAM is retained in all workloads.

Result discussion. Figure 5 illustrates the data processing speed comparison of the ORAM-applied machine learning workloads and the throughput comparison of the ORAM-backed Redis in varied ORAM capacities. The benchmark results closely match the tendencies from the microbenchmarks shown in §8.2, and the performance relationship between PIM-ORAM and CPU-ORAM is reenacted in all workloads. Even though PIM-ORAM initially underperforms compared to CPU-ORAM when the ORAM capacity is smaller than 1.5GB, PIM-ORAM shortly begins to outperform, starting at 1.5GB, and the gap grows larger as the capacity is increased. PIM-ORAM's performance advantage widens up to 3.19× difference over CPU-ORAM in the machine learning workloads, and 3.40× difference over CPU-ORAM in the YCSB workloads.

9. Discussion

This section provides the specific discussions relevant to PIM-ORAM's design.

9.1. Comparison with TEE-only ORAM

Performance and scalability. Our experiment results indicate the scalability characteristics of PIM-ORAM: PIM-ORAM provides substantial acceleration in large data computation, while the increased I/O overhead from CPU-PIM communication becomes a bottleneck with smaller data sizes. However, this property also applies to accelerators in general to a certain degree. Nevertheless, one contribution of our experiments is to clearly show the range of data size where PIM-ORAM becomes effective (Figure 4b). Another aspect is memory bandwidth reduction: PIM-ORAM provides a significant main bus bandwidth reduction in many processed data size intervals (Figure 4a). This characteristic would prove to be beneficial in saturating the compute capability of the cloud systems, alleviating the *memory wall* problem when many workloads are running simultaneously.

Required porting effort. PIM-ORAM does not require more porting effort than CPU-side TEE-only solutions. Porting a program to utilize ORAM to conceal its sensitive mem-

ory access trace involves replacing the memory accesses with ORAM access functions (e.g., `oram_read()`). PIM-ORAM's host-side software memory controller implementation exposes a set of ORAM access functions that are equivalent to those of the CPU-only ORAMs, and the PIM acceleration is transparent to the programmer.

Required hardware changes. As we already discussed in §4, PIM-ORAM requires the PIM hardware to receive a set of hardware extensions, similar to those introduced with NVIDIA's GPU confidential compute, to support acceleration of the confidential computing [68]. However, we leave the hardware prototype implementation as future work.

9.2. Interoperability with confidential computing

With the aforementioned hardware changes based on the ongoing standardization efforts on confidential computing, we expect that PIM-ORAM would interoperate with the confidential computing architectures such as Intel TDX [69], AMD SEV-SNP [82]. Notably, the TDISP standard [66] lays out the requirements and procedures by which the CPU TEE and accelerator establish trust. One difference between PIM-ORAM and the confidential compute accelerators modeled in TDISP [66] is that such accelerators are assumed to be of PCIe device class. In contrast, PIM-ORAM is a memory device. However, the type of the physical link itself (i.e., PCIe vs. main bus) is largely irrelevant to the protocol itself. Nevertheless, we regard a concrete hardware implementation that can further validate the feasibility of our design as future work.

9.3. WRAM capacity and recursive ORAM

PIM-ORAM adopts a two-level recursive ORAM to maximize the bank memory (MRAM) utilization by reducing the WRAM usage. This approach introduces a trade-off; recursively structured ORAMs, while substantially reducing the OCCs footprint in WRAM, introduce additional computational overhead. We found the two-level recursion to provide an ideal trade-off that yields a feasible solution by balancing the UPMEM hardware's subpar DPU performance and its current WRAM size. Therefore, we expect that an adaptation of PIM-ORAM design to future PIM hardware can use our implementation with the UPMEM hardware as a reference to find the right balance. For example, a hardware with much faster DPU and even smaller WRAM may consider adopting an N-level recursive ORAM.

10. Related Work

Hardware-based memory side-channel mitigation. Previous works have explored memory address side-channel mitigation with smart memory. Invisimem [47], and Obfusmem [58] suggested a cooperation model between the host CPU memory controller and 3D-stacked memory to randomize the accessed addresses observable on the system bus. SecureDIMM [18] proposed ORAM-based side-channel eliminated memory based on its own design that replaces the buffer of LRDIMM with ORAM-enabled ASIC buffer and introduces new custom DDR commands to handle the ORAM operations on the memory. The previous works used a simulated system to evaluate the design because they used new memory structures or custom hardware that were not realized in the real world. Furthermore, the CPU memory controller should be changed as they bring new memory commands or new memory control interfaces.

The PIM-ORAM design set its starting point as the commodity real PIM device to discuss and evaluate PIM-accelerated ORAM primitives at a practical level. Our PIM-based ORAM scheme is designed for the currently available commodity PIM with only the necessary set of hardware change assumptions, thereby proposing the most realistic design yet. Our evaluations are conducted with real PIM devices to report design feasibility, while the existing works have resorted to simulation with ideal settings.

Side-channel resistant external storage. Other previous works tried to introduce side-channel resistant secure storage to the system in accelerator form by using FPGA. TrustOre [65] made side-channel resistant storage for SGX on an FPGA chip and was connected to the host via PCIe. Phantom [19] made the FPGA-based Path-ORAM accelerator implemented on the specific FPGA computing platform and suggested a usage model as an IaaS form.

PIM-ORAM's PIM-accelerated ORAM does not require a dedicated accelerator and can be readily adapted to current and future memory devices. For this reason, we believe that PIM-ORAM is a necessary and practical design exploration for oblivious computing in the cloud.

11. Conclusion

We presented the design and implementation of PIM-ORAM that proposes an in-memory ORAM primitives for commodity PIM hardware. PIM-ORAM's design retrofits ORAM to run inside memory, where low-latency and parallelized memory access accelerate the very high overhead of ORAM. PIM-ORAM's recursive ORAM scheme and the *Split-Join* process were the key design components that overcome the inherent limitations of the current DRAM-based PIM. In all, PIM-ORAM design makes a case for PIM-accelerated ORAM as a new primitive for oblivious computation in the cloud.

12. Acknowledgements

This work was supported by grants from the National Research Foundation of Korea (NRF) (RS-2022-NR072053, RS-2025-00520023), the Institute of Information & Communications Technology Planning & Evaluation (IITP) funded by the Ministry of Science and ICT (MSIT) (RS-2022-II221199, RS-2022-II220688, RS-2024-00437306, RS-2024-00439819, RS-2024-00437849, RS-2024-00425503, RS-2024-00440780), and the Korea Internet & Security Agency (KISA) (1781000009). This work was also carried out as part of a collaborative research with NYU (G01240629).

References

[1] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution," ser. HASP '13. New York, NY, USA: Association for Computing Machinery, 2013.

[2] A. M. Devices, "Amd secure encrypted virtualization (sev)," https://developer.amd.com/sev/, 2022, last accessed Oct 15 , 2022,.

[3] T. Alves, "Trustzone : Integrated hardware and software security," 2004.

[4] I. Amazon Web Services, "Aws nitro enclaves," https://aws.amazon.com/ec2/nitro/nitro-enclaves/, 2021.

[5] M. Azure, "Azure confidential computing," https://azure.microsoft.com/en-us/solutions/confidential-compute/, 2021.

[6] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software grand exposure: SGX cache attacks are practical," in *11th USENIX Workshop on Offensive Technologies (WOOT 17)*. Vancouver, BC: USENIX Association, Aug. 2017.

[7] Y. Yarom and K. Falkner, "{FLUSH+ RELOAD}: A high resolution, low noise, l3 cache {Side-Channel} attack," in *23rd USENIX security symposium (USENIX security 14)*, 2014, pp. 719–732.

[8] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "Cachezoom: How SGX amplifies the power of cache attacks," *CoRR*, vol. abs/1703.06986, 2017.

[9] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE symposium on security and privacy*. IEEE, 2015, pp. 605–622.

[10] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," *Journal of the ACM (JACM)*, vol. 43, no. 3, pp. 431–473, 1996.

[11] E. Stefanov, M. V. Dijk, E. Shi, T.-H. H. Chan, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path oram: An extremely simple oblivious ram protocol," *J. ACM*, vol. 65, no. 4, apr 2018.

[12] L. Ren, C. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. van Dijk, and S. Devadas, "Constants count: Practical improvements to oblivious RAM," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 415–430.

[13] X. Wang, H. Chan, and E. Shi, "Circuit oram: On tightness of the goldreich-ostrovsky lower bound," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 850–861. [Online]. Available: https://doi.org/10.1145/2810103.2813634

[14] C. W. Fletcher, L. Ren, A. Kwon, M. van Dijk, and S. Devadas, "Freecursive oram: [nearly] free recursion and integrity verification for position-based oblivious ram," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 103116.

[15] S. Sasy, S. Gorbunov, and C. W. Fletcher, "Zerotrace: Oblivious memory primitives from intel sgx." in *NDSS*, 2018.

[16] K. Dinh Duy, J. Kim, H. Lim, and H. Lee, " IncognitOS: A Practical Unikernel Design for Full-System Obfuscation in Confidential Virtual Machines ," in *2025 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2025, pp. 4192–4209. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP61157.2025.00222

[17] A. Rane, C. Lin, and M. Tiwari, "Raccoon: Closing digital side-channels through obfuscated execution," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 431–446.

[18] A. Shafiee, R. Balasubramonian, M. Tiwari, and F. Li, "Secure dimm: Moving oram primitives closer to memory," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 428–440.

[19] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, and D. Song, "Phantom: Practical oblivious computation in a secure processor," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013, pp. 311–324.

[20] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *SIGARCH Comput. Archit. News*, vol. 23, no. 1, p. 20–24, mar 1995. [Online]. Available: https://doi.org/10.1145/216585.216588

[21] A. Gholami, Z. Yao, S. Kim, C. Hooper, M. W. Mahoney, and K. Keutzer, "Ai and memory wall," *IEEE Micro*, vol. 44, no. 3, pp. 33–39, 2024.

[22] A. Saulsbury, F. Pong, and A. Nowatzyk, "Missing the memory wall: the case for processor/memory integration," in *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, ser. ISCA '96. New York, NY, USA: Association for Computing Machinery, 1996, p. 90–101. [Online]. Available: https://doi.org/10.1145/232973.232984

[23] J. Nider, C. Mustard, A. Zoltan, J. Ramsden, L. Liu, J. Grossbard, M. Dashti, R. Jodin, A. Ghiti, J. Chauzi, and A. Fedorova, "A case study of Processing-in-Memory in off-the-Shelf systems," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, Jul. 2021, pp. 117–130.

[24] J. Gómez-Luna, I. E. Hajj, I. Fernandez, C. Giannoula, G. F. Oliveira, and O. Mutlu, "Benchmarking a new paradigm: An experimental analysis of a real processing-in-memory architecture," *CoRR*, vol. abs/2105.03814, 2021.

[25] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015*, D. T. Marr and D. H. Albonesi, Eds. ACM, 2015, pp. 105–117.

[26] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 27–39.

[27] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "Graphpim: Enabling instruction-level PIM offloading in graph computing frameworks," in *2017 IEEE International Symposium on High Performance Computer Architecture, HPCA 2017, Austin, TX, USA, February 4-8, 2017*. IEEE Computer Society, 2017, pp. 457–468.

[28] Y. Kwon, K. Vladimir, N. Kim, W. Shin, J. Won, M. Lee, H. Joo, H. Choi, G. Kim, B. An, J. Kim, J. Lee, I. Kim, J. Park, C. Park, Y. Song, B. Yang, H. Lee, S. Kim, D. Kwon, S. Lee, K. Kim, S. Oh, J. Park, G. Hong, D. Ka, K. Hwang, J. Park, K. Kang, J. Kim, J. Jeon, M. Lee, M. Shin, M. Shin, J. Cha, C. Jung, K. Chang, C. Jeong, E. Lim, I. Park, J. Chun, and S. Hynix, "System architecture and software stack for gddr6-aim," in *2022 IEEE Hot Chips 34 Symposium (HCS)*, 2022, pp. 1–25.

[29] J. H. Kim, S.-h. Kang, S. Lee, H. Kim, W. Song, Y. Ro, S. Lee, D. Wang, H. Shin, B. Phuah, J. Choi, J. So, Y. Cho, J. Song, J. Choi, J. Cho, K. Sohn, Y. Sohn, K. Park, and N. S. Kim, "Aquabolt-xl: Samsung hbm2-pim with in-memory processing for ml accelerators and beyond," in *2021 IEEE Hot Chips 33 Symposium (HCS)*, 2021, pp. 1–26.

[30] F. Devaux, "The true processing in memory accelerator," in *2019 IEEE Hot Chips 31 Symposium (HCS)*. Los Alamitos, CA, USA: IEEE Computer Society, aug 2019, pp. 1–24.

[31] D. Lavenier, J.-F. Roy, and D. Furodet, "Dna mapping using processor-in-memory architecture," in *2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, 2016, pp. 1429–1435.

[32] D. Lavenier, R. Cimadomo, and R. Jodin, "Variant calling parallelization on processor-in-memory architecture," in *2020 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, 2020, pp. 204–207.

[33] X. Zhang, G. Sun, C. Zhang, W. Zhang, Y. Liang, T. Wang, Y. Chen, and J. Di, "Fork path: Improving efficiency of oram by removing redundant memory accesses," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015, pp. 102–114.

[34] E. Stefanov, E. Shi, and D. Song, "Towards practical oblivious ram," in *NDSS*, 2012.

[35] K. Hsieh, S. Khan, N. Vijaykumar, K. K. Chang, A. Boroumand, S. Ghose, and O. Mutlu, "Accelerating pointer chasing in 3d-stacked memory: Challenges, mechanisms, evaluation," in *2016 IEEE 34th International Conference on Computer Design (ICCD)*, 2016, pp. 25–32.

[36] 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*, vol. 63, no. 6, pp. 3:1–3:19, 2019.

[37] Advanced Micro Devices, Inc, "Amd high bandwidth memory," https://www.amd.com/en/technologies/hbm, 2021, last accessed June 11, 2024.

[38] S. Newsroom, "Samsung develops industry's first high bandwidth memory with ai processing power," https://news.samsung.com/global/samsung-develops-industrys-first-high-bandwidth-memory-with-ai-processing-power, 2021, last accessed Aug 25, 2024.

[39] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, P. Ranganathan, and O. 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*, ser. ASPLOS '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 316–331.

[40] S. Lee, S.-h. Kang, J. Lee, H. Kim, E. Lee, S. Seo, H. Yoon, S. Lee, K. Lim, H. Shin, J. Kim, O. Seongil, A. Iyer, D. Wang, K. Sohn, and N. S. 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)*, 2021, pp. 43–56.

[41] A. Olgun, J. G. Luna, K. Kanellopoulos, B. Salami, H. Hassan, O. Ergin, and O. Mutlu, "Pidram: A holistic end-to-end fpga-based framework for processing-in-dram," *ACM Trans. Archit. Code Optim.*, vol. 20, no. 1, nov 2022. [Online]. Available: https://doi.org/10.1145/3563697

[42] S. Mosanu, M. N. Sakib, T. Tracy, E. Cukurtas, A. Ahmed, P. Ivanov, S. Khan, K. Skadron, and M. Stan, "Pimulator: A fast and flexible processing-in-memory emulation platform," in *Proceedings of the 2022 Conference & Exhibition on Design, Automation & Test in Europe*, ser. DATE '22. Leuven, BEL: European Design and Automation Association, 2022, p. 1473–1478.

[43] H. Kang, Y. Zhao, G. E. Blelloch, L. Dhulipala, Y. Gu, C. McGuffey, and P. B. Gibbons, "Pim-trie: A skew-resistant trie for processing-in-memory," in *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 1–14. [Online]. Available: https://doi.org/10.1145/3558481.3591070

[44] A. Lopes, D. Castro, and P. Romano, "Pim-stm: Software transactional memory for processing-in-memory systems," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '23. New York, NY, USA: Association for Computing Machinery, 2024, p. 815–827. [Online]. Available: https://doi.org/10.1145/3620665.3640428

[45] H. Kal, C. Yoo, and W. W. Ro, "Aespa: Asynchronous execution scheme to exploit bank-level parallelism of processing-in-memory," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 815–827. [Online]. Available: https://doi.org/10.1145/3613424.3614314

[46] J. T. Pawlowski, "Hybrid memory cube (hmc)," in *2011 IEEE Hot Chips 23 Symposium (HCS)*, 2011, pp. 1–24.

[47] S. Aga and S. Narayanasamy, "Invisimem: Smart memory defenses for memory bus side channel," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 94–106.

[48] A. O. Glova, I. Akgun, S. Li, X. Hu, and Y. Xie, "Near-data acceleration of privacy-preserving biomarker search with 3d-stacked memory," in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2019, pp. 800–805.

[49] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, June 2015, pp. 336–348.

[50] Y. Zhao, M. Gao, F. Liu, Y. Hu, Z. Wang, H. Lin, J. Li, H. Xian, H. Dong, T. Yang, N. Jing, X. Liang, and L. Jiang, "Um-pim: Dram-based pim with uniform & shared memory space," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, 2024, pp. 644–659.

[51] Y. Kwon, Y. Lee, and M. Rhu, "Tensordimm: A practical near-memory processing architecture for embeddings and tensor operations in deep learning," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 740–753. [Online]. Available: https://doi.org/10.1145/3352460.3358284

[52] D. Lee, D. Jung, I. T. Fang, C. che Tsai, and R. A. Popa, "An Off-Chip attack on hardware enclaves via the memory bus," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 487–504. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/lee-dayeol

[53] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: Cold boot attacks on encryption keys," in *Proceedings of the 17th USENIX Security Symposium, July 28-August 1, 2008, San Jose, CA, USA*, P. C. van Oorschot, Ed. USENIX Association, 2008, pp. 45–60.

[54] J. Choi, J. Kim, C. Lim, S. Lee, J. Lee, D. Song, and Y. Kim, "Guardiann: Fast and secure on-device inference in trustzone using embedded sram and cryptographic hardware," in *Proceedings of the 23rd ACM/IFIP International Middleware Conference*, ser. Middleware '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 15–28. [Online]. Available: https://doi.org/10.1145/3528535.3531513

[55] P. Colp, J. Zhang, J. Gleeson, S. Suneja, E. de Lara, H. Raj, S. Saroiu, and A. Wolman, "Protecting data on smartphones and tablets from memory attacks," *SIGARCH Comput. Archit. News*, vol. 43, no. 1, p. 177–189, mar 2015. [Online]. Available: https://doi.org/10.1145/2786763.2694380

[56] N. Zhang, K. Sun, W. Lou, and Y. T. Hou, "Case: Cache-assisted secure execution on arm processors," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 72–90.

[57] S. Zhao, Q. Zhang, Y. Qin, W. Feng, and D. Feng, "Minimal kernel: An operating system architecture for TEE to resist board level physical attacks," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. Chaoyang District, Beijing: USENIX Association, Sep. 2019, pp. 105–120. [Online]. Available: https://www.usenix.org/conference/raid2019/presentation/zhao

[58] A. Awad, Y. Wang, D. Shands, and Y. Solihin, "Obfusmem: A low-overhead access obfuscation for trusted memories," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 107119.

[59] Y. Wang, L. Orosa, X. Peng, Y. Guo, S. Ghose, M. Patel, J. S. Kim, J. Gomez-Luna, M. Sadrosadati, N. Mansouri-Ghiasi, and O. Mutlu, "FIGARO: improving system performance via fine-grained in-dram data relocation and caching," in *53rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2020, Athens, Greece, October 17-21, 2020*. IEEE, 2020, pp. 313–328. [Online]. Available: https://doi.org/10.1109/MICRO50266.2020.00036

[60] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Rowclone: fast and energy-efficient in-dram bulk data copy and initialization," in *The 46th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-46, Davis, CA, USA, December 7-11, 2013*, M. K. Farrens and C. Kozyrakis, Eds. ACM, 2013, pp. 185–197. [Online]. Available: https://doi.org/10.1145/2540708.2540725

[61] S. H. S. Rezaei, M. Modarressi, R. Ausavarungnirun, M. Sadrosadati, O. Mutlu, and M. Daneshtalab, "Nom: Network-on-memory for inter-bank data transfer in highly-banked memories," *IEEE Comput. Archit. Lett.*, vol. 19, no. 1, pp. 80–83, 2020. [Online]. Available: https://doi.org/10.1109/LCA.2020.2990599

[62] D. Chen, H. He, H. Jin, L. Zheng, Y. Huang, X. Shen, and X. Liao, "Metanmp: Leveraging cartesian-like product to accelerate hgnns with near-memory processing," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3579371.3589091

[63] K. K. Chang, P. J. Nair, D. Lee, S. Ghose, M. K. Qureshi, and O. Mutlu, "Low-cost inter-linked subarrays (LISA): enabling fast inter-subarray data movement in DRAM," in *2016 IEEE International Symposium on High Performance Computer Architecture, HPCA 2016, Barcelona, Spain, March 12-16, 2016*. IEEE Computer Society, 2016, pp. 568–580. [Online]. Available: https://doi.org/10.1109/HPCA.2016.7446095

[64] S. Volos, K. Vaswani, and R. Bruno, "Graviton: Trusted execution environments on gpus," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 681–696.

[65] H. Oh, A. Ahmad, S. Park, B. Lee, and Y. Paek, "Trustore: Side-channel resistant storage for sgx using intel hybrid cpu-fpga," ser. CCS '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 1903–1918.

[66] PCI-SIG, "Tee device interface security protocol (tdisp)," https://pcisig.com/tee-device-interface-security-protocol-tdisp, 2022, last accessed Sep 26, 2025.

[67] A. M. Devices, "Amd sev-tio: Trusted i/o for secure encrypted virtualization," https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/sev-tio-whitepaper.pdf, 2023, last accessed Sep 26, 2023,.

[68] NVIDIA, Corp, "Confidential compute on nvidia hopper h100," https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/HCC-Whitepaper-v1.0.pdf, 2023, last accessed September 20th, 2023,.

[69] Intel, "Intel® tdx connect architecture specification," https://www.intel.com/content/www/us/en/content-details/773614/intel-tdx-connect-architecture-specification.html, 2023, last accessed Oct 1, 2023,.

[70] N. Semiconductors, "Lpc540xx/lpc54s0xx user manual," 2019.

[71] U. SAS, "User manual," https://sdk.upmem.com/2024.1.0/, 2024, last accessed Aug 10, 2024.

[72] Intel, Inc, "Intel(r) software guard extensions for linux* os," https://github.com/intel/linux-sgx, 2023, last accessed June 4, 2023,.

[73] ——, "Intel® software guard extensions ssl," https://github.com/intel/intel-sgx-ssl, 2023, last accessed June 6, 2023,.

[74] S. Ghosh, L. S. Kida, S. J. Desai, and R. Lal, "A >100 gbps inline AES-GCM hardware engine and protected DMA transfers between SGX enclave and FPGA accelerator device," Cryptology ePrint Archive, Paper 2020/178, 2020. [Online]. Available: https://eprint.iacr.org/2020/178

[75] Design and Reuse, "Ahb aes with dma," https://www.design-reuse.com/sip/ahb-aes-with-dma-ip-46281, 2024, last accessed Aug 26, 2024,.

[76] K. D. Duy and H. Lee, "Se-pim: In-memory acceleration of data-intensive confidential computing," *IEEE Transactions on Cloud Computing*, pp. 1–18, 2022.

[77] sshsshy, "Zerotrace," https://github.com/sshsshy/ZeroTrace, 2022, last accessed Apr 28 , 2024,.

[78] J. Gu, B. Zhu, M. Li, W. Li, Y. Xia, and H. Chen, "A Hardware-Software co-design for efficient Intra-Enclave isolation," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 3129–3145. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/gu-jinyu

[79] Intel, Inc, "Intel performance counter monitor," https://github.com/intel/pcm, 2023, last accessed May 29, 2023,.

[80] Redis, Ltd, "Redis," https://redis.io/, 2024, last accessed February 29th, 2024,.

[81] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 143–154. [Online]. Available: https://doi.org/10.1145/1807128.1807152

[82] A. M. Devices, "Strengthening vm isolation with integrity protection and more," https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf, 2020, last accessed Sep 11 , 2025,.

Appendix A.
Access latency comparison between PIM-ORAM and ZeroTrace

We also provide the microbenchmark result of ZeroTrace [15], which is a representative SGX-based Path-ORAM implementation.

Total Data Size (GB)	**0.1**	**0.2**	**0.4**	**0.8**	**1.5**	**3.1**	**6.1**	**12.2**	**24.4**
Data Block Size (KB)	**1**	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**
ZeroTrace	7.99	14.60	27.81	54.34	107.27	213.40	**N/A**	**N/A**	**N/A**
PIM-ORAM	2.84	2.92	2.98	3.10	3.48	4.14	7.05	11.09	18.06
└*POMC*	0.43	0.51	0.57	0.69	1.07	1.73	4.64	8.68	15.65
└*DPU*	2.41	2.41	2.41	2.41	2.41	2.41	2.41	2.41	2.41

TABLE 2: Average access latency in milliseconds. *DPU* and *POMC* illustrate the PIM-ORAM's access latency breakdown of host-side (CPU) POMC execution vs. PIM-side DPU execution.

Table 2 shows the average access latency of PIM-ORAM and that of ZeroTrace. The access latency is measured by conducting 10,000 ORAM accesses. ZeroTrace shows higher latency than PIM-ORAM in overall. One of the main reasons is that ZeroTrace utilizes SHA256 and AES CTR mode to protect ORAM trees, whereas our implementation uses the lightweight integrity verification scheme that leverages AES GCM. However, such a different implementation hinders a fair comparison between PIM and CPU. Moreover, ZeroTrace failed to operate when the data block size exceeded 32KB, as a segmentation fault occurred. Thus, we decided to implement CPU-ORAM to effectively handle the given evaluation parameters and to make a fair comparison with PIM-ORAM by operating the identical Path-ORAM implementation.

Appendix B.
The performance comparison between SGX hardware mode and simulation mode

The SGX simulation mode provides functional correctness but shows different performance characteristics. However, we found that the simulation mode performs significantly better than its hardware counterpart. This is because the SGX simulation mode is mainly intended for correctness testing of SGX applications and does not perform memory encryption [78].

Block Size (KB)	**1**	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**
TPUT$_{SGXSim}$ **/ TPUT**$_{SGXHW}$	2.43	2.21	1.96	1.62	1.36	1.20	1.11	1.10	1.04

TABLE 3: Throughput comparison of 10,000 CPU-ORAM accesses on SGX simulation mode vs. on SGX hardware mode.

Table 3 reports the results from our local SGX-capable machine, equipped with Intel Core i9-10900K CPU (10 cores) @ 3.70GHz, and 128GB of DRAM, running CPU-ORAM in the simulation mode and also hardware mode. CPU-ORAM performs a large volume of memory access inside SGX simulation and unfairly receives an exemption on encryption overhead for all its memory accesses.