USENIX

THE ADVANCED COMPUTING
SYSTEMS ASSOCIATION

Treebeard: A Scalable and Fault Tolerant ORAM Datastore

Amin Setayesh, Cheran Mahalingam, Emily Chen,
and Sujaya Maiyya, *University of Waterloo*

https://www.usenix.org/conference/usenixsecurity25/presentation/setayesh

This paper is included in the Proceedings of the
34th USENIX Security Symposium.

August 13–15, 2025 • Seattle, WA, USA

978-1-939133-52-6

Open access to the Proceedings of the
34th USENIX Security Symposium is sponsored by USENIX.

Treebeard: A Scalable and Fault Tolerant ORAM Datastore

Amin Setayesh Cheran Mahalingam Emily Chen Sujaya Maiyya
University of Waterloo

Abstract

We present Treebeard - *the first scalable and fault tolerant* Oblivious RAM (ORAM) based datastore designed to protect applications from access pattern attacks. Current ORAM systems face challenges in practical adoption due to their limited ability to handle concurrent workloads, scale effectively, and ensure fault tolerance. We address all three limitations in Treebeard by utilizing a multi-layer architecture that scales horizontally, handling thousands of requests in parallel, while replicating the data to prevent data loss upon failures. Experimental evaluation demonstrates Treebeard's ability to scale linearly, achieving a throughput of 160k ops/sec with 16 machines; this behavior is similar to the enclave-based state-of-the-art, Snoopy. Being fault-tolerant, Treebeard recovers from failures with close to zero downtime and achieves 13.8x the throughput of QuORAM, the latest fault tolerant ORAM system, even without scaling.

1 Introduction

Cloud-based storage services offer high availability, performance, and scalability in a cost-effective manner. However, privacy concerns limit many applications from migrating to cloud storage: even completely encrypting an application's data falls short of protecting its privacy effectively. Access pattern attacks [7, 15, 19–21, 24–27, 44, 45, 51, 69] observe the access frequency of each encrypted data block and leverage the learned frequency distribution to uncover plaintext data. For example, a recent attack [26] reconstructed the plaintext values of an encrypted database with relative errors as low as $0.003\% - 2.9\%$.

Oblivious RAM or ORAM [17] is a cryptographic primitive that mitigates access pattern attacks by guaranteeing *workload independence*: the adversary-observable data access patterns remain independent of input workload. ORAM schemes protect data from semi-honest active adversaries who attempt to learn *any* non-trivial information about the data. The adversary can control the storage (e.g., observe the access frequency of each data block) and network channels (e.g., view or delay encrypted messages). Many storage systems have integrated ORAM to prevent access pattern attacks [8, 13, 14, 35, 55, 60, 62]. These systems obfuscate access patterns by transforming each logical data access to a sequence of random physical block accesses.

Most existing oblivious data stores rely on a centralized, stateful, trusted proxy to maintain encryption key and coordinate accesses between clients and the server [5, 13, 35, 54, 55, 60, 62]. Proxy-based ORAM systems offer an effective solution to serve concurrent requests in a multi-user setting [5, 13, 35, 55, 60]. However, a single proxy coordinating many concurrent requests quickly becomes a bottleneck and is a single point of failure. Plaintext databases *shard* the data across multiple machines for scalability and replicate stateful entities for fault tolerance. However, incorporating scalability and fault tolerance in ORAM poses non-trivial challenges.

1. ORAM systems scale poorly or suffer from security issues without careful design considerations: Naively partitioning the data into different shards can lead to non-uniform server accesses because a shard with hotkeys will receive many more requests than the other shards. The adversary can infer information about data locality by observing the access imbalance.

Oblivistore [60] and CURIOUS [5] attempt to scale by sharding proxy state across different machines (i.e., subORAMs). But they route client requests to appropriate shards via a single load balancing process, which becomes a scalability bottleneck. Moreover, Oblivistore's sharding mechanism has security vulnerabilities [5, 55]. Snoopy [14], a recent scalable ORAM solution, leverages trusted hardware enclaves to scale horizontally and perform order of magnitude higher than state-of-the-art baselines. However, it requires specialized hardware to offer obliviousness, which is not offered by all cloud providers.

2. Achieving fault tolerance and ensuring high availability in ORAM systems pose significant challenges: Apart from data loss and application downtime upon proxy failures, retrying failed requests may break security if multiple logical requests access the same physical location at failure boundaries.

QuORAM [35] solves the fault-tolerance problem securely by replicating the server-proxy pairs and ensures high availability even while faults occur. However, its throughput drops significantly even if one replica fails and replica recovery poses significant delays to the system.

In summary, although these works partially mitigate either the scalability or the fault tolerance challenges, none of them address both challenges *simultaneously*. Incorporating scalability and fault tolerance into an ORAM scheme, while handling multi-user requests in parallel, introduces significant security challenges (§3).

Our contributions: We propose Treebeard, the first ORAM-based storage system to achieve both scalability and fault tolerance. Treebeard utilizes a tree-based RingORAM [54]-like design with a trusted proxy architecture as its building block. Tree-based schemes structure the outsourced data as a tree and ensure obliviousness by accessing a seemingly random path from the server when processing client requests. Unlike existing ORAM schemes that handle concurrent client requests [5,13,35,55,60], Treebeard eliminates any scalability or fault tolerance bottlenecks introduced by a proxy.

First, Treebeard proposes a novel multi-layer architecture that dissects the state and functionality managed by a proxy and distributes them across multiple layers. The first layer batches client requests and routes them to the subsequent layer. The second layer maintains data-dependent information (including *stash*, a cache-like data structure) and decides which tree paths to retrieve from the server. The third layer communicates with the external storage to access these paths. This multi-layer design empowers Treebeard to scale horizontally and serve over 160k operations in parallel.

Second, each layer in Treebeard meticulously optimizes and parallelizes requests to maximize throughput. In addition to its multi-layer architecture, Treebeard's performance advantage stems from a *multi-path reading* technique: it aggregates a set of paths to be read from the server, ensuring that overlapping nodes across these paths are retrieved only once. Although this optimization temporarily increases storage at the proxy (i.e., *stash* size), it significantly reduces both the communication and computation bandwidths, enabling Treebeard to serve thousands of requests even without scaling. This is an independent contribution that can be integrated with other non-scalable tree-based ORAM schemes such as [54, 55, 62].

Lastly, to guarantee fault tolerance and high availability (under a bounded number of failures), Treebeard replicates processes in each layer using Raft [43]. Implementing and evaluating Treebeard demonstrates its ability to scale the throughput linearly with the number of machines, similar to Snoopy [14], which requires specialized hardware enclaves for security. Although the two systems vary in their threat model, our experiments under comparable scale factors indicate that Treebeard achieves 160 Kops/sec using 16 machines, which is **1.7x** the throughput of Snoopy. Being fault-tolerant, Treebeard recovers from failures with close to zero downtime and exhibits no performance degradation, unlike QuORAM [35]. It achieves **13.8x** of QuORAM's throughput even without scaling.

Motivating applications: Treebeard is well suited for organizations with sensitive data, such as government agencies or healthcare providers. Consider a newly established clinic that collects and stores medical data, including genomic profiles and IoT-generated measurements like heart rate or glucose levels. Medical data can require large storage (e.g., gigabytes per genomic profile). To handle storage demands without overprovisioning local infrastructure, the clinic can offload data to the cloud while using Treebeard to securely access it. Initially, the clinic may operate with only a few on-premises machines, and even in this unscaled setup, Treebeard outperforms existing baselines executing thousands of operations per second. As the clinic grows and the volume of concurrent IoT data accesses increases, Treebeard can be redeployed at a larger scale to accommodate the demand. Furthermore, Treebeard allows organizations to distribute their outsourced data across multiple cloud providers while maintaining a single, trusted proxy layer on-premises. This design reduces dependence on a single provider and enhances privacy, benefiting sensitive domains like healthcare and government.

2 Background

While a variety of ORAM schemes exist [1, 3, 6, 9–11, 42, 47, 49], tree-based ORAMs such as Path ORAM [62] have gained substantial popularity due to their low bandwidth overhead and simple design. Treebeard utilizes a Ring ORAM [54]-like design, a variant of Path ORAM, as its basis. This section provides a primer into both Path ORAM and Ring ORAM.

Path ORAM stores the data on an untrusted cloud storage in a binary tree format. Each node in the tree, called a *bucket*, stores exactly Z data blocks. Clients may request any data block residing on the tree. To hide the physical location of a client requested block, Path ORAM reads *the entire path* on which the block resides. However, repeated reads to the same tree path to serve repeated requests to a given block will generate non-uniform accesses on the server. Thus, Path ORAM maintains a **random path invariant**, which dictates that the mapping from a requested block to its tree path should be random and *change after every access to that block*. This ensures that the server always observes a random path being accessed, independent of the blocks requested by clients.

Path ORAM employs a single trusted proxy between the clients and the untrusted storage to maintain the random path invariant. Upon receiving a read or a write request: (1) The proxy reads the requested block's tree path ID p from a local data structure, *position map*, which maintains a mapping of each data block to its tree path ID. (2) It reads the entire path p from the storage server. (3) It assigns the requested block to a random path p' and updates the *position map*. This step is crucial to maintain the random path invariant as it randomizes

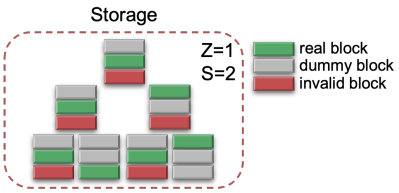


Figure 1: Overview of Ring ORAM.

the mappings from blocks to their tree paths. (4) The proxy updates the requested block's value (for write requests) and stores it *locally* in a cache-like data structure called the *stash*. (5) Finally, it *evicts* the path: simply storing the blocks of a path in the *stash* will cause its size to grow indefinitely. Thus, Path ORAM *evicts* the read path p by pushing each stash-block to the lowest non-full bucket intersecting with p and p', a block's newly assigned path. If no such bucket exists, the blocks continue to remain in the *stash*.

Ring ORAM: For each client request, Path ORAM reads $Z * logN$ blocks, where $logN$ is the tree height; Ring ORAM [54] optimizes Path ORAM to reduce the blocks read per request to $logN$. Figure 1 shows an overview of Ring ORAM's design. The main difference between the two schemes stems from their bucket structures: each Ring ORAM bucket contains exactly $Z+S$ slots, where at most Z slots store encrypted real blocks, and the other slots store encrypted dummy blocks. Ring ORAM operates similar to Path ORAM except for steps 2 and 5:

Step 2: Rather than retrieving Z blocks at each level of a path as in Path ORAM, Ring ORAM reads a single block from each bucket in the path. If a bucket contains the client requested block, it reads that block, and otherwise reads a dummy block, thus reducing the blocks read per request to $logN$. Ring ORAM stores additional metadata (stored at and retrieved obliviously from the server) for each bucket to efficiently identify whether a requested block exists in a given bucket or not. Since only one of the $logN$ blocks retrieved is real, Ring ORAM adds only that block to the *stash*.

Step 5: Path ORAM reads and evicts one path for each request; Ring ORAM decouples the two phases such that a single path is evicted after reading A paths, where A is a configurable parameter. It chooses the path to evict in a deterministic order (i.e., in reverse-lexicographic order) independent of the paths read.

Ring ORAM maintains two additional invariants:

∘ **permuted buckets invariant**: Ring ORAM randomizes the location of the $Z+S$ blocks in a bucket by randomly *permuting* them before pushing them back to the storage. Failing to do so will allow the adversary to distinguish between real and dummy blocks.

∘ **single access invariant**: Similar to Path ORAM, Ring ORAM reassigns every requested block to a randomly chosen path, which ensures that real blocks in a bucket are accessed at most once. Thus, Ring ORAM must also access dummy blocks in a bucket at most once to hide real blocks from dummy ones in a bucket. Ring ORAM achieves this by *invalidating* all accessed blocks—real or dummy—in a bucket by relying on additional metadata to store this. Additionally, to ensure that buckets have sufficient valid dummy blocks to read, the proxy permutes all buckets in a path that have been accessed S times immediately after reading that path. Ring ORAM calls this *early reshuffle*.

Intuitively, Ring ORAM's security stems from its ability to maintain the three invariants: i) request to any block retrieves a random path from the server; ii) each bucket stores real and dummy blocks in randomly permuted slots; and iii) each block - real or dummy - in a bucket is read at most once in between bucket permutations. Moreover, evictions also ensure workload independence by evicting a deterministically chosen path after every A read paths.

Why Treebeard uses Ring ORAM instead of Path ORAM?

Path ORAM retrieves $Z*logN$ blocks for each request and temporarily stores all these blocks in the *stash* until the eviction phase. In contrast, Ring ORAM stores only *one* block per client request in the *stash*, as each request retrieves just one real block from a path. To improve throughput, Treebeard adopts batching (§5). Intuitively, if Treebeard batches B requests, a Path ORAM-like approach would cause the *stash* size to grow to $B*Z*logN$, whereas a Ring ORAM-like approach would result in the *stash* growing to only B, which is why Treebeard uses Ring ORAM. §7.3 experimentally measures the stash size for both approaches and Appendix C provides a detailed analysis of the two alternates.

3 Challenges and approach

Although supporting parallelism, scalability, and fault-tolerance in plaintext databases are well-studied problems, these techniques cannot be trivially translated to oblivious databases, as we discuss in this section with Ring ORAM as the basis. We note that these challenges apply to other ORAM schemes as well.

1. Parallelism Challenges: Serving requests from multiple users in parallel requires synchronization at the proxy. This is necessary to avoid retrieving the same path twice for two concurrent requests accessing the same block. Existing concurrent ORAM schemes [5, 13, 35, 55] track concurrent and conflicting[1] requests and only issue a real read path request for the initial request. Subsequent concurrent requests to the same block initiate *fake read path* operations to random paths in the storage tree. Upon receiving the response from the real read path, the proxy responds to all waiting requests. Whether real or fake, each read path request reads only one path from

[1] Conflict here implies concurrent requests to the same object. Even two concurrent read requests are considered conflicting since they conflict on the path being fetched.

the server. Since in a tree, any two paths have at least one, but likely more, overlapping nodes, existing schemes waste significant WAN bandwidth[2] because each request redundantly fetches the overlapping buckets.

Approach: Treebeard amortizes the proxy-server WAN communication cost by securely *batching* read path requests in fixed time *epochs* and retrieves the overlapping buckets across these paths only once.

2. Scalability Challenges: Ring ORAM consists of a single proxy and a single storage server (i.e., one tree). Scaling the server by creating multiple trees, perhaps stored on different machines, can boost system performance by allowing concurrent accesses to different trees. However, sharding the data blocks and statically assigning each shard to different trees breaks workload independence because a tree with popular blocks will receive substantially more accesses than the other trees. Thus, a scalable ORAM scheme should maintain a new invariant to prevent such leakage: **random tree invariant**, which dictates that the mapping of a requested block to its tree should be random and change after every access.

Scaling the server in Ring ORAM, while helpful, fails to scale the system indefinitely due to the single proxy. We cannot simply scale the number of proxies such that each proxy can read or evict paths to any storage tree because two proxies may end up reading the same block in a bucket. Mitigating such behaviors will require meticulous coordination between the proxies. On the other hand, forcing each proxy to read or evict blocks to a single storage tree will break the random tree invariant. These challenges present three seemingly contradictory requirements: 1) only one proxy should read/evict data to a specific tree at any given time. 2) data blocks should be randomly mapped to storage trees. 3) the proxies should avoid inter-shard coordination.

Approach: Treebeard meets all three requirements by dividing the proxy functionality into two layers: the Stash layer and the ORAM layer. To avoid meticulous inter-shard coordination, each Stash layer process maintains data-dependent state related to a fixed and disjoint set of blocks. To enable a single proxy shard to read/evict data to a specific tree, the ORAM layer stores tree-dependent data such that each storage tree communicates with only one ORAM layer process. For the random tree invariant, Stash processes randomly reassign blocks to trees and evict the blocks to the storage trees via the respective ORAM process. Although not strictly necessary, Treebeard additionally adds a third Router layer to collect and batch requests.

3. Fault-Tolerance Challenges: Ring ORAM lacks mechanisms for handling failures. Addressing failures even with a single proxy, let alone a distributed proxy design, presents challenges. Since Ring ORAM proxy stores critical states such as the encryption key, *stash*, and *position map*, losing its state renders the outsourced data inaccessible. Systems like Obladi [13] back-up the proxy state on the server and retrieve the backup information upon proxy failure. However, this backup approach introduces significant periods of unavailability post failure, as clients must wait for the reinitialization process, prolonged by WAN communication delays.

More importantly, retrying client requests after a failure leads to security issues. Recall that while reading a path, Ring ORAM reads the real block from one bucket and dummy blocks from all the other buckets. Consider a scenario where the proxy crashes after partially reading a path (i.e., accessing a few buckets), and the client retries the request. Before the failure, the proxy may have already accessed the real block along with several dummy blocks. Upon retrying the request after the failure, the same real block but different dummy blocks will be accessed, which violates the single access invariant of Ring ORAM.

Approach: We resolve these issues by introducing fault tolerance into the design of the Stash and ORAM layers. Treebeard leverages a consensus protocol to replicate important states such as the *stash* and *position map*. While replicating stateful data structures is straightforward, replication alone does not resolve the security vulnerability discussed above. To address this, we devise carefully crafted mechanisms to manage failed and retried requests, ensuring that Treebeard upholds all four invariants independent of and across failure boundaries.

4 System and Threat Model

System Model: Treebeard adapts the standard proxy architecture used in academic [5, 13, 35, 55, 60, 62] and production systems [4, 32, 40, 48, 58] wherein an application outsources its storage to the cloud, and relies on a trusted proxy to route client requests. We model Treebeard as an encrypted key-value store that supports single object `Get` and `Put` requests. We leave extending the system to support more complex forms of data for future work.

Treebeard partitions the state and functionality of a centralized proxy and distributes them across multiple machines. The application only needs a small number of servers in its trusted domain to collectively act as the proxy. Figure 2 shows the high-level overview of the system. The trusted domain consists of three layers: the Router, Stash, and ORAM layers. We note that Treebeard can collocate multiple processes across layers on a single machine, as we describe in our experimental setup. Recent private data systems utilize similar multi-layer architectures for scalability [5, 14, 60, 63]. Since Treebeard has no central coordination point, the application can horizontally scale each layer for increased performance.

Components within the trusted domain can crash without prior notice. Crashed components become unresponsive and may eventually rejoin after recovering. Treebeard employs Raft [43] consensus algorithm to replicate data, although any consensus algorithm can be deployed for replication. In Raft, a designated *leader* replica handles requests and propagates

[2]The proxy resides on the application's trusted domain and communicates with the cloud server over WAN.

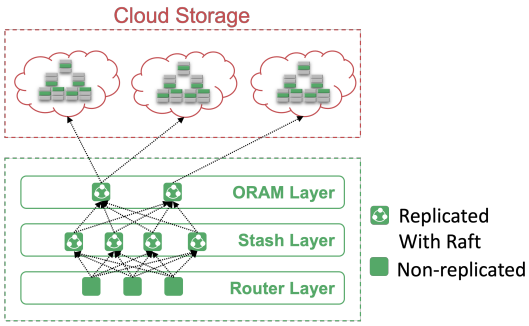


Figure 2: Treebeard System Architecture

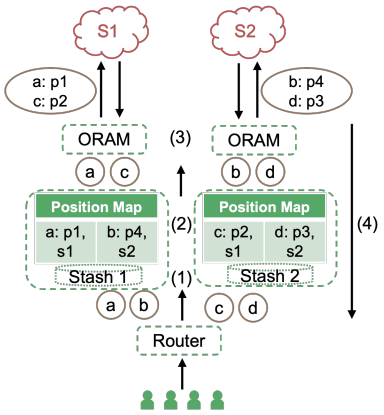


Figure 3: Overview of a Read Operation in Treebeard.

any state changes to other replicas. The Raft protocol, and consequently Treebeard, can tolerate up to f out of $2f+1$ failing components. On the server side, we assume the cloud ensures high availability of data, which we deem to be realistic.

Threat Model: Treebeard protects data from an honest-but-curious adversary who can observe accesses to the storage trees over time. This threat model is stronger than snapshot adversaries who can only take snapshots of the data storage [29, 46, 50, 52] without observing the access patterns. The adversary can mount access pattern attacks using techniques such as *frequency analysis* and l_p*-optimization* [28, 30, 41] to learn *any* non-public information about the data. The adversary can also view all network communications outside the trusted domain and can arbitrarily delay these (encrypted) messages. The adversary cannot control the trusted domain including observing or controlling timing of messages between the clients and the proxy. It also cannot induce machine failures. However, it learns of any failures within the trusted domain. We consider attacks performed by malicious adversaries or timing attacks based on response times to clients [55] or application query workload to be out of scope.

5 Treebeard Design

Overview: We first provide a high-level overview of the system before explaining each layer in detail. Figures 3 and 4 outline how Treebeard reads and writes (i.e., evicts) data to the server, respectively. Clients send their `Get` or `Put` request to an arbitrarily chosen Router.

1. Each Router process collects requests for a fixed time, called an *epoch*, and forwards each request to the Stash process in charge of the requested block by deterministically hashing the requested block's id. In Figure 3, the router sends blocks {a, b} to Stash 1 and {c, d} to Stash 2.
2. Treebeard statically but randomly partitions data blocks across Stash layer processes. Each Stash process stores data-dependent state such as the *stash* and *position map* for a subset of the blocks. Upon receiving a batch of requests from a Router, a Stash process checks its *position map* to determine the paths (on different trees) to be read from the server. It then batches and forwards all the requests accessing the same tree to the corresponding ORAM process. Importantly, it reassigns the requested blocks to randomly chosen paths and trees.
3. The ORAM layer maintains tree-dependent state such as the eviction order and access counts per tree. Each ORAM process manages one or more storage trees, but each tree maps to only one ORAM process. Upon receiving a batch of read path requests from a Stash process, an ORAM process retrieves those paths from the storage tree in parallel. Notably, it reads any overlapping buckets across paths only once, significantly reducing the bandwidth. In the figure, each of ORAM 1 and ORAM 2 retrieves two paths from a different storage tree.
4. The responses travel downstream from ORAM to Router processes, who then respond to the clients.

Figure 4 depicts the multi-path eviction process to highlight how Treebeard writes data back to the server. The ORAM processes oversee the evictions but are ignorant of the blocks in the *stashes*. Consequently, they collect these blocks from Stash processes during evictions as explained below:

1. Each ORAM process monitors the number of read-path operations for its trees and initiates a new eviction every A reads per tree. It then selects k paths to evict using the reverse-lexicographic order of path ids.
2. An ORAM process then requests blocks belonging to the evicting tree from the Stash processes. In Figure 4, the ORAM process requests blocks that were randomly assigned to tree *s1* from the Stash processes.
3. Based on the *position map*, each Stash process sends those blocks from its *stash* that belong to the tree being evicted. In the figure, Stash 1, lacking any blocks in its *stash* for tree *s1*, sends no response, whereas Stash 2 sends blocks {c, d} to the ORAM process.
4. The ORAM process then evicts the k selected paths similar to Ring ORAM's eviction mechanism.

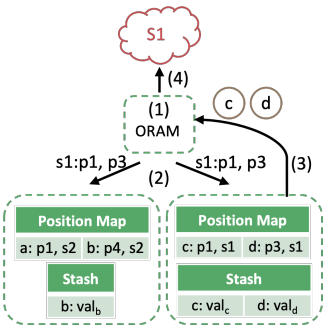


Figure 4: Overview of an Evict Operation in Treebeard.

The subsequent sections delve into the detailed design of each layer. Due to space constraints, the Appendix presents detailed algorithms for each layer.

5.1 Router Layer

The Router processes play a crucial role in improving Treebeard's throughput by batching and forwarding client requests to the Stash processes, as we discuss in this section.

5.1.1 Router Forwarding

Upon receiving client requests, a Router enqueues them and forwards them to Stash processes at the end of an epoch. We employ a straightforward hash-based routing strategy to statically map plaintext keys to Stash processes. This fixed mapping ensures that the Stash process that receives the client request will have the necessary *position map* and *stash* information for that block. Because Router processes are stateless, Treebeard does not replicate them. A crashed router can simply restart after recovery and start processing requests. We discuss the implications of failed requests in §5.1.2.

Epochs: Epochs are commonly used strategies to implement batching [13, 14]. Treebeard defines epochs as fixed-time consecutive intervals that are non-overlapping. Router processes collect all the epoch requests and transmit them collectively to the Stash processes. Once the higher layers process the requests and send the response to Router processes, they collectively forward all responses in an epoch to the respective clients.

Note that Treebeard makes no assumptions about synchronizing epochs across Routers and hence, multiple Routers can be in different epochs. This is crucial since our Router processes are distributed and time drifts are common. Moreover, restarting of a failed Router also causes its epoch boundaries to diverge from other routers.

While the Router layer is not strictly required for scaling or fault tolerance in Treebeard, its absence would force clients to handle key hashing, batch requests per epoch, and route them directly to the appropriate Stash processes. We introduce a stateless Router layer to offload this complexity and simplify client logic.

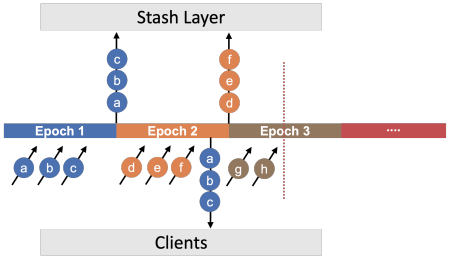


Figure 5: Timeline of a Router with a failure in epoch 3.

5.1.2 Router Failures

A Router failure should ensure protection against two potential issues: losing data consistency and breaking the invariants.

Let's analyze a Router process failure as depicted in Figure 5. At any given time, a client request can be in one of three stages at a Router: (i) it belongs to an ongoing epoch (e.g., epoch e_3). (ii) it belongs to an epoch that has finished collecting requests but hasn't yet sent responses (e.g., epoch e_2). (iii) it belongs to a successfully completed epoch with responses issued to clients (e.g., epoch e_1). Without loss of generality, let's consider the scenario in Figure 5, where a Router successfully responds only to requests from epoch e_1 before it fails. We assume that the clients will retry all other requests belonging to epochs e_2 and e_3. Retrying requests in e_3 is trivial since none of the above layers have seen requests in e_3. However, the higher layers will have received requests in e_2 and will continue to process them. Retrying these requests has no impact on database correctness or security because the Stash layer recognizes any retried requests and does not apply the operation again (§5.2). From an adversary's perspective, we assume it learns of any failures in the Router layer, and hence all requests that must be retried (e.g., requests in epochs e_2 and e_3). However, this leaks no non-trivial access behavior at the storage trees.

5.2 Stash Layer

Processes in the Stash layer maintain data-dependent information. In particular, each Stash process stores *stash* and *position map* for a subset of blocks, with blocks statically but randomly distributed across Stash processes. This section explains the role of the Stash layer in the read and evict phases.

5.2.1 Read phase

Figure 6 depicts how a Stash process handles a batch of requests received from a Router process. (1) For each block b_i in the batch, the Stash process determines whether to send a real or fake request to the ORAM process. This decision

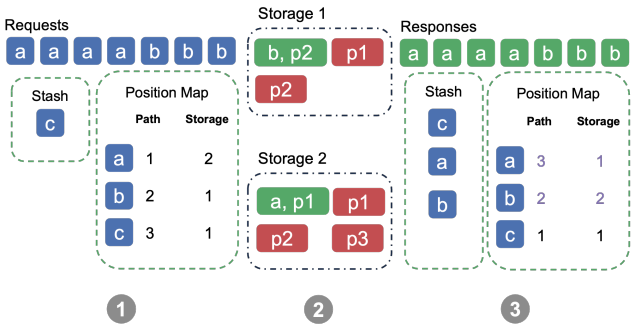


Figure 6: Read Phase Overview at a Stash process *S1*. (1) *S1* gets a batch of seven requests from the Router. It finds the position for blocks *a* (path 1 and storage 2) and *b* (path 2 and storage 1). It then creates two real and five fake read path requests. (2) *S1* sends each set of requests to the ORAM responsible for that storage. (3) *S1* stores the received blocks in the *stash* and responds to the Routers.

depends on whether another concurrent request has already initiated a read-path for b_i; if yes, the Stash process issues a fake read to a random path in a random tree [3] and otherwise a real request for b_i based on the *position map*. A fake read path reads only dummy blocks from all buckets. The Stash process next assigns a new random path and tree for b_i to maintain the random path and random tree invariants, and updates the *position map*. (2) Stash processes then batch all read-path requests belonging to an ORAM process based on the trees and forward each batch to the respective ORAM processes. (3) Finally, upon receiving responses from the ORAM processes, they cache the real values in the *stash* and forward responses of all pending requests to Router processes.

5.2.2 Eviction phase

Stash processes in Treebeard remain ignorant of how or when eviction happens; they merely store blocks in the *stash* and maintain the block's position. An ORAM process, upon deciding to evict paths from a tree, requests a fixed number of blocks from the Stash layer to move the stashed blocks to the server. Stash processes in response simply identify *stash* blocks belonging to the tree being evicted, based on the *position map*, and relay those blocks to the ORAM process.

Stash processes should consider two subtle cases when dealing with blocks sent for eviction. (1) A Stash process cannot simply delete a block from the *stash* after sending it to an ORAM process because the paths being evicted may lack the space to accommodate all the received blocks. Hence, Stash processes wait to receive an ack or a nack from the ORAM layer and can delete only those blocks with an ack. (2) However, not all blocks that receive an ack can be deleted because of possible block updates by concurrent requests triggered *after* eviction began. Otherwise, client updates can be permanently lost. Stash processes track such updates to the *stash* blocks using logical timestamps, i.e., counters. Each block in the *stash* initializes its counter to zero when an eviction starts. If a new request accesses a data block from the *stash*, the process increments the block's counter. Stash processes only remove those blocks that both receive an ack from the ORAM layer and have a zero counter value. Note that we allow each block to participate in one eviction at a time to avoid consistency issues.

5.2.3 Linearizability

A major benefit of Treebeard's modular design is that each data block belongs to a single Stash process and all requests to that block must proceed through the designated Stash process, which helps maintain linearizability. For each ongoing request, Stash processes assign a unique timestamp and adds it to a request log. The process executes its requests in the timestamp order such that for every request $r1$ that arrives before $r2$, $r2$ sees the effect of $r1$. Even when clients issue concurrent requests to a block, Treebeard guarantees a linearization order based on the unique (and sequential) timestamps assigned by a Stash process responsible for that data block.

5.2.4 Handling Failures

Since Stash processes store critical data, Treebeard must ensure their fault tolerance to avoid data loss by replicating important states using Raft [43] in the Stash layer. Each shard in the Stash layer has a leader node and a set of replicas that maintain data-dependent states: *stash* and *position map*, which capture the allocation of data blocks a shard is responsible for. Treebeard synchronously replicates any changes to the *position map* and *stash* occurring during the read or eviction phases. Because (a majority of) the replicas maintain consistent view of these states (i.e., accurate path information for blocks and identical stash blocks), when a shard leader fails, a replica detecting the failure becomes the new leader by running Raft's leader election [43] algorithm and starts processing new requests. While *stash* and *position map* are obvious choices to replicate, Treebeard also replicates the request log to ensure that a Stash process does not re-apply a client operation multiple times (due to clients re-issuing requests)[4]. From the adversary's view, when a leader (or a replica) in the Stash layer fails, it learns of the failure and no other information.

5.3 ORAM Layer

The ORAM layer maintains tree-dependent data including the deterministic eviction order and access counts per tree. Tree-

[3] Sending fake read paths is crucial to maintain indistinguishability between workloads that have skewed accesses vs. uniform accesses (§6)

[4] Most Raft libraries support auto-compaction of the replication log to avoid indefinitely growing logs.

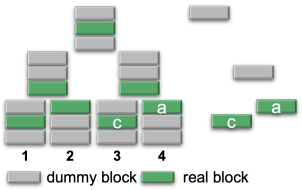


Figure 7: A batch of requests {c:path-3 and a:path-4} reads real blocks a and c and reads one dummy block from the overlapped buckets. It results in four read blocks instead of six blocks in Ring ORAM's read-path operation.

beard assigns each ORAM process a disjoint set of storage trees, and each process communicates with its storage trees to read and evict paths.

5.3.1 Multi-Path Reads

Treebeard introduces a multi-path read optimization to reduce the number of buckets retrieved for a batch of read-path requests. Existing tree-based ORAM schemes [8, 13, 55, 62], including Ring ORAM, retrieve an entire path, consisting of $logN$ buckets, for each client request. However, we observe that many buckets overlap in a batch of read-path requests, especially at the higher levels of a tree. Leveraging this insight, our multi-path retrieval technique avoids redundant accesses to overlapping buckets by identifying and tracking all such buckets based on a set of path ids, reading them only once (illustrated in Figure 7). Importantly, this optimization preserves the security guarantees of sequential Ring ORAM, as the paths requested by Treebeard adhere to the protocol dictated by Ring ORAM; Treebeard simply executes these requests in parallel. Our experimental analysis indicates that this optimization reduces the bandwidth consumption up to *51%* (§7.4).

However, naively reading multiple paths at once introduces security issues depending on the number of blocks accessed from each bucket. For example, if two (or more) real client-requested blocks reside in the same bucket and our approach reads two blocks from this bucket but a single dummy block from the other buckets, the adversary can distinguish between real and fake blocks. We may not be able to read two (or more) blocks from every bucket because some buckets may have less than two valid dummy blocks to read.

We can mitigate this issue with two approaches: (1) Create a maximum batch of S (the number of dummy blocks per bucket) requests such that buckets that do not contain any client-requested blocks will have enough dummy blocks to read. Systems like CURIOUS [5] and ConcurORAM [8] adapt this strategy of limiting the amount of concurrency. (2) Store a single real block in each bucket such that an ORAM process always reads one block from each bucket (real or dummy). Although the latter approach results in larger trees, it benefits from higher performance due to unrestricted batching. Since our goal in designing Treebeard is to push the boundaries of achievable throughput in ORAM systems, Treebeard's implementation chooses the second approach (§7). At first glance, setting $Z = 1$ might appear to undermine the advantages of Ring ORAM compared to Path ORAM. However, Ring ORAM retains its benefits over Path ORAM both in terms of stash sizes (experimentally shown in §7.3) and bandwidth (detailed analysis described in Appendix C).

5.3.2 Multi-Path Evictions

Similar to multi-path reads, Treebeard evicts multiple paths at once to keep the *stash* sizes small at the Stash processes. This is crucial since Treebeard's optimization reads batches of paths at once. Hence, each ORAM process deterministically chooses k paths to evict after every A batched-reads per tree. Similar to Ring ORAM's eviction, an ORAM process first reads all real blocks per bucket in the k paths, while gathering *stash* blocks from Stash processes in parallel. The ORAM process then fills every invalid real block in a path with a *stash* block and retains the valid real blocks in their buckets. It fills the buckets with enough dummy blocks so that all buckets have $Z+S$ blocks. Finally, it permutes each bucket and writes them back to the server. The frequency of eviction, A, plays a vital role in the trade-off between *stash* sizes and performance; we experimentally analyze this trade-off in §7.4.

5.3.3 Handling Failures

Since ORAM processes maintain stateful information including the order of paths evicted and the counter that triggers eviction after A accesses to a tree, each shard leader in the ORAM layer replicates this state synchronously whenever these states are updated. This replication ensures that, in the event of a shard leader failure, Raft [43] can elect a new leader from among the replicas without violating the eviction order or frequency. However, simply replicating the above states does not address subtle challenges that arise when a leader ORAM process crashes either in the read or eviction phases.

Failure while reading a path: The main risk of a failure while reading a path is breaking the single access invariant. Consider a scenario where for a given client request, a leader reads offset i corresponding to a dummy block from bucket B_1 and offset j of a real requested block from bucket B_2. If the leader fails and Raft elects a new leader, the new leader may decide to read a different dummy block at offset i' from B_1 but the same offset j for the real block from B_2. This breaks the single access invariant and leaks the location of the real block requested by the client. This problem may also occur when clients retry requests independent of a leader failure. To mitigate this leakage, Treebeard first identifies all offsets to be read per bucket for a given request and replicates this data *before* reading the path. This way, Treebeard ensures that any retries of a read-path request will fetch the same slots

from each bucket in the path as the previous leader. Since the adversary knows about a leader failure (or retried requests), it only learns of the failure and that the repeated read-path request will read *the same path and offsets again*, which does not leak any non-trivial information.

Failure while evicting: A leader failure during the eviction phase can lead to data loss if not handled correctly. Eviction reads all the real blocks in a path, and shuffles and merges them with the *stash* blocks received from the Stash layer. It then writes the path back starting from the leaf bucket to the root. Consider a case where an unaccessed real block b_i was initially stored at the leaf bucket of a path being evicted. After shuffling in the eviction phase, say another block b_j replaced b_i in the leaf node, which was written back to the server. If the leader process fails before writing back the other buckets, blocks such as b_i will be permanently lost.

Treebeard can trivially mitigate this data loss by replicating all the collected blocks (from the Stash layer and the tree paths). But this can overwhelm the bandwidth because the collected blocks can be many. Treebeard employs a more efficient approach wherein if a bucket has a real block with a valid offset (i.e., unaccessed), its position in the bucket will be shuffled but it will remain in the same bucket; this prevents the above problem. We need to carefully ensure that the offsets of a bucket reflect the change (i.e., the new offset of the real block) and that this change is made persistent at the same time we write the bucket back. Treebeard ensures this by issuing an atomic transaction to the server (exposed by most cloud storage engines) such that the server updates both the bucket and its metadata atomically. Therefore, the above approach guarantees that failures during eviction causes no data loss. Note that this technique does not impact the number of blocks evicted from the *stashes*; it merely restricts what buckets they can be evicted to.

6 Security Overview

We model Treebeard's security with the standard ORAM security model, which must hide *any* non-public information from an honest-but-curious adversary who can observe events such as storage accesses, encrypted network communications with the storage, and process failures.

Security Definition: Consider a sequence of accesses

$$seq = [(\mathsf{bid}_1, r_1, t_1), (\mathsf{bid}_2, r_2, t_2), \cdots, (\mathsf{bid}_m, r_m, t_m)]$$

where bid_i is the requested block, r_i is the router receiving the request, and t_i is the time at which router r_i receives the request. We prove Treebeard's security using an indistinguishability game. Given two sequences of accesses seq_0 and seq_1 with identical router and timing information but different blocks, the game picks a uniformly random challenge bit b and executes seq_b. The adversary observes the resulting events and outputs its guess b' of the challenge bit. We say Treebeard is secure if any polynomial-time adversary can guess the challenge bit with a probability negligibly higher than half.

Theorem 1. *Assuming the underlying encryption scheme is IND-CPA, Treebeard is oblivious.*

Proof Sketch: Due to space constraints, the complete security proof utilizing simulators is described in the Appendix B.

Treebeard's security (including all its optimizations) hinges on its ability to read and evict the *same* paths identified by executing sequential Ring ORAM [54]. Given a sequence of m accesses, *the leaf-ids (i.e., path-ids) accessed across storage trees in Treebeard's multi-path read phases are identical to the m leaf-ids prescribed by Ring ORAM*; Treebeard merely accesses paths in parallel. The fact that Treebeard retrieves the overlapping buckets once in a batch of read-path requests also leaks no additional information since the adversary can trivially derive this information given a set of leaf-ids. Moreover, Treebeard's replication ensures that despite failures, the system always accesses m paths from the server. For eviction, Treebeard evicts k deterministically chosen paths after every A accesses to a tree. The eviction order and the reshuffled buckets are identical to Ring ORAM's protocol. Hence, Treebeard's visible behavior remains identical to Ring ORAM's despite scaling, parallelizing, and replicating the scheme.

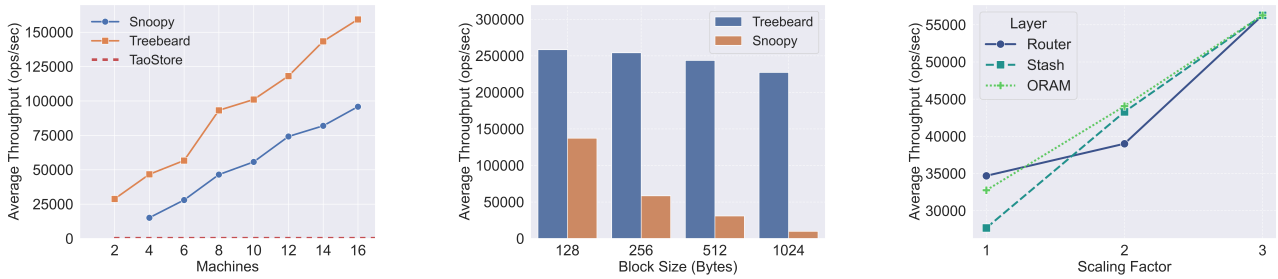
7 Evaluation

Our evaluation answers the following central questions:

- How does Treebeard scale compared to the existing baselines? What is the effect of scaling each layer? (§7.1)
- How does Treebeard compare to fault tolerant baselines? How does it behave under failures? (§7.2)
- How does Treebeard's optimizations affect its stash? (§7.3)
- How does epoch duration affect Treebeard's performance? Is Treebeard *workload-independent*? Can it accommodate growing database sizes?(§7.4)

Implementation: We implemented Treebeard in Go, consisting of approximately 8000 lines of code: https://doi.org/10.6084/m9.figshare.29230676.v1. Treebeard's implementation uses Hashicorp's Raft [23] wherein we customize the protocol by aggregating replication operations to improve replication efficiency. The implementation utilizes the Msgpack binary encoder to serialize and deserialize messages between Raft replicas.

We use a Yahoo! Cloud Serving Benchmark (YCSB)-like [12] client that utilizes YCSB-generated traces to send GET/PUT requests to Treebeard. YCSB ensures that the workload generated reflects real-world workloads. The experiments use Redis [53], an in-memory database, as the untrusted cloud storage, with its pipelining capabilities to efficiently process the read path and evict operations.



(a) Scaling Throughput **(b)** Impact of Block Size **(c)** Impact of Scaling Each Layer

Figure 8: Various aspects of scaling Treebeard and comparison to baselines.

Experimental Setup: The experimental setup deploys Treebeard and its baselines (except Snoopy) on Standard D8s v3 instances (8 vCPUs, 32GB RAM) on Azure [38]. The setup utilizes a more powerful machine (Standard F32s v2 with 32 vCPUs and 64GB RAM) for a multi-threaded client to simulate a large number of clients concurrently issuing `Get`/`Put` requests. To emulate machines within an application's trusted domain performing the proxy functionality, the setup deploys the client and proxy machines in a separate region from the storage servers, with ∼10 ms round-trip latency. This configuration reflects a realistic deployment scenario for applications outsourcing their storage to the cloud and communicating over WAN.

Each experiment consists of a database with 2M key-value pairs. Unless otherwise noted, the block size is 1kB and client queries are drawn from a highly skewed distribution (Zipf 0.99). Each reported data point is an average of 3 runs to account for the performance fluctuation in the cloud. The experiments use a default of $S = 6$, $A = 100$, and $k = 200$ (the values reflect Ring ORAM's recommendation as discussed in Appendix C) and set epoch time per Router to 1ms. Treebeard replicates Stash and ORAM layer processes 3 ways by placing the replicas across multiple availability zones on Azure.

Baselines: We evaluate Treebeard with three baselines, each providing a central feature offered by Treebeard (scalability, fault-tolerance, and concurrency).

1. Snoopy [14] is a scalable baseline that achieves significantly higher throughput compared to previous ORAM systems. It utilizes trusted hardware execution environments (TEEs) to hide access patterns. Snoopy deploys all its components, Load Balancers and Sub ORAMs, in the cloud to remove the centralized proxy. The clients first send their requests to a Load Balancer, which batches requests in an epoch and forwards them to Sub ORAMs. *For each batch and for each Load Balancer, Snoopy linearly scans the entire database*. Although Snoopy requires lower storage than Treebeard (due to S dummy blocks per bucket), scanning, decrypting, and encrypting the entire database per batch incurs higher compute overhead, which has higher dollar cost than storage [39]. TEE enabled machines also cost at least 3× higher than a regular machine on Azure presently. Moreover, Snoopy does not guarantee fault tolerance. Snoopy, however, is more suitable for use cases where the on-premise resources are scarce and the application needs to deploy the entire system in the cloud and the added cost is acceptable.

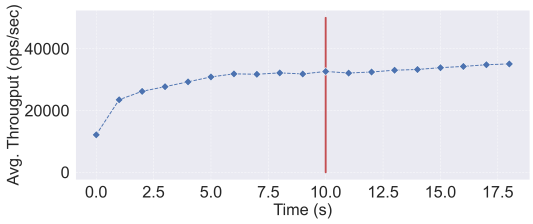
2. QuORAM [35] is a replicated highly available ORAM system. We deploy it with its default replica count of 3. QuORAM is not scalable, but it handles crash failures similar to Treebeard. It replicates proxy-server pairs and allows clients to read and write data to a quorum of replicas. Unlike Raft, QuORAM's replication is decentralized, with clients communicating with replicas rather than replicas communicating with each other.

3. TaoStore [55] is a tree-based concurrent ORAM solution that relies on a single proxy. It is neither scalable nor fault-tolerant.

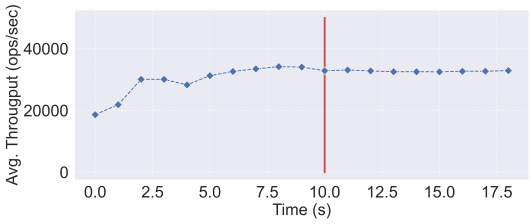
7.1 Scalability Analysis

The first set of experiments measure Treebeard's scaling capability and compares it with Snoopy. In line with Snoopy's default block size [14], these experiments uses blocks of size 160B for both systems. We also used the best parameters provided by Snoopy scripts to run its experiments. Snoopy fixes a latency upper bound for its experiments and reports the highest throughput achieved. It achieves the highest throughput with max 1s latency, which is what we report here, while reporting Treebeard's throughput with similar latency bound.

Figure 8a shows the throughput results of Treebeard compared with Snoopy and TaoStore, with TaoStore acting as a non-scalable but concurrent baseline. The experiment scales both Treebeard and Snoopy from 2 machines up to 16 machines (Snoopy starts with 4). At 16 machines, Treebeard consists of 8 server-side storage machines and 8 machines in the trusted domain executing various collocated proxy layer processes (and their replicas). Whereas for Snoopy, all 16 machines reside on the server. We acknowledge that there exists a fundamental difference in the system model of Treebeard and Snoopy. Our experimental setup takes a best-effort approach in a fair comparison of the two systems.



(a) A Stash leader failure



(b) An ORAM leader failure

Figure 9: **(a, b)** A Stash and ORAM leader failure has no noticeable effect on the throughput.

Comparing the results, unsurprisingly, TaoStore performs poorly compared to the other two systems because of a lack of scalability and read-path optimization, achieving ∼500 ops/sec. Meanwhile, both Snoopy and Treebeard scale linearly, with their throughput increasing steadily with each additional machine. However, the results indicate that Treebeard outperforms Snoopy by **1.7x** on average.

Effect of Block Size: This experiment evaluates Treebeard and Snoopy with increasing block sizes to measure the impact of block sizes on performance, if and when applications manage data blocks larger than 160B, with results depicted in Figure 8b. The experiment loosens the one second latency upper bound since Snoopy's throughput dropped to 0 with 1kB blocks. As the block sizes grow, Snoopy's performance degrades drastically, with its throughput dropping to 10 Kops/sec even with its highest scaling factor. In contrast, although Treebeard's throughput drops with higher block sizes, the reduction is only 12.07% compared to Snoopy's 92.80% drop. We attribute Snoopy's performance drop to the size of the enclave page cache (EPC). Until recently, SGX set its EPC to 128MB. Since Snoopy scans the entire database per batch, fitting 1kB blocks into the EPC likely triggered many paging in and out, causing the performance drop. Although SGX has expanded its EPC memory to gigabytes, we were unable to run Snoopy on a newer SGX machine but we expect its drop to be less drastic if a larger EPC can be utilized.

Per Layer Scaling: This experiment measures the performance impact of scaling each layer in Treebeard, with fixing the storage instances to 6. The experiment scales one layer at a time starting from 1 to 3 processes, while keeping the other two layers with 3 processes. Hence, the end point for each experiment reaches the same maximum of 3 processes per layer. Figure 8c demonstrates the result of scaling each layer. The maximum throughput Treebeard achieves is 55 Kops/s with a scale factor of 3 per layer.

This experiment indicates the Stash layer to have the most prominent impact on performance, with the throughput being the lowest at Stash scale factor of 1. This is because a single Stash process (replicated three ways) will be inundated with the request from all three routers, overwhelming it. The ORAM layer also plays a critical role in Treebeard with a single ORAM process dropping the max throughput by 41.79%.

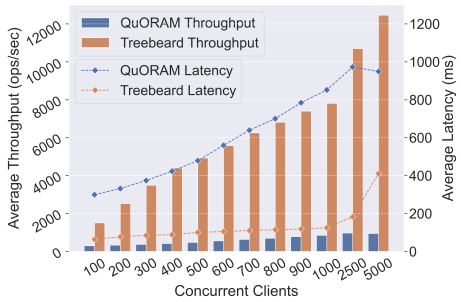


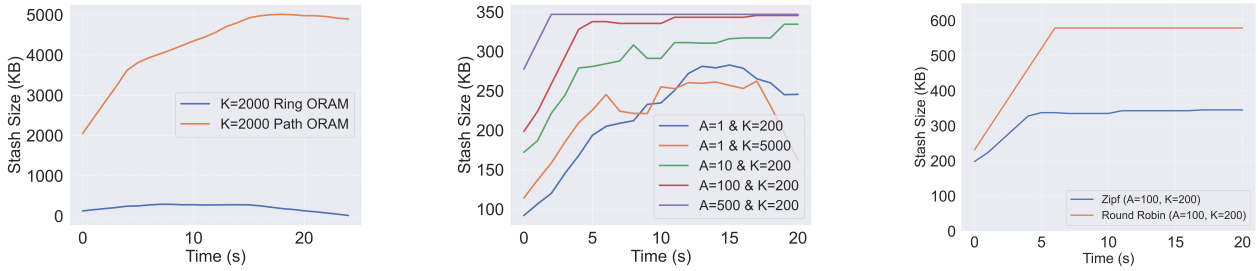
Figure 10: Treebeard and QuORAM throughput

A single ORAM will be overwhelmed by its interactions with 6 storage servers, limiting the system throughput. We anticipated Router layer to have negligible impact on performance because of its simplistic functionalities. However, as seen in Figure 8c, although this layer has the lowest impact compared to the other layers, a scale factor of 1 drops the max system performance by 38.35%. This experiment underscores the impact of a multi-layer architecture in building a scalable oblivious data system.

7.2 Fault-tolerance Analysis

The next set of experiments analyze Treebeard's ability to handle failures. We compare Treebeard with QuORAM [35], a state-of-the-art fault-tolerant and highly available ORAM system, with results in Figure 10. Similar to [35], this experiment measures throughput and latency for both systems while increasing client concurrency. Since QuORAM is non-scalable, this experiment deploys Treebeard with a scale factor of 1.

Treebeard consistently outperforms QuORAM in terms of throughput across all levels of concurrent clients, with the difference increasing as the concurrency increases. QuORAM's throughput saturates at ∼1000 ops/sec, similar to the results reported in [35], but Treebeard continues to achieve higher throughput. At 5000 concurrent clients, Treebeard executes 15 Kops/sec, whereas QuORAM only manages 949.44 ops/sec. The reason for QuORAM's saturation is because, similar to Ring and Path ORAM, it retrieves paths individually for each client request. Whereas, Treebeard batches a set of read-path



(a) Treebeard with Path ORAM vs. Ring ORAM stash size

(b) Impact of A & k on Treebeard's stash size

(c) Impact of varying workload on Treebeard's stash size

Figure 11: Stash experiments of Treebeard.

operations and retrieves the overlapping buckets only once. Hence, as the concurrency increases, Treebeard batches more read-path requests, avoiding redundant bucket reads along with amortizing the cost of communicating to the cloud. This increased throughput however comes at the cost of higher latency, with the latency spiking after 1000 concurrent clients.

Impact of Leader Failure: Figure 9 shows the effect of leader failures in both the Stash and ORAM layers (note that replica failure has no effect on system performance). This experiment consists of three processes in each layer with the Stash and ORAM processes replicated three ways (thus, $f = 1$). Upon a leader failure, Raft elects a new leader, who then continues to process requests. To analyze Treebeard's behavior upon leader failures, this experiment introduces leader failures at the 10^{th} second. Figure 9a and 9b depict the system performance when a Stash and an ORAM leader fail, respectively. The results indicate that Treebeard observes no performance drop upon a leader failure in either of the stateful layers. This experimentally underscores the fault tolerance and high availability aspect of Treebeard and how a new leader helps to recover from failures.

7.3 Stash Experiments

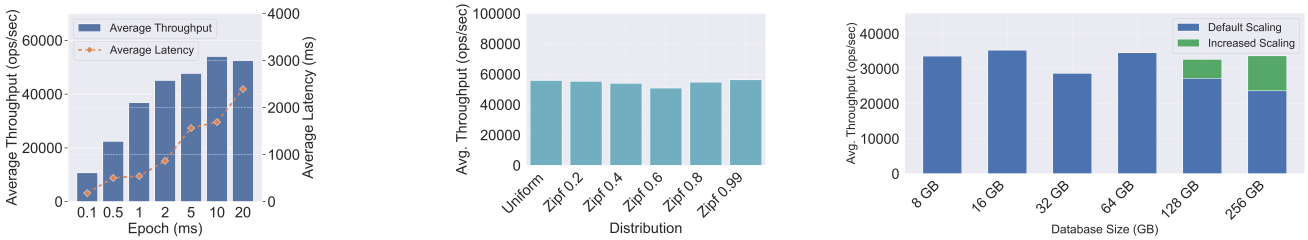
As noted in §5.3, Treebeard's design fixes real blocks per bucket, Z, to one. This, along with Treebeard's optimization to read multiple paths at once, impacts its stash size by potentially limiting the number of blocks it can evict from *stash*es. Despite Treebeard compensating for this by executing multi-path evictions instead of Ring ORAM's single path eviction, a natural question that arises is: would Path ORAM with $Z > 1$ be a better choice for Treebeard?

Path ORAM vs. Ring ORAM: We created a Path ORAM version of Treebeard to verify the benefits of Ring ORAM over Path ORAM when Ring ORAM's $Z = 1$. This experiment also highlights the modular design of Treebeard since we only had to change the ORAM layer to accommodate Path ORAM's scheme. In this version, Treebeard continues to execute multi-path reads and stores all retrieved blocks (i.e., $Z * logN$ blocks per path) in their respective *stash*es. In the subsequent eviction phase, triggered immediately after the read phase, Treebeard evicts k paths. This version sets $Z = 4$ because Path ORAM has optimal stash size with $Z = 4$ [62].

To ensure a fair comparison, we set the rate of eviction $A = 1$ in Ring ORAM to match Path ORAM's eviction rate and used $k = 2000$ for both implementations because Treebeard on average read 2000 paths in one batch. Figure 11a compares the stash size of Treebeard built with Path ORAM versus Ring ORAM measured at one second intervals with a moving average of a window size of ten. The results show that Treebeard built with Ring ORAM maintains a relatively small stash size and peaks at 347 blocks. However, with Path ORAM, the stash size peaks at 5,306 blocks, which is **15x** the stash size use of Ring ORAM. This underscores Treebeard's choice to use Ring ORAM despite setting $Z = 1$.

Impact of A and k on Treebeard: This experiment measures the impact of eviction rate on Treebeard's stash size and performance, with Figure 11b showing the results.

We expected both A, the frequency with which Treebeard evicts the path, and k, the number of paths evicted in parallel, to play an equally critical role in bounding the stash size. However, the experiment with $A = 1$ and $k = 200$ vs $k = 5000$ indicates that k's role is less prominent in bounding the stash size. In contrast, $A = 1$ consumes 21.43% lower stash size than $A = 100$, indicating a more prominent role. This is because $A = 1$ is the most aggressive eviction policy with an ORAM process triggering eviction after each multi-path read operation. A also impacts performance: the throughput at $A = 100$ is 32 Kops/sec compared to 27 Kops/sec at $A = 1$ – an 18.5% reduction. With $A = 100$ and $k = 200$ (default values for other experiments), the stash size is at most 350 KB per Stash process. Although this stash size is larger than a sequential ORAM scheme, *the system batches and serves thousands of requests in parallel*. We believe that 350 KB stash size to achieve 32 Kops/sec is a reasonable trade-off. Importantly, Treebeard can always achieve stash sizes equivalent to Ring ORAM by imposing a batch size of one at the cost of significantly reducing its throughput.



(a) Impact of Epoch Size on performance **(b)** Workload Independence in Treebeard **(c)** Increasing database size in Treebeard

Figure 12: Internal experiments of Treebeard.

Impact of varying workload on stash size: The stash experiments thus far use a highly skewed input distribution (Zipf=0.99). Under such skew, requested objects often already reside in the *stash*, reducing the number of real read-path requests and helping keep the stash size small. In contrast, uniform or round robin access patterns lead to more real read-path requests, potentially increasing stash size. To evaluate this, clients in this experiment generate round robin accesses, with the results shown in Figure 11c. This experiment uses the values of A=100 and k=200 (default values used in other experiments). As illustrated, the maximum stash size in a shard increases from 350 blocks under skewed access to 580 blocks with round robin accesses. Nonetheless, this accounts for only ∼0.028% of the database size, which we consider a reasonable trade-off for enabling concurrency and scalability. To understand whether this low percentage holds even when the database size increases, we measured the stash size when the database size increases from 2GB (i.e., 2M key-values with 1kB values) to 256GB. For the 256GB database size, the maximum stash size was 40k blocks for skewed and 41k blocks for round robin accesses, both accounting for ∼0.015% of the database size, indicating that the stash size remains small relative to the database size.

7.4 Treebeard Internal Experiments

The final set of experiments evaluate the effectiveness and observable security of Treebeard.

Bandwidth Analysis: In this experiment, we simulated Treebeard and Ring ORAM's read path protocols to illustrate the benefits of our multi-path read optimization for a database with 2M entries on a tree of 1M paths (tree height of 21). Table 1 provides simulation results of average number of blocks retrieved per client request for 10,000 requests per run. With a batch of size 2000, which is the average batching size observed in our experiments with an epoch of 1ms, *Treebeard's multi-path eviction reduces bandwidth by **51%** compared to Ring ORAM*. We also note that this relative bandwidth improvement will hold for other tree-based ORAM schemes.

Epoch Analysis: This experiment investigates the influence of epoch sizes on system performance. Epoch size refers to the duration that a Router waits to collect the requests before relaying them to the the Stash layer. To remove client concurrency as the bottleneck in measuring the full potential of an epoch size, we increase the client concurrency as the epoch size increases. With a scale factor of 3, the results shown in Figure 12a indicate that the throughput improves with the increase in epoch sizes, reaching a peak of 52 Kops/sec at 10 ms epoch. Beyond this, the throughput decreases due to increased concurrency, which overloads the system. The increase in throughput, however, comes at the cost of increased latency. This indicates a throughput vs. latency trade-off of Treebeard: higher epoch sizes enable more batching, which increases the throughput, but require client requests to wait in Router queues for larger durations.

	Ring ORAM	Trb. (B=5)	Trb. (B=100)	Trb. (B=500)	Trb. (B=2000)
Buckets	21.00	15.46	14.46	12.14	10.14

Table 1: Comparison on number of buckets retrieved by Ring ORAM and Treebeard (Trb.) with different batch sizes (B).

Workload Independence: Figure 12b shows Treebeard's throughput for different skewness – from uniform to 0.99 Zipfian – in the client workload. The results show negligible impact on system performance for different types of workloads. We attribute the small difference in throughput for different Zipf configurations to the experimental noise introduced by multi-tenancy in the cloud. This highlights the experimental aspect of the workload independence guarantees provided by Treebeard despite its distributed design. It also provides an experimental evidence that Treebeard is secure from attacks utilizing workload patterns [5, 55] that Oblivistore [60] is vulnerable to.

Increasing Database Size: This experiment measures Treebeard's throughput as the size of the database increases starting from 8GB to 256GB[5] as shown in Figure 12c. By default, this experiment uses a scale factor of 3 at the proxy, while

[5]We note that the experiments cannot simulate extremely large databases by mocking the storage functionalities because the correctness of RingORAM, and hence Treebeard, requires reading and writing correct metadata.

varying the number of cloud database instances from 5 to 10, increasing the tree node count eightfold as the database size grows. As seen in the figure, the throughput remains roughly the same (standard deviation of 2333.78 ops/sec) until the database size reaches 128GB. For 128GB and 256GB database sizes, the throughput starts to reduce despite provisioning additional storage servers, indicating that Treebeard is bottlenecked at the ORAM layer. We test this hypothesis by increasing the number of ORAM processes from 3 to 5 and 8 for 128GB and 256GB respectively. Additional resources at the proxy enables Treebeard to bridge the performance gap (as shown in the stacked bars in Figure 12c). This experiment highlights that Treebeard can scale to retain similar throughput as the database size increases to hundreds of gigabytes.

8 Related Work

The literature on ORAM schemes is extensive (e.g., [1, 54, 57, 62, 64]) so we mainly discuss ORAM systems that achieve: parallelism, scalability, or fault-tolerance. Additionally, this section does not discuss oblivious data systems under threat models weaker than ORAM's (Pancake [18], Waffle [36], and Shortstack [63]). We also do not discuss complex query processing systems on outsourced information, such as Opaque [70], Panda [37], ObliDB [16], and Obscure [22].

Parallel ORAM: While many theoretical constructions of parallel ORAM exist [2, 6, 10, 11, 42], the discussion is limited to practical ORAM schemes. Shroud [33] is an ORAM scheme that enables parallelism by reading multiple buckets in a single path at once. However, it processes client requests sequentially. PrivateFS [65] and ConcurORAM [8] serve client requests in parallel in a proxy-less setting. The client directly communicates with the cloud to synchronize concurrent requests, which differs from Treebeard. This incurs increased WAN communication overheads, negatively impacting system performance. Oblivistore [60] and TaoStore [55] process concurrent requests under asynchronous networks. But both schemes issue one read-path operation per client request, without batching them.

Meanwhile, Obi [66] retrieves a fixed number of paths in a batch but it does not deduplicate overlapping buckets, while Fork Path [68] deduplicates overlapping buckets but processes requests sequentially. Treebeard differs from these works in batching a dynamic number of concurrent read path requests and deduplicating the overlapping buckets, both choices contributing to its increased performance as discussed in §7.4.

Obladi [13] is the most optimized concurrent scheme and compares closely with Treebeard. It employs an epoch-based logic to batch a set of requests and amortize the cost of executing ORAM. Given a set of read-path operations, Obladi tracks the dependencies between operations to serialize them based on conflicts and executes non-conflicting operations in parallel. This mechanism introduces higher complexity compared to Treebeard's approach of reading multiple paths' buckets in a batch and reducing redundancy. Moreover, Obladi relies on a single proxy that cannot be scaled.

Scalable ORAM: Many ORAM schemes [31,34,59,61,67] consider multi-server ORAM where they scale the storage. However, the proxy remains the main bottleneck in these systems. Oblivistore [60], CURIOUS [5] and Snoopy [14] are the only current schemes to scale the ORAM processing units. Since §7 discusses Snoopy in detail, we focus the discussion on Oblivistore and CURIOUS. Both systems achieve scalability by distributing the data across multiple ORAM proxies (or sub-ORAMs), each responsible for one shard of data. However, for security they rely on a single oblivious load balancing unit to communicate with the ORAM nodes, which becomes a scalability bottleneck. Moreover, Oblivistore has proven security vulnerabilities [5,55] that reveal the amount of contention in client workloads. SEAL [15] is a recent ORAM system that can distribute ORAM processing across different processes but it leaks information on the request distribution across different ORAM processes to achieve parallelism. Moreover, none of the these works [5, 14, 15, 60] ensure fault tolerance (although some discuss possible extensions).

Fault-tolerant ORAM: Only two existing ORAM data stores, Obladi [13] and QuORAM [35], incorporate fault tolerance in their design. Obladi assumes that cloud storage is persistent and handles crash faults in the proxy layer. Obladi pushes its states to the cloud storage at epoch boundaries and retrieves them upon failures. However, the system becomes unavailable until the proxy recovers and reinitializes its state. QuORAM [35] mitigates Obladi's unavailability problem by replicating the server-proxy pairs and ensures high availability even when faults occur. Although QuORAM's replication handles failures well, the system cannot scale.

9 Conclusion

We introduce Treebeard - a high throughput ORAM system that scales linearly with the number of machines while ensuring fault tolerance. Treebeard utilizes a novel multi-layer design that disaggregates the functionality and state maintained by a single ORAM proxy across processes in multiple layers. Importantly, it ensures that no layer introduces scalability bottlenecks. Treebeard meticulously addresses information leakage introduced by each of its core features - parallelism, scalability, and fault tolerance. Primarily, it employs a multi-path read mechanism to read a set of paths from the server in a batch and avoids redundant fetching of overlapping buckets across those paths. It outperforms state-of-the-art scalable and fault-tolerant baselines and executes 160k ops/sec with 16 machines. As future work, Treebeard can be extended to provide transactional guarantees and support complex relational and analytical queries.

Acknowledgments

We gratefully acknowledge NSERC for grant RGPIN-2023-03448. This project was made possible in-part through the support of the National Cybersecurity Consortium and the Government of Canada (CSIN) (project number 2024-1261). We also thank the anonymous reviewers and shepherd for their insightful comments.

Ethics Considerations

No sensitive or personally identifiable data was used in Treebeard experiments. All experiments were done using synthetically generated data that do not contain private or sensitive user information. Our code development was entirely performed on infrastructure fully operated by the research team with no unauthorized access to sensitive information or real-world systems. Our evaluations were performed on Microsoft Azure but since the experiments utilized synthetic data, this did not reveal any sensitive information to Azure.

The techniques described in this paper enhance data security and privacy by addressing challenges in scalability and fault tolerance for ORAM-based systems. However, we recognize the potential risks of dual use, where malicious actors could exploit insights from Treebeard 's architecture to protect their data from other actors. We acknowledge this risk, but the benefits that the community gains from this system in advancing secure and privacy-preserving technologies outweigh the potential risks.

Beyond technological advancements and economic considerations, existing research recognizes the social implications of privacy breaches, particularly for marginalized communities. Sensitive and identifiable information collected by businesses in sectors such as banking and healthcare is often at risk. By devising a secure data management system, Treebeard aims to safeguard individual user data from intentional or unintentional misuse.

We have thoroughly analyzed the ethical considerations and ensured alignment with the principles outlined by the USENIX Security 2025 guidelines. Our methodology prioritizes the advancement of security and privacy while responsibly addressing potential risks.

Open Science

All the implementation and experiments are open-source and publicly available at https://doi.org/10.6084/m9.figshare.29230676.v1.

References

[1] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. OptORAMa: Optimal oblivious RAM. J. ACM, 70(1):1–70, 2022.

[2] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Enoch Peserico, and Elaine Shi. Optimal oblivious parallel RAM. In Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pages 2459–2521, 2022.

[3] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, and Elaine Shi. Oblivious RAM with worst-case logarithmic overhead. J. Cryptology, 36(2):7, 2023.

[4] Baffle. https://baffle.io.

[5] Vincent Bindschaedler, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, and Yan Huang. Practicing oblivious access on cloud storage: the gap, the fallacy, and the new way forward. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS), pages 837–849, 2015.

[6] Elette Boyle, Kai-Min Chung, and Rafael Pass. Oblivious parallel RAM and applications. In Theory of Cryptography: 13th International Conference (TCC), pages 175–204, 2016.

[7] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS), pages 668–679, 2015.

[8] Anrin Chakraborti and Radu Sion. ConcurORAM: High-throughput stateless parallel multi-client ORAM. In 26th Annual Network and Distributed System Security Symposium (NDSS), 2019.

[9] T-H Hubert Chan, Yue Guo, Wei-Kai Lin, and Elaine Shi. Oblivious hashing revisited, and applications to asymptotically efficient ORAM and OPRAM. In Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, pages 660–690, 2017.

[10] T.-H. Hubert Chan and Elaine Shi. Circuit OPRAM: unifying statistically and computationally secure ORAMs and OPRAMs. In Theory of Cryptography: 15th International Conference (TCC), pages 72–107, 2017.

[11] Binyi Chen, Huijia Lin, and Stefano Tessaro. Oblivious parallel RAM: improved efficiency and generic constructions. In Theory of Cryptography: 13th International Conference (TCC), pages 205–234, 2016.

[12] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud

serving systems with ycsb. In Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC), pages 143–154, 2010.

[13] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. Obladi: Oblivious serializable transactions in the cloud. In 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI), pages 727–743, 2018.

[14] Emma Dauterman, Vivian Fang, Ioannis Demertzis, Natacha Crooks, and Raluca Ada Popa. Snoopy: Surpassing the scalability bottleneck of oblivious storage. In Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP), pages 655–671, 2021.

[15] Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Saurabh Shintre. Seal: Attack mitigation for encrypted databases via adjustable leakage. In 29th USENIX Security Symposium (USENIX Security 20), pages 2433–2450, 2020.

[16] Saba Eskandarian and Matei Zaharia. Oblidb: Oblivious query processing for secure databases. Proc. VLDB Endow., 13(2):169–183, 2019.

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

[18] Paul Grubbs, Anurag Khandelwal, Marie-Sarah Lacharité, Lloyd Brown, Lucy Li, Rachit Agarwal, and Thomas Ristenpart. Pancake: Frequency smoothing for encrypted data stores. In 29th USENIX Security Symposium, pages 2451–2468, 2020.

[19] 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 Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pages 315–331, 2018.

[20] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. Learning to reconstruct: Statistical learning theory and encrypted database attacks. In 2019 IEEE Symposium on Security and Privacy (SP), pages 1067–1083, 2019.

[21] Zichen Gui, Oliver Johnson, and Bogdan Warinschi. Encrypted databases: New volume attacks against range queries. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, pages 361–378, 2019.

[22] Peeyush Gupta, Yin Li, Sharad Mehrotra, Nisha Panwar, Shantanu Sharma, and Sumaya Almanee. Obscure: Information-theoretically secure, oblivious, and verifiable aggregation queries on secret-shared outsourced data. IEEE Transactions on Knowledge and Data Engineeting (TKDE), 34(2):843–864, 2022.

[23] Hashicorp Raft. https://github.com/hashicorp/raft.

[24] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: ramification, attack and mitigation. In 19th Annual Network and Distributed System Security Symposium (NDSS), 2012.

[25] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'neill. Generic attacks on secure outsourced databases. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS), pages 1329–1340, 2016.

[26] Evgenios M Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. Data recovery on encrypted databases with k-nearest neighbor query leakage. In 2019 IEEE Symposium on Security and Privacy (SP), pages 1033–1050, 2019.

[27] Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. Improved reconstruction attacks on encrypted data using range query leakage. In 2018 IEEE Symposium on Security and Privacy (SP), pages 297–314. IEEE, 2018.

[28] Marie-Sarah Lacharité and Kenneth G Paterson. A note on the optimality of frequency analysis vs. ℓ_p-optimization. Cryptology ePrint Archive, 2015.

[29] Marie-Sarah Lacharité and Kenneth G Paterson. Frequency-smoothing encryption: preventing snapshot attacks on deterministically encrypted data. Cryptology ePrint Archive, 2017.

[30] Jingwei Li, Chuan Qin, Patrick PC Lee, and Xiaosong Zhang. Information leakage in encrypted deduplication via frequency analysis. In 2017 47th Annual IEEE/IFIP international conference on dependable systems and networks (DSN), pages 1–12. IEEE, 2017.

[31] Zheli Liu, Bo Li, Yanyu Huang, Jin Li, Yang Xiang, and Witold Pedrycz. Newmcos: towards a practical multi-cloud oblivious storage scheme. IEEE Transactions on Knowledge and Data Engineering (TKDE), 32(4):714–727, 2019.

[32] Lookout. https://www.lookout.com.

[33] Jacob R. Lorch, Bryan Parno, James Mickens, Mariana Raykova, and Joshua Schiffman. Shroud: Ensuring private access to Large-Scale data in the data center. In 11th USENIX Conference on File and Storage Technologies (FAST 13), pages 199–213, San Jose, CA, February 2013. USENIX Association.

[34] Steve Lu and Rafail Ostrovsky. Distributed oblivious ram for secure two-party computation. In Theory of Cryptography Conference (TCC), pages 377–396. Springer, 2013.

[35] Sujaya Maiyya, Seif Ibrahim, Caitlin Scarberry, Divyakant Agrawal, Amr El Abbadi, Huijia Lin, Stefano Tessaro, and Victor Zakhary. QuORAM: A quorum-replicated fault tolerant ORAM datastore. In 31st USENIX Security Symposium, pages 3665–3682, 2022.

[36] Sujaya Maiyya, Sharath Chandra Vemula, Divyakant Agrawal, Amr El Abbadi, and Florian Kerschbaum. Waffle: An online oblivious datastore for protecting data access patterns. Proceedings of the ACM on Management of Data (SIGMOD), 1(4):1–25, 2023.

[37] Sharad Mehrotra, Shantanu Sharma, Jeffrey D. Ullman, Dhrubajyoti Ghosh, Peeyush Gupta, and Anurag Mishra. PANDA: partitioned data security on outsourced sensitive and non-sensitive data. ACM Trans. Manag. Inf. Syst., 11(4):23:1–23:41, 2020.

[38] Microsoft Azure. https://azure.microsoft.com/en-ca.

[39] Microsoft Azure Cost Calculator. https://azure.microsoft.com/en-us/pricing. Accessed Jan 10, 2025.

[40] Navajo Systems. https://tinyurl.com/yc4z5nyf.

[41] Muhammad Naveed, Seny Kamara, and Charles V Wright. Inference attacks on property-preserving encrypted databases. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pages 644–655, 2015.

[42] Kartik Nayak and Jonathan Katz. An oblivious parallel RAM with O($\log^2$ N) parallel runtime blowup. Cryptology ePrint Archive, page 1141, 2016.

[43] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In 2014 USENIX Annual Technical Conference (Usenix ATC 14), pages 305–319, 2014.

[44] Simon Oya and Florian Kerschbaum. Hiding the access pattern is not enough: Exploiting search pattern leakage in searchable encryption. In USENIX Security Symposium, pages 127–142, 2021.

[45] Simon Oya and Florian Kerschbaum. IHOP: Improved statistical query recovery against searchable symmetric encryption through quadratic optimization. In 31st USENIX Security Symposium, pages 2407–2424, 2022.

[46] Antonis Papadimitriou, Ranjita Bhagwan, Nishanth Chandran, Ramachandran Ramjee, Andreas Haeberlen, Harmeet Singh, Abhishek Modi, and Saikrishna Badrinarayanan. Big data analytics over encrypted datasets with seabed. In OSDI, volume 16, pages 587–602, 2016.

[47] Sarvar Patel, Giuseppe Persiano, Mariana Raykova, and Kevin Yeo. PanORAMa: Oblivious RAM with logarithmic overhead. In IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS), pages 871–882, 2018.

[48] Perspecsys. https://tinyurl.com/45ubwnef.

[49] Benny Pinkas and Tzachy Reinman. Oblivious RAM revisited. In Advances in Cryptology–CRYPTO 2010: 30th Annual Cryptology Conference, pages 502–519, 2010.

[50] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. Arx: an encrypted database using semantically secure encryption. Cryptology ePrint Archive, 2016.

[51] Rishabh Poddar, Stephanie Wang, Jianan Lu, and Raluca Ada Popa. Practical volume-based attacks on encrypted databases. In 2020 IEEE European Symposium on Security and Privacy (EuroS&P), pages 354–369. IEEE, 2020.

[52] Raluca Ada Popa, Catherine MS Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In Proceedings of the twenty-third ACM Symposium on Operating Systems Principles (SOSP), pages 85–100, 2011.

[53] Redis. https://redis.io/.

[54] Ling Ren, Christopher W. Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. Constants count: Practical improvements to oblivious RAM. In USENIX Security Symposium, pages 415–430. USENIX Association, 2015.

[55] Cetin Sahin, Victor Zakhary, Amr El Abbadi, Huijia Lin, and Stefano Tessaro. TaoStore: Overcoming asynchronicity in oblivious data storage. In 2016 IEEE Symposium on Security and Privacy (SP), pages 198–217, 2016.

[56] Amin Setayesh, Cheran Mahalingam, Emily Chen, and Sujaya Maiyya. Treebeard: A scalable and fault tolerant ORAM datastore. Cryptology ePrint Archive, 2025.

[57] Elaine Shi, T-H Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with O($(logn)^3$) worst-case cost. In Advances in Cryptology–ASIACRYPT 2011: 17th International Conference on the Theory and Application of Cryptology and Information Security, pages 197–214, 2011.

[58] Skyhigh Networks. https://www.skyhighsecurity.com.

[59] Emil Stefanov and Elaine Shi. Multi-cloud oblivious storage. In Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, pages 247–258, 2013.

[60] Emil Stefanov and Elaine Shi. Oblivistore: High performance oblivious cloud storage. In 2013 IEEE Symposium on Security and Privacy (SP), pages 253–267, 2013.

[61] Emil Stefanov, Elaine Shi, and Dawn Xiaodong Song. Towards practical oblivious RAM. In 19th Annual Network and Distributed System Security Symposium (NDSS), 2012.

[62] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS), pages 299–310, 2013.

[63] Midhul Vuppalapati, Kushal Babel, Anurag Khandelwal, and Rachit Agarwal. SHORTSTACK: distributed, fault-tolerant, oblivious data accesss. In 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI), pages 719–734, 2022.

[64] Xiao Wang, Hubert Chan, and Elaine Shi. Circuit oram: On tightness of the goldreich-ostrovsky lower bound. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pages 850–861, 2015.

[65] Peter Williams, Radu Sion, and Alin Tomescu. PrivateFS: A parallel oblivious file system. In Proceedings of the 2012 ACM SIGSAC Conference on Computer and Communications Security (CCS), pages 977–988, 2012.

[66] Zhiqiang Wu and Rui Li. Obi: a multi-path oblivious ram for forward-and-backward-secure searchable encryption. In NDSS, 2023.

[67] Jinsheng Zhang, Qiumao Ma, Wensheng Zhang, and Daji Qiao. Tskt-oram: A two-server k-ary tree oram for access pattern protection in cloud storage. In MILCOM 2016-2016 IEEE Military Communications Conference, pages 527–532. IEEE, 2016.

[68] Xian Zhang, Guangyu Sun, Chao Zhang, Weiqi Zhang, Yun Liang, Tao Wang, Yiran Chen, and Jia Di. Fork path: improving efficiency of oram by removing redundant memory accesses. In Proceedings of the 48th International Symposium on Microarchitecture, pages 102–114, 2015.

[69] 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 Symposium, volume 2016, pages 707–720, 2016.

[70] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), pages 283–298, 2017.

A Security Definition

We adapt the standard ORAM definition, which informally states that a secure ORAM hides the following from an adversary: i) the exact data accessed by a user, ii) timing information on when an object was last accessed, iii) whether clients are accessing the same object or different objects, iv) the access pattern such as skewed vs. uniform, and v) whether a client requests a read or write operation.

We leverage the indistinguishability game to formally define Treebeard's obliviousness guarantees. But first, we formally define an ORAM scheme. An ORAM scheme consists of two modules ORAM = {Encode, OClient}. Encode shuffles and encrypts a given database D to produce D_{enc} along with the secret key K. An external server stores D_{enc} and a stateful module OClient stores K. In Treebeard, multiple storage servers store different partitions of D_{enc}, whereas the proxy unit (consisting of processes across three layera) retains the secret key K.

OClient answers all the read/write requests sent by clients after communicating with the storage server. Since Treebeard's OClient is distributed across multiple processes, and the system consists of multiple routers that receive data access requests, we define a sequence of accesses as:

$$seq = [(\mathsf{bid}_1, r_1), (\mathsf{bid}_2, r_2), \cdots, (\mathsf{bid}_m, r_m)]$$

where bid_i indicates the block requested in the i^{th} access and r_i indicates the Router process that receives the request. Note that since ORAM schemes behave identically for both read

and write requests, *seq* only lists the block ids without indicating the type of operation or operation values.

Security definition: Consider two different access sequences seq_0 and seq_1 with the same length and access timings:

$$seq_0 = [(\mathsf{bid}_1, r_1, t_1), (\mathsf{bid}_2, r_2, t_2), \cdots, (\mathsf{bid}_m, r_m, t_m)]$$

$$seq_1 = [(\mathsf{bid}'_1, r_1, t_1), (\mathsf{bid}'_2, r_2, t_2), \cdots, (\mathsf{bid}'_m, r_m, t_m)]$$

where t_i indicates the time at which router r_i received the request. Informally, an ORAM scheme is secure if a polynomial time adversary cannot distinguish between seq_0 and seq_1 by observing the accesses to the server. We define this formally with an indistinguishability game $\mathcal{G}$ with the following steps:

1. The game begins by sampling a uniformly random bit $b \in \{0,1\}$, called the challenge bit.
2. The game then issues seq_b to OClient, who then executes the ORAM scheme and accesses data on the storage.
3. The adversary observes the resulting events and outputs its guess b' of the challenge bit. The game returns *true* if the guess is correct and otherwise *false*.

We say that an ORAM scheme is oblivious if

$$Pr[\mathcal{G}_{ORAM} \Rightarrow true] - \frac{1}{2} = negl(\lambda) \quad (1)$$

where λ is the security parameter. In other words, an ORAM scheme is secure if any polynomial-time adversary A can guess the challenge bit with probability negligibly higher than half. This implies that an ORAM scheme is secure if any two sequences of requests are indistinguishable to the attacker.

B Security Proof

Ring ORAM upholds obliviousness by indiscriminately executing the following steps for any input workload: every client request reads a random path from the server, every S accesses to a bucket leads to its (early) reshuffling, and every A read paths causes the eviction of a deterministically chosen path. Proving the security of Treebeard hinges on arguing that scaling, replicating, and parallelizing Ring ORAM continues to uphold the above access behavior.

Treebeard uses an indistinguishability based security game defined in Appendix A. Our proof builds on the following central properties of Treebeard.

- At initialization, Treebeard uniformly and randomly partitions the data and stores each partition on a storage tree.
- The adversary observes that for m accesses, m' random paths, indicated by leaf-ids, are read across all storage trees. Treebeard's security builds on the fact that the m' leaf-ids it reads correspond to the same m' leaf-ids prescribed by sequential Ring ORAM.
- The adversary also learns that every A accesses to a storage tree triggers the eviction (reading and writing) of k deterministically chosen paths. Additionally, it also learns the reshuffled buckets that have been accessed S times.

In the game $\mathcal{G}$, the adversary's goal is to correctly derive the challenge bit b by observing any and all information visible to the adversary. For Treebeard to be secure, we need to show that the adversary has a negligible advantage over randomly guessing the challenge bit. We achieve this by constructing a simulator that only has access to publicly available information of the system. This implies that the simulator initializes the database with encryptions of dummy values (e.g., zero value) and replaces block values in each request also with dummy values. Due to the IND-CPA security of the underlying encryption scheme, the adversary cannot learn whether the encrypted blocks correspond to real values or dummy values. Note that since Treebeard reveals the request load and the proxy unit configuration (i.e., number of Router, Stash, and ORAM layer processes), the simulator must also *simulate* the same configuration to ensure security.

At a high level, the simulator simulates the following:

i) A router simulator collects requests in epochs and forwards the requests to a Stash simulator. The Stash simulator picks a random path from a random tree to fetch for each request in the batch, and forwards the read-path requests to an ORAM simulator. ii) The ORAM simulator reads the randomly chosen paths from the randomly chosen trees such that the overlapping buckets across the paths are retrieved only once. iii) It also simulates eviction phase (i.e., reading Z blocks per bucket and writing the path back) by evicting k deterministically chosen paths after A accesses to a storage tree. Note that none of these steps depend on the blocks (or their values) requested by clients. Additionally, the simulator adds delays equivalent to the communication delays incurred for replication or communication between layers.

The simulator is a theoretical ideal that is stateful and does not fail. Note that since Treebeard builds on top of Ring ORAM, our proof assumes that all steps executed using Ring ORAM are secure and that Ring ORAM provides both real and simulated functions to initialize a (sub-)tree, shuffle a bucket, access a path, and evict a path. Since Treebeard assumes that the response times to clients are hidden from the adversary, the attack presented in [55] does not apply to Treebeard; protecting against such an attack utilizing client response times requires a centralized request sequencer [55], which becomes a scalability bottleneck. Furthermore, Treebeard's security and database correctness guarantees are only upheld when at most f out of $2f+1$ replicas of an Stash or ORAM process fails (as is the case with plaintext databases). The functionality of Treebeard's various layers and the simulator's design in detail is discussed in the extended report [56].

C Ring ORAM vs Path ORAM in Treebeard

Recall that Treebeard's implementation sets $Z = 1$ for the underlying Ring ORAM protocol. A natural question that arises is: *for Treebeard, would Ring ORAM continue to hold the benefits over Path ORAM if $Z = 1$?* We answer this in the affirmative for two reasons.

1) Effect on stash size: Treebeard's multi-path read optimization retrieves multiple paths from the server. Suppose this value is B, i.e., Treebeard retrieves B paths from the server. With Ring ORAM like designs, this equates to B real blocks being read and added to the *stash*. Whereas, with Path ORAM, this will be $Z * logN$ potential blocks added to the *stash*. For equivalence, even setting $Z = 1$ in Path ORAM incurs a $logN$ factor overhead in the *stash* size. We experimentally evaluated this by replacing Ring ORAM with Path ORAM in Treebeard and the results in §7.3 indicate a significantly higher *stash* for Path ORAM compared to Ring ORAM (with $logN$=21).

2) Impact on bandwidth: One might argue that setting $Z = 1$ will cause Ring ORAM's overall bandwidth usage to be much higher than Path ORAM's. The answer to this depends on the parameters (A, S) set for Ring ORAM. Assuming $Z = 1$ for both Path and Ring ORAMs, evicting a path back in Path ORAM incurs $logN$ bandwidth whereas Ring ORAM incurs a higher bandwidth of $logN + (1 + S)logN$ (the first $logN$ is to first read the real blocks in the path being evicted). However, *Path ORAM evicts one path for every path read whereas base Ring ORAM evicts one path every A paths read.*

In Treebeard, this translates to: every $A * B$ paths read, k paths are evicted where B is the number of paths read in a batch. Therefore the eviction bandwidth for B requests for Path ORAM is $B * logN$ whereas for Ring ORAM it is $(2 + S)logN * k/(A * B)$. Note that Ring ORAM, and hence Treebeard, also performs early reshuffle wherein each bucket in a path that was read will be permuted after accessing that bucket S times. Ring ORAM indicates their overall bandwidth use to be $(L+1) + (L+1) * (2 * Z + S)/A * (1 + poisson.cdf(S, A))$, where L is the height of the tree [54]. In Treebeard, $A = A * B/k$ because in each read access to the server, Treebeard reads some B paths but instead of evicting one path, the system evicts k paths. In our experiments, the average batch size observed was $B = 2000$, and we set $A = 100$, $k = 200$, and $S = 6$. We have $N = 2M$, giving a tree height of ($logN =$) 21. Plugging in these values in Ring ORAM's bandwidth calculation, the overall bandwidth overhead (for both read and evict phases) per client request is 23x, with a tree of height 21. Whereas for Path ORAM, the bandwidth overhead per request is $2logN$, which is 42x for a tree of height 21. This indicates that even with $Z = 1$, choosing Ring ORAM helps reduce the bandwidth overhead for the parameters values our experiments used.