SONIC: Concurrent Oblivious RAM & Data Structures for Low-Latency and High-Throughput

Nihal Talur
UC Santa Cruz

Ioannis Demertzis
UC Santa Cruz

Abstract

Relying solely on encryption for privacy-preserving computations is prone to leakage-abuse/access-pattern attacks. TEEs, while cost-effective, are also vulnerable to side-channel attacks. Oblivious primitives, such as oblivious memory (ORAM) and data structures (ODS), are effective building blocks to mitigate these risks by concealing memory access patterns and side-channel information. Applications range from private contact discovery (Signal) to anonymous key transparency, encrypted email search, encrypted/oblivious databases, anonymous communication (Sparta/SP'25), private federated learning, LLM privacy (Compass/OSDI'25), and broader confidential computing efforts.

Tree-based ORAMs (EnigMap (USENIX'23), GraphOS (PVLDB'23), Oblix (SP'18)) offer low latency but limited parallelism. Partition-based solutions like Snoopy (SOSP'21) shard data across subORAMs (which build oblivious hashtables on incoming requests, then linearly scan them), achieving high throughput by trading off latency, theoretically enabling linear scalability. In practice, Snoopy's performance hinges on how quickly each subORAM can build the oblivious hashtable and complete its linear scan before exceeding latency targets, constraining server utilization and throughput. While supporting a TB-scale dataset with Snoopy is theoretically feasible, we estimate it would require 1000+ servers.

In this work, we reconcile the fractured landscape between low-latency and high-throughput ORAM designs. We introduce **SONIC**: the first **parallel/concurrent doubly-oblivious tree-based ORAM** for TEEs. SONIC achieves **156K-3.3M req/s** with a single server, tackling the core challenges of all tree-ORAM constructions: overcoming the sequential eviction bottleneck, enabling efficient batch evictions, and providing lock-free access/reshuffle/stash operations. SONIC achieves throughput **29-104**× higher than EnigMap, and **158-560**× higher than GraphOS, with lower latency. In the distributed, high-throughput setting, our SONIC-powered OMAP **PMCHAIN** can replace Snoopy's subORAM, supporting higher throughput and 64× larger datasets using the same hardware (reducing Snoopy's server requirements).

1 Introduction

Organizations increasingly rely on Trusted Execution Environments (TEEs) for private computation in the cloud [1]; however, end-to-end encryption alone cannot fully protect against leakage-abuse attacks [5, 14, 28, 38, 43–45, 48, 54, 55, 57–60, 62, 66, 69, 78, 91, 97] and side-channel attacks [56, 65, 90].

Applications. Signal [3] offers a service called *Private Contact Discovery* that employs ORAM/ODS, allowing users and the Signal server to learn only which of the user's contacts also use Signal. Recent work by Zhu et al. [99] introduces Compass, a semantic search system that leverages ORAMs to create privacy-preserving RAG databases and ensure user privacy. Fredrickson et al. [37] show how ORAM/ODS facilitate anonymous communication, while GraphOS [17] uses ORAM/ODS to execute graph queries on graph databases. SEAL [31] demonstrates how ORAM/ODS can be adapted to database queries with tunable leakage for improved efficiency, and Chang et al. [21] leverage ORAMs to perform equi-join queries. For blockchains such as Ethereum [36], ORAM/ODS can be used for privacy-preserving RPC nodes [67], mitigating surveillance [63, 92] and financial exploitation [26] attacks. ORAM/ODS can also be used for Key Management Systems, Google's Key Transparency [53], private federated computation (e.g., Apple's Private Cloud Compute) [6], TEE-based approaches, such as Google's Privacy Sandbox, Protected Audience [40], Titan Security Key [4], Asylo [77], and a wide range of confidential computing initiatives.

Prior State-of-the-art (Doubly) ORAM/ODS for TEEs. Oblix [71], GraphOS [17], and EnigMap [89] are *doubly-oblivious*, tree-based PATHORAM [87] derivatives designed for hardware enclaves, with a focus on low-latency operations. In trusted execution environments (TEEs), **double-obliviousness** [17, 70, 71, 74, 89] ensures that every aspect of the protocol remains oblivious, including components that were traditionally client-side in a standard client–server model. Oblix introduced a doubly-oblivious ORAM (DORAM) construction and built a doubly-oblivious map (DOMAP) using an oblivious AVL tree. A com-

mon challenge across doubly-oblivious tree-ORAM designs (Oblix [71], GraphOS [17], and EnigMap [89]) is their inherently sequential nature: accesses must lock paths when reading buckets. Additionally, blocks from the stash must periodically be re-randomized and evicted, requiring path locks. Because each PATHORAM access must be followed by an eviction that cannot be parallelized across different accesses, performance/concurrency suffers. Oblix proposed batching evictions [71], but GraphOS and EnigMap found this to be slower than performing single-path evictions [17, 89], where the required oblivious sort is too small to parallelize.

Snoopy [27] is a distributed DORAM/DODS architecture (each server runs in a TEE) that achieves high throughput by addressing the main limitation of tree-based PATHORAM constructions: their lack of concurrency and non-parallelizable eviction. Snoopy sets a target epoch latency such as 1 second, and uses batch processing to maximize per-epoch throughput. Snoopy uses load balancers, which route requests, and subORAMs, which store shards of the dataset. Load balancers obliviously gather, deduplicate, and route batches to the subORAMs. SubORAMs are black-box components that can be implemented with any DOMAP. Snoopy uses an oblivious two-tier hashtable [20]. Every epoch, each subORAM builds an oblivious hashtable over the requests, then parallel scans its stored dataset through hashtable queries. Both build and scan costs can be latency bottlenecks, imposing a hard limit on the per-subORAM dataset size.

In this paper, we pose and address the following question:

*How can we achieve concurrency in doubly-oblivious tree-based ORAMs (**Challenge-1**), effectively address the inherently sequential nature of eviction (**Challenge-2**), and design novel oblivious data structures exploiting concurrency and parallelism in next-generation low-latency and high-throughput ORAMs/ODSs able to scale to TBs of input (**Challenge-3**)?*

Our Contribution. In this work, we propose SONIC[1], which addresses this question by providing a scalable, high-throughput, and low-latency ORAM and ODS. In particular:

• We propose SONIC, the first parallel/concurrent derivative of RINGORAM [79] in the **single-server doubly-oblivious setting**. While all prior works [17, 71, 89] in this model rely on PATHORAM [87], we identify unique opportunities in RINGORAM's bucket structure and eviction schedule to enable concurrency and batch eviction. We propose novel algorithmic modifications to RINGORAM to exploit these opportunities and fully address these challenges.

• To address ***Challenge-1***, we observe that RINGORAM's periodic eviction schedule provides an opportunity for parallelizing accesses. To enable this capability, we: (1) use atomics to enable lock-free bucket reads, (2) make reshuffles *local* and *just-in-time*, and (3) enable lock-free access to the stash. For lock-free bucket reads, we precompute block offsets and index them with an atomic counter, guaranteeing that concurrent bucket accesses touch different blocks. The original RINGORAM protocol locks the tree and stash for reshuffles, offloading blocks from the stash when re-permuting the bucket. However, a careful analysis of their proof [79] reveals that reshuffles do not contribute to the stash analysis; we thus eliminate stash participation in reshuffles, making reshuffles bucket-local. Lastly, our stash allows concurrent reads/writes, eliminating a key point of synchronization. Thus, between evictions, accesses can proceed with high concurrency.

• To address ***Challenge-2***, we leverage RINGORAM's reverse-lexicographic (minimally overlapping) eviction schedule to propose two novel batch eviction techniques. First, we propose a modified *forest layout*: removing the top R levels of the tree, creating 2^R disjoint subtrees. Each subtree is given a stash, expanded to compensate for removed buckets and maintain the stash bound (3.4). Periodic batch evictions route blocks to subtree stashes, and the eviction schedule guarantees exactly one evict path per subtree; all subtrees are evicted in parallel. Second, to further optimize eviction within each subtree, we propose an algorithm to batch evict E paths *within each subtree* with a *single* oblivious sort, compaction, and two linear scans. Our algorithm groups buckets across multiple paths with a single sort, then pushes blocks upward and compacts to fill paths. Finally, we observe that when an application can ensure a disjoint access sequence (such as via Snoopy's deduplication [27] or disjoint ODS traversals [99]), evictions can be performed after access windows rather than interleaved. This is because RINGORAM's access protocol guarantees that disjoint accesses never touch the same real blocks. Deferring evictions improves performance in the high-throughput setting, and enables targeting the offline-maintenance [82] setting (3.2).

• To address ***Challenge-3***, we propose applying SONIC to both low-latency and high-throughput ODSs. In the low-latency setting, existing ODS solutions such as Oblix, GraphOS, and EnigMap [17, 71, 89] have limited potential for concurrency. In contrast, Compass [99] introduces a prefetching technique to retrieve useful data from neighboring nodes. In the high-throughput setting, ODS approaches like the oblivious hashtable [20] used in Snoopy [27] naturally benefit from concurrency; however, their reliance on scanning the entire dataset restricts scalability for large workloads. To address this limitation, we propose **PMCHAIN**, a hybrid construction using an oblivious hashtable for the position map, while SONIC stores the underlying dataset (each subORAM runs a self-contained SONIC instance). This shrinks the linear-scan overhead and makes it block-size-independent, improving dataset scalability. We use PMCHAIN as a drop-in replacement for Snoopy's original subORAM, allowing us to target the **multi-server high-throughput setting**. PMCHAIN delivers higher throughput and supports much larger datasets,

[1] SONIC stands for Scalable, Oblivious, Nimble, Instant, and Concurrent.

significantly reducing hardware costs.

• To ensure a meaningful comparison between our novel SONIC approach and GraphOS/EnigMap, we develop GRAPHMAP by combining the best ideas of both prior systems with all applicable optimizations used in SONIC. This yields GRAPHMAP, a state-of-the-art PATHORAM-based design achieving a 2.7×–10× performance improvement over the original EnigMap and GraphOS implementations [17, 89] respectively. We use GRAPHMAP, alongside existing implementations, as baselines for evaluating SONIC.

• Our approaches establish a **new state-of-the-art** in performance for both low-latency and high-throughput scenarios. In the **low-latency** setting, SONIC achieves throughput improvements of **16-58**× over our *strengthened* GRAPHMAP baseline, **29-104**× over EnigMap's ORAM, and **158-560**× over GraphOS's ORAM (for $N = 2^{27}, B = 64$). In the **high-throughput multi-server** setting, using 1 load balancer and **8 subORAMs** targeting 1s latency: PMCHAIN subORAMs scale to a **256GB** total dataset, while Snoopy's two-tier hashtable subORAMs cannot scale beyond 4GB. We estimate that to reach **1TB**, Snoopy's subORAM would require >1000 servers, while PMCHAIN would require 32.

2 Preliminaries

Obliviousness (Informally). A RAM, data structure, or algorithm is oblivious if and only if for any two sequences of operations of the same length, its observable behavior (such as access pattern) is computationally indistinguishable.

Oblivious RAM. Oblivious RAM (ORAM) [10, 39, 76, 79, 87, 88] conceals access pattern to an array of data blocks by retrieving multiple blocks per access, making it computationally infeasible for an adversary to determine the target block of each access. For TEE-based ORAMs, both client and server exist on the same machine, with the protocol acting as a way to conceal the client's access pattern to memory blocks.

ORAM notation. We denote with N the number of addressable blocks in the ORAM tree of height L, where each ORAM bucket stores Z real and S dummy blocks. GRAPHMAP uses only N, L, and Z. For SONIC, R is the tree depth where the 2^R disjoint subtrees (3.2) begin. We select base eviction rate A by the RINGORAM security analysis. We batch evict $2^R \cdot E$ paths every $A' = A \cdot 2^R \cdot E$ accesses. The ORAM is configured for T concurrent accesses. We explain parameterization methodology in 3.3 and 5.

PATHORAM. PATHORAM [87] is an ORAM protocol obfuscating client access patterns to server-stored blocks. The server holds a tree of Z-block buckets. The client stores a position map (mapping blocks to leaves) and a stash for unevicted blocks. PATHORAM maintains the *invariant* that blocks always reside along the path from the root to their assigned leaf. For `Access`, the client finds the target's leaf via the position map, reads the entire root-to-leaf path into the stash, and retrieves the block from the updated stash. The block is then remapped to a new random leaf, followed by an `Evict` operation. `Evict` pushes blocks from the stash back to the tree, placing them as deep as possible on their assigned paths. We provide a detailed refresher in Appendix A.1.

RINGORAM. RingORAM [79] maintains the PATHORAM invariant but improves its bandwidth and eviction schedule. Buckets store a permuted array of $Z + S$ real and dummy blocks with metadata describing the permutation. This allows reading exactly one block (the target if present, else a dummy) per bucket during access, reducing bandwidth compared to PATHORAM. An `EarlyReshuffle` procedure periodically re-permutes buckets to maintain dummy blocks. RingORAM uses a reverse-lexicographic eviction schedule ensuring minimally overlapping paths between evictions. This supports an improved stash analysis, evicting every A accesses (with A based on Z), unlike PATHORAM's eviction after each access. We provide a refresher in Appendix A.2.

Path Eviction/Stash Analysis. Both PATHORAM and RINGORAM use the same path eviction protocol. Evicted stash blocks are initially assigned to the deepest (towards leaves) bucket in the evict path where they can reside (maintaining the invariant). This initial assignment disregards buckets' finite capacity Z for real blocks, and we term this an *∞-ORAM assignment* [79, 87]. To correct for finite bucket capacity, a *Z-postprocessing* phase pushes blocks from overfull buckets into buckets towards the root or into the stash, enforcing the bucket capacity for Z real blocks. Security depends on a *stash analysis* that bounds maximum stash size with negligible overflow probability, aligning with the standard security definition for ORAMs [17, 71, 79, 87].

Hardware Enclaves/TEEs. Hardware enclaves, such as Intel's SGX [51] or ARM TrustZone [7], also called Trusted Execution Environments (TEEs), provide secure isolation of code and data to aid in private computation. Enclaves run alongside an untrusted operating system (OS), which can observe the memory access patterns, control flow, and timing of enclave programs. The untrusted OS is unable to read the registers or memory of an enclave program, which are encrypted by the CPU. Enclave programs which exhibit secret-dependent memory access patterns and control flow may leak information exploitable by software side-channel attacks [75].

Oblivious Operations. We use the following oblivious operations: $\overline{OSelect(a,b,cond)}$: Obliviously return a if $cond = 1$, otherwise return b. $\overline{OSwap(a,b,cond)}$: Obliviously swap the data at addresses a and b if $cond = 1$, otherwise leave both locations unmodified. We implement the primitives *OSwap* and *OSelect* using the code from the recent work on scalable oblivious sorting [74] and elaborate in Section 3.3.

Oblivious Sort, Shuffle, and Compaction. We make use of oblivious algorithms to perform sort [8, 47, 74], shuffle [47, 74, 81, 82], and order-preserving compaction [81]. $\overline{OSort(A,cmp)}$: Obliviously sort array A based on comparator function cmp. We implement our oblivious sorting using

the implementation of the recent work on scalable oblivious sorting [74]. $\overline{OShuffle(A)}$: Obliviously permute array A. We implement oblivious shuffle by performing oblivious sort with random keys. $\overline{OCompact(A, marks)}$: Obliviously perform order-preserving compaction on A: marked elements are moved to the front of the array with their relative order preserved, and unmarked elements are moved to the end. We implement *OCompact* using the ORCompact algorithm [81].

Double-Obliviousness for TEEs. Mishra et al. [71] introduced the notion of "double-obliviousness" for hardware enclaves. In hardware enclaves, algorithms must run with secret-independent access patterns and control flow to prevent leakage of client state. An algorithm is *doubly-oblivious* if and only if, for any two executions on inputs of the same size, both the memory access patterns and control flow (i.e. the access patterns within the executed client code) are indistinguishable. Numerous prior works operate under the same strict security model [17, 21, 47, 61, 70, 71, 74, 81, 82, 89].

Low-latency DORAMs and DOMAPs. Prior works proposed doubly-oblivious ORAMs (DORAMs) and doubly-oblivious OMAPs (DOMAPs) specifically designed for enclaves, with a primary goal of minimizing operation latency. Oblix [71] pioneered this research area, introducing a low-latency DORAM algorithm, constructing a DOMAP using an oblivious AVL tree [95] over their DORAM. Subsequent works, GraphOS [17] and EnigMap [89], offered distinct approaches to further improve performance; together, they represent the state-of-the-art in low-latency DORAMs and DOMAPs for enclaves. Notably, Oblix, GraphOS, and EnigMap are all doubly-oblivious derivatives of the highly influential PathORAM [87] construction, chosen for its favorable concrete performance characteristics within enclaves.

Significant effort in these works focused on optimizing the eviction process, which both GraphOS and EnigMap observed to be the primary performance bottleneck for doubly-oblivious PathORAM implementations. Oblix originally suggested performing batch evictions, where multiple paths are evicted after a sequence of accesses. However, both GraphOS and EnigMap independently identified the drawbacks of this approach: batch eviction leads to a larger stash, which requires costly oblivious sorting during each eviction step. Consequently, they demonstrated that evicting paths individually significantly outperforms the batch eviction strategy. Beyond this shared observation, GraphOS proposed an asymptotically better doubly-oblivious eviction, improving concrete eviction performance compared to the original Oblix. Both EnigMap and GraphOS also contributed asymptotically better initialization algorithms. EnigMap's [89] investigation into the performance of Oblix-like constructions identified a potential bottleneck when the ORAM tree size exceeds the available Enclave Page Cache (EPC) memory. They highlighted the performance cost associated with page swapping in this scenario. This finding was particularly relevant for SGXv1, given its extremely limited per-socket EPC size of only 128MB [35]. While this constraint is lessened in SGXv2 environments due to a drastically larger 128GB per-socket EPC and lower page swap costs [35], EnigMap's techniques remain applicable for datasets large enough to exceed even this capacity. To address scenarios where the EPC is exceeded, EnigMap proposed a packed memory layout designed to improve the locality of bucket fetches during ORAM accesses. This optimization reduces the required page swaps by a constant factor, reducing the performance impact of paging.

Oblivious two-tier hashtable. We make use of the oblivious two-tier hashtable [20] as a primitive for our high-throughput construction. This oblivious hashtable works by placing items into two levels of bins with two PRFs. Queries are performed by evaluating the PRFs and scanning target bins.

Snoopy. Snoopy [27] is a distributed architecture for a scalable high-throughput OMAP, using a combination of load balancers and subORAMs. Snoopy sacrifices latency to achieve high throughput, using batch processing to serve many requests at once. Load balancers obliviously batch and route requests to subORAMs, which contain shards of the dataset. The subORAMs are implemented with the two-tier hashtable. We provide a refresher in Appendix A.3.

Threat model. We adopt the same threat model as prior works on ORAMs, OMAP, and oblivious primitives for hardware enclaves [17, 21, 27, 70, 71, 74, 81, 82, 89]. We assume an adversary that holds full control over all software on the target machine. Additionally the adversary can observe all memory access patterns and control flow of the enclave program. However, the adversary cannot read the plaintext contents of registers and memory, as they are protected by the enclave's hardware-backed memory encryption. This data is encrypted with the processor's secret key, which is inaccessible to the adversary. In line with prior work, we consider out-of-scope denial-of-service attacks, rollback attacks, and hardware side-channels such as power-analysis or timing attacks [13, 41, 50, 65, 83, 90, 93]. There are complementary techniques [23, 24, 46, 85, 86] that may mitigate such attacks.

Related work. Parallel/concurrent ORAMs have been explored in other models [12, 16, 19, 22, 52, 73, 80, 84, 94], primarily in client-server and trusted-proxy architectures. Sonic is the first parallel/concurrent tree-based ORAM to target the **doubly-oblivious** model for TEEs, where the adversary can observe all access patterns and control flow. This setting requires the entire protocol, including what would traditionally be "client-side" logic, to be fully oblivious.

ConcurORAM [16] targets the client-server setting (server-side obliviousness), where multiple clients issue concurrent queries. These clients are mutually trusted, but do not share runtime state, making inter-client synchronization the primary challenge. ConcurORAM thus uses server-hosted oblivious data structures to store metadata used to serialize/resolve conflicts between concurrent requests, adding some overhead to each access. In contrast, Sonic focuses on synchronizing concurrent accesses from a *single* enclave client while

maintaining double-obliviousness. Because SONIC operates in shared-memory, we use cheap atomics and lock-free algorithms to achieve synchronization with minimal overhead.

Treebeard [84] explores concurrent ORAM scalability in a trusted-proxy/trusted-domain setting. Their system divides servers into a trusted domain, which is fully trusted, and an untrusted domain, where storage access patterns must be oblivious. Treebeard avoids the bottleneck of a single trusted proxy by distributing client/proxy functionality across multiple servers in the trusted domain, achieving scalability and fault tolerance under its trust model. In contrast, SONIC targets the multi-server without the assumption of trusted machines, using the doubly-oblivious Snoopy architecture where servers run in TEEs and network communication is also oblivious.

A concurrent work, H2O2RAM [98] develops a DORAM for TEEs following the design lineage of FutORAMa [11]. H2O2RAM is an amortized construction that performs lookups over a hierarchy of oblivious hashtables, requiring expensive periodic rebuilds (which can reach linear-size).

Other recent works advance the frontier in different system models. In the client-server model, MegaBlocks [9] proposes an I/O and bandwidth efficient hierarchical ORAM for the asymmetric setting with small logical blocks but large physical blocks. Palermo [96] proposes a specialized co-designed hardware controller providing ORAM functionality. BOLT [49] treats high-bandwidth memory (HBM) as an unobservable region to accelerate an OMAP. Both works rely on specialized hardware that differs from commodity TEEs targeted by SONIC, with different security models. BULKOR [64] proposes a new bulk loading algorithm for doubly-oblivious PATHORAM to reduce initialization cost.

A distinct, more theoretical line of work [12, 19, 52] studies Oblivious PRAM (OPRAM), a theoretical abstraction of oblivious parallel computation. In particular, we note that Circuit OPRAM [52] decomposes the tree into subtrees, creating a forest-like structure, similar to that of SONIC. Their construction is a theoretical model of lockstep-synchronized parallel access to non-overlapping paths. In contrast, SONIC supports concurrent access to overlapping paths without lockstep synchronization or coarse locking.

A broader area of work in encrypted-database systems studies private search and query processing techniques across the leakage/performance spectrum, spanning searchable encryption [15, 18, 25, 29, 30, 32–34, 72], relaxed obliviousness [31, 42, 68], and relational database operators [61, 70].

3 Designing a New Concurrent ORAM

Existing state-of-the-art enclave ORAMs such as GraphOS [17] and EnigMap [89] are based on PATHORAM [87], and inherit its limitation of being inherently sequential. This sequential nature comes from PATHORAM's theoretical analysis, which requires an eviction to be performed after each access. We make the observation that RINGORAM [79], while also sequential in its original design, has interesting core properties that are useful as a starting point for building a concurrent ORAM, though this requires substantial redesign and novel algorithmic contributions that we now describe.

Our modifications introduce parallelism for both *accesses and evictions*, directly addressing the critical bottlenecks inherent in prior sequential ORAMs. Crucially, our redesign preserves the stash analysis of the original RINGORAM [79], as we inherit the core eviction schedule and protocol (see 3.4). We transform the access phase to be as *lock-free* and *localized* as possible, and modify eviction to be parallelizable across multiple paths while maintaining equivalent stash analysis. Furthermore, we introduce a novel doubly-oblivious batch eviction algorithm leveraging the reverse-lexicographic schedule to evict many paths at once with a single oblivious sort.

We now illustrate our design process by listing the initial challenges and how we overcome them, resulting in our novel SONIC construction with parallel access and eviction.

Challenge 1: Locks and synchronization. The stash analysis of RINGORAM allows evictions to be performed every A accesses rather than after each access, suggesting the possibility of concurrent access. However, this is not possible in the original protocol, which requires locking target paths and the stash for each access. Without locks, concurrent accesses would conflict in ways that violate obliviousness [16, 80]. To ensure obliviousness, each block in a bucket may be read at most once between reshuffles; two concurrent accesses touching the same block would leak information about the bucket permutation and identity of real blocks. Additionally, both reshuffles and evictions remove blocks from the stash, requiring locks to prevent race conditions (and thus correctness violations). Therefore, enabling concurrent access requires minimizing locks, without violating correctness.

Lock-free offset selection. Our first goal is lock-free block offset selection during bucket accesses. For dummy blocks, we precompute offsets during reshuffle and use an atomic touch counter for lock-free selection of disjoint dummies. For real blocks, we require the application to prevent concurrent requests for the same block (straightforwardly achievable at the application level; see Section 4). Since real blocks are only touched when they match the target address, the same real block will never be touched concurrently. Further details are provided in Section 3.1.

Just-in-time reshuffle. Next, we address path locking during reshuffles, as seen in RINGORAM's `EarlyReshuffle` which attempts to sequentially reshuffle the entire path after each access. Observing that the goal of early reshuffles is ensuring dummy block availability (i.e., buckets touched $< S$ times), we maintain this with less synchronization via *just-in-time* reshuffling: a bucket is reshuffled *immediately* after its S-th touch by its last reader. Reads are lock-free until the atomic counter triggers on the S-th access. At this point, we lock the bucket, perform the reshuffle, then unlock it. The parameter S controls this lock-and-reshuffle frequency.

Localizing reshuffles. We now eliminate stash participation in reshuffles. The RINGORAM `EarlyReshuffle` involves: (1) downloading Z blocks to stash; (2) evicting Z blocks from stash to the bucket; (3) re-permuting. However, the RINGORAM paper explicitly states, and its stash analysis proof confirms, that eviction during reshuffles is not necessary to bound the stash to the stash bound: regularly-scheduled full-path evictions suffice [79]. We thus propose fully *bucket-local* reshuffles: download, re-permute, and upload the Z selected blocks within the bucket, adding fresh dummies. This preserves obliviousness (as we re-permute the bucket) and leaves stash analysis unaffected, while removing stash involvement. Combined with just-in-time reshuffles, this allows concurrent, conflict-free reshuffling of different buckets.

Lock-free stash. Because the `Access` operation reads and writes the global stash, we must ensure that concurrent accesses to it do not conflict, while maintaining double-obliviousness. We accomplish this with an append-only lock-free array with atomic counters. This global stash is flushed and routed to subtrees during evictions (see 3.2).

These modifications enable efficient concurrent access without conflict. This fully maintains the obliviousness of accesses but minimizes synchronization, unlocking concurrent execution of the A accesses *between* evictions. These techniques address the access-phase bottlenecks but highlight the remaining challenge: after every epoch of A accesses, an expensive, sequential eviction must occur.

Challenge 2: Expensive sequential evictions. A key challenge in all tree-based ORAMs is the high cost of eviction. Both PATHORAM and RINGORAM require a nearly-identical core eviction operation; the evict path must be filled with stash blocks, pushing blocks as deep as possible towards their target leaves. RINGORAM evicts paths according to the minimally-overlapping reverse-lexicographic schedule, enabling its tighter stash analysis. The doubly-oblivious path eviction algorithms found in Oblix [71], GraphOS [17], and EnigMap [89] all utilize oblivious sorts to group blocks into target buckets. The first doubly-oblivious approach, in Oblix [71], proposed batching evictions to reduce their impact. However, both GraphOS and EnigMap observed that batch eviction worsened overall latency, and that eagerly evicting single paths at a time yielded better performance.

We first explain why existing techniques are insufficient to alleviate the eviction bottleneck. Because doubly-oblivious eviction algorithms use oblivious sorts, and oblivious sorts are known to be highly parallelizable [74], one might consider parallelizing this sort. However, because the number of elements being sorted (<200) is small, parallelizing the sort provides no improvement, even with the best known approaches for small arrays: (1) bitonic sort (used by [17, 71, 89]); and (2) `ORShuffle` [81] combined with a nonoblivious sort.

Bucket relocation. Our key insight is that batch eviction can be made highly efficient by evicting multiple disjoint paths in parallel. However, in an ORAM tree, paths heavily overlap near the root, complicating multi-path batch eviction. We therefore *relocate* buckets where paths overlap, removing them from their original positions and making a local copy within each descendant subtree. Relocating even a few such levels suffices to unlock parallelism with minimal (<1%) space overhead. This allows paths to be evicted simultaneously *without coordination*, while preserving the semantics of the eviction procedure and maintaining the stash analysis. Blocks that would be evicted to a relocated bucket instead go into its subtree-local copy. Parallelizing evictions allows us to achieve state-of-the-art latency compared to prior approaches.

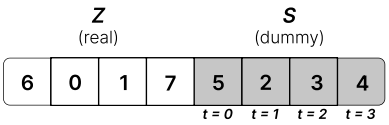


Figure 1: The `offsets` field of bucket metadata. Each access consumes one dummy slot of the bucket, up to S touches, upon which the bucket is reshuffled. The non-secret atomic touch counter t allows concurrent accesses to select distinct slots pointing to unique offsets.

Forest layout. Our first batch eviction technique removes the top R levels of the tree, resulting in a forest layout (see Figure 2) with 2^R subtrees. Each subtree maintains its own local stash, with additional capacity for relocated buckets to maintain the stash bound (see 3.4). Due to RINGORAM's reverse-lexicographic eviction schedule, we can batch evict every $A \cdot 2^R$ accesses, with one evict path per subtree, across all subtrees in parallel. We elaborate in 3.2.

One-sort batch eviction. We develop a novel batch eviction algorithm that efficiently evicts multiple paths at once with a single oblivious sort and compaction. This generalizes single-path eviction to two-dimensional coordinates, and increases eviction buffer sizes to be large enough to parallelize sorts. This can be applied in conjunction with forest eviction, performing batch eviction both across subtrees and within each subtree. We provide further details in 3.2.

Deferred eviction. Some applications can provide a guarantee that a sequence of accesses is *disjoint*, meaning that each is to a distinct target block. Under these conditions, we are free to reorder accesses and evictions arbitrarily (while maintaining the invariant that there must eventually be one eviction per A' accesses). For example, we can perform all accesses in parallel uninterrupted by evictions, then perform all queued evictions at the end. We describe the details in Section 3.2, and provide evaluations in Section 5.

3.1 Parallel Access

Lock-free offset selection. Each bucket contains exactly $Z+S$ blocks, as in RINGORAM. Z of these are *real slots*, which may hold real or dummy blocks, and S are *dummy slots*, which hold dummy blocks. Each bucket stores metadata

about the permutation and identity of its real blocks, and has an atomic counter t for its touch count, as shown in Figure 1. When an access attempts to read a target block from a bucket, `GetBlockOffset` obliviously selects the offset of either the target or a dummy block. If the metadata indicates that the target is in one of the Z real slots, we return that offset; otherwise, we return offsets$[Z+t]$, the offset of the tth dummy block. Since offsets are randomly permuted, and each one is read only a single time, offset selection can be made doubly oblivious using a linear scan and *OSelect* over metadata.

Concurrent path reads. The lock-free bucket structure above allows us to access buckets concurrently while preserving obliviousness. First, we observe that reshuffles are more frequent closer to the root, as more paths pass through those nodes. If a reshuffle is needed at one of these buckets, it may briefly stall concurrent accesses attempting to read an intersecting path. To reduce the impact of these stalls, we read paths from leaf to root, as buckets near the leaf level are reshuffled less frequently. When stalls occur, they impact less of the path, reducing their latency impact. When a bucket's atomic touch counter reaches the maximum value, we perform a lock-reshuffle-release, refreshing the bucket.

Local reshuffles. Every S touches, we must reshuffle a bucket in order to maintain fresh available dummies. First, we identify the Z untouched blocks with an oblivious double-for-loop over metadata: some may be dummies, but they are guaranteed to contain any remaining real blocks. We then directly read these blocks from the bucket; obliviousness is preserved because each of the $Z+S$ blocks is touched *exactly once* and their offsets were randomized by the previous reshuffle. While RingORAM places these blocks into the stash, then draws Z admissible blocks from the stash to evict to the bucket, we observe that this offloading does not contribute to RingORAM's stash bound (see A.2.1). Sonic instead pads the same Z extracted blocks with S fresh dummies to $Z+S$ blocks, then re-permutes and writes them. Our technique preserves the obliviousness of reshuffles while eliminating stash involvement and making them fully bucket-local.

3.2 Parallel Eviction

Forest eviction. We explain the forest batch eviction technique. We define a new parameter R (*routing depth*), and relocate the top R levels of the tree, resulting in a forest structure with 2^R subtrees, as shown in Figure 2. Each subtree maintains its own subtree-local stash, which holds only blocks belonging to this subtree. The subtree stash has space for eviction overflow, plus the ZR relocated capacity compensating for the R ancestor buckets removed for the forest cut. We connect this to RingORAM's original stash analysis in Section 3.4.

After each access, modified, leaf-remapped blocks are temporarily placed in the global stash. During evictions, we fully *flush* the global stash, obliviously routing each block to the unique subtree containing its assigned leaf. Each subtree thread clones the global stash and obliviously compacts to extract only blocks belonging to it, enabling parallel routing without synchronization. The number of routed blocks per subtree can be bounded with a balls-into-bins analysis [27].

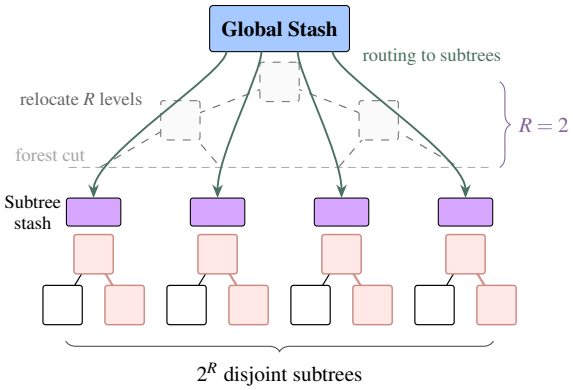


Figure 2: Forest eviction layout.

We evict a batch of 2^R paths at once; due to the reverse-lexicographic schedule, paths are evenly distributed across subtrees. Each subtree independently performs a path eviction, combining blocks from evict paths, blocks routed from the global stash, and blocks already in the subtree-local stash. We illustrate forest eviction in Figure 2.

Replacing one sort with compaction. We propose an improvement to the eviction algorithm from GraphOS [17]. Their eviction algorithm works as follows: (1) Pad the buffer with filler blocks, ensuring that each bucket is filled. (2) Assign blocks to their deepest evictable bucket without regard for capacity. (3) Sort blocks by assigned bucket to group them, with real blocks before filler blocks. (4) Scan from leaf to root and push blocks upwards as buckets are filled, enforcing finite Z-capacity. When all buckets are filled, mark remaining real blocks as excess. (5) Sort blocks again to move all non-excess blocks to the front, arranged as grouped buckets for the output path. We observe that the second oblivious sort can be replaced with a *compaction*, which is both asymptotically and concretely cheaper than sort [81]. We use this optimization for both GraphMap and Sonic. We take inspiration from this improvement to develop a new batch eviction algorithm.

One-sort batch eviction. We introduce a novel doubly-oblivious batch eviction algorithm that generalizes the single-path eviction described above. For this algorithm, we consider eviction within a single tree, which can be either an unmodified RingORAM tree or one of the forest subtrees.

Due to the reverse-lexicographic eviction schedule, 2^i consecutive evict paths within a tree overlap at i levels. To allow evicting multiple paths at once, we can relocate these overlap buckets to a copy for each descendant path segment. We therefore expand each subtree stash with $2^i \cdot i \cdot Z$ additional space. After this modification, there are $E = 2^i$ non-overlapping path segments within each subtree, and each has space in the sub-

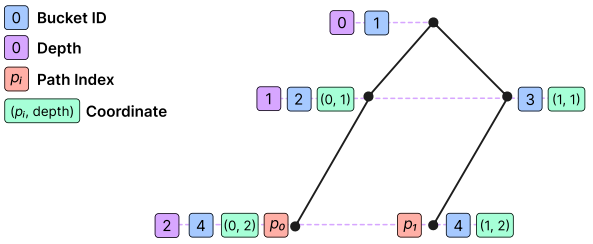


Figure 3: Eviction coordinates for multi-path batch eviction.

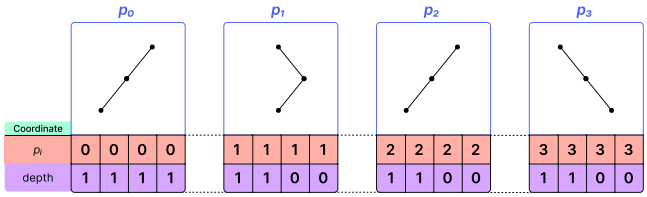


Figure 4: The buffer layout for multi-path batch eviction. Blocks are packed in path-major order, storing their assigned coordinate. Each path segment corresponds to one evict leaf.

tree stash corresponding to its top i overlap region levels.

Our key insight is that blocks can be tagged according to a two-dimensional *coordinate* (Figure 3) that indicates the target path index in addition to the target depth. We can then utilize the same approach of target bucket assignment (using a scan and sort) followed by Z-postprocessing (using a scan and compaction) to evict all the paths. Crucially, the entire algorithm uses only a single sort and compaction in addition to two linear scans. It evicts any $E = 2^i$ paths at once; for $E = 1$, it is equivalent to single-path eviction.

Since all evict paths are non-overlapping, each real block has a *unique* target coordinate for its initial assignment. This coordinate is also never in the overlap region; since all paths below are being evicted, *exactly one* of them is closest to the block's mapped leaf. Note that buckets in the overlap region may be pointed to by multiple coordinates, such as $(0,0)$ and $(1,0)$; blocks assigned here are placed in the corresponding copy of the relocated overlap buckets. We illustrate the layout of these coordinates for a batch eviction in Figure 3. When we sort blocks to group them by buckets, we first order by path index, then by depth, with real before dummy. This results in a sorted buffer that is segmented according to each path being evicted, illustrated in Figure 4. This is an ideal layout that allows us to generalize the Z-postprocessing scan, which resets its state for each path segment. After this scan, we compact the buffer, bringing the output blocks for each path to the front of the buffer again in a path-segmented layout. Output blocks are written to tree paths, and leftover blocks added to the subtree stash. We provide pseudocode in Algorithm 1.

We now explain the multi-segment Z-postprocessing scan that enforces bucket capacity after the initial coordinate assignment. Recall that path eviction pushes all blocks as deep

Algorithm 1: MultiPathEviction(evictLeaves, inBlocks)

▷ Let c be path length and i be overlap height
1 Assign ∞-ORAM coordinates to inBlocks
2 Add $E \cdot c \cdot Z$ filler blocks, assigning Z blocks to every coordinate
3 blocks $\leftarrow$ inBlocks $\cup$ fillerBlocks
4 Sort blocks by path then depth, placing real before filler
5 blocks $\leftarrow$ PostprocessBlocks(blocks)
6 evictedBlocks $\leftarrow$ blocks[$0..E \cdot (c-i) \cdot Z$]
7 leftoverBlocks $\leftarrow$ blocks[$E \cdot (c-i) \cdot Z..$end]
8 **return** evictedBlocks, leftoverBlocks

as possible towards their target leaf. However, as each bucket only has Z real capacity, blocks must be moved upwards towards the root, and eventually the stash, as buckets are filled. The scan begins with blocks assigned to the leaf level of the first path, and keeps track of the fullness of each bucket and the full path as it moves towards the root. Whenever a level becomes full, any real blocks still assigned below it are updated to the deepest remaining available bucket along the path. Filler blocks ensure at least Z blocks are initially assigned to each bucket. When a path becomes completely full, blocks are assigned to the stash. Each time a path boundary is encountered, the scan state is reset for processing the next path. At the end, blocks assigned to the overlap region and other leftover blocks are moved to the expanded subtree stash. The pseudocode is given in Algorithm 2.

Deferred eviction for disjoint sequences. Consider a *disjoint access sequence*, where each access is to a distinct target block. Since each SONIC access reads at most one real block (the target block) from the tree, disjoint accesses will never require the same target block. We can thus temporarily set aside blocks read from the tree to a *deferred buffer*, and postpone evictions according to a schedule optimal for the application. After the disjoint access sequence completes, we can incrementally flush the deferred buffer to the global stash and perform the required number of batch evictions. For example, we can evict according to a *windowed schedule*, breaking an arbitrarily long disjoint access sequence into windows of $k \cdot A'$ accesses. After each window of accesses, we then perform the k delayed evictions. For applications that can tolerate an offline phase, we can operate in a high-throughput access-only online phase, uninterrupted by eviction; evictions can be performed during an offline-maintenance phase [82].

3.3 Putting It All Together

Our modifications to the base RINGORAM design result in SONIC, our novel parallel/concurrent doubly-oblivious

SONIC: Protocol Summary

Parameters. Height L containing N blocks. Buckets have Z real and S dummy blocks. Base eviction rate A from RINGORAM; SONIC evicts every $A' = A \cdot 2^R \cdot E$. Access parallelism T, forest depth R (2^R subtrees), multi-path batch size E.

State. PositionMap[addr]→leaf; GlobalStash (flushed during eviction); SubtreeStash[k] for each subtree k. Each bucket stores secret permutation `offsets[0..Z+S-1]`, metadata (`valids`, `realAddrs`), and atomic `touchCtr`.

Access(addr, op, data′). *(concurrent access)*

1. **Barrier:** If eviction is running, wait for completion.
2. **Remap:** $\ell \leftarrow$ PositionMap[addr]; sample random ℓ' and set PositionMap[addr] $\leftarrow \ell'$.
3. **ReadPath (leaf→root):** for each bucket on path ℓ: Call RESERVETOUCH: wait for bucket to be open, then atomically reserve touch index $t \in [0, S)$. The final touch $S-1$ closes the bucket until reshuffle. Select one offset using GETBLOCKOFFSET: linear scan the metadata; iff a valid match for addr, read the real block, else read dummy `offsets[Z+t]`. If $t = S-1$, RESHUFFLEBUCKET after this read (just-in-time), then reopen fresh epoch.
4. **Retrieve/update:** insert the (real/dummy) fetched block from ReadPath into GlobalStash. Linear scan GlobalStash and SubtreeStash[SubtreeIndex(ℓ)] to obtain target block if dummy was read from path. Apply `op`/`data′`; set `leaf := ℓ′`; append updated block to GlobalStash.
5. **Round end:** Every A' accesses, pause new accesses, wait for in-flight accesses, then run EvictionRound.

EvictionRound. *(forest routing + parallel eviction)*

1. **Scheduled paths:** Compute the next batch of $2^R \cdot E$ scheduled eviction leaves (reverse-lex order) and partition them: each subtree k receives its next E leaves.
2. **Gather blocks:** In parallel for each subtree k: Make a local snapshot of GlobalStash, and obliviously compact blocks belonging to the subtree (mapped to subtree k's leaf range). Union with SubtreeStash[k] and blocks read from the E eviction paths in subtree k (as in reshuffle, obliviously select Z untouched blocks per bucket).
3. **Evict:** Run MULTIPATHEVICTION (Algorithm 1) for subtree k on its E leaves and input buffer, producing Z blocks per bucket on each evicted path and leftover blocks; store leftovers back into SubtreeStash[k].
4. **Write buckets:** Recreate each evicted bucket with its assigned Z blocks and S fresh dummy blocks, arranged in a fresh permutation.

ReshuffleBucket(bucket). Obliviously select exactly Z untouched blocks in the bucket using a double-for-loop over bucket metadata, to select all remaining valid real slots and enough valid dummy blocks. Pad with S fresh dummy blocks, then write them back to the bucket in a fresh permutation and reset `touchCtr := 0`.

PostprocessBlocks. After sorting blocks by (pathIx, depth) with real-before-filler, perform a linear scan to enforce Z capacity, pushing blocks upwards on overflow, then compact to output exactly Z blocks per bucket. See Algorithm 2.

Figure 5: Access and Eviction protocols for SONIC.

ORAM. We provide a full protocol description in Figure 5, and detailed evaluations in Section 5. We now describe the access algorithm which is SONIC's public interface.

Access algorithm. Our `Access` algorithm follows the outline of RINGORAM's original protocol. We allow A' concurrently executed accesses between evictions. Each access reads its target path, looking for a block with the target address. It finds either a real or dummy block along the path, and inserts it into the global stash. It then scans the global stash and relevant subtree stash to find the target block. When an eviction round is required, we temporarily suspend accesses while we run parallelized batch eviction.

Parameterization and performance. As our design introduces several tunable parameters, we now explain their impact on performance and concurrency. Z and S are the number of real and dummy blocks in a bucket, respectively. S controls the frequency of reshuffles. The total size $Z+S$ of the bucket determines the reshuffle cost. Because reshuffles lock buckets, reshuffle frequency and cost must be balanced. The size of the eviction sort is directly proportional to Z. The minimum base eviction rate A satisfying the stash analysis (3.4) increases with Z, allowing expensive evictions to be performed less frequently. T controls the number of parallel in-flight accesses at a time. Note that this is distinct from the eviction parallelism controlled by R. Excessive T can cause bucket lock contention, though optimal T increases with larger N, as shown in Figure 9. At depth R, there are 2^R subtrees, each of which use their own thread during evictions. Increasing R allows more eviction parallelism, but increases subtree stash sizes and required thread count. E (power-of-two) is the num-

Algorithm 2: PostprocessBlocks(blocks)

```
   /* Blocks are grouped by (pathIx, depth),
      with real before dummy                    */
 1 currPathIx ← ⊥
 2 waterlineDepth ← maxDepth
 3 waterlineFullness ← 0
 4 for each block in blocks do
 5     pathIx, depth ← GetEvictCoord(block)
       /* Path change.                          */
 6     isNewPath ← (pathIx ≠ currPathIx)
 7     currPathIx ← OSelect(pathIx, currPathIx,
         isNewPath)
 8     waterlineDepth ← OSelect(maxDepth,
         waterlineDepth, isNewPath)
 9     waterlineFullness ← OSelect(0, waterlineFullness,
         isNewPath)
       /* Placement conditions.                 */
10     pathFull ← (waterlineDepth < 0)
11     hasSpace ← (¬pathFull) ∧ (waterlineFullness < Z)
12     belowWaterline ← (depth > waterlineDepth)
       /* Decide whether to keep.               */
13     keepBlock ← (¬block.isDummy ∧ ¬pathFull) ∨
         (block.isDummy ∧ hasSpace ∧
         ¬belowWaterline)
       /* Update coordinate.                    */
14     newDepth ← OSelect(waterlineDepth, ⊥,
         keepBlock)
15     SetEvictCoord(block, pathIx, newDepth)
16     SetCompactMark(block, keepBlock)
       /* Update waterline.                     */
17     newFullness ← OSelect(waterlineFullness + 1,
         waterlineFullness, keepBlock)
18     bucketFull ← (newFullness = Z)
19     waterlineDepth ← OSelect(waterlineDepth - 1,
         waterlineDepth, bucketFull)
20     waterlineFullness ← OSelect(0, newFullness,
         bucketFull)
21 end
   /* Move marked blocks to the front          */
22 CompactMarkedBlocks(blocks)
23 return blocks
```

ber of paths we evict within a subtree during a single batch eviction operation described in Section 3.2. Increasing E increases eviction cost and subtree stash sizes but reduces eviction frequency as more paths are evicted at once.

Optimized Primitives. We use optimized routines for oblivious *sorting*, *shuffling*, and *compaction* based on the most recent results for each algorithm. For sort, we use the parallel bitonic sort implementation [74] from the distributed oblivious sort paper [74]. For compaction, we implement ORCompact [81]. For shuffle, we compare bitonic-sort-based shuffle with ORShuffle [81], and determine that for small buffers, our optimized bitonic sort provides better performance.

3.4 Security Analysis

Above, we described our doubly-oblivious access and eviction algorithms: using a combination of inherently oblivious constructs (linear scans/double-for-loops) and state-of-the-art oblivious primitives. We now focus on the stash analysis, as stash occupancy must be bounded to ensure security [79, 87].

Parameters. Our parameter selection of A and L exactly follows RingORAM. We select eviction rate A as a function of Z that results in negligible stash overflow probability according to RingORAM's theoretical analysis [79]. We choose tree height L according to the block count N and eviction rate A to ensure $N \leq A \cdot 2^{L-1}$. The stash space for overflow capacity is chosen with security parameter $\lambda = 80$ for overflow probability 2^{-80}, in line with prior work [17, 71, 89].

Forest layout and subtree stashes. Recall from 3.2 that our forest layout (Figure 2) removes the top R levels of the tree, creating 2^R disjoint subtrees of height $L-R$, each with its own *subtree-local* stashes; each subtree stash has ZR additional space for relocated ancestor buckets. Similarly, we add $2^i \cdot i \cdot Z$ (where $i = \log_2 E$) capacity to relocate the overlap region for E paths within each subtree. While the relocation of buckets into subtree-local copies increases subtree stash sizes, E and R are small constants in practice. They are chosen according to desired parallelism; $R = 3$ enables $2^R = 8$ subtree threads, and one-sort batch eviction performs best with $E \leq 4$ in practice, with only modest increases to combined stash sizes (for a typical $Z = 8$, $R = 3$ expands each subtree stash by 24 blocks; $E = 4$ expands each subtree stash by 64 additional blocks).

Stash occupancy. Our modifications preserve RingORAM's original analysis: bucket relocation by copying to descendants cannot increase overflow stash occupancy, and our eviction procedure performs the same greedy Z-postprocessing as RingORAM. We elaborate further in Appendix B.

4 From DORAMs to DOMAPs

The previous section described the design of our concurrent doubly-oblivious ORAM. We discuss using Sonic to construct doubly-oblivious OMAPs for various settings.

Low-latency DOMAPs. Existing low-latency DOMAP constructions [17, 71, 89] use search trees following the ODS framework for tree-like accesses [95]. However, they suffer from poor throughput due to the logarithmic factor overhead imposed by oblivious search trees, as has been previously demonstrated [27, 70]. Because existing DOMAP algorithms were developed alongside sequential DORAMs, they are inherently serial and cannot benefit from the novel concurrent DORAM capabilities unlocked by Sonic. The availability of these new capabilities presents a compelling opportunity to design DOMAPs for parallel ORAM accesses.

Recent work proposes designs that are naturally suited for concurrent accesses, such as the prefetching technique proposed in Compass [99]. While Compass used a sequential ORAM in the client-server setting, their approach creates opportunities to leverage concurrent ORAMs. Their prefetching technique proactively retrieves nearby nodes relative to the target, which contain useful data for future accesses. During the primary traversal towards a target, additional traversals are initiated into neighboring regions of the tree, which are divergent traversals across distinct sets of nodes. This pattern of disjoint ORAM accesses aligns with the concurrent access capabilities of SONIC. Therefore, SONIC enables future low-latency DOMAP designs better suited to exploit concurrent ORAM accesses. This opens a pathway to potentially significant throughput improvements for low-latency DOMAPs operating within enclaves. While designing such a concurrent ODS algorithm is beyond the scope of this work, we highlight this as a promising direction for future work.

Trading latency for higher throughput. Due to the limitations of existing low-latency OMAPs [95], we consider sacrificing some latency in order to achieve much higher throughput. This aligns with the approach taken by Snoopy [27], which sets a target latency then attempts to maximize the throughput given this constraint. We first examine Snoopy's approach to understand its limitations, then propose novel high-throughput approaches leveraging SONIC.

Snoopy implements subORAMs using an oblivious hashtable [20], chosen for its state-of-the-art throughput in the shared-memory setting. The hashtable is built by obliviously sorting the input data on two levels according to a PRF. This hashtable can then be queried by a set of unique requests. The uniqueness of requests is guaranteed by Snoopy's load balancer, which deduplicates queries before sending them to subORAMs. Each subORAM stores a dataset shard of N elements. Every time a batch of K requests is received, Snoopy builds a two-tier hashtable over the requests, and linearly scans the subORAM's dataset shard as queries to this hashtable. The hashtable is constructed this way because $N \gg K$ in practical scenarios, and oblivious sorts are much more expensive than linear scans.

Limitations of the two-tier hashtable. We identify a key bottleneck in the two-tier hashtable approach: to process each request batch, it must *linearly scan the entire dataset*. For a subORAM with N elements of size B, given a batch of K requests, assuming ideal parallelization with P threads: the subORAM must sort $\Theta(\frac{KB}{P})$ data, and linearly scan $\Theta(\frac{NB}{P})$ data. This scan imposes a fixed cost proportional to NB, limiting the size of the shard: the request batch cannot be processed without completing the full scan. Because linear scans are easily parallelizable, this approach is highly effective for small datasets. However, as datasets grow, this becomes inefficient; it is not viable to linearly scan arbitrarily large datasets. This limitation on the size of each subORAM means that (1) each subORAM underutilizes the memory available on its server, and (2) an unreasonably large number of subORAMs may be required, with infeasible hardware costs.

A new subORAM design. We propose **PMCHAIN**, a novel hybrid construction, combining low-latency SONIC with the two-tier hashtable. Our key insight is that we can store the position map using the oblivious hashtable, then dispatch concurrent accesses to the ORAM. This new design synergizes the strengths of each construction (the high throughput of the hashtable, and the low latency of the ORAM) and effectively bridges the scaling limitation identified above. Unlike the hashtable, which must linearly scan the full dataset for every request batch, an ORAM can be initialized once at startup, then queried with microsecond-scale latency per access.

Using the ORAM this way requires us to store a position map to map keys to values. Crucially, the size of this position map depends only on N, and is independent of B. Each position map entry must include a counter and block identifier: the counter is used with a PRF to map blocks to leaves, and the identifier allows us to match OMAP keys with their target records. We can thus store the position map using as little as 8 bytes per record. For datasets with $B > 16$, which is very common (Snoopy uses $B = 160$ [27]), the cost of the linear scan on the position map ($8N$) is much cheaper than the cost of the linear scan on the full dataset (NB). We treat the hashtable's linear scan phase as a fixed cost. In contrast, the cost of the ORAM phase is tunable, because each individual ORAM access has sub-millisecond latency even for large block sizes. Given a target latency, we select the largest N such that the hashtable phase completes within the latency budget; remaining time is used for the ORAM phase. We can thus support arbitrarily large B and tune latency-throughput tradeoffs per-workload. While the linear scan fundamentally limits the maximum size of Snoopy's subORAMs, our approach avoids this limitation and allows much more data to be stored per subORAM, as we show in Section 5.

5 Experimental Evaluation

Experimental Setup. We perform all experiments using Microsoft Azure's DCsv3-series of virtual machines which support Intel-SGXv2. Intel SGXv2 offers substantially more EPC memory than SGXv1, allowing us to store large datasets within enclave memory without incurring expensive paging overheads [35, 89]. We run all experiments on Standard_DC32s_v3 instances, which support Intel SGXv2 with 192GiB of EPC. Our code is written in C++ using the Intel SGX SDK [2]. All algorithms are doubly-oblivious in control flow and access pattern and implemented with the optimized primitives described in Section 3.3.

Measurements/Parameters. We evaluate our constructions for block counts in the range $N = 2^{10}$ to 2^{27}, as our experiments focus on the shared-memory setting. For cases which require exceeding the SGXv2 EPC memory, we can directly apply EnigMap's packed layout technique [89] to reduce the im-

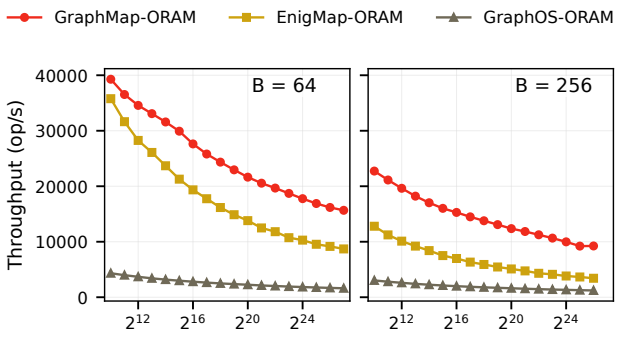


Figure 6: PATHORAM-variant access throughput for each implementation as N increases. B denotes ORAM block size.

pact of page swaps. For very large datasets which exceed the memory of a single server, we can leverage Snoopy's scalable distributed architecture [27] orthogonally to our techniques. We also evaluate performance across different access thread counts to demonstrate scaling characteristics. For SONIC, we evaluate both serial and parallel configurations with different parameters, and measure their latency and throughput across dataset sizes. To analyze ORAM performance, we perform 10^7 accesses and measure the average latency of each access (including evictions), and compute overall throughput based on the total time taken to perform all accesses. We select stash sizes for all schemes to ensure an overflow probability of 2^{-80} according to theoretical analyses [79, 87].

Baselines. The state-of-the-art baselines for doubly-oblivious ORAMs targeting hardware enclaves are GraphOS [17] and EnigMap [89], doubly-oblivious derivatives of PATHORAM [87]; both provide open-source implementations for SGXv2. Despite GraphOS's asymptotically better eviction algorithm, their implementation incurs overheads due to large metadata and hot-path memory allocations, understating the impact of their optimizations. EnigMap's doubly-oblivious PATHORAM implementation is well-engineered with strong concrete performance, and is the best existing baseline.

We analyze these existing state-of-the-art baselines [17, 71, 89], and combine their best ideas with our newly proposed ideas to create a *strengthened baseline* we call GRAPHMAP. GRAPHMAP aims to demonstrate the best possible performance of an optimized PATHORAM construction, combining prior optimizations with our improved primitives. We use GraphOS's asymptotically better eviction and the strong engineering standards of EnigMap as the foundation for our baseline. We additionally apply our observation in Section 3.2, replacing a sort with more efficient compaction. All implementations are parameterized with $Z = 4$, the best-performing configuration achieving negligible overflow probability [87]. In Figure 6, we compare GRAPHMAP to the existing implementations [17, 89] of both GraphOS and EnigMap.

Our evaluation in Figure 6 shows that our new baseline GRAPHMAP consistently outperforms other implementations across all values of N. For $N = 2^{27}$, with $B = 64$, GRAPHMAP is 1.80× faster than EnigMap and 9.72× faster than GraphOS. For $N = 2^{26}$ with $B = 256$, GRAPHMAP is 2.70× faster than EnigMap and 7.58× faster than GraphOS. We also observe that both the implementations of EnigMap and GraphOS fail to initialize at the largest N values due to increased memory overheads; in contrast, our implementation is able to support the largest experiment sizes. We therefore use GRAPHMAP as a new baseline for evaluating all other constructions.

Our Novel Enclave ORAM. We now evaluate SONIC against the optimized baseline described above. We evaluate the performance of SONIC separately for serial and parallel modes. Because SONIC is highly configurable and has a number of parameters that are tunable for parallel performance (S, Z, T, R, E), we explain the performance impacts of each parameter in Section 3.3. To aid in parameter selection, we develop a parameter search tool that rapidly tests and evaluates parameters to optimize latency and throughput for specific N and B. As a rule of thumb, micro-optimizing parameters for each point can provide modest (≈ 10–30%) throughput improvements. This tool is particularly useful for selecting strongly performing parameters for a given hardware configuration, indirectly identifying compute and bandwidth bottlenecks. Importantly, our parameter search tool uniquely benefits SONIC, as parameter tuning is not possible for PATHORAM, which always performs best with the minimum $Z = 4$. To simplify our evaluations, we *do not* optimize parameters for each point, and instead select fixed parameters for all points that provide good overall performance across different N and B. For our evaluations, we use only the following fixed parameterizations of SONIC: (1) A serial ($T = 1$) configuration with $Z = 8, S = 12, R = 3, E = 1$, labeled **Ser**; (2) A parallel ($T = 8$) configuration with $Z = 16, S = 16, R = 3, E = 2$, labeled **Par8**; (3) A parallel ($T = 16$) configuration with $Z = 16, S = 16, R = 3, E = 2$, labeled **Par16**. Note that the serial configuration supports only serial *accesses*, but still leverages parallelism only for eviction, with 2^R worker threads. Finally, in Figure 9, we use the same $Z = 16, S = 16, R = 3, E = 2$ family and vary T.

5.1 Performance Analysis

Scaling with block count. Figure 7 demonstrates that SONIC strongly outperforms all baselines. SONIC in both serial and parallel configurations achieves state-of-the-art throughput by a significant margin across all values of N. At $N = 2^{27}, B = 64$, parallel SONIC maintains a throughput 16.26× higher than GRAPHMAP, 29.25× higher than EnigMap, and 158.02× higher than GraphOS. Comparing the access latency of all constructions (Figure 8): serial-access SONIC demonstrates a state-of-the-art latency 0.29× that of the baseline GRAPHMAP. On the other hand, parallel SONIC demonstrates latency 0.98× that of GRAPHMAP while sustaining much higher throughput. As discussed above, SONIC's parameters

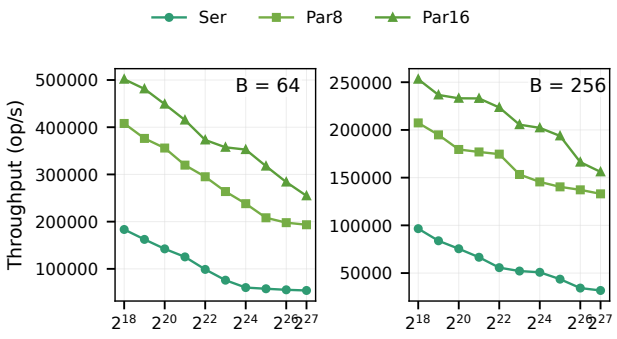


Figure 7: ORAM access throughput as N increases. Labels indicate SONIC parameters.

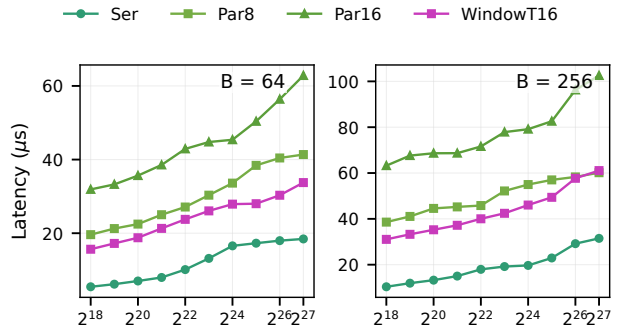


Figure 8: ORAM access latency as N (block count) is increased. The series labels indicate SONIC parameters.

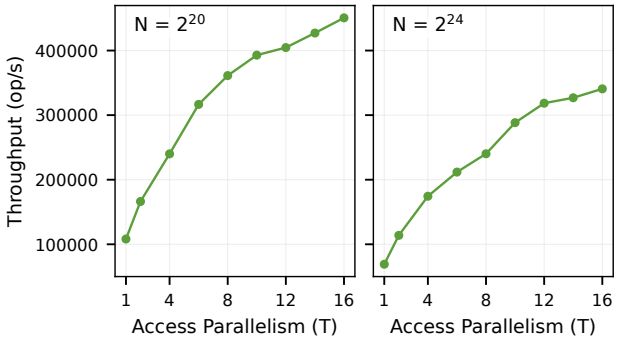


Figure 9: ORAM access throughput as T (access parallelism) is increased.

can be further tuned for lower latency at a given N.

Figure 7 also shows a promising trend in the throughput of parallel SONIC. While all other approaches drastically drop in throughput as block count increases, parallel SONIC's throughput decreases more moderately. This demonstrates the effectiveness of parallelism in overcoming the performance bottleneck of sequential ORAMs. At $N = 2^{27}, B = 64$, parallel SONIC achieves a new state-of-the-art throughput of over 254K op/s. In contrast, EnigMap's throughput is 8706 op/s, and GraphOS's throughput is 1612 op/s.

Scaling with access parallelism. Figure 9 shows the behavior of SONIC throughput as access parallelism is increased. We can see that larger ORAM trees benefit more from access parallelism: going from $T = 1$ to $T = 16$ improves throughput by 4.17× at $N = 2^{20}$ and 4.93× at $N = 2^{24}$. This demonstrates the increasing effectiveness of parallelism in larger ORAM trees. As demonstrated by Figure 7, sequential ORAMs suffer from a major performance bottleneck at large sizes, while parallel SONIC maintains reasonable throughput.

Deferred evictions for disjoint accesses. We now evaluate the deferred eviction strategy that can be used for sequences of disjoint accesses proposed in Section 3.2. When the calling application can guarantee that a sequence of accesses is disjoint (i.e. accessing unique blocks), we can defer evictions to improve throughput. This guarantee is enforced by design through request deduplication in Snoopy [27], and during traversals of non-overlapping regions of oblivious data structures such as Compass [99]. We evaluate three eviction schedules: (1) without using deferred eviction, evicting every A' accesses; this is the same **Par16** evaluated above; (2) windowed deferred eviction; and finally (3) online-only, where we evaluate throughput in only the online phase with no evictions, assuming evictions are performed as offline maintenance as described in Section 3.2. We name the windowed deferred eviction configuration **WindowT16**. We name the online/offline configuration **OnlineT16**. These parameters are selected by our optimization tool explained above. All configurations use $T = 16$ for a fair comparison.

Figure 10 shows the large throughput impact of being able to defer evictions. Compared to standard parallel SONIC, using windowed deferred eviction improves throughput by 1.86×, and the online phase reaches a throughput 3.54× higher. Compared to our baselines, the throughput of online SONIC is 58× higher than GRAPHMAP, 104× higher than EnigMap, and 560× higher than GraphOS.

Without deferred eviction, evictions periodically lock the tree and limit access concurrency. When evictions can be deferred, we can perform long sequences of accesses with minimal synchronization, through our techniques in Section 3.1. The most realistic and generally applicable schedule is windowed deferred eviction, with windows of accesses followed by eviction phases. We can directly apply windowed deferred eviction when running SONIC with disjoint accesses, such as for Snoopy [27]. Finally, this deferred eviction technique uniquely allows SONIC to effectively target the offline-maintenance setting motivated in recent work [82]. The online phase consists of accesses without any evictions; all evictions are deferred to the offline phase. This approach achieves drastically higher throughput in the online phase, but we note that it is only applicable to applications that underutilize hardware and tolerate an offline-maintenance phase.

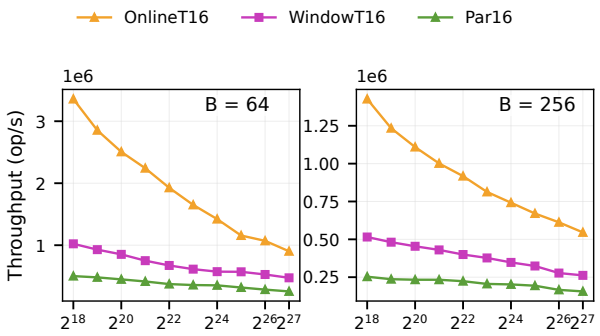


Figure 10: Parallel SONIC access throughput compared to windowed and online-only deferred eviction as N is increased.

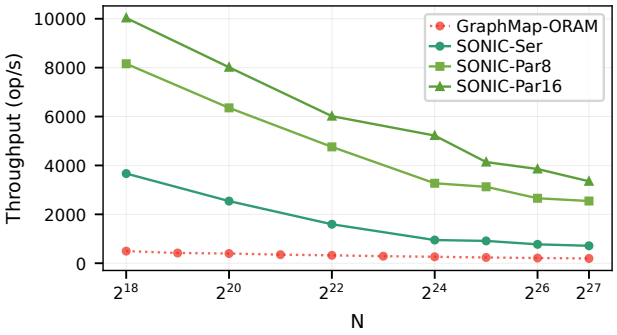


Figure 13: OMAP access throughput as N (dataset size) is increased. Block size is fixed at $B = 64$.

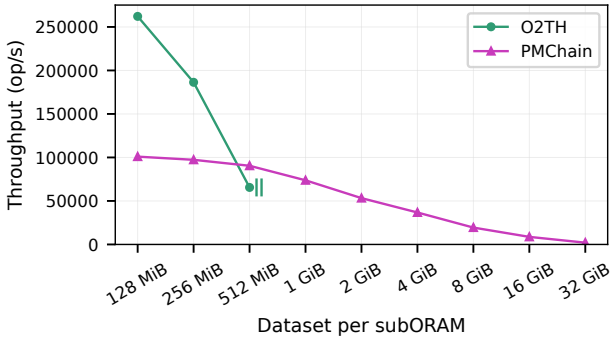


Figure 11: Isolated single-subORAM throughput ($N = 2^{23}$, latency target 1.0s)

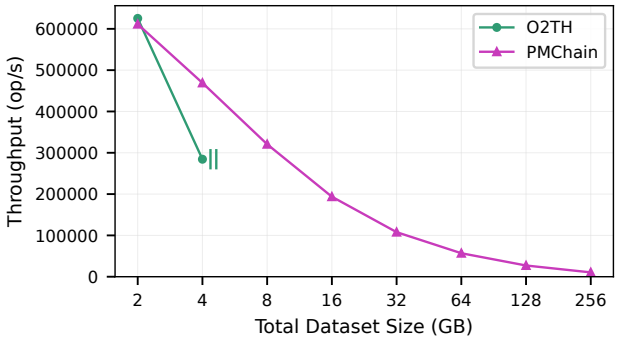


Figure 12: Throughput in a multi-server deployment with 1 load balancer and 8 subORAMs, including load balancer and network overheads. Dataset is evenly sharded across subORAMs ($N = 2^{23}$ per subORAM, latency target 1.0s)

Low-latency DOMAPs. As discussed conceptually in Section 4, the concurrent capabilities of SONIC create opportunities for low-latency DOMAP designs to achieve higher throughput by parallelizing ORAM accesses, for instance, via techniques like disjoint prefetching [99]. While we leave the development of low-latency concurrent DOMAPs to future work, we illustrate the potential impact of such a design in Figure 13; demonstrating the performance improvements available to future low-latency concurrent DOMAPs.

Comparison of subORAMs. We now evaluate our high-throughput DOMAP construction **PMCHAIN** described in Section 4 as a subORAM replacement for Snoopy's architecture [27]. For a strengthened baseline, we reimplement Snoopy for SGXv2 with an optimized two-tier hashtable (O2TH) using state-of-the-art oblivious primitives. Both O2TH and PMCHAIN use the same 8-thread configuration.

In Figure 11, we isolate a single subORAM node and measure the throughput it can sustain within a 1.0s latency target. Because in the Snoopy architecture, request batches are deduplicated and thus disjoint, we use SONIC's deferred eviction. We set N to the maximum that can be scanned within the latency target ($N = 2^{23}$), and measure the throughput of each subORAM as we scale the dataset by increasing B. The single-subORAM evaluation shows the expected behavior: O2TH is faster at the smallest sizes, where its linear scan remains cheap, but fails to scale beyond 512 MiB ($N = 2^{23}, B = 64$). In contrast, our PMCHAIN subORAM reaches 1.38× higher throughput at 512 MiB and supports datasets as large as 32 GiB ($N = 2^{23}, B = 4096$). This is because the position map lookup phase performs the linear scan with cost depending only on N and independent of B. All remaining latency budget can be used to perform as many ORAM accesses as possible.

Multi-server comparison. In Figure 12, we evaluate the same comparison in the full Snoopy deployment with 1 load balancer, 8 subORAMs, and a 1.0s latency target, as we scale the total dataset. We measure useful end-to-end throughput for each dataset size, comparing Snoopy's original subORAM with PMCHAIN (we do *not* assume offline-maintenance; both access and eviction must complete within the latency budget). Under this deployed measurement, which includes load-balancer routing and network communication, O2TH is faster only at the smallest total dataset. At 4GB, Snoopy's original subORAM limit, the PMCHAIN-based system supports **469K**

req/s, sustaining throughput 1.65× higher than O2TH (284K req/s). Beyond this point, Snoopy's original subORAM fails to meet the latency target, while PMCHAIN scales to **256GB** total and supports **10K req/s**. Note that this configuration runs each PMCHAIN subORAM with a 32GB dataset shard; overall system throughput could be scaled higher by adding more subORAMs and thus reducing per-shard size [27].

Together, these evaluations demonstrate the scalability limitation of Snoopy's subORAM, and show that PMCHAIN is an effective drop-in replacement to improve dataset scalability by 64×. We estimate that Snoopy with PMCHAIN-based subORAMs could support a **1TB** dataset with around 32 servers. Our hybrid design can be used to build a more practical and more scalable high-throughput distributed OMAP, supporting real-world dataset sizes with reduced hardware cost.

6 Conclusion

We presented SONIC, the first parallel/concurrent, doubly-oblivious tree-based ORAM for TEEs. Our design directly addresses all central bottlenecks of prior work, enabling concurrent access and two forms of parallel batch eviction. SONIC achieves state-of-the-art throughput and latency, and is tunable for different application requirements. We also show how SONIC can defer evictions to further decrease latency under the precondition of disjoint access sequences. We additionally introduced PMCHAIN as a more scalable subORAM for high-throughput, multi-server deployments, enabling the Snoopy architecture to scale to substantially larger datasets and higher throughput with the same hardware. Overall, SONIC unlocks parallelism/concurrency in doubly-oblivious ORAMs, advancing the scalability of practical oblivious data stores.

References

[1] Confidential computing summit. https://www.confidentialcomputingsummit.com/e/ccs25.

[2] Intel Software Guard Extensions SDK for Linux OS.

[3] Signal: Building a faster ORAM. https://signal.org/blog/building-faster-oram/.

[4] Titan security key. Google Cloud, 2024.

[5] Mohamed Ahmed Abdelraheem, Tobias Andersson, and Christian Gehrmann. Inference and record-injection attacks on searchable encrypted relational databases. *IACR Cryptol. ePrint Arch.*, 2017.

[6] Apple Security Engineering and Architecture (SEAR); User Privacy; Core OS; Services Engineering (ASE); and Machine Learning and AI (AIML). Private cloud compute: A new frontier for ai privacy in the cloud. *Apple Security Research Blog*, 2024.

[7] ARM Limited. ARM TrustZone Technology, 2004.

[8] Gilad Asharov, T-H. Hubert Chan, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. Bucket oblivious sort: An extremely simple oblivious sort. In *SOSA*, 2020.

[9] Gilad Asharov, Eliran Eiluz, Ilan Komargodski, and Wei-Kai Lin. Megablocks: Breaking the logarithmic i/o-overhead barrier for oblivious ram. In *CCS*, 2025.

[10] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, and Elaine Shi. Optorama: Optimal oblivious RAM. In *EUROCRYPT*, 2020.

[11] Gilad Asharov, Ilan Komargodski, and Yehuda Michelson. Futorama: A concretely efficient hierarchical oblivious ram. In *CCS*, 2023.

[12] Elette Boyle, Kai-Min Chung, and Rafael Pass. Oblivious parallel ram and applications. In *TCC*, 2016.

[13] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In *WOOT*, 2017.

[14] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In *CCS*, 2015.

[15] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel-Cătălin Roşu, and Michael Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. *NDSS*, 2014.

[16] Anrin Chakraborti and Radu Sion. Concuroram: High-throughput stateless parallel multi-client oram. *NDSS*, 2019.

[17] Javad Ghareh Chamani, Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Rasool Jalili. Graphos: Towards oblivious graph processing. In *VLDB*, 2023.

[18] Javad Ghareh Chamani, Dimitrios Papadopoulos, Mohammadamin Karbasforushan, and Ioannis Demertzis. Dynamic searchable encryption with optimal search in the presence of deletions. In *USENIX Security*, 2022.

[19] T-H Hubert Chan, Kai-Min Chung, and Elaine Shi. On the depth of oblivious parallel ram. In *ASIACRYPT*, 2017.

[20] T-H Hubert Chan, Yue Guo, Wei-Kai Lin, and Elaine Shi. Oblivious hashing revisited, and applications to asymptotically efficient oram and opram. In *ASIACRYPT*, 2017.

[21] Zhao Chang, Dong Xie, Sheng Wang, and Feifei Li. Towards practical oblivious join. In *SIGMOD*, 2022.

[22] Binyi Chen, Huijia Lin, and Stefano Tessaro. Oblivious parallel ram: improved efficiency and generic constructions. In *Theory of Cryptography Conference*, 2015.

[23] Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. Detecting privileged side-channel attacks in shielded execution with déjà vu. In *ASIACCS*, 2017.

[24] Victor Costan, Ilia A. Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security*, 2016.

[25] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *CCS*, 2006.

[26] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. In *SP*, 2020.

[27] Emma Dauterman, Vivian Fang, Ioannis Demertzis, Natacha Crooks, and Raluca Ada Popa. Snoopy: Surpassing the scalability bottleneck of oblivious storage. In *SOSP*, 2021.

[28] Jonathan L Dautrich Jr and Chinya V Ravishankar. Compromising privacy in precise query protocols. In *EDBT*, 2013.

[29] Ioannis Demertzis, Javad Ghareh Chamani, Dimitrios Papadopoulos, and Charalampos Papamanthou. Dynamic searchable encryption with small client storage. *NDSS*, 2020.

[30] Ioannis Demertzis, Dimitrios Papadopoulos, and Charalampos Papamanthou. Searchable encryption with optimal locality: Achieving sublogarithmic read efficiency. In *CRYPTO*, 2018.

[31] Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Saurabh Shintre. SEAL: attack mitigation for encrypted databases via adjustable leakage. In *USENIX Security*, 2020.

[32] Ioannis Demertzis, Stavros Papadopoulos, Odysseas Papapetrou, Antonios Deligiannakis, Minos Garofalakis, and Charalampos Papamanthou. Practical private range search in depth. In *TODS*, 2018.

[33] Ioannis Demertzis and Charalampos Papamanthou. Fast searchable encryption with tunable locality. In *SIGMOD*, 2017.

[34] Ioannis Demertzis, Rajdeep Talapatra, and Charalampos Papamanthou. Efficient searchable encryption through compression. In *VLDB*, 2018.

[35] Muhammad El-Hindi, Tobias Ziegler, Matthias Heinrich, Adrian Lutsch, Zheguang Zhao, and Carsten Binnig. Benchmarking the second generation of intel sgx hardware. In *DaMoN*, 2022.

[36] Ethereum Foundation. Ethereum.org, 2025.

[37] Kyle Fredrickson, Ioannis Demertzis, James P Hughes, and Darrell DE Long. Sparta: Practical anonymity with long-term resistance to traffic analysis. In *SP*, 2025.

[38] Matthieu Giraud, Alexandre Anzala-Yamajako, Olivier Bernard, and Pascal Lafourcade. Practical passive leakage-abuse attacks against symmetric searchable encryption. In *SECRYPT*, 2017.

[39] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 1996.

[40] Google Privacy Sandbox Team. Protected audience api (formerly fledge): Privacy-preserving ad auction mechanism. Privacy Sandbox Developer Documentation, 2023.

[41] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on Intel SGX. In *EuroSec*, 2017.

[42] Paul Grubbs, Anurag Khandelwal, Marie-Sarah Lacharité, Lloyd Brown, Lucy Li, Rachit Agarwal, and Thomas Ristenpart. Pancake: Frequency smoothing for encrypted data stores. In *USENIX Security*, 2020.

[43] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G. Paterson. Pump up the volume: Practical database reconstruction from volume leakage on range queries. In *CCS*, 2018.

[44] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G. Paterson. Learning to reconstruct: Statistical learning theory and encrypted database attacks. In *SP*, 2019.

[45] Paul Grubbs, Richard McPherson, Muhammad Naveed, Thomas Ristenpart, and Vitaly Shmatikov. Breaking web applications built on top of encrypted data. In *CCS*, 2016.

[46] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. Strong and efficient cache side-channel protection using hardware transactional memory. In *USENIX*, 2017.

[47] Tianyao Gu, Yilei Wang, Bingnan Chen, Afonso Tinoco, Elaine Shi, and Ke Yi. Efficient oblivious sorting and shuffling for hardware enclaves. In *USENIX Security*, 2025.

[48] Zichen Gui, Oliver Johnson, and Bogdan Warinschi. Encrypted databases: New volume attacks against range queries. In *CCS*, 2019.

[49] Yitong Guo, Hongbo Chen, Haobin Hiroki Chen, Yukui Luo, XiaoFeng Wang, and Chenghong Wang. Bolt: Bandwidth-optimized lightning-fast oblivious map powered by secure hbm accelerators. In *CCS*, 2025.

[50] Marcus Hähnel, Weidong Cui, and Marcus Peinado. High-resolution side channels for untrusted operating systems. In *USENIX ATC*, 2017.

[51] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *HASP*, 2013.

[52] T-H Hubert Chan and Elaine Shi. Circuit opram: Unifying statistically and computationally secure orams and oprams. In *TCC*, 2017.

[53] Ryan Hurst and Gary Belvin. Google's key transparency.

[54] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS*, 2012.

[55] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. Generic attacks on secure outsourced databases. In *CCS*, 2016.

[56] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. *Communications of the ACM*, 2020.

[57] Evgenios M Kornaropoulos, Nathaniel Moyer, Charalampos Papamanthou, and Alexandros Psomas. Leakage inversion: Towards quantifying privacy in searchable encryption. In *CCS*, 2022.

[58] Evgenios M. Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. Data recovery on encrypted databases with k-nearest neighbor query leakage. In *SP*, 2019.

[59] Evgenios M. Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. The state of the uniform: Attacks on encrypted databases beyond the uniform query distribution. In *SP*, 2020.

[60] Evgenios M Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. Response-hiding encrypted ranges: Revisiting security via parametrized leakage-abuse attacks. In *SP*, 2021.

[61] Simeon Krastnikov, Florian Kerschbaum, and Douglas Stebila. Efficient oblivious database joins. 2020.

[62] Marie-Sarah Lacharité, Brice Minaud, and Kenneth G. Paterson. Improved reconstruction attacks on encrypted data using range query leakage. In *SP*, 2018.

[63] Kai Li, Jiaqi Chen, Xianghong Liu, Yuzhe Richard Tang, XiaoFeng Wang, and Xiapu Luo. As strong as its weakest link: How to break blockchain dapps at rpc service. In *NDSS*, 2021.

[64] Xiang Li, Yunqian Luo, and Mingyu Gao. Bulkor: Enabling bulk loading for path oram. In *SP*, 2024.

[65] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *arXiv preprint arXiv:1801.01207*, 2018.

[66] Chang Liu, Liehuang Zhu, Mingzhong Wang, and Yu-an Tan. Search pattern leakage in searchable encryption: Attacks and new construction. *Inf. Sci.*, 2014.

[67] Zhongtang Luo, Rohan Murukutla, and Aniket Kate. Last mile of blockchains: Rpc and node-as-a-service. In *TPS-ISA*, 2022.

[68] Sujaya Maiyya, Sharath Chandra Vemula, Divyakant Agrawal, Amr El Abbadi, and Florian Kerschbaum. Waffle: An online oblivious datastore for protecting data access patterns. 2024.

[69] Evangelia Anna Markatou and Roberto Tamassia. Full database reconstruction with access and search pattern leakage. In *ISC*, 2019.

[70] Apostolos Mavrogiannakis, Xian Wang, Ioannis Demertzis, Dimitrios Papadopoulos, and Minos Garofalakis. {OBLIVIATOR}:{OBLIVIous} parallel joins and other {OperATORs} in shared memory environments. In *USENIX Security*, 2025.

[71] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. Oblix: An efficient oblivious search index. In *SP*, 2018.

[72] Priyanka Mondal, Javad Ghareh Chamani, Ioannis Demertzis, and Dimitrios Papadopoulos. {I/O-Efficient} dynamic searchable encryption meets forward & backward privacy. In *USENIX Security*.

[73] Kartik Nayak and Jonathan Katz. An oblivious parallel ram with $O(\log^2 N)$parallel runtime blowup. *Cryptology ePrint Archive*, 2016.

[74] Nicholas Ngai, Ioannis Demertzis, Javad Ghareh Chamani, and Dimitrios Papadopoulos. Distributed & scalable oblivious sorting and shuffling. In *SP*, 2024.

[75] Alexander Nilsson, Pegah Nikbakht Bideh, and Joakim Brorsson. A survey of published attacks on intel sgx. arXiv:2006.13598, 2020.

[76] Sarvar Patel, Giuseppe Persiano, Mariana Raykova, and Kevin Yeo. Panorama: Oblivious RAM with logarithmic overhead. In *FOCS*, 2018.

[77] Porter, Nelly and Garms, Jason. Introducing asylo: an open-source framework for confidential computing. Google Cloud Blog, 2018.

[78] David Pouliot and Charles V. Wright. The shadow nemesis: Inference attacks on efficiently deployable, efficiently searchable encryption. In *CCS*, 2016.

[79] Ling Ren, Christopher Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten Van Dijk, and Srinivas Devadas. Constants count: Practical improvements to oblivious {RAM}. In *USENIX Security*, 2015.

[80] Cetin Sahin, Victor Zakhary, Amr El Abbadi, Huijia Lin, and Stefano Tessaro. Taostore: Overcoming asynchronicity in oblivious data storage. In *SP*, 2016.

[81] Sajin Sasy, Aaron Johnson, and Ian Goldberg. Fast fully oblivious compaction and shuffling. In *CCS*, 2022.

[82] Sajin Sasy, Aaron Johnson, and Ian Goldberg. Waks-on/waks-off: Fast oblivious offline/online shuffling and sorting with waksman networks. In *CCS*, 2023.

[83] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using SGX to conceal cache attacks. In *DIMVA*, 2017.

[84] Amin Setayesh, Cheran Mahalingam, Emily Chen, and Sujaya Maiyya. Treebeard: A scalable and fault tolerant ORAM datastore. In *USENIX Security*, 2025.

[85] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *NDSS*, 2017.

[86] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. Preventing page faults from telling your secrets. In *AsiaCCS*, 2016.

[87] Emil Stefanov, Marten van Dijk, Elaine Shi, T-H Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: an extremely simple oblivious ram protocol. *JACM*, 2018.

[88] Emil Stefanov, Elaine Shi, and Dawn Song. Towards Practical Oblivious RAM. *NDSS*, 2012.

[89] Afonso Tinoco, Sixiang Gao, and Elaine Shi. EnigMap: External-Memory oblivious map for secure enclaves. In *USENIX Security*, 2023.

[90] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient {Out-of-Order} execution. In *USENIX Security*, 2018.

[91] Cédric Van Rompay, Refik Molva, and Melek Önen. A leakage-abuse attack against multi-user searchable encryption. *PETS*, 2017.

[92] Shan Wang, Ming Yang, Wenxuan Dai, Yu Liu, Yue Zhang, and Xinwen Fu. Deanonymizing ethereum users behind third-party rpc services. In *INFOCOM*, 2024.

[93] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *CCS*, 2017.

[94] Xiao Wang, Hubert Chan, and Elaine Shi. Circuit oram: On tightness of the goldreich-ostrovsky lower bound. In *CCS*, 2015.

[95] Xiao Shaun Wang, Kartik Nayak, Chang Liu, TH Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious data structures. In *CCS*, 2014.

[96] Haojie Ye, Yuchen Xia, Yuhan Chen, Kuan-Yu Chen, Yichao Yuan, Shuwen Deng, Baris Kasikci, Trevor Mudge, and Nishil Talati. Palermo: Improving the performance of oblivious memory using protocol-hardware co-design. In *HPCA*. IEEE, 2025.

[97] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In *USENIX Security*, 2016.

[98] Leqian Zheng, Zheng Zhang, Wentao Dong, Yao Zhang, Ye Wu, and Cong Wang. {H2O2RAM}: A {High-Performance} hierarchical doubly oblivious {RAM}. In *USENIX Security*, 2025.

[99] Jinhao Zhu, Liana Patel, Matei Zaharia, and Raluca Ada Popa. Compass: encrypted semantic search with high accuracy. In *OSDI*, 2025.

A Expanded Relevant Prior Work

A.1 PathORAM

We briefly recap the construction PathORAM. PathORAM is a protocol enabling a client to access an array of N data blocks stored on a server, without revealing which blocks it is accessing. The server stores a tree of height $L = \log_2 N$, with each node storing a bucket of Z blocks. The client stores a *position map* mapping each address to a random tree leaf, as well as a bounded stash that holds blocks between evictions. The protocol maintains the invariant that each block in the tree is always along the path to its assigned leaf.

Access. Each time the client accesses a block, it downloads all blocks along the path to its assigned leaf, and adds them to the stash. It then remaps the block to a fresh random leaf. After updating the target block, the client performs *eviction* along the path that was just read to offload the stash.

Eviction. The eviction process offloads blocks from the stash back to the ORAM tree. The target path is filled from leaf to root to maximize offloading. The stash analysis proves that a bucket size of $Z \geq 4$ ensures negligible overflow probability.

Doubly-oblivious variant. In the TEE setting, both client and server exist on the same machine. The client must thus access its own internal state obliviously: this combination of oblivious access to both client and server state is called double-obliviousness [71, 74, 81, 82]. Accesses can be made doubly oblivious with fixed-size linear scans, and evictions can be made doubly oblivious by combining oblivious sorts and linear scans [17, 71, 89].

A.2 RingORAM

RingORAM improves upon PathORAM, with a new bucket structure and eviction schedule. Buckets in RingORAM contain $Z + S$ blocks, with up to Z real blocks and S

dummy blocks, stored with a secret permutation. Both the permutation as well as the identity of real blocks are stored as bucket metadata. By scanning bucket metadata, the access operation reads only one block per bucket, improving bandwidth compared to PATHORAM. RINGORAM also features an improved stash analysis due to its reverse-lexicographic eviction schedule, which evicts minimally overlapping paths. This enables RINGORAM to evict every A accesses rather than after each one as in PATHORAM.

Access. To access a block, the client proceeds down each bucket along the mapped path, reading exactly one block per bucket. At each bucket, the metadata is scanned to determine which block to read. If the metadata indicates that the bucket contains the target block, its offset is selected, otherwise a random dummy is selected.

Reshuffle. To maintain the availability of fresh dummy blocks within each bucket, after each access, buckets that have reached S touches are *reshuffled*: the Z remaining blocks are extracted to the stash, and Z blocks along with S fresh dummies are permuted and written back to the bucket.

Eviction. Every A accesses, a path is evicted (according to the reverse-lexicographic schedule). The eviction algorithm works exactly like in PATHORAM (see Section A.1; evicted blocks for each bucket are permuted along with dummies and written back to buckets.

A.2.1 Reshuffles do not contribute to stash bound

The RINGORAM paper [79] explicitly states: *"We note here that the reshuffling of a bucket does not affect the occupancy of the bucket, and is thus irrelevant to the proof we present here"*. An examination of the proof confirms that they consider only the impact of the periodic, reverse-lexicographically ordered scheduled full-path evictions on tree occupancy.

A.3 Snoopy

Snoopy is a distributed ORAM and OMAP architecture addressing the scalability bottleneck of tree-based ORAMs due to their sequential nature. Snoopy's core idea is to sacrifice latency to achieve improved throughput, through parallelism and batch processing. Instead of storing the full ORAM in a single tree, Snoopy shards the dataset across multiple subORAMs. Rather than processing requests individually, load balancers collect large batches of requests; these requests are then obliviously routed to their target subORAMs.

Request processing. Snoopy operates in epochs of a specified target latency. Load balancers deduplicate requests and send them to the correct subORAM, then match responses to requests and return results back to the client. Load balancers route obliviously by distributing requests according to a balls-and-bins analysis: a set of R deduplicated requests (balls) are routed to S subORAMs (bins). This provides a batch size $f(R,S)$, denoting the maximum number of requests that will be sent to any one subORAM with negligible overflow probability. The routing process thus consists of obliviously sorting and compacting the request batch to deduplicate it, then constructing batches to be sent to each subORAM.

subORAM processing. Every epoch, each subORAM receives a batch of K requests from the load balancers. The subORAM must obliviously retrieve and update data to process the request batch. Snoopy's evaluation [27] found that the most performant subORAM implementation was based on the oblivious two-tier hashtable [20]. Because request batch size K is much smaller than subORAM dataset size N, and hashtable construction requires oblivious sort, a hashtable is constructed on the requests, then the dataset is queried through the hashtable. Results are then gathered and sent back to the load balancers.

B Stash/Security Analysis

We show that SONIC's overflow stash usage is bounded by that of RINGORAM. Recall from Section 3 that SONIC preserves RINGORAM's leaf-remap rule and reverse-lexicographic eviction schedule, but changes the physical layout used during eviction: we remove the top R levels to create a forest of 2^R subtrees, and within each subtree we batch $E = 2^i$ consecutive reverse-lexicographic path evictions. We choose A and L exactly as in RINGORAM, so that $N \leq A \cdot 2^{L-1}$.

SONIC then performs one eviction round after an epoch of $A' = A \cdot 2^R \cdot E$ accesses, processing the next $2^R E$ scheduled reverse-lexicographic paths in one batch. After subtree k completes its local eviction, we interpret the blocks written back into $\text{SubtreeStash}[k]$ as $\text{Overlap}_k \sqcup \text{Treetop}_k \sqcup \text{Overflow}_k$.

Overlap_k is the relocated capacity for the overlap region removed for one-sort batch eviction, and Treetop_k is the relocated capacity for the R removed ancestors above the subtree. The overlap and treetop regions *are filled first* before blocks are placed in overflow capacity Overflow_k. Thus, the load of Overflow_k is the true overflow from the subtree.

Intuition. SONIC's eviction schedule exactly matches that of *delayed* RINGORAM [79]. SONIC evicts the same set of leaves, but in a modified tree where some shared buckets have been duplicated and made private to individual subtrees or path segments. Copying a bucket by leaf range cannot make placement harder. Eviction first fills tree buckets, then relocated regions; what remains is true overflow. This true overflow is therefore no larger than the stash needed by the corresponding delayed RINGORAM execution.

Facts from RINGORAM. We summarize the RINGORAM stash analysis framework [79]. They analyze an ∞-ORAM with infinite bucket capacity, denoted $\text{ORAM}_L^{\infty,A}$, together with a greedy postprocessing map G_Z that enforces finite bucket capacity Z and places excess in the stash. Let S^∞ and S^Z denote the resulting ∞-ORAM and postprocessed states after the same access sequence. RINGORAM proves that

$G_Z(S^\infty) = S^Z$. They then characterize stash overflow by overload of rooted subtrees: for any $T \in ORAM_L^{\infty,A}$ with $n(T)$ nodes, capacity $c(T) = Z \cdot n(T)$, and pre-postprocessing load $X(T)$: $st(S^Z) > K \iff \exists T : X(T) > c(T) + K$. Finally, with eviction under reverse-lexicographic schedule every A accesses, $E[X(T)] \le n(T) \cdot A/2$, whenever $N \le A \cdot 2^{L-1}$, with an exponentially decaying tail.

Delayed reference process. Let $\mathsf{Ring}^{\mathrm{del}}$ denote RINGORAM on the same access sequence, with the same leaf remappings and reverse-lexicographic eviction order as SONIC; except that the next $2^R \cdot E$ evictions are delayed until the end of epoch length A'. This is in accordance with the *delayed eviction* strategy described for bursts in the RINGORAM paper [79].

Copy refinement. For any bucket u, let $\mathrm{Leaves}(u)$ be the set of leaves whose blocks may legally reside in u under the path invariant. A **copy refinement** of u is a set of capacity-Z copies $u_1, \ldots, u_m$ such that the sets $\mathrm{Leaves}(u_1), \ldots, \mathrm{Leaves}(u_m)$ are pairwise disjoint and partition $\mathrm{Leaves}(u)$.

Lemma B.1. *Replacing a bucket by a copy refinement cannot increase the minimum overflow required to place a set of leaf-labeled blocks.*

Proof. Take any feasible placement in the original tree. For each block assigned to u, move it to the unique copy whose leaf set contains that block's leaf. Because copied leaf sets partition $\mathrm{Leaves}(u)$, every block has exactly one destination copy. Because the original bucket held at most Z blocks, no copy receives more than Z blocks. Hence, a copy-modified tree allows a feasible placement with the same overflow. □

SONIC modifications are copy refinement. First, for the forest cut: each removed bucket above depth R is copied once for each descendant depth-R subtree. At removed depth $d < R$, there are 2^d buckets, each of which has 2^{R-d} descendant subtrees, so each removed depth contributes $2^d \cdot 2^{R-d} = 2^R$ copies. Summing over the R removed depths gives $R\,2^R$ copied buckets total, totaling R copied treetop buckets per subtree. Second, for one-sort batch eviction within a subtree, $E = 2^i$ consecutive reverse-lexicographic paths overlap at i levels. At subtree-local overlap depth $d < i$, there are 2^d overlap nodes, and each is copied for its 2^{i-d} descendant path segments, so each overlap depth contributes $2^d \cdot 2^{i-d} = 2^i = E$ copies. This sums to Ei copied buckets within the subtree. Thus, the ZR and EiZ terms in the subtree capacity expansion are the result of applying copy refinements to the original tree.

Theorem B.2. *Let* $\mathsf{Ring}^{\mathrm{del}}$ *be the delayed reference process described above. After one* SONIC *eviction round,*

$$\sum_{k=1}^{2^R} |Overflow_k| \le s = st(\mathsf{Ring}^{\mathrm{del}})$$

Proof. Let $s := st(\mathsf{Ring}^{\mathrm{del}})$. Consider the final placement of the delayed RINGORAM after the $2^R E$ scheduled evictions. Now apply SONIC's copy refinements described above; by Lemma B.1, this yields a feasible placement in the refined SONIC layout with true overflow at most s. The eviction procedure performs greedy Z-postprocessing on the refined layout, filling tree buckets, then $\mathrm{Overlap}_k$, then $\mathrm{Treetop}_k$, and only the remainder enters $\mathrm{Overflow}_k$. By the same rooted-subtree argument used in RINGORAM, greedy postprocessing on a rooted tree leaves no more overflow than any other feasible placement of the same blocks. Therefore the residual $|\mathrm{Overflow}_k|$ is at most the overflow of the refined delayed-RINGORAM placement on subtree k, and the sum across subtrees is bounded by s. □