Privacy-preserving Indexing and Query Processing for Secure Dynamic Cloud Storage

Minxin Du, Qian Wang, *Member, IEEE,* Meiqi He, and Jian Weng

***Abstract*—With the increasing popularity of cloud-based data services, data owners are highly motivated to store their huge amount of potentially sensitive personal data files on remote servers in encrypted form. Clients later can query over the encrypted database to retrieve files while protecting privacy of both the queries and the database, by allowing some reasonable leakage information. To this end, the notion of searchable symmetric encryption (SSE) was proposed. Meanwhile, recent literature has shown that most dynamic SSE solutions leaking information on updated keywords are vulnerable to devastating file-injection attacks. The only way to thwart these attacks is to design *forward-private* schemes.**

In this paper, we investigate new privacy-preserving indexing and query processing protocols which meet a number of desirable properties, including the multi-keyword query processing with conjunction and disjunction logic queries, practically high privacy guarantees with adaptive chosen keyword attack (CKA2) security and forward privacy, and the support of dynamic data operations, etc. Compared to previous schemes, our solutions are highly compact, practical and flexible. Their performance and security are carefully characterized by rigorous analysis. Experimental evaluations conducted over a large representative dataset demonstrate that our solutions can achieve modest search time efficiency, and they are practical for use in large-scale encrypted database systems.

***Index Terms*—Encrypted database, multi-keyword search, dynamic update, cloud computing.**

I. Introduction

In the cloud computing paradigm, providing database-as-a-service (DaaS) allows a third party service provider to host database as a service, providing its customers seamless mechanisms to create, store, and access databases at cloud with adequate storage resource, convenient data access and reduced management and infrastructure costs. But database outsourcing also raises data confidentiality and privacy concerns due to data owner's loss of physical data control. To provide privacy guarantees for sensitive data such as personal identities, health records, financial data, *etc.*, a straightforward approach is to encrypt the sensitive data locally before outsourcing [13], [14], [24], [28], [30], [31]. While providing strong end-to-end privacy, encryption becomes a hindrance to data computation or utilization, *i.e.*, it is hard to retrieve data files based on their content as in the plaintext search domain. In addition, clients are also concerned about their query privacy, expecting that the database server cannot learn the data content nor the query in plaintext form.

To address these challenges, the notion of searchable symmetric encryption (SSE) was first introduced by Song *et al.* [25]. Roughly speaking, a SSE scheme encrypts data in such a way that it can be privately queried through the use of a query-specific token generated with knowledge of the secret key. In recent years, researchers have put great efforts to make SSE solutions [6], [7], [9], [10], [12], [16], [17], [20], [26], [29] practical. Kamara *et al.* [17] designed an inverted index based searchable encryption scheme where this index can be incrementally updated. Afterwards, a parallelizable and dynamic SSE scheme based on the red-black tree data structure is proposed in [16]. In another work [5], Cash *et al.* also proposed a parallelizable and dynamic SSE scheme by leveraging the generic dictionary structure, which stores newly-added document-keyword pairs in an auxiliary encrypted dictionary and records the pair to be deleted in a revocation list. Moreover, their scheme shows good scalability when it searches on datasets with billions of document-keyword pairs. Hahn *et al.* [12] proposed a dynamic SSE construction which uses linked lists to construct the inverted index. However, the client needs to maintain an additional structure called history which stores previous search tokens, and the server has to traverse the encrypted index with time cost linear in the number of document-keyword pairs for a new search.

The above dynamic SSE schemes cannot achieve *forward privacy*, which means that the server cannot learn whether an updated data file contains a keyword w that has been searched or not in the past. In a recent work [33], by leveraging such update leakage, the authors showed some devastating file-injection attacks which can be run on almost all the existing SSE schemes. As a consequence, this work underlines the need for dynamic SSE constructions to provide the forward privacy guarantee. To address this problem, forward privacy was for the first time explicitly considered by Stefanov *et al.* in [26]. Their main idea is to store document-keyword pairs in a hierarchical structure of logarithmic levels by using techniques similar in oblivious RAM. Unfortunately, their search time becomes very long with the increasing number of document-keyword pairs, meanwhile, the updates require multiple rounds of communication between the server and the client with non-negligible overheads.

In this paper, we propose a new index design for processing queries over encrypted cloud storage which we call proba-

M. Du and Q. Wang are with the Key Laboratory of Aerospace Information Security and Trusted Computing, Ministry of Education, School of Cyber Science and Engineering, Wuhan University, China, and the State Key Laboratory of Integrated Services Networks, Xidian University, Xi'an, China (E-mail: {duminxin,qianwang}@whu.edu.cn).

M. He is with the Department of Computer Science, The University of Hong Kong, Hong Kong (e-mail: mqhe@cs.hku.hk).

J. Weng is with the Department of Computer Science, Jinan University, China (E-mail: cryptjweng@gmail.com).

bilistic inverted index coding structure. On the basis of this basic structure, we further propose two specific constructions with privacy guarantees. Our constructions carefully make a trade-off between query efficiency and privacy, achieving flexible query functionalities. In particular, among other very recent works [2], [3], [19] to be discussed later, our solutions provide practical privacy guarantees including *forward privacy*. Besides, our solutions have a compact index structure supporting multi-keyword queries (including both conjunction and disjunction logic queries) and data updates with modest overheads. Our main technical contributions are as follows.

We propose bucket-encrypting index structure with random generator, named BEIS-I. Based on the basic index structure, we encrypt *data identifier vectors* (DIVs), bit-vectors representing the collection of entire data files, using random numbers. BEIS-I allows practically efficient multi-keyword query with privacy preservation. Due to the decomposable index structure, BEIS-I naturally supports secure data updates.

We also propose bucket-encrypting index structure with homomorphic generator, named BEIS-II. To save bandwidth during the query process, BEIS-II strategically makes use of additively homomorphic encryption to encrypt DIVs. Moreover, BEIS-II can support effective and secure batch data processing (including updates) due to the ciphertext packing.

Finally, our formal security analysis shows that both our constructions enjoy security against adaptive chosen-keyword attacks (*i.e.*, CKA2-security) and also *forward privacy*. To demonstrate the practicality of our solutions, we conduct experimental evaluations using large representative datasets.

II. Related Work

A. Single-Keyword Search

The search time of the first SSE scheme proposed by Song *et al.* [25] is linear in the length of the file collection. By associating an encrypted index to each file, the SSE construction proposed by Goh [10] achieves search time that is proportional to $\#\mathbf{d}$, the number of files in the collection. In 2006, Curtmola *et al.* [9] generalized the security definitions of SSE (named CKA1 and CKA2) and proposed a new encrypted index structure that is associated with the entire file collection. Its search time is linear in $\#\mathbf{d}_w$, the number of files that contain keyword w. Kamara *et al.* [17] extended the inverted index approach [9] to support both addition and removal of data files. This construction achieves the overall best performance in single-core architectures. Motivated by advances in multi-core architectures, Kamara *et al.* [16] proposed a parallelizable and dynamic SSE scheme based on red-black tree data structure. The search time complexity of parallel execution is $O(\frac{\#\mathbf{d}_w}{p}\log\#\mathbf{d})$, where p is the number of the available processors. Hahn *et al.* [12] proposed a similar construction which uses linked lists to construct the inverted index. The client in their scheme needs to maintain a data structure history σ which stores previously-used search tokens in order to get rid of time cost $O(|d|\cdot|\sigma|)$ to perform file additions, where $|d|$ is the number of keywords in newly-added file d and $|\sigma|$ is the number of linked lists in the index γ_w. The advantage is that if the tokens were submitted before, then during the search, the server can return the matched results immediately according to the index γ_w. However, in general case, the server has to traverse the whole index γ_f and this time cost is linear in the number of document-keyword pairs. Cash *et al.* [5] proposed a parallelizable SSE scheme by leveraging the dictionary structure. With the help of an auxiliary encrypted database and a revocation list, it then can support addition and deletion operations. Though their scheme offers optimal search complexity $O(\#\mathbf{d}_w)$, the functionality that supports multi-keyword search is not realized yet.

The above dynamic SSE schemes such as [12], [16], [17] cannot achieve *forward privacy*, and they also leak a lot of additional information during the updates, *e.g.*, the keyword hashes shared between data files besides the keyword hashes of the queried keywords. The first SSE scheme that achieves *forward privacy* is proposed by Stefanov *et al.* [26]. It stores document-keyword pairs in a hierarchical structure of logarithmic levels, which is reminiscent of algorithmic techniques used in the ORAM literature. From the perspective of performance, its search complexity is $O(m\log^3 N)$, where m is the number of documents containing query keyword and N is the number of document-keyword pairs. Although the theoretical cost is sub-linear, the actual search time cost will be extremely large with the increase of N. On the other hand, each update in their scheme induces $O(\log(N))$ bandwidth and takes $O(\log^2(N))$ time in the worst-case. When one level needs to be rebuilt during update, it must ensure that the client has $O(N^\alpha)$ local storage for $0<\alpha<1$, and it also induces several rounds of communication between the server and the client. Moreover, it needs to resize the whole data structure when the number of deleted entries equals $N/2$, otherwise it will cause a blow-up in the space of the data structure. By resorting to the trapdoor permutation (TDP), the formal definition of *forward privacy* is first presented in a very recent work of Bost [2], and further generalized to any inverted index based SSE scheme. However, the construction of [2] only supports single-keyword searches and maintains additional client's storage (which could be unsatisfactory for constrained devices) needed for the token generation. In a follow-up work [3], Bost *et al.* realized single-keyword SSEs from constrained and puncturable cryptographic primitives. Through the fine-grained control afforded by these primitives, they for the first time studied the notion of backward privacy, and designed schemes achieving both forward privacy and different flavors of backward privacy. Kim *et al.* [19] proposed a new data structure, called *dual dictionary*, which contains linked dictionaries to represent both inverted and forward indexes. To achieve forward security, the proposed scheme encrypts the newly-added data with fresh keys that are not related to the previous search tokens.

B. Multi-Keyword Search

From the perspective of search functionality, one common limitation of the above SSE solutions is that they only support single-keyword search, the direct extension to multi-keyword search requires to compute the intersection of disjunctive queries for each keyword. Ballard *et al.* [1] presented two

schemes for performing conjunctive keyword searches over the encrypted data. While the first one is based on Shamir secret sharing with the size of search tokens linear in the total number of data files, and the second yields constant size tokens by leveraging bilinear maps. The multi-keyword search problem was investigated by Cao *et al.* [4] and a multi-keyword ranked search scheme was proposed using the secure kNN technique in [32]. However, their approach is limited by the large index storage cost due to its associated large dictionary, and the low search efficiency due to its high vector computation cost introduced in the one-to-one search [27]. To support generic boolean search including both conjunction and disjunction expressions, Moataz *et al.* [21] proposed an SSE design which transfers keywords to vectors and used the Gram-Schmidt process to orthogonalize and orthonormalize the resulting vectors. Their scheme also incurs huge computation costs because $O(\#\mathbf{d})$ inner products are needed to execute the encrypted search. By combining Bloom filters and Yao's garbled circuits, Pappas *et al.* [23] generated an encrypted search tree to allow arbitrary Boolean queries with sublinear overheads. Moreover, their construction shows practical scalability which is suitable for real-world deployment on large-scale databases with millions of records.

Cash *et al.* [6] designed the OXT construction supporting multi-keyword Boolean queries that can be expressed in searchable normal form. While OXT can only handle arbitrary disjunctive and boolean queries in linear time. For each query, their scheme first retrieves the results for one of the query keywords, and then filters them according to the given boolean query against each of remaining keywords. Yet, as with most of the existing dynamic SSE schemes, OXT does not achieve *forward privacy*. A follow-up work of Kamara *et al.* [15] proposed IEX which supports disjunctive queries and can be instantiated in various ways (*e.g.*, [5]). By sacrificing some security, IEX achieves optimal communication complexity and handles boolean queries via conjunctive normal form (CNF). Furthermore, it can be extended to a dynamic scheme called DIEX which naturally inherits the forward security of the underlying encrypted multi-maps and dictionaries.

III. Preliminaries and Definitions

A. Notations

In cloud-based database systems, the data owner outsources a large-scale collection of data files $\mathbf{d} = (d_1, \ldots, d_{\#\mathbf{d}})$ to the remote database server in encrypted form $\mathbf{c} = (c_1, \ldots, c_{\#\mathbf{c}})$. The data files $\mathbf{d}$ can represent text files or records in a relational database. The keyword universe (which contains all distinct keywords) extracted from $\mathbf{d}$ is denoted by $\mathbf{w} = (w_1, \ldots, w_{\#\mathbf{w}})$. We use $\mathtt{id}(d_j)$ to denote the identifier of data file d_j and $\mathtt{id}(\mathbf{d})$ to denote the identifiers of all data files in $\mathbf{d}$. Given a vector v, we refer its j^{th} element as $v[j]$ or v_j. In our construction, we use $\mathsf{div} \in \{0,1\}^{\#\mathbf{d}}$ to denote a data identifier vector (DIV), where the j^{th} entry of div is 1 if d_j is included; 0 otherwise. By this definition, given w_i, we can use div_i or div_{w_i} to denote all data files that contain w_i (*i.e.*, the corresponding entries in div_i are 1's). Different from most previous SSE constructions, we consider multi-keyword query $\mathbf{q} = \{w_1, \ldots, w_{\#\mathbf{q}}\}$. Hereafter, we use $\mathbf{d_q}$ to represent the data files for both conjunction and disjunction logic queries. In the following discussion, if $\#\mathbf{q} = 1$ then we use w to denote the single-keyword query, *i.e.*, $\mathbf{q} = \{w\}$. Similarly, we use $\mathbf{d}_w$ to denote all the data files that contain the single keyword w. Standard bitwise Boolean operations are defined on binary vectors: bitwise OR (union) "$\bigvee$" and bitwise AND (intersection) "$\bigwedge$", which also take the same notations for single bit OR and AND operations, respectively. The concatenation of n strings $s_1, \ldots, s_n$ is denoted by $s_1||\ldots||s_n$.

B. Problem Statement

To enable the query service over $\mathbf{c}$ for effective information retrieval, the data owner needs to build an encrypted index $\boldsymbol{\gamma}$ based on $\mathbf{d}$ and outsources ($\boldsymbol{\gamma}$, $\mathbf{c}$) to the remote database server. Later, a client (for ease of exposition, in the following statements when we mention a client, we refer it as the data owner or an authorized user) can use a multi-keyword query $\mathbf{q}$ to retrieve data files of interest in the database. To this end, the client generates a search token $\boldsymbol{\tau}$ and sends it to the remote cloud server. After receiving $\boldsymbol{\tau}$, the cloud executes the query on $\boldsymbol{\gamma}$ and returns all the encrypted data files $\mathbf{c_q}$ or their corresponding encrypted IDs that satisfy the query. Each of $\mathbf{c_q}$ contains all the query keywords in $\mathbf{q}$ for the 'AND' logic query, or it contains the ranked encrypted data files for the 'OR' logic query based on a specific similarity metric. Finally, the client locally decrypts $\mathbf{c_q}$ and obtains data contents in plaintext form.

The database system considered here is a dynamic one. That is, at any time the client may add or remove (a set of) data files to or from the remote database. To do so, the client generates an update token $\boldsymbol{\tau}_a$/$\boldsymbol{\tau}_d$ for the data files to be updated. Given $\boldsymbol{\tau}_a$/$\boldsymbol{\tau}_d$, the database server can then perform secure updates on $\boldsymbol{\gamma}$ and $\mathbf{c}$, respectively.

Formally, the core functionalities of our database system are defined below.

Definition 1. *A dynamic searchable encrypted database supporting multi-keyword query is a tuple of seven (probabilistic) polynomial-time algorithms* $(\mathsf{KeyGen}, \mathsf{BuildIndex}, \mathsf{Trapdoor}, \mathsf{Query}, \mathsf{Recovery}, \mathsf{UpdateToken}, \mathsf{Update})$ *such that:*

$K \leftarrow \mathsf{KeyGen}(1^k)$*: is a probabilistic key generation algorithm run by the data owner. It takes as input a security parameter k and outputs the key K.*

$(\boldsymbol{\gamma}, \mathbf{c}) \leftarrow \mathsf{BuildIndex}(K, \mathbf{d})$*: is a (possibly probabilistic) algorithm run by the data owner to build the encrypted index. It takes as input the key K and data collection $\mathbf{d}$ and outputs an encrypted index $\boldsymbol{\gamma}$ and the ciphertexts $\mathbf{c}$.*

$\boldsymbol{\tau} \leftarrow \mathsf{Trapdoor}(K, \mathbf{q})$*: is a (possibly probabilistic) algorithm run by the data owner or authorized users to generate a search token. It takes as input K and a given query request $\mathbf{q}$, and outputs the trapdoor $\boldsymbol{\tau}$.*

$\mathbf{c_q} \leftarrow \mathsf{Query}(\boldsymbol{\tau}, \boldsymbol{\gamma}, \mathbf{c})$*: is a deterministic algorithm run by the database server. It takes as input the search token $\boldsymbol{\tau}$, the encrypted index $\boldsymbol{\gamma}$ and $\mathbf{c}$ and outputs a sequence of encrypted data files $\mathbf{c_q}$.*

$\mathbf{d_q} \leftarrow \mathsf{Recovery}(K, \mathbf{c_q})$*: is a deterministic algorithm run by the data owner or authorized users. It takes as input the key* K *and the ciphertexts* $\mathbf{c_q}$*, and outputs files* $\mathbf{d_q}$*.*

$\boldsymbol{\tau}_a/\boldsymbol{\tau}_d \leftarrow \mathsf{UpdateToken}(K, d_j)$*: is a (possibly probabilistic) algorithm that takes as input the key* K *and a data file* d_j *to be added or deleted, and outputs an update token* $\boldsymbol{\tau}_a/\boldsymbol{\tau}_d$*.*

$(\boldsymbol{\gamma}', \mathbf{c}') \leftarrow \mathsf{Update}(\boldsymbol{\tau}_a/\boldsymbol{\tau}_d, \boldsymbol{\gamma}, \mathbf{c})$*: is a deterministic algorithm run by the database server. It takes as input the encrypted index* $\boldsymbol{\gamma}$ *and a update token* $\boldsymbol{\tau}_a/\boldsymbol{\tau}_d$ *and outputs the updated encrypted data collection* $\mathbf{c}'$ *and index* $\boldsymbol{\gamma}'$*.*

Note that in UpdateToken, the input d_j can be replaced by a set of data files and thus the output update token $\boldsymbol{\tau}_a/\boldsymbol{\tau}_d$ is used for batch file updates.

C. Cryptographic Primitives

1) Symmetric-key Encryption (SKE): An SKE scheme is a set of three polynomial-time algorithms $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$. Gen takes a security parameter k as input and outputs a secret key sk. Enc takes a key sk and message m as input and outputs a ciphertext c. Dec takes sk and c as input and outputs m iff c was encrypted under the key sk. Intuitively, an SKE scheme is secure against chosen-plaintext attacks (CPA) if the ciphertexts do not leak any useful information about the plaintext even to an adversary that can query an encryption oracle. Many common SKE schemes such as AES in counter mode satisfy this definition.

2) Pseudo-random Function (PRF): A keyed function F is a PRF if it is a polynomial-time computable function that there exists no probabilistic polynomial-time (PPT) adversary can distinguish it from random functions when the key is kept secret. We refer the readers to [18] for formal definitions.

3) Homomorphic Encryption (HE): HE allows certain computations to be carried out on ciphertexts to generate an encrypted result which matches the result of operations performed on the plaintext after being decrypted. Although fully homomorphic encryption (FHE) for arbitrary operations is prohibitively slow, homomorphic encryption for solely addition operations is practically efficient. In particular, we adopt the Paillier cryptosystem [22] which is additively homomorphic in our construction.

In the Paillier cryptosystem, the public (encryption) key is $pk_p = (n = pq, g)$, where $g \in \mathbb{Z}^*_{n^2}$, and p and q are two large prime numbers (of equivalent length) chosen randomly and independently. The private (decryption) key is $sk_p = (\varphi(n), \varphi(n)^{-1} \mod n)$. Given a message a, we write the encryption of a as $[\![a]\!]_{pk}$, or simply $[\![a]\!]$, where pk is the public key. The encryption of a message $x \in \mathbb{Z}_n$ is $[\![x]\!] = g^x \cdot r^n \mod n^2$, for some random $r \in \mathbb{Z}^*_n$. The decryption of the ciphertext is $x = L([\![x]\!]^{\varphi(n)} \mod n^2) \cdot \varphi^{-1}(n) \mod n$, where $L(u) = \frac{u-1}{n}$. The homomorphic property of the Paillier cryptosystem is given by $[\![x_1]\!] \cdot [\![x_2]\!] = (g^{x_1} \cdot r_1^n) \cdot (g^{x_2} \cdot r_2^n) = g^{x_1+x_2}(r_1 r_2)^n \mod n^2 = [\![x_1 + x_2]\!]$.

D. Security Model and Definitions

The security of an SSE scheme expresses the fact that the server should learn as little as possible about the content of the database and queries, which means the adversary cannot learn anything beyond some explicit leakage. We recall the semantic security definition of SSE schemes for single keyword queries from [8], [9], [17], which is typically captured using a real-world versus ideal-world formalization. The model is parameterized by a set of leakage functions that describe information leaked to the adversary when executing each operation. More precisely, the security says that the view of an adversary during an attack can be simulated given only the output of the leakage functions. Therefore, we follow this widely-accepted simulation-based framework [8] to analyze the security of our constructions which support multi-keyword search, namely, the security can be characterized via proper leakage definitions required for the ideal world simulation.

Definition 2. *(Dynamic CKA2-security). Let* SSE *be a searchable encrypted database scheme as in Definition 1 and consider following probabilistic experiments, where* $\mathcal{A}$ *is a stateful semi-honest adversary,* $\mathcal{S}$ *is a stateful simulator and* $\mathcal{L}_1$*,* $\mathcal{L}_2$*, and* $\mathcal{L}_3$ *are stateful leakage functions for setup, search, and update respectively:*

$\mathbf{Real}_{\mathcal{A}}(k)$*: the challenger runs* $\mathsf{KeyGen}(1^k)$ *to generate the secret key* K*.* $\mathcal{A}$ *outputs* $\mathbf{d}$ *and receives* $(\boldsymbol{\gamma}, \mathbf{c})$ *such that* $(\boldsymbol{\gamma}, \mathbf{c}) \leftarrow \mathsf{BuildIndex}(K, \mathbf{d})$ *from the challenger.* $\mathcal{A}$ *makes a polynomial number of adaptive multi-keyword queries* $\mathbf{q}$ *or update queries* d_j*. For each* $\mathbf{q}$*,* $\mathcal{A}$ *receives from the challenger a search token* $\boldsymbol{\tau} \leftarrow \mathsf{Trapdoor}(K, \mathbf{q})$*. For each file* d_j *to be updated,* $\mathcal{A}$ *receives from the challenger an update token* $\boldsymbol{\tau}_a/\boldsymbol{\tau}_d \leftarrow \mathsf{UpdateToken}(K, d_j)$*. Finally,* $\mathcal{A}$ *returns a bit* b *that is output by the experiment.*

$\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(k)$*:* $\mathcal{A}$ *outputs* $\mathbf{d}$*. Given* $\mathcal{L}_1(\mathbf{d})$*,* $\mathcal{S}$ *generates and sends a pair* $(\boldsymbol{\gamma}, \mathbf{c})$ *to* $\mathcal{A}$*. The adversary* $\mathcal{A}$ *makes a polynomial number of adaptive multi-keyword queries* $\mathbf{q}$ *or update queries* d_j*. For each* $\mathbf{q}$*,* $\mathcal{S}$ *is given* $\mathcal{L}_2(\mathbf{d}, \mathbf{q})$ *and returns an appropriate search token* $\boldsymbol{\tau}$*. For each file* d_j *to be updated,* $\mathcal{S}$ *is given* $\mathcal{L}_3(\mathbf{d}, d_j)$ *and returns an appropriate update token* $\boldsymbol{\tau}_a/\boldsymbol{\tau}_d$*. Finally,* $\mathcal{A}$ *returns a bit* b *that is output by the experiment.*

We say that SSE *is* $(\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3)$*-secure against adaptive dynamic chosen-keyword attacks (CKA2) if for all PPT adversaries* $\mathcal{A}$*, there exists a PPT simulator* $\mathcal{S}$ *such that*

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

where $\mathsf{negl}(k)$ *is a function negligible in* k*.*

The above definition ensures that an adversary $\mathcal{A}$ cannot learn any information beyond the leakage, otherwise it could use this extra information to distinguish the experiments.

IV. Our Constructions

A. Overview

On the basis of data identifier vectors (DIVs), we propose an encoding approach which allows identification of files to match the general multi-keyword queries. Our key idea is as follows. First, we map the keywords (contained in the keyword universe) onto a line (*i.e.*, an array) by leveraging a collection of independent hash functions, and then put the corresponding DIVs into the hit buckets. Hereafter, we call each entry in the array a bucket, which will be pointing

to the possible matching files. Afterwards, both conjunctive and disjunctive logic queries can be achieved by performing arithmetic operations on the DIVs hit by the issued search token, which is equivalent to post-processing a sequence of single-keyword queries. Specifically, for the conjunctive logic queries, files are returned iff the intersection of all hit bits in a row equals to 1. For the disjunctive logic queries, files are returned according to the similarity scores computed on all hit bits in a row.

Based on the above ideas, we further propose two schemes BEIS-I and BEIS-II with privacy preservation. The former masks the original bits of DIVs with the outputs of a PRF, the latter encrypts the bit-vectors using Paillier homomorphic cryptosystem. Intuitively, BEIS-I is computationally more efficient due to the use of symmetric encryption, while BEIS-II has lower communication cost since multiple encrypted DIVs can be aggregated via the additively homomorphic property and the ciphertext packing technique is adopted.

B. Probabilistic Coding Data Structure: Basic Construction

We first present our basic index construction, which does not protect the privacy of DIVs, as follows.

1) Initialization and Index Construction:

$\mathsf{KeyGen}(1^k)$: Given a security parameter k, sample a k-bit string sk_1 uniformly at random, and call $sk_2 \leftarrow \mathsf{SKE.Gen}(1^k)$. Output $K = (sk_1, sk_2)$.

$\mathsf{BuildIndex}(K, \mathbf{d})$:

i) Select a collection of r independent keyed hash functions $h_{i=[1,r]} = \{h_i | i \in [1,r]\}$, where $h_i : \{0,1\}^* \times \{0,1\}^k \rightarrow \{0,1\}^l$.

ii) Extract $\mathbf{w} = \{w_1, \ldots, w_{\#\mathbf{w}}\}$ from $\mathbf{d}$ and generate the corresponding DIVs $\{\mathsf{div}_1, \ldots, \mathsf{div}_{\#\mathbf{w}}\}$.

iii) Initialize an array $\boldsymbol{\gamma}$ consisting of m buckets where each bucket is set to be empty.

iv) For each $w_i \in \mathbf{w}$

(a) Compute the r hash bucket positions $x_1 = h_1(w_i, sk_1), \ldots, x_r = h_r(w_i, sk_1)$.

(b) For each position x_j, if the bucket $\boldsymbol{\gamma}[x_j]$ is empty, then store div_i in $\boldsymbol{\gamma}[x_j]$; otherwise, update it by storing the bitwise OR of div_i and the "old" DIV previously stored at this position, denoted by $\boldsymbol{\gamma}[x_j] = \boldsymbol{\gamma}[x_j] \bigvee \mathsf{div}_i$.

v) For $1 \le i \le \#\mathbf{d}$, let $c_i \leftarrow \mathsf{SKE.Enc}(sk_2, d_i)$.

vi) Output $(\boldsymbol{\gamma}, \mathbf{c})$, where $\mathbf{c} = (c_1, \ldots, c_{\#\mathbf{c}})$.

$\mathsf{Recovery}(K, \mathbf{c_q})$: Return $d_i \leftarrow \mathsf{SKE.Dec}(sk_2, c_i)$ for $c_i \in \mathbf{c_q}$.

2) Multi-keyword Search Over the Index:

$\mathsf{Trapdoor}(sk, \mathbf{q})$: For each $w_i \in \mathbf{q}$, compute positions $(x_1 = h_1(w_i, sk_1), \ldots, x_r = h_r(w_i, sk_1))$. Output the union of all positions as $\boldsymbol{\tau}$.

$\mathsf{Query}(\boldsymbol{\gamma}, \mathbf{c}, \boldsymbol{\tau})$: Use $\boldsymbol{\tau}$ to perform specified logic ('AND' or 'OR') query over $\boldsymbol{\gamma}$.

(a) *Conjunction logic query*. Extract the corresponding DIVs from all hit buckets in $\boldsymbol{\tau}$ and compute $\mathsf{div}_{\boldsymbol{\tau}} = \bigwedge_{y \in \boldsymbol{\tau}} \boldsymbol{\gamma}[y]$. Return encrypted data files $\mathbf{c_q} = \{c_i \in \mathbf{c} : \mathsf{div}_{\boldsymbol{\tau}}[i] = 1\}$. To state the correctness, let $\boldsymbol{\tau}_w$ be a sub-trapdoor for a single keyword w in $\mathbf{q}$. Thus, $\boldsymbol{\tau}_w \subset \boldsymbol{\tau}$ and

$$\mathsf{div}_{\boldsymbol{\tau}} = \bigwedge_{w \in \mathbf{q}} \bigwedge_{y \in \boldsymbol{\tau}_w} \boldsymbol{\gamma}[y] = \bigwedge_{w \in \mathbf{q}, y \in \boldsymbol{\tau}_w} \boldsymbol{\gamma}[y] = \bigwedge_{y \in \boldsymbol{\tau}} \boldsymbol{\gamma}[y]. \quad (1)$$

It is easy to see that the execution will return data files containing all keywords in the multi-keyword query $\mathbf{q}$.

(b) *Disjunction logic query*. Compute the similarity score, denoted by $score(d_j, \mathbf{q})$, between data file d_j and query $\mathbf{q}$. Specifically, the score is defined as

$$\begin{aligned} score(d_j, \mathbf{q}) = r \cdot \mathbf{1}\Big(\bigwedge_{y \in \{\boldsymbol{\tau}[1], \ldots, \boldsymbol{\tau}[r]\}} \boldsymbol{\gamma}[y][j] = 1\Big) + \\ \cdots + r \cdot \mathbf{1}\Big(\bigwedge_{y \in \{\boldsymbol{\tau}[\#\mathbf{q}r - r + 1], \ldots, \boldsymbol{\tau}[\#\mathbf{q}r]\}} \boldsymbol{\gamma}[y][j] = 1\Big), \end{aligned} \quad (2)$$

where $\mathbf{1}(\cdot)$ denotes an indicator function, and it equals to 1 if the condition in $\mathbf{1}(\cdot)$ holds; 0, otherwise. Sort the similarity scores in a descending order and return ranked encrypted data files.

Remarks. As can be seen from Eq. (1), for conjunction logic queries, the query result of our basic index structure is equivalent to that of post-processing a sequence of single-keyword queries. For disjunction logic queries, $score(d_j, \mathbf{q})$ can be seen as the number of common buckets mapped by the keywords in d_j and $\mathbf{q}$. Therefore, the larger the number of common buckets, the higher similarity between the data d_j and the query $\mathbf{q}$. Note that if a data file d_j contains at least one query keyword, then the $score(d_j, \mathbf{q}) \ge r$. Due to this property, the server can return all ranked data files with $score(d_j, \mathbf{q}) \ge r$ or the ranked results with top-k highest scores, where k could be user-specified.

3) Supporting Index Dynamics:

In practice, supporting efficient data dynamics is a natural requirement for full-fledged database systems. Both addition and removal of data files should be supported without either re-indexing the whole database from scratch or making use of generic and relatively expensive dynamization techniques.

$\mathsf{UpdateToken}(K, d_j)$: To add a new file d_j, the client locally encrypts it as c_j and constructs the sub-index $\boldsymbol{\gamma}^*$ for d_j which has the same structure as $\boldsymbol{\gamma}$. To this end, the client simply runs $\mathsf{BuildIndex}(K, d_j)$ and sets $\boldsymbol{\tau}_a = (c_j, \boldsymbol{\gamma}^*)$. To delete an existing file d_j, the client first determines its location, then generates $\boldsymbol{\tau}_d = (j : d_j \in \mathbf{d})$.

$\mathsf{Update}(\boldsymbol{\tau}_a / \boldsymbol{\tau}_d, \boldsymbol{\gamma}, \mathbf{c})$: For addition, merge $\boldsymbol{\gamma}^*$ into the original index $\boldsymbol{\gamma}$ and c_j into ciphertexts $\mathbf{c}$, respectively. For deletion, set j-th entry of each bucket in $\boldsymbol{\gamma}$ to 0, and delete c_j from $\mathbf{c}$.

C. BEIS-I: Using Random Generator

The above basic index construction supports efficient conjunctive or disjunctive logic queries without privacy preservation. On the basis of it, we first present BEIS-I, a bucket-encrypting index structure (BEIS) using random generator which is instantiated by a PRF. In BEIS-I, every original bit of each IFV is padded with a number generated by the PRF. Due to the use of symmetric key encryption and the inheritance of basic index structure, BEIS-I achieves practical efficiency when performing multi-keyword queries and updates. In the following, to avoid the reiteration of the same steps, we only present the key differences and modifications.

1) Initialization and Index Construction:

$\mathsf{KeyGen}(1^k)$: Additionally, we make use of a PRF F which is defined as $\{0,1\}^k \times \{0,1\}^* \to \{0,1\}^k$, and sample a k-bit string sk_3 uniformly at random to serve as the key of F.

$\mathsf{BuildIndex}(K, \mathbf{d})$: After obtaining the basic index structure, the following additional step to encrypt $\boldsymbol{\gamma}$ is added.

v) For each original bit b stored in the index $\boldsymbol{\gamma}[y][j]$ ($y \in [1,m], j \in [1, \#\mathbf{d}]$), we store its encrypted form as

$$\boldsymbol{\gamma}[y][j] = b + \zeta_{yj}, \tag{3}$$

where $\zeta_{yj} = F(sk_1, y||j||\mathit{flag})$ and the default value of *flag* is set to 0. As will be described in Section IV-C3, the value of *flag* will be changed to 1 when generating the update token $\boldsymbol{\tau}_d$.

2) Multi-keyword Search Over the Encrypted Index:

$\mathsf{Trapdoor}(sk, \mathbf{q})$: Identical to that in the basic construction.

Now for each multi-keyword query, the server needs to locate all hit encrypted DIVs and return intermediate encrypted results computed on the specified query logics to the client for further processing.

$\mathsf{Query}(\boldsymbol{\gamma}, \mathbf{c}, \boldsymbol{\tau})$: Use $\boldsymbol{\tau}$ to compute the intermediate encrypted results according to specified logic ('AND' or 'OR').

(a) *Conjunction logic query*. Locate all the hit encrypted DIVs, compute and return

$$\sum_{y \in \boldsymbol{\tau}, j \in [1, \#\mathbf{d}]} \boldsymbol{\gamma}[y][j]$$

to the client.

(b) *Disjunction logic query*. Locate all the hit encrypted DIVs, compute and return

$$\sum_{y \in \boldsymbol{\tau}[1,r], j \in [1,\#\mathbf{d}]} \boldsymbol{\gamma}[y][j], \ldots, \sum_{y \in \boldsymbol{\tau}[\#\mathbf{q}r-r+1, \#\mathbf{q}r], j \in [1,\#\mathbf{d}]} \boldsymbol{\gamma}[y][j]$$

to the client.

At the client side, it first can re-compute ζ_{yj} for each $y \in \boldsymbol{\tau}$ and $j \in [1, \#\mathbf{d}]$ by leveraging the secret key sk_1. Then according to different query logics:

(a) *Conjunction logic query*. The client locally computes $st_{\mathbf{q}} = (st_{\mathbf{q},1}, \ldots, st_{\mathbf{q},\#\mathbf{d}})$, where

$$st_{\mathbf{q},j} = \sum_{y \in \boldsymbol{\tau}} (1 + \zeta_{yj}) \quad (j \in [1, \#\mathbf{d}]).$$

Afterwards, it initializes an empty set $\mathbb{ID}$, and adds the corresponding identifier $\mathtt{id}(d_j)$ ($j \in [1, \#\mathbf{d}]$) into $\mathbb{ID}$ if

$$\sum_{y \in \boldsymbol{\tau}} \boldsymbol{\gamma}[y][j] = st_{\mathbf{q},j}.$$

(b) *Disjunction logic query*. The client locally computes $st_{\mathbf{q}} = (st_{\mathbf{q},1}, \ldots, st_{\mathbf{q},\#\mathbf{d}})$, where

$$st_{\mathbf{q},j} = (\sum_{y \in \boldsymbol{\tau}[1,r]} \zeta_{yj} + r, \ldots, \sum_{y \in \boldsymbol{\tau}[\#\mathbf{q}r-r+1, \#\mathbf{q}r]} \zeta_{yj} + r).$$

Then, for each $j \in [1, \#\mathbf{d}]$, it computes the similarity score as follows

$$score(d_j, \mathbf{q}) = r \cdot \mathbf{1}(\sum_{y \in \boldsymbol{\tau}[1,r]} \boldsymbol{\gamma}[y][j] = st_{\mathbf{q},j}[1]) + \ldots + r \cdot \mathbf{1}(\sum_{y \in \boldsymbol{\tau}[\#\mathbf{q}r-r+1, \#\mathbf{q}r]} \boldsymbol{\gamma}[y][j] = st_{\mathbf{q},j}[\#\mathbf{q}]).$$

Afterwards, it sorts the scores in a descending order and adds the corresponding identifier $\mathtt{id}(d_j)$ ($j \in [1, \#\mathbf{d}]$) into $\mathbb{ID}$ if $score(d_j, \mathbf{q})$ is larger than a pre-defined threshold.

For both cases, the client sends $\mathbb{ID}$ to the server. At last, the server returns the encrypted data files $\mathbf{c_q}$ back to the client.

The following theorem shows the effectiveness of BEIS-I.

Theorem 1. *BEIS-I using random generator is equivalent to searching over the basic index structure.*

Proof. i) For the conjunction logic query, as shown in Eq. (1), $\mathsf{div}_{\boldsymbol{\tau}}[j] = 1$ iff $\boldsymbol{\gamma}[y][j] = 1$ for each $y \in \boldsymbol{\tau}$. Similarly, in BEIS-I, for $\forall j \in [1, \#\mathbf{d}]$, if there exists $y \in \boldsymbol{\tau}$ such that the original bit $\boldsymbol{\gamma}[y][j]$ is '0', then $\sum_{y \in \boldsymbol{\tau}} \boldsymbol{\gamma}[y][j] < st_{\mathbf{q},j}$ since $\boldsymbol{\gamma}[y][j] = 0 + \zeta_{yj} < 1 + \zeta_{yj}$. Therefore, $\mathtt{id}(d_j)$ is added to $\mathbb{ID}$ if $\sum_{y \in \boldsymbol{\tau}} \boldsymbol{\gamma}[y][j] = st_{\mathbf{q},j}$, *i.e.*, $\boldsymbol{\gamma}[y][j] = 1 \ \forall y \in \boldsymbol{\tau}$.

ii) For the disjunction logic query, let b_{yj} be the original bit stored in $\boldsymbol{\gamma}[y][j]$ ($y \in \boldsymbol{\tau}, j \in [1, \#\mathbf{d}]$), we have

$$\begin{aligned} score(d_j, \mathbf{q}) &= r \cdot \mathbf{1}(\sum_{y \in \boldsymbol{\tau}[1,r]} (b_{ij} + \zeta_{yj}) = \sum_{y \in \boldsymbol{\tau}[1,r]} \zeta_{yj} + r) \\ &\quad + \cdots + r \cdot \mathbf{1}(\sum_{y \in \boldsymbol{\tau}[\#\mathbf{q}r-r+1, \#\mathbf{q}r]} (b_{ij} + \zeta_{yj}) \\ &= \sum_{y \in \boldsymbol{\tau}[\#\mathbf{q}r-r+1, \#\mathbf{q}r]} \zeta_{yj} + r) \\ &= r \cdot \mathbf{1}(\sum_{y \in \{\boldsymbol{\tau}[1,r]\}} b_{ij} = r) + \ldots + \\ &\quad r \cdot \mathbf{1}(\sum_{y \in \boldsymbol{\tau}[\#\mathbf{q}r-r+1, \#\mathbf{q}r]} b_{ij} = r) \end{aligned}$$

It is easy to see that the computation of $score(d_j, \mathbf{q})$ is equivalent to that of basic construction Eq. (2). □

3) Supporting Index Dynamics:

Next, we show our BEIS-I scheme can also support efficient data updates. To this end, based on the basic index construction, the following modifications are made.

$\mathsf{UpdateToken}(K, d_j)$: To add a new file d_j, the client encrypts each bit of the newly-obtained sub-index $\boldsymbol{\gamma}^*$ in a similar manner processed in $\mathsf{BuildIndex}$, then outputs the encrypted sub-index $\boldsymbol{\gamma}^*$ and c_j as $\boldsymbol{\tau}_a$. To delete an existing file d_j, the client generates an m-dimension row vector Del, where each entry is initialized to 0. Then it generates the encrypted version by computing $\mathsf{Del}_j[i] = 0 + \zeta_{ij}$, where $i \in [1, m]$ and $\zeta_{ij} = F(sk_1, i||j||\mathit{flag} + 1)$. Finally, it outputs the encrypted row vector Del as $\boldsymbol{\tau}_d$.

$\mathsf{Update}(\boldsymbol{\tau}_a/\boldsymbol{\tau}_d, \boldsymbol{\gamma}, \mathbf{c})$: For file addition, append the encrypted sub-index $\boldsymbol{\gamma}^*$ extracted from $\boldsymbol{\tau}_a$ to $\boldsymbol{\gamma}$ and c_j into $\mathbf{c}$, respectively. For file deletion, set the jth row of $\boldsymbol{\gamma}$ to the encrypted row vector Del extracted from $\boldsymbol{\tau}_d$ and delete c_j from $\mathbf{c}$ accordingly.

D. BEIS-II: Using Homomorphic Generator

To further reduce the communication cost, we next introduce BEIS-II by leveraging Paillier cryptosystem to encrypt DIVs stored in each bucket. Due to the additively homomorphic property, multiple hit DIVs can be aggregated directly at the server side, although in an encrypted form.

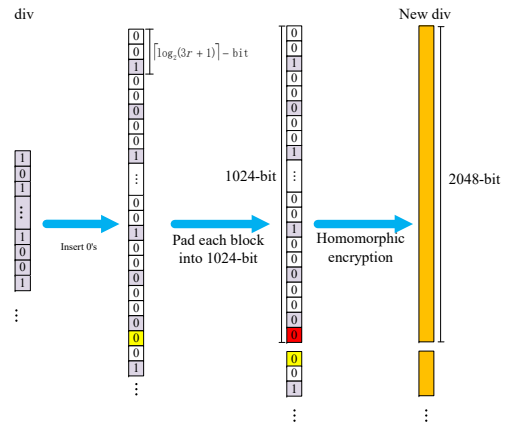


Fig. 1: An example of building index

Ciphertext Packing. It is worth noting that the typical parameters for Paillier cryptosystem can support big (*e.g.*, 1024-bit) integers as the plaintext space. In our setting, we want to encrypt bit vectors (*i.e.*, DIVs) of length $\#\mathbf{d}$, and therefore need to pack each bit vector into multiple 1024-bit integers. To ensure that the summation of bit values in one row across r different buckets will not affect the aggregated values of other rows during the query phase, we need to allocate at least $\lceil \log_2(r+1) \rceil$ bits for each original bit, *i.e.*, insert $(\lceil \log_2(r+1) \rceil - 1)$-bit of 0's before each bit. In addition, due to the fact that our deletion mechanism requires using an integer (instead of r) to indicate a certain file that has been deleted, more extra 0's need to be inserted before each bit. In particular, for removal of an existing data file d_j, we will set the second to last entry to 1 in an 1024-bit 0 vector corresponding to d_j and generate its encrypted version. Afterwards, this tailor-made encrypted vector will be used to perform additive homomorphic operations on all the DIVs stored in the buckets. Upon finishing the updates, the last two entries corresponding to d_j in all the buckets are modified to either "10" or "11". Then for any new future query, the decimal value of entries corresponding to d_j in the aggregated form will be turned to $3r$ or $2r$, which would not incur a "collision", *i.e.*, affect the correctness of checking equality for a new query. In light of the above observations, we will use a block of $\lceil \log_2(3r+1) \rceil$ bits in the plaintext space to represent a bit in the DIV, that is, pad $(\lceil \log_2(3r+1) \rceil - 1)$-bit of 0's before each original bit. Besides, the remaining bits which are insufficient to form a block are set to 0. At last, the packed integers are encrypted by the Paillier cryptosystem.

1) Initialization and Index Construction:

$\mathsf{KeyGen}(1^k)$: Generate (sk_p, pk_p) for the Paillier cryptosystem, and add this new key tuple to K.

$\mathsf{BuildIndex}(K, \mathbf{d})$: Based on the basic index structure, additional steps are performed as follows:

v) As shown in Fig. 1, extract div stored in the $\boldsymbol{\gamma}[y]$ $(y \in [1,m])$, and construct $\widehat{\text{div}}$ by inserting $(\lceil \log_2(3r+1) \rceil - 1)$-bit 0's before each bit of div. Pack $\widehat{\text{div}}$ into multiple 1024-bit plaintext blocks, and use Paillier cryptosystem to encrypt each block under the public key pk_p. Store the newly-obtained div $= [\![block_{y,1}]\!] || \ldots || [\![block_{y,v}]\!]$ into $\boldsymbol{\gamma}[y]$, where v denotes the maximum number of blocks.

2) Multi-keyword Search Over the Encrypted Index:

$\mathsf{Trapdoor}(sk, \mathbf{q})$: Identical to that in the basic construction.

After locating the hit buckets, the server homomorphically sums up these encrypted DIVs and returns the intermediate result of aggregation to the client.

$\mathsf{Query}(\boldsymbol{\gamma}, \mathbf{c}, \boldsymbol{\tau})$: Use $\boldsymbol{\tau}$ to compute the intermediate encrypted results as follows. For $t \in [1, \#\mathbf{q}]$, compute

$$\theta_t = \prod_{y \in \boldsymbol{\tau}[1,r]} [\![block_{y,1}]\!] || \ldots || \prod_{y \in \boldsymbol{\tau}[\#\mathbf{q}r - r + 1, \#\mathbf{q}r]} [\![block_{y,v}]\!],$$

and return $\Theta = \{\theta_1, \ldots, \theta_{\#\mathbf{q}}\}$ to the client.

Afterwards, the client decrypts the intermediate ciphertext Θ using sk_p, unpacks the blocks, and parses the decryption result as a concatenation of decimal values based on the aligned configuration of the above packing technique, *i.e.*, convert the decrypted binary string θ_t to $B_t = (b_1 \ldots b_{\#\mathbf{d}})_{10}$ $(t \in [1, \#\mathbf{q}])$, where b_i is the binary representation of a $\lceil \log_2(3r+1) \rceil$-bit integer.

(a) *Conjunction logic query*. Add the corresponding $\mathsf{id}(d_j)$ into $\mathbb{ID}$ iff $\forall t \in [1, \#\mathbf{q}]$, $B_t[j] = r$.

(b) *Disjunction logic query*. For each $j \in [1, \#\mathbf{d}]$, compute the similarity score

$$score(d_j, \mathbf{q}) = r \cdot \mathbf{1}(B_1[j] = r) + \cdots + r \cdot \mathbf{1}(B_{\#\mathbf{q}}[j] = r),$$

sort the scores in a descending order, and adds the corresponding $\mathtt{id}(d_j)$ into $\mathbb{ID}$ if $score(d_j, \mathbf{q})$ is larger than a pre-defined threshold.

Discussions. In practice, the number of keywords in a query, *i.e.*, $\#\mathbf{q}$, is usually limited. Motivated by this observation, we can insert $(\lceil \log_2(3\#\mathbf{q}r + 1) \rceil - 1)$-bit 0's before each bit in $\mathsf{BuildIndex}$ algorithm to adapt to the summation of $\#\mathbf{q}r$ "11"or "00". Hence, for the conjunction logic query, the output of Query can be further aggregated into a single encrypted result by computing $\Theta = \theta_1 \theta_2 \cdots \theta_{\#\mathbf{q}}$. At last, it adds the corresponding $\mathsf{id}(d_j)$ into $\mathbb{ID}$ by checking whether the decrypted decimal value equals to $\#\mathbf{q}r$.

Theorem 2. *BEIS-II using homomorphic generator is equivalent to searching over the basic index construction.*

Proof. The correctness proof is similar to the proof for Theorem 1. Details are omitted due to the space limitation. □

3) Supporting Index Dynamics:

Finally, we show (batch) file updates can be realized by our BEIS-II scheme. To this end, the following adaptions are made based on the basic construction.

$\mathsf{UpdateToken}(K, d_j)$: To add a new file d_j, for the sub-index $\boldsymbol{\gamma}^*$ generated in the basic construction, the client first pads it (*i.e.*, every single bit in m buckets) to binary strings of 1024 bits using the same packing method, and then encrypts them using Paillier cryptosystem under the pk_p. The encrypted sub-index $\boldsymbol{\gamma}^*$ and ciphertext c_j are output as $\boldsymbol{\tau}_a$. To delete an existing file d_j, the client initializes a 1024-bit 0 vector Del, sets the $((j \mod \lfloor \frac{1024}{\lceil \log_2(3r+1) \rceil} \rfloor) \cdot \lceil \log_2(3r+1) \rceil - 1)$-th entry to 1, encrypts it by leveraging Paillier homomorphic encryption, and then outputs the encrypted Del as $\boldsymbol{\tau}_d$.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2018.2818651, IEEE Transactions on Information Forensics and Security

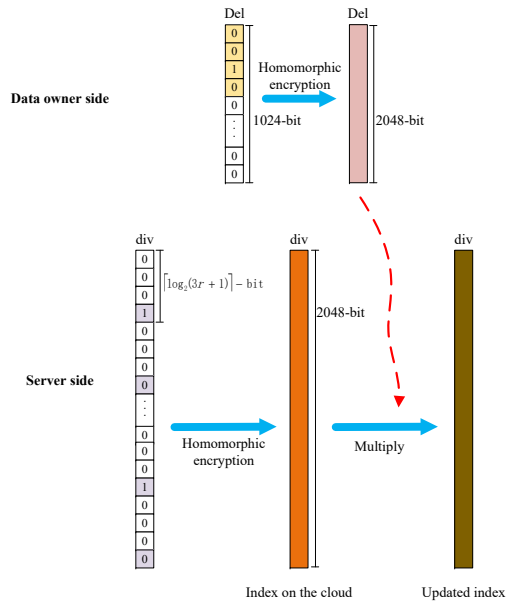


Fig. 2: An example of deleting a file

$\mathsf{Update}(\boldsymbol{\tau}_a/\boldsymbol{\tau}_d, \boldsymbol{\gamma}, \mathbf{c})$: Procedures for adding a new file d_j are the same as those in the basic construction. For file deletion, for the $\frac{j \cdot \lceil \log_2(3r+1) \rceil}{1024}$-th ciphertext block in all the buckets $\boldsymbol{\gamma}[y]$ $(y \in [1, m])$ as well as encrypted Del extracted from $\boldsymbol{\tau}_d$, the server computes $\boldsymbol{\gamma}[y] \leftarrow \boldsymbol{\gamma}[y] \cdot \mathsf{Del}$ by resorting to the additively homomorphic property and deletes c_j from $\mathbf{c}$. This works since the decimal value of entries corresponding to d_j is changed from r (or 0) to $3r$ (or $2r$).

Fig. 2 gives an illustrative example for the process of file deletion. Assume that the client wants to delete file d_1, which is implied in the first ciphertext block of each div. To do so, the client initializes a 1024-bit vector $\mathsf{Del} = (0010, 00\ldots)$ and encrypts it with homomorphic encryption. After receiving the Del, the server multiplies it with the first block of each div to complete the update.

Due to the fact that the message space of Paillier cryptosystem is always greater than the space allocated for a data file $(i.e., 1024 \gg \lceil \log_2(3r+1) \rceil)$, and therefore updates can be achieved in a batch manner, that is, a collection of data files can be added and removed simultaneously. Taking the removal of d_1 and d_2 for example, the client can generate a tailored Del vector with the form of $(0010, 0010, 00\ldots)$.

V. Theoretical Analysis and Experiments

A. Security Analysis

Both of our constructions that we discuss in this work leak limited information. Now we proceed to describe and formalize these leakages below.

- (*Leakage function $\mathcal{L}_1$ for setup*). Given the collection of data files $\mathbf{d}$, $\mathcal{L}_1(\mathbf{d}) = \{\#\mathbf{d}, [|d_i|]_1^{\#\mathbf{d}}, [\mathtt{id}(d_i)]_1^{\#\mathbf{d}}, m\}$, where $\#\mathbf{d}$ denotes the number of files, $|d_i|$ and $\mathtt{id}(d_i)$ denote the size and the identifier of file d_i, respectively, and m denotes the number of buckets.
- (*Leakage function $\mathcal{L}_2$ for search*). Given the collection of data files $\mathbf{d}$ and a multi-keyword query $\mathbf{q}$, $\mathcal{L}_2(\mathbf{d}, \mathbf{q}) = \{\pi(\mathbf{Q}), A_p(\mathbf{d}, \mathbf{q}), \mathsf{IP}\}$, where $\pi(\mathbf{Q})$ denotes the search pattern, $A_p(\mathbf{d}, \mathbf{q})$ denotes the access pattern and IP denotes the intersection pattern, all of which are described in the following definitions.

Definition 3. *(Search Pattern π). The search pattern leakage reveals whether the keywords in the query have appeared before. Formally, for two queries $\mathbf{q}, \mathbf{q}'$, define $\mathsf{sim}(\mathbf{q}, \mathbf{q}') = \{(i,j) | \mathbf{q}[i] = \mathbf{q}'[j], i \in [1, \#\mathbf{q}], j \in [1, \#\mathbf{q}']\}$, i.e., whether one of the keywords in $\mathbf{q}$ matches with the other one in $\mathbf{q}'$. Let $\mathbf{Q} = \{\mathbf{q}_1, \ldots, \mathbf{q}_\delta\}$ be a sequence of δ queries, its induced search pattern leakage $\pi(\mathbf{Q})$ is a $\delta \times \delta$ symmetric matrix with the entry (i,j) equals $\mathsf{sim}(\mathbf{q}_i, \mathbf{q}_j)$.*

Definition 4. *(Access Pattern A_p). Given a search query $\mathbf{q}$, the access pattern is defined as $A_p(\mathbf{d}, \mathbf{q}) = \{\mathtt{id}(\mathbf{c_q})\}$, i.e., the identifiers of data files in the result set $\mathbf{c_q}$.*

Definition 5. *(Intersection Pattern IP). Given $\mathbf{Q} = \{\mathbf{q}_1, \ldots, \mathbf{q}_\delta\}$, the intersection pattern IP is formally a $\delta \times \delta$ symmetric matrix defined by*

$$\mathsf{IP}[i,j] = \begin{cases} \mathbb{ID}(\mathbf{q}_i) \bigcap \mathbb{ID}(\mathbf{q}_j) & \textit{Conjunctive queries } \mathbf{q}_i \neq \mathbf{q}_j \\ \mathbb{ID}(\mathbf{q}_i) \bigcup \mathbb{ID}(\mathbf{q}_j) & \textit{Disjunctive queries } \mathbf{q}_i \neq \mathbf{q}_j \end{cases}$$

Informally, this part of leakage which can be derived from the access pattern means that for two different conjunctive/disjunctive queries (keywords in these two queries are not exactly the same), the set of data files (*i.e.*, identifiers) matching a new conjunctive/disjunctive query that contains all the keywords in these two issued queries is leaked (if no data file matching the new query then nothing is leaked). It can be seen as the price we pay for the rich functionality that allows for the computation of boolean queries.

Moreover, both of our constructions achieve *forward privacy* which is first explicitly introduced in [26] for dynamic SSE schemes. Intuitively, forward privacy means that an add update does not leak any information about the updated keywords. In other words, it guarantees that an adversarial server cannot determine whether a newly-added data file contains any keyword in any previously authorized query. Forward privacy is a strong security property that can be parameterized by carefully defining the leakage. To formalize it, we capture forward privacy in leakage function $\mathcal{L}_3$ for updates, which is also the case in the definition by Bost [2].

Definition 6. *(Forward Privacy). The above $\mathcal{L}$-adaptively-secure SSE scheme is forward private iff the leakage function $\mathcal{L}_3$ for update can be written as $\mathcal{L}_3(\mathbf{d}, d) = \{\mathtt{op}, |d|, \mathtt{id}(d)\}$, where op denotes the type of the update operation, i.e., addition or deletion, $|d|$ and $\mathtt{id}(d)$ denote the size and the identifier of the file d to be updated, respectively.*

Analogous to the work [2], the definition given here does not limit the types of the update, that is, we consider forward privacy for not only additions, but also deletions. As described in the file addition, the sub-index, one component of the update token $\boldsymbol{\tau}_a$, is constructed in a similar way as that in BuildIndex, namely, every entry in all m buckets is encrypted by either a random number or the Paillier cryptosystem. For file deletion, the encrypted update token $\boldsymbol{\tau}_d$ is applied to all m buckets in

a row representing the specified file. Therefore, beyond the stated leakage information (in Definition 6), nothing about the previously queried keywords is leaked if the underlying encryption schemes are secure.

Theorem 3. *Assuming the used symmetric-key encryption scheme* SKE *is CPA-secure, and the existence of one-way functions (resp., the decisional composite residuosity problem is hard), then BEIS-I (resp., BEIS-II) is CKA2-secure in the standard model.*

Proof. The only difference between BEIS-I and BEIS-II lies in how the DIVs are encrypted. In the following proof, we only relies on the fact that both underlying encryptions are CPA-secure without using the homomorphic property. Therefore, the proofs for both schemes are identical.

Now we proceed to prove the CKA2 security by describing a PPT simulator $\mathcal{S}$ such that for any PPT adversary $\mathcal{A}$, the outputs of $\mathbf{Real}_{\mathcal{A}}(k)$ and $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(k)$ are indistinguishable. **[Setup]** Given $\mathcal{L}_1(\mathbf{d}) = \{\#\mathbf{d}, [|d_i|]_1^{\#\mathbf{d}}, [\mathtt{id}(d_i)]_1^{\#\mathbf{d}}, m\}$, $\mathcal{S}$ simulates the encrypted database as below. It first constructs an array $\widetilde{\gamma}$ with m entries (buckets), where each entry stores an encryption of 0's by using the underlying encryption schemes (*i.e.*, either the outputs of a PRF or the Paillier cryptosystem). Here, the number of 0's encrypted is determined by the number of files $\#\mathbf{d}$ (and the packing technique presented in the construction for BEIS-II). Afterwards, it runs $sk_2' \leftarrow \mathsf{SKE.Gen}(1^k)$ and $c_i' \leftarrow \mathsf{SKE.Enc}(sk_2', \{0,1\}^{|d_i|})$ for $i \in [1, \#\mathbf{d}]$. Finally, it outputs $(\widetilde{\gamma}, \widetilde{\mathbf{c}})$, where $\widetilde{\mathbf{c}} = (c_1', \ldots, c_{\#\mathbf{d}}')$. $\mathcal{S}$ maintains internally a mapping from the m bucket positions to the m l-bit random strings.

[Simulating Searches] Given $\mathcal{L}_2(\mathbf{d}, \mathbf{q}) = \{\pi(\mathbf{Q}), A_p(\mathbf{d}, \mathbf{q}), \mathsf{IP}\}$, $\mathcal{S}$ simulates the search processes as follows. On the basis of search pattern, $\mathcal{S}$ first checks if any of the keyword in $\mathbf{q}$ has appeared before, and if so, it sets the keyword to match r l-bit random strings (*i.e.*, bucket positions) that are previously used, where r is the number of independent hash functions used for the rank. Otherwise, $\mathcal{S}$ chooses r out of m l-bit random strings (*i.e.*, bucket positions) as a new cluster previously unused to represent the keyword. Afterwards, $\mathcal{S}$ sends all the l-bit random strings corresponding to these clusters of bucket positions (representing the keywords in the query) to $\mathcal{A}$. Upon verifying the response from $\mathcal{A}$ against the index $\widetilde{\gamma}$, that is, $\mathcal{A}$ returns back the hit encryptions of 0's, $\mathcal{S}$ sends the file identifiers given in the access pattern $A_p(\mathbf{d}, \mathbf{q})$ to $\mathcal{A}$, who then returns the corresponding ciphertexts.

[Simulating Updates] For add updates, $\mathcal{S}$ is given $\mathcal{L}_3(\mathbf{d}, d) = \{\mathtt{op}, |d|, \mathtt{id}(d)\}$ ($\mathtt{op}$ is specified as addition), which are simulated as in the setup phase. For delete updates, $\mathcal{S}$ is given $\mathcal{L}_3(\mathbf{d}, d) = \{\mathtt{op}, |d|, \mathtt{id}(d)\}$, where $\mathtt{op}$ is specified as deletion. $\mathcal{S}$ simulates a fresh $\widetilde{\mathsf{Del}}$ as in the generation of update token, and sends $\widetilde{\mathsf{Del}}$ representing the file with identifier $\mathtt{id}(d)$ to $\mathcal{A}$.

In summary, the indistinguishability between $\mathbf{Real}_{\mathcal{A}}(k)$ and $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(k)$ follows from the CPA security of SKE as well as the underlying encryption schemes (used in BEIS-I and BEIS-II), and the pseudorandomness of the PRF. □

Specifically, both of our schemes only return aggregated encrypted DIVs (as intermediate results) hit by the search token, instead of immediately retrieving the final encrypted results for the client. That is why the simulator $\mathcal{S}$ can just return a set of random strings (representing the bucket positions) since the adversary cannot distinguish without the corresponding secret key for decryption. Besides, for retrieving the encrypted results (sent back to the client) at the second round, the simulator does not need to work on the encrypted index. Instead, the information available from the leakage function already allows it to do that. In other words, the adaptive security of our schemes can be achieved in the standard model due to the non-trivial communication overhead (*i.e.*, interactive solutions).

B. Query Accuracy Analysis

In our BEIS constructions, there exist a small number of false positives due to probabilistic encodings/decodings. Fortunately, we show that the probability of false decodings can be reduced to almost a negligible level by properly choosing system parameters (m, r). Formally, the false positive rate of a query can be naturally defined as the ratio of the amount of *falsely returned* encrypted data files to the number of all encrypted data files returned from the cloud server. We have the following theorem to characterize the false positive rate $\neg p_{\mathbf{q}}$.

Theorem 4. *The false positive rate of a query* $\mathbf{q}$ *is*

$$\neg p_{\mathbf{q}} = \begin{cases} \frac{exp}{\phi(\bigwedge_{w_i \in \mathbf{q}} \mathsf{div}_i) + exp}, & \text{for the conjunction logic} \\ \frac{exp}{\phi(\bigvee_{w_i \in \mathbf{q}} \mathsf{div}_i) + exp}, & \text{for the disjunction logic,} \end{cases} \tag{4}$$

where $\phi(\bigwedge_{w_i \in \mathbf{q}} \mathsf{div}_i)$ *and* $\phi(\bigvee_{w_i \in \mathbf{q}} \mathsf{div}_i)$ *denote the number of "1"s in* $\bigwedge_{w_i \in \mathbf{q}} \mathsf{div}_i$ *and* $\bigvee_{w_i \in \mathbf{q}} \mathsf{div}_i$*, respectively, and* exp *denotes the expectation of the number of the falsely returned data files for a given query* $\mathbf{q}$*.*

Proof. We compute exp by first discussing the tricky case where $\mathbf{q} \subset \mathbf{w}$. Essentially, the false returning of each encrypted data is independent of other encrypted data files. Thus, *for each data* $d_j \in \mathbf{d}$*, we use* p_j *to denote the probability that* d_j *is falsely returned.*

For ease of presentation, we start from a single keyword w_i ($w_i \in \mathbf{q}$ but $w_i \notin d_j$). Since $w_i \in \mathbf{w}$, div_i should have been encoded into the r buckets $\boldsymbol{\gamma}[h_i(w_i, sk_1)]$ for $i \in [1, r]$. Based on our bucket based encoding/decoding method, The false decoding of div_i happens *if and only if* all r buckets are encoded by data identifier vectors of other keywords besides that of w_i. Without loss of generality, we assume there are r additional data identifier vectors $\{\widehat{\mathsf{div}}_1, \ldots, \widehat{\mathsf{div}}_r\}$ which are repeatedly encoded into the r buckets $\{\boldsymbol{\gamma}[h_1(w_i, sk_1)], \ldots, \boldsymbol{\gamma}[h_r(w_i, sk_1)]\}$, respectively. Here, each $\widehat{\mathsf{div}}_i$ might be the joint of multiple data identifier vectors. In the decoding process, $\mathsf{div}_{\boldsymbol{\tau}}$ is decoded from the r buckets $\{\boldsymbol{\gamma}[h_1(w_i, sk_1)], \ldots, \boldsymbol{\gamma}[h_r(w_i, sk_1)]\} = \{\mathsf{div}_i \bigvee \widehat{\mathsf{div}}_1, \ldots, \mathsf{div}_i \bigvee \widehat{\mathsf{div}}_r\}$. We denote $\hat{p}_j$ as the probability that w_i is falsely considered as a membership of d_j. Let n_j be the number of distinct keywords in d_j, we have

$$\hat{p}_j \quad = \quad (1 - (1 - \frac{1}{m})^{r n_j})^r \approx (1 - e^{\frac{-r n_j}{m}})^r.$$

Now we discuss p_j as follows: 1) The conjunction logic query. For $\forall d_j \in \mathbf{d}$, d_j will be falsely returned if and only

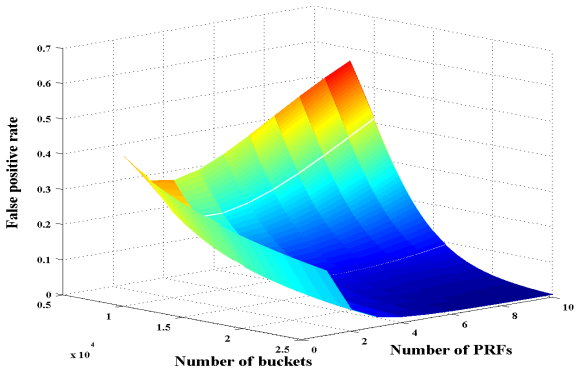


Fig. 3: The false positive rate of the query results

if all keywords in $\mathbf{q}$ are hit by d_j while there exists at least one keyword in $\mathbf{q}$ which is actually not contained in d_j. By contradiction, d_j will not be falsely returned if there always exists a keyword in $\mathbf{q}$ which is not contained and hit by d_j. Thus, we have $p_j = 1-(1-\hat{p}_j) = \hat{p}_j$. 2) The disjunction logic query. Similarly, for $\forall d_j \in \mathbf{d}$, d_j will be falsely returned if there exists at least one keyword in $\mathbf{q}$ that is hit by d_j, where all keywords in $\mathbf{q}$ are not contained in d_j. By contradiction, d_j will not be falsely returned if none of the keywords in $\mathbf{q}$ is hit by d_j. Thus, we have $p_j = 1-(1-\hat{p}_j)^{\#\mathbf{q}}$.

Then we can compute the expectation as

$$exp = \sum_{d_j \in \hat{\mathbf{d}}} p_j,$$

where $\hat{\mathbf{d}}$ denotes the collection of data files which might be falsely returned. In the conjunction logic query, $\#\hat{\mathbf{d}} = \#\mathbf{d} - \phi\{\bigwedge_{w_i \in \mathbf{q}} \mathsf{div}_i\}$, and $\#\hat{\mathbf{d}} = \#\mathbf{d} - \phi\{\bigvee_{w_i \in \mathbf{q}} \mathsf{div}_i\}$ in the disjunction logic query. When considering the case that $w_i \notin \mathbf{w}$ and the false positive decoding happens, the returning of any data files is a false decoding. In this case, we compute exp over $\mathbf{d}$, *i.e.*, $\hat{\mathbf{d}} = \mathbf{d}$. Based on exp, we can easily compute $\neg p_{\mathbf{q}}$ as Eq. (4). □

Fig. 3 shows the experimental evaluation of the false positive rate of the query results. In the experiment, we set $\#\mathbf{d} = 10^5$ with $\#\mathbf{w} = 8000$. Then, we first build different indices by using different values of m and r. For each index, we execute two types of queries over the index and each data point in the figure is the average of 10^3 queries. As shown from the results, the false positive rate is extremely small or even negligible by choosing suitable m and r. Note that, to obtain a low and acceptable false positive rate, it is required to properly choose different settings of m and r when $\#\mathbf{d}$ and $\#\mathbf{w}$ vary much. To get rid of rescaling the whole index during the update process, we conservatively set the parameters m and r to be relatively large values at the cost of minor variation in false positive rate. From a practical point of view, it implies our index constructions achieve desirable privacy guarantee without sacrificing the query accuracy and space efficiency much.

C. Performance Analysis

For the search time efficiency, $\#\mathbf{q}r$ hash operations are used to compute the bucket positions at the client side for both schemes. At the server side, for BEIS-I, add operations are applied to compute the summation of entries in DIVs which are hit by the the search token $\boldsymbol{\tau}$. Afterwards, the client re-computes the PRF and involves $\#\mathbf{d}$ (*resp.* $\#\mathbf{d} \cdot \#\mathbf{q}$) checking operations for conjunction queries (*resp.* disjunction queries) to evaluate the equality with the intermediate results. While BEIS-II needs to aggregate $\frac{\#\mathbf{d} \cdot \lceil \log_2(r+1) \rceil \cdot \#\mathbf{q}(r-1)}{1024}$ encrypted DIVs via additive homomorphic operations, then perform $\frac{\#\mathbf{d} \cdot \lceil \log_2(r+1) \rceil \cdot \#\mathbf{q}}{1024}$ decryption operations and $\#\mathbf{d} \cdot \#\mathbf{q}$ equality checking operations at the client side.

Although the searching time complexity for both schemes is theoretically $O(\#\mathbf{d})$ which is as good as previous solutions such as [1], [4], [6], [11], [21], [27], we still achieve practical efficiency. The computations involved in BEIS-I are hash and elementary operations, which are practically efficient. For BEIS-II, due to the use of packing technique, instead of $\#\mathbf{d}$ ciphertexts, only a fraction of it is being processed. Our experimental results also show that both homomorphic aggregation of ciphertexts at the server side and the decryption of aggregated (intermediate) results at the client side attain practical efficiency.

While the de facto standard of SSE efficiency is sub-linear in $\#\mathbf{d}$, most of these state-of-the-art solutions only work for single keyword search without supporting more complex queries (*e.g.*, conjunction and disjunction queries). Besides, it is worth noting that both our schemes are highly parallelizable; namely, during the search process, the hit data identifier vectors can be effectively split into multiple segments, each of which can be processed in the same manner by a separate thread (or process) in parallel.

The communication cost, as described above, is also theoretically $O(\#\mathbf{d})$ for both schemes. Note that this part of cost is dominated by the intermediate encrypted results returned to the client. Specifically, for BEIS-I, it requires, respectively, about 0.97MB for conjunction logic queries and about 4.83MB for disjunction logic queries to transmit the intermediate summations, when $\#\mathbf{d} = 100000$, $\#\mathbf{q} = 5$, $r = 5$ and ζ's are 80-bit random numbers. For BEIS-II, the aggregated (intermediate) ciphertexts Θ which need to be transferred are linear in $\#\mathbf{d}$. With Paillier modulus being 2048-bit as an additional configuration, the bandwidth consumption for both conjunction and disjunction logic queries is about 0.36MB. It is obvious that these communication overheads are all practically acceptable.

For the storage cost of the encrypted index, the theoretical complexity of our schemes is $O(\#\mathbf{d} \cdot m)$, where m is the number of buckets. While most of the state-of-the-art solutions such as [6], [12], [17] have the complexity of $O(N)$, where N is the total number of document-keyword pairs. In fact, the number N can be computed as $\sum_{i=1}^{\#\mathbf{d}} |d_i|$, and it is equivalent to $\#\mathbf{d} \cdot c$, where $|d_i|$ denotes the number of unique keywords in data file d_i, and c is a constant that is always smaller than $\#\mathbf{w}$. Hence, our schemes incur larger storage overhead due to the fact that m is larger than $\#\mathbf{w}$.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2018.2818651, IEEE Transactions on Information Forensics and Security

VI. Experimental Evaluation

In this section, we implement and evaluate our SSE schemes using a large representative dataset. We first summarize in Table I the differences between our schemes and other notable searchable encryption schemes in the literature. Our BEIS schemes can execute multi-keyword boolean search while achieving *forward privacy*, a strong privacy guarantee that the server cannot learn whether or not a newly added file contains a keyword we searched for in the past. In addition, the search tokens/keyword hashes of all keywords in the new file are not leaked. In BEIS-II, we can trade-off the time efficiency of file additions for *forward privacy*. As shown in Fig. 6, our experiments show that if *forward privacy* is not required, much more efficient file additions are obtained. As for file deletions, both BEIS-I and BEIS-II do not leak additional information.

The following experiments are implemented in Java 7. Either the operations by the server, or operations performed by the client, were executed on an Intel Core i5-4440M 2.80GHz CPU with 12GB RAM running on Windows 8. To minimize the I/O access time, we randomly generate $\#\mathbf{d}$ $\#\mathbf{w}$-bit $\{0,1\}$ vectors to simulate the database in the main memory during the experiments. As $\#\mathbf{d}$ denotes the number of data files (vectors) in the database, the i-th bit of the vector accounts for keyword w_i ($i \in \{1, \ldots, \#\mathbf{w}\}$). Specifically, if the bit at position i of a vector is 1, it means this file contains the keyword w_i. Note that, all the real-world datasets can be pre-processed to such format. In our experiment, we choose $\#\mathbf{d}$ ranging from 2^{13} to 2^{17}, and we choose $\#\mathbf{w}$ ranging from 6000 to 10000. In the following discussion, we use BEIS-I and BEIS-II to denote the dynamic BEIS with random generator and the BEIS with homomorphic generator, respectively.

In the experiments, we set ζ's to be 80-bit random values to encrypt the index in BEIS-I and Paillier homomorphic encryption system (with Paillier modulus is 1024 bits or 2048 bits) to encrypt the index in BEIS-II, respectively. In addition, we use the HMAC-SHA1 for PRFs. For simplicity, we omitted time measurements for encrypting and decrypting the original data files, since the time costs of such operations are well studied.

Fig. 4 shows the time costs of index construction process, which is a one-time cost. In the left figure, we set $\#\mathbf{w} = 8000$ while in the right figure, we set $\#\mathbf{d} = 2^{15}$. In BEIS-I, during the index construction, we use an 80-bit random number to mask each entry in div's, which leads to only a small computation cost, while during the index construction of BEIS-II, the use of Paillier homomorphic encryption system to encrypt each div leads to relatively higher computation cost.

Fig. 5 shows the query performance of our BEIS schemes. Note that, the time cost of query process consists of the time cost of generating trapdoors (tokens) on the client side and the time cost of file searching on the server side. Note that our BEIS schemes can naturally support multi-keyword search, without needing to post-process all results of single-keyword queries. In the left figure, we evaluate the single keyword query performance with the increase of $\#\mathbf{d}$. In the right figure, we evaluate the performance of the multi-keyword query. Since our solutions have nearly the same computation cost for disjunction logic and conjunction logic, and thus for simplicity, we only evaluate the performance of the multi-keyword query in 'AND' logic.

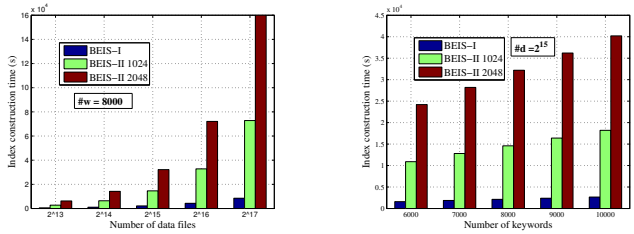


Fig. 4: The average time cost of index construction

Fig. 6 shows the time costs of addition and deletion operations over BEIS-I and BEIS-II. Obviously, the costs of file updates for BEIS-I increase with the number of data files to be updated, and both addition and deletion operations are very efficient. However, due to the unique encryptions of the index in BEIS-II, the time costs of file updates remain invariant. That implies that our BEIS-II better applies to updates of a bunch of data files simultaneously. During the process of file addition, we measure the time costs of BEIS-II with and without achieving *forward privacy* respectively. For the latter case, we do not need to generate m encrypted value for every bucket. In fact, for each file to be inserted, the client can only compute a single encrypted value of a 1024-bit vector and send the 2048-bit ciphertext to the server together with the position information of the buckets to be changed. We can greatly reduce the time costs for generating adding token if *forward privacy* is not required in practice.

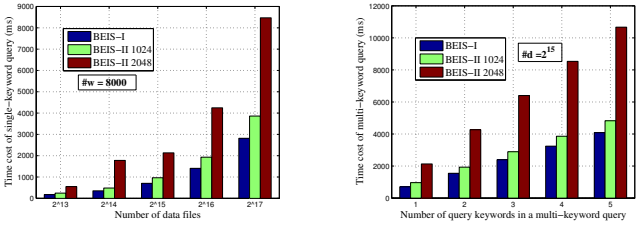


Fig. 5: The average time cost of query process

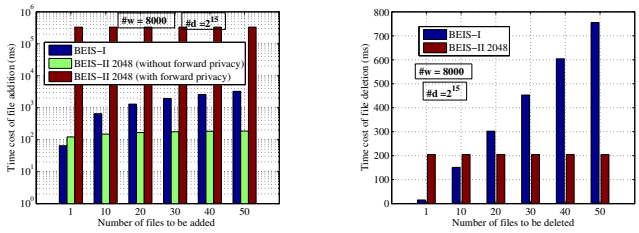


Fig. 6: The average time cost of file updates

VII. Conclusion

In this paper, we introduced a suite of new and novel SSE index designs for processing queries over large-scale encrypted databases. Our index constructions made trade-offs

Scheme	Multi-KW Search	Search Time	Client State Info.	Index Size	Update Cost	Forward Privacy	Security Strength
RBT-SSE [16]	No	$O(\#\mathbf{d}_w \cdot \log \#\mathbf{d})$	No	$O(\#\mathbf{d} \cdot \#\mathbf{w})$	$O(\#\mathbf{w} \cdot \log \#\mathbf{d})$	No	CKA2
OXT [6]	Yes	$O(\#\mathbf{d}_w)$	S-terms	$O(N + \#\mathbf{w})$	-	No	CKA2
DSSE [17]	No	$O(\#\mathbf{d}_w)$	No	$O(N + \#\mathbf{w})$	$O(\|d\|)$	No	CKA2
SP14 [26]	No	$O(\#\mathbf{d}_w \cdot \log^3 N)$	O-sort	$O(N)$	$O(\|d\| \cdot \log^2 N)$	Yes	CKA2
HK14 [12]	No	$O(N)$	History	$O(N + \#\mathbf{w})$	$O(\|d\|)$	No	CKA2
Sophos [2]	No	$O(\#\mathbf{d}_w)$	Counters	$O(N)$	$O(\|d\|)$	Yes	CKA2
BEIS-I	Yes	$O(\#\mathbf{d})$	No	$O(\#\mathbf{d} \cdot m)$	$O(m)$	Yes	CKA2
BEIS-II	Yes	$O(\#\mathbf{d})$	No	$O(\#\mathbf{d} \cdot m)$	$O(m)$	Yes	CKA2

TABLE I: An overview of complexities for the state-of-the-art searchable encryption schemes in a single-thread and single-keyword query setting. $\#\mathbf{d}_w$ is the number of data files that contain the keyword w, $\#\mathbf{d}$ is the number of data files in the file collection, N is the number of total document-keyword pairs, $\#\mathbf{w}$ is the number of keywords in the keyword collection, $|d|$ is the number of unique keywords in data file d, m is the number of buckets, RBT-SSE [16], DSSE [17] and HK14 [12] require one-round query, OXT [6] and SP14 [26] require multi-round query due to the use of matching protocol and the o-sort operation, respectively. For our schemes, both BEIS-I and BEIS-II require two-round query.

between query efficiency and query privacy, with flexible and comprehensive query functionalities. Through rigorous security analysis under strong security model and extensive experiments on real-world datasets, we demonstrated the effectiveness and practicality of our constructions.

ACKNOWLEDGMENTS

Qian's research is supported in part by National Natural Science Foundation of China (Grant Nos. U1636219, 61373167), the Outstanding Youth Foundation of Hubei Province (Grant No. 2017CFA047), and the Key Program of Natural Science Foundation of Hubei Province (Grant No. 2017CFA007). Jian's research is supported in part by National Key R&D Plan of China (Grant No. 2017YFB0802203), National Natural Science Foundation of China (Grant Nos. U1736203, 61732021), Guangzhou Key Laboratory of Data Security and Privacy Preserving, Guangdong Key Laboratory of Data Security and Privacy Preserving, and National Joint Engineering Research Center of Network Security Detection and Protection Technology. Qian Wang is the corresponding author.

REFERENCES

[1] L. Ballard, S. Kamara, and F. Monrose, "Achieving efficient conjunctive keyword searches over encrypted data," in *Proc. of ICICS'05*. Springer, 2005, pp. 414–426.

[2] R. Bost, "$\sum o\varphi o\varsigma$: Forward secure searchable encryption," in *Proc. of CCS'16*. ACM, 2016, pp. 1143–1154.

[3] R. Bost, B. Minaud, and O. Ohrimenko, "Forward and backward private searchable encryption from constrained cryptographic primitives," in *Proc. of CCS'17*. ACM, 2017, pp. 1465–1482.

[4] N. Cao, C. Wang, M. Li, K. Ren, and W. Lou, "Privacy-preserving multi-keyword ranked search over encrypted cloud data," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 1, pp. 222–233, 2014.

[5] D. Cash, J. Jaeger, S. Jarecki, C. S. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner, "Dynamic searchable encryption in very-large databases: Data structures and implementation," in *Proc. of NDSS'14*, 2014.

[6] D. Cash, S. Jarecki, C. S. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner, "Highly-scalable searchable symmetric encryption with support for boolean queries," in *Proc. of CRYPTO'13*. Springer, 2013, pp. 353–373.

[7] Y. Chang and M. Mitzenmacher, "Privacy preserving keyword searches on remote encrypted data," in *Proc. of ACNS'05*. Springer, 2005, pp. 442–455.

[8] M. Chase and S. Kamara, "Structured encryption and controlled disclosure," in *Proc. of ASIACRYPT'10*. Springer, 2010, pp. 577–594.

[9] R. Curtmola, J. A. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: Improved definitions and efficient constructions," in *Proc. of CCS'06*. ACM, 2006, pp. 79–88.

[10] E.-J. Goh *et al.*, "Secure indexes." *IACR Cryptology ePrint Archive*, vol. 2003, p. 216, 2003.

[11] P. Golle, J. Staddon, and B. Waters, "Secure conjunctive keyword search over encrypted data," in *Proc. of ACNS'04*. Springer, 2004, pp. 31–45.

[12] F. Hahn and F. Kerschbaum, "Searchable encryption with secure and efficient updates," in *Proc. of CCS'14*. ACM, 2014, pp. 310–320.

[13] K. He, J. Chen, R. Du, Q. Wu, G. Xue, and X. Zhang, "Deypos: Deduplicatable dynamic proof of storage for multi-user environments," *IEEE Transactions on Computers*, vol. 65, no. 12, pp. 3631–3645, 2016.

[14] S. Hu, Q. Wang, J. Wang, Z. Qin, and K. Ren, "Securing sift: Privacy-preserving outsourcing computation of feature extractions over encrypted image data," *IEEE Trans. on Image Processing*, vol. 25, no. 7, pp. 3411–3425, 2016.

[15] S. Kamara and T. Moataz, "Boolean searchable symmetric encryption with worst-case sub-linear complexity," in *Proc. of EUROCRYPT'17*. Springer, 2017, pp. 94–124.

[16] S. Kamara and C. Papamanthou, "Parallel and dynamic searchable symmetric encryption," in *Proc. of FC'13*. Springer, 2013, pp. 258–274.

[17] S. Kamara, C. Papamanthou, and T. Roeder, "Dynamic searchable symmetric encryption," in *Proc. of CCS'12*. ACM, 2012, pp. 965–976.

[18] J. Katz and Y. Lindell, *Introduction to modern cryptography*. CRC press, 2014.

[19] K. S. Kim, M. Kim, D. Lee, J. H. Park, and W.-H. Kim, "Forward secure dynamic searchable symmetric encryption with efficient updates," in *Proc. of CCS'17*. ACM, 2017, pp. 1449–1463.

[20] J. Li, Q. Wang, C. Wang, N. Cao, K. Ren, and W. Lou, "Fuzzy keyword search over encrypted data in cloud computing," in *Proc. of INFOCOM'10*. IEEE, 2010, pp. 441–445.

[21] T. Moataz and A. Shikfa, "Boolean symmetric searchable encryption," in *Proc. of AsiaCCS'13*. ACM, 2013, pp. 265–276.

[22] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *Proc. of EUROCRYPT'99*. Springer, 1999, pp. 223–238.

[23] V. Pappas, F. Krell, B. Vo, V. Kolesnikov, T. Malkin, S. G. Choi, W. George, A. Keromytis, and S. Bellovin, "Blind seer: A scalable private DBMS," in *Proc. of S&P'14*. IEEE, 2014, pp. 359–374.

[24] J. Shen, J. Shen, X. Chen, X. Huang, and W. Susilo, "An efficient public auditing protocol with novel dynamic structure for cloud data," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 10, pp. 2402–2415, 2017.

[25] D. X. Song, D. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in *Proc. of S&P'00*. IEEE, 2000, pp. 44–55.

[26] E. Stefanov, C. Papamanthou, and E. Shi, "Practical dynamic searchable encryption with small leakage," in *Proc. of NDSS'14*, 2014, pp. 23–26.

[27] B. Wang, S. Yu, W. Lou, and Y. T. Hou, "Privacy-preserving multi-keyword fuzzy search over encrypted data in the cloud," in *Proc. of INFOCOM'14*. IEEE, pp. 2112–2120.

[28] C. Wang, K. Ren, J. Wang, and Q. Wang, "Harnessing the cloud for securely outsourcing large-scale systems of linear equations," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 6, pp. 1172–1181, 2013.

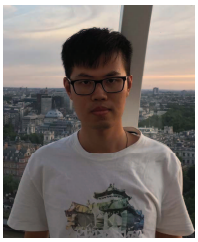
[29] Q. Wang, M. He, M. Du, S. S. M. Chow, R. W. F. Lai, and Q. Zou, "Searchable encryption over feature-rich data," *IEEE Transactions on Dependable and Secure Computing*, vol. PP, pp. 1–1, DOI: 10.1109/TDSC.2016.2 593 444, 2016.

[30] Q. Wang, S. Hu, K. Ren, M. He, M. Du, and Z. Wang, "Cloudbi: Practical privacy-preserving outsourcing of biometric identification in the cloud," in *Proc. of ESORICS'2015*. Springer, 2015, pp. 186–205.

[31] Q. Wang, C. Wang, K. Ren, W. Lou, and J. Li, "Enabling public auditability and data dynamics for storage security in cloud computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 5, pp. 847–859, 2011.

[32] W. K. Wong, D. W.-L. Cheung, B. Kao, and N. Mamoulis, "Secure knn computation on encrypted databases," in *Proc. of SIGMOD'09*. ACM, 2009, pp. 139–152.

[33] Y. Zhang, J. Katz, and C. Papamanthou, "All your queries are belong to us: The power of file-injection attacks on searchable encryption," in *Proc. of USENIX Security'16*, 2016, pp. 707–720.



Minxin Du received the B.S. degree in Computer Science and Technology from Wuhan University, China, in 2015. He is currently working towards the Master degree in the School of Cyber Science and Engineering in Wuhan University. His research interests include cloud computing, information security, and applied cryptography.

Qian Wang is a Professor with the School of Cyber Science and Engineering and School of Computer Science, Wuhan University, China. He received the B.S. degree from Wuhan University, China, in 2003, the M.S. degree from Shanghai Institute of Microsystem and Information Technology (SIMIT), Chinese Academy of Sciences, China, in 2006, and the Ph.D. degree from Illinois Institute of Technology, USA, in 2012, all in Electrical Engineering. His research interests include AI security, data storage, search and computation outsourcing security and privacy, wireless systems security, big data security and privacy, and applied cryptography etc. Qian is an expert under National "1000 Young Talents Program" of China. He is a recipient of IEEE Asia-Pacific Outstanding Young Researcher Award 2016. He is also a co-recipient of several Best Paper and Best Student Paper Awards from IEEE ICDCS'17, IEEE TrustCom'16, WAIM'14, and IEEE ICNP'11 etc. He serves as Associate Editors for IEEE Transactions on Dependable and Secure Computing (TDSC) and IEEE Transactions on Information Forensics and Security (TIFS). He is a Member of the IEEE and a Member of the ACM.

Meiqi He received the B.S. degree in Information Security from Wuhan University, China, in 2015. She is currently working towards the Ph.D. degree in the Computer Science Department at The University of Hong Kong. Her research interests include cloud computing, information security, applied cryptography and bioinformatics.

Jian Weng received the M.S. and B.S. degrees in computer science and engineering from South China University of Technology, in 2004 and 2000, respectively, and the Ph.D. degree in computer science and engineering from Shanghai Jiao Tong University, in 2008. From April 2008 to March 2010, he was a postdoc in the School of Information Systems, Singapore Management University. Currently, he is a professor and dean with the School of Information Technology, Jinan University. He has published more than 60 papers in cryptography conferences and journals, such as CRYPTO, Eurocrypt, TCC, Asiacrypt, PKC, CT-RSA, IEEE TDSC and IEEE TIFS. He served as PC cochairs or PC member for more than 30 international conferences, such as ISPEC 2011, RFIDsec 2013 Asia, ISC 2011, IWSEC 2012, *etc.*