

Obliv-DB Framework: Secure Query Processing on Cloud-based Database Systems using Distributed Oblivious Access

G. Dumindu Samaraweera and J. Morris Chang

Department of Electrical Engineering, University of South Florida,
4202 E. Fowler Avenue, Tampa, 33620, Florida, United States.

Contributing authors: samaraweera@usf.edu; chang5@usf.edu;

Abstract

The explosion of big data along with cloud computing architecture has empowered cloud-based database infrastructures to efficiently manage large and distributed data volumes in the cloud by facilitating numerous data-driven applications. Recently, different systems have been proposed that can additionally protect the privacy of these data by keeping them encrypted at the database level, and enabling them to make trusted query executions over encrypted data. This approach has given the potential and much safer means to outsource private information to untrusted and distributed cloud platforms. However, regardless of their proficiency towards handling a large volume of private information, these techniques are exposed to different access pattern attacks. Oblivious Random Access Machine (ORAM) is a security primitive well known for mitigating such attacks; however, direct integration of ORAM into cloud-based database systems is much more challenging due to high-performance penalties and minimal query functionalities. In this paper, we propose a novel data processing framework for database systems in the cloud using distributed ORAM techniques and oblivious data structures, making database queries resilient to access pattern attacks. We implemented our framework on a practical database setup and evaluated the performance based on different industrial metrics. The experimental results demonstrate that our distributed approach has significant benefits for cloud-based database systems compared to the direct integration of ORAM primitives at the database level.

Keywords: database, query processing, security, big data, performance

1 Introduction

From the very early days to the big data era, every new wave of computing technology has accelerated data growth in countless ways. In terms of managing such a volume of data, during the last decade, distributed storage architecture has given the potential for the existence of Database-as-a-Service (DBaaS) model deployed on the cloud, and it has become an integral part of both data owners and users. However, despite the fringe benefits offered, this model has serious data security and privacy concerns as the third-party cloud operators might have access to the sensitive information of other users [1].

Toward mitigating these privacy challenges, various privacy/security-enhancing technologies for database systems have been proposed over the past years to achieve confidentiality of data from curious insiders and malicious outsiders. These solutions can be classified into two main classes at the high level. The first approach is focused on database systems configured with secure hardware and executing the queries within trusted execution environments such as enclaves (e.g. Intel SGX or ARM TrustZone) [2], [3], [4]. The second set of solutions is the encrypted databases (EDBs) mainly rely on numerous Property Preserving Encryption (PPE) technologies such as SQL-aware encryption [5], [6], order preserving/revealing encryption (OPE) [7], [8], searchable symmetric encryption (SSE) [9], [10], [11] and so on. There is a rich set of literature on these techniques and are basically designed by taking the advantage of encrypting the content of the database using different cryptographic approaches/layers, enabling them to execute queries over encrypted data.

It is well known that having data encryption alone is inadequate to guarantee the protection for data in outsourced privacy-critical database applications. Some of the recent work have demonstrated that even with the encryption mechanisms, sensitive information can be revealed through query access patterns when the client executes queries over encrypted data [12], [13], [14], [15], [16]. Moreover, they have already shown that the access pattern attacks can be combined with some external prior knowledge to launch statistical inference attacks, which can cause severe damage to data in terms of information privacy. Thus, most of the well-known aforementioned encrypted database schemes are vulnerable and leak at least some information [13], [12]. To that end, protecting database solutions against query access pattern attacks beyond the use of encryption/cryptographic approaches is of utmost importance for many of today's data-driven applications.

Oblivious Random Access Machine (ORAM) is a security primitive known to hide query access pattern on client-server architectures. The concept was first proposed by Goldreich et al. [17] decades ago, and the basic idea behind this scheme was to allow a client to hide its access pattern to the remote storage by continuously shuffling and re-encrypting data as they accessed. The ORAM is a fascinating approach from a privacy standpoint, and hence it has been a topic of active research for a long time, focusing on how to design more robust and efficient ORAM schemes. As a result, over the past, diverse

set of ORAM schemes have been progressed [18], [19], [20], [21] serving different applications. However, on the flip side, it is very challenging to adopt ORAM into the cloud-based systems and sometimes impractical due to its foundational lower bounds [22]. There are very limited number of literature discussing the applicability of ORAM in actual database systems using specially designed data structures [23], [24], [25]. But, how to instrument them on a real production environment is still unclear due to limited query functionalities (in terms of database operations) and the performance. At this point, we seek an ORAM-based solution that can be deployable on existing database architectures (with minimal changes) which can protect privacy-critical data against query access pattern attacks.

In this paper, we introduce a novel framework called *Obliv-DB* for database systems toward mitigating query access pattern attacks. We are looking at the problem in a more practical approach, beyond the theoretical boundaries, that can be deployable on cloud-based production databases handling a large volume of data. Our main intention is to design a more practical solution so that it can be easily adopted into existing modern database architectures as an additional security/privacy layer to mitigate access pattern attacks. To the best of our knowledge, there is no other work addressing the security of cloud-based database systems using practically efficient distributed ORAM schemes. The results of our experiments demonstrate that the proposed approach performs better than the direct integration of ORAM on database systems, as some of the previous literature suggested [24]. Concretely, our contributions of this article are (1) analyzing the challenges of existing ORAM schemes toward integrating them for database operations (Section 3), (2) proposing a framework (Obliv-DB) with special data structure to handle different database operations in a distributed ORAM setting (Section 4), (3) analyzing the threat model and security of the proposed solution (Section 5), (4) implementing and evaluating the framework in terms of various performance metrics using realistic workloads with Yahoo! Cloud Serving Benchmark (YCSB) [26] (Section 6).

The rest of the paper is structured as follows: Section 2 discusses the evolution of ORAM schemes toward practical deployment. Section 3 discusses the limitations and challenges of existing approaches with reference to database operations. In section 4, the Obliv-DB framework, a new distributed ORAM approach for database systems, is introduced. In section 5, the security analysis is given, while section 6 discusses the performance evaluation of Obliv-DB and its counterparts. Finally, the paper concludes with section 7, showing further observations and future work.

2 Progression of ORAM Schemes Toward The Practical Deployment

Designing a bandwidth and computational efficient ORAM scheme has been in the discussion for more than a decade now. Thus, there are a wide range of

ORAM implementations proposed to-date serving different applications. However, there is only a limited number of literature in the past having a discussion of integrating ORAM for database systems. In this section we first discuss the basic construction of ORAM schemes in general and their progression over the years. At the latter part, it reviews the suitability and readiness of these ORAM designs for practical deployment leading to a discussion of limitations and the challenges of them to integrate on database systems, in the subsequent section.

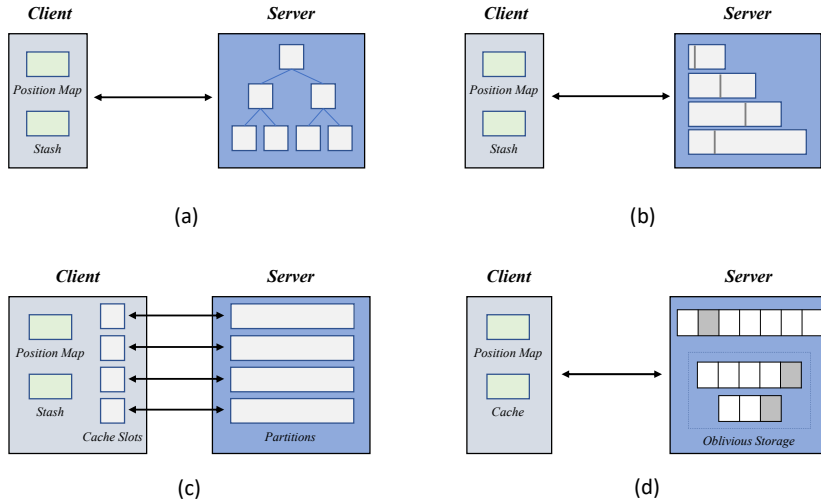


Fig. 1 The classification of ORAM designs. a) tree-based design b) layered design c) partition-based design d) large-message ORAM design.

2.1 Tree-based ORAM vs Other Architectures

In general, typical ORAM designs can be classified into four different groups; tree-based, layered, partition-based and large-message ORAMs [22] as illustrated in the Fig. 1. The tree-based design (e.g. [18]) usually consists with a binary tree whereas layered design (e.g. [27]) contains a hierarchical data structure. The partition-based ORAM (e.g. [28]) makes ORAM operations asynchronous by distributing across multiple servers with multiple partitions. Comparatively, large-message ORAMs (e.g. [29]) follow a more realistic oblivious storage model which can be comprehended in industrial workloads with a smaller client footprint. Despite this classification, it was found that most of the initial ORAM designs were costly in terms of bandwidth/performance; however, most of the recent bandwidth efficient schemes [20], [30], [31], [32] follow the architecture of tree-based design (or combination of tree-based [21]), and can achieve better performance with less communication overhead. Hence,

it appeared that tree-based ORAM architecture is one of the influential factors toward designing a new oblivious access method for database systems in cloud.

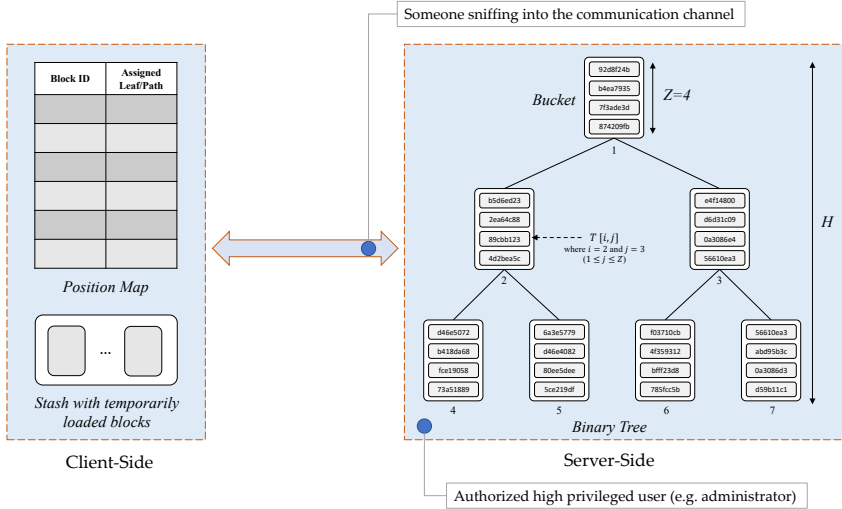


Fig. 2 Typical arrangement of a tree-based ORAM.

A typical tree-based ORAM consists of two data structures located at the server-side and client-side. The server is arranged as a binary tree whereas a tabular data structure called *position map* is stored at the client side. As depicted in Fig. 2, each node in the binary tree is a *bucket* and it can contain up to Z data blocks of size B . These data blocks usually has the same size and each can be identified using a unique identifier. The tree can have N leaf nodes (changes according to the height of the tree); hence, it can store N real blocks and the buckets are usually padded with dummy blocks whenever the slots are empty.

The access protocol of ORAM generally consists of two phases; retrieval and eviction. The *retrieval* operation refers to the client accessing/downloading a particular block in the tree by locating the path stored in the *position map*. Once the corresponding block is downloaded and processed, client then shuffles the location of the block, re-encrypts it and pushes back to the ORAM tree. This process is called as the *eviction* operation. The basic idea behind ORAM is to hide the data access pattern to the remote untrusted server or anyone sniffing the client-server communication by making an adversary not to learn anything about which data is actually being accessed, whether it is a read or write operation, how frequently it is accessed, or whether the same data is accessed twice.

2.2 Small Client vs Large Client Storage Schemes

With the growing interest of improving the efficiency and practicality of ORAM designs, research community has looked into the possibility of deploying an ORAM on storage/memory constrained client devices [33], [18], [34]. In 2013, Stefanov et al. [18] introduced Path ORAM, one of the extremely simple tree-based ORAM protocols, which consists of two data structures; *position map* and *stash* at the client-side and, a typical binary tree arrangement at the server-side. At any time, each block is assigned into a leaf bucket uniformly at random. Initially, the *stash* is empty and its role is to store data blocks temporarily during the block accesses. Whenever a block is read from the server, the entire path from the root bucket to the mapped leaf bucket is read into the *stash*. Once that block is processed, it then remaps to another leaf, encrypts the content with fresh randomness and sends back to the server along the path just read. The Path ORAM's basic principle is, at any given time each data block shall be stored either in a node along the path from the root to the corresponding leaf or it should be in the *stash*. The security of Path ORAM is mainly derived on the property of each data block is assigned to a leaf that is chosen uniformly at random and is updated after each data access. Even though each data access consists of both read and write operations, due to the shuffling and re-encryption process, only the client can distinguish reads from the writes mitigating the access pattern attacks.

Even though these small storage ORAM schemes perform reasonably well when dealing with single block accesses, it has been shown to be costly in terms of communication [22] for many other applications that requires large volume of asynchronous requests (e.g. cloud-based file access operation where file itself is composed of multiple ORAM blocks. Section 3.2 of the manuscript discusses these in details). Thus, in 2014 Ren et al. [19] introduced the first bandwidth efficient ORAM protocol, Ring ORAM, that improved the communication bottleneck of Path ORAM. The idea was very similar to Path ORAM scheme. However, the significant difference of this protocol to Path ORAM is that, unlike Path ORAM it only reads out one block from each bucket. This lowers the bandwidth overhead of the read operation. Since the *position map* only tracks the path containing the block of interest, Ring ORAM maintains some additional meta data along with the *position map* to locate the position of the block within a bucket.

As the frequency of eviction has a significant impact on communication efficiency of ORAM schemes, lately, large client storage systems looked into the ways of minimizing the number of evictions without degrading the security. Unlike Path ORAM, these schemes (e.g. Ring ORAM) do not reshuffle data in every access. Instead, they usually keep track of a counter of how many times a bucket is accessed. This counter then decides when to reshuffle the buckets. On the other hand, instead of using a randomized eviction procedure, evictions are typically performed in a deterministic/specific order called *reverse lexicographic order* as depicted in Fig. 3 by picking one leaf in every time step [35]. Specifically, for a degree- k and height- H tree, the leaf for eviction

path at the eviction count t is chosen by $\text{DigitReverse}_k(t \bmod k^H)$ where DigitReverse_k represents the order reversal of the binary string representation of the integer input. The idea of this operation is to reduce the frequency of eviction and to minimize the overlap between consecutive eviction paths so that it improves the quality of eviction [19].

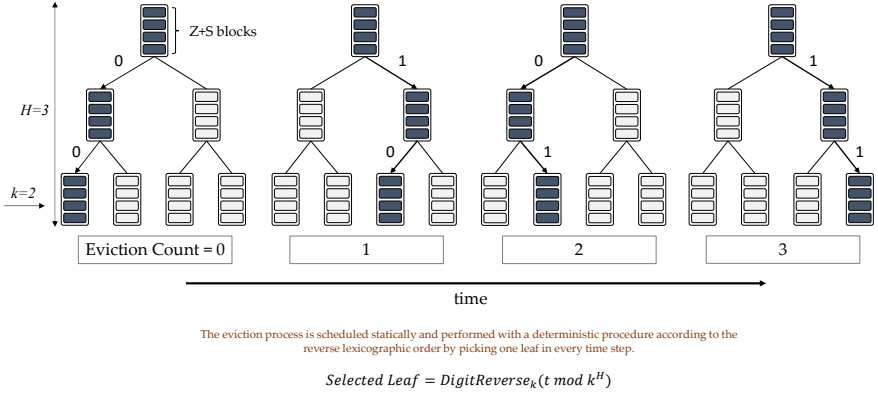


Fig. 3 How server performs the eviction in reverse lexicographic order of paths. Arrows indicate the buckets those on the eviction path. In the diagram, eviction path edges represent 0 and 1, where 0 means go to the left child and 1 means go to the right child. In this example, after eviction count=3, order repeats.

2.3 Taking the Advantage of Server-side Computation

Initial designs of the ORAM schemes have assumed to have a server as a simple storage device where client interacts with to read and write data but does not perform any computation. Lately, it was identified that untrusted server has a significant compute power possibly much greater than that of client, hence it can be utilized for heavy computations within the ORAM setting. To that end, different ORAM schemes have been evolved during the recent past allowing server-side computation while having the distinction between the amount of workload performed by the server and number of round-trips in terms of communication cost.

There are several different ways to harness the compute power of server within the ORAM paradigm [36], [37]. Among that, some of the ORAM schemes have made endeavors toward the direction of using homomorphic encryption (HE) while other schemes leverage the private information retrieval (PIR) techniques along with encryption. At a glance, the HE based schemes have evolved with the idea that instead of client moving data on the server, client can instruct the server to perform the ORAM operations and the eviction without revealing any data or its movements. While this is a very promising direction, it also suffers from several drawbacks when the ORAM is utilized

8 Secure Query Processing on Cloud-based Database Systems

with fully homomorphic encryption (FHE) due to the fact that the depth of the ORAM can grow unbounded with the increasing volume of ORAM requests.

In 2016, Devadas et al. [20] introduced a tree-based ORAM scheme called Onion ORAM which does not require FHE but employs an additively homomorphic encryption (AHE) that can achieve practical efficiency in the malicious setup. Again, this also follows typical architecture of tree-based ORAMs however, its eviction algorithm is different from others. During the eviction operation, Onion ORAM moves all blocks from parent bucket to two of its children depending on which move keeps the block on the path to its leaf. This is called *bucket triplet eviction* and its next eviction path is based on the *reverse lexicographic order* similar to previous designs. It is important to note here that the Onion ORAM's server storage consists of two types of data; data blocks and meta data. Each block in the bucket is tagged with meta data along with the logical address of the block and the leaf it is mapped to. The client generally reads and writes directly into this meta data and instructs the server to perform requests and evictions on actual data blocks homomorphically by sending some select vectors. Once the select vectors are received, server does the heavy work of performing computations over data blocks minimizing the cost of bandwidth.

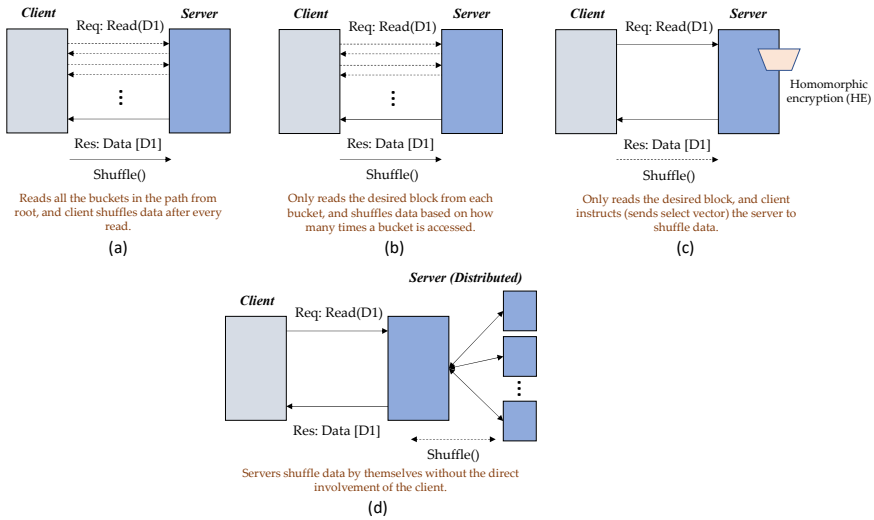


Fig. 4 The progression of different ORAM designs towards distributed setting: (a) Operation of tree-based design, (b) Minimizing the number of evictions, (c) Use of homomorphic encryption (HE) with server-side computation, (d) Circumventing homomorphic operations using distributed setup.

2.4 Circumventing Homomorphic Operations Using Distributed ORAMs

While HE operations can improve the computation efficiency of the underlying ORAM schemes, practically deploying them on cloud-based setup is more challenging since homomorphic operations are time consuming and hence, it incurs higher end-to-end delay. As a result, distributed ORAM schemes have been proposed. Stefanov et al. [38] introduced a multi-cloud oblivious storage that reduces client-cloud bandwidth cost, making ORAM more practical for bandwidth constrained clients. In [39] authors have proposed a multi-server oblivious storage using two non-colluding servers. The main idea of the solution is to let the two clouds shuffle the data among themselves without the direct involvement of the client. However, due to its partition ORAM framework [33] which requires $O(\sqrt{N})$ blocks of client storage it was costly for most of the memory limited clients. In 2015, Moataz et al. [39] proposed a constant communication ORAM without using homomorphic encryptions in exchange for expanding the traditional ORAM setting from single-server to multiple non-colluding servers. Later in 2017, Hoang et al. [32] introduced an ORAM scheme that also can achieve $O(1)$ client-server bandwidth while making it more practical solution for cloud-based applications. The authors have followed the idea of Shamir secret sharing protocol, PIR techniques along with a tree-based ORAM architecture to eliminate costly homomorphic operations.

The fig. 4 summarizes the evolution of these approaches in terms of data access and eviction operations. While different approaches/implementations of ORAMs have distinctive application scenarios, it is evident that having distributed ORAM construction as the underlying ORAM primitive is the most prominent way forward for the practical deployment of database operations, in terms of mitigating query access pattern attacks. On the other hand, having multiple server deployments is not an overhead for a database solutions as in many cases modern database systems itself are composed of multiple distributed servers for load balancing, data redundancy and high availability purposes. This is one of the influential factors for us to put forward a practically viable solution for database systems to mitigate access pattern attacks.

3 Limitations and Challenges of Existing Approaches in terms of DB Operations

As there are multiple different factors that govern the overall performance, it is quite challenging for a database system to integrate ORAM. Hence, the design of such system is kind of a balancing act between performance, security and the usability (practicality). Despite the limited attention given in the literature thus far, we believe it is important to precisely capture the threat model of cloud-based database solutions in order to discuss the possible counter measures using ORAM, toward mitigating query access pattern attacks. Thus, in this section, first we discuss the general adversarial threat model of database

systems to identify the inherent security benefits of ORAM schemes when they get implemented on database systems. Then we look into the problem that we are solving with regard to challenges and the limitations of existing approaches.

3.1 ORAM and the Database Adversarial Threat Model

The strongest threat to an outsourced database system in a cloud-based environment is the active attacker who has the ability to compromise the database server and perform arbitrary malicious operations. For an example, a system administrator of a cloud service provider usually have access to full set of server resources and rich set of privileges. However, latest security models usually focus on passive attacks that do not interfere with the functionality of database, assumed to be passive, acts in an honest manner but observes all its operations (honest-but-curious) [1].

- **Malicious Insider Attacks:**

With the increasing demand for Database-as-a-Service (DBaaS) model where the database resources are getting deployed on cloud-based computing infrastructures, the cloud system administrators and database administrators typically gain the access to privileged domains/resources. Hence, this threat is of utmost importance to address through a client-server combined approach or a mechanism deployed at server-side. Different encrypted database technologies can cryptographically isolate the direct access to privacy-critical information but it was found that EDBs cannot completely eliminate query access pattern attacks. Hence, the ORAM has been identified as a probable solution that can isolate this threat by continuously shuffling and re-encrypting data as they get accessed.

- **External Passive Attacks:**

While the malicious insider attacks originated within the boundary of cloud server, there can be another possible attacker who compromises the database server externally by just listening to the client-server communication or by listening to the conversation between server-server in a clustered environment. These type of external adversaries can observe the database operations including the queries that have been issued and the corresponding responses, and how server access data in response to these queries. External passive attacks can be diminished using secure communication protocols (e.g. SSL/TLS), however in many cases, most of the modern database systems do not facilitate this as a built-in feature [1].

A typical abstraction of database threat model along with a tree-based ORAM is depicted in Fig. 2.

3.2 Challenges of Integrating ORAM for DB Operations

Due to strong security grantees of ORAM, several different studies have investigated how to make an ORAM into a practical cloud-based storage. But in reality, most of these systems suffer from high bandwidth cost (and end-to-end

delay), making them unsuitable for bandwidth constrained clients. Consequently, some solutions along with encrypted databases (EDB) suggested to outsource the heavy workload to the cloud combining with trusted hardware to overcome the bandwidth cost [2], [3]. However, in this article, we are seeking a solution without the use of trusted hardware so that it can be easily adopted into many existing database architectures as an additional security/privacy layer to mitigate access pattern attacks

Regardless of the complexity in underlying ORAM structure, adoption of ORAM design into database landscape is quite challenging in many aspects. At first, security and performance model of ORAM is not in-line with today's cloud-based database applications. The ORAM concept was originated from the idea of memory protection hence, it does not take into the consideration unique performance requirements of modern database systems such as scalability, availability and dynamic resource allocation. These properties even do not appear to be the primary focus of the ORAM research.

Secondly, many ORAM designs essentially have synchronous data access schemes which make them ill-suited for many database operations that would be unusable without asynchronous access for large data volumes. One simplest solution to overcome this problem is to batch the requests together. However, it is more complicated than it appears to be. Bindschaedler et al. in their study [22] discovered an interesting observation that some ORAM schemes perform reasonably well when dealing with single block accesses but it incurs a huge slowdown when processing requests in batch. Their study even mentioned 71 times slower in some cases with the Path ORAM. Thus, in general, due to its own foundational lower bounds, most of the best known ORAM schemes incur about at least 10 to 20 times additional performance penalty.

In addition, even with the synchronous data accesses, due to the fact that most ORAM designs assume fixed-size blocks, it is sometimes more difficult to bring it into the database landscape as they need to utilize variable-size objects. For instance, if we consider mapping a simple database table into ORAM blocks, different columns might require different size of objects. Typical ORAM designs only support data blocks of the same size and when the block size varies they usually pad them up to a fixed size. But it could lead to performance degrade. Moreover, eviction is a crucial property of any ORAM scheme, but eviction process of many ORAM designs is not in accordance with database operations. For example, in practice, whenever the primary database fails one of the standby databases is transitioned to take over the primary instance. However, as in many ORAM schemes some part of the data is cached on the client-side (by design), and if the client fails, it is difficult to facilitate fail-over scenarios with ORAM. Therefore, it is required to correctly identify the gap between existing ORAM approaches and the current requirements of cloud-based database services to design an efficient framework that can integrate ORAM into database services.

Apart from some of the best known approaches of using ORAM as a simple cloud-based storage, there are a few studies that investigated the actual

implementation of ORAM in database systems. In 2016, Chang et al. [23] presented SEAL ORAM framework by implementing ORAM blocks in a row oriented approach where each block represents a unique row of a database table instance. This allows efficient query operations (insert, update, delete) on a row in a database table. Even though it shows the possibility of integrating ORAM with database systems, when it comes to column based query operations it requires to transfer all ORAM blocks to client-side which incurs huge additional overhead. Another possible approach is to allocate single ORAM block for each cell in the database table. However, on the down side, this approach increases size of the *position map* and it also increases the number of query requests when it requires operations on entire row or column. Recently in [24] Hoang et al. have investigated the application of special oblivious data structures along with ORAM to implement them on database services. This has motivated us to focus on integration of ORAM for cloud-based database systems to package it as a ready-to-deploy framework that can be used on existing database solutions.

4 Obliv-DB Framework

We propose Obliv-DB framework for database systems. Our solution follows the typical implementation of tree-based ORAMs in a distributed setup along with a special data structure that can efficiently manage/handle database operations securely. Fig. 5 shows a high level overview of Obliv-DB framework. We seek a solution that can be even deployable on memory limited clients, and that can be integrated into existing database solutions. Hence, Obliv-DB is an intermediary interface connecting client device and different server instances to perform secure database operations without revealing query access patterns.

We follow similar ideas proposed in [24], [25] by maintaining special data structure for row-based and column-based operations. However, unlike their design, our framework is based on a distributed ORAM as the underlying architecture which we believe more realizable toward the practical-end and our results outperform them. In addition, due to its underlying ORAM scheme [18], we noticed that the aforementioned solutions are practically incompetent to handle increasing realistic workloads. Thus, our intention is to explore the use of distributed ORAM in database systems; (i) to improve the overall performance by reducing the end-to-end delay using distributed approach (ii) to make an extensive solution that can be deployed even on memory/resource constrained clients without degrading the performance on increasing workloads (iii) to design a framework that can be conveniently pluggable into existing cloud-based environments/database services. More discussion on the implementation specific details is included in the section 6 of the manuscript. Further, the underlying ORAM of our solution is motivated by the ORAM structure proposed in [32] which utilizes Shamir secret sharing protocol to distribute data shares among servers, and we changed their structure according to the requirements of our solution geared toward the practical performance.

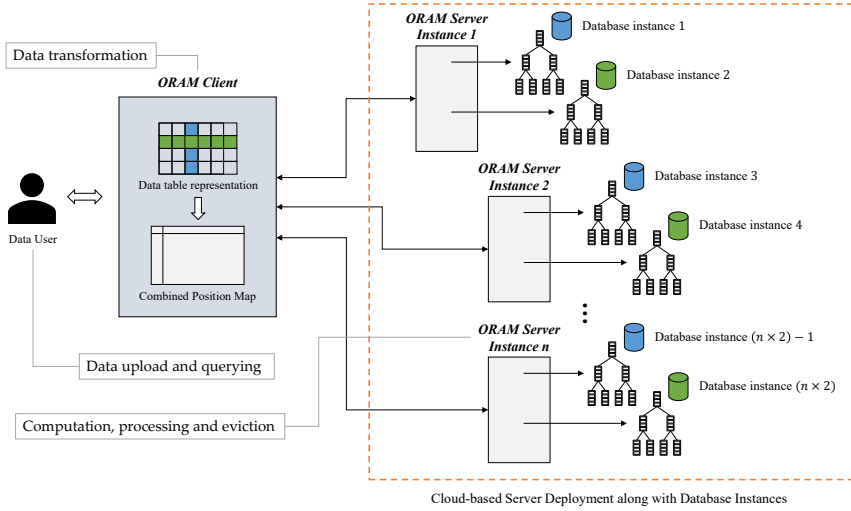


Fig. 5 The high level overview of Obliv-DB Framework.

4.1 Use of Shamir Secret Sharing for Distributed ORAMs

We now present the idea of using Shamir secret sharing [40] within the application context of ORAM proposed in [32]. The aim is to divide a secret into multiple shares where the same secret can be recovered later on by combining certain number of shares. The (t, l) -threshold scheme comprises of two basic algorithms; one algorithm to share the secret and other to recover. In order to create t -private shares of secret $\alpha \in \mathbb{F}_p$ among l parties, the client generates a random polynomial f where $f(0) = \alpha$. Then it evaluates $f(x_i)$ for each party P_i for $1 \leq i \leq l$, where x_i is a deterministic non-zero element of \mathbb{F}_p that uniquely identifies the party P_i . This $f(x_i)$ is called the share of the party P_i . In order to recover the secret α back, the shares of at least $t + 1$ parties have to be incorporated using Lagrange interpolation which is the recovery algorithm.

In the work of Hoang et al. [32], authors showed the use of additively and multiplicatively homomorphic properties of Shamir secret share scheme which can be used in a secure multi-party multiplication protocol. The idea is, given two values $\alpha_1, \alpha_2 \in \mathbb{F}_p$ shared by Shamir secret sharing, $2t + 1$ parties among l parties can compute the multiplication of α_1, α_2 without revealing the original value of α_1 and α_2 . In terms of accessing these shares from multiple servers, it is required to have secure mechanism that does not reveal the information which is being fetched. In general, multi-server Private Information Retrieval (PIR) techniques [41], [42] enables this functionality and the protocol usually consists of three basic algorithms to create the query, to retrieve the answers and finally to reconstruct the value. Let $b = (b_1, \dots, b_n)$ be a database formed of n items distributively stored in l servers. In order to retrieve an item b_i in b , the client creates set of queries (e_1, \dots, e_l) and distributes e_j to corresponding server S_j . These queries are typically designed in a way such that they do not

reveal any information about the identity of the item retrieving. The answer of each server a_j then be retrieved by the client. Upon receiving l answers, client computes the value of answer b_i using the reconstruction algorithm.

4.2 Oblivious Data Structure for Database Operations

Our implementation of the main data structure for oblivious access consists of multiple tree-based ORAMs. Suppose we need to outsource a table of M rows and N columns of privacy-critical data through Obliv-DB. At this point, system allocates two tree-based ORAMs, ROW-ORAM (which can store M data rows) for row-based operations while other, COLUMN-ORAM (store N data columns) to serve column-based operations. For example, each block in a bucket of ROW-ORAM represents a row of original table. The basic idea behind this representation is to efficiently access and process different types of row-based and column-based SQL operations, by minimizing the block accesses and round-trips for each query.

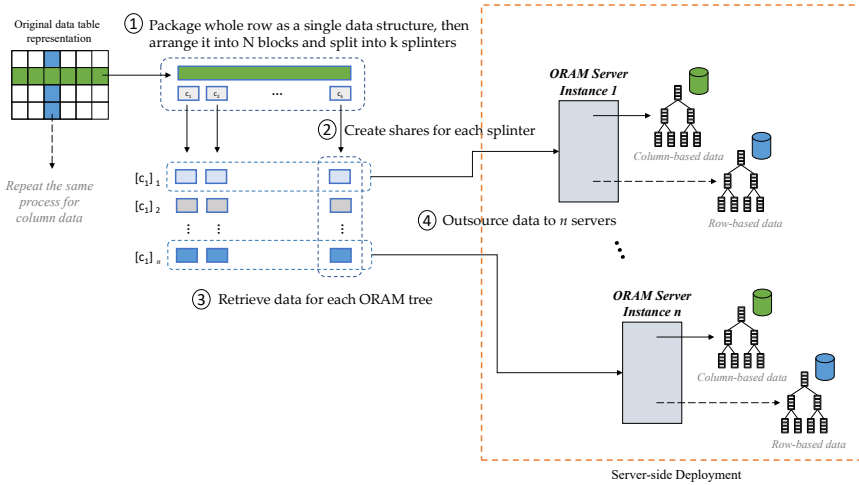


Fig. 6 The Obliv-DB data access procedure.

The server-side implementation of Obliv-DB framework is a typical binary tree which can store up to $N \leq A \cdot 2^{H-1}$ data blocks where H is the height of tree and A is the eviction frequency, described in the previous sections. As it contains two ORAMs to cater the demand for different row-based and column-based query operations, height of the tree usually changes with the size of the dataset. Each node in the tree represents a *bucket* in the ORAM where each bucket can store up to Z data blocks. The data blocks can be uniquely identified using the index i assigned to each *bucket* along with the *blockID* j assigned to each block within a bucket. We follow the tree-based architecture represented in Fig. 2, where $T[i, j]$ denotes the j^{th} block of i^{th} bucket. It

is noteworthy that size of the block varies according to the underlying data structure in terms of storing ROW-ORAM or COLUMN-ORAM data.

The position map of Obliv-DB contains position of the real data blocks represented using a data structure with three fields: *ublockID*, *pathID* and *pathIndex*. The *ublockID* has a unique ID for each real block while *pathID* ($2^H \leq \text{pathID} < 2^{H+1}$) carries the path along the tree assigned for each data block. And finally *pathIndex* ($1 \leq \text{pathIndex} \leq Z \cdot H + 1$) represents the location (index) of the block along that assigned path. Our Obliv-DB implementation requires two position maps (pm_{col}, pm_{row}) to hold each column and row in the original data representation. The basic construction of oblivious data structure is depicted in Fig. 6 and presented in Algorithm 1 as described below.

Algorithm 1 Obliv-DB Initialization

```

1: procedure GENERATETREE(dataTable, dim)
2:    $N \leftarrow 0$ 
3:   if dim.oram=COL then
4:      $N \leftarrow \text{dim}.N_{COL}$ 
5:   else
6:      $N \leftarrow \text{dim}.N_{ROW}$ 
7:   end if
8:   Arrange DataTable into blocks  $b = (b_1, \dots, b_N)$ 
9:    $T[i, j] \leftarrow \{0\}$  ▷ initialize all blocks with 0
10:  for each  $i \in 1, \dots, N$  do
11:     $x_i \leftarrow x_i \in \text{UniformRandom}(2^H, 2^{H+1})$ 
12:    select  $T[x_i, y_i]$  where  $(T[x_i, y_i] == \text{empty})$ 
13:     $T[x_i, y_i] \leftarrow b_i$  ▷ assign block to a random leaf
14:     $\text{pathID} \leftarrow x_i$ 
15:     $\text{pathIndex} \leftarrow (\lfloor \log_2 x_i \rfloor \cdot Z + y_i)$ 
16:     $pm_{\text{dim}}[uBlockID_i] \leftarrow (\text{pathID}, \text{pathIndex})$ 
17:  end for
18:  for each  $i \in 1, \dots, 2^{H+1} - 1$  do
19:    for each  $j \in 1, \dots, Z$  do
20:       $c_{i,j}^1, \dots, c_{i,j}^k \leftarrow T[i, j]$  ▷ split into k splinters
21:      for each  $c \in 1, \dots, k$  do
22:         $([c_{i,j}^c]_1, \dots, [c_{i,j}^c]_n) \leftarrow \text{GetShares}(c_{i,j}^c)$ 
23:      end for
24:       $[T[i, j]]_l \leftarrow ([c_{i,j}^1]_l, \dots, [c_{i,j}^k]_l) \text{ where } 1 \leq l \leq n$ 
25:    end for
26:  end for
27:  return  $([T]_1, \dots, [T]_n)$  ▷  $[T]_i$  goes to  $\text{Server}_i$ 
28: end procedure

```

Upon given the original dataset and the dimensions (whether ROW or COLUMN ORAM), Obliv-DB initialization protocol generates multiple shares of trees to be outsourced for n number of servers. The idea is, say for ROW-ORAM, first we rearrange all columns in a row as one data structure. Similarly, if it is COLUMN-ORAM, we package all rows in a column into a single data structure. Thereafter, this data structure is then split into N multiple blocks of B -bit by assigning unique $uBlockID$ for each. The client's intention here is to assign each block into a random leaf bucket while assigning 0 (dummy) for rest of the buckets in the tree. Instead of assigning the B -bit block directly to the leaf, in Obliv-DB we first generate shares of each block which are going to be distributed among n servers. In practice, size of this B -bit block can be varied with the usage and it might not be acceptable enough for computations at the server-side, especially with large volumes of data. Hence, each B -bit block is split into k equal-size splinters and client generate n shares for each splinter making n distributed ORAM trees. Meanwhile, client updates two position maps (pm_{col}, pm_{row}) according to the assigned $pathID$ and $pathIndex$. A high level view of the initialization protocol in steps is depicted in Fig. 6. It was also noted that many ORAM schemes proposed over the past support only storing blocks of the same size, and when there are variable size blocks, they get filled up with dummy data to make them equal size. However, with our design, variable block sizes also can be handled more efficiently compared to previous variable block size ORAM schemes as discussed in Sec. 4.6.

4.3 Data Access Scheme

The data access scheme of Obliv-DB operates similar to the previous schemes like Onion ORAM [20] which utilizes a select vector, and it also leverage the secure multi-party multiplication and private information retrieval protocol presented in [32] which follows the original presentation in [43], [44]. Each server S_i has a share of original data table containing $x = Z \cdot H + 1$ data items. Assume client requires to retrieve a specific block D_i from the tree. At this point, client first picks up the corresponding position map (based on the query whether it is row operation or column operation) and obtains the exact location of the tree ($pathIndex$) along with the $pathID$. Suppose j is the exact location to be retrieved. Now client creates a x -dimensional select vector \vec{v} according to the block location in the corresponding path where all items in the vector is 0 except the j^{th} position being set to 1. Thereafter, it generates secret shares for this select vector and distributes each share \vec{v}_i among all servers along with the same $pathID$. Note that, a share of vector is also a vector in which the corresponding components are the shares of the components in original vector \vec{v} .

Upon receiving, each server calculates the inner product of \vec{v}_i and its share in the tree along the path defined in $pathID$ and sends back to the client. It is noteworthy that, each location in the tree contains k separate splinters; thus, server needs to calculate the inner product for each splinter and send them to the client as a bundle. Finally, client accumulates all answers from n

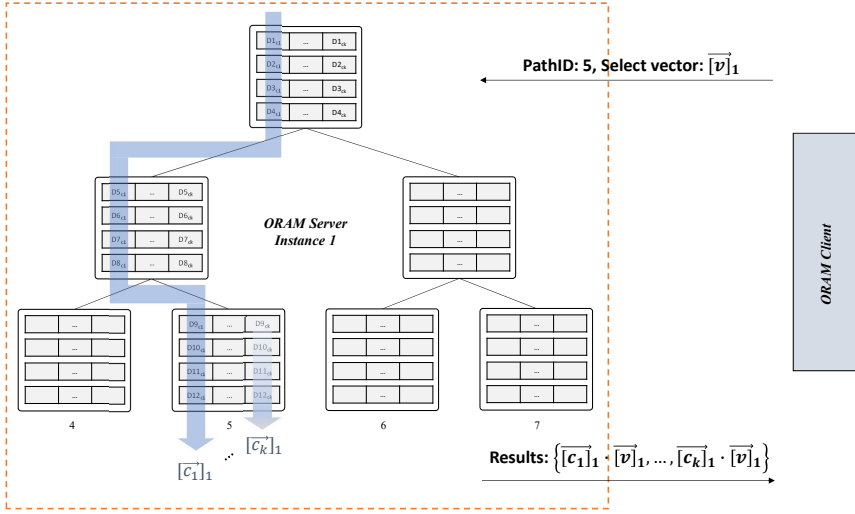


Fig. 7 The Obliv-DB data retrieval.

distributed servers to recover the desired data block. This data access scheme is presented in Fig. 7 in a simpler form and described in the Algorithm 2.

4.4 Data Eviction

Once the data block is retrieved and recovered from distributed servers, it is then required to perform the eviction operation after having A (a parameter originally proposed in [30]) successive retrievals. The Obliv-DB's eviction operation is inspired by the 'triplet eviction strategy' proposed in [20] and it follows a deterministic eviction path according to the reverse lexicographic order (refer Fig. 3). This eviction order can ensure that the nodes in two eviction paths performed consecutively will not overlap; hence majority of the splinters will not shuffle repeatedly. The idea is, in each eviction operation, data blocks of each parent bucket moved to one of its children such that these data blocks still remains in the same eviction path. We followed a similar idea proposed in 'SSS-SMP-based triplet eviction' in [32] to make it realizable in distributed setup. However, an additional proxy server is introduced in our design in terms of improving the time cost spent for eviction operations while obviously moving data from source bucket to child bucket(s).

When moving data from parent bucket to child, the client first generate a representative share vector (of Z -dimensional) for all data in each bucket. Then these two vectors (parent and child) are concatenated by forming a $2Z$ -dimensional single share vector \vec{u} . Thereafter, a permutation matrix $I \in \{0, 1\}^{2Z \times Z}$ is generated in a way such that the product of \vec{u} and I becomes Z -dimensional vector \vec{v} . In simpler terms the objective is to move all data in the parent section of the vector \vec{u} to the child data vector \vec{v} as depicted in Fig. 8, without changing the location of any existing data in \vec{v} . It is also noted that,

Algorithm 2 Obliv-DB Data Access Scheme

```

1: procedure DATAACCESS(id, dim)
2:   Client:
3:   Set  $e := (e_1, \dots, e_x)$  where  $e_j \leftarrow 1, e_i \leftarrow 0$  for all  $1 \leq i \neq j \leq x$ 
4:   for  $i = 1, \dots, x$  do ▷ generate select vector
5:      $([e_i]_1^{(t)}, \dots, [e_i]_n^{(t)}) \leftarrow \text{generateShares}(e_i, t)$ 
6:   end for
7:    $[e]_i^{(t)} := ([e_1]_i^{(t)}, \dots, [e_x]_i^{(t)})$ , for  $1 \leq i \leq n$ 
8:   Send  $(pm_{dim}.pathID, [e]_i^{(t)})$  to  $Server_i$  ▷ goes to  $Server_i$ 
9:   Server:
10:  Each  $Server_i$  with  $(pathID, [e]_i^{(t)})$ 
11:  for  $j = 1, \dots, k$  do
12:    Let  $[c_j]_i^{(t)}$  represents  $j$ -th splinter of  $Z$  slots along the  $pathID$ 
13:     $[c_j]_i^{(2t)} \leftarrow [e]_i^{(t)} \cdot [c_j]_i^{(t)}$  ▷ server calculates the inner product
14:  end for
15:  Send  $[[c_1]_i^{(2t)}, \dots, [c_k]_i^{(2t)}]$  to client
16:  Client:
17:  On receive from servers  $(\{[c_1]_i^{(2t)}\}_{i=1}^n, \dots, \{[c_k]_i^{(2t)}\}_{i=1}^n)$  ▷ receives share
    from each server
18:   $c_j \leftarrow \text{recoverShares}([c_j]_1^{(2t)}, \dots, [c_j]_n^{(2t)}, 2t)$ 
19:   $data \leftarrow (c_1, \dots, c_n)$ 
20:  return  $data$ 
21: end procedure

```

exact location of real data must be hidden even after the permutation. Thus when generating I , client actually generates secret shares for every element of I making them a share matrix. Finally, when executing the matrix product of two shares (in \vec{u} and I), in our design, servers also need to communicate with each other to recover the value. This is where the 'Proxy Server' comes in to the picture handling these server-server communications efficiently. It is noteworthy that the proxy server doesn't necessarily need to be trusted as the generation of secret shares happens at the client-side. The following section 4.5 describes the execution of matrix product of two shares collaboratively at the server-side.

4.5 Use of Secure Multi-party Multiplication in Secret Shared Values

Secure multi-party multiplication (SMP) protocols are in the discussion for a long time. Among different SMP schemes, most practical protocols in threshold-cryptography are being based on Gennaro et al.'s idea as proposed in [45]. Assuming (t, n) -Shamir sharing of two secrets a and b , the protocol allows to securely execute (t, n) -Shamir sharing of product $a \cdot b$ without revealing the values of a and b . In our implementations, we consider a similar idea

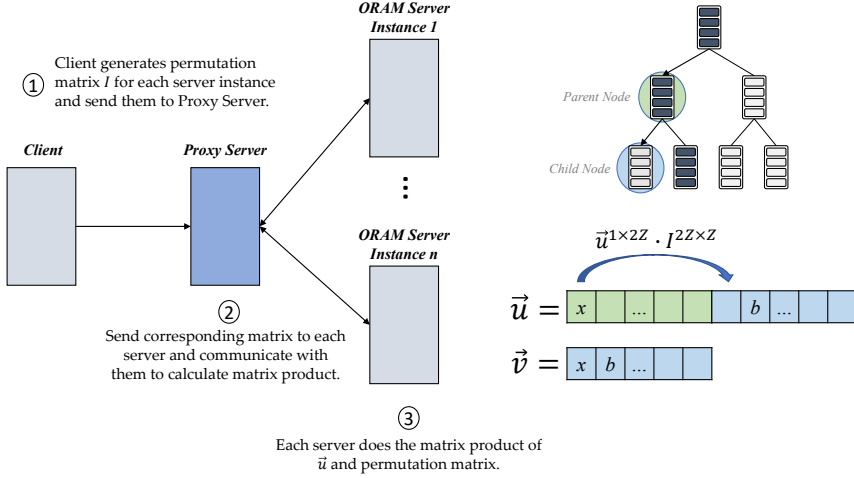


Fig. 8 The Obliv-DB data eviction operation.

proposed in [46] which uses one communication round with $O(n)$ bandwidth and helps us to eventually reduce the time cost spent on eviction operation. The protocol is summarized as follows.

Let K be a finite field where $K < n$ and x_1, \dots, x_n be non-zero elements of K while n is the number of parties. Each party P_i has a share of a (i.e.: a_i) and share of b (i.e.: b_i) and would like to compute the multiplication. Let $f(0) = a$ and $g(0) = b$ where f and g are polynomials of degree at most t such that $f(x_i) = a_i$ and $g(x_i) = b_i$ for all i . The resultant product now has a degree at most $2t$ (where $2t < n$). However, there exists a linear map $\phi : K^n \rightarrow K$ such that the result of $\phi(a_1 \cdot b_1, \dots, a_n \cdot b_n)$ is always ab which can be computed locally by the parties itself and can be uniquely determined by Lagrange interpolation. Therefore, first each party P_i locally multiplies a_i and b_i and creates shares of $a_i \cdot b_i$ (input sharing) by using a new random polynomial h_i of degree t . Thereafter, each party locally computes the inner product of received shares and $\phi(h_1(x_j), \dots, h_n(x_j))$ to obtain new shares of $a_i \cdot b_i$ which now can be represented as degree of t .

This process simplifies the secure multiplication into secure linear computation and hence it can be easily adopted into distributed environment in our design. We employed a dedicated proxy server to handle inter-server communications along with the secret sharing degree reduction process.

4.6 Use of Variable Block Size

A greater number of ORAM constructions are primarily designed to provide a generic solution in terms of protecting underlying data against access pattern leakages. Therefore, even most recent ORAM designs have overlooked the importance of block-size of data items present in the ORAM structure. Hence, typical ORAM designs only support data blocks of the same size and when

the block size varies they usually pad them up to a fixed size. However, for database operations, block size is a crucial factor that leads to overflow in storage space and/or network bandwidth, and eventually the performance degrade. Therefore, in our Obliv-DB design we tried to incorporate the variable-size data blocks in the ORAM structure without using trivial padding which makes our solution more practical and reasonable for a cloud-based database system. In 2016, Roche et al. [47] presented a first of its kind variable-sized storage block structure (vORAM) for ORAMs which requires each bucket to be split into two areas to store encryption keys and the data. Recently, the DivORAM architecture [48] is proposed, relatively similar to our design; however instead of directly using additively homomorphic encryption scheme, we take advantage of the homomorphic properties of Shamir secret sharing which makes our design more robust in the database landscape.

As we described the oblivious data structure in Sec. 4.2, each B -bit data block is divided into k equal-size splinters. This B -bit data block can be varied in size but our intention is to split this block into several splinters of the same size (s) in a way that it is no longer required to pad the block with dummy data. Then, different shares of the same splinter will spread among multiple servers on the same path. However, in a case of smaller block size (less than s) it is padded to s to maintain the security guarantees. During the implementation we change the client-side parameter of NO_OF_SPLINTERS along with additional meta-data to incorporate variable length block size in our solution. The size of B -bit data block is usually bounded by how ORAM data is packaged. For example, if it is ROW-ORAM, the size of the data block becomes size of the column. However, as discussed in the section 4.1, this block size cannot be any arbitrary size beyond $\lceil \log_2 p \rceil$ otherwise it will not be acceptable for the computation over \mathbb{F}_p . Thus, when generating equal size splinters, we assume that we select an appropriate prime p in a way such that every splinter is less than p , when interpreted as an element of \mathbb{F}_p .

5 Threat Model and Security Analysis

In this section, the security of the Obliv-DB framework is reviewed with the emphasis given to the threat model and the potential attacks in the multi-server setting.

5.1 Security Definition

Based on the literature, there are two ORAM security definitions exist [49]. The first one is the general ORAM security defined by most of the ORAM schemes in terms of distinguishing two consecutive access sequences of the same length. The second is defined as distinguishing two access sequences of arbitrary length operations proposed by Haider et al. [50] where the same access pattern can have completely different memory access sequences when it comes to implementations based on secure hardware architectures. However, in this paper, we consider the protection against access pattern attacks

while fetching data from cloud-based non-colluding multi-server setting, as for the case of database operations. Thus, by extending the security definition of prior ORAM designs [33], [18], [32], we define the security of Obliv-DB under distributed ORAM setting as follows.

Definition 1. Let $\vec{x} = ((op_1, id_1, Data_1), (op_2, id_2, Data_2), \dots)$ be private data access sequence of length q , where op can be either *read* or *write* request while id is the unique identifier of *data* to be read or written. Let λ be the security parameter, and $A(\vec{x})$ denote the sequence of message exchanges with the server S_i given a data request sequence \vec{x} .

Definition 2. A distributed multi-server ORAM is secure if (1) for any two data access sequences \vec{x}, \vec{y} of same length and their corresponding message access sequences $A(\vec{x}), A(\vec{y})$ are statistically and computationally indistinguishable for anyone but the client, (2) the ORAM construction is correct where it returns correct results for any access sequence \vec{x} with probability $\leq 1 - \text{negl}(|\vec{x}|)$.

5.2 Threat Model

In the Obliv-DB framework, the cloud-based servers are assumed to be honest-but-curious (semi-honest) in which each server performs their operations as per the instructions of the underlying ORAM scheme, but curious about the data being exchanged. In addition, we also assume that under the distributed setting cloud servers are non-colluding and different cloud operators do not share data with each other. It is understood that in most of the practical deployments, this assumption is realistic as the cloud service providers compete with each other. As such, Obliv-DB can handle up to t colluding servers according to the privacy level setting of (t, l) -threshold mechanism discussed in the section 4.1.

Attacks out of scope. As mentioned, we consider Obliv-DB's threat model to be semi-honest. Hence, any attack that breaks or tampering the protocols is out of scope. In addition, in the Obliv-DB framework a malicious cloud attacker usually has access to three types of timing information and may leak data: 1) time of arrival of client request, 2) time of inter-server message exchange, 3) time of response sent to the client. However, in the current version of the manuscript, we do not consider these timing attacks as well.

5.3 Security Analysis

In order to satisfy the security of the Obliv-DB framework, each data access operation including the ORAM selection process, query path selection, block access and evict operation should ensure that it leaks no information about access pattern. Accordingly, we emphasize the following points.

- a) *Row or column based data structure does not leak any additional information other than the type of data structure being accessed and the size of row/column.*

Let D represents the data structure where $D = 0$ if the operation being performed is on ROW-ORAM with size M , and $D = 1$ if it is on COLUMN-ORAM with size N . The ROW-ORAM and COLUMN-ORAM data structures in Obliv-DB are introduced to serve different row-based and column-based operations as described in Sec. 4.2 and operates independently. Hence, due to the obliviousness of the actual implementation of underlying ORAM primitive which satisfies Definition 1 and 2, a construction with row or column based data structure leaks no information about the particular node being accessed, other than D and the dimensions M, N . Therefore, as long as the query access sequences of each data structure are indistinguishable from each other, these data structures leak no additional information.

It is also noteworthy that as an additional security measure the access information whether it is a ROW-ORAM or COLUMN-ORAM can be hidden by performing a simultaneous access for each operation. On the other hand, the dimensions of ROW-ORAM and COLUMN-ORAM can also be hidden by configuring equal dimensions so that it is hard to distinguish. However, it has to be evaluated mostly case by case within the security-performance spectrum, as it may adversely impact the performance of some applications.

- b) *Query path selection and block access is oblivious.*

The path assigned to each block is decided uniformly at random and independent to each other. Once a specific block is accessed, it is again re-assigned to a new path selected randomly which is independent to query access pattern of client. Moreover, each individual block consists of k splinters spread across multiple servers thus the probability that an adversary can distinguish which splinter is being accessed is very low [32]. For instance, during the block access, each server receives a share of x -dimensional select vector \vec{v}_i . Given a data access sequence, let $\vec{b} = (b_1, b_2, \dots, b_N)$, where b_i is the individual block and let $s(b_i)$ denotes the splinters belongs to block b_i (i.e: c_1, c_2, \dots, c_k). The probability that an adversary can distinguish which splinter is being accessed in a path is

$$Pr[s(b_i)] = \frac{1}{(Z \cdot (H + 1))^k}, i \in \{1, 2, \dots, N\}$$

where Z is the number of slots in a bucket and H is the height of the tree. Therefore, block access along with query path selection in Obliv-DB is oblivious.

- c) *Evict operation leak no information.*

For each eviction process the eviction path it involves is deterministic (Obliv-DB eviction is based on triplet eviction [19]) and is independent

of query access pattern. During each eviction operation all data blocks on the path are re-encrypted and shuffled by all servers (with the help of proxy server) and the same number of blocks are written back. Hence, evict operation is random and independent to query access pattern and thereby does not leak any information about where the original data is now stored or along which path they are stored.

- d) *Any polynomial-time adversary cannot distinguish given two sequences.*

Even though the eviction operation happens in every A times, the obliviousness of query access pattern is still guaranteed by the block access operation as stated above. Thus, a polynomial-time adversary cannot distinguish given two access sequences.

6 Performance Evaluation

We implemented our Obliv-DB framework on a production-ready environment to evaluate its performance with a realistic workload with different configuration settings. We then compared the results of Obliv-DB framework with S3 ORAM and Path ORAM's implementation on CURIOUS framework [22], [24] with some additional integrations/changes to accommodate row and column based query operations.

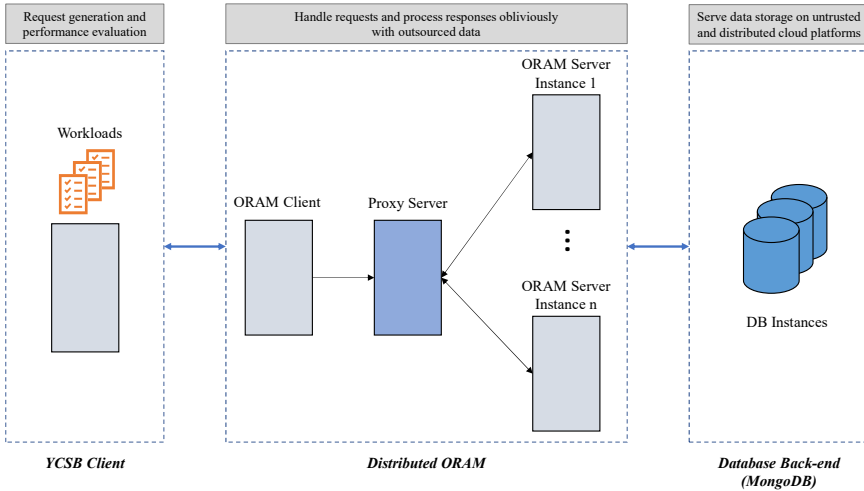


Fig. 9 A summary of experimental deployment setup of Obliv-DB.

6.1 Implementation, Experimental Setup and Configuration

We implemented Obliv-DB using C++ (GCC version 7.4.0) with the assistance of abstract class implementations in S³ORAM. We integrated additional

functionalities and performance metrics to the code to facilitate different row-based and column-based query operations. For low level thread safe modular multiplications and inner product calculations we used NTL C++ library (version 11.3.2). Moreover, for the implementation of asynchronous client-server and server-server communications we used ZeroMQ library (version 4.2.5) in the distributed setup. In terms of handling large numbers beyond the range of standard data types we used the Boost libraries. MongoDB is chose as the database engine for our implementation and we built the MongoDB CXX driver (version 3.4.0) and configured to work with our Obliv-DB framework in a parallel and distributed setup. We deployed several MongoDB instances of version 4.0 to facilitate back-end database for each ORAM server instance. It is important to note that, as per the design of Obliv-DB framework, it can be extended to apply into many forms of data models including NoSQL and relational (RDBMS). However, as NoSQL databases are rising in popularity, in the current version of the experiments, we only considered evaluating Obliv-DB and its counterparts with document oriented NoSQL model with MongoDB.

A summary of experimental deployment setup is depicted in Fig. 9. As shown, Yahoo! Cloud Serving Benchmark (YCSB) [26] is utilized in this work in terms of the performance evaluation. We have customized the YCSB 0.17.0 in a way to process queries through distributed Obliv-DB framework to obtain the performance measurements. In terms of executing a workload, YCSB has two phases: loading phase and transactions phase. In order to generate the performance statistics, YCSB is using this transactions phase which defines the operations to be executed against the data set. At the end of the execution of the workload, YCSB reports a series of performance statistics including average, min, max, 95th and 99th percentile latency for each operation type (read, update, etc.), a count of the return codes for each operation, and a histogram of latencies for each operation, and so on. We deployed our implementation of Obliv-DB framework on series of Virtual Machines (VMs) configured on a high-end server machine equipped with two Intel Xeon Gold 6148 CPUs, 128 GB of RAM running on Ubuntu 20.04. Each ORAM Server is deployed on a separate VM consists of 12 CPU cores and 8GB of RAM with specific resource allocations managed by VMware Workstation 15. To make it a production-ready realistic environment, inter VM communication has been divided into three separate virtual networks. Moreover, in the virtual network setup we have configured inter VLAN bandwidth as 100Mbps (for both incoming transfers and outgoing transfers) considering a moderate speed cable/leased line with an added average latency of 25ms. The high level virtual network configuration is depicted in the Fig. 10. To manage network level name resolutions, we employed a separate Domain Name Server (DNS) running on top of another Ubuntu 20.04 instance. For all our experiments, we used 7 ORAM server instances (i.e: l) and 3 as the Shamir Secret Sharing threshold (i.e: t).

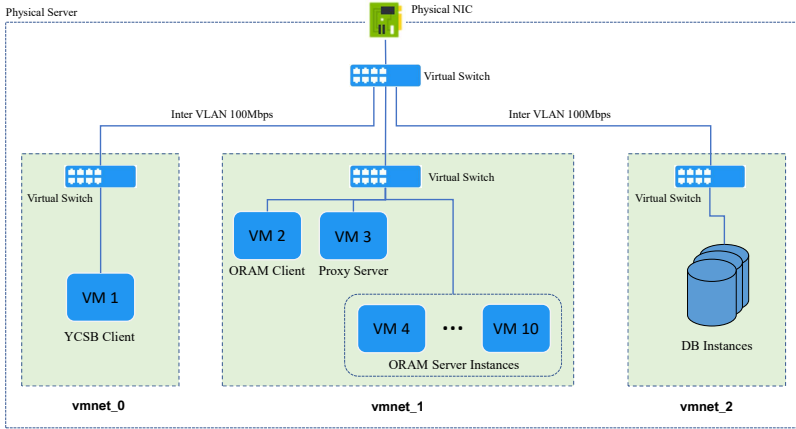


Fig. 10 The high-level virtual network configuration setup.

6.2 Evaluation Metrics and Experimental Results

We compared the performance of Obliv-DB based on multiple different metrics. At first, we looked into the ORAM preparation time. Even though ORAM preparation is a one-time cost during the stage of database setup/migration, it can affect the performance of the overall solution in a production environment. Then we looked into the query response time (the time taken for a query to complete a transaction) of different row/column oriented queries. Finally, we measured the throughput (the average number of operations executed per second) of the proposed scheme over increasing workloads. For a cloud-based database system, the response time and the throughput are highly crucial factors. Thus we compared the response time and throughput of the proposed scheme with its counterparts by including all components such as transmission delay during the client-server communication, decryption of data at client-side and re-encryption. In this case, the YCSB measurements were utilized to capture the actual duration at the point of client sends the first command/query until it receives the final result from the servers. From the throughput distribution, we analyzed how Obliv-DB behaves for an increasing workload, as it is an important factor for a database system. It is also noteworthy that for each of these experiments, we conducted the experiment for 10 times and the average number is reported in the results. We selected Path ORAM (implemented on CURIOUS framework) and S3 ORAM as the main counterparts for Obliv-DB. It is also noteworthy that there are a few (and recent) ORAM implementations discussing oblivious query processing in the distributed setting such as [4], [51] which consider similar deployment of Obliv-DB. However, these solutions are mainly rely on hardware enclaves such as Intel SGX while we are seeking for a solution that can be adopted into the existing environments without the use of hardware enclaves.

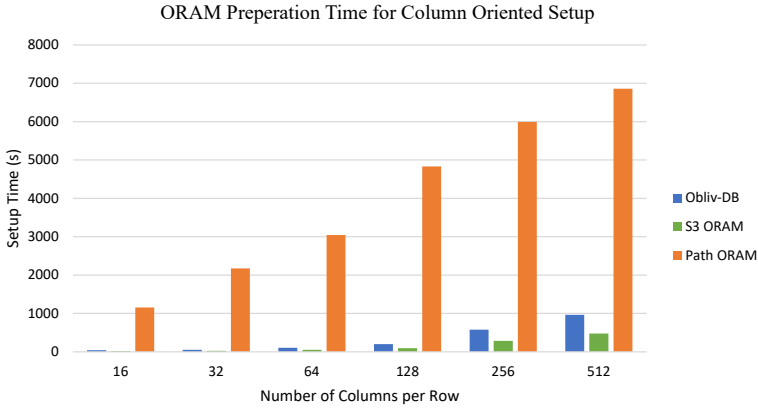


Fig. 11 The ORAM preparation time for column oriented setup.

6.2.1 Database Preparation Time

We first analyzed the database preparation time for underlying ORAM scheme. When a database system is migrating from on-premise to cloud along with ORAM primitives, it is impractical if it takes very large time just to construct a reasonable sized database. Thus, we looked into the database construction time for Obliv-DB and its counterparts as depicted in Fig. 11. For a database consists of 32.7K rows of data where each row contains 512 of columnar data (each of block size 4096), our Obliv-DB is $14.3\times$ faster than Path ORAM during the initial phase of database construction. However, since the underlying data structure of Obliv-DB contains both row data and column data as independent ORAMs, Obliv-DB is approximately $2\times$ slower than S3 ORAM during the initial preparation stage.

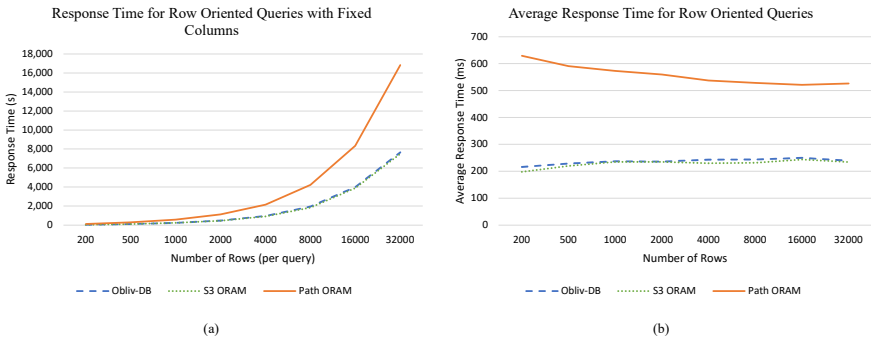


Fig. 12 The response time comparison for row oriented queries (varying row sizes) (a) total response time, and (b) average response time.

6.2.2 Query Response Time for Different Query Types

Next, the query response time is considered for the evaluation. In this case, we first measured the performance of Obliv-DB for row-based operations where client can execute SQL operations for a specific or set of rows. During this experiment we changed the number of rows from 128 to 32.7K while the number of columns fixed at 32. The results of the experiment is depicted in Fig. 12. Obliv-DB's response time for row-based operations is more saturated around 200ms while Path ORAM is $2.5\times$ slower as shown in the Fig. 12(b). However, when it compares with the S3 ORAM it can be seen that both S3 ORAM and Obliv-DB performs almost similar for row-based operations even though Obliv-DB's underlying data structure is different.

In addition, we have observed that Path ORAM's average response time is getting slightly lower with the increasing number of rows until 8000 rows and then saturated around 520ms. This can be mainly attributed to the fact that path ORAM has shuffling operation after every read which involves a lot of communication back and forth. Thus, each time when a row is accessed, the response time can be varied in a range of 200ms variance (This can vary in different network settings. As mentioned before, we are using a network setting of 100Mbps for both incoming transfers and outgoing transfers considering a moderate speed cable/leased line with an added average latency of 25ms). Therefore, when the values are averaged over large number of rows (for example 32K vs 200) this type of deviations are observable.

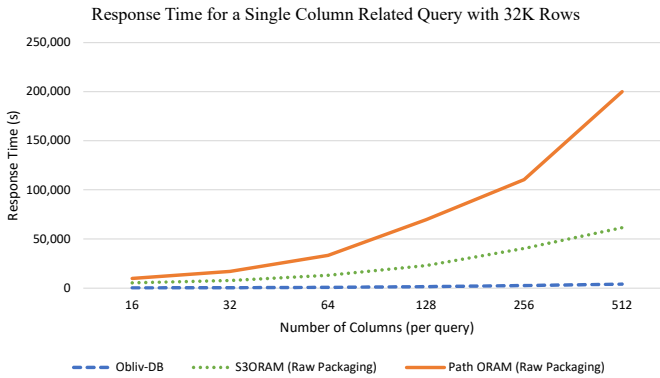


Fig. 13 The response time comparison for a column oriented query with varying column sizes.

We next looked into the column oriented operations to measure how quickly client can execute a column related operation such as statistical or conditional query (e.g. `SELECT AVG(net.salary) from Employees`). As such, the Fig. 13 presents the response time for different type of column-oriented queries, by varying column sizes from 16 to 512 while the number of rows is fixed at 32.7K.

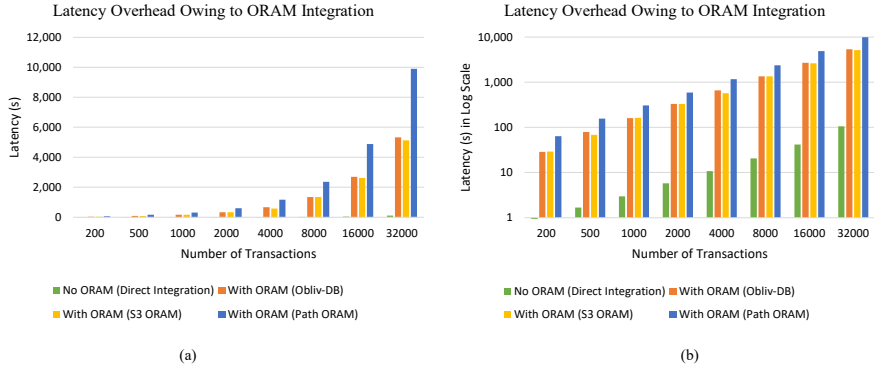


Fig. 14 Comparison of latency overhead with and without ORAM. (a) latency overhead in linear scale, and (b) same in log scale to emphasize the difference with and without ORAM primitives.

In order to facilitate varying number of columns at client, YCSB configuration parameters (FIELD_COUNT_PROPERTY, FIELD_LENGTH_PROPERTY) were changed accordingly. In addition, to accommodate variable column size for the ORAMs, we changed the block size of ORAM to different values in Obliv-DB, S3 ORAM and Path ORAM accordingly. The results emphasize that our Obliv-DB outperforms both S3 ORAM and Path ORAM, and it is significant even with the increasing number of columns in the database.

6.2.3 Latency Overhead

Thirdly, we explored the latency overhead owing to ORAM integration. In this case, the average latency was measured with and without using ORAM primitives, for an increasing workload as shown in the Fig. 14. It was noticed that whenever the security primitives are enabled, all ORAM designs follow a similar pattern in latency overhead; however, on average the Path ORAM has a latency overhead as twice as Obliv-DB, while S3 ORAM and Obliv-DB have similar properties. Moreover, in the case of a baseline with direct integration (without using ORAM), Obliv-DB has an average latency overhead of $54\times$ whereas Path ORAM shows an additional $101\times$ average latency overhead.

6.2.4 Overall Throughput Comparison

It is also important to look at how database system reacts in terms of throughput against increasing workloads especially when the security primitives are enabled. Thus, we measured the overall throughput of Obliv-DB and its counterparts for varying column sizes. In YCSB, typically we need to use the *-threads* and *-target* parameters to control the amount of offered load. For example, we can set 10 threads attempting a total of 100 operations per second (10 ops/sec per thread). The key point is, as long as the average latency

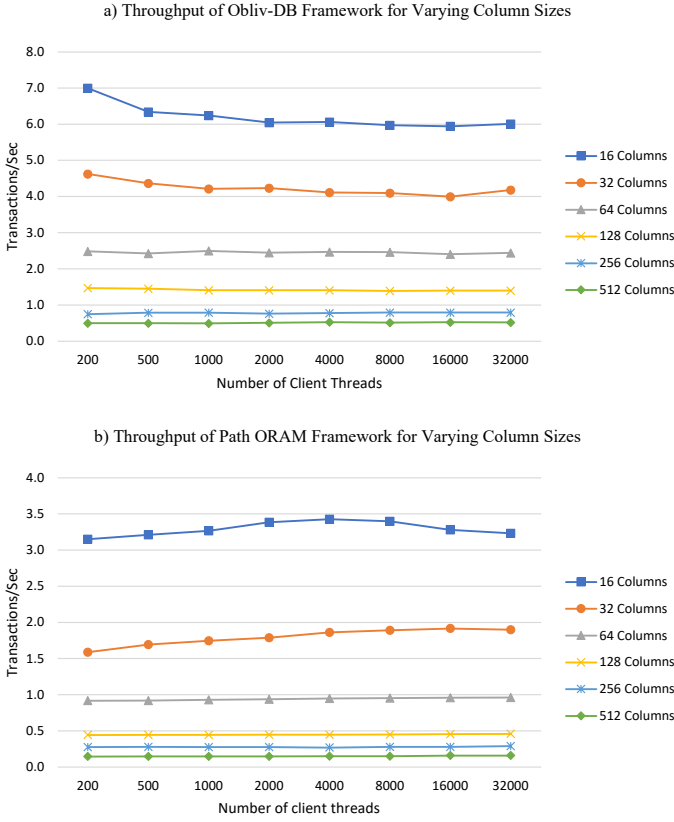


Fig. 15 Throughput of distributed Obliv-DB framework vs Path ORAM for increasing workloads.

of operations is not above 100 ms, each thread will be able to carry out its intended 10 operations per second. In general, it is required to have enough threads so that no thread is attempting more operations per second than is possible [52]. As such, the number of client threads in YCSB were changed from 200 to 32,000 (arbitrary queries with a balanced workload) and the results are depicted in Fig. 15. At an average, Obliv-DB's throughput is $2.4\times$ higher than that of Path ORAM and it concludes that Obliv-DB framework in distributed setup can handle higher number of transactions per second than Path ORAM's implementation. It is also noteworthy that because of Obliv-DB's underlying architecture, Obliv-DB demonstrated better throughput compared to S3 ORAM's implementation as shown in the Fig. 17.

Moreover, it is also noteworthy that in this work we consider each ORAM transaction as a single operation (read/write). As we mentioned earlier, we do not change the underlying database but ORAM primitives work as an intermediate facilitator to enable the security primitives. In order to handle concurrent

bucket accesses in ORAM we use a simple shared locking mechanism such that the second transaction is in wait till the first is over.

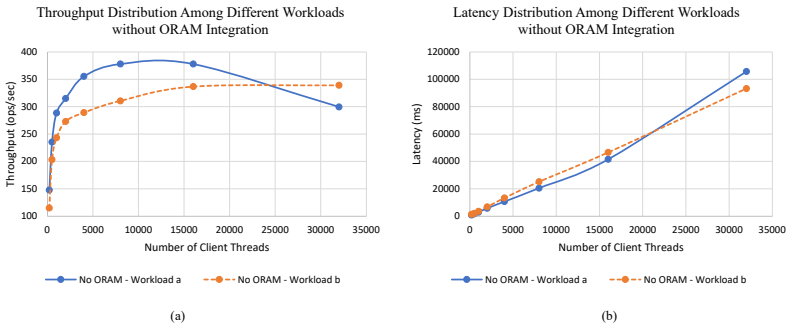


Fig. 16 Throughput and latency distribution among different workloads when ORAM is not enabled.

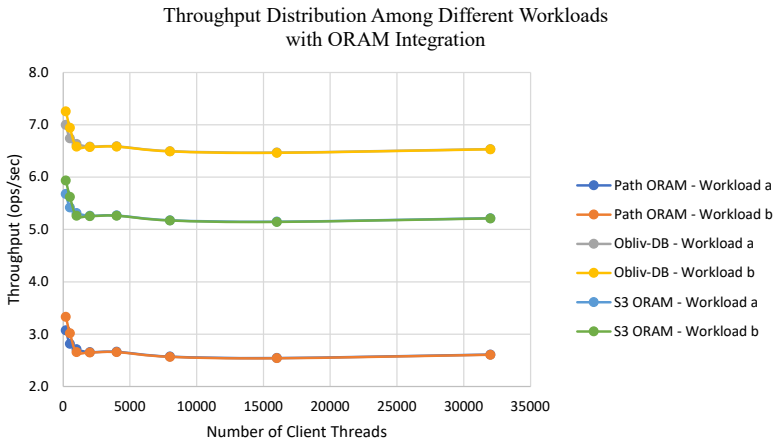


Fig. 17 Throughput distribution among different workloads when ORAM is enabled.

6.2.5 Use of Different Workloads

Next, we looked into the impact for throughput when having different workloads. For this experiment, two different workloads were defined where workload-a configured with 50% read (read, scan) operations and 50% write (insert, update) operations, and workload-b with 95% read operations and 5% write operations. At a high level, workload-a is a balanced workload where as workload-b can be defined as a read mostly workload. Accordingly,

the YCSB client was configured to generate queries for different workloads with Zipfian distribution and the throughput was measured against increasing client threads. The Fig. 16 shows the throughput distribution among different workloads when ORAM is not enabled.

As illustrated in the Fig. 16, there is a clear difference between the results of two workloads where, on average, workload-a produces a higher throughput while workload-b's throughput is considerably lower attributing to higher number of read operations. As database read operations have higher latency than write operations in general, it is expected to notice a drop in overall throughput whenever running a workload with high read portion. In addition, it can be seen a performance degrade in workload-a after exceeding 15000 clients. This could be mainly due to the impact of latency (as it can be seen from Fig. 16 (b)) on account of the resource limitations of the allocated virtual machine as workload-a has more write operations compared to workload-b.

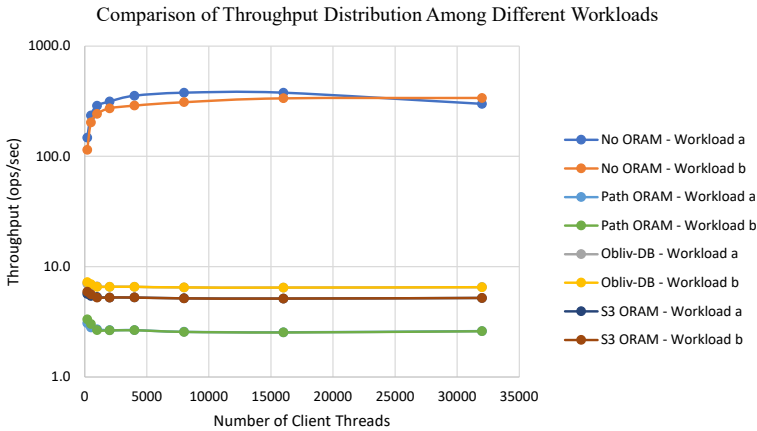


Fig. 18 Throughput distribution of different workloads for with and without ORAM schemes.

However, whenever the security is enabled (with ORAM), as shown in the Fig. 17, it is very clear that the workloads can not be distinguished as both workload-a and workload-b follow a similar pattern. This graph explains the importance of having ORAM as the underlying security primitive for the database operations. Moreover, it is noteworthy that an adversary can not distinguish which kind of operation it is or whether it is a read or write operation, not just looking at the operations even by looking at the throughput of the all operations.

Finally, we looked into the overall throughput overhead due to the use of ORAM as the security primitive. This can be used to have an idea about how much of an additional overhead accompanied by the ORAM schemes. The Fig. 18 explains the throughput distribution of different workloads when deployed

with and without ORAM schemes. As it can be seen, in overall Obliv-DB's throughput is $42\times$ slower than that of direct integration (no ORAM) whereas the throughput of Path ORAM is almost $103\times$ slower, and throughput of S3 ORAM is $53\times$ slower than the direct integration.

7 Conclusion

This paper takes the initiative toward implementing a secure and practical data processing framework called Obliv-DB for cloud-based database systems by using a distributed oblivious scheme and specially formulated data structures. Usually direct integration of ORAM into database systems severely affect the overall performance of the underlying data operations. However, we explored the potential of ORAM in the distributed context along with state-of-the-art horizontal-scaling capabilities of modern database engines to reduce the cost of direct integration of ORAM in the database landscape. We implemented our framework on a practical database setup with Yahoo! cloud serving benchmark and performed series of experiments on a realistic production-ready environment. The experiment results showed that Obliv-DB has significant performance improvements over direct integration of ORAM with database systems.

References

- [1] Samaraweera, G.D., Chang, M.J.: Security and privacy implications on database systems in big data era: A survey. *IEEE Transactions on Knowledge and Data Engineering* (2019)
- [2] Bajaj, S., Sion, R.: Trusteddb: A trusted hardware-based database with privacy and data confidentiality. *IEEE Transactions on Knowledge and Data Engineering* **26**(3), 752–765 (2013)
- [3] Priebe, C., Vaswani, K., Costa, M.: Enclavedb: A secure database using sgx. In: 2018 IEEE Symposium on Security and Privacy (SP), pp. 264–278 (2018). IEEE
- [4] Eskandarian, S., Zaharia, M.: Oblidb: Oblivious query processing for secure databases. *arXiv preprint arXiv:1710.00458* (2017)
- [5] Popa, R.A., Redfield, C., Zeldovich, N., Balakrishnan, H.: Cryptdb: protecting confidentiality with encrypted query processing. In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pp. 85–100 (2011). ACM
- [6] Papadimitriou, A., Bhagwan, R., Chandran, N., Ramjee, R., Haeberlen, A., Singh, H., Modi, A., Badrinarayanan, S.: Big data analytics over

- encrypted datasets with seabed. In: 12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16), pp. 587–602 (2016)
- [7] Kerschbaum, F.: Frequency-hiding order-preserving encryption. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pp. 656–667 (2015). ACM
 - [8] Liu, Z., Chen, X., Yang, J., Jia, C., You, I.: New order preserving encryption model for outsourced databases in cloud environments. *Journal of Network and Computer Applications* **59**, 198–207 (2016)
 - [9] Ishai, Y., Kushilevitz, E., Lu, S., Ostrovsky, R.: Private large-scale databases with distributed searchable symmetric encryption. In: Cryptographers’ Track at the RSA Conference, pp. 90–107 (2016). Springer
 - [10] Cao, N., Wang, C., Li, M., Ren, K., Lou, W.: Privacy-preserving multi-keyword ranked search over encrypted cloud data. *IEEE Transactions on parallel and distributed systems* **25**(1), 222–233 (2013)
 - [11] Bösch, C., Hartel, P., Jonker, W., Peter, A.: A survey of provably secure searchable encryption. *ACM Computing Surveys (CSUR)* **47**(2), 18 (2015)
 - [12] Grubbs, P., Ristenpart, T., Shmatikov, V.: Why your encrypted database is not secure. In: Proceedings of the 16th Workshop on Hot Topics in Operating Systems, pp. 162–168 (2017). ACM
 - [13] Naveed, M., Kamara, S., Wright, C.V.: Inference attacks on property-preserving encrypted databases. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pp. 644–655 (2015). ACM
 - [14] Grubbs, P., Sekniqi, K., Bindschaedler, V., Naveed, M., Ristenpart, T.: Leakage-abuse attacks against order-revealing encryption. In: 2017 IEEE Symposium on Security and Privacy (SP), pp. 655–672 (2017). IEEE
 - [15] Grubbs, P., Lacharité, M.-S., Minaud, B., Paterson, K.G.: Learning to reconstruct: Statistical learning theory and encrypted database attacks. (2019)
 - [16] Lacharité, M.-S., Minaud, B., Paterson, K.G.: Improved reconstruction attacks on encrypted data using range query leakage. In: 2018 IEEE Symposium on Security and Privacy (SP), pp. 297–314 (2018). IEEE
 - [17] Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious rams. *Journal of the ACM (JACM)* **43**(3), 431–473 (1996)

- [18] Stefanov, E., Van Dijk, M., Shi, E., Fletcher, C., Ren, L., Yu, X., Devadas, S.: Path oram: an extremely simple oblivious ram protocol. In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, pp. 299–310 (2013). ACM
- [19] Ren, L., Fletcher, C.W., Kwon, A., Stefanov, E., Shi, E., van Dijk, M., Devadas, S.: Ring oram: Closing the gap between small and large client storage oblivious ram. *IACR Cryptology ePrint Archive* **2014**, 997 (2014)
- [20] Devadas, S., van Dijk, M., Fletcher, C.W., Ren, L., Shi, E., Wichs, D.: Onion oram: A constant bandwidth blowup oblivious ram. In: *Theory of Cryptography Conference*, pp. 145–174 (2016). Springer
- [21] Li, B., Huang, Y., Liu, Z., Li, J., Tian, Z., Yiu, S.-M.: Hybridoram: practical oblivious cloud storage with constant bandwidth. *Information Sciences* **479**, 651–663 (2019)
- [22] Bindschaedler, V., Naveed, M., Pan, X., Wang, X., Huang, Y.: Practicing oblivious access on cloud storage: the gap, the fallacy, and the new way forward. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 837–849 (2015). ACM
- [23] Chang, Z., Xie, D., Li, F.: Oblivious ram: a dissection and experimental evaluation. *Proceedings of the VLDB Endowment* **9**(12), 1113–1124 (2016)
- [24] Hoang, T., Ozkaptan, C.D., Hackebeil, G.A., Yavuz, A.A.: Efficient oblivious data structures for database services on the cloud. *IEEE Transactions on Cloud Computing* (2018)
- [25] Wang, X.S., Nayak, K., Liu, C., Chan, T., Shi, E., Stefanov, E., Huang, Y.: Oblivious data structures. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pp. 215–226 (2014). ACM
- [26] Yahoo!: Yahoo! Cloud Serving Benchmark. [Online]. Available: <https://github.com/brianfrankcooper/YCSB>. [Accessed: 08-Jan-2022]. (2021). <https://github.com/brianfrankcooper/YCSB> Accessed 2021-02-16
- [27] Goodrich, M.T., Mitzenmacher, M., Ohrimenko, O., Tamassia, R.: Oblivious ram simulation with efficient worst-case access overhead. In: *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop*, pp. 95–100 (2011). ACM
- [28] Stefanov, E., Shi, E.: Oblivistore: High performance oblivious cloud storage. In: *2013 IEEE Symposium on Security and Privacy*, pp. 253–267 (2013). IEEE

- [29] Goodrich, M.T., Mitzenmacher, M., Ohrimenko, O., Tamassia, R.: Practical oblivious storage. In: *Proceedings of the Second ACM Conference on Data and Application Security and Privacy*, pp. 13–24 (2012). ACM
- [30] Ren, L., Fletcher, C., Kwon, A., Stefanov, E., Shi, E., Van Dijk, M., Devadas, S.: Constants count: Practical improvements to oblivious {RAM}. In: *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pp. 415–430 (2015)
- [31] Ma, Q., Zhang, W.: Towards practical protection of data access pattern to cloud storage. In: *MILCOM 2018-2018 IEEE Military Communications Conference (MILCOM)*, pp. 1–9 (2018). IEEE
- [32] Hoang, T., Ozkaptan, C.D., Yavuz, A.A., Guajardo, J., Nguyen, T.: S³ oram: A computation-efficient and constant client bandwidth blowup oram with shamir secret sharing. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 491–505 (2017). ACM
- [33] Stefanov, E., Shi, E., Song, D.: Towards practical oblivious ram. *arXiv preprint arXiv:1106.3652* (2011)
- [34] Zahur, S., Wang, X., Raykova, M., Gascón, A., Doerner, J., Evans, D., Katz, J.: Revisiting square-root oram: efficient random access in multi-party computation. In: *2016 IEEE Symposium on Security and Privacy (SP)*, pp. 218–234 (2016). IEEE
- [35] Gentry, C., Goldman, K.A., Halevi, S., Julta, C., Raykova, M., Wichs, D.: Optimizing oram and using it efficiently for secure computation. In: *International Symposium on Privacy Enhancing Technologies Symposium*, pp. 1–18 (2013). Springer
- [36] Mayberry, T., Blass, E.-O., Chan, A.H.: Efficient private file retrieval by combining oram and pir. In: *NDSS* (2014). Citeseer
- [37] Wang, X., Chan, H., Shi, E.: Circuit oram: On tightness of the goldreich-ostrovsky lower bound. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 850–861 (2015). ACM
- [38] Stefanov, E., Shi, E.: Multi-cloud oblivious storage. In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, pp. 247–258 (2013). ACM
- [39] Moataz, T., Blass, E.-O., Mayberry, T.: Chf-oram: a constant communication oram without homomorphic encryption. Technical report, Cryptology ePrint Archive, Report 2015/1116 (2015)

- [40] Shamir, A.: How to share a secret. *Communications of the ACM* **22**(11), 612–613 (1979)
- [41] Devet, C., Goldberg, I., Heninger, N.: Optimally robust private information retrieval. In: Presented as Part of the 21st {USENIX} Security Symposium ({USENIX} Security 12), pp. 269–283 (2012)
- [42] Olumofin, F., Goldberg, I.: Privacy-preserving queries over relational databases. In: International Symposium on Privacy Enhancing Technologies Symposium, pp. 75–92 (2010). Springer
- [43] Goldberg, I.: Improving the robustness of private information retrieval. In: 2007 IEEE Symposium on Security and Privacy (SP’07), pp. 131–148 (2007). IEEE
- [44] Gennaro, R., Rabin, M., Rabin, T., VSS, S.: Fast-track multiparty computations with applications to threshold cryptography. In: Proc of ACM PODC, vol. 98
- [45] Gennaro, R., Rabin, M.O., Rabin, T.: Simplified vss and fast-track multiparty computations with applications to threshold cryptography. In: Podc, vol. 98, pp. 101–111 (1998). Citeseer
- [46] Cramer, R., Damgård, I., de Haan, R.: Atomic secure multi-party multiplication with low communication. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques, pp. 329–346 (2007). Springer
- [47] Roche, D.S., Aviv, A., Choi, S.G.: A practical oblivious map data structure with secure deletion and history independence. In: 2016 IEEE Symposium on Security and Privacy (SP), pp. 178–197 (2016). IEEE
- [48] Liu, Z., Huang, Y., Li, J., Cheng, X., Shen, C.: Divoram: Towards a practical oblivious ram with variable block size. *Information Sciences* **447**, 1–11 (2018)
- [49] Liu, Z., Li, B., Huang, Y., Li, J., Xiang, Y., Pedrycz, W.: Newmcos: towards a practical multi-cloud oblivious storage scheme. *IEEE Transactions on Knowledge and Data Engineering* **32**(4), 714–727 (2019)
- [50] Haider, S.K., Khan, O., van Dijk, M.: Revisiting definitional foundations of oblivious ram for secure processor implementations. *arXiv preprint arXiv:1706.03852* (2017)
- [51] Dauterman, E., Fang, V., Demertzis, I., Crooks, N., Popa, R.A.: Snoopy: Surpassing the scalability bottleneck of oblivious storage. In: Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles,

pp. 655–671 (2021)

- [52] Yahoo!: Yahoo! Cloud Serving Benchmark. [Online]. Available: <https://github.com/brianfrankcooper/YCSB/wiki/Running-a-Workload>. [Accessed: 08-Jan-2022]. (2010). <https://github.com/brianfrankcooper/YCSB/wiki/Running-a-Workload> Accessed 2010-04-28