Update-Sensitive Structured Encryption with Backward Privacy

Zhiqiang Wu, Jin Wang *Senior Member, IEEE*, Keqin Li *Fellow, IEEE*

***Abstract*—Many recent studies focus on dynamic searchable encryption (DSE), which provides efficient data-search and data-update services directly on outsourced private data. Most encryption schemes are not optimized for update-intensive cases, which say that the same data record is frequently added and deleted from the database. How to build an efficient and secure DSE scheme for update-intensive data is still challenging. We propose UI-SE, the first DSE scheme that achieves single-round-trip interaction, near-zero client storage, and backward privacy without any insertion patterns. UI-SE involves a new tree data structure, named OU-tree, which supports oblivious data updates without any access-pattern leakage. We formally prove that UI-SE is adaptively secure under Type-1$^-$ backward privacy, which is stronger than Type-1 backward privacy proposed by Bost et al. in CCS 2017. Experimental data also demonstrate UI-SE has low computational overhead, low local disk usage, and high update performance on scalable datasets.**

***Index Terms*—Backward Privacy, Cloud Computing, Dynamic Searchable Encryption, Forward Privacy, Oblivious RAM.**

I. Introduction

A. Background and Motivations

Nowadays, most companies and users outsource their private data to the cloud for convenience and ubiquitous services. However, the cloud servers perhaps can be broken by hackers due to security vulnerabilities and untrusted network managers. To address these, the researchers proposed searchable encryption (SE) that encrypts the private data in a way that still allows the user to search on encrypted data directly [1]. To update the data, they also proposed dynamic searchable encryption (DSE), which allows the user to update outsourced encrypted data efficiently and privately [2].

A DSE scheme involves two parties, the user (the client) and the cloud (the server). Assume the user is trusted, and the cloud is untrusted. The cloud always wants to break the user's private data in any data searches and data updates. The user initially outsources an encrypted index to the cloud for data searching and updating, where the index can map any search queries into a set of encrypted file identifiers. Current studies show that forward and backward privacy is necessary for a DSE scheme [3]. The forward privacy says that any data updates should appear no correlations with the historical search queries and update queries. If a DSE scheme does not achieve forward privacy, the adaptive file injection attack proposed in [4] can easily break the DSE scheme. The backward privacy says that any data searches should have no correlations with the historical deleted entries. If a DSE scheme does not achieve backward privacy, a search query will reveal the timestamps of the historically deleted data, which are perhaps very important to the user.

Zhiqiang Wu and Jin Wang are with the School of Computer and Communication Engineering, Changsha University of Science and Technology, Hunan, China, 410114. (wzq@csust.edu.cn, jinwang@csust.edu.cn).

Keqin Li is with the Department of Computer Science, State University of New York, New Paltz, NY 12561. (lik@newpaltz.edu)

Jin Wang is the corresponding author.

Consider the following usage scenario. To delete a user from a financial trading system, most banks do not directly remove the user from the database. They only mark the user record as deleted. When the user comes back, the banks add the user to the database again. If the same user record has been frequently removed and added, this scenario is called update-intensive. In other words, all new updates only mark the existing records as inserted or deleted after database initialization, and there are no actual deletions. Our target is to build an efficient and secure update-intensive DSE scheme that achieves both forward privacy and backward privacy.

B. Limitations of Prior Art

Most backward-private DSE schemes do not consider the update-intensive cases. There are three parameters to describe the update length. For any updates to keyword w, the number of entries matching keyword w is labelled with a_w, the number of real files containing keyword w is denoted by n_w, and the number of entries matching keyword w, when the index is initialized, is described as c_w. For example, assuming an encrypted database consists of only one keyword-identifier pair $(w, 1)$ after initialization, and this pair has been alternately added and deleted 10000 times each, we have $a_w = 20000$, $c_w = 1$, and $n_w = 0$.

In TIFS 2020 [5], He et al. proposed a backward-private scheme CLOSE-FB using a fish-bone data structure to record every data updates. Unfortunately, the fish bone can not support intensive modifications since its efficiency is linear in a_w. The structure can be updated only in a fixed number of times. In TDSC 2019 [6], Li et al. introduced an efficient DSE solution Khons whose search complexity is proportional to only n_w. However, Khons has unscalable client-side storage that contains a large set of chain headers.

Many DSE schemes [7] achieve small-client storage and backward privacy by employing the state-of-the-art ORAM, OMAP proposed in [8]. However, OMAP still suffers from $O(\log N)$ client-server interactions and $O(\log^2 N)$ communication bandwidth per query, where N is the database size, the total number of keyword-identifier pairs.

TABLE I
COMPARISONS OF SMALL-CLIENT DSE SCHEMES ON UPDATE-INTENSIVE CASES

Scheme	Search			Update			CS	BP
	Computation	Communication	RT	Computation	Communication	RT		
TWORAM [9]	$O(a_w \log N + \log^3 N)$	$O(a_w \log N + \log^3 N)$	2	$\widetilde{O}(\log^2 N)$	$\widetilde{O}(\log^3 N)$	2	$O(1)$	-
QOS [10]	$O(n_w \log i_w + \log^2 m)$	$O(n_w \log i_w + \log^2 m)$	$O(\log m)$	$O(\log^3 N)$	$O(\log^3 N)$	$O(\log N)$	$O(1)$	(I, III)
SD$_d$ [10]	$O(a_w + \log N)$	$O(a_w + \log N)$	1	$O(\log^3 N)$	$O(\log^3 N)$	$O(\log N)$	$O(1)$	(I, II)
CLOSE-FB [5]	$O(a_w + CLen)$	$O(a_w)$	1	$O(CLen)$	$O(1)$	1	$O(1)$	(I, II)
Orion [7]	$O(n_w \log^2 N)$	$O(n_w \log^2 N)$	$O(\log N)$	$O(\log^2 N)$	$O(\log^2 N)$	$O(\log N)$	$O(1)$	(I, I)
UI-SE	$O(c_w \log N)$	$O(c_w \log N)$	1	$O(\log N)$	$O(\log N)$	1	$O(1)$	(I, I$^-$)

N is the number of keyword-identifier pairs, and m is the number of distinct keywords. n_w is the number of real files containing keyword w, a_w is the number of entries matching keyword w, i_w is the number of insertions, and c_w the number of entries matching keyword w when the index is initialized. $a_w \geq i_w \geq c_w \geq n_w$. RT denotes the number of interaction round trips. CS denotes client storage. BP denotes backward privacy. Update complexity is given per keyword-identifier pair. $CLen$ is a constant value that denotes the maximum number of updates.

C. Proposed Approach

To address all the above challenges, instead of adopting an oblivious RAM, we proposed an oblivious-update tree (OU-tree) structure to achieve backward privacy, small client-side storage, and single-round-trip interaction. The OU-tree puts the keyword-identifier pairs into the leaves and organizes the frequent updates into the leaf-to-root paths. Any update only accesses a leaf-to-root path whose leaf identifier is always in random distribution. Any search retrieves a leaf-to-root path that contains all the updates of the corresponding keyword. Even if the same keyword-identifier pair is updated frequently, the accessed nodes are always of a fixed size. With the OU-tree, the search efficiency is linear in only c_w instead of a_w. The OU-tree does not rely on any large client-side storage structures, such as local chain headers [6], since choosing an update path does not depend on anything.

D. Our Contributions

Our contributions are summarized as follows:

1) We propose UI-SE, the first update-intensive DSE scheme that achieves single-round-trip interaction, small-client storage, and insertion-pattern-hiding backward privacy.

2) We propose OU-tree, a new tree structure that supports efficient oblivious updates without relying on any ORAMs.

Table I lists the comparisons of typical small-client DSE schemes on update-intensive cases. TWORAM [9], QOS [10], SD$_d$ [10] and Orion [7] are ORAM-based DSE schemes. As for backward privacy, we only quantify the index accessing leakage.

II. RELATED WORK

Searchable encryption was first introduced by Song et al. in 2000 [11]. Curtmola et al. proposed the widely-used adaptive security definition in [1]. Dynamic searchable encryption first appeared in [2]. DSE provides many features, such as conjunctive queries [12], [13], range queries [14], Boolean queries [15], [16], fuzzy queries [17], [18], and graph queries [19]. Many subsequent works focused on the studies of forward privacy of DSE schemes [3], [7], [12], [20]–[23]. Almost all recent works rely on a client-side per-keyword map that stores the keyword states to improve DSE security [3], [7], [12], [21], [24], [25]. This structure is a heavy burden for resource-constrained devices.

An approach to eliminating the client-side per-keyword map is to employ a low-computation oblivious RAM, such as Path ORAM [26], [27], OMAP [8], and multi-server S^3-ORAM [28]. Many ORAMs are not suitable for DSE due to heavy computational overhead and large data-block size, such as Onion Ring ORAM [29], whose block size reaches 300 KB. TWORAM [9] achieves small data-block size, single-round-trip access, and small-client storage. Unfortunately, TWORAM relies on heavy garbled circuits [30], [31] that act as the practical bottleneck in performance. Liu et al. proposed an ORAM-based DSE scheme that wants to hide the size pattern [32].

Current studies showed that DSE schemes still suffer from many kinds of access-pattern-based attacks [4], [33]–[35]. One approach to reducing the access-pattern leakage is to employ the ORAMs [7], or shuffle the accessed locations in data reads and writes like this work.

Other encrypted searches include homomorphic encryption [36], order-preserving encryption [37], and secure multi-party computation [38].

III. NOTATIONS AND DEFINITIONS

In this section, we recall the encryption protocols and the security definitions, such as adaptive security and forward and backward privacy, which regularly appear in current DSE schemes.

A. DSE Protocols

Result-revealing. A result-revealing DSE scheme is a user-cloud protocol, whose file-identifier results generated by a search query are directly exposed to the cloud for retrieving data files. The file identifier is a bit string that uniquely represents the data file. The advantage of result-revealing SE is that the cloud can handle the data-search work in a single-round interaction (not single-round-trip).

Result-hiding. A result-hiding DSE scheme is a user-cloud protocol, whose file-identifier results generated by a search query are hidden from the cloud. The advantage of result-hiding DSE is that it helps the DSE scheme achieve backward privacy at the cost of client-side computation.

B. Privacy Definitions

We summary the privacy in DSE schemes: search pattern, full query pattern, update pattern, insertion pattern, size pattern, etc.

Search/full query pattern. Let Q be a sequence of search or update queries issued so far after index initialization. Each query has the form of $(i, search, w)$ or $(i, op, (w, id))$, where $(i, search, w)$ denotes the i-th search tuple for keyword w, and $(i, op, (w, id))$ denotes the i-th update (addition or deletion) with a keyword-identifier pair (w, id). The search pattern $sp(w)$ and the full query pattern $qp(w)$ are defined as follows,

$$sp(w) = \{i|(i, search, w) \in Q\}$$
$$qp(w) = \{(i, op)|(i, search, w) \in Q \ or \ (i, op, (w, id)) \in Q\},$$

where $op = \perp$, add, or del for a search, an addition, or a deletion, respectively. The search pattern is the repetition of the search tokens. The full query pattern includes not only the search pattern, but also the timestamps and the corresponding operation names of the historical updates.

Update pattern. The update pattern $up(w)$ induced by a search or an update query is defined as follows,

$$up(w) = \{(i, op)| \ (i, op, (w, id)) \in Q\}.$$

The update pattern is a part of the full query pattern. If an update query reveals the search pattern, this leakage is disastrous [4]. Therefore, most schemes allow only $up(w)$ leakage of the update query.

We also recall some notations used in prior works, $HistDB(w)$, $TimeDB(w)$, $Updates(w)$, and $DelHist(w)$:

$$\begin{aligned} &HistDB(w) = \{(i, id)|(i, add, (w, id)) \in Q \ \},\\ &TimeDB(w) = \{(i, id)|(i, add, (w, id)) \in Q\\ &\quad and \ \forall j, (j, del, (w, id)) \notin Q\},\\ &Updates(w) = \{i|(i, add, (w, id)) \in Q\\ &\quad or \ (i, del, (w, id)) \in Q\},\\ &DelHist(w) = \{(i, j)|\exists id : (i, add, (w, id)) \in Q\\ &\quad and \ (j, del, (w, id)) \in Q\}.\end{aligned}$$

$HistDB(w)$ is a sequence of file identifiers added to the database with their corresponding timestamps. $TimeDB(w)$ is a sequence of identifiers of files containing w, excluding the deleted identifiers, together with the timestamps when they are inserted. $Updates(w)$ is a sequence of update timestamps. $DelHist(w)$ is a sequence of pairs, whose second entry is the deletion timestamp on w, and whose first entry is the corresponding insertion time.

Insertion pattern. The insertion pattern $ip(w)$ is defined as follows,

$$\begin{aligned} ip(w) = \{i|(i, add, (w, id)) \in Q\\ and \ \forall j, (j, del, (w, id)) \notin Q\}.\end{aligned}$$

Size pattern. The size pattern $Size(w)$ reveals the number of matched entries from the index for w.

C. Security Definitions

Adaptive security. We adopt the adaptive-security definition, proposed in CCS 2006 [1], and modified by [2] and [24] for DSE. A DSE scheme is said to be $\mathcal{L}$-adaptively-secure if and only if for any probabilistic polynomial-time (PPT) adversary $\mathcal{A}$, who can adaptively query the random oracle for search and update tokens, there exists a simulator $\mathcal{S}$ such that the execution of $\mathcal{S}(\mathcal{L})$ and the execution of $\mathcal{A}$ are computationally PPT indistinguishable. The adaptive security guarantees that even if the adversary $\mathcal{A}$ can choose keywords for attacks adaptively (CKA2), $\mathcal{A}$ learns nothing from the index except $\mathcal{L}$. CKA2 differs from the nonadaptively choosing keywords for attacks (CKA) [39].

Forward privacy. An $\mathcal{L}$-adaptively-secure index-based DSE scheme is forward-private if and only if the update leakage function $\mathcal{L}^{Update}$ can be written as

$$\mathcal{L}^{Update}(op, w, id) = \mathcal{L}'(op),$$

where id denotes the identifier of the newly added file, w the updated keyword, and $\mathcal{L}'$ is a stateless function.

We consider only the leakage induced by accessing the encrypted index. Data-file retrieving as well as its leakage analysis are not our focus. A forward-private index means that its addition and deletion reveal no query patterns other than the operation name.

Backward privacy. Let Π be an $\mathcal{L}$-adaptively-secure DSE scheme parameterized by two leakage functions $\mathcal{L}^{Update}$ and $\mathcal{L}^{Search}$. Π is said to be BP-(x, y)-private if the leakage level of data queries satisfies the following definition:

$$\begin{cases} x = I : \mathcal{L}^{Update}(op, w, id) = \mathcal{L}'(op)\\ x = II : \mathcal{L}^{Update}(op, w, id) = \mathcal{L}'(op, w)\\ y = I : \mathcal{L}^{Search}(w) = \mathcal{L}''(sp(w), TimeDB(w))\\ y = II :\\ \quad \mathcal{L}^{Search}(w) = \mathcal{L}''(sp(w), TimeDB(w), Updates(w)))\\ y = III :\\ \quad \mathcal{L}^{Search}(w) = \mathcal{L}''(sp(w), TimeDB(w), DelHis(w))\\ y = I^{-} : \mathcal{L}^{Search}(w) = \mathcal{L}''(sp(w), Size(w)) \end{cases} \tag{1}$$

where $\mathcal{L}'$ and $\mathcal{L}''$ are stateless functions.

Type-I update leakage contains nothing except the operation name. Type-II update leakage has the update pattern. Type-I search leakage allows the leakage of when the same keyword w is searched, and the file identifiers matching w with the insertion timestamps, excluding the deleted identifiers. Type-II search leakage further allows when all the updates on w happen. Type-III search leakage further allows the leakage of which deletion update cancels which insertion update. According to the Bost et al.'s backward privacy definition [3], BP-(I, I) is called backward privacy with insertion pattern, BP-(*, II) is called backward privacy with update pattern, and BP-(*, III) is called weak backward privacy. According to the forward-privacy definition, BP-(I, I) and BP-(I, II) can be viewed as achieving both forward and backward privacy. Note that a DSE scheme that has $\mathcal{L}^{Update}$ does not imply the existence of $\mathcal{L}^{Search}$, and vice versa.

Backward privacy without insertion patterns. We give a straightforward definition that is slightly stronger than BP-(I, I). If the search leakage contains nothing of the insertion patterns, it is denoted as Type-I^{-}. BP-(I, I^{-}) is called backward privacy without insertion patterns, assuming the update leakage is also Type-I. BP-(I, I^{-}) allows the leakage of the search pattern and the size pattern on search queries.

The definition (1) differs from the Bost et al.'s backward-privacy definition [3] whose leakage function does not explicitly contain the search pattern. Note that given $TimeDB(w)$, the adversary can not deduce $sp(w)$. For example, consider the following access sequence, $((1, add, id_1, \{w_1\}), (2, search, w_1), (3, search, w_1))$. The leakage of the second search appears the same as the first according to $TimeDB(w)$, but the second search contains more leakage than the first. That is, $TimeDB(w_1)^{(2)} = TimeDB(w_1)^{(3)} = \{(1, id_1)\}$, yet $\mathcal{L}^{(2)Search}(w_1) \neq \mathcal{L}^{(3)Search}(w_1)$, where each (i) denotes the i-th time. Therefore, $sp(w)$ should be explicitly included in the leakage definition of backward privacy.

We note that forward privacy is not a necessity of backward privacy. Consider the following awkward encryption scheme, named AS, which consists of three algorithms, AS=$\{$Setup, Search, Update$\}$. In the setup stage, the user saves the unencrypted database DB locally. In the update stage, the user encrypts the keyword by a CPA-secure private-key algorithm and uploads it to the cloud. The user also saves the updates into DB. In the search stage, the user queries DB locally. It is easy to see that AS is adaptively BP-(II, $\perp$)-secure since the search leaks nothing. It has backward privacy but no forward privacy.

IV. OU-tree: An oblivious-update tree

In this section, we propose OU-tree, a new tree structure for highly-intensive keyword-identifier updates, such as frequently adding or deleting an existing (w, id) pair. The advantage of OU-tree is that it supports efficient updates without any access patterns.

A. *OU-tree*

An L-height oblivious-update tree (OU-tree) is an encrypted full binary tree containing $2^L - 1$ tree nodes with the following properties.

1) Elements in any tree node are triplets that have the form of (w, id, v), where (w, id) is a keyword-identifier pair, and $v = 1$ or 0 denotes a version, inserted or deleted, respectively.
2) Each internal node can hold a set of Z triplets at most. Each leaf has at most R triplets.
3) Any keyword-identifier pair in the tree has at most L different versions that follow a leaf-to-root path. The new version is always near the root.
4) In the initial stage, all the leaves randomly store a set of keyword-identifier pairs, and all the internal nodes are initially empty.

Figure 1 shows a 3-height OU-tree example. In the initial stage, the leaves store all the keyword-identifier pairs

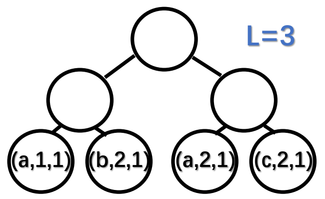

Fig. 1. An initialized-OU-tree example.

$\{(a, 1), (b, 2), (a, 2), (c, 2)\}$ that belong to two data files, where $(a, 2, 1)$ denotes $(a, 2)$ is inserted and $(a, 2, 0)$ for deletion of $(a, 2)$. All the internal nodes are initially empty. Thus, if all the leaves initially save N keyword-identifier pairs that are the whole database, and if all the internal nodes occupy $O(N)$ empty-triplet space, the tree requires storage space of $O(N)$ since the tree is balanced.

The OU-tree uses the following user-cloud protocol to hold the frequently updated keyword-identifier pairs. In the initial stage, the user initializes and encrypts an OU-tree, and then puts it into the cloud. In the update stage for updating a keyword-identifier pair (w, id), instead of directly accessing this pair, the user randomly chooses a leaf-to-root path and inserts a new triplet (w, id, v) into this path. Each (w, id) perhaps has many update versions. If (w, id) has been frequently added and deleted, only the latest version that nears the root is necessary, and other obsolete versions, if possible, can be removed though they perhaps still exist in the tree. More specifically, the protocol works with the following algorithms $\{$Setup, WritePath, ReadPath$\}$.

Setup. Setup is an algorithm that converts an unencrypted inverted index DB into an initialized OU-tree, as shown in Algorithm 1. Let F be a keyed collision-resistant hash function modelled as a random oracle, $\mathbb{K}$ be a user's secret key, L be the height of the tree, any keyword-identifier pair (w, id) will map to a leaf, denoted

$$MapLeaf(w, id) \overset{def}{=} F_{\mathbb{K}}(w||id)\%2^{L-1},$$

where leaf identifier ranges from 0 to $(2^{L-1} - 1)$. Given an unencrypted inverted index DB containing a set of keyword-identifier pairs, the algorithm puts any triplet $(w, id, 1)$ into the leaf whose identifier is $MapLeaf(w, id)$. All the leaf nodes are padded with dummy values to the same size R and are encrypted by an RCPA-secure private-key encryption algorithm. With the above approach, the OU-tree is initialized.

Algorithm 1 Setup an OU-tree.

```
Setup($DB$)
  1) initialize an empty $L$-height OU-tree $T$.
  2) for all $(w, id)$ in $DB$
      a) $leaf \leftarrow F_{\mathbb{K}}(w||id)\%2^{L-1}$
      b) put $(w, id, 1)$ into the leaf node $leaf$.
  3) encrypt all the leaves of $T$.
  4) return $T$
```

WritePath. WritePath is an algorithm to mark a keyword-identifier pair as inserted or deleted, as shown in Algorithm

2. Given an input triplet (w, id, v), the algorithm uses the following steps to mark (w, id) as v obliviously. 1) The user randomly generates a leaf identifier $leaf \in [0, 2^{L-1} - 1]$. 2) The cloud sends the whole $leaf$-to-root path to the user. 3) For any (w', id') in the path, the user removes all the obsolete keyword-identifier pairs related to (w', id'), and saves only one latest version that nears the root. 4) All the remaining triplets are put into the local stash $Stash$. 5) The user writes the original input (w, id, v) into $Stash$ for rebuilding the path later. 6) All the triplets will be evicted to the path with two conditions. One condition is that for any (w'', id'', v'') in Stash, it should be evicted to the overlapped nodes of the $leaf''$-to-root path and $leaf$-to-root, where $leaf'' = MapLeaf(w'', id'')$. The other condition is that all the triplets are stacked from bottom to top along the path sequentially. In an extreme case, if the path is full, some triplets remain in the local stash, but our experiments in the last section show that the probability of the stash size exceeding can be negligible. All the tree nodes in the path are padded with dummy values to the same size Z. Finally, the user encrypts the path using the RCPA-secure private-key encryption algorithm and uploads the whole path. The new triplet (w, id, v) is obliviously put into the tree with these steps.

In Algorithm 2, $CrossLevel(leaf, leaf')$ denotes the level of the intersection node of the $leaf$-to-root path and the $leaf'$-to-root path. For example, in Figure 3, $CrossLevel(C, D) = 2$. $CrossLevel(leaf, leaf') \leq i$ means that the current triplet can be evicted to the i-th level. The intuition behind this update algorithm is that even if $leaf$ is randomly chosen, the new-version keyword-identifier pair can still be accurately retrieved later through reading the $(MapLeaf(w, id))$-to-root path.

Algorithm 2 Update an OU-tree.

```
WritePath$(w, id, v)$
  1) $leaf \leftarrow random(0, 2^{L-1} - 1)$
  2) download the $leaf$-to-root path
  3) decrypt and put all the triplets into the $Stash$. For any
     different-version pairs, only the pair that nears the root
     is saved, and the other versions are discarded.
  4) write $((w, id), v)$ into $Stash$
  5) create $L$ empty nodes, $ND_1 = \{\}, \cdots, ND_L = \{\}$.
  6) for $i = L$ to 1
     a) for all $((w', id'), v')$ in $Stash$
        i) if $CrossLevel(MapLeaf(w', id'), leaf) \leq i$
           and $ND_i$ is not full,
           A) put $(w', id', v')$ into $ND_i$
           B) remove $((w', id'), v')$ from $Stash$
           C) if $ND_i$ is full, break
  7) encrypt $\{ND_L, ND_{L-1}, \cdots, ND_1\}$ and send them to
     the cloud
  8) the cloud overwrites the existing $leaf$-to-root path
```

Figure 2 shows an example of how to update an OU-tree. Assuming Node A, Node B, and Node C have the triplet $(a, 2, 1)$ each, and the triplet that nears the root is always the fresh one. To delete $(a, 2)$, instead of directly removing it from the leaves, the user inserts $(a, 2, 0)$ with the following

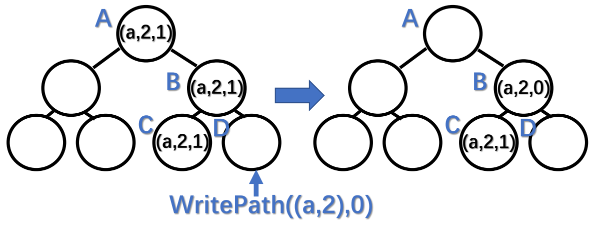

Fig. 2. Data updates in an OU-tree.

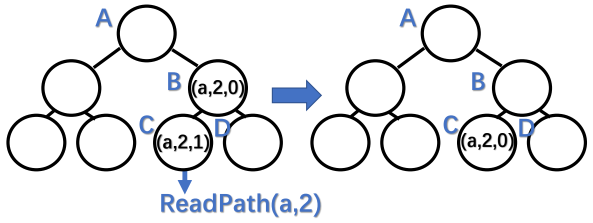

Fig. 3. Data reads in an OU-tree.

steps. First, the user randomly chooses a leaf identifier for insertion. Note that the randomly accessed leaf helps the access algorithm achieve obliviousness. Assuming Node D has been chosen, the access path is $\{A, B, D\}$. Second, the tree removes $(a, 2, 1)$ in both Node A and Node B since they are obsolete. Third, the tree inserts $(a, 2, 0)$ into Node B. Note that directly touching Node C is not allowed now since this will violate the access-pattern constrain, and the triplet can be inserted into only Node B or Node A.

ReadPath. ReadPath is an algorithm to test whether a (w, id) has been deleted from the tree or not. Given a leaf identifier $leaf$ generated by $MapLeaf(w, id)$, the cloud sends the whole $leaf$-to-root path to the user, who then decrypts the tree nodes and reads the desired file identifier. The user removes all the historical version (w, id) pairs, writes the latest version (w, id, v) into the leaf node, and reencrypts the whole $leaf$-to-root path.

Algorithm 3 Read from an OU-tree.

```
ReadPath$(w, id)$
  1) $leaf \leftarrow MapLeaf(w, id)$
  2) decrypt the $leaf$-to-root path
  3) read $v$ with $(w, id)$ from the node that is the nearest
     to the root
  4) set $v$ as the result
  5) remove $(w, id, 0)$ and $(w, id, 1)$ from the path
  6) write $(w, id, v)$ into the leaf node
  7) reencrypt the path
```

Figure 3 illustrates an example of how to read from an OU-tree. Assume $MapLeaf(a, 2) = C$. Given the pair $(a, 2)$, the algorithm touches Nodes $\{A, B, C\}$. If there exist two triplets $\{(a, 2, 1), (a, 2, 0)\}$ in the path, only the triplet $(a, 2, 0)$ that nears the root is saved, and it will be moved into leaf C.

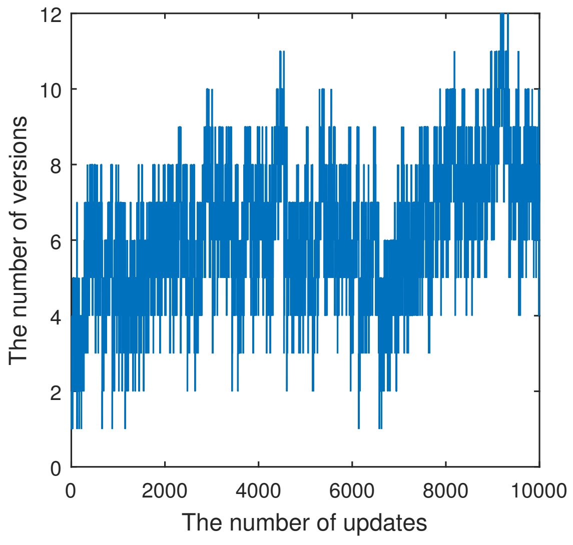

Fig. 4. The number of versions of the same pair.

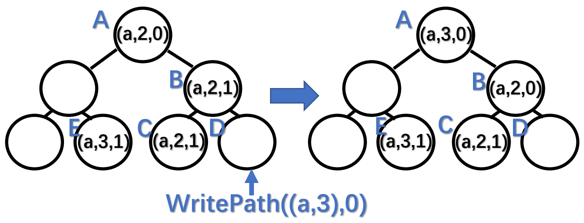

Fig. 5. Data updating and shuffling.

B. *Efficiency Analysis*

How many versions of a pair do exist in the tree? Intuitively, for any (w, id), the tree can hold at most L versions for this pair since the tree is L-height. We perform the following experiment to observe how many versions of a pair exist in the tree.

Consider that the pair has been updated s times. On the x-th updates, the number of versions of (w, id) is denoted by $V^{s,L}_{w,id}(x)$. Figure 4 illustrates the relationship between the number of updates and $V^{10000,16}_{w,id}(x)$. The pair has been updated 10000 times. As x grows larger, there are more versions in the tree with a considerable probability. In the average case, there are 6 versions in the tree if $L = 16$ and $s = 10000$.

In reality, there are fewer versions than those of the experiments. We observe two facts. 1) Any update on (w', id') $((w', id') \neq (w, id))$ will reduce the number of versions of (w, id) only if the updated leaf-to-root path is near the $MapLeaf(w, id)$-to-root path. For example, in Figure 5, the update $((a, 3), 0)$ on the path (A, B, D) rebuilds Nodes A and B. $(a, 2, 0)$ is moved into Node B, and $(a, 2, 1)$ is deleted. 2) Any read on (w, id) will reduce the number of versions of (w, id) to 1 since all the obsolete versions are removed. Therefore, the probability of the tree being full is negligible.

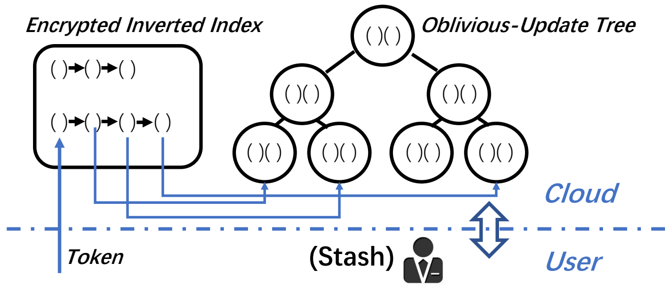

Fig. 6. A design overview.

V. UI-SE: An update-intensive backward-private DSE scheme

A. *A Design Overview*

Recall that the user should first know both the keyword and the identifier to read a value from an OU-tree. If the user knows only the keyword, how to read all the related identifiers? To address this, we store the OU-tree leaves into a new data structure, the encrypted inverted index (EII), which converts a keyword query to a set of accessing leaves. With EII and OU-tree, a new backward-private DSE scheme, named UI-SE, is proposed.

UI-SE is a three-tuple DSE scheme of UI-SE=(UI-Setup, Search, Update). UI-Setup is to build an index for later data outsourcing, Search is for keyword queries, and Update is to insert or delete an existing keyword-identifier pair frequently.

UI-SE involves three fundamental storage structures, an EII, an OU-tree, and a stash, as shown in Figure 6. The EII is a static data structure. After initialization, the EII is always unchanged. The OU-tree is a dynamic data structure that supports highly-intensive updates. Assuming these structures have been set up, a keyword query works as follows. The user sends a keyword token constructed from the keyword to the cloud. And then, a set of leaf identifiers is matched from EII. These leaves are used to download a set of leaf-to-root paths. The user decrypts the paths at the client-side, reads the final results, reencrypts the paths, and uploads the new paths. The local stash is designed for temporary data shuffling. For any updates, the algorithm touches only the OU-tree instead of the EII. Thus, no update pattern is revealed due to the oblivious access algorithm of the OU-tree.

B. *UI-SE: An Update-Intensive DSE Scheme*

WriteMap/ReadMap. To illustrate UI-SE, we first introduce two subprocedures, WriteMap and ReadMap. They are used to write/read from the enrypted inverted index that is stored in a hash table map. WriteMap is a procedure to encrypt and write (w, id) into map, as shown in Algorithm 4. Let H_1 and H_2 be two different collusion-resistant hash functions modeled as random oracles, F be the keyed pseudorandom function, and $\mathbb{K}$ be the secret key. The identifier id is encrypted by an RCPA-secure private-key encryption algorithm into the encrypted form of eid. Given a randomized unique input var, the algorithm first computes $key \leftarrow H_1(var)$ and $mask \leftarrow H_2(var)$. Next, the (w, id) pair is mapped to $var' \leftarrow F_{\mathbb{K}}(w||id)$, which is a new randomized unique value used in the next time. The encrypted key-value pair $(key, mask \oplus (var'||eid))$ is written into the hash table.

Algorithm 4 Two subprocedures for accessing an EII.

```
WriteMap($map, var, w, id$)
  1) $key \leftarrow H_1(var)$
  2) $mask \leftarrow H_2(var)$
  3) $var' \leftarrow F_{\mathbb{K}}(w||id)$
  4) encrypt $id$ into $eid$ by an RCPA-secure algorithm
  5) $map[key] \leftarrow mask \oplus (var'||eid)$, assuming $map$ is stored globally.
  6) return $var'$
ReadMap($map, var$)
  1) $key \leftarrow H_1(var)$
  2) $mask \leftarrow H_2(var)$
  3) $(var', eid) \leftarrow map[key] \oplus mask$
  4) return $(var', eid)$
```

ReadMap is an algorithm to read a tuple from the EII, as shown in Algorithm 4. Given map and an input var, the algorithm returns $map[H_1(var)] \oplus H_2(var)$, whose left part is var' the value to decode the next tuple, and whose right part is eid the encrypted file identifier. ReadMap is the counterpart of WriteMap.

Algorithm 5 Set up a UI-SE index.

```
UI-Setup($DB$)
  1) for all $w$ in $DB.W$
     a) $var \leftarrow F_{\mathbb{K}}(w)$
     b) for all $id$ in $DB(w)$
        i) $var \leftarrow WriteMap(map, var, w, id)$
  2) invoke $tree \leftarrow OU\text{-}tree.Setup(DB)$
  3) return $(map, tree)$
```

UI-Setup. Given an unencrypted inverted index DB that constructed from a set of files, the user locally sets up an EII and an OU-tree, as shown in Algorithm 5. Let $DB.W$ be the set of keywords that can be queried, and $DB(w)$ be the set of identifiers of the files containing keyword w. For any keyword w in $DB.W$ and for any (w, id) in $DB(w)$, the user stores all the (w, id) pairs through repeatedly invoking $var \leftarrow WriteMap(map, var, w, id)$. The first entry is $var \leftarrow F_{\mathbb{K}}(w)$, which can be viewed as the chain header. After the EII initialization, the user invokes $tree \leftarrow OU\text{-}tree.Setup(DB)$ to build the tree. The user then outsources $(map, tree)$ to the cloud.

Searching for a keyword. Searching for a keyword consists of three steps, reading the encrypted identifiers from the EII, reading the updates from the OU-tree, and reencrypting the accessed paths of the tree, as shown in Algorithm 6.

Given the chain header $var \leftarrow F_{\mathbb{K}}(w)$, the cloud retrieves all the leaves through repeatedly invoking $(var, eid) \leftarrow ReadMap(map, var)$. This step outputs two collections, one is the set of encrypted file identifiers D, whose length equals $c_w = |D|$, and the other collection is the set of leaf identifiers,

Algorithm 6 A search protocol.

```
Search($w$)
  1) **User**: $var \leftarrow F_{\mathbb{K}}(w)$; send it to the cloud;
  2) **Cloud**: $T \leftarrow \{\}$, $D \leftarrow \{\}$
  3) while $var \neq \perp$ do
     a) $(var', eid) \leftarrow ReadMap(map, var)$
     b) $D \leftarrow D \bigcup eid$
     c) $leaf = var' \% 2^{L-1}$
     d) put the $leaf$-to-root tree nodes into $T$
     e) $var \leftarrow var'$
  4) send $\{D, T\}$ to the user
  5) **User**:
  6) decrypt $D$ for generating a set of file identifiers $D'$
  7) decrypt $T$ and get an unencrypted small tree $T'$
  8) $T'' \leftarrow \{\}$
  9) for all $id$ in $D'$
     a) $leaf \leftarrow MapLeaf(w, id)$
     b) read the latest $(w, id)$ from $T'$, and get $v$.
     c) If $v = 1$, output one final result $id$.
     d) remove $(w, id, 0)$ and $(w, id, 1)$ from the $leaf$-to-root path of $T'$.
     e) write $(w, id, v)$ into node $leaf$ of $T'$.
     f) reencrypt the path nodes and store the reencrypted tree nodes into $T''$.
 10) upload $T''$ for replacing the old paths
```

which correspond to a set of leaf-to-root paths T. Recall that in an OU-tree, any (w, id) is mapped to its leaf by invoking $MapLeaf(w, id)$, which equals $F_{\mathbb{K}}(w, id)\%2^{L-1}$. T contains all the updates on w. The cloud sends (D, T) to the user.

The user gets a set of file identifiers by decrypting D. These are not the final results since some identifiers perhaps have been marked as deleted in the index. The user should decode T to test which identifiers are deleted. For any returned pair (w, id), the user invokes $OU\text{-}tree.ReadPath(w, id)$ to test whether the pair is deleted or not. Since all the related paths are downloaded and stored locally, this algorithm is independent of the cloud. The user outputs a set of filtered identifiers that are the final results, saves only the latest versions, and re-encrypts the paths by the RCPA-secure private-key algorithm.

Updating a pair. To update a pair (w, id) as inserted or deleted, the user only needs to invoke $OU\text{-}tree.Update(w, id, v)$ API with $v = 1$ for addition and $v = 0$ for deletion, as shown in Algorithm 7. The latest pair will be written into the tree on a randomly chosen path.

Algorithm 7 Update a frequently-accessed pair

```
Update($op, (w, id)$)
  1) if $op = add$,
        invoke $tree.Update(w, id, 1)$;
  2) if $op = del$,
        invoke $tree.Update(w, id, 0)$;
```

C. Single-Round-Trip Protocol

In the UI-SE search and update protocols, the user should always upload the final encrypted paths to the cloud for nodes replacing. This reencryption will lead to three-round interactions. Fortunately, the last communication packet that contains a set of new paths can be folded into the next query. Therefore, the protocols are completed in a single-round-trip user-cloud interaction.

VI. Security Analysis

UI-SE security is quantified by the leakage function, $\mathcal{L} = \{\mathcal{L}_{UI}^{Setup}, \mathcal{L}_{UI}^{Search}, \mathcal{L}_{UI}^{Update}\}$, where each is for describing the leakage in the setup stage, the search stage, and the update stage, respectively. $\mathcal{L}_{UI}^{Setup} = \{L, N\}$, where L is the height of the OU-tree, and N is the total number of keyword-identifier pairs in the inverted index. We focus the study on $\mathcal{L}_{UI}^{Update}$ and $\mathcal{L}_{UI}^{Search}$.

A. Update-Leakage Analysis

UI-SE has two main components, the encrypted inverted index (EII) and the OU-tree. Since the EII can be viewed as a static index, no update leakage from the EII is generated. The updates on the OU-tree are always oblivious. This is because 1) any update always accesses a random leaf-to-root path, and 2) all the tree nodes are encrypted by the RCPA-secure algorithm that guarantees the encrypted contents are indistinguishable from random. Since addition and deletion use the same access algorithm, the operation reveals only whether the name is a search or an update. From the above analysis, the update leakage function is written as

$$\mathcal{L}_{UI}^{Update}(op, w, id) = \{op\},$$

where $op = add$ or del, and (w, id) is a pair for the update.

B. Search-Leakage Analysis

Given a query token $F_{\mathbb{K}}(w)$, the cloud can use it to search the EII and obtain an array of key-value tuples $\{(k_1, v_1), (k_2, v_2), \cdots, (k_{c_w}, v_{c_w})\}$. From the OU-tree, the cloud learns $\{k_1\%2^{L-1}, \cdots, k_{c_w}\%2^{L-1}\}$, but no other knowledge is gained. All the above knowledge equals $\{(k_1, v_1), (k_2, v_2), \cdots, (k_{c_w}, v_{c_w})\}$ since L is a constant. We only need to consider the leakage generated from the adaptively-secure inverted index. Since each v_i has two parts $(k_{i+1} : eid_i)$, where eid_i is the i-th encrypted file identifier, the cloud's knowledge equals $\{(k_1, eid_1), \cdots, (k_{c_w}, eid_{c_w})\}$. Assuming each keyword has been queried many times, the leakage of this approach is just the search pattern. The randomized encrypted identifiers still guarantee that no correlations of two different keyword queries are exposed. From the above analysis, we have the search leakage function

$$\mathcal{L}_{UI}^{Search}(w) = (sp(w), c_w),$$

where $sp(w)$ is the search pattern, and c_w is the size pattern, the number of identifiers of files containing w when the index is initialized.

C. Backward Privacy without Insertion Patterns

According to $\mathcal{L}_{UI}^{Update}$ and $\mathcal{L}_{UI}^{Search}$, UI-SE achieves forward and backward privacy on the BP-(I, I$^-$) level.

Theorem 5.1 UI-SE reveals no insertion patterns.

Proof: For any (w, id) pair, even it has been frequently added and deleted, the token $F_{\mathbb{K}}(w)$ always matches the fixed-length entries in the EII and the fixed-length path in the OU-tree. c_w entries in the EII and $\Theta(c_w \log N)$ tree nodes are accessed. Since some nodes in the path have been reencrypted by the randomized encryption algorithm after any update, and the historical update paths are in random distribution, the search does not reveal the historical insertion time. Thus, UI-SE is forward and backward private without the insertion pattern, which is stronger than the BP-(I, I) privacy. Since $\mathcal{L}_{UI}^{Setup}$ and $\mathcal{L}_{UI}^{Update}$ are negligible compared to the search leakage, the leakage function is written as $\mathcal{L} \approx \{\mathcal{L}_{UI}^{Search}\}$. Formally, security is defined as follows.

Theorem 5.2 (Adaptive security). Assuming F is a pseudorandom function, H_1 and H_2 are modeled as random oracles, and there exists an RCPA-secure algorithm, then UI-SE is ($sp(w)$, c_w)-secure under an adaptive adversary.

Proof: Assume there exists an adversary $\mathcal{A}$, who owns the ability to adaptively query the random oracle for any tokens according to the DSE scheme. $\mathcal{A}$ wants to break the encryption protocol through issuing search queries and update queries. Consider such a simulator $\mathcal{S}$, who is given $\mathcal{L} = \{\mathcal{L}_{UI}^{Setup}, \mathcal{L}_{UI}^{Search}, \mathcal{L}_{UI}^{Update}\}$ to adaptively simulate $\mathcal{A}$'s operations, including searches, additions, and deletions. $\mathcal{S}$ uses the following steps to simulate $\mathcal{A}$.

In the index initialization stage, the adversary $\mathcal{A}$ has an encrypted index $(map, tree)$, and $\mathcal{S}$ also sets up a simulated index $(map^*, tree^*)$ by using random values according to (L, N) leakage.

In the update stage, $\mathcal{A}$ adaptively queries the random oracle for an update token. $\mathcal{A}$ creates an update query that will access a randomly-chosen leaf-to-root path P_{up}. Since $\mathcal{S}$ knows there is an update query, $\mathcal{S}$ also creates a simulated update query that will access a randomly-chosen leaf-to-root path P_{up}^*.

In the search stage, $\mathcal{A}$ adaptively queries the random oracle for a search token. Let the token be $tk \leftarrow F_{\mathbb{K}}(w)$. If the same token has been issued j times, $\mathcal{S}$ adaptively simulates it j times since $\mathcal{S}$ has $sp(w)$. $\mathcal{S}$ creates a simulated search token tk^* using a random value.

Consider the search in the encrypted inverted index. Assume that $\mathcal{A}$ successfully obtains an array of key-value tuples $((k_1, v_1), (k_2, v_2), \cdots, (k_{c_w}, v_{c_w}))$ from map, an array of encrypted file identifiers $(eid_1, \cdots, eid_{c_w})$, and an array of encrypted variables $(var_1, \cdots, var_{c_w})$, where each $k_i = H_1(var_i), v_i = H_2(var_i) \oplus (var_{i+1}||eid_i)$, and $var_1 = tk$. $\mathcal{S}$ also randomly chooses a sequence of key-value pair $((k_1^*, v_1^*), (k_2^*, v_2^*), \cdots, (k_{c_w}^*, v_{c_w}^*))$ from map^*, randomly generates an array of encrypted file identifiers $(eid_1^*, \cdots, eid_{c_w}^*)$, and randomly generates an array of variables $(var_1^*, \cdots, var_i^*)$, where $var_1^* = tk^*$. $\mathcal{S}$ now programs the random oracles H_1 and H_2. Let each $k_i^* = H_1(var_i^*), v_i^* = H_2(var_i^*) \oplus (var_{i+1}^*||eid_i^*)$. Finally, $\mathcal{A}$ outputs a set of results $(eid_1, eid_2, \cdots, eid_{c_w})$, and $\mathcal{S}$ also outputs a set of results $(eid_1^*, eid_2^*, \cdots, eid_{c_w}^*)$.

Consider the search in the OU-tree. $\mathcal{A}$ uses $(var_1, \cdots, var_{c_w})$ to access the tree. $\mathcal{A}$ outputs a sequence of tree paths $\{P_1, \cdots, P_{c_w}\}$, where each P_j is a sequence of leaf-to-root nodes whose leaf identifier equals $var_j\%2^{L-1}$. $\mathcal{S}$ also outputs a sequence of tree paths $\{P_1^*, \cdots, P_{c_w}^*\}$, where each P_j^* is a sequence of leaf-to-root nodes whose leaf identifier equals $var_j^*\%2^{L-1}$. After the access, $\mathcal{A}$ replaces the accessed nodes with new paths according to the protocol, and $\mathcal{S}$ also replaces the accessed nodes with random values.

Due to the psedurandom function, tk and tk^* are PPT computationally indistinguishable. Due to the RCPA-secure encryption algorithm, $(map, tree)$ and $(map^*, tree^*)$, P_{up} and P_{up}^*, $\{P_1, \cdots, P_{c_w}\}$ and $\{P_1^*, \cdots, P_{c_w}^*\}$, and $\{eid_1, \cdots, eid_{c_w}\}$ and $\{eid_1^*, \cdots, eid_{c_w}^*\}$ are PPT computationally indistinguishable. These implies that $\mathcal{A}$ learns nothing except $\mathcal{L}$. Therefore, UI-SE is $\mathcal{L}$-secure under the adaptive adversary with BP-(I, I$^-$) backward privacy, where $\mathcal{L} \approx (sp(w), c_w)$.

VII. Experiments and Evaluations

A. Experimental Methodologies

We conduct the experiments on a desktop computer equipped with a Core(TM) i9-10850K CPU 3.60GHz and 64 GB DDR4 memory. All the testing cases are coded in C++ 17. AES is for symmetric encryption, and Blake2b is for pseudorandom computation. The experiments randomly generate a set of databases that reach $1E8$ pairs.

The leaf capacity R is set to a fixed small value. Since F is a pseudorandom function, all the keyword-identifier pairs are nearly uniformly distributed in the leaves. Thus, the leaf capacity R can be set to a small value to hold all the keyword-identifier pairs. In the experiments, R is less than 7, and L is set to $(L \approx (\log_2 N + 4))$, where the additional height is used for providing more space for insertions.

For simplicity, let $a_w = a(w)$ be the number of updates on w, and $c_w = c(w)$ be the number of keyword-identifier pairs for w when the index is initialized. Assume the index is fully loaded into the memory. All communication time is ignored in the experiments. All testing cases create only one thread.

B. Experimental Data

Node size. Let t_1 be the number of empty triplets of the internal nodes, t_2 be the number of dummy triplets of the internal nodes, and t_3 be the number of real triplets of the internal nodes. The vacancy rate α is evaluated by

$$\alpha = \frac{t_1 + t_2}{t_1 + t_2 + t_3}.$$

TABLE II
The vacancy rate on update-intensive cases.

	Updates	L	Z	α
1	$4,000$	13	4	86.70%
2	$40,000$	13	4	85.46.%
3	$400,000$	13	4	85.04%
4	$4,000,000$	13	4	85.48%
5	$4,000$	13	6	91.13%
6	$40,000$	13	6	90.67%
7	$400,000$	13	6	90.30%
8	$4,000,000$	13	6	90.21%

The experimental data in Table II prove that the node size $Z = 4$ is enough for an OU-tree to provide much free space. The experiments are handled by updating the keyword-identifier pairs repeatedly after index initialization. $Z = 4$ denotes each internal node has at most four triplets. Even if the tree has been updated $4,000,000$ times, there is still 85.48% free space if $Z = 4$. This is because the shuffle algorithm always tries to remove the obsolete versions as much as possible. After the removals, the tree still has enough room to hold the new updates. The newly inserted pairs will be moved into the leaves by the shuffle algorithm when the random write accesses the leaves.

Stash size. The experiment data in Table III prove that UI-SE achieves near-zero client-side storage. The maximum stash size is the maximum of the stash used during a sequence of updates. The average stash size is near-zero most of the time, even if the pairs have been highly intensively updated. Note that if $Z = 1$, in the experiments, the maximum stash size can reach 400 or higher. Therefore, $Z = 4$ is a suitable parameter for highly intensive updates.

TABLE III
The stash size.

	Updates	L	Z	Average stash size	Maximum stash size
1	$400,000$	13	4	1	15
2	$4,000,000$	13	4	1	16
3	$400,000$	13	6	1	13
4	$4,000,000$	13	6	1	13
5	$400,000$	20	4	2	16
6	$4,000,000$	20	4	2	19

Index-construction time. The experimental data in Table IV prove that the OU-tree-based index can be efficiently set up. Since the index achieves the optimal index size $O(N)$, the index-construction speed reaches 199, 848 pairs/s when $L = 28$ and $Z = 4$ if $N = 2E7$. Setting up the whole index takes only 100 seconds, assuming the file-preprocess time is not considered.

TABLE IV
Index-initialization experiments.

	N	L	Z	Speed (pair/s)	Insertion time (s)
1	$2E5$	21	4	263,289	0.8
2	$2E6$	24	4	241,948	8.3
3	$2E7$	28	4	199,848	100.1
4	$2E5$	21	6	237,586	0.8
5	$2E6$	24	6	218,941	9.1
6	$2E7$	28	6	183,157	109.2

Update time/bandwidth. The experimental data in Figure 7 show that an update takes only several milliseconds even if the number of keyword-identifier pairs reaches 50 million. The line of 'Z=4' in the figure is more efficient than that of 'Z=6'. This is because a small tree node means less work for encryption.

The experimental data in Figure 8 show that the update bandwidth is in several KB even if the number of keyword-identifier pairs reaches 50 million. Each triplet occupies 48 bytes after encryption in the experiment, and each tree node contains $Z = 4$ triplets. If the height of the tree is 30, the communication bandwidth equals approximately 6 KB.

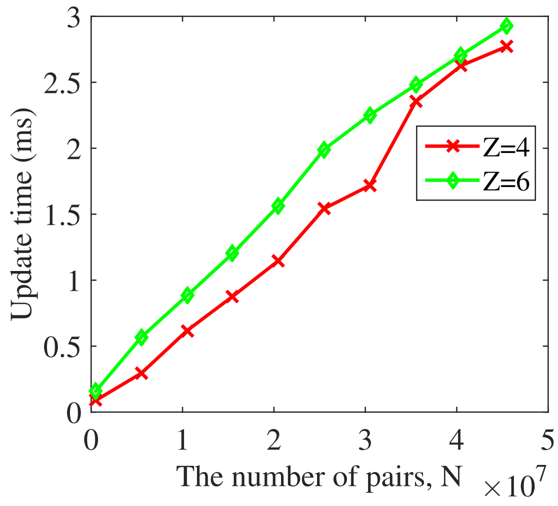

Fig. 7. The relationship between update time and database size.

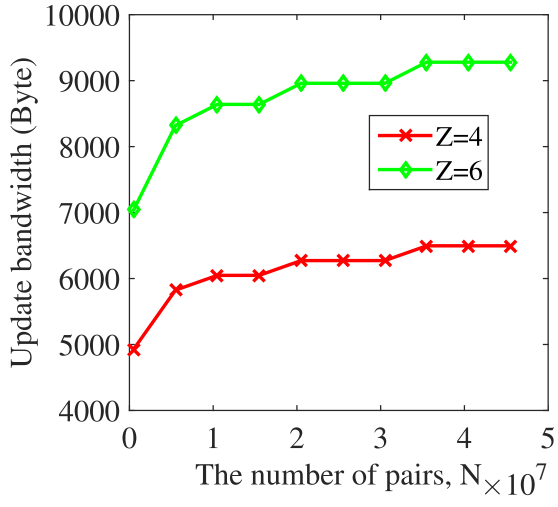

Fig. 8. The relationship between update bandwidth and database size.

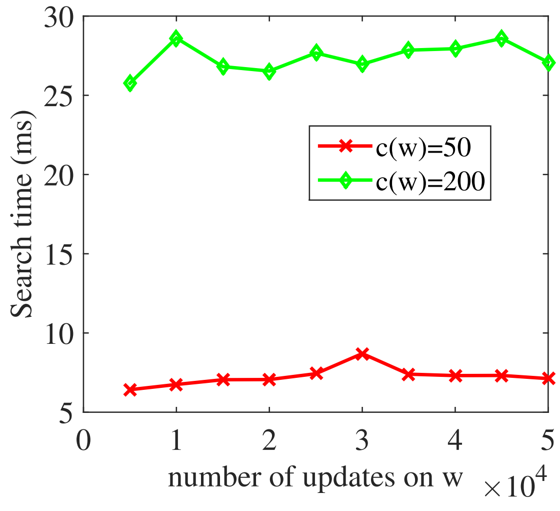

Fig. 9. The relationship between search time and the number of updates on the keyword.

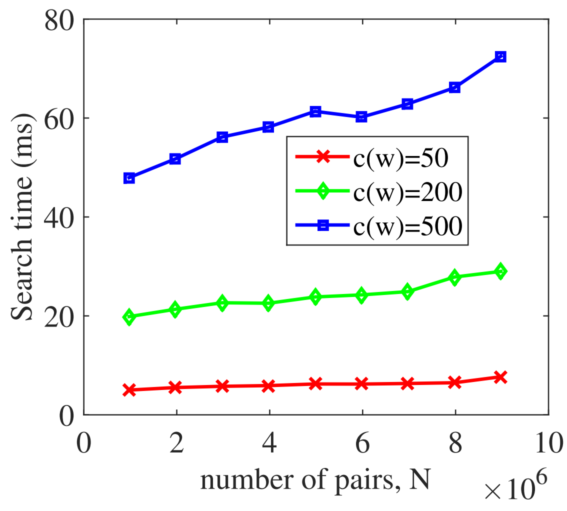

Fig. 10. The relationship between search time and database size.

Search time. The experimental data in Figure 9 prove that search time has no correlations with $a(w)$, the number of updates on keyword w. Even if the keyword has been updated 50,000 times, the search time remains unchanged, where $c(w) = 50$ means that there are 50 keyword-identifier pairs for w when the index is initialized, and so on.

The experimental data in Figure 10 prove that UI-SE achieves worst-case sublinear search complexity on scalable datasets. The search time has correlations with only two factors, $c(w)$ and $\log N$. These experiments are conducted on ten different indexes. The time cost grows slowly with the number of keyword-identifier pairs increasing. This is reasonable since the larger the database, the higher the OU-tree.

C. *Compared with Other Backward-Private Schemes*

We compare UI-SE with other typical small-client backward-private DSE schemes: SD$_d$ [10], CLOSE-FB [5], and Orion [7].

SD$_d$ and CLOSE-FB leak update patterns in the search queries. A search reveals their historical update timestamps about this keyword. That is, they achieve only BP-(I, II) privacy. SD$_d$ and CLOSE-FB are not suitable for highly intensive updates since their search time is linear in $a(w)$.

The oblivious map (OMAP) based DSE schemes, Orion and SD$_d$, suffer from the inherent defects of not only $O(\log N)$ client-server interactions, but also $O(\log^2 N)$ communication overhead per update. These defects limit their potential usages on cloud-hosted network applications.

VIII. Conclusions and Future Work

In this paper, we gave UI-SE, a new DSE scheme that supports highly intensive updates. The construction involves a new OU-tree structure, whose advantage is that the tree supports insertion-pattern-hiding backward-private updates. We focused the study on updating existing keyword-identifier pairs in the index. To update non-existing pairs, one can combine the traditional data structures, such as the fish-bone structure [5], with the OU-tree to support various kinds of data queries.

Acknowledgments

This work is supported by the Research Foundation of Education Bureau of Hunan Province, China, under Grant 19C0085, and the National Natural Science Foundation of China (61772454, 61811530332, 61811540410).

References

[1] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *the 13th ACM conference on Computer and Communications Security (CCS)*, pages 79–88. ACM, 2006.

[2] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. Dynamic searchable symmetric encryption. In *the 2012 ACM conference on Computer and Communications Security (CCS)*, pages 965–976. ACM, 2012.

[3] Raphaël Bost, Brice Minaud, and Olga 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)*, pages 1465–1482. ACM, 2017.

[4] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In *USENIX Security Symposium*, pages 707–720, 2016.

[5] Kun He, Jing Chen, Qinxi Zhou, Ruiying Du, and Yang Xiang. Secure dynamic searchable symmetric encryption with constant client storage cost. *IEEE Transactions on Information Forensics and Security*, 16:1538–1549, 2020.

[6] Jin Li, Yanyu Huang, Yu Wei, Siyi Lv, Zheli Liu, Changyu Dong, and Wenjing Lou. Searchable symmetric encryption with forward search privacy. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2019.

[7] Javad Ghareh Chamani, Dimitrios Papadopoulos, Charalampos Papamanthou, and Rasool 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)*, pages 1038–1055. ACM, 2018.

[8] Xiao Shaun Wang, Kartik Nayak, Chang Liu, TH Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious data structures. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 215–226. ACM, 2014.

[9] Sanjam Garg, Payman Mohassel, and Charalampos Papamanthou. Tworam: efficient oblivious ram in two rounds with applications to searchable encryption. In *Annual International Cryptology Conference (CRYPTO)*, pages 563–592. Springer, 2016.

[10] Ioannis Demertzis, Javad Ghareh Chamani, Dimitrios Papadopoulos, and Charalampos Papamanthou. Dynamic searchable encryption with small client storage. *IACR Cryptol. ePrint Arch.*, 2019:1227, 2019.

[11] Dawn Xiaoding Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *Security and Privacy (S&P)*, pages 44–55. IEEE, 2000.

[12] Zhiqiang Wu and Kenli Li. Vbtree: forward secure conjunctive queries over encrypted data for cloud computing. *The VLDB Journal*, 28(1):25–46, 2019.

[13] Rui Li and Alex X Liu. Adaptively secure conjunctive query processing over encrypted data for cloud computing. In *33rd International Conference on Data Engineering (ICDE)*, pages 697–708. IEEE, 2017.

[14] Rui Li, Alex X. Liu, Ann L. Wang, Bezawada Bruhadeshwar, Rui Li, Alex X. Liu, Ann L. Wang, and Bezawada Bruhadeshwar. Fast and scalable range query processing with strong privacy protection for cloud computing. *IEEE/ACM Transactions on Networking (TON)*, 24(4):2305–2318, 2016.

[15] Zhiqiang Wu, Kenli Li, Keqin Li, and Jin Wang. Fast boolean queries with minimized leakage for encrypted databases in cloud computing. *IEEE Access*, 7:49418–49431, 2019.

[16] Seny Kamara and Tarik Moataz. Boolean searchable symmetric encryption with worst-case sub-linear complexity. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 94–124. Springer, 2017.

[17] Zhangjie Fu, Xinle Wu, Chaowen Guan, Xingming Sun, and Kui Ren. Toward efficient multi-keyword fuzzy search over encrypted outsourced data with accuracy improvement. *IEEE Transactions on Information Forensics and Security (TIFS)*, 11(12):2706–2716, 2016.

[18] Wanyu Lin, Helei Cui, Baochun Li, and Cong Wang. Privacy-preserving similarity search with efficient updates in distributed key-value stores. *IEEE Transactions on Parallel and Distributed Systems*, 32(5):1072–1084, 2020.

[19] Russell WF Lai and Sherman SM Chow. Forward-secure searchable encryption on labeled bipartite graphs. In *International Conference on Applied Cryptography and Network Security*, pages 478–497. Springer, 2017.

[20] Raphael Bost, Pierre-Alain Fouque, and David Pointcheval. Verifiable dynamic symmetric searchable encryption: Optimality and forward security. *IACR Cryptology ePrint Archive*, 2016:62, 2016.

[21] Xiangfu Song, Changyu Dong, Dandan Yuan, Qiuliang Xu, and Minghao Zhao. Forward private searchable symmetric encryption with optimized i/o efficiency. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2018.

[22] Kee Sung Kim, Minkyu Kim, Dongsoo Lee, Je Hong Park, and Woo-Hwan Kim. Forward secure dynamic searchable symmetric encryption with efficient updates. In *the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1449–1463. ACM, 2017.

[23] Ghous Amjad, Seny Kamara, and Tarik Moataz. Forward and backward private searchable encryption with sgx. In *Proceedings of the 12th European Workshop on Systems Security*, page 4. ACM, 2019.

[24] Raphael Bost. $\sum o\varphi o\varsigma$: Forward secure searchable encryption. In *the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1143–1154. ACM, 2016.

[25] Cong Zuo, Shi-Feng Sun, Joseph K Liu, Jun Shao, and Josef Pieprzyk. Dynamic searchable symmetric encryption with forward and stronger backward privacy. In *European Symposium on Research in Computer Security (ESORICS)*, pages 283–303. Springer, 2019.

[26] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. A retrospective on Path ORAM. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2019.

[27] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In *the 2013 ACM SIGSAC conference on Computer and Communications Security (CCS)*, pages 299–310. ACM, 2013.

[28] Thang Hoang, Ceyhun D Ozkaptan, Attila A Yavuz, Jorge Guajardo, and Tam 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)*, pages 491–505. ACM, 2017.

[29] Hao Chen, Ilaria Chillotti, and Ling Ren. Onion ring ORAM: Efficient constant bandwidth oblivious RAM from (leveled) TFHE. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 345–360. ACM, 2019.

[30] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 784–796, 2012.

[31] Ebrahim M Songhori, Siam U Hussain, Ahmad-Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar. Tinygarble: Highly compressed and scalable sequential garbled circuits. In *2015 IEEE Symposium on Security and Privacy*, pages 411–428. IEEE, 2015.

[32] Zheli Liu, Yanyu Huang, Xiangfu Song, Bo Li, Jin Li, Yali Yuan, and Changyu Dong. Eurus: Towards an efficient searchable symmetric encryption with size pattern protection. *IEEE Transactions on Dependable and Secure Computing*, 2020.

[33] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In *the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 668–679. ACM, 2015.

[34] Vincent Bindschaedler, Paul Grubbs, David Cash, Thomas Ristenpart, and Vitaly Shmatikov. The tao of inference in privacy-protected databases. *Proceedings of the VLDB Endowment*, 11(11):1715–1728, 2018.

[35] Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. Improved reconstruction attacks on encrypted data using range query leakage. In *2018 IEEE Symposium on Security and Privacy (S&P)*, pages 19–36, 2017.

[36] Marten Van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. In *Annual Interna-*

tional Conference on the Theory and Applications of Cryptographic Techniques, pages 24–43. Springer, 2010.

[37] Florian Kerschbaum and Axel Schroepfer. Optimal average-complexity ideal-security order-preserving encryption. In *the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 275–286. ACM, 2014.

[38] Samee Zahur, Xiao Wang, Mariana Raykova, Adrià Gascón, Jack Doerner, David Evans, and Jonathan Katz. Revisiting square-root ORAM: efficient random access in multi-party computation. In *2016 IEEE Symposium on Security and Privacy (S&P)*, pages 218–234. IEEE, 2016.

[39] Rui Li, Alex X. Liu, Ann L Wang, and Bezawada Bruhadeshwar. Fast range query processing with strong privacy protection for cloud computing. In *Very Large Data Bases (VLDB)*, pages 1953–1964, 2014.

Zhiqiang Wu received the B.S. degree in computer application technology from the Central South University, China, in 2003, and received the Ph.D. degree in computer science from the Hunan University, China, in 2019. He currently works with the Changsha University of Science and Technology. His research interests include network security, data encryption, embedded systems, software architecture, high-performance computing, and big data computing. He has authored several papers in international journals, such as the VLDB Journal.

Jin Wang received the B.S. and M.S. degrees from the Nanjing University of Posts and Telecommunications, China, in 2002 and 2005, respectively, and the Ph.D. degree from Kyung Hee University, South Korea, in 2010. He is currently a Professor with the Changsha University of Science and Technology. He has published more than 300 international journal and conference papers. His research interests mainly include wireless sensor networks, network performance analysis, and optimization. He is a Senior Member of the IEEE and a member of ACM.

Keqin Li Dr. is a SUNY Distinguished Professor of computer science with the State University of New York. He is also a Distinguished Professor at Hunan University, China. His current research interests include cloud computing, fog computing and mobile edge computing, energy-efficient computing and communication, embedded systems and cyber-physical systems, heterogeneous computing systems, big data computing, high-performance computing, CPU-GPU hybrid and cooperative computing, computer architectures and systems, computer networking, machine learning, intelligent and soft computing. He has published over 630 journal articles, book chapters, and refereed conference papers, and has received several best paper awards. He currently serves or has served on the editorial boards of the IEEE Transactions on Parallel and Distributed Systems, the IEEE Transactions on Computers, the IEEE Transactions on Cloud Computing, the IEEE Transactions on Services Computing, and the IEEE Transactions on Sustainable Computing. He is an IEEE Fellow.