Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

TC 11 Briefing Papers

Achieve efficient position-heap-based privacy-preserving substring-of-keyword query over cloud

Fan Yin[a,b], Rongxing Lu[b,*], Yandong Zheng[b], Jun Shao[c], Xue Yang[d,e], Xiaohu Tang[a]

[a] The Information Security and National Computing Grid Laboratory, Southwest Jiaotong University, Chengdu, 611756, China
[b] The Canadian Institute for Cybersecurity, Faculty of Computer Science, University of New Brunswick, Fredericton, E3B 5A3 Canada
[c] School of Computer and Information Engineering, Zhejiang Gongshang University, Hangzhou, 310018 China
[d] The Tsinghua Shenzhen International Graduate School, Tsinghua University, Shenzhen, 518055, China
[e] The PCL Research Center of Networks and Communications, Peng Cheng Laboratory, Shenzhen, 518055, China

ARTICLE INFO

Article history:
Received 20 December 2020
Revised 7 July 2021
Accepted 3 August 2021
Available online 10 August 2021

Keywords:
Cloud computing
Outsourced encrypted data
Substring-of-keyword query
Position heap
Efficiency

ABSTRACT

The cloud computing technique, which was initially used to mitigate the explosive growth of data, has been required to take both data privacy and users' query functionality into consideration. Symmetric searchable encryption (SSE) is a popular solution to supporting efficient keyword queries over encrypted data in the cloud. However, most of the existing SSE schemes focus on the exact keyword query and cannot work well when the user only remembers the substring of a keyword, i.e., substring-of-keyword query. This paper aims to investigate this issue by proposing an efficient and privacy-preserving substring-of-keyword query scheme over cloud. First, we employ the position heap technique to design a novel tree-based index to match substrings with corresponding keywords. Then based on the tree-based index, we introduce our substring-of-keyword query scheme, which contains two consecutive phases. The first phase queries the keywords that match a given substring, and the second phase queries the files that match a keyword in which people are really interested. In addition, detailed security analysis and experimental results demonstrate the security and efficiency of our proposed scheme.

1. Introduction

The rapid development of information techniques, e.g., internet of things, smart building, etc., has been promoting the explosive growth of the data. According to IBM Marketing Cloud study Cloud (2010), more than 90% of data on the internet has been created since 2016. In order to mitigate the local storage and computing pressure, an increasing number of individuals and organizations tend to store and process their

* Corresponding author.
E-mail addresses: yinfan519@gmail.com (F. Yin), rlu1@unb.ca (R. Lu), yzheng8@unb.ca (Y. Zheng), chn.junshao@gmail.com (J. Shao), yang.xue@sz.tsinghua.edu.cn (X. Yang), xhutang@swjtu.edu.cn (X. Tang).

https://doi.org/10.1016/j.cose.2021.102432
0167-4048/© 2021 Elsevier Ltd. All rights reserved.

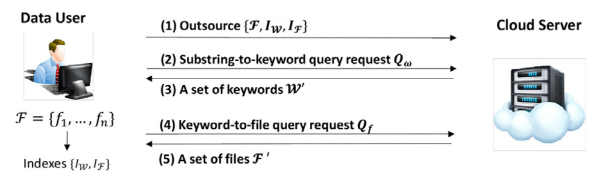
data in the cloud. However, since the cloud server may not be fully trustable, those data with some sensitive information (e.g., electronic health records) have to be encrypted before being outsourced to the cloud Zheng et al. (2019). Although the data encryption technique can preserve data privacy, it also hides some critical information such that the cloud server cannot well support some users' query functionality over the encrypted data, e.g., keyword query, which returns the collection of files containing some specific queried keywords. In order to address the challenge, the concept of symmetric searchable encryption (SSE) Song et al. (2000) was introduced, which enables the cloud server to search encrypted files in a very efficient way.

Over the past years, in order to improve the keyword query efficiency, a variant of secure keyword-based index techniques have been designed to match the keywords with corresponding files, such as inverted index Cash et al. (2014, 2013); Curtmola et al. (2006), tree-based index Goh et al. (2003); Shao et al. (2019); Yin et al. (2019), etc. Since the current keyword-based index techniques are built with exact keywords, the existing SSE schemes can only support exact keyword query, i.e., the queried keyword must be exactly the same keyword stored in cloud.

However, in practice, it is quite common that a user only remembers a fragment/substring of a keyword rather than the exact keyword and expects to achieve a substring-of-keyword query, i.e., *the user first queries some candidate keywords containing a substring to help him/her complete the queried keyword and then queries files that match the queried keyword*. Considering the Google website example, it automatically returns a list of candidate keywords after users enter a fragment of the queried keyword to the search bar. This feature can help users efficiently enter the correct queried keyword before a real search. Unfortunately, most SSE schemes with the current keyword-based index techniques cannot be directly used to support the substring-of-keyword query because their indexes do not contain the substring information. Although some SSE schemes Chase and Shen (2015); Hahn et al. (2018); Leontiadis and Li (2018); Mainardi et al. (2019) focus on the substring query and can be used to implement substring-of-keyword query, they cannot achieve high efficiency in terms of the computational cost of query processing and the overhead of storage at the same time.

To address the above challenge, in this paper, we consider a fine-grained SSE scheme supporting substring-of-keyword query, which consists of two consecutive phases. The first phase, called the substring-to-keyword query, is to query a list of candidate keywords containing a given specific substring, and then the user chooses the keyword that he/she needs from candidate keywords. The second phase, called the keyword-to-file query, is to query files that match the chosen keyword. Specifically, the main contributions of this paper are three-fold:

- First, based on the position heap technique, we design a storage-efficient index (i.e., modified position heap) to match substrings with corresponding keywords. We then use pseudo-random function and symmetric encryption scheme to encrypt this index, which can not only well support the substring-to-keyword query, but also preserve the privacy of queried substring as well as the plaintext of the keywords.
- Second, we proposed an efficient and privacy-preserving substring-of-keyword query scheme, which consists of a substring-to-keyword query and a keyword-to-file query. This scheme is suitable for critical applications in practice such as Google search.
- Finally, we analyze the security of our proposed scheme and conduct extensive experiments to evaluate its performance. The results show that our proposed scheme can achieve efficient queries in terms of low computational cost and communication overhead.


Data User
Cloud Server
(1) Outsource $\{\mathcal{F}, I_{\mathcal{W}}, I_{\mathcal{F}}\}$
(2) Substring-to-keyword query request Q_ω
(3) A set of keywords $\mathcal{W}'$
(4) Keyword-to-file query request Q_f
(5) A set of files $\mathcal{F}'$
$\mathcal{F} = \{f_1, \dots, f_n\}$
Indexes $\{I_{\mathcal{W}}, I_{\mathcal{F}}\}$
Fig. 1 – System model under consideration.

The remainder of the paper is organized as follows. We formalize the system model, security model, and design goals in Section 2. Then, we introduce some preliminaries including the position heap technique Ehrenfeucht et al. (2011), symmetric encryption scheme, and the security notion of substring-to-keyword query in Section 3. After that, we present our proposed scheme in Section 4, followed by security analyses and performance evaluation in Section 5 and Section 6, respectively. Some related works are discussed in Section 7. Finally, we draw our conclusions in Section 8.

2. Models and design goals

In this section, we formalize the system model, security model, and identify our design goals.

2.1. System model

In our system model, we consider two entities, namely a cloud server and a data user, as shown in Fig. 1.

- <u>Data user</u>: The data user has a collection of files $\mathcal{F} = \{f_1, f_2, \dots, f_n\}$ and each file $f_j \in \mathcal{F}$ consists of a set of keywords from a dictionary $\mathcal{W} = \{\omega_1, \omega_2, \dots, \omega_d\}$. Due to the limited storage space and computational capability, the data user intends to outsource the file collection $\mathcal{F}$ and its indices, i.e., $I_{\mathcal{W}}$ – index for substring-to-keyword query, $I_{\mathcal{F}}$ – index for keyword-to-file query, to the cloud server. Then, the data user launches a substring-of-keyword query with the cloud server. The substring-of-keyword query consists of two consecutive phases: a substring-to-keyword query and a keyword-to-file query. To be more specific, the data user first submits a substring-to-keyword query request Q_ω to the cloud server and retrieves a set of keywords $\mathcal{W}' \subseteq \mathcal{W}$ containing the given substring. Then, the data user chooses the queried keyword from $\mathcal{W}'$ and uses a queried keyword to submit a keyword-to-file query request Q_f to retrieve a set of files $\mathcal{F}' \subseteq \mathcal{F}$ containing the queried keyword.

- **Cloud server**: The cloud server is considered to be powerful in storage space and computational capability. The cloud server is assumed to efficiently store file collection $\mathcal{F}$ and its indices $\{I_{\mathcal{W}}, I_{\mathcal{F}}\}$ in local. In addition, the cloud server will process two types of query requests: substring-to-keyword query request Q_ω and keyword-to-file query request Q_f. For the former, the cloud server conducts search operation in the index $I_{\mathcal{W}}$ and responds a set of keywords $\mathcal{W}' \subseteq \mathcal{W}$; For the latter, the cloud server conducts search operation in the index $I_{\mathcal{F}}$ and responds a set of files $\mathcal{F}' \subseteq \mathcal{F}$.

2.2. *Security model*

In our security model, the data user is considered as trusted, while the cloud server is assumed as *honest-but-curious*, which means that the cloud server will i) honestly execute the query processing, return the query results without tampering it, and ii) curiously infer as much sensitive information as possible from the available data. The sensitive information could include the files $\mathcal{F}$, the indices $\{I_{\mathcal{W}}, I_{\mathcal{F}}\}$, the substring-to-keyword query request Q_ω, and the keyword-to-file query request Q_f. The formal simulated-based definition for this security model is described in Subsection 3.3.

2.3. *Design goals*

In this work, our design goal is to achieve an efficient and privacy-preserving substring-of-keyword query scheme. In particular, the following three requirements should be achieved.

- *Privacy preservation*. In the proposed scheme, all the data obtained by the cloud server, i.e., $\{\mathcal{F}, I_{\mathcal{W}}, I_{\mathcal{F}}, Q_\omega, Q_f\}$, should be privacy-preserving during the outsourcing, query, and update phases. Formally, the proposed scheme needs to satisfy security definition 1.
- *Efficiency*. In order to achieve the above privacy requirement, additional computational cost and storage overhead will inevitably be incurred. Therefore, in this work, we also aim to reduce the computational cost and communication overhead to be linear with the length of the queried substring.
- *Dynamics*. Update operations should be efficiently and securely supported after the initial outsourcing.

3. Preliminary

In this section, we recall some preliminaries including the position heap technique Ehrenfeucht et al. (2011), the symmetric key encryption scheme, and the security definition of substring-to-keyword query, which will be served as the basis of our proposed scheme.

3.1. *The (original) position heap technique*

Intuitively speaking, the (original) position heap $P(t)$ is a trie built from all the suffixes of t and can be used to achieve ef-

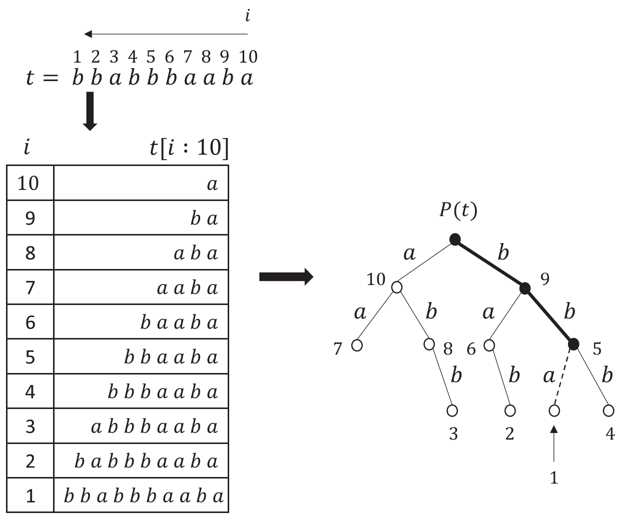


Fig. 2 – An example of building position heap P(t) for string t = *bbabbbaaba*. The solid edges in P(t) reflect the insertion path for suffix t[1 : 10].

ficient substring search for t. To construct the position heap $P(t)$ from a string $t = c_1c_2 \ldots c_m$, $P(t)$ is initialized as a root node and a set of suffixes $t[i : m] = c_i \ldots c_m$ ($i \in [m, \ldots, 1]$) are chosen and inserted to the $P(t)$. To do this, for each suffix $t[i : m]$ ($i \in [m, \ldots, 1]$), its longest prefix $t[i : j]$ ($i \leq j \leq m$) that is already represented by a path in $P(t)$ is found and a new leaf child is added to the last node of this path. The new leaf child is labeled with i and its edge is labeled with $t[j+1]$ (see Fig. 2). Compared to other data structures to achieve substring search, such as suffix tree Chase and Shen (2015) and suffix array Leontiadis and Li (2018), the position heap Ehrenfeucht et al. (2011) can achieve high efficiency in both storage and query time.

In the following, we formally describe the `PHBuild` and `PHSearch` algorithms of the position heap. Note that, we consider each node in the position heap stores two attributes: *edge* and *pos*, where the former stores the label of the node's edge and the latter stores the label of the node.

3.1.1. `PHBuild` *algorithm*

Given a string $t = c_1c_2 \ldots c_m$, the `PHBuild` (i.e., Algorithm 1) first initializes position heap $P(t)$ as a root node. Then, it visits the t from the right to left and inserts each suffix $t[i : m]$ ($i \in [m, \ldots, 1]$) to the position heap $P(t)$. In particular, for each suffix $t[i : m]$, the algorithm first finds its longest prefix $t[i : j]$ ($i \leq j \leq m$) that is already represented by a path in $P(t)$ (lines 4–10). Assume the last node of this path is N. Then the algorithm appends a new leaf child N' to the N, where $N'.edge = c_{j+1}$ and $N'.pos = i$ (lines 11–12). Fig. 2 depicts an example to build such a position heap for a string $t = bbabbbaaba$. During the insertion for suffix $t[1 : 10]$, this algorithm finds its longest prefix $t[1 : 2]$ represented by the solid path and appends a new leaf child N' to the last node of the solid path, where $N'.edge = a$ and $N'.pos = 1$.

Algorithm 1 Build a position heap $P(t)$ for the string $t = c_1c_2\dots c_m$.

```
1: initialize position heap $P(t)$ as a root node $R$, where $R.edge = Null$ and $R.pos = Null$;
2: for each $i$ in $[m, m-1, \dots, 1]$ do
3:     $N = R$;
4:     for each $j$ in $[i, i+1, \dots, m]$ do
5:         find the child $N\prime$ of $N$, where $N\prime.edge = c_j$;
6:         if $N\prime$ does exist then
7:             $N = N\prime$
8:         else
9:             $j = j - 1$;
10:            break;
11:    insert a new child node $N\prime$ to the $N$;
12:    $N\prime.edge = c_{j+1}, N\prime.pos = i$;
13: return $P(t)$;
```

3.1.2. PHSearch *algorithm*

Given a substring s and a position heap $P(t)$, the PHSearch (i.e., Algorithm 2) is supposed to find all the positions in t that are occurrences of s. The time complexity of this algorithm is $O(|s|^2 + d_s)$, where $|s|$ is the length of the queried substring and d_s is the number of matching positions. The details are as follows:

Algorithm 2 Search substring s in a position heap $P(t)$, where $s = s_1s_2\dots s_l$ and $t = c_1c_2\dots c_m$.

```
1: initial empty sets $L_1$ and $L_2$;
2: let $N$ be the root node of the $P(t)$;
3: for each $i$ in $[1, 2, \dots, l]$ do
4:     find the child $N\prime$ of $N$, where $N\prime.edge = s_i$;
5:     if $N\prime$ does exist then
6:         if $i = l$ then
7:             $L_2.add(N\prime.pos)$;
8:             for each descendant $X$ of $N\prime$ do
9:                 $L_2.add(X.pos)$;
10:        else
11:            $L_1.add(N\prime.pos)$;
12:        $N = N\prime$;
13:    else
14:        break;
15: for each $i$ in $L_1$ do
16:    if $c_ic_{i+1}\dots c_{i+l-1}$ is not equal to $s_1s_2\dots s_l$ then
17:        $L_1.remove(i)$;
18: return $L_1 \cup L_2$;
```

- The algorithm first finds the longest prefix s' of s that can be represented by a path in $P(t)$ and then denotes this path as the search path. Next, the algorithm lets L_1 be the set of *pos* stored in the intermediate nodes along the search path and L_2 be the set of *pos* stored in the descendants of the last node of the search path (lines 3–14). In particular, if $s' \neq s$, the *pos* stored in the last node of the search path is included in L_1. Otherwise, it is included in L_2.
- After completing the previous step, elements in L_2 must be matching positions but elements in L_1 may or may not be matching positions. Therefore, the algorithm reviews each position $i \in L_1$ in the string t to filter out unmatching positions and remove them from the L_1. Finally, this algorithm returns $L_1 \cup L_2$ (lines 15–17).

Take an example with Fig. 2. Given a substring $s = bb$, the PHSearch algorithm finds its longest prefix bb corresponding to the solid path. In this way, L_1 and L_2 are equal to $\{9\}$ and $\{5, 1, 4\}$. Then, this algorithm reviews the string t and confirms $i = 9 \in L_1$ is not an occurrence of s. Therefore, the position 9 is removed from the L_1, and L_1 is an empty set now. Finally, this algorithm returns all the *pos* in $L_1 \cup L_2 = \{5, 1, 4\}$.

3.2. *Symmetric key encryption scheme*

A symmetric key encryption scheme (SKE) Katz and Lindell (2014) is a set of three polynomial-time algorithms (Gen, Enc, Dec) such that Gen takes a security parameter λ and returns a secret key k; Enc takes a key k and a message m, then returns a ciphertext c; Dec takes a key k and a ciphertext c, then returns m if k was the key under which c was produced. In this work, we consider a SKE is indistinguishable under chosen plaintext attack (IND-CPA) Katz and Lindell (2014), which guarantees the ciphertext does not leak any information about the plaintext even an adversary can query an encryption oracle. We note that common private-key encryption schemes such as AES in counter mode satisfy this definition.

3.3. *Security definition of substring-to-keyword query*

In this subsection, we follow the security definition in Curtmola et al. (2006) to formalize the simulated-based security definition of substring-to-keyword query scheme by using the following two experiments: $Real_{\mathcal{A},\mathcal{C}}(\lambda)$ and $Ideal_{\mathcal{A},\mathcal{S}}(\lambda)$. In the former, the adversary $\mathcal{A}$, who represents the cloud server, executes the proposed scheme with a challenger $\mathcal{C}$ that represents the data user. In the latter, $\mathcal{A}$ also executes the proposed scheme with a simulator $\mathcal{S}$ who simulates the output of the challenger $\mathcal{C}$ through the leakage of the proposed scheme. The leakage is parameterized by a leakage function collection $\mathcal{L} = (\mathcal{L}_O, \mathcal{L}_Q, \mathcal{L}_U)$, which describes the information leaked to the adversary $\mathcal{A}$ in the outsourcing phase, query phase, and update phase respectively. If any polynomial adversary $\mathcal{A}$ cannot distinguish the outputs between the challenger $\mathcal{C}$ and the simulator $\mathcal{S}$, then we can say there is no other information leaked to the adversary $\mathcal{A}$, i.e., the cloud server, except the information that can be inferred from the $\mathcal{L}$. More formally,

- $Real_{\mathcal{A},\mathcal{C}}(1^\lambda) \to b \in \{0, 1\}$: Given a keyword dictionary $\mathcal{W}$ chosen by the adversary $\mathcal{A}$, the challenger $\mathcal{C}$ outputs encrypted index $I_{\mathcal{W}}$ by following the outsourcing phase of the proposed scheme. Then, $\mathcal{A}$ can adaptively send a polynomial number of query requests (or update requests) to the $\mathcal{C}$, who outputs corresponding encrypted query requests (or encrypted update requests). Eventually, $\mathcal{A}$ returns a bit b as the output of this experiment.
- $Ideal_{\mathcal{A},\mathcal{S}}(1^\lambda) \to b \in \{0, 1\}$: Given the leakage function $\mathcal{L}_O$, the simulator outputs simulated encrypted index $\overline{I_{\mathcal{W}}}$. Then, for each query request (or update request), the adversary $\mathcal{A}$ sends its leakage function $\mathcal{L}_Q$ (or $\mathcal{L}_U$) to the simulator $\mathcal{S}$, who generates the corresponding simulated encrypted

query request (or encrypted update request). Eventually, $\mathcal{A}$ returns a bit b as the output of this experiment.

Definition 1. A substring-to-keyword query scheme is $\mathcal{L}$-secure against adaptive attacks if for any probabilistic polynomial time adversary $\mathcal{A}$, there exists an efficient simulator $\mathcal{S}$ such that

$$|\Pr[Real_{\mathcal{A},\mathcal{L}}(\lambda) \to 1] - \Pr[Ideal_{\mathcal{A},\mathcal{S},\mathcal{L}}(\lambda) \to 1]| \le negl(\lambda).$$

4. Our proposed scheme

In this section, we will present our substring-of-keyword query scheme. Before delving into the details, we first introduce a modified position heap for keyword dictionaries, which is a basic building block of our proposed scheme.

4.1. *Modified position heap for keyword dictionaries*

In order to process substring-to-keyword query efficiently, we first design a modified position heap to index all the keywords in a dictionary, which consists of two algorithms: i) `MPHBuild` algorithm; ii) `MPHSearch` algorithm.

4.1.1. `MPHBuild` *algorithm*

Given a dictionary $\mathcal{W} = \{\omega_1, \omega_2, \dots, \omega_d\}$, the `MPHBuild` algorithm first transforms the dictionary $\mathcal{W} = \{\omega_1, \omega_2, \dots, \omega_d\}$ to a string $t_{\mathcal{W}} = \omega_1||\#||\omega_2||\#\dots\#||\omega_d$, where $||$ denotes the concatenation operation and # denotes a separate character that does not appear in any $\omega \in \mathcal{W}$. In the rest of this paper, we call this string $t_{\mathcal{W}}$ *dictionary string*. Then, this algorithm follows `PHBuild` algorithm (i.e., Algorithm 1) to build an original position heap for this dictionary string $t_{\mathcal{W}}$ and further modifies it to a modified position heap $P(t_{\mathcal{W}})$ as follows: i) For each node N, replace its $N.pos$ with the corresponding keyword in $t_{\mathcal{W}}$, called $N.keyword$. ii) At the same time, remove useless paths, whose edges starting with #. Fig. 3 depicts an example of building the modified position heap $P(t_{\mathcal{W}})$ for a dictionary $\mathcal{W} = \{\omega_1, \omega_2, \omega_3\}$.

4.1.2. `MPHSearch` *algorithm*

Given a substring s and a modified position heap $P(t_{\mathcal{W}})$, the `MPHSearch` algorithm follows the `PHSearch` algorithm (i.e., Algorithm 2) to return all the keywords in $\mathcal{W}$ that include s. There are two differences between `PHSearch` and `MPHSearch`: i) `PHSearch` returns a set of positions, but `MPHSearch` returns a set of keywords because all the $N.pos$ stored in $P(t_{\mathcal{W}})$ is replaced by the corresponding $N.keyword$. ii) `PHSearch` reviews each position $i \in L_1$ in the string t to filter out unmatching positions, but `MPHSearch` directly returns all the keywords in L_1. The reason is that the cloud server, who performs `MPHSearch` algorithm, is not allowed to access to the dictionary string $t_{\mathcal{W}}$ to filter out unmatching keywords in L_1. Therefore, the cloud server returns all the keywords in L_1 and leaves the filter operation to the data user. Fortunately, the computational cost of the filter operation, i.e., $O(|s|^2)$, is acceptable because the length of queried substring, i.e., $|s|$, is relatively small in practice.

4.2. *Description of our proposed scheme*

In this subsection, we will describe our proposed privacy-preserving substring-of-keyword query scheme, which consists of five phases: i) System Initialization; ii) Data Outsourcing; iii) Substring-of-keyword Query; iv) Update (Insertion); and v) Update (Deletion).

4.2.1. *System initialization*

Given a security parameter λ, the data user first initializes a secure pseudo-random function (PRF) $H_{k_1} : \{0,1\}^* \longrightarrow \{0,1\}^{\gamma}$, where k_1 is a λ-bit random key. Then, the data user initializes an IND-CPA secure SKE $\Pi = (Gen, Enc, Dec)$ and generates a secret key $k_2 = \Pi.Gen(1^{\lambda})$.

4.2.2. *Data outsourcing*

Assume the data user has a file collection $\mathcal{F} = \{f_1, f_2, \dots, f_n\}$, where each $f_j \in \mathcal{F}$ includes a set of keywords $\mathcal{W}_j \subseteq \mathcal{W}$. The data user generates secure indices $\{I_{\mathcal{W}}, I_{\mathcal{F}}\}$ and a set of encrypted files in the following steps:

Step 1: In order to support efficient substring-to-keyword query, the data user uses the `MPHBuild` algorithm, described in Section 4.1, to build a modified position heap $P(t_{\mathcal{W}})$ for the dictionary $\mathcal{W}$.

Step 2: For privacy, the data user encrypts $P(t_{\mathcal{W}})$ to a secure index $I_{\mathcal{W}}$ as follows:

- For each node N in the modified position heap (except the root), the data user uses $\Pi.Enc_{k_2}$ to encrypt its $N.keyword$.
- For each node N in the modified position heap (except the root), the data user concatenates each edge label, i.e., $N.edge$, along the path from the root to this node, and calculates the PRF output of the concatenation through H_{k_1}.

Considering the example in Fig. 4, the $I_{\mathcal{W}}$ is encrypted from Fig. 3(d). For each node, its keywords are encrypted through $\Pi.Enc_{k_2}$, and its edge label are transformed to a PRF output through H_{k_1}.

Step 3: In order to support efficient keyword-to-file query, the data user utilizes the inverted index proposed in Cash et al. (2014) to implement the index $I_{\mathcal{F}}$. This inverted index is implemented by a hash table, and each $< key, value >$ pair in it is the form of $< \omega, id >$, where ω is a keyword and id is a file identifier.

Step 4: Finally, the data user encrypts each file $f_j \in \mathcal{F}$ through $\Pi.Enc_{k_2}$ and sends these encrypted files to the cloud server with secure indices $\{I_{\mathcal{W}}, I_{\mathcal{F}}\}$.

4.2.3. *Substring-of-keyword query*

Given a substring $s = s_1 s_2 \dots s_l$, the data user launches a substring-of-keyword query with the cloud server. The substring-of-keyword query consists of two consecutive phases: a substring-to-keyword query and a keyword-to-file query, which are described in the following steps:

Step 1: First, the data user generates a substring-to-keyword query request Q_{ω} and submits it to the cloud server. To be more specific, the $Q_{\omega} = \{Q_1, Q_2, \dots, Q_l\}$ consists of l PRF outputs, where

$$Q_i = H_{k_1}(s_1||\dots||s_i),\ 1 \le i \le l \quad (1)$$

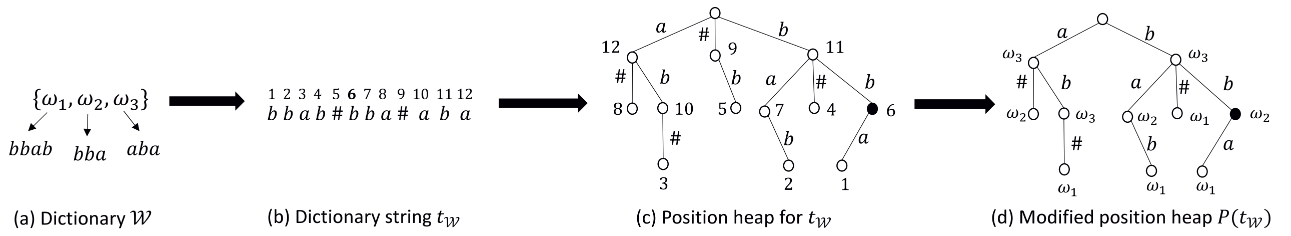

$\{\omega_1,\omega_2,\omega_3\}$
bbab bba aba
(a) Dictionary $\mathcal{W}$
1 2 3 4 5 6 7 8 9 10 11 12
b b a b # b b a # a b a
(b) Dictionary string $t_{\mathcal{W}}$
(c) Position heap for $t_{\mathcal{W}}$
(d) Modified position heap $P(t_{\mathcal{W}})$

Fig. 3 – An example of building a modified position heap for a dictionary $\mathcal{W}$. (a) $\mathcal{W} = \{\omega_1, \omega_2, \omega_3\}$ is a dictionary, where $\omega_1 = bbab$, $\omega_2 = bba$, and $\omega_3 = aba$. (b) To get dictionary string $t_{\mathcal{W}}$, concatenate all the keywords in $\mathcal{W}$ with character #. (c) Build an original position heap for $t_{\mathcal{W}}$. (d) For each node N, replace its N.*pos* with the corresponding keyword, called N.*keyword*. At the same time, remove useless paths from the $P(t_{\mathcal{W}})$.

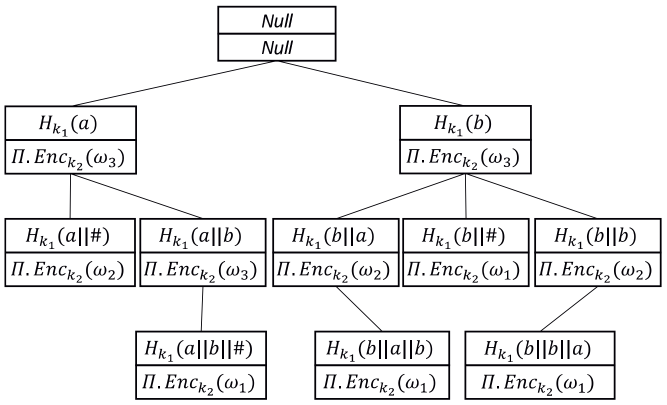

Null
Null
$H_{k_1}(a)$ $\Pi.Enc_{k_2}(\omega_3)$
$H_{k_1}(b)$ $\Pi.Enc_{k_2}(\omega_3)$
$H_{k_1}(a||\#)$ $\Pi.Enc_{k_2}(\omega_2)$
$H_{k_1}(a||b)$ $\Pi.Enc_{k_2}(\omega_3)$
$H_{k_1}(b||a)$ $\Pi.Enc_{k_2}(\omega_2)$
$H_{k_1}(b||\#)$ $\Pi.Enc_{k_2}(\omega_1)$
$H_{k_1}(b||b)$ $\Pi.Enc_{k_2}(\omega_2)$
$H_{k_1}(a||b||\#)$ $\Pi.Enc_{k_2}(\omega_1)$
$H_{k_1}(b||a||b)$ $\Pi.Enc_{k_2}(\omega_1)$
$H_{k_1}(b||b||a)$ $\Pi.Enc_{k_2}(\omega_1)$

Fig. 4 – An example of secure index $I_{\mathcal{W}}$, which is generated from the modified position heap $P(t_{\mathcal{W}})$ in Fig. 3(d).

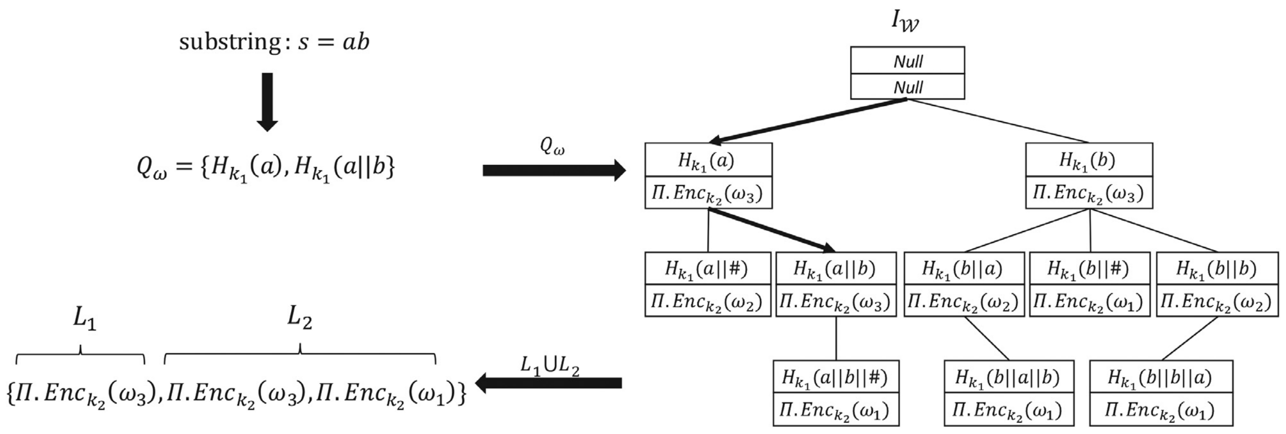

substring: $s = ab$
$Q_\omega = \{H_{k_1}(a), H_{k_1}(a||b)\}$
Q_ω
$I_{\mathcal{W}}$
L_1
L_2
$L_1 \cup L_2$
$\{\Pi.Enc_{k_2}(\omega_3), \Pi.Enc_{k_2}(\omega_3), \Pi.Enc_{k_2}(\omega_1)\}$

Fig. 5 – An example of substring-to-keyword query, where $I_{\mathcal{W}}$ is the secure index for dictionary string $t_{\mathcal{W}} = bbab\#bba\#aba$ and the given substring is $s = ab$.

Step 2: After receiving the query request Q_ω, the cloud server follows the `MPHSearch` algorithm, described in Section 4.1, to search encrypted keywords in secure index $I_{\mathcal{W}}$ and returns elements in $L_1 \cup L_2$ to the data user. Fig. 5 depicts an example of substring-to-keyword query, where the given substring is $s = ab$. In this example, the data user generates $Q_\omega = \{H_{k_1}(a), H_{k_1}(a||b)\}$ and sends it to the cloud server. After receiving the Q_ω, the cloud server performs `MPHSearch` to get $L_1 = \{\Pi.Enc_{k_2}(\omega_3)\}$, $L_2 = \{\Pi.Enc_{k_2}(\omega_3), \Pi.Enc_{k_2}(\omega_1)\}$ and returns $L_1 \cup L_2$ to the data user. Note that, since $\Pi.Enc$ is a randomized encryption, these encrypted keywords in $L_1 \cup L_2$ are indistinguishable for the cloud server.

Step 3: After receiving the encrypted keywords, the data user first decrypts them and filters out the unmatching keywords. Then, the data user chooses a queried keyword from the matching keywords and submits a keyword-to-file query to the cloud server. Since our paper just focuses on the design of substring-to-keyword query, we directly utilize the scheme

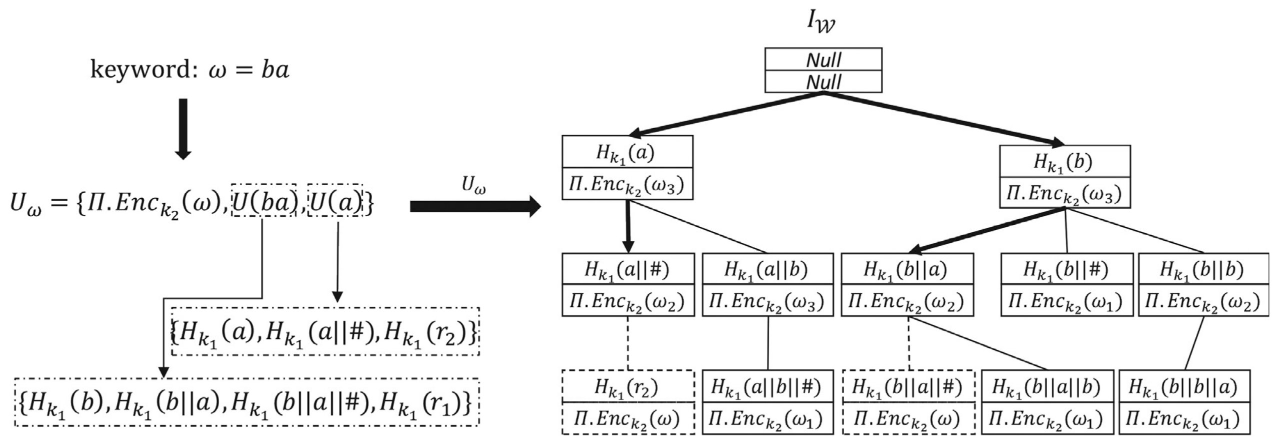

keyword: $\omega = ba$
$U_\omega = \{\Pi.Enc_{k_2}(\omega), U(ba), U(a)\}$
U_ω
$\{H_{k_1}(a), H_{k_1}(a||\#), H_{k_1}(r_2)\}$
$\{H_{k_1}(b), H_{k_1}(b||a), H_{k_1}(b||a||\#), H_{k_1}(r_1)\}$
$I_\mathcal{W}$
Null
Null
$H_{k_1}(a)$
$\Pi.Enc_{k_2}(\omega_3)$
$H_{k_1}(b)$
$\Pi.Enc_{k_2}(\omega_3)$
$H_{k_1}(a||\#)$
$\Pi.Enc_{k_2}(\omega_2)$
$H_{k_1}(a||b)$
$\Pi.Enc_{k_2}(\omega_3)$
$H_{k_1}(b||a)$
$\Pi.Enc_{k_2}(\omega_2)$
$H_{k_1}(b||\#)$
$\Pi.Enc_{k_2}(\omega_1)$
$H_{k_1}(b||b)$
$\Pi.Enc_{k_2}(\omega_2)$
$H_{k_1}(r_2)$
$\Pi.Enc_{k_2}(\omega)$
$H_{k_1}(a||b||\#)$
$\Pi.Enc_{k_2}(\omega_1)$
$H_{k_1}(b||a||\#)$
$\Pi.Enc_{k_2}(\omega)$
$H_{k_1}(b||a||b)$
$\Pi.Enc_{k_2}(\omega_1)$
$H_{k_1}(b||b||a)$
$\Pi.Enc_{k_2}(\omega_1)$

Fig. 6 – An example of inserting keyword $\omega = ba$ to $I_\mathcal{W}$, where $I_\mathcal{W}$ is the secure index for dictionary string $t_\mathcal{W} = bbab\#bba\#aba$.

proposed in Cash et al. (2014) to implement our keyword-to-file query. According to the scheme in Cash et al. (2014), the data user can submit efficient and privacy-preserving single keyword queries to the cloud server based on the index $I_\mathcal{F}$.

4.2.4. *Update (insertion)*

In the proposed scheme, there are two types of insertion operations: insert keywords to the index $I_\mathcal{W}$ and insert files to the index $I_\mathcal{F}$. Since Cash et al. Cash et al. (2014) has proposed a privacy-preserving insertion algorithm to deal with the insertion for the index $I_\mathcal{F}$, we just need to consider the insertion for the index $I_\mathcal{W}$.

Given a keyword $\omega = c_1c_2\dots c_z$, the data user is supposed to insert it to the index $I_\mathcal{W}$. Intuitively, assume the dictionary is $W = \{\omega_1, \omega_2, \dots, \omega_d\}$ and its corresponding dictionary string is $t_\mathcal{W} = \omega_1||\#||\omega_2||\#\dots\#||\omega_d$. The insertion operation will update the index $I_\mathcal{W}$ to a new version, called $I_{\mathcal{W}'}$, where its corresponding $t_{\mathcal{W}'} = \omega||\#||t_\mathcal{W}$. The details are described as follows:

Step 1: The data user chooses z random values $r_1, r_2, \dots, r_z$ to generate an update (insertion) request $U_\omega = \{\Pi.Enc_{k_2}(\omega), U(c_1c_2\dots c_z), U(c_2\dots c_z), \dots, U(c_z)\}$ and submits it to the cloud server. Specifically, each $U(c_i\dots c_z)$ $(1 \le i \le z)$ in U_ω consists of $(z - i + 3)$ PRF outputs, i.e., $\{U_i, U_{i+1}, \dots, U_{z+2}\}$, where

$$U_j = \begin{cases} H_{k_1}(c_i||\dots||c_j), & if\ i \le j \le z \\ H_{k_1}(c_i||\dots||c_j||\#), & if\ j = z+1 \\ H_{k_1}(r_i), & if\ j = z+2 \end{cases} \tag{2}$$

Fig. 6 depicts an example of the insertion operation for keyword $\omega = ba$. In this example, the $U_\omega = \{\Pi.Enc_{k_2}(\omega), U(ba), U(a)\}$, where $U(ba) = \{H_{k_1}(b), H_{k_1}(b||a), H_{k_1}(b||a||\#), H_{k_1}(r_1)\}$ and $U(a) = \{H_{k_1}(a), H_{k_1}(a||\#), H_{k_1}(r_2)\}$.

Step 2: After receiving the update (insertion) request U_ω, for each $U(c_i\dots c_z)$ $(1 \le i \le z)$ in it, the cloud server first finds its longest prefix $U_iU_{i+1}\dots U_h (1 \le h < z+2)$ that is already represented by a path in $I_\mathcal{W}$ and denotes this path as insertion path. Then the cloud server appends a new leaf child N' to the last node of the insertion path, where $N'.edge = U_{h+1}$ and $N'.keyword = \Pi.Enc_{k_2}(\omega)$. Note that, in practice, the h can not equal to $z+2$ because $U_{z+2} = H_{k_1}(r_i)$ is a random number. As shown in Fig. 6, the solid edges reflect the insertion paths for the $U(ba)$ and $U(a)$.

4.2.5. *Update (deletion)*

In the proposed scheme, there are two types of deletion operations: delete substrings from the index $I_\mathcal{F}$ and delete keywords from the index $I_\mathcal{W}$. Since Cash et al. Cash et al. (2014) has proposed a privacy-preserving deletion algorithm for the index $I_\mathcal{F}$, we just need to consider the deletion for the index $I_\mathcal{W}$.

We implement this deletion operation by maintaining a revocation list $I_{\mathcal{W}_r}$, which is also an encrypted modified position heap, in the cloud server. Specifically, in the data outsourcing phase, the data user builds a modified position heap $I_{\mathcal{W}_r}$ for an empty dictionary $\mathcal{W}_r = \{\}$ and sends the $I_{\mathcal{W}_r}$ to the cloud server with $\{I_\mathcal{W}, I_\mathcal{F}\}$. Then, to delete a keyword from the cloud server, the data user just follows the update (insertion) method in 4.2.4 to insert the keyword to $I_{\mathcal{W}_r}$. During a substring-to-keyword query, after receiveing a substring-to-keyword query request, the cloud server performs search operations over $I_\mathcal{W}$ and $I_{\mathcal{W}_r}$ separately, and returns two result sets to the data user. Finally, the data user decrypts the two result sets and calculates the difference between them to obtain the correct keywords.

Correctness. The correctness of our proposed is quite straightforward. The only issue is the collision among the edges' PRF outputs in $I_\mathcal{W}$. Since the domain size of PRF H_{k_1} is 2^γ, assuming that the number of nodes in $I_\mathcal{W}$ is m, the probability of collision is $O(\binom{m}{2}/2^\gamma) = O(m^2/2^\gamma)$. So we need to choose $\gamma = \lambda + 2log(m)$ such that $O(m^2/2^\gamma) = O(1/2^\lambda)$ is negligible over the security parameter λ.

5. Security analysis

In this paper, the proposed substring-of-keyword query scheme consists of two query schemes: a substring-to-keyword query scheme and a keyword-to-file query scheme. Since the security analysis in Cash et al. (2014) has shown that the keyword-to-file query scheme is secure, we mainly focus on the security analysis of the substring-to-keyword query scheme in this section.

5.1. *Leakage function collection*

The leakage function collection $\mathcal{L}$ consists of three leakage functions: $\mathcal{L}_O$, $\mathcal{L}_Q$, and $\mathcal{L}_U$. Before defining them, we first give some definitions for the leakage of this scheme.

Definition 2. (Query Path Pattern) Given the index $I_{\mathcal{W}}$, which contains a set of encrypted nodes $\{n_1, n_2, \ldots, n_m\}$, and a query request Q_ω, the *Query Path Pattern* reveals a set of identifiers of nodes in $I_{\mathcal{W}}$ that are reached by the Q_ω, i.e., nodes in the search path. Note that, in the proposed scheme the *Query Path Pattern* includes the *Access Pattern* (i.e., search result) and *Search Pattern* (i.e., which queries have the same queried substring).

Definition 3. (Insertion Path Pattern) Given the index $I_{\mathcal{W}}$, which contains a set of encrypted nodes $\{n_1, n_2, \ldots, n_m\}$, and an update (insertion) request U_ω, the *Insertion Path Pattern* reveals the set of identifiers of nodes in $I_{\mathcal{W}}$ that are reached by the U_ω, i.e., nodes in the insertion path.

Definition 4. (Deletion Path Pattern) The deletion method is implemented by a revocation list, which means the update (deletion) request is exactly the same as the update (insertion) request. Therefore, given the revocation list $I_{\mathcal{W}_r}$, which contains a set of encrypted nodes $\{n_1, n_2, \ldots, n_m\}$, and an update (deletion) request U_ω, the *Deletion Path Pattern* reveals the set of identifiers of nodes in $I_{\mathcal{W}_r}$ that are reached by the U_ω.

Now we define the leakage functions to capture the information leakage in different phases.

- Outsourcing Phase: Given the index $I_{\mathcal{W}}$, which contains a set of encrypted nodes $\{n_1, n_2, \ldots, n_m\}$. The leakage $\mathcal{L}_O$ consists of the following information:
  - m: the size of the dictionary string $t_{\mathcal{W}}$.
  - $\Gamma = \{(id_1, C_{id_1}), \ldots, (id_m, C_{id_m})\}$: the structure of index $I_{\mathcal{W}}$, where $id_i (1 \le i \le m)$ denotes the identifiers of encrypted node n_i and $C_{id_i} (1 \le i \le m)$ denotes all the identifiers of id_i's children.
- Query Phase: Given the index $I_{\mathcal{W}}$ and a substring-to-keyword query request Q_ω, the leakage $\mathcal{L}_Q$ is *Query Path Pattern*.
- Update Phase: Given the index $I_{\mathcal{W}}$, revocation list $I_{\mathcal{W}_r}$, and an update request U_ω, if update operation is insertion / deletion, the leakage $\mathcal{L}_U$ is *Insertion Path Pattern* / *Deletion Path Pattern*.

5.2. *Security proof*

We now prove the security of the substring-to-keyword query scheme based on the leakage function collection $\mathcal{L} = \{\mathcal{L}_O, \mathcal{L}_Q, \mathcal{L}_U\}$. Intuitively, we first define a simulator $\mathcal{S}$ based on the leakage function collection $\mathcal{L}$ and then analyze the indistinguishability between the outputs of the $\mathcal{S}$ in the ideal world and the challenger $\mathcal{C}$ (i.e., the data user) in the real world. Finally, we conclude that the proposed substring-to-keyword query scheme does not reveal any information beyond the leakage function collection $\mathcal{L}$ to the server. The details are as follows.

Theorem 1. *If the H is a secure pseudo-random function (PRF) and Π is an IND-CPA secure symmetric key encryption scheme (SKE), then our proposed scheme is $\mathcal{L}$-adaptively-secure.*

Proof. Based on the leakage function collection $\mathcal{L}$, we can build a simulator $\mathcal{S}$ as follows:

- Data Outsourcing: given the leakage function $\mathcal{L}_O = \{m, \Gamma\}$, the simulator $\mathcal{S}$ is supposed to generate a simulated $\overline{I_{\mathcal{W}}}$ (i.e., an encrypted modified position heap). Specifically, the simulator $\mathcal{S}$ first generates m empty nodes and identifies each node a unique identifier from $\{id_1, \ldots, id_m\}$. Then the simulator $\mathcal{S}$ constructs these nodes to a tree (i.e., $\overline{I_{\mathcal{W}}}$) based on Γ, which means the $\overline{I_{\mathcal{W}}}$ has the same tree structure as $I_{\mathcal{W}}$. Next, for each node in the $\overline{I_{\mathcal{W}}}$, the simulator $\mathcal{S}$ chooses a random number $\overline{H}$ from the domain of H as the PRF output of its edge and a random number $\overline{\Pi.Enc}$ from the domain of $\Pi.Enc$ as its encrypted keyword. Since the output of H and $\Pi.Enc$ are pseudo-random, the adversary $\mathcal{A}$ cannot distinguish between the $\overline{I_{\mathcal{W}}}$ in the ideal world and the $I_{\mathcal{W}}$ in the real world.
- Substring-to-keyword Query: given the leakage function $\mathcal{L}_Q$ for a substring-to-keyword query request Q_ω, the simulator $\mathcal{S}$ is supposed to generate a simulated encrypted substring-to-keyword query request $\overline{Q_\omega}$. Note that, in this phase, the simulator $\mathcal{S}$ not only has $\mathcal{L}_Q$ but also $\mathcal{L}_O$ and $\overline{I_{\mathcal{W}}}$ from the data outsourcing phase. Therefore, the simulator $\mathcal{S}$ can follow the *Query Path Pattern* in $\mathcal{L}_Q$ to find the search path in $\overline{I_{\mathcal{W}}}$ and output all the $\overline{H}$ stored in the search path as the $\overline{Q_\omega}$. Since the output of H is pseudo-random, the adversary $\mathcal{A}$ cannot distinguish between the elements in $\overline{Q_\omega}$ and Q_ω. At the same time, after receiving the $\overline{Q_\omega}$, the adversary $\mathcal{A}$ can use it to find matching encrypted keywords in $\overline{I_{\mathcal{W}}}$. Since these matching encrypted keywords in $\overline{I_{\mathcal{W}}}$ is encrypted through $\Pi.Enc$, the adversary $\mathcal{A}$ cannot distinguish them from the matching encrypted keywords in $I_{\mathcal{W}}$, which means the adversary $\mathcal{A}$ cannot distinguish between $\overline{Q_\omega}$ in the ideal world and Q_ω in the real world.
- Update: given the leakage function $\mathcal{L}_U$ for an update (insertion / deletion) request U_ω, the simulator $\mathcal{S}$ is supposed to generate a simulated encrypted update request $\overline{U_\omega}$. Note that, in this phase, the simulator $\mathcal{S}$ not only has $\mathcal{L}_U$ but also $\mathcal{L}_O$ and $\overline{I_{\mathcal{W}}}$ / $\overline{I_{\mathcal{W}_r}}$ from the data outsourcing phase. Therefore, the simulator $\mathcal{S}$ can follow the *Insertion Path Pattern* / *Deletion Path Pattern* in $\mathcal{L}_U$ to find the insertion paths in $\overline{I_{\mathcal{W}}}$ / $\overline{I_{\mathcal{W}_r}}$ and output all the $\overline{H}$ stored in these insertion paths as the $\overline{U_\omega}$. Since the output of H is pseudo-random, the adversary $\mathcal{A}$ cannot distinguish between $\overline{U_\omega}$ in the ideal world and U_ω in the real world.

In summary, as the adversary $\mathcal{A}$ cannot distinguish between the outputs from the simulator $\mathcal{S}$ and the challenger $\mathcal{C}$, we can conclude that our proposed substring-to-keyword query scheme is $\mathcal{L}$-adaptively-secure. □

6. Performance evaluation

In this section, we evaluate the performance of our proposed scheme from both theoretical and experimental perspectives.

Table 1 – Comparison between ours and existing schemes.

Scheme	Index Space	Query Time	Dynamism
Chase and Shen (2015)	$O(m)$	$O(\lvert s\rvert + d_s)$	static
Leontiadis and Li (2018)	$O(m)$	$O(\lvert s\rvert + d_s)$	static
Hahn et al. (2018)	$O(m)$	$O(\lvert s\rvert \cdot d_{\overline{kg}})$	dynamic
Moataz et al. (2018)	$O(m)$	$O(m)$	static
Mainardi et al. (2019)	$O(\lvert\Sigma\rvert \cdot m)$	$O(\lvert s\rvert + d_s)$	static
Our solution	$O(m)$	$O(\lvert s\rvert + d_s)$	dynamic

m is the size of dataset, $|s|$ is the size of queried substring s, d_s is the number of matching positions for s, $d_{\overline{kg}}$ is the average number of matching positions for a k-gram of s, and $|\Sigma|$ is the number of distinct characters in dataset.

6.1. *Theoretical analysis*

We perform a theoretical comparison of our proposed substring-to-keyword scheme with existing schemes (cf. Table 1) from three aspects: index space, query time, and dynamism. From Table 1, we can see that the schemes in Chase and Shen (2015) and Leontiadis and Li (2018) have the same index space (i.e., $O(m)$) and query time (i.e, $O(|s| + d_s)$). However, in practice, Chase and Shen (2015) will consume more index space than Leontiadis and Li (2018) due to its suffix tree index, which just stores position data in leaf nodes and does not utilize the space of inner nodes effectively. In fact, the number of nodes in the suffix tree can be up to $2m$, where m is the size of the dataset. In contrast, Leontiadis and Li (2018) utilizes Burrows-Wheeler Transformation (BWT) to build a suffix array index to support substring query, which has better storage-efficiency than the suffix tree at the cost of worse query-efficiency.

Later, based on the scheme in Leontiadis and Li (2018), Mainardi et al. (2019) uses Private Information Retrieval (PIR) technique to protect the access pattern, which causes high index space and query time. In addition to suffix tree and suffix array, there are other auxiliary data structures Hahn et al. (2018); Moataz et al. (2018) can be used to support substring query. However, their query time is unacceptable in practice.

Compared with these existing schemes, our proposed substring-to-keyword scheme can achieve high storage-efficiency and query-efficiency at the same time. In specific, our scheme can achieve $O(m)$ complexity for index space and $O(|s| + d_s)$ complexity for query time, which are the same as Chase and Shen (2015); Leontiadis and Li (2018) and better than Hahn et al. (2018); Mainardi et al. (2019); Moataz et al. (2018). In addition, our proposed scheme can support dynamic datasets, which cannot be supported by Chase and Shen (2015); Leontiadis and Li (2018). Further, due to the use of position heap technique, which is storage-efficient than the suffix tree and query-efficient than the suffix array, our proposed scheme consumes less index space than Chase and Shen (2015) and less query time than Leontiadis and Li (2018) in practice, which will be shown in the next subsection.

6.2. *Experimental analysis*

In this subsection, we evaluate the computational cost and storage overhead of the proposed substring-to-keyword scheme in terms of three phases: local data outsourcing, substring-to-keyword query, and update. Specifically, we implemented the proposed scheme in C++ (our code is open source Yin (2019)) and conducted experiments on a 64-bit machine with an Intel Core i5-8400 CPU at 2.8GHZ and 2GB RAM, running CentOS 6.6. We utilized the OpenSSL library for the entailed cryptographic operations, where the H and Π are instantiated using HMAC-SHA-256 and AES-512-CBC in the OpenSSL library, respectively. Note that, we implemented the data user and the cloud server on the same machine, which means there is no network delay between them. The underlying dataset (i.e., the dictionary $\mathcal{W}$) in our experiment was extracted from 29,378 articles from Wikivoyage wikivoyage (0000), and it contains 40,205 distinct keywords in total. The length distribution of the keywords in $\mathcal{W}$ can be found in Fig. 7.

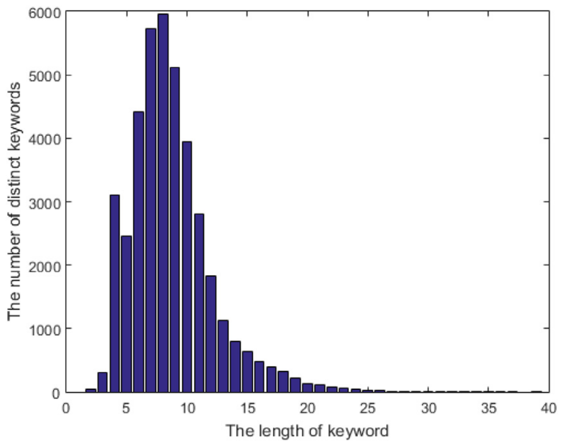


Fig. 7 – The length distribution of a total of 40,205 distinct keywords in $\mathcal{W}$.

In order to show the efficiency of our proposed substring-to-keyword scheme, we compare it with the schemes in Chase and Shen (2015); Leontiadis and Li (2018). Note that, in our experiment, we also use the schemes in Chase and Shen (2015); Leontiadis and Li (2018) to support substring query on the dictionary string $t_{\mathcal{W}}$, which is transformed from the dictionary $\mathcal{W}$ by the method described in Fig. 3(a-b).

6.2.1. *Data outsourcing*

In this part, we consider the storage overhead and computational cost of data outsourcing phase.

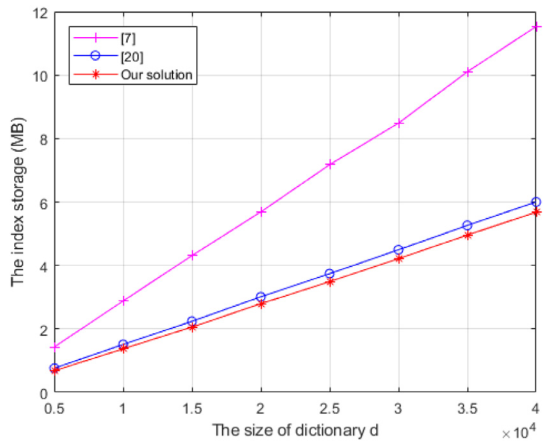


Fig. 8 – The storage overhead of the data outsourcing versus the size of dictionary *d*.

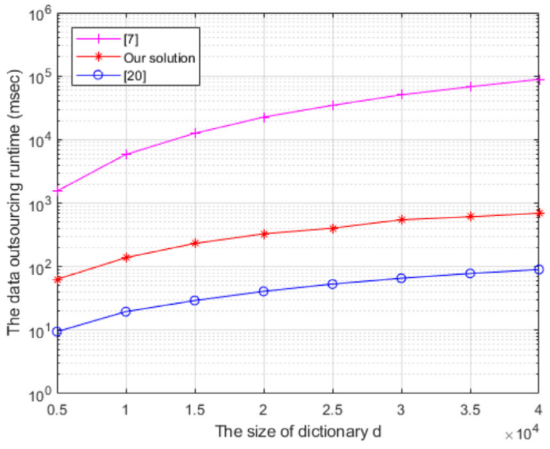


Fig. 9 – The data outsourcing runtime versus the size of dictionary *d*.

In general, given a dictionary string, our solution generates an encrypted position heap, Chase and Shen (2015) generates an encrypted suffix tree, and Leontiadis and Li (2018) generates an encrypted suffix array as the index, respectively. Fig. 8 and Fig. 9 (the y-axis is log scale) depict the storage overhead and the runtime versus the size of dictionary (i.e., d), where d varies from 5000 to 40,000 keywords. The figures show that Chase and Shen (2015) consumes much more storage overhead and computation cost than Leontiadis and Li (2018) and our solution in data outsourcing phase.

6.2.2. *Substring-to-keyword query*

In this part, we randomly choose queried substrings from the dictionary $\mathcal{W}$ and calculate their average queried time. Since the computational cost of substring-to-keyword query is limited by two factors: the size of dictionary (i.e., d) and the number of matching keywords (i.e., d_s), we analyze them separately in the following.

Fig. 10 (the y-axis is log scale) depicts the computational cost of the substring-to-keyword query versus the size of dictionary (i.e., d). This figure shows that the computational cost

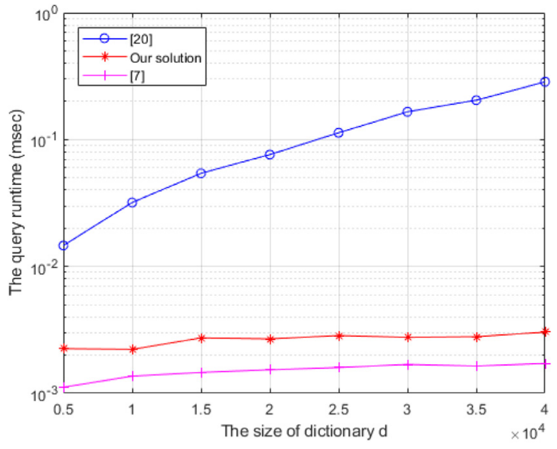


Fig. 10 – Substring-to-keyword query runtime versus the size of dictionary *d*, where the number of matching keywords d_s is 5.

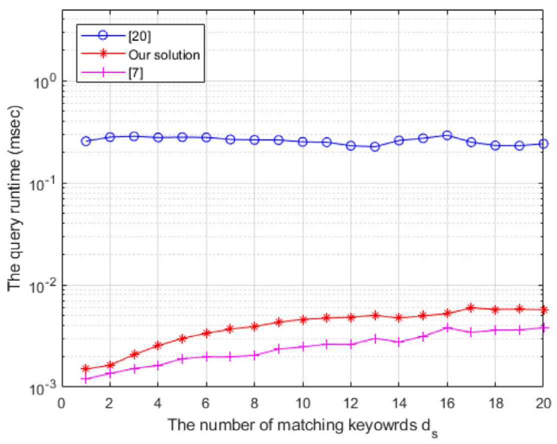


Fig. 11 – Substring-to-keyword query runtime versus the number of matching keywords d_s, where $d = 40000$.

of our solution and Chase and Shen (2015) are not affected by d when the number of matching keywords d_s is fixed. However, the computational cost of Leontiadis and Li (2018) increases linear with d even if d_s is fixed.

Fig. 11 (the y-axis is log scale) plots the runtime of the substring-to-keyword query versus the number of matching keywords d_s, in which d is fixed to 40000. From this figure, we can see that the computational cost of these three schemes are not affected too much by d_s. Meanwhile, our solution and Chase and Shen (2015) are significantly quicker than Leontiadis and Li (2018). For example, when $d_s = 20$, the computational cost of our solution and Chase and Shen (2015) are both about 0.004 ms, which is just about 1/60 compared to Leontiadis and Li (2018).

6.2.3. *Update*

In this part, we consider the update (insertion / deletion) phase. Since there is no secure update method in Chase and Shen (2015); Leontiadis and Li (2018), we only test the update performance of our solution. Since the deletion operation in

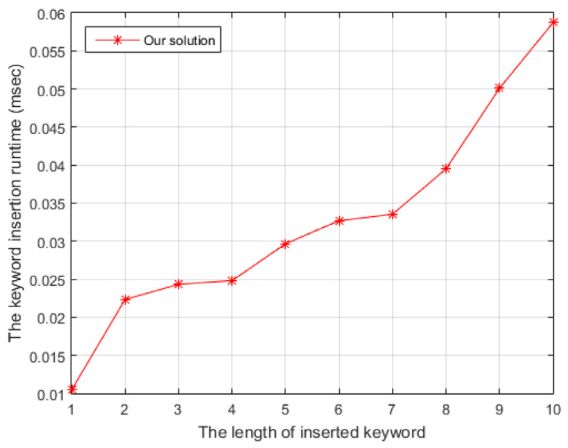


Fig. 12 – Insertion runtime versus the length of inserted keyword, where the size of original dictionary d is 5000.

our solution is the same as the insertion operation, we just evaluate the computational cost of the insertion operation.

Fig. 12 plots the computational cost of the insertion versus the length of inserted keyword, in which the size of original dictionary d is fixed to 5000. From this figure, we can see that the computational cost of our solution increases linearly with the length of inserted keyword.

7. Related work

A searchable encryption scheme can be realized with optimal security via powerful cryptographic tools, such as Fully Homomorphic Encryption (FHE) Gentry (2009, 2010) and Oblivious Random Access Memory (ORAM) Goldreich and Ostrovsky (1996); Ostrovsky (1990). However, these tools are extraordinarily impractical. Another set of works utilize property-preserving encryption (PPE) Bellare et al. (2007); Boldyreva et al. (2009, 2011); Yang et al. (2018) to achieve searchable encryption, which encrypts messages in a way that inevitably leaks certain properties of the underlying message. For balancing the leakage and efficiency, many studies focus on Searchable Symmetric Encryption (SSE). Song et al. Song et al. (2000) first used the symmetric encryption to facilitate keyword query over the encrypted data. Then, Curtmola et al. Curtmola et al. (2006) gave a formal definition of SSE, and proposed an efficient SSE scheme. However, their scheme cannot support update(insertion/deletion) operation. Later, Kamara et al. Kamara et al. (2012) proposed the first dynamic SSE scheme, which used a deletion array and a homomorphic encrypted pointer technique to securely update files. Unfortunately, due to the use of fully homomorphic encryption, the update efficiency is very low. In a more recent paper Cash et al. (2014), Cash et al. described a simple dynamic inverted index based on Curtmola et al. (2006), which utilizes the data unlinkability of hash table to achieve secure insertion. Meanwhile, to prevent the file-injection attacks Zhang et al. (2016), many works Bost (2016); Kim et al. (2017); Zuo et al. (2018, 2019) focused on the forward security, which ensures that newly updated keywords cannot be related to previous queried results.

Nevertheless, these above works only can support the exact keyword query. If the queried keyword does not match a preset keyword, the query will fail. Fortunately, fuzzy query can deal with this problem as it can tolerate minor typos and formatting inconsistencies. Li et al. Li et al. (2010) first proposed a fuzzy query scheme, which used an edit distance with a wildcard-based technique to construct fuzzy keyword sets. For instance, the set of CAT with 1 edit distance is $\{CAT, *CAT, *AT, C*AT, C*T, CA*T, CA*, CAT*\}$. Then, Kuzu et al. Kuzu et al. (2012) used LSH (Local Sensitive Hash) and Bloom filter to construct a similarity query scheme. Because an honest-but-curious server may only return a fraction of the results, Wang et al. Wang et al. (2013) proposed a verifiable fuzzy query scheme that not only supports fuzzy query service, but also provides proof to verify whether the server returns all the queried results. However, these fuzzy query schemes only support single fuzzy keyword query and address problems of minor typos and formatting inconsistency, which can not be directly used to achieve substring-of-keyword query.

In Chase and Shen (2015), Melissa et al. designed a SSE scheme based on the suffix tree to support substring query. Although this scheme can be used to implement the substring-of-keyword query and allows for substring query in $O(|s| + d_s)$ time, its storage cost $O(m)$ has a big constant factor. The reason is that suffix tree only stores position data in leaf nodes and does not utilize the space of inner nodes effectively. This leads the number of nodes in suffix tree can be up to $2m$, where m is the size of the dataset. In order to reduce the storage cost as much as possible, Leontiadis et al. Leontiadis and Li (2018) leveraged Burrows Wheeler Transform (BWT) to build an auxiliary data structure called suffix array, which can achieve storage cost $O(m)$ with a lower constant factor. However, its query time is relatively large. Later, Mainardi et al. Mainardi et al. (2019) optimizes the query algorithm in Leontiadis and Li (2018) to achieve $O(|s| + d_s)$ at the cost of higher index space, i.e., $O(|\Sigma| \cdot m)$, where $|\Sigma|$ is the number of distinct characters in the dictionary. Although authors in this article considered datasets with small $|\Sigma|$ (e.g., DNA dataset), the $|\Sigma|$ can be large in practice. In addition to suffix tree and suffix array, there are other auxiliary data structures can be used to support substring query. In 2018, Florian et al. Hahn et al. (2018) designed an index based on k-grams. When a user needs to perform a substring query, the cloud performs a conjunctive keyword query for all the k-grams of the queried substring. However, its query time is relatively large due to the computational cost of intersection operations in the conjunctive keyword query. In the same year, Tarik et al. Moataz et al. (2018) proposed a new substring query scheme based on the idea of letter orthogonalization, which allows testing of string membership by performing efficient inner product. Again, the disadvantage of this scheme comes its $O(m)$ query time.

8. Conclusion

In this paper, we have proposed an efficient and privacy-preserving substring-of-keyword query scheme over cloud. Specifically, based on the position heap technique, we first designed a tree-based index to support substring-to-keyword query and then applied a PRF and a SKE to protect its privacy. After that, we proposed a novel substring-of-keyword query scheme, which contains two consecutive phases: a substring-to-keyword query that queries the keywords matching a given substring, and a keyword-to-file query that queries the files matching a keyword that the user is really interested. The proposed scheme is very suitable for many critical applications in practice such as Google search. Detailed security analysis and performance evaluation show that our proposed scheme is indeed privacy-preserving and efficient. In our future work, we will take more security properties into consideration, e.g., achieving forward and backward security.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

This work is supported in part by NSERC Discovery Grants (no. Rgpin 04009), Natural Science Foundation of Zhejiang Province (grant no. LZ18F020003), National Natural Science Foundation of China (grant no. U1709217), and NSFC Grant (61871331).

REFERENCES

Bellare M, Boldyreva A, O'Neill A. Deterministic and efficiently searchable encryption. In: Advances in Cryptology - CRYPTO 2007, 27th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19–23, 2007, Proceedings; 2007. p. 535–52.

Boldyreva A, Chenette N, Lee Y, O'Neill A. Order-preserving symmetric encryption. In: Advances in Cryptology - EUROCRYPT 2009, 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cologne, Germany, April 26–30, 2009. Proceedings; 2009. p. 224–41.

Boldyreva A, Chenette N, O'Neill A. Order-preserving encryption revisited: Improved security analysis and alternative solutions. In: Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14–18, 2011. Proceedings; 2011. p. 578–95.

Bost R. $\sum o\varphi o\varsigma$: Forward secure searchable encryption. In: Weippl ER, Katzenbeisser S, Kruegel C, Myers AC, Halevi S, editors. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24–28, 2016. ACM; 2016. p. 1143–54.

Cash D, Jaeger J, Jarecki S, Jutla CS, Krawczyk H, Rosu M-C, Steiner M. Dynamic Searchable Encryption in Very-Large Databases: Data Structures and Implementation. In: NDSS. Citeseer; 2014. p. 23–6.

Cash D, Jarecki S, Jutla CS, Krawczyk H, Rosu M, Steiner M. Highly-scalable searchable symmetric encryption with support for boolean queries. In: Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2013. Proceedings, Part I; 2013. p. 353–73.

Chase M, Shen E. Substring-searchable symmetric encryption. Proceedings on Privacy Enhancing Technologies 2015;2015(2):263–81.

Cloud, I. M., 2010. key marketing trends for 2017 and ideas for exceeding customer expectations.

Curtmola R, Garay JA, Kamara S, Ostrovsky R. Searchable symmetric encryption: improved definitions and efficient constructions. In: Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006, Alexandria, VA, USA, Ioctober 30 - November 3, 2006; 2006. p. 79–88.

Ehrenfeucht A, McConnell RM, Osheim N, Woo S. Position heaps: a simple and dynamic text indexing data structure. J. Discrete Algorithms 2011;9(1):100–21.

Gentry C. Fully homomorphic encryption using ideal lattices. In: Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31, - June 2, 2009; 2009. p. 169–78.

Gentry C. Computing arbitrary functions of encrypted data. Commun. ACM 2010;53(3):97–105.

Goh E-J, et al. Secure indexes.. IACR Cryptology ePrint Archive 2003;2003:216.

Goldreich O, Ostrovsky R. Software protection and simulation on oblivious rams. J. ACM 1996;43(3):431–73.

Hahn F, Loza N, Kerschbaum F. Practical and secure substring search. In: Proceedings of the 2018 International Conference on Management of Data; 2018. p. 163–76.

Kamara S, Papamanthou C, Roeder T. Dynamic searchable symmetric encryption. In: Proceedings of the 2012 ACM conference on Computer and communications security. ACM; 2012. p. 965–76.

Katz J, Lindell Y. Introduction to modern cryptography. CRC press; 2014.

Kim KS, Kim M, Lee D, Park JH, Kim W-H. Forward secure dynamic searchable symmetric encryption with efficient updates. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. ACM; 2017. p. 1449–63.

Kuzu M, Islam MS, Kantarcioglu M. Efficient similarity search over encrypted data. In: 2012 IEEE 28th International Conference on Data Engineering. IEEE; 2012. p. 1156–67.

Leontiadis I, Li M. Storage efficient substring searchable symmetric encryption. In: Proceedings of the 6th International Workshop on Security in Cloud Computing; 2018. p. 3–13.

Li J, Wang Q, Wang C, Cao N, Ren K, Lou W. Fuzzy keyword search over encrypted data in cloud computing. In: 2010 Proceedings IEEE INFOCOM. IEEE; 2010. p. 1–5.

Mainardi N, Barenghi A, Pelosi G. Privacy preserving substring search protocol with polylogarithmic communication cost. In: Proceedings of the 35th Annual Computer Security Applications Conference; 2019. p. 297–312.

Moataz T, Ray I, Ray I, Shikfa A, Cuppens F, Cuppens N. Substring search over encrypted data. J. Comput. Secur. 2018;26(1):1–30.

Ostrovsky R. Efficient computation on oblivious rams. In: Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, May 13–17, 1990, Baltimore, Maryland, USA; 1990. p. 514–23.

Shao J, Lu R, Guan Y, Wei G. Achieve efficient and verifiable conjunctive and fuzzy queries over encrypted data in cloud. IEEE Trans. Serv. Comput. 2019.

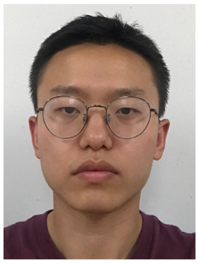
Song DX, Wagner DA, Perrig A. Practical techniques for searches on encrypted data. In: 2000 IEEE Symposium on Security and Privacy, Berkeley, California, USA, May 14–17, 2000; 2000. p. 44–55.

Wang J, Ma H, Tang Q, Li J, Zhu H, Ma S, Chen X. Efficient verifiable fuzzy keyword search over encrypted data in cloud computing.. Comput. Sci. Inf. Syst. 2013;10(2):667–84.

wikivoyage,. https://www.wikivoyage.org/Accessed Nov. 2019.

Yang W, Xu Y, Nie Y, Shen Y, Huang L. TRQED: secure and fast tree-based private range queries over encrypted cloud. In:

Database Systems for Advanced Applications - 23rd International Conference, DASFAA 2018, Gold Coast, QLD, Australia, May 21–24, 2018, Proceedings, Part II; 2018. p. 130–46.
Yin, F., 2019. An implementation of our proposed scheme.
Yin F, Zheng Y, Lu R, Tang X. Achieving efficient and privacy-preserving multi-keyword conjunctive query over cloud. IEEE Access 2019;7:165862–72.
Zhang Y, Katz J, Papamanthou C. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In: 25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10–12, 2016; 2016. p. 707–20.
Zheng Y, Lu R, Li B, Shao J, Yang H, Choo KR. Efficient privacy-preserving data merging and skyline computation over multi-source encrypted data. Inf. Sci. 2019;498:91–105.
Zuo C, Sun S-F, Liu JK, Shao J, Pieprzyk J. Dynamic searchable symmetric encryption schemes supporting range queries with forward (and backward) security. In: European Symposium on Research in Computer Security. Springer; 2018. p. 228–46.
Zuo C, Sun S-F, Liu JK, Shao J, Pieprzyk J. Dynamic searchable symmetric encryption with forward and stronger backward privacy. In: European Symposium on Research in Computer Security. Springer; 2019. p. 283–303.



Fan Yin received the B.S. degree in information security from the Southwest Jiaotong University, Chengdu, China, in 2012. He is currently working toward the Ph.D. degree in information and communication engineering, Southwest Jiaotong University, and also a visiting student at Faculty of Computer Science, University of New Brunswick, Canada. His research interests include searchable encryption, privacy-preserving and security for cloud security and network security.

Rongxing Lu is currently an associate professor at the Faculty of Computer Science (FCS), University of New Brunswick (UNB), Canada. He is a Fellow of IEEE. His research interests include applied cryptography, privacy enhancing technologies, and IoT-Big Data security and privacy. He has published extensively in his areas of expertise, and was the recipient of 9 best (student) paper awards from some reputable journals and conferences. Currently, Dr. Lu serves as the Vice-Chair (Conferences) of IEEE ComSoc CIS-TC (Communications and Information Security Technical Committee). Dr. Lu is the Winner of 2016-17 Excellence in Teaching Award, FCS, UNB.

Yandong Zheng received her M.S. degree from the Department of Computer Science, Beihang University, China, in 2017 and She is currently pursuing her Ph.D. degree in the Faculty of Computer Science, University of New Brunswick, Canada. Her research interest includes cloud computing security, big data privacy and applied privacy.

Jun Shao received the Ph.D. degree from the Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China, in 2008. He was a Postdoctoral Fellow with the School of Information Sciences and Technology, Pennsylvania State University, State College, PA, USA, from 2008 to 2010. He is currently a Professor with the School of Computer Science and Information Engineering, Zhejiang Gongshang University, Hangzhou, China. His current research interests include network security and applied cryptography.

Xue Yang received the Ph.D. degree in Information and Communication Engineering from Southwest Jiaotong University, Chengdu, China, in 2019. She was a visiting student at the Faculty of Computer Science, University of New Brunswick, Canada, from 2017 to 2018. She is currently a Postdoctoral Fellow in the Tsinghua Shenzhen International Graduate School, Tsinghua University, China. Her research interests include big data security and privacy, applied cryptography and federated learning.

Xiaohu Tang received the B.S. degree in applied mathematics from Northwest Polytechnic University, Xi'an, China, in 1992, the M.S. degree in applied mathematics from Sichuan University, Chengdu, China, in 1995, and the Ph.D. degree in electronic engineering from Southwest Jiaotong University, Chengdu, in 2001. From 2003 to 2004, he was a Research Associate with the Department of Electrical and Electronic Engineering, The Hong Kong University of Science and Technology. From 2007 to 2008, he was a Visiting Professor with the University of Ulm, Germany. Since 2001, he has been with the School of Information Science and Technology, Southwest Jiaotong University, where he is currently a Professor. His research interests include coding theory, network security, distributed storage, and information processing for big data. Dr. Tang was a recipient of the National excellent Doctoral Dissertation Award in 2003 (China), the Humboldt Research Fellowship in 2007 (Germany), and the Outstanding Young Scientist Award by NSFC in 2013 (China). He served as an Associate Editor for several journals, including the IEEE TRANSACTIONS ON INFORMATION THEORY and IEICE Transactions on Fundamentals, and served for a number of technical program committees of conferences.