ORTOA: One Round Trip Oblivious Access

Sujaya Maiyya[1] Yuval Steinhart[2] Divyakant Agrawal[2] Prabhanjan Ananth[2] Amr El Abbadi[2]
[1]*University of Waterloo,* [2]*University of California Santa Barbara*
[1]*smaiyya@uwaterloo.ca,* [2]*{yuval, divyagrawal, prabhanjan, elabbadi}@ucsb.edu*

Abstract

Use of cloud based storage-as-a-service has surged due to its many advantages such as scalability and pay-as-you-use cost model. However, storing data in the clear on third-party servers creates vulnerabilities, especially pertaining to data privacy. Applications typically encrypt their data before offloading it to cloud storage to ensure data privacy. To serve a client's read or write request, an application either reads or updates the encrypted data on the cloud, revealing the type of client access to the untrusted cloud. An adversary however can exploit this information leak to compromise a user's privacy by tracking read/write access patterns. Existing approaches (used in Oblivious RAM (ORAM) and frequency smoothing datastores) hide the type of client access by always reading the data followed by writing it, *sequentially*, irrespective of a read or write request, rendering one of these rounds redundant with respect to a client request. To mitigate this redundancy, we propose ORTOA- a One Round Trip Oblivious Access protocol that reads or writes data stored on remote storage *in one round without revealing the type of access*. To our knowledge, ORTOA is the first generalized protocol to obfuscate the type of access in a single round, reducing the communication overhead in half. ORTOA hides the type of individual access as well as the read/write workload distribution of an application, and due to its generalized design, it can be integrated with many existing obliviousness techniques that hide access patterns such as ORAM or frequency smoothing. Our experimental evaluations show that for objects of 160B size ORTOA's throughput is **1.4-1.7x** that of a baseline that requires two rounds to hide the type of access; and the baseline incurs **1.5-1.9x** higher latency than ORTOA.

1 Introduction

Due to the high cost of owning and maintaining an on-premise storage fleet, many modern applications outsource their data storage to third party cloud providers such as Amazon AWS or Microsoft Azure. However, outsourcing an application's data in plaintext can reveal sensitive information to a potentially non-trustworthy cloud provider. Many applications protect their data with the standard technique of data encryption.

Encrypted databases such as CryptDB [40] consist of a trusted *front-end* that stores the encryption key and routes all client requests to the untrusted storage. A simple encrypted key-value store design (supporting single object `GET/PUT` requests) serves client requests as follows: for read requests the front-end reads the appropriate encrypted value from the storage, decrypts it, and responds to the client. Whereas for write requests, the front-end encrypts the value updated by the client and writes the encrypted value to the storage.

This common approach of reading and writing encrypted data allows an adversary controlling the cloud to distinguish between read and write requests since only write requests update the database. Revealing the type of access – read vs. write – can violate an end user's or an application's privacy.

At an individual user's level, consider a banking application example where a user either views their balance or updates it upon a purchase. Even with the balance information encrypted, an adversary learns when a user updates their balance. This information combined with location data, which many mobile applications track implicitly, can reveal with a high probability when (and where) a user transacted for goods or services, violating the user's privacy. In fact, a recent attack by John et. al. [32] utilized observing only write accesses to perform a privacy attack.

At an application level, an application is incentivized to hide the type of service it provides because side channel attacks such as [31] exploit such meta-data to reveal sensitive information. However, an application cannot maintain anonymity of its service even with encrypting its data because the read vs. write pattern of an application often reveals the type of service it provids. For example, social network applications tend to be extremely read-heavy [14], whereas IoT applications lean write-heavy [17].

Essentially, revealing the type of access on encrypted data poses privacy challenges both at an individual and an application level. A straightforward approach to address this privacy challenge is to hide the type of operation by always reading an object followed by writing it, irrespective of the type of client request (oblivious datastores that use ORAM [28] or frequency smoothing [30] deploy this two round technique to hide the type of operation).

This sequential two round solution doubles the end-to-end latency for *each* user access compared to plaintext datastores. The trusted front-end, from here on referred to as *proxy*, often communicates with the untrusted storage server over WAN, aggravating the latency problem. For companies such as Amazon and Google, end-to-end latency directly impacts revenue: Amazon loses 1% revenue (worth $3.8 billion!) for every 100*ms* lag in loading its pages [1]; Google's traffic drops by 20% if search results take an additional 500*ms* to load [5].

Furthermore, with increasing privacy laws such as GDPR [3] that prohibit data movement across continents, requiring two rounds of cross-continent communication for each request becomes too expensive. With restricted data movement and due to the high penalty of increased end-to-end latency, we believe that new protocols should trade-off sending larger amounts of data for reduced number of communication rounds.

Rooted in this motto, this work proposes *ORTOA*, a one round trip data access protocol that hides the type of client access to efficiently address the privacy challenges caused by revealing the type of access. ORTOA hides the type of individual client access as well as the read/write distribution of an application.

1.1 Challenges with designing a one round access oblivious protocol

To highlight the challenges of designing a one-round access oblivious protocol, we first present two naive solutions. To hide the type of client operation from an adversary, it is necessary for both read and write requests to be indistinguishable. Hence, both operations need to read *and* write a given physical location.

More specifically, a read request should write back the same value it read, while a write request should write the new value, potentially distinct from the value it read. The two round protocol executes this as follows: (i) fetch the requested data object by performing a read, (ii) decrypt the value, (iii) either encrypt the new value for writes or re-encrypt the fetched value for reads, and (iv) write the freshly encrypted value back to the server. Note that standard encryption schemes such as AES produce different ciphertexts even if the same value is encrypted multiple times; hence, an adversary cannot distinguish between updated value encryptions or same value re-encryptions.

Reducing the two rounds of this protocol to a single round is straightforward for write requests: for each write request, the proxy encrypts the new value and propagates the encrypted value to the server without fetching the object's value first (steps i and ii). But this technique does *not* work for read requests: the proxy cannot re-encrypt an object's value (which is stored only at the server) without fetching the value first. Hence, the proxy needs to perform a read before writing the re-encrypted value, rendering the one round approach moot.

Another naive solution to perform read-followed-by-write in a single round trip is to treat all client requests as read-modify-write transactions. Typically, for read-modify-write transactions, a client interactively reads an object (after acquiring a write-lock), modifies the read value, and writes back the updated value. This can be converted to a non-interactive approach by modifying the storage server to support this type of operation without communicating with the client (proxy, in our case). In this naive solution, the proxy sends an encrypted new value for writes or encrypted dummy value for reads and the server performs read-modify-write by (i) executing a read, (ii) writing the (encrypted) value sent by the proxy, and (iii) responding to the proxy with the read value. But the challenge here is that any subsequent reads after the first read operation will fetch a dummy value, leading to the application permanently losing its data! Making the server's logic more sophisticated such that the read-modify-write operation should re-write the existing value for read requests or update the value for write requests, reveals the type of client query to the server. Hence, a single round solution such as this cannot be used without losing data or compromising privacy.

1.2 Intuitions for ORTOA

We observe that the above discussed challenges exist primarily because of the way data values are stored at the server, i.e, using standard encryption of plaintext values, which disallows the server from updating values obliviously. To mitigate these challenges, ORTOA explores alternate approaches to represent and store data values. In particular, ORTOA, inspired by garbled circuit constructions [34, 45], represents plaintext values in a binary format and encodes each bit with a *secret label* generated using pseudo-random functions (PRFs); only these encoded labels are stored at the server. PRFs are deterministic encoding functions that produce the same output when invoked any number of times with the same input list. If a plaintext value for an object with key k is `01`, then the server stores labels $< l_0, l_1 >$, which are the outputs of $< PRF(k,0), PRF(k,1) >$. Intuitively, ORTOA updates the labels after each access – read or write – to an object because updating the labels only for write requests will reveal the type of operation. The core idea of ORTOA lies in how the proxy and the server communicate to update the labels for both read and write requests *in a single round* (§4).

2 System and Security Model

2.1 System Model

ORTOA is designed for key-value stores where a unique key identifies a given data object, wherein the datastore supports single key `GET` and `PUT` operations. The data is stored on an external server(s) managed by a third party, analogous to renting storage servers from third party cloud providers.

We assume the external server that stores the data to be untrusted. Furthermore, the system uses a proxy model commonly deployed in many privacy preserving data systems [18,30,36,40–42]. The proxy is assumed to be trusted and the clients interact with the external server by routing requests through the proxy. Alternately, the system can also be viewed as a single trusted client interacting with the externally stored data on behalf of users from within the trusted domain. The proxy is a stateful entity and remains highly available; ensuring high availability of the proxy is orthogonal to the protocol presented here. Note that the state stored at the proxy is an order of magnitude smaller (i.e., megabytes) than the state at the external server (i.e., gigabytes).

All communication channels – clients to proxy, proxy to server – are asynchronous, unreliable, and insecure. The adversary can view (encrypted) messages, delay message deliveries, or reorder messages. All communication channels use encryption mechanisms such as transport layer security [7] to mitigate message tampering.

2.2 Data and Storage Model

Each object consists of a unique key and a value, where all values are of equal length – an assumption necessary to avoid any leaks based on the length of the values (equal length can be achieved by padding). Neither an object's key nor its value is stored in the clear at the server. For a given key-value object $< k, v >$, the keys are encoded using pseudorandom functions (PRFs). A PRF's determinism permits a proxy to encode a given key multiple times while resulting in the same encoding; this encoding can then be used to access the value of a given key from the server. We use a procedure *Enc* to encode the values (this procedure differs from §3 to §4). For a key k and its corresponding value v, the server essentially stores $< PRF(k), Enc(v) >$.

2.3 Threat Model

As mentioned earlier, this work focuses on hiding the type of access generated by clients. We assume an honest-but-curious adversary that wants to learn the type of client accesses without deviating from executing the designated protocol correctly. The adversary can control the external server as well as all the communication channels – proxy to external server and clients to proxy. We further assume the adversary can access (encrypted) queries to and from a sender and can inject queries (say by compromising clients), a commonly used adversarial model [19,37,41,42].

Non-goals: ORTOA does not hide the actual physical locations accessed by client requests and hence is vulnerable to attacks based on access patterns, similar to encrypted databases such as CryptDB [40] or Arx [39] (however, ORTOA protects encrypted databases from attacks based on exposing the type of operation). ORTOA does not aim to protect an application from timing based side channel attacks or implementation based backdoor attacks.

3 FHE based solution

After discussing a few non-private or incorrect one round naive solutions in §1, this section presents a one round mechanism to hide the type of accesses using an existing cryptographic primitive, Fully Homomorphic Encryption (FHE) [13,22,25]. This is a theoretically viable but practically infeasible solution. We first provide a high-level overview of FHE before presenting a one round solution.

3.1 Fully Homomorphic Encryption (FHE)

Homomorphic encryption is a form of encryption scheme that allows computing on encrypted data without having to decrypt the data, such that the result of the computation remains encrypted. Homomorphic encryption schemes add a small random term, called *noise*, during the encryption process to guarantee security. A homomorphic encryption function $\mathcal{HE}$ takes a secret-key sk, a message m, and a noise value n as input and produces the ciphertext, ct, as output as shown in Equation 1. The corresponding homomorphic decryption function $\mathcal{HD}$ takes the secret-key and the ciphertext as input to produce message m:

$$ct = \mathcal{HE}(sk, m, n); \qquad m = \mathcal{HD}(sk, ct) \tag{1}$$

An important property of a homomorphic encryption scheme is that the noise must be small; in fact, the decryption function fails if the noise becomes greater than a threshold value, a value that depends on a given FHE scheme.

Homomorphic encryption schemes allow computing over encrypted data. Some homomorphic encryption schemes support addition [11,38] and some other schemes support multiplication [21]. A fully homomorphic encryption (FHE) scheme supports both addition and multiplication on encrypted data [13,22,25]. An FHE scheme, $\mathcal{FHE}$, applied on two messages $m1$ and $m2$ (and two noise values $n1$ and $n2$) can perform the following two operations:

$$\mathcal{FHE}(m1;n1) + \mathcal{FHE}(m2;n2) = \mathcal{FHE}(m1+m2;n1+n2)$$

$$\mathcal{FHE}(m1;n1) * \mathcal{FHE}(m2;n2) = \mathcal{FHE}(m1*m2;n1*n2)$$

For small noise values $n1$ and $n2$, decrypting $\mathcal{FHE}(m1+m2;n1+n2)$ results in the plaintext addition of $m1+m2$, and similarly decrypting $\mathcal{FHE}(m1*m2;n1*n2)$ results in the plaintext multiplication of $m1*m2$. As illustrated above, each homomorphic operation increases the amount of noise included in the encrypted value.

3.2 One-round oblivious read-write using FHE

We propose a technique to use FHE to execute read and write operations in a single round of communication to the external key-value store. Specifically, this section uses an FHE scheme as the encoding procedure *Enc* specified in Section 2.2 to encrypt the values of the key-value store. For a given key-value pair, the server stores $< PRF(k), \mathcal{FHE}(v) >$.

Let v_{old} be the current value of a given data object, which is stored only at the server (after encrypting $\mathcal{FHE}(v_{old})$), and let v_{new} be the updated value of the object, for a write operation (and an 'empty' value for a read). The challenge is to develop an FHE procedure (or computation) Pr with parameters $\mathcal{FHE}(v_{old})$ and $\mathcal{FHE}(v_{new})$ such that:

$$For\ reads: \textsc{Pr}(\mathcal{FHE}(v_{old}) \ , \ \mathcal{FHE}(v_{new})) = \mathcal{FHE}(\boldsymbol{v}_{old})$$
$$For\ writes: \textsc{Pr}(\mathcal{FHE}(v_{old}) \ , \ \mathcal{FHE}(v_{new})) = \mathcal{FHE}(\boldsymbol{v}_{new})$$

The external server can execute the same procedure Pr for both read and write requests but the result of Pr would vary depending on the type of access. If we can design such a procedure, since the server already stores $\mathcal{FHE}(v_{old})$, the proxy only needs to send $\mathcal{FHE}(v_{new})$ in a single round and expect the correct result for either type of operations.

To develop such a procedure, the proxy creates a two-dimensional binary vector $\mathcal{C} = [c_r, c_w]$ where c_r is 1 for read operations (otherwise 0) and c_w is a 1 for write operations (otherwise 0). To see how the vector can be helpful, briefly disregard any data encryption and consider the data in the plain. We construct a procedure Pr$'$:

```
PROCEDURE  Pr'($v_{old}$, $v_{new}$, [$c_r$, $c_w$]):
    RETURN  ($v_{old}$ * $c_r$) + ($v_{new}$ * $c_w$)
```

For reads, when $c_r = 1$ and $c_w = 0$, the result of Pr$'$ is v_{old}; otherwise, for writes when $c_r = 0$ and $c_w = 1$, the result of Pr$'$ is v_{new}. The above procedure gives us the desired functionality, albeit with no encryption. Given that FHE encrypted values can be added and multiplied, Pr$'$ can be refined to procedure Pr to include FHE encrypted inputs:

```
PROCEDURE
Pr($\mathcal{FHE}(v_{old})$, $\mathcal{FHE}(v_{new})$, [$\mathcal{FHE}(c_r)$, $\mathcal{FHE}(c_w)$]):
  RETURN $\mathcal{FHE}(v_{old}) * \mathcal{FHE}(c_r) + \mathcal{FHE}(v_{new}) * \mathcal{FHE}(c_w)$
```

With Procedure Pr that results in the desired outcomes, the next steps elaborate on the specific operations of the proxy and the server:

(1) Upon receiving either a `Read(`k`)` or a `Write(`k, v_{new}`)` request from a client, the proxy creates vector $\mathcal{C}$ such that for reads, $\mathcal{C} = [1,0]$ and for writes, $\mathcal{C} = [0,1]$.

(2) The proxy then sends $\mathcal{FHE}(\mathcal{C})$, i.e. $[\mathcal{FHE}(c_r), \mathcal{FHE}(c_w)]$, along with $\mathcal{FHE}(v_{new})$, where $v_{new} = \perp$ for reads. It also sends $PRF(k)$ so that the server can identify the location to access.

(3) The server, upon receiving the encoded key along with the 3 encrypted entities, reads the value currently stored at key $PRF(k)$. The server then executes Procedure Pr by using the stored value $\mathcal{FHE}(v_{old})$ and the 3 entities sent by the proxy. The server then updates its stored value to the output of the computation and sends the output back to the proxy.

(4) Given that either c_r or c_w is 0, Procedure Pr's output will either be $\mathcal{FHE}(v_{old})$ for reads or $\mathcal{FHE}(v_{new})$ for writes. Given that FHE schemes produce different ciphertexts even if the same value is encrypted multiple times, an adversary cannot distinguish between updated value encryptions or same value re-encryptions. For reads, the proxy decrypts $\mathcal{FHE}(v_{old})$ using FHE's secret-key to retrieve the data object's value. For writes, the proxy ignores the returned value.

Thus, by leveraging the properties of FHEs that allow computing on encrypted data, specifically executing Procedure Pr, we theoretically showed how to read or write data in one round without revealing the type of access.

3.3 Challenges with FHE based solution

Although FHE allows reading or writing data obliviously in one round, this approach is not practically feasible, mainly due to the noise n necessary for homomorphic encryption. As noted above, the noise increases with each homomorphic computation, with the increase being especially drastic for multiplications, which the Procedure Pr requires for both read and write accesses.

To gauge the practicality of the above described FHE based solution, we developed and evaluated a prototype of the solution. The prototype used Microsoft SEAL [6] FHE library with BFV [22] scheme. The evaluation used values of size 160B and 128-bit secret keys, and BFV coefficients set to their default in the SEAL library.

Our experiments revealed that after about 10 accesses to a specific object, the noise value grew too large for the FHE decryption to succeed, essentially rendering this solution impractical for any use in real deployments. The inevitable multiplication in Procedure Pr for both reads and writes is the root cause of this infeasibility. We believe that our proposed FHE solution can be used in the future if better performing FHE schemes are invented that control the amount of noise amplification.

4 ORTOA

Having shown that the use of an existing cryptographic primitive, Fully Homomorphic Encryption (FHE), as-is is impractical to provide the desired one round trip oblivious access approach, we propose a novel protocol, ORTOA, that avoids FHE.

Since the existing encryption scheme, FHE, failed to provide the desired result, we take a step further and define a rather unique way of encoding the data values stored at the external server. We first consider the plaintext value in its binary format. For each binary bit of the plaintext, the server stores a secret label generated by the proxy using pseudorandom functions. This idea of encoding bits using secret labels is inspired by garbled circuit constructions [34,45]. More precisely, if k is a data object's key and v its plaintext value in binary, then the server stores:

$$< PRF(k), (sl_{b_1}^{(1)}, \ldots, sl_{b_j}^{(j)}, \ldots, sl_{b_\ell}^{(\ell)}) >$$

where $\ell = |v|$, $sl_{b_j}^{(j)}$ is a secret label corresponding to the j^{th} index of v from the left (indicated as the superscript) where j goes from 1 to ℓ, and $\forall j$, $b_j \in \{0,1\}$ represents bit value 0 or 1 (indicated as the subscript). For example if $\ell = 3$ and $v = 101$ (in binary notation) , then the server stores $(sl_1^{(1)}, sl_0^{(2)}, sl_1^{(3)})$. The proxy generates secret labels using a pseudorandom function of the form $PRF(k, j, b, ct)$ that takes as input the key k, position index j from left, the corresponding bit value b, and an access counter ct. Because PRFs are deterministic functions, invoking the chosen PRF with the same inputs any number of times will result in the same output label.

The goal of ORTOA is to read and write data in one round-trip, without revealing the type of access. Intuitively, it becomes evident that to hide reads from writes, every access to an object must write the data, which is what ORTOA does at a high level: it updates the secret labels of an object whenever a client accesses the object – be it for a read or a write. We use notation *ol* to represent the *old* secret label currently stored at the server and *nl* to represent the *new* label that would replace the old label. To be able to regenerate the last array of secret labels for a given object, the proxy maintains an access counter indicating the total access count of an object. We note that although maintaining access counters for all objects is $O(N)$, where N is the database size, this requires a small amount of memory (8MB for 1M objects) compared to storing the plaintext values at the proxy (in GBs). Many practical oblivious schemes [18,30,36,41–43] also maintain such $O(N)$ datastructures (e.g., position maps) at the proxy in exchange for higher performance.

4.1 An Illustrative Example

For ease of exposition, we first explain how ORTOA executes reads and writes using a simple example and formally present the protocol in the next section.

Recall that all data values are of the same length, ℓ bits, indexed 1 to ℓ. In this example, let $\ell = 1$, and let k be the specific key accessed by a client where the corresponding plaintext key-value tuple is $< k, 0 >$, i.e., the value associated with k is 0. The server in-turn stores the corresponding encoded tuple $< PRF(k)\ ,\ ol_0^{(1)} >$ where $ol_0^{(1)}$ is a secret label for bit value 0 (indicated as the subscript) at index 1 (indicated as the superscript).

1. Client: The client either sends a `Req(Read, k)` or a `Req(Write, k, v')` request to the proxy, where v' is an updated value for k. In this example, we assume v' is 1.

2. Proxy: The proxy, in response, executes the following steps:

2.1 The proxy generates two ***old*** secret labels $< ol_0^{(1)}, ol_1^{(1)} >$ (where *ol* indicates old label) both for index 1 by calling $PRF(k, 1, b, ct)$ where $b \in \{0, 1\}$ and ct is k's access counter. For each index, the proxy needs to generate labels for both bit values 0 and 1 *since it does not know the actual value, which is stored only at the serve*r.

2.2 The proxy next generates two ***new*** labels $< nl_0^{(1)}, nl_1^{(1)} >$ (where *nl* indicates new label) both for index 1 by calling $PRF(k, 1, b, ct+1)$ where $b \in \{0, 1\}$ and it updates k's access count to $ct + 1$.

2.3 The details of this step depend on the type of access: for reads, the proxy encrypts each new secret label using the corresponding old secret label, thus generating two encryptions for index 1:
$E = [< Enc_{ol_0^{(1)}}(nl_0^{(1)}),\ Enc_{ol_1^{(1)}}(nl_1^{(1)}) >]$
Whereas for writes, assuming the updated value $v' = 1$, the proxy encrypts only the new label corresponding to the updated value $v' = 1$ using the old labels, i.e.:
$E = [< Enc_{ol_0^{(1)}}(\boldsymbol{nl_1^{(1)}}),\ Enc_{ol_1^{(1)}}(\boldsymbol{nl_1^{(1)}}) >]$

2.4 The proxy next shuffles E pairwise, i.e, randomly reorders the two encryptions, to ensure that the first encryption does not always refer to bit 0 and the second to bit 1, and sends E to the external server.

3. Server: The external server, upon receiving E does the following:

3.1 For the pair of encryptions received, the server tries to decrypt both encryptions using its locally stored label. But since it stores only one old label at index 1, it succeeds in decrypting only one of the two encryptions. In this example, the server decrypts $Enc_{ol_0^{(1)}}(nl_0^{(1)})$ for reads or $Enc_{ol_0^{(1)}}(nl_1^{(1)})$ for writes using the stored $ol_0^{(1)}$.

3.2 The server then updates index 1's secret label to the newly decrypted value, in this case, $nl_0^{(1)}$ for reads or $nl_1^{(1)}$ for writes. For writes, since both encryptions for an index encrypt only one new label $nl_1^{(1)}$, either decryptions will result in the desired, updated label that reflects the new value of $< k, 1 >$. Whereas for reads, the server ends up with $nl_0^{(1)}$, reflecting the existing value of $< k, 0 >$. The server sends the output of the decryption to the proxy and since the proxy knows the mapping of secret labels to plaintext bit values, the proxy learns the value of k to be 0 for reads and ignores the output for writes.

4.2 Protocol

This section formally presents the protocol, described in the two functions depicted in Figure 1. Table 1 defines the variables used in explaining ORTOA.

The Init(kv) procedure describes the data initialization process in ORTOA. Upon receiving the plaintext key-value pairs as input, for each pair (line 3), the procedure generates PRF labels at each of the ℓ indexes corresponding to bit b of the value (represented in binary form) (line 7). All the labels appended together represent the value (line 11) and the procedure returns the encoded keys and labels to be stored at the external server.

When a client sends `Req(Read,k)` or a `Req(Write,k,v')` to the proxy, the proxy and the server execute the following steps.

1. Proxy: The proxy, upon receiving a `Req(Read, k)` or a `Req(Write, k, v')` request from a client, where v' is an updated value for k, invokes the ProcessClientRequest procedure as defined in Figure 1, which internally executes the following steps:

1.1 The proxy retrieves key k's access counter ct (line 1).

1.2 For each of the ℓ indexes of the value, the proxy generates the two *old* labels corresponding to both bit-values 0 and 1 by passing the current access counter ct to the PRF (line 5):
$\{ol_0^{(1)} \leftarrow PRF(k, 1, 0, ct),\ ol_1^{(1)} \leftarrow PRF(k, 1, 1, ct),$
$\ldots,$
$ol_0^{(\ell)} \leftarrow PRF(k, \ell, 0, ct),\ ol_1^{(\ell)} \leftarrow PRF(k, \ell, 1, ct)\}$

1.3 For each of the ℓ indexes of the value, the proxy next generates two *new* secret labels corresponding to both bit-values 0 and 1 by passing the updated access counter $ct + 1$ (accounting for the ongoing access) to the PRF (line 6):
$\{nl_0^{(1)} \leftarrow PRF(k, 1, 0, ct+1),\ nl_1^{(1)} \leftarrow PRF(k, 1, 1, ct+1),$
$\ldots,$
$nl_0^{(\ell)} \leftarrow PRF(k, \ell, 0, ct+1),\ nl_1^{(\ell)} \leftarrow PRF(k, \ell, 1, ct+1)\}$

1.4 The details of this step depend on the type of access: for reads, the proxy encrypts each new secret label using the corresponding old secret label and generates two encryptions for each of the ℓ indexes (line 8):

$$E = [< Enc_{ol_0^{(1)}}(nl_0^{(1)}),\ Enc_{ol_1^{(1)}}(nl_1^{(1)}) >, \ldots,$$
$$< Enc_{ol_0^{(\ell)}}(nl_0^{(\ell)}),\ Enc_{ol_1^{(\ell)}}(nl_1^{(\ell)}) >]$$

```
Procedure Init($kv$):
1  $kv' \leftarrow \emptyset$
2  $ct \leftarrow 1$ // indicates an access count of 1
3  for $(k,v) \in kv$ do
4      $labels \leftarrow \emptyset$
5      $i \leftarrow 1$ // starting index
       // $v$ is in binary representation
6      for each bit $b \in v$ starting from left most position do
7          $l \leftarrow PRF(k,i,b,ct)$
8          $labels \overset{\cup}{\leftarrow} l$
9          $i \leftarrow i+1$
10     end
11     $kv' \overset{\cup}{\leftarrow} \{PRF(k), labels\}$
12 end
13 Return $kv'$
```

```
Procedure ProcessClientRequest( $op, k, val$ )
1  Retrieve key $k$'s $ct$ // $k'$s latest access count
2  $E \leftarrow \emptyset$
3  $i \leftarrow 1$ // starting index
   // $val$ is in binary representation
4  for each bit $b \in val$ starting from left most position do
5      $ol_0^{(i)} \leftarrow PRF(k,i,0,ct),\ ol_1^{(i)} \leftarrow PRF(k,i,1,ct)$
6      $nl_0^{(i)} \leftarrow PRF(k,i,0,ct+1)$,
        $nl_1^{(i)} \leftarrow PRF(k,i,1,ct+1)$
7      if $op = read$ then
8          $E \overset{\cup}{\leftarrow} \{Enc_{ol_0^{(i)}}(nl_0^{(i)}),\ Enc_{ol_1^{(i)}}(nl_1^{(i)})\}$
9      else
10         $E \overset{\cup}{\leftarrow} \{Enc_{ol_0^{(i)}}(nl_{b_i}^{(i)}),\ Enc_{ol_1^{(i)}}(nl_{b_i}^{(i)})\}$
11     end
12     $i \leftarrow i+1$
13 end
14 $ct \leftarrow ct+1$
15 Return $E$
```

Figure 1: ORTOA's algorithms to initialize a set plaintext key value pairs kv and process an individual client request for operation type op, key k, and updated value val.

For writes, assuming b_i is the updated bit value at index i, the proxy encrypts only the new labels corresponding to the updated value v' using the old labels (line 10):

$$E = [< Enc_{ol_0^{(1)}}(\boldsymbol{nl}_{b_1}^{(1)}), Enc_{ol_1^{(1)}}(\boldsymbol{nl}_{b_1}^{(1)}) >, \dots,$$
$$< Enc_{ol_0^{(\ell)}}(\boldsymbol{nl}_{b_\ell}^{(\ell)}), Enc_{ol_1^{(\ell)}}(\boldsymbol{nl}_{b_\ell}^{(\ell)}) >]$$

Note that for writes, at each index i, both the old labels encrypt only one new label $nl_{b_i}^{(i)}$ corresponding to v'.

Symbol	Meaning
$ol_{b_j}^{(j)}$	Secret label of a single bit of plaintext value
j	Index from 1 to ℓ starting from the left of plaintext value
b_j	Bit value (0 or 1) at index j of plaintext value
ct	Access counter
$nl_{b_j}^{(j)}$	New secret label of a single bit of plaintext value

Table 1: Variables used in ORTOA.

1.5 The proxy increments k's access counter (line 14) and pairwise shuffles each of the ℓ pairs of encryptions and sends this encryption to the external server.

2. Server: The server upon receiving the encryption E from the proxy performs the following steps:

2.1 For each of the ℓ pairwise encryptions, the server tries to decrypt both encryptions using the locally stored label. However, since it stores only one old label per index, it succeeds in decrypting only one of the two encryptions per index. *Note that ORTOA uses authenticated encryption to ensure the server identifies successful decryptions.*

At index j, the server either stores $ol_0^{(j)}$ or $ol_1^{(j)}$, and hence, it can successfully decrypt only one of $< Enc_{ol_0^{(j)}}(nl_0^{(j)})$, $Enc_{ol_1^{(j)}}(nl_1^{(j)}) >$ obtaining $nl_0^{(j)}$ or $nl_1^{(j)}$ for reads. For writes, since both encryptions encrypt $nl_{b_j}^{(j)}$, either decryptions will result in the new label corresponding to the updated bit b_j at index j.

2.2 The server then updates each index's secret label to the newly decrypted value and sends the output to the proxy. Since the proxy knows the mapping of secret labels to plaintext bit values at each index, the proxy learns the value of k for reads and it ignores the output for writes.

The server always updates its stored secret labels after executing ORTOA to access an object. For reads, the updated labels reflect the *existing value* of the object; for writes, the updated labels reflect the *updated value* of the object. Thus by choosing a unique data representation model and taking advantage of that model, ORTOA provides a one round-trip oblivious access protocol without restricting the number of accesses, unlike the FHE approach.

4.3 Complexity Analysis

4.3.1 Space Analysis

Proxy: The only information the proxy needs to maintain to support ORTOA is the access counter for each key in the database. While the complexity of storing access counters for all the keys is *O(N)*, where N is the database size, the actual space it consumes is quite low. For example if a single counter requires 8 bytes, for a database of size 1 million objects, the proxy requires about 8mB space to store the counters. Note that this space size is much lower compared to storing plaintext objects at the proxy.

Server: While the storage cost at the proxy is insignificant to support ORTOA, the same is not true for the server. The exact space analysis at the server is as follows: if ℓ represents the length of a plaintext value (and all values have same length), r the output size (in bits) of

A few plaintext bit combinations	1-label-per-bit representation
0000	$sl_0^{(1)}, sl_0^{(2)}, sl_0^{(3)}, sl_0^{(4)}$
0001	$sl_0^{(1)}, sl_0^{(2)}, sl_0^{(3)}, sl_1^{(4)}$
0010	$sl_0^{(1)}, sl_0^{(2)}, sl_1^{(3)}, sl_0^{(4)}$
0011	$sl_0^{(1)}, sl_0^{(2)}, sl_1^{(3)}, sl_1^{(4)}$

Table 2: When $\ell = 4$ and each secret label represents one bit of plaintext data, i.e, $y = 1$.

A few plaintext bit combinations	1-label-per-2-bits representation
0000	$sl_{00}^{(1,2)}, sl_{00}^{(3,4)}$
0001	$sl_{00}^{(1,2)}, sl_{01}^{(3,4)}$
0010	$sl_{00}^{(1,2)}, sl_{10}^{(3,4)}$
0011	$sl_{00}^{(1,2)}, sl_{11}^{(3,4)}$

Table 3: When $\ell = 4$ and each secret label represents two bits of plaintext data, i.e, $y = 2$.

the PRF that generates secret labels, and N the database size, then server's storage space in bits can be calculated as:

$$\underbrace{(r \cdot N)}_{Space\ for\ keys} + \underbrace{(r \cdot \ell \cdot N)}_{Space\ for\ values}$$

4.3.2 Communication and Computation Analysis

Every bit of plaintext can have 2 possible values – either a 0 or a 1. Since the data values, or rather the data value encodings, are stored only at the server, the proxy generates both possible secret label encodings, and the corresponding 2 encryptions, for each bit of the plaintext. The proxy then sends 2 encryptions per bit to the server. If ℓ be the length of data values and E_{len} the length of encrypted ciphertexts, for every object accessed by a client, ORTOA incurs the communication cost of:

$$\underbrace{2 \cdot E_{len}}_{Encryptions\ per\ bit} \cdot \underbrace{\ell}_{Number\ of\ bits}$$

In terms of computation, the proxy and the server perform $2 * \ell$ encryptions and decryptions, respectively.

5 Optimizations

5.1 Space optimized solution

In this section, we discuss a technique to optimize storage space by trading off communication cost. Recall that for every bit of plaintext data, the server stores a secret label of r bits; in other words, r bits are used to represent a single bit of plaintext data. To optimize space, the next logical question we ask is: can we use r bits to represent multiple bits of plaintext data?

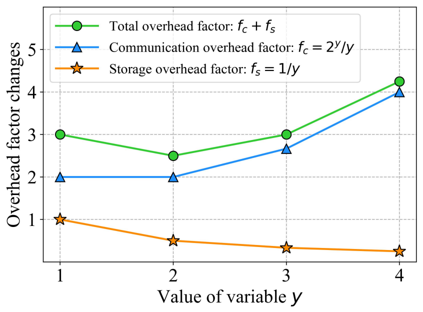


Figure 2: Storage vs. communication overhead factor analysis to find optimal y value - the value that indicates how many plaintext bits are represented by a single label.

One label represents two bits of plaintext: We start with a simple case where a single label represents two bits of plaintext data (Table 3), instead of one (Table 2). In this case, the server stores $\ell/2$ labels for every data item (instead of ℓ), reducing the storage space by half. For example, if the plaintext value is 0010, then the server stores $[sl_{00}^{(1,2)}, sl_{10}^{(3,4)}]$ where, say label $sl_{10}^{(3,4)}$ corresponds to plaintext values 1 and 0 at indexes 3 and 4 respectively.

There are $2^2 = 4$ unique bit combinations for every 2 indexes of the plaintext – 00, 01, 10, and 11. Since the proxy does not know the value, it generates 4 secret labels for every 2-bits, i.e., labels for all possible unique bit combinations, and creates 4 corresponding encryptions for every two bits of plaintext data. The proxy then sends these 4 encryptions per 2-bits to the server, which then tries to decrypt all 4 encryptions. Since the server stores only one label per 2-bits, it succeeds in decrypting only one of the 4 encryptions per 2-bits, which becomes the new label for those 2-bits.

One label represents y bits of plaintext: The above approach can be further generalized where a single label represents y bits of plaintext. For example a label $sl_{b_1 \ldots b_y}^{(1,\ldots,y)}$ corresponds to bits $b_1 \ldots b_y$ from indexes 1 to y. This approach reduces the storage space by a factor of y, i.e., ℓ/y. Note that if the length of values, ℓ, is not divisible by y, we can pad the plaintext with a specific character to indicate the bit value at that index is invalid.

Communication and computation complexity increase: While the space optimized solution reduces the storage space at the server by a factor of y, it incurs increased communication and computation overhead as more labels need to be communicated from the proxy to the server, as analysed next. Recall the communication complexity of the non-space-optimized solution is $(2 \cdot E_{len} \cdot \ell)$. Generalising this to when one secret label represents y bits, there are 2^y possible unique combinations for every y bits of plaintext and the server stores ℓ/y labels. So the communication complexity becomes $(2^y \cdot E_{len} \cdot \ell/y)$ bits and the computation complexity increases to $2^y * \ell/y$, i.e, a factor of $2^y/y$ increase compared to the non-space-optimized solution.

Calculating optimal y value: The above discussion implies that there exists a trade-off between the storage space and the amount of communication (and computation) with the increase in y. When y increases, the storage space reduces by a factor $f_s = 1/y$ and the communication expense increases by a factor $f_c = 2^y/y$, i.e., while the storage space decreases non-linearly, the amount of communication (and computation) increases exponentially.

To calculate the optimal value of y, we compare the overhead factors f_s, f_c, and the total combined overhead of the system, $f_s + f_c$, as depicted in Figure 2. As expected and as seen in the figure, the storage factor reduces with increasing y, and communication factor increases with y. The total overhead plot is interesting: the overall overhead decreases for $y = 2$ and starts increasing from $y = 3$. This is because when $y = 2$, the storage space reduces by half, meanwhile the communication factor remains the same for $y = 1$ and $y = 2$, i.e., $f_c = 2$. For any $y > 2$, the communication factor increases more rapidly than the storage factor reduction, causing the total overhead factor to increase with y. Since the total overhead is the least at $y = 2$, that becomes the optimal y for ORTOA.

5.2 Reducing the number of decryptions

Given that ORTOA has the least overhead for $y = 2$, i,e, a single label representing 2-bits of plaintext, this implies that the proxy sends $2^y = 2^2 = 4$ encryptions for every 2-bits of plaintext. Since the server stores a single label for every 2-bits of plaintext (Table 3), the server can successfully decrypt only one of the 4 encryptions. In the protocol presented in §4, the four encryptions per 2-bits are randomly shuffled by the proxy, and hence, the server attempts to decrypt all encryptions until it succeeds (authenticated encryption schemes used in ORTOA allows identifying successful decryptions). Essentially, the server wastes computation trying to identify the right encryption. To mitigate this inefficiency and reduce the number of potential decryptions on the server from 4 to 1 for every 2-bits of plaintext, ORTOA adapts the point-and-permute [10] optimization.

To reduce the number of decryptions, instead of sending the 4 encryptions per 2-bits in a randomly shuffled manner, the proxy generates the four entries in a deterministic way. For ease of exposition, let us assume that the 4 encryptions are sent as a table where each of the four entries are indexed in binary notation: `00`, `01`, `10`, and `11` indicating the 1st, 2nd, 3rd, and 4th entry of the table.

Intuitively, the proxy generates two additional bits of information *per label* indicating which of the four entries to decrypt upon the next access; we term them ***decryption bits*** d_1d_2. The server stores bits d_1d_2 along with its corresponding secret label. For example, if the server stores a label ($sl_{00}^{(1,2)}$, **10**) for the plaintext indexes (1,2) of an object, the decryption bits 10 indicate that the server should decrypt only the `10`th entry, i.e., the third entry, in the encryption table sent by the proxy for plaintext indexes (1,2). We discuss how the proxy generates the two decryption bits, d_1d_2, next.

To simplify the explanation of the optimization, let us consider $\ell = 2$. The server stores a single label, $ol_{b_1b_2}$, corresponding to two bits of plaintext of an object, and the decryption bits d_1d_2. The main constraint that the proxy needs to guarantee while generating the encryption table when a client accesses the object next is: the encryption entry at index d_1d_2 should use the label $ol_{b_1b_2}$, i.e., $d_1d_2^{th}$ entry in the table is $Enc_{ol_{b_1b_2}}(nl_{b'_1b'_2})$ where $b'_1b'_2$ is b_1b_2 for reads or the updated bits for writes. This constraint is necessary because with this optimization, we are stating that the server decrypts only $d_1d_2^{th}$ entry in the table but the server can only decrypt an encryption that used ol_{b_1b2} (since that is the only label it stores). Essentially, the proxy needs to deterministically 'link' d_1d_2 with b_1b_2 but also randomize this link for every access. The proxy achieves this by leveraging two random bits, r_1r_2, which act as one-time padding bits to link encryption table indexes with labels. Note that the proxy does not store these two bits r_1r_2 explicitly; they can be derived with any PRF (e.g., a PRF $\mathcal{P}$ that takes the access counter ct and key k as input to generate the two bits).

First, let us consider a simplified case where ORTOA supports accessing a data object only once, and hence decryption bits d_1d_2 need not be updated. To access a given object, the proxy generates the four encryption entries for the 2-bits of plaintext by first generating the old and new labels as described in Steps 1.2 and 1.3 of §4.2. Next the proxy creates $d_1d_2^{th}$ entry and links it to the labels by xor-ing with bits r_1r_2: For reads

$$d_1d_2^{th}entry : Enc_{ol_{d_1d_2 \oplus r_1r_2}}(\ nl_{d_1d_2 \oplus r_1r_2}\ )$$

For writes where $nl_{b'_1b'_2}$ represents the label for updated value (essentially all entries encrypt the same new label, refer §4.2)):

$$d_1d_2^{th}entry : Enc_{ol_{d_1d_2 \oplus r_1r_2}}(\ nl_{b'_1b'_2}\ )$$

Generalizing this to where ORTOA supports any number of accesses to an object, the two decryption bits need to be updated after each access. Essentially, at *each* access, we update the decryption bits to $d'_1d'_2$ indicating which entry to decrypt upon the *next* access. The proxy achieves this by generating two new bits r'_1 and r'_2 using the same PRF that generated $r1$ and $r2$ (e.g., invoke PRF $\mathcal{P}$ with updated access counter $ct + 1$ and k). The proxy generates the encryption table with four entries as follows:

For reads:

$$d_1d_2^{th}entry : Enc_{ol_{d_1d_2 \oplus r_1r_2}}(\underbrace{nl_{d_1d_2 \oplus r_1r_2}}_{New\ label}, \underbrace{d_1d_2 \oplus r_1r_2 \oplus r'_1r'_2}_{Bits\ d'_1d'_2})$$

For writes where $nl_{b'_1b'_2}$ represents the new label :

$$d_1d_2^{th}entry : Enc_{ol_{d_1d_2 \oplus r_1r_2}}(\underbrace{nl_{b'_1b'_2}}_{New\ label}, \underbrace{d_1d_2 \oplus r_1r_2 \oplus r'_1r'_2}_{Bits\ d'_1d'_2})$$

The server upon receiving the encryption table decrypts one entry based on the decryption bits d_1d_2. A decryption yields both the new label as well as the updated bits $d'_1d'_2$, which determines what entry to decrypt for the next access. This approach can be generalized to values of any arbitrary length ℓ. Thus by constructing an optimization similar to point-and-permute technique, ORTOA reduces the potential number of decryptions performed by the server from 4 to 1. This reduces the server's computation complexity to $\ell/2$, i.e., one decryption per 2-bits of plaintext.

6 Protocol evaluation

In this section, we discuss the merits and limitations of ORTOA by conducting experimental evaluations.

Baseline: In evaluating ORTOA, we consider a two-round-trip (2RTT) protocol as the baseline: the baseline system also consists of a proxy necessary to maintain the encryption key, which routes client requests to the external server. The baseline proxy translates each request by a client – read or write – into a read request followed by a write request, ensuring read-write indistinguishability. This technique is on par with how most existing obliviousness solutions hide the type of operation [18, 30, 36, 41–43].

Goals: We aim to answer four questions through evaluations:

1. How does ORTOA compare with the 2RTT baseline when the proxy-to-server distance varies? (§6.1)

2. How does ORTOA's performance change with changing configurations such as concurrency or read-write ratio? (§6.2)
3. When and how should an application choose between ORTOA and the 2RTT baseline? (§6.3)
4. How does the two protocols compare for a range of real-world applications? (§6.4)

Experimental Setup: We evaluated ORTOA and its baseline on AWS. The clients and proxy were deployed on a c6i.32xlarge instance with 8GiB memory and 128 cores @ 3.5GHz; and the server on an r5.xlarge instance with 8GiB of memory and 4 cores @ 3.1GHz. The client and proxy were located in the US-West1 (California) datacenter and in most of our experiments, the server was hosted in the US-West2 (Oregon) datacenter. ORTOA's implementation can be found at **REDACTED**.

Unless stated otherwise, in each experiment a multi-threaded client (with a default of 32 threads) sends requests concurrently to the proxy, while each thread sends requests sequentially, i.e., it waits until its current request is answered before sending the next one. Each data point plotted in all the experiments is an average of 3 runs to account for performance variability caused by AWS. In our experiments, the servers for both ORTOA and the baseline store $\sim 2^{20}$ (1M) data objects and unless stated otherwise, all experiments use synthetic data for evaluations. Each client thread picks an object to access uniformly at random, and unless stated otherwise, it decides to read or write the data also uniformly at random. Most of the experiments choose a 160B value size, $\ell = 1280$ bits (this size is in line with other oblivious data systems [19,37] as well as with a range of real-world applications §6.4). Each experiment measures *latency*, the time interval between when a client sends a request to when it receives the corresponding response; and *throughput*, the number of operations executed per second.

Real world datasets: In addition to detailed experiments on synthetic data, we measure ORTOA's performance on three real world datasets: (i) An Electronic Health Record (EHR) data consisting of patients' heart disease records [2], (ii) SmallBank [9] data focusing on single object read/write requests rather than transactional workloads, and (iii) e-Commerce dataset [8] from UCI's machine learning repository consisting of records on customers' online retail purchases. §6.4 discusses more details on the datasets and ORTOA's performance on the datasets.

6.1 ORTOA vs. two round trip baseline

In the first set of experiments, our goal is to measure the effect of proxy-to-server distance on throughput and latency. We compare ORTOA with the 2RTT baseline where the proxy and clients are located in the US-West1 (California) datacenter and the server is placed at increasingly farther datacenters of US-West2 (Oregon), US-East1 (N. Virginia), EU-West2 (London), and AP-South1 (Mumbai). Table 4 notes the round-trip time (RTT) latencies from California to the other datacenters. in the same datacenter as the proxy and the clients so as to mimic realistic behavior where between 79%-95% of cloud users face more than 10 ms latency when accessing a cloud server [16]. Further, this experiment runs 32 concurrent client requests and Figure 3a plots the average latency per client request (i.e., the effect of proxy-to-server distance on individual client requests), along with the system's throughput.

As seen in Figure 3a, as the physical distance between the proxy and the server increases, latency increases and throughput decreases

	Oregon	N. Virginia	London	Mumbai
California	21.84	62.06	147.73	230.3

Table 4: RTT latencies across different datacenters in ms.

	Oregon	N. Virginia	London	Mumbai
Computation (ms)	0.76	0.75	0.8	0.82
Communication (ms)	23.4	64.62	152.31	235.38
Total time (ms)	24.15	65.35	153.11	236.2

Table 5: Time spent in computation (creating old and new labels and encrypting them) vs. time spent in communication, in ms, when the proxy and clients are located in California and the server is located at different datacenters.

for both ORTOA and the 2RTT baseline. But the latency of the 2RTT baseline is at minimum **1.5x** higher than ORTOA at Oregon and at most **1.9x** higher at Mumbai (this difference exists because ORTOA's computation time matters more when the proxy-to-server distance is low and becomes negligible when the proxy-to-server distance increases, as will be reported next). Inversely, ORTOA's throughput is about **1.4-1.7x** that of the baseline. This experiment highlights the benefits of constructing a single round access oblivious protocol, as compared to the state-of-the-art two-round approach.

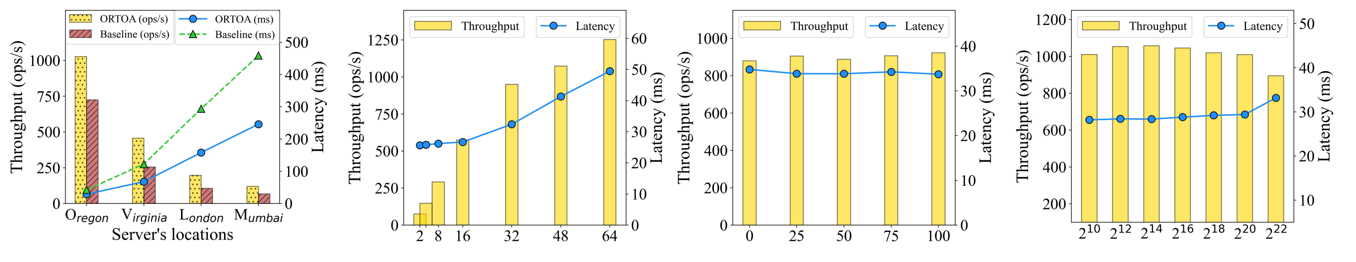
Latency breakdown of ORTOA Since ORTOA's computation cost is high due to generating old and new labels for every 2-bits of plaintext and performing four encryptions for every 2-bits of data, here we report the time the proxy spends in computation vs. in communication. Recall that this experiment consists of 32 concurrent clients generating requests to access data objects of size 160B and the server is placed at increasingly farther distances from California, where the proxy and the clients reside. Table 5 notes the average computation time vs. communication time and the total time, in milliseconds, spent by ORTOA in executing a request. As shown in Table 5, ORTOA consistently spends less than 1 ms in computing the labels and encrypting the data. This latency breakdown indicates that of the total time spent per request, ORTOA spends the majority of the time in communication.

6.2 Micro Benchmarking

This set of experiments evaluate ORTOA's behavior across different configurations, starting with increasing concurrent client requests. These experiments place the server in US-West2 (Oregon) and the proxy and the clients in US-West1 (California) datacenters.

6.2.1 Increasing Concurrency

To understand how ORTOA behaves when clients' request load increases, this experiment measures the protocol's throughput and latency while the number of concurrent clients (implemented via threads) increases starting from 2, and the results are depicted in Figure 3b. As seen in the figure, ORTOA's performance strikes a neat balance at 32 clients with an average latency of ~30 ms and a throughput of ~1000 ops/s. This throughput is about 12.6x of the



(a) ORTOA vs Baseline (b) Varying concurrency (c) Varying % of write requests. (d) Varying the database size.

Figure 3: (a) Throughput and latency for ORTOA and the 2RTT baseline, where the proxy lies in the California datacenter and the server is placed at increasingly farther datacenters. (b) ORTOA's throughput measured with increasing the number of concurrent clients. ORTOA's performance is optimal at 32 clients with latency at ∼30 ms and throughput at ∼1000 ops/s. (c) ORTOA's throughput and latency measured with increasing the percent of PUTs highlights its effectiveness in hiding the read/write ratios of an application. (d) ORTOA's throughput and latency measured while increasing the database size, i.e., number of objects, from 2^{10} to 2^{22} (∼4.2M). The performance degrading is mostly due to a single server storing a large database in memory.

throughput at 2 clients. Although the throughput at 64 clients is 26% higher than at 32 clients, the latency is 54% higher at 64 clients. Since a concurrency of 32 clients has the lowest latency while providing high throughput, the following (and the previous) experiments choose the concurrency of 32 clients, all sending requests in parallel.

6.2.2 Varying the percent of writes

This experiment measures ORTOA's throughput and latency while increasing the percent of PUT (or write) operations from 0 to 100%, as shown in Figure 3c. In this experiment, the server resides in Oregon and 32 concurrent clients read or write the data. As seen in the figure, the throughput and the latency values remain more or less constant at ∼920 ops/s and 33 ms latency (a maximum difference of 40 ops/s for throughput and 2 ms for latency). This experimentally demonstrates the access-oblivious guarantee of ORTOA in that its performance remains the same regardless of the percentage of read or write operations in the client workload. This highlights that ORTOA protects applications from vulnerabilities exploited by observing the overall read/write ratios of an application.

6.2.3 Varying *N*: the database size

This experiment evaluates ORTOA's performance when the overall database size, i.e., the number of objects stored, increases from 2^{10} to 2^{22} (∼4.2 million objects) and the results are depicted in Figure 3d. As shown in the figure, throughput and latency change minimally up until 2^{20} (∼1M objects) and the performance gracefully degrades by 11% at 2^{22} objects. The primary reason for this degradation is due to a single server storing increasingly larger number of objects in memory, which reduces the resources available to execute data access requests and impedes performance. This is an expected behavior of database systems and a standard approach to overcome this performance degradation is to scale the storage.

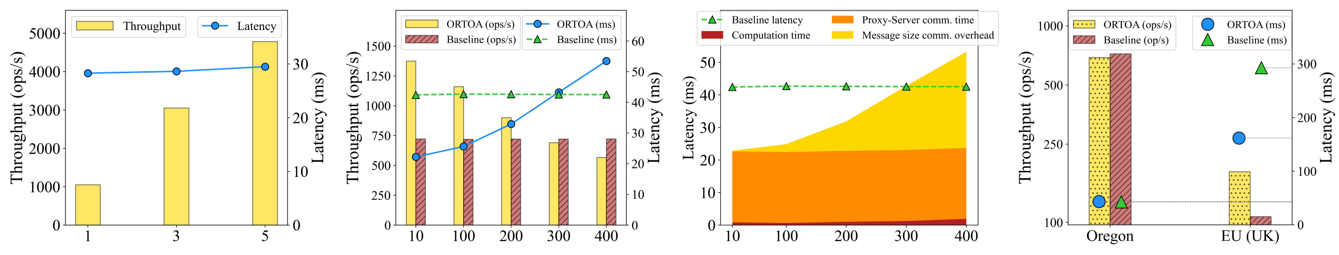
6.2.4 Scaling ORTOA

In this set of experiments, we address the observed performance reduction due to increasing database size by sharding the data across multiple servers and proxies, i.e., by scaling both storage and compute. This experiment increases the number of storage servers and proxies from 1 to 5, by pairing each storage server with a proxy and scaling them pairwise. Since ORTOA aims to hide the type of access performed by a client (and not the overall access pattern), the system can scale the number of proxies without compromising security. For each scaling factor s, the client concurrency is also increased by the scaling factor, i.e., by $32 * s$. This experiment places all the proxies and clients in California and the servers in Oregon and each server stores 1M objects. The resulting throughput and latency are shown in Figure 4a. ORTOA scales near-linearly with the increasing number of servers and proxies: its peak throughput at a scale factor of 5 is about **5x** the throughput at a scale factor of 1. The latency remains constant (a maximum difference of 1.2 ms) across different scale factors. This experiment emphasizes the linear scaling of ORTOA– a highly desired property of data management systems.

6.3 ORTOA vs the 2RTT baseline: Varying ℓ – the length of values

Since the storage, communication, and computation complexity of ORTOA are directly proportional to ℓ (see §4.3), in this experiment, we measure ORTOA's throughput and latency while increasing the size of the values (where all values have the equal length) from 10B to 400B with 32 concurrent clients sending requests and compare the performance with the 2RTT baseline; the results are depicted in Figure 4b. Note that this experiment places the server in Oregon and the proxy and clients in California. Interestingly, this experiment reveals the turning point at which the baseline outperforms ORTOA. As expected, ORTOA's throughput decreases and latency increases as the value size grows. At 300B both the baseline and ORTOA have comparable performance and the baseline starts outperforming ORTOA after that.

Latency breakdown: We speculated the primary reason for ORTOA's performance degradation to be the increased computation at the proxy as it has to generate many more labels, and then encrypt, and decrypt the labels. To validate this hypothesis, we measured latency breakdowns while increasing the value sizes; this breakdown in shown in Figure 4c. Surprising to us, while the computation time does increase for larger values (by 1ms), the primary bottleneck is actually the additional communication time required to transfer larger amounts of data (see the communication overhead analysis in §4.3.2). Figure 4c plots the overall latency of the baseline to contrast with ORTOA's latency, which consists of computation time, the constant communication latency of 21.8ms, and the additional



(a) Increasing the scale factor (b) Varying value size ℓ – in bytes (c) Varying value size ℓ – in bytes (d) Servers in Oregon and EU

Figure 4: (a) ORTOA's throughput and latency measured when the number of servers and proxies in the system are scaled up to a factor of 5. Throughput scales near-linearly with the scale factor, highlighting the scalability of ORTOA. (b) Throughput and latency measured for ORTOA and the baseline while increasing the size of data values from 10B to 400B. Due to the large-message communication overhead of ORTOA, the baseline outperforms ORTOA starting at 300B. (c) Latency breakdown of ORTOA: computing time spent generating labels and encryptions, communication latency between the proxy (US-Ca) and server (US-Or), and additional communication overhead due to exchanging larger messages for higher value sizes. (d) Throughput (in log scale) and latency comparison between ORTOA and the baseline when the server is placed in Oregon vs. EU for values sized 300B.

communication overhead time. We see after 300B ORTOA's overall latency becomes greater than the baseline's latency. However, we cannot blindly claim that for objects greater than 300B, the 2RTT baseline is always a better choice because where the server is located with regard to the proxy also plays a vital role in this.

How to choose between ORTOA and the 2RTT baseline? To help an application choose between ORTOA and the baseline, we provide the following equation: Let c be the cross-datacenter communication time between the server and the proxy, let p be ORTOA's processing or computation time, and let o be ORTOA's communication overhead time due to exchanging large messages. ORTOA is a better choice for an application if:

$$c > p + o$$

If communicating with the server one extra round is worse than the combined processing time and additional large-message overhead delays, then ORTOA will yield better performance than the 2RTT baseline; and vice versa. To highlight this point, we conduct an experiment with objects of 300B by placing the server in EU, as an example to show the impact on latency and performance when an application complies with GDPR, which disallows moving data outside of EU. The results are shown in Figure 4d.

As seen in the figure, when the server is placed in Europe, $c = 147.7ms$ and for ORTOA, $p + o = 13.7ms$ and ORTOA's throughput is **1.7x** that of the baseline. This underscores our hypothesis that having fewer rounds of communication at the cost of increased message sizes is worthwhile when the communication latency between the proxy and server is large compared to the processing and communication overhead of ORTOA. Even with low proxy-to-server communication latency, ORTOA can be a better choice for performance than the 2RTT baseline for small object sizes, as discussed in 6.1. Whereas with low proxy-to-server communication latency but large value sizes, the 2RTT solution performs better than ORTOA.

6.4 Real world datasets

To assess ORTOA's behavior for real world applications, this experiment measures and compares ORTOA's performance with the baseline for three practical applications with strict privacy needs: health care, banking, and e-commerce. For each application, we initialize the database with real world datasets: (i) An Electronic Health Record (EHR) dataset consisting of heart disease information [2] with 14 attributes. For this dataset, we chose two attributes: a UUID to identify unique patients and their resting blood pressure data. The size of resting blood pressure attribute is 10B (80 bits). Because the original dataset consists of only 1024 (2^{10}) entries, we repeat this dataset to create a database of size 2^{20} (1M) objects. (ii) A SmallBank [9]-like dataset for banking applications where, although SmallBank [9] supports transactional queries, this experiment focuses on single object read/write requests from clients, which aligns with the type of requests supported by ORTOA. This dataset also consists of 1M entries with a UUID attribute to identify bank customers and a 50B (400 bits) combined balance attributes consisting of checking balance, savings balance, and account numbers. (iii) An e-commerce dataset [8] from UCI's machine learning repository with 8 attributes. For the experiment we pick 3 attributes, invoiceId as object keys and concatenated customerId (with 5 character limit) and productDescription (with 35 character limit) attributes as values. Hence, in total, the plaintext values for this dataset amounts to 40B (320 bits). While the original dataset consists of 541,909 entries, we re-use the dataset to build a database with 1M entries.

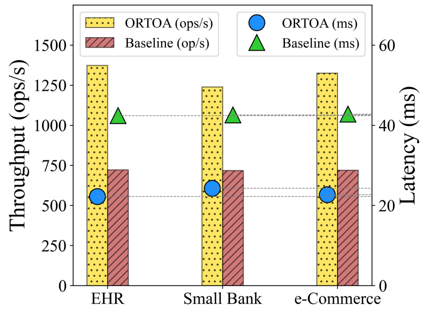


Figure 5: Throughput and latency comparison between ORTOA and the baseline for three practical applications based on real world datasets - Electronic Health Records (EHR), SmallBank data, and e-commerce data.

This experiment measures the latency and throughput of ORTOA on real world datasets and contrasts the performance with the 2RTT baseline with 32 concurrent client threads generating the read/write workload. As depicted in Figure 5, ORTOA's throughput is **1.9x** of the baseline for EHR, **1.7x** for SmallBank, and **1.8x** for e-commerce (varying value sizes, i.e., 10B, 50B, and 40B respectively, causes this difference in performance). Conversely, the baseline's latency is **1.7-1.9x** higher than that of ORTOA. This experiment indicates that for a variety of popular applications that have strong privacy requirements, ORTOA outperforms the 2RTT baseline.

6.5 Dollar cost analysis

We have shown the benefits of a single round access oblivious protocol through the above discussed experimental evaluations. Since ORTOA incurs high storage and communication overheads, in this section, we discuss the estimated dollar cost of deploying ORTOA. To calculate the estimates, we consider the storage, communication, and compute costs of Google Cloud [4, 29], whose costs are comparable to other cloud providers. Google Cloud charges \$0.02 per GB of storage per month, \$0.12 per GB of network usage, and \$0.4 per million function invocations with a 1.4 GHz CPU costing \$0.00000165 per 100ms (ORTOA needs 2 ms to encrypt/decrypt labels). In estimating the dollar cost, we consider the optimized protocol with $y = 2$, and PRFs that produce 128-bit labels, i.e., $r = 128$, with data values of size 160B, i.e., $\ell = 1280$, and with encryption schemes that produce 128-bit ciphertexts, i.e., $E_{len} = 128$. Please refer to §4.3 to recall the storage, communication, and compute complexity of ORTOA. With the above configuration, consider running ORTOA with a large dataset consisting of 1 million data objects. This costs an application **\$1.52** in storage per month, and executing 1 million accesses will cost **\$18.3** in terms of bandwidth and **\$3.7** in terms of compute (function calls). Taking into account the cost of a single access, ORTOA incurs a cost of **\$0.000023** per request – a reasonable price considering the advantage over the 2RTT baseline, which incurs up to 1.9x higher latency overhead and serves up to 1.7x less requests compared to ORTOA.

7 Security of ORTOA

This section defines the security guarantees of ORTOA and provides intuitions of the proof; the Appendix presents the formal security proof. ORTOA aims to hide the type of client access – read or write – from an adversary that controls the external database server. The security definition closest to capturing this indistinguishability lies in ORAM [27]; however ORAM's security definition focuses primarily on *access pattern* indistinguishability and hence cannot to employed to capture the desired goals of ORTOA. Therefore, we introduce a new security definition to express the desired read or write obliviousness called real-vs-random read-write indistinguishability or ROR-RW indistinguishability. We note that the new definition is the best possible definition for settings that hide the type of access without hiding the location of the accessed object.

Security definition: Consider a sequence of m client accesses

$$A = \{(op_1, k_1, val_1), \cdots, (op_i, k_i, val_i), \cdots, (op_m, k_m, val_m)\}$$

where for i^{th} request, op_i indicates the type of operation (read or write), k_i denotes the key, and val_i is either an updated value for writes or $\perp$ for reads. We use a security game-based definition that provides the sequence of accesses A as input to both the real system and an ideal system (simulator based), where both are stateful entities,

```
Real(A)
1 $output \leftarrow \emptyset$
2 for $a_i \in A$ do
3     $output \overset{\cup}{\leftarrow} Process-ClientRequest(a_i)$
4 end
5 Return $output$
```

```
Ideal(K)
1 $output \leftarrow \emptyset$
2 for $k_i \in K$ do
3     $output \overset{\cup}{\leftarrow} Simulator(k_i)$
4 end
5 Return $output$
```

Figure 6: Security game where given a sequence of client generated accesses A, the Real world takes A as input and the Ideal world takes the sequence of keys accessed in A as input and both produce as output a sequence of encryptions that are sent to the external server.

```
Procedure Simulator( k )
1 $E \leftarrow \emptyset$
  // Iterate over each of the $\ell$ indexes
2 for ( $i = 0; i < \ell; i++$ ) do
3     Retrieve the old label $ol^{(i)}$ for $k$
4     $nl^{(i)} \overset{\$}{\leftarrow} \{0,1\}^{\lambda}$
5     $ol'^{(i)} \overset{\$}{\leftarrow} \{0,1\}^{\lambda}$
6     $E \overset{\cup}{\leftarrow} \{Enc_{ol^{(i)}}(nl^{(i)}),\ Enc_{ol'^{(i)}}(0)\}$
7     $ol^{(i)} \leftarrow nl^{(i)}$
8 end
9 Return $E$
```

Figure 7: Simulator pseudocode accessed in the Ideal algorithm.

and both produce outputs Out_{Real} and Out_{Sim} respectively consisting of a sequence of accesses to the external server. A system is said to be ROR-RW secure if, given the two outputs, an adversary can distinguish between the two with negligible probability, i.e., *For all probabilistic polynomial adversaries $\mathcal{A}$*,

$$|\, Pr[A(Out_{Real}) \rightarrow 1] - Pr[A(Out_{Sim}) \rightarrow 1] \,| \leq negl$$

To argue for ORTOA's correctness, we consider a game $\mathcal{G}$, as shown in Figure 6. We assume the length ℓ of data values to be 1 for simplicity but the same argument holds for data values of any arbitrary length. The game either executes Real or Ideal algorithm with uniformly random probability and provides the output to an adversary. ORTOA is ROR-RW secure if the adversary, based on the received output, can identify the algorithm selected by the security game with negligible probability.

The Real algorithm invokes ORTOA's *ProcessClientRequest* procedure (defined in Figure 1) for each of the m accesses in A and appends the output of each access to produce Out_{Real}. The Ideal algorithm, on the other hand, invokes a simulated function, *Simulator*, defined in Figure 7. The Ideal algorithm (and its Simulator) has no access to the type of requests op_i or the data values in $\mathcal{A}$; it generates m pairs of encryptions of *dummy* values. The collation of these dummy encryptions forms Out_{Sim}. If we can prove that the output generated by the Real algorithm appears indistinguishable to Out_{Sim}, it proves that ORTOA is `ROR-RW` secure.

Proof intuition: Intuitively, we first show that a read and a write access to *ProcessClientRequest* procedure are indistinguishable, and then show that *ProcessClientRequest*'s output is indistinguishable from that of the Simulator. Figure 8 captures the argument for this indistinguishability. The basis of our argument lies in the PRF deployed in ORTOA: ORTOA's PRF, *PRF*, produces labels that are indistinguishable from a uniformly sampled random variable $r \overset{\$}{\leftarrow} \{0,1\}^{\lambda}$. The argument in Figure 8 invokes *ProcessClientRequest* procedure once to read an object k and once to update k with bit value b' (assuming the length of the value $\ell = 1$). As shown in the figure, given that the server stores only one old label, say ol_b, and given *PRF*'s security, the output produced by both invocations of *ProcessClientRequest* are identical.

When the Real algorithm invokes *ProcessClientRequest* m times (for m accesses in A), the output of the Real algorithm based on the argument shown in Figure 8 becomes indistinguishable from that of Out_{Sim}, which is essentially m pairs of encryptions of λ length random values. We utilize this intuition in developing the formal security proof using hybrids (refer Appendix).

8 Discussion on related work

To the best of our knowledge, ORTOA is the only solution that tackles the problem of hiding the type of operation in a generalized manner. The literature on ORAM constructions consists of a few specialized solutions that achieve single round communication complexity [12, 24, 26, 27, 35]. All these solutions are based on Garbled-RAM which requires the server to store and evaluate a garbled circuit *per request*. Garbled-RAMs do not take fixed length inputs and their execution time varies based on the input size as well as value size. All these properties primarily differ from ORTOA's, which has a simple server model, fixed length inputs, and constant execution time. These solutions primarily focus on hiding the data access patterns, with mechanisms to hide the type of access tightly coupled with hiding access pattern. ORTOA on the other hand focuses on hiding the type of access in a more *generalized* way that can be adapted to construct obliviousness solutions such as ORAM or frequency smoothing [30]. On the other hand, although a few ORAM based datastores that do not use Garbled-RAM such as [20, 23, 44] have single *online* rounds, they need *offline* rounds per request to write the data back. Hence, they are not truly single-round solutions. Due to hiding access patterns, all of the above ORAM schemes have a lower bound bandwidth cost of *log(N)*, where N is the number of data objects [28, 33] or $\sqrt{N}$ lower bound when the data storage server performs no computations [15]. Since ORTOA focuses on obfuscating the type of access, it has a constant bandwidth cost *independent* of N (as discussed in §4.3).

9 Conclusion and Future Work

Encrypted databases leak information on when a client performs a read vs. a write operation to an adversary; by observing individual read/write accesses, the adversary can learn the overall read/write workload of an application. An adversary can exploit this information leak to violate privacy at an individual user level or at an application level. Existing solutions to hide the type of operation (deployed in ORAM or frequency smoothing techniques) consist

```
Read: $(read, k, \perp)$
1 $\{Enc_{ol_b}(nl_{b'}),\ Enc_{ol_{1-b}}(nl_{1-b'})\} \leftarrow$
    $ProcessClientRequest(read, k, \perp)$
  // Because the server has only $ol_b$, it cannot
     decrypt $Enc_{ol_{1-b}}(nl_{1-b'})$. So it can be replaced
     with a random string.
2 $\equiv \{Enc_{ol_b}(nl_{b'}),\ Enc_{ol_{1-b}}(\{0,1\}^{\lambda})\}$
  // From $PRF$'s security, the new label can be
     replaced with a random string of length $\lambda$.
3 $\equiv \{Enc_{ol_b}(\{0,1\}^{\lambda}),\ Enc_{ol_{1-b}}(\{0,1\}^{\lambda})\}$ // From
     $PRF$'s security, the old labels can be
     replaced with random strings of length $\lambda$.
4 $\equiv \{Enc_{\{0,1\}^{\lambda}}(\{0,1\}^{\lambda}),\ Enc_{\{0,1\}^{\lambda}}(\{0,1\}^{\lambda})\}$
```

```
Write: $(write, k, b')$
1 $\{Enc_{ol_b}(nl_{b'}),\ Enc_{ol_{1-b}}(nl_{b'})\} \leftarrow$
    $ProcessClientRequest(write, k, b')$
  // Because the server has only $ol_b$, it cannot
     decrypt $Enc_{ol_{1-b}}(nl_{b'})$. So it can be replaced
     with a random string.
2 $\equiv \{Enc_{ol_b}(nl_{b'}),\ Enc_{ol_{1-b}}(\{0,1\}^{\lambda})\}$
  // From $PRF$'s security, the label can be
     replaced with a random string of length $\lambda$.
3 $\equiv \{Enc_{ol_b}(\{0,1\}^{\lambda}),\ Enc_{ol_{1-b}}(\{0,1\}^{\lambda})\}$
  // From $PRF$'s security, the old labels can be
     replaced with random strings of length $\lambda$.
4 $\equiv \{Enc_{\{0,1\}^{\lambda}}(\{0,1\}^{\lambda}),\ Enc_{\{0,1\}^{\lambda}}(\{0,1\}^{\lambda})\}$
```

Figure 8: Intuition for read-write indistinguishability when a key k is accessed where the server stores label ol_b corresponding to k's plaintext value $b \in \{1,0\}$. The write request updates k's value to bit b'. The PRF deployed in ORTOA generates labels of length λ.

of always reading an object followed by writing it, irrespective of the client request. This incurs one round of redundant communication *per request* and doubles the end-to-end latency compared to plaintext datastores. In this work, we propose ORTOA, a One Round Trip Oblivious Access protocol that accesses data on remote storage and hides the type of access in a single round. This is the first protocol to focus on hiding access type on encrypted databases. Experimentally evaluating ORTOA and comparing it with a baseline that requires two rounds to hide the type of access confirms the benefits of designing a single round solution: the baseline incurred **1.5-1.9x** higher latency and serves **1.4-1.8x** less requests per second than ORTOA for objects of size 160B. This work also presents a theoretically sound one round trip oblivious access solution using Fully Homomorphic Encryption and discusses its improbability of practical use due to the expensive multiplication operation. As future work, we aim to integrate ORTOA into an end-to-end system that hides access pattern by integrating it with existing techniques such as frequency smoothing or by designing novel ORAM schemes that leverage ORTOA to access data in a single round.

References

[1] Amazon loses 1% revenue for every 100ms page load delay. https://www.contentkingapp.com/academy/page-speed-resources/faq/amazon-page-speed-study/. Accessed May 9, 2022.

[2] EHR dataset of heart diseases. https://www.kaggle.com/datasets/johnsmith88/heart-disease-dataset. Accessed October 14, 2022.

[3] GDPR. https://gdpr-info.eu/. Accessed May 9, 2022.

[4] Google function pricing. https://cloud.google.com/functions/pricing. Accessed August 15, 2021.

[5] Google loses 20% traffic for 0.5s page load delay. https://medium.com/@vikigreen/impact-of-slow-page-load-time-on-website-performance-40d5c9ce568a. Accessed May 9, 2022.

[6] Microsoft seal. https://docs.microsoft.com/en-us/azure/architecture/solution-ideas/articles/homomorphic-encryption-seal. Accessed June 15, 2021.

[7] TLS. https://datatracker.ietf.org/doc/html/rfc5246. Accessed April 14, 2022.

[8] Uci online retail data set. https://archive.ics.uci.edu/ml/datasets/online+retail. Accessed May 9, 2022.

[9] Alomari, M., Cahill, M., Fekete, A., and Rohm, U. The cost of serializability on platforms that use snapshot isolation. In *2008 IEEE 24th International Conference on Data Engineering* (2008), IEEE, pp. 576–585.

[10] Beaver, D., Micali, S., and Rogaway, P. The round complexity of secure protocols. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing* (1990), pp. 503–513.

[11] Benaloh, J. Dense probabilistic encryption. In *Proceedings of the workshop on selected areas of cryptography* (1994), pp. 120–128.

[12] Boneh, D., Mazieres, D., and Popa, R. A. Remote oblivious storage: Making oblivious ram practical. *http://dspace.mit.edu/handle/1721.1/62006.TechnicalReport* (2011).

[13] Brakerski, Z. Fully homomorphic encryption without modulus switching from classical gapsvp. In *Annual Cryptology Conference* (2012), Springer, pp. 868–886.

[14] Bronson, N., Amsden, Z., Cabrera, G., Chakka, P., Dimov, P., Ding, H., Ferris, J., Giardullo, A., Kulkarni, S., Li, H., et al. Tao:facebook's distributed data store for the social graph. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)* (2013), pp. 49–60.

[15] Cash, D., Drucker, A., and Hoover, A. A lower bound for one-round oblivious ram. In *Theory of Cryptography Conference* (2020), Springer, pp. 457–485.

[16] Charyyev, B., Arslan, E., and Gunes, M. H. Latency comparison of cloud datacenters and edge servers. In *GLOBECOM 2020-2020 IEEE Global Communications Conference* (2020), IEEE, pp. 1–6.

[17] Copie, A., Fortiş, T.-F., and Munteanu, V. I. Benchmarking cloud databases for the requirements of the internet of things. In *Proceedings of the ITI 2013 35th International Conference on Information Technology Interfaces* (2013), IEEE, pp. 77–82.

[18] 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.

[19] 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.

[20] Dautrich, J., Stefanov, E., and Shi, E. Burst {ORAM}: Minimizing {ORAM} response times for bursty access patterns. In *23rd USENIX Security Symposium (USENIX Security 14)* (2014), pp. 749–764.

[21] ElGamal, T. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory 31*, 4 (1985), 469–472.

[22] Fan, J., and Vercauteren, F. Somewhat practical fully homomorphic encryption. *IACR Cryptol. ePrint Arch. 2012* (2012), 144.

[23] Fletcher, C., Naveed, M., Ren, L., Shi, E., and Stefanov, E. Bucket oram: single online roundtrip, constant bandwidth oblivious ram. *Cryptology ePrint Archive* (2015).

[24] Garg, S., Lu, S., Ostrovsky, R., and Scafuro, A. Garbled ram from one-way functions. In *Proceedings of the forty-seventh annual ACM symposium on Theory of computing* (2015), pp. 449–458.

[25] Gentry, C., et al. *A fully homomorphic encryption scheme*, vol. 20. Stanford university Stanford, 2009.

[26] Gentry, C., Halevi, S., Lu, S., Ostrovsky, R., Raykova, M., and Wichs, D. Garbled ram revisited. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques* (2014), Springer, pp. 405–422.

[27] Goldreich, O. Towards a theory of software protection and simulation by oblivious rams. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing* (1987), pp. 182–194.

[28] Goldreich, O., and Ostrovsky, R. Software protection and simulation on oblivious rams. *Journal of the ACM (JACM) 43*, 3 (1996), 431–473.

[29] Google. Google cloud pricing. https://cloud.google.com/storage/pricing. Accessed August 15, 2021.

[30] Grubbs, P., Khandelwal, A., Lacharité, M.-S., Brown, L., Li, L., Agarwal, R., and Ristenpart, T. Pancake: Frequency smoothing for encrypted data stores. In *29th USENIX Security Symposium (USENIX Security 20)* (2020), pp. 2451–2468.

[31] ISLAM, M. S., KUZU, M., AND KANTARCIOGLU, M. Access pattern disclosure on searchable encryption: ramification, attack and mitigation. In *Ndss* (2012), vol. 20, p. 12.

[32] JOHN, T. M., HAIDER, S. K., OMAR, H., AND VAN DIJK, M. Connecting the dots: Privacy leakage via write-access patterns to the main memory. *IEEE Transactions on Dependable and Secure Computing 17*, 2 (2017), 436–442.

[33] LARSEN, K. G., AND NIELSEN, J. B. Yes, there is an oblivious ram lower bound! In *Annual International Cryptology Conference* (2018), Springer, pp. 523–542.

[34] LINDELL, Y., AND PINKAS, B. A proof of security of yao's protocol for two-party computation. *Journal of cryptology 22*, 2 (2009), 161–188.

[35] LU, S., AND OSTROVSKY, R. How to garble ram programs? In *Annual International Conference on the Theory and Applications of Cryptographic Techniques* (2013), Springer, pp. 719–734.

[36] 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. In *31st USENIX Security Symposium (USENIX Security 22)* (2022), pp. 3665–3682.

[37] MISHRA, P., PODDAR, R., CHEN, J., CHIESA, A., AND POPA, R. A. Oblix: An efficient oblivious search index. In *2018 IEEE Symposium on Security and Privacy (SP)* (2018), IEEE, pp. 279–296.

[38] PAILLIER, P. Public-key cryptosystems based on composite degree residuosity classes. In *International conference on the theory and applications of cryptographic techniques* (1999), Springer, pp. 223–238.

[39] PODDAR, R., BOELTER, T., AND POPA, R. A. Arx: A strongly encrypted database system. *IACR Cryptol. ePrint Arch. 2016* (2016), 591.

[40] POPA, R. A., REDFIELD, C. M., ZELDOVICH, N., AND BALAKRISHNAN, H. Cryptdb: Protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), pp. 85–100.

[41] 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.

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

[43] 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), pp. 299–310.

[44] WILLIAMS, P., AND SION, R. Single round access privacy on outsourced storage. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), pp. 293–304.

[45] YAO, A. C.-C. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science* (1986), IEEE, pp. 162–167.

Appendix

Because no existing security definitions capture ORTOA's goal of hiding the type of access (without focusing on hiding the location of accessed location, we propose a new security definition called real-vs-random read-write indistinguishability or `ROR-RW` to capture the goal of this work of hiding the type of access performed by a client.

Security definition: Consider a sequence of m client accesses

$$A = \{(op_1, k_1, val_1), \cdots, (op_i, k_i, val_i), \cdots, (op_m, k_m, val_m)\}$$

where for i^{th} request, op_i indicates the type of operation (read or write), k_i denotes the key, and val_i is either an updated value for writes or $\perp$ for reads. This is a security definition based on a game $\mathcal{G}$ defined in Figure 6. The game takes the sequence of accesses A and provides it as input to both the real system and an ideal system (simulator based), where both are stateful entities, and both produce outputs Out_{Real} and Out_{Sim} respectively consisting of a sequence of accesses to the external server. A system is said to be `ROR-RW` secure if, given the two outputs, an adversary can distinguish between the two with negligible probability, i.e.,
For all probabilistic polynomial adversaries $\mathcal{A}$,

$$| \, Pr[A(Out_{Real}) \to 1] - Pr[A(Out_{Sim}) \to 1] \, | \leq negl$$

For simplicity in arguing for ORTOA's security, the proof assumes $\ell = 1$; however, the same proof argument extends to values of arbitrary length. Further, our proof considers the non-optimized protocol as presented in §4.2 but the proof easily extends to the optimized versions as well.

For Real algorithm in Figure 6, the game sends a sequence of m accesses in A produced by clients where the algorithm in-turn calls ORTOA's *ProcessClientRequest* procedure (defined in Figure 1) for each access in A. Note that the *ProcessClientRequest* procedure is a stateful algorithm. Let λ be the length of old and new labels generated by a PRF and let *Enc* be the encryption scheme deployed in the *ProcessClientRequest* procedure to encrypt new labels of length λ using old labels of length λ. Since we assume $\ell = 1$, *ProcessClientRequest* produces two encryptions for each access to send to the server. The Real algorithm collates the output of *ProcessClientRequest* method, consisting of a pair of encryptions for each of the m accesses; this collation of encryptions is the Real algorithm's output, represented as:

$$Out_{Real} \leftarrow \{Enc_{ol_b}(nl_{b'}), Enc_{ol_{1-b}}(nl_{b''})\}^m$$

where for each read access ($b' = b$) and ($b'' = 1 - b$), and for write accesses ($b' = b'' = \hat{b}$), the updated bit.

For the Ideal algorithm in Figure 6, the game provides the sequence of keys accessed in A as input where the algorithm in-turn calls a Simulator defined in Figure 7. The Simulator's goal is to produce encryptions similar to the *ProcessClientRequest* procedure but with arbitrary values; one can notice the analogies between the two procedures. To achieve this, we assume the Simulator to be stateful and it stores one old label ol per index i of a key k's value

– these are the labels stored at the external server. The procedure takes key k as input and iterates over each of the ℓ indexes (where ℓ is the value's plaintext length). At each index, the Simulator retrieves the corresponding old label; it then generates two randomly sampled labels $nl^{(i)}$ and $ol'^{(i)}$ of length λ (same as the PRF used in *ProcessClientRequest*). It uses $ol^{(i)}$ to encrypt $nl^{(i)}$ and $ol'^{(i)}$ to encrypt an invalid value, 0. This does not reveal any information to the adversary that controls the external server because the server only stores label $ol^{(i)}$ and can decrypt only one of the two encryptions sent by the Simulator. The Simulator shuffles the two encryptions at each index and appends it a list E to send to the server. It also updates the old labels $ol^{(i)}$ with the newly and randomly generated label $nl^{(i)}$. Because the Simulator encrypts random values of length λ, the Ideal algorithm's output is, assuming $\ell = 1$:

$$Out_{Sim} \leftarrow \{Enc_{\{0,1\}^\lambda}\{0,1\}^\lambda, Enc_{\{0,1\}^\lambda}\{0,1\}^\lambda\}^m$$

Formal proof: We now formally prove that the real and the ideal worlds are computationally indistinguishable using a standard hybrid argument.

Hybrid_1: This corresponds to the real experiment and the output of this hybrid is Out_{Real}.

Hybrid_2: We modify the real experiment where the labels generated using *PRF* in the *ProcessClientRequest* procedure are now sampled from the uniform distribution.

The computational indistinguishability of Hybrid_1 and Hybrid_2 follows from the security of PRF.

$\mathsf{Hybrid}_{3.i}$ for $i \in [m]$: In the sequence of m accesses in A, consider the i^{th} access, in which the *ProcessClientRequest* procedure generates $2 * \ell = 2 * 1 = 2$ encryptions ($\ell = 1$). Since the server stores only one label per index and can only decrypt one of the two encryptions, the other encryption sent has no significance: let the two ciphertexts be CT_0 and CT_1 where both the ciphertexts are encrypted with respect to two different old labels ol_0 and ol_1. Note that the server has exactly one label ol_b for some bit b. Replace the message in CT_{1-b} with 0s - this encryption becomes *insignificant* since the server cannot decrypt it. This hybrid replaces encryptions of all such insignificant entries with the encryptions of an invalid value, say 0.

The computational indistinguishability of $\mathsf{Hybrid}_{3.i}$ and $\mathsf{Hybrid}_{3.i-1}$ follows from the security of encryption.

Hybrid_4: This corresponds to the ideal experiment, i.e., Out_{Real} is equivalent to Out_{Sim}.

The hybrids Hybrid_4 and $\mathsf{Hybrid}_{3.m}$ are identically distributed. The transition from $\mathsf{Hybrid}_{3.m}$ to Hybrid_4 is as follows: in $\mathsf{Hybrid}_{3.m}$, the labels are still associated with bits and only one of the two encryptions per index generated using the labels is valid. This implies that only one label per index has significance. But note that in $\mathsf{Hybrid}_{3.m}$, the labels are independent of the bits associated with them (due to Hybrid_2). This essentially leads to the conclusion that irrespective of the type of operation, only one of the two encryption is valid and the valid encryption encrypts a label generated uniformly at random (new label) using another label generated uniformly at random (old label). This is equivalent to the encryptions generated by the Simulator in the ideal world. Hence, the output of this hybrid corresponds to the output of the simulator, Out_{Sim}.