Privacy-Preserving Similarity Search With Efficient Updates in Distributed Key-Value Stores

Wanyu Lin, *Member, IEEE*, Helei Cui, *Member, IEEE*, Baochun Li, *Fellow, IEEE*, and Cong Wang, *Fellow, IEEE*

Abstract—Privacy-preserving similarity search plays an essential role in data analytics, especially when very large encrypted datasets are stored in the cloud. Existing mechanisms on privacy-preserving similarity search were not able to support secure updates (addition and deletion) efficiently when frequent updates are needed. In this article, we propose a new mechanism to support parallel privacy-preserving similarity search in a distributed key-value store in the cloud, with a focus on efficient addition and deletion operations, both executed with sublinear time complexity. If search accuracy is the top priority, we further leverage Yao's garbled circuits and the homomorphic property of Hash-ElGamal encryption to build a secure evaluation protocol, which can obtain the top-R most accurate results without extensive client-side post-processing. We have formally analyzed the security strength of our proposed approach, and performed an extensive array of experiments to show its superior performance as compared to existing mechanisms in the literature. In particular, we evaluate the performance of our proposed protocol with respect to the time it takes to build the index and perform similarity queries. Extensive experimental results demonstrated that our protocol can speedup the index building process by up to 800$\times$ with 2 threads and the similarity queries by up to $\sim$7$\times$ with comparable accuracy, as compared to the state-of-the-art in the literature.

Index Terms—Searchable symmetric encryption, key-value stores, data privacy, similarity search, cloud storage, efficient updates

1 Introduction

With large and rapidly growing volumes of data to be managed, it is customary to outsource the storage of such data to the cloud. Yet, storing data in the cloud leads to legitimate and serious concerns about the privacy of such data. To mitigate these concerns, searchable symmetric encryption (SSE) has been introduced to provide a rich set of search operations on encrypted data (e.g., [1], [2], [3]).

Real-world systems that implemented SSE to support secure search operations, such as CryptDB [4] and Seabed [5], have all been designed as centralized databases. In contrast, distributed key-value stores, such as Redis and MongoDB, have emerged as prevalent data storage solutions in public cloud storage platforms, due to their scalability, performance, and strong support for multiple data models. As a result, encrypted systems based on distributed key-value stores have been proposed [6], [7] to inherit the salient advantages in modern distributed key-value stores, such as high performance and excellent scalability.

Given the widespread use of distributed key-value stores in the cloud, it is important to support *similarity search*, which has become indispensable in a wide range of applications such as cloud image sharing [8], and real-time decisions [9]. EncSIM [7], a state-of-the-art mechanism to support privacy-preserving similarity search over distributed key-value stores in the literature, only supported efficient search operations, but failed to support efficient update operations (adding and deleting data), which are naturally needed for applications with frequent updates. For example, in the application of cloud-assisted image sharing for mobile devices, data generated from mobile devices are updated frequently. Therefore, efficient updates are essential and highly desirable in secure distributed key-value stores. In addition, if search accuracy is the top priority, EncSIM would have to perform decryption and ranking at the client side, which incurred unnecessary client-side computation.

In this paper, we propose a new privacy-preserving similarity search mechanism focusing on its practicality, especially in the case where frequent updates (addition and deletion) occur in distributed key-value stores. Specifically, our objectives are to address the following two challenges. First, appropriate cryptographic primitives for similarity search need to be carefully chosen to balance security and practicality. The selective primitives should not diminish the advantages of distributed key-value stores and provably secure with guaranteed strength. Second, the leakage should be minimized while performing updates.

Essentially, the core of privacy-preserving similarity search relies on building an inverted index, combining standard locality sensitive hashing (LSH) [9] and standard cryptographic primitives, including Pseudorandom Functions (PRFs) and Symmetric-Key Block Ciphers. Solely based on such an inverted index, while deletion queries occur, [1], [7] deployed a revocation list to support secure deletion, and do not reclaim space in the server after each deletion.

- *Wanyu Lin and Baochun Li are with the Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON M5S 1A1, Canada. E-mail: wanyu.lin@mail.utoronto.ca, bli@ece.toronto.edu.*
- *Helei Cui is with the School of Computer Science, Northwestern Polytechnical University, Xi'an, Shaanxi 710129, China. E-mail: chl@nwpu.edu.cn.*
- *Cong Wang is with the Department of Computer Science, City University of Hong Kong, Hong Kong, China. E-mail: congwang@cityu.edu.hk.*

Manuscript received 2 July 2020; revised 25 Oct. 2020; accepted 1 Dec. 2020. Date of publication 9 Dec. 2020; date of current version 15 Dec. 2020. (Corresponding author: Wanyu Lin.) Recommended for acceptance by W. Yu Ph.D. Digital Object Identifier no. 10.1109/TPDS.2020.3042695

To support actual data deletion in a privacy-preserving manner, the core of our mechanism relies on a *dual index*, which uses a linked index to represent both the inverted index and the forward index. In particular, to ensure sublinear similarity search over encrypted high-dimensional datasets, our proposed mechanism first builds the inverted index, combining a standard locality sensitive hashing (LSH) [9] and SSE. It then applies SSE to develop the corresponding secure forward index, deriving from the associated data identifiers, in order to support data deletion.

While studying the problem of dynamic SSE schemes, it is crucial to support *forward security*. Zhang *et al.* [10] highlighted that file-injection attacks become particularly effective when the dynamic SSE is not forward-secure. Forward security has been extensively studied in the literature [11], [12], [13], [14], and required that newly added records cannot be related to previous queries. Inspired by [12], the index associated with previous queries would be established with fresh keys after each search is performed in our mechanism.

In order to maintain a high degree of parallelism, our dual-index is designed specifically for secure and distributed key-value stores with high-dimensional datasets. However, due to the probabilistic nature of LSH, the effectiveness of similarity search relies heavily upon the performance of LSH, which introduces false positives. In order to further reduce such false positives, and just return the top-R accurate results in the encrypted domain, we propose to further apply a secure two-party computation protocol based on Yao's garbled circuits [15] and a homomorphic encryption scheme, Hash-ElGamal [16], to evaluate the query candidates. Our protocol is able to avoid expensive decryption during circuit evaluation [17], [18], which leads to a significant reduction of the overall query time. To further optimize the overall query efficiency, we perform the evaluation process in parallel across a cluster of storage nodes.

In a nutshell, our objectives are to guarantee strong data confidentiality and to support similarity search for forward-secure addition and efficient deletion, while maintaining a high degree of parallelism for the best possible performance. Highlights of our original contributions are as follows. *First*, we propose a series of protocols, including index construction and its corresponding addition, deletion and search operations, to support similarity search with efficient updates over encrypted data in distributed key-value stores, i.e., Redis. Specifically, the encrypted database and encrypted index are physically stored as key-value pairs in Redis, in which they are accessible by native Redis API, i.e., **Put**, **Get**, **Update**. *Second*, in order to guarantee the search accuracy and release the computation burden of the client, we introduce a secure two-party computation protocol based on Yao's garbled circuit and Hash-ElGamal to efficiently compute the distance of the candidates and the query in the cloud. As such, we can obtain the top-R accurate results before returning these results to the client. We have formally analyzed the security strength of our mechanism. *Finally*, to evaluate our proposed mechanism experimentally, we have implemented it in our real-world prototype implementation and conducted an extensive array of performance evaluations using real-world datasets.

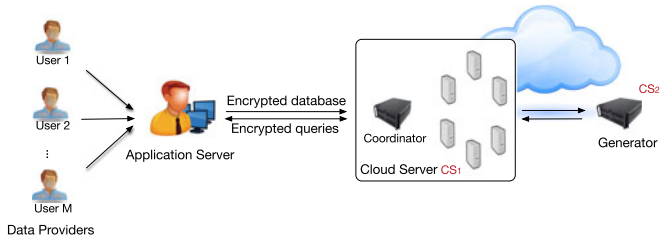

Fig. 1. The system architecture of our mechanism.

2 Problem Formulation

2.1 System Architecture

In this paper, we consider a typical scenario of storing data in the cloud, as shown in Fig. 1, where a group of users, called data providers (DPs), upload their data to the public cloud via a trusted local server, called an *application server* (AS). Such a trustworthy application server could be deployed in the same organization as the data providers. In addition, we introduce two types of cloud servers, CS_1 and CS_2, to jointly perform secure and accurate data operations on the encrypted dataset.

More precisely, each data provider i has a small subset of the data collection that can be represented as high-dimensional data records $\{S_i\}$, where data can be images, videos and websites. Take an image dataset as an example. Each data record is a fingerprint of an image, which is a 64-bit array in our experiments. Each data provider encrypts its dataset $\{S_i\}$ using a public key issued by the application server—$\{S_i^*\}$, and sends both the plain and encrypted records to the application server. The application server collects the data providers' data, and globally distributes them within CS_1 in an encrypted form $\mathbf{S}^* = \{s_1^*, s_2^*, \ldots, s_n^*\}$.[1] It is capable of issuing similarity search, addition and deletion queries on behalf of the users. CS_1 consists of a coordinator and a cluster of storage nodes. It is delegated by the AS to execute specific protocols over encrypted data. CS_2, on the other hand, initializes the evaluation parameters according to the keys pre-issued by the application server.

Our system for effective and privacy-preserving similarity search with efficient updates consists of five polynomial-time protocols *SimSE = (Setup, Addition, Deletion, Search, Evaluation)*. The service flow of our system is divided into the following four phases:

Preparation. The data provider i first encrypts its data records using the public key of Hash-ElGamal pre-issued by the application server. After that, each data provider sends its data, $\{S_i^*, S_i\}$, to the application server, where $\{S_i\}$ is used for subsequent construction of the secure index at the application server, while $\{S_i^*\}$ is for subsequent data partition and evaluation within the cloud servers. Such a data pre-encryption process in each data provider can effectively balance the computation overhead of the application server, especially for applications with massive amounts of data.

Setup. Once the application server collects the data from all of the data providers, it enters this phase to build a

1. To be clear, $\{S_i\}$ is a subset of data records from data provider i, and s_j or s_j^* represents a particular data record or an encrypted data record.

secure, searchable index **I** that supports efficient search operations with sublinear complexity in the distributed key-value store. Our *Setup* protocol partitions the encrypted database (EDB) according to the encrypted data records and builds an encrypted searchable index for each partition separately, as performed in [6], [7]. The EDB, containing the encrypted index **I**, the encrypted data and the encrypted high-dimensional representations $\mathbf{S}^*$, is then deployed evenly across the cluster of nodes. For subsequent evaluations of secure candidates, the application server is delegated to share a pair of Hash-ElGamal keys with CS_2.

Search. In this phase, the application server generates an encrypted query token **t** by executing the *SearchToken* protocol and sends it to the coordinator. The coordinator then broadcasts the tokens to the cluster of nodes in CS_1. Upon receiving the token, the nodes are responsible for processing the encrypted index by running the *SimSearch* protocol in parallel. To eliminate false positives from the initial query candidates, CS_2 prepares garbled circuits for each node, which securely evaluates whether the distance of the encrypted candidates and the query is within a pre-defined threshold by running the *Evaluation* protocol in parallel. One may consider applying full homomorphic encryption (FHE) instead, but it may be too computationally expensive to be practical [19].

Updates. In this phase, if data is to be added, the application server first generates indexes for the newly added records by running the *Addition* protocol. By locating the storage node via consistent hashing, the coordinator dispatches the indexes and encrypted records to that particular node. CS_1 outputs an updated EDB with the newly added records and indexes. If data is to be deleted, the application server generates the deletion token and sends it to CS_1. By executing the *Deletion* protocol, CS_1 outputs an updated EDB with that particular data record deleted.

2.2 Threat Assumptions

We assume that the cloud servers, CS_1 and CS_2, are "semi-honest" and can be seen as adversaries [20], [21]. CS_1 maintains the storage nodes, executes the data operation protocols (*Search, Addition, Deletion*), and evaluates the query results with the help of CS_2 as required. We assume that CS_1 and CS_2 do not collude with each other. Precisely, CS_2 does not expose its Hash-ElGamal secret key to CS_1. As CS_1 can access the memory and disks of all the storage nodes [6], it is "curious" and may be able to infer and analyze the EDB; it can monitor the query protocols as well. As such, it may learn additional information about the EDB [20], [22], the query tokens, or access the index entries and query results. The data application is considered trustworthy, building its secure index properly and issuing queries for the authorized users. We do not consider the case where the cloud servers would corrupt the queries and the evaluation process.

As a convenience, we summarize the notations to be used throughout this paper in Table 1.

TABLE 1
Notation

Symbol	Meaning
$\mathcal{K}$	security parameter
lsh	LSH parameters
$g_l, l \in [1, lshL]$	$lshL$ LSH hash functions
(pk, sk)	Hash-ElGamal public/ secret keys
$\mathbf{HE}(Enc_{pk}, Dec_{sk})$	Hash-ElGamal encryption scheme
$\mathbf{SE}(Enc, Dec)$	SSE scheme
F, F_1, F_2, F_3, F_4	Pseudo-random functions
$\mathbf{S} = \{s_1, s_2, \ldots, s_n\}$	high-dimensional data records collection
$\mathbf{S}^* = \{s_1^*, s_2^*, \ldots, s_n^*\}$	encrypted data records collection
$\mathbf{id} = \{id_1, id_2, \ldots, id_n\}$	records' identifiers (physical address)
CS_1	the primary cloud server for DBMS
N	number of storage nodes in S_1
CS_2	garbled circuit generator
t	a search token for a query
I	search index
s_q	similarity query
δ	pre-defined distance threshold
$\hat{\delta}$	the garbled input of threshold
$\hat{s^*}$	the garbled input of s^*

2.3 Preliminaries

Locality Sensitive Hashing (LSH). LSH provides a method to solve the problem of fast approximate nearest neighbour search in a high-dimensional space [9]. Basically, it hashes input data records to a universe $\mathcal{U}$, so that similar data records are mapped into the same buckets with higher probability than those that are far apart. An LSH family H is defined for a metric space $\mathcal{M} = (C, \mathcal{D})$. This LSH family is (r_1, r_2, p_1, p_2)-locality-sensitive if any two data records $s_1, s_2 \in C$ satisfy:

if $\mathcal{D}(s_1, s_2) \leq r_1$, then $P_r[g(s_1) = g(s_2)] \geq p_1$; if $\mathcal{D}(s_1, s_2) \geq r_2$, then $P_r[g(s_1) = g(s_2)] \leq p_2$,

where C is the domain of the data records, and $\mathcal{D}(s_1, s_2)$ is the distance between the data records s_1 and s_2.

Searchable Symmetric Encryption (SSE). A symmetric encryption scheme *SE =(KeyGen, Enc, Dec)* consists of three algorithms: the key generation algorithm *KeyGen* that takes a security parameter $\mathcal{K}$ as an input to return the secret keys; the encryption algorithm *Enc* that takes a key K_2 and a data identifier id as inputs to return a ciphertext id^*; and the decryption algorithm *Dec* that takes K_2 and id^* as input to return id if K_2 is exactly the secret key to encrypt id.

Pseudo-Random Function. A family of pseudo-random functions (PRF) is defined as $G : \mathcal{K} \times X \rightarrow R$, if for all probabilistic polynomial time, adversary $\mathcal{A}$, $|\mathbf{Pr}[\mathcal{A}^{G(K,\cdot)} = 1 | K \leftarrow \{0,1\}^{\mathcal{K}}] - \mathbf{Pr}[\mathcal{A}^{g} = 1 | g \leftarrow \{\mathbf{Func} : X \rightarrow R\}]| < negl(\mathcal{K})$, where $negl(\mathcal{K})$ is a negligible function in $\mathcal{K}$.

Hash-ElGamal Encryption (HE). Hash-ElGamal Encryption is a semantically secure asymmetric encryption scheme, under the Decisional Composite Residuosity assumption [16]. Let $\mathbf{Enc}_{pk}$ and $\mathbf{Dec}_{sk}$ be the HE encryption and decryption functions with the keys pk and sk, respectively. A Hash-ElGamal cryptosystem has the following homomorphic property: the ciphertext of a given message m, under HE, is $c = \langle u, v \rangle$. With the ciphertext, one can add a random mask μ by computing $c^{\mu} = \langle u, v \oplus \mu \rangle$. Then the one with sk can derive $m \oplus \mu$ based on c^{μ}.

Paillier Cryptosystem The Paillier cryptosystem [23] is a semantically secure public key encryption scheme based on the Decisional Composite Residuosity assumption. Let $\mathbf{Enc}_{pk}$ and $\mathbf{Dec}_{sk}$ be the Paillier encryption and decryption functions with the keys pk and sk, respectively. We use $[m]$ to denote an encryption of a message m under Paillier cryptosystem, that is, $[m] = \mathbf{Enc}_{pk}(m)$.

Paillier cryptosystem is homomorphically additive. Given the ciphers $[u]$ and $[v]$, there exists an efficient algorithm to compute the encryption of $u+v$. The additive property satisfies $[u] \oplus [v] = [u+v]$.

Yao's Garbled Circuit (GC). The Yao's Garbled Circuit is a generic methodology for secure multi-party computation [15], with two parties holding their private inputs s_1^* and s_2^* respectively. It allows them to jointly compute an arbitrary function $\mathcal{D}(s_1^*; s_2^*)$. After computation, the two parties learn the value of $\mathcal{D}(s_1^*; s_2^*)$, but no party learns any other information beyond the output value.

The basic idea of Yao's Garbled Circuit is as follows. The first party, called the generator, builds a "garbled" circuit for computing $\mathcal{D}$ and then it sends the garbled circuit with its garbled-circuit input values $\hat{s_1^*}$ to the second party, called the evaluator. Upon receiving the circuits, the evaluator is required to obtain the garbled-circuit input value $\hat{s_2^*}$ from the generator through a 1-out-of-2 oblivious transfer protocol. Then, the evaluator can obtain the encrypted output of $\mathcal{D}$ based on the value of $\hat{s_1^*}$ and $\hat{s_2^*}$. In this paper, we will use $\hat{s^*}$ to denote the garbled-input value of s^*.

3 Our Proposed Design

In this section, we first introduce a basic construction of EncSIM [7], then describe the intuition of our design for supporting forward security with efficient updates, followed by a detail description of our proposed mechanism.

EncSIM [7] represents the state-of-the-art that supports privacy-preserving similarity search in distributed key-value stores, with the ability of scaling up to millions of data records. Its architecture inherits modern features of distributed key-value stores, including high performance, incremental scalability, and parallel processing, yet still maintaining data confidentiality.

By applying a standard partition algorithm — consistent hashing [24] — the cloud server can globally distribute an encrypted database in a cluster of storage nodes. Due to its salient features, we follow this partition algorithm to support similarity search in parallel, with the exception that high-dimensional data records are encrypted using Hash-ElGamal encryption by each data provider. With this particular encryption, we guarantee that the candidates' evaluation is performed securely and efficiently in CS_1 with the help of CS_2, ensuring the quality of the query results while not incurring additional client-side computation. Algorithm 1 illustrates our proposed partition mechanism.

Algorithm 1. Setup$(\mathcal{K}, \{\mathbf{S}^*, \mathbf{S}\}, N, \mathbf{lsh})$

Require: Application server's security parameter: $\mathcal{K}$; Dataset: $\{\mathbf{S}^*, \mathbf{S}\} = \{\langle s_1^*, s_1\rangle, \langle s_2^*, s_2\rangle, \ldots, \langle s_n^*, s_n\rangle\}$; The number of servers: N; LSH parameters: **lsh**.
Ensure: Encrypted index: **I**;
1: **for** $i = 1$ to n **do**
2: $j \leftarrow$ **ConsistHash**(s_i^*);
3: add s_i to partition $\mathbf{P_j}$;
4: **end for**
5: **for** $j = 1$ to N **do**
6: $I_j \leftarrow$ **BuildIndex**$(\mathcal{K}, \mathbf{P_j}, \mathbf{lsh})$;
7: **end for**

Initial database	
s1	id1
s2	id2
s3	id3
…	…
…	…

Fig. 2. An example of the initial index: id_1, id_2, id_3 are identifiers of the data collection, while s_1, s_2, s_3 are associated high-dimensional representations. In this collection, s_1 and s_3 are similar items. In other words, $g_l(s_1||l) = g_l(s_3||l)$.

While EncSIM supports efficient and secure search by using an inverted index with a combination of LSH [9] and SSE, semantic security precludes the possibility to support update operations. Even though it allows for forward-secure addition by maintaining a state table at the client side, it is still unable to support efficient data deletion. A naive way to support deletion is to delete data in the same way as adding data, by just adding a "deleted ID" metadata into the encrypted index, as if a new record is to be added [25]. This, however, increases the index size unnecessarily as the number of delete operations increases, and may not be suitable if updates are frequently made.

To describe our proposed mechanism, let us consider a data collection with three items, each of which is represented by its high-dimensional data s_i and associated identifier, as shown in Fig. 2. We will use this as our running example to illustrate the intuition in our proposed mechanism.

The core design of our mechanism is a dual-index that combines the inverted index and the forward index, while EncSIM only contains the inverted index. The former is used to guarantee search operations with sublinear complexity in distributed key-value stores, while the latter is used for secure data deletion. Note that our design aims to be practical for secure cloud applications with frequent updates. More precisely, in each data partition, we first construct the inverted index by combining LSH and SSE in order to ensure efficient similarity search with sublinear complexity. In particular, for each record, lshL tokens are computed based on lshL pre-defined LSH functions.

By integrating LSH and SSH, both security and query efficiency can be guaranteed. As file-injection attacks [10] highlight the importance of forward security in dynamic SSE, we undoubtedly need to consider forward security while building our mechanism. Different from EncSIM, our mechanism achieves forward security by using fresh keys after each search has been processed: the newly added data associated with the previously queried records would be established with a fresh key. By doing so, the old search tokens would be unusable, and forward security is provided as a result.

With our design, nodes in CS_1 are able to proceed with the search protocol, and obtain encrypted query candidates in parallel. Algorithm 2 gives a formal description of the dual-index construction. More precisely, the application server maintains two hash tables. H is used to manage two key-counter pairs for each LSH value (H_j represents the hash table for partition j), where crt reflects the number of records mapped to an LSH value in the EDB, and ucrt

H_j	
LSH value	crt ‖ ucrt ‖ K ‖ Ku
$g_1(s_1 \| 1)$	1 ‖ 0 ‖ K ‖ Ku
$g_2(s_1 \| 2)$	1 ‖ 0 ‖ K ‖ Ku
$g_1(s_3 \| 1)$	2 ‖ 0 ‖ K ‖ Ku
…	…

(a) H_j is to manage two key-counter pairs for each LSH value for partition j.

$Hcrt_{(id)}$	
Identifier	$crt_{(id)}$
id_1*	1
id_2*	1
id_3*	1
…	…

(b) $crt^{(id)}$ is to reflect the number of forward index entries for "id".

Fig. 3. An example construction of two hash tables maintained by the application server. In our running example, as $g_1(s_1||1) = g_1(s_3||1)$, the counter crt is incremented, while the addition counter ucrt is set to 0 initially, as no addition occurs in the beginning. (Symbol $*$ in our running example represents the value after PRF computation, for example, $id^* \leftarrow F(\hat{K}, id)$).

Encrypted dual-index	
PRF(id_2*,1)	PRF($g_1(s_2 \| 1)$*, 1)
PRF($g_1(s_1 \| 1)$*, 1)	PRF(id_1*, 1)‖Enc(id_1)
…	…
PRF($g_1(s_3 \| 1)$*,2)	PRF(id_3*, 1)‖Enc(id_3)
PRF(id_1*, 1)	PRF($g_1(s_1 \| 1)$*, 1)
…	…

Fig. 4. An example construction of the dual-index maintained in CS_1.

represents the number of newly added records mapped to that particular LSH value. K is the key to encrypt the data stored before a query for that LSH value, K_u is the key for subsequent addition operations (H_j is updated at line 20 / 23 in Algorithm 2). In our running example, Fig. 3 shows the two hash tables maintained by the application server. By encrypting the newly added records using new keys, the adversary would not be able to learn the association between the newly added records and the previously queried ones.

Each entry in $H_{crt(\mathrm{id})}$, $crt^{(\mathrm{id})}$ reflects the number of forward index entries for a record with id. Notably, only the counters crt and $crt^{(\mathrm{id})}$ are updated sequentially, when initializing an EDB. The addition counter ucrt is set to 0 initially, as no addition occurs in the beginning as shown in Fig. 3a. The dual-index of each record consists of two entries: the forward index in line 25 and the inverted index in line 26 (see Algorithm 2). The former entry is used for efficient deletion in sublinear time while the latter is to ensure similarity search in sublinear time, where F_3 and F_4 are secure PRFs. Algorithms 1 and 2 are corresponding to the *Setup* phase, as mentioned in Sec. 2. According to our construction mechanism, the dual-index of our running example is shown as Fig. 4. Note that, for simplicity, we use PRF as a general pseudorandom function in our running example.

Addition. To add a record, the data provider first encrypts it via Hash-ElGamal encryption and sends both the ciphertext and the plaintext $\{s_a^*, s_a\}$ to the application server. Notably, the pre-encryption process for each record is to mitigate the computation overhead of the application server such that the system can scale up to millions of data records. When the application server receives the record to be added, it locates the storage node via consistent hashing. After that, lshL LSH values are computed based on lshL hashes. By retrieving from its maintaining hash table H that contains key/counter pairs for each LSH value (the same process while building the index — refer to line 15 in Algorithm 2), it can obtain the uploading key K_u for corresponding LSH value $\tilde{t}_l$ in the plaintext.

To this end, the newly added indexes are derived from its uploading key and LSH values. For each value, if the corresponding key/counter pairs are not found in H, the new index entries encrypted by the initial uploading key will be added. In other words, when the first time a newly LSH value added in H, its uploading key is the one that initialized it at the beginning. Notably, only the ucrt and $crt^{(id)}$ are incremented in this phase. Algorithm 3 formally describes the addition process.

Algorithm 2. BuildIndex$(\mathcal{K}, \mathbf{P_j}, \mathbf{lsh})$

Require: The application server's security parameter: $\mathcal{K}$; The number of servers: N; LSH parameters: **lsh**.
Ensure: Encrypted Index: I_j;
1: $(\hat{K}, K, K_u) \leftarrow \{0,1\}^{\mathcal{K}}$; //$\hat{K}$ is the key to compute PRF value of id; K is the encryption key before queries; K_u is the encryption key for subsequently addition operations.
2: For each partition/ node, initialize the inverted index hash table H_j, the forward index hash table $H_{crt(\mathrm{id})_j}$ and I_j, lshL;
3: **for** $\forall s \in \mathbf{P_j}$ **do**
4: $K^{(\mathrm{id})} \leftarrow F(\hat{K}, \mathrm{id})$;
5: **for** $l \leftarrow 1$ to lshL **do**
6: $\tilde{t}_l \leftarrow (g_l(s)||l)$;
7: $K_1 \leftarrow F_1(K, \tilde{t}_l)$, $K_2 \leftarrow F_2(K, \tilde{t}_l)$;
//forward index counter updates
8: $crt^{(\mathrm{id})} \leftarrow H_{crt(\mathrm{id})_j}$.**Get**$(K^{(\mathrm{id})})$;
9: **if** $crt^{(id)} \neq null$ **then**
10: $crt^{(id)}++$;
11: $H_{crt(\mathrm{id})_j}$.**Update**$(K^{(\mathrm{id})}, crt^{(\mathrm{id})})$;
12: **else**
13: $crt^{(id)} \leftarrow 1$;
14: $H_{crt(\mathrm{id})_j}$.**Put**$(K^{(\mathrm{id})}, crt^{(\mathrm{id})})$;
15: **end if**
//inverted index counter updates
16: $crt||ucrt||K||K_u \leftarrow H_j$.**Get**$(\tilde{t}_l)$;
//track the counter and encrypted key
17: **if** $crt||ucrt||K||K_u \neq \perp$ **then**
18: parse $(crt||ucrt||K||K_u)$ to get crt, ucrt, K, K_u;
19: $crt++$;
20: H_j.**Update**$(\tilde{t}_l, crt||ucrt||K||K_u)$;
21: **else**
22: $crt \leftarrow 1, ucrt \leftarrow 0$;
23: H_j.**Put**$(\tilde{t}_l, (crt||ucrt||K||K_u))$;
24: **end if**
25: I_j^1.**Put**$(F_3(K^{(\mathrm{id})}, crt^{(\mathrm{id})}), F_4(K_1, crt))$;
26: I_j^2.**Put**$(F_4(K_1, crt), F_3(K^{(\mathrm{id})}, crt^{(\mathrm{id})})||$**Enc**$(K_2, \mathrm{id}))$;
27: **end for**
28: **end for**

We then need to guarantee forward security, i.e., previous queries do not leak any information of the newly added records. To achieve this goal, similar to [12], the uploading key for each LSH is updated when a query for retrieving

that LSH value occurs, which will be shown in the search phase later. Informally, forward security can be achieved in the sense that the cloud could not infer the association between the retrieved records and the newly added records, as they were encrypted using different encryption keys.

Algorithm 3. Addition$((s_a^*, s_a, id), \mathbf{lsh})$

Require: LSH parameters: **lsh**; Hash tables: H, $H_{crt^{(id)}}$.
Ensure: Updated Encrypted Index: **I**;
1: *Application Server:*
2: $j \leftarrow$ **ConsistHash**(s_a^*);
3: $K^{(id)} \leftarrow F(\hat{K}, id)$;
4: Initial added index I_{aj}.
5: **for** $l \leftarrow 1$ to $lshL$ **do**
6: $\tilde{t}_l \leftarrow (g_l(s_a)||l)$;
//forward index counter updates
7: $crt^{(id)} \leftarrow H_{crt^{(id)}_j}$.**Get**$(K^{(id)})$;
8: **if** $crt^{(id)} \neq null$ **then**
9: $crt^{(id)}++$;
10: $H_{crt^{(id)}_j}$.**Update**$(K^{(id)}, crt^{(id)})$;
11: **else**
12: $crt^{(id)} \leftarrow 1$;
13: $H_{crt^{(id)}_j}$.**Put**$(K^{(id)}, crt^{(id)})$;
14: **end if**
//inverted index counter updates
15: $crt||ucrt||K||K_u \leftarrow H_j$.**Get**$(\tilde{t}_l)$;
16: **if** $crt||ucrt||K||K_u \neq \perp$ **then**
17: $ucrt++$;
18: H_j.**Update**$(\tilde{t}_l, crt||ucrt||K||K_u)$;
19: **else**
20: $ucrt \leftarrow 1, crt \leftarrow 0$;
21: K, Ku are the most initial encryption keys setup at the beginning;
22: H_j.**Put**$(\tilde{t}_l, (crt||ucrt||K||K_u))$;
23: **end if**
24: $crt||ucrt||K||K_u \leftarrow H_j$.**Get**$(\tilde{t}_l)$;
25: parse $(crt||ucrt||K||K_u)$ to get crt, $ucrt$, K, K_u;
26: $K_1 \leftarrow F_1(K_u, \tilde{t}_l)$;
27: $K_2 \leftarrow F_2(K_u, \tilde{t}_l)$;
28: I_{ja}^1.**Put**$(F_3(K^{(id)}, crt^{(id)}), F_4(K_1, ucrt))$;
29: I_{ja}^2.**Put**$(F_4(K_1, ucrt), F_3(K^{(id)}, crt^{(id)})||$**Enc**$(K_2, id))$;
30: **end for**
31: Send I_a^1, I_a^2, s_a^* to Node j;
32: *Node j:*
33: Put all key-value pairs in I_{ja}^1, I_{ja}^2 to I_j^1, I_j^2;

Deletion. As shown in Algorithm 4, deletion is quite straightforward with the forward index. The application server first locates the storage node via consistent hashing, and then sends the deletion token derived from the data identifier to the storage node it found. Combining the inverted index and the forward index would lead to twice the amount of storage at the beginning. However, when the number of deletion operations exceeds a certain value, our scheme requires less storage than the solution used in [25], since it ensures that the records are explicitly deleted from the EDB.

Let us consider the following example. Suppose we have n records and would delete m records. Using the naive solution to support deletion for EncSIM, the storage overhead will be $(n+m) \cdot \mathrm{lshL}$. With the dual-index, the storage overhead of building our index would be $2n \cdot \mathrm{lshL}$: $n \cdot \mathrm{lshL}$ for the inverted index and $n \cdot \mathrm{lshL}$ for the forward index. With the deletion process, $2m \cdot \mathrm{lshL}$ index entries would be deleted. It thereby costs a storage overhead of $2(n-m) \cdot \mathrm{lshL}$ in total with our scheme. When $m > (1/3)n$, the overhead of our scheme is less than that of the naïve solution. Another remarkable advantage is that we do not need to rebuild the entire EDB to remove the deletion information. Search is also more efficient since there is no need to perform additional searches over deleted records, as we will explain next.

Algorithm 4. Deletion$(\hat{K}, (s_d^*, s_d, id))$

Require: Application server's encryption $\hat{K}$.
Ensure: Updated Encrypted Index: I;
1: *Application Server:*
2: $j \leftarrow$ **ConsistHash**(s_d^*);
3: $dtok \leftarrow F(\hat{K}, id)$;
4: Send $dtok$ to Node j;
5: *Node j:*
6: $c \leftarrow 1$;
7: **while** I_j^1.**Get**$(F_3(dtok, c)) \neq \perp$ **do**
8: $dval \leftarrow I_j^1$.**Get**$(F_3(dtok, c)$;
9: **if** $dval =$ null **then**
10: return $\perp$;
11: **else**
12: I_j^1.**Del**$(F_3(dtok, c))$; I_j^2.**Del**$(dval)$; $c++$;
13: **end if**
14: **end while**

Search. Algorithm 5 describes our search protocol. To find similar records with a given query s_q, the application server first computes search tokens for the coordinator. Note that a query may occur after some addition operations. As a result, similar records to be retrieved may exist in both the original EDB encrypted with K and the newly added records encrypted with K_u. Accordingly, we process the query using the tokens derived from K for the original EDB for that LSH value, and the tokens derived from K_u for the newly added records (see line 10-12, 21-22 in Algorithm 5).

To provide forward security, we require a re-encryption process in the cloud server to invalidate the old search tokens (see line 8-11 in Algorithm 6). When a record is retrieved, the corresponding indexes are re-encrypted using a new token nt derived from a new encryption key nK. At the same time, the uploading key is updated as nK_u for subsequently secure addition operations. With this re-encryption, the application server would always maintain two key/counter pairs for each LSH value in H. At this point, the old search tokens associated with this query would be unusable, and forward security is provided. In the end, each node of CS_1 holds a subset of the query candidates.

Evaluation. Due to the probabilistic property of LSH, the initial query results can involve several false positives, where the actual distance of the candidates and the query record exceeds a pre-defined threshold [18]. To perform high-quality similarity search, those false positive candidates should be eliminated on the server side. Thus, we introduce an additional evaluation step by resorting to Yao's GCs, whose performance has steadily increased over the years [15].

Algorithm 5. Search$(s_q, \mathbf{lsh}, \mathbf{I}, \mathbf{S}^*, nK)$

Require: Query: s_q; Hash table H; LSH parameters: **lsh**; Encrypted indexes: **I**; Encrypted dataset: $\mathbf{S}^*$
Ensure: Candidate results: **R**;
1: *Application Server:* **SearchToken**
2: $\{nK, nK_u\} \leftarrow \{0,1\}^{\lambda}$;
3: **for** $l \leftarrow 1$ to lshL **do**
4: $\tilde{t}_l \leftarrow (g_l(s_q)||l)$;
5: $crt||ucrt||K||K_u \leftarrow H_j$.**Get**$(\tilde{t}_l)$;
6: $i \leftarrow 0$;
7: **if** $crt||ucrt||K||K_u \neq \perp$ **then**
8: $i++$;
9: parse $crt||ucrt||K||K_u$ to get K, K_u;
10: $t_l \leftarrow (F_1(K, \tilde{t}_l)||F_2(K, \tilde{t}_l))$;
11: $t_{ul} \leftarrow (F_1(K_u, \tilde{t}_l)||F_2(K_u, \tilde{t}_l))$;
12: $nt_l \leftarrow (F_1(nK, \tilde{t}_l)||F_2(nK, \tilde{t}_l))$;
13: $(t_i, crt_i, t_{ui}, ucrt_i, nt_i) \leftarrow (t_l, crt, t_{ul}, ucrt, nt_l)$;
14: H.**Update**$(\tilde{t}_l, crt||ucrt||nK||nK_u)$;
//update the re-encryption key and the uploading key.
15: **end if**
16: **end for**
17: Send $\mathbf{t} = \{(t, crt, t_u, ucrt, nt)\}_i$ to the coordinator;
18: *The cluster of nodes:* **SimSearch**
19: **for all** $(t_i, crt_i, t_{ui}, ucrt_i, nt_i) \in \mathbf{t}$ **do**
20: $ncrt_i \leftarrow 0$; $R_i \leftarrow \emptyset$;
21: **SubSearch**$((t_i, crt_i, nt_i, ncrt_i), \mathbf{R_i})$;
22: **SubSearch**$((t_{ui}, ucrt_i, nt_i, ncrt_i), \mathbf{R_i})$;
23: **end for**
24: Conduct candidates evaluation and obtain the top-R results in parallel with the help of CS_2; sends the $ncrt$ and the R query results to the application server;

Here, we apply a similar approach as proposed in [18], which combines a random masking scheme based on homomorphic encryption with Yao's GCs to avoid data decryption inside the circuits. Inspired by [17], we select the Hash-ElGamal encryption scheme [16] rather than the Paillier encryption used in [18], as it is more efficient and suffices in our case. More precisely, once the node of CS_1 locates a candidate, it adds random masks to s_q^* and s_i^*, respectively. After CS_2 obtains the masked encrypted data records, it first decrypts them via sk and then generates the garbled input of $\hat{s_q}^{\mu_q}$, $\hat{s_i}^{\mu_i}$ for CS_1, where μ_q and μ_i are random masks for s_q^* and s_i^*, respectively. Through a 1-out-of-2 oblivious transfer protocol, the node of CS_1 obtains the garbled input of $\hat{\mu_q}$, $\hat{\mu_i}$ and $\hat{\delta}$, where δ is the pre-defined distance threshold. Then the node runs the evaluation and obtains a boolean result. Fig. 5 illustrates the detailed construction of our circuit. It is worth noting that the *unmasking* function inside the circuits is just an XOR gate, i.e., $s = s^{\mu} \oplus \mu = (s \oplus \mu) \oplus \mu$.

Recall that we aim to ensure that the cloud only needs to return the top-R accurate results, where R is a configurable parameter set up by the application server when a similarity query is issued. Before the evaluation process, the coordinator is required to collect the candidates' identifiers from the cluster, and to determine the top-R candidates roughly based on the number of mapping records for each id in the first round. This is based on the property of LSH that, if a record in an index is highly similar to the queried one, the number of the corresponding index is expected to be higher [9]. After that, the coordinator dispatches the top-R candidates to the corresponding nodes, who would run the evaluation phase subsequently to verify the quality of their candidates in parallel. The decision of query candidates between the coordinator and the cluster stopped only when the R results are qualified.

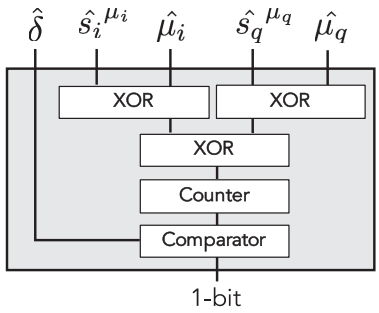

Fig. 5. An illustration of our circuit design.

Algorithm 6. SubSearch$((t, crt, nt, ncrt), \mathbf{R})$

Require: Search token **t**; New encrypted token nt.
Ensure: Updated EDB; updated query candidates; **R**
1: parse $t \rightarrow (K_1, K_2)$, parse $nt \rightarrow (nK_1, nK_2)$;
2: **for** $c = 1$ to crt **do**
3: **if** I_j^2.**Get**$(F_4(K_1, c)) \neq \perp$; **then**
4: parse I_j^2.**Get**$(F_4(K_1, c))$ to get $id^* \leftarrow$ **Enc**(K_2, id) and $F_3(K^{(id)}, crt^{(id)})$;
5: $id \leftarrow$ **Dec**(K_2, id^*);
6: put s_{id}^* to **R**;
7: $ncrt \leftarrow ncrt + 1$;
8: I^1.**Update**$(F_3(K^{(id)}, crt^{(id)})), F_4(nK_1, ncrt))$;
9: I^2.**Del**$(F_4(K_1, c))$;
10: I^2.**Put**$(F_4(nK_1, ncrt), F_3(K^{(id)}, crt^{(id)})||$
11: **Enc**$(nK_2, id))$;
12: **end if**
13: **end for**

Remarks. We target similarity services over encrypted data in distributed key-value stores. The number of query candidates may be considerable and they are distributed across the cluster of nodes in CS_1. According to [17], [26], the GC only offers one-time security. In other words, each candidate's evaluation requires an entirely new circuit with fresh keys. Evaluating large numbers of query candidates will lead to long query latencies, which is not suitable for big data analytics.

To reduce the evaluation latencies, we can further adopt the latest advances of component-based GC [26], which allows the use of offline pre-processing to prepare the entire GC, with only the inputs specified during the online evaluation. Hence, CS_2 can pre-deploy a number of GCs across the cluster of storage nodes. Once the candidates are located, CS_2 only needs to decrypt and garble the masked inputs of each candidate and the query for the corresponding node of CS_1. As the evaluation function and input format are fixed in our scenario, such offline/online GC designs [26] can significantly reduce the overall evaluation time.

Discussions on Backward Privacy. Another relevant privacy property for dynamic SSE schemes is *Backward Privacy*, where (fresh) search queries should not leak matching

records that have been deleted. It was formally studied only recently [13], [14], [27]. Existing schemes that ensure this privacy would incur extra overhead in either computation or communication, which somewhat limits its potential for adoption in practice. In this paper, our focus is on privacy-preserving similarity search with efficient updates in distributed key-value stores. Designing practical mechanisms with backward privacy will be left as future work.

4 Security Analysis

In this section, we will evaluate the security strength of our proposed scheme. In particular, we follow the framework of Kim *et al.*'s forward secure dynamic SSE. Beyond this framework, we also analyze the similarity leakage among queries like [7], [28]. Regarding security in the evaluation phase, the integration of Yao's GCs and Hash-ElGamal ensures the security for the candidates and the queries against the two cloud servers; the output of the evaluation only reveals whether the distance of the candidate and the query is within a threshold.

We will first define leakage functions of our scheme, denoted as $\mathcal{L}$, that restrict the type of information leaked to the adversaries while processing addition, deletion and search queries, respectively. We will then define the security notion based on $\mathcal{L}$, and then prove that our scheme is $\mathcal{L}$-*secure* against adaptive chosen-keyword attacks.

The leakage functions are given as follows:

$$\mathcal{L}_{\mathbf{Setup}}(\mathbf{S}^*) = (N, \{n_1, n_2, \ldots, n_N\}, |s^*|, (|l|, |v|)) \tag{1}$$

$$\mathcal{L}_{\mathbf{Addition}}(s_a) = (|s_a^*|, (|l|, |v|)) \tag{2}$$

$$\mathcal{L}_{\mathbf{Deletion}}((s_d)) = (id) \tag{3}$$

$$\begin{aligned}\mathcal{L}^{sp}_{\mathbf{Search}}(s_q) = (\{|\{g_1(s_q), \ldots, g_{lshL}(s_q)\} \cap \\ \{g_1(s_i), \ldots, g_{lshL}(s_i)\}|\}_{i=1}^{q})\end{aligned} \tag{4}$$

$$\mathcal{L}^{ap}_{\mathbf{Search}}(s_q) = (\{\{\langle l, v\rangle\}_{m_j}, S_j^*\}_{j=1}^{N}), \tag{5}$$

where N is the number of partitions/storage nodes, $\{n_1, \ldots, n_N\}$ is the number of local encrypted records in each node; $|s^*|$ and $(|l|, |v|)$ are the bit length of the encrypted records and the key-value pairs of the encrypted indexes.

We observe that $\mathcal{L}_{\mathbf{Setup}}$ is the only information that can be discovered by CS_1, since the *semi-honest* cloud can access both encrypted indexes and the encrypted data collection. Because our newly added records are encrypted with a fresh key unrelated to the previously retrieved records, the leakage during addition queries only contains the bit length of the added data as shown in Eq. (2). Considering the deletion process, as our scheme ensures actual data deletion, it is inevitable to leak its id. Such leakage shown above does not reveal the actual content of the data records and indexes.

Regarding leakage during the *Search* process, particularly the search pattern, the queries' similarity in addition to the equality can be known according to the size of the intersections between two queries. A token $\mathbf{t_q}$, consisting of $lshL$ LSH values for a given query, is $\{g_1(s_q), \ldots, g_{lshL}(s_q)\}$. Considering another query s_i, if $\mathbf{t_i} = \{g_1(s_i), \ldots, g_{lshL}(s_i)\}$ has at least one matched token as $\mathbf{t_q}$, s_q and s_i are likely similar. The more intersections between s_q and s_i, the more closeness [9] of these two queries. We can therefore define the search pattern against the adversaries as shown in Eq. (4). Since each storage node can observe the retrieved key-value pairs and the matched encrypted data records, the leakage function of the access pattern can be defined as Eq. (5), where m_j denotes the number of the retrieved indexes, and $|S_j^*|$ indicates the number of the retrieved candidates in node j.

Given the leakage functions, we can define the security notion as follows:

Definition 1. *Let* $\prod$ = (Setup, Addition, Deletion, SearchToken, SimSearch) *be the dynamic similarity encryption scheme. Given the leakage functions*

$$(\mathcal{L}_{\mathbf{Setup}},\ \mathcal{L}_{\mathbf{Addition}},\ \mathcal{L}_{\mathbf{Deletion}},\ \mathcal{L}^{sp}_{\mathbf{Search}},\ \mathcal{L}^{ap}_{\mathbf{Search}}),$$

a probabilistic polynomial time adversary $\mathcal{A}$ *and a simulator* $\mathcal{S}$*, we can define two games, including a real game* $\mathbf{Game}_{R,\mathcal{A}}$ *and a ideal game* $\mathbf{Game}_{S,\mathcal{A}}$.

$\mathbf{Game}_{R,\mathcal{A}}(\mathcal{K})$: $\mathcal{A}$ chooses a dataset $\mathbf{S}$. A challenger generates secret keys under the parameter $\mathcal{K}$, builds the encrypted index $\mathbf{I}$ via proceeding **Setup**, and outputs $\mathbf{I}$ to $\mathcal{A}$. Then $\mathcal{A}$ issues addition queries. It first generates the encrypted indexes I_a under the uploading key via proceeding **Addition**. Then the challenger stores the newly added record for $\mathcal{A}$. To delete a record, $\mathcal{A}$ generates the deletion token. With this deletion token, the challenger deletes the corresponding data entries via **Deletion** for $\mathcal{A}$. After a certain number of addition and deletion operations, $\mathcal{A}$ adaptively issues $|q|$ similarity queries. For each similarity query, the challenger generates the token sequence $\mathbf{t}$ via **SearchToken** for $\mathcal{A}$ to query the server through **SimSearch**. Finally, $\mathcal{A}$ returns a bit as the output.

$\mathbf{Game}_{S,\mathcal{A}}(\mathcal{K})$: $\mathcal{A}$ chooses a dataset $\mathbf{S}$. the simulator $\mathcal{S}$ generates simulated index $\hat{\mathbf{I}}$ based on $\mathcal{L}_{\mathbf{Setup}}$. When an addition query is issued, $\mathcal{S}$ generates the transcript of the newly added data entries based on $\mathcal{L}_{\mathbf{Addition}}$. During the simulation of the addition operation, the simulator would also simulate the transcript of the deletion tokens based on the record identifiers. From $\mathcal{L}_{\mathbf{Deletion}}$, it simulates the deletion tokens via search from the transcript. For $|q|$ adaptive similarity queries, $\mathcal{S}$ generates simulated $\hat{\mathbf{t}}$ based on $\mathcal{L}^{sp}_{\mathbf{Search}}$ and $\mathcal{L}^{ap}_{\mathbf{Search}}$ for $\mathcal{A}$ to query the server. Finally, $\mathcal{A}$ returns a bit as the output.

$\prod$ is $(\mathcal{L}_{\mathbf{Setup}}, \mathcal{L}_{\mathbf{Addition}}, \mathcal{L}_{\mathbf{Deletion}}, \mathcal{L}^{sp}_{\mathbf{Search}}, \mathcal{L}^{ap}_{\mathbf{Search}})$-*secure* against the adaptive chosen-keyword attacks if for all probabilistic polynomial time adversaries $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that

$$\left|\mathbf{Pr}[\mathbf{Game}_{R,\mathcal{A}}(\mathcal{K})] - \mathbf{Pr}[\mathbf{Game}_{S,\mathcal{A}}(\mathcal{K})]\right| \leq negl(\mathcal{K}),$$

where $negl$ is a negligible function in $\mathcal{K}$.

Theorem 1. $\prod$ *is* $(\mathcal{L}_{\mathbf{Setup}}, \mathcal{L}_{\mathbf{Addition}}, \mathcal{L}_{\mathbf{Deletion}}, \mathcal{L}^{sp}_{\mathbf{Search}}, \mathcal{L}^{ap}_{\mathbf{Search}})$-secure *against the adaptive chosen-keyword attacks in*

the random oracle model if SE and HE are semantically secure, and F, $F_1 \sim F_4$ are secure PRFs.

The Proof of Theorem 1. *Setup:* Given $\mathcal{L}_{\textbf{Setup}}$, the simulator $\mathcal{S}$ generates a random string s^* with the bit length of $|s^*|$ to simulate each encrypted records. For each partition j, $j \in [1, N]$, $\mathcal{S}$ generates $2 \times lshL \times n_j$ dummy key-value pairs with the bit length of $|l|$ and $|v|$ to simulate the index entries $\hat{\mathbf{I}}_\mathbf{j}$. We can observe that the real and simulated indexes, and the real and simulated encrypted records are computationally indistinguishable. □

Addition. From $\mathcal{L}_{\textbf{Addition}}$, $\mathcal{S}$ can generate the transcript of added indexes when a query $\textbf{Addition}(s_a)$ is issued. Particularly, $\mathcal{S}$ generates a $|s^*|$-bit random string s^* to simulate the added record and $2 \times lshL$ dummy key-value pairs to simulate the corresponding index. Both of the real and simulated indexes, and the real and simulated encrypted records are indistinguishable if the PRFs are semantic secure.

Deletion. During the addition tokens simulation, the deletion token $dtok$ corresponding to the data records' identifiers has been generated in a deletion token transcript. When a deletion query is issued, based on $\mathcal{L}_{\textbf{Deletion}}$, $\mathcal{S}$ can just search the deletion token from the transcript. We observe that the real and the simulated deletion tokens are indistinguishable.

Search. From $\mathcal{L}^{ap}_{\textbf{Search}}$, $\mathcal{S}$ can simulate the first query and the result. In particular, $\mathcal{S}$ generates the transcript of the search token $\{(\hat{t}_i, \hat{crt}_i, \hat{t_{u_i}}, \hat{ucrt}_i, \hat{nt}_i)\}_{lshL}$ with random strings, when a search query $\textbf{Search}(s_q)$ is issued. For each partition j, $j \in [1, N]$, $\mathcal{S}$ uses random oracle H to find key-value pairs. It first obtains the tokens $\{\hat{K}_1, \hat{K}_2\}$ for every t and t_u. Then it locates the indexes via random oracle, i.e., $\hat{\mathbf{I}}_\mathbf{j}.\textbf{Get}(H(\hat{K}_1, c))$ for all tokens, where c increments from 0 and the sum of c_L is equal to $|m_j|$. By replacing $\textbf{Enc}(\hat{K}_2, id)$ with $H(\hat{K}_2||r) \oplus id$, where r is a random string, id can be derived. Simultaneously, $\mathcal{S}$ simulates the re-encrypted entries under the corresponding $\hat{nt}_i$. Based on the $\{id\}_{m_j}$, the simulator can give the simulated retrieved encrypted records $\{S^*_j\}$.

For the subsequent kth query, $\mathcal{S}$ learns the size of the intersection token set from $\mathcal{L}^{sp}_{\textbf{Search}}$, denoted as b. It also learns the number of matched indexes. After that, $\mathcal{S}$ selects b simulated tokens from the first query and generates random tokens for the rest of the $2 \times lshL - b$ tokens, which leads to b overlapped tokens. For the tokens appeared before, $\mathcal{S}$ copies previous matching indexes. Otherwise, it follows the same way of simulating results as in the first query. Due to the semantic security of secure PRFs, the real and simulated tokens, and the real and simulated matching candidates are indistinguishable.

5 Experimental Evaluation

In order to evaluate the effectiveness and efficiency of our proposed work, we have implemented a system prototype in Java, and performed an extensive array of experiments on Microsoft Azure. We utilized virtual machine instances with Intel Xeon E5-2673 v4 (8 vCPU @ 2.3 GHz) and 32 GB of RAM, running the Linux operating system (Ubuntu Server 16.04 LTS). In our implementation, we used Java Cryptography Architecture (JCA) to implement secure pseudo-random functions via HMAC-SHA1.

To show the effectiveness and efficiency of candidate evaluation using garbled circuits, we have implemented our computation protocol based on ObliVM-GC, a well-known secure two-party computation framework [15]. In addition, we used the LSH parameter settings of [18], i.e., both $lshL$ and $lshK$ are set to 5. The Hash-ElGamal encryption library[2] is imported to encrypt the data records for circuit evaluation.

We used the INRIA Copydays dataset[3] to create an EDB. It is a real-world dataset that contains 157 unmodified images, and a group of modifications in different JPEG qualities and cropping percentages. As a result, we have 14,130 images as our plaintext dataset. We use the perceptual image hashing algorithm[4] to extract fingerprints from these original images as our data records collection, where each record is a 64-bit array. The similarity of two images can be measured by Hamming distance of their data records. Furthermore, in order to evaluate the performance of our encrypted index design, including the time of building indexes, additions, and deletions, we also generated 1 million random fingerprints in our experiments. Although our protocols are evaluated on the image dataset, they are readily applied to other data modalities, such as textual data.

5.1 Performance Evaluation

Our performance evaluation focuses on building the encrypted index, performing similarity queries, and carrying out secure evaluations with garbled circuits. Since few approaches can support parallel privacy-preserving similarity search in distributed key-value stores and offer efficient and secure updates simultaneously, we have no baseline to have comprehensive comparisons. An alternative option is the dynamic searchable encryption for large encrypted datasets, proposed by Cash *et al.* [1]. This scheme focuses on the data structure design but could not ensure forward security. For efficiency demonstration, we use Cash *et al.*'s scheme as one of our baselines. For a fair comparison, we implemented and adapted Cash *et al.*'s scheme into our system prototype. We also implemented Cui *et al.*'s approach [18], which focused on similarity search over encrypted near-duplicate data with a public-key based searchable encryption for supporting the "many-to-many" scenario.

Setup. Fig. 6 shows the time cost of building the encrypted index (see Algorithm 2). We observed that the time cost is linear to the number of data records, and the overall speed is very fast compared with the design in [18], completing the task in 114 seconds for one million records with one thread. We have also tested the performance with two threads, which further accelerated this procedure. We observe that our time cost is slightly more than Cash *et al.*' design. This is, of course, an initial one-time cost for a given dataset, which is inevitable for security. Note that our index

2. Hash-ElGamal Lib in Java: https://github.com/harrycui/HashElGamal

3. INRIA Copydays dataset: http://lear.inrialpes.fr/people/jegou/data.php

4. ImageHash library: https://pypi.python.org/pypi/ImageHash

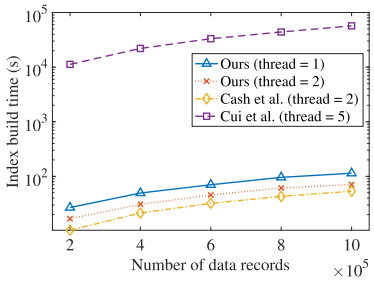

Fig. 6. Index building time comparison.

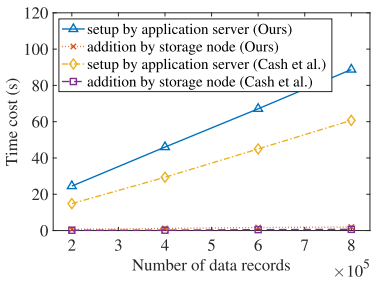

Fig. 7. Addition time (on one node).

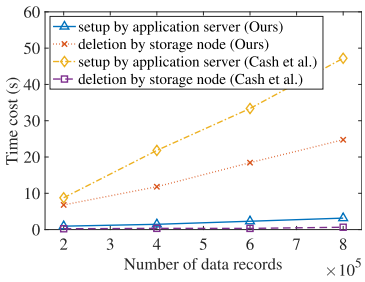

Fig. 8. Deletion time (on one node).

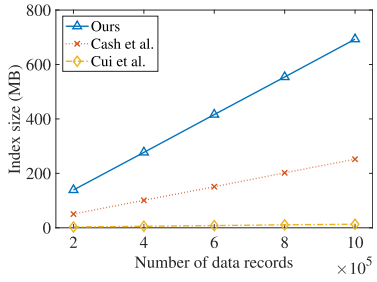

Fig. 9. Index size comparison.

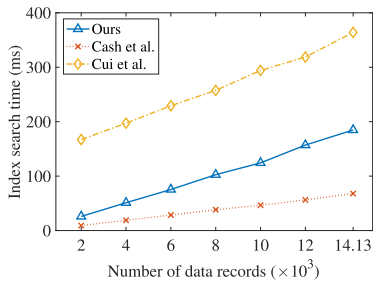

Fig. 10. Search time comparison.

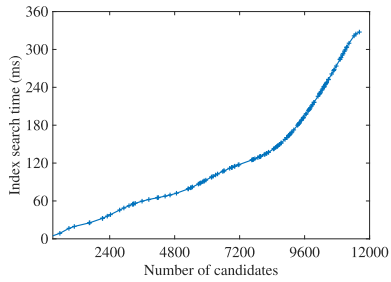

Fig. 11. Initial search time (on one node).

is specialized for similarity search and ensures efficient updates with forward security.

Further, we also measured the space consumption of the prepared encrypted index by using Java serialization. Fig. 9 shows the size of the serialized encrypted index in different scales. For example, for one million data records, the total size is about 693 MB, which could be de-serialized (reconstructed) on each storage node. Here, the index size of Cui *et al.* [18] is significantly small because it is not an index design and only stores small-sized digests (20 bytes for each) together with encrypted identifiers (4 bytes for each). Compared with Cash *et al.* 's scheme, our construction contains an approximate double size of the index, which is not unexpected due to the dual-index design for efficient updates and ensuring forward security.

Dynamic Updates. To evaluate the efficiency of dynamic updates to the EDB, we further perform addition and deletion operations with different numbers of data records. More precisely, we measure the time cost on the side of AP and storage node separately.

In Fig. 7, we observe that the setup time for dynamic addition linearly increases with the number of added records, while the writing time is quite stable and very short, which does not incur much runtime overhead. We note that most of the addition time is consumed by the AP, which is inevitable due to the generation of those encrypted key-value pairs. The overall time cost remains largely the same as the index building operation due to the use of similar processing routines.

In Fig. 8, we observe that the application server completes the setup for deletion very quickly. Deletion on a storage node is slower but still very efficient even for 800,000 records, taking 25 seconds. This is quite different from the addition process, because the application server only needs to generate a token (*dtok*) for each record to be deleted, while the storage node needs to generate multiple keys for the actual key-value pairs in the index.

Search Efficiency. Fig. 10 depicts the average time cost when performing similarity search over the encrypted index, built from 14,130 images. We use the original 157 images as our query dataset, and run multiple rounds on our testing server. Specifically, Fig. 10 shows that our search operation on each storage node is faster than that of Cui *et al.* [18], at millisecond levels, and the overall time cost linearly increases with the number of data records. Fig. 11 further reports the relationship between the search time cost and its located candidates. It is worth noting that the search time is not affected by the dynamic updates (both addition and deletion) in our test, and is only positively affected by the number of located candidates. That can be explained by

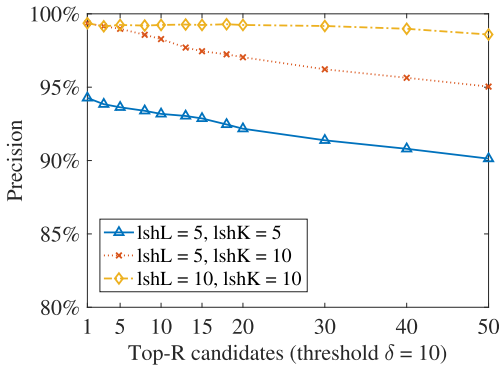

Fig. 12. Precision of initial candidates.

the fact that, with the dual-index data structure, the search operation is performed with sublinear time complexity.

Accuracy Evaluation. Fig. 12 shows the precision of the Top-R results in the initial search phase with different LSH parameters, where both $lshL$ and $lshK$ are 5 in our default setting. Here, *precision* is measured by the percentage of true positives. We can observe that the results are roughly ranked over the number of matched LSH values that still contain false positives (due to the inherent approximation feature of the LSH algorithm), and the precision can be improved by changing the LSH parameters.

In practice, increasing $lshL$ can reduce the number of false positives in the Top-R results (by ranking the number of matched LSH values), but inevitably incurs extra storage overhead. This indicates a tradeoff between query precision and storage costs. Besides that, increasing $lshK$ will reduce the number of located data records, which, without a doubt, contains fewer false positives but incurs more false negatives. Furthermore, we observe that the precision degrades with the increase of R. This confirms the fact that our secure evaluation phase is necessary for high-quality services, which return only the true positive results to the application server.

Moreover, to understand the effect of the underlying homomorphic encryption on the performance of our secure candidates' evaluation, we implemented both our design with Hash-ElGamal encryption and Cui *et al.*'s scheme with Paillier [18], based on ObliVM-GC. Table 2 illustrates a detailed comparison between the two circuits. It shows that the evaluation speed is faster using Hash-ElGamal, which leads to a reduction of 40 percent on the overall evaluation time. This can be explained by the fact that the mask removal module implemented via an XOR gate is efficient, leading to a significant reduction in the overall evaluation time. In addition, our scheme can reduce the bandwidth costs between the two cloud servers by 25 percent.

TABLE 2
An Evaluation of the Secure Evaluation Phase

Design	Num of *and* gate	Num of *xor* gate	Bandwidth (KB)	Time (ms)
Paillier [18]	320	1537	16	121.11
hashElGamal	192	1025	12	78.03

6 Related Work

Encrypted Database Systems. Several practical encrypted database systems have been proposed to support rich queries over encrypted data (e.g., [4], [6], [7], [29]). As examples, CryptDB, one of the first fully functional systems, proposed to use onion encryption for queries with different functionalities [4]. BlindSeer, on the other hand, were designed to achieve better search privacy and support arbitrary boolean queries [29]. Both of CryptDB and BlindSeer are designed as centralized systems.

More recently, several encrypted data analytic systems based on distributed key-value stores have been proposed and implemented [6], [7]. They inherited the features of modern distributed key-value stores, such as high performance and incremental scalability, yet still enabling specific query functions over encrypted data. EncKV [6] is specifically design for exact match and range match, while EncSIM [7] specializes in similarity search on encrypted high-dimensional data records.

Similarity Search Over Encrypted Data. Kuzu *et al.*'s work [28] was one of the first works to address the problem of sub-linear similarity search over encrypted data while preserving data confidentiality. An LSH-based inverted index construction scheme has been proposed, which enables privacy-preserving similarity search in sublinear time complexity. However, dynamic update operations were not considered. Followed by this work, Yuan *et al.* extended the secure similarity index design to support over millions of encrypted records [30]. Such a design is especially suitable for low latency applications, where only a small number of similar records are retrieved for each query. More Recently, Liu *et al.* proposed a privacy-preserving similarity search over distributed key-value stores [7]. Their scheme allows for parallel similarity search over encrypted high-dimensional data records, and supports data addition with guaranteed forward security as well. Yet, it does not support secure and efficient updates, and requires client-side post-processing after decryption.

Dynamic Searchable Symmetric Encryption (DSSE). Kamara *et al.* proposed the first dynamic SSE scheme with sublinear time [3] and later improved this construction by reducing the leakage (of keyword hashes when performing the update operation) at the cost of more space complexity on the server [31]. However, their designs still not achieve forward security. Recall that the information leaked by a DSSE scheme can be captured by backward and forward security. The backward security focuses on the EDB and the updates to it during queries, while the forward security considers the privacy of the EDB and previous queries during update operations. [11], [32], and [12] were proposed to consider forward security of DSSE. $\Sigma o\varphi o\varsigma$ [11] is specialized for the search and insertion operations, but does not support actual deletion, while [12] ensures the secure updates which are specialized for exact-keyword search [12] over encrypted documents. Those schemes for dynamic SSE reduce the efficiency of SE, especially in the communication costs between the client and the server. Vo *et al.* [33] proposed an SGX-based solution to ease the bottleneck. Specifically, they proposed using SGX to take over the client's most tasks and further developed batch data processing and state compression techniques to reduce the communication overhead.

Secure Two-Party Computation Using Garbled Circuits. Huang *et al.* leveraged Yao's garbled circuits to design an

efficient privacy-preserving fingerprint recognition system [34]. Nikolaenko *et al.* proposed an efficient and secure ridge regression protocol, combining an additive homomorphic encryption and Yao's garbled circuits [35]. In [17], by applying a more efficient encryption scheme with homomorphic property — Hash-ElGamal encryption, Nikolaenko *et al.* built a more efficient and secure matrix factorization scheme on top of Yao's garbled circuits. Moreover, Zhang *et al.* proposed a non-interactive framework for binary descriptor based search [36], and Cui *et al.* proposed a secure near-duplicate detection services in the context of encrypted in-network storage [18].

7 Conclusion

In this paper, we have proposed an efficient and privacy-preserving similarity search mechanism with update support in outsourced distributed key-value stores. At the core of our construction lies a dual-index data structure, which not only allows parallel secure search in distributed storage nodes, but also supports forward-private addition and deletion, with sub-linear time complexity. Moreover, to ensure the quality of the search results and release the computation burden of the client, we introduced a secure two-party computation protocol based on Yao's garbled circuit and Hash-ElGamal to securely and efficiently eliminate false positives among the query candidates in the cloud. We have implemented a prototype on Microsoft Azure and conducted an extensive array of performance evaluations using a real-world dataset.

Acknowledgments

This work was supported in part by the Natural Science Basic Research Program of Shaanxi under Grant 2020JQ-215, in part by the Research Grants Council of Hong Kong under Grant CityU 11217819 and Grant CityU 11217620, and in part by the Fundamental Research Funds for the Central Universities under Grant 3102019QD1001.

References

[1] D. Cash *et al.*, "Dynamic searchable encryption in very-large databases: Data structures and implementation," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2014, pp. 23–26.
[2] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: Improved definitions and efficient constructions," *J. Comput. Secur.*, vol. 19, no. 5, pp. 895–934, 2011.
[3] S. Kamara, C. Papamanthou, and T. Roeder, "Dynamic searchable symmetric encryption," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2012, pp. 965–976.
[4] R. A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan, "CryptDB: Protecting confidentiality with encrypted query processing," in *Proc. ACM Symp. Operating Syst. Princ.*, 2011, pp. 85–100.
[5] A. Papadimitriou *et al.*, "Big data analytics over encrypted datasets with seabed," in *Proc. USENIX Symp. Operating Syst. Des. Implementation*, 2016, pp. 587–602.
[6] X. Yuan, Y. Guo, X. Wang, C. Wang, B. Li, and X. Jia, "EncKV: An encrypted key-value store with rich queries," in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, 2017, pp. 423–435.
[7] X. Liu, X. Yuan, and C. Wang, "EncSIM: An encrypted similarity search service for distributed high-dimensional datasets," in *Proc. IEEE/ACM Int. Symp. Qual. Service*, 2017, pp. 1–10.
[8] Y. Hua, W. He, X. Liu, and D. Feng, "SmartEye: Real-time and efficient cloud image sharing for disaster environments," in *Proc. IEEE Conf. Comput. Commun.*, 2015, pp. 1616–1624.
[9] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," in *Proc. IEEE Symp. Foundations Comput. Sci.*, 2006, pp. 459–468.
[10] Y. Zhang, J. Katz, and C. Papamanthou, "All your queries are belong to US: The power of file-injection attacks on searchable encryption," in *Proc. USENIX Secur. Symp.*, 2016, pp. 707–720.
[11] R. Bost, "$\Sigma o \varphi o \varsigma$: Forward secure searchable encryption," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 1143–1154.
[12] K. S. Kim, M. Kim, D. Lee, J. H. Park, and W.-H. Kim, "Forward secure dynamic searchable symmetric encryption with efficient updates," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 1449–1463.
[13] R. Bost, B. Minaud, and O. Ohrimenko, "Forward and backward private searchable encryption from constrained cryptographic primitives," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 1465–1482.
[14] J. G. Chamani, D. Papadopoulos, C. Papamanthou, and R. Jalili, "New constructions for forward and backward private symmetric searchable encryption," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 1038–1055.
[15] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi, "ObliVM: A programming framework for secure computation," in *Proc. IEEE Symp. Secur. Privacy*, 2015, pp. 359–376.
[16] B. Chevallier-Mames, P. Paillier, and D. Pointcheval, "Encoding-free ELGamal encryption without random oracles," in *Proc. Int. Workshop Public Key Cryptogr.*, 2006, pp. 91–104.
[17] V. Nikolaenko, S. Ioannidis, U. Weinsberg, M. Joye, N. Taft, and D. Boneh, "Privacy-preserving matrix factorization," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2013, pp. 801–812.
[18] H. Cui, X. Yuan, Y. Zheng, and C. Wang, "Enabling secure and effective near-duplicate detection over encrypted in-network storage," in *Proc. IEEE Conf. Comput. Commun.*, 2016, pp. 1–9.
[19] R. A. Popa *et al.*, "Building web applications on top of encrypted data using Mylar," in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2014, pp. 157–172.
[20] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart, "Leakage-abuse attacks against searchable encryption," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2015, pp. 668–679.
[21] N. Cao, C. Wang, M. Li, K. Ren, and W. Lou, "Privacy-preserving multi-keyword ranked search over encrypted cloud data," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 1, pp. 222–233, Jan. 2014.
[22] M. Naveed, S. Kamara, and C. V. Wright, "Inference attacks on property-preserving encrypted databases," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2015, pp. 644–655.
[23] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *Proc. Int. Conf. Theory Appl. Cryptographic Techn.*, 1999, pp. 223–238.
[24] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *Proc. ACM Symp. Theory Comput.*, 1997, pp. 654–663.
[25] E. Stefanov, C. Papamanthou, and E. Shi, "Practical dynamic searchable encryption with small leakage," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2014, pp. 72–75.
[26] A. Groce, A. Ledger, A. J. Malozemoff, and A. Yerukhimovich, "CompGC: Efficient offline/online semi-honest two-party computation," *IACR Cryptol. ePrint Arch.*, vol. 2016, pp. 458–476, 2016, Art. no. 458.
[27] S.-F. Sun *et al.*, "Practical backward-secure searchable encryption from symmetric puncturable encryption," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 763–780.
[28] M. Kuzu, M. S. Islam, and M. Kantarcioglu, "Efficient similarity search over encrypted data," in *Proc. IEEE Int. Conf. Data Eng.*, 2012, pp. 1156–1167.
[29] V. Pappas *et al.*, "Blind seer: A scalable private DBMS," in *Proc. IEEE Symp. Secur. Privacy*, 2014, pp. 359–374.
[30] X. Yuan, H. Cui, X. Wang, and C. Wang, "Enabling privacy-assured similarity retrieval over millions of encrypted records," in *Proc. Eur. Symp. Res. Comput. Secur.*, 2015, pp. 40–60.
[31] S. Kamara and C. Papamanthou, "Parallel and dynamic searchable symmetric encryption," in *Proc. Financial Cryptogr. Data Secur.*, 2013, pp. 258–274.

[32] J. Li *et al.*, "Searchable symmetric encryption with forward search privacy," *IEEE Trans. Dependable Secure Comput.*, to be published, doi: 10.1109/TDSC.2019.2894411.
[33] V. Vo, S. Lai, X. Yuan, S.-F. Sun, S. Nepal, and J. K. Liu, "Accelerating forward and backward private searchable encryption using trusted execution," in *Proc. Int. Conf. Appl. Cryptogr. Netw. Secur.*, 2020, pp. 1–12.
[34] Y. Huang, L. Malka, D. Evans, and J. Katz, "Efficient privacy-preserving biometric identification," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2011, pp. 1–40.
[35] V. Nikolaenko, U. Weinsberg, S. Ioannidis, M. Joye, D. Boneh, and N. Taft, "Privacy-preserving ridge regression on hundreds of millions of records," in *Proc. IEEE Symp. Secur. Privacy*, 2013, pp. 334–348.
[36] L. Zhang, T. Jung, C. Liu, X. Ding, X.-Y. Li, and Y. Liu, "POP: Privacy-preserving outsourced photo sharing and searching for mobile devices," in *Proc. Int. Conf. Distrib. Comput. Syst.*, 2015, pp. 308–317.

Wanyu Lin (Member, IEEE) received the BEngr degree from the School of Electronic Information and Communications, Huazhong University of Science and Technology, China, in 2012, the MPhil degree from the Department of Computing, The Hong Kong Polytechnic University, in 2015, and the PhD degree from the Department of Electrical and Computer Engineering, University of Toronto. Her research interests include data outsourcing security in cloud computing, machine learning in adversarial settings, and model interpretation.

Helei Cui (Member, IEEE) received the BE degree in software engineering from Northwestern Polytechnical University, in 2010, the MS degree in information engineering from the Chinese University of Hong Kong, in 2013, and the PhD degree in computer science from the City University of Hong Kong, in 2018. He is currently an associate professor with the School of Computer Science, Northwestern Polytechnical University, China. His research interests include cloud security, mobile security, and multimedia security.

Baochun Li (Fellow, IEEE) received the BEngr degree from the Department of Computer Science and Technology, Tsinghua University, China, in 1995, and the MS and PhD degrees from the Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, in 1997 and 2000, respectively. Since 2000, he has been with the Department of Electrical and Computer Engineering, University of Toronto, where he is currently a professor. He holds the Bell Canada Endowed Chair in Computer Engineering since August 2005. His research interests include cloud computing, distributed systems, datacenter networking, and wireless systems. He was the recipient of the IEEE Communications Society Leonard G. Abraham Award in the Field of Communications Systems, in 2000. In 2009, he was a recipient of the Multimedia Communications Best Paper Award from the IEEE Communications Society, and a recipient of the University of Toronto McLean Award. He is a member of ACM.

Cong Wang (Fellow, IEEE) is an associate professor with the Department of Computer Science, City University of Hong Kong. His current research interests include data and network security, blockchain and decentralized applications, and privacy-enhancing technologies. He is one of the Founding Members of the Young Academy of Sciences of Hong Kong. He received the Outstanding Researcher Award (junior faculty), in 2019, the Outstanding Supervisor Award, in 2017 and the President's Awards, in 2019 and 2016, all from City University of Hong Kong. He is a co-recipient of the IEEE INFOCOM Test of Time Paper Award 2020, Best Student Paper Award of IEEE ICDCS 2017, and the Best Paper Award of IEEE ICDCS 2020, ICPADS 2018, and MSN 2015. He serves/has served as associate editor for the *IEEE Transactions on Dependable and Secure Computing*, *IEEE Internet of Things Journal*, *IEEE Networking Letters*, and *Journal of Blockchain Research*, and TPC co-chairs for a number of IEEE conferences/workshops. He is a member of ACM.