Optimizing ORAM Datastores for Scalability, Fault Tolerance, and Performance

by

Amin Setayesh

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2025

© Amin Setayesh 2025

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Oblivious RAM (ORAM) mitigates access pattern attacks, where adversaries infer sensitive data by observing access patterns. These attacks can compromise privacy even when data is encrypted. While ORAM ensures privacy by obfuscating these patterns, its adoption in cloud environments faces significant challenges, particularly related to scalability, fault tolerance, and performance.

This thesis presents Treebeard: an ORAM-based datastore that addresses these challenges through a novel multi-layer architecture. Unlike traditional ORAM systems that rely on a centralized proxy to manage data access and security, this design separates responsibilities across specialized layers that are independently scalable. Each layer handles distinct functionalities and efficiently batches and processes requests. Treebeard facilitates horizontal scaling, and adds fault tolerance by eliminating single points of failure.

Experiments show that Treebeard is scalable, highly performant, and fault-tolerant. Treebeard outperforms existing ORAM systems in terms of throughput while simultaneously addressing scalability and fault tolerance in its design.

Acknowledgements

I would like to express my gratitude to my supervisor, Sujaya Maiyya. Her guidance, support, and dedication were essential in helping me complete this thesis. She not only provided valuable feedback and insights for my research but also encouraged my personal growth throughout this journey. Her continuous support kept me motivated during challenging times. I would also like to thank my committee members, Florian Kerschbaum, and Ken Salem for reading my thesis and their advice.

I am also thankful to my friends who stood by me throughout this process. Your support during difficult times made this experience enjoyable.

Finally, I want to thank my family. Their constant love, even from afar, provided me with the strength and determination to keep going. I am deeply grateful for their support.

To everyone who contributed to this journey, thank you for being a part of it.

Dedication

To my parents and my sister, for believing in me long before I believed in myself.

Table of Contents

Author's Declaration ii

Abstract iii

Acknowledgements iv

Dedication v

List of Figures ix

List of Tables x

1 Introduction 1

1.1 Contributions 2

1.2 Organization 4

2 Background and Challenges 5

2.1 Oblivious RAM 5

2.2 Obliviousness Challenges 8

3 Design **11**

3.1 System and Threat Model . . . 11

3.2 Treebeard Architecture . . . 13

3.2.1 Router Layer . . . 15

3.2.2 Stash Layer . . . 16

3.2.3 ORAM Layer . . . 20

3.3 Ring ORAM vs Path ORAM in Treebeard . . . 23

4 Security **25**

4.1 Security Definition . . . 25

4.2 Security Proof . . . 27

4.2.1 Initialization . . . 28

4.2.2 Router layer . . . 29

4.2.3 Stash layer . . . 30

4.2.4 ORAM layer . . . 32

5 Implementation & Evaluation **36**

5.1 Implementation . . . 36

5.2 Experimental Setup . . . 37

5.3 Baselines . . . 37

5.4 Evaluation . . . 38

5.4.1 Scalability Analysis . . . 38

5.4.2 Fault-tolerance Analysis . . . 40

5.4.3 Stash Experiments . . . 42

5.4.4 Treebeard Internal Experiments . . . 43

6 Related Work **46**

6.1 Parallelism in ORAM Systems 46

6.2 Scalability in ORAM Systems 51

6.3 Fault tolerance in ORAM Systems 52

7 Conclusion **53**

7.1 Future Research Directions 53

References **55**

List of Figures

2.1 Overview of Ring ORAM. 7

3.1 Treebeard System Architecture 12

3.2 Overview of a Read Operation in Treebeard. 14

3.3 Overview of an Evict Operation in Treebeard. 15

3.4 Timeline of a Router with a failure in epoch 3. 17

3.5 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. 18

3.6 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. 20

5.1 Various aspects of scaling Treebeard and comparison to baselines. 39

5.2 Treebeard and QuORAM throughput 41

5.3 **(a, b)** A Stash and ORAM leader failure has no noticeable effect on the throughput. 42

5.4 Stash and internal experiments of Treebeard. 45

List of Tables

5.1 Comparison of the number of buckets retrieved by Ring ORAM and Treebeard (Trb.) with different batch sizes (B). 44

Chapter 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 [20, 17, 21, 22, 6, 14, 16, 18, 23, 36, 43, 54, 37] observe the access frequency of each encrypted data block and leverage the learned frequency distribution to uncover plaintext data. For example, a recent attack [22] reconstructed the plaintext values of an encrypted database with relative errors as low as $0.003\% - 2.9\%$.

Oblivious RAM or ORAM [15] 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 [13, 29, 47, 51, 50, 7, 12]. 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 keys and coordinate accesses between clients and the server [51, 12, 47, 50, 29, 46, 4]. Proxy-based ORAM systems offer an effective solution to serve concurrent requests in a multi-user setting [47, 12, 29, 50, 4]. 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 [50] and CURIOUS [4] 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 [4, 47]. Snoopy [13], 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 limited to select 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 [29] 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 (2.2).

1.1 Contributions

We propose Treebeard, the first ORAM-based storage system to achieve both scalability and fault tolerance. Treebeard utilizes a tree-based RingORAM [46]-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 [12, 47, 29, 50, 4], 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.

Lastly, to guarantee fault tolerance and high availability (under a bounded number of failures), Treebeard replicates processes in each layer using Raft [35]. Implementing and evaluating Treebeard demonstrates its ability to scale the throughput linearly with the number of machines, similar to Snoopy [13], 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 [29]. It achieves 13.8x of QuORAM's throughput even without scaling.

1.2 Organization

In Chapter 2, the necessary background regarding ORAM systems and challenges on extending previous ORAM systems are discussed. In Chapter 3, we present the system model and the system architecture with each layer in detail. Chapter 4 discusses the security of our design and provides a formal proof. In Chapter 5, we perform experiments to compare Treebeard with previous ORAM systems. Chapter 6 summarizes the related work. Finally, Chapter 7 presents the conclusions and discusses potential future work.

Chapter 2

Background and Challenges

Oblivious RAM (ORAM) systems are cryptographic protocols designed to hide access patterns to data stored on untrusted servers, preserving privacy even when the data itself is encrypted. This chapter provides an explanation of two ORAM systems: Path ORAM and Ring ORAM. It also explains the challenges on extending existing ORAM systems to support parallelism, scalability, and fault-tolerance.

2.1 Oblivious RAM

While a variety of ORAM schemes exist [8, 39, 41, 1, 2, 5, 9, 10, 34], tree-based ORAMs such as Path ORAM [51] have gained substantial popularity due to their low bandwidth overhead and simple design. Treebeard utilizes a Ring ORAM [46]-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 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 the stash-blocks 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 [46] optimizes Path ORAM to reduce the blocks read per request to $logN$. Figure 2.1 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

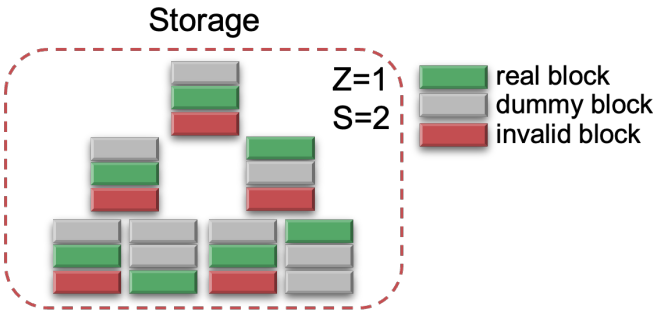


Figure 2.1: Overview of Ring ORAM.

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 determinist 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.

2.2 Obliviousness Challenges

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 [4, 47, 12, 29] 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 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

[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.

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

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 [12] 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.

Chapter 3

Design

3.1 System and Threat Model

System Model: Treebeard adapts the standard proxy architecture used in academic [51, 12, 47, 50, 29, 4] and production systems [3, 32, 27, 48, 40] 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 3.1 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 [13, 52, 50, 4]. 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 [35] consensus algorithm to replicate data, although any consensus algorithm can be deployed for replication. In Raft, a designated *leader* replica handles requests and propa-

gates 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.

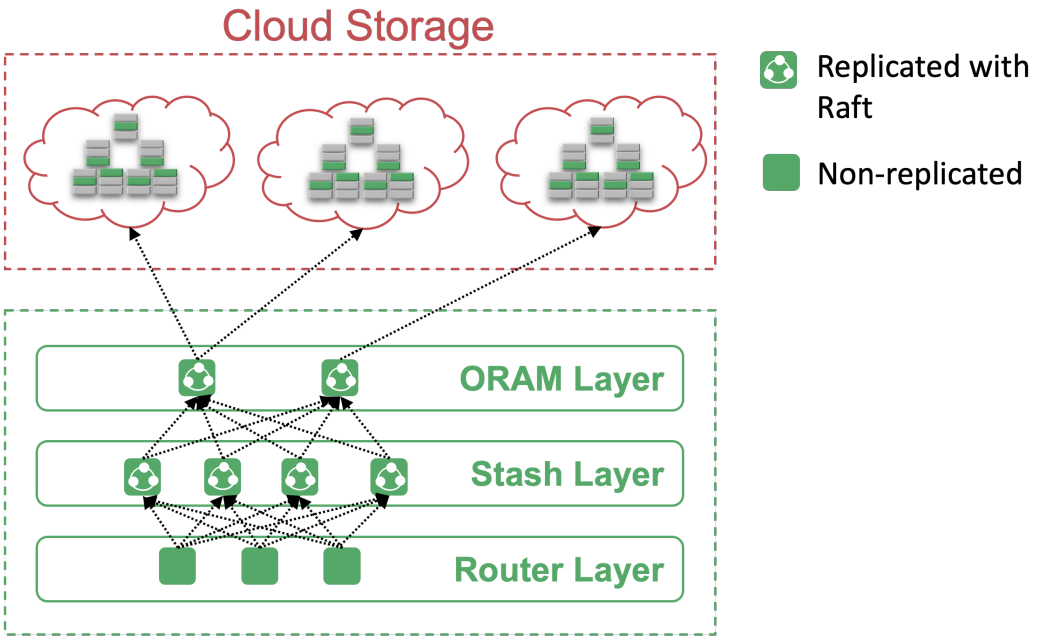


Figure 3.1: Treebeard System Architecture

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 [44, 42, 25, 38] without observing the access patterns. The adversary can mount access pattern attacks using techniques such as *frequency analysis* and l_p*-optimization* [33, 24, 26] 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 application query workload or response times

to clients[1] to be out of scope.

3.2 Treebeard Architecture

Overview: We first provide a high-level overview of the system before explaining each layer in detail. Figures 3.2 and 3.3 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 3.2, 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 3.3 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

[1]Protecting against attacks utilizing client response times requires a centralized request sequencer [47], which becomes a scalability bottleneck.

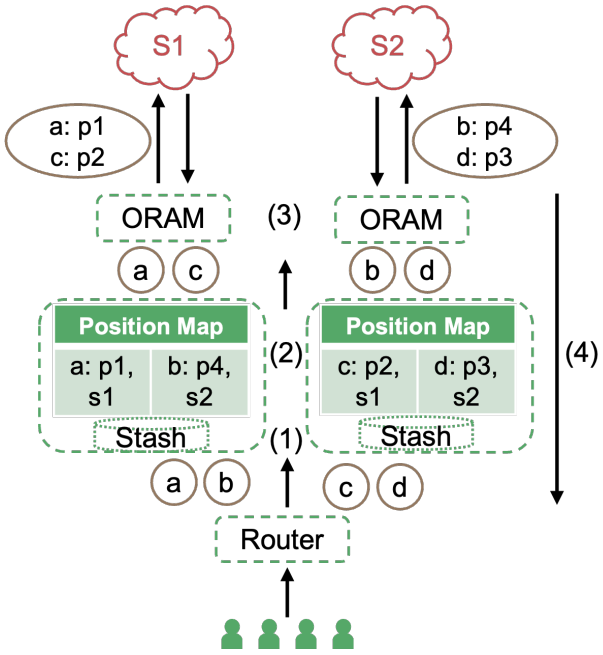


Figure 3.2: Overview of a Read Operation in Treebeard.

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 3.3, 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.

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

3.2.1 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.

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 section 3.2.1.

Epochs: Epochs are commonly used strategies to implement batching [13, 12]. 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.

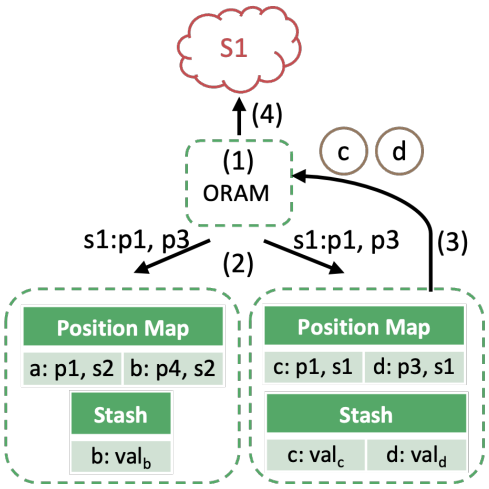


Figure 3.3: Overview of an Evict Operation in Treebeard.

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.

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 3.4. 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 response received (e.g., epoch e_1). Without loss of generality, let's consider the scenario in Figure 3.4, where a Router successfully responds only to requests from epoch e_1 before it fails. We assume that the clients can 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. Clients re-issuing these requests has no effect (i) on database correctness because linearizability is maintained by Stash processes (section 3.2.2); and (ii) on security because this case is similar to Treebeard handling repeated requests to a single block across different routers. Moreover, all four security invariants are maintained by the other two layers, which ensure fault tolerance. From an adversary's perspective, we assume it learns of any failures in the Router layer, and consequently all requests that must be retried (e.g., requests in epochs e_2 and e_3). However, this leaks no information about the contents of the requests or non-trivial access behavior at the storage trees.

3.2.2 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.

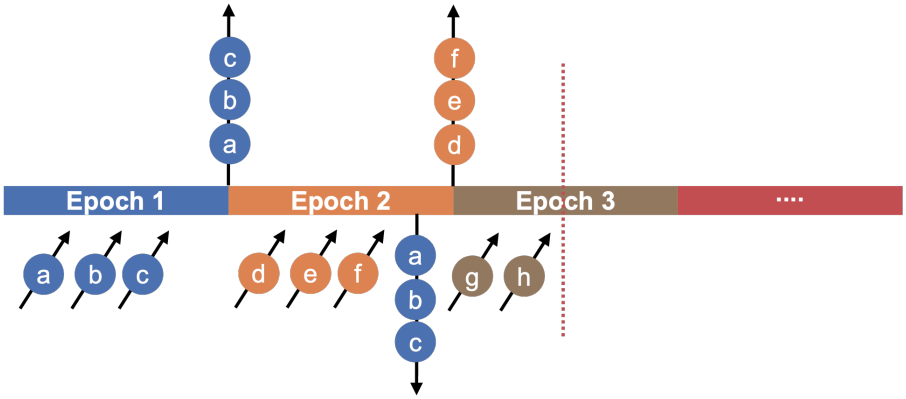


Figure 3.4: Timeline of a Router with a failure in epoch 3.

Read phase

Figure 3.5 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 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 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.

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

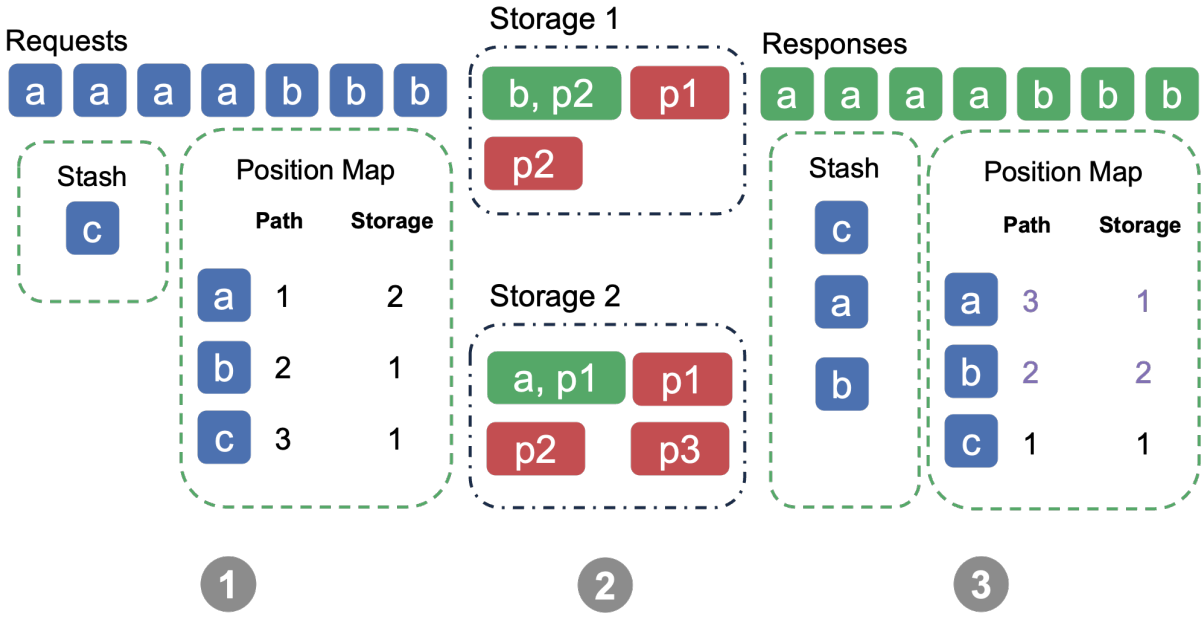


Figure 3.5: Read Phase Overview at a Stash process *S1*. **1.** *S1* gets a batch of seven requests from the Router. It finds the position for blocks *a* (path 1 and storage 2) and *b* (path 2 and storage 1). It then creates two real and five fake read path requests. **2.** *S1* sends each set of requests to the ORAM responsible for that storage. **3.** *S1* stores the received blocks in the *stash* and responds to the Routers.

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.

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 maintain an ongoing request map of request-id to its timestamp and the response after receiving it from an ORAM process. A Stash process executes its requests in the timestamp order such that for every request $r1$ that arrives before $r2$, $r2$ sees the effect of $r1$. Upon a Router failure, when clients retry their requests using the same request-ids, Stash processes identify it as a repeated request and do not process it further. If and when a response exists in the map, the processes simply forward that downstream. And Router processes signal the Stash processes to remove the ongoing requests from the map after sending back the responses to the clients. Note that deletions to the ongoing requests and *stash*es are independent of each other.

Handling Failures

Since Stash processes store critical data, Treebeard must ensure their fault tolerance to avoid data loss by replicating important states using Raft [35] in the Stash layer. While the *stash* and *position map* are obvious choices to replicate, we also need to replicate the ongoing requests map in a Stash process to avoid security issues. Consider a scenario where the Raft leader fails after sending a real read-path operation to an ORAM process. If Treebeard 'forgets' about this request and issues a fresh real request for that block, the server will observe two read-path requests touching the same real block but different dummy blocks, thus breaking the single access invariant. Therefore, along with the *stash* and *position map*, Treebeard replicates ongoing requests and ensures that the new Raft leader will refrain from re-issuing read-path requests. This ensures that for any m client requests, Treebeard reads exactly m paths independent of failures, which is crucial for Treebeard's security as we will discuss in section 4.2. Thus, when a leader (or a replica) in the Stash layer fails, the adversary learns of the failure and no other information.

3.2.3 ORAM Layer

The ORAM layer maintains tree-dependent data including the deterministic eviction order and access counts per tree. Treebeard assigns each ORAM process a disjoint set of storage trees, and each process communicates with its storage trees to read and evict paths.

Multi-Path Reads

Treebeard introduces a simple yet novel multi-path read optimization to reduce the number of buckets retrieved for a batch of read-path requests. Existing tree-based ORAM schemes [51, 47, 7, 12], 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 3.6). 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%* (section 5.4.4).

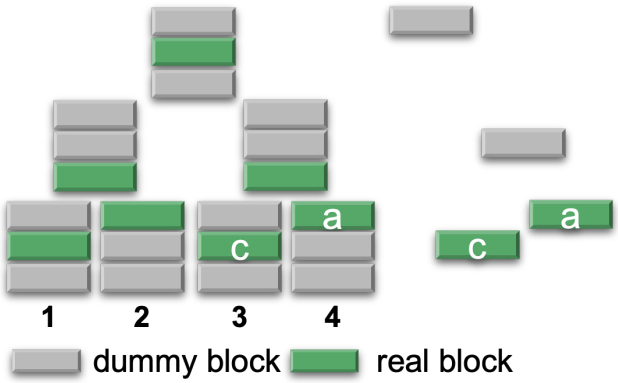


Figure 3.6: A batch of requests {c:path-3 and a:path-4} reads real blocks a and c and reads one dummy block from the overlapped buckets. It results in four read blocks instead of six blocks in Ring ORAM's read-path operation.

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 [4] and ConcurORAM [7] 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 (section 5.4). 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 section 5.4.3) and bandwidth.

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 section 5.4.4.

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, Treebeard replicates the ORAM layer using Raft. However, simply replicating the state 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 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. To mitigate this leakage, Treebeard first identifies all offsets to be read per bucket and replicates this data *before* reading the path. This way, Treebeard ensures that if a new leader retries the read-path step, it fetches the same slots from each bucket in the path as the previous leader. Since the adversary knows about a leader failure, it only learns that the leader failed while reading a path and that the new leader 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.

3.3 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 section 5.4.3 indicate a **15x** higher *stash* for Path ORAM compared to Ring ORAM (the increase factor is close to $logN$=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, k) 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. Therefore the eviction bandwidth for B requests for Path ORAM is $BlogN$ 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. 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.

Treebeard is based on Ring ORAM but differs from the original Ring ORAM design in several ways: multi-path reads, setting Z to one, and multi-path evictions. These modifications affect the behavior and assumptions of the original Ring ORAM design. In particular, we focus on the impact of fixing Z to one while performing multi-path reads and evictions.

Setting $Z = 1$ increases Ring ORAM's bandwidth usage compared to Path ORAM. With $Z = 1$, both Path ORAM and Ring ORAM read $Mlog(N)$ blocks from the tree, where M is the number of paths read. However, during evictions, Path ORAM writes back to $klog(N)$ blocks, while Ring ORAM writes to $k(1 + S)log(N)$ blocks, as it still needs to fill full buckets containing $(Z = 1) + S$ blocks. This results in less efficient bandwidth overhead for Ring ORAM compared to its original design.

Another important implication is the stash size. In our multi-path read step, Ring ORAM only reads the B requested blocks and adds them to the stash. In contrast, Path ORAM reads all $Mlog(N)$ blocks from the paths and writes them to the stash for later eviction. During the eviction phase, Treebeard writes back blocks from the stash to $kpaths$. Since M, the number of paths read, is usually larger than k to keep the throughput high, Path ORAM gathers significantly more blocks in its stash compared to Ring ORAM.

Chapter 4

Security

4.1 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 $\mathsf{ORAM} = \{\mathsf{Encode}, \mathsf{OClient}\}$. 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 $\mathsf{OClient}$ 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.

$\mathsf{OClient}$ answers all the read/write requests sent by clients after communicating with the storage server. Since Treebeard's $\mathsf{OClient}$ 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 $\mathsf{OClient}$, 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

$$Pr[\mathcal{G}_{ORAM} \Rightarrow true] - \frac{1}{2} = negl(\lambda) \tag{4.1}$$

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.

4.2 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 4.1. 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 requests received by the system, 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.

We next discuss the functionality of Treebeard's various layers and the simulator's design in detail. 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 [47] does not apply to Treebeard. 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).

4.2.1 Initialization

Treebeard

To initialize a database, our system takes as input a database D of plaintext key-value pairs and a set of configurations indicating the number of Router, Stash, and ORAM layer processes, along with the replication count. The configuration also dictates all other parameters defined in Table 1. Treebeard then creates all the necessary processes based on the configuration and executes them across P physical machines. Treebeard then invokes the Encode function to generate a secret key K and encrypted key-value pairs D_{enc}. The system randomly shuffles and partitions D_{enc} across T sub-trees, where each sub-tree calls Ring ORAM's initialization (which ensures dummy padding at each bucket and creating the necessary metadata) with $Z = 1$. Ring ORAM initialization outsources each sub-tree along with its metadata to the external storage. Treebeard assigns each sub-tree to one of the ORAM layer processes and shares K across all ORAM processes. Treebeard also statically shards all plaintext keys across different Stash layer processes using a keyed

hash function, whose key is shared with the Router processes. A Stash process stores the *position map* for all plaintext keys it is responsible for, along with an empty *stash*. At this point, the external server and all the layers of Treebeard are initialized.

Simulator

Similar to Treebeard, the simulator receives the dataset D and a set of configurations consisting of all the publicly known configuration details. The simulator is a stateful entity that stores necessary information to simulate publicly visible behavior of Treebeard in a data or workload-independent manner. During initialization, the simulator replaces all block values with dummy entries and encrypts them, generating D'_{enc}, and stores the encryption key. The simulator partitions D'_{enc} into T sub-trees and uses Ring ORAM's simulated initialization function to outsource each of the T sub-trees. This completes the simulator's data initialization process.

4.2.2 Router layer

Treebeard

Each Router process in Treebeard collects client requests in sequential epochs, with each epoch spanning a fixed time interval. A Router process batches requests per Stash process based on the keyed hash function and forwards the batches. Upon receiving the responses from higher layers, it relays the responses to all the clients. Upon a Router process failure, the system takes no action, while the clients retry failed requests with the same request ids. The security proof assumes that the two sequences seq_0 and seq_1 account for repeated requests (i.e., only the final request sent to a router is captured). Since the adversary learns of the failed routers and can see the two access sequences seq_0 and seq_1, which indicates the requests received by each router, the adversary learns all the failed and retried requests. Retrying of failed requests has no impact on the security or correctness of the system, as discussed in section 3.2.1. Algorithm 1 presents the pseudocode of Treebeard's Router layer.

Algorithm 1 Router algorithm

1: **Inputs:** B_i: a batch of client requests for the ith epoch, e_i: the current epoch, H: the hash function.
2: **Variables:** S: an array to store requests per Stash process
3: **function** FORWARD(B_i, e_i, H)
4: $S \leftarrow []$
5: **for** $req \in B_i$ **do**
6: $s \leftarrow H(req.blockID)$
7: $S[s] \leftarrow S[s] \cup req$
8: **end for**
9: **for** $s \in (1, |S|)$ **do**
10: Send $S[s]$ batch of requests to Stash process s
11: **end for**
12: **end function**

Simulator

A simulated Router layer simulates multiple routers collecting requests in epochs and forwarding the requests to a simulated Stash layer. Since the adversary cannot see response times, the simulator does not communicate back to the clients. The simulator also replaces the real values in every request with dummy values.

4.2.3 Stash layer

Treebeard

The Stash layer holds data-dependent information (such as the *stash* and *position map*) and is responsible for ensuring the **random path** and **random tree** invariants. For each block in a batch of requests received from a Router process, a Stash process decides whether to issue a real read path or a fake read path request based on the existence of concurrent and conflicting requests. Importantly, similar to Ring ORAM, each Stash process assigns a requested block to a randomly chosen path on a randomly chosen tree. Treebeard's security hinges on the fact that this randomizing of blocks to paths (and trees) is identical to Ring ORAM. This implies that the ***paths (i.e., leaf-ids) accessed by Treebeard is identical to the paths read by sequential Ring ORAM; Treebeard merely batches***

them and executes them in parallel. Failure of a Stash process has no externally visible behavior since Treebeard replicates each process using Raft, and Raft elects a new leader process with negligible impact on performance (section 5.4.2). By replicating all the ongoing requests, Treebeard ensures that each client request triggers exactly one read path request (real or fake) from the storage even under failures. Algorithm 2 presents the high-level pseudocode for Treebeard's Stash layer.

Algorithm 2 Stash algorithm

Inputs: B: batch of client requests received from a Router process.
State: *req_log*: request log for tracking ongoing requests, *pos_map*: position map (mapping of block id to randomly assigned tree and path id), *stash*: stash holding not yet evicted blocks.
function QUERY(B)
    for $req \in B$ **do**
        $req_log[req.blockID] \leftarrow req_log[req.blockID] \cup req$
        if $len(req_log[req.blockID]) > 1$ **then** ▷ *An ongoing request for this block already exists*
            $p \xleftarrow{\$} |paths|$
            $s \xleftarrow{\$} |storages|$
            Send fake read path request for path p on storage s
        else
            $p, s \leftarrow pos_map[req.blockID]$
            $pos_map[req.blockID] \xleftarrow{\$} |paths|, |storages|$
            Send real read path request for path p on storage s
            Update *stash* based on the response
        end if
        Remove request req from req_log
    end for
end function

Simulator

A Stash layer simulator receives batches of requests from the Router simulator and for each request, the Stash simulator chooses a random path from a random tree to fetch. The stateful simulator retains the request to path (and tree) mapping in a map mp, which

is necessary to handle retried requests. When clients retry a request, the simulator first checks if *mp* has an entry for the request id. If yes, the simulator refrains from issuing a new read path request, and otherwise issues a read path request to a random path in a random tree. This behavior mimics the real system (section 3.2.2) without making any data dependent decisions.

4.2.4 ORAM layer

Treebeard

The ORAM layer maintains tree-dependent data with each process responsible for communicating with a disjoint set of trees. This layer upholds the **permuted buckets** and **single access** invariants (chapter 2). Treebeard's core parallelization strategy stems from the ORAM layer executing multiple read paths at once and reducing the overall number of buckets retrieved from the storage. Specifically, any two paths in a tree overlap at least at the root node (and perhaps other higher level nodes); the ORAM processes retrieve the overlapping nodes just once for each batch of read path requests. *This optimization leaks no new information to an adversary because the overlap across a set of paths is deterministic, which the adversary can deduce even for sequential Ring ORAM.* Treebeard merely batches these requests, with the overlapping nodes retrieved just once. ORAM layer also evicts paths in batches such that overlapping buckets are written just once for a set of k paths.

Failure of an ORAM process (if it is a Raft leader) has externally visible effect, as discussed in section 3.2.3. If an ORAM leader process fails while reading a path, Treebeard's replication ensures that the new leader will read the path again with the same offset per bucket as the old leader. If a leader fails while evicting a path, the new leader will re-evict the path without losing any data. Since an adversary learns about ORAM process failures, it can deduce the effect of re-executing a read or evict step by a new leader, which does not leak any workload dependent information. Algorithm 4 presents the high-level pseudocode for Treebeard's ORAM layer. For detailed implementation, please refer to open-sourced implementation.

Simulator

The ORAM layer simulator receives a set of randomly chosen paths and trees to retrieve from the Stash simulator. The simulator calls Ring ORAM's simulated read path function to retrieve data from the storage. The stateful simulator retains the randomly chosen offsets read from each bucket. For each tree, the simulator evicts k deterministically chosen paths after A read paths by invoking Ring ORAM's simulated evict function. Similarly, for each bucket in a tree accessed S times, it invokes Ring ORAM's simulated early reshuffle function. To simulate the failure behavior, the ORAM simulator takes external inputs that indicate (i) whether a leader process failed while reading or evicting a path, (ii) the i^{th} read or evict request that failed, and (iii) the buckets that were already retrieved or evicted at the time of failure. The simulator mimics the real system by first retrieving or evicting the same buckets indicated in the input. This is the pre-failure step. To simulate retries after failures, it either re-issues the i^{th} read path to the storage with the same offsets (for failure during reads) or re-evicts the i^{th} path by reading and writing the same path (for failures during eviction). □

From the above arguments, we can assert that the adversary observable view can be simulated using a simulator that has no data or workload-dependent behavior. Since both the real system and the simulator produce similar interactions with the external storage, the adversary cannot distinguish between the two with a significant advantage, which proves the security of Treebeard.

Algorithm 3 ORAM read path algorithm

1: **Inputs:** B: batch of read path requests.
2: **Variables:** *bucket_set*: distinct buckets in the requested paths, $offsets$: block offset to read from each bucket, *blocks*: blocks read from a tree.
3: **function** MULTIPATHREAD(B)
4: **for** $req \in B$ **do** ▷ *Gather all distinct buckets in paths*
5: $bucket_set \leftarrow bucket_set \cup GetBuckets(req.path)$
6: **end for**
7: **for** $bucket \in bucket_set$ **do** ▷ *Gather the block offsets per bucket to be read*
8: $offsets \leftarrow offsets \cup$
$GetBlockOffset(bucket, B.blockIDs)$
9: **end for**
10: **for** $offset \in offsets$ **do** ▷ *Read the actual blocks*
11: $blocks \leftarrow blocks \cup ReadBlock(offset)$
12: **end for**
13: Execute early reshuffle
14: Return *blocks*
15: **end function**
16: **function** GETBUCKETS(p)
17: Return distinct buckets in path p
18: **end function**
19: **function** GETBLOCKOFFSET(*bucket*, *blocks*)
20: Return a real block's offset if any of the *blocks* exists in this bucket, else returns a random valid offset ▷ *Identical to Ring ORAM's implementation*
21: **end function**
22: **function** READBLOCK(offset)
23: Return block at $offset$ from Bucket ▷ *Identical to Ring ORAM's implementation*
24: **end function**

Algorithm 4 ORAM eviction algorithm

Inputs: *paths*: paths to evict, *storage*: storage to evict
Variables: *bucket_set*: distinct buckets in the requested paths, *col_blocks*: collected blocks from the tree and Stash processes during eviction.
function MULTIPATHEVICT(*paths*, *storage*)
    for $path \in paths$ **do**
        $bucket_set \leftarrow bucket_set \cup GetBuckets(path)$
    end for
    for $bucket \in BucketsSet$ **do** ▷ *Collect all blocks from the server for paths being evicted*
        $col_blocks \leftarrow col_blocks \cup ReadBucket(bucket)$
    end for
    for $s \in Stashes$ **do** ▷ *Collect stash blocks from the Stash layer*
        $col_blocks \leftarrow col_blocks \cup$
            $RequestStashBlocks(s, paths, storage)$
    end for
    for $bucket \in BucketsSet$ **do**
        $WriteBucket(bucket, col_blocks)$
    end for
end function
function GETBUCKETS(p)
    Return distinct buckets in path p
end function
function REQUESTSTASHBLOCKS(s, *paths*, *storage*)
    From Stash process s, request blocks (randomly) assigned to *paths* and *storage*
end function
function READBUCKET(*bucket*)
    Return the valid buckets in the *bucket* ▷ *Identical to Ring ORAM's implementation*
end function
function WRITEBUCKET(*bucket*, *stash*)
    Write buckets from the *stash* to the *bucket*. ▷ *Identical to Ring ORAM's implementation*
end function

Chapter 5

Implementation & Evaluation

5.1 Implementation

We implemented Treebeard in Go, open source and available at: https://github.com/dsg-uwaterloo/treebeard. To ensure efficient communication between layers, we utilize gRPC to send and receive messages between layers; we use Protocol Buffers (protobuf) to serialize these messages. Treebeard's implementation uses Hashicorp's Raft [19] 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 implemented a Yahoo! Cloud Serving Benchmark (YCSB)-like [11] client that utilizes YCSB-generated traces to send GET/PUT requests to Treebeard. YCSB ensures that the workload generated reflects real-world workloads. Additionally, since Treebeard consists of many layers and components in its implementation, our implementation supports observability with low overhead. We use OpenTelemetry to efficiently send processing times per layer to a central telemetry aggregator. The experiments use Redis [45], as the untrusted cloud storage, with its pipelining capabilities to efficiently process the read path and evict operations.

The implementation reflects our layered design. It is implemented with separately deployable modules for every layer with clear APIs. To implement a new layer one can implement it in any language and expose the few APIs the other layers expect.

5.2 Experimental Setup

The experimental setup deploys Treebeard on Standard D8s v3 instances (8 vCPUs, 32GB RAM) on Azure [30]. 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 $k = 200$ 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.

5.3 Baselines

We evaluate Treebeard with three baselines, implemented by the baseline authors, each providing a central feature offered by Treebeard (scalability, fault-tolerance, and concurrency).

1. Snoopy [13] 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 [31]. 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 [29] 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 [47] is a tree-based concurrent ORAM solution that relies on a single proxy. It is neither scalable nor fault-tolerant.[1]

5.4 Evaluation

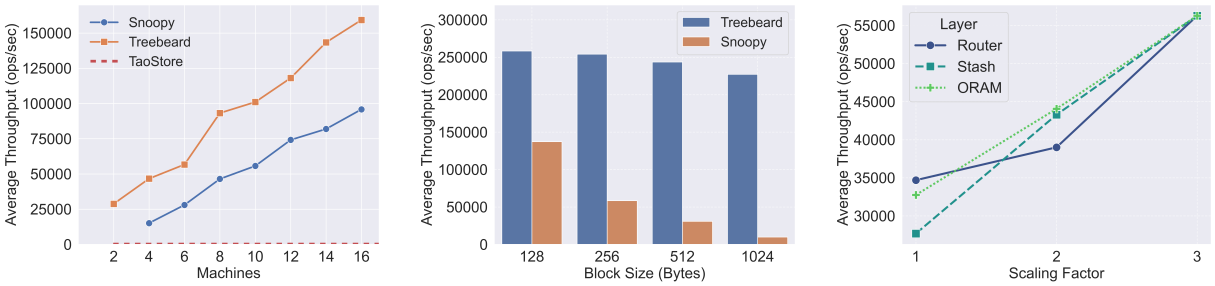
Our evaluation answers the following central questions:

- How does Treebeard scale compared to the existing baselines? What is the effect of scaling each layer? (section 5.4.1)
- How does Treebeard compare to fault tolerant baselines? How does it behave under failures? (section 5.4.2)
- How does Treebeard's optimizations affect its stash? (section 5.4.3)
- What effect do epochs have on Treebeard's performance? Is Treebeard *workload-independent*? (section 5.4.4)

5.4.1 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 [13], this 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,

[1] Although a better baseline, we do not compare Treebeard with Obladi [12] since we were unable to acquire an executable artifact online [49] or from the authors.



(a) Scaling Throughput

(b) Impact of Block Size

(c) Impact of Scaling Each Layer

Figure 5.1: Various aspects of scaling Treebeard and comparison to baselines.

which is what we report here, while reporting Treebeard's throughput with similar latency bound.

Figure 5.1a 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.

Comparing the results, unsurprisingly, TaoStore performs poorly compared to the other two systems. 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 5.1b. 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 10k ops/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 5.1c demonstrates the result of scaling each layer. The maximum throughput Treebeard achieves is 55k ops/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%. 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 5.1c, 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.

5.4.2 Fault-tolerance Analysis

The next set of experiments analyze Treebeard's ability to handle failures. We compare Treebeard with QuORAM [29], a state-of-the-art fault-tolerant and highly available ORAM system, with results in Figure 5.2. Similar to [29], 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 [29], but

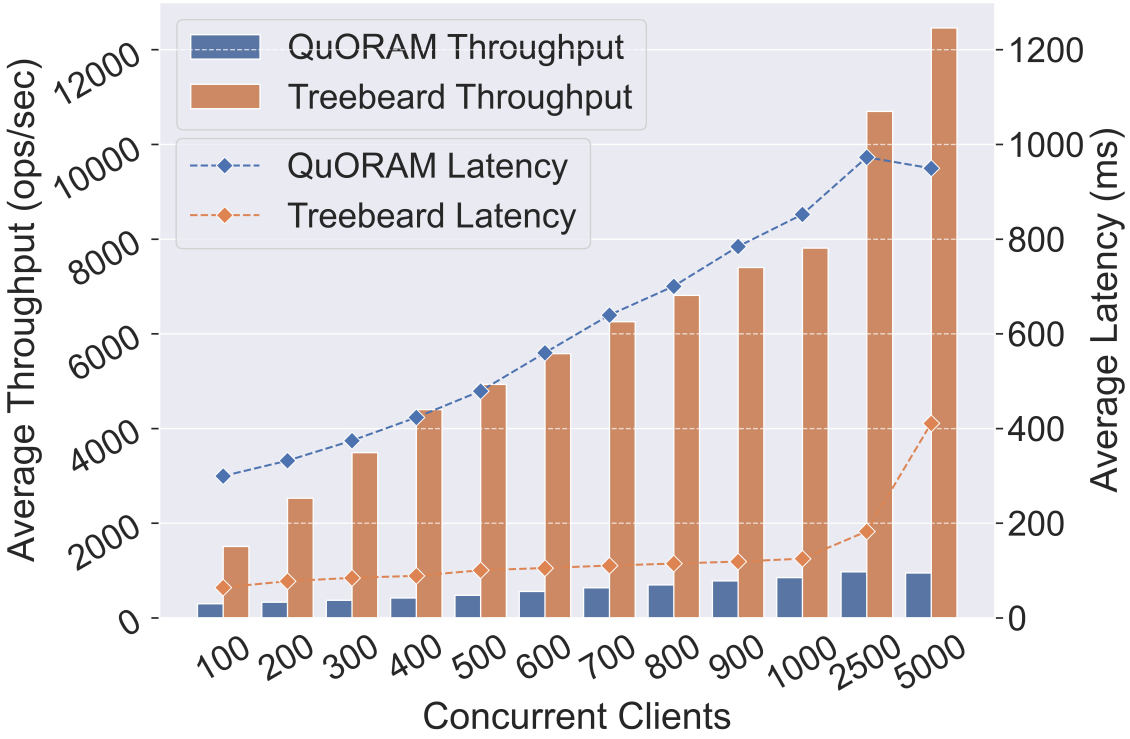
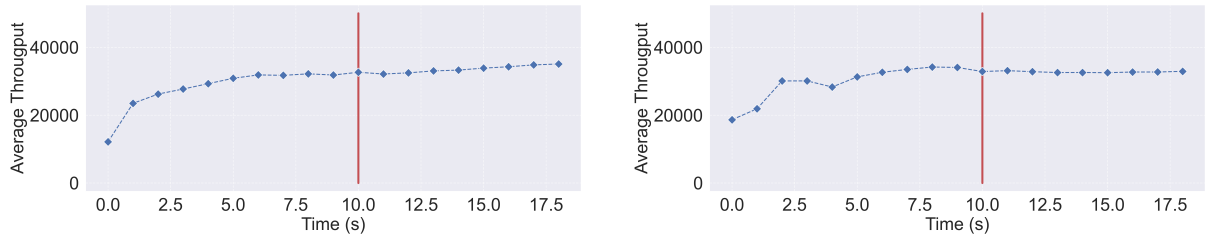


Figure 5.2: Treebeard and QuORAM throughput

Treebeard continues to achieve higher throughput. At 5000 concurrent clients, Treebeard executes 15k ops/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 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 5.3 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,



(a) A Stash leader failure

(b) An ORAM leader failure

Figure 5.3: **(a, b)** A Stash and ORAM leader failure has no noticeable effect on the throughput.

this experiment introduces leader failures at the 10^{th} second. Figure 5.3a and 5.3b 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.

5.4.3 Stash Experiments

As noted in section 3.2.3, 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 $Z = 4$ [51].

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 5.4a 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: The second stash experiment measures the impact of eviction rate on Treebeard's stash size and performance. Figure 5.4b shows the results of this experiment.

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 32k ops/sec compared to 27k ops/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 32k ops/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.

5.4.4 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 5.1 provides

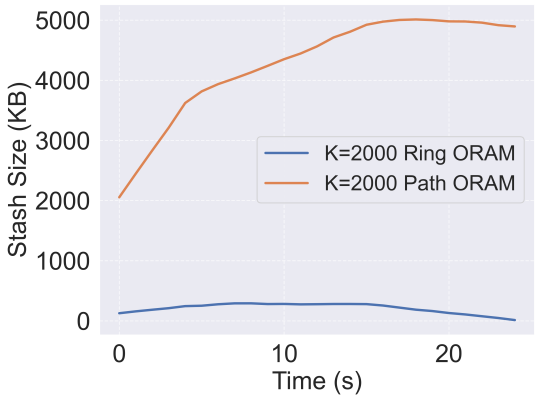
	Ring ORAM	Trb. (B=5)	Trb. (B=100)	Trb. (B=500)	Trb. (B=2000)
Buckets	21.00	15.46	14.46	12.14	10.14

Table 5.1: Comparison of the number of buckets retrieved by Ring ORAM and Treebeard (Trb.) with different batch sizes (B).

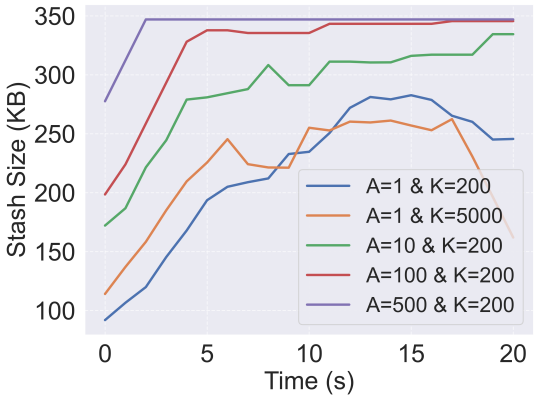
simulation results of average number of blocks retrieved per client request for 10,000 requests per run. With a batch of size 2000, which is the average batching size observed in our experiments with an epoch of 1ms, *Treebeard's multi-path eviction reduces bandwidth by **51%** compared to Ring ORAM*. 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 5.4c indicate that the throughput improves with the increase in epoch sizes, reaching a peak of 52k ops/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.

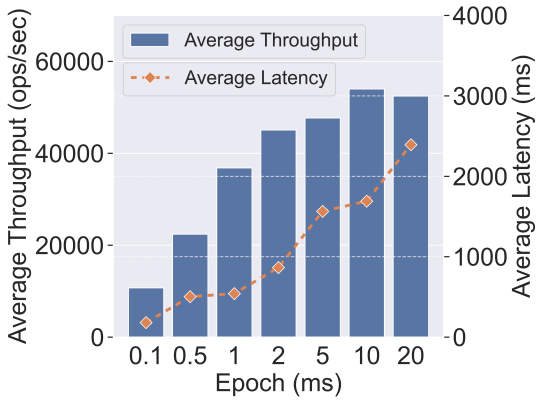
Workload Independence: Figure 5.4d shows Treebeard's throughput for different skewness – from uniform to 0.99 Zipfian – in the client workload. The results show negligible impact on system performance for different types of workloads. We attribute the small difference in throughput for different Zipf 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 [4, 47] that Oblivistore [50] is vulnerable to.



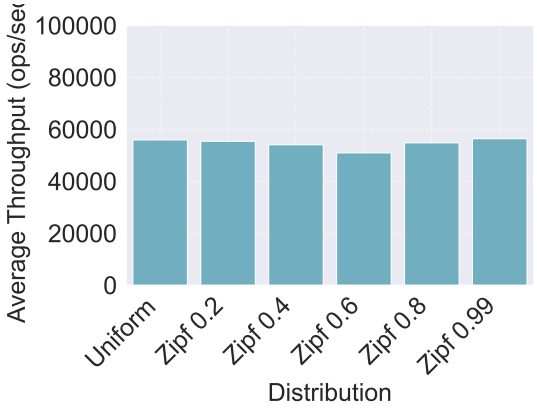
(a) Treebeard with Path ORAM vs. Ring ORAM stash size



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



(c) Impact of Epoch Size on performance



(d) Workload Independence in Treebeard

Figure 5.4: Stash and internal experiments of Treebeard.

Chapter 6

Related Work

This section explores key advancements in practical ORAM systems, focusing on parallelism, scalability, and fault tolerance. We begin by discussing how various ORAM schemes leverage parallelism to improve performance, including techniques for inter-request and intra-request concurrency. Next, we examine scalable ORAM designs, particularly those that distribute storage and processing across multiple servers while addressing bottlenecks in centralized proxies. Finally, we review fault-tolerant ORAM approaches that ensure availability in the presence of system failures.

6.1 Parallelism in ORAM Systems

Early ORAM schemes processed requests one at a time, but later systems introduced parallelism to improve throughput. Shroud [28] was one of the practical ORAM designs that introduced intra-request parallelism by reading multiple buckets of an ORAM tree path simultaneously (i.e., multiple levels at once). This allows a single ORAM access to use parallel hardware and reduce its latency. However, Shroud still handled client operations sequentially. In contrast, PrivateFS [53] and ConcurORAM [7] focused on inter-request parallelism by allowing multiple clients to access the ORAM store at the same time without a central proxy. PrivateFS introduced a parallel oblivious file system that uses a partitioned ORAM (PD-ORAM) as a building block, enabling clients to perform concurrent file operations securely. Similarly, ConcurORAM eliminates the centralized

proxy by having each client communicate directly with the cloud storage. This proxy-less approach avoids a bottleneck at a central server and lets requests proceed in parallel, substantially improving aggregate throughput. One trade-off is that such designs can incur higher network overhead (e.g. additional round trips for coordination) in wide-area settings, since clients must synchronize with the cloud storage to maintain ORAM invariants without a proxy.

Later systems combined parallelism with support for asynchronous operations. ObliviStore [50] and TaoStore [47] were designed for cloud environments where multiple users issue requests concurrently, possibly with network delays. Both adopt a trusted proxy model: a single intermediary batches and translates client requests into ORAM accesses on the untrusted storage server. These systems can handle concurrent requests arriving out-of-sync, ensuring that security is not compromised by asynchronous scheduling. ObliviStore was a pioneering distributed ORAM that sharded data across sub-ORAM shards to scale, but it required the centralized proxy to route every access to the correct shard in an oblivious manner. A subtle flaw was later discovered: if two clients accessed the same data near-simultaneously, the proxy's scheduling could leak timing information; CURIOUS [4] was proposed to fix this vulnerability, preventing an adversary from observing any timing information. CURIOUS improved ObliviStore's security for concurrent access patterns, but it still relied on a centralized load-balancing proxy, and subsequent analysis showed it remained insecure under fully adversarial asynchronous conditions [47]. TaoStore [47], on the other hand, tackled asynchrony by introducing a sequencer that maintains a consistent ordering, preventing leakage of any timing information.

A significant attempt to maximize parallel ORAM throughput is Obladi [12]. Obladi is an oblivious cloud storage system that supports ACID transactions, and it explicitly batches operations to exploit parallelism across multiple requests. The core idea is to group incoming requests into epochs and reshuffle the ORAM after each epoch rather than after each individual access. Within an epoch, Obladi tracks dependencies between operations: if two operations need to access the same tree path or data block, it orders them to preserve correctness, but otherwise it executes independent ORAM accesses concurrently. By using this epoch-based scheduling, Obladi amortizes the cost of ORAM operations and allows many non-conflicting reads/writes to proceed in parallel. This approach significantly increases throughput: experiments showed that inter-request batching yields an order-of-magnitude speedup. Obladi still employs a single trusted proxy to coordinate each epoch that can become a performance bottleneck and a single point of failure.

Multi-Path ORAM Approaches (OBI and Fork Path): In addition to the above schemes, researchers have explored multi-path operations in ORAM to boost throughput. Traditional tree-based ORAMs (e.g. Path ORAM) read and evict a single root-to-leaf path per request, but newer designs show benefits in accessing multiple paths per operation. Two notable systems in this vein are OBI and Fork Path ORAM, which both leverage multi-path access but in different contexts. OBI (Wu and Li, 2024) is a multi-path ORAM designed for forward-and-backward private searchable encryption. It tackles the "large-stash" problem that arises when a single query needs to fetch or insert a large set of blocks. Instead of treating each block access as a separate ORAM operation, OBI accesses multiple tree paths in one batch for a given query. By fetching all required paths together, OBI amortizes the overhead of ORAM access and drastically reduces the round-trip count for bulk operations. To make this viable, OBI introduces new eviction strategies tailored for batched accesses: a k-Nearest Neighbor Eviction Algorithm and a Partition-Based Eviction Algorithm. These algorithms efficiently flush a large stash of blocks by evicting multiple paths' worth of data in a single round, achieving several orders of magnitude better eviction throughput than naive Path ORAM when the stash is large. OBI's evaluation demonstrates that multi-path ORAM operations can significantly outperform conventional single-path ORAM in scenarios with high-result-count queries. However, OBI operates in a single-client context (one user's query processed at a time) and is built on a standard tree ORAM (similar to Path ORAM). It focuses on batch handling and eviction efficiency for one large request rather than on multi-user concurrency. Notably, OBI does not emphasize deduplicating overlapping path segments during a batch read – each path is accessed as needed – but rather concentrates on coping with the many blocks in the stash via improved eviction. In summary, OBI demonstrated that reading and evicting many paths as one operation is feasible and can improve throughput for certain workloads (like search queries returning many results), though it required careful algorithmic changes to the Path ORAM design to remain secure and efficient (ensuring forward/backward privacy in the encryption scheme's context).

Fork Path ORAM (Zhang et al., MICRO 2015) takes a different approach to multi-path operations, aiming to remove redundant memory accesses across ORAM requests. The insight behind Fork Path is that if two ORAM requests traverse partially overlapping root-to-leaf paths (a common occurrence, as many paths share prefix buckets near the root), the shared segments need not be fetched twice. Fork Path ORAM therefore introduces path merging: it processes multiple pending ORAM operations together and merges their

access paths into a single "forked" access pattern, reading common buckets once and then forking into separate branches when the paths diverge. To enable this, the system reorders and schedules ORAM requests to maximize overlap (so that requests with similar accessed paths are served together) and employs a merging-aware caching mechanism to temporarily cache recently read buckets for reuse by other requests in the queue. By leveraging these techniques, Fork Path significantly reduces the total number of block reads and writes per batch of operations without breaking ORAM's security invariants. In essence, it achieves an intra-batch deduplication similar to Treebeard's multi-path read optimization: overlapping levels of the tree are accessed only once. Fork Path was implemented at the ORAM controller level in a secure processor setting, demonstrating up to 58

Comparison with Treebeard: Treebeard's approach draws inspiration from the idea of multi-path access but differs markedly in design goals and capabilities. All three systems – Treebeard, OBI, and Fork Path – recognize that reading one path per operation under-utilizes available parallelism, and they instead perform batched/multi-path reads to improve throughput. However, Treebeard is distinguished by its system-level scope and unrestricted concurrency. Treebeard introduces a multi-path read mechanism that aggregates an arbitrary set of ORAM paths and reads each bucket at most once (thanks to tracking overlapping nodes). This is conceptually similar to Fork Path's path merging (both avoid redundant bucket reads), and indeed Treebeard sets each bucket's capacity Z = 1 specifically to simplify this deduplication – each bucket holds at most one real block, so an ORAM process always reads exactly one block per bucket regardless of how many paths are in the batch. In contrast, OBI and Fork Path rely on Path ORAM with Z ¿ 1 buckets. OBI's design must cope with scenarios where multiple real blocks from a single bucket might be needed in one multi-path operation (since Path ORAM buckets can contain several data blocks); it handles this by algorithmically ensuring security (e.g., possibly reading extra dummy blocks or limiting batch size) and by using powerful eviction algorithms to empty the stash. Fork Path, using standard Path ORAM, limits how many requests merge at once based on dummy block availability and uses careful scheduling to avoid security leaks. Treebeard circumvents these complications by using a Ring ORAM–inspired scheme with Z = 1, effectively enforcing that no bucket ever contains more than one real block. This design choice, along with a consistent dummy block strategy, lets Treebeard batch unlimited numbers of requests (limited only by system resources) without having to restrict concurrency to avoid multi-block leakage – a key advantage over prior multi-path proposals. As a result, Treebeard can serve hundreds or thousands of concurrent read requests

in parallel, whereas prior systems often had to cap the batch size or concurrency window to maintain security invariants (for example, ConcurORAM and CURIOUS deliberately restrict the degree of concurrency, and Fork Path's efficiency plateaus beyond a certain queue size).

Another difference lies in the eviction (write-back) phase. Treebeard performs periodic multi-path evictions: after every A batched reads, each ORAM worker selects k paths and evicts blocks along those paths in one combined operation (filling each bucket with one real block if available and dummies otherwise, akin to Ring ORAM's procedure). This keeps the stash size at the proxy manageable despite the high request volume. OBI similarly evicts multiple paths together, but it uses its custom algorithms (KNNEA, PBEA) to do so in a single round with optimal or near-optimal time complexity when handling a large stash. Fork Path did not introduce a new eviction algorithm; it essentially still performs the standard Path ORAM eviction for each merged request (which is simpler because each merged request was treated as one extended ORAM access). In practice, Treebeard's eviction approach is less algorithmically complex than OBI's – Treebeard relies on the underlying Ring ORAM eviction pattern (augmented to multiple paths) and on tuning the eviction frequency A to balance stash growth, whereas OBI provides provable efficiency bounds for evicting a very large stash in one go. This means OBI's scheme might be more optimized for extreme batch sizes in theory, while Treebeard's approach is sufficient in a system setting where A can be adjusted to prevent the stash from ever growing too large.

Crucially, Treebeard targets a multi-user, scalable cloud datastore environment, which sets it apart from both OBI and Fork Path. OBI's multi-path ORAM is presented in the context of a single-client protocol (e.g., one user searching an encrypted index) and does not address concurrent accesses by multiple independent clients. Fork Path is even more restrictive in scope, focusing on a single CPU's memory accesses. Treebeard, by contrast, is architected to handle many clients and ORAM requests in parallel across a distributed system. Its multi-layer design (batching layer, stash management layer, storage layer) and use of replicated processes for fault tolerance address challenges (like load balancing and failures) that OBI and Fork Path do not consider. In Treebeard, concurrency is on two levels: many requests can be batched together (like OBI/Fork Path's idea, but without a fixed limit), and many clients/threads can be served simultaneously due to the system's scaled-out, proxy-free architecture. Furthermore, Treebeard's multi-path read and write optimizations are integrated into this larger system with careful attention to crash recovery and consistency (e.g., using Raft to replicate ORAM state and handle failures during

batched operations). Neither OBI nor Fork Path provide fault tolerance or a distributed architecture – they assume a benign execution environment with no crashes.

In summary, Treebeard, OBI, and Fork Path all improve ORAM performance by using multiple-path operations, but they do so with different priorities. Treebeard's distinctive features are its unlimited batching of requests and a Ring ORAM–based design (Z=1) that enables deduplicated reads at scale, all within a fault-tolerant, multi-user system. OBI demonstrates the algorithmic feasibility of multi-path ORAM on standard Path ORAM for large single queries, introducing sophisticated eviction techniques but remaining a single-client solution. Fork Path shows the benefits of merging ORAM requests at a low level, removing redundant accesses through scheduling and caching, albeit in a single-processor scenario. Treebeard builds on the spirit of these approaches but extends it to a full-scale system, achieving both high throughput and robustness by combining multi-path operations with system-level parallelism and replication. This comparison highlights that Treebeard is not an isolated idea – it sits at the intersection of prior work on ORAM parallelism – yet its contribution lies in integrating deduplicated multi-path access with a scalable, concurrent architecture that pushes ORAM performance to new levels.

6.2 Scalability in ORAM Systems

To support large-scale data and high request numbers, ORAM systems must scale—ideally distributing both storage and workload across multiple servers. ObliviStore [50] was a significant effort to create a scalable ORAM-backed cloud storage. It distributed the ORAM storage across multiple machines by dividing the dataset into sub-ORAMs, each responsible for a partition of the address space. A centralized trusted proxy keeps track of which sub-ORAM holds each data block and routes each client request to the appropriate server shard. To prevent the cloud from learning which shard is accessed (since that would reveal which partition the address falls into), ObliviStore uses an oblivious load balancer: every request is passed through a shuffling procedure so that an outside observer cannot easily link a specific query to a particular shard. While this design achieved high performance, it encountered two key issues when scaling up to many clients. First, the reliance on a single load-balancing proxy introduced a new scalability bottleneck. Second, as mentioned above, researchers discovered security vulnerabilities in ObliviStore under its security model because of timing attacks. CURIOUS [4] was introduced to refine ObliviStore's design.

A more recent scalable ORAM system is Snoopy [13], which takes a different approach by leveraging trusted hardware enclaves for scalability. Snoopy distributes the ORAM index and data across many servers and runs the ORAM controller logic inside secure enclaves on each server. By doing so, Snoopy can scale horizontally: as the data or query load grows, more enclave-equipped servers can be added. The enclaves assure that even though each server operates on only a subset of the data, the access patterns to its local memory are not visible to the cloud provider. Snoopy's design thereby avoids a centralized proxy and achieves higher throughput than prior state-of-the-art ORAM systems by exploiting parallelism and batch processing across enclaves. The main drawback is the reliance on specialized hardware (Intel SGX or similar). Only certain cloud providers offer these trusted execution environments, and enclaves come with their own limitations (memory size, potential side-channel attacks, etc.).

6.3 Fault tolerance in ORAM Systems

Only two existing ORAM data stores, Obladi [12] and QuORAM [29], incorporate fault tolerance in their design. Obladi assumes that cloud storage is persistent and handles crash faults in the proxy layer. Specifically, Obladi pushes its state—including position maps, stashes, and relevant metadata—to cloud storage at the boundaries of predefined epochs. In the event of a failure, the proxy retrieves this saved state to resume operations. However, this design introduces a period of unavailability, as the system must wait for the proxy to recover and reinitialize its state before it can resume serving requests.

In contrast, QuORAM mitigates Obladi's unavailability problem by replicating the server-proxy pairs. This replication ensures high availability since, if one replica fails, another can immediately take over without significant service interruption. Although QuORAM provides robust fault tolerance by ensuring continuous operation during failures, its design does not scale.

Chapter 7

Conclusion

In this thesis, 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 novel 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.

7.1 Future Research Directions

While Treebeard represents a significant advancement in scalable, fault-tolerant ORAM systems, several avenues for future research remain:

- **Support for Complex Queries:** Currently optimized for simple GET/PUT operations, Treebeard can be extended to support complex relational queries. Integrating query optimization techniques and extending Treebeard's capabilities to handle SQL-like queries could improve its adoption for practical use cases.

- **Transactional Guarantees:** Extending Treebeard to support different transactional guarantees would enable users to tune the system for their own use case. Lowering the guarantees will benefit the performance at the cost of weaker transactional gurantees.
- **Other Security Models:** While Treebeard effectively mitigates information leakage through its multi-path read mechanism, future work could explore stronger adversarial models, such as those involving side-channel attacks or timing attacks.
- **Dynamic Scaling and Resource Management:** Implementing dynamic scaling capabilities would allow Treebeard to adapt to varying workloads by automatically adjusting the number of active machines and optimizing resource utilization. This could improve both performance and cost efficiency in cloud-based deployments.

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, and Elaine Shi. Oblivious RAM with worst-case logarithmic overhead. J. Cryptology, 36(2):7, 2023.

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

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

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

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

[26] 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.

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

[28] 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.

[29] 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.

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

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

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

[33] 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.

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

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

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

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

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

[43] 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.

[44] 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.

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

[46] 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.

[47] 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.

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

[49] Snoopy Repository. https://github.com/ucbrise/snoopy. Accessed Jan 10, 2025.

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

[51] 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.

[52] 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.

[53] 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.

[54] 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.