Efficient Searchable Symmetric Encryption for Storing Multiple Source Data on Cloud

Chang Liu[†], Liehuang Zhu[‡], and Jinjun Chen[†],
[†]Faculty of Engineering and Information Technology, University of Technology, Sydney, NSW 2007, Australia
Email: {changliu.bit, jinjun.chen}@gmail.com
[‡]Beijing Engineering Research Center of Massive Language Information Processing and Cloud Computing Application,
School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China
Email: liehuangz@bit.edu.cn

Abstract

Cloud computing has greatly facilitated large-scale data outsourcing due to its cost efficiency, scalability and many other advantages. Subsequent privacy risks force data owners to encrypt sensitive data, hence making the outsourced data no longer searchable. Searchable Symmetric Encryption (SSE) is an advanced cryptographic primitive addressing the above issue, which maintains efficient keyword search over encrypted data without disclosing much information to the storage provider. Existing SSE schemes implicitly assume that original user data is centralized, so that a searchable index can be built at once. Nevertheless, especially in cloud computing applications, user-side data centralization is not reasonable, e.g. an enterprise distributes its data in several data centers. In this paper, we propose the notion of Multi-Data-Source SSE (MDS-SSE), which allows each data source to build a local index individually and enables the storage provider to merge all local indexes into a global index afterwards. We propose a novel MDS-SSE scheme, in which an adversary only learns the number of data sources, the number of entire data files, the access pattern and the search pattern, but not any other distribution information such as how data files or search results are distributed over data sources. We offer rigorous security proof of our scheme, and report experimental results to demonstrate the efficiency of our scheme.

Keywords

Searchable Symmetric Encryption, Multiple Data Sources, Data Outsourcing, Cloud Computing.

I. Introduction

As one of the most successful cloud computing techniques, cloud storage offers elastic storage services within a "pay-as-you-go" mode. More and more cloud users outsource their data to cloud storage providers such as Dropbox and iCloud to diminish the cost of data storage and management. However, security and privacy risks are still the most concern in practical cloud storage usage [1], as cloud users will lose the physical control over their data. Encrypting data before outsourcing is an effective way to guarantee data confidentiality. As a result, electronic health records (ERHs) are legislatively required to be encrypted in several countries [16]

A new challenge comes that encrypted data is not searchable, which will severely influence the retrievability of the data. To address this issue, an advanced cryptographic primitive has been proposed, commonly known as Searchable Symmetric Encryption (SSE). SSE allows a storage provider to answer keyword search queries on encrypted data without learning much information about the data as well as searched keywords. Most SSE schemes [7], [13], [12], [11], [4], [19], [17], [3] require a data owner to build a secure searchable index at a setup phase, so that subsequent keyword searches can be executed in an efficient way.

As far as we know, existing SSE schemes implicitly assume that the searchable index can be directly built by a certain user. This assumption only makes sense when user data is extremely light weight and stored centrally, which is however inconsistent with many cloud computing scenarios. Consider a data owner, whose data is separately stored in several data centers. It is not possible for such data owner to centralize all the data and build a searchable index at once. Personal social data such as chatting records, light weight though, are often stored in several different devices (laptop, ipad, mobile phone). It is not reasonable to require the user to move all data into one device. Moreover, previous SSE schemes are not suitable for data sharing scenarios, for instance that a number of hospitals need to share ERHs. Therefore, it is apparent that we need SSE schemes with support for multiple data sources, which we call Multi-Data-Source SSE (MDS-SSE).

Suppose there are k data sources. A naive approach is that each data source uses any previous SSE scheme to build his/her own searchable index and uploads it to the server. To perform a search, the server needs to search in k indexes respectively. The drawbacks of this approach are two-fold: (1) Much information is leaked as the server learns how data files and search results are distributed over k data sources. Such non-trivial information leakage might lead to severe privacy compromise (e.g., by statistical attacks). (2) There are at least k index searches no matter how many search results will be found.

To hide data file distribution, one should break the correlation between data files and data sources, which means data files should be anonymously transported to the server. This can be achieved using anonymous communication techniques such as Onion Routing [10]. To hide search result distribution, indexes built by each data source should be indistinguishable to each other. It implicitly requires that indexes can be merged at the server side (otherwise the server can search in each index and disclose search result distribution).

We observe that dynamic searchable symmetric encryption (DSSE) schemes [13], [12], [19], [17], [3] can be used in the following way to build only one index rather than k indexes: one data source firstly builds its own searchable index and uploads it to the server while other data sources subsequently append their data using the Update algorithm in DSSE. Nevertheless, this approach still leaks search result distribution because the server is able to identify which results are for the initial index and which are for the updated ones. Moreover, this approach will lead to either more additional leakage [13], [17], or larger data structures [12], or higher computation/communication overheads [19], [3].

We also observe that the basic scheme in [3] can be extended to a secure MDS-SSE scheme (see details in Appendix A). However, the resultant scheme requires data sources to maintain an online table during index building. On one hand, it is not always easy to make every data source have access to the online table. On the other hand, the usage of the online table has potential risks in real-world systems, e.g., by observing how many times a data source touches the table an adversary can learn the number of keywords for this data source.

The contributions of this paper are summarized as follows:

- To the best of our knowledge, we are the first to address the MDS-SSE issue and formally define the notion of MDS-SSE.
- We propose a novel MDS-SSE scheme which is efficient in terms of index size and search time. Our scheme is proven to be secure against adaptive chosen-keyword attacks (CKA2) in the standard model.
- We have implemented our scheme and the scheme in [3]. Experimental results on different types of datasets demonstrate the efficiency of our scheme.

II. Related Work

Searching on remote encrypted data can achieve optimal security guarantee (i.e., nothing is leaked) by using Oblivious Random Access Memory (ORAM) [9]. Though recent works [20], [21] make ORAM much more practical, ORAM remains unacceptable for large scale data outsourcing applications. To maintain practical search, several early SSE works [18], [8], [5], [7] tried to find a proper tradeoff between efficiency and security. Curtmola et al. [7] proposed sound security models for SSE, which called non-adaptive/adaptive[1] (CKA1/CKA2) security. Chase and Kamara [6] further generalized the security models by using leakage functions to parameterize information leakage.

Under CKA1 or CKA2 security model, a series of efficient SSE schemes [7], [6], [13], [12], [19], [17], [3] have been proposed. Their common idea is to build an searchable index before data outsourcing. Each entry in the index is a keyword/identifier pair. Given a keyword, all identifiers whose corresponding data files containing the keyword can be efficiently searched out. In [7], keyword/identifier pairs are organized as linked lists and stored in an array. The authors constructed a look-up table to locate the head node for each linked list. This scheme is proved to be CKA1-secure and achieves optimal search time complexity that only linearly scale with the number of search results. The authors also gave a CKA2-secure scheme which requires much larger index and higher communication overhead. Subsequent SSE schemes use different index structures to extend the functionality of SSE:

More Data Types and Query Types. In [6], Chase and Kamara considered looked-up queries on matrix data, search queires on labeled data, neighbor and adjacency queries on graph data, and subgraph queries on labeled graph data. Jin Li et al. considered fuzzy keyword search queries in [15]. Cash et al. proposed a SSE scheme supporting multi-keyword boolean search in [4].

Dynamic Updates. Dynamic SSE (DSSE) schemes were proposed in [13], [19], [17], [3], which support data dynamic updates. The schemes in [13] and [3] use additional data structures to manage newly added data or deleted data. The scheme in [19] achieves smallest leakage during updates, but requires poly-logarithmic overhead on top of the overhead of [13] and [3]. In [17], the authors present a novel data structure called Blind Storage, which requires more storage space (four times size of original data) but makes the server computation free.

Parallel Search. Parallel search was considered in [12] and [3]. The scheme in [12] uses tree-based index to achieve $O((r/p)\log n)$ parallel search time while the scheme in [3] uses dictionary-based index to achieve $O(r/p)$ parallel search time, where r stands for the number of search results and p stands for the number of processors.

Multiple Users. The notion of *Multi-user* SSE (MSSE) was first proposed in [7]. Using MSSE a data owner can authorize the search ability for arbitrary subset of multiple data users. We can see that MSSE and MDS-SSE are two different notions because the former supports multiple data users while the latter supports multiple data owners.

[1]The adaptiveness depends on whether the adversary can generate query depend on previous answers and the index.

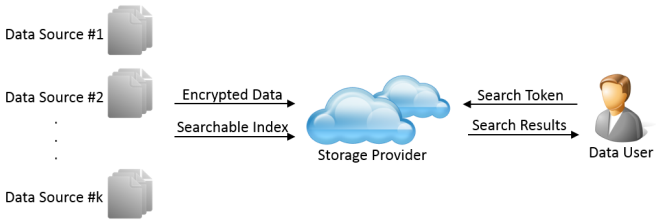


Fig. 1. The system model of MDS-SSE: an illustration

III. PRELIMINARIES

In this section, we first introduce the system model and adversary model of MDS-SSE and then give formal definitions of a typical MDS-SSE scheme and its security model. At last, we introduce several tools and data structures we will use in our construction.

System Model. There are three roles in a MDS-SSE system: (1) k data sources, denoted as $\mathsf{DS} = \{\mathsf{DS}_1, ..., \mathsf{DS}_k\}$, who own k collections of data files. (2) Data users, denoted as DU, who issue search queries for interested keywords. (3) Storage provider, denoted as SP, who stores encrypted user data files and responses DU's search queries. Note that DU can be DS themselves, or any authorized users who share the secret key with DS. As illustrated in Fig. 1, DS encrypt data files and build searchable indexes before data oursourcing. Upon receiving k indexes from DS, SP merges them. DU use secret key to issue search tokens and afterwards SP searches the index and returns the identifiers of data files containing searched keywords.

Adversary Model. We treat untrusted SP as the adversary, who behaves "honest-but-curious". On one hand, the adversary follows all the operations required by the MDS-SSE system model. On the other hand, SP tries to deduce private information about the original data or searched keywords.

MDS-SSE and Security Definition. The following defines a typical MDS-SSE scheme. One can properly extend algorithm's input or output in specific constructions.

Definition 1 (MDS-SSE): A Multi-Data-Source Searchable Symmetric Encryption (MDS-SSE) scheme is a collection of five algorithms (KeyGen, BuildIndex, MergeIndex, TokenGen, Search) and two protocols (Setup, Query), defined as:

- $K \leftarrow \mathsf{KeyGen}(1^\lambda)$: takes as input a security parameter λ and outputs a secret key K.
- $I_{sid} \leftarrow \mathsf{BuildIndex}(K, \mathcal{D}, sid)$: takes as input a secret key K, a collection of data files $\mathcal{D}$ and an identity of a data source sid. It outputs a secure searchable index I_{sid}. The sid can simply be integers so that we denote $1, ..., k$ as the sid for k data sources respectively.
- $I \leftarrow \mathsf{MergeIndex}(\{I_{sid}\}_{1 \le sid \le k})$: takes as input k indexes generated from k data sources. It outputs a merged secure searchable index I.
- $\tau_w \leftarrow \mathsf{TokenGen}(K, w)$: takes as input a secret key K and a search word w. It outputs a search token τ_w.
- $ID(w) \leftarrow \mathsf{Search}(\tau_w, I)$: takes as input a search token τ_w and an index I. It outputs $ID(w)$, a collection of identifiers whose corresponding data files containing the search word w.

- Setup: is performed by DS and SP. During the setup protocol, DS generate and share a secret key, and then prepare all information needed to build indexes. Finally DS outsource encrypted data files and secure searchable indexes while SP stores all the data and merges all indexes.
- Query: is performed by DU and SP. During the query protocol, DU send search tokens to SP and the latter returns search results.

To attain efficiency, most SSE schemes leak the access pattern and the search pattern to the adversary, which are defined as:

Definition 2 (Access Pattern): Let $\mathbf{w} = w_1, ..., w_n$ be a sequence of n searched keywords and $ID(w_1), ..., ID(w_n)$ be their corresponding search results. The access pattern over $\mathbf{w}$ is a tuple $\varphi_{\mathbf{w}} = (ID(w_1), ..., ID(w_n))$

Definition 3 (Search Pattern): Let $\mathbf{w} = w_1, ..., w_n$ be a sequence of n searched keywords. The search pattern over $\mathbf{w}$ is an $n \times n$ matrix $\rho_{\mathbf{w}}$ where $\rho_{\mathbf{w}}[i][j] = 1$ if $w_i = w_j$ and $\rho_{\mathbf{w}}[i][j] = 0$ otherwise ($1 \le i, j \le n$).

We follow the "state-of-the-art" SSE security model in [7], [6] while making slight modifications to fit the multi-data-source scenario. The security model is parametrized by leakage functions $\mathcal{L}_{setup}$ and $\mathcal{L}_{query}$ that indicating information leakage during Setup and Query protocols. Our scheme achieves CKA2 security, which guarantees that nothing is leaked more than the output of $\mathcal{L}_{setup}$ and $\mathcal{L}_{query}$ even if the adversary has the ability to perform adaptive chosen-keyword attacks in probability polynomial time (PPT). We give simulation-based definitions as follows:

Algorithm 1: KeyGen

Input: λ: A security parameter
Output: K: A secret key
1 Randomly select K from $\{0,1\}^\lambda$
2 Output K

Definition 4 (CKA2-security): Let Π = (KeyGen, BuildIndex, MergeIndex, TokenGen, Search, Setup, Query) be an MDS-SSE scheme and $\mathcal{L}_{setup}, \mathcal{L}_{query}$ be leakage functions. Let k be the number of data sources. For an adversary $\mathcal{A}$ and a simulator $\mathcal{S}$, we define the following experiments:

- $\textbf{Real}_{\mathcal{A}}(\lambda)$: the challenger generates a secret key $K = \mathsf{KeyGen}(1^\lambda)$. $\mathcal{A}$ chooses k collections of data files $\mathcal{D}_1, ..., \mathcal{D}_k$ containing n data files in total and a keyword universe $\mathcal{W}$. $\mathcal{A}$ receives $(\{c_j\}_{1\le j\le n}, \{I_j\}_{1\le j\le k})$ such that c_j is an encrypted data file and $I_j \leftarrow \mathsf{BuildIndex}(K, \mathcal{D}_j, j)$. Then $\mathcal{A}$ makes a polynomial number of adaptive queries. For each queried keyword w, $\mathcal{A}$ receives a search token $\tau_w \leftarrow \mathsf{TokenGen}(K, w)$ from the challenger. Finally, $\mathcal{A}$ returns a bit b that is output by the experiment.
- $\textbf{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)$: $\mathcal{A}$ chooses k collections of data files $\mathcal{D}_1, ..., \mathcal{D}_k$ containing n data files in total and a keyword universe $\mathcal{W}$. Given $\mathcal{L}_{setup}(\{\mathcal{D}_j\}_{1\le j\le k}, \mathcal{W})$, $\mathcal{S}$ simulates and sends $(\{c_j^*\}_{1\le j\le n}, \{I_j^*\}_{1\le j\le k})$ to $\mathcal{A}$. Then $\mathcal{A}$ makes a polynomial number of adaptive queries. For each queried keyword w_i, let $\mathbf{w}$ denote the sequence of existing i queried keywords such that $\mathbf{w} = \{w_1, ..., w_i\}$. Given $\mathcal{L}_{query}(\mathbf{w})$, $\mathcal{S}$ simulates and sends a search token $\tau_{w_i}^*$ to $\mathcal{A}$. Finally, $\mathcal{A}$ returns a bit b that is output by the experiment.

We say that Π is $(\mathcal{L}_{setup}, \mathcal{L}_{query})$-secure against adaptive chosen-keyword attacks if for all PPT adversary $\mathcal{A}$, there exists a PPT simulator $\mathcal{S}$ such that

$$|\Pr[\textbf{Real}_{\mathcal{A}}(\lambda) = 1] - \Pr[\textbf{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda) = 1]| \le \mathrm{negl}(\lambda).$$

where negl stands for a negligible function.

Data Structures and Tools. In scheme descriptions we make use of standard data structures including arrays, lists and dictionaries. We use the notation $A[i]$ to denote the value stored at location i of array A. A list simply supports "add" and "delete" operations. A dictionary stores key/value pairs. We use the notation $D[l]$ to denote the value labeled with the key l in dictionary D. $D[l]$ returns $\perp$ if l is not exist in D. The dictionary structure is search efficient such that given a key l, $D[l]$ can be returned in $O(1)$ time. We also make use of cryptographic primitives including variable-input-length pseudo-random function (PRF), pseudo-random permutation (PRP) and symmetric-key encryption (SKE) scheme. The security definitions of PRF, PRP and SKE are clearly presented in [14].

IV. Our MDS-SSE scheme

We first introduce the five algorithms as listed in Definition 1. We then use the five algorithms to construct Setup and Query protocols. In the rest of the paper, let n be the number of total data files, n_i be the number of data files at DS_i and k be the number of data sources. We use $\mathcal{W}$, fid to respectively denote a keyword universe and an identifier of a data file.

A. Key Generation. The key generation algorithm simply chooses a random λ-bit string (see Algorithm 1). The key can be generated by any one of DS or a trusted third party and shared under any secure key distribution protocols such as [2], [22]. In this paper, we assume that a secret key is properly shared among DS and DU.

B. Index Building. Each DS uses BuildIndex algorithm to build a local index for his/her own data. The local index is constructed as an array structure. Each line of the array stores a keyword and a n-bit string indicating which data files contain the keyword. DS set the i-th ($1 \le i \le n$) bit as 1 if the data file with $fid = i$ contains the keyword. For example, supposing $n = 10$, the string 0110000001 means the keyword appears in file 2,3 and 10. The input L is a list of integers indicating which bits (among the total n bits) are assigned for a specific DS. Later, we will discuss how to assign L for each DS.

After obtaining keyword/strings pairs for all keywords in $\mathcal{W}$, let's consider how to encrypt them and how to insert them into the array. Here, we employ two variable-input-length pseudo-random functions PRF and PRF′ with the following parameters:

$$\mathsf{PRF} : \{0,1\}^\lambda \times \{0,1\}^* \to \{0,1\}^\lambda$$
$$\mathsf{PRF}' : \{0,1\}^\lambda \times \{0,1\}^* \to \{0,1\}^n$$

A keyword w is encoded as $\alpha = \mathsf{PRF}(K, 1||w)$ where K is the secret key. Line 11-12 in Algorithm 2 makes the encryption key for a bit string both keyword-specific and data-source-specific. Bit string encryption is simply XOR operation. Note that $\mathcal{W}$ is not required to be confidential, namely, $\mathcal{W}$ is assumed to be known to the adversary[2]. Therefore, DS cannot insert all the

[2]This may happen when DS choose a public keyword list as $\mathcal{W}$.

Algorithm 2: BuildIndex

Input: K: A secret key
$\mathcal{D}$: A collection of data files
$\mathcal{W}$: A keyword universe
L: A list of $|\mathcal{D}|$ integers
sid: An identity of data source
n: The total number of data files
Output: A: An array of keyword/string pairs

1 Set $K_{shuffle} = \mathsf{PRF}(K, 1)$
2 Initialize an array A of length $|\mathcal{W}|$.
3 Initialize a counter $\mathsf{ctr} = 1$
4 **for** $w \in \mathcal{W}$ **do**
5 Initialize a counter $i = 1$
6 Initialize an n-bit zero string γ
7 **for** $f \in \mathcal{D}$ **do**
8 **if** $w \in f$ **then**
9 Set $\gamma[L[i]] = 1$
10 $i = i + 1$
11 Set $\alpha = \mathsf{PRF}(K, 1||w)$, $\beta = \mathsf{PRF}(K, 2||w)$
12 Set $\kappa_{sid} = \mathsf{PRF}'(\beta, sid)$
13 Set $\gamma = \gamma \oplus \kappa_{sid}$
14 Set $A[\mathsf{PRP}(K_{shuffle}, \mathsf{ctr})] =< \alpha, \gamma >$
15 $\mathsf{ctr} = \mathsf{ctr} + 1$
16 Output A

Algorithm 3: MergeIndex

Input: $\{A_i\}_{1 \le i \le k}$: Arrays generated by k data sources
Output: I: A searchable index

1 Initialize an empty dictionary I
2 **for** $i \in \{1, ..., k\}$ **do**
3 Parse A_i as $< \alpha_{i,1}, \gamma_{i,1} >, ..., < \alpha_{i,|\mathcal{W}|}, \gamma_{i,|\mathcal{W}|} >$
4 **for** $j \in \{1, ..., |\mathcal{W}|\}$ **do**
5 Set $\alpha = \alpha_{1,j}$ (note that $\alpha_{1,j} = \alpha_{2,j} = ... = \alpha_{k,j}$)
6 Set $\gamma = \bigoplus_{i=1}^{k} \gamma_{i,j}$
7 Set $I[\alpha] = \gamma$
8 Output I

keyword/string pairs into the array in the order as same as the order in $\mathcal{W}$. To achieve random insertion while keep a same order among all DS, we employ a pseudo-random permutation PRP with the following parameters:

$$\mathsf{PRP} : \{0,1\}^{\lambda} \times \{1, ..., |\mathcal{W}|\} \to \{1, ..., |\mathcal{W}|\}$$

The secret key for PRP, which we call the shuffle key, is the output of $\mathsf{PRF}(K, 1)$. Thus all DS get the same shuffle key and insert keyword/string pairs into the array in a same shuffled order. BuildIndex algorithm is described in detail in Algorithm 2.

C. Index Merging. Upon receiving k arrays (i.e. local indexes) from DS, SP merges them using MergeIndex algorithm. The resultant index is formed as a dictionary structure storing (encrypted) keyword/string pairs. SP merges k arrays line by line. For each line, SP needs to merge k keywords and k bit strings respectively. As k keywords should be same to each other, SP just selects any one of them as the result of "merging" (line 5 in Algorithm 3). To merge k bit strings, SP computes the XOR results for all strings (line 6 in Algorithm 3). SP then inserts the merged keyword/string pair into the dictionary and process the next line. Algorithm 3 displays the details of MergeIndex.

D. Token Generation. DU use the secret key to generate search tokens for keywords they want to search for. A search token is a $< \alpha, \kappa >$ tuple, in which α will be used to locate the correct entry of the dictionary while κ will be used to decrypt the corresponding bit string. Algorithm 4 displays the details of TokenGen.

Algorithm 4: TokenGen

Input: K: A secret key
w: A keyword for search
k: The number of data sources
Output: τ_w: A search token
Set $\alpha = \mathsf{PRF}(K, 1||w)$, $\beta = \mathsf{PRF}(K, 2||w)$
Set $\kappa = \bigoplus_{i=1}^{k} \mathsf{PRF}'(\beta, i)$
Output $\tau_w =< \alpha, \kappa >$

Algorithm 5: Search

Input: τ_w: A search token
I: A searchable index
Output: $ID(w)$: A set of identifiers
Parse τ_w as $< \alpha, \kappa >$
Set $\gamma = I[\alpha]$
Initialize an empty set $ID(w)$
if $\gamma \neq \perp$ **then**
    Set $\gamma = \gamma \oplus \kappa$
    Scan the n-bit string γ and add fid_i in $ID(w)$ if the bit at position i is 1
Output $ID(w)$

E. Search. Upon receiving a search token $\tau_w =< \alpha, \kappa >$ from a DU, SP searches the index using α and obtains a encrypted bit string, and then uses κ to decrypt the bit string. Algorithm 5 displays the details of Search.

We now construct Setup and Query protocols using the five algorithms above.

F. Setup. During the setup protocol, DS encrypt their data files and build searchable indexes. SP receives encrypted data files and indexes from DS and merges all indexes. Firstly, each DS obtains n as follows: $\mathsf{DS}_i (1 \leq i \leq k)$ encrypts n_i and sends it to SP. SP then sends all encrypted n_i to every DS. Each DS thus can decrypt all n_i and add them together to obtain n. Secondly, DS_i needs to know which n_i bits in the n-bit string are assigned for him/her (i.e., the list L in Algorithm 2). To achieve random assignment[3], we use another pseudo-random permutation PRP' with the following parameters:

$$\mathsf{PRP}' : \{0,1\}^{\lambda} \times \{1, ..., n\} \rightarrow \{1, ..., n\}$$

to shuffle the sequence $(1, ..., n)$. DS_1 selects serial n_1 integers from the shuffled sequence, and DS_2 selects serial n_2 subsequent integers, and so on. Thirdly, DS encrypt all data files and send them to SP via anonymous communication. Fourthly, DS build local indexes using algorithm BuildIndex and send them to SP. Fifthly, SP merges all indexes using algorithm MergeIndex. The details of Setup are listed in Protocal 1. ENC and DEC in Protocal 1 are encryption and decryption algorithms from a secure SKE scheme.

G. Query. During the query protocol, DU issue search tokens using algorithm TokenGen and then SP returns search results output by algorithm Search. The details of Query are listed in Protocal 2.

V. Security

In this section, we analyze the security of our MDS-SSE scheme. We first define the leakage functions in our scheme as follows:

- $\mathcal{L}_{setup}(\{\mathcal{D}_j\}_{1 \leq j \leq k}, \mathcal{W}) = (k, n, \{|c_j|\}_{1 \leq j \leq n}, \mathcal{W})$
- $\mathcal{L}_{query}(\mathbf{w}) = (\varphi_{\mathbf{w}}, \rho_{\mathbf{w}})$

where $k, n, |c_j|, \mathcal{W}, \varphi_{\mathbf{w}}, \rho_{\mathbf{w}}$ stands for the number of data collections, the total number of data files, the length of encrypted file c_j, the keyword universe, the access pattern over the query sequence $\mathbf{w}$ and the search pattern over the query sequence $\mathbf{w}$, respectively.

Theorem 1: If $\mathsf{PRF}, \mathsf{PRF}', \mathsf{PRP}, \mathsf{SKE}$ are secure cryptographic primitives, then our scheme is $(\mathcal{L}_{setup}, \mathcal{L}_{query})$-secure against adaptive chosen-keyword attacks (CKA2) in the standard model.

[3]This randomization is not required in our security proof, but to defend adversaries with additional background knowledge. For example, if we assign a serial substring for each data source and an adversary has the knowledge that a data source is most likely to search his/her own data, then the adversary can observe an intensively touched substring and roughly estimate the number of data files for this data source.

Protocol 1: Setup

(Assume all DS share a secret key $K \leftarrow \mathsf{KeyGen}(\lambda)$)

$\underline{\mathsf{DS}_i}$ $(1 \le i \le k)$:
Set $n'_i = \mathsf{ENC}(K, n_i)$
Send n'_i to SP

$\underline{\mathsf{SP}}$:
Send $\{n'_1, ..., n'_k\}$ to every DS

$\underline{\mathsf{DS}_i}$ $(1 \le i \le k)$:
for $j \in \{1, ..., k\}$ **do**
  $n_j = \mathsf{DEC}(K, n'_j)$
Set $n = \sum_{j=1}^{n} n_j$, $n' = \sum_{j=1}^{i-1} n_j$
$K'_{shuffle} = \mathsf{PRF}(K, 2)$
Shuffle $(1, ..., n)$ as $(a_1, ..., a_n)$ by computing $a_j = \mathsf{PRP}'(K'_{shuffle}, j)$ for $j \in \{1, ..., n\}$
Initialize an empty list L
for $j \in \{n' + 1, n' + n_i\}$ **do**
  Append a_j to L
for $f_j \in \mathcal{D}_i (1 \le j \le n_i)$ **do**
  Set $fid_j = L[j]$
  Let fid_j be the identifier of f_j
  $c_j = \mathsf{ENC}(K, f_j)$
  Send $< fid_j, c_j >$ to SP via anonymous communication
$A_i \leftarrow \mathsf{BuildIndex}(K, \mathcal{D}_i, \mathcal{W}, L, i, n)$
Send A_i to SP

$\underline{\mathsf{SP}}$:
Set $I \leftarrow \mathsf{MergeIndex}(\{A_i\}_{1 \le i \le k})$

Protocol 2: Query

$\underline{\mathsf{DU}}$:
Input a search word w
$\tau_w \leftarrow \mathsf{TokenGen}(K, w, k)$
Send τ_w to SP.

$\underline{\mathsf{SP}}$:
Set $ID(w) \leftarrow \mathsf{Search}(\tau_w, I)$
Send $ID(w)$ to DU

Proof: In the experiment **Real** (described in Definition 4), $\mathcal{A}$ receives $(\{c_j\}_{1 \le j \le n}, \{I_j\}_{1 \le j \le k}, \{\tau_j\}_{1 \le j \le q})$. In addition to the typical definition, $\mathcal{A}$ also receives k encrypted integers $\{n'_j\}_{1 \le j \le k}$ during the Setup protocol. We construct a simulator $\mathcal{S}$ in the experiment **Ideal** (described in Definition 4), who simulates $(\{n'^*_j\}_{1 \le j \le k}, \{c^*_j\}_{1 \le j \le n}, \{I^*_j\}_{1 \le j \le k}, \{\tau^*_j\}_{1 \le j \le q})$ using the output of $\mathcal{L}_{setup}$ and $\mathcal{L}_{query}$. We prove that $\mathcal{A}$ cannot distinguish between experiments **Real** and **Ideal**. In other words, $\mathcal{A}$ cannot distinguish between $(\{n'^*_j\}_{1 \le j \le k}, \{c^*_j\}_{1 \le j \le n}, \{I^*_j\}_{1 \le j \le k}, \{\tau^*_j\}_{1 \le j \le q})$ and $(\{n'_j\}_{1 \le j \le k}, \{c_j\}_{1 \le j \le n}, \{I_j\}_{1 \le j \le k}, \{\tau_j\}_{1 \le j \le q})$. In the following proof, when we say "indistinguishable" or "cannot distinguish" we mean the advantage in distinguishing two variables is limited by $\mathrm{negl}(\lambda)$.

Simulating $\{n'_j\}_{1 \le j \le k}$: $\mathcal{S}$ randomly selects a key K^* of length λ and k integers $n^*_1, ..., n^*_k$. For each integer $n^*_i (1 \le i \le k)$, $\mathcal{S}$ sets $n'^*_i = \mathsf{ENC}(K^*, n^*_i)$. As SKE is secure, $(n'^*_1, ..., n'^*_k)$ and $(n'_1, ..., n'_k)$ are indistinguishable to $\mathcal{A}$.

Simulating $\{c^*_j\}_{1 \le j \le n}$: For $1 \le j \le n$, $\mathcal{S}$ randomly selects a bit string c^*_j of length $|c_j|$. As SKE is secure, $\{c^*_j\}_{1 \le j \le n}$ and $\{c_j\}_{1 \le j \le n}$ are indistinguishable to $\mathcal{A}$.

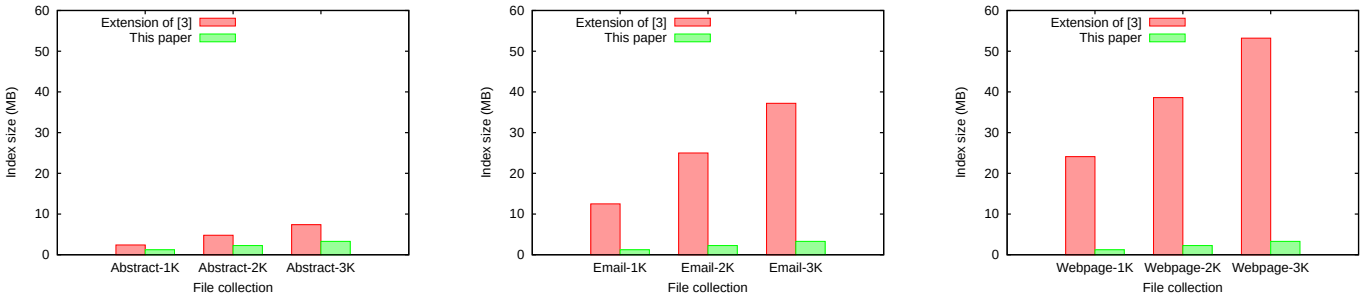
Simulating $\{I^*_j\}_{1 \le j \le k}$: $\mathcal{S}$ initialize k arrays $\{I^*_j\}_{1 \le j \le k}$ of size $|\mathcal{W}|$. $\mathcal{S}$ randomly selects $|\mathcal{W}|$ bit strings $\{\alpha^*_j\}_{1 \le j \le |\mathcal{W}|}$ of length λ and $k \cdot |\mathcal{W}|$ bit strings $\{\gamma^*_{c,j}\}_{1 \le c \le k, 1 \le j \le |\mathcal{W}|}$ of length n. $\mathcal{S}$ sets $I^*_c[j] =< \alpha^*_j, \gamma^*_{c,j} >$ for $1 \le c \le k, 1 \le j \le |\mathcal{W}|$.

As PRF is a secure pseudo-random function, $\{I^*_j\}_{1 \le j \le k}$ and $\{I_j\}_{1 \le j \le k}$ are indistinguishable to $\mathcal{A}$.

Simulating $\{\tau^*_j\}_{1 \le j \le q}$: $\mathcal{S}$ computes $I \leftarrow \mathsf{MergeIndex}(\{I^*_j\}_{1 \le j \le k})$. For $1 \le i \le q$, according to $\rho_{\mathbf{w}}$

Name	# of files	# of words
Abstract-1K	1,000	186,938
Abstract-2K	2,000	369,747
Abstract-3K	3,000	576,051
Email-1K	1,000	882,210
Email-2K	2,000	1,780,264
Email-3K	3,000	2,723,640
Webpage-1K	1,000	5,353,872
Webpage-2K	2,000	8,004,950
Webpage-3K	3,000	10,821,006

TABLE I. DOCUMENT COLLECTIONS



(a) Abstract-* (b) Email-* (c) Webpage-*

Fig. 2. Index size evaluation: Varying # of data files.

- if the i-th query has never appeared before, then $\mathcal{S}$ randomly selects an α_i^* from $\{\alpha_j^*\}_{1\leq j\leq |\mathcal{W}|}$ (as the security of PRP guarantees that each query touches a random index entry), making sure α_i^* has not been selected before. From $\varphi_{\mathbf{w}}$ $\mathcal{S}$ extracts the search results of the i-th query, denoted as $ID(w_i)$, and generates a bit string γ^* of length n, making the j-th bit of γ^* be 1 if $j \in ID(w_i)$ and 0 otherwise. $\mathcal{S}$ computes $\kappa_i^* = \gamma^* \oplus I[\alpha_i^*]$. $\mathcal{S}$ sets $\tau_i^* =< \alpha_i^*, \kappa_i^* >$
- if the i-th query is the same with a prior query, which we suppose to be the j-th $(j < i)$ query. $\mathcal{S}$ sets $\tau_i^* =< \alpha_j^*, \kappa_j^* >$.

In such a way, $\mathcal{S}$ simulates correct search tokens which have the same search results as in the experiment **Real**. Therefore, $\mathcal{A}$ cannot distinguish between $\{\tau_j^*\}_{1\leq j\leq q}$ and $\{\tau_j\}_{1\leq j\leq q}$.

In summary, $\mathcal{A}$ cannot distinguish between the view in the experiment **Real** and the view in the experiment **Ideal**. Thus we have

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

■

VI. PERFORMANCE EVALUATION

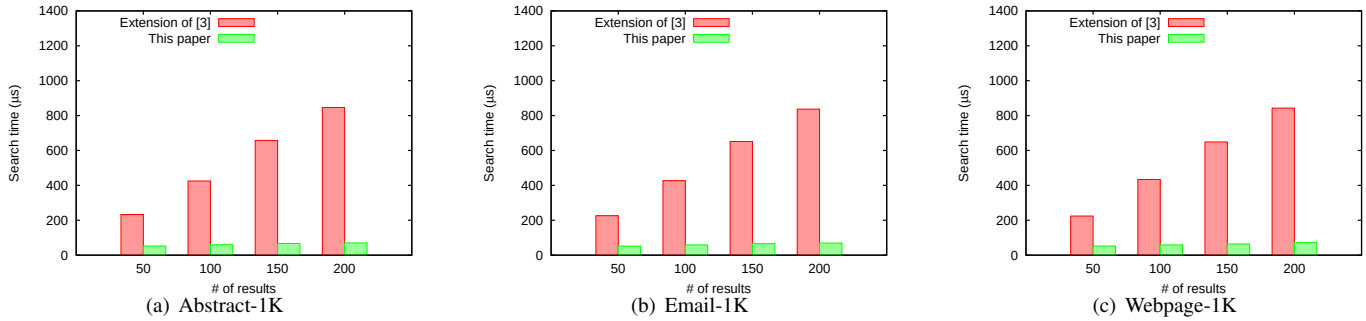
We implemented our scheme and the extended scheme of [3] (Appendix A) using Python 2.7.3. In this section we compare the performance of the two schemes in terms of index size and search time under different dataset settings. All test programs were performed on an Intel Core i5-4570 3.20GHz computer with 8GB RAM running Windows 8.1. Each data point in the figures is an average of 10 executions.

Dataset. We chose three types of real-world text datasets, as shown in Table I. The Abstract-* collections were extracted from PubMed dataset[4]. Each data file contained an abstract of a paper. The Email-* collections were extracted from Enron dataset[5]. Each data file was an email without attachments. The Webpage-* collections were extracted from DBLife dataset[6]. Each data file was an webpage of a personal homepage. In all tests, we used a keyword universe of 3000 common English words. We fixed k as 5. Note that k value does not influence the index size and the search time for both schemes.

Fig. 2 reports index sizes for nine data collections. From individual subfigures we can observe that the index size of both schemes grows linearly to the number of data files. For our scheme, as the length of bit string equals to the number of data files, the increase of the length of bit string leads to the increase of the index size. For [3], the increase of the index size is due to the increase of the number of keyword/identifier pairs.

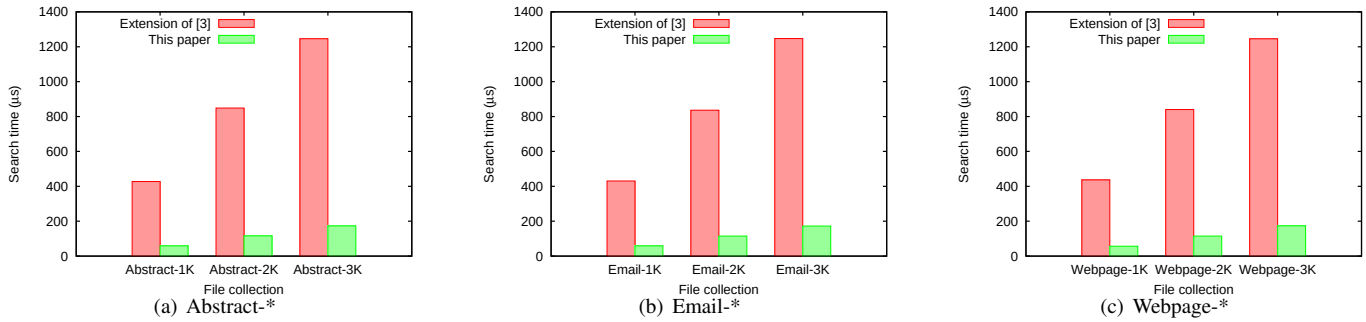
While comparing the three subfigures 2(a)-2(c), we can see that the index size of [3] is sensitive to the total number of words. The more words in the collection, the more keyword/indentifier pairs need to be indexed. Such a case will happen until the collection has included all keywords in the keyword universe. By contrast, the index size of our scheme remains stable in three types of data collections, because its index size is only decided by the size of keyword universe and the number of data files.

[4]Crawled from http://www.ncbi.nlm.nih.gov/pubmed
[5]Downloaded from http://www.cs.cmu.edu/~./enron/
[6]Crawled from http://dblife.cs.wisc.edu/



(a) Abstract-1K (b) Email-1K (c) Webpage-1K

Fig. 3. Search time evaluation: Varying # of search results.



(a) Abstract-* (b) Email-* (c) Webpage-*

Fig. 4. Search time evaluation: Varying # of data files.

In all 9 collections, index sizes of our scheme are smaller than that of [3]. Therefore our scheme is storage efficient and well-suited to applications where data files are large.

For both schemes, the search time cost is from three kinds of operations, namely, dictionary search, result decryption and result extraction. The actual number of operations needed is decided by the number of data files and/or the number of search results. Thus we tested two schemes' search performance by respectively varying the above two parameters.

In the first test (Fig. 3), we fixed the number of data files as 1000. Different queried keywords result in different numbers of search results, so we report the search time of four specific keywords with 50, 100, 150, 200 results respectively. From any of subfigures 3(a)-3(c) we can see that the search time of both schemes increases when more and more search results are returned. For our scheme, more search results leads to more result extractions, but brings no change in dictionary search and result decryption. Moreover, the result extraction in our scheme is merely to scan a bit string and find all positions where the bit is 1. Such operations are extremely efficient so that the increase in search time is slight. But for [3], more search results means more dictionary searches, more result decryptions and more result extractions. That's why the search time of [3] is much more sensitive to the number of results than that of our scheme.

In the second test (Fig. 4), we kept the number of search results accounting for 10% of the entire collection rather than a fixed number, because in practice a specific query usually associates a fixed percentage of data files. From subfigures 4(a)-4(c) we can observe how the increasing number of data files affects the search time. For our scheme, the search time increases because result decryption spends more time on a longer bit string. Result extraction also requires a little more time on string scanning but such time can be ignored. For [3], more results will be returned when the number of data files grows, which results in more time needed in a search.

While comparing between subfigures 3(a)-3(c), 4(a)-4(c), we can see there is no significant change in the search time for both schemes when the number of words increases. Though the index size of [3] increases when more words are included, the usage of dictionary structure guarantees a stable search time. From the results we can see that the search time of our scheme is much smaller than that of [3].

VII. Conclusion

Motivated by the practical phenomenon in data outsourcing scenarios that user data is often separately distributed, we propose a novel MDS-SSE scheme. The work of [3] is the only existing work that imports no additional information leakage in the multiple data source setting after proper modification. Our scheme outperforms the scheme in [3] in two aspects. Firstly, their scheme has to maintain an online table while ours has no such requirement. Secondly, our scheme has better performance in terms of index size and search time under different kinds of data collections.

REFERENCES

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, 2010.

[2] M. Bellare and P. Rogaway. Entity authentication and key distribution. In *Advances in Cryptology - CRYPTO'93*, pages 232–249, 1993.

[3] D. Cash, J. Jaeger, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner. Dynamic searchable encryption in very large databases: Data structures and implementation. In *Network and Distributed System Security Symposium*, NDSS'14, 2014.

[4] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Roşu, and M. Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *Advances in Cryptology - CRYPTO'13*, pages 353–373. 2013.

[5] Y.-C. Chang and M. Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In *Applied Cryptography and Network Security*, pages 442–455, 2005.

[6] M. Chase and S. Kamara. Structured encryption and controlled disclosure. In *Advances in Cryptology - ASIACRYPT'10*, pages 577–594. 2010.

[7] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: Improved definitions and efficient constructions. In *Proceedings of the 2006 ACM Conference on Computer and Communications Security*, CCS'06, pages 79–88, 2006.

[8] E.-J. Goh. Secure indexes. Cryptology ePrint Archive, Report 2003/216, 2003. http://eprint.iacr.org/.

[9] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996.

[10] D. Goldschlag, M. Reed, and P. Syverson. Onion routing. *Commun. ACM*, 42(2):39–41, 1999.

[11] S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Outsourced symmetric private information retrieval. In *Proceedings of the 2013 ACM conference on Computer and Communications Security*, CCS'13, pages 875–888, 2013.

[12] S. Kamara and C. Papamanthou. Parallel and dynamic searchable symmetric encryption. In *Financial Cryptography and Data Security*, pages 258–274. 2013.

[13] S. Kamara, C. Papamanthou, and T. Roeder. Dynamic searchable symmetric encryption. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS'12, pages 965–976, 2012.

[14] J. Katz and Y. Lindell. *Introduction to modern cryptography: principles and protocols*. CRC Press, 2007.

[15] J. Li, Q. Wang, C. Wang, N. Cao, K. Ren, and W. Lou. Fuzzy keyword search over encrypted data in cloud computing. In *Proceedings of the 2010 IEEE INFOCOM Mini-Conference*, pages 1–5, 2010.

[16] H. Löhr, A.-R. Sadeghi, and M. Winandy. Securing the e-health cloud. In *Proceedings of the 2010 ACM International Health Informatics Symposium*, IHI'10, pages 220–229, 2010.

[17] M. Naveed, M. Prabhakaran, and C. A. Gunter. Dynamic searchable encryption via blind storage. In *IEEE Symposium on Security and Privacy*, SP'14, 2014.

[18] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searching on encrypted data. In *IEEE Symposium on Security and Privacy*, SP'00, pages 44–55, 2000.

[19] E. Stefanov, C. Papamanthou, and E. Shi. Practical dynamic searchable encryption with small leakage. In *Network and Distributed System Security Symposium*, NDSS'14, 2014.

[20] E. Stefanov and E. Shi. Oblivistore: High performance oblivious cloud storage. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP'13, pages 253–267, 2013.

[21] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path oram: An extremely simple oblivious ram protocol. In *Proceedings of the 2013 ACM Conference on Computer and Communications Security*, CCS'13, pages 299–310, 2013.

[22] M. Steiner, G. Tsudik, and M. Waidner. Diffie-hellman key distribution extended to group communication. In *Proceedings of the 1996 ACM Conference on Computer and Communications Security*, CCS'96, pages 31–37, 1996.

APPENDIX A
EXTENSION OF THE SCHEME IN [3]

We show how to properly extend the basic scheme in [3] to fit the MDS-SSE setting. A compact description is listed in Scheme 1. To prevent conflicts in using counters[7] among data sources, the scheme has to import an online table (denoted as T in Scheme 1) during index building. The online table associates an counter initialized as 0 for all keywords in $\mathcal{W}$ (as illustrated in Fig. 5(a)). Every time a DS_i processes a keyword, DS_i retrieves the current counter of this keyword from the online table, and then updates it by adding $|ID_i(w)|$. The retrieved counter will be the initial counter for this keyword at DS_i's side (which is always 0 in the original scheme). The illustrative tables shown in Fig. 5(b) and Fig. 5(c) are updated by two DS who has 37 and 25 data files containing w_1, respectively.

w_1	0
w_2	0
...	...

(a) Initial state

w_1	37
w_2	0
...	...

(b) After first updating

w_1	62
w_2	0
...	...

(c) After second updating

Fig. 5. Online table: an illustration.

[7] In [3], each keyword/identifier pair is assigned an integer counter number.

Scheme 1: Extension of the scheme in [3]

Setup (by DS_i and SP)

DS_i $(1 \leq i \leq k)$:

Share a secret key $K \in \{0,1\}^\lambda$ with other DS
Initialize an empty list L_i
for $w \in \mathcal{W}$ **do**
    $K_1 = F(K, 1||w)$, $K_2 = F(K, 2||w)$
    Retrieve the current counter c of w in T
    Update c to $c + |ID_i(w)|$ in T
    for $fid \in ID_i(w)$ **do**
        $l = F(K_1, c)$, $d = \mathsf{ENC}(K_2, fid)$, $c = c + 1$
        Add $< l, d >$ into L_i
for $f \in \mathcal{D}_i$ **do**
    $f = \mathsf{ENC}(K, f)$
Send $\mathcal{D}_i$ to SP via anonymous communication
Slice L_i into fragments of equal size and send each fragment to SP via anonymous communication

SP:

Initialize an empty dictionary I
Insert keyword/identifier pairs in each fragment into I

Query (by DU and SP)

DU:

Input a search word w
$K_1 = F(K, 1||w)$, $K_2 = F(K, 2||w)$
Send (K_1, K_2) to SP

SP:

for $c = 0$ *until* $d = \perp$ **do**
    $d = I[F(K_1, c)]$, $fid = \mathsf{DEC}(K_2, d)$, $c = c + 1$
Send each fid to DU