Enabling Privacy-Assured Similarity Retrieval over Millions of Encrypted Records

Xingliang Yuan, Helei Cui, Xinyu Wang, and Cong Wang(✉)

City University of Hong Kong, Kowloon, Hong Kong
{xl.y,helei.cui}@my.cityu.edu.hk
{xinywang,congwang}@cityu.edu.hk

Abstract. Searchable symmetric encryption (SSE) has been studied extensively for its full potential in enabling exact-match queries on encrypted records. Yet, situations for similarity queries remain to be fully explored. In this paper, we design privacy-assured similarity search schemes over millions of encrypted high-dimensional records. Our design employs locality-sensitive hashing (LSH) and SSE, where the LSH hash values of records are treated as keywords fed into the framework of SSE. As direct combination of the two does not facilitate a scalable solution for large datasets, we then leverage a set of advanced hash-based algorithms including multiple-choice hashing, open addressing, and cuckoo hashing, and craft a high performance encrypted index from the ground up. It is not only space efficient, but supports secure and sufficiently accurate similarity search with constant time. Our designs are proved to be secure against adaptive adversaries. The experiment on 10 million encrypted records demonstrates that our designs function in a practical manner.

Keywords: Cloud security · Encrypted storage · Similarity retrieval

1 Introduction

Massive datasets are being outsourced to public clouds today, but outsourcing sensitive data without necessary protection raises acute privacy concerns. To address this problem, searchable encryption, as a promising technique that allows data encryption without compromising the search capability, has attracted wide-spread attention recently [2,5,6,9,12,14,22,24]. While these works provide solutions with different trade-offs among security, efficiency, data update, etc., most of them only support exact-match queries over encrypted data. Although useful in certain applications, they can be somewhat restrictive for situations where exact matches rarely exist, and approximate queries, particularly similarity queries are more desired. For instance, in multimedia databases or data mining applications, heterogeneous data like images, videos, and web pages are usually represented as high-dimensional records. In those contexts, finding similar records or nearest neighbors with respect to a given query record are much

G. Pernul et al. (Eds.): ESORICS 2015, Part II, LNCS 9327, pp. 40–60, 2015.
DOI: 10.1007/978-3-319-24177-7_3

more common and crucial to selectively retrieve the data of interest, especially in very large datasets [1,10,16,19,23].

In this work, we study the problem of privacy-assured similarity search over very large encrypted datasets. Essentially, we are interested in designing efficient search algorithms with sublinear time complexity. This requirement excludes all public key based approaches that usually demand linear search, and drives us to only focus on symmetric key based approaches, in particular, efficient encrypted searchable index designs. We first note that searchable symmetric encryption (SSE) has wide applicability as long as one can access the data via keywords. Thus, this problem could be theoretically handled by a direct combination of locality-sensitive hashing (LSH) [1] and SSE [13]. More technically, LSH, a well-studied algorithm for fast similarity search, hashes high-dimensional records such that the close ones collide with much higher probability than distant ones. Then by treating LSH hash value(s) as "keyword(s)", one may directly apply known SSE to realize private similarity search [15].

Such a straightforward solution, however, does not achieve practical efficiency as the sizes of datasets are continuously growing. Take the work in [15] for example: due to random padding, their proposed encrypted index needs to be augmented quadratically in the size of dataset. Even by combining LSH with one of the latest advancements of SSE [2] that achieves asymptotically optimal space complexity, the resulting index can still be prohibitively large due to the inherent issues from LSH [16] like its imbalanced structures and its demand of a large number of hash tables for accurate search. Besides, those issues will also readily turn most queries into a so-called "big query" [10,16], where almost every query could comprise a large number of matched records, leading to substantial I/O resources and long search latency. Most of previous SSE constructions focused on exact keyword search for document retrieval. They are generic primitives without considering the above performance issues, and thus do not necessarily scale well in the context of similarity retrieval over large number of records.

Therefore, rather than just assembling off-the-shelf designs in a blackbox manner, we must consider security, space utilization, time efficiency, and search accuracy simultaneously, and build a new construction from the ground up. As an initial effort, we resort to recent advancements in high performance hash-based structures [10,16,18,19] in the plaintext domain. Our goal is to intelligently incorporate their design philosophies into our encrypted index structure so as to make a practical design fully customized and thoroughly optimized. In particular, we explore multiple choice hashing, open addressing, and cuckoo hashing [18] to balance the index load, resolve the severe imbalance of LSH, and yield constant search time with a controllable trade-off on accuracy. Each query only requires $O(1)$ lookup and retrieves a small constant number of similar records with low latency. Such design also makes it possible that any form of post processing on retrieved records (e.g., distance ranking) can be efficiently completed at local.

For security, we apply pseudo-random functions to protect sensitive LSH hash values, use symmetric encryption to encrypt the index content, and implement the above hash-based optimizations in a random fashion. Through crafted algo-

rithm designs, our proposed encrypted index can perform favorably even over a large number of data records. For completeness, we also propose a dynamic version of the index design to support secure updates of encrypted data records. We note that one trade-off of SSE is the compromise on security to pursue functionality and efficiency. Similar to previous definitions for SSE, our security strength is evaluated by capturing the controlled leakage in the context of LSH-based similarity search, and we formally prove the security against adaptive chosen keyword attacks. Our contributions are summarized as follows:

- We propose a novel encrypted index structure with optimal space complexity $O(n)$, where n is the number of the data records. It supports secure similarity search with constant time while achieving good accuracy.
- We extend this index structure to enable the server to perform secure dynamic operations over the encrypted index, i.e., Insert and Delete.
- We formalize the leakage functions in the context of LSH-based similarity search, present the simulation-based security definition, and prove the security against adaptive chosen-keyword attacks.
- We implement our schemes with practical optimizations, and deploy them to Amazon cloud for 10 million 10,000-dimensional records extracted from Common Crawl[1]. The evaluations show that our security designs are efficient in time and space, and the retrieved records are desired with good accuracy.

The rest of the paper is organized as follows. The related works are summarized in Sect. 2. The preliminaries are introduced in Sect. 3. The security definition is given in Sect. 4. Then we present the proposed schemes in Sect. 5. After that, we formally define the leakage functions and prove our schemes achieve the security against adaptive chosen-keyword attacks. Section 7 shows our experiment results. Finally, Sect. 8 makes the conclusion.

2 Related Works

Song et al. first introduce the notion of searchable encryption [21]. Then Goh develops a per-file index design via Bloom filter [7], and Chang et al. [4] also give a per-file index design. Curtmola et al. improve the security notions known as SSE and introduce new constructions against non-adaptive and adaptive chosen-keyword attacks [6]. Chase et al. generalize SSE by introducing the notion of structured encryption and give the first efficient scheme (i.e., sublinear time) with adaptive security [5]. Afterwards, Kamara et al. propose a dynamic SSE scheme with sublinear time and introduce the security framework that captures the leakage of dynamic operations [14]. Then Kamara et al. give the first dynamic SSE scheme supporting parallelizable search [12]. Meanwhile, several works extend SSE to support sophisticated functionalities. Cash et al. propose

[1] Common Crawl Corpus: an open repository of web crawl data, on line at http://commoncrawl.org/.

an SSE scheme for boolean queries that firstly achieves the asymptotically optimal search time [3], and Jarecki et al. extend that scheme in the multi-client scenario [11].

Very recently, Cash et al. implement a dynamic SSE for large databases when the index is stored in the hard disks [2]. The proposed hybrid packing approach considers the locality of documents, and improves I/O parallelism. Stefanov et al. propose a dynamic SSE that achieves *forward privacy* [22]. Yet, their design relies on an ORAM-like index with hierarchical structure. The search time complexity is polylogarithmic, and the client needs to rebuild the index periodically. Naveed et al. present a building block called *blind storage* as a sub-component of SSE for the privacy of document set, i.e., hiding the number of documents and the document sizes [17]. Their design splits each document into blocks and randomly inserts them into a huge array. The required storage cost is several times larger than the original document set. Besides, multiple round interactions are also needed when retrieving a large document. Hahn et al. propose an SSE scheme with secure and efficient updates, where the update operations leak no more information than the access pattern [9]. Specifically, an encrypted file index is built in advance, which stores encrypted keywords for each file. When one sends search queries, an inverted index will be built gradually. Because adding files only updates the file index, the server will not know whether those files contain the keywords searched before or not. The search cost is initially linear, then amortized over time.

SSE is applicable for any forms of private retrieval based on keywords [13]. Built on locality-sensitive hashing (LSH) or other distance embedding techniques, similarity search will be transformed to keyword search. Kuzu et al. [15] build an encrypted index from a LSH-based inverted index. Each distinct LSH hash value is associated with an n-bit vector, where n is the total number of records in a dataset, and each bit indicates a matched record. In their encrypted index, a large amount of random padding is added to hide the number of distinct LSH hash values and the imbalance in the number of matched records. Consequently, the index has a quadratic space overhead as worst as $O(n^2)$.

3 Preliminaries

Cuckoo Hashing: Cuckoo hashing [18] is a variant of multiple choice hashing. It allows items moving between hash tables so as to achieve high load factors[2]. Let $\mathcal{X}$ be the universal domain, and cuckoo hashing is defined as:

Definition 1 (Cuckoo Hashing). *Given two hash tables T_1 and T_2 with w capacity, two independent and random hash functions $u_1, u_2 : \mathcal{X} \rightarrow \{0, w-1\}$ are associated to T_1 and T_2. Item $x \in \mathcal{X}$ can be placed either in bucket $T_1[u_1(x)]$ or in bucket $T_2[u_2(x)]$.*

[2] The load factor refers to the ratio between the number of items and the number of buckets in the index.

When inserting an item without a vacancy, the item in one of those two occupied buckets will be kicked out and moved to another hash table. We denote such operation as *cuckoo-kick*. *cuckoo-kick* will not stop until all "kicked" items are re-inserted within a threshold of iterations. When the number of such iterations exceeds the threshold, rehash will be activated such that all items are inserted again by newly selected hash functions. To reduce the probability of rehash, it is natural to extend cuckoo hashing from two hash tables to multiple ones so that each item has more buckets to place.

Locality-Sensitive Hashing: Locality-sensitive hashing (LSH) [1] is the state-of-the-art algorithm to solve the problem of approximate nearest neighbors in high-dimensional spaces. The functions in the LSH family project high-dimensional records such that the hashes of similar records collide with much higher probability than those of distant ones. From [1], the LSH family is defined in Appendix A.

Cryptographic Primitives: A private-key encryption scheme $\mathsf{SE}(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ consists of three algorithms: The probabilistic key generation algorithm Gen takes a security parameter k to return a secret key K. The probabilistic encryption algorithm Enc takes a key K and a plaintext $M \in \{0,1\}^*$ to return a ciphertext $C \in \{0,1\}^*$; The deterministic decryption algorithm Dec takes k and $C \in \{0,1\}^*$ to return $M \in \{0,1\}^*$. Define a pseudo-random function (PRF) family is a family $\mathcal{F}$ of functions such that it is computationally infeasible to distinguish any function in $\mathcal{F}$ from a uniformly random function.

4 Notations and Definitions

This section gives the notations and the security definitions used throughout the paper. D is defined as a y-dimensional record, and D^* is the ciphertext of D. A is the record identifier, which can also be its physical address. $\mathbf{D}$ is a record set $\{D_1, \cdots, D_n\}$, and n is its cardinality. V represents a vector, where v_j is its j-th component. We denote T as the hash table, w as its capacity, and $T[i]$ as its i-th bucket, where $i \in [0, w)$. $0^{|a|}$ denotes a string with $|a|$ bits of '0'. Given two strings x and y, their concatenation is written as $x||y$. P, G, and F are the pseudo-random function (PRF). Our scheme contains the functions for key generation, index construction, and query operations. We give the definitions with specified inputs and outputs as follows:

$K \leftarrow \mathsf{GenKey}(1^k)$: takes as input a security parameter k, and outputs a secret key K.
$\mathcal{I} \leftarrow \mathsf{Build}(K, \mathbf{D})$: takes as input K and a record set $\mathbf{D}$, and outputs an encrypted index $\mathcal{I}$.
$t \leftarrow \mathsf{GenTpdr}(K, D)$: takes as input K and D, and outputs a trapdoor t.
$\mathbf{A} \leftarrow \mathsf{Search}(\mathcal{I}, t)$: takes as input $\mathcal{I}$ and t, and outputs a set of identifiers $\mathbf{A}$.
$\mathcal{I}' \leftarrow \mathsf{Insert}(\mathcal{I}, t, D)$: takes as input $\mathcal{I}$, t, and D for insertion, and outputs the updated index $\mathcal{I}'$.
$\mathcal{I}' \leftarrow \mathsf{Delete}(\mathcal{I}, t, D)$: takes as input $\mathcal{I}$, t, and D for deletion, and outputs the updated index $\mathcal{I}'$.

Our scheme follows the security notion of SSE stated in [6,14]: the server cannot infer any sensitive information of data records from the encrypted index before search; the server can only learn the limited information about the requested queries and the results. Like prior SSE schemes, there will be the leakage of access pattern and query pattern, for enabling search and updates over the encrypted index. Explicitly, the access pattern includes the results of queries; the query patten indicates not only the equality of query records but also the *similarity* between each other, where the latter is additional in contrast to other SSE schemes for exact keyword search (also recognized in [15]). Based on the simulation-based model [8] and the definition verbatim from [6,14], we give the formal security definition in Appendix B.

5 Our Proposed Schemes

5.1 Main Scheme

In this section, we present the main scheme for secure and scalable similarity search on encrypted high-dimensional records. Our core design is a high performance encrypted index built from the scratch. It supports secure and non-interactive search with constant time while preserving good accuracy of search results. Below, we give our design rationale before introducing the details.

Design Rationale: As mentioned, one can treat LSH hash values as keywords and employ any known SSE to make secure similarity search functionally correct [13,15]. However, directly applying existing SSE indices [2,6,12,14,15,22,24] will cause large space consumption and long query latency. First, the number of matched records varies for different LSH hash values. Such imbalance will make the space complexity as worst as quadratic in the size of dataset [15], because of random padding used in the inverted index based SSE schemes [6,14,24]. Second, the query latency scales with the number of matched records. It could be painfully long for large datasets. Third, multiple composite LSH functions are usually applied to each record for good accuracy. For l composite LSH functions, each record will have l hash values. Therefore, even the latest key-value pair based SSE indices [2,22] will result in an index with space complexity $O(ln)$, where n is the number of the records. It might still be huge since l can be as large as several hundred [16]. As analyzed, combining LSH and SSE directly appears to be neither practically efficient nor scalable for large datasets.

To address the above issues, we propose to build an advanced encrypted index, which aims to inherit the design benefits of LSH indices in the plaintext domain for performance while minimizing the extra overhead incurred by security. In particular, we resort to recent advancements on hash-based indices, which utilize high performance hashing algorithms such as multiple choice hashing [16,19], open addressing [16], and cuckoo hashing [10]. We also observe that most applications of similarity search suffice for an approximate result, e.g., high-value statistics such as nearest neighbors and Top-K analysis [1,15]. Besides, an appropriate approximation algorithm will improve search efficiency by orders of

$\mathsf{Build}(K, \mathbf{D})$:
CLIENT:

1. Setup stage:
   (a) call $\mathsf{GenKey}(1^k)$ to generate the key set $K = (K_1, K_2, K_3)$;
   (b) set the index load factor τ and initiate l hash tables: $\{T_1, \cdots, T_l\}$ with the capacity $w = \lceil \frac{n}{\tau l} \rceil$ for each;
   (c) assign a universal hash function u_j: $U \rightarrow \{0, w-1\}$ to T_j, $\forall j \in [1, l]$;
   (d) set the *cuckoo-kick* threshold α and the initial *random probing* step d;
   (e) $\forall D \in \mathbf{D}$, compute $lshV(D) = \{v_1, \cdots, v_l\}$, where $v_j = g_j(D)$.
2. Insertion stage, $\forall D \in \mathbf{D}$:
   (a) set $\beta = 0$ to mark the iterations of *cuckoo-kick*;
   (b) select T_j randomly from $\{T_1, \cdots, T_l\}$ and compute $\{G(K^1_{v_j}, i)\}_d$, where $K^1_{v_j} = P(K_1, v_j)$ and i is from 1 to d;
   (c) scan $\{T_j[u_j(G(K^1_{v_j}, i))]\}_d$ from T_j to the rest of tables incrementally and place A to the very first vacant bucket;
   (d) if none of those buckets is empty, randomly select one of them and *cuckoo-kick* A' inside by replacing it with A; increment β and re-insert A' via step b), c) and d) iteratively;
   (e) if β reaches α, randomly select T_j and increment the cached d of v_j. Place A to $T_j[u_j(G(K^1_{v_j}, d))]$ if it is empty; iterate this step until A is inserted.
3. Encryption stage:
   (a) encrypt occupied buckets: $B^* = A||0^{|a|} \oplus r$, where $|a|$ is the length of check tag, $r = F(K^2_{v_j}, b)$, $K^2_{v_j} = P(K_2, v_j)$, and b is the address offset of bucket.
   (b) fill empty buckets with random strings: $B^* = \mathsf{Enc}(K_3, 0^{|B^*|})$.

$\mathsf{GenTpdr}(K, D)$:
CLIENT:

1. compute $\{v_j = g_j(D)\}_l$ for j from 1 to l;
2. generate $t = (\{K^1_{v_j}\}_l, \{K^2_{v_j}\}_l)$, where $K^1_{v_j} = P(K_1, v_j)$, and $K^2_{v_j} = P(K_2, v_j)$.

$\mathsf{Search}(\mathcal{I}, t)$: for each $K^1_{v_j}$ in $\{K^1_{v_j}\}_l$:
SERVER:

1. locate $\{T_j[u_j(G(K^1_{v_j}, i))]\}_{d_{max}}$ for i from 1 to d_{max};
2. compute $r = F(K^2_{v_j}, b)$ and $r \oplus B^*$ for each, where $B^* = T_j[u_j(G(K^1_{v_j}, i))]$;
3. if the least significant $|a|$ bits are all '0', push A to $\mathbf{A}$.

Fig. 1. Index Build function and Search operation in the main scheme

magnitude while only introducing a small loss in accuracy [1,10,16]. Therefore, we incorporate this design philosophy into the framework of SSE and show how to build a provably secure and space efficient index with constant search time and good accuracy.

Main Scheme in Detail: Essentially, the client will build the index in three stages, i.e., setup, insertion, and encryption. The setup stage initializes the index structure and the prerequisite system parameters tuned on the input dataset; The insertion stage places all record identifiers to the buckets of index in a

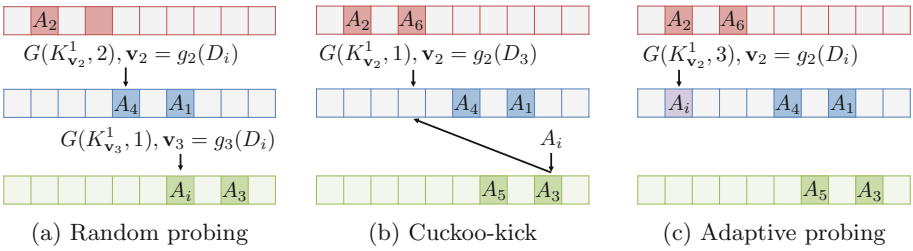

Fig. 2. The example illustrates three cases when inserting an identifier A_i. The number of hash functions l is 3. The random probing step d is 2. The one in purple is an adaptive probing bucket.

random manner; The encryption stage encrypts the identifiers inside and fills empty buckets with random padding. The Build function is presented in Fig. 1, and an example in Fig. 2 illustrates how an identifier is inserted. For easy presentation, we describe the insertion stage and the encryption stage, along which the system parameters are introduced.

In the insertion stage, the identifiers are sequentially inserted to the index buckets without loss of data confidentiality. We first introduce *secure multiple choice hashing* to balance the index load, and then combine it with *random open addressing* to handle the LSH imbalance. Moreover, we apply *cuckoo hashing* to build a very compact index. For security and correctness, those techniques are realized by utilizing different PRFs.

It is known that multiple choice hashing provides multiple positions for each inserted item so as to achieve load balance. It can be naturally extended in LSH indices [10,16,19], i.e., l composite LSH functions $\{g_1, \cdots, g_l\}$ are associated with l hash tables $\{T_1, \cdots, T_l\}$ respectively. The record identifiers are inserted into those hash tables which are indexable by LSH hash values. Given a record D, the bucket of its identifier A is determined by $lshV(D) = \{v_1, \cdots, v_l\}$, where $v_j = g_j(D)$. Yet, such procedure does not consider security. Because LSH functions are not the cryptographic hash function, D could be leaked from where it is stored. Thus, we use PRF to protect $lshV(D)$: $\{P(K_1, v_1), \cdots, P(K_1, v_l)\}$ shown at Stage 2.b of Build function in Fig. 1, where P is PRF. The transformed hash values are now used to find available buckets. We note that such treatment is also seen in prior works [15,20]. Besides, multiple choice hashing can eliminate redundancy compared to the inverted index. It is not necessary to store all l copies for each identifier to have good search accuracy. This advantage enables flexible approximate algorithms to make a trade-off between efficiency and accuracy [10,16,19]. In our case, we pursue practical efficiency at large scale and store a single copy of each identifier to achieve $O(n)$ space complexity. The size of index only scales with the number of data records.

Secure multiple choice hashing balances the load of index, but it might still not provide sufficient vacant buckets to handle the imbalance of LSH, i.e., a large number of records matched with the same LSH hash value could readily

exist. One straightforward solution is to introduce more bucket choices by adding more hash tables, but accessing a large number of hash tables will degrade the performance. Therefore, we adopt open addressing to resolve LSH collisions [10,16]. The key idea is to seek a number of alternative buckets within each hash table. But applying basic mechanisms of open addressing will disclose the locality of similar records. For example, if linear probing is used, the server will learn that similar records are placed next to each other. To hide such information, a probing sequence should be scrambled. Thus, we utilize *random probing* to generate a random probing sequence, $\{G(K^1_{v_j}, 1), \cdots, G(K^1_{v_j}, d)\}$, where $K^1_{v_j}$ denotes the transformed LSH hash value $P(K_1, v_j)$, G is PRF, and d is the probing step.

To compact our encrypted index, we further utilize the idea of cuckoo hashing, a variant of multiple choice hashing. It allows the identifiers to relocate, moving across different hash tables. In our design, when inserting a record D, if all $l * d$ probing buckets are occupied, one of them will be randomly selected. The identifier A' inside will be kicked, and A will be placed. Then A' is re-inserted back. Such *cuckoo-kick* operation will loop until no identifier is not placed. We observe that *cuckoo-kick* will facilitate the refinement of clustering similar records and improve the search accuracy. Less similar records will be excluded via iterative *cuckoo-kick*. The empirical results will be shown later in Sect. 7.

It is worth noting that rehash may happen in cuckoo hashing when relocating items in an endless loop. Consequently, all identifiers should be re-inserted, which could be quite expensive for a large dataset. To sidestep this issue, we propose *adaptive probing* to seek more vacant buckets at each hash table in a heuristic way. When the number of *cuckoo-kick* reaches a given threshold α, we start to randomly select $v_j \in lshV(D)$ and increment its probing step d in T_j so that one more bucket can be used for relocation. As a result, each d for a given v_j is cached, and the maximum probing step d_{max} will be notified to the server after the index is built. It is used for search operations in such a way that each table will process constant d_{max} buckets for a given query.

To achieve the security guarantees stated in Sect. 4, we have to encrypt the entire index and make each bucket indistinguishable. Considering security, efficiency, and correctness, we investigate the underlying data structure on bucket encryption. As introduced in Sect. 3, cuckoo hashing uses weak hash functions for a compact index. At Stage 1.c of Build function in Fig. 1, the output range of universal hash is $[0, w-1]$, where w is the hash table capacity. Because a weak hash function is not collision-resistant, two different LSH hash values might collide at the same bucket. To get correct results, one may append an encrypted LSH hash value with its identifier in the bucket, but it introduces additional storage overhead.

Tactfully, we embed LSH hash values into random masks for bucket encryption, so the matched results will be found on the fly. In particular, we concatenate the identifier with a check tag $A||0^{|a|}$, and encrypt the concatenated string by XORing a random mask r: $B^* = A||0^{|a|} \oplus r$, where $r = F(K^2_{v_j}, b)$, $K^2_{v_j} = P(K_2, v_j)$, and b is the address offset of bucket from the base address of

index. Only if $0^{|a|}$ is correctly recovered, the bucket will be the matched one. And because b is unique for each bucket, each bucket is encrypted with different random mask even the same LSH hash value is embedded. Finally, the rest of empty buckets are filled with random padding to make all buckets indistinguishable.

Search Operation: Based on the index construction, the server can perform Search to return a constant number of encrypted similar records for a query record. In Search of Fig. 1, the client generates the trapdoor $t = (\{K^1_{v_j}\}_l, \{K^2_{v_j}\}_l)$ for a query D, which enables the server to locate and unmask the matched buckets. Upon receiving $\{K^1_{v_j}\}_l$, the server locates d_{max} buckets in each hash table via PRF $G(K^1_{v_j}, i)$ for i from 1 to d_{max}, where d_{max} is the maximum probing step of all distinct LSH hash values. Then it computes random masks from $\{K^2_{v_j}\}_l$ via PRF $F(K^2_{v_j}, b)$, where b is the bucket address offset. Only if the identifiers inside are associated with matched v_j, the check tag will be all "0" and the identifier will be considered as the correct result. Meanwhile, the server cannot unmask the identifiers inside if they have different LSH hash values to the query record's.

Regarding security, the random mask is generated from the unique address offset of bucket, and thus each bucket is encrypted via a different mask. Such design ensures that the server knows nothing before search. We also note that multiple choice hashing is designed for parallel search. The buckets in l independent hash tables can be processed in parallel. Thus, the time complexity can achieve $O(c/min(p, l))$, where p is the number of processors, and c is a constant $l * d_{max}$. The number of retrieved ciphertext is bounded by $O(c)$.

5.2 Dynamic Scheme

To support secure dynamic operations, we design a variant of bucket construction. Accordingly, Insert and Delete are proposed to add and remove an identifier from the encrypted index, respectively. We note that updating a given record causes the change of its LSH hash values. As a result, its identifier will be relocated by first triggering Delete and then Insert.

Explicitly, we store the state information of ciphertext at the server, and ask the client to use fresh random masks generated from the state information to re-encrypt all the buckets which have been accessed in a given query. During the update, the bucket, whose underlying content is truly modified, is hidden due to the re-encryption. In particular, we design the bucket in the format such as: $B^* = (P(K_5, s) \oplus A||v_j, Enc(K_4, s))$, where K_4 and K_5 are private keys, v_j is used to guarantee the correctness of Search, and the fresh random mask is generated by updating a random seed s. We note that the setup phase and the insertion phase remain unchanged when building the dynamic index. Only the encryption phase is different.

To insert a new record, one straightforward solution is to follow the insertion stage in Build. However, such procedure could trigger *cuckoo-kick* and cause many interactions between client and server. Besides, the client needs to re-encrypt all accessed buckets in each interaction, which will introduce computational burdens.

Alternatively, we employ a similar approach of *adaptive probing* to moderate the communication and computation overhead. To insert an identifier A of given D, the client generates the trapdoor to enable the server to iteratively return l buckets (one for each hash table) until A finds a vacancy to stay. As we only notify d_{max} to the server, it is required to use a map to record the latest probing step δ_{v_j} for each distinct $K^1_{v_j}$, where δ_{v_j} starts from $d_{max}+1$. We also note that setting a less aggressive index load factor will help to insert A. As a result, the client can retrieve the latest probing buckets $\{T_j[u_j(G(K^1_{v_j}, \delta_{v_j}))]\}_l$ in each table. If no bucket is empty, the client will keep on asking the server to return l new probing buckets by incrementing each δ_{v_j} for j from 1 to l, i.e., one bucket in each of l tables. To hide the location of A, the last l buckets are re-encrypted via fresh random masks.

To delete A, the client generates the trapdoor to retrieve the corresponding $\sum_{j=1}^{l} \delta_{v_j}$ buckets, where one of them stores A. After decryption, the client locates the bucket of A and replaces it with $\perp$. Likewise, it re-encrypts the accessed buckets with fresh random masks to hide the emptied one. Compared to our main scheme, Search now needs to return $\sum_{j=1}^{l} \delta_{v_j}$ encrypted buckets for a given trapdoor. The decryption is conducted at the local client. v_j that matches the LSH hash value of the query record is considered as the correct result.

6 Security Analysis

In this section, we evaluate the security strength of main scheme Ω_1 and dynamic scheme Ω_2 under the security framework of SSE. We first define the leakage in search and update operations, and specifically discuss the security for LSH-based similarity search. Based on well-defined leakage functions, we prove that both schemes are secure against adaptive chosen-keyword attacks.

Security on the Main Scheme: Our scheme endows the server with an ability to find encrypted similar records by borrowing techniques from SSE. As a result, it does have the same limitation as prior SSE constructions. In particular, the server initially knows certain attributes of index without responding any Search query; that is, the capacity of encrypted index, the number of hash tables, and the bit length of encrypted bucket. As long as Search begins, the access pattern and the query pattern are subsequently revealed. Essentially, the access pattern for each query includes a set of identifiers of similar records and the accessed encrypted buckets of index. While for the query pattern, the notion in our scheme extends from the notion of SSE for keyword search, but it reveals more information, the similarity between each query. In keyword search, each query produces one single trapdoor. The query pattern is captured by recording those deterministic trapdoors, where the repeated ones indicate the keywords searched before. While in our scheme, each query generates a trapdoor that consists of multiple sub trapdoors. Therefore, if two trapdoors have an intersection, it will indicate that they are similar; that is, their underlying query records have matched LSH hash values. Accordingly, we quantify this similarity by θ defined as the size of the intersections between the two composite LSH hash values.

Formally, we define the leakage $\mathcal{L}^1_{\Omega_1}$ and $\mathcal{L}^2_{\Omega_1}$ as follows: $\mathcal{L}^1_{\Omega_1}(\mathbf{D}) = (N, l, |B^*|)$, where N is the index capacity, l is the number of hash tables, and $|B^*|$ is the length of encrypted bucket; $\mathcal{L}^2_{\Omega_1}(D) = (accp_{\Omega_1}(D), simp_{\Omega_1}(D))$. $accp_{\Omega_1}(D)$ is the access pattern for a query record D defined as $(\{A\}_m, \{B^*\}_{ld_{max}})$, where $\{A\}_m$ is the identifiers of returned m similar records, and $\{B^*\}_{ld_{max}}$ is the accessed buckets. $simp(D)_{\Omega_1}$ is the query pattern defined as $(\{\theta = |lshV(D) \cap lshV(D_i)|\}_q, i \in [1, q])$, where D_i is one of q queried records, and θ is the size of the intersections between l composite hash values of D and D_i.

Regarding the access pattern $accp_{\Omega_1}$, for a given query record D, the server decrypts d_{max} buckets at each hash table, total ld_{max} buckets for l tables, to recover the result identifiers $\{A\}_m$, where $m \leq ld_{max}$. Therefore, the server knows where the identifiers are stored and how many identifiers are correctly recovered at each table. From the perspective of security, revealing d_{max} does not appear to be harmful. It only informs the server when to stop random probing in each hash table. Regarding the query pattern $simp_{\Omega_1}$, the similarity of query records is known in addition to the equality. A trapdoor t for a given D contains l sub trapdoors: $\{P(K_1, g_1(D)), \cdots, P(K_1, g_l(D))\}$. Considering another D_i, if $t_i = \{P(K_1, g_1(D_i)), \cdots, P(K_1, g_l(D_i))\}$ has at least one matched sub trapdoor as t, D_i and D are likely similar. From the definition of LSH [1], the θ between D and D_i will further tell their closeness. The bigger θ is, the closer they are.

We adopt the simulation-based definition in [6]. Given the leakage functions $\mathcal{L}^1_{\Omega_1}$ and $\mathcal{L}^2_{\Omega_1}$, a probabilistic polynomial time (P.P.T.) simulator $\mathcal{S}$ can simulate an index, respond a polynomial number of Search queries, and generate corresponding trapdoors. To prove the adaptive security defined in Appendix B, we show that any P.P.T. adversary $\mathcal{A}$ cannot differentiate: (1) the real index and the simulated index; (2) the real search results and the simulated results, the real trapdoors and the simulated trapdoors for a polynomial number of adaptive queries. We present Theorem 1 and the formal proof in Appendix C.

Security on the Dynamic Scheme: The dynamic scheme is built on the design of the main scheme. The underlying index structure is exactly the same, which does not show extra information. Thus, the leakage function $\mathcal{L}^1_{\Omega_2}$ is the same as $\mathcal{L}^1_{\Omega_1}$. Regarding $\mathcal{L}^2_{\Omega_2}$, the trapdoors for Search, Insert, and Delete are transformed via PRF from the LSH hash values of query records, so the similarity between inserted, deleted and searched records are also known. Therefore, the query pattern $simp_{\Omega_2}(D)$ is the same as $simp_{\Omega_1}(D)$. For the access pattern $accp_{\Omega_2}(D)$, the server needs to maintain the state information δ_{v_j} for each distinct LSH hash value to enable Insert and ensure the correctness of Delete and Search. We note that revealing δ_{v_j} does not compromise on security. Newly allocated probing sequences all start from $d_{max} + 1$ and each Insert interaction is a batch update on l buckets. Thus, the server does not know which buckets are truly modified.

Most of efficient dynamic SSE schemes on exact keyword search leak the information such that: a keyword belongs to a newly added encrypted file if that keyword is searched before; a keyword belongs to a deleted encrypted file if that keyword is searched later. The former is defined as *forward leakage*, and

the latter is defined as *backward leakage* in [22]. In our scheme, the server knows that the records are similar to a newly inserted D, if those records appear in search results before, since their LSH hash values have intersections with D; the server also knows that the records are similar to a deleted D, if those records appear in search results later. Formally, we define the leakage as $\mathcal{L}^3_{\Omega_2}(D) = (addp(D), delp(D))$, where $addp(D) = (\{\forall A_i : lshV(D) \cap lshV(D_i) \neq \emptyset\}_q, i \in [1, q])$ and $delp(D) = (\{\forall A_i : lshV(D) \cap lshV(D_i) \neq \emptyset\}_q, i \in [1, q])$. Given the leakage functions, we give Theorem 2 and prove it in Appendix C to demonstrate that our dynamic scheme is secure against an adaptive adversary.

7 Implementation and Evaluation

Implementation: Most of SSE schemes do not specifically address the cost for building the encrypted index when the size of dataset goes large. Such cost could be prohibitively expensive for the client with limited resources. To address this issue, we carefully select system parameters and optimize the implementation for Build. Given a dataset with n records, the index load factor τ should be pre-set to determine the index capacity N. We set τ as 90 % based on empirical experience of cuckoo hashing based designs [10] to build a very compact index. Then we create l arrays with continuous addresses as the underlying structure of hash tables, where l is the number of composite LSH functions, trained via E2LSH package[3] on a sample dataset. The array capacity w is set by $\lceil \frac{n}{\tau l} \rceil$.

We allocate a shared memory with total $|A| * N$ bits, excluding the check tag or the state information, so as to increase the capacity of index held in client's memory. We encrypt each bucket in memory, and dump it to the hard disk simultaneously or send it directly to the server as long as it is encrypted. Such method will avoid memory overflow at the clients with restricted physical memory. Meantime, we carefully set the *cuckoo-kick* threshold α and the initial *random probing* step d. If α is set as a large number, the chance of rehash can be reduced. As a trade-off, the insertion will take more time due to the expensive *cuckoo-kick*. Here, we pursue efficiency and set $\alpha = 50$ instead of hundreds. We note that the value of d also has a trade-off on building efficiency and querying efficiency. If we set a large d, our index can have a large number of bucket choices to handle LSH collisions so as to reduce the iterations of *cuckoo-kick*, but search will process a large number of buckets and thus increase the latency.

Experiment Setup: we implement the main scheme and the dynamic scheme in *Java* to demonstrate the feasibility of our design at a large scale. We evaluate the performance and the search accuracy on a dataset with 10 million high-dimensional records. For cryptographic primitives, we use OpenSSL toolkit (version 1.0.1*h*) to implement the symmetric encryption via *AES-128*, and pseudo-random function (PRF) via *HMAC-SHA1*. Our source code is available at Git[4]. To demonstrate the practicality, we deploy our implementation on a

[3] E2LSH package: online at http://web.mit.edu/andoni/www/LSH.
[4] SimSSE: on line available at https://github.com/harrycui/SimSSE.

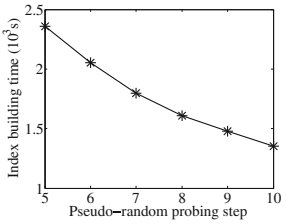

Fig. 3. Build time

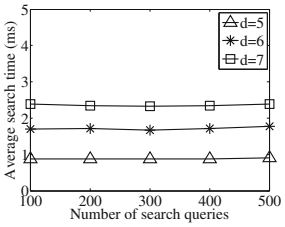

Fig. 4. Search time in Ω_1

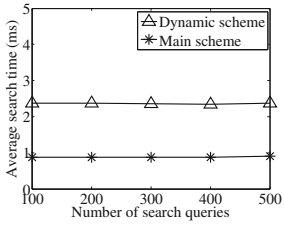

Fig. 5. Search comparison

AWS EC2 large instance "r3.4xlarge". We generate 10 million records from a public dataset on AWS, "Common Crawl Corpus", which contains over billions of web pages. Because it is stored on Amazon S3, we directly launch map-reduce jobs on AWS Elastic MapReduce (EMR) to process these web pages on a cluster of EC2 instances. For each web page, we generate a 10,000-dimensional Bag-of-Words (BoW) vector according to a dictionary with 10,000 top frequent words, where such BoW model is commonly used in methods of web page clustering and similar web page detection. Here, we apply Euclidean distance as the distance metric and use the E2LSH package to train the parameters l and m defined in Appendix A. For training, 10 % vectors are randomly selected and the distance threshold r is set to 0.5. Accordingly, tunable LSH parameters l and m are derived as 20 and 8 respectively.

Performance Evaluation: We evaluate our proposed schemes on index building cost, index space consumption, bandwidth cost, search and dynamic operation performance, and search accuracy. Figure 3 reports the index building time. For a fixed number of hash tables l, if the random probing sequence d is small, few buckets can be used to resolve the imbalanced LSH collisions. Thus, more iterations of *cuckoo-kick* will be required. In fact, the building cost is proportional to the iterations of *cuckoo-kick*. As shown, increasing d will reduce the iterations of *cuckoo-kick*, shortening the overall time, but it will introduce more bandwidth cost because the number of retrieved records is related to d for fixed l. Although the building time is not moderate, over 2,000 s in Fig. 1, it is a one-time cost, and we will improve it via concurrent programming in future.

Because we encrypt and dump the bucket simultaneously, the client only needs to allocate 4 bytes, the length of identifier, for each bucket in memory. In Table 1, for a dataset with 1 billion records, client only needs to allocate 4.4GB for our index with a load factor 90 %. In our main scheme, an encrypted bucket is in the format as: $A||0^{|a|} \oplus r$. The mask r is an output of *HMAC-SHA1*. Thus, each encrypted bucket is 20 bytes long. In our dynamic scheme, an encrypted bucket is in the format as: $A||v_j \oplus P(K_5, s), Enc(K_4, s)$, where the mask $P(K_5, s)$ is also 20 bytes long, and $Enc(K_4, s)$ is 16 bytes long with *AES-128*. We show the space consumption for different scales of datasets in Table 1. For datasets with 4 billions of records, our index consumes 160GB memory, which can fit into the main memory (244GB) of the largest Amazon EC2 instance. As mentioned, directly applying the implementations of prior SSE will consume much more

Table 1. Index space (GB) with a load factor $\tau = 90\,\%$.

Schemes	$n = 10^7$	$n = 10^8$	$n = 10^9$	$n = 4 * 10^9$
Ω_1, Ω_2 at client	0.04	0.4	4.4	17.8
Ω_1 at server	0.22	2.2	22.2	88.9
Ω_2 at server	0.40	4.0	40.0	160.0

space due to inherent issues of LSH. The comparison of space complexity is shown in Table 2 in Appendix E. Those indices will contain ln key-identifier pairs at least, where l is a tunable LSH parameter based on a predetermined distance threshold and selected training datasets, usually at scale of tens or hundreds. Such constant factor could result in an excessively huge index, e.g., more than 1000GB for $l = 20$ on 1 billion records. Our design is highly space efficient. The size of proposed index only scales with the size of dataset.

The bandwidth evaluation for Search is shown in Table 3 in Appendix D. For comparison, we build an inverted index with same LSH parameters. Then we randomly sample hundreds of search queries and calculate the average number of returned identifiers for each. In Table 3-(a), the statistics of LSH imbalance are reported for our selected 10 million records. We can see that the largest number of matched records is nearly 20 thousand. And most of LSH hash values have hundreds of matched records. In Table 3-(b), the size of results from the inverted index is over thousands which is huge and not scalable. In our design, Search costs a small sized bandwidth. For $l = 20$ and $d = 5$, the number of retrieved encrypted records is 100 at most. From Table 3-(b), comparing with an inverted index, our Search saves dozens of times of bandwidth cost. Because cuckoo hashing utilizes weak hash functions, records with different LSH hash values might be grouped. Recall that a specialized random mask and a check tag are used to enable the server to get the correct search results. Therefore, the number of result identifiers is less than 100 in Table 3-(b). From the parameter setting, 20 trapdoors (20 bytes for each) and 100 encrypted records (40 KB for each record) totally cost approximate 4 MB.

As proposed, Search only accesses a constant number of encrypted buckets, so our design can scale well when dealing with a "big query". Figure 4 reports Search performance of the main scheme. We randomly sample different numbers of queries and evaluate the average time for different probing step d. When the number of accessed buckets is equal to 100, it takes less than 1ms. Figure 5 compares Search performance of the main scheme and the dynamic scheme. Because Search in the dynamic scheme asks the client to perform the decryption of random masks, it needs more time than computing random masks directly at the cloud in the main scheme. Therefore, it is slower, but it still achieves millisecond latency, less than 3ms to process 100 encrypted buckets.

Multiple choice hashing is designed to enable lookup in each hash table concurrently. To measure the concurrency, we implement Search of our main scheme

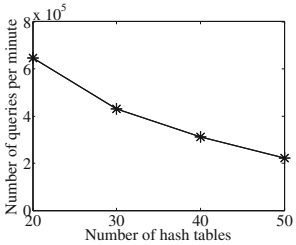

Fig. 6. Throughput

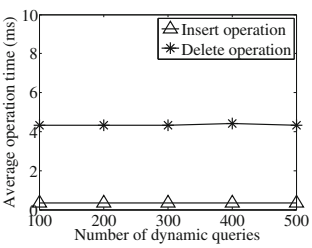

Fig. 7. Insert and Delete

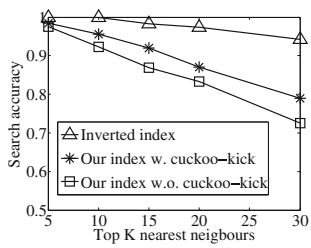

Fig. 8. Search accuracy

in parallel threads. In particular, we create 16 threads simultaneously and measure the throughput for various number of hash tables in Fig. 6. It shows that our implementation can handle over 60 thousands of queries per minute. When the number of hash table increases, the throughputs will decrease. The reason is that accessing too many hash tables degrades the search performance. Therefore, the parameter l should be kept relatively small for high concurrency.

The performance of Insert and Delete is shown in Fig. 7. The total time consists of the time for locating encrypted buckets at the server, and the time for decrypting and re-encrypting them at the client. For Insert, we conduct hundreds of Insert queries on the index with a load factor 90 %. Because it adopts *adaptive probing* to find available buckets by open addressing rather than expensive *cuckoo-kick*. The results show that even the index load is heavy, Insert can be fast. Insert succeeds in one interaction by only accessing 20 buckets for $l = 20$ and the average time is less than 1ms. Delete has to retrieve, decrypt, and re-encrypt all related $\sum_{j=1}^{l} \delta_{v_j}$ buckets. Therefore, the time is much longer. We do not perform Insert before Delete, so δ_{v_j} can be treated as $d_{max}+1$, which is equal to 12 for this dataset. As shown in Fig. 7, the average time for Delete is around 4ms.

The search accuracy is measured based on the definition of $\frac{1}{K}\sum_{i=1}^{K}\frac{||D'_i - D_q||}{||D_i - D_q||}$ in [23], where D_q is the query record and D_i is the i-th closest record to D_q. This metric reflects a general quality of Top-K neighbors. It quantifies the closeness between the Euclidean distances of Top-K records from LSH-based indices and the ground truth nearest records via linear scan. For comparison, we also compute the accuracy of Top-K results from an inverted index. Figure 8 shows that the results of inverted index are closer to the ground truth. We note that our design introduces a little loss in accuracy, because our index employs an approximation such that only one copy of each record identifier is stored and the search results do not include all the matched records. But we can see that *cuckoo-kick* improves accuracy a bit. The reason is that even if one of two similar records is kicked, it is probably still moved back to one of previous corresponding buckets. On the contrary, the less similar one might be kicked out since they have fewer matched LSH hash values. As a result, our design still achieves acceptable accuracy and saves dozens of times on the index space consumption, and the query latency and bandwidth.

8 Conclusion

We investigated secure and fast similarity search over large encrypted datasets. Our design starts from two building blocks, LSH and SSE. As we target for the high performance index design, we have explored practical hashing schemes, including multiple choice hashing, open addressing, and cuckoo hashing, to achieve a construction with superior space efficiency and low query latency. Adapting from the security framework of SSE, we carefully capture the information leakage and prove the security against adaptive chosen keyword attacks. We have implemented our schemes over 10 million encrypted high-dimensional data records at Amazon AWS. The experimental results are indeed promising.

Acknowledgment. This work was supported in part by Research Grants Council of Hong Kong (Project No. CityU 138513), grant from City University of Hong Kong (Project No. 7004279), and an AWS in Education Research Grant award.

A Definition of Locality-Sensitive Hashing

Definition 2 (LSH Family $\mathcal{H}$). *Given the distance r, cr, where $c > 1$, and the probability value p_1, p_2, where $p_1 > p_2$, a function family $\mathcal{H}$ is (r, cr, p_1, p_2)-sensitive if for any points $D, D' \in \mathcal{R}^d$ and any $h \in \mathcal{H}$: if distance $dist(D, D') \leq r$, $P[h(D) = h(D')] \geq p_1$; if distance $dist(D, D') > cr$, $P[h(D) = h(D')] \leq p_2$;*

In practice, the composite LSH function $\{g_1, \cdots, g_l\}$ is applied to enlarge the gap between p_1 and p_2. One explicit composite function g_i contains m independent LSH functions, which are randomly selected from $\mathcal{H}$: $g_i = (h_1, \ldots, h_m)$. As a result, for any $D, D' \in \mathcal{R}^d$: if $dist(D, D') \leq r$, $P[\exists i \in [1, l] : g_i(D) = g_i(D')] \geq 1 - (1 - p_1^m)^l$; $dist(D, D') > cr$, $P[\exists i \in [1, l] : g_i(D) = g_i(D')] \leq 1 - (1 - p_2^m)^l$.

B Simulation-Based Security Definition

Definition 3. *Let $\Omega =$ (GenKey, Build, GenTpdr, Search, Insert, Delete) be our scheme for secure similarity search, and let $\mathcal{L}_\Omega^1$, $\mathcal{L}_\Omega^2$, and $\mathcal{L}_\Omega^3$ be the stateful leakage function. Given an adversary $\mathcal{A}$ and a simulator $\mathcal{S}$, define the following probabilistic games $\mathbf{Real}_\mathcal{A}(k)$ and $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(k)$:*

$\mathbf{Real}_\mathcal{A}(k)$*: a challenger calls* $\mathsf{GenKey}(1^k)$ *to output a key K. $\mathcal{A}$ selects $\mathbf{D}$ and asks the challenger to build $\mathcal{I}$ via* Build*. Then $\mathcal{A}$ adaptively performs a polynomial number of* Search, Insert *or* Delete *queries, and asks for the trapdoor t of each query q from the challenger. Finally, $\mathcal{A}$ returns a bit as the game's output.*

$\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(k)$*: $\mathcal{A}$ selects $\mathbf{D}$, and $\mathcal{S}$ generates $\widetilde{\mathcal{I}}$ based on $\mathcal{L}_\Omega^1(\mathbf{D})$. Then $\mathcal{A}$ adaptively performs a polynomial number of queries. From $\mathcal{L}_\Omega^2(D)$ and $\mathcal{L}_\Omega^3(D)$ of each query q, $\mathcal{S}$ returns the ciphertext and generates the corresponding $\widetilde{t}$. Finally, $\mathcal{A}$ returns a bit as the game's output.*

Our proposed scheme Ω is $(\mathcal{L}^1_\Omega, \mathcal{L}^2_\Omega, \mathcal{L}^3_\Omega)$-secure against adaptive chosen-keyword attacks if for all probabilistic polynomial time adversaries $\mathcal{A}$, there exists a probabilistic polynomial time simulator $\mathcal{S}$ such that

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

where $\epsilon(k)$ is a negligible function in k.

C Security Proofs

Theorem 1. *Ω_1 is $(\mathcal{L}^1_{\Omega_1}, \mathcal{L}^2_{\Omega_1})$-secure against adaptive chosen-keyword attacks in the random oracle model if* SE *is CPA-secure, and F, P, G are PRF.*

Proof. We will demonstrate that, based on $\mathcal{L}^1_{\Omega_1}$, $\mathcal{S}$ can first simulate an index $\widetilde{\mathcal{I}}$, which is indistinguishable from $\mathcal{I}$. Then $\mathcal{S}$ can simulate result identifiers $\{\{\widetilde{A}\}_m\}_q$ and trapdoors $\{\widetilde{t}\}_q$ based on $\mathcal{L}^2_{\Omega_1}$ for q adaptive queries, which are also indistinguishable from $\{\{A\}_m\}_q$ and $\{t\}_q$. To achieve the indistinguishability, $\{\widetilde{A}\}_m$ should be correctly recovered via $\widetilde{t}$ from $\widetilde{\mathcal{I}}$ for all q queries. It means $\{\widetilde{t}\}_q$ should also be consistent with each other, which implicitly asks $\mathcal{S}$ to trace the dependencies between each query. The simulation is presented as follows:

- Simulate $\widetilde{\mathcal{I}}$: given $\mathcal{L}^1_{\Omega_1}(\mathbf{D}) = (N, l, |B^*|)$, $\mathcal{S}$ initializes an empty index $\widetilde{\mathcal{I}}$ with l hash tables and total N capacity, which are exactly the same as $\mathcal{I}$. After that, $\mathcal{S}$ generates $\widetilde{K_B}$ via $\mathsf{Gen}(1^k)$. Each bucket in $\widetilde{\mathcal{I}}$ is filled with $\mathsf{Enc}(\widetilde{K_B}, 0^{|B^*|})$, where $|B^*|$ is the bit length of bucket in $\mathcal{I}$.
- Simulate the first Search query: given $accp_{\Omega_1}(D)$ from $\mathcal{L}^2_{\Omega_1}(D)$, $\mathcal{S}$ outputs $\{\widetilde{A}\}_m$ which is identical to $\{A\}_m$, and then generates the trapdoor $\widetilde{t} = (\{\widetilde{K^1_j}\}_l, \{\widetilde{K^2_j}\}_l)$, where $\widetilde{K^1_j}$ and $\widetilde{K^2_j}$ are random strings with equal length of $K^1_{v_j}$ and $K^2_{v_j}$. By operating random oracles, $\mathcal{S}$ can use $\widetilde{t}$ to recover $\{A\}_m$. In particular, PRF G and F are replaced by two random oracles H_1 and H_2. We note that $\mathcal{L}^2_{\Omega_1}$ also tells $\mathcal{S}$ where the identifiers are stored and how many identifiers are correctly recovered at each table. Thus, on the input $\widetilde{t}$ and $\{\widetilde{A}\}_m$, $\mathcal{S}$ can locate the buckets $\{\widetilde{B^*}\}_{ld_{max}}$ with identical locations of $\widetilde{B^*}_{ld_{max}}$ via $\{H_1(\widetilde{K^1_j}||d)\}_{ld_{max}}$, where $d \in [1, d_{max}]$, and outputs $\{H_2(\widetilde{K^2_j}||b)\}_{ld_{max}}$ such that $H_2(\widetilde{K^2_j}||b) \oplus \widetilde{B^*} = A$ for buckets that can be correctly decrypted.
- Simulate subsequent Search queries: given $simp_{\Omega_1}(D)$ from $\mathcal{L}^2_{\Omega_1}(D)$, $\mathcal{S}$ can know whether there are similar query records that appear before or not. If $\{\theta\}_q$ are all 0, which means D is a record which is distant to others, $\mathcal{S}$ follows

the same way for the first query to simulate trapdoors and search results. As long as there exists θ which is larger than 0, $\mathcal{S}$ uses the same random strings $\widetilde{K_j^1}$ and $\widetilde{K_j^2}$ of D_i based on the intersection, and accesses the same buckets in $\widetilde{T_j}$. Meanwhile, $\mathcal{S}$ generates fresh random strings for sub trapdoors which did not appear before, and outputs the accessed buckets and the result identifiers from $\mathcal{L}^2_{\Omega_1}$ as described above.

We emphasize that $\mathcal{I}$ and $\widetilde{\mathcal{I}}$ have an identical structure with N buckets and l hash tables. The bucket $\mathcal{B}^*$ and $\widetilde{\mathcal{B}^*}$ are filled by ciphertext with equal length. Thus, $\mathcal{I}$ and $\widetilde{\mathcal{I}}$ are indistinguishable. For a given query, the result identifiers $\{A\}_m$ and $\{\widetilde{A}\}_m$ are identical, and the accessed buckets $\{\widetilde{B^*}\}_{ld_{max}}$ locates identically as $\{B^*\}_{ld_{max}}$. Due to the pseudo-randomness of F, P and G, the trapdoors t and $\widetilde{t}$ are indistinguishable. Meanwhile, the simulated $\{\widetilde{t}\}_q$ are consistent with each other, and the intersections among $\{\widetilde{t}\}_q$ are identical to the intersections $\{t\}_q$. Therefore, the outputs of $\mathbf{Real}_{\mathcal{A}}(k)$ and $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(k)$ are computationally indistinguishable.

Theorem 2. *Ω_2 is ($\mathcal{L}^1_{\Omega_2}$, $\mathcal{L}^2_{\Omega_2}$, $\mathcal{L}^3_{\Omega_2}$)-secure against adaptive chosen-keyword attacks in random oracle model if* SE *is CPA-secure, and P, G are PRF.*

Proof. As stated in the proof of Theorem 1, simulator $\mathcal{S}$ can simulate an indistinguishable index $\widetilde{\mathcal{I}}$ from $\mathcal{L}^1_{\Omega_2}$. It can also generate consistent trapdoors, the access pattern and the query pattern from $\mathcal{L}^2_{\Omega_2}$, which are indistinguishable from real ones. For Search, Insert and Delete, $\mathcal{S}$ returns $\{\widetilde{B^*}\}_{\sum_{j=1}^{l}\delta_{v_j}}$ with identical locations of real buckets via operating random oracle $H_1(\widetilde{K_j^1}, d_j)$, where $d_j \in [1, \delta_{v_j}]$. Meantime, $\mathcal{S}$ can operate random oracle $H_2(\widetilde{K_j^2}, \widetilde{s})$ to recover the result identifier $A = H_2(\widetilde{K_j^2}, \widetilde{s}) \oplus B^*$, and update the encrypted bucket $B^* = H_2(\widetilde{K_j^2}, \widetilde{s}) \oplus A$ or $B^* = H_2(\widetilde{K_j^2}, \widetilde{s}) \oplus \perp$ so that the subsequent search results will be consistent. Note that for Insert and Delete, the buckets accessed by the server will be re-encrypted. $\mathcal{S}$ can generate new buckets via $\mathsf{Enc}(\widetilde{K_B}, 0^{|B^*|})$. From $\mathcal{L}^3_{\Omega_2}$, $addp(D)$ and $delp(D)$ show the identifiers in some of those updated buckets if they are searched either before or after, so $\mathcal{S}$ can generate the consistent trapdoors and masks via H_2 on the input of B^*, A and $\perp$. Due to the CPA-secure of SE and the pseudo-randomness of P and G, the adversary $\mathcal{A}$ cannot differentiate $\widetilde{\mathcal{I}}$ and $\mathcal{I}$, $\widetilde{t}$ and t, and $\widetilde{B^*}$ and B^* respectively. Therefore, the outputs of $\mathbf{Real}_{\mathcal{A}}(k)$ and $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(k)$ are indistinguishable (Table 2).

D Comparison with Prior Work

Table 2. We compare existing SSE schemes with our schemes by treating LSH hash values as keywords. $\#w$ is the number of keywords, $\#id_w$ is the number of matched identifiers for a given keyword w, M_w is the maximum number of matched identifiers over all the keywords, n is the number of records, c is the retrieval constant, ($c \ll \#id_w$ shown in our experiment), p is the number of used processors, and l is the number of composite LSH functions.

Scheme	Index size	Search time	Security	Index leak
CGKO'06 [6]	$O(\sum_w \#id_w + \#w)$	$O(\#id_w)$	NonAd	#w
CK'10 [5]	$O(\#wM_w)$	$O(\#id_w)$	Ad	#w
vLSDHJ'10 [24]	$O(\#wM_w)$	$O(\log \#w)$	Ad	#w
KPR'12 [14]	$O(\sum_w \#id_w + \#w)$	$O(\#id_w)$	Ad	#w
KIK'12 [15]	$O(ln^2)$	$O(l)$	Ad	-
KP'13 [12]	$O(\#wn)$	$O((\#id_w \log n)/p)$	Ad	#w
SPS'14 [22]	$O(\sum_w \#id_w)$	$O(\#id_w + \log \sum_w \#id_w)$	Ad	-
CJJJKRS'14 [2]	$O(\sum_w \#id_w)$	$O(\#id_w/p)$	Ad	$\sum_w \#id_w$
Our scheme	$O(n)$	$O(c/min(p, l))$	Ad	-

E Bandwidth Consumption Switch Appendix D with Appendix E

Table 3. Bandwidth evaluation.

Matched IDs	$< 1K$	$1K - 4K$	$> 4K$
$\#\ lshV$	15465K	1861	68

(a) Statistics of LSH imbalance.

#Samples	100	200	300	400	500
$\#A_{inv}$	2652	3730	4280	5285	3824
$\#A$	95	96	93	90	95
Saving	27 ×	38 ×	45×	58 ×	39 ×

(b) Bandwidth comparison and saving.

References

1. Andoni, A., Indyk, P.: Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. Commun. ACM **51**, 117–122 (2008)
2. Cash, D., Jaeger, J., Jarecki, S., Jutla, C., Krawczyk, H., Rosu, M.C., Steiner, M.: Dynamic searchable encryption in very large databases: Data structures and implementation. In: Proceedings of NDSS (2014)
3. Cash, D., Jarecki, S., Jutla, C., Krawczyk, H., Roşu, M.-C., Steiner, M.: Highly-scalable searchable symmetric encryption with support for boolean queries. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013, Part I. LNCS, vol. 8042, pp. 353–373. Springer, Heidelberg (2013)

4. Chang, Y.-C., Mitzenmacher, M.: Privacy preserving keyword searches on remote encrypted data. In: Ioannidis, J., Keromytis, A.D., Yung, M. (eds.) ACNS 2005. LNCS, vol. 3531, pp. 442–455. Springer, Heidelberg (2005)
5. Chase, M., Kamara, S.: Structured encryption and controlled disclosure. In: Abe, M. (ed.) ASIACRYPT 2010. LNCS, vol. 6477, pp. 577–594. Springer, Heidelberg (2010)
6. Curtmola, R., Garay, J., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: improved definitions and efficient constructions. In: Proceedings of ACM CCS (2006)
7. Goh, E.J.: Secure indexes. Cryptology ePrint Archive (2003)
8. Goldreich, O.: Foundations of Cryptography: Volume 2, Basic Applications, vol. 2. Cambridge University Press, New York (2009)
9. Hahn, F., Kerschbaum, F.: Searchable encryption with secure and efficient updates. In: Proceedings of ACM CCS (2014)
10. Hua, Y., Xiao, B., Liu, X.: Nest: Locality-aware approximate query service for cloud computing. In: Proceedings of IEEE INFOCOM (2013)
11. Jarecki, S., Jutla, C., Krawczyk, H., Rosu, M., Steiner, M.: Outsourced symmetric private information retrieval. In: Proceedings of ACM CCS (2013)
12. Kamara, S., Papamanthou, C.: Parallel and dynamic searchable symmetric encryption. In: Proceedings of Financial Cryptography (2013)
13. Kamara, S., Papamanthou, C., Roeder, T.: CS2: A searchable cryptographic cloud storage system. Microsoft Research, Technical report MSR-TR-2011-58 (2011)
14. Kamara, S., Papamanthou, C., Roeder, T.: Dynamic searchable symmetric encryption. In: Proceedings of ACM CCS (2012)
15. Kuzu, M., Islam, M.S., Kantarcioglu, M.: Efficient similarity search over encrypted data. In: Proceedings of IEEE ICDE (2012)
16. Lv, Q., Josephson, W., Wang, Z., Charikar, M., Li, K.: Multi-probe lsh: Efficient indexing for high-dimensional similarity search. In: Proceedings of VLDB (2007)
17. Naveed, M., Prabhakaran, M., Gunter, C.: Dynamic searchable encryption via blind storage. In: Proceedings of IEEE S&P (2014)
18. Pagh, R., Rodler, F.F.: Cuckoo hashing. J. Algorithms **51**(2), 122–144 (2004)
19. Panigrahy, R.: Entropy based nearest neighbor search in high dimensions. In: Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithm (SODA) (2006)
20. Rane, S., Boufounos, P.T.: Privacy-preserving nearest neighbor methods: comparing signals without revealing them. IEEE Sig. Process. Mag. **30**(2), 18–28 (2013)
21. Song, D., Wagner, D., Perrig, A.: Practical techniques for searches on encrypted data. In: Proceedings of IEEE S&P (2000)
22. Stefanov, E., Papamanthou, C., Shi, E.: Practical dynamic searchable symmetric encryption with small leakage. In: Proceedings of NDSS (2014)
23. Tao, Y., Yi, K., Sheng, C., Kalnis, P.: Quality and efficiency in high dimensional nearest neighbor search. In: Proceedings of ACM SIGMOD (2009)
24. van Liesdonk, P., Sedghi, S., Doumen, J., Hartel, P., Jonker, W.: Computationally efficient searchable symmetric encryption. In: Jonker, W., Petković, M. (eds.) SDM 2010. LNCS, vol. 6358, pp. 87–100. Springer, Heidelberg (2010)