Palermo: Improving the Performance of Oblivious Memory using Protocol-Hardware Co-Design

Haojie Ye†, Yuchen Xia†, Yuhan Chen†, Kuan-Yu Chen†, Yichao Yuan†, Shuwen Deng‡, Baris Kasikci*, Trevor Mudge†, Nishil Talati†
†University of Michigan, USA; ‡Tsinghua University, China; *University of Washington, USA
Email: yehaojie@umich.edu

***Abstract*—Oblivious RAM (ORAM) hides the memory access patterns, enhancing data privacy by preventing attackers from discovering sensitive information based on the sequence of memory accesses. The performance of ORAM is often limited by its inherent trade-off between security and efficiency, as concealing memory access patterns imposes significant computational and memory overhead. While prior works focus on improving the ORAM performance by prefetching and eliminating ORAM requests, we find that their performance is very sensitive to workload locality behavior and incurs additional management overhead caused by the ORAM stash pressure.**

This paper presents Palermo: a protocol-hardware co-design to improve ORAM performance. The key observation in Palermo is that classical ORAM protocols enforce restrictive dependencies between memory operations that result in low memory bandwidth utilization. Palermo introduces a new protocol that overlaps large portions of memory operations, within a single and between multiple ORAM requests, without breaking correctness and security guarantees. Subsequently, we propose an ORAM controller architecture that executes the proposed protocol to service ORAM requests. The hardware is responsible for concurrently issuing memory requests as well as imposing the necessary dependencies to ensure a consistent view of the ORAM tree across requests. Using a rich workload mix, we demonstrate that Palermo outperforms the RingORAM baseline by 2.9×, on average, incurring a negligible area overhead of 5.78mm^2 (less than 2% in 12th generation Intel CPU after technology scaling) and 2.14W without sacrificing security. We further show that Palermo also outperforms the state-of-the-art works PageORAM, PrORAM, and IR-ORAM.

I. Introduction

Building a secure oblivious RAM (ORAM) design for cloud workloads is critical because it ensures the confidentiality of accessing sensitive data. ORAMs are designed to offer a secure memory access pattern, concealing the information being accessed and safeguarding the privacy of sensitive data [16]. This is particularly important in the context of modern AI workloads (*e.g.,* recommendation and large language models), where the memory access pattern can reveal sensitive prior product selections, such as user preferences or prompts to the language model. ORAM can protect against these potential vulnerabilities by *obfuscating the access pattern to the memory*, preventing an attacker from inferring any useful information about where the data is being accessed.

However, the obfuscation of the memory access patterns using ORAM incurs a *significant performance overhead*. For example, to hide memory access in a 16GB protected memory space, classical ORAM implementations such as PathORAM [46] and RingORAM [42] on average convert a single access to 576 and 470 accesses, respectively. This is because the critical metadata structure to protect the memory space `PosMap` is too large to fit on-chip. Addressing this issue necessitates a hierarchical ORAM design, often involving three levels, to establish a mapping between the physical and ORAM addresses [12]. Therefore, optimizing the performance of ORAM implementations is *crucial for practicality*.

Prior works [8], [39], [41], [50] present ways to optimize the performance of ORAM by prefetching and eliminating as many ORAM requests as possible. However, we find two limitations of these works. First, the benefit of these designs is highly sensitive to the application behavior, benefiting workloads with high spatial locality. They fall short in optimizing other workloads with low data locality. Second, prefetching-based optimizations modify the original PathORAM algorithm, which selects ORAM leaves independently and uniformly at random. These designs map consecutive physical addresses in the original memory space to the same leaf in the ORAM tree. This practice significantly increases the ORAM stash pressure and introduces background evictions, ultimately resulting in diminishing returns and a capped speedup at 3.2× even with a synthetic trace exhibiting perfect locality.

This paper presents Palermo—a protocol-hardware co-design to improve ORAM performance. The design objective of Palermo is to enhance performance without compromising prior ORAM security guarantees [46]. Using the performance analysis of RingORAM [42], we discover that the average memory bandwidth utilization of executing RingORAM on the tested benchmarks is less than 30%. Considering the memory-intensive nature of ORAM, this is surprising; our further analysis of reduced bandwidth utilization reveals that enforced dependencies in the protocol limit memory-level parallelism.

To this end, we design Palermo that re-architects the ORAM protocol, and designs a novel hardware ORAM controller architecture to fully unlock the potential of our protocol. The protocol is designed to improve the concurrency in serving ORAM requests. In particular, the Palermo protocol introduces intra- and inter-request parallelism to increase memory bandwidth utilization. Intra-request parallelism involves overlapping memory requests related to different steps in accessing various hierarchical levels of ORAM. Inter-request parallelism entails concurrent processing of memory requests across distinct ORAM requests. While overlapping memory requests, the Palermo protocol identifies and enforces minimal dependencies, ensuring a consistent view of the ORAM tree after each address remapping.

To support the proposed protocol, we also present the design of an ORAM controller. The hardware architecture consists of on-chip memory structures for stash, position map, tree-top cache [30], and a mesh of Processing Engines (PEs) that enable concurrent serving of ORAM memory requests. We present both qualitative and quantitative security analyses of Palermo to assess the uniform randomness of the attacker's view of the DRAM traffic and the isolation of each LLC miss latency, even when concurrently serving multiple ORAM memory requests at the ORAM controller.

To demonstrate the effectiveness of Palermo, we use a wide range of workloads including SPEC17 [3], graph analytics [2], [31], deep learning [18], [19], [37], and key-value accesses [23]. Palermo outperforms PathORAM [46] and RingORAM [42] by 2.4× and 2.2×, on average. Borrowing ideas from prior prefetching works [8], [39], [50], we show that Palermo can further improve this performance gain by 3.2× and 2.9×, respectively. We also show improvements against PageORAM [38], PrORAM [50], and IR-ORAM [41] while upholding the same security level as in these works. The performance improvements are attributed to improved memory bandwidth utilization from 21% in the RingORAM baseline to 59% in Palermo. Our post-synthesis RTL results reveal that Palermo only consumes 5.78mm^2 silicon area (less than 2% in Intel 12th Gen CPU after technology scaling [48], providing a low-complexity practical solution) and 2.14W. The key contributions of this work are as follows.

- Performance analysis of prior ORAM implementations revealing further optimization opportunities.
- Design of a new ORAM protocol that enables overlapping memory requests within the same ORAM request and between multiple ORAM requests.
- Design of the ORAM controller architecture to apply the minimal required protocol synchronization and improve memory-level parallelism.
- Palermo: end-to-end protocol-hardware co-design that improves an average performance of RingORAM by 2.9×.

II. ORAM Background and Threat Model

A. A Case For Oblivious RAM/Memory

The evolution of cloud services has facilitated the execution of applications with exceedingly large memory and computing demands. While clients enjoy the benefit of using these cloud services, clients may unintentionally give up the privacy of running their applications even when data is fully encrypted. We briefly discuss an example of how an untrusted cloud can learn sensitive information from clients.

Consider a client executing a Large Language Model (LLM) inference workload using untrusted outsourced memory to store the token feature table. The external memory party (attacker) has the capability to monitor the complete memory bus and eavesdrop on the memory request traces of the victim. Even though the transferred feature values are fully encrypted, the attacker can still observe the full address request trace and infer about token address and frequency distribution. The attacker can iterate over common LLM models and map memory traces to corresponding tokens. In this way, the attacker can fully reconstruct all the user's prompts and extract sensitive knowledge from the user. The aforementioned example highlights various recent incidents and concerns related to corporate information leakage by ChatGPT [20], [35]. Such outcomes impose a substantial trust burden on the reliance on third-party cloud memory services.

This calls for an urgent need for privacy in using cloud memory services. One attractive privacy solution is to use the oblivious memory protocol (ORAM). ORAM enables the user to hide the memory access addresses, types, and patterns from the attacker when using the untrusted cloud memory service so that their access patterns are computationally indistinguishable from random accesses. In fact, ORAM has been incorporated into Signal's ecosystem [10], facilitating users to conduct contact discovery while safeguarding the privacy of each user's social map through ORAM.

B. Threat model

We model a common scenario that a shared server with cloud settings is launched to serve remote clients. The server is equipped with a standalone secure processor, *i.e.,* the Trusted Computing Base (TCB), that includes cores and a small amount of on-chip memories. The cloud service can use any commodity off-chip memory modules and can snoop the memory bus as an attacker. The processor runs a private or public program on private data. On LLC misses, the processor issues to a trusted on-chip ORAM controller to access data in untrusted external memory. The attacker is "honest but curious" and uses any possible information to gather insights from the victim process, such as memory access hotspots, memory bus contents, timings, *etc.* Any distinguishable behavior from the original LLC miss trace is considered an obliviousness violation [46]. This is a common threat model that is present in most prior ORAM works [12], [34], [40], [41], [50].

C. ORAM Protocol

Many ORAM protocols have been proposed to protect users' memory traces [9], [15], [21], [25], [42], [46]. This work focuses on an optimized protocol RingORAM [42]. Algorithm 1 shows the pseudocode of the RingORAM protocol. RingORAM manages the untrusted cloud memory as a binary tree—ORAM tree. Each node in the tree is a bucket and consists of multiple blocks of data at the cache line granularity. The blocks are randomly permuted and store up to `Z` blocks that contain real blocks, and at least `S` blocks that contain dummy blocks. All data is encrypted with different keys. Each physical address `PA` in the secure logical memory space does not directly correspond to `PA` on the cloud memory; instead, it maps to a specific `leaf` (top node) of the ORAM tree. In the secure domain, the `PosMap` data structure keeps track of `leaf` mappings and the actual node location of every

Algorithm 1 Pseudocode for RingORAM algorithm to serve an ORAM request.

```
procedure RingORAMAccess(PA, op, data')
    Global: PosMap: Position map that stores the mapped leaf ID for each PA
// PosMap stored in recursive ORAM due to linear size growth with DRAM capacity
    Global: Stash: Buffer that temporarily stores the data read from DRAM
    Global: round: Enforces a stash eviction for every A accesses
    leaf = PosMap[PA]
    leaf' = UniRandLeaf
    PosMap[PA] = leaf'
    data = ReadPath(leaf, PA)
    if op == READ then
        return data to processor from Stash
    else
        Stash[PA] = data'
    if (round++) % A == 0 then
        EvictPath()
    EarlyReshuffle(leaf)
    return

procedure ReadPath(leaf, PA)
    Input: leaf: leaf ID
    Input: PA: Physical address that misses the LLC
    Global: NodeMetadata: Keeps track of the ORAM tree per node metadata
// NodeMetadata stored in DRAM due to linear size growth with DRAM capacity
    for all NodeID ∈ leaf along root do
        i = unused_fake_blk if PA in NodeID else real_blk
        Stash = Stash ∪ ReadBucket(NodeID, i)
        NodeMetadata[NodeID].update()

procedure EarlyReshuffle(leaf)
    Input: leaf: leaf ID ▷ Check along leaf whether buckets can be further used
    for all NodeID ∈ leaf along root do
        if NodeMetadata[NodeID].accessed == S then
            ResetBucket(NodeID)

procedure EvictPath()
    Global: G: leaf ID to exercise eviction
// RingORAM uses a deterministic ring counter eviction leaf sequence
    for all NodeID ∈ G along root do
        ResetBucket(NodeID)
    G = next ring counter

procedure ResetBucket(NodeID)
    Input: NodeID: Node ID in the ORAM tree that needs a bucket reset
    Fetch_offset = NodeMetadata[NodeID].unused_real_blk()
    Fetch_offset.pad(NodeMetadata[NodeID].unused_fake_blk())
// Fetched offset size is padded to Z to ensure obliviousness on the memory bus
    for all i ∈ Fetch_offset do
        Stash = Stash ∪ ReadBucket(NodeID, i)
    WriteBucket(NodeID, Stash)    ▷ Attempt to push to the ORAM tree
    NodeMetadata[NodeID].reset()
```

PA, and Stash is a small buffer (of typical size 256) that holds the blocks streaming from cloud memory to the trusted processor. RingORAM maintains an invariant that PA in the logical memory space lies along the path from its mapped leaf connecting to the root **or** lies in the Stash.

When LLC encounters an R/W miss on a physical address (PA), it queries PosMap to locate the mapped leaf and the position of the node. RingORAM then loads one block from each node along the path from the mapped leaf to the root, generating a stream of memory addresses. Each node selects the real block if it corresponds to the actual location of PA, otherwise a dummy block is chosen. All touched blocks are invalidated for further use (❶). Fig. 1 left shows the example of LLC missing on the light blue block. After consulting the PosMap and finding its mapped leaf, exactly one block from each node is streamed into Stash and it is guaranteed to have the block of interest and the rest being dummy blocks. After loading along the path, Stash decrypts the block of interest to

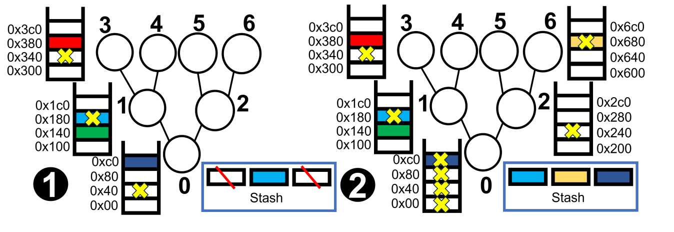


Fig. 1. **A toy ORAM access example for illustration purposes. The shown ORAM tree has Z and S set to 2. In practice, Z and S are much higher. On LLC miss on light blue block, the missed physical address is converted to leaf number to launch accesses along the path and pull blocks into the stash. Once any node is touched S times, a reset routine is launched.**

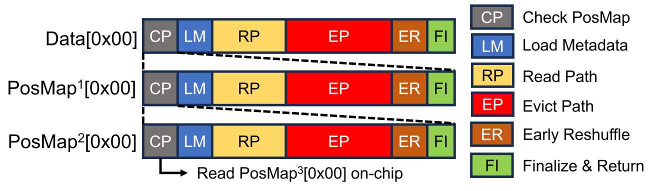


Fig. 2. **Hierarchical ORAM memory spaces. Because the secret data structure PosMap exceeds the on-chip memory capacity, a second-level ORAM protocol is launched to protect the access to data structure PosMap. The recursive process continues until PosMap of the protected data structure can be stored on-chip.**

serve the LLC miss, holds the block temporarily, and discards the dummy blocks. After each access of PA, its mapped leaf is randomly selected again. If S blocks in a node are marked as invalid, violating the future read routine due to the absence of available dummy blocks, the node initiates a reset routine. RingORAM first loads Z blocks from the node to Stash and clears the node, then exhaustively iterates through the blocks in Stash to push them back to the reset node (❷). Fig. 1 right shows the example of LLC missing on the yellow block. After streaming from the mapped leaf, node 0 is accessed S times and thus requires a reset routine. To reset, all blocks in node 0 are streamed into Stash before Stash pushes its content back to the reset node as much as possible. All block permutations and valid bits are reset afterward. Periodic eviction occurs after every A ORAM requests, where a leaf is selected, initiating a reset routine for all nodes along the path from the leaf to the root. RingORAM provides theoretical and empirical evidence that the probability of Stash size overflowing 256 is negligible ($< 2^{-103}$) [42].

D. Practical ORAM Implementation

While protecting a large memory space with ORAM, the capacity of PosMap typically exceeds the on-chip memory capacity on a CPU. For example, to protect 16GB user space, PosMap takes 2GB of capacity to store the leaf mapping and node position of all data blocks. Hence, hierarchical ORAM memory spaces are established to access the encrypted PosMap in the same oblivious access fashion [12]. In Fig. 2, an example is illustrated, wherein an LLC miss occurs for a 0x00 read operation in the protected user space. In response, the ORAM controller initiates a `PosMap` check for block 0x00. This launches a sub-ORAM read of the data structure `PosMap` at 0x00. The sub-ORAM launches recursively until the recursive `PosMap` can be stored on-chip. Only after the `PosMap` access, the protocol can find the mapped leaf and resume the RingORAM algorithm execution.

To distinguish the recursive `PosMaps`, we name $\texttt{PosMap}^1$ that keeps track of the protected memory space, and $\texttt{PosMap}^2$ that keeps track of $\texttt{PosMap}^1$, and so on. Similar to prior works, we use 3 levels of `PosMaps` and store $\texttt{PosMap}^3$ on-chip, the same as all prior works [41], [42], [46]. RingORAM imposes a dependency between levels of `PosMaps` and distinct requests. In other words, when presented with requests R1 and R2 for the protected memory space, RingORAM accesses recursive sub-ORAMs to fulfill R1 ($\texttt{PosMap}^2$, $\texttt{PosMap}^1$, `Data`) first and then proceeds to R2 ($\texttt{PosMap}^2$, $\texttt{PosMap}^1$, `Data`) in sequence.

III. Analysis of Prior ORAM Proposals

A. *Analysis of Classical ORAM Implementations*

Considering the memory-intensive nature of ORAM, it might be assumed that it fully utilizes the available memory bandwidth. Moreover, since RingORAM reduces the number of memory accesses, one might intuitively expect it to proportionally outperform the PathORAM baseline. *However, we discover that this is not the case.* The RingORAM protocol only marginally outperforms PathORAM by 10% despite a significant 42% of reduction in the number of accesses.

Fig. 3 shows the performance breakdown of RingORAM using detailed methodology discussed in §VII. The baseline ORAM controller issues all resulting reads and incurred stash evictions of an ORAM request to the memory controller. Without waiting for the writes to commit, the ORAM controller keeps issuing the next ORAM request to saturate the memory controller queues for memory-level parallelism. In the RingORAM protocol, ORAM requests are served one after another to avoid a block being read by multiple requests before being updated. Otherwise, it gives the attacker accurate information about the dummy location and violates obliviousness. For a single request, after selecting the leaf-to-root path, the request needs to load metadata on-chip to determine which blocks in the nodes are usable. The data load can only start after their dependent metadata are loaded. Additionally, the protocol specifies that `Stash` can only push blocks to the ORAM tree after `ReadPath` and `ResetBucket` pull all their required blocks. These requirements introduce substantial synchronization overhead, as pull requests are held up while waiting for their dependent metadata to be in place, and the `Stash` is delayed until all pull requests are in position.

We refer to the memory controller stall cycles resulting from these reasons as ORAM-sync cycles. Surprisingly, ORAM-sync accounts for around 72.4% of the execution time. In essence, RingORAM dedicates a substantial portion of time to stalling the memory controller, awaiting the completion of preceding high-latency pull requests before it can advance with the protocol. These stalls are inherent to executing the

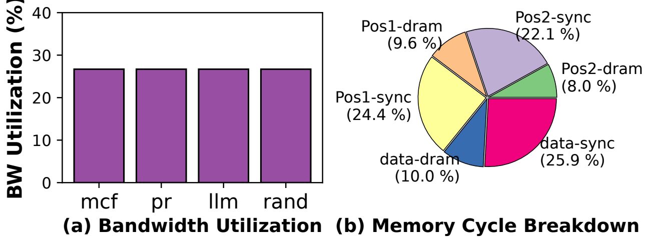


Fig. 3. **RingORAM protocol bandwidth utilization and performance breakdown. RingORAM incurs less than 30% bandwidth utilization, which is similar across workloads due to the application of the ORAM protocol. ORAM-sync overhead accounts for 72.4% of the execution time, which indicates the memory stays idle and spends most of the time waiting for long-latency pull requests to be serviced.**

RingORAM protocol, even with a multi-issue ORAM controller attempting to saturate the memory bandwidth. Notably, the average DRAM bandwidth utilization remains below 30% across all workloads. We additionally provide the approximate analytical calculation to support the cycle-accurate simulation results. The DRAM request latency for row-hits and row-misses are tCL and (tCL+tRP+tRCD), respectively. We find that 48.2% of the requests are row-hits in RingORAM. Furthermore, the average memory controller queue occupancy across all channels is 21.1 due to frequent dependency stalls of the protocol. Using DRAM timing parameters for DDR4-3200, the average bandwidth we find is 64B $\times$ 21.1 / 46.9ns = 28.8GB/s (theoretical-maximum: 102.4GB/s): close to 28.1% bandwidth utilization. Note that despite the varying nature of these workloads in terms of memory traffic, each LLC miss request undergoes conversion into a single ORAM request by mapping to a unique ORAM tree leaf on every occasion. The homogenization of memory bandwidth utilization occurs due to applying the ORAM protocol, resulting in indistinguishable memory traffic across different workloads, thereby achieving the primary objective of ORAM.

The access to sub-ORAMs accounts for 64.1% of the time, which is due to the capacity limitation of on-chip structures (§II-D). Additionally, the protocol forces these accesses to issue and complete in order, servicing one request at a time. This study reveals that there is an untapped potential for memory-level parallelism in serving different requests because different `PosMaps` are protecting different memory spaces. For instance, concurrently serving accesses R1 $\texttt{PosMap}^1$ and R2 $\texttt{PosMap}^2$ does not result in conflicts. No prior work takes advantage of this optimization opportunity.

B. *Prefetch-based Optimization Strategies*

To improve ORAM performance, many prior works [39], [41], [50] propose to eliminate ORAM requests. For example, PrORAM and its variants [8], [39], [50] use PathORAM [46], a prior state-of-the-art protocol, as their baseline. PathORAM manages the ORAM tree similarly to RingORAM, with a key distinction being that in PathORAM, a `node` does not differentiate between real or dummy blocks. Each LLC miss

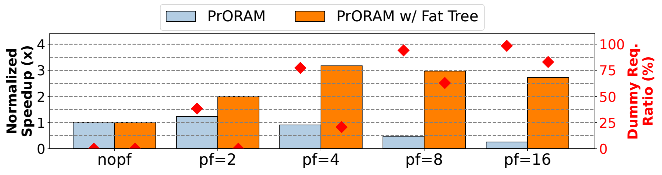


Fig. 4. **Normalized speedup of PrORAM and LAORAM (PrORAM w/ Fat Tree) running stm, a synthetic workload where consecutive cache line addresses are missed subsequently by the LLC. pf=X refers to forcing mapping to the same leaf for a prefetch length of X. A high dummy request ratio limits the performance scaling despite the present locality.**

on PA in PathORAM results in loading **all** blocks along the nodes connecting from its mapped leaf to the root. Additionally, upon loading from a leaf, PathORAM evicts the same leaf path immediately. The PrORAM protocol [50] forces the mapping of consecutive physical addresses in the original memory space to the same leaf in the ORAM tree. This way, one load of a physical address effectively *prefetches* multiple data blocks from the ORAM tree to Stash and then to LLC. Subsequent accesses, if hit the LLC, bypass the ORAM protocol. This solution achieves a notable speedup by mitigating the ORAM workload when the original memory access exhibits high locality.

However, PrORAM optimization suffers from two limitations, which we discuss through qualitative and quantitative analyses as follows. First, the prefetch performance benefit is sensitive to the locality pattern. This optimization has little effect on workloads with low to moderate locality in their original LLC miss trace such as some SPEC benchmarks, real-world graph analytics, and random key-value accesses, with demonstrated in performance evaluation in §VIII-A. Second, forcing the mapping of consecutive physical addresses in the original memory space to the same leaf in the ORAM tree limits the scope that Stash can distribute its blocks back to the ORAM tree. This violates the premise in PathORAM proof that blocks are assigned "independently and uniformly at random" into leaves of the ORAM binary tree [46]. As a result, background eviction is introduced when Stash overflows in this scenario [43].

Whenever the Stash size exceeds the threshold, a dummy ORAM request is inserted to read from and write to a random dummy path. This path access does not contribute to fulfilling any LLC requests but only helps to further clear the Stash. Fig. 4 shows the quantitative results of how these dummy requests affect performance. The experiment models PrORAM with a 1024-entry Stash protecting stm, a synthetic workload where consecutive cache line addresses are missed subsequently by the LLC. Ideally, a higher prefetch length configured in PrORAM should perform better than a lower one because the perfect locality of the trace can eliminate more of the ORAM requests. However, we find that PrORAM performance does not scale with the prefetch length.

At a prefetch length of 4, although the original ORAM leaf accesses are reduced by 4×, the Stash pressure has to repeatedly insert dummy requests, and the fraction of dummy requests accounts for 77.3% of the total ORAM requests. This leads to even a slowdown compared to the no prefetch case. PrORAM proposes to use a threshold to adapt to the background eviction frequency and dynamically disable or limit the prefetch length. This achieves up to 1.5× speedup over the no prefetch case (see Fig. 7 in PrORAM [50]).

LAORAM proposes a Fat-Tree structure to allocate 2× block size at the root level and gradually decrease the bucket size going up towards the leaf [39]. This significantly reduces the Stash pressure and decreases the dummy request ratio. However, its performance is capped at 3.2× at a prefetch length of 4 even with perfect locality in stm. Note that LAORAM reports a few over 3.2× speedups in some workloads. This is because LAORAM uses a software-managed Stash and allows the Stash to grow above 3600 after 12500 ORAM accesses and remain unbounded throughout the workload lifetime (see Fig. 8 in LAORAM [39]). This reduces the dummy request ratio observed in LAORAM. However, because *all* entries in Stash need to be probed on every ORAM access, a high-performance on-chip ORAM solution requires Stash to be in hardware and remain small due to the need for area efficiency and probing with high associativity [30]. Such an unbounded Stash cannot be justified in a high-performance on-chip hardware solution.

C. *Summary of Challenges Optimizing ORAM Performance*

The ORAM performance is limited due to its memory-intensive nature, where a single memory request in the unsecured domain is converted into 100s of memory requests [38], [42]. As a response, prior works [8], [39], [41], [50] attempt to improve performance by bypassing the ORAM protocol for a subset of memory requests. While these designs yield some performance improvement, the benefits are highly sensitive to workload behavior. Additionally, we uncover that the benefit of these optimizations does not scale with prefetch length even with perfect locality due to frequent Stash overflows and the introduction of dummy ORAM requests. The primary challenge in enhancing the performance of ORAM lies in accelerating a broad class of workloads with varying locality patterns and avoiding dummy requests. Palermo addresses this open research question.

IV. Introducing Concurrency using Palermo

A. *Key Design Goals*

The primary goal of Palermo is to improve performance by enhancing the memory bandwidth utilization without sacrificing security. The goals of Palermo design are:

- *Simultaneously processing multiple ORAM requests.* As discussed in §III-A, accessing multiple ORAM requests concurrently is the key to overcoming the long ORAM request latency and improving bandwidth utilization.
- *Minimizing execution bubbles.* Facilitating concurrent access to multiple ORAM requests necessitates identifying minimal dependencies between them, as this is integral to achieving high-performance ORAM while preserving memory functionality correctness and obliviousness.

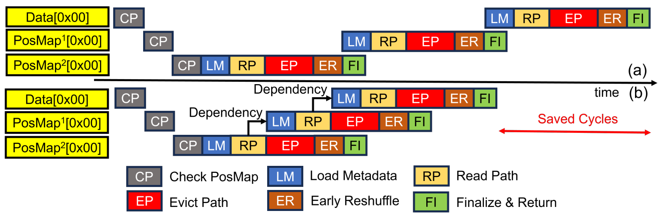


Fig. 5. **Intra-request parallelism in serving a single ORAM request.**

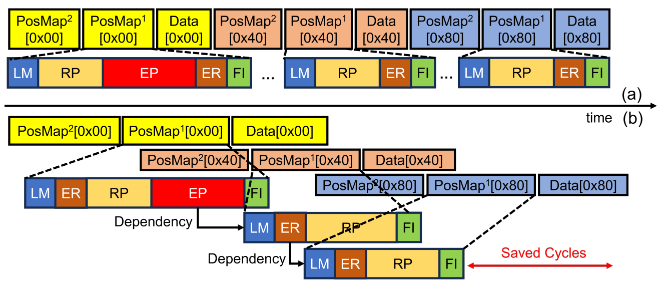


Fig. 6. **Inter-request parallelism in serving multiple ORAM requests.**

- *No compromise on security.* The protocol must be carefully designed such that any concurrent access behavior does not compromise the security guarantee of the original protocol.

B. Unlocking Parallelism in ORAM Protocol

To enhance performance, we analyze the minimal dependencies between ORAM requests that the ORAM controller must adhere to for correctness. We crucially observe that there are two categories of achievable parallelism to concurrently serve ORAM requests while ensuring correct functionality.

Intra-Request Parallelism. In Fig. 5(a), the baseline RingORAM protocol serving a single ORAM request of 0x00 with a hierarchical design is illustrated. Each level of ORAM launches a recursive sub-ORAM to check the PosMap value of the requested block. We make a crucial observation that each sub-ORAM memory space is exclusive; in other words, the ORAM memory space to protect $\texttt{PosMap}^2$ is entirely exclusive from $\texttt{PosMap}^1$. Thus, accesses to one level of sub-ORAM can be concurrently executed with accesses to another. Fig. 5(b) shows the minimal dependency that needs to be respected when serving different sub-ORAMs for a single request. $\texttt{PosMap}^1$ `LM` step can start as soon as the mapped leaf is known. This is resolved as soon as $\texttt{PosMap}^2$ `RP` is completed. The write to $\texttt{PosMap}^2$ ORAM tree (`EP`) and the read of $\texttt{PosMap}^1$ ORAM tree (`RP`) can be executed in parallel without any conflicts because they access exclusive memory spaces. Using these observations, we design the Palermo protocol that unlocks intra-request parallelism.

Inter-Request Parallelism. Fig. 6(a) shows the baseline RingORAM protocol serving multiple ORAM requests. Assuming LLC misses on addresses 0x00, 0x40, and 0x80, in the baseline, ORAM access to $\texttt{PosMap}^2$[0x00] executes the RingORAM protocol in order followed by the access to $\texttt{PosMap}^1$[0x00]; ORAM access to $\texttt{PosMap}^2$[0x40] is serialized after the access to `Data`[0x00]. We observe that sub-ORAMs on the same level (*e.g.*, $\texttt{PosMap}^1$[0x00] and $\texttt{PosMap}^1$[0x40]) can be concurrently accessed in most conditions, even though they share the same memory space. This is a distinctive characteristic of RingORAM because the protocol ensures that it invalidates precisely one data block position in the node upon being touched, and subsequent access to the node always selects a different position in the node if available. This unique characteristic of the RingORAM protocol unlocks massive opportunities for parallelism. This exclusivity access guarantee, however, can be violated once the node resets. This is because the reset operation pushes values from the stash and alters the content of the ORAM tree, and subsequent requests must have the updated view of the tree to maintain correctness. Hence, the modification of the ORAM tree (`EP` and `ER`) by one ORAM request and the read (`LM` and `RP`) by the subsequent request forms the minimal critical section for concurrently accessing multiple requests on the same sub-ORAM level. The `ER` step is executed on every access, whereas `EP` is executed after every `A` access.

To overcome this challenge, we propose to re-order and hoist the execution of the `ER` step to the earliest stage possible. The purpose of this is to resolve the write-to-read critical section as soon as possible. This approach allows the current request to modify the ORAM tree at the earliest opportunity to a “good to read” state and then pass the tree to the subsequent request. Meanwhile, `EP` is serialized after `RP` to uphold the theoretical guarantee that the stash remains bound to a fixed size, regardless of the concurrency order. In Fig. 6(b), our proposed protocol is depicted to minimize the critical path of concurrently running multiple requests. Consider the $\texttt{PosMap}^1$ ORAM tree, the execution of $\texttt{PosMap}^1$[0x40] can be issued as soon as $\texttt{PosMap}^1$[0x00] `EP` stage is complete. The execution of $\texttt{PosMap}^1$[0x80] can be issued once $\texttt{PosMap}^1$[0x40] `ER` stage is complete. All `RP` (non-store) steps from different requests in the same memory space can also be overlapped, which saves a significant amount of cycles. This design eliminates unnecessary dependencies.

C. Palermo Protocol Details

Algorithm 2 presents pseudocode for the proposed Palermo protocol that includes the following changes. First, we elevate the execution order of `EarlyReshuffle` to preserve correctness. Second, Palermo protocol serves an arbitrary number of requests concurrently while using `CommitHead` to synchronize the memory serving order to the same order they are issued. This ensures that using concurrent ORAM does not introduce any memory consistency issues. For each request, the `CommitHead` waits for all previous tree locks to be released. Then, it locks the ORAM tree until all operations that possibly modify the ORAM tree finish.

Palermo protocol (Palermo-SW) enables concurrent ORAM accesses and exploits the inter-request parallelism as multiple `ReadPath()` can be overlapped. However, the coarse-grained nature of software synchronization limits the benefit of Palermo. For example, the synchronization primitive

Algorithm 2 Pseudocode for Palermo algorithm to serve ORAM requests concurrently. Changes are marked in red.

```
procedure PALERMO ORAMACCESS(PA, op, data', GlobalID)
    Input: GlobalID: Global issue ID when accessing concurrently
    Global: CommitHead: Synchronization for original memory request order
    while(CommitHead != GlobalID) {;}        ▷ sleep and wait for sync
    leaf = UniRandomLeaf if PA pending else PosMap[PA]
    leaf' = UniRandomLeaf
    Mark PA as pending
    PosMap[PA] = leaf'
    EarlyReshufflePreCheck(leaf)
    if GlobalID % A != 0 then
        AtomicAdd(&CommitHead, 1)
    data = ReadPath(leaf, PA)
    if op == READ then
        return data to processor from Stash
    else
        Stash[PA] = data'
    if GlobalID % A == 0 then
        EvictPath()  ▷ Remove pending status upon blocks evicted from stash
        AtomicAdd(&CommitHead, 1)
    return

procedure EARLYRESHUFFLEPRECHECK(leaf)
    for all NodeID ∈ leaf along root do
        if NodeMetadata[NodeID].accessed == S - 1 then
            ResetBucket(NodeID)     ▷ Mark Node as bypassed in ReadPath()
```

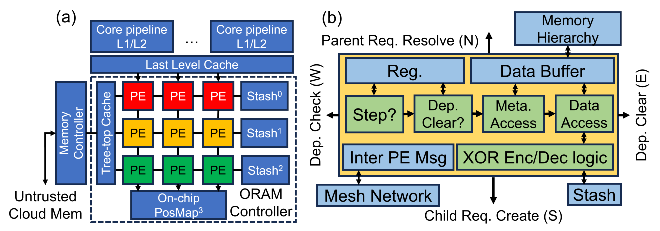


Fig. 7. **(a) Palermo ORAM controller consists of a 2D array of PEs, where each row of PE serves separate hierarchical sub-ORAMs, and each column of PE serves multiple ORAM requests concurrently, and (b) a single PE architecture for Palermo.**

placed around the PosMap check prevents the intra-request parallelism. To address this, we present the design of ORAM controller architecture to exploit parallelism at both levels.

V. Palermo ORAM Controller Design

This section presents the hardware design of an ORAM controller to fully support the memory-level parallelism introduced by Palermo protocol.

A. Hardware Support for Unlocking Additional Parallelism

Fig. 7 shows the proposed hardware architecture of Palermo ORAM controller. The architecture consists of 3 levels of Processing Elements (PEs). Each row of PEs serves a separate protected memory space of Data, PosMap^1, and PosMap^2. Each column of PEs serves a single request issued from an LLC miss. For example, three misses will be mapped to three columns of PEs. Within each PE, the finite state machine executes Algorithm 2 independently. Protocol-level dependencies are managed by the communication with neighboring PEs. The execution flow operates as follows.

Check PosMap: PE receives the query for PosMap from the parent (from the north) and sends it to the child (to the south), awaiting the response from the child to retrieve the leaf ID mapped to the requested data block.

Load Metadata: PE waits until the sibling dependency from the west is cleared. PE then loads metadata from the leaf to the root of the ORAM tree.

Early Reshuffle: Checking the loaded metadata, PE resets all nodes along the path that need an early reshuffle. After all reshuffles are issued, it clears the dependency for the sibling to the east (*i.e.,* ORAM tree is ready for next read).

Read Path: Processing the metadata, PE issues requests and loads data along the leaf to root, decrypts blocks, and responds to the parent with the values of the requested block.

Evict Path: If there is an eviction being scheduled, the stash is re-encrypted and an eviction process is initiated. This clears the dependency for the sibling to the east.

Finalize: PE is ready to retire and clears up when all rows (sub-ORAMs) of the same request are finalized.

All PEs execute an identical workflow and concurrently issue memory requests to enhance bandwidth utilization. The PE array forms a ring, and once a new request occupies the left-most PE, the right-most PE sends clear signals to the left-most PE regarding sibling dependency.

B. A Walk-Through Example

Figure 8 illustrates the proposed hardware's operation with a 3×3 PE mesh. Initially, at t = 0, LLC has three outstanding misses: 0x00R, 0x40R, and 0x80R. Palermo controller first registers all 3 requests in a separate column in Data PE. The on-chip storage of PosMap^1 for address 0x00R is absent, resulting in the designation of the Data-0 PE phase as CP, initiating an access to address PosMap^1[0x00] in the sub-ORAM. This launches another recursive access, setting the PosMap^1-0 PE the CP phase. This process continues until the position map is found on-chip at the third level. PosMap^2-0 PE accesses this on-chip mapping and progresses to the LM phase. While this request progresses, other requests are still stalled at the CP phase. At t = 1, PosMap^2-0 PE finishes the ER phase. This signals the completion of the writing phase of PosMap^2, prompting the transmission of a dependency clear signal (red arrow) to the PE to the east, instructing PosMap^2-1 to execute LM. As the Palermo protocol ensures that the subsequent execution of PosMap^2-0 and PosMap^2-1 will not access the same address, these two PEs can operate concurrently. At t = 2, PosMap^2-0 completes RP, transmitting the CP request response to the requester PosMap^1-0, while PosMap^2-1 finishes ER, unlocking PosMap^2-2 to execute LM. Note that for PE of request-id being multiples of A, the ORAM tree write phase is complete only when EP is complete.

The execution frontier moves in a waveform and unlocks a massive level of parallelism across different requests with minimal dependencies. At the column level, multiple requests can issue the time-consuming stage ReadPath() concurrently. Upon issuing the ORAM tree modification of the current request to the memory controller, the neighboring PE in the east can promptly start issuing DRAM requests without waiting for the modification to complete. This uplifts the

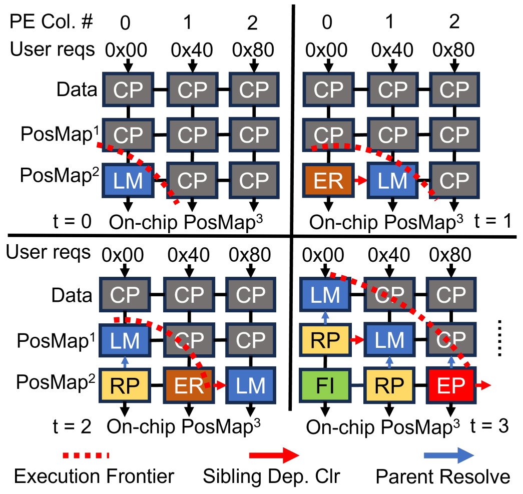
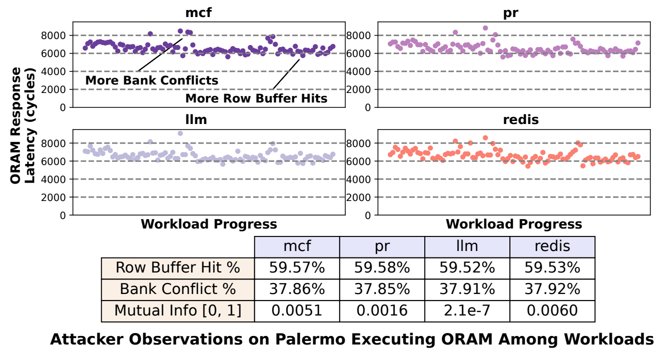


Fig. 8. **Palermo architecture walkthrough example.**



	mcf	pr	llm	redis
Row Buffer Hit %	59.57%	59.58%	59.52%	59.53%
Bank Conflict %	37.86%	37.85%	37.91%	37.92%
Mutual Info [0, 1]	0.0051	0.0016	2.1e-7	0.0060

Fig. 9. **ORAM response latency in Palermo. The variations are due to the leaf selections, memory controller, and DRAM architecture, all of which are public information. Quantitatively, the mutual information in all workloads is shown in the table. The close to 0 result indicates the attacker's knowledge gain is no better than random in the original program by observing the DRAM timings. Qualitatively, this is expected because all DRAM timings are only determined by the statistically random leaf sequences, thus fully decoupled with any program behavior.**

memory bandwidth utilization rate significantly. At the row level, different sub-ORAMs can issue requests at the same time. When the ORAM tree has completed the `ReadPath()` phase, the parent request can immediately start to issue.

C. *Palermo Integration with Prefetching*

Inspired by prior works [8], [39], [50], Palermo can also support prefetching to further improve performance. To enable prefetching of contiguous data blocks, Palermo maps multiple data blocks/cache lines to a single `Data` block in the ORAM tree. This is achieved seamlessly upon address translation from `Data` ORAM tree block accesses in `RP` phase to a sequence of *prefetch length* DRAM accesses, while `PosMap`^{1} and `PosMap`^{2} ORAM tree accesses remain unchanged. As shown in Fig. 12, Palermo prefetch scheme does not increase the `Stash` pressure as opposed to PrORAM and thus eliminates the necessity for injecting any dummy ORAM requests. Prefetching does not change the Palermo protocol and only requires a wider `Stash` design to accommodate wider data blocks in hardware. Notably, prefetching is not a necessary design choice, but can be optionally used with Palermo based on ideas presented in prior works [8], [39], [50]. This feature is configured by the user before execution and is fully decoupled with the underlying protected workload behavior.

D. *Discussion: Comparison with Software-based Parallel ORAM Protocols*

Software-based ORAM optimizations [6], [45], [49] have been proposed to query untrusted databases. To unlock parallel transaction processing for multiple clients, ConcurORAM issues multiple ORAM requests one after the other with "order-based synchronization" without waiting for their evictions to be committed. All evictions are tracked in an append-based log and are processed in the background.

Crucially, ConcurORAM is a software library that adopts the ORAM protocol algorithm to access an external untrusted database securely. While ConcurORAM achieves memory saturation, it refers to achieving peak performance in software. In contrast, Palermo designs a novel ORAM protocol and an underlying hardware implementation to access an untrusted RAM efficiently. In fact, this co-design unlocks memory-level parallelism at the memory controller to reduce execution bubbles and improve DRAM bandwidth utilization.

E. *Discussion: Optimizing RingORAM over PathORAM*

RingORAM presents major design improvements over PathORAM to reduce DRAM traffic. Despite the 42% less traffic, RingORAM marginally outperforms PathORAM by 10% mainly due to the blocking issue nature and frequent dependency stalls presented in §III-A. Palermo finds the unique opportunity in RingORAM to respect the minimal dependency at a hardware level and exploit higher memory-level parallelism. PathORAM, on the other hand, does not have RingORAM's access exclusivity guarantee that subsequent accesses always select distinct blocks in a node if available (§IV-B). Thus, a similar strategy in PathORAM gains limited performance benefits because PathORAM protocol in its nature has much higher total DRAM traffic with fewer dependency bubbles, leaving little room for improvement at the memory controller. §VIII-A presents the detailed performance analysis of all studied baselines based on the two protocols.

VI. Security Analysis of Palermo

The following qualitative and quantitative analyses show that Palermo design upholds the same security guarantees as the RingORAM protocol.

Qualitative analysis. We first qualitatively show that Palermo's access traces are indistinguishable. In Palermo, the ORAM controller issues LLC misses at a constant rate and pads dummy ORAM requests when LLC issues none, same as [41]. Each ORAM request `PA` in Palermo protected memory space is mapped to an ORAM leaf. Upon accessing this leaf as it becomes visible on the DRAM memory bus, Palermo immediately re-maps the `PA` to another random ORAM leaf without revealing it, following the remapping scheme from RingORAM. Thus, an attacker monitoring the memory traffic that executes DRAM bus attacks can only infer the underlying stream of statistically random leaf selections at a constant rate and the statically configured *prefetch length*, both of which are uncorrelated with the original program behavior. In conclusion, Palermo follows the same level of security guarantee as RingORAM in hiding the memory access patterns.

Quantitative analysis. Palermo's distinct characteristic from other ORAM optimizations is that it overlaps multiple ORAM requests. Palermo supports overlapping ORAM requests rooted from LLC misses issued by different processes (such as concurrent KV access in redis or token feature table accesses in llm among different users) for better resource availability in the cloud settings. While overlapping ORAM requests incur sharing of the ORAM controller and may interfere with other requests, we quantitatively show that Palermo by design ensures isolation of service latencies at the ORAM controller of each LLC miss. Fig. 9 shows the analysis of ORAM response latencies for four workloads - mcf, pr, llm, and redis. These figures clearly show that the access latencies are closely clustered together. This is because ORAM requests are issued at a constant rate and once an ORAM request is issued at the ORAM controller, it always attempts to issue DRAM requests at its earliest time of resolving protocol dependencies and contend the memory controller with a fixed number of ORAM requests before its issue. Thus, the response latency of any ORAM request is a deterministic function of a constant number of uniformly random leaf selections before issue, leading to fully decoupled behavior of LLC misses, regardless of which process launched it.

Note that all the access latencies cannot be identical due to the hardware architecture of memory controllers and DRAM. However, the variance due to this does not render a design insecure as it is based only on the *public information*, including the memory controller design, DRAM access protocol, and PA to ORAM leaf mapping (which is decoupled from PA as this is remapped randomly at each access). For example, modern memory controllers reorder requests to exploit bank-level parallelism and row-buffer locality in DRAM. This always leads to variations in DRAM response timings, and hence ORAM response timings. Palermo design does not affect this behavior. For the evaluated workloads, about 59% of DRAM requests in Palermo access an already open row leading to lower latency, versus 37% of accesses that require precharging a row and activating a new row leading to higher latency. The similarity in these statistics for different workloads is owing to the ORAM protocol (similar to Fig. 3(a)).

$$\begin{aligned} \mathrm{M} \equiv & \frac{p_1}{2} log_2 \frac{2p_1}{p_1+p_2} + \frac{p_2}{2} log_2 \frac{2p_2}{p_1+p_2} \\ & + \frac{1-p_1}{2} log_2 \frac{2(1-p_1)}{2-p_1-p_2} + \frac{1-p_2}{2} log_2 \frac{2(1-p_2)}{2-p_1-p_2} \end{aligned} \tag{1}$$

TABLE I
PROBABILITIES OF DIFFERENT VICTIM BEHAVIORS B AND ATTACKER OBSERVATIONS O.

		Attacker's observation O	
		Longer timing	Shorter timing
Victim's behavior B	Requested block is in the Stash	p_1	1 - p_1
	Requested block is in the ORAM Tree	p_2	1 - p_2

TABLE II
REAL-WORLD SERVICES THAT DEMAND OBLIVIOUSNESS.

Category	Workload Name	Description
SPEC17	mcf (mcf)	Route planning computation [3]
	lbm (lbm)	Fluid dynamics computation [3]
Graph	PageRank (pr)	Score ranking [2] on Livejournal [1]
	Motif Mining (motif)	Graph mining [31] on Wikipedia [26]
DL	DLRM(MemBound) (rm1)	Meta RM [18], [19] on Criteo 1T [22]
	DLRM(Balanced) (rm2)	Alibaba RM [54] on DBLP [44]
	Large Lang. Model (llm)	GPT-2 [37] on OpenORCA [27]
KV	Redis (redis)	Redis KV access [23]
	Streaming (stm)	Streaming memory access
	Random (rand)	Random memory access

We further use mutual information M [11], [17] between victim behavior B and attacker observation O to quantify the amount of information about the response latency that an attacker can gain. From the potential attacker's sample of its access latency, we model the attacker's best guess about whether the victim's behavior hits the attacker's past access address is based on whether the access latency is lower/higher than the median observed latency. Using Table. I, Equation 1, and experimental measurements, the calculated mutual information M in these workloads is close to 0 (see table in Fig. 9). M close to 0 indicates that p1, p2 $\approx$ 0.5: the attacker observes longer and shorter than median timings with almost 50-50 probabilities. Therefore, the potential attacker cannot extract any information gain about the private program behavior out of Palermo.

VII. EVALUATION METHODOLOGY

A. Real-World Cloud Services

We use a variety of cloud services for evaluation as shown in Table II. The goal of ORAM is to hide the user's sensitive access patterns, such as accessed node IDs in graphs, item embedding IDs in deep-learning-based recommendation models (DLRMs), and token IDs in large language models (LLMs). We measure up to 50M ORAM requests in the protected memory space and use the first half of the execution as a warmup, which translates into more than 3B memory instructions to the untrusted cloud memory.

B. State-of-the-art ORAM Baselines

PathORAM [46] is a widely adopted protocol that achieves low algorithmic complexity and small on-chip requirements.
RingORAM [42] builds on top of PathORAM with significant bandwidth improvement.
PageORAM [38] uses PathORAM as the base protocol and introduces *sibling node* accesses. Sibling node accesses can

TABLE III
PALERMO SYSTEM CONFIGURATION.

System Component	Modeled Parameters
Host processor	32-OoO cores, 4-wide issue, 2.66GHz frequency, 128-entry ROB, 3-level inclusive cache hierarchy
L1 cache	32KB private per core, 4-way associative
L2 cache	256KB private per core, 8-way associative
L3 cache	8MB shared cache, 16-way associative
Protected memory space	16GB user data protected
Tree-top caches [30]	24× banks of 32 KB scratchpad (3×256 KB total)
PosMap^3	16× banks of 1 MB EDRAM (16 MB total)
Stash^0, Stash^1, Stash^2	3× cache bank of 16 KB SRAM cache (48 KB total)
PE layout	3 (row) × 8 (column) PEs, 1.6GHz frequency
Outsourced DRAM	4-channel DDR4-3200, 102.4 GB/s peak bandwidth

expand the options for a block's residence and capitalize on the row buffer locality associated with accessing the nodes in PathORAM. The size of tree buckets can be reduced in this way, thereby reducing the overhead of each ORAM access.

IR-ORAM [41] uses hardware to keep track of the tree-top cache blocks' `PosMap` mappings. When IR-ORAM detects the accessed block hits in the tree-top cache, the accesses to the recursive `PosMap` ORAM are bypassed. Additionally, IR-ORAM shrinks the bucket size in the middle part of the ORAM tree to reduce the overhead of each ORAM access. IR-ORAM is similar and outperforms an early work Rho [34]. We use IR-ORAM to represent this optimization idea.

PrORAM [50] and its variants [8], [39] use prefetch to eliminate ORAM accesses (see §III-B). The performance is measured after sweeping for best-performing prefetch lengths with Fat-Tree optimizations [39].

C. Simulation Parameters and Infrastructure

Table III shows the modeled system configuration. Similar to prior works, Palermo uses 3 levels of sub-ORAM trees. The protected user memory space is 16GB. We use 256 KB tree-top caches, and 16KB stash for each level of sub-ORAM, the same cache size provision as prior works [41].

To accurately estimate the performance of Palermo, we implement a detailed two-phase simulation methodology. First, we model all hardware components (except caches) using System Verilog HDL. We synthesize this design using a commercial 28 nm technology library using the Synopsys Design Compiler. Using detailed post-synthesis RTL simulations, we extract the critical path delay of our circuits and set Palermo clock frequency at 1.6 GHz. Additionally, we collect the power and area numbers using RTL. We use CACTI [33] to estimate the performance/power/area of SRAM caches. Second, to model DRAM performance, We utilize the widely-used, cycle-accurate Ramulator [24] to model DRAM. We additionally model the cycle-accurate behavior of the Palermo controller at the front end of the Ramulator, interacting with Ramulator memory events. To measure the effect on end-to-end performance, we use Sniper [5]. This simulator faithfully models all system components. We validate the simulator's functionality by comparing its memory traces with Palermo software version to ensure the absence of missed events. Palermo is open-sourced at https://github.com/Linestro/Palermo-ORAM.

VIII. EVALUATION RESULTS

A. Performance Analysis

Palermo versus prior works. Fig. 10 compares the end-to-end performance of Palermo with the state-of-the-art ORAM optimizations. In absolute numbers, Palermo issues and resolves at 3.8E6 LLC misses per second and RingORAM is at 1.7E6 LLC misses per second. The proportion of dummy ORAM requests becomes negligible after the warm-up phase. Given the fully DRAM-bound execution after applying ORAM, the DRAM-bound execution in nature diminishes the need for dummy requests when all benchmarks execute with ORAM. Palermo achieves an average of 3.2× and 2.4× speedup with and without prefetch over PathORAM while RingORAM, PageORAM and IR-ORAM achieve 1.1×, 1.2×, and 1.1×. PrORAM achieves superior performance by avoiding the ORAM protocol for a subset of memory requests when their data is prefetched and is present in LLC. This is further evident by observing a stark performance difference in stm and rand, where PrORAM achieves suboptimal performance for workloads with little to no spatial locality. Palermo-SW shows the 1.2× protocol-level-only speedup over PathORAM with software mutex synchronizing the inter-request parallelism. Palermo shows an additional 2.6× performance brought by the co-designed hardware. The overall design aspects contribute a total of 3.2× to performance gain.

The performance improvements of Palermo are attributed to unlocking massive opportunities for parallelism. The performance gains of Palermo can be further illustrated using the results in Fig. 11. This figure compares the bandwidth utilization and average number of outstanding DRAM requests in the memory controller. For better illustration, we show Palermo without prefetch such that the total DRAM traffic is identical between RingORAM and Palermo. Using Palermo protocol-hardware co-design, multiple requests can be issued to the memory controller as soon as their minimal dependencies are met. Therefore, the average number of outstanding DRAM requests in Palermo increases by 2.8×, which translates to a 2.2× improvement in the memory bandwidth utilization.

To compare the prefetch effect, we sweep the best performing prefetch length for PrORAM for each workload, and apply the same prefetch length for Palermo. This ensures the LLC miss traffics are the same after prefetch filtering between PrORAM and Palermo for a fair comparison. Palermo outperforms PrORAM by 1.9× due to better memory-level parallelism and no background eviction incurred by `Stash` pressure.

Note that the bandwidth utilization remains consistent across various workloads. Our chosen workloads represent a wide range of popular applications and exhibit diverse memory behaviors in the absence of ORAM. The uniformity in bandwidth across these workloads stems from applying the ORAM protocol. Each memory address is mapped to a random leaf on the ORAM tree on every access. This consistent behavior on the memory bus illustrates how ORAM effectively obscures memory access patterns from potential attackers.

Bounded stash size in Palermo. Stash overflows lead to

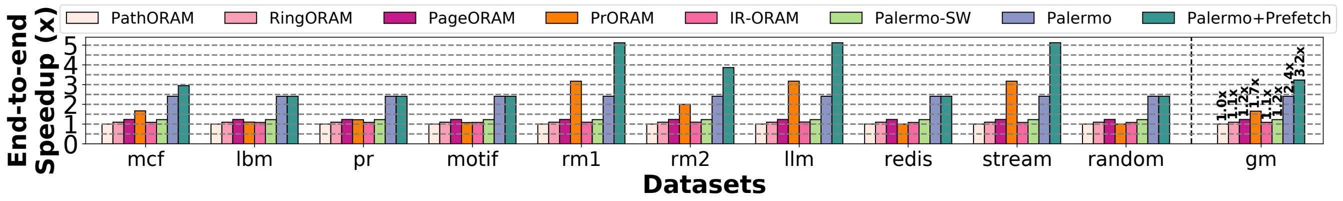


Fig. 10. **End-to-end performance improvements while executing a variety of cloud service benchmarks with RingORAM [42], PageORAM [38], PrORAM [50], IR-ORAM [41], Palermo software only version, and Palermo normalized to PathORAM performance. Higher is better. PrORAM performance is measured after sweeping for the best-performing prefetch length with Fat-Tree optimizations. Palermo+Prefetch always applies the same prefetch length as PrORAM selects in each workload such that LLC miss traffics are the same.**

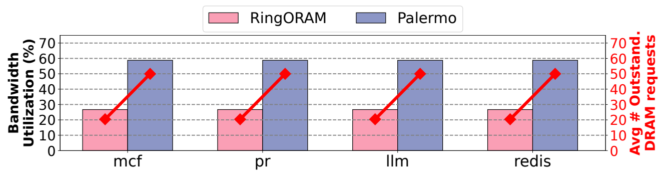


Fig. 11. **DRAM bandwidth utilization comparison and average number of outstanding DRAM requests in RingORAM and Palermo without prefetch optimization. Palermo architecture improves parallelism and enqueues 2.8× more outstanding requests on average in the memory controller, resulting in a 2.2× higher DRAM bandwidth utilization.**

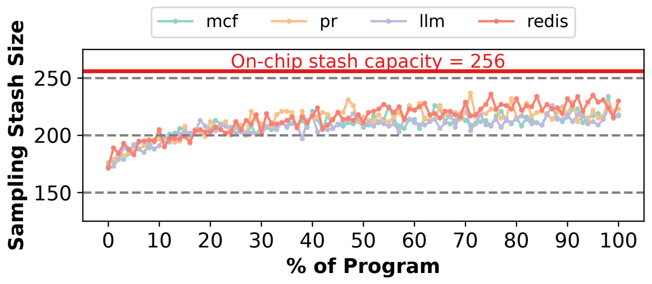


Fig. 12. **Palermo stash utilization over time while executing various workloads. The empirical result shows that even with the introduction of concurrency with Palermo, the stash utilization is bounded.**

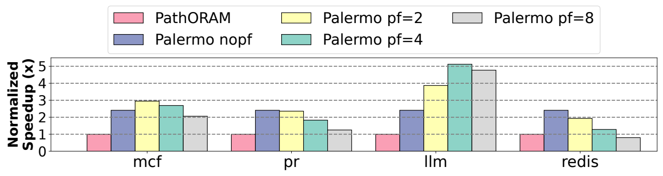


Fig. 13. **Performance sensitivity of Palermo with different prefetch lengths. pf=X refers to converting Palermo `Data` ORAM tree block accesses in `RP` phase to a sequence of X DRAM accesses.**

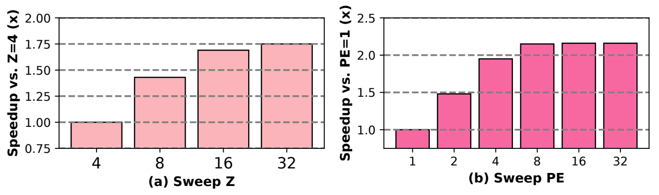


Fig. 14. **Palermo performance sensitivity study with respect to (a) ORAM protocol parameter `Z`, and (b) the number of PE columns in Palermo architecture.**

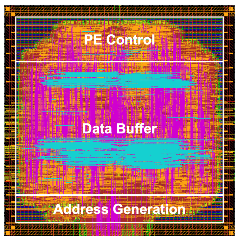
background evictions, which adversely affect the performance of the ORAM protocol. A high-performance on-chip ORAM controller solution must achieve a negligible probability of overflowing even with a small stash (*e.g.*, $< 2^{-103}$ in RingORAM with a 256-size stash). One of the key design goals of Palermo is to improve performance with a bounded on-chip stash and avoid dummy requests. Therefore, we study the occupancy of the stash in Palermo. Palermo protocol synchronizes the `EP` phase after the `RP` phase to provide a theoretical bound of the stash size while concurrently serving requests. Fig. 12 shows the sampled maximum stash size after every 1% progress during the execution of mcf, pr, llm, and redis. For the represented benchmarks, the maximum stash size throughout the program execution is 234, 237, 228, and 236. This empirical result shows evidence that the intelligently designed Palermo protocol and hardware architecture maintain stash boundness and improve performance.

B. *Palermo Performance Analysis with Prefetching*

Fig. 13 shows Palermo performance study of representative workloads with different prefetch lengths. For SPEC, Graph, and KV access workload with moderate locality, Palermo performance only moderately changes and consistently outperforms PathORAM when switching prefetch length from 1 to 4. Thus, Palermo performance is not critically dependent on selecting the best prefetch length. For embedding access workloads, selecting prefetch length closing to the size of an embedding row offers maximized benefits. When configured with prefetch length of X, Palermo `Stash` contains X cachelines for each SRAM entry. Thus, the amount of data in `Stash` grows by the dimension of X. However, the number of `Stash` tags remains bounded and stays below 256, regardless of the prefetch length. This is because Palermo prefetch strategy does not change the protocol algorithm of handling blocks but only magnifies the block size by a factor of X in `Data` ORAM tree.

C. *Sensitivity Analysis*

ORAM Protocol Design Parameters. Palermo offers a rich protocol parameter sweep including the number of real and dummy blocks (`Z`, `S`) per `node` and eviction frequency



Component	mm^2 Area	mW Power
Control Logic (24x)	0.22	25.8
Address Generation (24x)	0.13	38.6
PE data buffer (24x)	1.05	1709
Stash (3x, shared)	0.11	25.6
Treetop Cache (3x, shared)	1.33	240.9
Mesh Network (1x, shared)	0.05	58.1
PosMap^3 (1x, shared)	2.89	42.9
Total	**5.78 mm^2**	**2.14 W**

Fig. 15. **A single PE layout, area, and power analysis of the entire Palermo. Power is measured at 1.6 GHz.**

A. We sweep all valid parameters shown in [4], [42]. Fig. 14(a) shows the memory throughput of different protocol parameters. Interestingly, Palermo exploits more benefits from larger (S, A) because they create fewer write barriers for concurrent serving of multiple ORAM requests. With larger (Z, S, A) Palermo achieves up to 1.8× performance over (4, 5, 3) with the same capacity of protected memory space. Larger A can create a higher pressure on Stash to store more blocks temporarily. Palermo adopts (16, 27, 20) configuration that can accommodate a modest stash size of 256 blocks.

Number of PEs. To demonstrate the performance sensitivity of Palermo architecture, Fig. 14(b) shows the performance of executing rand sweeping PE array size from 3×1 to 3×32. With fewer PEs, the workload is bound by structural hazards, instead of real dependencies between concurrent requests. Adding more PEs enables increased concurrency in issuing ORAM requests. Our evaluation shows that with 3×8 PEs, the workload starts to saturate the memory bandwidth and achieves 2.2× memory throughput over 3×1 PEs.

D. Area and Power Analysis

Fig. 15 shows the layout of a single PE, area, and power estimates for a full Palermo ORAM controller design. The power results include both leakage and dynamic power consumption. The dynamic power is averaged over all workloads. The table shows that Palermo consumes an area of 5.78mm^2 and 2.14W. A majority of this is consumed in on-chip tree-top caches and PE data buffers. A 2D pipeline of PE buffers is the key component of Palermo architecture to unlock memory-level parallelism by issuing concurrent request. The tree-top cache [30] stores the bottom level of the tree, which exhibits the highest access intensity. Therefore, the share of their area and power is justified. In comparison, the past FPGA work [13], [30] Phantom operates at 200MHz and takes more than 20mm^2 using at least 5% of the logic cells in high-performance Virtex-7 FPGAs. Other works optimizing ORAM [39]–[41] do not report area and power numbers. To compare, Palermo offers a high-performance on-chip ORAM controller solution that serves LLC misses obliviously and interacts with DDR memory at 1.6GHz. Palermo takes 5.78mm^2 area in 28nm technology and consumes 2.14W at 1.6GHz operating frequency.

IX. Related Work

Memory-level parallelism in ORAM. To enhance DRAM throughput of the ORAM protocol, prior works [4], [7], [14], [36], [47], [51], [52] implement on-chip and memory controller-level optimizations. They result in sub-optimal speedup because they strictly serve ORAM requests one after the other. Palermo addresses this aspect by proposing an optimized protocol and hardware architecture to overlap multiple ORAM requests and improve performance.

ORAM capacity utilization optimizations. PathORAM uses 50% of capacity to store dummy blocks [46], while RingORAM has an even higher dummy block percentage [42]. AB-ORAM [40] observes that RingORAM has many invalid blocks during the execution that can be recycled. To recycle a block, AB-ORAM allocates blocks at a remote bucket that are marked as invalid. Cao *et al.* [4] applies ”green blocks” that can reduce the block count in each bucket to save memory capacity. Palermo is orthogonal to improving capacity.

Other oblivious approaches. Private Information Retrieval (PIR) [28], [29], [32], [53] is an orthogonal approach to ORAM. PIR keeps data blocks static and hides the target of each query using costly homomorphic encryption.

X. Conclusion

To optimize ORAM performance, we introduced Palermo, a protocol and hardware co-design that enables the concurrent processing of multiple ORAM requests to maximize memory throughput while preserving ORAM correctness and security guarantees. Using a diverse workload mix, we demonstrated that Palermo achieves 2.9× performance on average compared to a RingORAM baseline, with a negligible area overhead of 5.78mm^2 on a CPU, without compromising security.

XI. Artifact Appendix

A. Abstract

This paper presents a protocol-hardware co-design to improve ORAM performance. Palermo introduces a new protocol that overlaps large portions of memory operations, within a single and between multiple ORAM requests, without breaking correctness and security guarantees. Subsequently, we propose an ORAM controller architecture that executes the protocol to service ORAM requests.

This document briefly describes how to reproduce the main result of our paper. Our instructions include 1) how to download the code repository, 2) how to compile and run the script, 3) how to reproduce the Fig. 10 in the paper. Expected result: Generated performance figure showing that Palermo can achieve performance gain by 3.2× and 2.9× over PathORAM [46] and RingORAM [42], respectively.

B. Artifact check-list (meta-information)

- **Program:** `c++` and `python3`
- **Compilation:** `g++ 9.4.0`
- **Dataset:** The evaluated workload LLC miss traces are included within the code repository. The evaluated datasets are: mcf (mcf), lbm (lbm), PageRank (pr), Motif Mining (motif), DLRM(MemBound) (rm1), DLRM(Balanced) (rm2), Large

Lang. Model (llm), Redis (redis), Streaming (stm), and Random (rand). See Table. II for details.

- **Run-time environment:** Implementation should run natively.
- **Hardware:** CPU
- **Execution:** Bash script for automatic compilation and execution.
- **Output:** The reproduced Fig. 10 in the paper.
- **Experiments:** Different ORAM technique performance on executing different workloads.
- **How much disk space required (approximately)?:** 10GB
- **How much time is needed to prepare workflow (approximately)?:** 10 minutes
- **How much time is needed to complete experiments (approximately)?:** 20 hours
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** MIT License

C. Description

1) How to access?: The artifact code base can be downloaded from https://github.com/Linestro/Palermo-ORAM/. The README file in the root directory of the Palermo repository contains instructions to reproduce Fig. 10.

2) Hardware dependencies: any commodity CPU should be adequate for running the code implementation.

3) Software dependencies: we use python3.9 and g++ 9.4.0 on Ubuntu 20.04.5 LTS (GNU/Linux 5.4.0-135-generic x86_64). It is possible to run on a different Linux distribution with other python3 versions and g++ versions, but we did not test on other environments.

4) Datasets: We use real-world workloads for evaluation. The applications are run through Sniper [5] to generate LLC miss traces.

D. Installation

Download the Palermo code base from https://github.com/Linestro/Palermo-ORAM/.

E. Experiment workflow

Due to the large number of commands, please refer to `Palermo-ORAM/README.md` for the commands for each step to run.

Step 1: Generate Palermo main results (6 hours).
Step 2: Generate Palermo software only results (2 hours)
Step 3: Generate RingORAM results (2 hours)
Step 4: Generate PrORAM results (6 hours)
Step 5: Generate other baselines (4 hours)
Step 6: Plot results in the paper (1 minute)

F. Evaluation and expected results

After the scripts have been completed running, the raw results are in `*_results/`. The reproduced figure is in `Palermo-ORAM/fig/`, and it should be the same as the Fig. 10 in the paper.

References

[1] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. 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*, 2006, pp. 44–54.

[2] S. Beamer, K. Asanović, and D. Patterson, “The gap benchmark suite,” *arXiv preprint arXiv:1508.03619*, 2015.

[3] J. Bucek, K.-D. Lange, and J. v. Kistowski, “Spec cpu2017: Next-generation compute benchmark,” in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, 2018, pp. 41–42.

[4] D. Cao, M. Zhang, H. Lu, X. Ye, D. Fan, Y. Che, and R. Wang, “Streamline ring oram accesses through spatial and temporal optimization,” in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 14–25.

[5] T. E. Carlson, W. Heirman, and L. Eeckhout, “Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation,” in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 1–12.

[6] A. Chakraborti and R. Sion, “Concuroram: High-throughput stateless parallel multi-client oram,” *arXiv preprint arXiv:1811.04366*, 2018.

[7] Y. Che, Y. Hong, and R. Wang, “Imbalance-aware scheduler for fast and secure ring oram data retrieval,” in *2019 IEEE 37th International Conference on Computer Design (ICCD)*. IEEE, 2019, pp. 604–612.

[8] Y. Che and R. Wang, “Multi-range supported oblivious ram for efficient block data retrieval,” in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 369–382.

[9] K.-M. Chung, Z. Liu, R. Pass *et al.*, “Statistically-secure oram with õ ($\log^2$ n) overhead,” 2014.

[10] G. Connell, “Technology deep dive: Building a faster oram layer for enclaves,” https://signal.org/blog/building-faster-oram/, 2022.

[11] S. Deng, W. Xiong, and J. Szefer, “Secure tlbs,” in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 346–359.

[12] 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*, 2015, pp. 103–116.

[13] C. W. Fletcher, L. Ren, A. Kwon, M. Van Dijk, E. Stefanov, D. Serpanos, and S. Devadas, “A low-latency, low-area hardware oblivious ram controller,” in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2015, pp. 215–222.

[14] N. Fujieda, R. Yamauchi, and S. Ichikawa, “Last path caching: A simple way to remove redundant memory accesses of path oram,” in *2016 Fourth International Symposium on Computing and Networking (CANDAR)*. IEEE, 2016, pp. 347–353.

[15] C. Gentry, K. A. Goldman, S. Halevi, C. Julta, M. Raykova, and D. Wichs, “Optimizing oram and using it efficiently for secure computation,” in *Privacy Enhancing Technologies: 13th International Symposium, PETS 2013, Bloomington, IN, USA, July 10-12, 2013. Proceedings 13*. Springer, 2013, pp. 1–18.

[16] 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.

[17] A. J. Goldsmith and P. P. Varaiya, “Capacity of fading channels with channel side information,” *IEEE transactions on information theory*, vol. 43, no. 6, pp. 1986–1992, 1997.

[18] U. Gupta, S. Hsia, V. Saraph, X. Wang, B. Reagen, G.-Y. Wei, H.-H. S. Lee, D. Brooks, and C.-J. Wu, “Deeprecsys: A system for optimizing end-to-end at-scale neural recommendation inference,” in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 982–995.

[19] U. Gupta, X. Wang, M. Naumov, C. Wu, B. Reagen, D. Brooks, B. Cottel, K. M. Hazelwood, B. Jia, H. S. Lee, A. Malevich, D. Mudigere, M. Smelyanskiy, L. Xiong, and X. Zhang, “The architectural implications of facebook’s dnn-based personalized recommendation,” *CoRR*, vol. abs/1906.03109, 2019. [Online]. Available: http://arxiv.org/abs/1906.03109

[20] M. Gurman, “Samsung bans staff’s ai use after spotting chatgpt data leak,” https://www.bloomberg.com/news/articles/2023-05-

02/samsung-bans-chatgpt-and-other-generative-ai-use-by-staff-after-leak#xj4y7vzkg, 2023.

[21] T. Hoang, C. D. Ozkaptan, A. A. Yavuz, J. Guajardo, and T. Nguyen, "S3oram: A computation-efficient and constant client bandwidth blowup oram with shamir secret sharing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 491–505.

[22] Kaggle, "Display advertising challenge," https://www.kaggle.com/c/criteo-display-ad-challenge, 2014.

[23] Kaggle, "Amazon us customer reviews dataset," https://www.kaggle.com/datasets/cynthiarempel/amazon-us-customer-reviews-dataset/data?select=amazon_reviews_multilingual_US_v1_00.tsv, 2021.

[24] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," *IEEE Comput. Archit. Lett.*, vol. 15, no. 1, p. 45–49, Jan. 2016. [Online]. Available: https://doi.org/10.1109/LCA.2015.2414456

[25] E. Kushilevitz, S. Lu, and R. Ostrovsky, "On the (in) security of hash-based oblivious ram and a new balancing scheme," in *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*. SIAM, 2012, pp. 143–156.

[26] J. Leskovec, D. Huttenlocher, and J. Kleinberg, "Signed networks in social media," in *Proceedings of the SIGCHI conference on human factors in computing systems*, 2010, pp. 1361–1370.

[27] W. Lian, B. Goodson, E. Pentland, A. Cook, C. Vong, and "Teknium", "Openorca: An open dataset of gpt augmented flan reasoning traces," https://https://huggingface.co/Open-Orca/OpenOrca, 2023.

[28] J. Lin, L. Liang, Z. Qu, I. Ahmad, L. Liu, F. Tu, T. Gupta, Y. Ding, and Y. Xie, "Inspire: in-s torage p rivate i nformation re trieval via protocol and architecture co-design," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 102–115.

[29] H. Lipmaa and B. Zhang, "Two new efficient pir-writing protocols," in *Applied Cryptography and Network Security: 8th International Conference, ACNS 2010, Beijing, China, June 22-25, 2010. Proceedings 8*. Springer, 2010, pp. 438–455.

[30] 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.

[31] P. Mackey, K. Porterfield, E. Fitzhenry, S. Choudhury, and G. Chin, "A chronological edge-driven approach to temporal subgraph isomorphism," in *2018 IEEE international conference on big data (big data)*. IEEE, 2018, pp. 3972–3979.

[32] T. Mayberry, E.-O. Blass, and A. H. Chan, "Efficient private file retrieval by combining oram and pir," *Cryptology ePrint Archive*, 2013.

[33] N. Muralimanohar *et al.*, "Cacti 6.0: A tool to understand large caches," in *HP laboratories*, 2009.

[34] C. Nagarajan, A. Shafiee, R. Balasubramonian, and M. Tiwari, "ρ: Relaxed hierarchical oram," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 659–671.

[35] J. Novet, "Microsoft briefly restricted employee access to openai's chatgpt, citing security concerns," https://www.cnbc.com/2023/11/09/microsoft-restricts-employee-access-to-openais-chatgpt.html, 2023.

[36] H. Omar, S. K. Haider, L. Ren, M. Van Dijk, and O. Khan, "Breaking the oblivious-ram bandwidth wall," in *2018 IEEE 36th International Conference on Computer Design (ICCD)*. IEEE, 2018, pp. 115–122.

[37] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[38] R. Rajat, Y. Wang, and M. Annavaram, "Pageoram: An efficient dram page aware oram strategy," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2022, pp. 91–107.

[39] R. Rajat, Y. Wang, and M. Annavaram, "Laoram: A look ahead oram architecture for training large embedding tables," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–15.

[40] M. Raoufi, J. Yang, X. Tang, and Y. Zhang, "Ab-oram: Constructing adjustable buckets for space reduction in ring oram," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 361–373.

[41] M. Raoufi, Y. Zhang, and J. Yang, "Ir-oram: Path access type based memory intensity reduction for path-oram," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2022, pp. 360–372.

[42] 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)*, 2015, pp. 415–430.

[43] L. Ren, X. Yu, C. W. Fletcher, M. Van Dijk, and S. Devadas, "Design space exploration and optimization of path oblivious ram in secure processors," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013, pp. 571–582.

[44] R. Rossi and N. Ahmed, "The network data repository with interactive graph analytics and visualization," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 29, no. 1, 2015.

[45] C. Sahin, V. Zakhary, A. El Abbadi, H. Lin, and S. Tessaro, "Taostore: Overcoming asynchronicity in oblivious data storage," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 198–217.

[46] 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," *Journal of the ACM (JACM)*, vol. 65, no. 4, pp. 1–26, 2018.

[47] R. Wang, Y. Zhang, and J. Yang, "Cooperative path-oram for effective memory bandwidth sharing in server settings," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 325–336.

[48] WikiChip, "Alder lake - microarchitectures - intel," https://en.wikichip.org/wiki/intel/microarchitectures/alder_lake, 2023.

[49] P. Williams, R. Sion, and A. Tomescu, "Privatefs: A parallel oblivious file system," in *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012, pp. 977–988.

[50] X. Yu, S. K. Haider, L. Ren, C. Fletcher, A. Kwon, M. Van Dijk, and S. Devadas, "Proram: dynamic prefetcher for oblivious ram," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 616–628.

[51] X. Zhang, G. Sun, P. Xie, C. Zhang, Y. Liu, L. Wei, Q. Xu, and C. J. Xue, "Shadow block: Accelerating oram accesses with data duplication," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 961–973.

[52] 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 *Proceedings of the 48th International Symposium on Microarchitecture*, 2015, pp. 102–114.

[53] H. Zhong, C. Hua, R. Shi, and W. Li, "Pir-writing protocol that supports information share," in *2012 8th International Conference on Wireless Communications, Networking and Mobile Computing*. IEEE, 2012, pp. 1–5.

[54] G. Zhou, X. Zhu, C. Song, Y. Fan, H. Zhu, X. Ma, Y. Yan, J. Jin, H. Li, and K. Gai, "Deep interest network for click-through rate prediction," in *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, 2018, pp. 1059–1068.