Practical and Secure Query Processing for Large-scale Encrypted Cloud Storage Systems

Fangquan Cheng[1], Qian Wang[1], Kui Ren[2], Zhiyong Peng[1]

Wuhan University[1], University at Buffalo, SUNY[2]
{cheng,qianwang,peng}@whu.edu.cn, kuiren@buffalo.edu

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 of interest while preventing database servers from learning private information about the contents of files and queries. In this paper, we investigate new and novel SSE designs which meet all practical properties, including one-round multi-keyword query, comprehensive and practical privacy protection, sublinear search time, and efficient dynamic data operation support. Moreover, our solutions can well support parallel search and run for very large-scale cloud databases. Compared to the existing SSE solutions, our solution is highly compact, efficient and flexible. Its performance and security are carefully characterized by rigorous analysis. Experimental evaluations conducted over large representative real-word data sets demonstrate that compared with the state-of-the-art our solution indeed achieves desirable properties for large-scale encrypted database systems.

Key words: Cloud storage, encrypted data, one-round multi-keyword query, data dynamics, parallel search

1 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 [1, 2]. 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 and financial data etc., a straightforward approach is to encrypt the sensitive data locally before outsourcing [2, 3, 3, 4]. While providing strong end-to-end privacy, encryption becomes a hindrance to data computation/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 both the data content and query purposes in plaintext not to be leaked to the remote database server.

To allow a client to retrieve encrypted data files of interest, many searchable symmetric encryption (SSE) solutions have been proposed [5–12]. The existing SSE solutions, however, either have large search time complexity and privacy guarantee under weaker security model [5,6], or cannot support efficient data operations and parallel computations [8,9], or introduce relatively large storage cost and low search efficiency [11,12]. Besides, to achieve efficient searches, researchers have weaken the privacy guarantee without the protection of *access pattern* and *search pattern*. In recent years, researchers have put great effort into the practicality of SSE constructions. Kamara et al. [10] designed a two-level dynamic encrypted index based on the inverted index approach. As a following work, Kamara et al. [13] further proposed a parallelizable and dynamic SSE scheme based on the red-black tree data structure, achieving stronger privacy guarantee during data updates. In the most recent work [14], Cash et al. proposed a new SSE construction, supporting sublinear multi-keyword Boolean queries. However, their schemes have to execute multi-round interactions for the security concern, with relatively high computational burden on the client. To the best of our knowledge, none of existing SSE solutions satisfy all desirable properties in terms of *one-round* multi-keyword query, comprehensive and practical privacy protection, sublinear search time and efficient dynamic data operation support. Thus, there still exists much room for us to explore more efficient, flexible and practically-secure index constructions for processing queries over encrypted cloud storage systems.

In this paper, we introduce a suite of new and novel SSE index designs for processing queries over encrypted cloud storage systems. Our construction addresses the main limitations of each of the existing inverted index approaches by carefully making a trade-off between query efficiency and query privacy, with flexible and comprehensive query functionalities. In particular, our solution has a very compact index construction achieves practical privacy guarantee and supports *one-round* multi-keyword query, scalability, parallel computation and efficient data updates. Our technical contributions are as follows.

– We propose a bucket-encrypting index structure (BEIS) with moderate computational cost, based on a new bloom filter-based probabilistic information coding approach. The BEIS naturally allows efficient *one-round* multi-keyword query without sacrificing query privacy.
– We propose an enhanced version of BEIS to further improve the query efficiency, based on a new hierarchical data structure while keeping the same privacy guarantee. Due to the nice and decomposable index structure, query processing over BEIS and its hierachical version (HBEIS) can both be parallelized in multi-core architectures and support efficient and secure data updates.
– We present a formal security analysis of our schemes, showing that the proposed schemes enjoy security against adaptive chosen-keyword attacks (CKA2) and can help mitigate the leakage of search pattern privacy. To demonstrate the practicality of our solutions, we conduct experimental evaluations using large representative real-world data sets. It is shown that our HBEIS construction can increase the query performance from approximately 6 to 10^2 times with a representative dataset of $5 \cdot 10^5$ data files.

2 Related Work, Preliminaries and Definitions

2.1 Related Work

Following the idea of trading security for efficiency, the first notable SSE scheme along this direction was proposed by Song et al. [5] with weaken security, and the search time of the scheme is linear in the length of the file collection. By associating an encrypted index to each file, the SSE construction proposed by Goh [6] achieves search time that is proportional to n, the number of files in the collection. In particular, a formal security model (IND-CKA) with respect to index indistinguishability was first introduced to prove the semantic security of indexes, and one appealing property of this solution is that it handles dynamic data efficiently and securely (without revealing any private information about the contents of data and keywords). In 2006, Curtmola et al. [8] generalized security definitions of SSE (CKA1 and CKA2) and proposed a new encrypted index structure that associates with the entire file collection. The resulting search time is optimal and sublinear in $|\mathbf{d}_w|$, the number of files that contain keyword w. Recently, Kamara et al. [10] extended the inverted index approach [8] and designed a two-level dynamic encrypted index which allows for both addition and removal of data files. This construction achieves the overall best performance in single-core architectures due to its high query efficiency and dynamic index. Motivated by advances in multi-core architectures, Kamara et al. [13] proposed a parallelizable and dynamic SSE scheme based on red-black tree data structure. The search executes in parallel $O(\frac{|\mathbf{d}_w|}{p}\log|\mathbf{d}|)$ time, where p and $|\mathbf{d}|$ are the number of the available processors and the total data files, respectively. Among others, one 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. Besides, to trade privacy for efficiency, the existing SSE designs leak the access pattern and the search pattern. The multi-keyword search problem over encrypted data was investigated by Cao et.al [12] and a multi-keyword ranked search scheme was proposed using the secure kNN computation technique [15]. 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. Still, the access pattern and the search pattern during query execution are still in risk without any protection. In the latest literature [14], the authors designed the first SSE scheme which supports sublinear multi-keyword Boolean queries by providing trade-offs between performance and privacy. Unfortunately, for each query, their schemes have to execute two/three rounds of interaction for security concerns, and put too much computational burden on the client (the time complexity of the trapdoor generation in terms of *token* is $O(q|\mathbf{d}_w|)$, where a *token* can be roughly expressed as g^x and q is the number of query keywords. To enrich the functionalities, the notion of fuzzy and/or similarity keyword search over encrypted data have been proposed and investigated in [11, 16]. Li et.al [11] put forth a fuzzy keyword search scheme by building a large "fuzzy set" associated with both the encrypted index and the search queries. Based on locality sensitive hashing, Kuzu et.al [16] proposed a more generic solution for distinct similarity contexts. A two-server model is considered to mitigate the risk of access pattern leakage, but similar to existing SSE solutions which leak the search pattern, it leaks the similarity pattern.

2.2 Problem Statement and 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. 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 $\mathbf{q} = \{w_1, \ldots, w_{\#\mathbf{q}}\}$ (note that for a relational database, keywords are attribute-value pairs) to query and retrieve data files of interest in the database. To this end, the client generates an encrypted query token $\boldsymbol{\tau}$ (*i.e.*, trapdoor of $\mathbf{q}$) 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 $\mathbf{c_q}$ that satisfy the query, *i.e.*, each of $\mathbf{c_q}$ contains all the query keywords in $\mathbf{q}$ for the 'AND' logic query, or $\mathbf{c_q}$ contains the ranked (possible top-k) encrypted data files for the 'OR' logic query based on a specific similarity measurement. Finally, the client can locally decrypt $\mathbf{c_q}$ and obtain data contents in plaintext form.

The database system is a dynamic one. That is, at any time the data owner or authorized users may add or remove one or more data files $\mathbf{d}_u$ from the remote database. To do so, the client generates an encrypted update token $\boldsymbol{\tau}_u$. Given $\boldsymbol{\tau}_u$, the database server can execute *secure* updates on $\mathbf{c}$ and the corresponding index $\boldsymbol{\gamma}$. In multi-core architectures, the database server is expected to execute parallel computations or say parallel keyword search in our application scenario. Formally, a searchable encrypted database system is defined as below.

Definition 1. *(A searchable encrypted database) A searchable encrypted database consists of six polynomial-time algorithms (KeyGen, BuildIndex, Trapdoor, Query, UpdateInfo, Update) such that:*

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

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

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

$\mathbf{c_q} \leftarrow \mathsf{Query}(\boldsymbol{\tau}, \boldsymbol{\gamma}, \mathbf{c})$: *is a deterministic algorithm that 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 $\mathbf{c_q}$.*

$\boldsymbol{\tau}_u \leftarrow \mathsf{UpdateInfo}(sk, \mathbf{d}_u)$: *is a (possibly probabilistic) algorithm that takes as input the secret key sk and a set of data files $\mathbf{d}_u$* to be added or deleted, and outputs an update token $\boldsymbol{\tau}_u$.

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

We use $\mathtt{id}(d_j)$ to denote the identifier of the data file d_j and $\mathtt{id}(\mathbf{d})$ to denote the identifiers of all data files in $\mathbf{d}$. The keyword universe (which contains all distinct keywords) extracted from $\mathbf{d}$ is denoted by $\mathbf{w} = (w_1, \ldots, w_{\#\mathbf{w}})$. 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.). We also use $\mathbf{d}_w$ to denote the all data files that contain w and $\phi(\mathtt{div})$ to denote the number of 1's in $\mathtt{div}$. In addition, we use $\widehat{\mathsf{div}}$ to denote a joint data identifier vector computed from multiple divs based on pre-defined computations. Standard bitwise Boolean operations are defined on binary vectors: bitwise OR (union) "$\bigvee$" and bitwise AND (intersaction) "$\bigwedge$", which also take the same notations for single bit OR and AND operations, respectively. Given a vector v, we refer its j^{th} element as $v[j]$ or v_j.

2.3 Security Model and Definitions

We follow the widely-accepted security definitions of SSE in [6, 8–10] for our construction. However, in our application scenario $\mathbf{q}$ is a multi-keyword query instead of a single-keyword one.

Definition 2. *(Search Pattern π). Given a sequence of s query requests $\boldsymbol{Q} = \{\boldsymbol{q}_1, \ldots, \boldsymbol{q}_s\}$, the search pattern privacy is defined as a symmetric binary $s \times s$ matrix π, where the element $\pi[i,j] = 1$ if $\boldsymbol{q}_i = \boldsymbol{q}_j$ and $\pi[i,j] = 0$ otherwise.*

Definition 3. *(Access Pattern A_p). Given a sequence of s query requests $\boldsymbol{Q} = \{\boldsymbol{q}_1, \ldots, \boldsymbol{q}_s\}$. Let $\boldsymbol{\tau} = \{\boldsymbol{\tau}_1, \ldots, \boldsymbol{\tau}_s\}$ be the corresponding trapdoors of $\boldsymbol{Q}$ and $\{\boldsymbol{c}_{\boldsymbol{q}_1}, \ldots, \boldsymbol{c}_{\boldsymbol{q}_s}\}$ be the sequence of query results. The access pattern privacy then is defined as $\{A_p(\boldsymbol{\tau}_1) = \boldsymbol{id}(\boldsymbol{c}_{\boldsymbol{q}_1}), \ldots, A_p(\boldsymbol{c}_{\boldsymbol{q}_s}) = \boldsymbol{id}(\boldsymbol{c}_{\boldsymbol{q}_s})\}$.*

Definition 4. *(History). A history $H = (\boldsymbol{d}, \boldsymbol{Q})$, which contains the interaction history between a client and the cloud database server, consists of a collection of data files $\boldsymbol{d}$ and s query requests $\boldsymbol{Q} = \{\boldsymbol{q}_1, \ldots, \boldsymbol{q}_s\}$.*

Definition 5. *(View). A view of adversaries contains all information that adversaries can access directly. Let $\boldsymbol{c} = \{c_1, \ldots, c_{\#\boldsymbol{c}}\}$ be a collection of encrypted data files, $\boldsymbol{id}(c_j)$ be the identifier of the encrypted data file c_j, $\boldsymbol{\gamma}$ be the encrypted index of $\boldsymbol{c}$, $\boldsymbol{\tau_Q} = \{\boldsymbol{\tau}_{\boldsymbol{q}_1}, \ldots, \boldsymbol{\tau}_{\boldsymbol{q}_s}\}$ be the trapdoors for a history $H = (\boldsymbol{d}, \boldsymbol{Q})$. Then, $View(H) = \{\boldsymbol{id}(c_1), \ldots, \boldsymbol{id}(c_{\#\boldsymbol{c}}), \boldsymbol{c}, \boldsymbol{\gamma}, \boldsymbol{\tau_Q}\}$ is defined as the view of H.*

Definition 6. *(Trace). A trace presents information users allow adversaries to access. Given a history $H = (\boldsymbol{d}, \boldsymbol{Q})$, let $|c_j|$ be the size of the encrypted form c_j of the data file d_j ($d_j \in \boldsymbol{d}$), $\boldsymbol{id}(c_j)$ be the identifier of the encrypted data file c_j, $\pi(H)$ and $A_p(H)$ be the search and access patterns for H. Then, $Trace(H) = \{|c_1|, \ldots, |c_{\#\boldsymbol{c}}|, \boldsymbol{id}(c_1), \ldots, \boldsymbol{id}(c_{\#\boldsymbol{c}}), \pi(H), A_p(H)\}$ is defined as the trace of H.*

Definition 7. *(Adaptive Security (CKA2) for Searchable Encrypted Database). A searchable encrypted database is said to be adaptively secure, if for all $s \in \mathbb{N}$ and for all (non-uniform) probabilistic polynomial-time adversaries $\mathcal{A} =$*

$\{\mathcal{A}_0, \ldots, \mathcal{A}_{s-1}\}$*, then there exists a (non-uniform) probabilistic polynomial-time simulators* $\mathcal{S} = \{\mathcal{S}_0, \ldots, \mathcal{S}_{s-1}\}$ *such that* $\mathcal{S}$ *can simulate the adversary's view of a s-query history* H *from* $Trace(H)$ *with probability negligibly close to 1. Formally, for all polynomials and a sufficiently large* κ*, let* $s = poly(\kappa)$ *and* H *be a s-query history, there exists a simulator* $\mathcal{S}$ *such that for all polynomial-size distinguishers* $\mathcal{D}$*:*

$$|Pr[\mathcal{D}(\mathcal{A}(View(H))) = 1] - Pr[\mathcal{D}(\mathcal{S}(Trace(H))) = 1]| \leq \mathit{negl}(k),$$

where the probabilities are taken over H *and the internal coins of the key generation and the encryption.*

3 Our Constructions

3.1 Probabilistic Coding Data Structure

Base on Definition 1, we present the basic structure of our index construction.
$\mathsf{KeyGen}(1^k)$: Given a security parameter k, generate two random k-bit strings sk_1 and r. Call $sk_2 \leftarrow \mathsf{SKE.KeyGen}(1^k, r)$, where SKE is a PCPA-secure symmetric encryption scheme. Output $sk = (sk_1, sk_2)$.
$\mathsf{BuildIndex}(sk, \mathbf{d})$: Extract $\mathbf{w} = \{w_1, \ldots, w_{\#\mathbf{w}}\}$ from $\mathbf{d}$ and generate $\{\mathsf{div}_1, \ldots, \mathsf{div}_{\#\mathbf{w}}\}$. Construct a data structure $\boldsymbol{\gamma}$ consisting of m buckets. Each bucket of $\boldsymbol{\gamma}$ is initially set to be empty. Select a collection of K independent keyed hash functions $h_{i=[1,K]} = \{h_i | i \in [1, K]\}$ (or pseudo-random functions), where $h_i : \{0,1\}^* \times \{0,1\}^k \to [m]$. For each w_i ($i \in [1, \#\mathbf{w}]$), compute ($x_1 = h_1(w_i, sk_1), \ldots, x_K = h_K(w_i, sk_1)$). For each x_j ($j \in [1, K]$), insert div_i into bucket $\boldsymbol{\gamma}[x_j]$ according to the following rule: If the bucket $\boldsymbol{\gamma}[x_j]$ is empty, store div_i in the bucket, denoted by $\boldsymbol{\gamma}[x_j] = \mathsf{div}_i$; otherwise update the bucket by storing the "union" of div_i and the "old" DIV previously stored in bucket $\boldsymbol{\gamma}[x_j]$, denoted by $\boldsymbol{\gamma}[x_j] = \boldsymbol{\gamma}[x_j] \bigvee \mathsf{div}_i$. Let $\mathbf{c} \leftarrow \mathsf{Enc}(sk_2, \mathbf{d})$. Output $(\boldsymbol{\gamma}, \mathbf{c})$.
$\mathsf{Trapdoor}(sk, \mathbf{q})$: For each w_i ($i \in [1, \#\mathbf{q}]$), compute ($x_1 = h_1(w_i, sk_1), \ldots, x_K = h_K(w_i, sk_1)$). Generate the union of all positions as $\boldsymbol{\tau}$. Output $\boldsymbol{\tau}$ and the query logic information ('AND' or 'OR').
$\mathsf{Query}(\boldsymbol{\gamma}, \mathbf{c}, \boldsymbol{\tau})$: Execute $\boldsymbol{\tau}$ over $\boldsymbol{\gamma}$ according to the client-specified query logic:
1) *Conjunction logic query.* The server computes $\mathsf{div}_{\boldsymbol{\tau}}$ from the buckets at the all positions in $\boldsymbol{\tau}$ as $\mathsf{div}_{\boldsymbol{\tau}} = \bigwedge_{y \in \boldsymbol{\tau}} \boldsymbol{\gamma}[y]$. Recover data identifiers from $\mathsf{div}_{\boldsymbol{\tau}}$ and output the corresponding encrypted data files. To state the correctness, let $\boldsymbol{\tau}_w$ be the sub-trapdoor for a single keyword w in $\mathbf{q}$. Thus, $\boldsymbol{\tau}_w \subset \boldsymbol{\tau}$ and

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

2) *Disjunction logic query.* The server computes the similarity score, denoted by $score(d_j, \mathbf{q})$, between data file d_j and query $\mathbf{q}$. We define the score against the trapdoor $\boldsymbol{\tau}$ as

$$score(d_j, \mathbf{q}) = \sum_{y \in \boldsymbol{\tau}} \boldsymbol{\gamma}[y][j]. \tag{2}$$

Output the ranked encrypted data files with $score(d_j, \mathbf{q}) \geq K$, where K is the number of the keyed hash functions.

Remarks. For multi-keyword conjunction queries, as can be seen from Eq. (1), the query result of the index structure is equivalent to that of the post-processing of results of single-keyword queries. For multi-keyword disjunction queries, a keyword contained in both the data d_j (during the index construction) and the query $\mathbf{q}$ will be mapped to the same set of buckets in the index. Therefore, $score(d_j, \mathbf{q})$ is simply the number of common buckets mapped by the keywords in d_j and $\mathbf{q}$. Intuitively, the larger number of common buckets, the higher similarity between the data d_j and the query $\mathbf{q}$. If a data file d_j contains at least one query keyword, there must be $score(d_j, \mathbf{q}) \geq K$. Due to this property, the server can send back all ranked encrypted data items with $score(d_j, \mathbf{q}) \geq K$ or the ranked encrypted data items with the top-k highest scores, where k could be user-specified.

3.2 BEIS: Bucket-Encrypting Index Structure

On the basis of probabilistic coding data structure, we present a bucket-encrypting index structure (BEIS), enabling efficient query processing over encrypted data and providing semantic security against adaptive adversaries (CKA2). BEIS inherits the bucket-based index structure (the probabilistic coding data structure) to gain efficient query performance and encrypts all information in the index structure. Our analysis and experimental evaluation show that compared to the state-of-the-art of SSE, BEIS enjoys better performance at both the client and cloud sides and comprehensive and practical privacy guarantee. Moreover, BEIS supports *one-round* search with different query logics instead of multiple rounds of interactions for each query.

BEIS with Random Generator. The centerpiece of BEIS is to design a new non-trivial and non-deterministic encryption function which encrypts the bits of the data identifier vector stored in the indexing buckets and a search algorithm which outputs data identifiers of the target encrypted data files without exposing the bit values of vectors that have been searched.

$\mathsf{KeyGen}(1^k)$: Given a security parameter k, choose two prime numbers α, β and two k-bit strings sk_1, r uniformly at random. Generate $sk_2 \leftarrow \mathsf{SKE.KeyGen}(1^k, r)$, where SKE is a PCPA-secure symmetric encryption scheme. Output $sk = (\alpha, \beta, sk_1, sk_2)$.

Let $F : \{0,1\}^k \times \{0,1\}^* \rightarrow \{0,1\}^k$ be a pseudo-random function, which is a polynomial-time computable function that cannot be distinguished from a random function by any polynomial-time adversaries.

$\mathsf{BuildIndex}(sk, \mathbf{d})$: After constructing the bucket based index structure $\boldsymbol{\gamma}$ from data files $\mathbf{d}$, we encrypt $\boldsymbol{\gamma}$ as follows: For each $\boldsymbol{\gamma}[y][j]$ $(y \in [1, m], j \in [1, \#\mathbf{d}])$, *i.e.*, the j^{th} bit of the vector stored in the y^{th} bucket of $\boldsymbol{\gamma}$:

- Generate the two associated secret parameters denoted by $\zeta_{yj} = F(sk_1, y||j||1)$ and $\lambda_{yj} = F(sk_1, y||j||2)$. The notation $||$ denotes string concatenation. Let $b = \boldsymbol{\gamma}[y][j]$ and store the encrypted form of b into $\boldsymbol{\gamma}[y][j]$ as

$$\boldsymbol{\gamma}[y][j] = \alpha(b\beta + \zeta_{yj}) + \lambda_{yj}, \qquad (3)$$

  where ζ_{yj} and λ_{yj} can be considered as privacy parameters to prevent b, α and β from the greatest common divisor (GCD) based attack. This encryption algorithm has been shown to be provably secure on the basis of approximate GCD problem [17]. Please refer [17] for details.

The client outsources the encrypted data files $\mathbf{c}$ encrypted under sk_2 and $\boldsymbol{\gamma}$ to the cloud server.

$\mathsf{Trapdoor}(sk, \mathbf{q})$: The trapdoor consists of two components. One is the bucket positions $P_{\mathbf{q}}$ mapped by all query keywords, which is denoted by $P_{\mathbf{q}} = \{h_j(w_i, sk_1) | j \in [1, K] \text{ and } i \in [1, \#\mathbf{q}]\}$. With these positions and sk_1, the client recoveries associated secret parameters ζ_{yj} and λ_{yj} for each $y \in P_{\mathbf{q}}$ and $j \in [1, \#\mathbf{d}]$. Another component is called the search token, denoted by $st_{\mathbf{q}}$, which is generated under the positions $P_{\mathbf{q}}$ according to different query logics:
1) Conjunction logic. Compute the search token for the 'AND' logic

$$st_{\mathbf{q},j} = \sum_{y \in P_{\mathbf{q}}} [\alpha(\beta + \zeta_{yj}) + \lambda_{yj}] \;\text{ for } j \in [1, \#\mathbf{d}]. \tag{4}$$

2) Disjunction logic. Generate randomly a privacy parameter q_j from $[1, \alpha\beta]$ for $j \in [1, \#\mathbf{d}]$ and compute the search token for the 'OR' logic

$$st_{\mathbf{q},j} = \sum_{y \in P_{\mathbf{q}}} [\alpha\zeta_{yj} + \lambda_{yj} - q_j] \;\text{ for } j \in [1, \#\mathbf{d}], \tag{5}$$

where q_j is used to guarantee the security of our algorithm, *i.e.*, preventing α and β from being exposed in query processing. Note that q_j is constrained within $[1, \alpha\beta)$ so that q_j do not affect the ranking of results.

Then the client sends the trapdoor $\boldsymbol{\tau} = \{P_{\mathbf{q}}, st_{\mathbf{q}} = \{st_{\mathbf{q},1}, \ldots, st_{\mathbf{q},\#\mathbf{d}}\}\}$ and the query logic information ('AND' or 'OR') to the cloud server.

$\mathsf{Query}(\boldsymbol{\gamma}, \mathbf{c}, \boldsymbol{\tau})$: Receiving the trapdoor $\boldsymbol{\tau} = \{P_{\mathbf{q}}, st_{\mathbf{q}}\}$, the cloud server excutes the query according to the user-specified query logic. Let IDs be the sequence of identifiers of the returned data files in the following discussion.
1) Conjunction logic. For each $j \in [1, \#\mathbf{d}]$, add $\mathtt{id}(d_j)$ into IDs if

$$\sum_{y \in P_{\mathbf{q}}} \boldsymbol{\gamma}[y][j] = st_{\mathbf{q},j}. \tag{6}$$

2) Disjunction logic. For each $j \in [1, \#\mathbf{d}]$, add $\mathtt{id}(d_j)$ into IDs and compute the similarity score $score(d_j, \mathbf{q})$

$$score(d_j, \mathbf{q}) = \sum_{y \in P_{\mathbf{q}}} \boldsymbol{\gamma}[y][j] - st_{\mathbf{q},j}. \tag{7}$$

Rank the identifiers in IDs in accordance with the similarity scores. Finally, the cloud server returns the (or ranked) encrypted data files.

The following theorem shows the effectiveness of BEIS, *i.e.*, the returned data files and their similarity scores in BEIS are the same as those of the plaintext search over the basic index structure (without encrypting the index).

Theorem 1. *Searching over BEIS with random generator is equivalent to searching over the basic index structure.*

Proof. See Appendix A.

BEIS with Homomorphic Generator. In the above BEIS construction scheme with the random generator, the client has to pay the cost for generating and transmitting search tokens in $\mathsf{Trapdoor}$. To reduce the communication

$\mathsf{KeyGen}(1^k)$: choose $sk_1, r \xleftarrow{\$} \{0,1\}^k$ and two prime numbers α, β. Generate $sk_2 \leftarrow \mathsf{SKE.KeyGen}(1^k, r)$. Output $sk = (\alpha, \beta, sk_1, sk_2)$.

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

1. for each $y \in [1, m]$, generate secret roots $\zeta_{y0} = F(sk_1, y||1)$, $\lambda_{y0} = F(sk_1, y||2)$ and $\mu_{y0} = F(sk_1, y||3)$.

2. for $\forall y \in [1, m]$, $\forall j \in [1, \#\mathbf{d}]$, generate two secret parameters:

$$\lambda_{yj} = \lambda_{y0} + \frac{1}{2} f_h(h(j)\mu_{y0}), \tag{8}$$

$$\zeta_{yj} = \zeta_{y0} + \frac{1}{2\alpha} f_h(h(j)\mu_{y0}). \tag{9}$$

3. encrypt the plaintext index $\boldsymbol{\gamma}$ according to Eq. (3) and let $\mathbf{c} \leftarrow \mathsf{Enc}(sk_2, \mathbf{d})$.

4. output $(\boldsymbol{\gamma}, \mathbf{c})$.

$\mathsf{Trapdoor}(sk, \mathbf{q})$:

1. generate bucket positions $P_{\mathbf{q}} = \{h_j(w_i, sk_1) | j \in [1, K] \text{ and } i \in [1, \#\mathbf{q}]\}$.

2. for each $y \in P_{\mathbf{q}}$, recover ζ_{y0}, λ_{y0} and μ_{y0} .

3. compute the root token st_r:

$$st_r = \sum_{y \in P_{\mathbf{q}}} [\alpha(\beta + \zeta_{y0}) + \lambda_{y0}]. \tag{10}$$

4. Output $\boldsymbol{\tau} = \{P_{\mathbf{q}}, st_r, f_h(\sum_{y \in P_{\mathbf{q}}} \mu_{y0})\}$ and the query logic ('AND' or 'OR').

$\mathsf{Query}(\boldsymbol{\gamma}, \mathbf{c}, \boldsymbol{\tau})$:

1. $\forall j \in [1, \#\mathbf{d}]$, generate the search token $st_{\mathbf{q}}$:

$$st_{\mathbf{q},j} = st_r + f_h(h(j)) f_h(\sum_{y \in P_{\mathbf{q}}} \mu_{y0}). \tag{11}$$

2. execute the query:

   1) conjunction logic. For each $j \in [1, \#\mathbf{d}]$, add $\mathtt{id}(d_j)$ into IDs if

$$\sum_{y \in P_{\mathbf{q}}} \boldsymbol{\gamma}[y][j] \bmod p = st_{\mathbf{q},j} \bmod p. \tag{12}$$

   2) disjunction logic. For each $j \in [1, \#\mathbf{d}]$, compute the similarity score $score(d_j, \mathbf{q})$

$$score(d_j, \mathbf{q}) = (st_{\mathbf{q},j} - \sum_{y \in P_{\mathbf{q}}} \boldsymbol{\gamma}[y][j]) \bmod p \tag{13}$$

and add $\mathtt{id}(d_j)$ into IDs if $score(d_j, \mathbf{q}) \geq K\alpha\beta \bmod p$. Rank identifiers in IDs.

3. Output $\mathbf{c}_{\mathbf{q}}$.

Fig. 1. The query processing over BEIS with homomorphic generator.

overhead, we next present another BEIS construction with the homomorphic generator used for generating the secret parameters in BuildIndex and recovering them in Trapdoor. With the homomorphic generator, a tiny cost will be put on the client to generate a root token st_r of the search token $st_{\mathbf{q}}$. After receiving st_r, the cloud server can generate the complete search token $st_{\mathbf{q}}$ by using pre-defined functions without reducing the level of security. By doing so, our scheme achieves a more practical value by migrating the computational task of the search token generation from clients to the cloud, eliminating the large communication cost. We show the details of the homomorphic generator in the following processing steps: the generation of the secret parameters ζ_{yj}, λ_{yj} for each required bucket position y and each bit position j of DIV, the generation of the root token st_r and the complete search token $st_{\mathbf{q}}$. The other processing steps are the same as BEIS construction with random generator and will not be discussed here due to the space limitation. Let $h : \{0,1\}^* \rightarrow \{0,1\}^k$ be a random oracle. We start the discussion from an additive and multiplicative homomorphic generator f_h: Let s be a sensitive value to be protected, p and q be two distinct large prime numbers,

$$f_h(s) = (s + rp) \bmod pq, \tag{14}$$

where r is a randomly-chosen integer and $f_h^{-1}(f_h(s)) = f_h(s) \bmod p$. Assuming that s_1, s_2 are two randomly-selected private values, it is easy to show f_h's homomorphic properties: $f(s_1)+f(s_2) \Leftrightarrow f(s_1+s_2)$ and $f(s_1)f(s_2) \Leftrightarrow f(s_1 s_2)$, where the notation $\Leftrightarrow$ represents the equivalency relation. In the following discussion, we will utilize the homomorphic properties of f_h for our index construction and query processing. In Query, we use IDs to record identifiers of the returned data files, and the value $K\alpha\beta \bmod p$ can be provided by the client when generating search tokens. The query processing over BEIS construction with homomorphic generator is described in Fig. 1.

Theorem 2. *Searching over BEIS with homomorphic generator is equivalent to searching over BEIS construction with random generator.*

Proof. See Appendix B.

Theorem 3. *The BEIS construction with homomorphic generator is adaptively secure under the CKA2 model.*

Proof. See Appendix C.

Discussions of search pattern privacy mitigation. Due to the probabilistic nature of our index construction, it can help mitigate the search pattern privacy leakage. Let τ_1, τ_2 be the trapdoors of two queries $\mathbf{q}_1$ and $\mathbf{q}_2$. According to Definition 2, the mitigation of the search pattern leakage should make $\mathbf{q}_1$ indistinguishable from $\mathbf{q}_2$ even though $\tau_1 = \tau_2$. We next analyze the probability p_{sp} of the case $\tau_1 = \tau_2$ but $\mathbf{q}_1 \neq \mathbf{q}_2$. We use $\hat{\mathbf{q}}$ to denote the supper set of all queries. For ease of analysis, we consider the following equivalent case: Given $\mathbf{q}_1 \subset \hat{\mathbf{q}}$ and τ_1, what is the probability that there exists $\mathbf{q}_2 \subset \hat{\mathbf{q}}$ ($\mathbf{q}_2 \neq \mathbf{q}_1$) such that $\tau_1 = \tau_2$. $\forall w \in \hat{\mathbf{q}} - \mathbf{q}_1$ and its trapdoor τ_w, there must exist one position $y \in \tau_1$ with two possible cases: 1) $\tau_w \subset \tau_1 \setminus \{y\}$; 2) $y \in \tau_w$ and there exists at least one position in τ_w that locates in $\{1, \ldots, m\} \setminus \{\tau_1\}$. Therefore, we can compute $p_{sp} = 1 - [(\frac{\#\tau_1 - 1}{m})^K + \frac{K}{m}(1 - (\frac{\#\tau_1}{m})^{K-1})]^{\#\mathbf{q} - \#\mathbf{q}_1}$. Undoubtedly, $K, \#\tau_1, \#\mathbf{q}_1 \ll m, \#\hat{\mathbf{q}}$ and $\hat{\mathbf{q}} \supseteq \mathbf{w}$. Then, we have $p_{sp} \geq \frac{1}{2}$ since $\#\hat{\mathbf{q}}$ is relatively large. Therefore, from a practical point of view, for all probabilistic polynomial-time adversaries there exists no probabilistic polynomial-time simulator that can distinguish τ_1 from τ_2.

4 HBEIS: Hierarchical BEIS Construction

In BEIS, the index construction and query processing do not involve any expensive encryption operations as compared to most of existing SSE solutions, so the query efficiency can be very high. However, in the execution of queries, the cloud server needs to check each data file to generate the identifiers of returned data files. To accommodate large-scale storage systems, in this section, we extend our index construction to a hierarchical BEIS (HBEIS) to address this problem and gain much higher query efficiency on the cloud side, at the price of a slightly higher cost for index construction, storage and update.

Given a data set $\mathbf{d}$, a bucket-based index $\boldsymbol{\gamma}$ in plaintext form will be constructed. Then, we construct the hierarchical bucket-based structure (HBS), which is a bucket-based binary tree. At each leaf of the HBS, a data file identifier is stored. At the root or each internal node u of the HBS, a m-bucket vector $\boldsymbol{\gamma}_u$ is stored. All buckets of $\boldsymbol{\gamma}_u$ will be set a boolean value in the HBS initialization. Specifically, if $\boldsymbol{\gamma}_u[y] = 1$ (the y-th bucket of $\boldsymbol{\gamma}_u$), then there is at least one path from u to a leaf node that stores an identifier $\mathsf{id}(d_j)$, such that $\boldsymbol{\gamma}[y][\mathsf{id}(d_j)] = 1$. That means that the data identifier vector $\boldsymbol{\gamma}[y]$ stored in the y-th bucket of $\boldsymbol{\gamma}$ includes the data file d_j. Intuitively, in order to guarantee the above property of the bucket vector $\boldsymbol{\gamma}_u$, we generate $\boldsymbol{\gamma}_u$ as follows: Let l and r be the left child storing the bucket vector $\boldsymbol{\gamma}_l$ and the right child of r storing the bucket vector $\boldsymbol{\gamma}_r$, respectively. The bucket vector $\boldsymbol{\gamma}_u$ is computed recursively as follows: $\boldsymbol{\gamma}_u = \boldsymbol{\gamma}_l \bigvee \boldsymbol{\gamma}_r$. Note that each leaf v storing the data identifier $\mathsf{id}(d_j)$ is associated with a virtual bucket vector $\boldsymbol{\gamma}_v$, which will not be physically stored in the disk. Then, we let $\boldsymbol{\gamma}_v[y] = \boldsymbol{\gamma}[y][\mathsf{id}(d_j)]$ for each $y \in [1, m]$.

After building the plaintex index $\boldsymbol{\gamma}$ and the HBS, we discuss the encryption of the buckets and the query processing following the logic in section 3.2. For the root or every internal node u of HBS, it has an unique identifier $\mathsf{id}(u)$.

$\mathsf{KeyGen}(1^k)$: Given a security parameter k, choose two additional prime numbers $\hat{\alpha} \xleftarrow{\$} \{0,1\}^k$ and $\hat{\beta} \xleftarrow{\$} \{0,1\}^k$ uniformly at random in addtion to $(\alpha, \beta, sk_1, sk_2)$. Output $sk = (\alpha, \beta, \hat{\alpha}, \hat{\beta}, sk_1, sk_2)$.

$\mathsf{BuildIndex}(sk, \mathbf{d})$: The generation of $\boldsymbol{\gamma}$ is shown in section 3.2 and will not be repeated here. For the encryption of HBS, the client generates additional secret roots $\hat{\zeta}_{y0} = F(sk_1, y||4)$, $\hat{\lambda}_{y0} = F(sk_1, y||5)$, $\hat{\mu}_{y0} = F(sk_1, y||6)$ for each $y \in [1, m]$. Then, for the root and each internal node u of the HBS, the client generates two secret parameters for each bucket position $y \in [1, m]$ as follows:

$$\hat{\lambda}_{y\mathsf{id}(u)} = \hat{\lambda}_{y0} + \frac{1}{2} f_h(h(\mathsf{id}(u))\hat{\mu}_{y0}) \tag{15}$$

$$\hat{\zeta}_{y\mathsf{id}(u)} = \hat{\zeta}_{y0} + \frac{1}{2\hat{\alpha}} f_h(h(\mathsf{id}(u))\hat{\mu}_{y0}), \tag{16}$$

where we let $h(\mathsf{id}(u)) = 1$ if u is the root of the HBS. Let $b = \boldsymbol{\gamma}_u[y]$ and then store the encrypted b into $\boldsymbol{\gamma}_u[y]$ as follows:

$$\boldsymbol{\gamma}_u[y] = \hat{\alpha}(b\hat{\beta} + \hat{\zeta}_{y\mathsf{id}(u)}) + \hat{\lambda}_{y\mathsf{id}(u)}. \tag{17}$$

Finally, the client outsouces the index (including the encrypted $\boldsymbol{\gamma}$ and HBS) to the cloud together with the encrypted data files.

$\mathsf{Trapdoor}(sk, \mathbf{q})$: Given a query $\mathbf{q}$, the client generates the bucket positions $P_{\mathbf{q}}$

mapped by all query keywords and the two root tokens denoted by st_r (used for the search over $\boldsymbol{\gamma}$) and $\hat{st}_r$ (used for the search over the encrypted HBS). We show the generation of $\hat{st}_r$ following the generation of st_r:

$$\hat{st}_r = \sum_{y \in P_{\mathbf{q}}} [\hat{\alpha}(\hat{\beta} + \hat{\zeta}_{y0}) + \hat{\lambda}_{y0}]. \tag{18}$$

Then the client sends the trapdoor $\boldsymbol{\tau} = \{P_{\mathbf{q}}, st_r, \hat{st}_r, f_h(\sum_{y \in P_{\mathbf{q}}} \mu_{y0}), f_h(\sum_{y \in P_{\mathbf{q}}} \hat{\mu}_{y0})\}$ and the query logic information ('AND' or 'OR') to the cloud.

$\mathsf{Query}(\boldsymbol{\gamma}, \mathbf{c}, \boldsymbol{\tau})$: After receiving the trapdoor $\boldsymbol{\tau} = \{P_{\mathbf{q}}, st_r, \hat{st}_r, f_h(\sum_{y \in P_{\mathbf{q}}} \mu_{y0}),$ $f_h(\sum_{y \in P_{\mathbf{q}}} \hat{\mu}_{y0})\}$, the cloud server first performs the query over the encrypted HBS as follows: starting from the root of the HBS, the cloud server checks the buckets at positions $P_{\mathbf{q}}$ of node u and continues to check u's children if u meets the query. Return all data identifiers stored in the leaves that have been reached after this traversal is over. More specifically, let l and r be the left and the right child of a node u respectively, and let $\hat{IDs}$ be the sequence of the output data identifiers, we recursively define a checking algorithm $\mathsf{check}(u)$ as follows:

- Compute the search token $\hat{st}_{P_{\mathbf{q}},u} = \hat{st}_r + f_h(h(\mathsf{id}(u)))f_h(\sum_{y \in P_{\mathbf{q}}} \hat{\mu}_{y0})$.
- As for the conjunction query, if $\sum_{y \in P_{\mathbf{q}}} \gamma_u[y] \bmod p \neq \hat{st}_{\mathbf{q},u} \bmod p$, return; As for the disjunction query, if $score(u, \mathbf{q}) = (\hat{st}_{\mathbf{q},j} - \sum_{y \in P_{\mathbf{q}}} \gamma_u[y]) \bmod p < K\hat{\alpha}\hat{\beta} \bmod p$, return. Note that $K\hat{\alpha}\hat{\beta} \bmod p$ is provided by the query client.
- If u is a leaf, add the associated data identifier to $\hat{IDs}$, else call $\mathsf{check}(l)$ and $\mathsf{check}(r)$.

Then for each data identifier $\mathsf{id}(d_j) \in \hat{IDs}$, the cloud server computes the search token $st_{P_{\mathbf{q}},j}$ with the root token st_r and performs the query over $\boldsymbol{\gamma}[y][j](y \in P_{\mathbf{q}})$ according to the user-specified query logic. The detailed query processing is shown in section 3.2 and will not be repeated here.

5 The Support of Data Dynamics and Parallel Search

5.1 The Support of Dynamic Data Operations

In practice, the support of efficient data dynamics is highly required by an (encrypted) indexing scheme, *i.e.*, data adding and data deletion operations are supported without needing to either re-index the entire data collection or make use of generic and expensive dynamization techniques. To the best of our knowledge, the existing SSE solutions that can support data dynamics were proposed in [9] and [10]. The construction in [10] relies on an additional deletion index that has the same size with the search index. Besides, the update involves complex operations (*i.e.*, encryption and decryption operations) which may affect the update efficiency with a large index size. As for the privacy, the update operations may reveal a non-trivial amount of information, *e.g.*, the single data file adding operation in both [9] and [10] will leak the number of keywords contained in the data file.

Our constructions can address the above problems and easily fulfill more efficient data adding and deletion operations. We start from the update of $\boldsymbol{\gamma}$. To delete existing data files $\mathbf{d}_u$, the client can submit the deletion token $\boldsymbol{\tau}_u$

consisting of the identifiers of $\mathbf{d}_u$ to the cloud server. Then the server checks all non-empty buckets of the index $\boldsymbol{\gamma}$ and resets $\boldsymbol{\gamma}[y][\mathtt{id}(d_j)] = 0$ for each $d_j \in \mathbf{d}_u$ and each bucket of $\boldsymbol{\gamma}$. The time complexity of updating $\boldsymbol{\gamma}$ is $\mathcal{O}(\hat{m})$, where $\hat{m}$ represents the number of non-empty buckets of $\boldsymbol{\gamma}$. Obviously, it will be very time-consuming when $\hat{m}$ is relatively large. A better solution is that the client constructs an auxiliary index structure which is called as *Deletion Table* while not sacrificing any data privacy guarantee. The *deletion table* is constructed to quickly look up the buckets encoding the data files to be deleted. Firstly, we build the plaintext deletion table as a bidimensional table $\mathbf{T} = \{[\mathtt{id}(d_j), T_{\mathtt{id}(d_j)}] | d_j \in \mathbf{d}\}$, where $T_{\mathtt{id}(d_j)} = \{y | \boldsymbol{\gamma}[y][\mathtt{id}(d_j)] = 1\}$. Then $\mathbf{T}$ is encrypted as follows: $\mathbf{T} = \{[F(sk_1, \mathtt{id}(d_j)), \mathsf{SKE.Enc}(k_j, T_{\mathtt{id}(d_j)})] | d_j \in \mathbf{d}\}$, where $k_j = \mathsf{SKE.Enc}(sk_1, d_j)$. Then, $\mathbf{T}$ will be outsourced to the cloud together with the encrypted index. Obviously, the deletion table does not leak the number of keywords contained in the updated index since each data file is always associated with the same number of keywords. The cloud will find and decrypt the corresponding items of $\mathbf{T}$ with $\{F(sk_1, \mathtt{id}(d_j)), \mathsf{SKE.Enc}(sk_1, d_j) | d_j \in \mathbf{d}_u\}$ submitted by the client. Finally, the cloud updates the corresponding buckets of the index $\boldsymbol{\gamma}$ and deletes the corresponding items of $\mathbf{T}$.

To add new data files $\mathbf{d}_u$, the client locally encrypts the data files as $\mathbf{c}_u$ and builds the corresponding sub-index $\boldsymbol{\tau}_u$ for $\mathbf{d}_u$ which has the same features with the index $\boldsymbol{\gamma}$ on the cloud. $\boldsymbol{\tau}_u$ will be used to update the index $\boldsymbol{\gamma}$ or say be merged into $\boldsymbol{\gamma}$ stored on the remote database server. The construction of sub-index $\boldsymbol{\tau}_u$ is the same as the initial index construction shown in section 3.2. On the cloud side, the server performs the adding update by directly merging $\boldsymbol{\tau}_u$ and $\mathbf{T}_u$ into $\boldsymbol{\gamma}$ and $\mathbf{T}$, respectively. With respect to the update of the HBS, it is slightly more complicated than the update of $\boldsymbol{\gamma}$ since it may involve the update of the structural update of the HBS. To add or delete the data files $\mathbf{d}_u$ (with the corresponding identifiers), the update of the HBS will require two rounds of communication in order to guarantee the update security. We show the sketch of the HBS update as follows due to the space limitation.

- The client sends the identifiers of $\mathbf{d}_u$ and the type of update token (adding or deleting) to the cloud server.
- The cloud server performs the structural update of the HBS only based on the node identifiers of the HBS, the data identifiers of $\mathbf{d}_u$ and the type of data update request. As shown in section 4, the query time is proportional to the height of the HBS. Therefore, the HBS can be viewed as a red-black tree and the structural update of the HBS will involve the necessary rotations that are performed during an update of a red-black tree, so that the height of the HBS can always be maintained to be logarithmic. Please refer to [13] for more details of such update operations on a red-black tree. After updating the structure of the HBS, the cloud server returns the update auxiliary information AI to the client. The information AI contains portion of the HBS that consists of all nodes (along with the encrypted bucket vectors stored at them) accessed during the update. That is, AI suffices to perform the update of the HBS for supporting the dynamic operations of $\mathbf{d}_u$, in absence of the rest of the tree which is a complementary tree of AI in the HBS.
- After receiving AI which can be considered as a sub-HBS, if the update is an adding operation of $\mathbf{d}_u$, the client first encrypts the data files $\mathbf{d}_u$

as $\mathbf{c}_u$ and computes the virtual bucket vector $\boldsymbol{\gamma}_u$ for each $u \in AI$ which stores the identifier of the data file in $\mathbf{d}_u$. Then, the client decrypts the encrypted bucket vectors stored in AI and performs the bottom-up update of all plaintext bucket vectors storing the nodes of AI. For each node u of the updated AI, the client changes its identifier from $\mathsf{id}(u)$ to $\mathsf{id}(u')$. Finally, the client encrypts AI (*i.e.*, encrypts all bucket vectors stored in AI) and sends the encrypted AI to the cloud together with the encrypted data $\mathbf{c}_u$. As shown in the section 4, in the encryption of AI, the change of the node identifier will give rise to the re-computation and re-randomization of the secret parameters of all nodes in AI, thus preventing the cloud server from inferring information about the plaintext bucket vectors stored in AI by comparing the initial AI with the updated AI.
- After receiving $\mathbf{c}_u$ and the updated AI, the cloud copies AI to the already structurally-updated HBS (note that the HBS was structurally updated in step 2) and outputs the new HBS and the updated $\mathbf{c}$.

5.2 The Support of Parallel Search

To support parallel search, the cloud server constructs a compact addressing hash table which stores a pair: a data identifier and the address pointing to the encrypted data for each file in $\mathbf{c}$. Let 0,1,. . . ,$p-1$ be the processors that are available. Parallel search is run over $\boldsymbol{\gamma}$ as follows: each processor searches separately over the partitions of $\boldsymbol{\gamma}$, each of which encodes the same number of data files and is stored separately (*i.e.*, the partitions can be denoted by $\{\boldsymbol{\gamma}[y][j] | y \in [1,m], j \in [1, \frac{\#\mathbf{d}}{p}]\}, \ldots, \{\boldsymbol{\gamma}[y][j] | y \in [1,m], j \in [\frac{(p-1)\#\mathbf{d}}{p}, \#\mathbf{d}]\}$). The parallel search over HBS is executed in a similar manner as in [13]. Due to the space limitation, we eliminate the execution details here.

6 Theoretical Analysis and Experiments

6.1 Query Accuracy Analysis

In our index structure, there exists a tiny false positive probability due to the probabilistic multi-hash based encoding/decoding. Fortunately, the probability of false decodings can be reduced to almost a negligible level by properly choosing system parameters m, K.

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

$$\neg p_{\boldsymbol{q}} = \begin{cases} \frac{exp}{\phi(\bigwedge_{w_i \in \boldsymbol{q}} \mathsf{div}_i)+exp}, & \text{for the conjunction logic} \\ \frac{exp}{\phi(\bigvee_{w_i \in \boldsymbol{q}} \mathsf{div}_i)+exp}, & \text{for the disjunction logic,} \end{cases} \tag{19}$$

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

Proof. See Appendix D.

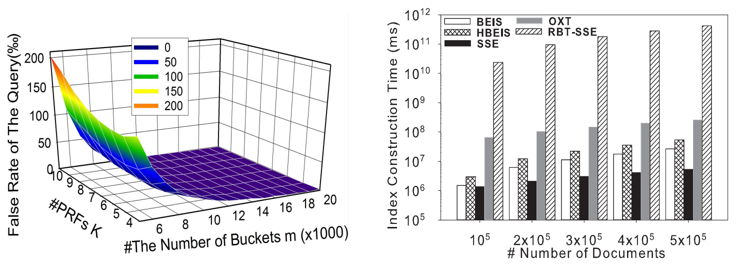

Fig. 2. (a). The false positive rate of the query. (b). The time cost of index construction.

6.2 Performance Evaluation

To demonstrate the feasibility and scalability, we implement our constructions (BEIS and HBEIS) and the following state-of-the-art to present a comparative analysis of the feasibility and scalability: 1) **SSE** [10], the first to achieve simultaneously the sublinear search time, dynamic update and CKA2 security; 2) **OXT** [14], the first to achieve multi-keyword boolean query with sublinear search time; 3) **RBT-SSE**, the first parallel and dynamic SSE scheme [13] based on Red-Black-Tree (RBT). We implement all schemes in Java programming language and execute them in 64-bit OpenJDK 1.7, and the cryptographic algorithm is implemented using the open APIs: including the 128-bit AES, SHA-256 and NIST 224p elliptic curve. The experiments are performed on a 64-bit Windows 7 operation system with Intel Core i3-2330M processor 2.20GHz and 4GB RAM. The experimental projects are run on Eclipse and single-threaded on the processors. To show the practical viability of our solution, we choose a real-word dataset for the experiments: Enron Email Dataset (EED) [18], where an email is regarded as a data file to be outsourced.

The false positive rate of the query. Fig. 2(a) shows the experimental evaluation of the false positive rate of the query. In the experiment, we set $\#\mathbf{d} = 20000$ and $\#\mathbf{w} = 20000$. Then, we first build different indices by using different values of m and K. 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 K. From a practical point of view, it implies our scheme achieves desirable privacy guarantee without sacrificing the query accuracy and space efficiency much.

The time cost of index construction. In SSE and OXT, the time for building index is linear with the number of global keyword-data pairs and the construction of them involves the encryption of the distinct keywords and all keyword-data pairs. Like SSE and OXT, our BEIS construction also involves the encryption of the index buckets. Fortunately, BEIS achieves privacy guarantee by using highly efficient and compact operations which involve only simple algebra operations (*i.e.*, addition, subtraction and multiplication) over integers, and it is much more efficient than other cryptographic operations used in the literature [10, 13, 14]. In Fig. 2(b), it is shown that the index construction time of BEIS is a slightly higher than that of SSE scheme but much lower than those of OXT and RBT-SSE.

The time cost of query performance. Fig. 3 shows the time cost of single keyword queries versus the number of data files (as large as 500000 data files).

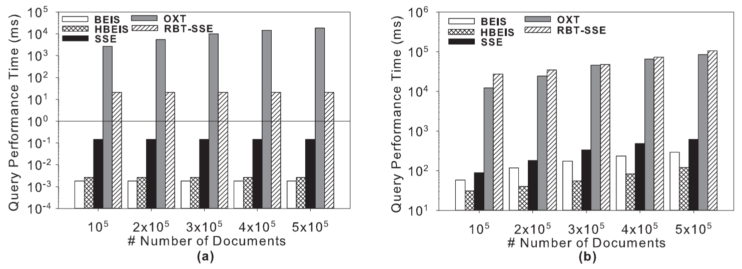

Fig. 3. The time cost of query performance. (a) At the client side. (b) At the cloud side.

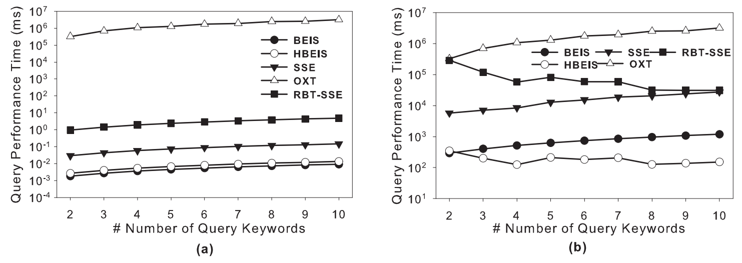

Fig. 4. The time cost of the ‘AND’ logic query performance. (a) At the client side. (b) At the cloud side.

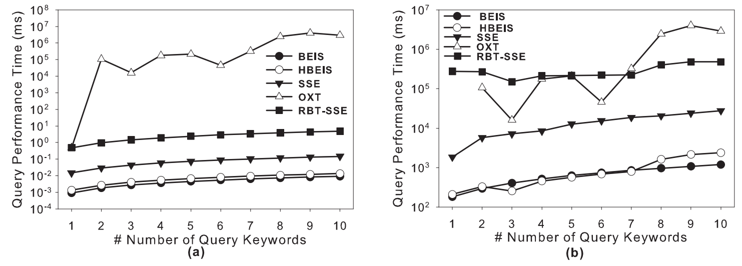

Fig. 5. The time cost of the ‘OR’ logic query performance. (a) At the client side. (b) At the cloud side.

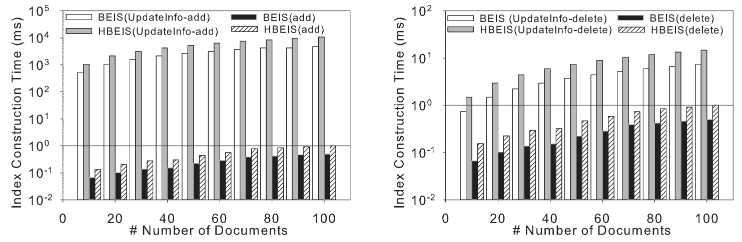

Fig. 6. The time cost of index update. (a). Add operation. (b). Delete operation.

Not surprisingly, due to the use of lightweight algebra operations for token generation and query processing, HBEIS and BEIS provide much higher query efficiency than three benchmarks at both the client and the cloud sides. Figs. 4 and 5 show the time cost of multi-keyword query (AND and OR logics) versus the number of keywords (used for multi-keyword query) from 2 to 10. As can be seen from Figs. 4 and 5, the time cost at the client side is almost linear with the number of query keywords in our constructions (BEIS, HBEIS), SSE and RBT-SSE. In OXT, however, the time cost at the client side is associated to the prevalence of one of the query words in the data collection, and it is relatively higher because the client needs to execute the exponent arithmetic (such as g^x). For the time cost at the cloud side, SSE depends on the prevalence of the query words in the data collection, and OXT depends on the number of the pairs, each of which consists of a query keyword and one of data files queried by the selected query keyword. As can be seen, HBEIS and BEIS achieve comparable query performance which is better than SSE, OXT and RBT-SSE. To obtain a fair comparison, we implement multi-keyword query in SSE and RBT-SSE by post-processing of all results of single-keyword queries. The query time in both HBEIS and RBT-SSE depends on the number of "reached" nodes in the hierarchical structure during the query processing. With the increase of keywords, the AND logic query will reduce the number of "reached" nodes, while the OR logic query will increase the value.

The time cost of dynamic index update. In our experiment, the number of data files to be added or deleted is ranging from 10 to 100. Fig. 6(a) and (b) show the execution time to add and delete data files, respectively. As shown in Fig. 6, the costs of adding files mostly fall on the client since the client has to build the secure sub-index for the new data files. Even so, the cost is only associated with the data files to be added, and it will not incur additional cost for the original index stored in the cloud. For the deletion operation, the client only needs to send a small deletion information (*i.e.*, a deletion request) to the cloud, and the cloud will execute the index update after finding the right positions (to be updated) in the index.

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

References

1. Hacigms, H., Mehrotra, S., Iyer, B.R.: Providing database as a service. In: ICDE, IEEE (2002) 29–38
2. Hacigms, H., Lyer, B., Li, C., Mhrotra, S.: Executing sql over encrypted data in the database-server-provider model. In: SIGMOD, ACM (2002) 216–227
3. Kamara, S., Lauter, K.: Cryptographic cloud storage. In: Financial Cryptography Workshops. (2010) 136–149

4. Gonzalez, L.M.V., Rodero-Merino, L., Caceres, J., Lindner, M.A.: A break in the clouds: Towards a cloud definition. Computer Communication Review (CCR) **39**(1) (2009) 50–55
5. Song, D.X., Wagner, D., Perrig, A.: Practical techniques for searches on encrypted data. In: IEEE Symposium on Security and Privacy, IEEE (2000) 44–55
6. Goh, E.J.: Secure indexes. IACR (2003)
7. Chang, Y., Mitzenmacher, M.: Privacy preserving keyword searches on remote encrypted data. In: ACNS. (2005) 442–455
8. Curtmola, R., Garay, J.A., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: Improved definitions and efficient constructions. In: ACM CCS, ACM (2006) 79–88
9. van Liesdonk, P., Sedghi, S., Doumen, J., Hartel, P.H., Jonker, W.: Computationally efficient searchable symmetric encryption. In: Workshop on Secure Data Management, ACM (2010) 87–100
10. Kamara, S., Papamanthou, C., Roeder, T.: Dynamic searchable symmetric encryption. In: ACM CCS, ACM (2012) 965–976
11. Li, J., Wang, Q., Wang, C., Cao, N., Ren, K., Lou, W.: Fuzzy keyword search over encrypted data in cloud computing. In: INFOCOM, IEEE (2010) 441–445
12. Cao, N., Wang, C., Li, M., Ren, K., Lou, W.: Privacy-preserving multi-keyword ranked search over encrypted cloud data. In: INFOCOM, IEEE (2011) 829–837
13. Kamara, S., Papamanthou, C.: Parallel and dynamic searchable symmetric encryption. In: Financial Cryptography. (2013) 258–274
14. Cash, D., Jarecki, S., Jutla, C.S., Krawczyk, H., Rosu, M.C., Steiner, M.: Highly-scalable searchable symmetric encryption with support for boolean queries. In: CRYPTO. (2013) 353–373
15. Wong, W.K., Cheung, D.W.L., Kao, B., Mamoulis, N.: Secure knn computation on encrypted databases. In: SIGMOD, ACM (2009) 139–152
16. Kuzu, M., Islam, M.S., Kantarcioglu, M.: Efficient similarity search over encrypted data. In: ICDE, IEEE (2012) 1156–1167
17. van Dijk, M., Gentry, C., Halevi, S., Vaikuntanathan, V.: Fully homomorphic encryption over the integers. In: EUROCRYPT. (2010) 24–43
18. Cohen, W.W.: Enron email dataset. (http://www.cs.cmu.edu/∼enron/)

A Proof of Theorem 1

Proof. i) For the conjunction logic query, as shown in Eq. (1), $\mathsf{div}_{\boldsymbol{\tau}}[j] = 1$ if and only if $\boldsymbol{\gamma}[y][j] = 1$ for each $y \in P_{\mathbf{q}}$. Similarly, in BEIS, for $\forall j \in [1, \#\mathbf{d}]$, if there exists $y \in P_{\mathbf{q}}$ such that the original plaintext bit value of $\boldsymbol{\gamma}[y][j]$ is '0', then $\sum_{y \in P_{\mathbf{q}}} \boldsymbol{\gamma}[y][j] < st_{\mathbf{q},j}$ since $\boldsymbol{\gamma}[y][j] = \alpha(0\beta + \zeta_{yj}) + \lambda_{yj} < \alpha(\beta + \zeta_{yj}) + \lambda_{yj}$. Therefore, $\mathtt{id}(d_j) \in IDs$ if and only if $\boldsymbol{\gamma}[y][j] = 1$ for each $y \in P_{\mathbf{q}}$.

ii) For the disjunction logic query, for $\forall y \in P_{\mathbf{q}}$ and $\forall j \in [1, \#\mathbf{d}]$, let b_{yj} be the original plaintext bit value of $\boldsymbol{\gamma}[y][j]$. For $\forall j \in [1, \#\mathbf{d}]$, we have

$$\begin{aligned} score(d_j, \mathbf{q}) &= \sum_{y \in P_{\mathbf{q}}} \boldsymbol{\gamma}[y][j] - st_{\mathbf{q},j} \\ &= \sum_{y \in P_{\mathbf{q}}} \boldsymbol{\gamma}[y][j] - \sum_{y \in P_{\mathbf{q}}} [\alpha\zeta_{yj} + \lambda_{yj} - q_j] \\ &= \sum_{y \in P_{\mathbf{q}}} [\alpha(b_{yj}\beta + \zeta_{yj}) + \lambda_{yj}] - \sum_{y \in P_{\mathbf{q}}} [\alpha\zeta_{yj} + \lambda_{yj} - q_j] \\ &= \sum_{y \in P_{\mathbf{q}}} [\alpha\beta b_{yj}] + q_j = \alpha\beta \sum_{y \in P_{\mathbf{q}}} b_{yj} + q_j. \end{aligned}$$

Here, q_j will not affect the result ranking since $q_j \in [1, \alpha\beta)$. We can conclude that $score(d_j, \mathbf{q})$ is proportional to $\sum_{y \in P_{\mathbf{q}}} b_{yj}$ shown in Eq. (2). This completes the proof.

B Proof of Theorem 2

Proof. i) For the conjunction query, given $\boldsymbol{\gamma}, \boldsymbol{\tau}$ and a data $d_j \in \mathbf{d}$, suppose that the original plaintext bit value of $\boldsymbol{\gamma}[y][j]$ is '1' for each $y \in P_{\mathbf{q}}$. Because $\boldsymbol{\gamma}[y][j]$ is encrypted as in Eq.(3) and by the homomorphic properties of f_h, we have

$$\begin{aligned}
\sum_{y \in P_{\mathbf{q}}} \boldsymbol{\gamma}[y][j] \bmod p &= \sum_{y \in P_{\mathbf{q}}} [\alpha(1\beta + \zeta_{yj}) + \lambda_{yj}] \bmod p \\
&= \sum_{y \in P_{\mathbf{q}}} [\alpha(1\beta + \zeta_{y0} + \frac{1}{2\alpha} f_h(h(j)\mu_{y0})) + \lambda_{y0} + \frac{1}{2} f_h(h(j)\mu_{y0})] \bmod p \\
&= \sum_{y \in P_{\mathbf{q}}} [\alpha(\beta + \zeta_{y0}) + \lambda_{y0} + f_h(h(j)\mu_{y0})] \bmod p \\
&= [st_r + f_h(h(j)) f_h(\sum_{y \in P_{\mathbf{q}}} \mu_{y0})] \bmod p \\
&= st_{\mathbf{q},j} \bmod p.
\end{aligned}$$

Based on the above analysis, for $\forall j \in [1, \#\mathbf{d}]$ suppose that there exist some bucket positions $P \in P_{\mathbf{q}}$ such that the original plaintext bit value of $\boldsymbol{\gamma}[y][j]$ is '0' for each $y \in P$. Then we must have $st_{\mathbf{q},j} - \sum_{y \in P_{\mathbf{q}}} \boldsymbol{\gamma}[y][j] = \#P\alpha\beta$, where $\#P$ denotes the number of positions in P. Because in practice $\#P \ll p$ and p is a prime number, we have $\#P\alpha\beta \bmod p \neq 0$, and there must be $\sum_{y \in P_{\mathbf{q}}} \boldsymbol{\gamma}[y][j] \bmod p \neq st_{\mathbf{q},j} \bmod p$.

ii) For the disjunction query, we follow the same proof logic as Theorem. 1. $\forall y \in P_{\mathbf{q}}$ and $\forall j \in [1, \#\mathbf{d}]$, we let b_{yj} be the original plaintext bit value of $\boldsymbol{\gamma}[y][j]$. As shown in Eq. (13), $\forall j \in [1, \#\mathbf{d}]$, we have

$$\begin{aligned}
score(d_j, \mathbf{q}) &= (st_{\mathbf{q},j} - \sum_{y \in P_{\mathbf{q}}} \boldsymbol{\gamma}[y][j]) \bmod p \\
&= (st_r + f_h(h(j)) f_h(\sum_{y \in P_{\mathbf{q}}} \mu_{y0}) - \sum_{y \in P_{\mathbf{q}}} \boldsymbol{\gamma}[y][j]) \bmod p \\
&= (\alpha\beta \sum_{y \in P_{\mathbf{q}}} b_{yj}) \bmod p.
\end{aligned}$$

When $\alpha\beta \bmod p \leq \frac{p}{\#P_{\mathbf{q}}}$, $score(d_j, \mathbf{q})$ is proportional to $\sum_{y \in P_{\mathbf{q}}} b_{yj}$. This completes the proof.

C Proof of Theorem 3

Proof. We consider a polynomial-size simulator $\mathcal{S} = \{\mathcal{S}_0, \ldots, \mathcal{S}_s\}$ such that for all polynomial-size adversaries $\mathcal{A} = \{\mathcal{A}_0, \ldots, \mathcal{A}_s\}$, the outputs of the two experiments: $\mathbf{Real}^*_{SSE,\mathcal{A}}(k)$ and $\mathbf{Sim}^*_{SSE,\mathcal{A},\mathcal{S}}(k)$ defined in [8] are computationally indistinguishable. That is, the simulated view $Views_S(H)$ and the real view

$View_R(H)$ of a s-query history H are computationally indistinguishable. Consider the simulator $\mathcal{S}$ adaptively generates the simulated view $Views_S(H) = \{\mathtt{id}(c_1)^*, \ldots, \mathtt{id}(c_{\#\mathbf{c}})^*, c_1^*, \ldots, c_{\#\mathbf{c}}^*, \boldsymbol{\gamma}^*, \boldsymbol{\tau}_{q_1}^*, \ldots, \boldsymbol{\tau}_{q_s}^*\}$ as follows:

$\mathcal{S}_0(1^k, H(\mathbf{d}))$: $\mathcal{S}_0$ chooses n random Strings $\{c_1^*, \ldots, c_{\#\mathbf{c}}^*\}$ such that $|c_1| = |c_1^*|, \ldots, |c_{\#\mathbf{c}}| = |c_{\#\mathbf{c}}^*|$. Because $\mathcal{A}$ does not know the key sk_2, the PCPA-security of SKE guarantees that each c_j^* is indistinguishable from a real ciphertext c_j. $\mathcal{S}_0$ simply copies the identifier list in the trace, *i.e.*, $\mathtt{id}(c_1)^* = \mathtt{id}(c_1), \ldots, \mathtt{id}(c_{\#\mathbf{c}})^* = \mathtt{id}(c_{\#\mathbf{c}})$. Obviously, the identifer lists of $Views_S(H)$ and $View_R(H)$ are distinguishable. $\mathcal{S}_0$ constructs a bucket-based data structure $\boldsymbol{\gamma}^*$, where each bucket will be used to store a vector, and $\boldsymbol{\gamma}^*[y][j]$ denotes the j^{th} bit of the vector stored in the y^{th} bucket. The number of the buckets and the size of a vector can be obtained from the trace, and thus they are indistinguishable from that of $\boldsymbol{\gamma}$. $\mathcal{S}_0$ chooses two random values R_{y0}^1 and R_{y0}^2 for each bucket position y. For each y, $\mathcal{S}_0$ chooses a random values $\mathtt{id}(j)^*$, and then computes $R_{y0}^1 + f(\mathtt{id}(j)^* R_{y0}^2)$ and inserts it into $\boldsymbol{\gamma}^*[y][j]$ for each bit j of the vector stored in the bucket $\boldsymbol{\gamma}^*[y]$. According to Eqs. (3) and (14), for any bucket position y and any vector bit j, $\boldsymbol{\gamma}^*[y][j]$ is indistinguishable from $\boldsymbol{\gamma}[y][j]$, because $\mathcal{A}$ does not know the secret keys $\{\alpha, \beta, sk_1\}$. $\mathcal{S}_0$ includes $\boldsymbol{\gamma}^*$ and R_{y0}^1, R_{y0}^2 for each bucket position y in the state $st_{\mathcal{S}}$ and outputs $\{c_1^*, \ldots, c_{\#\mathbf{c}}^*, \boldsymbol{\gamma}^*, st_{\mathcal{S}}\}$.

$\mathcal{S}_1(1^k, H(\mathbf{d}, \mathbf{Q} = \mathbf{q}_1))$: Let $\#\mathbf{q}_1$ be the number of keywords in $\mathbf{q}_1$. $\mathcal{S}_1$ randomly chooses bucket positions for $\#\mathbf{q}_1 K$ times. Let $P_{\mathbf{q}_1}^*$ be the chosen bucket positions. $\mathcal{S}_1$ sets $\boldsymbol{\tau}_{\mathbf{q}_1}^* = \{P_{\mathbf{q}_1}^*, st_r^* = \sum_{y \in P_{\mathbf{q}_1}^*} R_{y0}^1, f(\sum_{y \in P_{\mathbf{q}_1}^*} R_{y0}^2)\}$. According to the definition of PRF, $P_{\mathbf{q}_1}^*$ is indistinguishable from $P_{\mathbf{q}_1}$. st_r^* and $f(\sum_{y \in P_{\mathbf{q}_1}^*} R_{y0}^2)$ are distinguishable from st_r and $f(\sum_{y \in P_{\mathbf{q}_1}} \mu_{y0})$ respectively, because $\mathcal{A}$ does not know the keys $\{\alpha, \beta, sk_1\}$. Thus, $\boldsymbol{\tau}_{\mathbf{q}_1}^*$ is indistinguishable form $\boldsymbol{\tau}_{\mathbf{q}_1}$. $\mathcal{S}_1$ includes $\boldsymbol{\tau}_{\mathbf{q}_1}^*$ in $st_{\mathcal{S}}$, and outputs $(\boldsymbol{\tau}_{\mathbf{q}_1}^*, st_{\mathcal{S}})$. With the trapdoor $\boldsymbol{\tau}_{q_1}^*$, the adversary $\mathcal{A}_1$ is able to execute the query over $\boldsymbol{\gamma}^*$, in a way that the query returns the same results as the access pattern $A_p(H)$. This can be achieved by appropriate use of the random oracle (the output of which is indistinguishable from a random value). Namely, the output $\mathtt{id}(j)^* = h(j)$ of the random oracle in **Query** is programmed by the simulator in the following manner: if d_j is contained in the access pattern $A_p(H)$, then $\mathtt{id}(j)^*$ is the value such that $[R_y^1 0 + f(\mathtt{id}(j)^*) f(R_{y0}^2)] \bmod p = \boldsymbol{\gamma}^*[y][j] \bmod p$ for each $y \in P_{\mathbf{q}_1}^*$. Otherwise, $\mathtt{id}(j)^*$ is the value such that $[R_{y0}^1 + f(\mathtt{id}(j)^*) f(R_{y0}^2)] \bmod p \neq \boldsymbol{\gamma}^*[y][j] \bmod p$ for at least one position $y \in P_{\mathbf{q}_1}^*$. The simulator can do this since it stored the state $st_{\mathcal{S}}$.

$\mathcal{S}_i(1^k, H(\mathbf{d}, \mathbf{Q} = \{\mathbf{q}_1, \ldots, \mathbf{q}_i\}))$: For $2 \leq i \leq s$, $\mathcal{S}_i$ first checks whether $\mathbf{q}_i$ has appeared before by using the search pattern $\pi(H)$. If $\mathbf{q}_i$ has not appeared before, then $\mathcal{S}_i$ generates the trapdoor in the $\boldsymbol{\tau}_{\mathbf{q}_i}^*$ in the same way as $\mathcal{S}_1$ does. Otherwise, $\mathcal{S}_i$ retrieves the trapdoor previously used for $\mathbf{q}_i$ and uses it as $\boldsymbol{\tau}_{\mathbf{q}_i}^*$. $\mathcal{S}_i$ includes $\boldsymbol{\tau}_{\mathbf{q}_i}^*$ in $st_{\mathcal{S}}$, and outputs $(\boldsymbol{\tau}_{\mathbf{q}_i}^*, st_{\mathcal{S}})$. Similarly, $\boldsymbol{\tau}_{\mathbf{q}_i}^*$ is indistinguishable form $\boldsymbol{\tau}_{\mathbf{q}_i}$ since $\mathcal{A}$ does not know secret keys $\{\alpha, \beta, sk_1\}$. This completes the proof.

D Proof of Theorem 4

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 $d_j \in \boldsymbol{d}$, we use p_j to denote the probability of 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 $w_i \in \mathbf{w}$, $\texttt{idv}_i$ should have been encoded into the K buckets $\boldsymbol{\gamma}[h_i(w_i, sk_1)]$ for $i \in [1, K]$. The false decoding of $\texttt{idv}_i$ happens *if and only if* all K buckets are encoded by data identifier vectors of other keywords besides that of w_i. Without loss of generality, we assume there are K additional data identifier vectors $\{\widehat{\texttt{idv}}_1, \ldots, \widehat{\texttt{div}}_K\}$ which are repeatedly encoded into the K buckets $\{\boldsymbol{\gamma}[h_1(w_i, sk_1)], \ldots, \boldsymbol{\gamma}[h_K(w_i, sk_1)]\}$, respectively. Here, each $\widehat{\texttt{div}}_i$ might be the joint of multiple data identifier vectors. In the decoding process, $\texttt{div}_{\boldsymbol{\tau}}$ is decoded from the K buckets $\{\boldsymbol{\gamma}[h_1(w_i, sk_1)], \ldots, \boldsymbol{\gamma}[h_K(w_i, sk_1)]\} = \{\texttt{div}_i \bigvee \widehat{\texttt{div}}_1, \ldots, \texttt{div}_i \bigvee \widehat{\texttt{div}}_K\}$. We denote $\hat{p}_j$ as the probability of w_i which is falsely considered as a membership of d_j. Let n_j be the number of distinct keywords in d_j, we have

$$\hat{p}_j = (1 - (1 - \frac{1}{m})^{Kn_j})^K \approx (1 - e^{-Kn_j/m})^K. \tag{20}$$

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 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 conjunction 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 $exp = \sum_{d_j \in \hat{\mathbf{d}}} p_j$, where $\hat{\mathbf{d}}$ denotes the collection of data files which might be falsely returned. In the conjunction logic query, $\#\hat{\mathbf{d}} = \#\mathbf{d}$, 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. (19).