EasyChair Preprint

№ 15459

RouterORAM: an O(1)-Latency and Client-Work ORAM

Sumit Paul and David Knox

EasyChair preprints are intended for rapid dissemination of research results and are integrated with the rest of EasyChair.

November 24, 2024

RouterORAM: An $O(1)$-latency and client-work ORAM

Sumit Kumar Paul[0000−0002−6232−8090] and D.A.Knox[0009−0007−6138−2283]

University of Ottawa, Ottawa, Canada {Sumit.Paul,dknox}@uottawa.ca

Abstract. An adversary can learn a lot only by studying data storage access patterns, even if the actual data being accessed remains encrypted. Oblivious RAM (ORAM) is a cryptographic primitive that hides access patterns. However, to achieve this privacy, the client has to perform a significant amount of additional work per access, which not only causes very high access latency but is also often impractical for resource-constrained clients. As a result, ORAMs are still not usable in most of the scenarios. This paper proposes *RouterORAM*. The central idea of it is to harness the server's otherwise unutilized computation power to steer the deliberately misplaced blocks to their destined locations. To the best of our knowledge, *RouterORAM* is the first ORAM to drag both the access latency and the client's burden together down to the asymptotic minimum level, $O(1)$. It exploits the properties of homomorphic encryption to achieve the desired level of server obliviousness, and its privacy is proven with rigorous theoretical analysis. The long-term behavior of *RouterORAM* is captured with simulation, which vouches for its suitability for practical usage scenarios.

Keywords: ORAM · Latency · Client-work · Homomorphic Encryption

1 Introduction

Increasingly, users are storing and accessing larger amounts of personal data from cloud storage [20]. However, even if the data remains encrypted, a server can learn a lot of private information just by monitoring access patterns [6]. For instance, during a file access, a user/client requests the server to send the encrypted content of the file (or a part of it). Hence, the server learns which file (or which part of the file) is being accessed. The server can detect a file write/update operation by observing any change in the ciphertext content of the accessed location.

Golreich and Ostrovsky first analyzed this problem and proposed the Oblivious RAM (ORAM) algorithm [2]. In an ORAM, the client stores data as encrypted blocks in random locations on a remote server, keeps track of those locations, and later retrieves the required blocks from those random-looking stored locations and decrypts them locally.

With the ORAM method, additional work, re-encryption and re-shuffling of some part of the server content, is required for *each* distinct access. These additional works not only make reading and writing indistinguishable, but they also hide the access frequency of the blocks. However, this overhead increases access latency significantly. Moreover, for the sake of privacy, all these additional tasks are required to be done by the client, which is often impractical for resource-constrained client devices.

Several ORAM schemes have been proposed over the last thirty years, with the aim of minimizing the overhead. However, there is a limitation. The amount of overhead has

been proven to be at least $O(log N)$ per access [2], where N is the number of outsourced blocks. This means that, with the trend of increasing size of remote storage, ORAM will become even less practical.

The proof covers the most general use case scenario, where the client can read or write any item, in any order, at any time. For example, the client might remain active and access the remote storage around the clock, and, out of N-outsourced blocks, might only write to a particular block every time.

In reality, the client's access patterns are less extreme and occur in bursts rather than being continuous accesses [14]. Moreover, reads occur more often than writes [12, 13]. Therefore, rather than attempting to protect all possible access patterns (already proven to have $\Omega(log N)$ overhead), we restrict the problem space to protecting these *common* access patterns and can then reduce latency and client's work significantly.

We made some interesting observations. In any ORAM scheme, the client is required to store a large number of encrypted dummy blocks in the server for the purpose of obfuscation. However, after encryption, dummy and real blocks become indistinguishable. So, we might take some advantage by storing encryption of the replicas of real blocks instead of storing the encryption of dummies. Another observation is that there is an implicit assumption that the server can only read and write blocks whenever the client instructs and is idle otherwise. However, since the server has processing capability, theoretically, it could perform some useful work in that idle time.

By considering these observations and targeting the specified common access patterns, we propose *RouterORAM*, which protects the users' access pattern. The novel aspect of *RouterORAM* is: to the best of our knowledge, it is the first ORAM that can protect access pattern privacy with only $O(1)$-latency **and** $O(1)$-client work. The name *RouterORAM* comes from the idea of utilizing the server's computational capability to obliviously route the deliberately misplaced blocks to their destined locations. Additionally, *RouterORAM* stores multiple replicas of each blocks on the server. So, if the client requests a block that has not yet reached its destination, the server can return a replica.

The organization of this paper is as follows: We discuss the related work in Section 2. In Section 3, we describe *RouterORAM* in detail. In the subsequent section, we conduct a theoretical analysis and prove its claimed privacy. Section 5 shows the simulation results. In the next section, we discuss its theoretical contributions and compare *RouterORAM* with other schemes. Finally, in section 7, we conclude this paper.

2 Related work

Goldreich and Ostrovsky first envisioned the concerns of privacy leakage from access patterns and proposed hierarchical ORAM [2], which allows a client to hide its access pattern to the remote storage server. In their seminal paper, the authors not only proposed hierarchical ORAM, but also proved that $\Omega(log N)$ overhead is essential to achieve perfect privacy. However, their own construction's overhead was $O(log N^3)$ per access.

As a natural consequence, the research community started finding an ORAM scheme that could meet the theoretical limit. PathORAM [3], having $O(log N^2)$ overhead, is one of the most popular breakthroughs in this direction [7]. In PathORAM, the server stores the outsourced N data blocks in encrypted format in a binary tree having $log N$ levels. The client maps each block to a random path, which means the block resides somewhere on that path. To access a block, the client looks at its local *position map*

to find the mapped path. Then reads all the blocks on that path. The requested block is then remapped to a new random path and placed somewhere in that remapped path or in the client's local storage, called *stash*. Additionally, the client tries to push the newly remapped block towards the leaf of the remapped path whenever possible.

Finally, Asharov et. al. constructed the optimal $O(logN)$-ORAM, OptRAMa [4]. However, this theoretical minimal $O(logN)$ overhead is also impractical for medium-sized storage accesses. Generally, $O(logN)$-overhead does not only mean the client has to incur $O(logN)\times$ computation; it also means the client has to incur $O(logN)\times$ bandwidth while accessing each block. Ren et al. proposed Ring ORAM [5], in which they showed that it is possible to reduce the bandwidth blowup from $O(logN)\times$ to $O(1)\times$ by keeping some metadata in the server along with the outsourced data. Also, they used the *XOR*-trick, in which the server *XOR*s multiple encrypted blocks together and sends a single compressed response to the client. However, unlike the bandwidth reduction, it is not possible to reduce the computation overhead of an ORAM scheme below $O(logN)$ due to its theoretical bound, *at least when both read and write operations are supported*.

Roche et al. proposed a write-only ORAM, WoORAM [8], which surprisingly has only $O(1)$-overhead. However, having no read capability makes it less useful as, in most situations, the client outsources some data to remote storage with the intention of accessing (reading) it later. Tople et al. proposed PRO-ORAM [9], a read-only ORAM having $O(1)$ overhead. It uses a trusted execution environment on the server side which launches $\sqrt{N}$ threads to shuffle the entire storage in $O(1)$ time after each access. However, if N is not relatively small, launching $\sqrt{N}$ treads is not practical, even for a powerful server.

Reducing latency might be possible if the shuffling can be broken down into online and offline phases. The online phase occurs when the client accesses the remote storage for read/write. Some bare minimum tasks must be done during this phase; it determines the client-observed access latency. The offline phase can be postponed to a less busy period when the client and server jointly perform the remaining shuffling tasks. Stefanov and Shi proposed ObliviStore [10], which requires only a *constant* amount of reshuffling work in its online phase, resulting in an $O(1)$-latency ORAM. Dautrich et al. extended it to cope with the client's bursty access patterns and proposed Burst ORAM [11]. Burst ORAM does not perform any reshuffling work at all during the online phase, so it becomes even faster. However, here, ultimately, in *some idle* period, the *client* has to perform all the accumulated pending tasks to keep the ORAM in a usable state. In fact, if the burst size is too large, the client might have to download the entire database locally and perform $O(N)$ of work, which might be too much for its *idle* period.

In the majority of the situations, the remote server is treated only as a *storage device*, having no processing capability. On client's instruction, it can only read and write from certain locations. In reality, the remote server is independently capable of performing some useful work. So, theoretically, the client must be able to outsource some part of the reshuffling task to the server. The only challenge is, how it can be done without leaking any private information. If the client stores the data in the remote server after encrypting with fully homomorphic encryption (FHE), then theoretically, it is possible for the server to perform any computation on that, which might help the client to outsource some part of the reshuffling task to the server. Apon et. al. analyzed this idea and formalized the notion of *Verifiable Oblivious Storage* [15].

Devadas et al. applied this idea and proposed a PathORAM-based construction, Onion ORAM [16]. Additionally, in Onion ORAM the bandwidth is reduced by treating each path of the server tree as a remote database and fetching the required block from that path by performing PIR. After accessing each block, it is remapped to a new random path but placed in the root. When the root is about to become full, the client and server collaborate and evict the old blocks from it to make space for newer ones. During the eviction phase, instead of downloading the encrypted data and locally performing reshuffling, the client only downloads some small metadata, and *instructs* the server about how to do the reshuffling. The server performs the homomorphic evaluation over the server data as per the client's instruction which results the intended reshuffling.

Subsequently, Chen et al. improved Onion ORAM and proposed Onion Ring ORAM [17]. They showed how the homomorphic permutation can be done more efficiently and how by performing homomorphic expansion on the server side, the bandwidth requirement can be reduced even more. Not only that, they showed that FHE-based ORAM schemes are no longer theoretical possibilities by implementing their construction with the currently available FHE building blocks like cMUX-gate. However, the computation done by the client, especially during the offline eviction phase, is still quite heavy.

Recently Cong et al. proposed Panacea [18], another FHE-based ORAM scheme. It does not require any involvement of the client during the offline phase at all and achieves $O(1)$-client work per access. However, the access latency is high in this case. Naively, in Panacea, the server is required to perform $O(N)$ amount of computation per query. However, if only 1.5$\times$ bandwidth and 3$\times$ expansion in server storage are sacrificed, then the concept of probabilistic batch coding can be applied. In that case, instead of responding to queries individually, the server can process queries in a batch of size k, bringing the amortized computation cost per query down to $O(N/k)$.

3 RouterORAM Protocol

In this section, we first give an informal overview of the protocol. Then, we describe the details of the outsourced data and required storage. Finally, we discuss all the parts of our protocol in detail and how the client can access the remote storage whilst hiding the access pattern.

3.1 Overview of RouterORAM

RouterORAM is a PathORAM [3]-based construction; the server stores the outsourced blocks in a binary tree structure. After accessing a block, the client remaps that block to a new random leaf. However, to confuse the server, the client next deliberately "misplaces" the block on the server (i.e. places the block in a different and unrelated random position from the newly remapped leaf).

The server always runs a background routing process, ensuring that all of the misplaced blocks eventually reach their remapped location. To facilitate routing, the destination location (i.e. the remapped leaf) is specified in the block metadata. The server utilizes that information to route the block to its proper destination. Routing is done by leveraging the properties of homomorphic encryption, and the server learns nothing.

In *RouterORAM*, only a constant amount of work is required during each block access. Indeed, only two constant-size buckets need to be *touched*[1] at all on the server.

[1] In ORAM context, *touching* means replacing the existing ciphertext with a new ciphertext.

Beyond this $O(1)$-work, no other work is required from the client or server during each access. As a result, *RouterORAM* achieves $O(1)$-latency. However, the server's routing work does not disappear. *RouterORAM* allows the server to perform that work at more convenient times, with better distribution over time, which has advantages in the common case where servers have both busy and idle periods.

In *RouterORAM*, the client can store multiple replicas of each block in the server. In this way, some replicas of a block may be present in their mapped locations while others are in transit. As long as at least one replica is *at rest*, the content of the block can be accessed. In addition to reading and writing a block, *RouterORAM* offers a third feature: controlling the number of replicas. So, by dynamically adjusting the number of replicas of the blocks according to their anticipated access frequencies, the client can effectively mask the time the server requires to complete the routing.

We use some notations in our paper, which are summarized in Table:1.

Table 1. Notations

Notation	Explanation
N	Total number of distinct outsourced blocks, same as the number of leaves.
L	Height of the binary tree ($\lfloor logN \rfloor + 1$).
a	Identifier of a block, $a \in [N]$.
$\mathbb{D}(a)$	Data of the block, identified by a (or block a).
$\#(a)$	Number of replicas of block a, stored in the server.
Z	Number of slots in a bucket. Each slot can hold an encrypted block.
β_b	The bucket having label b, where $b \in_{\mathbb{N}} [1, 2^{L-1}]$.
$\beta_b[i]$	Content of the ith slot of β_b.
$\overline{(\cdot)}$	Ciphertext of $(\cdot)$ generated under the fully homomorphic encryption, FHE.
$\mathsf{e}_{\hat{b},\check{b}}$	Edge connecting the upper layer bucket $\beta_{\hat{b}}$ with its child $\beta_{\check{b}}$.
blk.m.a	Block metadata part storing the identifier of the block, blk.m.a $\in [N]$.
blk.m.x	Block metadata part storing the mapped leaf label, blk.m.x $\in_{\mathbb{N}} [2^{L-1}, 2^L - 1]$.
$\langle b_1, .., b_{\#(a)} \rangle :=$ pos[a]	The content of the client-local position map corresponding to the block a. It is a list holding the details of all the replicas of the block a.

3.2 Client storage and ORAM initialization

The client is only required to store and maintain the position map, pos[], which is an array indexed by the block identifier $a \in [N]$. It stores N-different lists. The client maps different replicas of each block with independent locations in the server, and the information about all the replicas of block a is kept in the list stored in pos[a]. Because of the deliberate misplacement, all the replicas may not be available immediately (the background routing might take some time to place them) at their mapped leaf bucket.

Hence, the availability information also needs to be stored. Thus $\forall_{a \in [N]}$, pos[a] is actually a list having the format: $\langle (b_1, \tau_1), .., (b_{\#(a)}, \tau_{\#(a)}) \rangle$, where each element is a tuple consisting of the label of the mapped leaf bucket (b_i) and the expected timestamp (τ_i), when the replica will be available at β_{b_i}. The client always ensures that pos[a] always remains sorted in ascending order of the availability timestamps of the replicas.

3.3 Outsourced data

Outsourced data is stored and accessed in terms of fixed-size blocks. Each block blk, has a data part (blk.d) having size B-bits and a fixed size metadata part (blk.m). The

client wants to outsource N different blocks to the server. Each block is addressed by an identifier denoted by a, where $a \in [N]$ and it is also stored as a part of block metadata ($\mathsf{blk.m.a}$). The other part of the block metadata ($\mathsf{blk.m.x}$) stores the block's destination (i.e., the currently mapped leaf). In *RouterORAM*, the client might store multiple replicas of each individual block in the server. $\#(a)$ denotes the number of replicas of block a, and the client is allowed to control this number from time to time. Along with *real* data blocks (and their replicas), some *dummy* blocks are also outsourced. The plain text content of a dummy block is all zeros (i.e., $\mathsf{blk.m.a} = \mathsf{blk.m.x} = 0$ and $\mathsf{blk.d} = \{0\}^{\mathsf{B}}$).

3.4 Server storage

The server arranges its storage as a full binary tree. Each node of the tree is a bucket having a fixed number (Z) of slots. $\ell \in_{\mathbb{N}} [1, L]$ denotes the levels of the tree, where the root has $\ell = 1$ and all the leaves have $\ell = L$. In each level ℓ, the tree has $2^{\ell-1}$ nodes/buckets. In *RouterORAM*, the number of leaf buckets is set to N, which means, $L = \lfloor log N \rfloor + 1$.[2] Each individual bucket is labeled with $b \in_{\mathbb{N}} [1, 2^L - 1]$. The bucket having label b is denoted by β_b. The labeling is done in such a way that for bucket β_b, β_{2b} is its left child, and β_{2b+1} is its right child. The root bucket is labeled with $b = 1$.

Each bucket-slot stores an encrypted real or dummy block. $\overline{\beta_{\mathsf{b}}[\mathsf{i}]}$ represents the stored ciphertext in i^{th} slot of β_b and $\beta_{\mathsf{b}}[\mathsf{i}]$ represents corresponding plaintext. Leaf buckets are special; they additionally store a list having Z-elements, called the invalidation list ($\beta_{\mathsf{b}}.\mathsf{il}$). Elements of $\beta_{\mathsf{b}}.\mathsf{il}$ are either a valid block identifier $\in [N]$ or 0. Since everything is encrypted under an IND-CPA secure fully homomorphic encryption scheme, FHE, real blocks, their replicas, and dummy blocks are not distinguishable from each other.

ORAM initialization In this one-time activity, the client securely outsources real data blocks (along with some replicas) and some dummy blocks to the remote server. *Alg:1*, shows the details, where both the client and server participate jointly during the red-dashed steps, and the rest of the steps are performed only by the client.

The client first chooses a randomly permuted order of to-be-outsourced blocks. Then, it encrypts the content of each replica (i.e., data and metadata) with FHE and stores that in a uniform randomly chosen leaf bucket. Accordingly, the client updates its local position map. Since, during initialization, each replica is placed on the mapped leaf, they are available immediately. Hence, the client specifies the current timestamp (τ_{cur}) as the expected availability for each replica. After completing real data outsourcing, the client fills the remaining empty slots of the server with the encryption of dummy blocks.

3.5 ORAM Access

In *RouterORAM*, the client can interact with the server through only one basic building block, $\mathsf{Access}(\mathsf{a}^{\mathsf{R}}, \mathsf{a}^{\mathsf{I}}, \mathbb{D}(\mathsf{a}^{\mathsf{I}}))$-call (*alg:2*). During each $\mathsf{Access}()$-call, one block, a^{R} is removed from the server tree, and another block, a^{I} having the data content $\mathbb{D}(\mathsf{a}^{\mathsf{I}})$ is inserted. By attentively choosing the parameter set of the $\mathsf{Access}()$-calls, the client can achieve the high-level operations: reading a block, writing a block, and controlling the number of replicas. *RouterORAM* ensures that individual $\mathsf{Access}()$-call leaks no privacy. Thus, any higher-layer operations built upon it will also remain privacy-preserving.

[2] Unless otherwise stated, in this paper $log(\cdot)$ denotes $log_2(\cdot)$

Algorithm 1 Init()

```
1: π ← Choose a random permutation of [N]
2: for a ∈ π do
3:     for i ∈ [#(a)] do
4:         b_i ←$ [2^{L-1}, 2^L - 1]
5:         pos[a] ∪ (b_i, τ_cur)
6:         Store {a, b_i, D(a)} in β_{b_i}
7: for Remaining empty slots do
8:     Store dummy in an empty-slot of the server tree
```

Algorithm 2 Access(a^R, a^I, $\mathbb{D}(a^I)$)

```
1: data ← Remove(a^R)
2: if Removal fails then
3:     Insert(0, *)  ▷ Dummy insertion to maintain same touch pattern
4:     Access(a^R, a^I, D(a^I))  ▷ Try again
5: else  ▷ Successfully removed
6:     if Insert(a^I, D(a^I)) ≠ success then
7:         Access(0, a^I, D(a^I))  ▷ Partial retry with dummy removal
8: return data
```

Algorithm 3 Remove(a^R)

```
1: data ← ∅
2: if a^R ≠ 0 then
3:     (b_1, τ_1) ← popFront(pos[a^R]), x ← b_1
4:     Fetch β_x from server
5:     β_x ← FHE.dec(β_x)
6:     for i ∈ [Z] do
7:         if β_x[i].a = a^R then
8:             data ← β_x[i].d, β'_x[i] ← dummy
9:         else
10:            β'_x[i] ← β_x[i]   ▷ i.e.,re-encrypt
11:    if data = ∅ then
12:        β_x.il ← β_x.il ∪ {a^R}
13:    β'_x.il ← β_x.il
14:    Replace β_x with β'_x in server
15: else  ▷ It's a dummy removal
16:    x ←$ [2^{L-1}, 2^L - 1]
17:    Replace β_x in the server, with its re-encrypted version β'_x
18: return data
```

Algorithm 4 Insert(a^I, $\mathbb{D}(a^I)$)

```
1: st ← failure
2: x ←$ [2^{L-1}, 2^L - 1], w ←$ [1, 2^{L-1} - 1]
3: if a^I ≠ 0 then
4:     Fetch β_w from server
5:     β_w ← FHE.dec(β_w)
6:     for i ∈ [Z] do
7:         if (β_w[i].a = 0) and st ≠ success then
8:             β'_w[i] ← {a^I, x, D(a^I)}
9:             st ← success
10:        else
11:            β'_w[i] ← β_w[i]
12:    Replace β_w with β'_w in server
13:    if st = success then
14:        τ_exp ← calcExpTime(x, w, τ_cur)
15:        Insert (x, τ_exp) at proper location in pos[a_I]
16: else  ▷ It's a dummy insertion
17:    Replace β_w in the server, with its re-encrypted version β'_w
18:    st ← success
19: return st
```

RouterORAM does not only keep the parameter sets of the Access()-calls private but also leaks nothing from the observable trace of the server-storage accesses during its execution. On rare occasions, some steps of Access() may fail. However, *RouterORAM* ensures that observed bucket touch patterns always remain the same: touching one leaf bucket, x, followed by touching another non-leaf bucket, w. The aim is, irrespective of the situation, an adversary (in this case, the server) must not learn anything by observing the storage trace. Specifically, the ORAM must satisfy the following privacy definition.

Definition 1 (ORAM Privacy). *Let* $\vec{y} = ((a^R_A, a^I_A, \mathbb{D}(a^I_A)), .., (a^R_1, a^I_1, \mathbb{D}(a^I_1)))$ *denotes an access sequence of length A, where* $(a^R_i, a^I_i, \mathbb{D}(a^I_i))$ *denotes the parameter set of the* i^{th} Access()*-call. Let* ORAM($\vec{y}$) *be the observable trace on the server storage due to* $\vec{y}$*. Then RouterORAM guarantees that for any two access sequences* $\vec{y}_1$ *and* $\vec{y}_2$ *where* $|\vec{y}_1| = |\vec{y}_2|$*, an adversary* $\mathcal{A}$*, cannot distinguish between* ORAM($\vec{y}_1$) *and* ORAM($\vec{y}_2$).

ORAM Removal At the first step of access, the client removes the block a^R by invoking Remove()(*alg:3*). Since pos[a^R] is already sorted, the client finds the earliest available location (β_x) of a^R from the front of the list. β_x might contain up to Z different blocks, so the client fetches entire β_x from the server and decrypts all the slots of it to figure out which one is a^R.

After finding a^R, the client reads its content and makes the corresponding slot empty by writing all zeros (i.e., dummy) in it. In some special situations, a^R might not be available in β_x; in that case, the client only updates some metadata part of the bucket (detail is discussed in *section: 3.7*). At the end of Remove(), the client stores the updated bucket (β'_x) in the same leaf of the server. To protect privacy, the client re-encrypts every component of the bucket before sending that to the server.

Sometimes (e.g. to deal with failures), the client might be required to issue a dummy Remove()-call (i.e., $a^R = 0$). In that case, the client just replaces a random leaf with its re-encryption without altering anything in its local position map. The goal is to make the actual and dummy removal access pattern indistinguishable from the server.

ORAM Insertion Each removal is paired with an insertion (*alg: 4*). To insert the block a^I with data content $\mathbb{D}(a^I)$, the client first locally maps the block with a random *leaf* label: x, but deliberately misplaces the block to another random and independent *non-leaf* bucket of the server, β_w. During insertion as well, the client re-encrypts all the content of the touched bucket (β_w) in the server.

The background routing process obliviously ensures that the inserted block reaches from β_w to β_x so that the client can access the block in the future, from the newly mapped bucket, β_x. Since the background routing pattern is deterministic, and both x and w are known to the client, so the client can locally compute the expected time, τ_{exp}, when the inserted block reaches its destined leaf bucket. To maintain the sorted order of pos[a^I], after a successful insertion, the client adds (x, τ_{exp}) as the ith element of pos[a^I], such that $\tau_{i-1} \leq \tau_{exp} \leq \tau_{i+1}$.

3.6 Higher layer operations and their latency

The client can achieve higher layer operations by controlling the parameter set of the Access()-calls. Block reading and manipulating the number of replicas are straightforward. The client chooses $a^R = a_1$ and $a^I = 0$ (i.e., removes a_1 with a dummy insertion) to read the block a_1 and decrease $\#(a_1)$ by one. To read a_1, without affecting $\#(a_1)$, the client invokes Access() with $a^R = a_1$, $a^I = a_1$ and $\mathbb{D}(a^I) = \mathbb{D}(a_1)$. With $a^R = a_1$, $a^I = a_2$ and $\mathbb{D}(a^I) = \mathbb{D}(a_2)$, client can read a_1 with the effect of decreasing $\#(a_1)$ and increasing $\#(a_2)$ at the same time. Similarly, by choosing $a^R = 0$ and $a^I = a_2$ ($\neq 0$), only $\#(a_2)$ can be increased. Since all these operations mentioned above can be performed by invoking only one Access(), the latency and the client work for all these operations is $O(1)$.

Although *RouterORAM* is targeted for read-dominated access patterns, by no means block writing is unsupported. Since the server may store multiple replicas of each block, block writing is slightly sophisticated than other operations. Suppose during writing, the client wants to update the content of block a_1 from $\mathbb{D}(a_1)$ to $\mathbb{D}'(a_1)$. So, if the block a_1 currently has $\#(a_1)$-replicas in the server, all those replicas must be updated. To do that, at first, all stale replicas of a_1 must be removed from the server, and new $\#(a_1)$-replicas having the updated content must be inserted. This can be achieved by invoking

$\mathsf{Access}(\mathsf{a}^{\mathrm{R}} = \mathsf{a}_1, \mathsf{a}^{\mathrm{I}} = \mathsf{a}_1, \mathbb{D}'(\mathsf{a}_1))$, $\#(\mathsf{a}_1)$ times. As a result, the latency of the write operation will be $O(\#(\mathsf{a}_1))$. Since $\#(\mathsf{a}_1)$ is the client-controlled constant (independent of N), complexity theory-wise, the write operation will also have $O(1)$ latency. However, practically, the client has to incur $\#(\mathsf{a}_1)\times$ work (as well as $\#(\mathsf{a}_1)\times$ latency) during writing block a_1, in comparison with reading the same block.

3.7 ORAM Background processing/routing

While interacting with the client, the server runs a routing process in the background, continuously. After the client inserts block a^{I} to β_w, the routing process ensures it reaches its actual destination, β_x. For example (*fig:1*), at time instant t_1, the client maps a_1 with β_{14}, but places that at a different and random bucket, β_5. The routing process then ensures that a_1 reaches its mapped leaf bucket by going through the following path: $\beta_5 \rightarrow \beta_2 \rightarrow \beta_1 \rightarrow \beta_3 \rightarrow \beta_7 \rightarrow \beta_{14}$. Similarly, another inserted block, a_2 must reach β_{11} from β_4.

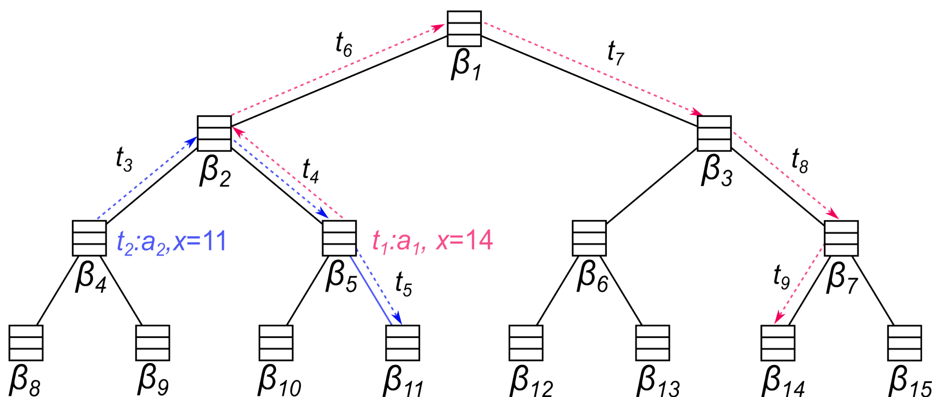

Fig. 1. ORAM tree and routing

The routing algorithm *(alg:5)* executes in a series of steps. In each step, an edge $(\mathsf{e}_{\hat{b},\check{b}})$ of the tree is processed, which connects a lower-level bucket $\beta_{\check{b}}$ with its parent, $\beta_{\hat{b}}$. During background routing, the edges are processed in a fixed order, starting from the top-left edge of the binary tree and ending at the bottom-right edge (i.e., $\mathsf{e}_{1,2} \rightarrow \mathsf{e}_{1,3} \rightarrow \mathsf{e}_{2,4} \rightarrow .. \rightarrow \mathsf{e}_{N-1,2N-3} \rightarrow \mathsf{e}_{N-1,2N-2}$). Eventually, after processing each edge (i.e., $2 \times (N-1)$ steps), the entire tree will have been processed, and the routing cycle repeats forever.

The processing of each edge is simply a homomorphic evaluation over the encrypted blocks that reside in $\beta_{\hat{b}}$ and $\beta_{\check{b}}$. As a result, it is possible for the server to do the routing correctly without knowing the details of the blocks that are in transit. In addition, the design of our routing algorithm must also ensure that the server does not learn anything from the effects of the routing as well. For example, during the execution of the routing algorithm, the ciphertext content of some locations may change while the remaining locations remain unaltered. The server may try to use that information to backtrack the path of the inserted block. Our algorithm protects against these.

Algorithm 5 Route()

1: **while** True **do**
2:   **for** $e_{\hat{b},\check{b}} \in$ {All edges in the tree} **do**
3:     $(\overline{\mu_{\hat{b}}}, \overline{\mu_{\check{b}}}) \leftarrow \mathsf{MoveVerdict}(\overline{\beta_{\hat{b}}}, \overline{\beta_{\check{b}}})$
4:     $\mathsf{Move}(\overline{\mu_{\hat{b}}}, \overline{\mu_{\check{b}}}, \overline{\beta_{\hat{b}}}, \overline{\beta_{\check{b}}})$

Algorithm 6 MoveVerdict$(\overline{\beta_{\hat{b}}}, \overline{\beta_{\check{b}}})$

1: $\ell_{\hat{b}} \leftarrow \lfloor log(\hat{b}) \rfloor + 1, \ell_{\check{b}} \leftarrow \ell_{\hat{b}} + 1$
2: **for** $i \in [Z]$ **do**
3:   $\overline{\mathsf{bit}} \leftarrow \overline{(\check{b} = (\beta_{\hat{b}}[i].m.x \gg (L - \ell_{\check{b}})))}$
4:   $\overline{\mu_{\hat{b}}[i]} \leftarrow \mathsf{CMux}(\overline{\mathsf{bit}}, \overline{\Leftrightarrow}, \overline{\nLeftrightarrow})$
5:   $\overline{\mathsf{bit}} \leftarrow \overline{(\beta_{\hat{b}}[i].m.x = 0)}$
6:   $\overline{\mu_{\hat{b}}[i]} \leftarrow \mathsf{CMux}(\overline{\mathsf{bit}}, \overline{\emptyset}, \overline{\mu_{\hat{b}}[i]})$
7: **for** $i \in [Z]$ **do**
8:   $\overline{\mathsf{bit}} \leftarrow \overline{(\check{b} = (\beta_{\check{b}}[i].m.x \gg (L - \ell_{\check{b}})))}$
9:   $\overline{\mu_{\check{b}}[i]} \leftarrow \mathsf{CMux}(\overline{\mathsf{bit}}, \overline{\nLeftrightarrow}, \overline{\Leftrightarrow})$
10:   $\overline{\mathsf{bit}} \leftarrow \overline{(\beta_{\check{b}}[i].m.x = 0)}$
11:   $\overline{\mu_{\check{b}}[i]} \leftarrow \mathsf{CMux}(\overline{\mathsf{bit}}, \overline{\emptyset}, \overline{\mu_{\check{b}}[i]})$
12: **return** $(\overline{\mu_{\hat{b}}}, \overline{\mu_{\check{b}}})$

Algorithm 7 Swap$(\overline{\mathsf{bit}}, \overline{d_0}, \overline{d_1})$

1: $\overline{\mathsf{tmp}} \leftarrow \mathsf{CMux}(\overline{\mathsf{bit}}, \overline{d_0}, \overline{d_1})$
2: $\overline{d_0} \leftarrow \overline{d_1} + \overline{d_1} - \overline{\mathsf{tmp}}$
3: $\overline{d_1} \leftarrow \overline{\mathsf{tmp}}$
4: **return** $(\overline{d_0}, \overline{d_1})$

Algorithm 8 Move$(\overline{\mu_{\hat{b}}}, \overline{\mu_{\check{b}}}, \overline{\beta_{\hat{b}}}, \overline{\beta_{\check{b}}})$

1: **for** $i \in [Z]$ **do** ▷ *Swap* $\beta_{\hat{b}} \leftrightarrow \beta_{\check{b}}$
2:   **for** $j \in [Z]$ **do**
3:     $\overline{\mathsf{bit}} \leftarrow \overline{(\mu_{\hat{b}}[i] = \Leftrightarrow) \textbf{ and } (\mu_{\check{b}}[j] = \Leftrightarrow)}$
4:     $(\overline{\beta_{\hat{b}}[i]}, \overline{\beta_{\check{b}}[j]}) \leftarrow \mathsf{Swap}(\overline{\mathsf{bit}}, \overline{\beta_{\hat{b}}[i]}, \overline{\beta_{\check{b}}[j]})$
5:     $\overline{\mu_{\hat{b}}[i]} \leftarrow \mathsf{CMux}(\overline{\mathsf{bit}}, \overline{\nLeftrightarrow}, \overline{\mu_{\hat{b}}[i]})$
6:     $\overline{\mu_{\check{b}}[j]} \leftarrow \mathsf{CMux}(\overline{\mathsf{bit}}, \overline{\nLeftrightarrow}, \overline{\mu_{\check{b}}[j]})$
7: **for** $i \in [Z]$ **do** ▷ *Move up* $\beta_{\check{b}} \rightarrow \beta_{\hat{b}}$
8:   **for** $j \in [Z]$ **do**
9:     $\overline{\mathsf{bit}} \leftarrow \overline{(\mu_{\hat{b}}[j] = \emptyset) \textbf{ and } (\mu_{\check{b}}[i] = \Leftrightarrow)}$
10:     $(\overline{\beta_{\hat{b}}[i]}, \overline{\beta_{\check{b}}[j]}) \leftarrow \mathsf{Swap}(\overline{\mathsf{bit}}, \overline{\beta_{\hat{b}}[i]}, \overline{\beta_{\check{b}}[j]})$
11:     $\overline{\mu_{\hat{b}}[j]} \leftarrow \mathsf{CMux}(\overline{\mathsf{bit}}, \overline{\nLeftrightarrow}, \overline{\mu_{\hat{b}}[j]})$
12:     $\overline{\mu_{\check{b}}[i]} \leftarrow \mathsf{CMux}(\overline{\mathsf{bit}}, \overline{\emptyset}, \overline{\mu_{\check{b}}[i]})$
13: **if** $\beta_{\check{b}}$ is a leaf bucket **then** ▷ *Invalidation*
14:   **for** $i \in [Z]$ **do**
15:     **for** $j \in [Z]$ **do**
16:       $\overline{\mathsf{bit}} \leftarrow \overline{(\beta_{\check{b}}.il[j] = \beta_{\hat{b}}[i].m.a)}$
17:       $\overline{\beta_{\hat{b}}[i]} \leftarrow \mathsf{CMux}(\overline{\mathsf{bit}}, \overline{\mathsf{dummy}}, \overline{\beta_{\hat{b}}[i]})$
18:       $\overline{\beta_{\check{b}}.il[i]} \leftarrow \mathsf{CMux}(\overline{\mathsf{bit}}, \overline{0}, \overline{\beta_{\check{b}}.il[i]})$
19:       $\overline{\mu_{\check{b}}[i]} \leftarrow \mathsf{CMux}(\overline{\mathsf{bit}}, \overline{\emptyset}, \overline{\mu_{\check{b}}[i]})$
20: **for** $i \in [Z]$ **do** ▷ *Move down* $\beta_{\hat{b}} \rightarrow \beta_{\check{b}}$
21:   **for** $j \in [Z]$ **do**
22:     $\overline{\mathsf{bit}} \leftarrow \overline{(\mu_{\hat{b}}[i] = \Leftrightarrow) \textbf{ and } (\mu_{\check{b}}[j] = \emptyset)}$
23:     $(\overline{\beta_{\hat{b}}[i]}, \overline{\beta_{\check{b}}[j]}) \leftarrow \mathsf{Swap}(\overline{\mathsf{bit}}, \overline{\beta_{\hat{b}}[i]}, \overline{\beta_{\check{b}}[j]})$
24:     $\overline{\mu_{\check{b}}[j]} \leftarrow \mathsf{CMux}(\overline{\mathsf{bit}}, \overline{\nLeftrightarrow}, \overline{\mu_{\check{b}}[j]})$

In theory, the evaluation under a fully homomorphic scheme can perform any computation. However, current publicly available homomorphic libraries (e.g., TFHE [19]) only support a few computation building blocks that can be used for the homomorphic evaluation process. Therefore, we use only those available building blocks to make our routing algorithm practically viable. Specifically, we use a few basic mathematical operations and the $\mathsf{CMux}(\overline{\mathsf{bit}}, \overline{d_0}, \overline{d_1})$-gates [19]. Depending on the encrypted control-bit input $(\overline{\mathsf{bit}})$, CMux() outputs re-encryption of either d_0 or d_1. However, since FHE is IND-CPA secure, by looking at the output, it is not possible to tell whether it is $\overline{d_0}$ or $\overline{d_1}$.

During processing of an edge $e_{\hat{b},\check{b}}$ (line:3-4 in *alg: 5*), the server actually re-arranges the blocks that reside in the slots of $\beta_{\check{b}}$ and $\beta_{\hat{b}}$. With each successive re-arrangement, each block-in-transit reduces its distance from its final destination. Referring to *fig:1*, at t_4, edge $e_{\hat{b}=2,\check{b}=5}$ is being processed. Block a_1 is currently located at β_5 and block a_2 is at β_2. After processing, $e_{2,5}$, a_1 has moved to β_2 and a_2 will end up at β_5.

To achieve this, MoveVerdict() (*alg:8*) homomorphically evaluates the metadata of all the blocks that currently reside in $\beta_{\hat{b}}$ and $\beta_{\check{b}}$ and outputs two encrypted arrays $(\mu_{\hat{b}}, \mu_{\check{b}})$ storing movement decisions regarding each individual block. If $\beta_{\check{b}}$ is nearer to the final destination of the block residing at $\beta_{\hat{b}}[i]$, then the block requires a movement from $\beta_{\hat{b}}$ to $\beta_{\check{b}}$, which is encoded as $\mu_{\hat{b}}[i] = \Leftrightarrow$. Otherwise, $\beta_{\hat{b}}[i]$ does not require a movement during the current edge processing and is encoded as $\nLeftrightarrow$. Similarly, $\mu_{\check{b}}[]$ is generated for $\beta_{\check{b}}[]$.

For the blocks of both buckets, MoveVerdict(), evaluates the same condition: whether $\beta_{\check{b}}$ is the common ancestor of the block in question and sets the bit (line 3 & 8). Depending on which, CMux() chooses the movement encoding. Dummy slots are encoded as $\emptyset$.

Finally the existing blocks of $\beta_{\hat{b}}$ and $\beta_{\check{b}}$ are being moved according to $\mu_{\hat{b}}$ and $\mu_{\check{b}}$ (*alg:8*). Basically, there could be three kinds of scenarios: a) Swap a block in $\beta_{\hat{b}}$, which requires a downward movement with a block in $\beta_{\check{b}}$ requiring an upward movement (e.g., in *fig: 1* processing $e_{2,5}$ at t_4) b) A block in $\beta_{\check{b}}$ is required to move up, but there is no block in $\beta_{\hat{b}}$ to go down. Hence, the block must be moved to an empty slot in $\beta_{\hat{b}}$ (in our example processing $e_{2,4}$ at t_3) c) Similarly, move a block down from $\beta_{\hat{b}}$ to an empty slot of $\beta_{\check{b}}$ (processing $e_{5,11}$ at t_5). Moving a block to an empty slot means, after the movement, the source slot becomes empty (i.e., stores a dummy block). Hence, interestingly, not only (a) but also scenarios (b) and (c) can be accomplished by performing Swap() operation (line: 4, 10, and 23 of *alg: 7*). After completion of the movement, the locations corresponding to the source and destination blocks are updated in $\mu_{\hat{b}}[]$ and $\mu_{\check{b}}[]$.

Note that, during writing a block, the client must remove all the existing replicas of it. However, at that moment, all the replicas might not yet reach their mapped location. So, the client is required to somehow notify the routing process to invalidate those replicas upon arrival. Client achieves that by adding the block's identity in the invalidation list (β.il) of the mapped buckets (*alg: 3*, line:12). So, before moving down any block to a leaf node (i.e., scenario (c)), Move() performs some additional checks. It evaluates the condition whether the identity of the block-in-transit ($\beta_{\check{b}}$.m.a) matches with any of the elements in $\check{\beta}$.il (line:15-16). If there is a match, then the block is not moved down but deleted by replacing the corresponding slot with dummy.

4 Privacy Analysis

We analyze the privacy properties of *RouterORAM* against an honest but curious adversary, which is the common threat model for ORAM [2, 4, 5, 8, 10, 11, 17, 18]. Since everything remains encrypted in the remote storage, the adversary $\mathcal{A}$ (the server in this case) cannot learn the outsourced data. However, $\mathcal{A}$ can gradually observe a series of ciphertext changes at specific locations on the server storage (i.e., the trace left due to the execution of the protocols). Like any other ORAM, the *RouterORAM* restricts $\mathcal{A}$ from learning anything from the generated trace due to the client's access. Additionally, it guarantees that $\mathcal{A}$ cannot learn anything during the execution of the routing process as well. Next, we analyze the privacy properties of our protocol in gradual steps.

Theorem 1 (Privacy of Remove()). *If* FHE *is IND-CPA secure, the adversary $\mathcal{A}$ learns nothing during a* Remove() *call, except x, the label of the touched leaf bucket.*

Proof. During Remove(), the client touches a leaf bucket, β_x. Since the remote storage is controlled by $\mathcal{A}$, it can notice the touched location, x. Depending on the input parameter of the Remove()-call, the content of β_x changes differently (e.g., if $\mathsf{a}^{\mathsf{R}} = 0$, then nothing is changed at all). Moreover, occasionally, the Remove() may fail. However, irrespective of the situation, the client always re-encrypts all the contents of the touched bucket. So, $\mathcal{A}$ only observes that the entire ciphertext of the touched bucket changes from $\overline{\beta_x}$ to $\overline{\beta'_x}$. Since the encryption scheme (FHE) is IND-CPA secure, $\mathcal{A}$ is neither able to learn the plaintext content of the bucket nor can guess whether any slot or the metadata of the

bucket has changed. In other words, if λ is the security parameter of FHE, then:

$$\forall_{i\in[Z]} \left| \Pr\left[\mathcal{A}(\overline{\beta_x}, \overline{\beta'_x}) \to 1 \mid \beta_x.[i] = \beta'_x.[i]\right] - \Pr\left[\mathcal{A}(\overline{\beta_x}, \overline{\beta'_x}) \to 1 \mid \beta_x.[i] \neq \beta'_x.[i]\right]\right| =$$

$$\left| \Pr\left[\mathcal{A}(\overline{\beta_x}, \overline{\beta'_x}) \to 1 \mid \beta_x.li = \beta'_x.li\right] - \Pr\left[\mathcal{A}(\overline{\beta_x}, \overline{\beta'_x}) \to 1 \mid \beta_x.li \neq \beta'_x.li\right]\right| \leq \mathrm{negl}(\lambda)$$

Thus, $\mathcal{A}$ learns nothing except the label x during a Remove() call. □

Corollary 1. *A real and a dummy removal are indistinguishable from each other.*

Proof. Suppose $\mathsf{Remove}_1()$ is a real removal which touches β_{x_1} and $\mathsf{Remove}_2()$ a dummy removal which touches β_{x_2}. According to *theorem: 1*, $\mathcal{A}$ only learns x_1 and x_2. Since the client chooses both x_1 and x_2 uniform randomly from the same distribution ($[2^{L-1}, 2^L - 1]$), $\mathcal{A}$ cannot differentiate them. □

Theorem 2 (Privacy of Insert()). *If FHE is IND-CPA secure, the adversary $\mathcal{A}$ learns nothing during an Insert() call, except w, the label of the touched non-leaf bucket.*

Proof. The argument is similar with *theorem:1*. As the client re-encrypts all the content of the touched non-leaf bucket from $\overline{\beta_w}$ to $\overline{\beta'_w}$, it is impossible for $\mathcal{A}$ to guess which part of β_w has changed, or whether anything has changed at all. Hence, $\mathcal{A}$ only learns w. □

Corollary 2. *A real and a dummy insertion are indistinguishable from each other.*

Proof. The argument is similar to *corollary: 1*. In both situations, the client chooses the touched non-leaf bucket uniform randomly from the same distribution. □

Corollary 3. *During a completely successful Access() call, $\mathcal{A}$ learns nothing except x and w, the label of the touched leaf bucket, and the label of the touched non-leaf bucket.*

Proof. During a completely successful Access(), no retrial is required and the client only makes one successful Remove() call and one successful Insert() call. Since none of them reveal anything except x and w, Access() also reveals nothing more. □

Theorem 3 (Unnoticeable failure during an Access() call). *After observing a single Access()-call, $\mathcal{A}$ cannot tell whether it has completed with any failure or not.*

Proof. Access() consists of one Remove() and one Insert(). On rare occasions, either of them may fail. As a result, overall Access() may fail as well. In the case of failure, another Access() is invoked (for retrial), but during the current Access()-call $\mathcal{A}$ cannot notice whether there is any failure.

Remove() fails when the client expects the block a^{R} ($\neq 0$) at its mapped leaf bucket β_x, but cannot find it there. In that case, Remove() returns $\emptyset$, and Access() issues a dummy insertion instead of inserting a^{I}. However, $\mathcal{A}$ can neither notice Remove() has failed (*theorem: 1*), nor that the subsequent insertion is a dummy one (*corollary: 2*). Thus, $\mathcal{A}$ cannot determine whether the Access() call completed with a failed removal or not.

Insert() fails, when the randomly chosen bucket β_w is already full, and a^{I} ($\neq 0$) cannot be inserted into β_w. Note that $\mathsf{Insert}(\mathsf{a}^{\mathsf{I}}, \mathbb{D}(\mathsf{a}^{\mathsf{I}}))$ is invoked only after a successful Remove(), and no additional step is performed within the same Access()-call, in the case of insertion failure. Since $\mathcal{A}$ cannot notice the failed insertion *(theorem: 2)*, $\mathcal{A}$ cannot say whether the observed Access() call completed with a failed insertion. □

Corollary 4. *$\mathcal{A}$ cannot distinguish between an original* Access() *call and a retrial of it.*

Proof. Suppose $\mathsf{Access}_2()$ is a retrial call, invoked because of the failure of either Remove() or Insert() during $\mathsf{Access}_1()$. So, $\mathcal{A}$ notices a pair of buckets $(\beta_{x_1}, \beta_{w_1})$ touched during $\mathsf{Access}_1()$ and another pair $(\beta_{x_2}, \beta_{w_2})$ touched during $\mathsf{Access}_2()$. The touched leaf and non-leaf locations in a single access are statistically independent. We have to show that during the retrial as well, the touched buckets are statistically independent.

If Remove() fails during $\mathsf{Access}_1()$, x_1 is removed from $\mathsf{pos}[\mathsf{a}^\mathsf{R}]$ and $\mathsf{Access}_2()$ touches β_{x_2}, the mapped location of next replica of a^R. Since mapped locations of all the replicas of a^R are chosen randomly and independently, x_2 and x_1 are statistically independent. In the case of Insert() failure in $\mathsf{Access}_1()$, a dummy removal is made during $\mathsf{Access}_2()$, which also chooses x_2 randomly and independently. As a result, x_1 and x_2 are always independent of each other.

Irrespective of the nature of failure in $\mathsf{Access}_1()$, the Insert() call during $\mathsf{Access}_2()$, chooses w_2 randomly and independently. Hence, there is no relation between w_1 and w_2. Thus, x_1, x_2, w_1, and w_2 are always statistically independent, and $\mathcal{A}$ cannot distinguish between an original Access() call and a retrial of it. □

Lemma 1. *Irrespective of the situation, by observing a series of* Access() *calls, $\mathcal{A}$ only learns the labels of a sequence of the bucket labels, nothing else.*

Corollary 5 (Access pattern privacy). *RouterORAM meets ORAM privacy definition:1*

Proof. From $\mathsf{ORAM}(\vec{\mathsf{y}})$, $\mathcal{A}$ only learns a sequence of touched bucket labels $\vec{\mathsf{b}} = \langle (x_A, w_A), .., (x_1, w_1) \rangle$ (*lemma: 1*). Since, irrespective of the situation (i.e., irrespective of the input parameters, success or failure, original or retrial, etc.), labels of all the touched buckets (i.e., $\{x_1, .., x_A\} \cup \{w_1, ..w_A\}$) are statistically independent of each other. Thus, $\mathcal{A}$ cannot distinguish between two such series of ORAM accesses: $\mathsf{ORAM}(\vec{\mathsf{y}_1})$ and $\mathsf{ORAM}(\vec{\mathsf{y}_2})$, when $|\vec{\mathsf{y}_1}| = |\vec{\mathsf{y}_2}|$. □

Theorem 4 (Privacy of Route()). *$\mathcal{A}$ learns nothing during the routing process.*

Proof. Not only the outsourced data (and the metadata) but also any information derived during routing (i.e., $\mu_{\hat{\mathsf{b}}}$, $\mu_{\check{\mathsf{b}}}$, the computed bit value in *alg: 6, 8, etc.*) always remain encrypted under FHE. The server can only process them homomorphically. By definition, the server cannot learn the underlying plain text during the homomorphic evaluation. Since FHE is IND-CPA secure, $\mathcal{A}$ cannot learn anything by observing the changes in the bucket ciphertexts due to homomorphic evaluation as well. Moreover, the pattern of touched buckets during routing is always fixed. Hence, $\mathcal{A}$ also learns nothing from the generated trace due to routing.

However, in-general the execution path of an algorithm may depend on its input (e.g., number of iterations, branching, etc.). $\mathcal{A}$ may try to closely monitor the execution flow of the routing algorithm and learn something. Nevertheless, the design of *RouterORAM* ensures that the execution flow of all the routing-related algorithms (*alg: 5- 8*) remains fixed and that all the steps are always executed. Hence, $\mathcal{A}$ cannot learn anything by observing the execution path of the routing process as well. □

5 Simulation results

Access() might fail. Fortunately, in *RouterORAM*, failure does not affect the privacy or read/write functionality. To cope with access failure, the client is just required to retry. However, if the failure rate is too high, then *effectively* the client might have to touch much more than two buckets per access. If true, then the *effective* latency and client work may no longer remain $O(1)$. So, to verify the amount of failures, we perform multiple simulations of our protocol with our simulator [1]. We simulate the steady, long-term (~5 months) behaviors of *RouterORAM* with different daily usage amounts (100MB/day to 12.8GB/day remote data access). During simulation, conservatively, we assume a very slow disk (40MB/s). *RouterORAM* is expected to produce fewer failures with the faster disks because, in that case, the server will require less time to handle the client's I/O requests and can spend more time/resources to perform routing.

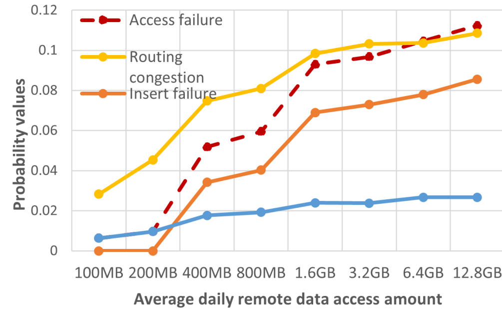

Fig. 2. Probabilities of failures and congestion

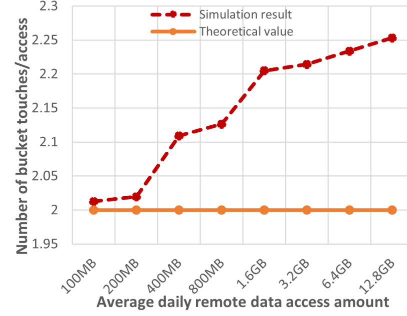

Fig. 3. Number of bucket touches

The detailed simulation results are shown in *fig: 2*. Routing congestion might occur when a block is supposed to be moved, but the destination bucket is already full. For each access call, one block is misplaced in a non-leaf bucket. So, the more frequently the client accesses the remote storage, the more slots in the non-leaf buckets remain occupied with misplaced blocks, and the probability of routing congestion increases. Routing congestion incurs some delay for the affected block in reaching its destination, which might cause Remove()-failure, but not always. Remove()-failure occurs *only if* the client tries to access the *affected* block replica *before its delayed arrival time*.

Since the expected number of full non-leaf buckets increases with access frequency, the probability of randomly chosen β_w during Insert() to be full also rises with access frequency. Which is nothing but the failure probability of Insert().

In the case of either Remove() or Insert() failure, the client invokes another Access() (i.e., additional two bucket touches), which might also fail with the same probability, and this might create an infinite series. Suppose F_A is the failure probability of Access() call, then by following the geometric distribution, we can compute the expected number of bucket touches per access is:

$$2 \times \sum_{k=1}^{\infty} k \times F_A^{k-1} \times (1 - F_A) = 2(1 - F_A) \sum_{k=1}^{\infty} k F_A^{k-1} = \frac{2(1 - F_A)}{(1 - F_A)^2} = \frac{2}{(1 - F_A)} \quad (1)$$

Since Access() fails, whenever either of Insert() or Remove() fails, F_A is the sum of the failure probabilities of Insert() and Remove(). So, by combining the results of *fig: 2* and

equation: 1 we get *fig: 3*. From this, it can be observed that the expected number of bucket touches increases with the usage amount. However, the rate of increase is very slow, and the expected number of bucket touches per access remains under 2.26 (instead of 2), even for very high usage settings. As a result, it can be safely concluded that *RouterORAM*'s latency and client work remain truly $O(1)$ in practical usage scenarios.

6 Discussions

In the past, BurstORAM [11] attempted to minimize the latency to $O(1)$ in a bursty setting. However, in their model, the client must remain active after their bursty access period to perform the pending reshuffling task. Due to this, the effective client work and bandwidth blowup per access become $O(logN)$. Also, in BurstORAM, the client must be able to store a large part of the outsourced database ($O(\sqrt{N})$) locally.

Theoretically, Apon et al. [15] first showed that with the server computation model, bandwidth blowup can be reduced to $O(1)$, and Onion-ring ORAM [17] is the latest scheme in that model. However, to date, no ORAM scheme that supports both read and write operations has been able to achieve $O(1)$-latency along with $O(1)$ client work. The theoretical contribution of this paper is to show how to achieve that by trading off the server's storage space and computation power. Another benefit is that $RouterORAM$ accomplishes this without locally storing any part of the outsourced database (stash).

Table 2. Comparisons

Scheme	Allowed modes	Bandwidth blowup	Total server storage	Total client work	Total server work	Latency
PathORAM [3]	R/W	$O(logN^2)$	$O(N)$	$O(logN^2)$	$O(logN^2)$	$O(logN^2)$
PRO-ORAM [9]	R-only	$O(1)$	$O(N)$	$O(1)$	$O(\sqrt{N})$	$O(1)$
WoORAM [8]	W-only	$O(1)$	$O(N)$	$O(1)$	$O(1)$	$O(1)$
BurstORAM [11]	R/W	$O(logN)$	$O(N)$	$O(logN)$	$O(logN)$	$O(1)$
Onion-Ring ORAM [17]	R/W	$O(1)$	$O(N)$	$O(logN)$	$O(logN)$	$O(logN)$
Panacea [3] [18]	R/W	$O(1)$	$O(N)$	$O(1)$	$O(N/k)$	$O(N/k)$
RouterORAM	R/W	$O(1)$	$O(N)$	$O(1)$	$O(logN)$	$O(1)$

Table 2 compares $RouterORAM$ with existing ORAM schemes. Although WoORAM and PRO-ORAM also have $O(1)$ latency and client work, none of them support both reading and writing. In the read/write model, Panacea recently minimized the total client work to $O(1)$ by removing offline work altogether. However, its latency is still proportional to the outsourced data size (N). From that aspect, $RouterORAM$ is the first read/write ORAM scheme that achieves $O(1)$ latency and client work. It is to be noted that the total amount of server work does not go away. It is still $O(logN)$ due to the theoretical limit of R/W ORAM.

To clarify the complexity of the write operation, it remains $O(1)$ as long as the access pattern is bursty and read-dominated, irrespective of data size or server configurations. As discussed in section 3.6, the write complexity is directly proportional to the number of existing replicas of the written block (i.e., $\#(a)$), which should be a small constant for a bursty read-dominated access, which is the targeted usage scenario of $RouterORAM$.

[3] It works in a batched setting, and k is the batch size

However, the client has the flexibility to choose any value for $\#(a)$. If this value escalates to $O(N)$, the complexity of the write operation would then increase to $O(N)$ as well.

Particularly, the choice of $\#(a) = O(N)$ makes sense only if the client creates continuous (rather than bursty) heavy traffic of write operations, that too only on the specific block a. We acknowledge that in this specific situation, the write operation of $RouterORAM$ will no longer remain efficient (however, the read will still remain $O(1)$).

Another interesting aspect of $RouterORAM$ is that it reduces the number of dummies on the server and places replicas in those spaces. As a result, it does not necessarily always consume more storage than other tree-based ORAM schemes with server computation (e.g., Onion-ring ORAM), despite trading off storage with latency. It *might* take more storage if the total number of replicas becomes quite high, which can happen if the client sets up $RouterORAM$ to support continuous (rather than bursty) accesses. But in any case, the server storage will not become unmanageable (e.g., exponential).

7 Conclusions and future work

We propose *RouterORAM*, which protects remote storage access pattern privacy with only $O(1)$-latency and $O(1)$-client work. Although it does not have any restriction regarding the client's access pattern, its benefits can be exploited maximally for a bursty and read-dominated remote access pattern, which is a very common remote storage access pattern. *Router-ORAM* is based on the unique idea of server-assisted routing of deliberately misplaced blocks. It utilizes the server's unconsumed computation capability, with homomorphic evaluation, to minimize the access latency and the client's burden without compromising any privacy. We give theoretical proofs for all claimed privacy properties. We demonstrate its practicality, by simulating its long-term behavior.

RouterORAM achieves the minimal possible latency and client work from a theoretical complexity viewpoint, but further research, especially under the lens of implementation efficiency, is still possible. The constant $O(1)$ term in the analysis is dominated by the execution time of the operations of the underlying homomorphic scheme, which might be optimized by exploiting some special properties of different homomorphic encryption schemes. Utilizing trusted execution environments (TEEs) to instantiate *RouterORAM* might be an interesting research direction. TEEs would not only improve performance but also might allow stronger adversaries to be withstood (e.g., an actively malicious server rather than the conventional honest-but-curious one).

Acknowledgments. This project was made possible in-part through the support of the National Cybersecurity Consortium and the Government of Canada (CSIN). The authors would also like to thank the anonymous reviewers for their valuable comments and suggestions, which greatly improved the quality of this paper.

References

1. S. Paul, ORAM-Simulator. (September 16, 2024). Rust. [Online]. Available: https://github.com/sumitkumarpaul/oram
2. O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious RAMs," J. ACM, vol. 43, no. 3, pp. 431–473, May 1996, doi: 10.1145/233551.233553.
3. E. Stefanov et al., "Path ORAM: an extremely simple oblivious RAM protocol," in Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security - CCS '13, Berlin, Germany: ACM Press, 2013, pp. 299–310. doi: 10.1145/2508859.2516660.

4. G. Asharov, I. Komargodski, W.-K. Lin, K. Nayak, E. Peserico, and E. Shi, "OptORAMa: Optimal Oblivious RAM," in Advances in Cryptology – EUROCRYPT 2020, vol. 12106, A. Canteaut and Y. Ishai, Eds., in Lecture Notes in Computer Science, vol. 12106. , Cham: Springer International Publishing, 2020, pp. 403–432. doi: 10.1007/978-3-030-45724-2_14.
5. Ren L., Fletcher C., Kwon A., Stefanov E., Shi E., Van Dijk M., Devadas S. Constants count: Practical improvements to oblivious RAM, in Proceedings of the 24th USENIX Security Symposium, 2015, pp. 415 - 430
6. M. S. Islam, M. Kuzu, and M. Kantarcioglu, "Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation".
7. R. Balasubramonian, "Memory Security," in Innovations in the Memory System, in Synthesis Lectures on Computer Architecture. , Cham: Springer International Publishing, 2019, pp. 81–101. doi: 10.1007/978-3-031-01763-6_11.
8. D. S. Roche, A. Aviv, S. G. Choi, and T. Mayberry, "Deterministic, Stash-Free Write-Only ORAM," in Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, Dallas Texas USA: ACM, Oct. 2017, pp. 507–521. doi: 10.1145/3133956.3134051.
9. S. Tople, Y. Jia, and P. Saxena, "PRO-ORAM: Practical Read-Only Oblivious RAM," 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019), pp. 197–211, 2019.
10. E. Stefanov and E. Shi, "ObliviStore: High Performance Oblivious Cloud Storage," in 2013 IEEE Symposium on Security and Privacy, Berkeley, CA: IEEE, May 2013, pp. 253–267. doi: 10.1109/SP.2013.25.
11. J. Dautrich, E. Stefanov, and E. Shi, "Burst ORAM: Minimizing ORAM Response Times for Bursty Access Patterns," presented at the 23rd USENIX Security Symposium (USENIX Security 14), 2014, pp. 749–764.
12. Y. Chen, K. Srinivasan, G. Goodson, and R. Katz, "Design implications for enterprise storage systems via multi-dimensional trace analysis," in Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, Cascais Portugal: ACM, Oct. 2011, pp. 43–56. doi: 10.1145/2043556.2043562.
13. A. W. Leung, S. Pasupathy, G. Goodson, and E. L. Miller, "Measurement and Analysis of Large-Scale Network File System Workloads," presented at the ATC'08: USENIX 2008 Annual Technical Conference, 2008, pp. 213–226.
14. W. W. Hsu and A. J. Smith, "Characteristics of I/O traffic in personal computer and server workloads," IBM Syst. J., vol. 42, no. 2, pp. 347–372, 2003, doi: 10.1147/sj.422.0347.
15. D. Apon, J. Katz, E. Shi, and A. Thiruvengadam, "Verifiable Oblivious Storage," in Public-Key Cryptography – PKC 2014, vol. 8383, pp. 131–148.
16. S. Devadas, M. Van Dijk, C. W. Fletcher, L. Ren, E. Shi, and D. Wichs, "Onion ORAM: A Constant Bandwidth Blowup Oblivious RAM," in Theory of Cryptography, vol. 9563, E. Kushilevitz and T. Malkin, Eds., in LNCS, vol. 9563, 2016, pp. 145–174.
17. H. Chen, I. Chillotti, and L. Ren, "Onion Ring ORAM: Efficient Constant Bandwidth Oblivious RAM from (Leveled) TFHE," in Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, 2019, pp. 345–360.
18. K. Cong, D. Das, G. Nicolas, and J. Park, "Poster: Panacea — Stateless and Non-Interactive Oblivious RAM," in Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, pp. 3585–3587.
19. TFHE-rs: A Pure Rust Implementation of the TFHE Scheme for Boolean and Integer Arithmetics Over Encrypted Data. (2022). Zama. [Online]: https://github.com/zama-ai/tfhe-rs
20. K. S. Tinani, B. Choithwani, B. Patil, P. Faiyazkhan, and T. Salat, "Study on Usage Pattern of Public Cloud Storage," ijcse, vol. 7, no. 6, pp. 922–927, Jun. 2019, doi: 10.26438/ijcse/v7i6.922927.