Parallel and Dynamic Structured Encryption

Russell W.F. Lai and Sherman S.M. Chow(✉)

Department of Information Engineering,
The Chinese University of Hong Kong, Sha Tin, N.T., Hong Kong
{russell,sherman}@ie.cuhk.edu.hk

Abstract. We design a searchable symmetric encryption scheme for structured data which supports dynamic updates and parallel computation. The abstract data type supported by our scheme not only can represent the usual keyword-file search but also other data type such as graph structure. Unlike previous parallelizable schemes, search complexity of our scheme is optimal, namely, linear in the number of matches divided by the number of processors. Moreover, previous parallel and dynamic schemes require an interactive update protocol to minimize the leakage caused by the updates. It is thus a major technical challenge to mandate non-interactive updates. While achieving multiple requirements simultaneously, our scheme leverages a simple tree structure. Our scheme is secure against adaptive chosen query attack. We also evaluate the efficiency of our scheme with synthetic data (of higher edge density) and real-life data for the application of online social network where connections among users are represented by graphs.

Keywords: Searchable symmetric encryption · Structured encryption · Non-interactive · Dynamic · Parallel · Graph encryption

1 Introduction

In searchable symmetric encryption (SSE), the key used for encryption has an additional capability of generating a *search token*, with which the encrypted content can be queried efficiently without leaking the plaintext data. A common application of SSE is to outsource the storage of a set of documents to an untrusted server. The ability to search is especially critical to mobile devices where transmission speed and storage space are usually limited.

Structured Encryption. Since the seminal work of Song *et al.* [11], many SSE schemes focus on keyword search over files. Later schemes extended the query type to more complex keyword searches, such as range search [13], similarity

Sherman S.M. Chow is supported in part by General Research Fund Grant No. 14201914 and the Early Career Award from Research Grants Council, Hong Kong; and Huawei Innovation Research Program (HIRP) 2015 (Proj. No. YB2015110147). We thank Chen Change Loy and Multimedia Lab, Dept. of Information Engineering, CUHK for their computational resources we used for some of our initial experiments.

R. Deng et al. (Eds.): SecureComm 2016, LNICST 198, pp. 219–238, 2017.
DOI: 10.1007/978-3-319-59608-2_12

search [14], *etc.* Chase and Kamara [1] generalize SSE to *structured encryption* for supporting queries over arbitrary structured data.

Leakage. Ideally, an SSE scheme should satisfy two security requirements: (1) the encrypted database does not reveal any information about the plaintext, and (2) the tokens for adaptively issued queries and updates do not reveal any further information beyond the query results. Typically, SSE schemes often reveal the access and search pattern [2,3]. Yet they are *non-interactive*, which means that the client only needs to delegate the search token and needs not to provide further help for any subsequent searches. There are *interactive* solutions like oblivious RAM [4] which can hide the access pattern, yet at the cost of efficiency.

Beyond access and search patterns, other information about the plaintext could be leaked to the server. This information can be precisely defined by a set of leakage functions [1,2,7]. Informally, we say that an SSE scheme is secure against adaptive chosen query attack (CQA2), a generalization of adaptive chosen keyword attack (CKA2) [2], if any adversary issuing a polynomial number of queries adaptively cannot distinguish a real SSE scheme from one simulated with the knowledge of the leakages. Note that the adversaries for different schemes (of different efficiency) are often given different sets of leakage functions.

Existing Parallel and Dynamic SSE. SSE schemes proposed by Kamara *et al.* [7], Kamara and Papamanthou [6], and Hahn and Kerschbaum [5] (denoted by KPR, KP, and HK respectively) support *dynamic* updates of files, *i.e.*, files can be added or removed. This can be done via the help of an *update token*. A recent SSE scheme proposed by Stefanov *et al.* [12] (denoted by SPS) can update individual (keyword, file) pairs dynamically, but is unable to directly remove a file, *i.e.*, the client needs to manually remove all the (keyword, file) pairs for the unwanted file.

Supporting update poses more challenges in preventing leakage. For supporting efficient dynamic updates, early work (*e.g.*, KPR [7]) made compromise in allowing more leakage when compared with some prior static SSE schemes. Moreover, KPR uses linked list as its internal data structure which is inherently sequential, making the scheme not parallelizable and less practical to be used in parallel computing architecture.

Recent parallel and dynamic schemes (KP [6] and SPS [12]) made the tradeoff by requiring *interaction* between the data owner and the server in *every updates* to minimize leakage. These schemes adopt different design principles in addressing the same problem. From a high-level point of view, KP employs a simple and direct approach which passes the data structure maintenance problem incurred by the update back to the owner. On the other hand, SPS relies on an interactive cryptographic protocol known as oblivious sorting. One can view these two schemes as adopting approaches at two ends of a spectrum. The former method requires the data owner to locally decrypt the relevant part of the data structure, and upload again an encryption of them after maintenance for keeping the parallel efficiency. The use of oblivious sorting requires *local storage* at the client side (apart from the private key) and makes the resulting scheme relatively heavyweight. In short, both approaches require quite a large amount of communication and work at the client side. These schemes also store redundant information which required to be traversed during a search, thus the full power of parallel computation diminishes. In more details, KP stores the actual data only in the leaf nodes of a tree and SPS firstly creates a "Delete" node during deletion rather than actually removing the data.

HK uses a simplistic approach for handling data dynamic by exploiting the leakage incurred from the first search on any keyword. While the majority of the existing SSE schemes required a pre-computed inverted index, HK simply stores the encrypted files as sequences of encrypted keywords in the database, and creates a simple inverted index on the fly using the leaked access pattern. Therefore, adding and deleting files in HK are as easy as adding or removing the corresponding sequence of encrypted keywords as a whole, and updating the rather small inverted index. Subsequent search can be easily parallelized as the inverted index is stored in plaintext. However, as the search history becomes longer, the inverted index becomes larger which slows down the addition and deletion algorithms.

To summarize, it is fair to say that designing SSE with a desirable trade-off between functionality, security, and efficiency is a challenging problem.

Our Contribution. We propose a searchable symmetric encryption scheme RBT which supports dynamic updates and parallel computation. In summary, our scheme makes technical contributions in two dimensions.

First, we extend structured encryption for dynamic abstract data type which allows updates to both the data space and the query space. Specifically, RBT allows updates to individual (query, data) pairs. This requires a more fine-grained access control over the encrypted database. Under this abstraction, RBT allows deletion of data which automatically deletes all (query, data) pairs related to the piece of data in question. To the best of our knowledge, our scheme is the first to support both types of updates. In addition to returning all data related to a given query, our scheme also supports meta-query to check if a (query, data) pair exists in the database, *i.e.*, that the query is related to the data. This contribution will be presented in Sect. 2. We will illustrate the applicability of this abstract data type, particularly to representing connections in online social network, in Sect. 2.2.

Second, in the premise of parallel SSE, we aim at the optimal search complexity linear in the number of matches divided by the number of processors, simultaneously ensuring that searches only leak search and access patterns, while minimizing the leakages during updates. This will be presented in Sect. 4. Despite making the above improvements, our scheme leverages a simple randomized binary tree (hence the name RBT) to achieve non-interactive queries and updates.

Finally, we show that our scheme is secure against adaptive chosen query attack, and demonstrate its performance in Sect. 5 using both synthetic data for general scenarios and real-life data for online social networks.

Performance Comparison. We compare our scheme with KPR, KP, and SPS and HK in Table 1. Yet, we remark that it is a simplified discussion due to the differences in leakages (of different data-structures), the interaction

Table 1. The search complexities of KPR, KP, SPS, HK, and RBT (m, N, and p denote the number of matches, number of all files/data, and number of processors resp.)

Scheme	Search complexity
KPR	$O(m)$
KP	$O(m/p) \log N$ ($\because$ storing data only in leaf nodes)
SPS	$O(m/p) \log^3 N$ ($\because$ rebuild mechanism)
HK	$O(N/p)$ (first time, $\because$ no pre-built inverted index)
	$O(m/p)$ (subsequent search)
RBT	$O(m/p)$

requirements, *etc.* In particular, during updates, KPR leaks local information; RBT leaks the affected sub-trees' traversal information μ_{t} (Table 2); KP and SPS leak nothing by interaction (throwing back the update-task to the client and performing interactive oblivious-updates respectively) as we explained.

2 Our Dynamic Abstract Data Type

2.1 Definition

We extend the definition of static data type by Chase and Kamara [1] to dynamic data type. A dynamic abstract data type $\mathcal{T}$ is defined by a data space $\mathcal{D}$ with a query operation $\mathsf{Query} : \mathcal{D} \times \mathcal{Q} \to \mathcal{R}$ and an update operation $\mathsf{Update} : \mathcal{D} \times \mathcal{U} \to \mathcal{D}$, where $\mathcal{Q}$ is the query space, $\mathcal{R}$ is the response space, and $\mathcal{U}$ is the update space.

As in most of the other SSE schemes, the responses to the queries are prepared during encryption. Without loss of generality, we let a data structure δ of type $\mathcal{T}$ and size parameter (M, N) to have the following structure:

– Data set: $\delta \subset \delta^* = \{(q_i, r_j)\}_{i=1,j=1}^{M,N} \in \mathcal{D}$
– Query space: $\mathcal{Q}(\delta) = \{q : \exists r \text{ s.t. } (q, r) \in \delta\}$
– Response space: $\mathcal{R}(\delta) = \{r : \exists q \text{ s.t. } (q, r) \in \delta\}$
– Update space: $\mathcal{U}(\delta) = \{(\text{“Add”}, d) : d \in \delta^* \setminus \delta\} \cup \{(\text{“Del”}, d) : d \in \delta\}$

where q_i is a query, r_j is a piece of data corresponding to a query, and δ^* is considered to be the largest possible collection of data. The operations Query and Update are defined in the natural way. This representation expresses each of the possible query-response pairs as a data item.

It can be useful to check if a certain pair of query and response exists. We therefore build extra “meta-queries” based on the normal query-response pairs. Concretely, we extend the query space to $\mathcal{Q}' = \mathcal{Q} \cup \delta$ and the response space to $\mathcal{R}' = \mathcal{R} \cup \{\mathsf{true}, \mathsf{false}\}$. The query operation is also extended so that, given a “meta-query” $d = (q, r)$, it checks if (q, r) is in the data set. If so, it returns true. Otherwise, it returns false. The update operation is extended in the natural way.

2.2 Instantiating Our Abstract Data Type

To illustrate the generality and flexibility of our abstract data type, we show how it covers (the common) searches for keyword in files, and other common data types considered in existing structured encryption of Chase and Kamara [1].

For keyword search, each keyword is encoded as a query, all the files containing a certain keyword are the corresponding responses. Via the meta-query, our data type further supports the query for checking if a certain keyword exists in a particular file, which minimizes the unnecessary traversal (and leakage) of other files containing the same keyword.

For lookup queries on matrix-structured data (e.g., pixel-based images) [1], we just encode the matrix data (e.g., the colors in different models like RGB and CMYK) as the responses. There can be various instantiations according to the specific needs of the application, e.g., one may assign (the index of) a row as the query and all the responses as the entries of that row, or one may assign a multi-dimension index (e.g., (row, column) pair in a 2D matrix) as the query, and our list of responses allow storing more than one data item in a single (indexed) entry. Looking ahead, with the dual structure storing both (query, response) and (response, query) pairs, our schemes can be extended to support transpose-related operations on matrices natively.

Finally, for graph, one natural representation is to assign nodes with outgoing edges as queries, and those with incoming edges as responses. The existing structured encryption [1] scheme supports neighbor queries and adjacency queries. Neighbor queries return all the nodes adjacent to a given node i. It is apparent that i will be the query and the adjacent nodes are all stored as its response. For queries to check if two nodes are adjacent, it can be easily supported by our meta-query. As mentioned in the original application [1], this allows us to support controlled disclosure of friendship graphs of a social network, for example.

3 Cryptography Background

3.1 Basic Notations

Let λ be the security parameter. All sets and other parameters depend on λ implicitly. $\{0,1\}^n$ denotes the set of all binary strings of length n. $\{0,1\}^*$ denotes the set of all finite length binary strings. $\mathbf{0}$ denotes the λ-bit string with all zeros. $\mathbf{0}_k$ denotes k consecutive zero strings $\mathbf{0}$. ϕ denotes the empty set. If X is a set, $x \leftarrow X$ denotes the sampling of an element x uniformly from X. If A is an algorithm, $x \leftarrow A$ means that x is the output of A. "$\oplus$" denotes the bit-wise exclusive OR (XOR) operation. If $x, y \in \{0,1\}^n$, $|y|$ denotes the length of y, *i.e.*, n; and $x \oplus= y$ denotes $x = x \oplus y$, *i.e.*, assigning $x \oplus y$ as the new value of variable x. ";" denotes string concatenation.

3.2 Pseudorandom Functions and Symmetric-Key Encryption

Pseudorandom functions (PRFs), informally, is a class of polynomial-time computable function family such that no polynomial-time adversary can distinguish between a randomly chosen function among this family and a truly random function (whose outputs are sampled uniformly and independently at random), with a significant advantage relative to the security parameter. Each PRF takes a secret key and an input. The secret key serves as an index to determine which function in the family to use.

To build a symmetric-key encryption scheme with computational security, one can use a PRF to output the mask to be XOR-ed with the message. Note that the input of the PRF should be unique to ensure security.

3.3 Dynamic Symmetric Structured Encryption

We combine and simplify existing definitions of dynamic SSE and (static) structured encryption to dynamic structured encryption for our abstract data type defined in Sect. 2. The standard security notion of SSE designed for keyword search over files is the notion of security against adaptive chosen keyword attack (CKA2). Below we generalize it to the notion of security against adaptive chosen query attack (CQA2) for structured encryption. For modeling the security of our dynamic structured encryption, we also extend dynamic CKA2 and (static) CQA2 security [1,7] to dynamic CQA2.

Definition 1. *Let $\mathcal{T}$ be a dynamic abstract data type with query operation* $\mathsf{Query} : \mathcal{D} \times \mathcal{Q} \rightarrow \mathcal{R}$ *and update operation* $\mathsf{Update} : \mathcal{D} \times \mathcal{U} \rightarrow \mathcal{D}$. *A dynamic symmetric-key structured encryption scheme for $\mathcal{T}$ is a tuple of six probabilistic polynomial-time algorithms* $\mathsf{DSSE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{QryTkn}, \mathsf{Qry}, \mathsf{UdtTkn}, \mathsf{Udt})$*:*

- $K \leftarrow \mathsf{Gen}(1^\lambda)$*: The key generation algorithm inputs a security parameter λ and outputs a secret key K.*
- $\gamma \leftarrow \mathsf{Enc}(K, \delta)$*: The encryption algorithm inputs a secret key K and a data structure δ of type $\mathcal{T}$. It outputs an encrypted data structure γ.*
- $\tau_q \leftarrow \mathsf{QryTkn}(K, q)$*: The query token generation algorithm inputs a secret key K and a query $q \in \mathcal{Q}$. It outputs a query token τ_q.*
- $\mathcal{R} \leftarrow \mathsf{Qry}(\tau_q, \gamma)$*: The query algorithm inputs a query token τ_q and an encrypted data structure γ. It outputs a sequence of identifiers $\mathcal{R}$.*
- $\tau_u \leftarrow \mathsf{UdtTkn}(K, u)$*: The update token generation algorithm inputs a secret key K and an update $u \in \mathcal{U}$. It outputs an update token τ_u.*
- $\gamma' \leftarrow \mathsf{Udt}(\tau_u, \gamma)$*: The update algorithm inputs an update token τ_u and an encrypted data structure γ. It outputs a new encrypted data structure γ'.*

We say that DSSE is correct if for all $\lambda \in \mathbb{N}$, for all K output by $\mathsf{Gen}(1^\lambda)$, for all $\delta \in \mathcal{D}$, for all γ output by $\mathsf{Enc}(K, \delta)$, for all sequences of queries and updates, the queries always return the correct sequences of identifiers of the responses from δ matching to the queries.

Definition 2 *(Dynamic CQA2-security)***.** *Let DSSE be a structured encryption scheme as defined in Definition 1. Consider two probabilistic experiments, where $\mathcal{A}$ is a stateful adversary, $\mathcal{S}$ is a stateful simulator, and $\mathcal{L}_e$, $\mathcal{L}_q$, $\mathcal{L}_u$ are stateful leakage algorithms:*

– $\mathbf{Real}_{\mathcal{A}}(1^\lambda)$*: the challenger runs* DSSE *with the input data structure* δ *specified by* $\mathcal{A}$*.* $\mathcal{A}$ *returns a bit* b *that is output by the experiment.*
– $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(1^\lambda)$*:* $\mathcal{A}$ *outputs* δ*. Given* $\mathcal{L}_e(\delta)$*,* $\mathcal{S}$ *generates and sends* γ *to* $\mathcal{A}$*.* $\mathcal{A}$ *makes a polynomial number of adaptive updates* u *and queries* q*. For queries,* $\mathcal{S}$ *is given* $\mathcal{L}_q(\delta, q)$*. It returns a query token* τ_q *and a response* R*. For updates,* $\mathcal{S}$ *is given* $\mathcal{L}_u(\delta, u)$*. It returns an update token* τ_u *and an encrypted data structure* γ*. Finally,* $\mathcal{A}$ *returns a bit* b *that is output by the experiment.*

We say that DSSE *is (*$\mathcal{L}_e$*,* $\mathcal{L}_q$*,* $\mathcal{L}_u$*)-secure against adaptive dynamic chosen-query attacks if for all* PPT *adversaries* $\mathcal{A}$*, there exists a* PPT *simulator* $\mathcal{S}$ *such that*

$$|\Pr[\mathbf{Real}_{\mathcal{A}}(1^\lambda) = 1] - \Pr[\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(1^\lambda) = 1]| \leq \mathsf{negl}(\lambda).$$

4 DSSE from Random Binary Tree

Our goal is to construct a dynamic SSE scheme for structured data, such that: (1) the computation complexity of the server during queries is optimal up to a constant time overhead, and (2) updates are non-interactive. Our solution is to represent the response spaces using random binary search trees. We use the concept of normal and dual nodes to support updates like KPR [7]. For any *data* $(q, r) \in \delta$, there are a normal node and a dual node storing (q, r) which is indexed by q and r respectively.

4.1 Intuition

Take keyword search over files as an example. All keyword-file pairs are prepared; and an index is built where the pairs with the same keyword are grouped into sets. Searching for a keyword (or making a query q) is then equivalent to traversing through a set (of responses $\{r : (q, r) \in \delta\}$). Yet, the server can only traverse the set upon receipt of the corresponding token; otherwise, it can identify all (encrypted) responses to a specific (unknown) query by traversing a set.

To delete a file, the server needs to retrieve all the keywords associated with it. Hence, one can consider it as "file search over keywords" instead of keyword search over files. This explains the role played by the set of dual nodes.

The simplest method to represent either kind of set is to use a linked list, as adopted in, for example, KPR. Yet traversing a linked list is inherently sequential. Another way is to use binary trees (*e.g.*, KP). While traversing a binary tree can be parallelized, updating a binary tree requires balancing or the tree will eventually degenerate to a linked list. However, balancing a tree often requires finding a suitable "replacement" node which can be at a branch "faraway" from the position where the modification was originally made. Reaching this node requires traversal and hence the client needs to leak sufficient secret to the server. To avoid balancing the tree explicitly, we use binary search trees with random addresses as their search keys [10].

4.2 High-Level Description

We first describe our scheme RBT in high-level. This part emphasizes on the encryption and decryption part, in particular, how to use different kinds of keys in the tokens (listed in Table 3) to retrieve the information stored in each cell (listed in Table 2).

(a) Setup: RBT consists of dictionaries I and A, where I is an index pointing to some cell of A, and the cells of A are connected in random binary trees. For each data $(q;r) \in \delta$, query $q \in \mathcal{Q}(\delta)$, and response $r \in \mathcal{R}(\delta)$, a *normal node* and a *dual node* are created and stored at *random* addresses in A. Each node stores multiple types of information labeled as μ_{s}, μ_{t}, μ_{d}, and μ_{a} as explained in Table 2. This information is masked by XOR-ing with a pseudo-random function (PRF) output computed from a key and the randomness stored in μ_{a} of the node. The keys for masking each type of information are listed in Table 3.

The dictionary I maps an index to a masked address of A, where the index and the mask are computed by applying PRFs to the corresponding data, query, or response. The *normal nodes* in A correspond to the data $(q, \cdot)$. Data corresponding to the same q are connected in a random binary search tree using random addresses as their *search keys*. Similarly, the *dual nodes* correspond to the data $(\cdot, r)$ and response r are connected in a random binary search tree. Figure 1 shows a toy-example of an encrypted database. Since our binary search trees use random addresses as their search keys, the trees are roughly balanced even after a sequence of insertion and deletion [10], hence expect no balancing.

Table 2. The information stored in an array cell of RBT, with subscript in boldface in the description: μ_{t} of a node stores the traversal keys of its children, which thus grants the access to all μ_{t} down its sub-tree

Info.	Description
μ_{s}	The response r to be returned upon search query corresponding to the data $(q;r)$
μ_{t}	The addresses of the parent and children nodes, and the $\mathsf{traversal}$ keys of the children nodes used for traversal during queries *and* updates
μ_{d}	The address and $\mathsf{traversal}$ key of the **d**ual node used for delete updates *only*
μ_{a}	The randomness used (in PRF to derive the key) for masking the above

Table 3. The keys required for masking the information stored in an array cell: S, T_b and D_b are PRFs where b is the type (0:normal; or 1:dual) of the node

Info	Key
μ_{s} of all $(q;\cdot)$	The search key $S(q)$
μ_{t} of $(q;r)$	The $\mathsf{traversal}$ key $T_b(q;r)$
μ_{d} of $(q;r)$	The dual key $D_0(\hat{q})$ or $D_1(\hat{r})$

As in KPR [7], one reason for storing a dual structure is to support the deletion of queries and responses. For example, to delete a response r', all nodes corresponding to r', namely $\{(q, r') \in \delta\}$, must also be removed from the database. The dual structure provides a mechanism for updating each (q, r) which belongs to different trees.

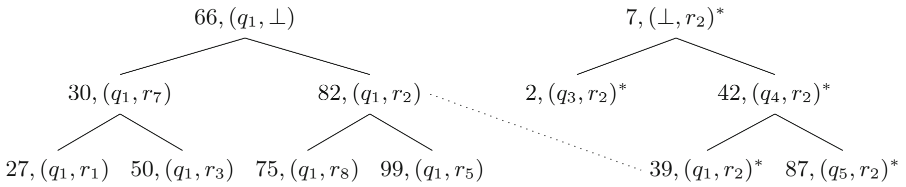

Fig. 1. Setup: Tree for q_1 and dual tree for r_2; Searching q_1 returns $r_1, r_7, r_3, r_8, r_2, r_5$ (in-order traversal based on the randomly assigned addresses $27, 30, 50, 66, 75, 82, 99$)

(b) Queries: μ_{t} of a node is masked using a traversal key stored in its parent node. So, to query q, the client computes and sends the following to the server: the index (in I), the index mask (to unmask the entry in I), the search key (to unmask μ_{s} and get back response r of a node), and the traversal key of q.

In more details, by unmasking the appropriate index of I, the server locates the root node of q, and traverses down by unlocking the traversal key of the children nodes iteratively. Parallel traversal is done by traversing both the left and right sub-trees of a node simultaneously. Upon arrival at a node, it uses the search key to unmask μ_{s}. The response to client contains all μ_{s} obtained during traversal.

(c) Meta-Queries: For meta-query (q, r), the client only sends the index and the index mask to the server (while the search key and traversal key are replaced by random strings). This means that the server is able to locate the node corresponding to (q, r), but cannot obtain the μ_{s} stored nor traverse down the sub-tree. Nevertheless, the server performs the same operations as for (normal) queries and returns the "unmasked" μ_{s} if a node is located. The client interprets the response as false if the server returns the empty set ϕ, or true otherwise.

(d) Add and Link Updates: The server creates a new node to be inserted under a random address in A. Adding a new query q or response r are considered to be *Add* updates, while adding a new data $d = (q, r)$ is a *Link* update.

For the *Add* update, the new node for q or r serves as the root node. For the *Link* update, node d is inserted into the tree corresponding to query q. To do this, the update token includes the traversal key of the root node, so that the server can use it to unmask the traversal keys of its children, traverse down the tree, and update the tree linkage. The same procedure is then repeated for adding the dual node of d. Figure 2 shows an example of a "Link" update.

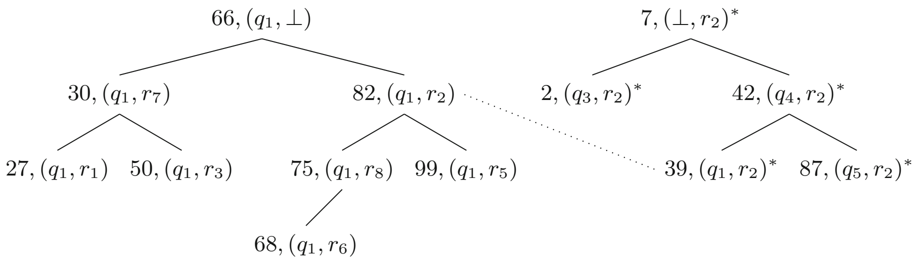

Fig. 2. Adding (q_1, r_6) to address 68

(e) Unlink Updates: Deleting $d = (q, r)$ from the database is considered to be an *Unlink* update. The server looks up I and locates the normal node for d in A, traverses down the sub-tree using the traversal key $T_b(d)$ to find the right-most left-sibling (or left-most right-sibling), and replaces the target node with the sibling. The same procedure is repeated for removing the dual node of d. Figure 3 shows an example of an "Unlink" update.

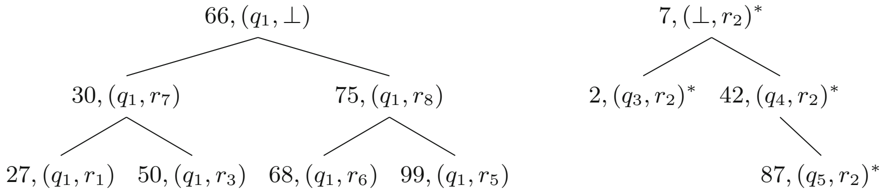

Fig. 3. Removing (q_1, r_2) from address 82 (replaced by (q_1, r_8) in address 75) and $(q_1, r_2)^*$ in address 39

(f) Delete Updates: To delete a response r, the server traverses the dual tree corresponding to r and delete all the dual nodes down the tree. Each dual of the dual nodes, which is a normal node, is also deleted from the corresponding normal tree. Parallel deletion is possible by deleting the left and right sub-trees simultaneously. Similar procedures can be done to delete a query q.

4.3 Concrete Construction

Now we give the details in how to construct our RBT scheme, according to the high-level description explained in the last sub-section. This part will be especially helpful for those who want to implement or possibly optimize our scheme. Recall that in last sub-section we have explained the encryption/decryption part of RBT. The rest is mostly about tree traversal and addition/deletion of nodes, which should be simple to understand for any computer scientists. While conceptually simple, writing down the actual steps in algorithm require a careful management of the pointers involved in (possibly more than one kinds of) the tree. Readers who are interested in its security can go straight to Sect. 4.4, or the performance evaluation in Sect. 5 which also explains part of the codes below and their sub-routines in Appendix A.

Let δ be a data structure of type $\mathcal{T}$ of size parameter (M, N) as defined in Sect. 2. Let $*$ be a special symbol denoting an empty string. Let $\mathcal{F} = \{\{F_b, G_b, T_b, D_b\}_{b \in \{0,1\}}, S\}$ be a set of PRFs such that for each $f \in \mathcal{F}$, $f : \{0,1\}^\lambda \times \{0,1\}^* \rightarrow \{0,1\}^\lambda$. Let $H_s : \{0,1\}^\lambda \times \{0,1\}^* \rightarrow \{0,1\}^\lambda$, $H_t : \{0,1\}^\lambda \times \{0,1\}^* \rightarrow \{0,1\}^{5\lambda}$, and $H_d : \{0,1\}^\lambda \times \{0,1\}^* \rightarrow \{0,1\}^{2\lambda}$ be another three PRFs to be modeled as random oracles. All PRFs use different keys. For brevity, we will not specify the key each time we use a PRF.

Our scheme $\mathsf{RBT} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{QryTkn}, \mathsf{Qry}, \mathsf{UdtTkn}, \mathsf{Udt}, \mathsf{Dec})$ is defined as follows, and the sub-routines $\mathsf{QryTrav}$, Ins, Del, $\mathsf{DelTrav}$, and replc are defined in Appendix A.

```
Algorithm $K \leftarrow \mathsf{Gen}(1^\lambda)$:
 1: Generate $\lambda$-bit random strings as keys of each PRF
 2: Output $K$ which includes all the generated keys
```

```
Algorithm $\gamma \leftarrow \mathsf{Enc}(K, \delta)$:
 1: Initialize empty dictionaries $I$ and $A$
 2: Set $\gamma = (I, A)$
 3: for all $d = (q; r) \in \delta$ do
 4:     if $\hat{q} = (q; *)$ is not added then
 5:         Run $\tau_u \leftarrow \mathsf{UdtTkn}(K, (\text{“Add”}, (\hat{q})))$
 6:         Run $\gamma \leftarrow \mathsf{Udt}(\tau_u, \gamma)$
 7:     end if
 8:     if $\hat{r} = (*; r)$ is not added then
 9:         Run $\tau_u \leftarrow \mathsf{UdtTkn}(K, (\text{“Add”}, (\hat{r})))$
10:         Run $\gamma \leftarrow \mathsf{Udt}(\tau_u, \gamma)$
11:     end if
12:     Run $\tau_u \leftarrow \mathsf{UdtTkn}(K, (\text{“Add”}, d))$
13:     Run $\gamma \leftarrow \mathsf{Udt}(\tau_u, \gamma)$
14: end for
15: Return $\gamma$
```

```
Algorithm $\tau_q \leftarrow \mathsf{QryTkn}(K, d)$:
 1: Parse $d$ as $(q, r)$
 2: if $r = *$ then
 3:     Return $\tau_q = (F_0(d), G_0(d), T_0(d), S(d))$
 4: else
 5:     Return $\tau_q = (F_b(d), G_b(d), t, s)$, where $b \leftarrow \{0,1\}$ and $t, s \leftarrow \{0,1\}^\lambda$
 6: end if
```

```
Algorithm $\mathcal{R} \leftarrow \mathsf{Qry}(\tau_q, \gamma)$:
 1: Parse $\gamma$ as $(I, A)$ and $\tau_q$ as $(\tau_1, \tau_2, \tau_3, \tau_4)$
 2: Abort if $\tau_1$ is not in $I$
 3: Retrieve $\mathsf{addr} = I[\tau_1] \oplus \tau_2$
 4: Run $\mathcal{R} \leftarrow \mathsf{QryTrav}(\mathsf{addr}, \tau_3, \tau_4)$
 5: Return $\mathcal{R}$
```

Algorithm $\tau_u \leftarrow \mathsf{UdtTkn}(K, (mode, d))$:

```
 1: Parse $d$ as $(q, r)$
 2: if $mode$ = "Add" and $(q = * \text{ or } r = *)$ then
 3:     Set $b = (q = *)$
 4:     Set $\overline{\mu_\mathsf{s}} \leftarrow \{0,1\}^\lambda$
 5:     Set $\overline{\mu_\mathsf{t}} = H_t(T_b(d), r_t)$
 6:     Set $\overline{\mu_\mathsf{d}} \leftarrow \{0,1\}^{2\lambda}$
 7:     Set $\mu_\mathsf{a} = (r_s, r_t, r_d) \leftarrow \{0,1\}^{3\lambda}$
 8:     Set $\tau_u = (\text{"Add"}, F_b(d), G_b(d), (\overline{\mu_\mathsf{s}}, \overline{\mu_\mathsf{t}}, \overline{\mu_\mathsf{d}}, \mu_\mathsf{a}))$
 9: else if $mode$ = "Add", $q \neq *$ and $r \neq *$ then
10:     Set $\mu_\mathsf{a} = (r_s, r_t, r_d) \leftarrow \{0,1\}^{3\lambda}$
11:     Set $\mu_\mathsf{a}' = (r_s', r_t', r_d') \leftarrow \{0,1\}^{3\lambda}$
12:     Set $\overline{\mu_\mathsf{s}} = r \oplus H_s(S(\hat{q}), r_s)$.
13:     Set $\overline{\mu_\mathsf{s}'} \leftarrow \{0,1\}^\lambda$
14:     Set $\overline{\mu_\mathsf{t}} = H_t(T_0(d), r_t)$
15:     Set $\overline{\mu_\mathsf{t}'} = H_t(T_1(d), r_t')$
16:     Set $\overline{\mu_\mathsf{d}} = (0, T_1(d)) \oplus H_d(D_0(d), r_d)$
17:     Set $\overline{\mu_\mathsf{d}'} = (0, T_0(d)) \oplus H_d(D_1(d), r_d')$
18:     Set
            $\tau_u = (\text{"Link"},$
            $F_0(\hat{q}), G_0(\hat{q}), T_0(\hat{q}), F_0(d), G_0(d), T_0(d),$
            $F_1(\hat{r}), G_1(\hat{r}), T_1(\hat{r}), F_1(d), G_1(d), T_1(d),$
            $(\overline{\mu_\mathsf{s}}, \overline{\mu_\mathsf{t}}, \overline{\mu_\mathsf{d}}, \mu_\mathsf{a}), (\overline{\mu_\mathsf{s}'}, \overline{\mu_\mathsf{t}'}, \overline{\mu_\mathsf{d}'}, \mu_\mathsf{a}'))$
19: else if $mode$ = "Del" and $(q = * \text{ or } r = *)$ then
20:     Set $b = (q = *)$
21:     Set $\tau_u = (\text{"Del"}, F_b(d), G_b(d), T_b(d), D_b(d))$
22: else if $mode$ = "Del", $q \neq *$ and $r \neq *$ then
23:     Set $\tau_u =$
            $(\text{"Unlink"}, F_0(d), G_0(d), T_0(d), F_1(d), G_1(d), T_1(d))$
24: else
25:     Set $\tau_u = \perp$
26: end if
27: Return $\tau_u$
```

Algorithm $\gamma' \leftarrow \mathsf{Udt}(\tau_u, \gamma)$:

```
 1: Parse $\gamma$ as $(I, A)$ and $\tau_u$ as $(mode, \tau_1, \tau_2, \ldots)$
 2: if $mode$ = "Add" then
 3:     Recall that
            $\tau_u = (\text{"Add"}, F_b(d), G_b(d), (\overline{\mu_\mathsf{s}}, \overline{\mu_\mathsf{t}}, \overline{\mu_\mathsf{d}}, \mu_\mathsf{a}))$
```

```
4:      Abort if $\tau_1$ or $\tau_4$ is in $I$
5:      **repeat**
6:          Sample $\mathsf{root} \leftarrow \{0,1\}^\lambda$
7:      **until** $A[\mathsf{root}]$ is free
8:      Set $I[\tau_1] = \mathsf{root} \oplus \tau_2$
9:      Set $A[\mathsf{root}] = \tau_3$
10: **else if** $\mathit{mode}$ = "Link" **then**
11:     Recall that
```

$$\begin{aligned}\tau_u = (&\text{“Link”},\\ &F_0(\hat{q}), G_0(\hat{q}), T_0(\hat{q}), F_0(d), G_0(d), T_0(d),\\ &F_1(\hat{r}), G_1(\hat{r}), T_1(\hat{r}), F_1(d), G_1(d), T_1(d),\\ &(\overline{\mu_\mathsf{s}}, \overline{\mu_\mathsf{t}}, \overline{\mu_\mathsf{d}}, \mu_\mathsf{a}), (\overline{\mu_\mathsf{s}'}, \overline{\mu_\mathsf{t}'}, \overline{\mu_\mathsf{d}'}, {\mu_\mathsf{a}}'))\end{aligned}$$

```
12:     Abort if $\tau_1$ or $\tau_7$ is not in $I$
13:     Set $\mathsf{root_q} = I[\tau_1] \oplus \tau_2$
14:     Set $\mathsf{root_r} = I[\tau_7] \oplus \tau_8$
15:     **repeat**
16:         Sample $\mathsf{tgt}, \mathsf{dual} \leftarrow \{0,1\}^\lambda$
17:     **until** $A[\mathsf{tgt}]$ and $A[\mathsf{dual}]$ are free
18:     Set $I[\tau_4] = \mathsf{tgt} \oplus \tau_5$ and $I[\tau_{10}] = \mathsf{dual} \oplus \tau_{11}$
19:     Set $A[\mathsf{tgt}] = \tau_{13}$ and $A[\mathsf{dual}] = \tau_{14}$
20:     Set $A[\mathsf{tgt}].\overline{\mu_\mathsf{d}}.\mathsf{dual} \mathrel{\oplus}= \mathsf{dual}$
21:     Set $A[\mathsf{dual}].\overline{\mu_\mathsf{d}}.\mathsf{dual} \mathrel{\oplus}= \mathsf{tgt}$
22:     Run $A \leftarrow \mathsf{Ins}(\mathsf{root_q}, \tau_3, \mathsf{tgt}, \tau_6)$
23:     Run $A \leftarrow \mathsf{Ins}(\mathsf{root_r}, \tau_9, \mathsf{dual}, \tau_{12})$
24: **else if** $\mathit{mode}$ = "Del" **then**
25:     Recall that
```

$$\tau_u = (\text{“Del”}, F_b(d), G_b(d), T_b(d), D_b(d))$$

```
26:     Abort if $\tau_1$ is not in $I$
27:     Set $\mathsf{root} = I[\tau_1] \oplus \tau_2$
28:     Run $A \leftarrow \mathsf{DelTrav}(\mathsf{root}, \tau_3, \tau_4)$
29: **else if** $\mathit{mode}$ = "Unlink" **then**
30:     Recall that
```

$$\tau_u = (\text{“Unlink”}, F_0(d), G_0(d), T_0(d), F_1(d), G_1(d), T_1(d))$$

```
31:     Set $\mathsf{tgt} = I[\tau_1] \oplus \tau_2$
32:     Set $\mathsf{dual} = I[\tau_4] \oplus \tau_6$
33:     Run $A \leftarrow \mathsf{Del}(\mathsf{tgt}, \tau_3)$
34:     Run $A \leftarrow \mathsf{Del}(\mathsf{dual}, \tau_6)$
35: **end if**
36: Set $\gamma = (I, A)$
37: Return $\gamma$
```

4.4 Security Analysis

We follow the existing framework [7] which describes the security of SSE schemes against an honest-but-curious server by a set of leakage functions $(\mathcal{L}_e, \mathcal{L}_q, \mathcal{L}_u)$ for encryption, queries, and updates respectively. RBT leaks information about the internal data structure when performing updates on the tree structure. Its security is asserted in Theorem 1 while the details of $(\mathcal{L}_e,\ \mathcal{L}_q,\ \mathcal{L}_u)$ are specified in its proof. The proof can be found in Appendix B.

Theorem 1. *The dynamic searchable symmetric encryption scheme on structured data presented above is $(\mathcal{L}_e,\ \mathcal{L}_q,\ \mathcal{L}_u)$-secure against adaptive dynamic chosen-query attacks in the random oracle model.*

5 Efficiency Evaluation

5.1 Complexities Analysis

Let p be the number of processors and m be the number of data related to a given query q or response r. It is easy to see from Qry algorithm that (after Line 1–3 which takes $O(1)$ time) it just applies QryTrav to traverse from the root of a tree. The algorithm QryTrav (after Line 4–8 which recovers the key for unwrapping the two child pointers in particular) just applies QryTrav to traverse the tree recursively. So the query complexity of our scheme is optimal, namely $O(m/p)$.

The update algorithm Udt encapsulates different modes of updates, namely, "Add", "Link", "Unlink", and "Delete". For "Add" update, which just samples a free address (Line 5–7) and masks them (Line 8–9) from the corresponding keys in the update token (Line 3), is constant time. "Link" and "Unlink" updates have complexity $O(\log m)$. Here we just explain "Link". Similar to "Add', it firstly parses the update token (Line 11). From there, the root addresses for q and r are obtained (Line 13–14). To insert the new node, it samples a target address (tgt) for storing the node itself and dual for storing its dual (Line 15–17), sets them up (*e.g.*, masking) appropriately (Line 18–21), and eventually calls Ins (Line 22–23) for locating the actual place to insert into an existing tree. Ins then calls itself recursively if needed just like the traversal in QryTrav. The longest traversal happens when it is inserted at the leaves level of the tree having m nodes, hence the complexity is $O(\log m)$.

Finally, "Delete" mode of update, *i.e.*, DelTrav, traverses the tree to find the node to be deleted similar to Qry. This traversal can be done in parallel, results in a complexity of $O(m/p)$. The sub-routine Del in DelTrav performs the actual deletion. It updates the pointers related to the normal node and the dual node accordingly after finding the replacement node, which is in $O(1)$ time. The step of finding replacement node via Replc simply traverses a tree which can be done in $O(\log m)$ time. To summarize, the complexity of the whole DelTrav algorithm is $O(m/p)$.

5.2 Experiments on Implementations

To demonstrate the applicability of RBT, we consider a privacy-preserving version of decentralized social networks where user connections are represented by graphs. The connections between users are encrypted by RBT, and are searchable by the users possessing search tokens delegated by the host. As described in Sect. 2.2, our scheme naturally supports "friends of friends" and "are Alice and Bob friends" types of queries.

To evaluate the performance of our scheme we implemented RBT in C++ using Crypto++ 5.6.2 library for cryptographic primitives and Intel Threading Building Blocks 4.2 Update 3 library for multi-threading. All PRFs are implemented by HMAC-SHA256. All computations were performed locally in memory (without network transfer). A distinctive feature of RBT over existing schemes is that it supports non-interactive parallel queries and updates. Computations are sequential unless specified.

The experiments were conducted on a machine with Intel Core i5-4590 at 3.50 GHz and 8.00 GB of memory running Windows 8.1. In each experiment, we used RBT to encrypt a set of synthetic data or real-life data. For real-life data, we used a graph [9] representing some Facebook social circles with 4039

Table 4. Timing for RBT ("//" denotes parallel computation)

Type	Synthetic	Synthetic	Facebook
M	500	1000	4039
N	500	1000	4039
$\|\delta\|$	125,000	500,000	176,468
Density	50%	50%	1.08%
Enc	88 s	451 s	110 s
QryTkn (Normal)	15 µs	17 µs	15 µs
QryTkn (Meta)	12 µs	12 µs	12 µs
Qry (Meta)	420 µs	1173 µs	5 µs
Qry (Normal, //)	39 µs	101 µs	59 µs
Qry (Normal)	64 µs	168 µs	94 µs
UdtTkn (Add)	122 µs	121 µs	121 µs
UdtTkn (Link)	139 µs	137 µs	141 µs
UdtTkn (Delete)	14 µs	20 µs	12 µs
UdtTkn (Unlink)	11 µs	11 µs	12 µs
Udt (Add)	577 µs	780 µs	453 µs
Udt (Delete, //)	29 ms	89 ms	5 ms
Udt (Delete)	38 ms	133 ms	9 ms
Udt (Link)	582 µs	783 µs	472 µs
Udt (Unlink)	360 µs	566 µs	353 µs

nodes and 88,234 undirected (*i.e.*, 176,468 directed) edges. The edge density is relatively small for this set of data. Hence, we also perform experiments on synthetic data which better model other application scenarios. The synthetic data contains graphs with 500 and 1000 nodes respectively with 50% of edge density.

The timing for encryption, "Add" updates, and "Link" updates, are computed by taking the average time needed for the respective operations for building the encrypted database from scratch. The timing for queries, "Delete" updates, and "Link" updates are computed by taking the average time needed for 100 times of the respective operations selected at random. For the timing of normal queries, the values are further divided by the number of responses returned by each query.

Our implementations were hardly optimized, yet the results show the moderate efficiency of our scheme; in particular, parallel computation effectively reduces the time for queries and especially for deletion (Table 4).

6 Conclusion

Searchable symmetric encryption (SSE) has been extensively studied in recent years. One can view the researches on designing SSE as finding a desirable tradeoff between functionalities, security, and efficiency. As shown in the literature, devising an SSE scheme which simultaneously achieves a number of desirable properties across these three domains is not an easy task. In this paper, we presented an SSE scheme on structured data supporting parallel traversal. Our aim is to achieve optimal query efficiency while minimizing leakage and communication incurred by the updates.

The abstract data type supported by our SSE scheme can represent queries over many common structured data. In particular, we consider an online social network such as Facebook. The connections between users can be represented by graphs, and common types of queries such as "friends of Alice" and "are Alice and Bob friends" can be represented by neighbor and adjacency queries respectively, which naturally correspond to the normal and meta queries over our abstract data type.

Moreover, we demonstrated the practicality of our scheme by evaluating its efficiency against both real-life graph data of online social network, and synthetic data for graphs in general. We believe our work makes an important step in advancing the field of SSE.

A Sub-routines in Our Construction

To make our scheme easier to understand, we modularize a number of operations for traversal during a query, insertion, and (the traversal needed for) deletion. Specifically, now we give the details of the operations[1] performed in the subroutines $\mathsf{QryTrav}$, Ins, Del, $\mathsf{DelTrav}$, and Replc. Algorithms $\mathsf{QryTrav}$ and $\mathsf{DelTrav}$

[1] Our poster [8] suggested a preliminary idea of using tree structure, but gave no details on the actual construction.

are invoked by Qry and "Del" mode of Udt respectively to traverse the binary search trees. They are identical to ordinary tree traversal algorithms except that the addresses of the children nodes need to be unmasked by using the traversal key. Due to the nature of tree traversal, QryTrav and DelTrav are parallelizable.

```
Algorithm $\mathcal{R} \leftarrow \mathsf{QryTrav}(\mathsf{tgt}, \mathsf{tkey}, \mathsf{k_{srch}})$:
 1: if $A[\mathsf{tgt}]$ is free then
 2:     Return $\phi$
 3: end if
 4: Compute $h_0 = H_s(\mathsf{k_{srch}}, A[\mathsf{tgt}].\mu_\mathsf{a}.r_s)$
 5: Compute $h_1 = H_t(\mathsf{tkey}, A[\mathsf{tgt}].\mu_\mathsf{a}.r_t)$
 6: Compute $\mu_\mathsf{s} = A[\mathsf{tgt}].\overline{\mu_\mathsf{s}} \oplus h_0$
 7: Compute $\mu_\mathsf{t} = A[\mathsf{tgt}].\overline{\mu_\mathsf{t}} \oplus h_1$
 8: Parse $\mu_\mathsf{t}$ as $(\mathsf{prt}, \mathsf{chd}_0, \mathsf{k}_0, \mathsf{chd}_1, \mathsf{k}_1)$
 9: Set $\mathcal{R}_0 \leftarrow \mathsf{QryTrav}(\mathsf{chd}_0, \mathsf{k}_0, \mathsf{k_{srch}})$
10: Set $\mathcal{R}_1 \leftarrow \mathsf{QryTrav}(\mathsf{chd}_1, \mathsf{k}_1, \mathsf{k_{srch}})$
11: Return $\mathcal{R} = \mathcal{R}_0 \cup \mathcal{R}_1 \cup \{\mu_\mathsf{s}\}$
```

```
Algorithm $A' \leftarrow \mathsf{DelTrav}(\mathsf{tgt}, \mathsf{tkey}, \mathsf{dkey})$:
 1: if $A[\mathsf{tgt}]$ is free then
 2:     Return $A$
 3: end if
 4: Compute $h_1 = H_t(\mathsf{tkey}, A[\mathsf{tgt}].\mu_\mathsf{a}.r_t)$
 5: Compute $h_2 = H_d(\mathsf{dkey}, A[\mathsf{tgt}].\mu_\mathsf{a}.r_d)$
 6: Compute $\mu_\mathsf{t} = A[\mathsf{tgt}].\overline{\mu_\mathsf{t}} \oplus h_1$
 7: Compute $\mu_\mathsf{d} = A[\mathsf{tgt}].\overline{\mu_\mathsf{d}} \oplus h_2$
 8: Parse $\mu_\mathsf{t}$ as $(\mathsf{prt}, \mathsf{chd}_0, \mathsf{k}_0, \mathsf{chd}_1, \mathsf{k}_1)$
 9: Parse $\mu_\mathsf{d}$ as $(\mathsf{dual}, \mathsf{k_D})$
10: Run $A \leftarrow \mathsf{Del}(\mathsf{dual}, \mathsf{k_D})$
11: Run $A \leftarrow \mathsf{DelTrav}(\mathsf{chd}_0, \mathsf{k}_0, \mathsf{dkey})$
12: Run $A \leftarrow \mathsf{DelTrav}(\mathsf{chd}_1, \mathsf{k}_1, \mathsf{dkey})$
13: Remove tgt from $A$
14: Return $A$
```

Ins is identical to an ordinary tree insertion algorithm except that it uses the traversal key to unmask the addresses of the children. Note that Ins determines the position of the insertion based on the address, which is chosen at random.

```
Algorithm $A' \leftarrow \mathsf{Ins}(\mathsf{root}, \mathsf{tkey_{root}}, \mathsf{tgt}, \mathsf{tkey_{tgt}})$:
 1: Compute $h_1 = H_t(\mathsf{tkey_{root}}, A[\mathsf{root}].\mu_\mathsf{a}.r_t)$
 2: Compute $\mu_\mathsf{t} = A[\mathsf{root}].\overline{\mu_\mathsf{t}} \oplus h_1$
 3: Parse $\mu_\mathsf{t}$ as $(\mathsf{prt}, \mathsf{chd}_0, \mathsf{k}_0, \mathsf{chd}_1, \mathsf{k}_1)$
 4: Set $b = (\mathsf{addr} > \mathsf{root})$
 5: if $\mathsf{chd}_b = 0$ then
 6:     Set $A[\mathsf{root}].\overline{\mu_\mathsf{t}}.\mathsf{chd}_b \oplus= \mathsf{tgt}$
 7:     Set $A[\mathsf{root}].\overline{\mu_\mathsf{t}}.\mathsf{k_b} \oplus= \mathsf{tkey_{tgt}}$
 8:     Set $A[\mathsf{tgt}].\overline{\mu_\mathsf{t}}.\mathsf{prt} \oplus= \mathsf{root}$
```

```
9: else
10:     Run $A \leftarrow \mathsf{Ins}(\mathsf{chd}_b, \mathsf{k_b}, \mathsf{tgt}, \mathsf{tkey}_{\mathsf{tgt}})$
11: end if
12: Output $A$
```

Del is identical to an ordinary tree deletion algorithm except that it uses the traversal key to unmask the addresses of the children nodes, and uses the dual key to unmask the addresses of the dual node. It also truly deletes the target node by updating its children and parent to point to the replacement node, instead of just copying the values as in some tree deletion algorithms, or just marked as "deletion pending" which requires housekeeping later. The replacement node is either the right-most left-sibling or the left-most right-sibling of the tree, computed using the algorithm Replc.

Algorithm $A' \leftarrow \mathsf{Del}(\mathsf{tgt}, \mathsf{tkey}_{\mathsf{tgt}})$:

```
1: if $A[\mathsf{tgt}]$ is free then
2:     Return $A$
3: end if
4: Compute $h_1 = H_t(\mathsf{tkey}_{\mathsf{tgt}}, A[\mathsf{tgt}].\mu_{\mathsf{a}}.r_t)$
5: Compute $\mu_{\mathsf{t}} = A[\mathsf{tgt}].\overline{\mu_{\mathsf{t}}} \oplus h_1$
6: Parse $\mu_{\mathsf{t}}$ as $(\mathsf{prt}, \mathsf{chd}_0, \mathsf{k}_0, \mathsf{chd}_1, \mathsf{k}_1)$
   ▷ Updating Replacement Node and its Neighbors
7: Sample $b \leftarrow \{0, 1\}$
8: Compute $\Delta \leftarrow \mathsf{replc}(b, \mathsf{chd}_b, \mathsf{k_b})$
9: Parse $\Delta$ as $(\mathsf{replc}, \mathsf{tkey}_{\mathsf{replc}}, {\mu_{\mathsf{t}}}')$
10: Parse ${\mu_{\mathsf{t}}}'$ as $(\mathsf{prt}', \mathsf{chd}_0', {\mathsf{k}_0}', \mathsf{chd}_1', {\mathsf{k}_1}')$
11: Set $A[\mathsf{prt}'].\overline{\mu_{\mathsf{t}}}.\mathsf{chd}_{1-b} \oplus= \mathsf{replc} \oplus \mathsf{chd}_b'$
12: Set $A[\mathsf{prt}'].\overline{\mu_{\mathsf{t}}}.\mathsf{k}_{1-\mathsf{b}} \oplus= \mathsf{tkey}_{\mathsf{replc}} \oplus {\mathsf{k_b}}'$
13: Set $A[\mathsf{chd}_b'].\overline{\mu_{\mathsf{t}}}.\mathsf{prt} \oplus= \mathsf{replc} \oplus \mathsf{prt}'$
14: Set $A[\mathsf{replc}].\overline{\mu_{\mathsf{t}}}.\mathsf{prt} \oplus= \mathsf{prt}' \oplus \mathsf{prt}$
15: Set $A[\mathsf{replc}].\overline{\mu_{\mathsf{t}}}.\mathsf{chd}_0 \oplus= \mathsf{chd}_0' \oplus \mathsf{chd}_0$
16: Set $A[\mathsf{replc}].\overline{\mu_{\mathsf{t}}}.\mathsf{k}_0 \oplus= {\mathsf{k}_0}' \oplus \mathsf{k}_0$
17: Set $A[\mathsf{replc}].\overline{\mu_{\mathsf{t}}}.\mathsf{chd}_1 \oplus= \mathsf{chd}_1' \oplus \mathsf{chd}_1$
18: Set $A[\mathsf{replc}].\overline{\mu_{\mathsf{t}}}.\mathsf{k}_1 \oplus= {\mathsf{k}_1}' \oplus \mathsf{k}_1$
   ▷ Updating Target Node and its Neighbors
19: Set $b = (\mathsf{tgt} > \mathsf{prt})$
20: Set $A[\mathsf{prt}].\overline{\mu_{\mathsf{t}}}.\mathsf{chd}_b \oplus= \mathsf{tgt} \oplus \mathsf{replc}$
21: Set $A[\mathsf{prt}].\overline{\mu_{\mathsf{t}}}.\mathsf{k_b} \oplus= \mathsf{tkey}_{\mathsf{tgt}} \oplus \mathsf{tkey}_{\mathsf{replc}}$
22: Set $A[\mathsf{chd}_0].\overline{\mu_{\mathsf{t}}}.\mathsf{prt} \oplus= \mathsf{tgt} \oplus \mathsf{replc}$
23: Set $A[\mathsf{chd}_1].\overline{\mu_{\mathsf{t}}}.\mathsf{prt} \oplus= \mathsf{tgt} \oplus \mathsf{replc}$
24: Remove $\mathsf{tgt}$ from $A$
25: Return $A$
```

Algorithm $(\mathsf{replc}, \mathsf{tkey}_{\mathsf{replc}}, \mu_{\mathsf{t}}) \leftarrow \mathsf{Replc}(b, \mathsf{tgt}, \mathsf{tkey}_{\mathsf{tgt}})$:

```
1: if $A[\mathsf{tgt}]$ is free then
2:     Return $(\mathsf{tgt}, \mathsf{tkey}_{\mathsf{tgt}}, \mathbf{0}_5)$
3: end if
```

```
 4: Compute $h_1 = H_t(\mathsf{tkey}_{\mathsf{tgt}}, A[\mathsf{tgt}].\mu_{\mathsf{a}}.r_t)$
 5: Compute $\mu_{\mathsf{t}} = A[\mathsf{tgt}].\overline{\mu_{\mathsf{t}}} \oplus h_1$
 6: Parse $\mu_{\mathsf{t}}$ as $(\mathsf{prt}, \mathsf{chd}_0, \mathsf{k}_0, \mathsf{chd}_1, \mathsf{k}_1)$
 7: if $A[\mathsf{chd}_{1-b}]$ is free then
 8:     Return $(\mathsf{tgt}, \mathsf{tkey}_{\mathsf{tgt}}, \mu_{\mathsf{t}})$
 9: else
10:     Return $\mathsf{replc}(b, \mathsf{chd}_{1-b}, \mathsf{k}_{1-\mathsf{b}})$
11: end if
```

B Security Proof

Proof (of Theorem 1). The leakage of our scheme is implied by the capability of the search keys skey, traversal key tkey, and dual keys dkey. Initially, with the encrypted database, $\mathcal{L}_e$ leaks the size of itself, namely $|\mathcal{Q}| + |\mathcal{R}| + 2|\delta|$. Suppose each of the $|\mathcal{Q}| + |\mathcal{R}| + 2|\delta|$ nodes has a unique identifier. $\mathcal{L}_q$ leaks upon a query q the access pattern, or precisely all μ_{s} stored in the normal nodes corresponding to $(q; \cdot)$. It also leaks the identifiers of these nodes. $\mathcal{L}_u$ leaks upon an update the type of the update. In addition, a "Link" update for $d = (q; r)$ leaks the identifiers of the normal nodes for $(q; \cdot)$, and the identifiers of the dual nodes for $(\cdot; r)$; an "Unlink" update for $d = (q; r)$ leaks the identifiers of the normal nodes for $(q; \cdot)$ under the sub-tree rooted at the normal node for $(q; r)$, and the identifiers of the dual nodes for $(\cdot; r)$ under the sub-tree rooted at the dual node for $(q; r)$; a "Delete" update for $(q; *)$ (resp. $(*; r)$) leaks the identifiers of the normal (resp. dual) nodes for $(q; *)$ (resp. $(*; r)$), as well the identifiers of the corresponding dual (resp. normal) nodes of these nodes.

To prove the security of our scheme, we need to construct a simulator $\mathcal{S}$ which interacts with an adversary $\mathcal{A}$ in the experiment $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(1^\lambda)$ defined in Definition 2. Due to space limitation, we provide the essential idea here, but remark that the simulation is straightforward given the set of leakage functions and follows the same structure of existing proofs [2,6,7].

The simulator simulates the encrypted database by random dictionaries of appropriate sizes given by the leakage function $\mathcal{L}_e$. It simulates all PRF by random functions, and all random oracles (RO) by maintaining and programming the corresponding tables.

For each query/update, the respective leakage $(\mathcal{L}_q/\mathcal{L}_u)$ reveals the identifiers of some of the nodes stored in A. The simulator programs the corresponding RO (*e.g.*, H_s and H_t for queries) such that on input the corresponding simulated key (*e.g.*, skey) and randomness (*e.g.*, r_s) stored in μ_{a} of the entry, it produces the suitable mask.

Finally, for answering the random oracle queries, it checks whether the answer for this query to the random oracle is programmed to some particular value. If so, it outputs the programmed value. Otherwise, it outputs a random value. The only possibility that an adversary can distinguish the simulated database from the real database is when it queries the random oracle for a valid pair of (key, randomness), while the corresponding information is not yet revealed in any queries or updates. However, since all simulated keys are produced by random functions, the probability of having such collision is negligible. □

References

1. Chase, M., Kamara, S.: Structured encryption and controlled disclosure. In: Abe, M. (ed.) ASIACRYPT 2010. LNCS, vol. 6477, pp. 577–594. Springer, Heidelberg (2010). doi:10.1007/978-3-642-17373-8_33
2. Curtmola, R., Garay, J.A., Kamara, S., Ostrovsky, R., Encryption, S.S.: Improved definitions and efficient constructions. J. Comput. Secur. **19**(5), 895–934 (2011)
3. Goh, E.-J.: Secure Indexes. Cryptology ePrint Archive, Report 2003/216
4. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious RAMs. J. ACM **43**(3), 431–473 (1996)
5. Hahn, F., Kerschbaum, F.: Searchable encryption with secure and efficient updates. In: ACM Conference on Computer and Communications Security (CCS), pp. 310–320 (2014)
6. Kamara, S., Papamanthou, C.: Parallel and dynamic searchable symmetric encryption. In: Financial Cryptography, pp. 258–274 (2013)
7. Kamara, S., Papamanthou, C., Roeder, T.: Dynamic searchable symmetric encryption. In: ACM Conference on Computer and Communications Security (CCS), pp. 965–976 (2012)
8. Lai, R., Chow, S.: Structured encryption with non-interactive updates and parallel traversal. In: IEEE International Conference on Distributed Computing Systems (ICDCS), pp. 776–777 (2015)
9. Leskovec, J., Krevl, A., Datasets, S.: Stanford Large Network Dataset Collection, June 2014. http://snap.stanford.edu/data
10. Reed, B.A.: The height of a random binary search tree. J. ACM **50**(3), 306–332 (2003)
11. Song, D.X., Wagner, D., Perrig, A.: Practical techniques for searches on encrypted data. In: IEEE Symposium on Security and Privacy, pp. 44–55 (2000)
12. Stefanov, E., Papamanthou, C., Shi, E.: Practical dynamic searchable encryption with small leakage. In: NDSS (2014)
13. Wang, B., Hou, Y., Li, M., Wang, H., Li, H.: Maple: scalable multi-dimensional range search over encrypted cloud data with tree-based index. In: ASIACCS, pp. 111–122 (2014)
14. Wang, Q., He, M., Du, M., Chow, S.S.M., Lai, R.W.F., Zou, Q.: Searchable encryption over feature-rich data. IEEE Trans. Depend. Secure Comput. (to appear). doi:10.1109/TDSC.2016.2593444