OBI: a multi-path oblivious RAM for forward-and-backward-secure searchable encryption

Zhiqiang Wu
Changsha University of Science and Technology
cxiaodiao@hnu.edu.cn

Rui Li
Dongguan University of Technology
ruili@dgut.edu.cn

***Abstract*—Dynamic searchable encryption (DSE) is a user-cloud protocol for searching over outsourced encrypted data. Many current DSE schemes resort to oblivious RAMs (ORAM) to achieve forward privacy and backward privacy, which is a concept to describe security levels of the protocol. We show that, however, most prior ORAM-based DSE suffers from a new problem: it is inefficient to fetch/insert a large set of data blocks. We call this the large-stash eviction problem. To address the problem, we present OBI, a multi-path Oblivious RAM, which accesses multiple tree paths per query for handling a large set of data blocks. We classify traditional tree-based ORAMs as single-path ORAMs if they access a single path per query. OBI has two new high-throughput multi-path eviction algorithms that are several orders of magnitude more efficient than the well-known PATH-ORAM eviction algorithm when the stash is large. We prove that the proposed multi-path ORAM outperforms the traditional single-path ORAM in terms of local stash size and insertion efficiency. Security analysis shows that OBI is secure under the strong forward and backward security model. OBI can protect the well-known DSE leakage, such as the search pattern and the size pattern. We also show that OBI can be applied to oblivious file systems and oblivious conjunctive-query DSE schemes. We conduct experiments on the Enron dataset. The experimental results demonstrate that OBI is far more efficient than the state-of-the-art ORAM-based DSE schemes.**

I. INTRODUCTION

A. Background and Motivation

Nowadays, many corporations and users outsource their sensitive data to cloud servers for low costs and global services. To search for documents efficiently, the users usually put data into traditional databases, such as Lucene [1] and Cassandra [2]. To protect privacy, the users should encrypt their private data before outsourcing since hackers and honest-but-curious network managers can break the cloud system. The users usually adopt two types of encryption techniques, oblivious RAM (ORAM) [3]–[5] and dynamic searchable encryption (DSE) [6], [7] . ORAM supports oblivious reading and writing without leaking the access pattern at the cost of continuously shuffling and re-encrypting the accessed data. DSE is a user-cloud protocol that provides efficient search and update services on encrypted data. DSE generally achieves excellent efficiency at the cost of leaking some search/access/size patterns, which are information relating to how often a keyword is being queried, how frequently the same location is being accessed, and how large result size is, respectively.

An ORAM is logically considered a key-value storage structure that aims to hide the access pattern. There are two types of ORAMs, tree-based ORAMs and layer-based ORAMs. The tree-based ORAM has smaller communication and computation cost in data shuffle operation than the layer-based ORAM based on the research of [8]. PATH ORAM [5] is a typical tree-based ORAM since it is compact, low-cost, and has been deployed on resource-constrained devices [9]. Many DSE schemes resort to the tree-based ORAMs to achieve forward privacy or backward privacy [10]–[13], which is a concept relating to the data-search and data-update privacy leakage. It is well known that, however, adopting an ORAM implies inefficiency. One reason is that these schemes rely on the ORAM that adopts a single-path accessing algorithm to read/re-encrypt/write a tree path. This type of ORAM might as well be named a single-path ORAM.

A problem existing in the single-path ORAM is that it is inefficient to fetch/insert a large set of data blocks. There are two approaches to reading a set of data blocks existing in an ORAM tree: reading a single path or reading multiple paths. Due to encryption, the set of data blocks is generally randomly distributed in multiple paths of the ORAM tree. To fetch the blocks, we should access all the paths. If we repeatedly read a path containing the data block by using the single-path ORAM algorithm, this will incur high round complexity. In cloud environment, one read implies a user-cloud interaction. Retrieving a large result set with this approach means high interactions. Another approach is to read multiple paths containing all the desired data blocks in one time (or many times). To the best of our knowledge, this approach is not well studied by researchers currently since this operation might incur poor efficiency. If an ORAM accesses multiple paths per query, we call this type of tree-based ORAM a multi-path ORAM, which has a multi-path eviction algorithm to read/re-encrypt/write the set of accessed paths. Our motivation is to study the multi-path ORAMs for improving ORAM-based DSE efficiency.

B. Limitations of Prior Art

The prior non-ORAM-based DSE schemes, such as [14], [15] are vulnerable to the search-pattern-based attacks [16], [17] since they have search pattern leakages. The prior ORAM-based DSE schemes, SD_d in [10], Horus in [11], Moneta in [12], Orion in [11], and Eurus in [13] suffer from the large-stash eviction problem that remains unsolved. Informally, the problem is stated as follows: when we search, insert, or delete

The corresponding author is Rui Li.

Network and Distributed System Security (NDSS) Symposium 2023
27 February - 3 March 2023, San Diego, CA, USA
ISBN 1-891562-83-5
https://dx.doi.org/10.14722/ndss.2023.24423
www.ndss-symposium.org

TABLE I. COMPARISONS OF TYPICAL FORWARD-AND-BACKWARD-SECURE DSE SCHEMES

Scheme	Search		Update		CS	FP	BP	SPH	APH	ZPH	OPH
	CC	RT	CC	RT							
SD$_d$ [10]	$O(a_w + \log N)$	1	$O(q \log^3 N)$	$O(q \log N)$	$O(1)$	$\surd$	$\surd$	$\times$	$\times$	$\times$	$\times$
CLOSE-FB [14]	$O(a_w + C)$	1	$O(q \cdot C)$	1	$O(1)$	$\surd$	$\surd$	$\times$	$\times$	$\times$	$\times$
Zuo et al. [15]	$O(a_w)$	1	$O(q)$	1	$O(m)$	$\surd$	$\surd$	$\times$	$\times$	$\times$	$\times$
Horus [11]	$O(n_w \log d_w \log N)$	$O(\log d_w)$	$O(q \log^2 N)$	$O(q \log N)$	$O(m)$	$\surd$	$\times$	$\surd$	$\times$	$\times$	$\times$
Moneta [12]	$\widetilde{O}(a_w \log N + \log^3 N)$	2	$\widetilde{O}(q \log^2 N)$	$O(q)$	$O(1)$	$\surd$	$\surd$	$\surd$	$\times$	$\times$	$\times$
Orion [11]	$O(n_w \log^2 N)$	$O(\log N)$	$O(q \log^2 N)$	$O(q \log N)$	$O(1)$	$\surd$	$\surd$	$\surd$	$\surd$	$\times$	$\times$
OBI (this paper)	$O(r_w \log N)$	1	$O(q \log N)$	1	$O(m)$	$\surd$	$\surd$	$\surd$	$\surd$	$\times$	$\times$
Eurus [13]	$O(M^2 \log^2 m)$	1^*	$O(M^2 \log^2 m)$	1^*	$O(m)$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\times$
ZPH-OBI (this paper)	$O(M \log N)$	1	$O(M \log N)$	1	$O(m)$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$

N is the total number of keyword-file-identifier pairs, q is the number of keyword-file-identifier pairs to be inserted, and m is the number of distinct keywords. M is the maximum number of matched file identifiers in the index. $M = max_{w \in W}|DB(w)|$, $DB(w)$ is a set of file identifiers matching keyword w, and W is all the keywords that can be queried. n_w is the number of real files containing keyword w, a_w is the number of entries matching keyword w, d_w is the number of deletions, and r_w is the number of entries matching keyword w after the last search on w. $a_w \geq r_w \geq n_w$. RT denotes the number of interaction round trips. CC denotes Computation cost and Communication cost per query. CS denotes client storage. FP denotes forward privacy. BP denotes backward privacy. SPH denotes Search-Pattern-Hiding. APH denotes Access-Pattern-Hiding. ZPH denotes siZe-Pattern-Hiding. OPH denotes OPeration-kind-Hiding, which captures the information of $op \in \{search, add, del\}$. Update complexity is given per q keyword-file-identifier pairs. C is a constant value that denotes the maximum number of updates. (1^*) denotes a client-server round trip and many server-server interactions. $\widetilde{O}$ denotes an approximation.

a large number of data blocks, the local stash will contain a large number of blocks to be re-encrypted and evicted. We call this the large-stash eviction problem, which might incur high round complexity or poor insertion efficiency. SD$_d$, Horus, and ORION make black-box use of the oblivious map (OMAP) [18], an oblivious data structure embedded in PATH ORAM [5]. Undoubtedly, they share the inherent drawback of the OMAP that requires $O(\log N)$ user-cloud rounds per access, where N is the number of keyword-file-identifier pairs. Thus, all these three schemes need $O(q \log N)$ rounds for inserting q pairs into the index, which is extremely expensive in the cloud environment. Moneta [12] still has insertion pattern leakage, though it adopts TWORAM [19]. PATH ORAM, OMAP, and TWORAM are all single-path ORAMs.

Liu et al. proposed Eurus [13], a DSE scheme for protecting the DSE size pattern. Their encryption approach relies on S^{3}ORAM [20] and padding. Since data padding incurs a large number of pairs temporarily stored in the stash, the large-stash eviction problem appears, resulting in unscalable search/update time complexity that we list in Table I. Eurus can be viewed as a single-path ORAM since it accesses only one tree path per query.

C. Proposed Approach

In this paper, we propose OBI, a multi-path ORAM. We embed an inverted index into the ORAM tree, the structure of the non-recursive PATH ORAM [5], for oblivious key-set mapping. We introduce the following new techniques to address the large-stash eviction problem.

Reducing position map. A position map is a storage structure that stores locations of randomly-distributed keyword-file-identifier pairs. We randomly inserted all the keyword-file-identifier pairs into the ORAM tree in the initial stage. To save insertion locations, we employ a locally-stored keyword hash table, which maps each keyword to the corresponding keyword information, including the number of searches and identifiers of this keyword. In addition, we use a pseudo-random function to map the keyword information to a set of locations of the keyword. This computation helps us to avoid storing the large non-recursive PATH-ORAM position map of size $O(N)$. The user maintains only the keyword hash table of size $O(m)$, where m is only the number of keywords, and it is much smaller than N.

Reducing eviction complexity. PATH ORAM takes $O(r^2)$ time to evict r data blocks in the stash when r is large. The eviction algorithm is quite inefficient. To address this, we design two algorithms, a k-Nearest Neighbour Eviction Algorithm (KNNEA) and a Partition-Based Eviction Algorithm (PBEA). KNNEA employs a sorted array to improve eviction time complexity. PBEA partitions the ORAM tree into subtrees to improve eviction time complexity since each subtree can be processed individually. PBEA reduces the eviction time complexity of PATH ORAM from $O(r^2)$ to $O(r \log N)$.

Based on KNNEA and PBEA, we implement two schemes, OBI and ZPH-OBI. OBI is a multi-path ORAM and also a DSE scheme without the search and access pattern leakage. ZPH-OBI is a siZe-Pattern-Hiding scheme, which reduces computational complexity of Eurus from $O(M^2 \log^2 m)$ to $O(M \log N)$ (in Table I).

D. Our Contributions

- We propose OBI, a multi-path ORAM for oblivious key-set mapping. We prove that the stash overflow probability decreases exponentially in r, which is the number of multiple paths per query.
- OBI is the first DSE scheme that satisfies the conditions of leaking no search and access patterns, getting data in single-round-trip access, and achieving quasi-optimal search efficiency in the worst-case.
- We propose two high-throughput multi-path eviction algorithms, KNNEA and PBEA. KNNEA is far more efficient than the PATH ORAM eviction algorithm [5], [9] when the stash is large. PBEA supports single-round-trip large-batch evictions with quasi-optimal time complexity.
- We propose ZPH-OBI scheme to hide both operation kinds and size patterns of DSE.
- We give an oblivious file system prototype, and an oblivious conjunctive-query DSE scheme.

II. RELATED WORK

Searchable encryption (SE) schemes can be classified into dynamic schemes and static ones, where dynamic schemes support updating operations and static schemes cannot. Most of early SE works are static, such as [21], [22]. Compared with static SE, dynamic SE (DSE) faces more challenges because there are new privacy leakages. A newly inserted data maybe reveal the previous issued queries, and a later issued query maybe accesses deleted data, which were studied in [23]–[25] and [11], [15], [26], respectively.

Naveed initiated the formal study of using ORAM to reduce access pattern leakage for SE [27]. Naveed claimed that eliminating leakage in SE is impossible, and achieving even weaker classes of leakage either requires communication more than downloading the entire database or does not provide meaningful leakage reduction. There are many new ORAM choices for DSE, such as [5], [18], [28]–[33]. Unfortunately, as mentioned above, there are still many big challenges that prevent DSE from using the ORAMs. If naively employing the ORAMs to hide the search/access patterns, the scheme suffers from new problems, such as a large user-side position map, high round complexity [34], or heavy computational overhead [19]. Upgrading DSE to oblivious DSE is still challenging, since DSE provides not only single-keyword queries, but also Boolean queries [7], [35]–[38], range queries [39]–[44], fuzzy queries [45], [46], etc. Chang et al. investigated oblivious range and kNN queries in [47]. They use oblivious batch processing and caching techniques to improve throughput, but the scheme is still highly interactive. TaoStore [48] is built on top of a tree-based ORAM scheme that processes client requests concurrently and asynchronously in a non-blocking fashion. ConcurORAM [49] is the first ORAM to achieve parallelism for stateless ORAM clients without the need for direct inter-client communication. There are some ORAMs that partially protect data-write privacy or data-read privacy [50], [51].

Another direction for protecting access patterns is to adopt secure hardware enclaves. ObliDB [52] is an enclave-based oblivious database engine that runs general relational read workloads over multiple access methods. ObliDB makes black-box use of the non-recursive Path ORAM [5], whose client-side is stored in a trusted hardware enclave, and whose server-side resides in untrusted memory. However, the subsequent studies [34], [53] showed that simply putting the ORAM client inside the enclave is insecure. To address the challenge, Oblix [34] uses a doubly-oblivious ORAM, which guarantees that the accesses to the ORAM server are oblivious, but also those to the ORAM client's internal memory. Krastnikov et al. proposed an oblivious algorithm for database equi-joins using sorting networks in [54].

III. Notations and Definitions

A. DSE Framework

A DSE scheme consists of three polynomial-time user-cloud protocols $(Setup, Search, Update)$. We assume that the user is trusted and the cloud is honest but curious. The cloud can honestly execute the protocols, but it always wants to obtain user's private data. In the $Setup$ protocol, the user first builds and uploads an index generated by encrypting a set of plain-text files. In the $Search$ protocol, the user sends an encrypted query relating to a keyword w. Then the cloud searches the index and returns the results $DB(w)$ that matches the query, where $DB(w)$ denotes the set of file identifiers of files containing w. In the $Update$ protocol, the user also sends an encrypted query to add or delete a set of keyword-file-identifiers from the index. All the protocols perhaps require multiple rounds. To protect access patterns, we store the encrypted data files in a file ORAM. We use following two steps to retrieve files in a DSE scheme. First, we obtain the file identifiers of the files obliviously. Second, we retrieve the files from the file ORAM via the file identifiers. We focus on file identifier retrieving in this paper.

B. DSE Security and Privacy

We adopt the adaptive security definition studied in [6], [12], [21]. A DSE scheme parameterized by a stateful leakage function $\mathcal{L} = \{\mathcal{L}^{Setup}, \mathcal{L}^{Query}\}$ is said to be $\mathcal{L}$-adaptively-secure, if for any probabilistic polynomial-time (PPT) adversary $\mathcal{A}$, there exists a PPT simulator $\mathcal{S}$ such that the following two executions are PPT computationally indistinguishable. In the real execution, $\mathcal{A}$, who can adaptively issue a polynomial number of queries $\{search, add, del\}$ without accessing the user's private data, is initially given a real encrypted database. Next, $\mathcal{A}$ adaptively issues queries for attacks. In the ideal execution, $\mathcal{S}$ initially generates a simulated encrypted database $\mathcal{S}(\mathcal{L}^{Setup})$. Next, $\mathcal{S}$ adaptively performs simulated queries generated by $\mathcal{S}(\mathcal{L}^{Query})$. Adaptive security guarantees that even if the adversary can adaptively choose keywords for attacks, the adversary learns no more than the information described by the predefined leakage function $\mathcal{L}$, which contains only a part of the search/access/size/kind patterns.

The search pattern is the information relating to how frequently a keyword is searched. It is defined as $sp(w) = \{i : (i, w) \in Q\}$, where Q is the query list. The access pattern is a set of result file identifiers and timestamps for accessing data files or the index. Note that the search/access patterns can, perhaps, be leaked or partially leaked by both search or update operations. Search/access-pattern-hiding means the DSE protocol leaks nothing of $sp(w)$ or access patterns from any operations, respectively. Size pattern is the result size of a query. Operation kind pattern is the information of $op \in \{search, add, del\}$.

We also adopt the forward privacy and backward privacy definitions studied in [12], [13], [55]. An adaptively-secure DSE scheme has forward privacy only if update queries leak no more than the information about operation kinds, update identifiers, and the number of keywords to be updated. Forward privacy implies that update queries can not expose historically searched records. An adaptively-secure DSE scheme has insertion-pattern-revealing backward privacy only if a search operation leaks no more than the information about insertion time stamps, insertion file identifiers, and the result size. Backward privacy implies that search queries can not match historically deleted records.

Strong forward and backward security model. Let $\boldsymbol{D}$ be a set of keyword-file-identifier pairs to be updated, where $\boldsymbol{D} = \{(w_1, id_1), \cdots, (w_r, id_r)\}$. An $\mathcal{L}$-adaptively-secure DSE scheme is strong Forward and Backward (strong-FB) secure if the query leakage function can be written as $\mathcal{L}^{Query}(op, \boldsymbol{D}) = (op, r)$, where $r = |\boldsymbol{D}|$ is called the size pattern and $op = \{search, add, del\}$ the operation-kind pattern. If op=$search$ to search for w, the input is $(search, \{(w, \perp), \cdots, (w, \perp)\})$. If op=add or del, the input is (add, $\boldsymbol{D}$) or (del, $\boldsymbol{D}$), respectively. The strong-FB security can protect the search and access patterns of DSE. If $\mathcal{L}^{Query}$ can be written as $\mathcal{L}^{Query}(op, \boldsymbol{D}) = (M)$, where $M = max_{w \in W}(|DB(w)|)$, the scheme achieves operation-hiding-FB security, which further protects both operation kinds and the size pattern.

C. *Oblivious Data Structures*

The tree-based ORAMs can be classified into single-path ORAMs and multi-path ORAMs.

Single-path ORAM. It is a tree-based user-cloud protocol, where the user can access only one path of the ORAM tree stored in the cloud per query. Most prior single-path ORAMs are logically considered a key-value store structure that aims to hide the data access pattern. We give a brief overview of the well-known PATH ORAM (for more details, see [5], [9]). A non-recursive PATH ORAM has three parts, a full binary tree stored in the cloud, a position map stored in local trusted memory, and a local stash. Each tree node can hold Z data blocks. Initially, each data block is mapped to a random path, whose leaf identifier is stored in the local position map. To retrieve one block, PATH ORAM should look up the position map first for obtaining the leaf identifier. PATH ORAM downloads the whole leaf-to-root path and writes it into the stash. To hide the access pattern, PATH ORAM reallocates a new random leaf identifier for the accessed block. Finally, PATH ORAM re-encrypts the path using an eviction algorithm, which evicts the stash entries to a single path. A recursive PATH ORAM can reduce the position map size at the cost of several interactions. We regard the recursive PATH ORAM also as a single-path ORAM since it accesses only one tree path every time. Note that, if an ORAM $\mathcal{O}$ consists of a set of single-path ORAMs, such as [48], [49], [56], where each ORAM is executed in parallel, we still consider $\mathcal{O}$ as a single-path ORAM.

Multi-path ORAM. A multi-path ORAM is a tree-based user-cloud protocol, where the user can access r paths of the ORAM tree stored in the cloud per query, and r ($r > 1$) is a constant. This type of ORAM must have a multi-path eviction algorithm, which evicts the stash entries to the accessed multiple tree paths. The multi-path ORAM is mainly designed for oblivious key-set mapping. The multi-path ORAM is secure if for any two access sequences to the tree with the same length, the two executions are probabilistic polynomial-time computationally indistinguishable. The ORAM security implies that all the accessed paths must be indistinguishable from random. In the next sections, we show that the multi-path ORAM has the merit of smaller stash size, compared with the single-path ORAM.

A key-set mapping multi-path ORAM. Most traditional ORAMs consider only oblivious key-value mapping. In modern databases, most data blocks work in bulk access mode. Without the key-set mapping ORAM, the data blocks can be accessed only one by one. Thus, we study the following key-set mapping ORAM for improving throughput.

Given a plain-text inverted index DB, we encrypt and embed it into the ORAM tree. For each keyword w (or called a key) in the index, we assume the result set $DB(w)$ are randomly distributed in multiple paths of the ORAM tree. A key-set mapping multi-path ORAM is a tree-based strong-FB-secure DSE scheme for mapping each w to $DB(w)$ efficiently without leaking the search and access patterns from DSE operations $op \in \{search, add, del\}$. This structure can be viewed as an oblivious inverted index. Note that the keyword w and the file identifier id can be arbitrary fixed-size values for different purposes. The key-set mapping multi-path ORAM is not equivalent to the oblivious parallel RAMs (OPRAMs) [48], [49], [56] since the key-set mapping ORAM works in single-thread mode.

IV. A Key-Set Multi-Path ORAM: An Implementation

A. *OBI Data Structures*

To obliviously and efficiently perform DSE operations, such as searching and updating, we redesign the data structures of the tree-based ORAM. There are three data structures, an encrypted ORAM tree stored in the cloud, a local keyword hash table, and a local stash, denoted by $(\boldsymbol{O_T}, \boldsymbol{H_T}, \boldsymbol{S_T})$, respectively, as shown in Figure 1. $\boldsymbol{O_T}$ is used to save the encrypted DSE index. $\boldsymbol{H_T}$ and $\boldsymbol{S_T}$ are designed for the user to generate encrypted queries.

The ORAM tree $\boldsymbol{O_T}$ is an encrypted full binary tree with the following properties: 1) Each tree node has Z triplets. 2) Each triplet has three fields $(key, value, leaf)$, where $(key, value)$ is a key-value pair for storing arbitrary fixed-size data, and $leaf$ is a leaf identifier that indicates the current triplet exists in the $leaf$-to-root path. The value field of the triplet is called a data block. Each data block consists of u file identifiers. The user encrypts the whole tree node by using the private-key counter-mode AES algorithm. The user stores the tree $\boldsymbol{O_T}$ into an array such that the cloud can efficiently retrieve an entire encrypted node by a node identifier or a path. The tree is initially empty with sufficient preallocated memory to hold all the triplets.

The local keyword hash table, denoted by $\boldsymbol{H_T}$, is a data structure designed for mapping each keyword w to a structure named keyword information, denoted by $K_I = (len, count)$, where len is the length of blocks for w (i.e., the number of data blocks in the index with the key w), and $count$ is the number of searches for w. Every time the user searches for w, $\boldsymbol{H_T}[w].count$ is increased by one. This value is used to re-encrypt accessed triplets. The local keyword hash table can be considered as the position map of PATH ORAM since $\boldsymbol{H_T}$ is used to save the positions of all the triplets stored in the cloud. The difference between these two structures is that not all triplet leaves are saved in $\boldsymbol{H_T}$. In comparison, PATH ORAM should save all the triplet positions in the position map. We later show how to compute a triplet leaf.

The stash, denoted by $\boldsymbol{S_T}$, is a data structure stored in a hash table for mapping each key to a triplet $(key, value, leaf)$. Given a triplet t, the stash saves t by $\boldsymbol{S_T}[t.key] \leftarrow t$. Note that we can not discard the leaf field of t when t is in the stash since $t.leaf$ denotes the future position of t in the ORAM tree, though it is in the stash now. When t has been evicted into the tree, t surely exists in the $(t.leaf)$-to-root path since our eviction algorithms provide this feature.

The ORAM tree supports only one operation, $ReadAndReplace(\{x_1, x_2, \cdots, x_r\})$, where each x_i is a leaf identifier generated by the user every time. The protocol includes three steps: 1) the user computes $\{x_1, x_2, \cdots, x_r\}$ and sends them to the cloud for downloading all the paths, x_1-to-root, $\cdots$, and x_r-to-root path; 2) the user decrypts, writes all the paths into the stash, and re-encrypts them by using a multi-path eviction algorithm; 3) the cloud replaces all the accessed tree paths with the newly encrypted paths.

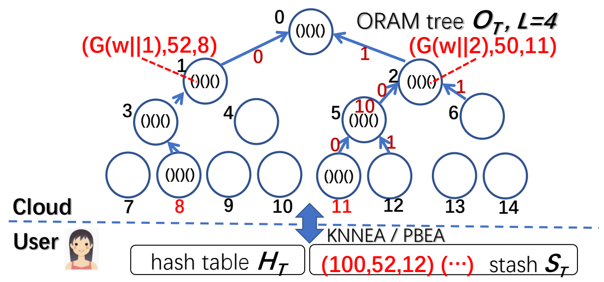

Fig. 1. An OBI overview

We use the following ORAM-tree eviction principle. Let t be a triplet stored the stash. Let x be a leaf identifier, corresponding to one of the currently accessed paths. Let $\mathcal{P}(x,j)$ denote the path of the node that exists in the j-th level of the x-to-root path. We want to evict t into the currently accessed paths. For a tree node that exists in the j-th level of the x-to-root path, if t can be evicted to this node, we assume the following equation always holds,

$$\mathcal{P}(x,j) = \mathcal{P}(t.leaf,j). \tag{1}$$

This equation will help the user efficiently access a data block after data shuffling. For example, in Figure 1, one of the currently accessed paths is $x = 11$, and $\mathcal{P}(11,3) =$ '10' corresponding to Node 5. A triplet stored in the stash is $t = (100,50,12)$. Since $\mathcal{P}(11,3) = \mathcal{P}(12,3)$, t can be evicted into Node 5. The triplet t can be also evicted into Node 2 or Node 0 only if the node is not full. However, t can not be evicted into Node 11 since $\mathcal{P}(11,4) \neq \mathcal{P}(12,4)$. Note that t can not be evicted into non-accessed paths. Otherwise, access pattern leakage will be induced.

Embedding an inverted index. We embed a plain-text inverted index DB into the ORAM to build an oblivious inverted index. There are two steps. Assume $DB(w) = \{id_1, id_2, \cdots, id_r\}$. First, we convert each keyword-file-identifier pair (w, id_i) ($i \in [1,r]$) of DB into triplet $t = (key, value, leaf)$. Let F be a keyed collision-resistant hash function modeled as a random oracle. Let G be a collision-resistant hash function, $\mathbb{K}$ be a user's secret key, and L be the tree height. We set $t.key$ to $G(w||i)$ and $t.value$ to $\{id_i\}$, where the symbol $||$ denotes an string concatenation. Since $t.value$ has u identifiers, let $u = 1$ here for simplicity. The field $t.leaf$ is set to $(2^{L-1}-1) + (F_{\mathbb{K}}(w||i||c)\%2^{L-1})$, where $c = \boldsymbol{H_T}[w].count$, which is a search counter stored in the local hash table $\boldsymbol{H_T}$. Repeatedly do these, until all the keyword-file-identifier pairs of DB have been processed. Second, we evict all the triplets from the stash to the ORAM tree by using the following multi-path eviction algorithms.

Let $token(w,i,c) = (2^{L-1}-1) + (F_{\mathbb{K}}(w||i||c)\%2^{L-1})$. $token(w,i,c)$ is the leaf identifier to retrieve the keyword-file-identifier pair (w, id_i). Since the user has $token(w,i,c)$ all the time, the user can always fetch the desired triplets. Note that the user cannot directly access the tree node containing the pair. Otherwise, access patterns will be leaked.

We show an OBI example with two embedded keyword-file-identifier pairs: $\{(w,50), (w,52)\}$ in Figure 1. The pairs are converted into two triplets, $\{(G(w||1),50,8), (G(w||2),52,11)\}$, where 8 and 11 are leaf identifiers generated by computing $\{token(w,1,\boldsymbol{H_T}[w].count) = 8, token(w,2,\boldsymbol{H_T}[w].count) = 11\}$, respectively. Here, $\boldsymbol{H_T}[w].count = 0$. The local keyword hash table $\boldsymbol{H_T}$ stores $\{(w,(2,0))\}$ that denotes there are two keyword-file-identifier pairs of w. The user can download the full 8-to-root path and the 11-to-root path to obtain $\{50, 52\}$.

B. Muti-Path Eviction Algorithms

A challenge in OBI is how to insert a large number of entries to multi-paths of the ORAM tree.

Large-stash eviction problem. We are given a large set of randomly distributed triplets stored in the stash. How to insert all the stash triplets into multi-paths of the ORAM tree obliviously and efficiently?

The large-stash eviction problem involves three aspects. First, the eviction should be oblivious. That is, the chosen multi-paths must be indistinguishable from random. Second, the eviction should be round-efficient and computationally efficient. Third, after the eviction, a well-designed multi-path eviction algorithm should satisfy the condition that the probability of remaining triples in the stash is extremely small.

Three events maybe lead to a temporarily large stash. First, the user wants to search a keyword with a large set of results, and then the stash is large since all the accessed blocks should be re-encrypted. Second, the user wants to insert a large number of triplets into the index. Third, the user wants to delete a large number of triplets.

A straightforward idea. We can slightly extend the single-path eviction algorithm [5] of PATH ORAM to support a multi-path eviction to address the large-stash eviction problem. Assume r eviction paths are given, and the stash size is $|\boldsymbol{S_T}|$, including the last retrieved data blocks. We want to evict these data blocks into the multiple paths. We add an additional loop, compared with the original single-path eviction algorithm, for running r times on the algorithm. From bottom to top, we fill all the paths with the stash blocks. The algorithm complexity is $O(Z \cdot L \cdot r \cdot |\boldsymbol{S_T}| \cdot t_{path})$, where t_{path} is the time to test a valid candidate. If $|\boldsymbol{S_T}| = O(r)$ on the large-stash occasions, the time complexity can be considered as $O(r^2)$, which is a heavy computational overhead of the user. The main drawback of this algorithm is that it must scan the entire stash to choose a candidate in each path level. To improve this, we propose two multi-path eviction algorithms.

We present a k-nearest neighbor eviction algorithm (KNNEA) and a partition-based eviction algorithm (PBEA). KNNEA is for processing a small result set, and PBEA is designed for retrieving or inserting a large collection of results or a large set of files. We list the parameters of the two eviction algorithms in Table II.

k-nearest-neighbor eviction algorithm. KNNEA is a multi-path eviction algorithm to evict the user-side triplets to cloud-side multiple paths, as shown in Algorithm 1. Intuitively, since the bottom level of the paths has more room than top levels, the algorithm should put stash entries to the bottom of the paths first. Let $\boldsymbol{T_N}$ be a set of tree nodes stored in a temporary hash table, whose key is a path string and whose value is a tree node. $\boldsymbol{T_N}$ is initially empty. Given a set of accessed leaves, denoted by $\boldsymbol{I} = \{x_1, x_2, \cdots, x_r\}$ relating to a set of leaf-to-root paths, $KNNEA$ evicts stash entries to the paths with a

TABLE II. PARAMETERS AND NOTATIONS

Parameter	Meaning
N	the number of blocks in the tree
L	the height of the tree (root, L=1)
Z	the maximum number of real blocks per tree node
k	$k = 2Z + 1$, # of blocks scanned per node insertion
$leaf$	a leaf identifier
$\boldsymbol{I}$	$\{x_1, x_2, \cdots, x_r\}$, a set of leaf identifiers
r	$r = \lvert\boldsymbol{I}\rvert$, the number of reading paths
$\boldsymbol{T_N}$	a set of tree nodes stored in a hash table
$\boldsymbol{T_A}$	a triplet array sorted by the leaf property of a triplet
$\mathcal{P}(x, l)$	a path of the node at level l along the x-to-root path
t	$(key, value, leaf)$, a triplet
p	a path string
$\boldsymbol{C_T}$	a set of candidate triplets

Algorithm 1: k-nearest-neighbor eviction alg. (KNNEA)

```
1  $\underline{KNNEA}(\boldsymbol{T_N}, \boldsymbol{I})$:
2  $\boldsymbol{T_A} \leftarrow \boldsymbol{S_T}$;
3  for $j = L$ to $1$ do
4      for $leaf \in \boldsymbol{I}$ do
5          $\boldsymbol{C_T} \leftarrow findKNN(\boldsymbol{T_A}, leaf, k)$;
6          for $t \in \boldsymbol{C_T}$ do
7              $p \leftarrow \mathcal{P}(leaf, j)$;
8              if $p = \mathcal{P}(t.leaf, j) \wedge \boldsymbol{T_N}[p]$ is not full then
9                  $\boldsymbol{T_N}[p] \leftarrow \boldsymbol{T_N}[p] \cup \{t\}$;
10                 remove $t$ from $\boldsymbol{S_T}$;
11                 remove $t$ from $\boldsymbol{T_A}$;
12 return $\boldsymbol{T_N}$;
```

bottom-to-top eviction strategy, which fills the (L, $L-1$, $\cdots$, 1) levels of the paths gradually. After invoking Algorithm 1, $\boldsymbol{T_N}$ is a set of newly evicted tree nodes corresponding to the set of paths.

We note that KNNEA is a local eviction algorithm handled by the trusted user. This algorithm evicts the triplets of the stash to a locally stored empty multi-path tree cache. KNNEA has no privacy leakage. After eviction, the user encrypts the multi-path cache and uploads to the cloud for data replacing.

KNNEA relies on $\boldsymbol{T_A}$, which is a triplet array sorted by the leaf property of a triplet, to hold the stash triplets initially. To avoid the same limitation of the PATH-ORAM eviction algorithm, we scan only k-nearest-neighbor triplets to the current path instead of scanning the whole stash. More concretely, for each node $\mathbb{N}$ in each path, if the leaf identifier associated with $\mathbb{N}$ is $leaf$, we read $(Z+1)$ array elements whose leaf identifiers are less or equal than $leaf$, and read $(Z+1)$ array elements whose leaf identifiers are larger or equal than $leaf$. Let $k = (2Z+1)$. This procedure is called $findKNN(\boldsymbol{T_A}, leaf, k)$, whose input is $\boldsymbol{T_A}$, $leaf$, and k. $findKNN$ returns only k nearest candidate triplets. Since each node can hold Z triplets at most, the $k = (2Z+1)$ nearest triplets are enough to fill node $\mathbb{N}$. If the candidate triplets can reside in the node, the algorithm removes the triplet from the sorted array and the stash. In PATH ORAM, the candidate triplets $\boldsymbol{C_T}$ are the whole stash. In comparison, the size of $\boldsymbol{C_T}$ in OBI is only k.

The main intuition behind KNNEA is that it only scans the k nearest neighbors of the current leaf in each level. Scanning the whole stash in each path level is not necessary when the stash is a sorted array. However, the user should always maintain the sorted array when the user needs to remove a candidate from the array, which takes $O(|\boldsymbol{S_T}|)$ time per deletion. Since $findKNN$ consumes only $O(k + \log|\boldsymbol{S_T}|)$ time, it is not the bottleneck. Assume the total number of evicted triplets is v, which equals the number of removals. Since each triplet can be removed only once, $KNNEA$ eviction time complexity is $O(v \cdot |\boldsymbol{S_T}| + Z \cdot L \cdot r \cdot t_{enc})$, where $Z \cdot L \cdot r \cdot t_{enc}$ is the time used to encrypt the final results, $v \cdot |\boldsymbol{S_T}|$ is the eviction cost, and t_{enc} is the time for encrypting a data block. Note that v is less than $Z \cdot r \cdot L$. We define $\beta \overset{def}{=} \frac{v}{Z \cdot L \cdot r}$. In general, $\beta \approx \frac{1}{logN}$. Compared to PATH-ORAM eviction time complexity, KNNEA avoids two multiplicative factors, $\frac{1}{\beta}$ and t_{path}, where t_{path} approximates $O(\log N)$. The small-stash KNNEA eviction time complexity is $O(Z \cdot L \cdot r \cdot t_{enc}) \approx O(r \log N)$. We omit the factor Z and t_{enc} for simplicity.

If the result size r and $|\boldsymbol{S_T}|$ are large (e.g., $r > 1,000$, or $10,000$), the first part $v \cdot |\boldsymbol{S_T}|$ may be the main factor that is a heavy computational overhead. To further address the large-stash eviction problem, we use the following PBEA.

Partition-based eviction algorithm. PBEA is a multi-path eviction algorithm to evict a large set of user-side triplets to cloud-side paths. The core idea of PBEA is dividing the leaf-identifier range into a set of fixed-size partitions, whose size is a power of two (e.g., 65536). After this, we insert the stash triplets into the corresponding partitions according to leaf identifiers. We can then process each partition with a small-result KNNEA. PBEA involves two stages. In the first stage, PBEA evicts most triplets by handling each non-empty partition individually. Thus, linearly scanning the whole stash in each level is avoided. In the second stage, PBEA invokes KNNEA further to evict the remaining triplets in the stash. Since stash size is small after all the partition evictions, processing the remaining result set with KNNEA is efficient.

We list the parameters of PBEA in Table III. The function $f(.)$ maps a leaf identifier to a partition number. As shown in Algorithms 2, $PBEA$ takes as input a set of leaf identifiers, denoted by $\boldsymbol{I}$, and outputs a set of encrypted tree nodes. To save temporary tree nodes, we adopt a globally-stored hash table $\boldsymbol{T_N}$ designed for mapping each string path to a tree node. $\boldsymbol{T_N}$ is initially empty. For each partition, $PBEA$ invokes $KNNEA^*$ to evict triplets to the temporary paths $\boldsymbol{T_N}$. The difference between $KNNEA$ and $KNNEA^*$ is that $KNNEA$ tries to evict the whole stash triplets to the whole paths. In comparison, $KNNEA^*$ tries to evict the partitioned triplets to the partitioned paths. Note that $PBEA$ relies on $KNNEA$ further to evict the triplets in the last stage since there are perhaps still some triplets in the stash after the partition evictions.

With the above processes, the triplets in every partition have been put into $\boldsymbol{T_N}$. If a tree node is not full, we pad it with dummy values to size Z. The user encrypts all the tree nodes in $\boldsymbol{T_N}$ with an RCPA-secure private-key encryption algorithm [36], [57], such as the counter-mode AES. A private-key encryption scheme is said to be Random-ciphertext-secure against the Chosen-Plaintext Attack (RCPA) if the ciphertexts that it outputs are computationally indistinguishable from random even to the adversary that can adaptively query the encryption oracle.

For simplicity, assume the stash contains only the retrieved blocks, and most remaining blocks have been evicted to the tree. The next section shows that permanent stash size (excluding the temporarily retrieved blocks) is near zero. $PBEA$ eviction time complexity is $O(Z \cdot L \cdot r \cdot t_{enc})$, which we write as $O(r \log N)$ for simplicity. PBEA reduces the eviction time complexity of PATH ORAM from $O(r^2)$ to $O(r \log N)$ when $|\boldsymbol{S_T}| = O(r)$.

Correctness analysis. Even if a triplet t has been shuffled and evicted, the user still knows how to access t. The reason is 1) the user can always compute $t.leaf$; and 2) when the triplet has been moved to the j-th level, the location of the triplet at the j-th level is $\mathcal{P}(t.leaf, j)$, which remains in the $(t.leaf)$-to-root path, according to Equation (1).

TABLE III. PARAMETERS OF PBEA

Parameter	Meaning
$\boldsymbol{P}$	an array of partitions stored in a hash table
2^d	the size of each partition
$\boldsymbol{P}[j].ts$	the j-th triplet partition, which is a set of triplets
$\boldsymbol{P}[j].ls$	the j-th leaf partition, which is a set of leaves
$f(.)$	$f(x) = \lceil \frac{x-2^{L-1}+1}{2^d} \rceil$

Algorithm 2: Partition-based eviction algorithm (PBEA)

```
1  $\underline{PBEA}(\boldsymbol{T_N}, \boldsymbol{I})$:
2  $\boldsymbol{P} \leftarrow \{\}$;
3  for $t \in \boldsymbol{S_T}$ do
4      $j \leftarrow f(t.leaf)$;
5      $\boldsymbol{P}[j].ts \leftarrow \boldsymbol{P}[j].ts \cup \{t\}$;
6  for $leaf \in \boldsymbol{I}$ do
7      $j \leftarrow f(leaf)$;
8      $\boldsymbol{P}[j].ls \leftarrow \boldsymbol{P}[j].ls \cup \{leaf\}$;
9  for $j = 0$ to $|\boldsymbol{P}| - 1$ do
10     if $\boldsymbol{P}[j].ts \neq \perp \wedge \boldsymbol{P}[j].ls \neq \perp$ then
11         $\boldsymbol{T_N} \leftarrow KNNEA^*(\boldsymbol{T_N}, \boldsymbol{P}[j].ts, \boldsymbol{P}[j].ls)$;
12 invoke $\boldsymbol{T_N} \leftarrow KNNEA(\boldsymbol{T_N}, \boldsymbol{I})$;
13 return $\boldsymbol{T_N}$;
```

C. Interaction Protocols

The OBI protocols include adding a set of documents, searching for a keyword, and immediately/lazily deleting a keyword or a set of keywords. All the protocols satisfy the following encryption principle.

Encryption principle. For each data block relating to a keyword-file-identifier pair (w, id_i) in the tree or in the stash, we ensure that the value $(w||i||c)$ is unique all the time, where $c = \boldsymbol{H_T}[w].count$, and $i \in [1, \boldsymbol{H_T}[w].len]$.

When a block corresponding keyword w is searched, we increase c by one. When adding a block that corresponds to keyword w, we use $token(w, \boldsymbol{H_T}[w].len, c)$ as the new leaf identifier for this block, and increase $\boldsymbol{H_T}[w].len$ by one, but c remains unchanged. Therefore, $(w||i||c)$ can be unique in the index all the time.

Bulk inserting a set of documents. Given a set of documents for bulk insertion, the user first writes keyword-file-identifier pairs of the documents into the stash. Next, the user performs a reading-and-replacing action to evict blocks containing the pairs into r paths of the tree. To provide more insertion space, the user sets r to a value that is no less than the number of blocks in the stash empirically. Since the user uses PBEA, bulk-insertion complexity is $O(r \log N)$.

Searching for keywords. Keyword searching is also a reading-and-replacing action. The user generates a set of leaf identifiers to search for keyword w. Recall that the leaf identifier for the i-th keyword-file-identifier pair of w is $token(w, i, c)$. There are $\boldsymbol{H_T}[w].len$ tokens in total. The user sends $\boldsymbol{I}$ to the cloud, where $\boldsymbol{I} = \{token(w, 1, c), \cdots, token(w, \boldsymbol{H_T}[w].len, c)\}$. Algorithm 3 shows the search protocol, where $ReadPaths$ is to read a set of tree paths, and $ReplacePaths$ is to replace the original tree nodes with the new encrypted ones.

Recall that if $u = 1$, a keyword-file-identifier pair corresponds to a keyword-block pair because each data block contains only one identifier. If $u > 1$, $\boldsymbol{H_T}[w].len$ denotes the number of blocks relating to w. A candidate block, denoted by $block^*$, is a block containing at most u file identifiers relating to w. Since all the identifiers are in plain text, the user can easily extract the final results. Note that there are perhaps some dummy identifiers in a candidate block, and the user needs to filter them out.

Reading all the leaf-to-root paths of $\boldsymbol{I}$ can retrieve the corresponding result-set data blocks, assuming all keyword-identifier pairs have been initially put into the tree. To avoid the same limitation of most search-pattern-leaked schemes, we increase $\boldsymbol{H_T}[w].count$ by one after the search on w. This increment implies that the accessed data blocks are obliviously mapped to new random paths. Since all the data blocks relating to w are in the local stash now, the user can choose desired values for final results. The user uses KNNEA for processing a small result set or PBEA for processing a large result set.

Search time complexity of OBI is $O(r_w \cdot L)$, where r_w is the number of $(w, block)$ pairs for w, $(r_w = \boldsymbol{H_T}[w].len)$. For simplicity, let $r_w = r$. The cloud takes $O(r \cdot L)$ access time to read the set of leaf-to-root paths, and the user requires shuffling the blocks and encrypting them. The total communication bandwidth is $O(B \cdot Z \cdot r \cdot L)$ bits per query, where B is the block size in bits. Thus search time is $O(r \cdot L) \approx O(r \log N)$. Since $L \approx \log N << n$, where n is the number of data files, the search complexity is quasi-optimal in the worst-case.

Algorithm 3: A protocol for keyword searching

```
1  $\underline{Search}(\boldsymbol{I})$:
2  Cloud: $\boldsymbol{T_N} \leftarrow ReadPaths(\boldsymbol{O_T}, \boldsymbol{I})$;
3  User:
4      write $\boldsymbol{T_N}$ into $\boldsymbol{S_T}$;
5      $\boldsymbol{H_T}[w].count \leftarrow \boldsymbol{H_T}[w].count + 1$;
6      $\boldsymbol{R} \leftarrow \{\}$;
7      for $i = 1$ to $\boldsymbol{H_T}[w].len$ do
8          $key \leftarrow G(w||i)$;
9          $leaf^* \leftarrow token(w, i, \boldsymbol{H_T}[w].count)$;
10         $\boldsymbol{S_T}[key].leaf \leftarrow leaf^*$;
11         $block^* \leftarrow \boldsymbol{S_T}[key].value$;
12         $\boldsymbol{R} \leftarrow \boldsymbol{R} \cup \{block^*\}$;
13     output all the file identifiers from $\boldsymbol{R}$;
14     $\boldsymbol{T'_N} \leftarrow KNNEA(\perp, \boldsymbol{I})$;
15     encrypt $\boldsymbol{T'_N}$;
16 Cloud: $\boldsymbol{O_T} \leftarrow ReplacePaths(\boldsymbol{O_T}, \boldsymbol{T'_N})$;
```

A search operation can be finished in a single-round-trip interaction. The last round of the search operation is to replace the original paths. This operation can be folded into the next

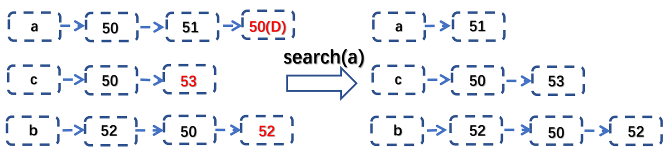

Fig. 2. Delete (a,50); Insert(c, 53); Insert(b, 52); Search(a).

query. The user only needs to buffer the last retrieved tree nodes temporarily.

There are two types of deletion algorithms, immediate deletion, and lazy deletion. The first algorithm deletes a pair immediately from the index. The second algorithm postpones the actual deletion to the subsequent search for w. All deletions rely on result-set reorganization techniques.

Immediate deletion. This algorithm is almost the same as the search algorithm. Recall that the result set $DB(w)$ is entirely rebuilt after every search on w. The user can remove the undesired (w, id) pairs when re-encrypting $DB(w)$. To immediately delete a (w, id) pair, the user only needs to perform a search and remove the identifiers when they exist in the stash. The immediate-deletion complexity is $O(r \log N)$, which is the search complexity.

Lazy deletion. A lazy deletion can be considered as an addition operation with a deletion bit. The real deletion is postponed to the time the search for this keyword is performed.

Given a file identifier, we use its highest bit as an indicator. If it is 0, the identifier should be added; Otherwise, be deleted. For example, in Figure 2, to delete $(a, 50)$, the user inserts a pair $(a, 50(D))$, where $50(D)$ means the identifier should be deleted later. When searching for a, the user rebuilds $DB(a)$ by using $(a, \{50, 51, 50(D)\})$. The lazy deletion time is $O(\log N)$, which is the block addition time.

Result-set reorganization. Result-set reorganization is an approach to reorganizing a result set at the user-side when the user searches for w. The purpose of the reorganization is for future efficient search queries. The algorithm includes two parts, removing duplicated (w, id) pairs, and handling lazy deletions. After a search, all redundant information relating to w has been removed.

D. ZPH-OBI: An Operation-Kind Hiding DSE Scheme

OBI leaks result size per query. However, we can, if possible, pad every query, including searching, inserting, and deletion to the maximum size $\frac{M}{u}$ (blocks), where $M = max_{w \in W}|DB(w)|$, and W is all the keywords that can be queried. We name the new DSE scheme ZPH-OBI. With the benefit of KNNEA/PBEA, ZPH-OBI takes $O(M \log N)$ time per query. ZPH-OBI hides not only the size pattern but also operation kinds, the information about $op \in \{search, add, del\}$. Liu et al. showed that the size pattern combined with the operation-kind pattern can lead to search-update correlations in [13]. Thus, hiding the operation kinds is meaningful.

ZPH-OBI is designed for only search-intensive DSE. That is, an update query with many search queries. If the user want to frequently update big data blocks, the result size pattern and operation kind pattern might be reasonably leaked.

V. OBI and Applications

In this section, we first analyze stash size. Next, we present an index-building algorithm. At last, we give OBI-based applications, such as oblivious file systems and oblivious conjunctive queries.

A. Stash-Usage Estimation

Load factor. Let $\mathcal{N}$ be the number of real blocks inserted into the tree. Let $\beta = \frac{\mathcal{N}}{Z(2^L-1)}$, called the load factor. Intuitively, we should reserve some space for insertions. Otherwise, most blocks will overflow to the stash.

Let $Y_i = (y_1, y_2, \cdots, y_i)$ be a sequence of accesses to OBI. Let $OBI(Y_i)$ be the running state of the i-th access. Let $St(OBI(Y_i))$ denote the local stash size after the i-th query, and $St(OBI(Y_{i+1}))$ denote the stash size after the $(i+1)$-th query. We give the following theorems.

Assumptions. Let U_{i+1} denote a set of stash triplets, including the old triplets and the newly downloaded triplets, before the $(i+1)$-th eviction. Let the $(i+1)$-th operation access the set of leaves, denoted by $\boldsymbol{I}_{i+1}$, where $r = |\boldsymbol{I}_{i+1}|$, assuming r is large. We make three assumptions. Assumption (1): $\boldsymbol{I}_{i+1}$ is in uniform distribution. Assumption (2): the number of retrieved real blocks is less than $Z \cdot L \cdot r \cdot \beta$ in the $(i+1)$-th operation. Assumption (3): $\beta < \frac{\gamma}{L}$, where $\gamma = 0.8 < 4 \ln 1.25$.

Theorem 5.2 (Oblivious bulk insertion). Let $X_{i+1} = St(OBI(Y_{i+1}))$. After an access sequence Y_i, we insert a large set of triplets into the local stash. Let the number of stash triplets be ξ. We will perform a reading-and-replacing action y_{i+1}. For any integers $\xi \geq 0$ and $R \geq 0$, if Assumptions (1-3) hold, and if $r > \frac{\xi}{Z \cdot (\gamma - L \cdot \beta)}$, we have

$$Pr[X_{i+1} > R \mid X_i = \xi] \leq e^{-\alpha_1 r - \alpha_2 R}, \qquad (2)$$

where $\alpha_1 = 0.0231Z$ and $\alpha_2 = 0.2231$.

Proof: We use $E(.)$ to denote a mean. Since $\boldsymbol{I}_{i+1}$ is in uniform distribution, the tree is divided into r same-size subtrees. According to Assumption (1), each subtree contains only one eviction path. This path is used to hold the stash triplets. According to KNNEA, in the worst case, the triplets can be evicted to the root node of the subtree. Let $E(U_{i+1}^T)$ be the number of the triplets that belong to a subtree T in the average case. Since $St(OBI(Y_i)) = \xi$, thus $E(U_{i+1}) \leq \xi + Z \cdot L \cdot r \cdot \beta$. Let $\eta = \xi + Z \cdot L \cdot r \cdot \beta$. We have $\eta \leq Zr\gamma$, due to $r > \frac{\xi}{Z \cdot (\gamma - L \cdot \beta)}$. Since the triplets U_{i+1} are randomly distributed, then $E(U_{i+1}^T) = \frac{E(U_{i+1})}{r} \leq \frac{\eta}{r} \leq Z\gamma$. Since the root of the subtree has Z triplets, this node is large enough to hold the triplets, whose size is $E(U_{i+1}^T)$. Intuitively, the sub-roots are large enough to hold U_{i+1}. We want to make a more accurate analysis. We assume each root of each subtree has infinite size for only stash analysis. If a sub-root contains more then Z triplets, we want to count the number of overflowing triplets. Since we can move the overflowing triplets into the stash, this assumption does not affect the final conclusion.

Define a random variable $G_j \in \{0, 1\}$, which represents the j-th triplet state in all the subtree roots. $G_j = 1$ means the triplet is a real block after the eviction, otherwise $G_j = 0$. Let $p_j = Pr[G_j = 1]$. Observe that each G_j is independent. Given a positive constant value $t > 0$, we first infer

$E(e^{t\underset{T}{\Sigma}U_{i+1}^T}) = E(e^{t\underset{j}{\Sigma}G_j}) = E(\prod_j e^{tG_j}) = \prod_j (p_j(e^t - 1) + 1) \leq \prod_j (e^{p_j(e^t-1)}) = e^{(e^t-1)\underset{j}{\Sigma}p_j} = e^{(e^t-1)E(\underset{T}{\Sigma}U_{i+1}^T)}$.

Next, according to the Markov inequality, we infer

$$\begin{aligned} Pr[X_{i+1} > R] &\leq Pr[\underset{T}{\Sigma}U_{i+1}^T > r \cdot Z + R] \\ &= Pr[e^{t\underset{T}{\Sigma}U_{i+1}^T} > e^{t(r\cdot Z+R)}] \\ &\leq E(e^{t\underset{T}{\Sigma}U_{i+1}^T}) \cdot e^{-t(r\cdot Z+R)} \\ &\leq e^{(e^t-1)E(\underset{T}{\Sigma}U_{i+1}^T)} \cdot e^{-t(r\cdot Z+R)} \\ &\leq e^{(e^t-1)rZ\gamma} \cdot e^{-t(r\cdot Z+R)} \\ &= e^{-tR+r((e^t-1)Z\gamma - tZ)}. \end{aligned}$$

We choose a value $t > 0$ such that $(e^t - 1)Z\gamma - tZ < 0$. Let $t = \ln 1.25 = 0.2231$, then

$$Pr[X_{i+1} > R] \leq e^{-0.0231\cdot r\cdot Z - 0.2231R}. \tag{3}$$

Therefore, Inequality (2) holds.

We now prove that all the assumptions are reasonable. Since each leaf identifier of $\boldsymbol{I}_{i+1}$ is generated by a pseudorandom function, whose input key is unique in the index all the time, Assumption (1) is reasonable. Since each path has the same access probability, if r is large, Assumption (2) can be satisfied. As for Assumption (3), we can increase the height of the tree L to provide more insertion space such that Assumption (3) holds. Note that OBI using PBEA still satisfies Inequality (2) since the proof is similar. If considering only data searching, we have the following theorem.

Theorem 5.3 (Oblivious multi-path search). Assume there is a sequence of s searches $Y_s = (y_1, \cdots, y_s)$. Let $X = St(OBI(Y_s))$. Let the stash size before the i-th reading-and-replacing action be ξ_{i-1}. Let the number of accessed leaves of the i-th reading-and-replacing action be r_i. Assume $r_i > \frac{\xi_{i-1}}{Z\cdot(\gamma - L\cdot\beta)}, (i \in [1, s], r = r_s)$. If Assumptions (1-3) hold, and if $R \geq 0$, then

$$Pr[X > R] \leq e^{-\alpha_1 r - \alpha_2 R}, \tag{4}$$

where $\alpha_1 = 0.0231Z$ and $\alpha_2 = 0.2231$.

Proof. Theorem 5.3 is similar to Theorem 5.2.

Theorems 5.2 and 5.3 prove that the multi-path eviction algorithm outperforms the single-path eviction algorithm in terms of stash size. For example, if we set $Z = 4$ and $r = 1$, then $Pr[X = 0] \geq 0.9117$; if $r = 100$, then $Pr[X = 0] \geq 1 - 9.7 \cdot 10^{-5}$; and if $r = 1000$, we have $Pr[X = 0] \geq 1 - 7.4 \cdot 10^{-41}$. This theorem implies that the stash size of multi-path reading r data blocks in one time is smaller than that of single-path reading r data blocks in r times. This is because the data blocks in the stash have more opportunities of going to the multi-paths.

B. Index Setup

Offline setup. We propose an Offline Index Setup Algorithm (OISA). Given a set of data files to create an oblivious index, the user directly writes the keyword-file-identifier pairs into the tree leaves without data shuffling. Since the whole index encryption happens at the trusted user-side, no information is revealed. After index initialization, the user outsources the whole index to the cloud.

OISA consists of five steps. The user first creates a local plain-text inverted index DB of the data files. For any keyword w, $DB(w)$ is divided into fixed-size blocks with each data block containing u file identifiers. The user puts all $(w, block)$ pairs into the stash. Next, all the triplets in the stash are inserted into the leaves according to leaf identifiers. If the leaves are full, the triplets remain in the stash. After OBI initialization, the user uploads the tree to the cloud.

Since F is a pseudorandom function, the leaves $\{token(w, i, c)\}_{(w,i)\in DB}$ are in uniform distribution, where c is initially 0. Therefore, every tree leaf contains the same number of triplets with high probability. Thus, the stash can be nearly empty after index initialization. The core idea behind OISA is that all the stash triplets are evicted to the leaves without data shuffling. Therefore, OISA achieves the optimal time complexity of $O(N)$.

C. Choosing a Partition Size

Recall that in PBEA, the size of each partition is 2^d. Our goal is to choose a proper value d such that the execution time of PBEA is minimal. Since PBEA invokes KNNEA to process all the partitions in the beginning stage and the tree top part in the final stage, where the tree top height is $(L-d)$, a large or a small partition is not suitable for PBEA. Thus, we empirically set $d = \lceil L/2 \rceil$.

D. OBI and Applications

An oblivious file system. We give an OBI-based non-interactive Bulk-Insertion Oblivious File System prototype named BI-OFS, which differs from prior interactive oblivious file systems [58], [59]. BI-OFS is an oblivious data structure that aims to hide the operation names, file identifiers, and file contents. BI-OFS logically contains a set of directories and files. Assume each file has the same size. Each directory contains a set of files and a set of subdirectories. BI-OFS provides privacy-preserving directory-read and bulk-file-write services to the user.

We consider a keyword-file-identifier pair as a file-content pair (f, c), where f consists of an absolute directory path and a file name (e.g., /etc/passwd), and c is a fixed-size data file. The user maintains a local directory that contains all the directory paths of the file system. The cloud has all the encrypted files. File meta data, such as file names and attributes, are put into file headers. Since the directory names are not large enough in general, the user-side can always hold the local directory. To read a file, the user should download the current directory, excluding the subdirectories. To read a full directory, the user first reads the subdirectory names from the local directory, and then the user performs a reading-and-replacing action to access r tree paths, where r is a constant. The user hides the directory size pattern by adding fake queries to make every reading-and-replacing action access the same number of tree paths.

BI-OFS enjoys the following merits. First, it supports bulk inserting a set of files. Second, it can read an entire directory containing subdirectories efficiently. Third, the read is non-interactive. The user can read/add/remove a directory in a single-round-trip interaction. Similar to ZPH-OBI, BI-OFS is suitable for only search-intensive file systems.

An oblivious conjunctive-query scheme. OBI supports only single-keyword query. To achieve efficient conjunctive queries, such as "select * from users where name='Tom' and sex='male' ", we design a new scheme.

Algorithm 4 gives an x-term Oblivious Conjunctive-Query searchable encryption demo, named OCQ. Let OBI be an OBI scheme, and $OMAP$ be the oblivious map [18]. OCQ consists of three subroutines, $Setup$, which is for locally setting up an index, $Search$, which is for searching $w_1 \wedge w_2 \cdots \wedge w_x$, and $Update$, which is for adding or deleting a keyword-file-identifier pair (w, id). We list the parameters of OCQ in Tables IV.

TABLE IV. PARAMETERS OF OCQ

Parameter	Meaning
$\mathcal{F}$	a set of plain-text data files
$OMAP$	an oblivious MAP
$results$	a set of file identifiers, the final results
$temp$	a set of file identifiers, the temporary results
$w_1, w_2, \cdots, w_x$	keywords

Algorithm 4: An oblivious conjunctive-query DSE scheme (OCQ)

```
Setup($\mathcal{F}$):
  $DB = CreateInvertedIndex(\mathcal{F})$.
  $OBI.Setup(\mathcal{F})$
  $OMAP = \{\}$
  for all $(w, id)$ in $DB$ do
      $OMAP.write(w||id, 1)$
  return $(OBI, OMAP)$

Search($w_1 \wedge w_2 \wedge \cdots \wedge w_x$):
  choose the short term, assuming it is $w_1$.
  $results \leftarrow \{\}$
  $temp \leftarrow \{\}$
  $temp \leftarrow OBI.Search(w_1)$
  for all $id \in temp$ do
      if $OMAP.read(w_2||id) = 1$
      $\wedge \cdots$
      $\wedge$ $OMAP.read(w_x||id) = 1$ then
          $results \leftarrow results \cup \{id\}$;
  return $results$

Update($op$, $w$, $id$):
  if $op =$ '$add$' then
      $OMAP.write(w||id, 1)$
      $OBI.add(w, id)$
  if $op =$ '$del$' then
      $OMAP.write(w||id, 0)$
      $OBI.delete(w, id)$
```

In the setup protocol, the user first builds a plain-text inverted index DB from a set of files $\mathcal{F}$, where $CreateInvertedIndex$ is the plain-text data processing algorithm. This protocol employs two encryption schemes, OBI and OMAP. OBI is used here for storing the final keyword-file-identifier pairs. OMAP is designed for oblivious conjunctive queries. For any keyword-file-identifier pair (w, id) in DB, OMAP writes the pair into the map by invoking $OMAP.write(w||id, 1)$. The main difference between OBI and OMAP is that OBI is for key-set mapping. In comparison, OMAP is for key-value mapping.

In the search protocol, to perform a conjunctive search query, $w_1 \wedge w_2 \wedge \cdots \wedge w_x$, the user first choose the short term, which is the keyword that matches the smallest results among $\{w_1, \cdots, w_x\}$. Since OBI stores all the keyword information at the user-side, the user can easily choose the short term. Then, the user searches the short term and gets an intermediate result set, which is a set of file identifiers. With OMAP, the users can filter out incorrect file identifiers that do not match the conjunctive query by testing each $OMAP.read(w^*||id^*)$ is 1 or not, where w^* is from $\{w_2, \cdots, w_x\}$, and id^* is from the intermediate result set.

In the update protocol, to perform an update query, the user can efficiently invoke $OMAP.write$, $OMAP.read$, $OBI.add$, and $OBI.delete$ APIs to update the oblivious index. $OBI.add(w, id)$ denotes an addition of the keyword-file-identifier pair (w, id), and so on.

We give a comparison with the typical conjunctive-query DSE scheme VBtree [23] in Table V. VBtree is a forward secure DSE scheme that supports efficient conjunctive queries with leaking the search and access patterns. The advantage of VBtree is that it can process a conjunctive query in the cloud without any user-cloud interactions (except for returning the final results). The results in the table show that OCQ nearly reaches the search efficiency of the insecure baseline. OCQ has search time of $O(Z \cdot x \cdot r_1 \cdot \log^2 N)$, where Z is the number of blocks in a tree node.

VI. SECURITY ANALYSIS

A. Leakage Analysis

Claim 1: The OBI protocol leaks no information to the cloud except the number of accessed leaves.

Proof: There are three stages in the reading-and-replacing protocol, reading, shuffling, and replacing.

In the read stage, the user downloads r paths from the ORAM tree. Thus, the size pattern r is leaked now. There are two types of paths: a fake path and a real path. A fake path is for data padding or providing more insertion space for data shuffling. Since the fake path is randomly generated, its leaf identifier is indistinguishable from random. A real path contains the desired data blocks to be retrieved. Assume x_i is the real-path leaf identifier. Recall that x_i is generated by the keyed pseudorandom function, whose input is $(w||i||c)$. x_i is still indistinguishable from random since $(w||i||c)$ is unique in the index all the time according to the data shuffle algorithm of KNNEA/PBEA, even if the same block has been accessed. Therefore, there is no other information leakage in this stage.

In the data shuffle stage, KNNEA/PBEA works at the trusted user-side. Thus, the cloud learns nothing about shuffling contents of KNNEA/PBEA. The user evicts the accessed blocks to new locally-stored cache paths, and ensures that each $(w||i||c)$ is unique in the index all the time. After data shuffle, the locally-stored paths are encrypted by the RCPA-secure algorithm for preparing to upload to the cloud. Therefore, there is no information leakage in this stage.

In the data replacing stage, the accessed paths are overwritten by the re-encrypted paths. In the reading-and-replacing action, the original paths are replaced with re-encrypted data. Due to the RCPA-secure algorithm, the encrypted path contents are indistinguishable from random. Even if the block resides in the original location, the cloud observes only the random ciphertext and gains no more information in this stage. Therefore, there is no information leakage in this stage.

TABLE V. A COMPARISON WITH THE TYPICAL CONJUNCTIVE-QUERY SCHEME VBTREE

Schemes	Cloud storage size	Local storage size	Search time	Round trips	Security
VBtree [23]	$O(N \log n)$	$O(m)$	$O(x \cdot r_1 \log n)$	1	forward secure
OCQ(this paper)	$O(N)$	$O(m)$	$O(Z \cdot x \cdot r_1 \log^2 N)$	$O(\log N)$	strong forward and backward secure

Assume $w_1 \wedge \cdots \wedge w_x$ is a conjunctive query. N is the total number of keyword-file-identifier pairs, m is the number of keyword-file-identifier pairs in the index, n is the number of files. Z is the number of data blocks in a tree node. x is the number of conjunctive terms. $r_1 = \min\{|DB(w_i)|\}_{i\in[1,x]}$.

From the above analysis, we conclude that the reading-and-replacing protocol leaks only the size pattern r.

B. DSE Security

Theorem 6.1 (Strong-FB security). Assuming the existence of the RCPA-secure algorithm, and F is a pseudorandom function, then OBI is strong forward and backward secure.

Proof. Let $\mathcal{A}$ be a probabilistic polynomial-time (PPT) adversary. $\mathcal{A}$ can adaptively issue search and update queries without accessing the user-side secret information since we assume the user-side data structures, including the keyword hash table and the stash, are trusted in the security model. We now prove that there exists a PPT simulator $\mathcal{S}$, who can adaptively simulate $\mathcal{A}$'s queries, including data searches and data updates.

$\mathcal{S}$ adaptively simulates $\mathcal{A}$'s queries. For a query q at time t, $\mathcal{A}$ has a set of leaf-to-root paths, $PATH = \{x_1\text{-}to\text{-}root, \cdots, x_{r_t}\text{-}to\text{-}root\}$. $\mathcal{S}$ also randomly chooses a set of leaf-to-root paths $PATH^* = \{x_1^*\text{-}to\text{-}root, \cdots, x_{r_t}^*\text{-}to\text{-}root\}$, where $|PATH| = |PATH^*| = r_t$, according to $\mathcal{L}^{Query}$. After the query, $\mathcal{A}$ has a set of newly updated paths. $\mathcal{S}$ also updates the accessed paths with random values. If a path of the query is a real access, since x_j is generated from a unique bit string in every data searching, data inserting, and deletion, each keyword-file-identifier pair corresponds to a unique string. According to the pseudorandom function F, each leaf identifier x_j and x_j^* ($j \in [1, r_t]$) are PPT computationally indistinguishable. If a path of the query is a fake access, then x_j is a randomly-generated value. x_j and x_j^* are also PPT computationally indistinguishable. According to the RCPA-secure encryption algorithm, $PATH$ and $PATH^*$ are PPT computationally indistinguishable.

The above analysis implies that the adversary learns nothing from the OBI protocol except $\mathcal{L}^{Query}$. Thus, OBI achieves strong-FB security. If the result size is padded to the constant value M, ZPH-OBI achieves operation-hiding-FB security.

VII. Performance Analysis

A. Experimental Methodologies

The experiments are performed on a desktop computer that runs Windows 10 with an Intel(R) Core(TM) i9-10850K CPU and 64 GB DDR4 memory. Blake2b is employed as the pseudorandom function, and counter-mode AES acts as the RCPA-secure encryption algorithm. The scheme and testing cases are programmed with C++ 20. N denotes the number of keyword-file-identifier pairs, m is the number of keywords in the index, n is the number of files, L is the height of the tree, Z is the number of blocks in a tree node, β is the load factor, and u is the number of file identifiers in one block. r is the number of reading paths per query. We empirically set $r = \xi$ for the large-stash eviction problem.

Assume the entire oblivious index can be fully loaded into the memory. Each file identifier is a 64-bit integer that includes a deletion bit. We ignore all communication time of the experiments. For every experiment, we create a single thread to evaluate performance. The stash size does not include the number of temporarily retrieved triplets in every access. We do not consider the case that the ORAM tree is full since we always reserve some space for insertions.

Datasets. We use the well-known Enron email dataset [60], which contains 517 thousand text documents in total. We extract keywords by splitting every document with a space character. The four sub-datasets are generated with the following approach. First, we create an unencrypted inverted index for the whole Enron dataset. Second, we generate the sub-datasets by randomly choosing a set of keywords from the inverted index, as shown in Table VI.

TABLE VI. THE ENRON DATASETS

Datasets	N	m	n
DB1	8,228,457	3,000	517,401
DB2	34,607,795	30,000	517,401
DB3	81,762,592	300,000	517,401
DB4	105,436,113	2,686,391	517,401

B. Index-Setup Evaluation

In Table VII, we list the experimental data of index setup, where KM size denotes the size of the keyword hash table. The number of file identifiers in a block is set to $u = 32$. The size of the tree node is set to $Z = 4$. The height of the tree for {DB1, DB2, DB3, DB4} is set to $L = \{23, 26, 27, 27\}$, respectively. DB4 is the whole Enron dataset, whose index size is 7.73 GB. The setup time is exclusive of data preprocessing.

TABLE VII. INDEX SIZES AND CONSTRUCTION TIME

Dataset	Index size	KM size	Setup time	Ind. speed(pairs/s)
DB1	381 MB	141 KB	7 s	$1.2 \cdot 10^6$
DB2	1.57 GB	1.4 MB	38 s	$9.5 \cdot 10^5$
DB3	2.28 GB	13.7 MB	103 s	$7.9 \cdot 10^5$
DB4	7.73 GB	122 MB	215 s	$4.9 \cdot 10^5$

Index size. The experimental data in Table VII demonstrate that OBI has the optimal index size. The whole oblivious-index size is 7.73 GB, which is proportional to the size of the entire unencrypted inverted index of 747 MB.

Setup time. The experimental data in Figure 3 and Table VII demonstrate that the offline setup achieves the optimal complexity. The online setup uses KNNEA for per-keyword batch insertion. That is, the number of insertion rounds is m. The offline setup algorithm is OISA. The offline indexing speed of DB4 reaches $4.9 \cdot 10^5$ pairs/s, which is far more efficient than the online indexing speed of $2.5 \cdot 10^4$ pairs/s when $L = 27$. This is because OISA does not involve data shuffling in every access.

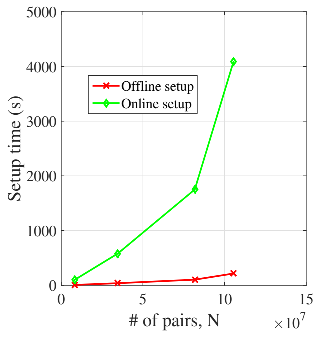

Fig. 3. Online/offline index-setup time on the Enron datasets.

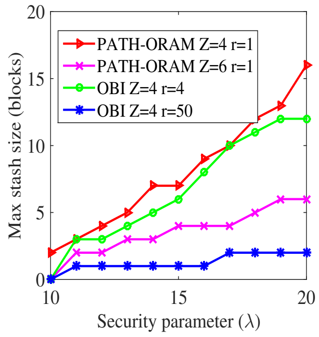

Fig. 4. Security parameters of PATH-ORAM and OBI

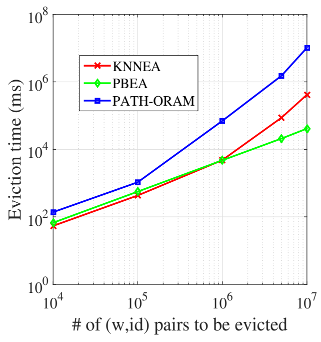

Fig. 5. A comparison of eviction algorithms (logarithmic scale).

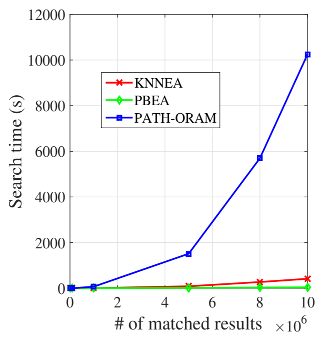

Fig. 6. Search time of the eviction algorithms ($L = 22$, $u = 32$, $Z = 4$).

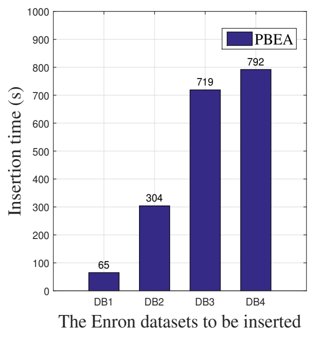

Fig. 7. Bulk insertion of the Enron dataset with PBEA, where $L = 27$, $u = 32$, and $Z = 4$.

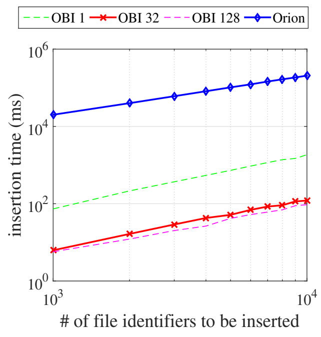

Fig. 8. Comparison with Orion on file-identifier insertion (L=22, Z=4). OBI 32 denotes u=32, and so on.

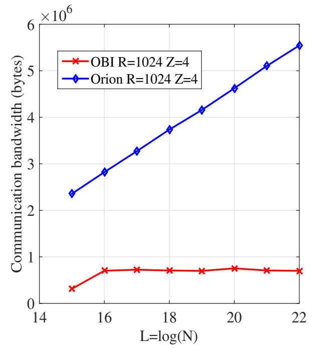

Fig. 9. Comparison with Orion on bandwidth, where the number of matched files R=1024

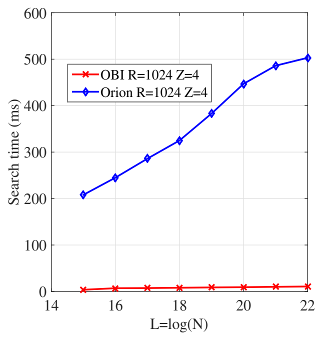

Fig. 10. Comparison with Orion on search time, where the number of matched files R=1024

Client storage. Experimental data in Table VII show that client storage of OBI can be acceptable. The user has two data structures, the keyword hash table and the stash. Since the local stash is nearly empty, the user maintains only the $O(m)$-byte keyword hash table. The user can choose only a set of desired keywords for index building and discards meaningless keywords, such as stop words and strange symbols. Thus the keyword hash table is still scalable, even if a large set of data files are given.

The $O(m)$-byte keyword hash table is a storage-efficiency data structure. The table in our OBI scheme acts as the position map in the non-recursive PATH ORAM, but the size of the table in our OBI scheme is much smaller than the size of the position map in the non-recursive PATH ORAM, because the size of the position map in the non-recursive PATH ORAM is $O(N)$ and $m << N$, where N is the number of keyword-identifier pairs.

Many non-oblivious DSE indexes also have a client-side keyword hash table, such as [12], [55], [61], [62]. They show that $O(m)$-byte local storage is acceptable on most modern devices. To the best of our knowledge, it seems that there are no better solutions to outsourcing the keyword table in a non-recursive DSE scheme. On the occasion of large m (e.g., $m = N$), we can use the recursive techniques [18] to outsource the storage to the cloud.

C. Stash Usage

Stash size. *The experimental results in Figure 4 demonstrate that the multi-path eviction algorithms outperform the single-path eviction algorithm of PATH-ORAM [5] in terms of stash size*. We set $Z = 4$, $L = 15$, and $\beta = \frac{1}{L}$ for both PATH-ORAM and OBI. Since a larger Z implies poor access efficiency, we do not choose $Z = 6$. If $r = 1$, the OBI stash size is similar to PATH-ORAM. The line $r = 4$ means four reading paths per query. Each point of each line is an experiment performed by accessing 2^λ ($\lambda \in [10, 20]$) times (with different locations) to the index. In PATH ORAM, the stash size of reading a block in one time and the stash size of reading four blocks in four times are similar. This is because the stash size of PATH ORAM has a small number of relations with the reading times. Its stash size is mainly determined by the last round reading operation. In OBI, reading four blocks in one time has a smaller stash size compared with PATH ORAM according to the experimental results of Figure 4.

In OBI for a DSE scheme, the number of reading paths per query r is variable. When $r = 1$, the OBI ORAM can be viewed as PATH ORAM with the same stash analysis. The value Z is a constant after index building. Based on Emil Stefanovy et al.'s conclusions, Z should be set to $Z \geq 4$ to avoid exceeding stash capacity. Thus, we let $Z = 4$ in our scheme. Prior single-path ORAM schemes, such as OMAP, also set Z to 4. Note that the larger the value Z is, the more encryption time will be cost in every query.

An ORAM with the security parameter (R, λ) means the probability of the permanent stash size exceeding R is less than $2^{-\lambda}$ ([5]). These results match Inequality 3, which proves the stash overflow probability decreases exponentially in r. We note that the DB1 Enron dataset returns $r = 85$ result blocks per query in the average case if $u = 32$. Thus, the permanent stash occupancy is trivial.

Eviction experiments. To evaluate the actual performance of KNNEA and PBEA, we extend the single-path eviction algorithm of PATH ORAM to support multi-path evictions. Compared with the original PATH ORAM, the extended version has an additional loop for multi-path eviction. Figure 5 shows the comparison of three eviction algorithms, KNNEA, PBEA, and PATH ORAM. Experimental results demonstrate that PBEA is two orders of magnitude faster than PATH ORAM if $r \cdot u > 10^7$. KNNEA is one order of magnitude faster than PATH ORAM when $r \cdot u = 10^6$.

In Table VIII, we list the permanent stash size of three eviction algorithms for bulk insertion and show that the permanent stash size is zero with considerable probability. The experimental data show that KNNEA, PBEA, and PATH ORAM can evict all the stash triplets into the ORAM tree in single round trip. After evicting 10^7 keyword-file-identifier pairs to $r = 312,500$ paths, the stash of PBEA is empty.

TABLE VIII. STASH SIZE AFTER LARGE-BATCH EVICTIONS

# of pairs to be evicted	r	KNNEA/PBEA/PATH ORAM
10^4	313	0/ 0/ 0
10^5	3, 125	0/ 0/ 0
10^6	31, 250	0/ 0/ 0
10^7	312, 500	0/ 0/ 0

Figure 6 demonstrates search efficiency of PBEA and KNNEA is 5X $\sim$ 270X higher than that of PATH ORAM. PATH ORAM consumes more than three hours to search a keyword with 10^7 matched results if $u = 32$. KNNEA consumes 510 seconds. However, PBEA takes only 40 seconds.

D. Insertion Efficiency

Large-batch insertion. We insert the entire Enron dataset into an empty oblivious inverted index. These experiments differ from the online setup algorithm shown in Figure 3, which is highly interactive. Here, we complete the insertion in a single-round-trip interaction. We first write the dataset into the stash. Next, if the stash has r triplets, we perform a reading-and-replacing action to put the dataset into r paths in one batch.

Experimental results in Figure 7 demonstrate that the multi-path ORAM outperforms the single-path ORAM in terms of insertion efficiency. PBEA supports efficient bulk insertion. Inserting the Enron DB1 dataset takes only 65 seconds, and inserting DB4 takes only 792 seconds. However, we consume more than 12 hours to insert the smallest DB1 dataset with the extended PATH-ORAM algorithm and more than one day to insert the DB1 data blocks with the original PATH-ORAM algorithm. This is because PATH ORAM should scan the whole stash in every query to shuffle the stash data blocks. When the stash size is small, data shuffling cost is trivial. In the experiments, a partition of size 65536 is suitable for PBEA to process each partitioned stash efficiently. When the stash size is large, data shuffling time complexity is proportional to the square of the stash size. Thus, we should use PBEA to partition a large ORAM into a set of small ORAMs such that each partitioned ORAM does not suffer from the large-stash eviction problem. In comparison, we take only 303 seconds to insert DB1 with KNNEA, but it is still not suitable for large-batch insertion. Note that the offline setup of DB1 with OISA takes only 7 seconds in Table VII. This is because OISA does not involve time-consuming data shuffling. We omit the remaining experiments on PATH ORAM since they will consume at least several days.

E. Compared with Orion, Eurus, and Other Works

Experimental results in Figures 9 and 10 demonstrate that OBI is far more efficient than Orion proposed in CCS 2018 [11]. We use an open-source C++ Orion implementation to perform the experiments [63]. Note that the number of deletions or lazy additions in the matched results is not a key concern since OBI can rebuild the whole result set to the optimal size at any time. For a frequently searched keyword, its result size is always near the optimal value since redundant information has been removed.

We set the OMAP parameter of Orion to $Z = 4$. The number of matched results equals $R = 1024$. We set OBI to $u = 32$ and $Z = 4$. That is, a data block contains $u = 32$ file identifiers. Figure 10 shows a comparison of OBI and Orion on search time. When $L = 22$, an OBI search takes 10.7 ms, but Orion consumes 503 ms. This is because Orion enjoys an additional multiplicative logarithmic overhead in every query. Another reason is that OBI has improved access locality. If client-side storage is considered, since Orion achieves $O(1)$ client storage, OBI can also achieve $O(1)$ client storage by employing the OMAP to outsource $O(m)$ storage to the cloud. This conversion will incur additional $O(\log^2 m)$ computational overhead and $O(\log m)$ rounds per query. Compared with Orion of $O(n_w \log^2 N)$, $O(\log^2 m)$ can be ignored. Figure 9 shows a comparison of the two schemes on bandwidth usage. OBI uses less bandwidth since Orion shuffles more data blocks than OBI.

The file-insertion speed of OBI is three orders of magnitude higher than that of Orion, as shown in Figure 8. For a fair comparison, the tree height is set to 22 in both schemes. To insert a file with 10 000 distinct keywords, OBI takes only 69.97 ms. However, Orion consumes 206 seconds. This is because OBI has the fast multi-path eviction algorithms, KNNEA and PBEA. Orion can only black-box invoke the slow single-path ORAM (OMAP) interfaces to update data. The insertion efficiency of OBI-(u=128) is 10X faster than that of OBI-(u=1). However, we choose OBI-(u=32) since communication bandwidth is also a performance factor. OBI can be initialized in the optimal $O(N)$ time, yet there are no available solutions to initializing an Orion index in $O(N)$ time to date. In the experiment, if naively invoking the OMAP APIs to online build the Orion index, setting up the whole Enron index consumes more than one day.

Compared with the insecure baseline, the typical conjunctive-query DSE scheme VBtree in [23], the OBI-based DSE scheme OCQ has two orders of magnitude slowdown if communication time is not considered, as shown in Table 11. For simplicity, we write $DB(w) = 15$ to denote a keyword w that matches 15 results, and so on. The experiment is performed by testing the query processing time of a conjunctive query ($x \wedge w$), where x and w are keywords. Since $DB(w)$=15, the keyword w is the short term. The query processing time is slightly dependent on $DB(x)$. Since $DB(x)$ can be the entire

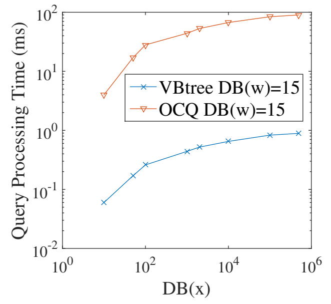

Fig. 11. OCQ and VBtree, running conjunctive queries (x ∧ w).

dataset, OCQ uses an oblivious index to avoid a linear scan of the whole database for conjunctive queries.

We compare ZPH-OBI with the size-pattern-hiding Eurus in TDSC 2020 [13]. We use the dataset DB1 and set $M = 10^4$ and $u = 32$ for both schemes. ZPH-OBI takes 50 ms per query. In comparison, Eurus runs out of the memory due to a required matrix of size $O(M^2 \log^2 m)$ bytes in its thin-path eviction algorithm. This is because Eurus suffers from the large-stash eviction problem. Moreover, OBI and ZPH-OBI protocols can be readily switched by the user for efficiently retrieving either a large result set or a small result set since OBI and ZPH-OBI share the same storage structure. In comparison, Eurus can retrieve only the maximum result set due to padding.

Compared with the non-oblivious DSE schemes, such as Cash et al.'s scheme [57], OBI consumes more time to shuffle and re-encrypt the result set of every data query at the user-side. This operation incurs an additional logarithmic multiplicative overhead. Based on our experimental results, Cash et al.'s scheme is still two orders of magnitude more efficient than OBI in data searches, assuming the same data locality is optimized in both schemes. However, result-set shuffling is unavoidable. Otherwise, the search and access patterns will be revealed and the attacks are still available [16], [17], [64]–[69].

OBI still has two limitations. First, OBI is suitable for only search-intensive applications instead of update-intensive DSE since OBI uses a lazy-deletion strategy. Second, a key-set mapping ORAM is not equivalent to a key-value mapping ORAM. OBI is optimized for only key-set mapping. The key-set mapping ORAM allows for some size pattern leakage in return for good throughput only if the user can partially sacrifice such privacy.

VIII. Conclusions

In this paper, we proposed OBI, a multi-path ORAM that supports oblivious bulk operations for protecting the search/access/size/kind patterns of DSE. OBI relies on two new multi-path eviction algorithms to provide notable features, including the optimal indexing speed, single-round-trip access, small stash size, and high bulk-insertion efficiency. We gave extensive experimental evaluations to show the merits of the multi-path eviction algorithms compared with PATH-ORAM and Orion. Despite the advantages, using ORAM means data are frequently shuffled and re-encrypted, which is very harmful to database systems. Regarding future work, one direction would be to design multi-user multi-path oblivious RAMs.

Acknowledgment

This work was supported in part by the National Key Research and Development Program of China under Grant 2021YFB3101300, the National Natural Science Foundation of China under Grants No. (61972089, 61972146), and the Hunan Provincial Natural Science Foundation of China under Grant (2022JJ30622, 2021JJ30743).

References

[1] "Apache Lucene," https://lucene.apache.org/, 2021.

[2] "Apache Cassandra," https://cassandra.apache.org/, 2021.

[3] R. Ostrovsky, "Efficient computation on oblivious RAMs," in *the twenty-second annual ACM Symposium on Theory of computing (STOC)*. ACM, 1990, pp. 514–523.

[4] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," *Journal of the ACM (JACM)*, vol. 43, no. 3, pp. 431–473, 1996.

[5] 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 *the 2013 ACM SIGSAC conference on Computer and Communications Security (CCS)*. ACM, 2013, pp. 299–310.

[6] S. Kamara, C. Papamanthou, and T. Roeder, "Dynamic searchable symmetric encryption," in *the 2012 ACM conference on Computer and Communications Security (CCS)*. ACM, 2012, pp. 965–976.

[7] S. Kamara and C. Papamanthou, "Parallel and dynamic searchable symmetric encryption," *Financial Cryptography and Data Security (FC)*, vol. 7859, pp. 258–274, 2013.

[8] 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*, 2015, pp. 837–849.

[9] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "A retrospective on Path ORAM," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2019.

[10] I. Demertzis, J. G. Chamani, D. Papadopoulos, and C. Papamanthou, "Dynamic searchable encryption with small client storage." in *ISOC Network and Distributed System Security Symposium (NDSS)*, 2020.

[11] J. Ghareh Chamani, D. Papadopoulos, C. Papamanthou, and R. Jalili, "New constructions for forward and backward private symmetric searchable encryption," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2018, pp. 1038–1055.

[12] R. Bost, B. Minaud, and O. Ohrimenko, "Forward and backward private searchable encryption from constrained cryptographic primitives," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2017, pp. 1465–1482.

[13] Z. Liu, Y. Huang, X. Song, B. Li, J. Li, Y. Yuan, and C. Dong, "Eurus: Towards an efficient searchable symmetric encryption with size pattern protection," *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2020.

[14] K. He, J. Chen, Q. Zhou, R. Du, and Y. Xiang, "Secure dynamic searchable symmetric encryption with constant client storage cost," *IEEE Transactions on Information Forensics and Security (TIFS)*, vol. 16, pp. 1538–1549, 2020.

[15] C. Zuo, S.-F. Sun, J. K. Liu, J. Shao, and J. Pieprzyk, "Dynamic searchable symmetric encryption with forward and stronger backward privacy," in *European Symposium on Research in Computer Security (ESORICS)*. Springer, 2019, pp. 283–303.

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

[17] S. Oya and F. Kerschbaum, "Hiding the access pattern is not enough: Exploiting search pattern leakage in searchable encryption," in *30th USENIX Security Symposium*, 2021.

[18] 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 (CCS)*. ACM, 2014, pp. 215–226.

[19] S. Garg, P. Mohassel, and C. Papamanthou, "Tworam: efficient oblivious ram in two rounds with applications to searchable encryption," in *Annual International Cryptology Conference (CRYPTO)*. Springer, 2016, pp. 563–592.

[20] T. Hoang, C. D. Ozkaptan, A. A. Yavuz, J. Guajardo, and T. Nguyen, "S^3 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 (CCS)*. ACM, 2017, pp. 491–505.

[21] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: improved definitions and efficient constructions," in *the 13th ACM conference on Computer and Communications Security (CCS)*. ACM, 2006, pp. 79–88.

[22] M. Chase and S. Kamara, "Structured encryption and controlled disclosure," in *the Theory and Application of Cryptology and Information Security (ASIACRYPT)*. Springer, 2010, pp. 577–594.

[23] Z. Wu and K. Li, "Vbtree: forward secure conjunctive queries over encrypted data for cloud computing," *The VLDB Journal*, vol. 28, no. 1, pp. 25–46, 2019.

[24] Z. Liu, S. Lv, Y. Wei, J. Li, J. K. Liu, and Y. Xiang, "Ffsse: Flexible forward secure searchable encryption with efficient performance," *IACR Cryptology ePrint Archive*, 2017.

[25] X. Song, C. Dong, D. Yuan, Q. Xu, and M. Zhao, "Forward private searchable symmetric encryption with optimized i/o efficiency," *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2018.

[26] G. Amjad, S. Kamara, and T. Moataz, "Forward and backward private searchable encryption with SGX," in *Proceedings of the 12th European Workshop on Systems Security*. ACM, 2019, p. 4.

[27] M. Naveed, "The fallacy of composition of oblivious ram and searchable encryption," *IACR Cryptol. ePrint Arch.*, vol. 2015, p. 668, 2015.

[28] L. Ren, C. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. Van Dijk, and S. Devadas, "Constants count: Practical improvements to oblivious RAM," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 415–430.

[29] Z. Chang, D. Xie, and F. Li, "Oblivious RAM: a dissection and experimental evaluation," *Proceedings of the VLDB Endowment*, vol. 9, no. 12, pp. 1113–1124, 2016.

[30] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li, "Oblivious RAM with $O(\log^3 N)$ worst-case cost," in *International Conference on The Theory and Application of Cryptology and Information Security (ASIACRYPT)*. Springer, 2011, pp. 197–214.

[31] D. S. Roche, A. Aviv, and S. G. Choi, "A practical oblivious map data structure with secure deletion and history independence," in *2016 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2016, pp. 178–197.

[32] B. Li, Y. Huang, Z. Liu, J. Li, Z. Tian, and S.-M. Yiu, "Hybridoram: practical oblivious cloud storage with constant bandwidth," *Information Sciences*, vol. 479, pp. 651–663, 2019.

[33] H. Chen, I. Chillotti, and L. Ren, "Onion Ring RAM: Efficient constant bandwidth oblivious RAM from (leveled) TFHE," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2019, pp. 345–360.

[34] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa, "Oblix: An efficient oblivious search index," in *2018 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2018, pp. 279–296.

[35] Z. Wu, K. Li, K. Li, and J. Wang, "Fast boolean queries with minimized leakage for encrypted databases in cloud computing," *IEEE Access*, vol. 7, pp. 49 418–49 431, 2019.

[36] S. Kamara and T. Moataz, "Boolean searchable symmetric encryption with worst-case sub-linear complexity," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2017, pp. 94–124.

[37] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Roşu, and M. Steiner, "Highly-scalable searchable symmetric encryption with support for boolean queries," in *Annual Cryptology Conference (CRYPTO)*. Springer, 2013, pp. 353–373.

[38] V. Pappas, F. Krell, B. Vo, V. Kolesnikov, T. Malkin, S. G. Choi, W. George, A. Keromytis, and S. Bellovin, "Blind seer: A scalable private dbms," in *Security and Privacy (S&P)*. IEEE, 2014, pp. 359–374.

[39] R. Li, A. X. Liu, A. L. Wang, and B. Bruhadeshwar, "Fast range query processing with strong privacy protection for cloud computing," in *Very Large Data Bases (VLDB)*, 2014, pp. 1953–1964.

[40] R. Li, A. X. Liu, A. L. Wang, B. Bruhadeshwar, R. Li, A. X. Liu, A. L. Wang, and B. Bruhadeshwar, "Fast and scalable range query processing with strong privacy protection for cloud computing," *IEEE/ACM Transactions on Networking (TON)*, vol. 24, no. 4, pp. 2305–2318, 2016.

[41] R. Li and A. X. Liu, "Adaptively secure conjunctive query processing over encrypted data for cloud computing," in *33rd International Conference on Data Engineering (ICDE)*. IEEE, 2017, pp. 697–708.

[42] ——, "Adaptively secure and fast processing of conjunctive queries over encrypted data," *IEEE Transactions on Knowledge and Data Engineering*, 2020.

[43] J. Chi, C. Hong, M. Zhang, and Z. Zhang, "Fast multi-dimensional range queries on encrypted cloud databases," in *International Conference on Database Systems for Advanced Applications*. Springer, 2017, pp. 559–575.

[44] I. Demertzis, S. Papadopoulos, O. Papapetrou, A. Deligiannakis, and M. Garofalakis, "Practical private range search revisited," in *the 2016 International Conference on Management of Data (SIGMOD)*. ACM, 2016, pp. 185–198.

[45] B. Wang, S. Yu, W. Lou, and Y. T. Hou, "Privacy-preserving multi-keyword fuzzy search over encrypted data in the cloud," in *IEEE INFOCOM 2014-IEEE Conference on Computer Communications*. IEEE, 2014, pp. 2112–2120.

[46] Z. Fu, X. Wu, C. Guan, X. Sun, and K. Ren, "Toward efficient multi-keyword fuzzy search over encrypted outsourced data with accuracy improvement," *IEEE Transactions on Information Forensics and Security (TIFS)*, vol. 11, no. 12, pp. 2706–2716, 2016.

[47] Z. Chang, D. Xie, F. Li, J. M. Phillips, and R. Balasubramonian, "Efficient oblivious query processing for range and knn queries," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2021.

[48] C. Sahin, V. Zakhary, A. El Abbadi, H. Lin, and S. Tessaro, "Taostore: Overcoming asynchronicity in oblivious data storage," in *2016 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2016, pp. 198–217.

[49] A. Chakraborti and R. Sion, "Concuroram: High-throughput stateless parallel multi-client oram," *arXiv preprint arXiv:1811.04366*, 2018.

[50] ——, "Sqoram: Read-optimized sequential write-only oblivious RAM," *Proceedings on Privacy Enhancing Technologies*, vol. 2020, no. 1, pp. 216–234, 2020.

[51] Z. Wu, Z. Cai, X. Tang, Y. Xu, and T. Deng, "A forward and backward private oblivious ram for storage outsourcing on edge-cloud computing," *Journal of Parallel and Distributed Computing*, vol. 166, pp. 1–14, 2022.

[52] S. Eskandarian and M. Zaharia, "Oblidb: oblivious query processing for secure databases," *Proceedings of the VLDB Endowment*, vol. 13, no. 2, pp. 169–183, 2019.

[53] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter, "Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017, pp. 2421–2434.

[54] S. Krastnikov, F. Kerschbaum, and D. Stebila, "Efficient oblivious database joins," *Proceedings of the VLDB Endowment*, vol. 13, no. 11, 2020.

[55] R. Bost, "$\sum o\varphi o\varsigma$: Forward secure searchable encryption," in *the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2016, pp. 1143–1154.

[56] E. Boyle, K.-M. Chung, and R. Pass, "Oblivious parallel ram and applications," in *Theory of Cryptography Conference*. Springer, 2016, pp. 175–204.

[57] 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," in *ISOC Network and Distributed System Security Symposium (NDSS)*, vol. 14. Citeseer, 2014, pp. 23–26.

[58] P. Williams, R. Sion, and A. Tomescu, "Privatefs: A parallel oblivious file system," in *Proceedings of the 2012 ACM conference on Computer and communications security (CCS)*, 2012, pp. 977–988.

[59] A. Ahmad, K. Kim, M. I. Sarfaraz, and B. Lee, "Obliviate: A data oblivious filesystem for intel sgx." in *ISOC Network and Distributed System Security Symposium (NDSS)*, 2018.

[60] "Enron email dataset," https://www.cs.cmu.edu/~./enron/, 2015.

[61] K. S. Kim, M. Kim, D. Lee, J. H. Park, and W.-H. Kim, "Forward secure dynamic searchable symmetric encryption with efficient updates," in *the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2017, pp. 1449–1463.

[62] J. Li, Y. Huang, Y. Wei, S. Lv, Z. Liu, C. Dong, and W. Lou, "Searchable symmetric encryption with forward search privacy," *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2019.

[63] R. Bost, "Opensse schemes," https://github.com/OpenSSE/, 2019.

[64] M. S. Islam, M. Kuzu, and M. Kantarcioglu, "Access pattern disclosure on searchable encryption: Ramification, attack and mitigation," in *ISOC Network and Distributed System Security Symposium (NDSS)*, vol. 20, 2012, p. 12.

[65] Y. Zhang, J. Katz, and C. Papamanthou, "All your queries are belong to us: The power of file-injection attacks on searchable encryption," in *USENIX Security Symposium*, 2016, pp. 707–720.

[66] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart, "Leakage-abuse attacks against searchable encryption," in *the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2015, pp. 668–679.

[67] V. Bindschaedler, P. Grubbs, D. Cash, T. Ristenpart, and V. Shmatikov, "The tao of inference in privacy-protected databases," *Proceedings of the VLDB Endowment*, vol. 11, no. 11, pp. 1715–1728, 2018.

[68] M.-S. Lacharité, B. Minaud, and K. G. Paterson, "Improved reconstruction attacks on encrypted data using range query leakage," in *2018 IEEE Symposium on Security and Privacy (S&P)*, 2017, pp. 19–36.

[69] G. Kellaris, G. Kollios, K. Nissim, and A. O'Neill, "Generic attacks on secure outsourced databases," in *the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2016, pp. 1329–1340.