ORTOA: A Family of One Round Trip Protocols For Operation-Type Obliviousness

Sujaya Maiyya
University of Waterloo

Yuval Steinhart
UC Santa Barbara

Adrian Davila
University of Waterloo

Jason Du
University of Waterloo

Divyakant Agrawal
UC Santa Barbara

Prabhanjan Ananth
UC Santa Barbara

Amr El Abbadi
UC Santa Barbara

ABSTRACT

Many applications relying on cloud storage services typically encrypt their data to ensure data privacy. However, reading or writing the encrypted data to serve client requests reveals the type of client operation to a potentially untrusted cloud. An adversary can exploit this information leak to compromise a user's privacy by tracking read/write access patterns. Existing approaches such as Oblivious RAM (ORAM) schemes hide the type of client access by always reading and then writing the data *sequentially* for both reads and writes, rendering one of these rounds redundant with respect to a client request. To mitigate this redundancy, we propose ORTOA- a family of protocols enabling single-round data access on remote storage *without revealing the operation type*. Specifically, we propose three protocols, two using existing cryptographic primitives of fully homomorphic encryption and trusted execution environments (TEEs), and a new primitive inspired by garbled circuits. Each of these protocols has different trust assumptions, allowing an application to choose the option best suited for its needs. To our knowledge, ORTOA is the first to propose generalized protocols to obfuscate the type of access in a single round, reducing the communication overhead in half. The proposed techniques can pave the way for novel ORAM schemes that hide both the type of access and the access pattern in a single round. Our experimental results show ORTOA achieving throughput gains of **1.7x-3.2x** compared to a baseline requiring two rounds for access type concealment, with the baseline incurring latency **1.5-1.9x** that of ORTOA for 160B-sized objects.

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 (e.g., CryptDB [46] or Arx [45]) typically 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, as explained next.

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. [35] utilized observing only write accesses to perform a privacy attack. The core idea of such attacks is to have multiple snapshots of memory (or a database) to observe all entries that are modified in between snapshots and uncover sensitive data. Hiding reads and writes by modifying data even for reads can help mitigate or at least weaken the accuracy of such attacks. Hiding reads and writes can also add potential protection against multiple snapshot adversaries (e.g., [14]).

At an application level, an application is incentivized to hide the type of service it provides because side channel attacks such as [34] exploit these meta-data to reveal sensitive information. However, an application cannot maintain anonymity of its service even while encrypting its data because the read vs. write pattern of an application often reveals the type of service it provides. For example, social network applications tend to be extremely read-heavy [9], whereas IoT applications lean write-heavy [12].

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 either Oblivious RAM [28] or other techniques [32, 40] utilize two rounds 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 [2]; Google's traffic drops by 20% if search results take an additional 500*ms* to load [31].

Furthermore, with increasing privacy laws such as GDPR [24] 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 family of one round trip data access protocols that hide the type of client access to efficiently address the privacy challenges caused by revealing the type of access. Specifically, we propose three different single-round access type hiding protocols, two using existing cryptographic primitives of fully homomorphic encryption (FHE) [25] (§3) and trusted execution environments (TEEs) such as Intel SGX [33] or ARM TrustZone [4] (§4), and a new primitive inspired by garbled circuits [37, 63] (§5). Each of these protocols has different trust assumptions, allowing an application to choose an option best suited for its needs. These protocols hide the type of individual client access as well as the read/write distribution of an application. Note that the ORTOA protocols only hide the type of client access and not the object accessed by clients. These protocols are designed to be a stepping stone for building novel ORAM and other oblivious schemes.

1.1 Challenges with designing a one round access-type hiding protocol

To highlight the challenges of designing a one-round protocol to hide the type of access, 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 non-deterministic encryption schemes such as AES produce different ciphertexts even if the same value is encrypted multiple times; hence, an adversary cannot distinguish between new 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 client 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 client cannot re-encrypt an object's value (which is stored only at the server) without fetching the value first. Hence, the client 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. In this naive solution, the client sends an encrypted new value for writes or an 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 client, and (iii) responding to the client with the read value. But the challenge here is that any subsequent reads after the first read operation will fetch a dummy value, permanently losing an application's 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 server's inability to trivially perform any checks or computations in a secure manner. A few cryptographic primitives such as fully homomorphic encryption (FHE), trusted enclaves (TEEs), or secure multi-party computation (MPC) allow computing on encrypted data. MPC schemes either require multiple rounds of communications to securely execute a computation [37, 63] or need multiple non-colluding servers [53], making them incompatible with the goals of ORTOA. Meanwhile, both FHE and TEEs can be leveraged to design a one round access-type hiding protocol, as we show in §3 and §4. The core idea is to formulate a mathematical equation whose execution either retains the old value for reads (albeit with re-encrypted ciphertext) or updates the value for writes. However, FHE allows only for a small number of ciphertext computations when the computation involves multiplication, whereas TEEs require access to specialized hardware, which can potentially leak side-channel information compromising privacy. We discuss these proposed solutions and their limitations, after providing a background, in the respective sections §3 and §4.

To overcome the limitations of the FHE-based and TEE-based protocols, termed FHE-ORTOA and TEE-ORTOA respectively, we present a novel *label-based* solution. Both FHE and TEE-based ORTOA encrypt and store data values using homomorphic encryption and symmetric encryption respectively, whereas the label-version, which we term LBL-ORTOA, explores alternate approaches to represent and store data values. In particular, inspired by garbled circuit constructions [37, 63], LBL-ORTOA represents plaintext values in a binary format and encodes each bit with a *secret label* generated using pseudo-random functions (PRFs); the server only stores these encoded labels. 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, LBL-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 LBL-ORTOA lies in how the clients communicate with the server (via a proxy) to update the labels for both read and write requests *in a single round* (§5).

1.3 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. Oblivious RAM (or ORAM) schemes provide stronger privacy than ORTOA by not only hiding the type of operation but also obfuscating the exact object accessed by clients. While most ORAM schemes (or other oblivious mechanisms [32, 40]) require two rounds to access, the literature consists of a few specialized ORAM solutions that achieve single round *online* communication complexity [16, 22, 23, 26, 27, 38, 62]. Many of these solutions are based on Garbled-RAM or Garbled-circuits, which require the server to store and evaluate a garbled program *per request* [23, 26, 27, 38]. Garbled-RAMs do not take fixed length inputs and their execution time varies based on the input size as well as the data size. Specifically, evaluating garbled programs incur $O(polylogN)$ or $O(N^e)$ complexity (where N is the data size and e is a constant > 0) [23, 26, 38]. Importantly, these schemes cannot handle adaptively chosen queries, i.e., all client queries must be known a priori, and also require an offline pre-processing step to construct and outsource the garbled program. All these properties primarily differ from ORTOA, which has a simplistic server model, fixed length inputs, constant execution time, and requires no offline steps. On the other hand, a few ORAM based datastores that do not use Garbled-RAM such as [16, 22, 62] also have single *online* rounds. However, they need *offline* rounds per request to evict, i.e., write the data back, deeming them a multi-round solution. Note that although offline eviction can reduce the latency in the critical, 'online' step of serving a request, this limits concurrency by allowing either only a read or a write to occur, to avoid corrupting the data or returning inconsistent results. Additionally, 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 novel ORAM schemes or integrate with oblivious schemes such as [32, 40]. To show the possibility of designing such schemes, we briefly outline a sketch of a novel PathORAM [58]-like tree-based ORAM scheme that executes operations in one round using ORTOA in §8.

Contributions and roadmap:
This work proposes ORTOA, a family of one round trip access type hiding protocols. Particularly, we make the following contributions:
1. A homomorphic encryption based protocol, FHE-ORTOA (§3).
2. A trusted hardware based protocol, TEE-ORTOA (§4).
3. A novel technique of label based protocol, LBL-ORTOA (§5).
4. An extensive evaluation of the proposed protocols and a comparison with the two round trip baseline protocol §6.
5. A security analysis of the proposed protocols §7.

2 SYSTEM AND SECURITY MODEL

2.1 System Model

ORTOA protocols are designed for key-value stores where a unique key identifies a given data object, and 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 [13, 32, 39, 46, 50, 57]. 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. Although stateful, 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). Note that for the FHE-ORTOA and TEE-ORTOA versions, the only reason the system assumes a proxy is to store the secret key necessary to encrypt the queries and decrypt the results. If we assume the secret key is shared with all clients, these two versions do not require a proxy.

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 [59] 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 always 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 across the three versions of ORTOA). 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 [15, 42, 50, 57].
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 [46] or Arx [45] (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: FHE-ORTOA

After discussing a few non-private or incorrect one round naive solutions in §1, this section presents FHE-ORTOA, a one round mechanism to hide the type of accesses using an existing cryptographic primitive, Fully Homomorphic Encryption (FHE) [7, 21, 25]. 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:

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

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 [6, 44] and some other schemes support multiplication [20]. A fully homomorphic encryption (FHE) scheme supports both addition and multiplication on encrypted data [7, 21, 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 (explained at a conceptual high level):

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

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

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.

3.1 One-round oblivious read-write using FHE

We propose FHE-ORTOA, a mechanism that uses 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 $< PRF(k), \mathcal{FHE}(v) >$. Note that if all clients do not share the secret-key used for data encryption, an application will need a light-weight 'gateway' proxy to encrypt and decrypt data or queries on behalf of clients. However, for practical purposes, we consider this version of ORTOA to be proxy-less.

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 ProcessClientRequest, or Pcr for short, with parameters $\mathcal{FHE}(v_{old})$ and $\mathcal{FHE}(v_{new})$ such that:

$$\textit{For reads}: \text{Pcr}(\mathcal{FHE}(v_{old}) \ , \ \mathcal{FHE}(v_{new})) = \mathcal{FHE}(v_{old})$$

$$\textit{For writes}: \text{Pcr}(\mathcal{FHE}(v_{old}) \ , \ \mathcal{FHE}(v_{new})) = \mathcal{FHE}(v_{new})$$

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

Procedure Pcr′(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 Pcr′ is v_{old}; otherwise, for writes when $c_r = 0$ and $c_w = 1$, the result of Pcr′ 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, Pcr′ can be transformed to procedure Pcr to include FHE encrypted inputs:

Procedure
Pcr($\mathcal{FHE}(v_{old})$, $\mathcal{FHE}(v_{new})$, $[\mathcal{FHE}(c_r), \mathcal{FHE}(c_w)]$):
 return $\mathcal{FHE}(v_{old}) * \mathcal{FHE}(c_r) + \mathcal{FHE}(v_{new}) * \mathcal{FHE}(c_w)$

With Procedure Pcr that results in the desired outcomes, the next steps elaborate on the specific operations of a client and the server:

(1) Upon deciding to perform either a `Read(k)` or a `Write(k, v_new)` request, a client creates vector C such that for reads, $C = [1, 0]$ and for writes, $C = [0, 1]$.

(2) The client then sends $\mathcal{FHE}(C)$, 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) While at rest, we assume the PRF-encoded keys and encrypted values are stored in any standard key-value store such as Redis [48] or Apache Cassandra [3] within the server. The server, upon receiving the encoded key along with the 3 encrypted entities, reads the value currently stored at key $PRF(k)$ from the key-value store. It then executes Procedure Pcr by using the stored value $\mathcal{FHE}(v_{old})$ and the 3 entities sent by the client. The server then updates its stored value to the output of the computation and sends the output back to the client.

(4) Given that either c_r or c_w is 0, Procedure Pcr's output will either be $\mathcal{FHE}(v_{old})$ for reads or $\mathcal{FHE}(v_{new})$ for writes. Since 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 client decrypts $\mathcal{FHE}(v_{old})$ using FHE's secret-key to retrieve the data object's value. For writes, the client ignores the returned value.

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

3.2 Complexity Analysis

3.2.1 ***Space Analysis.*** In FHE-ORTOA, the server stores all keys encoded using a PRF and all values encrypted using FHE. If r is the output size (in bits) of the PRF that generates encoded key, FHE_{len} is the length of the FHE encrypted ciphertexts, and N the database size, then the server's storage space in bits can be calculated as:

$$\underbrace{r \cdot N}_{\textit{Space for keys}} + \underbrace{FHE_{len} \cdot N}_{\textit{Space for values}}$$

3.2.2 ***Communication Analysis.*** To access an object, each client sends three FHE encrypted ciphertexts, one each of c_r and c_w, and one for v_{new}, rendering the bits of data communicated from a client to the server as:

$$3 \cdot FHE_{len}$$

Note that the plaintext to ciphertext length expansion factor for most FHE schemes is quite large (~225x for the library we used, as will be explained in the next section).

3.3 Challenges with FHE based solution

Although FHE allows hiding the type of access in one round, this approach is not practically feasible, mainly due to the noise n of FHE. As noted above, the noise increases with each homomorphic computation, with the increase being especially drastic for multiplications, which the Procedure Pcr 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 [41] FHE library with BFV [21] scheme. The evaluation used BFV coefficients set to the following: degree=32768, default coefficient modulus, and default plain modulus with 20 bits. With these setting, we could encrypt a plaintext value of up to 32768 bytes into a ciphertext of size 7404922 bytes (7.4 MB), which has a ~225x length expansion factor.

Our experiments revealed that within 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 Pcr for both reads and writes is the root cause of this infeasibility. Due to this limitation, we do not perform any more experimental analysis or evaluations of this approach. However, we believe that our proposed FHE solution can be used in the future when better performing FHE schemes are invented that control the amount of noise amplification.

4 TEE BASED SOLUTION: TEE-ORTOA

This section proposes an alternate one round trip solution to hide the type of access using trusted execution environments (TEEs) such as Intel SGX [33] and ARM TrustZone [4]. TEEs are secure areas within a main processor that protect the code and data loaded inside it by ensuring data confidentiality and integrity. TEEs provide isolation for code and data from the operating system using CPU hardware-level isolation and memory encryption. Many existing data systems utilize TEEs to provide data confidentiality guarantees [47, 56, 64]. If a cloud vendor can provide hardware enclaves (i.e., TEEs), an application can deploy its entire system on the cloud, which enables the data and the trusted component to reside together, significantly reducing the communication latency compared to a trusted proxy-based system. Note that, similar to FHE-ORTOA, we consider this version of ORTOA to be proxy-less by assuming that clients share the encryption-key.

4.1 One-round oblivious read-write using TEEs

The core idea of TEE-ORTOA is to execute the ProcessClientRequest function described in Procedure Pcr′ of §3 within a trusted enclave rather than using FHE. However, utilizing TEEs require careful partitioning of a program into trusted and untrusted components. Any sensitive portion of a program should belong to the trusted component to be executed within the enclave, whereas non-sensitive code can be executed outside the enclave.

Similar to §3, a client that wants to read or write an object constructs 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). For reads, the client sets $v_{new} = \perp$; and otherwise, to an updated value. However, instead of encrypting the vector and v_{new} using homomorphic encryption, the client encrypts them using a standard symmetric key encryption scheme such as AES. It then sends the encrypted vector and v_{new}, along with the PRF-encoded key, to the server.

The server's task upon receiving a client request is to first fetch v_{old} from the underlying key-value store (e.g., Redis [48]) and then execute the computation in Procedure Pcr′. Since retrieving encrypted values from the underlying data store is non-sensitive, TEE-ORTOA executes this portion of the code outside the enclave. It then sends all 3 encrypted entities, C, v_{old}, and v_{new} to the enclave, which decrypts them all, executes Procedure Pcr′ within the enclave, and finally encrypts the result using standard encryption scheme. The result is sent outside the enclave and the server then updates the value of the PRF-encoded key to this result, as well as forwards it to the client. In fact, we simplify this protocol further wherein the client only sends a one-dimensional vector, c_r, which is set to 1 for reads and 0 for writes. The enclave code decrypts c_r and depending on its value, re-encrypts either v_{old} or v_{new}. Since the server cannot distinguish if the output of the enclave code has re-encrypted the old value or has updated the value, this solution hides the type of client request using TEEs in a single round of client-server communication.

4.2 Complexity Analysis

4.2.1 ***Space Analysis.*** The space and communication complexity analysis of TEE-ORTOA are similar to that of FHE-ORTOA. However, since the data values are encrypted using standard libraries such as AES, this version does not suffer from as high a length-expansion-factor from plaintext to ciphertext as in FHE. If r is the output size (in bits) of the PRF that generates encoded keys, E_{len} is the length of the encrypted ciphertext, and N the database size, then the server's storage space in bits can be calculated as:

$$\underbrace{r \cdot N}_{Space\ for\ keys} + \underbrace{E_{len} \cdot N}_{Space\ for\ values}$$

4.2.2 ***Communication Analysis.***
To access an object, each client sends two encrypted ciphertexts, one for c_r and one for v_{new}, rendering the bits of data communicated from a client to the server as:

$$2 \cdot E_{len}$$

4.3 Challenges with TEE based solution

While TEE-ORTOA does not suffer from severe performance limitations as in FHE-ORTOA, it has two main challenges. The first is the necessity of requiring specialized hardware support from the cloud providers. While many popular cloud vendors currently provide some form of TEE support, they lack uniformity, which makes it challenging for applications to migrate their system from one cloud vendor to another. The second, and more pressing of the challenges, is the vulnerability exposed by side-channel leakages in TEEs [8, 43, 52, 61]. These attacks at a high level track behaviours such as memory access patterns, page faults, or cache accesses to successfully reconstruct encryption keys, severely limiting the guarantees of TEEs. Solutions that protect against these side-channel attacks incur significant performance overheads and often require complex program redesigning [51, 54, 55]. Despite these challenges, TEE-backed deployments are quite popular today. We implement TEE-ORTOA without these expensive protection mechanisms and evaluate its performance in §6; we leave as future work, developing a TEE-based one round protocol that protects against side-channel attacks.

5 LABEL BASED SOLUTION: LBL-ORTOA

Having shown that using existing cryptographic primitives, FHE, as-is is impractical to provide the desired one round trip oblivious access approach, while the TEE-based solution requires unique hardware and may suffer from side-channel attacks, we propose a novel technique that uses encoded labels to build ORTOA, called LBL-ORTOA.

In designing this version of ORTOA, 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 [37, 63]. 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.

Since the goal of ORTOA of hiding reads from writes can only be achieved if every access to an object writes the data, LBL-ORTOA 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 system needs to maintain an access counter per object indicating the total access count of an object. For this solution to be proxy-less, this counter should be maintained by all clients. But ensuring that after a client updates a counter, it propagates the update to all other clients requires some notion of consensus across clients, complicating the system design. Hence, LBL-ORTOA relies on a trusted proxy to maintain such stateful information and all clients route their requests through the proxy. 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 oblivious schemes [13, 32, 39, 50, 57, 58] that handle *concurrent* requests also maintain such $O(N)$ datastructures (e.g., position maps) at the proxy in exchange for higher performance.

5.1 An Illustrative Example

For ease of exposition, we first explain how LBL-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 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 $PRF(k, 1, b, ct + 1)$ 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.

5.2 LBL-ORTOA Protocol

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

The Init(kv) procedure describes the data initialization process in LBL-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.

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         $labels \xleftarrow{\cup} l$
9         $i \leftarrow i + 1$
10   **end**
11    $kv' \xleftarrow{\cup} \{PRF(k), labels\}$
12 **end**
13 Return kv'

PROCEDURE PCR(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         $E \xleftarrow{\cup} \{Enc_{ol_0^{(i)}}(nl_0^{(i)}), \ Enc_{ol_1^{(i)}}(nl_1^{(i)})\}$
9   **else**
10        $E \xleftarrow{\cup} \{Enc_{ol_0^{(i)}}(nl_{b_i}^{(i)}), \ Enc_{ol_1^{(i)}}(nl_{b_i}^{(i)})\}$
11   **end**
12    $i \leftarrow i + 1$
13 **end**
14 $ct \leftarrow ct + 1$
15 Pairwise shuffle E
16 Return E

Figure 1: LBL-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.

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, or PROCEDURE PCR for short, 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):
$\{ol_0^{(1)} \leftarrow PRF(k, 1, 0, ct), ol_1^{(1)} \leftarrow PRF(k, 1, 1, ct),$
$\ldots,$
$ol_0^{(\ell)} \leftarrow PRF(k, \ell, 0, ct), ol_1^{(\ell)} \leftarrow PRF(k, \ell, 1, ct)\}$

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 LBL-ORTOA.

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):

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

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):

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

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

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 LBL-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 LBL-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, LBL-ORTOA hide the type of operation in one round without restricting the number of accesses, unlike the FHE approach, or requiring specialized hardware.

5.3 Complexity Analysis

5.3.1 ***Space Analysis.***
Proxy: The only information the proxy needs to maintain to support LBL-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 LBL-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 the PRFs that generate secret labels and encoded keys, and N the database size, then server's storage space in bits can be calculated as:

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

5.3.2 ***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, LBL-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.

5.4 Tolerating malicious adversaries

Although we primarily consider protection against honest-but-curious adversaries, LBL-ORTOA can be extended to protect against data tampering by malicious adversaries. We briefly outline the mechanism here. Since the proxy in LBL-ORTOA has the mapping of a plaintext bit-value to its corresponding label value, when it reads an object, it can easily detect any data tampering by checking whether each read label of a value matches with the labels for either 0 or 1. Note that the adversary can only corrupt the data; it can never correctly change the label corresponding to say bit 0 to bit 1 since the PRF key necessary to correctly generate labels is stored only at the proxy.

5.5 Challenges with label based solution

The main challenge with LBL-ORTOA is that its storage and communication complexities grow with the size of the data values, as is evident from §5.3. We experimentally measure the performance cost (measured in throughput and latency) of LBL-ORTOA as the value size grows in §6.3. With regard to computation, the server needs to (attempt to) decrypt all encryptions per bit of plaintext despite being able to successfully decrypt only one of them, incurring wasteful computations. We address some of these challenges with optimizations summarized in the next section. Another challenge with this version is the necessity of a stateful trusted proxy. This proxy does *not* pose a scalability bottleneck: LBL-ORTOA can easily scale by adding additional proxies without compromising correctness or security. However, the proxy poses a fault tolerance challenge since it stores information necessary to execute the protocol. We leave the task of exploring efficient techniques to ensure proxy fault tolerance to future work.

5.6 Optimizations

We perform two major optimizations to LBL-ORTOA: one to reduce the storage size by half without increasing the communication or computation complexity, and one to enable the server to decrypt only one encryption per plaintext bit to avoid wasteful computations. Due to space constraints, the Appendix provides complete details of the optimization techniques. At a high level, for storage size reduction, 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. The optimization morphs this such that r bits represent *two* bits of plaintext rather than one, cutting down the storage cost by half. To reduce the number of decryptions, we utilize the point-and-permute [5] optimization of garbled circuits. This technique involves strategically permuting the possible encryptions per bit of plaintext and generating two additional bits of information (i.e., r+2 bits) indicating the exact encryption to decrypt upon the next access. This reduces the server's computation cost in half since the server only decrypts one of the two encryptions sent per bit of plaintext.

6 EXPERIMENTAL EVALUATION

In this section, we discuss the merits and limitations of various versions of ORTOA by conducting experimental evaluations. In particular, we only experimentally measure the performance of TEE-ORTOA and LBL-ORTOA since FHE-ORTOA using existing FHE implementations show impractical results (§3). If future efficient FHE implementations are developed, the practical viability of FHE-ORTOA can be reevaluated.

Baseline: In evaluating ORTOA, we consider a two-round-trip (2RTT) protocol as the baseline wherein each request by a client – read or write – translates 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 [13, 32, 39, 50, 57, 58].

Goals: We aim to answer four questions through evaluations:

(1) How does the TEE and label version of ORTOA compare with the 2RTT baseline when the client-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 LBL-ORTOA and the baseline on AWS, whereas TEE-ORTOA on Azure due to the availability of Intel SGX machines. For simplicity, even TEE-ORTOA and the baseline utilize a proxy to store the encryption key and all (concurrent) client requests are routed through the proxy. On AWS, the clients, proxy, and server were deployed on r5.xlarge instances each 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. On Azure, we deployed Intel SGX supported machines of spec Standard DC48s v3, 48 vcpus and 384 GiB memory. We note that Azure supports SGX enabled machines in a limited number of datacenters, including in the Virginia datacenter, where we placed the server. To have identical communication latencies as in LBL-ORTOA, the TEE version placed the client in the Virginia datacenter as well and simulated the proxy-to-server latency using the Linux `tc` command. ORTOA's implementation can be found at https://github.com/dsg-uwaterloo/ORTOA.

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 and Azure. In our experiments, the servers for both ORTOA protocols 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 [15, 42] 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 performances on three real world datasets: (i) An Electronic Health Record (EHR) data consisting of patients' heart disease records [19], (ii) SmallBank [1] data focusing on single object read/write requests rather than transactional workloads, and (iii) e-Commerce dataset [60] from UCI's machine learning repository consisting of records on customers' online retail purchases. §6.4 discusses more details on the datasets and the performance of the two versions.

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 the two ORTOA protocols 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 2 notes the round-trip time (RTT) latencies from California to the other datacenters. Since the TEE approach has a major limitation wherein only a limited datacenters support SGX enabled machines, TEE-ORTOA emulated this setup using the tc command to simulate similar cross datacenter latencies as in Table 2. Note that we do not place the server 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 [11]. Further, this experiment runs 32 concurrent client requests and Figure 2a 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.

	Oregon	N. Virginia	London	Mumbai
California	21.84	62.06	147.73	230.3

Table 2: RTT latencies across different datacenters in ms.

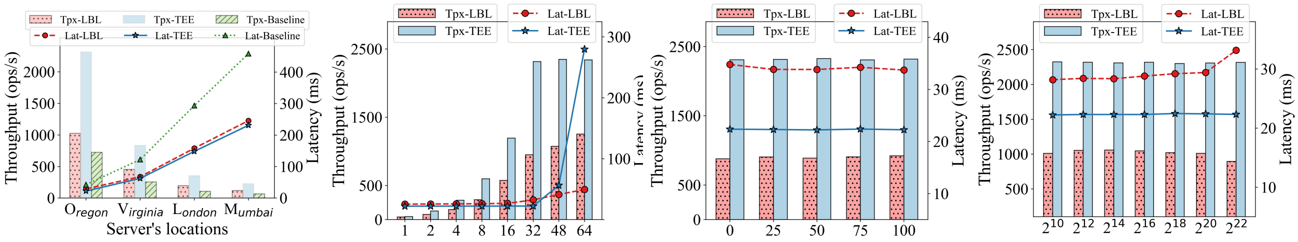
As seen in Figure 2a, as the physical distance between the proxy and the server increases, latency increases and throughput decreases for both the ORTOA protocols and the 2RTT baseline. Comparing the two versions of ORTOA, the TEE version consistently outperforms the label version with TEE's throughput values between **0.9-1.2x** higher than LBL's and its latency is about **20%** lower than the LBL version. The reason for this performance difference primarily stems from the increased amount of computation required both at the proxy and the server side for LBL-ORTOA compared to the simplistic computation that occurs in TEE-ORTOA. This indicates that if and when trusted enclaves are available at a cloud vendor, utilizing it helps improve ORTOA's performance. However, as noted in the experimental setup section, Azure supports SGX machines in limited datacenters. Hence, the TEE-ORTOA version may lose its performance benefits when the SGX enabled servers reside far from majority of the clients. For example, say if Virginia is the only datacenter with TEE enabled machines but an application has all of its clients in Asia, then based on Figure 2a, the LBL version is a better choice than TEE-ORTOA.

Comparing the two versions of ORTOA with the two round trip baseline, the experiment indicates that across all server locations, the two versions of ORTOA outperform the 2RTT baseline. In particular, the latency of the 2RTT baseline is **1.5x** to **1.9x** that of the two versions of ORTOA. Inversely, LBL-ORTOA's throughput is about **1.7x** and TEE-ORTOA's is about **3.2x** that of the baseline. The primary reason for the baseline's lower performance stems from incurring higher communication latency since its computation latency is negligible compared to ORTOA. This experiment highlights the benefits of constructing a single round access type oblivious protocol, as compared to the state-of-the-art two-round approach.

6.2 Micro Benchmarking

This set of experiments evaluate the behavior of ORTOA protocols 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 (the TEE version emulates this setup).

6.2.1 ***Increasing Concurrency.*** To understand how the ORTOA protocols behave when clients' request load increases, this experiment measures their throughput and latency while the number of concurrent clients (implemented via threads) increases starting from 1, and the results are depicted in Figure 2b. As seen in the figure, LBL-ORTOA's performance strikes a neat balance at 32 clients with an average latency of ~30 ms and a throughput



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

Figure 2: (a) Throughput and latency for TEE- and LBL-ORTOA and the 2RTT baseline, where the proxy lies in the California datacenter and the server is placed at increasingly farther datacenters. (b) Performance measured with increasing the number of concurrent clients for TEE- and LBL-ORTOA. Both versions perform optimally at 32 clients. (c) Throughput and latency measured for both versions of ORTOA while increasing the percent of PUTs highlights their effectiveness in hiding the read/write ratios of an application. (d) TEE- and LBL-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 of LBL-ORTOA is mostly due to a single server performing large computations while storing increasing amounts of data in memory.

of ~1000 ops/s. This throughput is about 24x of the throughput at 1 client. Although the throughput at 64 clients is 26% higher than at 32 clients, the latency is 54% higher at 64 clients, making 32 a better choice. Similarly, for TEE-ORTOA, the obvious optimal concurrency is 32; the performance plateaus after that whereas the latency starts spiking after 32 clients. The reason for the stark increase in latency is that the server machines had 48 cores; so as the client concurrency approached and went beyond 48, the latency spiked. Additionally, the increased context switching (i.e., paging in and out) between the trusted enclave and untrusted host processing also increases the latency, which is a common behavior observed in TEEs. Note that both versions exhibit an increase in throughput compared to a client concurrency of 1 because when a single client injects requests, the system remains under-utilized and the client is the primary bottleneck. Since a concurrency of 32 clients has optimal throughput and latency for both versions of ORTOA, the following (and the previous) experiments choose the concurrency of 32 clients, all sending requests in parallel.

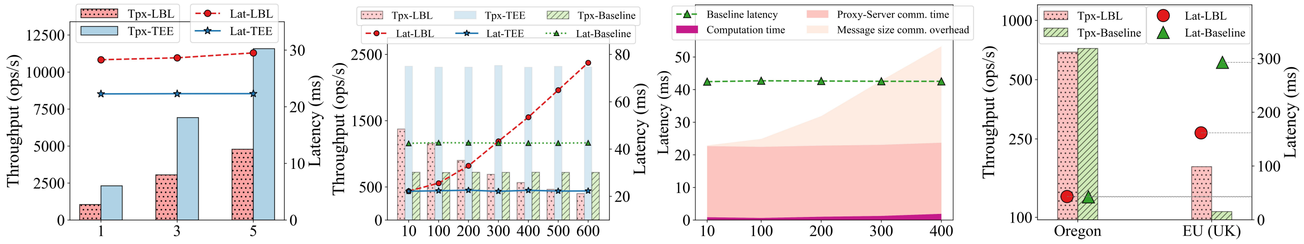
6.2.2 ***Varying the percent of writes.*** This experiment measures throughput and latency of the two versions while increasing the percent of PUT (or write) operations from 0 to 100%, as shown in Figure 2c. In this experiment, the server resides in Oregon and 32 concurrent clients read or write the data. As seen in the figure, the throughput and the latency values of LBL-ORTOA 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). Similarly, the TEE version has a consistent throughput of ~2320 ops/s incurring an average latency of ~23ms. This experimentally demonstrates the access-oblivious guarantee of ORTOA in that the performance remains the same regardless of the percentage of read or write operations in the client workload for both versions. 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 2d. As shown in the figure, for TEE-ORTOA, the throughput and latency remain mostly constant as the database size increases. Whereas, for LBL-ORTOA, 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 the computation (i.e., one decryption for each bit of the value) necessary to serve each request, impeding performance. The TEE version does not suffer from this degradation due to the limited amount of computation it requires in serving each client request. A standard approach to overcome the performance degradation in database systems is to scale the storage, which is what we do in the next experiment.

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 (TEE-ORTOA emulates this setup) and each server stores 1M objects. The resulting throughput and latency are shown in Figure 3a. Both versions of ORTOA scale near-linearly with the increasing number of servers and proxies: their peak throughput at a scale factor of 5 is about **5x** the throughput at a scale factor of 1. The latency remains constant across different scale factors for both versions. 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 LBL-ORTOA are directly proportional to ℓ (see §5.3), in this experiment, we measure throughput and latency of both versions while increasing the size of the values (where all values have equal length) from 10B to 600B with 32 concurrent clients sending requests and compare the performance with the 2RTT baseline; the results are depicted in Figure 3b. 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 LBL-ORTOA.



(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 3: (a) TEE- and LBL-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 TEE- and LBL-ORTOA, and the baseline while increasing the size of data values from 10B to 600B. Due to the large-message communication overhead of LBL-ORTOA, the baseline outperforms LBL-ORTOA starting at 300B, whereas TEE-ORTOA's performance remains unchanged. (c) Latency breakdown of LBL-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 LBL-ORTOA and the baseline when the server is placed in Oregon vs. EU.

As expected, LBL-ORTOA's throughput decreases and latency increases as the value size grows. At 300B both the baseline and LBL-ORTOA have comparable performance and the baseline starts outperforming LBL-ORTOA after that. Whereas, comparing the baseline with TEE-ORTOA, both protocols exhibit no performance fluctuations as the value sizes increase. Although the TEE version has this significant advantage compared to the LBL version, not all applications can benefit from and choose the TEE version due to the as yet limited support of trusted enclaves from all cloud vendors. Moreover, the side-channel leakages in TEEs [8, 43, 52, 61] may also limit the adoption of TEE-ORTOA. Given that the LBL version has no such limitations, the next section delves deeper to understand why its performance degrades as the value sizes increase and studies when is the 2RTT baseline better than LBL-ORTOA.

6.3.1 ***Latency breakdown of LBL-ORTOA***. We speculated the primary reason for LBL-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 3c. Surprisingly, 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 §5.3.2). Figure 3c plots the overall latency of the baseline to contrast with LBL-ORTOA's latency, which consists of computation time, the constant communication latency of 21.8ms, and the additional communication overhead time. We see after 300B LBL-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.

6.3.2 ***How to choose between LBL-ORTOA and the 2RTT baseline?*** To help an application choose between LBL-ORTOA and the baseline (assuming that TEEs are not a viable option), we provide the following equation: Let c be the cross-datacenter communication time between the server and the proxy, let p be LBL-ORTOA's processing or computation time, and let o be LBL-ORTOA's communication overhead time due to exchanging large messages. LBL-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 LBL-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 laws such as GDPR, which may disallow moving data outside of EU. The results are shown in Figure 3d.

As seen in the figure, when the server is placed in Europe, $c = 147.7ms$ and for LBL-ORTOA, $p + o = 21.7ms$, LBL-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 LBL-ORTOA. Even with low proxy-to-server communication latency, LBL-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 (such as images or videos), the 2RTT solution performs better than LBL-ORTOA.

6.3.3 ***Dollar cost analysis of LBL-ORTOA***. Since LBL-ORTOA incurs high storage and communication overheads, in this section, we discuss the estimated dollar cost of deploying LBL-ORTOA. To calculate the estimates, we consider the storage, communication, and compute costs of Google Cloud [29, 30], 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 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 §5.3 to recall the storage, communication, and compute complexity of LBL-ORTOA. With the above configuration, consider running LBL-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, LBL-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 LBL-ORTOA) when trusted enclaves are not a viable option.

6.4 Real world datasets

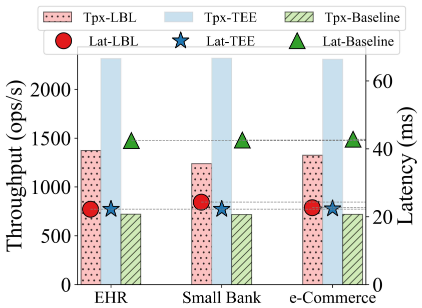


Figure 4: Throughput and latency comparison between the ORTOA protocols and the baseline for three practical applications based on real world datasets - Electronic Health Records (EHR), SmallBank data, and e-commerce data.

To assess ORTOA's behavior for real world applications, this experiment measures and compares the performance of its two versions 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 [19] 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[1]-like dataset for banking applications where, although SmallBank [1] 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 [60] 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.

This experiment measures the latency and throughput of the two versions 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 4, TEE-ORTOA's throughput is roughly **3.2x** that of the 2RTT baseline for all three applications. Whereas, LBL-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** that of the two versions of ORTOA. These performance differences are consistent with the performance differential between ORTOA and the baseline on synthetic datasets. This experiment indicates that for a variety of popular applications that have strong privacy requirements, ORTOA outperforms the 2RTT baseline.

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.

Real(A)

$output \leftarrow \emptyset$
for $a_i \in A$ **do**
  $output \xleftarrow{\cup} Process-ClientRequest(a_i)$
end
Return $output$

Ideal(K)

$output \leftarrow \emptyset$
for $k_i \in K$ **do**
  $output \xleftarrow{\cup} Simulator(k_i)$
end
Return $output$

Figure 5: 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.

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, and both produce outputs Out_{Real} and Out_{Sim} respectively consisting of a sequence of accesses to the external server. Note that A can be adaptively generated; ORTOA does not require A to be known a-priori. 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 correctness of ORTOA protocols, we consider a game $\mathcal{G}$, as shown in Figure 5. The game either executes Real or Ideal algorithm with uniformly random probability and provides the output to an adversary. Protocols of ORTOA are ROR-RW secure if the adversary, based on the received output, can identify the algorithm selected by the security game with negligible probability. Note that the signature for Procedure ProcessClientRequest (or PCR) differs syntactically but not semantically for the FHE and TEE versions and for the label version. For the security analysis, we simply assume that a client transforms an access in A to the necessary format (by encrypting the values either using FHE or standard encryption for the FHE and TEE versions respectively).

The Real algorithm invokes ORTOA's respective *ProcessClientRequest* procedure version 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*. Each version utilizes its own simulator so as to match the output of the respective real ORTOA protocol. 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 outputs that depend only on *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.

Theorem 1: A sequences of accesses $\mathcal{A}$ generated by the protocols of ORTOA is ROR-RW secure.
Proof: The formal proof can found in Appendix.

8 FUTURE WORK

1. Designing novel ORAM schemes: Apart from mitigating attacks that exploit the type of access on encrypted datastores, the ORTOA protocols proposed in this work can pioneer new oblivious schemes that hide both the access type and the accessed object *in a single round*. To show the possibility of designing such schemes, we briefly outline a sketch of a novel PathORAM [58]-like tree-based ORAM scheme that executes operations in one round. As the name suggests, tree-based ORAM schemes such as [39, 49, 50, 58] structure the outsourced data as a tree and store each outsourced object in a randomly chosen path. Specifically, in RingORAM [49], each node in the tree stores a fixed (maximum) number of real objects and dummy objects. To serve a client request, RingORAM reads the entire path on which the object resides, fetching all but one dummy objects at each level of the path. It temporarily stores the read object in a cache-like datastructure called *stash*, and finally shuffles the stash objects to store them in the path that was read, and writes the path back in an *eviction* step. This incurs two rounds of communication: once to read a path and once to evict it, i.e., write it back after shuffling. Although RingORAM and many other schemes [10, 39, 50] optimize by evicting the path as an offline process, they still require another round of communication. We can design a novel RingORAM-like scheme where reading and evicting a path can *occur in a single round* as follows: given that when a client requests an object, the adversary observes a random path, p, being accessed, the new scheme can identify at each level of p whether an object from this level is being read or being written. Reads would correspond to fetching the client requested object and writes are for evicting the objects in the stash. This negates the necessity of an offline eviction process. Similar to existing schemes, the read object would reside in the stash and be evicted upon subsequent accesses to the server. Even if the stash is empty, the scheme should access one object per level to avoid any information leakage. Such a scheme not only reduces the rounds of communication but also improve the concurrency since paths are accessed only once per request.

2. Supporting complex operations: Another direction we can expand ORTOA is by supporting more complex operations such as range queries, joins, and aggregates. We believe that reading and writing data objects form the core operations of any database system. Although ORTOA is proposed for a key-value store that supports read and write operations, the protocols as-is can support reading and writing on relational data based on primary keys (PKs). Support for queries such as point queries on non-PK attributes or range queries or joins can be achieved with additional data structures such as private indexing [18, 36], similar to SEAL [17], which builds a complex relational data system on top of a simple get-put supporting ORAM scheme.

9 CONCLUSION

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 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 family of one round data access protocols that hide the type of access. Leveraging cryptographic primitives like fully homomorphic encryption, trusted execution environments (TEEs), and a novel garbled circuits-inspired primitive, ORTOA offers flexibility in opting for suitable trust assumptions for applications. This is the first proposal 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 **1.7-3.2x** less requests per second than ORTOA for objects of size 160B.

REFERENCES

[1] 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.

[2] 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.

[3] APACHE CASSANDRA. https://cassandra.apache.org/_/index.html. Accessed December 14, 2023.

[4] ARM TRUSTZONE. https://www.arm.com/technologies/trustzone-for-cortex-m. Accessed October 1, 2023.

[5] 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.

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

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

[8] BRASSER, F., MÜLLER, U., DMITRIENKO, A., KOSTIAINEN, K., CAPKUN, S., AND SADEGHI, A.-R. Software grand exposure:{SGX} cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)* (2017).

[9] 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.

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

[11] 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.
[12] 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.
[13] Crooks, N., Burke, M., Cecchetti, E., Harel, S., Agrawal, R., and Alvisi, L. Obladi: Oblivious serializable transactions in the cloud. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (2018), pp. 727–743.
[14] Czeskis, A., Hilaire, D. J. S., Koscher, K., Gribble, S. D., Kohno, T., and Schneier, B. Defeating encrypted and deniable file systems: Truecrypt v5. 1a and the case of the tattling os and applications. In *HotSec* (2008).
[15] 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.
[16] 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.
[17] Demertzis, I., Papadopoulos, D., Papamanthou, C., and Shintre, S. {SEAL}: Attack mitigation for encrypted databases via adjustable leakage. In *29th USENIX Security Symposium (USENIX Security 20)* (2020), pp. 2433–2450.
[18] Demertzis, I., Papadopoulos, S., Papapetrou, O., Deligiannakis, A., and Garofalakis, M. Practical private range search revisited. In *Proceedings of the 2016 International Conference on Management of Data* (2016), pp. 185–198.
[19] EHR Dataset of Heart Diseases. https://www.kaggle.com/datasets/johnsmith88/heart-disease-dataset. Accessed October 14, 2022.
[20] ElGamal, T. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory 31*, 4 (1985), 469–472.
[21] Fan, J., and Vercauteren, F. Somewhat practical fully homomorphic encryption. *IACR Cryptol. ePrint Arch. 2012* (2012), 144.
[22] Fletcher, C., Naveed, M., Ren, L., Shi, E., and Stefanov, E. Bucket oram: single online roundtrip, constant bandwidth oblivious ram. *Cryptology ePrint Archive* (2015).
[23] 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.
[24] GDPR. https://gdpr-info.eu/. Accessed May 9, 2022.
[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 Cloud Pricing. https://cloud.google.com/storage/pricing. Accessed August 15, 2021.
[30] Google Function Pricing. https://cloud.google.com/functions/pricing. Accessed August 15, 2021.
[31] 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.
[32] 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.
[33] Intel SGX. https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/overview.html. Accessed October 1, 2023.
[34] 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.
[35] 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.
[36] Kamara, S., and Moataz, T. Sql on structurally-encrypted databases. In *Advances in Cryptology–ASIACRYPT 2018: 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2–6, 2018, Proceedings, Part I 24* (2018), Springer, pp. 149–180.
[37] 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.
[38] 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.
[39] 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.
[40] Maiyya, S., Vemula, S., Agrawal, D., El Abbadi, A., and Kerschbaum, F. Waffle: An online oblivious datastore for protecting data access patterns. *Cryptology ePrint Archive* (2023).
[41] Microsoft SEAL. https://docs.microsoft.com/en-us/azure/architecture/solution-ideas/articles/homomorphic-encryption-seal. Accessed June 15, 2021.
[42] 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.
[43] Moghimi, A., Irazoqui, G., and Eisenbarth, T. Cachezoom: How sgx amplifies the power of cache attacks. In *Cryptographic Hardware and Embedded Systems–CHES 2017: 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings* (2017), Springer, pp. 69–90.
[44] 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.
[45] Poddar, R., Boelter, T., and Popa, R. A. Arx: A strongly encrypted database system. *IACR Cryptol. ePrint Arch. 2016* (2016), 591.
[46] 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.
[47] Priebe, C., Vaswani, K., and Costa, M. Enclavedb: A secure database using sgx. In *2018 IEEE Symposium on Security and Privacy (SP)* (2018), IEEE, pp. 264–278.
[48] Redis. https://redis.io/. Accessed March 14, 2022.
[49] Ren, L., Fletcher, C. W., Kwon, A., Stefanov, E., Shi, E., van Dijk, M., and Devadas, S. Ring oram: Closing the gap between small and large client storage oblivious ram. *IACR Cryptol. ePrint Arch. 2014* (2014), 997.
[50] 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.
[51] Sasy, S., Gorbunov, S., and Fletcher, C. W. Zerotrace: Oblivious memory primitives from intel sgx. *Cryptology ePrint Archive* (2017).
[52] Schwarz, M., Weiser, S., Gruss, D., Maurice, C., and Mangard, S. Malware guard extension: Using sgx to conceal cache attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 14th International Conference, DIMVA 2017, Bonn, Germany, July 6-7, 2017, Proceedings 14* (2017), Springer, pp. 3–24.
[53] Shamir, A. How to share a secret. *Communications of the ACM 22*, 11 (1979), 612–613.
[54] Shih, M.-W., Lee, S., Kim, T., and Peinado, M. T-sgx: Eradicating controlled-channel attacks against enclave programs. In *NDSS* (2017).
[55] Shinde, S., Chua, Z. L., Narayanan, V., and Saxena, P. Preventing page faults from telling your secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security* (2016), pp. 317–328.
[56] Sinha, R., and Christodorescu, M. Veritasdb: High throughput key-value store with integrity. *Cryptology ePrint Archive* (2018).
[57] Stefanov, E., and Shi, E. Oblivistore: High performance oblivious cloud storage. In *2013 IEEE Symposium on Security and Privacy* (2013), IEEE, pp. 253–267.
[58] 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.
[59] TLS. https://datatracker.ietf.org/doc/html/rfc5246. Accessed April 14, 2022.
[60] UCI Online Retail Data Set. https://archive.ics.uci.edu/ml/datasets/online+retail. Accessed May 9, 2022.
[61] Van Bulck, J., Weichbrodt, N., Kapitza, R., Piessens, F., and Strackx, R. Telling your secrets without page faults: Stealthy page {Table-Based} attacks on enclaved execution. In *26th USENIX Security Symposium (USENIX Security 17)* (2017), pp. 1041–1056.
[62] 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.
[63] Yao, A. C.-C. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science* (1986), IEEE, pp. 162–167.
[64] Zheng, W., Dave, A., Beekman, J. G., Popa, R. A., Gonzalez, J. E., and Stoica, I. Opaque: An oblivious and encrypted distributed analytics platform. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (2017), pp. 283–298.

APPENDIX

10 OPTIMIZATIONS OF LBL-ORTOA

As seen in §5.3, LBL-ORTOA incurs high storage and computation costs. This section provides two optimization techniques, one to reduce the storage cost in half and the other to reduce the computation of LBL-ORTOA.

10.1 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?

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

One label represents two bits of plaintext: We start with a simple case where a single label represents two bits of plaintext data (Table 4), instead of one (Table 3). 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

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

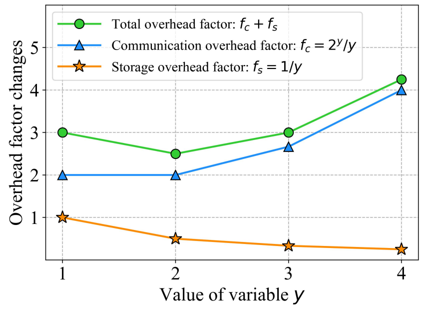


Figure 6: Storage vs. communication overhead factor analysis to find optimal y value - the value that indicates how many plaintext bits are represented by a single label.

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 6. 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 LBL-ORTOA.

10.2 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 4), the server can successfully decrypt only one of the 4 encryptions. In the protocol presented in §5, 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 LBL-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, LBL-ORTOA adapts the point-and-permute [5] 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 1^{st}, 2^{nd}, 3^{rd}, and 4^{th} 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 $(sl_{00}^{(1,2)}, \mathbf{10})$ for the plaintext indexes (1,2) of an object, the decryption bits 10 indicate that the server should decrypt only the $\texttt{10}^{th}$ 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 LBL-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 §5.2. 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 §5.2)):

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

Generalizing this to where LBL-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, LBL-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.

11 SECURITY ANALYSIS

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 5. 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* $\mathcal{A}$,

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

As noted in §7, this is an indistinguishability based definition that utilizes the real-vs-ideal security game defined in Figure 5. For Real algorithm in Figure 5, the game sends a sequence of m accesses, potentially adaptively generated, in A produced by clients where the game in-turn calls ORTOA's respective *ProcessClientRequest* procedure version for each access in A. Although the signature for Procedure ProcessClientRequest (or Pcr) differs syntactically but not semantically for the FHE and TEE versions and for the label version, for the security analysis, we simply assume that a client transforms an access in A to the necessary format (with either FHE or standard encryption for the FHE and TEE versions respectively). The Ideal algorithm, on the other hand, invokes a simulated function, *Simulator*. Each version utilizes its own simulator so as to match the output of the respective real ORTOA protocol. 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 outputs that depend only on *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.

In arguing for the security of different versions of ORTOA, we first briefly discuss the security of FHE-ORTOA and TEE-ORTOA, since these versions use existing cryptographic primitives, whereas we provide an elaborate proof for the LBL-ORTOA version as it introduces a new cryptographic technique.

11.1 Security of FHE-ORTOA and TEE-ORTOA versions

In proving Theorem 1, we split it into two parts, one for the FHE-ORTOA and TEE-ORTOA versions, and one for the LBL-ORTOA version. Since much of the proof in this subsection relies on the security of the underlying cryptographic primitives, the discussion is a brief proof sketch.

Theorem 1.1: A sequences of accesses $\mathcal{A}$ generated by FHE-ORTOA and TEE-ORTOA is ROR-RW secure.

Proof sketch: For the FHE-ORTOA version, the real algorithm encrypts the three entities c_r, c_w, v_{old}, and v_{new} using a fully homomorphic encryption scheme. The output of the real algorithm for each of the m accesses is the output of the Procedure ProcessClientRequest, or Pcr, defined in §3.1 that either retains the old value for reads or updates the value for writes, while the output of the procedure is always an FHE produced ciphertext. Whereas, the simulator generates a dummy value and encrypts it using the same FHE scheme as the real algorithm, and produces that as the output for each of the m accesses. FHE schemes are proven to be secure such that an adversary cannot distinguish if an FHE encrypted ciphertext corresponds to the values meaningful to the protocol or to some dummy value [25]. Hence, for this version, due to the security guarantees of FHE, an adversary cannot distinguish between the outputs produced by a simulator from the one produced by the real algorithm, proving the security of FHE-ORTOA.

For the TEE-ORTOA version, assuming that the protocol is executed within a trusted hardware enclave, the security proof boils down to the security of the encryption mechanism used to encrypt c_r, v_{old}, and v_{new}. The output of the real algorithm here produces one encrypted output (corresponding either to v_{old} or v_{new} depending on the type of access) for each of the m accesses. Whereas, the simulator generates a dummy value and encrypts it using the same encryption mechanism as the real algorithm for each of the m accesses. Assuming that the encryption scheme is IND-CPA, the adversary cannot distinguish if a ciphertext corresponds to meaningful plaintext values or to dummy values chosen by the simulator. Since the c_r, v_{old}, and v_{new} values are only decrypted within the enclave, by relying on the confidentiality guarantees of the enclave and that of the underlying encryption scheme, this version also ensures that the output sequence of the real algorithm is indistinguishable from that of the simulator.

11.2 Security of LBL-ORTOA version

This section provides an elaborate discussion on the security of LBL-ORTOA, unlike the ones for the other two version, since this is a new cryptographic technique proposed in the paper. In this version, the real algorithm calls the *ProcessClientRequest*, or Pcr, defined in Figure 1. Note that this procedure is a stateful algorithm. Let λ be the length of old and new labels generated by a PRF and let *Enc* be the IND-CPA secure encryption scheme deployed in the *ProcessClientRequest* procedure to encrypt new labels of length λ using old labels of length λ. Without loss of generality, for this proof, we assume the length ℓ of data values to be 1. Further, our proof considers the non-optimized protocol as presented in §5.2 but the proof easily extends to the optimized versions as well. 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 5, 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 $\ell = 1$:

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

Theorem 1.2: A sequences of accesses $\mathcal{A}$ generated by LBL-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

Procedure Simulator(k)

$E \leftarrow \emptyset$
`// Iterate over each of the ℓ indexes`
for ($i = 0; i < \ell; i++$) **do**
  Retrieve the old label $ol^{(i)}$ for k
  $nl^{(i)} \xleftarrow{\$} \{0,1\}^\lambda$
  $ol'^{(i)} \xleftarrow{\$} \{0,1\}^\lambda$
  $E \xleftarrow{\cup} \{Enc_{ol^{(i)}}(nl^{(i)}),\ Enc_{ol'^{(i)}}(0)\}$
  $ol^{(i)} \leftarrow nl^{(i)}$
end
Return E

Figure 7: Simulator pseudocode accessed in the Ideal algorithm.

Read: $(read, k, \perp)$

$\{Enc_{ol_b}(nl_{b'}),\ Enc_{ol_{1-b}}(nl_{1-b'})\} \leftarrow ProcessClientRequest(read, k, \perp)$

// 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.

$\equiv \{Enc_{ol_b}(nl_{b'}),\ Enc_{ol_{1-b}}(\{0,1\}^{\lambda})\}$

// From PRF's security, the new label can be replaced with a random string of length λ.

$\equiv \{Enc_{ol_b}(\{0,1\}^{\lambda}),\ Enc_{ol_{1-b}}(\{0,1\}^{\lambda})\}$ // From PRF's security, the old labels can be replaced with random strings of length λ.

$\equiv \{Enc_{\{0,1\}^{\lambda}}(\{0,1\}^{\lambda}),\ Enc_{\{0,1\}^{\lambda}}(\{0,1\}^{\lambda})\}$

Write: $(write, k, b')$

$\{Enc_{ol_b}(nl_{b'}),\ Enc_{ol_{1-b}}(nl_{b'})\} \leftarrow ProcessClientRequest(write, k, b')$

// 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.

$\equiv \{Enc_{ol_b}(nl_{b'}),\ Enc_{ol_{1-b}}(\{0,1\}^{\lambda})\}$

// From PRF's security, the label can be replaced with a random string of length λ.

$\equiv \{Enc_{ol_b}(\{0,1\}^{\lambda}),\ Enc_{ol_{1-b}}(\{0,1\}^{\lambda})\}$

// From PRF's security, the old labels can be replaced with random strings of length λ.

$\equiv \{Enc_{\{0,1\}^{\lambda}}(\{0,1\}^{\lambda}),\ Enc_{\{0,1\}^{\lambda}}(\{0,1\}^{\lambda})\}$

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 LBL-ORTOA generates labels of length λ.

the argument for this indistinguishability. The basis of our argument lies in the PRF deployed in LBL-ORTOA: the PRF function, *PRF*, produces labels that are indistinguishable from a uniformly sampled random variable $r \xleftarrow{\$} \{0,1\}^{\lambda}$. The argument in Figure 8 invokes *ProcessClientRequest* procedure once to read an object k and once to update k with bit value b'. 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.

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.

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 Hybrid$_{3.i}$ and 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 Hybrid$_{3.m}$ are identically distributed. The transition from Hybrid$_{3.m}$ to Hybrid$_4$ is as follows: in 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 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}.