Efficient Oblivious Data Structures for Database Services on the Cloud*

Thang Hoang† Ceyhun D. Ozkaptan‡ Gabriel Hackebeil§ Attila A. Yavuz¶

Abstract

Database-as-a-service (DBaaS) allows the client to store and manage structured data on the cloud remotely. Despite its merits, DBaaS also brings significant privacy issues. Existing encryption techniques (e.g., SQL-aware encryption) can mitigate privacy concerns, but they still leak information through access patterns, which are vulnerable to statistical inference attacks. Oblivious Random Access Machine (ORAM) can seal such leakages; however, the recent studies showed significant challenges on the integration of ORAM into databases. That is, the direct usage of ORAM on databases is not only costly but also permits very limited query functionalities. In this paper, we propose new oblivious data structures called *Oblivious Matrix Structure* (OMAT) and *Oblivious Tree Structure* (OTREE), which allow tree-based ORAM to be integrated into database systems in a more efficient manner with diverse query functionalities supported. OMAT provides special ORAM packaging strategies for table structures, which not only offers a significantly better performance but also enables a broad range of query types that may not be efficient in existing frameworks. On the other hand, OTREE allows oblivious conditional queries to be performed on tree-indexed databases more efficiently than existing techniques. We implemented our proposed techniques and evaluated their performance on a real cloud database with various metrics, compared with state-of-the-art counterparts.

1 Introduction

Services for outsourcing data storage and related infrastructure to the cloud have grown in the last decade due to the savings it offers to companies in terms of capital and operational costs. For instance, major cloud providers (e.g., Amazon, Microsoft) offer Database-as-a-service (DBaaS) that provides relational database management systems on the cloud. This enables a client to store and manage structured data remotely. Despite its merits, DBaaS raises privacy issues. The client may encrypt the data with standard encryption; however, this also prevents searching or updating information on the cloud, thereby invalidating the effectiveness of database utilization.

Various privacy enhancing technologies have been developed toward addressing the aforementioned privacy *vs.* data utilization dilemma. For instance, the client can use special encryption techniques such as SQL-aware encryption (e.g., [19, 20]) or searchable encryption with various security, efficiency and query functionality trade-offs (e.g., [5, 30, 14, 3, 26, 27, 31, 32]) to achieve the data confidentiality and usability on the cloud. However, even such encryption techniques might not be sufficient for privacy-critical database applications (e.g., healthcare) since sensitive information may be revealed through access

*Full version of the paper to be appeared in IEEE Transactions on Cloud Computing. Work done when the second, the third and the fourth authors were employed at Oregon State University. E-mail: {ozkaptac, hackebeg, attila.yavuz}@oregonstate.edu.

†School of EECS, Oregon State University, Corvallis, OR, 97331. Email: hoangmin@oregonstate.edu.

‡Department of Electrical and Computer Engineering, The Ohio State University, Columbus, OH 43210. E-mail: ozkaptan.1@osu.edu.

§Department of Industrial and Operations Engineering, University of Michigan, Ann Arbor, MI, 48109.

¶Department of Computer Science and Engineering, University of South Florida, Tampa, FL, 33620. E-mail: attilaayavuz@usf.edu.

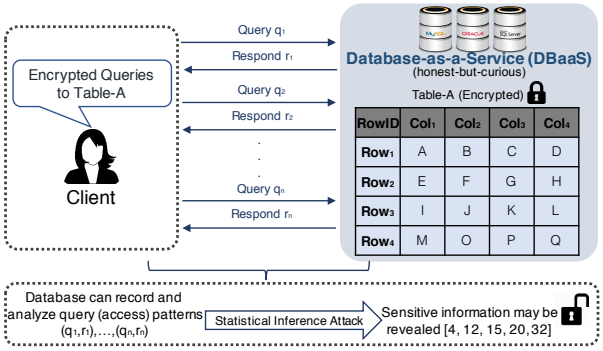


Figure 1. Information leakages through query access patterns over an encrypted database.

patterns when the client execute encrypted queries on the encrypted database. Recent work (e.g., [4, 13, 16, 21, 33]) showed that information leakage through the access pattern can be combined with some prior contextual knowledge to launch statistical inference attacks thereby, revealing vital information about encrypted queries and database. For example, such information leaks may expose the prognosis of illness for a patient or types/timing of financial transactions over valuable assets based on encrypted queries. Therefore, hiding the access pattern an important requirement for privacy-critical database applications.

Oblivious Random Access Machine (ORAM) [11] can be used to hide the client access patterns for such encrypted databases. Preliminary ORAM schemes (e.g., [11, 18]) were costly, but recent ORAM constructions (e.g., [7, 23, 24, 25, 28, 29]) have shown promising results. Most efficient ORAM schemes (e.g., [9, 22, 25]) follow the tree paradigm [23], and achieve $\mathcal{O}(\log N)$ communication overhead. Despite these improvements, there are several research gaps towards achieving efficient integration of ORAM into database applications. In the following, we discuss the research gaps and limitation of state-of-the-art approaches.

1.1 Limitations of Existing Approaches

The direct application of ORAM to the structured encrypted data has been shown to be costly in the context of searchable encryption [2, 12]. Meanwhile, there is a limited number of studies on the application and integration of ORAM for encrypted database systems. Chang et al. in [6] were among the first to investigate the use of ORAM in real database systems with a framework called SEAL-ORAM. In SEAL-ORAM, various ORAMs were implemented and compared on a MongoDB database platform, where ORAM blocks are constructed in a row-oriented manner. While it shows the possibility of using ORAM for encrypted databases, the functionalities and performance offered by such a direct adaptation seem to be limited. We outline the limitations of two direct ORAM applications in Figure 2, and further elaborate them as below.

• *Limitations of Row-Oriented Approach:* In SEAL-ORAM, each row of the database table is packaged into an ORAM block. We refer to this approach as RowPKG. RowPKG allows efficient insert/delete/update queries on a row in the database table. However, to execute oblivious insert/delete/update on a column, RowPKG requires to transfer *all* blocks in ORAM, which is not only bandwidth-costly but also client-storage expensive. Similarly, the execution of any column-related queries (e.g., statistical, conditional queries) is also inefficient because they require transferring all the ORAM blocks, which may not be practical for large databases. R RowPKG (i.e., row-oriented packaging) and its limitations are outlined in Figure 2-(a).

• *Limitations of Cell-Oriented Approach:* Another approach is to package each cell of the database table into an ORAM block. This approach increases the size of position map, which is an imperative component stored at the client in tree-based ORAMs. To eliminate the position map, oblivious 2D-grid structure (referred to as ODS-2D) [29] can be used to store the database table by clustering each $O(\log(N))$ cells

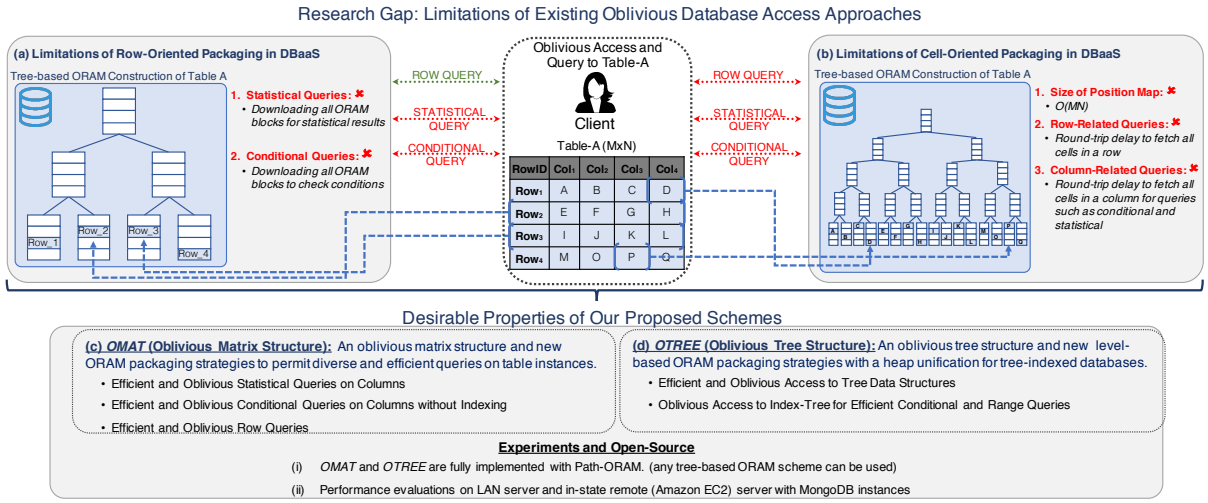


Figure 2. Research gap to be addressed and desirable properties of the proposed schemes.

into an ORAM block and using the pointer trick to link the blocks together. However, this approach may increase the number of requests when the query requires fetching an entire row or column. This incurs end-to-end delay due to a large number of round-trip delays, and therefore, is not suitable for large databases. Cell-oriented packaging and its limitations are summarized in Figure 2-(b). The above discussion indicates that there is a significant need for an efficient oblivious data structure that permits diverse types of queries on encrypted databases. Hence, in this paper, we seek answers to the following research questions:

> *"Can we create an efficient oblivious data structure for encrypted databases that allows diverse types of queries with a low overhead? Can we harness asymptotically optimal ORAMs over structured data to create an oblivious data structure?"*

1.2 Our Contributions

Given the availability of asymptotically-optimal ORAM building blocks, our objective is to create new oblivious data structures by harnessing such ORAMs in efficient manners. Specifically, we propose two efficient oblivious data structures that permit various types of queries on encrypted databases:

(i) Our first scheme is referred to as *Oblivious Matrix Structure* (OMAT) (Section 3.1). The main idea behind OMAT is to create an oblivious matrix structure that permits efficient queries over table objects in the database not only for the row but also column dimension. This is achieved via various strategies that are specifically tailored for the matrix structure with a delicate balance between the query diversity and the ORAM bandwidth overhead. This allows OMAT to perform various types of oblivious queries without streaming a large number of ORAM blocks or maintaining a very large position map at the client. *(ii)* Our second scheme is referred to as *Oblivious Tree Structure* (OTREE) (Section 3.2), which is designed for oblivious accesses on tree-indexed database instances. Given a column whose values can be sorted into a tree structure (i.e., numeric values), OTREE allows efficient oblivious conditional queries (e.g., a range query).

We illustrate desirable properties of our schemes in Figure 2-(c,d), and further discuss them as follows.

• *Highly efficient and diverse oblivious queries:* OMAT supports a diverse set of queries to be executed with ORAM. Specifically, OMAT permits oblivious statistical queries over value-based columns such as `SUM`, `AVG`, `MAX` and `MIN`. Moreover, oblivious queries on rows (e.g., insert, update) can be executed on an attribute with a similar cost. As shown in Table 1, with the given parameters and experimental setup, executing a column-related query such as statistical or conditional query with OMAT is approximately

Table 1. Transmission cost and client storage for compared schemes.

Scheme	Communication Cost[a]	Efficiency[b]	Client Storage[c]	End-to-End Delay[d]	
				Moderate Network	High Network
single column-related query (e.g., statistical, conditional queries)					
RowPKG [6]	$Z \cdot (B_1 \cdot N) \cdot (2M-1)$	1.00	$O(M \cdot N) \cdot w(1)$	6096 s	776 s
ODS-2D [29]	$(M/4) \cdot [Z \cdot (16 \cdot B_1) \cdot \log_2(M \cdot N/16)]$	17.04	$O(M \cdot \log(M \cdot N)) \cdot w(1)$	1245 s	292 s
OMAT	$Z^2 \cdot (B_1 \cdot M) \cdot \log_2(N)$	**28.44**	$O(M \cdot \log(N)) \cdot w(1)$	**475 s**	**60 s**
single row-related query (e.g., insert/delete/update queries)					
RowPKG [6]	$Z \cdot (B_2 \cdot N) \cdot \log_2(M)$	**1.00**	$O(N \cdot \log(M)) \cdot w(1)$	**567 ms**	**56 ms**
ODS-2D [29]	$(N/4) \cdot [Z \cdot (16 \cdot B_2) \cdot \log_2(M \cdot N/16)]$	0.19	$O(N \cdot \log(M \cdot N)) \cdot w(1)$	2380 ms	350 ms
OMAT	$Z^2 \cdot (B_2 \cdot N) \cdot \log_2(M)$	0.25	$O(N \cdot \log(M)) \cdot w(1)$	2032 ms	128 ms
traversal on database tree index (e.g., range queries)					
non-caching					
ODS-Tree [29]	$2 \cdot Z_1 \cdot B \cdot (H+1)^2$	1.00	$O(H) \cdot w(1)$	7929 ms	1318 ms
OTREE	$Z_2 \cdot B \cdot (H+1) \cdot (H+2)$	**1.60**	$O(H) \cdot w(1)$	**3762 ms**	**592 ms**
half-top caching					
ODS-Tree [29]	$2 \cdot Z_1 \cdot B \cdot \lceil \frac{H+1}{2} \rceil \cdot (H+1)$	1.00	$O(\sqrt{2^H}) + O(H) \cdot w(1)$	5979 ms	1008 ms
OTREE	$Z_2 \cdot B \cdot \lceil \frac{H+1}{2} \rceil \cdot (\lceil \frac{H+1}{2} \rceil + 1)$	**3.20**	$O(\sqrt{2^H}) + O(H) \cdot w(1)$	**1676 ms**	**272 ms**

• *Table Notations*: M and N denote the total number of (real) rows and columns in the matrix data structure, respectively. H is the height of the tree data structure. Z and B denote the bucket size and size of each block (in bytes), respectively.
• *Settings*: We instantiate our schemes and their counterparts with underlying Path-ORAM for a fair comparison. The bottom half of the table compares OTREE and ODS-Tree when combined with tree-top caching technique proposed in [17], in which we assume the top half of tree-based ORAM is cached on the client during all access requests.
• *Server Storage*: All of the oblivious matrix structures require $O(MN)$ server storage, however, the storage of OMAT is a constant (e.g., $Z = 4$) factor larger than others. OTREE is twice more storage efficient than ODS.
[a] Represents the total cost in terms of bytes to be processed (e.g., communication/computation depends on the underlying ORAM scheme) between the client and the server for each request. For OMAT , ODS-2D and RowPKG, the cost is for one access operation per query. For OTREE and ODS, the cost is for traversing an arbitrary path in a binary tree.
[b] Denotes the communication cost efficiency compared to chosen baseline, where $Z = 4, B_1 = 64, B_2 = 128, M = 2^{15}, N = 2^9$ for ODS-2D, RowPKG and OMAT, and $Z_1 = 4, Z_2 = 5$ (for stability), $B = 4096, H = 20$ for ODS-Tree and OTREE.
[c] Client storage consists of the worst-case stash size to keep fetched data. Additionally, the position map of OMAT and RowPKG are $O((M+N)\log(M+N))$ and $O(M \cdot \log(M))$, respectively. For ODS based structures and OTREE, position map requires $O(1)$ storage due to pointers and half-top cached blocks are also included in client storage.
[d] The delays were measured with a MongoDB instance running on Amazon EC2 connected with the client on two different network settings which are described in Section 5.1.

28× more communication efficient than that of RowPKG and this enables OMAT to perform queries approximately 13× faster than that of RowPKG. Compared to ODS-2D, although OMAT is only 1.6 × more communication-efficient, it performs approximately 5× faster in practice due to the large number of additional round-trip delays. OTREE achieves better performance than ODS for obliviously accessing the database index, which is constructed from the values of a column as a tree data structure. The communication cost of OTREE is 1.6× less than that of ODS without caching. This gain can be increased up to 3.2× with the caching strategy.

• *Generic Instantiations from Tree-based ORAM Schemes:* We notice that *any* tree-based ORAM scheme (e.g., [25, 28]) can be used for both OMAT and OTREE instantiations. This provides a flexibility in selecting a suitable underlying ORAM scheme, which can be adjusted according to the performance requirements of specific applications. Note that, in this paper, we instantiated our schemes with Path-ORAM [25] due to its efficiency, simplicity and not requiring any server-side computation.

• *Comprehensive Experiments and Evaluations:* We implemented OTREE, OMAT, and their counterparts under the same framework. We evaluated their performance with a MongoDB database instance unning on a remote AmazonEC2 server with two different network settings: (1) moderate-speed network and (2) high-speed network. This permits us to observe the impact of real network and cloud environment.

2 Preliminaries

We now present cryptographic techniques and implementation frameworks that are used by or are relevant to our proposed schemes.

Table 2. Summary of notations in tree-based ORAM.

Symbol	Description
N	Total number of nodes in the tree-based ORAM
H	Height of the ORAM tree structure
b, B	Block and Block size
Z	Capacity (in blocks) of each node
$\mathcal{P}(i)$	Path from leaf node i to root bucket in the tree
$\mathcal{P}(i, \ell)$	Bucket at level ℓ along the path $\mathcal{P}(i)$
$\mathcal{S}$	Client's local stash (optional)
pm	Client's local position map
$i :=$ pm[id]	block identified by id is currently associated with leaf node i, i.e., it resides somewhere along $\mathcal{P}(i)$ or (optional) in the stash.

2.1 Tree-based ORAM

ORAM enables a client to access encrypted data on an untrusted server without exposing the access patterns (e.g., memory blocks, their access time and order) to the server [11]. Existing ORAM schemes rely on IND-CPA encryption [15] and an oblivious shuffling to ensure that any data access patterns of the same length are computationally indistinguishable by anyone but the client.

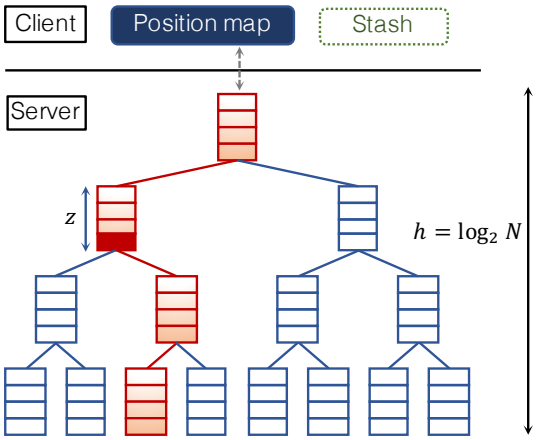


Figure 3. Tree-based ORAM structure [23].

Recent ORAMs (e.g., [8, 9, 22, 25]) follow the tree paradigm [23], which consists of two main data structures: A full binary tree data structure stored at the server side and a position map (denoted as pm) stored at the client side (Figure 3). Each node in the tree is called a bucket (denoted B) which can store up to Z data blocks (e.g., $Z = 4$). Each block b has a unique identifier id and all blocks are of the same size B (4 KB). A tree-based ORAM with N leaf nodes can store up to N real blocks, and other empty slots are filled with dummy data. $\mathcal{P}(i)$ denotes a path from the root to leaf i of the tree. The position map pm holds the location among 2^N possible paths $\mathcal{P}(i)$ for every block with identifier id. The size of pm is $\mathcal{O}(N \log N)$ which can be reduced to $\mathcal{O}(1)$ by using recursive ORAMs to store pm on the server with the $\mathcal{O}(\log N)$ increase of communication rounds for each access operation. Table 2 summarizes notations being used for tree-based ORAM scheme.

There are two basic phases in tree-based ORAMs: retrieval and eviction. For each access operation, the client gets the path ID of accessing block from the position map and sends the path ID to the server who responds with all blocks residing in the requested path. The client decrypts and processes the received data to obtain the desired block and executes the eviction function, which re-encrypts downloaded block(s) and pushes them back to the ORAM tree. Notice that although recently proposed ORAM schemes that follow the tree paradigm (e.g., [25, 22, 9]) provide different trade-offs between

communication and computation overhead, they all rely on the aforementioned basic operations.

Path-ORAM: In Path-ORAM [25], all real blocks are downloaded and stored temporarily in a so-called stash at the client in the retrieval phase. A new random address is then assigned to the accessed block and the local position map is updated. Next, the blocks in the stash are evicted according to the retrieval path. Path-ORAM offers asymptotically optimal communication and computation cost of $\mathcal{O}(\log N)$ by storing $\mathcal{O}(N \log N)$-sized position map. As the recursive ORAMs are known to be highly costly, we do not discuss them in this work.

2.2 Oblivious Data Structure

Oblivious Data Structure (ODS) proposed by Wang et al. [29] leverages "pointer techniques" to reduce the storage cost of position map components in non-recursive ORAM schemes to $\mathcal{O}(1)$, if the data to be accessed have some specific structures (e.g., grid, tree, etc.). For instance, given a binary search-sorted array as illustrated in Figure 4, the ORAM block is augmented with $k+1$ additional slots that hold the position of the block along with the positions and identifiers of its children as $b := (\mathsf{id}, \mathsf{data}, \mathsf{pos}, \mathsf{childmap})$, where id is the block identifier, data is the block data, pos is its position in ORAM structure, and $\mathsf{childmap}$ is a miniature position map with entries $(\mathsf{id}_i, \mathsf{pos}_i)$ for k children. To ensure that the $\mathsf{childmap}$ is up to date, a child block must be accessed through at most one parent at any given time. If a block does not have a parent (e.g., the root of a tree), its position will be stored in the client. A parent block should never be written back to the server without updating positions of its children blocks.

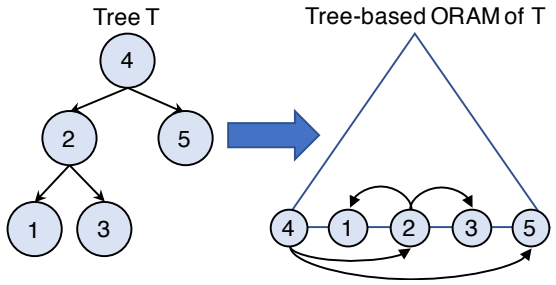


Figure 4. Oblivious Data Structure for a tree structure [29].

2.3 ORAM Implementation Framework

One of the most reliable and complete ORAM frameworks is CURIOUS [2], which gives a complete implementation of the state-of-the-art ORAM schemes (e.g., Path-ORAM [25]) in Java. In this paper, we chose CURIOUS to implement our oblivious data structures as it can be adopted with database drivers such as MongoDB or MySQL.

3 Proposed Techniques

We now present our proposed oblivious data structures, which are specially designed for efficient operations in database settings. We propose two schemes including *Oblivious Matrix Structure* (OMAT) and *Oblivious Tree Structure* (OTREE). OMAT supports efficient oblivious statistical queries on generic table instances, while OTREE supports range and conditional queries on tree-indexed instances. For our oblivious data structures, we choose Path-ORAM [25] as the underlying ORAM for the following reasons: (i) It is simple yet achieves asymptotic efficiency. (ii) Unlike some recent ORAMs [9, 22] that require computations at the server side, it requires only read/write operations. This is useful since such advanced cryptographic operations might not be readily offered by well-known database instances (e.g., MongoDB, MySQL). (iii) The availability of Path-ORAM implementations on existing frameworks (e.g.,

CURIOUS [2]) enables a fair experimental comparison of the proposed techniques with the state-of-the-art.

3.1 Oblivious Access on Table Structures

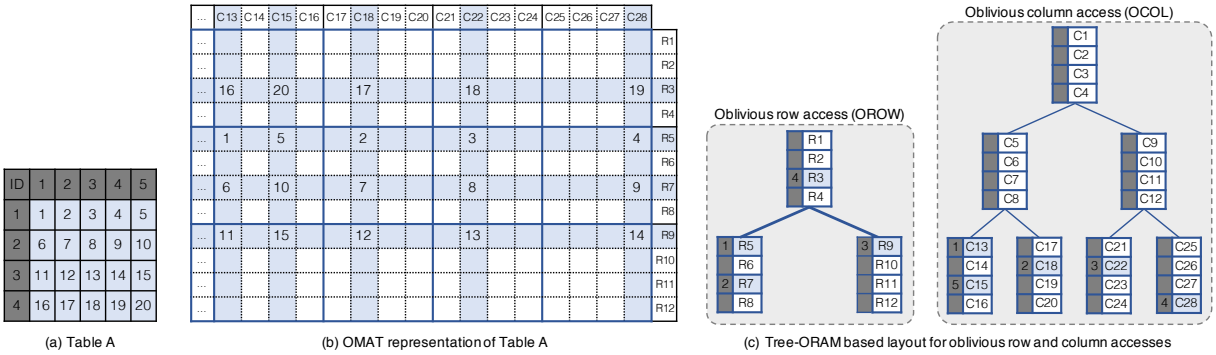


Figure 5. OMAT structure for oblivious access on table.

The direct application of tree-based ORAMs to access encrypted tables in general [2] and database systems in specific [6] have been shown to be inefficient for large datasets. Specifically, if each row in the table is packaged into an ORAM block as in [6], then performing queries to fetch a column in such a table (e.g., statistics) would require the client to stream all blocks in the ORAM structure, which might be impractical. On the other hand, packaging each cell in the table into an ORAM incurs a high network delay and client storage overhead. Thus, we investigate on how to translate the table into an oblivious data structure so that each row and column of it can be both accessed efficiently by a given ORAM scheme. Below, we first describe our oblivious data structure and then present our OMAT access scheme on top of it.

• *Oblivious Data Structure for OMAT.* The main data structure that we use for oblivious access on a table is a matrix. Given an input table $\mathbf{T}$ of size $M \times N$, we allocate a matrix $\mathbf{M}$ of size $Z \cdot 2^{\lceil \log_2(M) \rceil - 1} \times Z \cdot 2^{\lceil \log_2(N) \rceil - 1}$. We arrange tree-based ORAM building blocks for oblivious access as follows:

The layout of OMAT matrix $\mathbf{M}$ can be interpreted as two logical tree-based ORAMs defined as oblivious rows (denoted as OROW) and oblivious columns (denoted as OCOL) as illustrated in Figure 5. That is, the ORAM for row access on OROW is formed by a set of blocks $b_i := (\mathsf{id}_i, \mathsf{data}_i)$, where id_i is either a unique identifier if b_i contains the content of a row of the table $\mathbf{T}$ or null otherwise, and $\mathsf{data}_i \leftarrow \mathbf{M}[i, *]$. We group Z subsequent rows in $\mathbf{M}$ to form a bucket (i.e., node) in the OROW structure. Similarly, the ORAM for column access on OCOL is formed by $b_j = (\mathsf{id}_j, \mathsf{data}_j)$. Each (bucket) node in OCOL is formed by grouping Z subsequent blocks.

We assign each row $\mathbf{T}[i', *]$ $(i' = 1, \ldots, M)$ and each column $\mathbf{T}[*, j']$ $(j' = 1, \ldots, N)$ with a random leaf node IDs $u_{i'}$ and $v_{j'}$ in OROW and OCOL, respectively. That is, the data of $\mathbf{T}[i, *]$ and $\mathbf{T}[*, j]$ reside in some rows and columns of $\mathbf{M}$ along the assigned paths $\mathcal{P}(u_i)$ in OROW and $\mathcal{P}(v_j)$ in OCOL, respectively. In other words, $\mathbf{M}[i, j] \leftarrow \mathbf{T}[i', j']$, where $\mathbf{M}[i, *] \in \mathcal{P}(u_{i'})$ in OROW and $\mathbf{M}[*, j] \in \mathcal{P}(v_{j'})$ in OCOL. Our construction requires two position maps ($\mathsf{pm}_{\mathsf{row}}$ and $\mathsf{pm}_{\mathsf{col}}$) to store the assigned path for each row $\mathbf{T}[i', *]$ and each column $\mathbf{T}[*, j']$ of table $\mathbf{T}$ in OROW and OCOL, respectively. Our position maps store all necessary information to locate the exact position of a row/column data in the tree-based ORAM structures as $\mathsf{pm} := (\mathsf{id}, \langle \mathsf{pathID}, \mathsf{level}, \mathsf{order} \rangle)$, where $0 \leq \mathsf{level} \leq \log_2(N)$ indicates the level of the bucket, in which the row/column with id resides, and $1 \leq \mathsf{order} \leq Z$ indicates its order in the bucket.

• *The Proposed OMAT Access Scheme.* We present our OMAT scheme, which is instantiated with Path-ORAM, in Algorithm 1. Specifically, given a *column* (resp. *row*) identifier (id) to be accessed[1], the

[1]The access can be any types of operation such as read/add/delete/modify.

Algorithm 1 data $\leftarrow$ OMAT.Access(op, dim, id)

1: $b \leftarrow \mathsf{pm}_{\mathsf{dim}}[\mathsf{id}].\mathsf{pathID}$
2: **if** dim = col **then**
3: $\quad \mathsf{pm}_{\mathsf{dim}}[\mathsf{id}].\mathsf{pathID} \xleftarrow{\$} \{1, \ldots, 2^{\lceil \log_2(N) \rceil - 1}\}$
4: $\quad H \leftarrow \lceil \log_2(N) \rceil$
5: **else**
6: $\quad \mathsf{pm}_{\mathsf{dim}}[\mathsf{id}].\mathsf{pathID} \xleftarrow{\$} \{1, \ldots, 2^{\lceil \log_2(M) \rceil - 1}\}$
7: $\quad H \leftarrow \lceil \log_2(M) \rceil$
▷ *Read all rows/columns on the path* $\mathcal{P}(b)$
8: **for** each $\ell \in \{0, \ldots, H\}$ **do**
9: $\quad \mathcal{S}_{\mathsf{dim}} \leftarrow \mathcal{S}_{\mathsf{dim}} \cup \mathsf{ReadBucket}(\mathsf{dim}, \mathcal{P}(b, \ell))$
10: data $\leftarrow$ Read row/column with id from $\mathcal{S}_{\mathsf{dim}}$
11: data $\leftarrow \mathsf{FilterDummy}(\mathsf{data}, \mathcal{S}_{\neg\mathsf{dim}})$
12: $\mathcal{S}_{\neg\mathsf{dim}} \leftarrow \mathsf{Update}(\mathcal{S}_{\neg\mathsf{dim}}, \mathsf{pm}_{\mathsf{dim}})$
13: **if** op = write **then**
14: $\quad \mathcal{S}_{\mathsf{dim}} \leftarrow (\mathcal{S}_{\mathsf{dim}} \setminus \{(\mathsf{id}, \mathsf{data})\}) \cup \{(\mathsf{id}, \mathsf{data}^*)\}$
▷ *Evict blocks from the stash*
15: **for** each $\ell \in \{H, \ldots, 0\}$ **do**
16: $\quad \mathcal{S}'_{\mathsf{dim}} \leftarrow \{(\mathsf{id}', \mathsf{data}') \in \mathcal{S}_{\mathsf{dim}} | \mathcal{P}(b, \ell) = \mathcal{P}(\mathsf{pm}_{\mathsf{dim}}[\mathsf{id}'].\mathsf{pathID}, \ell)\}$
17: $\quad \mathcal{S}'_{\mathsf{dim}} \leftarrow$ Select $\min(|\mathcal{S}'_{\mathsf{dim}}|, Z)$ blocks from $\mathcal{S}'_{\mathsf{dim}}$
18: $\quad \mathcal{S}_{\mathsf{dim}} \leftarrow \mathcal{S}_{\mathsf{dim}} \setminus \mathcal{S}'_{\mathsf{dim}}$
19: $\quad o \leftarrow 1$
20: $\quad$ **for** each $(\mathsf{id}', \mathsf{data}') \in \mathcal{S}'_{\mathsf{dim}}$ **do**
21: $\quad\quad \mathsf{pm}[\mathsf{id}'].\mathsf{level} \leftarrow \ell$
22: $\quad\quad \mathsf{pm}[\mathsf{id}'].\mathsf{order} \leftarrow o$, $o \leftarrow o + 1$
23: $\quad \mathsf{WriteBucket}(\mathsf{dim}, \mathcal{P}(b, \ell), \mathcal{S}'_{\mathsf{dim}})$
24: **return** data

client retrieves its location from the *column* (resp. *row*) position map (step 1). The client then assigns the *column* (resp. *row*) to a new location selected uniformly at random (steps 2–7). The client reads all *columns* (resp. *rows*) residing on the same path according to tree-based ORAM layout (as depicted in Figure 5-(c) to the stash (steps 8–9). In this case, we modify the original ReadBucket subroutine of Path-ORAM, where it now takes an extra parameter (dim) that indicates the dimension to be read, and outputs the corresponding Z columns/rows in the bucket. The client retrieves the *column* (resp. *row*) with id from the stash (step 10). One might observe that according to OMAT structure, the retrieved *column* (resp. *row*) will contain data from dummy *rows* (resp. *columns*) as depicted by empty blue cells in Figure 5-(b). Therefore, to obtain only the real data of the requested *column* (resp. *row*), the client filters all data from dummy *rows* (resp. *columns*) (step 11). Moreover, since the position of the retrieved column (resp. *row*) is moved to a new random position (steps 2–7), it is required to update all *rows* (resp. *columns*) that are currently stored in the stash at this column (resp. *row*) position to achieve the consistency (step 12). If the access is to update, the client then updates the *column* (resp. *row*) with new data (steps 13–14) Finally, the client performs eviction as described in Path-ORAM to flush *columns* (resp. *rows*) from the stash back to the OMAT structure in the server (steps 15–23).

Notice that all columns/rows are IND-CPA decrypted and re-encrypted as they are read and written to/from the server, respectively. We assume that it is not required to hide the information whether a column or a row is being accessed. However, this can be achieved with the cost of performing oblivious accesses on both row and column (one of them is dummy selected randomly) for each access.

• *Use Case: Statistical and Conditional Queries.* Recall that, in row-oriented packaging, implementing secure statistical queries on a column requires downloading the entire ORAM blocks from the database. In contrast, OMAT structure allows queries such as `add`, `delete`, `update` not only on its row but also on its column dimension. Thus, we can implement statistical queries (e.g., `MAX`, `MIN`, `AVG`, `SUM`, `COUNT`, etc.)

over a column in an efficient manner via OMAT. Note that OMAT can also permit conditional query on rows with `WHERE` statement. Similar to statistical queries, the query can be implemented by reading the attribute column on which the `WHERE` clause looks up OCOL first to determine appropriate records that satisfy the condition, and then obliviously fetching such records on OROW structure. For example, assume that we have the following SQL-like conditional search.

```
SELECT * FROM A WHERE C > k
```

It can be implemented by:

1. Read the column C with id' on OCOL as $\mathbf{C}[*, \mathsf{id}'] \leftarrow \mathsf{OMAT.Access}(\mathsf{read}, \mathsf{col}, \mathsf{id}')$.
2. Get IDs of rows whose value larger than k, and such IDs are in $\mathsf{pm}_{\mathsf{row}}$ as $\mathcal{I} \leftarrow \{\mathsf{id} | \mathsf{id} \in \mathsf{pm}_{\mathsf{row}}.\mathsf{id} \wedge \mathbf{C}[\mathsf{id}, \mathsf{id}'] > k\}$
3. Access on OROW to get the desired result as $\mathbf{R}[\mathsf{id}, *] \leftarrow \mathsf{OMAT.Access}(\mathsf{read}, \mathsf{row}, \mathsf{id})$, for each $\mathsf{id} \in \mathcal{I}$.

The aforementioned approach can work with any unindexed columns. In the next section, we propose an alternative approach that can offer a better performance if the columns can be indexed with certain restrictions.

3.2 Oblivious Access on Tree Structures

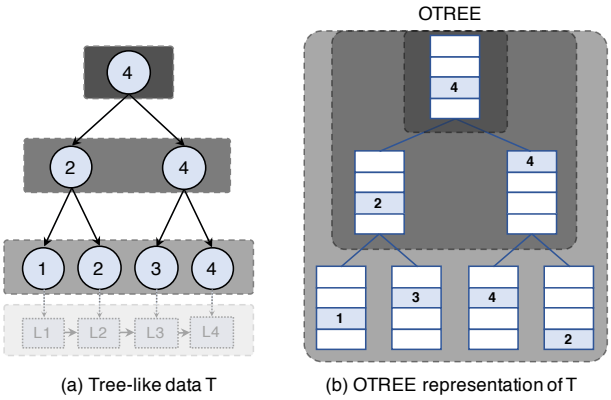


Figure 6. The OTREE layout for a tree data.

In the unencrypted database setting, conditional queries can be performed more efficiently, if column values can be indexed by a search-efficient tree data structure (e.g., Range tree, B+ tree, AVL tree). Figure 9 illustrates an example of a column indexed by a range tree for (non)-equality/range queries, in which each leaf node points to a node in another linked-list structure that stores the list of matching IDs. We propose an oblivious tree structure called OTREE, in which indexed data for such queries are translated into a balanced tree structure. As in OMAT, *OTREE can be instantiated from any tree-based ORAM scheme*. Notice that oblivious access on a tree was previously studied in [29]. Our method requires less amount of data to be transmitted and processed, since the structure of indexed values (i.e., the tree data structure) is not required to be hidden, and the client is merely required to traverse an arbitrary path of the tree. We present the construction of OTREE as follows.

• *Oblivious Data Structure for OTREE:* Given a tree-indexed data $\mathbf{T}$ of height H as input, we first construct the OTREE structure of height H with ORAM buckets as illustrated in Figures 6-(a,b). Then, each node of $\mathbf{T}$ at level ℓ is assigned to a random path and placed into a bucket of OTREE which resides on the assigned path at level ℓ' where $\ell' \leq \ell$. In other words, *any node of* $\mathbf{T}$ *at level* $0 \leq \ell \leq H$ *will reside*

in a bucket at level ℓ or lower in OTREE. If there is no empty slot in the path, the node will be stored in the stash if OTREE is instantiated with stash-required ORAM schemes (e.g., Path-ORAM).

We assume $\mathbf{T}$ is sorted by nodes' id and the position of nodes at level ℓ is stored in its parent node at level $\ell-1$ using the pointer technique proposed in [29]. Hence, each node of $\mathbf{T}$ is considered as a separate block in OTREE structure as: $b := (\mathsf{id}, \mathsf{data}, \mathsf{childmap})$, where id is the node identifier sorted in $\mathbf{T}$ (e.g., indexed column value), data indicates the node data, and childmap is of structure $\langle \mathsf{id}, \mathsf{pos} \rangle$ that stores the position information of node's children.

• *The Proposed OTREE Access Scheme:* OTREE can be instantiated with any tree-based ORAM schemes (e.g., Ring-ORAM [22], Circuit-ORAM [28]) , as similar to OMAT in Section 3.1, by modifying corresponding retrieval/eviction procedures while preserving the constraints of OTREE regarding the deepest level of nodes. OTREE also receives a significant benefit from caching mechanisms like top-tree caching [17], which can speed up bulk access requests.

We give the proposed OTREE scheme instantiated with Path-ORAM in Algorithm 2. Specifically, given the node identifier id to be accessed in the id-sorted tree structure, the client first reads the root bucket of Path-ORAM structure to obtain the root node of the tree (steps 1–3). The client then compares the requested id with the root id to decide which child of the root node should be accessed in the next step. The client accesses this child by reading its path in the Path-ORAM structure from level 0 to level 1. We notice that for each node at level l in the tree to be accessed, the client only accesses the path in the Path-ORAM structure up to level l. The process repeats until the desired id is found (steps 4–16). Finally, the client performs eviction to flush read nodes back to the Path-ORAM structure, wherein nodes at level l in the tree must reside somewhere in the Path-ORAM structure from level 0 to level l (steps 17–21).

The construction and constraints of OTREE require a stability analysis to ensure that tree-based ORAM scheme on OTREE behaves similarly to ODS in terms of the stash overflow probability. We provide an empirical stability analysis of OTREE with Path-ORAM as follows.

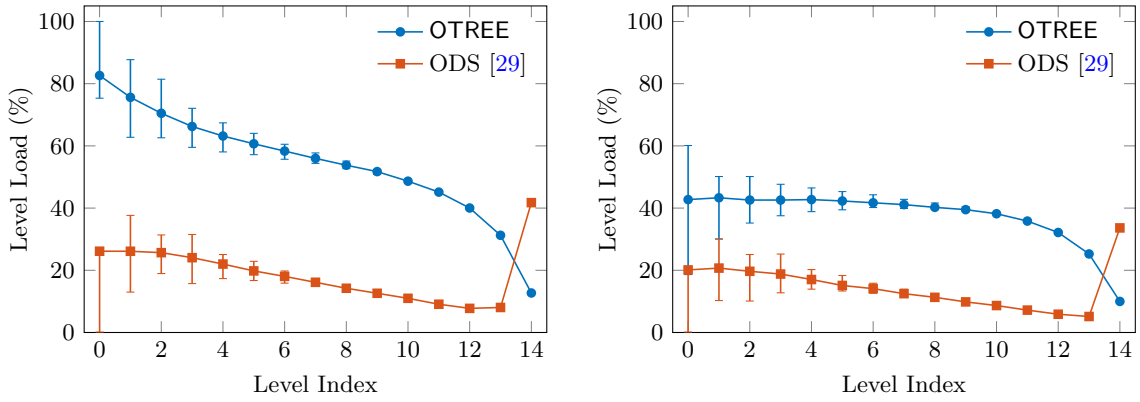
Algorithm 2 $(\mathsf{data}) \leftarrow \mathsf{OTREE.Access}(\mathsf{op}, \mathsf{id}, \mathsf{data}^*)$

1: $x_0 \leftarrow \mathsf{RootPos}$
2: $\mathcal{S} \leftarrow \mathcal{S} \cup \mathsf{ReadBucket}(\mathcal{P}(x_0, 0), 0)$
3: $b_0 \leftarrow$ Read block with $\mathsf{id}_0 = 0$ from $\mathcal{S}$
4: **for** each $\ell \in \{0, \ldots, H-1\}$ **do**
5: **if** $\mathsf{compare}(\mathsf{id}, \mathsf{id}_\ell) = \mathsf{go\_right}$ **then**
6: $(\mathsf{id}_{\ell+1}, x_{\ell+1}) \leftarrow b_\ell.\mathsf{child}[1]$
7: $b_\ell.\mathsf{child}[1].\mathsf{pos} \xleftarrow{\$} \{0, \ldots, 2^\ell - 1\}$
8: **else**
9: $(\mathsf{id}_{\ell+1}, x_{\ell+1}) \leftarrow b_\ell.\mathsf{child}[0]$
10: $b_\ell.\mathsf{child}[0].\mathsf{pos} \xleftarrow{\$} \{0, \ldots, 2^\ell\}$
11: $\mathcal{S} \leftarrow \mathcal{S} \cup \mathsf{ReadBucket}(\mathcal{P}_{\ell+1}(x_{\ell+1}, \ell+1))$
12: $b_\ell \leftarrow$ Read block id_ℓ from $\mathcal{S}$
13: **if** $\mathsf{id} = \mathsf{id}_\ell$ **then**
14: $\mathsf{data} \leftarrow b_\ell.\mathsf{data}$
15: **if** $\mathsf{op} = \mathsf{write}$ **then**
16: $S \leftarrow (S \setminus \{b_\ell\}) \cup \{(\mathsf{id}, \mathsf{data}^*, \mathsf{child})\}$
17: **for** each $\ell' \in \{\ell, \ldots, 0\}$ **do**
18: $\mathcal{S}' \leftarrow \{b' \in \mathcal{S} : \mathcal{P}_\ell(b'.\mathsf{pos}, \ell') = \mathcal{P}_\ell(b_\ell.\mathsf{pos}, \ell') \wedge b'.\mathsf{level} = \ell\}$
19: $\mathcal{S}' \leftarrow$ Select $\min(|\mathcal{S}'|, z)$ blocks from $\mathcal{S}'$
20: $\mathcal{S} \leftarrow \mathcal{S} \setminus \mathcal{S}'$
21: $\mathsf{WriteBucket}(\mathcal{P}_\ell(x_\ell, \ell'), \mathcal{S}')$
22: **return** data

• *Stability Analysis of OTREE:* We analyze the stability of OTREE in terms of the average bucket load in each level of the ORAM tree. Intuitively, one would expect an increase in average bucket load near the top of the ORAM tree, and a possible increase in the average client stash size if a Path-ORAM variant (e.g., [22, 17]) is used. We show empirically by our simulations, that OTREE behaves almost similar to ODS with a bucket size of $Z \geq 4$ with Path-ORAM. With $Z = 5$, bucket usage with OTREE structure approaches that of the stationary distribution when using an infinitely large bucket size.

Our empirical study considered experiments with an ORAM tree of height $H = 14$ storing $N = 2^{15} - 1$ blocks. We ran the experiments with different bucket sizes to observe its effect on the stash size and bucket usage. We treated ORAM blocks as nodes in a full binary tree of $H = 14$. We inserted nodes into storage according to the breadth-first order via access functions followed by a series of $(H + 1)$-length access requests, each of which consists of accessing a path of nodes from the root to a random leaf node in the binary tree. A single-round experiment was the execution of 2^{14} random root-to-leaf access sequences as described.

Figures 7 - 8 show the results of these experiments for ODS and OTREE with different bucket sizes. The results were generated by first running 1000 warm-up rounds after the initialization, and then collecting statistics over 1000 test rounds. Figure 7 depicts that with a bucket size $Z = 5$, buckets near the root of the OTREE structure contain roughly two non-empty blocks (one more than the average number of



(a) $Z = 4$ **(b)** $Z = 5$

Figure 7. Average bucket load within each level of the ORAM tree for different bucket sizes, where y-axis shows the average percentage of bucket being used and x-axis shows the bucket levels from 0 (root) to 14 (leaf).

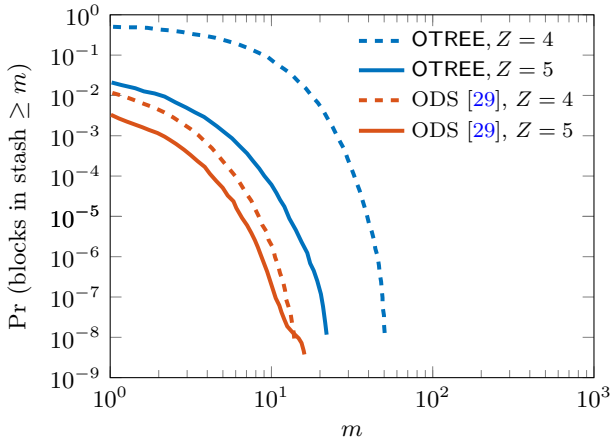


Figure 8. Probability of stash size exceeding the threshold.

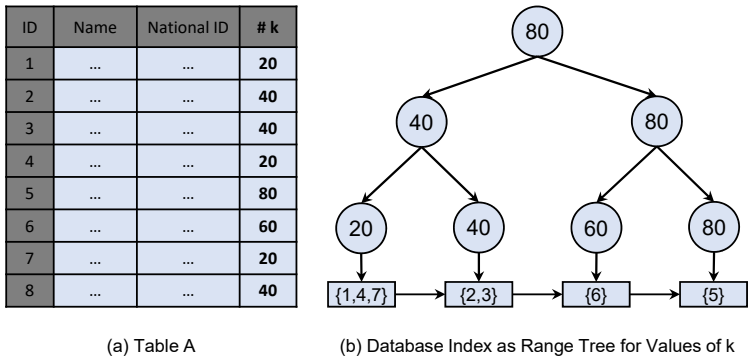


Figure 9. Values in a column indexed as a tree, and a linked list to retrieve matching IDs for conditional queries.

blocks assigned to them). Figure 8 illustrates that with $Z \leq 4$, the probability of the stash size exceeding $O(H)$ for OTREE diminishes quickly. These results suggest that using $Z = 5$ for OTREE in order to make underlying ORAM scheme in OTREE behaves similarly to that with $Z = 4$ on ODS.

• *Use Case: Conditional Query on Columns:*

We exemplify an implementation of a database index structured as OTREE for conditional queries as follows: Consider a column whose values are indexed by a sorted tree $\mathbf{T}$ of height h by putting distinct values as keys on leaf nodes as depicted in Figure 9. The leaf nodes of $\mathbf{T}$ points to a node ID in a linked-list structure that contains a list of matching IDs with the key. We translate $\mathbf{T}$ into OTREE, where each node at level $\ell < H$ stores the position maps of its children. We store a list of IDs in each linked-list node using an inverted index with compression. As the data structure for the linked-list, we employ ODS to store it in another ORAM structure (see [29] for details). Hence, each leaf node of $\mathbf{T}$ stores the position map of a linked-list node in ODS it points to. An example of a given conditional query as follows.

$$\texttt{SELECT} \ * \ \texttt{FROM A WHERE C} = \texttt{k}$$

where the column C is indexed into OTREE. It can be executed obliviously as follows.

1. Traverse a path with OTREE to get a leaf node as $b \leftarrow \mathsf{OTREE.Access}(k)$.
2. Get ID and position map of linked-list node which b points to as $(\mathsf{id}, \mathsf{pos}) \leftarrow b.\mathsf{childmap}$
3. Access on ODS to get the desired result as $\mathcal{R} \leftarrow \mathrm{ODS.Access}(\mathsf{id}, \mathsf{pos}, \cdot)$

The overall cost for this approach is: $O(\log^2 N + k \cdot O(\log(N))$, where k is the distance from the first element of the linked-list. The first part is the overhead of OTREE and the second part is the overhead of ODS (without padding).

4 Security Analysis

Our security analysis, as in Path-ORAM [25], is concise as the security of our proposed schemes are evident from their base ORAM.

Definition 1 (ORAM security [25])**.** *Let* $\vec{y} := ((\mathsf{op}_1, \mathsf{id}_1, \mathsf{data}_1), \ldots, (\mathsf{op}_M, \mathsf{id}_M, \mathsf{data}_M))$ *be a data request sequence of length* M*, where each* op_i *denotes a* $\mathsf{read}(\mathsf{id}_i)$ *or a* $\mathsf{write}(\mathsf{id}_i, \mathsf{data})$ *operation. Let* $A(\vec{s})$ *denote the sequence of accesses made to the server that satisfies the user data request sequence* $\vec{s}$*. An ORAM construction is secure if:* (i) *For any two data request sequences* $\vec{x}$ *and* $\vec{y}$ *of the same length, the access patterns* $A(\vec{x})$ *and* $A(\vec{y})$ *are computationally indistinguishable to an observer, and* (ii) *it returns the data that is consistent with the input* $\vec{s}$ *with probability* $\geq 1 - \mathsf{negl}(|\vec{s}|)$*. That is, the ORAM fails with only a negligible probability.*

Corollary 1. *Accessing OMAT leaks no information beyond* (i) *the size of rows and columns,* (ii) *whether the row or column dimension being accessed, given that the ORAM scheme being used on top is secure by Definition 1.*

Proof. Let $\mathbf{M}$ be an OMAT structure consisting of two logical tree-based ORAM structures OROW and OCOL as described in Section 3.1 with dimensions M and N, respectively. Let the bit $B = 0$ if the query is on OROW and $B = 1$, otherwise. A construction providing OMAT leaks no information about the location of a node u being accessed in $\mathbf{M}$ beyond the bit B and dimensions (M, N). This is due to the fact that OMAT uses a secure ORAM that satisfies Definition 1 to access each block of OROW and OCOL in $\mathbf{M}$. Thus, as long as the node accessed within OROW or OCOL is not distinguishable from any other node within that OROW and OCOL through the number of access requests, it is indistinguishable by Definition 1. □

Note that the information on whether the row or column was accessed can be hidden by performing a simultaneous row and column access on both dimensions for each query. This poses a security-performance trade-off. One can also hide the size of row and column by setting OMAT matrix with equal dimensions, but this may introduce some cost for certain applications.

Corollary 2. *Accessing OTREE leaks no information about the actual path being traversed, given that the ORAM scheme being used on top is secure by Definition 1.*

Proof. Let $\mathbf{T}$ be a tree data structure of height H. Let $\mathbf{T}_\ell$ be the set of nodes at level $0 \leq \ell \leq H$ in the tree. A construction providing OTREE leaks no information about the location of a node $u \in \mathbf{T}_\ell$ being accessed in the tree beyond that it is from $\mathbf{T}_\ell$. This is due to OTREE uses a secure ORAM that satisfies Definition 1 to access each level of the tree. Thus, as long as a node accessed within level ℓ is not distinguishable from any other node within that level through the number of access requests, it will be indistinguishable according to Definition 1. □

Side-channel leakages in Path-ORAM. There are several side-channel attacks on Path-ORAM (e.g., [1, 10]) when it is executed by the secure CPU playing on behalf of the ORAM client. In this context, since the secure CPU resides in the untrusted party, the adversary has a partial view on it to exploit the timing leakage (e.g., [1]). In our model, we assume that the client is fully trusted and it is totally apart from the adversary view (i.e., untrusted database server). Therefore, we do not consider these side-channel leakages due to the difference between our model and the secure CPU context.

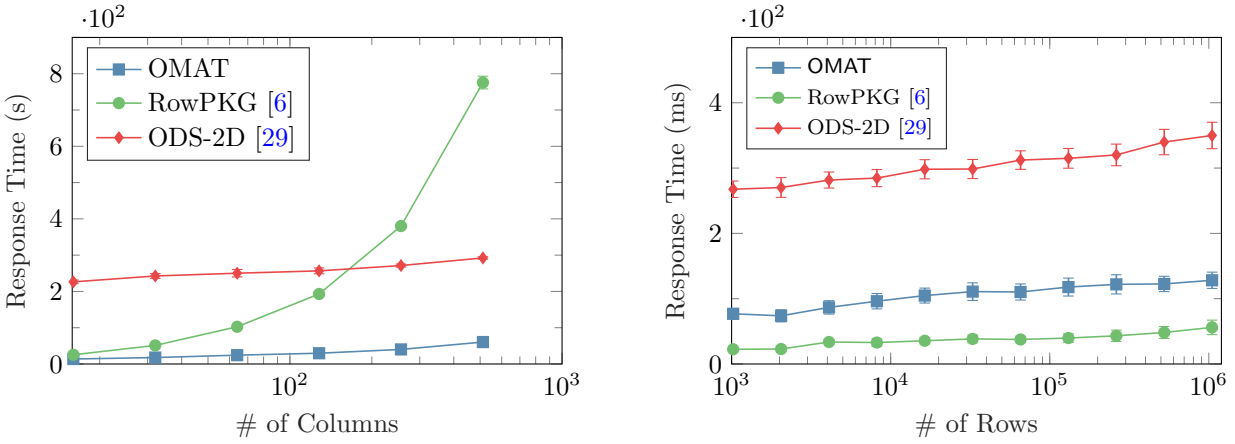
5 Performance Evaluation

5.1 Configurations

• *Implementation:* We implemented our schemes and their counterparts on CURIOUS framework [2]. We integrated additional functionalities into the framework to perform batch read/write operations to prevent unnecessary round-trip delays, and also to communicate with MongoDB instance via MongoDB Java Driver. We chose MongoDB as our database and storage engine. We preferred MongoDB since its Java Driver library is well-documented and easy to use. Moreover, it supports batch updates without restrictions, which is important for consistent performance analysis.

• *Data Formatting:* We created our database table with randomly generated data with a different number of rows, columns, and field sizes. We then used the table to construct tree-based ORAMs for compared schemes. For instance, in OMAT, we created OROW and OCOL structures from this table, while an oblivious tree structure is created for OTREE, as described in Section 3.

• *Experimental Setup and Configurations:* For our experiments, we used two different client machines on two different network settings: (i) A desktop computer that runs CentOS 7.2 and is equipped with Intel Xeon CPU E3-1230, 16 GB RAM; (ii) A laptop computer that runs Ubuntu 16.04 and is equipped with Intel i7-6700HQ, 16 GB RAM. For our remote server, we used AmazonEC2 with t2.large instance



(a) Column-related queries with 2^{15} rows **(b)** Row-related queries with 2^5 columns

Figure 10. End-to-end delay of queries for OMAT and counterparts with high-speed network setting.

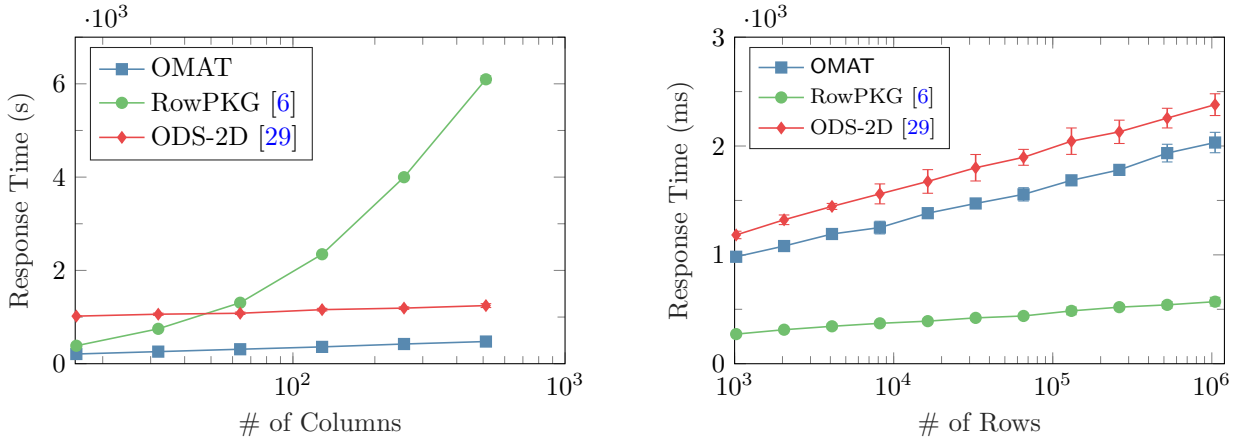
type that runs Ubuntu Server 16.04. While the connection between the desktop and the server was a *high-speed network* with download/upload speeds of 500/400 Mbps and an average latency of 11 ms, the connection between the laptop and the server was a *moderate-speed network* with download/upload speeds of 80/6 Mbps and an average latency of 30 ms.

• *Evaluation Metrics:* We evaluated the performance of our schemes and their counterparts based on the following metrics: (i) The Response time (i.e., end-to-end delay) including decryption, re-encryption and transmission times to perform a query; (ii) Client storage including the size of stash and position map; (iii) Server storage including the size of OMAT or OTREE. We compared the response times of OMAT and its counterparts for both row- and column-related queries (e.g., statistical, conditional). For OTREE and ODS-Tree, we compared the response times of traversing an arbitrary path on the tree-indexed database. To measure the end-to-end delay, we used the `std::chrono` C++ library to get the actual duration at the client side, from the time the client sends the first command until he receives the last response from the Amazon server. For each experiment, we ran 50 times and took the average number as the final response time reported in this section. We now describe our experimental evaluation results and compare our schemes with their counterparts.

5.2 Experimental Results

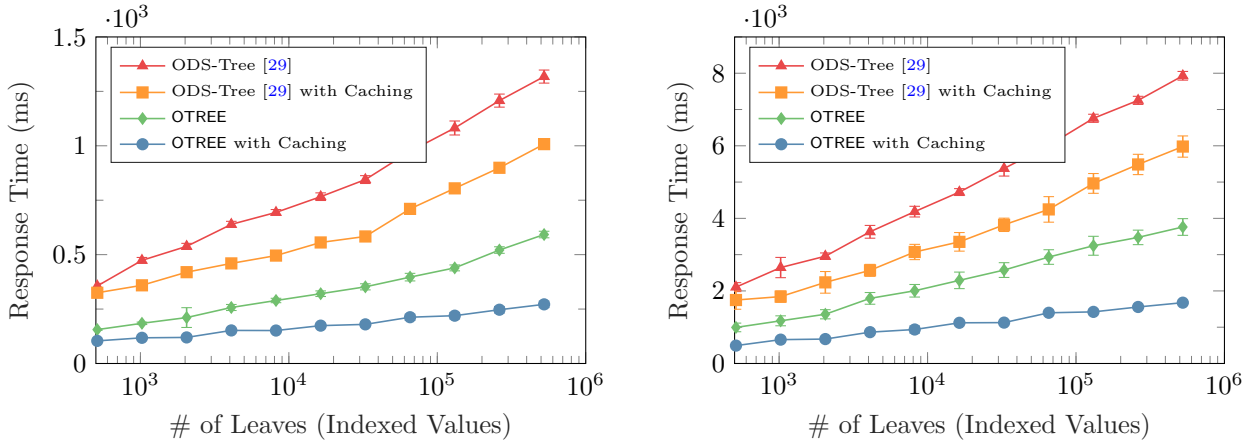
• *Statistical and Conditional Queries (Column-Related):* We first analyze the response time of column-related queries for OMAT, ODS-2D and RowPKG. With these queries, the client can fetch a column from the encrypted database for statistical analysis or a conditional search. Given a column-related query, the total number of bytes to be transmitted and processed by each scheme are shown in Table 1. RowPKG's transmission cost is the size of all ORAM buckets, where $Z \cdot (B \cdot N)$ and $(2M-1)$ denote the bucket size and the total number of buckets, respectively. As for OMAT, its oblivious data structure OCOL allows efficient queries on column dimension with $O(\log(N))$ communication overhead, which outperforms the linear overhead of $O(N)$ of RowPKG. While OMAT and RowPKG can fetch the whole column with one request, it requires $M/4$ synchronous requests for ODS-2D where each request costs $Z \cdot (16 \cdot B_1) \cdot \log_2(M \cdot N/16)$ bytes due to 4×4 clustering of the cells.

We measured the performance of OMAT and its counterparts with arbitrary column queries. In this experiment, we set parameters as $B = 64$ bytes and $Z = 4$. The number of columns N varies from 2^4 to 2^9, where the number of rows is *fixed* to be $M = 2^{15}$. Figures 10a and 11a illustrate the performance of the schemes on two different network settings with two different client machines as described in Section 5.1. For a database table with 2^{10} rows and 2^9 columns, OMAT's average query times are 60 s and 475 s compared to RowPKG's 775 s and 6100 s, and ODS-2D's 292 s and 1245 s on high- and moderate-speed



(a) Column-related queries with 2^{15} rows **(b)** Row-related queries with 2^5 columns

Figure 11. End-to-end delay of queries for OMAT and counterparts with moderate-speed network setting.



(a) Moderate-Speed Network Setting **(b)** High-Speed Network Setting

Figure 12. End-to-end delay of traversal on tree-indexed database for OTREE and ODS-Tree.

networks, respectively. This makes OMAT about 13× faster than RowPKG. While OMAT performs 2.6× faster than ODS-2D on the moderate-speed network, it becomes 4.9× on high-speed network since the latency starts to dominate the response time of ODS-2D with $M/4$ requests due to its construction with pointers.

• *Single Row-Related Queries:* We now analyze the response time of row-related queries for OMAT and its counterparts. Given a row-related query, the total number of bytes to be transmitted and processed by OMAT and its counterparts are summarized in Table 1. For OMAT and RowPKG, $(B \cdot N)$ and $Z \cdot \log_2(M)$ denote the total row size and the overhead of Path-ORAM, respectively. Due to OMAT's OCOL and OROW structures, OMAT is always a constant factor of $Z = 4$ more costly than RowPKG. Clustering strategy of ODS-2D also introduces more cost and makes ODS-2D 4.2× more costly than RowPKG when $N = 32$.

We measured the performance of OMAT and its counterparts with arbitrary row queries, where the number of rows M varies from 2^{10} to 2^{20}. The block size is $B = 128$ bytes and the number of columns is fixed as $N = 32$. By this setting, the total row/record size is $B \cdot N = 4096$ KB. Figures 10b and 11b illustrate the performance of the compared schemes for both network settings. We can see that OMAT performs slower than RowPKG by a constant factor of approximately 2.3× and 3.6× on high

and moderate-speed network, respectively. As for ODS-2D, Figure 10b explicitly shows the effect of the round-trip delay introduced by network latency on ODS-2D due to $N/4$ synchronous requests. Although ODS-2D has similar cost with OMAT, it performs approximately 220 ms and 380 ms slower than OMAT.

• *Traversal on Tree-indexed Database:* We analyze the response time of oblivious traversal on database index that is constructed as a range tree by putting distinct values of a column to the leaf of the tree. Figure 9 exemplifies the constructed range tree, and this structure is used along with its linked list to perform conditional queries (e.g., equality, range) on an indexed column, and fetch matching IDs. We compare our proposed OTREE and ODS-Tree with no caching and half-top caching strategies.

Given a database index tree constructed with values of the column, the total number of bytes to be transmitted and processed by OTREE and ODS-Tree without caching are $Z_2 \cdot B \cdot (H+1) \cdot (H+2)$ and $2 \cdot Z_1 \cdot B \cdot (H+1)^2$, respectively, where H is the height of tree data structure. While ODS traverses the tree with $O(H)$, the additional overhead of Path-ORAM makes the total overhead to be $O(H^2)$. As for OTREE, its level restriction on ORAM storage reduces the transmission overhead by 1.6×. With half-top caching strategy, overheads of both schemes reduce as shown in Table 1, however, OTREE's construction benefits more from caching by performing traversal 3.2× less costly than ODS-Tree.

For this experiment, we set the block size $B = 4$ KB, the number of blocks inside a bucket for ODS-Tree is $Z_1 = 4$, and the number of blocks inside a bucket for OTREE is $Z_2 = 5$ (see Section 3.2 for the stability analysis). We benchmarked OTREE and ODS-Tree with arbitrary equality queries when the number of indexed values varies from 2^9 to 2^{19}. The number of indexed values is set to 2^{19} for large database setting. For both network settings, Figure 12 demonstrates the effect of half-top caching strategy and how the structure of OTREE gives more leverage in response time. While OTREE without caching performs around 2× faster than its counterpart, caching allows OTREE to perform 3.6× faster than ODS-Tree with caching for both network settings.

5.3 Client and Server Storage

We now analyze the client storage overhead of our schemes and their counterparts. The position map of OMAT requires $O((M+N) \cdot \log(M+N))$ storage, while RowPKG requires $O(M \cdot \log(M))$, since only the position map of rows are stored. However, the dominating factor is M, since large databases have more rows than columns. ODS's pointer technique allows it to operate with O(1) storage for position map. Moreover, the worst-case stash size changes with the query type, because stash is also used to store currently fetched data and the worst-case storage costs are summarized in Table 1. For row-related queries, the worst-case stash storage is the same for both OMAT and RowPKG but ODS-2D requires more storage due to clustering. For column-related queries, RowPKG requires storing $O(M \cdot N)$ that corresponds to all ORAM buckets. Besides the query performance issues, this also makes RowPKG infeasible for very large databases to perform column-related queries. In addition, ODS-2D also requires $O(\log(M))$ times more client storage compared to OMAT. While RowPKG and ODS-2D have the same server storage size, OMAT requires constant $Z\times$ more storage due to additional dummy blocks.

Since OTREE and ODS do not require the position map to operate, the client storage consists of the stash and additionally cached block according to the caching strategy used. For the worst-case, both schemes have the same client storage with the same caching strategy; however, the stash of OTREE may be more loaded than ODS as shown in Figure 8 due to its level restriction. Moreover, server storage of OTREE is 2× less than ODS, since Path-ORAM of ODS requires one more level than OTREE.

6 Conclusions

In this paper, we introduced two new oblivious data structures called OMAT and OTREE. The proposed techniques can be instantiated with any tree-based ORAM scheme to enable efficient private queries on database instances. OMAT enables various statistical and conditional queries on generic database tables, which may be highly inefficient for its counterparts relying rely on row-oriented packaging. On the other hand, OTREE provides more efficient range queries on tree-indexed database than existing ODS

techniques, and also receives more benefit from caching optimizations. These properties allow OMAT and OTREE to be ideal data structures to construct oblivious database services on the cloud, which offers high security and privacy guarantee for the users.

References

[1] C. Bao and A. Srivastava. Exploring timing side-channel attacks on path-orams. In *2017 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pages 68–73. IEEE, 2017.

[2] V. Bindschaedler, M. Naveed, X. Pan, X. Wang, and Y. Huang. 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*, pages 837–849. ACM, 2015.

[3] N. Cao, C. Wang, M. Li, K. Ren, and W. Lou. Privacy-preserving multi-keyword ranked search over encrypted cloud data. *IEEE Transactions on parallel and distributed systems*, 25(1):222–233, 2014.

[4] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart. Leakage-abuse attacks against searchable encryption. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 668–679. ACM, 2015.

[5] D. Cash, J. Jaeger, S. Jarecki, C. S. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. *IACR Cryptology ePrint Archive*, 2014:853, 2014.

[6] Z. Chang, D. Xie, and F. Li. Oblivious ram: a dissection and experimental evaluation. *Proceedings of the VLDB Endowment*, 9(12):1113–1124, 2016.

[7] B. Chen, H. Lin, and S. Tessaro. Oblivious parallel ram: Improved efficiency and generic constructions. In *Theory of Cryptography Conference*, pages 205–234. Springer, 2016.

[8] J. Dautrich and C. Ravishankar. Combining oram with pir to minimize bandwidth costs. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, pages 289–296. ACM, 2015.

[9] S. Devadas, M. van Dijk, C. W. Fletcher, L. Ren, E. Shi, and D. Wichs. Onion oram: A constant bandwidth blowup oblivious ram. In *Theory of Cryptography Conference*, pages 145–174. Springer, 2016.

[10] C. W. Fletcher, L. Ren, X. Yu, M. Van Dijk, O. Khan, and S. Devadas. Suppressing the oblivious ram timing channel while making information leakage and program efficiency trade-offs. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 213–224. IEEE, 2014.

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

[12] T. Hoang, A. Yavuz, and J. Guajardo. Practical and secure dynamic searchable encryption via oblivious access on distributed data structure. In *Proceedings of the 32nd Annual Computer Security Applications Conference (ACSAC)*. ACM, 2016.

[13] M. S. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *Annual Network and Distributed System Security Symposium – NDSS*, volume 20, page 12, 2012.

[14] S. Kamara, C. Papamanthou, and T. Roeder. Dynamic searchable symmetric encryption. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pages 965–976. ACM, 2012.

[15] J. Katz and Y. Lindell. *Introduction to Modern Cryptography*. Chapman & Hall/CRC, 2007.

[16] C. Liu, L. Zhu, M. Wang, and Y.-a. Tan. Search pattern leakage in searchable encryption: Attacks and new construction. *Information Sciences*, 265:176–188, 2014.

[17] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, and D. Song. Phantom: Practical oblivious computation in a secure processor. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 311–324. ACM, 2013.

[18] B. Pinkas and T. Reinman. Oblivious ram revisited. In *Advances in Cryptology–CRYPTO 2010*, pages 502–519. Springer, 2010.

[19] R. A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 85–100. ACM, 2011.

[20] R. A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan. Cryptdb: processing queries on an encrypted database. *Communications of the ACM*, 55(9):103–111, 2012.

[21] D. Pouliot and C. V. Wright. The shadow nemesis: Inference attacks on efficiently deployable, efficiently searchable encryption. In *Proceedings of the 2016 ACM Conference on Computer and Communications Security*. ACM, 2016.

[22] L. Ren, C. W. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. van Dijk, and S. Devadas. Ring oram: Closing the gap between small and large client storage oblivious ram. *IACR Cryptology ePrint Archive*, 2014:997, 2014.

[23] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li. Oblivious ram with o ((logn) 3) worst-case cost. In *Advances in Cryptology–ASIACRYPT 2011*, pages 197–214. Springer, 2011.

[24] E. Stefanov, E. Shi, and D. Song. Towards practical oblivious ram. *arXiv preprint arXiv:1106.3652*, 2011.

[25] E. Stefanov, M. Van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path oram: an extremely simple oblivious ram protocol. In *Proceedings of the 2013 ACM SIGSAC conference on Computer and Communications security*, pages 299–310. ACM, 2013.

[26] B. Wang, Y. Hou, M. Li, H. Wang, and H. Li. Maple: scalable multi-dimensional range search over encrypted cloud data with tree-based index. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 111–122. ACM, 2014.

[27] B. Wang, M. Li, and H. Wang. Geometric range search on encrypted spatial data. *IEEE Transactions on Information Forensics and Security*, 11(4):704–719, 2016.

[28] X. Wang, H. Chan, and E. Shi. Circuit oram: On tightness of the goldreich-ostrovsky lower bound. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 850–861. ACM, 2015.

[29] X. S. Wang, K. Nayak, C. Liu, T. Chan, E. Shi, E. Stefanov, and Y. Huang. Oblivious data structures. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 215–226. ACM, 2014.

[30] A. A. Yavuz and J. Guajardo. Dynamic searchable symmetric encryption with minimal leakage and efficient updates on commodity hardware. In *Selected Areas in Cryptography – SAC 2015*, Lecture Notes in Computer Science. Springer International Publishing, August 2015.

[31] H. Yin, Z. Qin, J. Zhang, L. Ou, and K. Li. Achieving secure, universal, and fine-grained query results verification for secure search scheme over encrypted cloud data. *IEEE Transactions on Cloud Computing*, 2017.

[32] R. Zhang, R. Xue, L. Liu, and L. Zheng. Oblivious multi-keyword search for secure cloud storage service. In *Web Services (ICWS), 2017 IEEE International Conference on*, pages 269–276. IEEE, 2017.

[33] Y. Zhang, J. Katz, and C. Papamanthou. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 707–720, Austin, TX, 2016.