QuORAM: A Quorum-Replicated Fault Tolerant ORAM Datastore

Sujaya Maiyya, Seif Ibrahim, Caitlin Scarberry, Divyakant Agrawal, and Amr El Abbadi, *UC Santa Barbara;* Huijia Lin and Stefano Tessaro, *University of Washington;* Victor Zakhary, *Oracle*

https://www.usenix.org/conference/usenixsecurity22/presentation/maiyya

This paper is included in the Proceedings of the 31st USENIX Security Symposium.

August 10–12, 2022 • Boston, MA, USA

978-1-939133-31-1

QuORAM: A Quorum-Replicated Fault Tolerant ORAM Datastore

Sujaya Maiyya
UC Santa Barbara

Seif Ibrahim
UC Santa Barbara

Caitlin Scarberry
UC Santa Barbara

Divyakant Agrawal
UC Santa Barbara

Amr El Abbadi
UC Santa Barbara

Huijia Lin
University of Washington

Stefano Tessaro
University of Washington

Victor Zakhary
Oracle

Abstract

Privacy and security challenges due to the outsourcing of data storage and processing to third-party cloud providers are well known. With regard to data privacy, Oblivious RAM (ORAM) schemes provide strong privacy guarantees by not only hiding the contents of the data (by encryption) but also obfuscating the access patterns of the outsourced data. But most existing ORAM datastores are not fault tolerant in that if the external storage server (which stores encrypted data) or the trusted proxy (which stores the encryption key and other metadata) crashes, an application loses all of its data. To achieve fault tolerance, we propose ***QuORAM***, the first ORAM datastore to replicate data with a quorum-based replication protocol. QuORAM's contributions are three-fold: (i) it obfuscates access patterns to provide obliviousness guarantees, (ii) it replicates data using a novel lock-free and decentralized replication protocol to achieve fault tolerance, and (iii) it guarantees linearizable semantics. Experimentally evaluating QuORAM highlights counter-intuitive results: QuORAM incurs negligible cost to achieve obliviousness when compared to an insecure fault-tolerant replicated system; QuORAM's peak throughput is **2.4x** of its non-replicated baseline; and QuORAM performs **33.2x** better in terms of throughput than an ORAM datastore that relies on CockroachDB, an open-source geo-replicated database, for fault tolerance.

1 Introduction

Due to the cloud's core policy of *pay-by-use*, individuals and organizations are increasingly shifting from managing their own storage servers to renting storage from third-party cloud providers. Today, many products with high traffic, such as Twitter [5], Spotify [4], and Netflix [3], rely on cloud storage for some or all of their data storage requirements.

The cloud's convenience, however, comes at the cost of potentially compromising the privacy of the outsourced data. This privacy concern slows down the adoption of cloud services for many businesses [11]. Even with the encrypted data, users' access patterns can leak sensitive information to the cloud provider. Consider an example where a doctor stores patient records in a third-party cloud. If the doctor accesses a given patient's record more frequently than usual over a period of time, an intruder can infer some information about the patient's medical status. In fact, many works [9,15,17,19,21,22] have shown concrete inference attacks by exploiting access patterns alone.

The privacy of outsourced data requires first to hide the data content through encryption, and then to *obfuscate* the access pattern to that encrypted data. Oblivious RAM, or ORAM, a cryptographic primitive originally introduced by Goldreich and Ostrovsky [16], achieves access pattern obliviousness. Although ORAM originally protected software executing on a single machine from an adversary on that same machine [16], ORAM's functionalities are now extended to protect data accesses on remote storage [7, 10, 13, 25, 31–34]. Summarizing the general idea in these works: they break up the data into logical blocks, each stored at a unique physical addresses on the external server. After each access to a logical block, the ORAM scheme shuffles the physical address, thereby mapping any sequence of logical memory accesses to a sequence of random physical memory accesses.

Broadly speaking, many remote ORAM system architectures [7, 13, 14, 31, 33] consist of three-layers: *an untrusted cloud storage server*, *a trusted proxy*, and *the clients*. An application encrypts its data under a key K and outsources the encrypted data onto an untrusted storage server. The trusted proxy holds the key K and accesses the storage server on behalf of the application's clients. Clients send read and write requests to the proxy, which then communicates with the server according to an *ORAM scheme* and responds back to the clients. An ORAM scheme translates client requests into a sequence of storage server accesses that are *indistinguishable* from other client request translations.

Recent proposals enhance the efficiency of ORAM schemes [7, 8, 10, 13, 14, 31, 33, 39, 40] by supporting concurrent and asynchronous client accesses. However, in most of these proposals, the proxy and the storage server are not fault tolerant, deeming both components as single points of failure. If either crashes, the data becomes unavailable to users. Putting it differently, mitigating the privacy concerns of cloud storage derails one of the most significant advantages of the cloud: *fault tolerance*.

To date, Obladi [13] is the only ORAM system to tolerate crash failures without losing the system's state. For the storage server, Obladi relies on the standard fault tolerance guarantees of cloud storage servers and assumes a highly available server. For the proxy, Obladi meticulously pushes 'valid' proxy states to the cloud storage such that after a crash, the proxy resets to the last valid state stored fault-tolerantly in the cloud. The main problem with this approach is that although a proxy's relevant state can be recovered from the storage after a crash, the system cannot progress *while* the proxy is down. Moreover, delegating fault tolerance to the cloud incurs higher latencies than an ORAM system with inherent fault tolerance guarantees, as shown in the later sections of this paper.

In distributed systems, the gold standard for fault tolerance is state machine *replication*. Zakhary et al. [41] discuss replication to tolerate failures in ORAM systems and demonstrate the challenges of employing standard design choices – such as locking and quorum-based read-writes – in an ORAM system. The authors discuss only the risks of standard design choices for replication in ORAM systems rather than provide any solution to tolerate failures.

In this paper, we present, ***QuORAM***, the first (quorum) replicated fault-tolerant ORAM system, consisting of multiple untrusted cloud storage instances and trusted proxies. QuORAM replicates the data on multiple storage instances, where each storage instance is accessed through its independent trusted proxy. A subset of these replicas serve each client request, thus allowing the system to tolerate some failures at both the storage and the proxy layers.

Serving client requests from only a subset of replicas raises the challenge of *consistency*, which we define using linearizable semantics: "each operation applied by concurrent processes [appears to take] effect instantaneously at some point between its invocation and its response" [18]. Note that the operations themselves need not take effect instantaneously across all replicas (and cannot, in the presence of asynchronous network delay); they only need to *appear* instantaneous to the clients. We address this challenge and prove that QuORAM guarantees linearizable semantics.

Apart from obliviousness and fault tolerance, QuORAM achieves the following additional functionalities:

1. It supports multiple concurrent reads and writes,
2. It has no single point of failure,
3. It replicates data across multiple (possibly colluding) cloud storage servers, and
4. It guarantees linearizable semantics.

In the rest of the paper, §2 provides background on the ORAM scheme on which we build QuORAM; §3 describes the system and failure model of QuORAM; §4 defines the security model of QuORAM; §5 proposes the replication and ORAM scheme designs on QuORAM; and §6 experimentally evaluates QuORAM with three baselines. Appendices A and B detail the security and linearizability proofs of QuORAM.

2 Background

This section introduces an ORAM scheme, TaORAM [31], that acts as a building block of QuORAM. TaORAM ensures obliviousness in the presence of concurrent, arbitrarily-scheduled accesses while preserving linearizable semantics. TaoStore's [31] ORAM scheme, TaORAM, builds upon another ORAM scheme Path ORAM [35]. Path ORAM organises data into a tree of *buckets*, each of which contains multiple

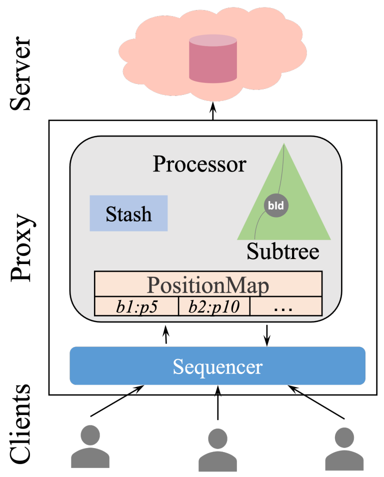


Figure 1: TaORAM's architecture

data *blocks*. Path ORAM maps each block's position to a leaf node *lf*, and stores the block in any one of the buckets along the path from the root to that leaf *lf*. TaoStore [31] extends Path ORAM for asynchronous and concurrent queries. TaoStore's system architecture (Figure 1) consists of a storage server, a proxy, and the clients. The storage server stores the encrypted data in a tree and the clients access the data by sending read/write requests to the trusted proxy; the proxy accesses the storage server on behalf of the clients (using the encryption key it stores) according to the TaORAM protocol.

The proxy consists of two components: a *Sequencer* and a *Processor*. The Sequencer communicates with clients and the Processor communicates with the server. The Sequencer maintains a FIFO request queue, which stores client requests in the order they arrive. When the proxy finds a response to a client request (after communicating with the server), the Sequencer forwards responses to clients in the request queue's FIFO order. The Processor maintains three pieces of local state: a position map, a local subtree, and a stash. The position map stores a block's leaf node id *lf* on whose path the block resides. The local subtree consists of blocks already fetched from the storage server (and possibly updated) but not yet written back, whereas blocks that do not fit in the subtree are stored in the stash. After the Processor fetches k paths, where k is a system configuration constant, a background thread writes those paths back to the server and deletes their contents from the local subtree. As k increases, the amount of memory consumed by the proxy also increases.

At a high level, TaORAM executes the following steps for both reads and writes to a block B:

1) Let P be the path containing block B. TaORAM fetches P from the server if not already fetched; otherwise, it performs a *fake read* by fetching a random path.

2) TaORAM adds the read path to the local subtree. For write operations, it updates the value of B in the local subtree.

3) TaORAM answers the client's request with B's value.

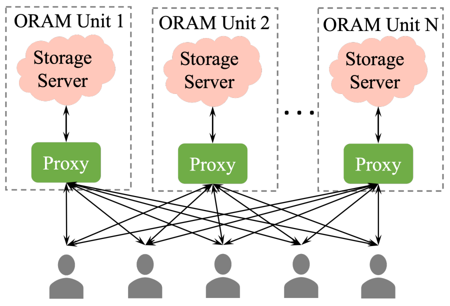


Figure 2: QuORAM Architecture

4) It assigns B to a new random path P' and updates the position map accordingly.
5) TaORAM next executes *flushing*: it reassigns each block in the subtree's path P or in the stash to the lowest non-full bucket intersecting with P and P', the block's newly assigned path. If no such bucket exists, TaORAM moves the block to the stash. TaORAM [31] proves that the stash size is bounded.
6) If TaORAM fetched k paths since the last write-back (where k is a system configuration constant), it writes those k paths from the subtree to the storage server. It then deletes all blocks in these k paths with no in-progress requests and retains blocks modified since initiating the write-back.

Although TaORAM preserves linearizability (as the authors proved in [31]), by itself, TaORAM does not tolerate failures. A user loses access to the data if the proxy or the storage server become unavailable. Additionally, the data cannot be recovered if the proxy or/and the storage server lose data.

3 System and Failure Model

Given the lack of fault tolerance in TaORAM and almost all existing ORAM datastores, we propose QuORAM, an ORAM datastore that provides fault tolerance via replication. This section presents the system and failure models of QuORAM.

3.1 System Model

QuORAM is a replicated oblivious data storage system that supports single key read and write operations on a key-value store, modeled as GET() and PUT() requests.[1] QuORAM has the same three-layered structure as a non-replicated ORAM system: untrusted storage servers to store encrypted data, application controlled trusted proxies to answer client requests by accessing the storage server, and clients who send read/write requests to the proxies. Typically in non-replicated ORAM systems, the overall state of the data is split between the proxy and the external storage. Extending an ORAM system to include replication also requires maintaining this one-to-one correspondence between a proxy and a storage server. Hence QuORAM replicates storage servers and proxies *in pairs* such that each proxy contacts exactly one storage server, and no two proxies contact the same storage server.

[1]Inserts and deletes are modeled using GET() and PUT() requests.

We refer to a pair of ORAM server and proxy as an *ORAM unit* and depict the system architecture in Figure 2. Although not a requirement, since QuORAM aims to tolerate crash failures, we envision QuORAM to be a geo-replicated datastore wherein the ORAM units and the clients accessing the data are all geo-distributed.

Within each ORAM unit, the external server S stores encrypted data while the corresponding proxy stores the respective secret key that encrypts S's data. The proxies in QuORAM also store other metadata necessary for the ORAM scheme (explained more in §5). All proxies in the system run the same ORAM scheme, translating each ORAM operation into a sequence of storage server operations. From a client's perspective, it treats an ORAM unit as a black box that exposes a read-write interface.

3.2 Failure Model

Crash failures: Our goal in developing a replicated ORAM system is to provide durability and failure tolerance comparable to production cloud storage. An ORAM unit enters a failed state when its storage server and/or its proxy crashes or when network partitions occur. These failures are effectively equivalent to the *entire* unit being unreachable: since the proxy holds the encryption secret key, the data accessed from the storage server cannot be decrypted without the proxy's decryption key, and the proxy's key is useless without the data from its corresponding storage server. As such, we consider an ORAM unit failure to be a single failure event, regardless of which component actually failed.

To tolerate a maximum of f failures, QuORAM replicates data onto $2f+1$ ORAM units. When a failed unit (server and proxy) resumes operation after a crash, it resumes the state before the crash. If an application assumes that a failed unit does *not* recover its previous state upon crash recovery, then the recovered unit can copy the current state from a *majority* of the ORAM units (this is because QuORAM relies on majority quorums to replicate the data and reading data from a majority guarantees reading the latest values of data, as will be discussed in §5.1).

All communication channels – clients to proxies, proxies to servers – are asynchronous, unreliable, and insecure. QuORAM secures all communication channels by employing encryption mechanisms such as transport layer security to mitigate message tampering.

Threat model: QuORAM assumes an *honest-but-curious* adversary that executes the designated protocol correctly. An adversary may control one or all external storage servers and can observe, track, and analyze data accesses to and from the server and perform inference attacks based on the access patterns. The adversary can control the *asynchronicity* of the network and also schedule read/write requests via a compromised client. Crash failures are consistent with the *honest-but-curious* adversarial model, hence we do not consider more severe malicious failure modes in this paper. The goal is to design an oblivious data storage system that tolerates catastrophic crash failures under the aforementioned adversarial model.

4 Security Model: Obliviousness in a Replicated ORAM Setting

Existing definitions of obliviousness are insufficient to capture the security of a replicated ORAM system because even if a single proxy-server pair provides ORAM guarantees, the choice of replication protocol may leak non-trivial information. Consider quorum-based replication protocols such as CRAQ [37] or Hermes [20]. In these works, read requests access a single node (i.e., single-node read quorums) and write requests access *all* the nodes in the system (i.e., all-node write quorums, which intersect with all single-node read quorums). Deploying such schemes allows an adversary to distinguish between read and write operations by merely observing how many units are accessed for an operation, regardless of whether the ORAM scheme leaks any information about the operation type.

To formalize the above information leak, we develop a new definition of obliviousness, adapted from the notion of *aaob-security* (adaptive asynchronous obliviousness) from TaORAM [31]. Intuitively, an ORAM scheme is aaob-secure if any two sequences of operations and any two data sets are indistinguishable to the attacker. This section first defines the ORAM scheme of QuORAM and then presents a security game based on which we define the security of replicated ORAM datastores.

4.1 ORAM scheme definition

A typical asynchronous ORAM scheme consists of two modules ORAM = {Encode, OClient}. Encode encrypts data D, and produces D_{enc} and a secret key K. An external server stores D_{enc} and a stateful ORAM client, OClient, stores K. QuORAM uses the above definition of ORAM = {Encode, OClient} for individual ORAM units but extends it to a list: Rep-ORAM = (ORAM$_1$, ORAM$_2$, ..., ORAM$_n$) for n ORAM units. Each ORAM unit ORAM$_i$'s Encode module receives the same data D. Given D, the Encode module outputs a secret key K_i and the data set D_{encK_i} encrypted with K_i after internally shuffling the data in a random order. The shuffling mitigates identical access patterns across different storage servers at the beginning of execution. The i^{th} external server stores D_{encK_i} and the corresponding i^{th} OClient retains K_i – both the server and OClient (executed by proxy) form an ORAM unit, ORAM$_i$.

Individual OClient's execute ORAM requests denoted as (op, bid, v) where op $\in$ {read, write}, bid represents a data block's id, and v=$\perp$ for reads or a new block value for writes. These operations result in read/write accesses to the storage server. While an OClient process recognizes a single type of operation – ORAM operation – represented by (op, bid, v), QuORAM distinguishes between two types of operations: *logical* and *ORAM*. Logical operations are client requested read/write operations[2] represented as (lop, bid, v) – where lop $\in$ {read, write}, bid is a data block's id, and v=$\perp$ for reads or an updated value for writes. Each logical operation in-turn translates to a sequence of ORAM operations (op, bid, v)$_i$ sent to an ORAM unit i. For example: a logical read can translate to a set of ORAM reads sent to a quorum of ORAM units followed by ORAM writes sent to that quorum.

4.2 Security definition

A replicated ORAM system, such as QuORAM, requires a slightly different security definition compared to aaob-security. The attack presented at the beginning of this section of using CRAQ [37] or Hermes [20] replication protocol clearly indicates that an aaob-secure system can still leak the type of logical operation. Hence, we extend aaob-security to include *logical obliviousness* i.e., *l-aaob-security*. *l-aaob-security* is an indistinguishability based security definition, which we define using a game $\mathcal{G}$. The steps of the game are:

- The game picks a uniformly random bit $b \in \{0,1\}$, called the challenge bit.
- An adversary $\mathcal{A}$ generates two same-sized sets of data D_0 and D_1. The game calls Rep-ORAM on D_b, i.e., it calls $D^b_{encK_i}, K_i \leftarrow \text{Encode}_i(D_b)$ for each ORAM unit i. The external server and OClient of an ORAM unit i store the encrypted data D_{encK_i} and the secret key K_i, respectively.
- The adversary, at any point in time, schedules two logical operations $(lop_{i,0}, lop_{i,1})$ consisting of arbitrary logical reads/writes. The game picks only one of the operations $lop_{i,b}$ and executes a replication protocol chosen by the replicated ORAM system by sending ORAM read/write operations to the ORAM units. The game notifies the adversary once the operation terminates without revealing the actual result, as the adversary can easily guess the challenge bit b based on the result.
- Throughout the above process, the adversary can read, delay, drop, and learn the timing of (but not modify) messages. The adversary can also cause any storage server, proxy, and/or client to crash, with at most f proxy/storage server failures.
- Finally, after scheduling any number of logical operations, the adversary decides on the value of the challenge bit b. The game $\mathcal{G}$ returns True if the adversary chooses the right bit; and otherwise returns False. At this point, the game terminates.

We define *l-aaob-advantage* of the adversary $\mathcal{A}$ against Rep-ORAM as

$$Adv^{l-aaob}_{Rep-ORAM} = 2 * Pr[\mathcal{G}^{l-aaob}_{Rep-ORAM} \Rightarrow \text{True}] - 1 \qquad (1)$$

A replicated ORAM system is *l-aaob-secure* if $Adv^{l-aaob}_{Rep-ORAM}$ is negligible for any polynomial time

[2]Logical reads/writes are equivalent to a key-value store's GETs/PUTs.

adversary $\mathcal{A}$, i.e., any polynomial-time adversary can guess the challenge bit with probability negligibly higher than half. In other words, an ORAM scheme is *l-aaob-secure* if any two sequences of logical operations[2] and any two data sets are indistinguishable to the attacker.

5 QuORAM: a replicated ORAM datastore

This section presents the design of the replicated ORAM datastore, QuORAM. In designing QuORAM, we aim to achieve three goals: (i) obfuscate access patterns to achieve privacy and *l-aaob-security*, (ii) replicate the data for fault tolerance, and (iii) achieve the above two goals while preserving linearizable semantics.

To describe how we achieve the above goals, this section first discusses the design of a data replication protocol that preserves linearizability, followed by the ORAM scheme that hides access patterns.

5.1 QuORAM's replication protocol

In describing QuORAM's replication protocol, for now, we assume the system employs a state-of-the-art ORAM algorithm, TaORAM, as a black-box (this is relaxed in §5.2) and focus only on the replication protocol that provides linearizability guarantees. Choosing an existing replication protocol or designing one is a non-trivial task due to preserving obliviousness. To highlight the challenges in replicating an ORAM datastore, we propose a naive solution followed by QuORAM's replication design.

Naive solution:

As discussed in §4, deploying optimized replication solutions such as Hermes [20] or CRAQ [37] breaks obliviousness because they access varying numbers of replicas for logical read and write operations. The naive solution presented here mitigates this challenge by deploying a single round replication protocol wherein a client accesses the same number of ORAM units for both read and write operations. Note that to ensure linearizability, the sites that handle read and write requests, *read quorum* and *write quorum*, must intersect with each other (e.g., majority quorums). In this single round multicast protocol, assuming majority quorums, a client reads from a majority and writes to a majority of the ORAM units.

While this solution is efficient since a client communicates with the ORAM units only once, it violates linearizability. We show how this solution breaks linearizability by providing an example. Consider a system with 3 replicated ORAM units where clients read or write from 2 out of the 3 replicas. A client *c1* sends a write request for a data item identified by key k, $(k = v')$ to ORAM units 1 and 2. Since the communication channels are asynchronous, assume that ORAM unit 1 receives the request and updates k's value to v' while ORAM 2's write request is in-transit. Now, another client $c2$ performs two consecutive reads on key k, once from ORAM units 1 and 2 and subsequently from ORAM units 2 and 3. For each request, the client chooses a read value corresponding to the latest timestamp (typically achieved using totally ordered timestamps [23]). For the first request, the client $c2$ reads the most up-to-date value v', whereas for the second request, it reads only the older value of k.

This is a linearizability violation, as from the external client's perspective, the operations on k appear non-linear.

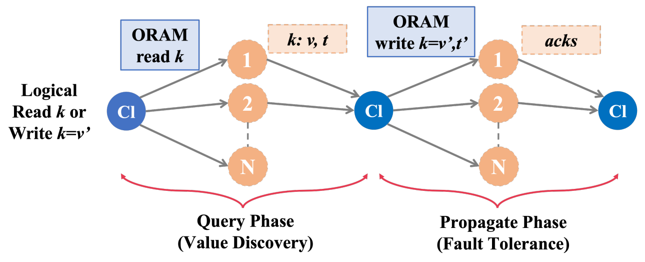
To circumvent this problem, the proxies can either deploy a locking mechanism (as is typical in database systems and as in Hermes [20]) or add another round of communication to ensure the correct ordering of requests. But employing a locking mechanism can breach obliviousness as locking leads to deadlocks, and detecting/resolving deadlocks in distributed systems requires additional communication across replica units. Since the adversary controls all communication channels, such additional communication leaks non-trivial information. Due to these reasons, QuORAM replicates the data in a lock-free approach that uses two rounds of communication between a client and the ORAM units.

QuORAM's replication

QuORAM's replication protocol design is inspired by Lynch and Shvartsman's replication protocol [27]. In designing the replication protocol, we follow the abstractions defined in the Consensus and Commitment (C&C) framework [29], which consists of four phases: *Leader election, Value Discovery, fault tolerance* , and *Decision*. The C&C framework [29] describes that most replication protocols are centralized in that one of the replicas acts as a *leader* and drives the protocol by communicating with other replicas. In such compositions, the leader node can be overloaded and become a bottleneck.

QuORAM chooses a different decentralized approach in which a client interested in reading or writing the data takes on the role of a leader and communicates with all ORAM units. This choice reduces the additional overhead on a single leader unit and avoids an adversarial case where an adversary delays the leader's communication links, thwarting the system performance.

Following the abstractions of the C&C framework, QuORAM's replication has two phases: in the first phase, a client identifies the most up-to-date value of an item by reading from a read quorum and in the second phase, it writes either the identified value (for read requests) or the updated value (for write requests) onto a write quorum of ORAM units, where the read and write quorums have non-empty intersection. Using the terminology of Lynch and Shvartsman's protocol [27], we term the first phase as the *query* phase and the second as the *propagate* phase. Given that some replica units' states may diverge due to crash or network failures, to easily identify the most up-to-date value of a given data item, each data item in QuORAM additionally maintains a monotonically increasing *tag* consisting of a sequence number and client id, $t =< seqNum, clientId >$. This is analogous to version or timestamp-based datastores.



ORAM read k
k: v, t
ORAM write k=v',t'
acks
Logical Read k or Write k=v'
Cl
1
2
N
Cl
1
2
N
Cl
Query Phase (Value Discovery)
Propagate Phase (Fault Tolerance)

Figure 3: QuORAM's replication protocol. Each circle represents an ORAM unit and a client *Cl* executes the protocol.

Overview: Figure 3 represents a high-level description of QuORAM's replication protocol. A client who wants to logically read or write a key k executes the replication protocol in two phases: query and propagate. The client first sends ORAM read requests for key k to a read quorum of ORAM units and waits to receive a response consisting of value v and tag t from the read quorum. The actions of the propagate phase depend on the type of client request: for logical reads (GETs), the client selects the value v with the highest tag t and multicasts ORAM write with v and t to a write quorum of units. For logical writes (PUTs), the client creates a new tag t' by incrementing the highest tag t (how will be explained later) and multicasts the ORAM write with v' and t' to a write quorum of units where v' is the new value. Upon receiving the ORAM write request, proxies in QuORAM update the value and tag *if and only if* the received tag t' is greater than its own tag value. The propagate phase terminates when the client receives acknowledgments from the write quorum. *For both logical read and write requests, a client considers its request to be complete only after completing both phases.*

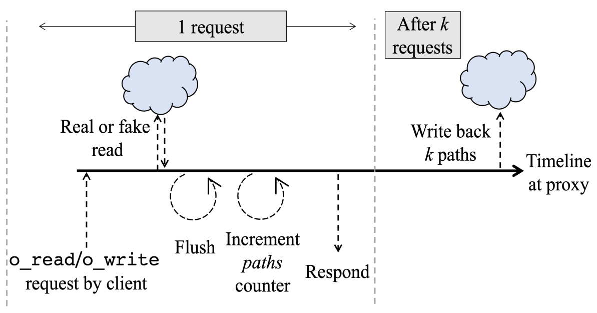
From this overview, it is clear that if a client chooses different read and write quorums in the query and propagate phases, then both sets of quorum fetch a path, shuffle, and write it back onto external servers. This creates unnecessary bandwidth and compute overheads. QuORAM addresses this issue by utilizing the same quorum for both query and propagate phases. Since QuORAM reuses read and write quorums interchangeably, we stop distinguishing between read and write quorums and impose a requirement that *any two quorums must intersect with each other* (rather than imposing read and write quorums must intersect). This way, a client can pick any quorum for both query and propagate phases. While for simplicity, QuORAM chooses majority quorums [38], i.e., sets of $\lceil (N+1)/2 \rceil$ ORAM units, the application can pick any other quorum composition that guarantees non-empty intersection between any two quorums (e.g., tree quorums [6] or grid quorums [30]). Informally, utilizing the same quorum for both the query and propagate phases does not leak any additional information since an attacker already observes what ORAM units are accessed while querying.

QuORAM's choice to communicate with only a quorum of ORAM units, instead of all, may result in a client not receiving a full quorum of responses (due to individual unit failures or message losses), even if globally, a majority of the units are alive. To ensure system progresses as long as a majority of ORAM units are live, we use timeouts to detect an unresponsive unit in a quorum and replace it with another. This brings us to the final design of QuORAM's replication protocol, whose pseudocode is described in Algorithm 1. Algorithm 1 and the rest of the paper distinguishes logical reads and writes from ORAM reads and writes by denoting logical operations as `l_read` and `l_write` (indicating GET() and PUT() requests respectively of a key value store), and ORAM operations as `o_read` and `o_write` (representing the query and propagate phase messages, respectively). Algorithm 1:

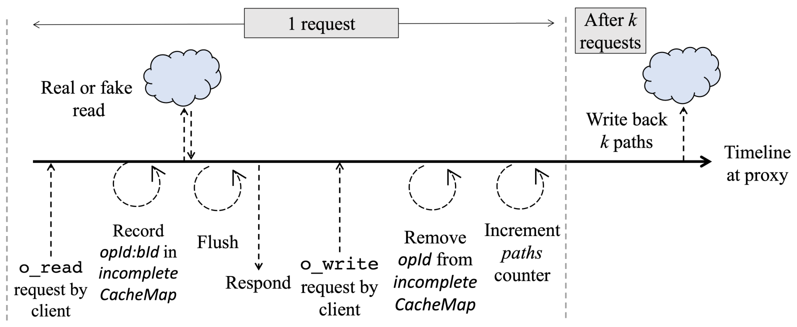
Algorithm 1 Pseudocode for QuORAM's replication protocol executed by a client with id *cId* for an operation of $opType \in$ `l_read, l_write` on block *bId* and update value v.

```
Query Phase:
1:  Q ← randomly select a set of $\lceil (N+1)/2 \rceil$ ORAM units
2:  opId ← a globally unique operation ID
3:  Multicast o_read(opId, bId) to all ORAM units in Q.
    Collect each response ($v_i$, $tag_i$), where $tag_i$ is a tuple of
    ($seqNum_i$, $cId_i$)
4:  While waiting for all responses from Q, if a read request
    sent to ORAM unit U times out:
      (a) U′ ← randomly selected unit not in Q
      (b) Q ← Q + U′ - U
      (c) Send o_read(opId,bId) to U′
5:  Upon receiving responses from all Q units, select the
    response $r$ with the highest tag
6:  If opType = l_write, set $t' \leftarrow (r.tag.seqNum + 1, cId)$
    and $v' \leftarrow v$
7:  If op_type = l_read, set $t' \leftarrow r.tag$ and $v' \leftarrow r.v'$
Propagate Phase:
8:  Multicast o_write(opId,bId,$v'$,$t'$) to all units in Q
9:  While waiting for all responses from Q, if a write request
    sent to ORAM unit U times out:
      (a) Execute steps 4(a) to 4(c)
      (b) Send o_write(opId, bId, $v'$, $t'$) to U′, without
          changing $t'$ and $v'$ sent in Step 8
10: Upon receiving acknowledgements from $Q$, the client
    considers the (logical) operation complete
```

1. A client C that wants to logically read or write a block *bId* starts the protocol by picking a quorum Q of randomly chosen majority of ORAM units (line 1).
2. The client assigns its operation a globally unique operation id, *opId*, (e.g., a sequence number and a client's unique id) as shown in line 2. This *opId*, a separate identifier from a data item's tag, is important to identify in-progress operations at both the client and proxies.



(a) Timeline of the proxy in TaORAM


1 request
After k requests
Real or fake read
Write back k paths
Timeline at proxy
o_read request by client
Record *opId:bId* in *incomplete CacheMap*
Flush
Respond
o_write request by client
Remove *opId* from *incomplete CacheMap*
Increment *paths* counter

(b) Timeline of a proxy in QuORAM

Figure 4: This figure captures the difference between the functionalities of a proxy in TaORAM vs. a proxy in QuORAM.

3. The client then multicasts `o_read(opId, bId)` to the proxies in quorum Q, who in-turn may fetch the block and the associated tag from their respective storage servers and retain it in the subtree until the block is written back (§5.2 will explain steps executed by a proxy). The client waits to receive responses consisting of the block's value *and* tag from *all* proxies in Q (line 3).
4. If the client times-out while waiting for a response from an ORAM unit U, it updates its quorum by removing U and adding another randomly selected unit U' to Q. The client then sends the `o_read` request to U'.
5. Upon receiving Q responses, the client picks the response r with the highest tag (line 5).
6. If client C's operation is `l_write`, it updates the tag (t') by incrementing the sequence number of the highest tag and updating the tag's client id to C's id and sets the value (v') to the block's new value v.
7. If client C's operation is `l_read`, it retains the highest tag (t') and its corresponding value (v') of the response r identified in Step 5.
8. Client C then broadcasts `o_write(opId, bId,` v'`,` t'`)` with the respectively updated value v' and tag t' to the proxies in Q and waits for their acknowledgements. A proxy P that receives the `o_write()` message sends an acknowledgement to C. However, the proxy P updates the value and tag ***if and only if*** the received tag t' is greater than its own tag value.
9. If the client times-out while waiting for an acknowledgement from a unit U (line 9), the client re-executes steps 4(a) to 4(c), essentially updating the quorum Q and sending `o_read` to the newly added unit U$'$. The client then sends the `o_write` request to U$'$, *without changing the value v' or tag t' sent in Step 8*, which is important to preserve linearizability. Note that even though only the write part of the operation timed-out, the client sends `o_read` before retrying `o_write` on the newly added unit to ensure the proxy fetches the necessary block and update its data structures accordingly.
10. Once the client receives acknowledgments from the quorum Q, the client considers the logical operation to be successful.

This concludes the discussion of QuORAM's replication protocol. This protocol guarantees linearizability, as will be discussed in §5.3.

5.2 QuORAM's ORAM Scheme

Having presented the replication protocol of QuORAM that preserves linearizability, this section discusses QuORAM's goal of providing obliviousness by hiding access patterns. QuORAM builds its ORAM scheme on top of TaORAM, described in §2 and we suggest reviewing it before proceeding.

Challenge of using TaORAM as-is: If proxies in QuORAM implement the ORAM scheme as-is in TaORAM, for each logical request the proxies fetch the requested block's path twice and write it back to the server twice, incurring unnecessary communication and compute overhead. The reason for the inefficiency is as follows: in a single execution of the replication protocol described in §5.1, a given proxy is either part of the quorum or not. If part of the quorum, the proxy always receives an `o_read` request in the query phase followed by an `o_write` request in the propagate phase, regardless of the type of logical request (Figure 3). Recall from §2 that for every ORAM request, TaORAM fetches a path, flushes it, and writes it back (after k requests) to the server. If the proxy treats the `o_read` and `o_write` as two separate and independent ORAM operations, then it fetches a path (real or fake) and writes it back to the server for ***both*** ORAM requests, incurring unnecessary overhead.

Solution: To mitigate the double fetching/writing of a block's paths, all proxies in QuORAM treat the two ORAM operations as correlated, and execute a single fetch and a single write-back for each logical operation. We discuss what happens when an adversary suppresses an `o_read` or `o_write` later. Figure 4 illustrates the details of a proxy's interactions between a client and its external storage in QuORAM and contrasts them with the corresponding interactions in TaORAM. We now discuss in more detail how QuORAM manages the execution of logical operations.

Challenge of asynchronously receiving `o_read` and `o_write`: QuORAM considers an `o_read` followed by an `o_write` as a single logical request, but they arrive sequentially; an adversary who controls the communication channels can control the interval between the two ORAM requests. This implies a proxy needs to *remember* for which request it has already fetched a path from the server and for which request it has not.

Solution: We achieve this by introducing a new data structure in TaORAM's Processor called incompleteCacheMap, as depicted in Figure 5. The incompleteCacheMap tracks client operations that are read but not written by mapping an operation to its requested block, i.e., *opId* to *bId*. If multiple operations access the same block, the incompleteCacheMap tracks them all. For the incompleteCacheMap, we use an LRU-based cache with a bounded number of elements for our evaluations (but any other cache design can be used). The size of the incompleteCacheMap is a system configuration and we assume the adversary knows this size.

Another change in QuORAM's ORAM scheme compared with TaORAM is in deciding when to write-back fetched paths (Figure 4). Conceptually, both ORAM schemes write-back k paths to the server after serving k requests, and both schemes track the number of requests served with a counter denoted by *paths*. But the main difference lies in how the two schemes define a single client request: TaORAM considers an `o_read` or an `o_write` as an independent, single client request, whereas QuORAM considers an `o_read` followed by an `o_write` with matching *opId* as a single client request. Due to this difference, TaORAM increments *paths* immediately after fetching a path from the server, indicating the accessed path is ready to be written back; whereas QuORAM waits until receiving the corresponding `o_write` before incrementing *paths*. Both schemes write-back when the *paths* counter value reaches a multiple of k.

Figure 5 provides the stepwise interactions between the various components of QuORAM. In the figure, Subtree, TaORAM Logic, and TaORAM Sequencer denote TaORAM's unmodified subtree, Processor and Sequencer logic (see Section 2). The steps depicted in Figure 5 are as follows:

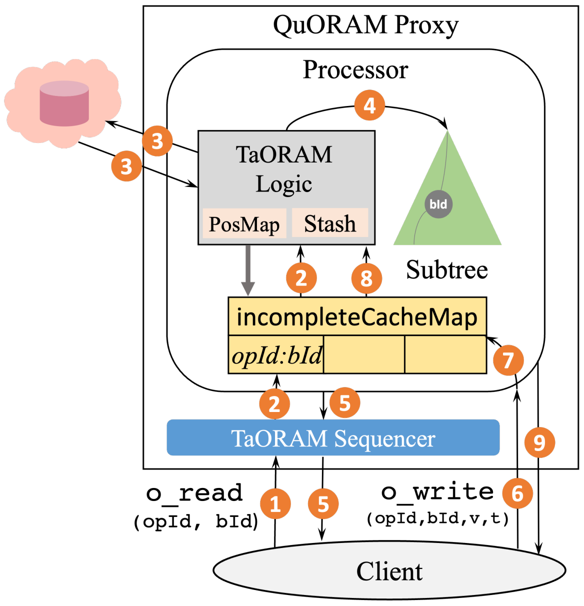


Figure 5: QuORAM's ORAM scheme built atop of TaORAM.

1. A client sends an `o_read(opId,bId)` request to a quorum of proxies (Figure depicts interaction with one). The unmodified TaORAM Sequencer records the request and forwards it to the Processor.
2. The Processor adds a new entry *opId* : *bId* to the incompleteCacheMap. If the cache is full, it evicts an entry based on the cache policy before adding the new entry; cache eviction increments *paths* (§5.2.1 describes the reasoning). The Processor then forwards the request to the TaORAM Logic, which abstractly represents all the unmodified data structures and execution logic of TaORAM's Processor.
3. The TaORAM Logic then fetches a path - real or fake - from the external server.
4. The Processor moves the fetched path, real or fake, to the Subtree.
5. Irrespective of real or fake reads from the server, the Processor sends the read response back to the client, through the Sequencer. For fake reads, the block's real value can be found either in the Subtree or the Stash. For real reads, the Processor assigns the block *bId* to a new path. The Processor then flushes the fetched path – real or fake (see §2 for details on flushing).
6. The client (after receiving responses from a quorum and updating the value and tag according to Algorithm 1) sends an `o_write(opId,bId,v′,t′)` to the chosen quorum of proxies.
7. Since `o_write` requests *do not* access the external server, they can be processed directly by the Processor bypassing the Sequencer, without breaking obliviousness. Upon receiving `o_write`, the Processor of a proxy checks if incompleteCacheMap has an entry for *opId* and *bId* : if yes, it executes step 8; if no, i.e., the cache evicted *opId* : *bId* entry in between `o_read` and `o_write`, then it executes step 9 by sending a *negative* acknowledgment to the client, indicating this request has failed.
8. The Processor removes the *opId* : *bId* entry from the incompleteCache, increments the *paths* counter and forwards the `o_write` request to TaORAM Logic. When *paths* reaches a multiple of k, TaORAM Logic asynchronously writes back k paths to the server. After receiving a write acknowledgement from the server, TaORAM Logic deletes the k paths from the Subtree. Importantly,

while deleting the paths, TaORAM Logic *does not* delete blocks that are pointed to by incompleteCacheMap.

9 The Processor then sends a positive acknowledgment to the client, and after receiving acknowledgments from the chosen quorum, the client considers its operation complete. If a client receives at least one negative acknowledgement from any proxy, it deems its request as unsuccessful. Based on the application, the client may retry the failed request.

5.2.1 Discussion on incompleteCacheMap eviction

Along with tracking ongoing client requests, incompleteCacheMap's other main role is to limit an adversary from causing a memory overflow at a proxy. An adversary can send only `o_read` messages of clients and suppress all `o_write` messages. Because the ORAM scheme fetches paths on `o_reads` and it writes-back paths and clears their memory upon receiving k `o_writes`, if a proxy receives only `o_reads` without any `o_writes`, its memory can overflow. To mitigate such adversarial behavior, we choose a limited-size cache-like datastructure that dictates how many in-progress requests a proxy can serve at a given time. As described in Step 2, if the Processor finds incompleteCacheMap to be full when a new `o_read` arrives, it evicts an entry based on the cache eviction policy and increments the *paths* counter. The counter increment is necessary to ensure a proxy writes-back paths *even if it receives no* `o_writes`. Because we assume an adversary knows the incompleteCacheMap size, writing k paths back after k combined `o_writes` and cache evictions does not leak any non-trivial information to an adversary.

An important detail for obliviousness and linearizability lies in the details of what happens when a block gets evicted from the incompleteCacheMap. *Eviction from incompleteCacheMap does not mean eviction from the proxy*. Eviction merely allows the proxy to *forget* that the evicted block had an in-progress request and allows the proxy to treat it as a block whose logical operations are complete. When the incompleteCacheMap evicts an entry, $opId : bId$, the operation's `o_write` request becomes a no-op because whatever the proxy read in the `o_read` operation is no longer guaranteed to be present in the proxy. Hence, the proxy notifies a client if its `o_write` request failed by sending a negative acknowledgement (7) and the application can decide how to handle negative acknowledgements. We assume that the adversary knows the incompleteCacheMap size; hence revealing the type of acknowledgement – positive or negative – to the adversary *does not* break obliviousness.

5.2.2 Discussion on a proxy's memory usage

As discussed earlier, QuORAM writes-back k paths to the server after serving k client requests. But as seen in step 8, after a write-back completes, QuORAM deletes only those blocks with no pointers in the incompleteCacheMap (i.e., QuORAM retains blocks accessed by ongoing requests).

Memory Issue: QuORAM's logic of not deleting certain blocks in the k paths after a write-back can cause a proxy's memory, i.e., Subtree, to grow unbounded (more precisely, it is bounded by N, the database size) if the retained blocks are never accessed again (a larger Subtree may indirectly cause a larger Stash). To see why, we consider a simple example where $k = 1$ and two concurrent logical operations $op1$ and $op2$ access the same block, $b1$. Say, a proxy receives $op1$'s `o_read` first, upon which it fetches a real path containing $b1$ from the external server. While the path is being fetched, it receives $op2$'s `o_read` and since the proxy already asked to read $b1$'s real path, it reads a fake path from the server for $op2$. When both `o_reads` are answered, the proxy receives $op1$'s `o_write`, which increments *paths* and initiates a write-back (because $k = 1$). The proxy writes the path back but cannot delete $b1$ because it has not yet received $op2$'s `o_write` request (and $op2$ read a fake path). If $op2$ updates the block and the path that block $b1$ resides on is *never* accessed and hence never written back again, then $b1$ may permanently reside in the proxy. If many such contending requests occur for different blocks at k write-back boundaries, a proxy's memory may grow unbounded. We note that in practical scenarios, this type of memory growth is improbable since clients will likely access some block in $b1$'s path over time and $b1$ will be opportunistically written back to the server, freeing it's memory. But the unbounded memory issue is a theoretical possibility.

Solution: To mitigate the unbounded memory growth problem, QuORAM creates a daemon process in the proxies wherein the daemon process *simulates* a client access every preset interval of time (e.g., 100 ms). The background process mimics both `o_read` and `o_write` requests within a proxy and that proxy fetches a path – real or fake – in accordance with the ORAM algorithm, flushes the path, and writes-back k paths after k accesses, *including the accesses generated by the background process*. We assume the adversary is aware of this behavior, where irrespective of client requests, each proxy performs its own access at regular intervals.

To further ensure that a proxy's Subtree (and hence it's Stash) does not grow in between the background thread's access intervals, we add a new datastructure called *excessBlocks*. Going back to the memory issue example, excessBlocks stores all blocks retained by the proxy after a write back to accommodate ongoing client requests. Introducing this new datastructure modifies Step 8 of the ORAM logic: after receiving a write acknowledgment of k paths from the server, a proxy moves to excessBlocks all blocks in those k paths that are pointed to by the incompleteCacheMap and which would otherwise have been deleted by TaORAM. This allows TaORAM Logic to free up all k paths from Subtree. We experimentally show (§6) that the size of excessBlocks remains low, irrespective of contention in workloads. Appendix C formally analyzes the size of Stash, which is of order $O(logN)$.

Regarding how the daemon process selects the blocks to access, it can be sequential, pseudorandom, or blocks in excessBlocks. If an application chooses to access blocks in excessBlocks, it must be noted that only blocks with no entries in incompleteCacheMap can be accessed and if no such blocks exist or if excessBlocks is empty, then the daemon process *must* continue to access blocks at preset intervals of time. Intuitively, how the daemon process selects blocks has no implications on obliviousness because this process *simulates* client requests; if an ORAM scheme hides how and what blocks are accessed by clients, then it also hides how and what blocks are accessed by the background process.

5.3 Security and linearizability of QuORAM

SECURITY:
The following theorem captures QuORAM's security.
Theorem 1: *Assuming individual ORAM units are aaob-secure, QuORAM is l-aaob-secure.*

Appendix A describes the detailed proof of the theorem. The core idea of the proof lies in how QuORAM replicates data: for *all* types of logical requests, QuORAM executes query and propagate phases. Both phases access the same number (i.e., majority) of ORAM units, even in the presence of failures. All system configurations – k the write-back frequency parameter, the incompleteCacheMap size, and the access interval of a proxy's daemon process – are known to an adversary, and hence any decision made based on these configurations does not leak any new information to an adversary.

LINEARIZABILITY:
Theorem 2: *QuORAM provides linearizability.*

Arguing for linearizability – defined per data item – in replicated data systems, especially semi-honest ones, is non-trivial. Appendix B provides a detailed proof of how QuORAM guarantees linearizable semantics.

Intuitively, QuORAM's linearizability proof captures two main relations between any two operations: (i) the tag values of any two completed logical operations have a strict less-than or less-than-or-equal-to relation; and (ii) a given logical operation – read or write – is atomic. The former point captures the relative ordering of logical operations and this order is particularly important for conflicting operations. The latter point implies that if an operation op_i wrote a block, then an operation op_j immediately succeeding op_i must read the block written by op_i; and if operation op_i merely read a block without writing it, then operation op_j immediately succeeding op_i must also read the same value as op_i. We further note that even a compromised client executing QuORAM's replication protocol does not violate linearizability.

6 Evaluation

In this section, we discuss QuORAM's experimental evaluations and contrast its performance with multiple baselines. Of particular interest is a baseline that resembles Obladi [13]'s approach to fault tolerance. As noted earlier, to date Obladi is the only other ORAM-based system that tolerates trusted proxy failures. Obladi achieves this by relying on the fault tolerance guarantees of cloud databases; Obladi pushes the necessary state of the proxy periodically to the external fault-tolerant database and recovers the proxy's state from the database if and when the proxy fails. While Obladi provides many additional guarantees, such as oblivious ACID transactional guarantees, we focus on its design choice for fault tolerance.

	N.California	Ohio	N. Virginia
N. California	6.3ms	51.32ms	62.19ms
Ohio	53.34ms	3.24ms	13.26ms
N. Virginia	63.48ms	11.98ms	4.87ms

Table 1: RTT latencies across different datacenters in ms.

While replication forms the core of fault tolerance, the two systems choose contrasting designs to replicate data: Obladi relies on the external cloud database to manage replicas and QuORAM manages replicas itself. To precisely measure how the choice of replication affects performance, we build a baseline consisting of a single TaORAM proxy (since TaoStore is the basis of QuORAM's ORAM scheme) that relies on a fault-tolerant open source database, CockroachDB [36], to replicate data. The goal of this baseline is to contrast the performance when an ORAM datastore (such as Obladi) relies on a replicated database for fault tolerance vs. using QuORAM.

6.1 Experimental Setup

We evaluated QuORAM and its baselines on AWS using r5.xlarge instances with 32GB of memory, Intel Xeon Platinum 8000 CPU with 4 cores @ 3.1GHz, and a gp2 SSD. Storage servers for QuORAM and its baselines persist the data on disk. We run our experiments on three different datacenters N. California, Ohio, and N. Virginia and Table 1 records the round-trip-time (RTT) latencies across and within the three datacenters. All the experiments place an ORAM unit (server & proxy) and a client process in each datacenter. Each client process creates 100 concurrent threads to achieve concurrency. We believe this reflects a setup for real-world applications where geo-distributed clients access data replicated across different datacenters. Note that we chose a replication factor of 3 as current state-of-the-art databases typically choose a replication factor of 3 [1,2].
Baselines:

Along with the CockroachDB-backed baseline, we evaluate QuORAM with 2 other baselines as well. Note that all baselines and QuORAM receive requests from geo-distributed clients. The 3 baselines are:
1. Insecure Replication Baseline: To measure the cost of providing obliviousness guarantees, we compare QuORAM with an insecure replication baseline that implements QuORAM's replication protocol (§5.1). More precisely, a client

queries from a majority quorum; for reads it picks the value corresponding to the highest tag and for writes it increments the highest tag and updates the value; it propagates the (potentially updated) tag and value to the same quorum it read from. In this baseline, the clients interact directly with the datastore replicas, eliminating the need for proxies, and clients do not encrypt their data or perform any ORAM related operations.

2. Secure No Replication (TaoStore): To measure the costs and benefits of fault tolerance, we use as a baseline the original non-replicated TaoStore [31] design consisting of a trusted proxy and an external server, both located in N. California. We choose TaoStore as the non-replicated baseline over other concurrent ORAM schemes because QuORAM 's ORAM logic closely relates to TaoStore's and hence, TaoStore forms a better baseline for evaluating the costs-benefits of replication, without accounting for performance differences due to ORAM scheme disparities.

3. CockroachDB Baseline: This baseline deploys TaoStore for obliviousness guarantees and CockroachDB [36] for fault tolerance (via replication managed by CockroachDB). We use a single trusted proxy (analogous to Obladi's single-proxy design) placed in N. California and a three-node CockroachDB cluster with replicas distributed across N. California, Ohio, and N. Virginia data centers, similar to QuORAM's setup.

6.2 Implementation details

We implemented QuORAM as well as the three baselines by modifying an open-source Java implementation of TaoStore, which forms the base ORAM scheme of QuORAM. The implementation consists of ∼9,400 lines of Java code. The implementation of QuORAM and its baselines can be found in https://github.com/SeifIbrahim/QuORAM/. To evaluate the systems, we use YCSB-like [12] benchmarking.

The storage server stores 1 GB of data with a block size of 4096 bytes and a bucket size of 4 blocks (i.e., each node in the tree stored at the external server consists of 4 blocks). To simulate an increasing load on the system, multiple client threads request logical read/write operations. By default, the experiments use 300 concurrent and geo-distributed clients accessing data at once (unless noted otherwise in an experiment). Each client chooses a type of operation at random, sends the request, waits for the response, and then repeats the process. Each run of the experiment lasts three minutes, and all clients end exactly the same time. For each operation, the block to be read or written is chosen randomly among all the blocks in a Zipfian distribution with an exponent of 0.9 (unless stated otherwise in an experiment), and the operation type is picked uniformly at random between read and write. In all the experiments, each data point represents an average of 3 runs and also marks the confidence interval. For system configurations, we use a default value $k = 40$ and the daemon process pseudo-randomly accesses blocks every 100ms, and an incompleteCacheMap of size 1000 blocks.

	Query phase	Propagate phase
QuORAM	12ms	0.55ms
Insecure Replication	0.05ms	0.03ms

Table 2: Processing time spent in the query and propagate phases by replicas in QuORAM vs. the Insecure replication baseline.

6.3 Experimental Results

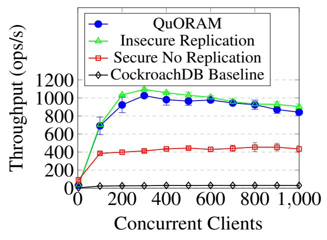
6.3.1 Throughput and Latency

In the first set of experiments, we compare the throughput and latency of QuORAM with the three baselines and see the most counter-intuitive result in this work. Figures 6a and 6b respectively show throughput and latency observed while increasing the number of concurrent clients.

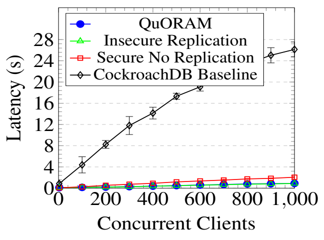
i. QuORAM vs. Insecure Replication Baseline

We first compare QuORAM with an insecure baseline that replicates data using QuORAM's replication protocol (§5.1). As seen in Figures 6a and 6b, QuORAM's throughput and latency values are closely comparable with that of the insecure baseline despite QuORAM providing privacy and obliviousness guarantees. To better understand the minor performance differences between QuORAM and the insecure baseline, we measured the average processing times spent by a replica in both the query and propagate phases of the two protocols. Table 2 records the processing time breakdown. As noted in the table, QuORAM's query phase requires the most time because a proxy communicates with its server to fetch a path. This includes 3-6ms intra-datacenter communication latency (Table 1). The proxy also decrypts the read path, merges it with the Subtree, and flushes the path, all of which incur processing latency. Meanwhile, the propagate phase merely updates a block in the Subtree. Although as noted in Table 2, the processing time for both phases of the insecure baseline requires extremely low latency compared to QuORAM, the communication cost (Table 1) overwhelms the processing time of either protocols, causing both protocols to be latency bound. Due to this reason, both QuORAM and the insecure baseline have comparable performances. This experiment indicates that in geo-replicated datastores, the overhead of encrypting and hiding access patterns of data is negligible compared to communicating with geo-distributed replicas.

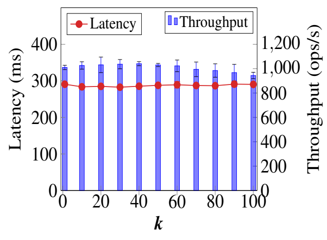
ii. QuORAM vs. Secure No Replication Baseline This baseline compares QuORAM's performance with a non-fault tolerant baseline (TaoStore as-is). Because replication involves additional communication with replicas and maintaining additional data structures (e.g., incompleteCacheMap), one can expect a replicated solution to perform worse than its non-replicated counterpart. The reason why QuORAM outperforms a non-replicated TaoStore datastore is because TaoStore consists of a single proxy, located in N. California, which receives an increasingly higher number of concurrent client requests, whereas the client load is balanced across the three proxies in QuORAM. More importantly, since the experiment consists of geo-distributed clients and the proxy resides in just



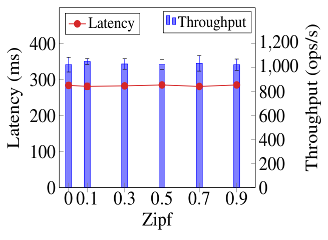
(a) Throughput vs Concurrency



(b) Latency vs. Concurrency



(c) Throughput and Latency while Varying *k*



(d) Throughput and Latency while Varying Zipf Coefficient

Figure 6: (a) QuORAM's throughput is comparable with the Insecure replication baseline, 2.4x of the No Replication baseline, and 33.2x higher than using CockroachDB for fault tolerance. (b) QuORAM's latency is comparable with the Insecure replication baseline, whereas the No replication baseline and CockroachDB suffer from a bottle-necked single proxy. (c) Varying the write-back frequency parameter *k* has no significant effect on throughput or latency of QuORAM. (d) Varying Zipfian exponent to produce low to high contention workloads has no significant effect on throughput or latency of QuORAM.

one location for TaoStore, the clients farther from the proxy face large access latencies, reducing the overall performance. Due to both load balancing and geo-replication, QuORAM's peak throughput is **2.4x** that of the non-replicated baseline.

iii. QuORAM vs. CockroachDB Baseline

Finally, comparing QuORAM with a replicated ORAM scheme that relies on a fault-tolerant database, CockroachDB, both in terms of throughput and latency, QuORAM clearly outperforms CockroachDB. The two main reasons causing CockroachDB to perform poorly are: (i) This baseline also consists of a single proxy that utilizes the read/write interface of CockroachDB to read and write the data on the external database. This single proxy, located in N. California, suffers from the same bottleneck issues as the non-replicated baseline. To mitigate the single proxy bottleneck, deploying multiple proxies – where a client communicates with any one proxy to access data – is a non-trivial task. This is because each access updates only one proxy's position map, stash, and subtree data structures, and the other proxies now have inconsistent data or position maps. Such solutions can neither guarantee linearizability nor obliviousness; (ii) The second reason causing CockroachDB to perform poorly is its choice of replication design: CockroachDB has a single leader for a given data item and this leader sequentially replicates data across replicas. Because of this single leader approach, since every read or write operation accesses the root node of the ORAM storage tree, *all* client operations are executed sequentially. QuORAM, on the other hand, employs a decentralized replication protocol, mitigating the single leader bottleneck. Because of the above two bottlenecks, CockroachDB performs worse with increasingly concurrent client requests.

6.3.2 Varying write-back threshold *k*

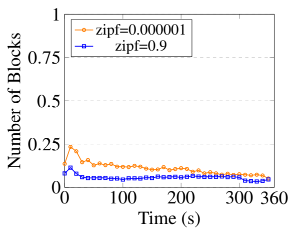
This set of experiments measures the throughput and latency of client accesses while varying the write-back threshold *k*, as seen in Figure 6c. The parameter *k* resembles a batching threshold: the higher the value of *k*, the higher the number of paths written back together and vice versa. Although proxies in QuORAM process and maintain larger number of paths locally with higher *k* values, it also results in fewer write-backs. Moreover, because a background thread executes write-backs, *k* values do not have a significant impact on throughput (with a range of 980-1030 ops/sec) or latency (about 290 ms), as can be seen in Figure 6c. This indicates that the QuORAM's performance is independent of the frequency of write-backs.

6.3.3 Varying contention

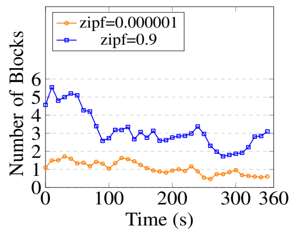
This experiment measures QuORAM's performance – throughput and latency – while varying the contention levels in client generated workloads and the results are shown in Figure 6d. Low contention, achieved by setting Zipfian exponent close to 0, implies clients select blocks uniformly at random from a pool of 262,140 blocks (the size of our dataset). High contention, achieved by setting Zipfian exponent to 0.9, indicates clients pick a small percent of the blocks (e.g., 10%) with a high probability. Typically, in non-oblivious datastores, contention in client workloads directly impacts the performance with higher contention causing low performance and vice versa. But the performance of an oblivious datastore, such as QuORAM, must remain independent of the contention in client workloads; otherwise an adversary can infer contention in client workloads just by observing requests served per second. As Figure 6d clearly indicates, QuORAM's throughput and latency values remain mostly constant with increasing contention (increasing Zipfian exponent) in client workloads. This experiment highlights the effectiveness of QuORAM in remaining impervious to contention in client workloads.

6.3.4 Stash and excessBlocks size analysis

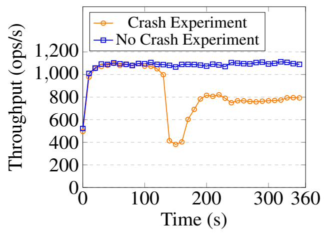
In the next set of experiments, we measure the average sizes of Stash and excessBlocks data structures over a 10-second window, calculated for a duration of 6 minutes, as shown in Figures 7a and 7b, respectively. Both figures depict the size of the respective data structures for two different Zipfian distributions in client workloads: Zipfian exponent close to 0



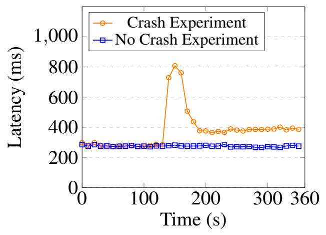
(a) 10-second moving average number of blocks in Stash



(b) 10-second moving average number of blocks in excessBlocks



(c) Throughput over time



(d) Latency over time

Figure 7: (a,b) The number of blocks in both Stash and excessBlocks remains low. The Stash's 10-second moving average size is under 1 block (implies the Stash has at least one block in the last 10 seconds) and excessBlocks at peak has 6 blocks ($= 0.33 \cdot logN$ where N=262140 blocks and $logN \approx 18$). (c,d) When an ORAM unit crashes, after a short adjustment period, both throughput and latency values stabilize and the stabilized values are higher for throughput and lower for latency compared to the non-replicated baseline.

(≈ 0.00001) indicates low contention (i.e., most requests access unique blocks) and the Zipfian exponent of 0.9 implies high contention (i.e., most requests access a small subset of blocks). Moreover, this experiment executes with the write-back threshold k set to 1. The reason we choose to analyze the sizes of Stash and excessBlocks with varying contention and with $k = 1$ is because of the memory issue discussed in §5.2.2. Recall that the memory issue is caused when say two logical operations access the same block and the second operation triggers a fake read. If the second operation's `o_write` arrives *after* the proxy initiates a write-back, the proxy cannot delete the block after receiving a write acknowledgment from the server (as TaoStore would have). To ensure the size of Subtree, which impacts the size of Stash, remains low, we move blocks that cause the memory issue into excessBlocks. Because excessBlocks's size can vary based on contention as well as when the write-back occurs frequently, we measure its sizes across two extreme contention values and the worst case write-back threshold. First, analyzing the Stash size, Figure 7a highlights that the size of the stash remains less than 1 over a 10-second window, matching QuORAM's theoretical Stash size guarantees of $logN$. Second, analyzing the size of excessBlocks, Figure 7b indicates that even though excessBlocks's size in larger for high contention, for both high and low contention workloads, it's size remains low (at worse ($0.33 \cdot logN$) with N=262140 and $logN = 18$). We note that choosing various strategies of how the daemon process in a proxy accesses blocks – sequential, pseudorandom, or blocks from excessBlocks – has no significance on the size of excessBlocks. This experiment clearly highlights that both Stash and excessBlocks remain small for all types of contention in workloads.

6.3.5 Crash Experiment

The final experiment measures QuORAM's performance when one (N. California) of the three ORAM units crashes when 300 clients execute operations and the crashed unit remains unavailable for the remainder of the experiment. The throughput and latency over time is depicted in Figures 7c and 7d respectively. As the figures indicate, the throughput drops and the latency increases steeply as soon as the crash occurs; both values stabilize afterward. In both figures, QuORAM's throughput stabilizes at ~800 ops/s and latency stabilizes at ~400ms. Even when failures occur, QuORAM performs better than the non-replicated baseline. The drop in QuORAM's throughput, which is ~300 ops/s, is roughly one-third of the overall throughput ~1080 ops/s. In fact, the reason the drop in throughput is less than one-third of the total throughput (~300 instead of ~360) is because this experiment crashes the proxy in N. California, which adversely affects only one set of clients. Whereas the clients in Ohio and N. Virginia continue to benefit from forming a quorum of two nearby proxies (Table 1). This experiment shows that QuORAM performs better than the non-replicated baseline even while tolerating f ORAM unit failures.

7 Related Work

While the literature on ORAM schemes consists of many works [7, 13, 16, 25, 31–34], to date, Obladi [13] by Crooks et al. is the only system to consider the fault tolerance aspect of an ORAM system. While Obladi provides transactional (ACID) guarantees in an ORAM setting, it compares to QuORAM in its *durability* or fault tolerance aspect. Obladi assumes the external and untrusted cloud storage server to be inherently fault-tolerant – a property guaranteed by most cloud providers – and relies on this guarantee to make the ORAM proxy fault-tolerant as well. Obladi pushes the state of the stateful proxy to the external server at periodic intervals; if the proxy crashes, it is restored to the last state pushed to the server. QuORAM has two main advantages over Obladi's design choice of fault tolerance : i) in spite of backing up the proxy's state at set intervals, Obladi becomes unavailable *during* proxy failures and recovery, and ii) as shown in the experiments, relying on cloud providers for fault tolerance incurs performance penalties compared to QuORAM's choice of fault tolerance . Another work EHAP-ORAM [24] relies on Non-volatile Memory (NVM) based hardware to persist data to recover from crashes. But the proposed solution cannot be generalized for non-NVM based ORAM datastores.

In Pharos [41], Zakhary et al. are one of the first to demonstrate the challenges of extending ORAM schemes to include replication. The authors show that naively replicating an ORAM system leaks non-trivial sensitive information. However, no correct ORAM fault-tolerant solution is proposed.

In a separate line of work, many works [10, 25, 26, 32–34, 42] have looked at extending a single ORAM server model to multi-server, multi-cloud settings. In SSS-ORAM [34] Stefanov et al. propose partitioned ORAM: an ORAM of N items split into $\sqrt{N}$ ORAMs, each of $\sqrt{N}$ size, albeit with a single cloud assumption. In [26], Lu et al. propose a distributed two-server ORAM from a theoretical perspective. They show that with two non-colluding servers, client bandwidth can be reduced to O(logN). In [32] Stefanov et al. extend [34] to propose a multi-cloud oblivious storage solution to reduce client-cloud bandwidth cost. An ORAM of N items is split across two non-colluding servers where after each data block's access, the two servers perform *two-cloud shuffling* to randomly shuffle the accessed block before its next access. In [25] Liu et al. build on [32] to optimize both the client-server and the cloud-cloud bandwidths, leading to reduced overall response time. Oblivistore [33] extends SSS-ORAM [34] to not only incorporate asynchronous concurrency but also to distribute an N item ORAM into multiple servers. The work also proposes ways to dynamically add ORAM nodes and external storage servers. CURIOUS [7] proposes a simpler solution to distribute data across multiple storage servers and serves concurrent client requests. ConcurORAM [10] allows a constant c number of concurrent clients to query at a time and requires APIs for fine-grained locking and additional datastructures from the server.

While the above works extend a partition-based ORAM scheme ([34]) to multi-server or multi-cloud schemes, in [42] Zhang et al. extend the tree-based ORAM ([35]) into a two-server setting by splitting the storage tree across two non-colluding servers to enhance performance. While the above proposals distribute data across storage servers, their deployment uses a single proxy. Recently Snoopy [14] partitions the data *and* the proxies where for scalability, proxies executing on trusted hardware serve different sets of client requests.

The main differences between prior proposals [14, 25, 26, 32–34, 42] and QuORAM are: i). the former proposals are non-replicated, i.e. each server stores a disjoint set of data items, whereas in QuORAM all servers store the same set of data items; ii) the former proposals are not fault-tolerant and can lose the data if a server or an ORAM client fails, unlike in QuORAM that tolerates server and ORAM client failures.

8 Conclusion

This work proposed QuORAM, a quorum-replicated ORAM datastore that provides fault tolerance and linearizable semantics. To date, QuORAM is the first system to replicate data while preserving obliviousness by hiding access patterns. QuORAM's novel replication protocol avoids locking – a standard technique to guarantee linearizability in distributed data systems – as employing locking can leak non-trivial information. Because QuORAM's replication protocol chooses a decentralized design, QuORAM performs **33.2x** better in throughput compared to relying on CockroachDB for fault tolerance, which consists of a centralized replication protocol. QuORAM's evaluation with a non-replicated ORAM baseline establishes the performance benefits of replication: due to geo-replication, clients can access data from close-by replicas thus causing QuORAM's peak throughput to be **2.4x** of the non-replicated baseline. Finally, the experiments indicate that QuORAM incurs negligible overhead to achieve obliviousness compared to the cost of fault tolerance due to communication among geo-distributed replicas.

Acknowledgements

We thank the Usenix Security reviewers for their insightful feedback and our shepherd Patrick McDaniel for his help in revising the paper. Sujaya Maiyya is partially supported by an IBM PhD Fellowship. Huijia Lin is supported by NSF grants CNS-1936825 (CAREER), CNS-2026774, a JP Morgan AI research Award, a Cisco research award, and a Simons Collaboration on the Theory of Algorithmic Fairness. Stefano Tessaro is partially supported by NSF grants CNS-1930117, CNS-1926324, CNS-2026774, a Sloan Research Fellowship, and a JP Morgan Faculty Award. This work is partially funded by CNS-1703560.

This work is dedicated to the loving memory of Caitlin Scarberry. This project started as part of her Masters Research requirements in 2018-19. In Spring 2019, Caitlin graduated from UCSB and joined Facebook. We are deeply saddened that she passed away in October 2021. Caitlin was an amazing person and had exceptional acumen as a student and researcher. This work benefited extensively from her insights. Her loss is immense for all of us and the research community.

References

[1] Default Replica Count For CockroahDB. https://www.cockroachlabs.com/docs/stable/configure-replication-zones.html. Accessed Jan 10, 2021.

[2] Default Replica Count For Spanner. https://cloud.google.com/spanner/docs/instances. Accessed Jan 10, 2021.

[3] Neflix uses AWS for all compute and storage needs. https://aws.amazon.com/solutions/case-studies/netflix/. Accessed October 5, 2021.

[4] Spotify backend infrastructure moves to Google Cloud. https://variety.com/2016/digital/news/spotify-goes-cloud-no-more-data-centers-1201712891/. Accessed October 5, 2021.

[5] Twitter selects AWS to power user feeds. https://press.aboutamazon.com/news-releases/news-

release-details/twitter-selects-aws-strategic-provider-serve-timelines/. Accessed October 5, 2021.

[6] Agrawal, D., and El Abbadi, A. An efficient and fault-tolerant solution for distributed mutual exclusion. *ACM Transactions on Computer Systems (TOCS) 9*, 1 (1991), 1–20.

[7] Bindschaedler, V., Naveed, M., Pan, X., Wang, X., and Huang, Y. 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* (2015), pp. 837–849.

[8] Boyle, E., Chung, K.-M., and Pass, R. Oblivious parallel ram and applications. In *Theory of Cryptography Conference* (2016), Springer, pp. 175–204.

[9] Cash, D., Grubbs, P., Perry, J., and Ristenpart, T. Leakage-abuse attacks against searchable encryption. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security* (2015), pp. 668–679.

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

[11] Chow, R., Golle, P., Jakobsson, M., Shi, E., Staddon, J., Masuoka, R., and Molina, J. Controlling data in the cloud: Outsourcing computation without outsourcing control. In *Proceedings of the 2009 ACM Workshop on Cloud Computing Security* (New York, NY, USA, 2009), CCSW '09, ACM, pp. 85–90.

[12] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing* (2010), pp. 143–154.

[13] Crooks, N., Burke, M., Cecchetti, E., Harel, S., Agarwal, R., and Alvisi, L. Obladi: Oblivious serializable transactions in the cloud. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)* (2018), pp. 727–743.

[14] Dauterman, E., Fang, V., Demertzis, I., Crooks, N., and Popa, R. A. Snoopy: Surpassing the scalability bottleneck of oblivious storage. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (2021), pp. 655–671.

[15] Demertzis, I., Papadopoulos, D., Papamanthou, C., and Shintre, S. {SEAL}: Attack mitigation for encrypted databases via adjustable leakage. In *29th {USENIX} Security Symposium ({USENIX} Security 20)* (2020), pp. 2433–2450.

[16] Goldreich, O., and Ostrovsky, R. Software protection and simulation on oblivious rams. *J. ACM 43*, 3 (May 1996), 431–473.

[17] Grubbs, P., Lacharité, M.-S., Minaud, B., and Paterson, K. G. Learning to reconstruct: Statistical learning theory and encrypted database attacks. In *2019 IEEE Symposium on Security and Privacy (SP)* (2019), IEEE, pp. 1067–1083.

[18] Herlihy, M. P., and Wing, J. M. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. 12*, 3 (July 1990), 463–492.

[19] Islam, M. S., Kuzu, M., and Kantarcioglu, M. Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012* (2012).

[20] Katsarakis, A., Gavrielatos, V., Katebzadeh, M. S., Joshi, A., Dragojevic, A., Grot, B., and Nagarajan, V. Hermes: a fast, fault-tolerant and linearizable replication protocol. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (2020), pp. 201–217.

[21] Kellaris, G., Kollios, G., Nissim, K., and O'Neill, A. Generic attacks on secure outsourced databases. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), pp. 1329–1340.

[22] Kornaropoulos, E. M., Papamanthou, C., and Tamassia, R. Data recovery on encrypted databases with k-nearest neighbor query leakage. In *2019 IEEE Symposium on Security and Privacy (SP)* (2019), IEEE, pp. 1033–1050.

[23] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (1978), 558–565.

[24] Liu, G., Li, K., Xiao, Z., and Wang, R. Ehap-oram: Efficient hardware-assisted persistent oram system for non-volatile memory. *arXiv preprint arXiv:2011.03669* (2020).

[25] Liu, Z., Li, B., Huang, Y., Li, J., Xiang, Y., and Pedrycz, W. Newmcos: towards a practical multi-cloud oblivious storage scheme. *IEEE Transactions on Knowledge and Data Engineering 32*, 4 (2019), 714–727.

[26] Lu, S., and Ostrovsky, R. Distributed oblivious ram for secure two-party computation. In *Theory of Cryptography Conference* (2013), Springer, pp. 377–396.

[27] Lynch, N. A., and Shvartsman, A. A. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS '97)* (Washington, DC, USA, 1997), FTCS '97, IEEE Computer Society, pp. 272–.

[28] Maiyya, S., Ibrahim, S., Scarberry, C., Agrawal, D., El Abbadi, A., Lin, H., Tessaro, S., and Zakhary, V. Quoram: A quorum-replicated fault tolerant oram datastore. *Cryptology ePrint Archive* (2022).

[29] Maiyya, S., Nawab, F., Agrawal, D., and Abbadi, A. E. Unifying consensus and atomic commitment for effective cloud data management. *Proceedings of the VLDB Endowment 12*, 5 (2019), 611–623.

[30] Naor, M., and Wieder, U. Scalable and dynamic quorum systems. *Distributed Computing 17*, 4 (2005), 311–322.

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

[32] Stefanov, E., and Shi, E. Multi-cloud oblivious storage. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), pp. 247–258.

[33] STEFANOV, E., AND SHI, E. Oblivistore: High performance oblivious cloud storage. In *2013 IEEE Symposium on Security and Privacy* (2013), IEEE, pp. 253–267.

[34] STEFANOV, E., SHI, E., AND SONG, D. Towards practical oblivious ram. *arXiv preprint arXiv:1106.3652* (2011).

[35] STEFANOV, E., VAN DIJK, M., SHI, E., FLETCHER, C., REN, L., YU, X., AND DEVADAS, S. Path oram: an extremely simple oblivious ram protocol. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), ACM, pp. 299–310.

[36] TAFT, R., SHARIF, I., MATEI, A., VANBENSCHOTEN, N., LEWIS, J., GRIEGER, T., NIEMI, K., WOODS, A., BIRZIN, A., POSS, R., ET AL. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (2020), pp. 1493–1509.

[37] TERRACE, J., AND FREEDMAN, M. J. Object storage on craq: High-throughput chain replication for read-mostly workloads. In *USENIX Annual Technical Conference* (2009), no. June, San Diego, CA, pp. 1–16.

[38] THOMAS, R. H. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems (TODS) 4*, 2 (1979), 180–209.

[39] TOPLE, S., JIA, Y., AND SAXENA, P. Pro-oram: Practical read-only oblivious {RAM}. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2019)* (2019), pp. 197–211.

[40] WILLIAMS, P., SION, R., AND TOMESCU, A. Privatefs: A parallel oblivious file system. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), pp. 977–988.

[41] ZAKHARY, V., SAHIN, C., EL ABBADI, A., LIN, H., AND TESSARO, S. Pharos: Privacy hazards of replicating oram stores. In *Proceedings of the 21th International Conference on Extending Database Technology, EDBT 2018* (2018).

[42] ZHANG, J., MA, Q., ZHANG, W., AND QIAO, D. Tskt-oram: A two-server k-ary tree oram for access pattern protection in cloud storage. In *MILCOM 2016-2016 IEEE Military Communications Conference* (2016), IEEE, pp. 527–532.

Appendix

A Security of replicated ORAM datastores

This section discusses obliviousness of QuORAM. Recall the ORAM scheme and the security definitions defined in Section 4.2. While the underlying ORAM scheme TaORAM [31] is proved to be *aaob-secure* (adaptive asynchronous obliviousness), QuORAM extends *aaob-secure* definition to include *logical operations* and defines *l-aaob-security* in Section 4.2. Logical operations are client requested read and write operations, which may internally consist of ORAM read and write operations. *l-aaob-secure* is an indistinguishability based security definition defined using a security game $\mathcal{G}$ in Section 4.2.

Theorem 1: Assuming individual ORAM units are *aaob-secure*, QuORAM is *l-aaob-secure*.
Proof (Sketch): Sahin et al. proved the obliviousness of TaORAM in [31]. The most important property of TaORAM (and tree-based ORAMs in general) is that every logical access translates into fetching a random path from the server to the TaORAM Processor, right after the Processor receives the logical access request. TaORAM achieves this by initially randomly shuffling the dataset before uploading to the storage server, and assigning a new random position to a block after each access. The position map in TaORAM's Processor keeps track of the random positions of all blocks.

We here focus only on the obliviousness of QuORAM, showing that it is *l-aaob-secure*. The security game $\mathcal{G}$ is defined in Section 4.2. Because the actual proof involves similar steps as TaORAM's, we omit the full proof due to lack of space but we outline the main steps necessary for the formal argument. The following are the key properties of QuORAM in arguing for its *l-aaob-security*:

1) During initialization, the game shuffles the data set D_b (after encryption) chosen by the adversary as done in TaORAM. Note that a consequence of this is that no two external servers store D_b in the same order.

2) QuORAM's replication protocol always accesses a quorum (majority) of ORAM units for the query phase and the same quorum for the propagate phase. An adversary $\mathcal{A}$ observing the communication between a client and the ORAM units sees 2 rounds of communication between the client and a quorum, for either type of logical operations, irrespective of the address or content of the block accessed.

3) In executing a logical operation, a proxy, p, is either part of the quorum or not. If p is part of the quorum, it always receives `o_read` before `o_write` (if `o_read` was dropped, the proxy sends negative acknowledgement for the `o_write`).

4) Given the fixed order of ORAM read and write requests for each logical request, in response to `o_read`, a proxy always fetches exactly one random path, either real or fake, from the server. There are three ways in which a path may become ready to be written back to the server. 1) The client sends an `o_write`, and then the path fetched for the corresponding `o_read` becomes ready to be written back. 2) The incompleteCacheMap becomes full and it chooses an entry to evict according to the eviction policy; the path associated with that entry becomes ready to be written back. 3) A path fetched by the daemon process is ready to be written back. When the number of paths to be written back accumulates to k, the proxy writes them back in a batch. Importantly, the adversary can predict the trigger for each of the case above, since 1) it observes every `o_read` and `o_write` requests from the client and knows the random path fetched for each `o_read`, 2) it can deduce the entries that reside in incompleteCacheMap and when it becomes full and which entry should be evicted, and 3) the adversary predicts the access from the daemon process (based on the preset interval). Therefore, observing the write-backs to the server reveals no non-trivial information.

5) The incompleteCacheMap in QuORAM identifies blocks that are read but not yet written. Maintaining this information crucially avoids re-fetching a path from the server for a given logical request. Further, even if the incompleteCacheMap evicts an in-progress block, the proxy still retains the block locally until it is written back to the server.

6) If an adversary $\mathcal{A}$ crashes either a server or a proxy, especially in the middle of a query or a propagate phase, $\mathcal{A}$ observes the client, executing the protocol, randomly access another ORAM unit and send two sequential requests (query followed by propagate) to this additional unit.

7) The game notifies the completion of a logical operation to the adversary only *after* a quorum of ORAM units complete executing both the query and propagate phase. If the adversary delays scheduling one or more messages in either of the phases, it receives delayed notification from the game.

In the security game (defined in game $\mathcal{G}$ in §4.2), an adversary generates two data sets of the same size D_0 and D_1 and schedules multiple but finite pairs of logical requests $(lop_{0,m}, lop_{1,m})$, where m identifies each request pair generated by the adversary. The game randomly picks the challenge bit $b \in \{0,1\}$ and stores only D_b in QuORAM and executes only $lop_{b,m}$ from each request pair. To store D_b in QuORAM, the game calls Rep-ORAM on D_b by invoking $D^b_{encK_i}, K_i \leftarrow \mathsf{Encode}_i(D_b)$ for each ORAM unit i. The external server and the proxy of an ORAM unit i store the encrypted data D_{encK_i} and the secret key K_i, respectively. The game executes QuORAM's replication protocol as defined in §5.1 for each logical request $lop_{b,m}$. The adversary does not see the output value of any operation it schedules (if it did, it would be trivial to guess the challenge bit). To prove that QuORAM is *l-aaob-secure*, we need to argue that an adversary has negligible advantage over randomly guessing the value of challenge bit b.

To do this, we show that from the adversary's point of view, it cannot distinguish a real execution of the game with a simulated game that does not use D_b or $lop_{b,m}$ for either b. First, instead of storing D_b, the simulated game stores encryption of dummy blocks (e.g., zero-value) and replaces block values in each $lop_{b,m}$ logical request also with encryption of dummy blocks. Next, it simulates the view of the adversary as follows:

(i). For each `o_read` request, a quorum (majority) of ORAM unit proxies are accessed; (ii). After the first access, the proxies always fetch one random path from the server and upon receiving the server response, proxies send a (response) message back; (iii) For each `o_write` request, the same quorum of ORAM unit proxies are accessed the second time, and they return to the client a small (acknowledgement) message; (iv) The simulator keeps track of the paths that are ready to be written-back triggered by `o_write`, as well as entries evicted from the incompleteCacheMap and accesses by the daemon process, and batch-write k paths back to the server, whenever k paths become ready.

Based on the above discussed properties of QuORAM, we assert that the adversary cannot distinguish the access behavior in the real and simulated cases, even in the presence of crash failures. This implies the *l-aaob-secure* of QuORAM.

B Linearizability

As noted in TaoStore [31], the correctness of a read or write operation differs from the obliviousness of the operation. Similar to TaORAM [31], QuORAM defines correctness using linearizability or *atomic semantics*: to an external observer, a client operation appears to take effect at a specific instance between the operation's invocation and its response indicating the operation's success. This section proves the correctness of QuORAM.

To argue for the correctness of QuORAM, we use the game $\mathcal{G}$ defined in Section 4.2 where the adversary schedules logical read/write operations but with a slight modification where the adversary now receives the response values and hence the challenge bit is non-existent. We call the modified game $\mathcal{G}_{corr}$ and use it in arguing correctness.

Definitions: A history Hist represents a sequence of logical read/write operations, viewed as the transcript after executing game $\mathcal{G}_{corr}$. Each operation op_i in Hist consists of an invocation event inv_i and a response event resp_i (which occurs after a successful propagate phase in QuORAM). A history is said to be complete if for every invocation event inv_i in the history there exists a corresponding response event resp_i; and otherwise the history is said to be partial.

We represent each operation op_i as $(op_{id}, bId, tag_i, v_i, u_i)$ where op_{id} identifies a globally unique logical operation, bId identifies a data block, tag_i represents a non-decreasing tag associated with the block, v_i equals $\perp$ for read operations and otherwise block's value to be updated with, and u_i indicates the existing value of the block prior to executing op_i, derived by a client after the query phase of op_i.

Similar to [31], $\leq_{lin}$ defines a linearizable relation between any two operations op_i and op_j: $op_i \leq_{lin} op_j$ implies resp_i precedes inv_j in a given history. We note that linearizability is defined for a single data block, i.e., both op_i and op_j operate on the same block bId. Given a complete and finite history of operations executed by QuORAM, this section proves QuORAM is linearizable, provided any adversary $\mathcal{A}$ eventually delivers all messages (after delaying and/or reordering).

Lemma 1: A block bId's response value u_i, derived by a client after a successful query phase of an operation op_i, corresponds to bId's highest tagged value.

Proof: Since each logical request in QuORAM reads from and writes to a (majority) quorum, there exists at least one over-lapping ORAM unit between any two logical requests. For each ORAM unit, TaORAM [31] guarantees that the unit maintains fresh-subtree invariant: "The contents on the paths in the local subtree and stash are always up-to-date, while the server contains the most up-to-date content for the remaining blocks". Thus, when a client executes the query phase of a logical operation op_i, at least one ORAM unit answers with block *bId*'s value u_i corresponding to the highest tag (either from the ORAM unit's proxy or the server), proving Lemma 1 holds. □

Lemma 2: Tags of a block *bId* maintained by an ORAM unit (either at the proxy or at the server) are monotonically non-decreasing.
Proof: As described in Algorithm 1, clients in QuORAM either retains tag values (for reads) or increments them (for writes) but never decrements tag values. Lemma 1 shows that a client always receives the highest tag for a block while executing the query phase, which it may retain or increment based on the type of the operation. Further, as discussed in §5.1, an ORAM unit's proxy updates a block's tag after receiving an `o_write` request *if and only if the new tag is greater than the block's current tag*. Based on the above arguments, it is shown that Lemma 2 holds. □

In our proposed system, linearizability captures two main relations between any two operations in a history: (i) the tag values of any two completed logical operations have a strict $<$ or $\leq$ relation; and (ii) a given logical operation – read or write – is atomic. The former point captures the relative ordering of logical operations. The latter point implies that if an operation op_i wrote a block, then an operation op_j immediately succeeding op_i must read the block written by op_i; and if operation op_i merely read a block without writing it, then operation op_j immediately succeeding op_i must also read the same value as op_i. We formally define the two relations captured by linearizability as follows.
Definition 1: A complete and finite history Hist is linearizable if for any two logical operations $op_i = (bId, tag_i, v_i, u_i)$ and $op_j = (bId, tag_j, v_j, u_j)$, and op_i, $op_j \in$ Hist, the following conditions hold:

❶ if op_i precedes op_j, then (i) $tag_i < tag_j$ if op_j is a write operation, or (ii) $tag_i \leq tag_j$ if op_j is a read operation.

❷ if op_i precedes op_j such that tag_i is the highest tag less than or equal to tag_j, then (i) $u_j = v_i$ if $v_i \neq \perp$ (op_i is a write), or (ii) $u_j = u_i$ if $v_i = \perp$ (op_i is a read).

Theorem 2: QuORAM provides linearizability.
Proof: ❶ To prove the first condition, we consider the two possible types of operations op_j can be:

(i) *If op_j is a write*: From Lemma 1 and 2, a logical write always increments the highest tag of a block. Since op_j is a write, and op_i may or may not be, due to the quorum intersection, op_j receives the highest tag in its query phase and increments it. Hence, the tag of op_j is strictly greater than that of op_i.

(ii) *If op_j is a read*: From Lemma 1 and 2, given the tag of a block is monotonically non-decreasing, we know that $tag_j \not< tag_i$, as op_i precedes op_j. Since tags are incremented only on writes, if no write took place between op_i and op_j, then $tag_i = tag_j$; whereas if a write operation op_k occurred after op_i and before op_j, then $tag_i < tag_k$ (from step (i)), and by transitivity, $tag_i < tag_j$. This is true for any number of write operations between op_i and op_j. Hence, $tag_i \leq tag_j$.

❷ Given that tag_i is the highest tag less than or equal to tag_j, irrespective of the type of operation of op_j, due to Lemma 1, when op_j executes the query phase, it receives the current highest tag of the block, i.e., tag_i and its associated value. (i) Now, if op_i wrote the block, then the block's value is v_i and hence when op_j queries the block, it receives v_i. Thus $u_j = v_i$. This shows that writes are atomic as any operation executing after a write reads the updated value.

(ii) If op_i merely read the value, which was equal to u_i, then since op_j immediately succeeds op_i for block *bId*, op_j's read value also equals u_i as no other operation updated the block. Thus $u_i = u_j$. This shows that reads are atomic. □

C Stash size analysis

This section analyzes QuORAM's stash size and the space utilized at the proxy can be found in the extended report [28].
Lemma 3: Similar to TaORAM, QuORAM's stash size is bounded by any function $F(N) = \omega \cdot logN$, except with negligible probability in N.
Proof: The core idea of this proof lies in mapping the execution of QuORAM to that of TaORAM in a straight-forward way. TaORAM's stash size is proved to be bounded by a function $F(N) = \omega \cdot logN$ (e.g., $F(N) = (logloglogN) \cdot logN$) and by mapping QuORAM's execution to that of TaORAM we prove that QuORAM has the same stash size guarantees as TaORAM.

To analyze QuORAM's stash size, recall the details of the unbounded space issue and its solution discussed in §5.2.2. The memory issue is caused due to the asynchrony in receiving `o_read` and `o_write` requests for a logical request; if a proxy initiates a write-back in between receiving the two requests, and if the `o_read` had triggered a fake read, the proxy cannot delete the block after receiving a write acknowledgement from the server. This is because the block's latest `o_write` arrived *after* the proxy initiated the write-back. In the unlikely case that this block or any block in its path is never accessed again, this block will always reside in the Subtree. This may in-turn affect the size of the Stash. QuORAM mitigates this issue by moving such blocks to excessBlocks datastructure and the daemon process in each proxy accesses (i.e., mimics `o_reads` and `o_writes`) blocks in the excessBlocks at pre-set intervals of time. This can be viewed as, from TaORAM's perspective, all blocks that can be deleted after receiving a write-back acknowledgement from the server will be deleted from the Subtree (and some may move to excessBlocks). As seen with this abstraction, QuORAM relies on TaORAM's logic of freeing the Subtree, without any changes, and hence QuORAM's stash size analysis follows that of TaORAM and the size is bounded by any function $F(N) = \omega \cdot logN$, except with negligible probability in N. □