ORTOA: One Round Trip Oblivious Access

Sujaya Maiyya, Divyakant Agrawal, Prabhanjan Ananth, Amr El Abbadi
University of California Santa Barbara
{sujaya_maiyya,agrawal,prabhanjan,amr}@cs.ucsb.edu

ABSTRACT

Cloud based storage-as-a-service is quickly gaining popularity due to its many advantages such as scalability and pay-as-you-use cost model. However, storing data using third party services, on third party servers, creates vulnerabilities, especially pertaining to data privacy. While data encryption is an obvious choice to achieve data privacy, attacks based on access patterns have shown that mere encryption is insufficient to fully hide the data from the storage vendor. Solutions such as Oblivious RAM (ORAM) and Private Information Retrieval (PIR) propose techniques to hide data access patterns. Hiding access pattern involves hiding both the specific data item accessed and the type of access – read or write – on the item. Most existing obliviousness solutions focus on hiding the accessed data item; whereas to hide the type of access, these techniques communicate twice with the remote storage, sequentially: once to read and once to write, even though one of the rounds is redundant with regard to a user's access request. To mitigate this redundancy, we propose ORTOA, One Round Trip Oblivious Access protocol that reads or writes data on remote storage *in one round without revealing the type of access*. To our knowledge, ORTOA is the first protocol to obfuscate the type of access in a single round, reducing the communication overhead in half. ORTOA focuses on hiding the type of access and due to its generalised design, it can be integrated with many existing obliviousness techniques that hide the specific data item accessed.

1 INTRODUCTION

Data privacy is becoming one of the major challenges faced by the database community today. Industries such as Facebook [6] and Google [11] are fined millions to billions of dollars for violating user data privacy, creating an urgent need to build systems that provide data privacy. When we think of data privacy, *data encryption* is the first solution that comes to mind. But many works [20, 19, 14, 27] have shown the inefficiency of preserving data privacy when only data encryption is used; they show that just by observing the data access patterns, an adversary can infer non-trivial information about the data or the user. These type of inference attacks are termed *access pattern attacks*.

Solutions such as Oblivious RAM (ORAM) [9, 26, 2, 24, 23] and Private Information Retrieval (PIR) [4, 29, 15] provide mechanisms that hide access patterns. Most of these works assume trusted clients who host their data on untrusted servers, typically managed by third party cloud providers. ORAM solutions achieve access pattern obliviousness by shuffling the physical locations of the data stored on the untrusted server after every access. While there are PIR schemes to write the data [2, 17], most PIR schemes focus on retrieving a data item without revealing the identity of the retrieved item to the external server. More recently, Pancake [13] proposed *frequency smoothing* to obfuscate access patterns wherein the access frequencies to all entries in the database are smoothed so that an adversary cannot infer any non-trivial insights on the data. Albeit using a less stringent but realistic security model, Pancake's frequency smoothing is shown to be highly pragmatic, with significantly better performance than ORAM based solutions.

In general, the access pattern obliviousness in the above schemes consists of two aspects: (i) hiding the exact data item, or rather the exact physical location of the data item accessed by a client; (ii) hiding the *type* of access, i.e. a read vs a write, requested by a client. To our knowledge, most existing solutions for access pattern obliviousness focus on proposing novel ways to solve aspect (i); whereas for aspect (ii), the most commonly [26, 24, 13, 23] adapted solution is to always perform a read followed by a write, irrespective of the type of client's request. Always reading followed by writing to hide the type of access incurs *two sequential rounds* of accesses between the clients and the external server resulting in significant overhead; *eliminating this additional overhead is the focus of this paper.*

The goal of this work is to provide a one round solution to read or write the data stored on an external server *without revealing the type of access*. This work does not focus on hiding what physical locations are accessed by the clients, which is orthogonal to hiding the type of access.

We propose, ORTOA, a novel One Round Trip Oblivious Access protocol to access a data item stored on an external untrusted server without revealing the type of access, *in a single round*. This reduction in one round of communication not only reduces the bandwidth cost but also plays a vital role in reducing end-to-end latency, especially in geo-distributed settings.

The paper is organised as follows: Section 2 presents the system and security model; Section 3 proposes a one round oblivious access solution using an existing cryptographic primitive, fully homomorphic encryption, and discusses the impracticality of this approach. Section 4 then presents our novel protocol, ORTOA, to obliviously read or write in one round, followed by a space-optimization of the proposed protocol. Section 5 provides a cost analysis of ORTOA.

2 SYSTEM AND SECURITY MODEL

2.1 System Model

ORTOA is designed for key-value stores where each data item is identified by a unique key and the operations supported are single key `GET`s and `PUT`s. The data is stored on an external server(s) managed by a third party, analogous to renting servers from third party cloud providers.

We assume the clients interacting with the data are trusted and the external server that stores the data is untrusted. Furthermore,

Author's address: Sujaya Maiyya, Divyakant Agrawal, Prabhanjan Ananth, Amr El AbbadiUniversity of California Santa Barbara, {sujaya_maiyya,agrawal,prabhanjan, amr}@cs.ucsb.edu.

the system uses a proxy model commonly used in many privacy preserving data systems [22, 24, 13, 25]. 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 is assumed to be highly available; ensuring high availability of the proxy is orthogonal to the protocol presented here.

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 or secure socket layer to mitigate message tampering.

2.2 Data and Storage Model

Each data item 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 item's key nor its value is stored in the clear at the server. For a given key-value item $< k, v >$, the keys are encoded using pseudorandom functions (PRFs) [1]. The determinism of PRFs 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. A procedure *Enc* is used to encode the values (this procedure is different in Section 3 and Section 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 only the type of access made by clients and *not the actual physical locations accessed by client requests*. We assume an honest-but-curious adversary that wants to learn the type of access performed by clients without deviating from executing the designated protocol correctly. The adversary can control all communication channels – the proxy to external server and the clients to proxy – as well as the external server. We further assume that while the adversary can access (encrypted) queries to and from a sender, it cannot inject any queries (say by compromising clients).

3 FHE BASED SOLUTION

This section presents a one round mechanism to hide the type of accesses using Fully Homomorphic Encryption (FHE) [8, 3, 7]. We first give a high level overview of FHE, and then present a one-round read-write solution that uses FHE, and finally discuss the practicality of the 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 and produces the ciphertext, ct, as shown in Equation 1; the corresponding 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 is greater than a threshold value, which is dependent on the given FHE scheme.

Homomorphic encryption schemes allow computing over encrypted data. Some homomorphic encryption schemes support addition [21, 1] and some other schemes support multiplication [5]. A fully homomorphic encryption scheme supports both addition and multiplication on encrypted data [8, 3, 7]. 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:

$$\mathcal{FHE}(m1; n1) + \mathcal{FHE}(m2; n2) = \mathcal{FHE}(m1 + m2; n1 + n2) \tag{2}$$

$$\mathcal{FHE}(m1; n1) * \mathcal{FHE}(m2; n2) = \mathcal{FHE}(m1 * m2; n1 * n2) \tag{3}$$

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

To hide the type of client access, i.e., read or write, from an adversary who might control the storage server, it is necessary for both read and write requests to be indistinguishable. Hence both operations need to both 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. This is especially challenging as the value to be read is stored only at the external server; due to this challenge, existing solutions use one round of communication to read an item and another round to write it.

Our goal is to use FHE to support the execution of read and write operations in a single round of communication on a 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 item (where the server stores $\mathcal{FHE}(v_{old})$), and v_{new} be the updated value of the data item, if the operation is a write. The challenge is to develop an FHE procedure (or computation) Pr with parameters $\mathcal{FHE}(v_{old})$ and $\mathcal{FHE}(v_{new})$ such that:

$$\begin{aligned} &For\ reads: \Pr(\mathcal{FHE}(v_{old}),\ \mathcal{FHE}(v_{new})) = \mathcal{FHE}(\boldsymbol{v_{old}}) \\ &For\ writes: \Pr(\mathcal{FHE}(v_{old}),\ \mathcal{FHE}(v_{new})) = \mathcal{FHE}(\boldsymbol{v_{new}}) \end{aligned} \tag{5}$$

With such a procedure, the external server can execute the same procedure Pr for both read and write requests but the result of Pr would vary based on the type of access. If we can design such a procedure, since the server already stores $\mathcal{FHE}(v_{old})$, the proxy

[1] Alternate to PRFs, searchable encryption schemes can also be used. The main requirement is a deterministic encoding of plaintext keys.

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 $C = [c_r, c_w]$ 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 $(v_{old} * c_r) + (c_w * v_{new})$

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 $\mathcal{FHE}(v_{old}) * \mathcal{FHE}(c_r) + \mathcal{FHE}(c_w) * \mathcal{FHE}(v_{new})$

With Procedure Pr that gives the desired outcomes as defined in Equation 5, 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 C such that for reads, $C = [1, 0]$ and for writes, $C = [0, 1]$.

(2) Proxy then sends $\mathcal{FHE}(C)$ i.e., $[\mathcal{FHE}(c_r), \mathcal{FHE}(c_w)]$, along with $\mathcal{FHE}(v_{new})$, where the value of v_{new} is a dummy value 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 evaluates Procedure Pr by using the read 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, as shown in Equation 7:

$$\begin{aligned} \textit{For reads} : \mathcal{FHE}(v_{old}) * \mathcal{FHE}(1) \ + \ \mathcal{FHE}(0) * \mathcal{FHE}(v_{new}) \\ = \mathcal{FHE}(v_{old}) \\ \textit{For writes} : \mathcal{FHE}(v_{old}) * \mathcal{FHE}(0) \ + \ \mathcal{FHE}(1) * \mathcal{FHE}(v_{new}) \\ = \mathcal{FHE}(v_{new}) \end{aligned} \tag{7}$$

The proxy uses $\mathcal{FHE}$'s secret-key to decrypt the output for reads to retrieve the data item's value; and ignores the output for writes.

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

While we have shown the theoretical feasibility of using FHE to read or write data obliviously in one round, this approach is not practically feasible, primarily due to the noise n necessary for homomorphic encryption (as shown in Equation 1). As noted earlier, the noise increases with each homomorphic computation, with the increase being especially drastic for homomorphic multiplications, which is used in Procedure Pr 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 [18] FHE library with BFV [7] scheme. The evaluation used values of size 1kb 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 data item, the noise value grew too large for the FHE decryption to succeed, essentially rendering this solution impractical to be of any use in real deployments. The inevitable multiplication in Procedure Pr′ for both reads and writes is the root cause of this in-feasibility. 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 the use of FHE. Since the existing value 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 mapping bits to secret labels is inspired by garbled circuit constructions [28, 16].

More precisely, if k is a data item's key and v its plaintext binary value, then the server stores:

$$< PRF(k), (sl_{b1}^{(1)}, \ldots, sl_{bj}^{(j)}, \ldots, sl_{b\ell}^{(\ell)}) >$$

where $|v| = \ell$, $sl_{bj}^{(j)}$ 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 $\forall j \ bj \in \{0, 1\}$ represents bit value 0 or 1 (indicated as the subscript). For example if $\ell = 3$ and $v = 101$, then the server stores $(sl_1^{(1)}, sl_0^{(2)}, sl_1^{(3)})$. The secret labels are generated using a pseudorandom function of the form $PRF(k, j, b, ts)$ that takes key k, position index j from left, the corresponding bit value b, and a timestamp ts. Because PRFs are deterministic functions, invoking the chosen PRF with the same set of inputs any number of times will result in the same output secret label.

The goal of ORTOA is to read and write data in one round-trip, without revealing the type of access. Intuitively, it is evident that to hide reads from writes, every access to a data item must write the data, which is what ORTOA does at a high level: it updates the secret labels of a data item whenever the item is accessed – be it for a read or a write. We use the 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 re-generate the last set of secret labels for a given data item, the proxy maintains the last accessed timestamp for each data item.

4.1 An Illustrative Example

For ease of exposition, we first explain how ORTOA executes reads and writes using a simple example. We 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 key-value tuple is $< k, 0 >$, i.e, the value associated with k is 0. Since the server does not store keys or values in plain, the corresponding tuple stored at the server is $< PRF(k)\ ,\ ol_0^{(1)} >$ 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 `Read(`k`)` or a `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 $PRF(k, 1, b, ts)$ where where $b \in \{0, 1\}$ and ts is the last accessed timestamp of k. For a given index, the proxy needs to generate labels for both bit values 0 and 1 since it does not know the actual value, stored only at the server.

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 $PRF(k, 1, b, ts')$ where $b \in \{0, 1\}$ and ts' is the current timestamp and it updates the last accessed timestamp of k to ts'.

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 for 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 a single new label $nl_1^{(1)}$, decrypting them using either of the old labels will result in the desired, updated labels that reflect 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

Having described a simple example that uses ORTOA to read and write data without revealing the type of operation, this section formally presents the ORTOA protocol.

1. Proxy: The proxy, upon receiving a `Read(`k`)` or a `Write(`k, v'`)` request from a client, where v' is an updated value for k, performs the following steps:

1.1 For each of the ℓ indexes of the value, the proxy generates two *old* secret labels corresponding to both bit-values 0 and 1 by passing the last accessed timestamp ts to the PRF:
$(ol_0^{(1)} \leftarrow PRF(k, 1, 0, ts),\ ol_1^{(1)} \leftarrow PRF(k, 1, 1, ts), \ldots,$
$ol_0^{(\ell)} \leftarrow PRF(k, \ell, 0, ts),\ ol_1^{(\ell)} \leftarrow PRF(k, \ell, 1, ts))$

1.2 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 current timestamp ts' to the PRF:
$(nl_0^{(1)} \leftarrow PRF(k, 1, 0, ts'),\ nl_1^{(1)} \leftarrow PRF(k, 1, 1, ts'), \ldots,$
$nl_0^{(\ell)} \leftarrow PRF(k, \ell, 0, ts'),\ nl_1^{(\ell)} \leftarrow PRF(k, \ell, 1, ts'))$

1.3 The proxy updates k's last accessed timestamp to ts'.

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:

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

Whereas for writes, assuming b_j represents the updated bit value at index j, the proxy encrypts only the new labels corresponding to the updated value v' using the old labels:

$$E = [< Enc_{ol_0^{(1)}}(nl_{b1}^{(1)}),\ Enc_{ol_1^{(1)}}(nl_{b1}^{(1)}) >, \ldots,$$
$$< Enc_{ol_0^{(\ell)}}(nl_{b\ell}^{(\ell)}),\ Enc_{ol_1^{(\ell)}}(nl_{b\ell}^{(\ell)}) >]$$

As noted above for writes, at each index j, both the old labels encrypt only one new label $nl_{bj}^{(j)}$ corresponding to v'.

1.5 The proxy pairwise shuffles each of the ℓ pairs of encryptions at each index 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. 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. Note that for writes, since both encryptions encrypt $nl_{bj}^{(j)}$, either decryptions will result in the new label corresponding to the updated bit bj 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.

After executing ORTOA for a data item the server always updates its stored secret labels. For reads, the updated labels reflect the existing value of the data item; for writes, the updated labels reflect the updated value of the data item. 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 last accessed timestamp for each key in the database. While the complexity of storing timestamps 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 timestamp is stored using 10 bytes, for a database of size 1 million items, the space used by the proxy is about 10mB.

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 is length of a plaintext value (and all values have same length), r the output of the PRF (in bits) that generates secret labels, and N the database size, then the total storage space at the server in bits is

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

4.3.2 *Communication Analysis*.

Every bit of the plaintext can have 2 possible values, either a 0 or a 1. Since the data values, or rather the data value mappings, are stored only the server, the proxy generates 2 possible secret labels, and the corresponding 2 encryptions, for each bit of the plaintext. The proxy then communicates 2 encryptions per bit to the server. If ℓ is the length of data values and E_{len} is the length of encrypted ciphertexts, for every data item accessed by a client, the communication cost is calculated as:

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

4.4 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?

One label represents two bits of the plaintext: We start with a simple case of using a single label to represent two bits of plaintext data, instead of one. In this case, the server stores $\ell/2$ labels for every data item, 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 $sl_{10}^{(3,4)}$ is a label corresponding 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. Since the proxy does not know the value, which is stored only at the server, it generates 4 secret labels for every 2-bits, 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, who then decrypts 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 the plaintext: This approach can be further extended where, in general, a single label represents y bits of the plaintext. For example $sl_{b_1 \ldots b_y}^{(1,\ldots,y)}$ is a label corresponding to bits $b_1 \ldots b_y$ from indexes 1 to y. This approach reduces the storage space to ℓ/y, i.e, the storage space reduces by a factor of y. Note that if 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 complexity increase: While the space optimized solution reduces the storage space at the server by a factor of y, it comes with an increased communication + compute overhead as more labels need to be communicated from the proxy to the server, as analysed next. As discussed in Section 4.3, the communication complexity of non-space-optimized solution is $(2 \cdot E_{len} \cdot \ell)$. Generalising this to when one secret label to represent y bits, there are 2^y possible unique combinations for every y bits of the plaintext and the server stores ℓ/y labels. So the communication (and compute) complexity becomes $(2^y \cdot E_{len} \cdot \ell/y)$, i.e, a factor of $2^y/y$ increase compared to non-space-optimized solution.

5 COST ANALYSIS

This section analyses the space vs. communication complexities of ORTOA to find the overall optimal value of y, followed by a dollar cost estimate of deploying ORTOA.

5.1 Calculating optimal y value

As discussed in Section 4.4, there exists a trade-off between the storage space and the amount of communication (and compute) 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 compute) increases exponentially.

To calculate the optimal value of y, we compare the overhead factors f_s, f_c, and the combined overall overhead of the system, $f_s + f_c$, as depicted in Figure 1. 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 $y = 2$, with $f_c = 2$. For any $y > 2$, the communication factor increases more swiftly 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.

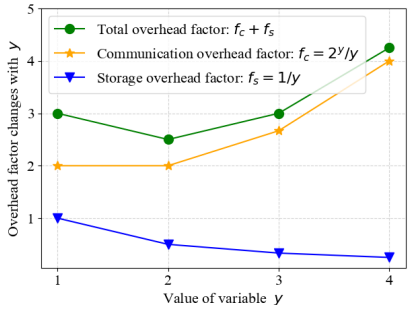

Figure 1: Storage vs communication overhead factor analysis to find optimal y value.

5.2 Dollar cost estimates

To gauge the practicality of ORTOA, we estimate the dollar cost of deploying ORTOA. To compute the estimates, the storage and communication costs of Google Cloud [10, 12] is considered, whose costs are comparable with 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. With the optimal y value of 2, and PRFs that produce 128-bit labels, i.e., $r = 128$, with data values of size 1kB, i.e., $\ell = 8000$, and with encryption schemes that produce 128-bit ciphertexts, i.e., $E_{len} = 128$, the following are the storage (per month) and communication cost estimates:

Storage: The dollar cost of storing 1 million data items is **\$1.28**.

Network: The bandwidth dollar cost of 1 million accesses is **\$30.72**

Compute: The dollar cost of 1 million function invocations, each at least 100ms long, is **\$7.00**

These dollar cost estimates indicate the practicality of ORTOA.

6 CONCLUSION AND FUTURE WORK

In this work, we propose ORTOA, a One Round Trip Oblivious Access protocol that reads or writes data stored on remote storage, potentially controlled by an adversary, in a single round of communication without revealing the type of access. Oblivious access techniques consists of two components: obfuscating the data item accessed by a client and hiding the type of client's access, i.e., read or write. Most existing obliviousness solutions focus on obfuscating the data item accessed by a client; whereas to hide the type of access, they require two rounds of communication. To our knowledge ORTOA is the first protocol to hide the type of access in a single round. ORTOA can be integrated with many existing techniques that hide access patterns. 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.

We plan to expand the paper, specifically extend ORTOA in three directions: (i) to provide formal security proof to argue for the obviousness guarantees of ORTOA; (ii) to experimentally evaluate ORTOA and compare it with two baselines: (1) a non-oblivious data access protocol, and (2) a two round oblivious data access protocol. The first baseline measures the overhead of obliviousness compared to a non-oblivious solution, and the second baseline measures the benefits of ORTOA over existing two-round-trip solutions; (iii) to discuss the scalability of ORTOA. The current design of ORTOA can be easily expanded to include multiple storage servers and/or multiple proxies, highlighting the scalability of ORTOA.

REFERENCES

[1] Josh Benaloh. "Dense probabilistic encryption". In: *Proceedings of the workshop on selected areas of cryptography*. 1994, pp. 120–128.

[2] Dan Boneh et al. "Public key encryption that allows PIR queries". In: *Annual International Cryptology Conference*. Springer. 2007, pp. 50–67.

[3] Zvika Brakerski. "Fully homomorphic encryption without modulus switching from classical GapSVP". In: *Annual Cryptology Conference*. Springer. 2012, pp. 868–886.

[4] Benny Chor et al. "Private information retrieval". In: *Proceedings of IEEE 36th Annual Foundations of Computer Science*. IEEE. 1995, pp. 41–50.

[5] Taher ElGamal. "A public key cryptosystem and a signature scheme based on discrete logarithms". In: *IEEE transactions on information theory* 31.4 (1985), pp. 469–472.

[6] *Facebook fined \$5B over data privacy violation*. "https://www.ftc.gov/news-events/press-releases/2019/07/ftc-imposes-5-billion-penalty-sweeping-new-privacy-restrictions". Accessed June 7, 2021.

[7] Junfeng Fan et al. "Somewhat practical fully homomorphic encryption." In: *IACR Cryptol. ePrint Arch.* 2012 (2012), p. 144.

[8] Craig Gentry et al. *A fully homomorphic encryption scheme*. Vol. 20. 9. Stanford university Stanford, 2009.

[9] Oded Goldreich et al. "Software protection and simulation on oblivious RAMs". In: *Journal of the ACM (JACM)* 43.3 (1996), pp. 431–473.

[10] *Google Cloud Pricing*. "https://cloud.google.com/storage/pricing". Accessed August 15, 2021).

[11] *Google fined \$57M over GDPR violation*. "https://digitalguardian.com/blog/google-fined-57m-data-protection-watchdog-over-gdpr-violations". Accessed June 7, 2021.

[12] *Google Function Pricing*. "https://cloud.google.com/functions/pricing". Accessed August 15, 2021).

[13] Paul Grubbs et al. "Pancake: Frequency smoothing for encrypted data stores". In: *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 2020, pp. 2451–2468.

[14] Mohammad Saiful Islam et al. "Access pattern disclosure on searchable encryption: ramification, attack and mitigation." In: *Ndss*. Vol. 20. 2012, p. 12.

[15] Eyal Kushilevitz et al. "Replication is not needed: Single database, computationally-private information retrieval". In: *Proceedings 38th annual symposium on foundations of computer science*. IEEE. 1997, pp. 364–373.

[16] Yehuda Lindell et al. "A proof of security of Yao's protocol for two-party computation". In: *Journal of cryptology* 22.2 (2009), pp. 161–188.

[17] Helger Lipmaa et al. "Two new efficient PIR-writing protocols". In: *International Conference on Applied Cryptography and Network Security*. Springer. 2010, pp. 438–455.

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

[19] Arvind Narayanan et al. "Myths and fallacies of" personally identifiable information"". In: *Communications of the ACM* 53.6 (2010), pp. 24–26.

[20] Arvind Narayanan et al. "Robust de-anonymization of large sparse datasets". In: *2008 IEEE Symposium on Security and Privacy (sp 2008)*. IEEE. 2008, pp. 111–125.

[21] Pascal Paillier. "Public-key cryptosystems based on composite degree residuosity classes". In: *International conference on the theory and applications of cryptographic techniques*. Springer. 1999, pp. 223–238.

[22] Raluca Ada Popa et al. "CryptDB: Protecting confidentiality with encrypted query processing". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 2011, pp. 85–100.

[23] Ling Ren et al. "Constants Count: Practical Improvements to Oblivious {RAM}". In: *24th USENIX Security Symposium ({USENIX} Security 15)*. 2015, pp. 415–430.

[24] Cetin Sahin et al. "Taostore: Overcoming asynchronicity in oblivious data storage". In: *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2016, pp. 198–217.

[25] Emil Stefanov et al. "Oblivistore: High performance oblivious cloud storage". In: *2013 IEEE Symposium on Security and Privacy*. IEEE. 2013, pp. 253–267.

[26] Emil Stefanov et al. "Path ORAM: an extremely simple oblivious RAM protocol". In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 2013, pp. 299–310.

[27] Frank Wang et al. "Splinter: Practical private queries on public data". In: *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*. 2017, pp. 299–313.

[28] Andrew Chi-Chih Yao. "How to generate and exchange secrets". In: *27th Annual Symposium on Foundations of Computer Science*. IEEE. 1986, pp. 162–167.

[29] Sergey Yekhanin. "Private information retrieval". In: *Locally Decodable Codes and Private Information Retrieval Schemes*. Springer, 2010, pp. 61–74.