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, serving client requests by reading or writing the encrypted data 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 communication overhead by 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 hiding, with the baseline incurring latency **1.5-1.9x** that of ORTOA for 160B-sized objects.

1 INTRODUCTION

Many modern applications, seeking to reduce the high costs associated with owning and maintaining on-premise storage, opt to outsource data storage to third-party cloud providers like Amazon AWS or Microsoft Azure. However, storing an application's data on the cloud in plaintext poses a risk of exposing sensitive information to potentially untrustworthy providers. To mitigate this, many applications employ data encryption techniques. Encrypted databases, such as CryptDB [41] or Arx [40], typically utilize a trusted front-end, often termed a *proxy*, to store the encryption key and direct all client requests to the untrusted storage. In a simple design for an encrypted key-value store supporting single-object GET/PUT requests, the proxy handles read requests by retrieving the appropriate encrypted value from storage, decrypting it, and responding to the client. For write requests, the proxy encrypts the value provided by the client and stores it in 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. [30] utilized observing only write accesses to perform a privacy attack. The core idea of such attacks is to uncover sensitive information by taking multiple snapshots of the memory (or a database) and observing all entries modified between snapshots. 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 [29] 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 [25] or other techniques [27, 34] 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 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 [26]. Given the substantial financial implications of increased latency, we advocate for new protocols that prioritize trading larger amounts of data for a reduced number of communication rounds.

Rooted in this motto, this work proposes *ORTOA*, a family of one round trip data access protocols addressing the above privacy challenges by concealing the type of client access. Specifically, we propose three different single-round access type hiding protocols, two using existing cryptographic primitives of fully homomorphic encryption (FHE) [22] (§3) and trusted execution environments (TEEs) such as Intel SGX [28] or ARM TrustZone [4] (§4), and a new primitive inspired by garbled circuits [31, 58] (§5). These protocols, each with different trust assumptions, empower applications to choose the most suitable option. They effectively hide both the type of individual client access and the read/write distribution of an application. ORTOA protocols hide the type of access and not the objects accessed by the clients. These protocols are designed to be a stepping stone for building novel single-round ORAM and other oblivious schemes that hide both types of sensitive information.

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

To illustrate the challenges of designing a one-round protocol to hide the type of access, we present two naive solutions. For both read and write requests to be indistinguishable, it is essential for each operation to involve both reading and writing at a given physical location. The two-round protocol achieves this by fetching the requested data, decrypting it, and either encrypting a new value for writes or re-encrypting the fetched value for reads before writing it back to the server. Note that standard non-deterministic encryption schemes such as AES guarantee that an adversary cannot distinguish between new value encryptions or same value re-encryptions.

Although reducing the two rounds of this protocol to a single round is straightforward for write requests by just updating the value without reading it, it proves challenging for reads: a client cannot re-encrypt an object's value without fetching the value first, rendering the one-round approach impractical.

Another naive solution is to treat all client requests as read-modify-write transactions. In typical read-modify-write transactions, a client interactively reads an object, modifies the read value, and writes back the updated value. The non-interactive version involves modifying the server to support this operation without client interaction. In this naive solution, the client sends an encrypted new value for writes or an encrypted dummy value for reads and the server performs a non-interactive read-modify-write by writing the (encrypted) value sent by the client and 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! If the server's logic is enhanced to handle read-modify-write transactions differently for read and write requests, it reveals the type of client query to the server. Therefore, such a single-round solution is not viable without compromising privacy or losing data.

1.2 Intuitions for ORTOA

The above discussed challenges exist primarily because of the server's inability to securely perform any checks or computations. Cryptographic primitives such as fully homomorphic encryption (FHE), trusted enclaves (TEEs), or multi-party computation (MPC) allow computing on encrypted data. MPC schemes either involve multiple communication rounds for secure computation [31, 58] or require multiple non-colluding servers [48], making them incompatible with the goals of ORTOA. Therefore, we leverage FHE and TEEs to design one round access-type hiding protocols. The core idea is to formulate a computation whose execution either retains the old value for reads or updates the value for writes. However, FHE has limitations in ciphertext computations involving multiplication, and TEEs require specialized hardware with potential side-channel leakage. The solutions and their limitations are discussed in detail in §3 and §4, after a primer on their backgrounds.

To overcome the limitations in FHE-based (FHE-ORTOA) and TEE-based (TEE-ORTOA) protocols, we introduce a novel *label-based* solution termed LBL-ORTOA. Unlike FHE-ORTOA and TEE-ORTOA, which encrypt and store data values using homomorphic encryption and symmetric encryption, LBL-ORTOA represents plaintext values in a binary format, encoding each bit with a *secret label* generated using pseudo-random functions. These encoded labels, rather than encrypted values, are stored at the server. LBL-ORTOA updates labels after each access to an object, both for reads and writes, in a single round, to prevent revealing the type of operation. The details are discussed in §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. While Oblivious RAM (ORAM) schemes (or other oblivious mechanisms [27, 34]) provide stronger privacy by hiding both the operation type and the specific object accessed, they often require two rounds for access. Some specialized ORAM solutions achieve single-round *online* communication complexity [16, 20, 21, 23, 24, 32, 57]. 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* [21, 23, 24, 32]. 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) [21, 23, 32]. 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. This necessary pre-processing step marks these as multi-round protocols, unlike ORTOA. Other ORAM-based datastores without Garbled-RAM also feature single online rounds but involve offline rounds per request to evict data, i.e., write the data back, rendering them multi-round solutions [16, 20, 57]. 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 read-write or write-write conflicts. ORTOA's focus on concealing access type in a generalized manner distinguishes it from solutions primarily targeting access patterns. ORTOA can be adapted to construct novel ORAM schemes or be integrated with oblivious schemes such as [27, 34]. To show the possibility of designing such schemes, we briefly outline a sketch of a novel PathORAM [53]-like access pattern hiding 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, LBL-ORTOA uses a proxy model commonly deployed in many privacy preserving data systems [13, 27, 33, 41, 45, 52]. The proxy is assumed to be trusted and the clients interact with the external server by routing requests through the proxy. 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., giga to tera bytes).

All communication channels – between clients, proxy, and 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 [54] 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 client 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 between any sender and receiver. 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, 36, 45, 52].
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 [41] or Arx [40] (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, 19, 22].

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 [6, 18, 22, 39]. These schemes add a small random term, called *noise*, to 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. 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.

Unlike partial homomorphic encryption [6, 18, 39], fully homomorphic encryption supports both adding and multiplying encrypted data [7, 19, 22], with the results remaining encrypted. Conceptually, for two values, $v1$ and $v2$, encrypted with $\mathcal{FHE}$, decrypting the output of $\mathcal{FHE}(v1) + \mathcal{FHE}(v2)$ results in plaintext addition of $v1$ and $v2$. Similarly, decrypting the output of $\mathcal{FHE}(v1) * \mathcal{FHE}(v2)$ results in the product of plaintext $v1 * v2$.

3.1 Hiding access type 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 this version is considered to be proxy-less by assuming that all clients share the secret-key used for data encryption; if clients do not share the secret key, an application will need a light-weight 'gateway' proxy to encrypt and decrypt data or queries on behalf of clients.

Let v_{old} be the current value of a given data object, which is stored only at the server (stored after encryption $\mathcal{FHE}(v_{old})$), 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 a procedure ProcessClientRequest, or Pcr for short, with parameters $\mathcal{FHE}(v_{old})$ and $\mathcal{FHE}(v_{new})$ such that:

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

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 server stores the encrypted key-value pairs in any standard key-value store such as Redis [43] or Apache Cassandra [3]. The server, upon receiving the client request, 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 FHEs to compute 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:

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

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).

3.2.2 ***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:

$$3 \cdot FHE_{len}$$

3.3 Challenges with FHE based solution

Although FHE allows hiding the type of access in one round, this solution is impractical primarily due to the noise (n) associated with FHE. The noise increases with each homomorphic computation, the increase being substantial for multiplications, which is required in both read and write accesses, as seen in Procedure PCR.

To assess the practicality of FHE-ORTOA, we developed and evaluated a prototype utilizing the Microsoft SEAL [35] FHE library with the BFV [19] scheme. The evaluation employed 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. 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 [28] 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 [42, 51, 59]. 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 Hiding access type 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 and c_w is a 1 for write operations. 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 [43]) 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 key-value store with 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 avoids severe performance limitations seen in FHE-ORTOA, it faces two primary challenges. First, it relies on specialized hardware support from 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, 37, 47, 56]. 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 [46, 49, 50]. 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 [31, 58]. 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 access 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.

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 and let its plaintext value be 0. The server 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. Proxy: The proxy, upon receiving a `Req(Read, k)` or a `Req(Write, k, v′=1)` request from a client, executes the following steps:

1.1 The proxy generates two ***old*** secret labels $< ol_0^{(1)}, ol_1^{(1)} >$ 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.*

1.2 The proxy next generates two ***new*** labels $< nl_0^{(1)}, nl_1^{(1)} >$ 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 $ct + 1$.

1.3 The details of this step depend on the type of access: for reads, the proxy encrypts each new secret label with its 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.:
$E = [< Enc_{ol_0^{(1)}}(\boldsymbol{nl_1^{(1)}}),\ Enc_{ol_1^{(1)}}(\boldsymbol{nl_1^{(1)}}) >]$

1.4 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.

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

2.1 The server tries to decrypt both encryptions it received 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 $ol_0^{(1)}$.

2.2 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. Since the proxy knows the mapping of secret labels to plaintext bit values, it learns the value of k to be 0 for reads and ignores the output for writes.

5.2 Hiding access types using LBL-ORTOA

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.

1. Proxy: When a client sends Req(Read, k) or a Req(Write, k, v') to the proxy, the proxy invokes the procedure ProcessClientRequest, or Pcr for short, as defined in Figure 1. Similar to §5.1,

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 (line 5):
$\{ol_0^{(1)} \leftarrow PRF(k, 1, 0, ct),\ ol_1^{(1)} \leftarrow PRF(k, 1, 1, ct),$
$\ldots,$
$ol_0^{(\ell)} \leftarrow PRF(k, \ell, 0, ct),\ ol_1^{(\ell)} \leftarrow PRF(k, \ell, 1, ct)\}$

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

```
PROCEDURE 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          $E \overset{\cup}{\leftarrow} \{Enc_{ol_0^{(i)}}(nl_0^{(i)}),\ Enc_{ol_1^{(i)}}(nl_1^{(i)})\}$
9      else
10         $E \overset{\cup}{\leftarrow} \{Enc_{ol_0^{(i)}}(nl_{b_i}^{(i)}),\ Enc_{ol_1^{(i)}}(nl_{b_i}^{(i)})\}$
11     end
12     $i \leftarrow i + 1$
13 end
14 $ct \leftarrow ct + 1$
15 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.

1.3 For each of the ℓ indexes of the value, the proxy next generates two *new* secret labels corresponding both bit values by passing the updated access counter $ct + 1$ 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 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

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.

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 secret label at each index of $PRF(k)$ 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, as in the FHE approach, or requiring specialized hardware, as in the TEE version.

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 technical report provides complete details of the optimization techniques [38]. 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 indicating the exact encryption to decrypt upon the next access. This reduces the server's computation cost to decrypting exactly one encryption 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, 27, 33, 45, 52, 53].

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 do the ORTOA and 2RTT 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 c6i.32xlarge instance with 8GiB memory and 128 cores @ 3.5GHz. 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.

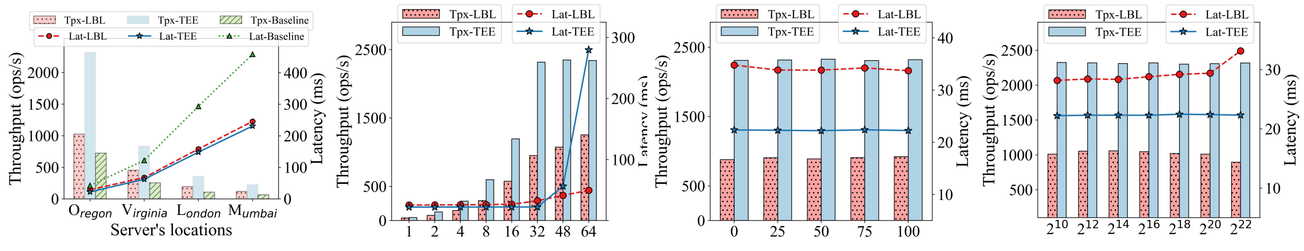
Unless stated otherwise, in each experiment a multi-threaded client (with a default of 32 threads) sends requests concurrently to the proxy, while each thread sends requests sequentially, i.e., it waits until its current request is answered before sending the next one. Each data point plotted in all 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, 36] 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, as measured by clients.

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 [17], (ii) SmallBank [1] data focusing on single object read/write requests rather than transactional workloads, and (iii) e-Commerce dataset [55] 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 placed the client and server in Virginia and emulated the above 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.

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



(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 highlight 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.

	Oregon	N. Virginia	London	Mumbai
California	21.84	62.06	147.73	230.3

Table 2: RTT latencies across different datacenters in ms.

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 a majority of 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 over 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 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. 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, 37, 47, 56] 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:

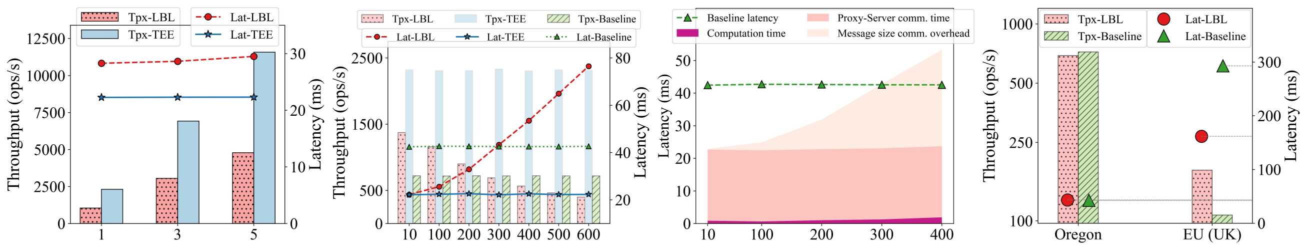
$$c > p + o$$

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.

6.4 Real world datasets

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 [17] with 14 attributes. For this dataset, we chose two attributes: a UUID to identify unique patients and their



(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.

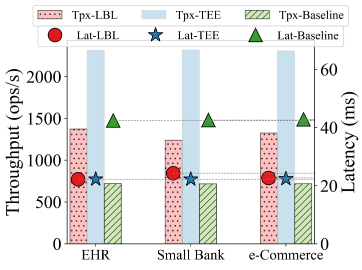


Figure 4: Throughput and latency comparison between the ORTOA protocols and the baseline for three practical applications based on real world datasets - Electronic Health Records (EHR), SmallBank data, and e-commerce data.

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 [55] 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 on real world data are consistent with those on synthetic data. 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. 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 [24]; 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($A$)
1 $output \leftarrow \emptyset$
2 for $a_i \in A$ do
3     $output \overset{\cup}{\leftarrow} Process-ClientRequest(a_i)$
4 end
5 Return $output$
```

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

Figure 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 $\mathcal{G}$ that 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, along with a detailed security definition, can be found in the technical report [38].

8 FUTURE WORK

Designing novel ORAM schemes: Apart from mitigating attacks exploiting access type on encrypted datastores, the ORTOA protocols 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 [53]-like tree-based ORAM scheme that executes operations in one round. As the name suggests, tree-based ORAM schemes such as [33, 44, 45, 53] structure the outsourced data as a tree and store each outsourced object in a randomly chosen path. Specifically, in RingORAM [44], 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 real 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, 33, 45] optimize by evicting the path as an offline process, they still require one round of communication for writing. 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 improves the concurrency since paths are accessed only once per request.

9 CONCLUSION

Encrypted databases leak information about 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. Specifically, we propose three versions of ORTOA, each varying in its trust assumptions. Leveraging cryptographic primitives like fully homomorphic encryption, trusted hardware enclaves, 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. ORTOA can be utilized to develop novel ORAM schemes that hide access patterns in one round. 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's latency is **1.5-1.9x** that of ORTOA, with **1.7-3.2x** throughput difference than the ORTOA protocols 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., Agarwal, R., and Alvisi, L. Obladi: Oblivious serializable transactions in the cloud. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (2018), pp. 727–743.

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

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

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

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

[21] 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.

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

[23] 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.

[24] 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.

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

[26] 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.

[27] 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.

[28] Intel SGX. https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/overview.html. Accessed October 1, 2023.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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).

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

[36] 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.

[37] 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.

[38] ORTOA Extended Version. https://cs.uwaterloo.ca/~smaiyya/assets/papers/ORTOA.pdf. Accessed October 1, 2023.

[39] 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.

[40] Poddar, R., Boelter, T., and Popa, R. A. Arx: A strongly encrypted database system. *IACR Cryptol. ePrint Arch. 2016* (2016), 591.

[41] 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.

[42] 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.

[43] Redis. https://redis.io/. Accessed March 14, 2022.

[44] 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.

[45] 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.

[46] Sasy, S., Gorbunov, S., and Fletcher, C. W. Zerotrace: Oblivious memory primitives from intel sgx. *Cryptology ePrint Archive* (2017).

[47] 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.

[48] Shamir, A. How to share a secret. *Communications of the ACM 22*, 11 (1979), 612–613.

[49] Shih, M.-W., Lee, S., Kim, T., and Peinado, M. T-sgx: Eradicating controlled-channel attacks against enclave programs. In *NDSS* (2017).

[50] 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.

[51] Sinha, R., and Christodorescu, M. Veritasdb: High throughput key-value store with integrity. *Cryptology ePrint Archive* (2018).

[52] Stefanov, E., and Shi, E. Oblivistore: High performance oblivious cloud storage. In *2013 IEEE Symposium on Security and Privacy* (2013), IEEE, pp. 253–267.

[53] 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.

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

[55] UCI Online Retail Data Set. https://archive.ics.uci.edu/ml/datasets/online+retail. Accessed May 9, 2022.

[56] 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.

[57] 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.

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

[59] 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.