2017

Dynamic Searchable Symmetric Encryption with Physical Deletion and Small Leakage

Peng Xu
Huazhong University of Science and Technology

Shuai Liang
Huazhong University of Science and Technology

Wei Wang
Huazhong University of Science and Technology

Willy Susilo
University of Wollongong, wsusilo@uow.edu.au

Qianhong Wu
Beihang University, qhw@uow.edu.au

See next page for additional authors

Publication Details

Xu, P., Liang, S., Wang, W., Susilo, W., Wu, Q. & Jin, H. (2017). Dynamic Searchable Symmetric Encryption with Physical Deletion and Small Leakage. Lecture Notes in Computer Science, 10342 207-226. 22nd Australasian Conference on Information Security and Privacy (CISP2017)

Dynamic Searchable Symmetric Encryption with Physical Deletion and Small Leakage

Abstract
Dynamic Searchable Symmetric Encryption (DSSE) allows a client not only to search over ciphertexts as the traditional search- able symmetric encryption does, but also to update these ciphertexts according to requirements, e.g., adding or deleting some ciphertexts. It has been recognized as a fundamental and promising method to build secure cloud storage.

Keywords
small, deletion, leakage, encryption, symmetric, physical, dynamic, searchable

Disciplines
Engineering | Science and Technology Studies

Publication Details
Xu, P., Liang, S., Wang, W., Susilo, W., Wu, Q. & Jin, H. (2017). Dynamic Searchable Symmetric Encryption with Physical Deletion and Small Leakage. Lecture Notes in Computer Science, 10342 207-226. 22nd Australasian Conference on Information Security and Privacy (CISP2017)

Authors
Peng Xu, Shuai Liang, Wei Wang, Willy Susilo, Qianhong Wu, and Hai Jin

Dynamic Searchable Symmetric Encryption with Physical Deletion and Small Leakage

Peng Xu[1], Shuai Liang[1], Wei Wang[2], Willy Susilo[3], Qianhong Wu[4] and Hai Jin[1]

[1] Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China
[2] Cyber-Physical-Social Systems Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China
[3] Centre for Computer and Information Security Research, School of Computing and Information Technology, University of Wollongong, Australia
[4] Electronics and Information Engineering, Beihang Univerisity, Beijing, China
Email: xupeng@mail.hust.edu.cn

Abstract. Dynamic Searchable Symmetric Encryption (DSSE) allows a client not only to search over ciphertexts as the traditional searchable symmetric encryption does, but also to update these ciphertexts according to requirements, e.g., adding or deleting some ciphertexts. It has been recognized as a fundamental and promising method to build secure cloud storage. In this paper, we propose a new DSSE scheme to overcome the drawbacks of previous schemes in the state-of-art. The biggest challenge is to realize the physical deletion of ciphertexts with small leakage. We employ both logical and physical deletions, and run physical deletion in due course to avoid extra information leakage. Our instantiation achieves noticeable improvements throughout all following aspects: search performance, storage cost, functionality, and information leakage when operating its functions. We also demonstrate its provable security under adaptive attacks and practical performance according to experimental results.

1 Introduction

Symmetric-key encryption with keyword search (or searchable symmetric encryption, SSE for short) allows clients to upload their keyword searchable ciphertexts to a server, and then delegate keyword search to the server and retrieve files of an expected keyword. A secure SSE scheme can keep the privacy of keywords not only to outside attackers but also to the server. The details are as follows: for all keywords of a file, a client respectively generates the corresponding keyword searchable ciphertexts and the encrypted file in symmetric-key setting, and stores these ciphertexts in the server; to retrieve the files of an expected keyword, the client delegates a keyword search token to the server, and then the server finds out all matching keyword searchable ciphertexts, decrypts out

their encrypted file identifiers, and returns the corresponding encrypted files of these identifiers to the client; finally, the client decrypts out these files. Since the encryption of files can be separately processed with an independent symmetric-key encryption scheme, SSE only focus on the generation of keyword searchable ciphertexts. Hence, unless the clear statement, all encryptions or ciphertexts are searchable in this paper.

In the past decade, most of researches on SSE focus on improving security, accelerating search performance or searching with multiple keywords. Until 2012, Kamara *et al.* [1] first proposed a dynamic SSE (DSSE) scheme (called KPR'12 in our paper) by constructing hidden chains to connect all searchable ciphertexts of the same keyword. With a keyword search trapdoor, these hidden chains will be partially disclosed and guide the server to efficiently find out all matching ciphertexts. In addition, KPR'12 can add new ciphertexts to their corresponding chains or delete old ciphertexts from these chains. Clearly, DSSE is more flexible than the traditional SSE both in theory and practice.

But The KPR'12 scheme causes significant information leakage in updating ciphertexts. Specifically, when adding or deleting ciphertexts, it will leak some information of the corresponding chains, e.g., the number of ciphertexts in a chain. These leakage information makes the server having a noticeable advantage to guess keywords. In 2013, Kamara *et al.* [2] modified their previous work by sharply increasing the size of searchable ciphertexts. Technically, this new DSSE scheme (called KP'13 in our paper) generates secure vectors for all keywords, where each vector is of size linear with the number of all files, and then constructs a tree structure for these vectors to accelerate search performance. Since each keyword is contained only by a part of files in practice, these vectors contain many redundancies. Hence, KP'13 takes a high storage complexity.

In 2014, Cash *et al.* [3] proposed a DSSE scheme (called CJJ'14) by applying private counters to constructing hidden relationships among all keyword searchable ciphertexts. Technically, each keyword has a private counter which is initiated as "1"; to generate a searchable ciphertext of a keyword, the current value of the corresponding counter will be taken as input, and after the generation, the counter will be added with "1"; when receiving a keyword search trapdoor, the server can efficiently find out all matching ciphertexts by traversing all possible values of a counter. CJJ'14 is more convenient than KPR'12 and KP'13. But it cannot physically delete ciphertexts. In other words, only logical deletion is achieved by taking extra storage to remember the deleted ciphertexts. When searching a keyword, the deleted ciphertexts will not be taken into account even if they contain the keyword. Hence, its storage complexity will consistently increase with the total number of both adding or deleting operations. This disadvantage also appears in the work of [4].

Rough comparisons of the above DSSE schemes are listed in Table 1 (the exact comparisons will be given in Section 7). The summary is that no previous work is good at all aspects of search complexity, storage complexity, functionality and information leakage. Hence, we are interested in proposing a new DSSE scheme to complete this work in the state-of-art.

Table 1: Rough comparisons. **Search** is to find out the files containing a queried keyword. **AddFile** is to add all keyword searchable ciphertexts of a new file. **AddKeyword** is to add a new keyword searchable ciphertext of an existing file. **DeleteFile** is to delete all keyword searchable ciphertexts of an existing file. **DeleteKeyword** is to delete a keyword searchable ciphertext of an existing file. Information leakage denotes the information leaked by running previous functions.

Scheme	**Search** Complexity	Storage Complexity	Functions				Information Leakage
			AddFile	**AddKeyword**	**DeleteFile**	**DeleteKeyword**	
KPR'12[1]	Low	Normal	Achieved	Failed	Physical	Failed	Large
KP'13[2]	Normal	High	Achieved	Failed	Physical	Failed	Normal
CJJ'14[3]	Low	Low	Achieved	Achieved	Failed	Logical	Small
Ours	Low	Low	Achieved	Achieved	Physical	Physical	Small

1.1 Our Main Ideas

Before introducing our main ideas, some basic concepts are needed. In the paradigmatic application of DSSE, each file has an unique identifier and several keywords. It is common to let file-keyword pair (id, w) denote that the file with identifier id has keyword w. And suppose database DB consists of all such pairs, which are derived from the application.

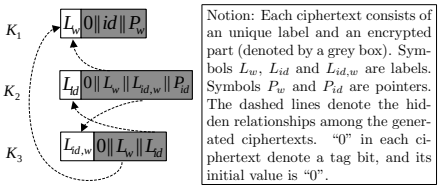


Fig. 1: The generated ciphertexts for pair (id, w).

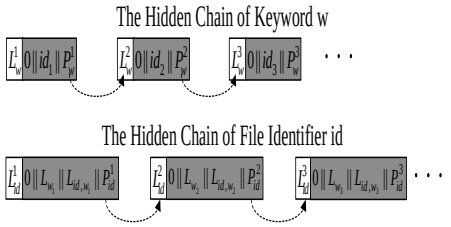


Fig. 2: The generated hidden chain relationship among ciphertexts, where $P_w^1 = L_w^2$, $P_w^2 = L_w^3$, $P_{id}^1 = L_{id}^2$ and $P_{id}^2 = L_{id}^3$.

For each pair in DB, our scheme will generate three kinds of searchable ciphertexts, called K1, K2 and K3, respectively. The three kinds of ciphertexts are used to accordingly achieve functions **Search**, **DeleteFile** and **DeleteKeyword**, respectively. Taking pair (id, w) as an example, Figure 1 shows the generated three ciphertexts, including their hidden relationships. All K1 ciphertexts of the same keyword construct a hidden chain relationship, like the chain of keyword w in Figure 2; all K2 ciphertexts of the same file identifier also construct a hidden chain relationship, like the chain of identifier id in Figure 2; no hidden relationship among K3 ciphertexts is needed.

When searching a keyword with the corresponding keyword search token, function **Search** finds out the hidden chain of the keyword, and follows the guidance of the chain to rapidly find out all matching K1 ciphertexts. By decrypting these matching ciphertexts, function **Search** finally obtains some file identifiers, which refer to the files containing the queried keywords. However, our chains among K1 ciphertexts have some differences with that of KPR'12. One difference is that we do not generate the deterministic heads for all possible chains when initializing a DSSE scheme as KPR'12 did. On the contrary, the head of a chain in our DSSE scheme is dynamically generated when the keyword corresponding to the chain is the first time to be used. So we do not take extra storage for any chain's deterministic head.

When adding a new keyword searchable ciphertext of an existing file with identifier id (let w' be the new keyword), function **AddKeyword** generates three above mentioned ciphertexts of pair (id, w'). The generated K1 ciphertext will be linked to the end of the w''s chain, and similarly the K2 ciphertext will be linked to the end of the id's chain. Summarily, our idea is to link the new generated ciphertexts to the end of chains. This idea is inspired by the drawback of KPR'12 that linking ciphertexts to the middle of chains causes some impact on the old ciphertexts in the chains, i.e. extra information leakage. In addition, we just need to support function **AddKeyword**, since function **AddFile** can be achieved by executing function **AddKeyword** multiple times.

The biggest challenge in our work is to realize the physical deletion with small leakage when running functions **DeleteKeyword** and **DeleteFile**. Roughly, both functions need to delete some ciphertexts from chains. If the deletion is physical, some operations after the deletion are necessary to repair the broken chains. For example, suppose that the second ciphertext in the hidden chain of keyword w in Figure 2 is physically deleted. To repair the broken chain, values L_w^1 and L_w^3 must be known, and then set $P_w^1 = L_w^3$. These operations leak the fact that the ciphertexts with labels L_w^1 and L_w^3 are in the same chain with the ciphertext with label L_w^2. In the worst case, all ciphertexts in a chain will be leaked. To overcome this challenge, our idea is to employ both logical and physical deletions, and run physical deletion in due course to avoid extra information leakage. The details are as follows.

First, our K3 ciphertexts are used to support function **DeleteKeyword**. When deleting all searchable ciphertexts of a file-keyword pair (id, w), a delete token allows server to quickly find out the matching K3 ciphertext and then decrypt out two indices. Referring to Figure 1, these two indices respectively correspond to a K1 ciphertext and a K2 ciphertext. Then the K1 and K2 ciphertexts are logically deleted by setting their tag bits to be "1". Finally, that matching K3 ciphertext is physically deleted.

Second, our K2 ciphertexts are used to support function **DeleteFile**. When deleting all searchable ciphertexts of a file with identifier id, a delete token allows the server to find out the hidden chain of id. Following this chain, all matching K2 ciphertexts can be rapidly found. Referring to Figure 1, each matching K2 ciphertext can be decrypted out two indices, and these indices allow server to

find out all related K1 and K3 ciphertexts. Finally, all found K1 ciphertexts are logically deleted by setting their tag bits to be "1", and all found K2 and K3 ciphertexts are physically deleted.

Third, our above ideas show the physical deletion of K2 and K3 ciphertexts and the logical deletion of K1 ciphertexts. Hence, our final goal is to physically delete the K1 ciphertexts. This work is elegantly achieved when the search process is rebooted per keyword according to our extra design. Then all logically deleted K1 ciphertexts of the queried keyword will be physically deleted, and the corresponding chain will be repaired. Summarily, we do the physical deletion of the K1 ciphertexts by function **Search**, not by function **DeleteFile** as previous schemes did. This method does not cause extra information leakage, since the inherent information leakage of function **Search** is enough for repairing a broken chain which is caused by the physical deletion.

1.2 Our Contributions

Compared with KPR'12, we extend the definition of DSSE by additionally defining functions **AddKeyword** and **DeleteKeyword**. As a result, the new DSSE definition consists of five protocols that are **Setup**, **AddKeyword**, **DeleteFile**, **DeleteKeyword** and **Search**. A client runs protocol **Setup** to generate searchable ciphertexts, which will be stored in a server. The other Protocols allow the client to delegate the corresponding operations to the server. We also extend the traditional security definition, which is called indistinguishability under adaptively chosen keyword attacks (IND-CKA2). In our new security definition, a more powerful adversary is modeled. Specifically, in either a real or an ideal attack game, an adversary is allowed to make a polynomial number of adaptive operations to engage in protocols **AddKeyword**, **DeleteFile**, **DeleteKeyword** or **Search**.

Before proposing our complete DSSE scheme, we construct two basic DSSE schemes to help the understanding of our ideas. The first one shows how to construct the hidden chains among searchable ciphertexts, add new searchable ciphertexts to the corresponding chains, and search a keyword according to the guidance of the corresponding chain. The second one shows how to employ both logical and physical deletions to delete all searchable ciphertexts of a file. From these two basic DSSE schemes, we construct our complete DSSE scheme provably IND-CKA2 secure in the random oracle (RO) model. The two basic DSSE schemes are also of independent interest in the applications in which limited functionalities are sufficient.

We make thorough comparisons between our complete DSSE scheme and the related KPR'12, KP'13 and CJJ'14 schemes. The comparisons show that our scheme has noticeable advantages in all aspects of search complexity, storage complexity, functionality and information leakage. Finally, we show the practicality of our scheme with extensive experimental results in executing its main functions.

1.3 Organization of The Remainder

Sections 2 and 3 respectively define some symbols, two common data structures, DSSE and its IND-CKA2 security. Sections 4 and 5 respectively show our two basic DSSE schemes. Section 6 proposes our complete DSSE scheme and its provable IND-CKA2 security. In Section 7, we exactly compare our complete DSSE scheme with some previous schemes, and then show the practice of our scheme by numerical results. The other related works on SSE are reviewed in Section 8. Section 9 concludes this paper.

2 Defining Symbols and Data Structures

We let symbol $k \in \mathbb{N}$ denote the security parameter. The set of all binary strings of length $n \in \mathbb{N}$ is denoted as $\{0,1\}^n$, and the set of all finite binary strings denoted as $\{0,1\}^*$. We write $x \xleftarrow{\$} \mathcal{X}$ to represent an element x being sampled uniformly at random from the set $\mathcal{X}$. The output x of an algorithm $\mathbf{A}$ is denoted by $x \leftarrow \mathbf{A}$. $|\mathcal{X}|$ represents the size of set $\mathcal{X}$ or the total number of members in set $\mathcal{X}$. $\mathcal{W}$ denotes the keyword space. Each file is denoted by an unique identifier. $\mathcal{ID}$ is the set of all the file identifiers. (id, w) is a file-keyword pair, where $w \in \mathcal{W}$ and $id \in \mathcal{ID}$. DB is a database or a set of different (id, w) pairs. $|DB|$ is the total number of the pairs in DB (or the size of DB). $DB(w)$ is the set of the file identities that pair with keyword $w \in \mathcal{W}$ in DB. Similarly, $DB(id)$ is a set of the keywords that pair with file $id \in \mathcal{ID}$ in DB. If pair $(id, w) \in DB$ holds, $|DB(id, w)| = 1$, otherwise $|DB(id, w)| = 0$. Symbol $||$ denotes the concatenation of strings.

Our schemes will employ two standard data structures *List* and *Dictionary*. When $\mathcal{T}$ is a *List*, $|\mathcal{T}|$ denotes the total number of records in $\mathcal{T}$. There are four operations on dictionary $\mathcal{D}$. We define them as follows:

- **Creat**$(\mathcal{T})$: Take a list $\mathcal{T}$ of label-data pairs as input (where each label is unique), and return a dictionary $\mathcal{D}$;
- **Get**$(\mathcal{D}, L)$: Take a dictionary $\mathcal{D}$ and a label L as inputs, return the corresponding data D if $(L, D) \in \mathcal{D}$, otherwise return $NULL$;
- **Update**$(\mathcal{D}, (L, D))$: Take a dictionary $\mathcal{D}$ and a label-data pair (L, D) as inputs, insert (L, D) into $\mathcal{D}$ if L does not exist in $\mathcal{D}$, otherwise update the original data of label L into the new data D, and finally return $\perp$;
- **Remove**$(\mathcal{D}, L)$: Take a dictionary $\mathcal{D}$ and a label L as inputs, delete record (L, D) from $\mathcal{D}$ and return $\perp$;

Note that the dictionary algorithm **Creat**$(\mathcal{T})$ is history-independent [3]. It means that for any list $\mathcal{T}$ the distribution of $\mathcal{D} \leftarrow$ **Creat**$(\mathcal{T})$ depends only on the records of $\mathcal{T}$ not on the members' order in $\mathcal{T}$. In addition, the time complexity of algorithm **Get** is $O(1)$.

3 Defining DSSE and Its Security

To simplify the description of our DSSE concept, we generalize several algorithms defined in KPR'12, and then add two new protocols **AddKeyword** and **DeleteKeyword**. We also makes the following assumptions: (1) All file identifiers will never be re-used; (2) The searchable ciphertexts of the same file-keyword pair will never be re-added; (3) Keyword space $\mathcal{W}$ and set $\mathcal{ID}$ have $\mathcal{W} \bigcap \mathcal{ID} = \emptyset$.

Definition 1 (DSSE). *A DSSE scheme consists of the following five protocols between a client and a server:*

- **Setup***: The client takes a security parameter and a database DB as inputs, generates an initial encrypted database EDB, some secret parameters like secret keys, and sends EDB to the server. The server stores EDB.*
- **AddKeyword***: To add the searchable ciphertexts of a new file-keyword pair (id, w), the client takes the file-keyword pair and his secret parameters as inputs, generates and sends the corresponding searchable ciphertexts to the server. The server takes the encrypted database EDB as input, and inserts these ciphertexts into EDB.*
- **DeleteFile***: To delete all searchable ciphertexts of a file with identifier id, the client takes the file's identifier and his secret parameters as inputs, generates and sends a* delete *token to the server. The server takes the encrypted database EDB as input, deletes all corresponding searchable ciphertexts from EDB.*
- **DeleteKeyword***: To delete the searchable ciphertexts of a file-keyword pair (id, w), the client takes the file-keyword pair and his secret parameters as inputs, generates and sends a* delete *token to the server. The server takes the encrypted database EDB as input, deletes the corresponding searchable ciphertexts from EDB.*
- **Search***: To find out the files containing an expected keyword w, the client generates takes the keyword and his secret parameters as inputs, generates and sends a* search *token to the server. The server takes the encrypted database EDB as input, outputs the file identifiers which means that the corresponding files contain the keyword.*

The IND-CKA2 security of a DSSE scheme defines two games: a real game $\mathbf{Real}_{\mathcal{A}}$ between an adversary $\mathcal{A}$ and a challenger and an ideal game $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}$ between the adversary $\mathcal{A}$ and a simulator $\mathcal{S}$. In game $\mathbf{Real}_{\mathcal{A}}$, the challenger sets up a real DSSE scheme, and the adversary $\mathcal{A}$ adaptively engages in every protocol of DSSE by querying the challenger. On the contrary, in game $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}$, the simulator $\mathcal{S}$ sets up a simulated DSSE scheme. It means that $\mathcal{S}$ never knows the real database DB chosen by $\mathcal{A}$, and it only takes leakage functions as inputs to simulate the functions of the challenger. If $\mathcal{A}$ cannot distinguish games $\mathbf{Real}_{\mathcal{A}}$ and $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}$, we say that the DSSE scheme is IND-CKA2 secure. Moreover, the smaller the leakage, the stronger the IND-CKA2 security.

Definition 2 (IND-CKA2 Security). *Let* **DSSE** = (**Setup**, **AddKeyword**, **DeleteFile**, **DeleteKeyword**, **Search**) *be a DSSE scheme, $\mathcal{A}$ be a stateful*

adversary, $\mathcal{S}$ be a stateful simulator, and ($\mathcal{L}_{Setup}$, $\mathcal{L}_{AddKeyword}$, $\mathcal{L}_{DeleteFile}$, $\mathcal{L}_{DeleteKeyword}$, $\mathcal{L}_{Search}$) be stateful leakage functions. Consider the following probabilistic experiments:

- **Real**$_{\mathcal{A}}(k)$*: $\mathcal{A}$ chooses DB. A challenger runs* **Setup** *to generate some secret parameters and the encrypted database EDB of DB. $\mathcal{A}$ receives EDB and makes a polynomial number of adaptive operations to engage in protocol* **AddKeyword**, **DeleteFile**, **DeleteKeyword** *or* **Search**. *For each query, the challenger returns the corresponding result such as the searchable ciphertexts that will be added to EDB, a token to delete all searchable ciphertexts of a file, a token to delete the searchable ciphertexts of a file-keyword pair or a* search *token. Finally, $\mathcal{A}$ returns one bit b as the output of this experiment.*
- **Ideal**$_{\mathcal{A},\mathcal{S}}(k)$*: $\mathcal{A}$ chooses DB. Given $\mathcal{L}_{Setup}$, $\mathcal{S}$ simulates and sends EDB to $\mathcal{A}$. $\mathcal{A}$ makes a polynomial number of adaptive operations to engage in protocols* **AddKeyword**, **DeleteFile**, **DeleteKeyword** *or* **Search**. *For each query, $\mathcal{S}$ is with the corresponding leakage $\mathcal{L}_{AddKeyword}$, $\mathcal{L}_{DeleteFile}$, $\mathcal{L}_{DeleteKeyword}$ or $\mathcal{L}_{Search}$, then returns an appropriate result such as the searchable ciphertexts that will be added to EDB, a token to delete all searchable ciphertexts of a file, a token to delete the searchable ciphertexts of a file-keyword pair or a* search *token. Finally, $\mathcal{A}$ returns one bit b as the output of this experiment.*

If $|Pr[\mathbf{Real}_{\mathcal{A}}(k) = 1] - Pr[\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(k) = 1]|$ is negligible, we say that **DSSE** *is IND-CKA2 secure with leakage functions ($\mathcal{L}_{Setup}$, $\mathcal{L}_{AddKeyword}$, $\mathcal{L}_{DeleteFile}$, $\mathcal{L}_{DeleteKeyword}$, $\mathcal{L}_{Search}$),.*

4 Our Basic DSSE Scheme D-I

Our basic DSSE scheme D-I only consists of protocols **Setup**, **AddKeyword** and **Search**. Given a database DB, protocol **Setup** shows the generation of K1 ciphertexts, so that all ciphertexts of the same keyword are connected by a hidden chain. To add the K1 ciphertext of a new file-keyword pair, protocol **AddKeyword** connects the new generated ciphertext to the end of the corresponding chain. With a keyword search trapdoor, protocol **Search** shows how to quickly find out the related file identifiers. Let $\mathbf{F} : \{0,1\}^k \times \{0,1\}^* \rightarrow \{0,1\}^k$ be a key-based pseudo-random function. Let $\mathbf{H} : \{0,1\}^* \rightarrow \{0,1\}^{2k}$ be a cryptographic hash functions. The basic scheme D-I is shown in Figure 3.

In protocol **Setup**, each keyword w has a pointer parameter P_w, and each K1 ciphertext is a label-data pair. When generating the K1 ciphertext for a file-keyword pair (id, w), the generated label L_w is equal to $\mathbf{F}_{k_1}(w)$ if it is the first time to generate a ciphertext for keyword w, otherwise it is equal to P_w, and the data D_w is the encryption of file identifier id and a new value of P_w. With the same method, all K1 ciphertexts of the same keyword are connected by a hidden chain, since the value of P_w encrypted by the former one of any two neighboring ciphertexts in a chain is equal to the label of the latter one. In addition, the final value of P_w is privately recorded by the client at the end of

★ Protocol **Setup**$((k, DB), NULL)$:
- Client: Take k and DB as inputs, randomly choose two secret keys $\mathcal{K} = (k_1, k_2)$, initialize two empty lists $\mathcal{T}_P$ and $\mathcal{T}_W$, and do the following steps:
  1. For each keyword w in DB, initialize pointer parameter $P_w = NULL$;
  2. For each file-keyword pair $(id, w) \in DB$
     (a) If $P_w = NULL$, set label $L_w = \mathbf{F}_{k_1}(w)$, otherwise set $L_w = P_w$; Set $\{R, P_w\} \xleftarrow{\$} \{0,1\}^{2k}$;
     (b) Generate a K1 ciphertext $(L_w, D_w = (D_{w,1} = (\mathbf{H}(\mathbf{F}_{k_2}(w), R) \oplus (id||P_w)), D_{w,2} = R))$, and add this ciphertext to $\mathcal{T}_W$ in the lexicon order;
  3. For each keyword w in DB, add tuple (w, P_w) into $\mathcal{T}_P$; Generate dictionaries $\mathcal{D}_P \leftarrow \mathbf{Creat}(\mathcal{T}_P)$ and $\mathcal{D}_W \leftarrow \mathbf{Creat}(\mathcal{T}_W)$; Keep $\mathcal{K}$ and $\mathcal{D}_P$ secret, and send the encrypted database $EDB = \mathcal{D}_W$ to the server;
- Server: Store EDB.

★ Protocol **AddKeyword**$((\mathcal{K}, \mathcal{D}_P, id, w), EDB)$:
- Client: Take $\mathcal{K} = (k_1, k_2)$, $\mathcal{D}_P$ and a file-keyword pair (id, w) as inputs, retrieve $P_w \leftarrow \mathbf{Get}(\mathcal{D}_P, w)$ according to w, and do the following steps:
  1. If $P_w = NULL$, set $L_w = \mathbf{F}_{k_1}(w)$, otherwise set $L_w = P_w$; Set $\{R, P_w\} \xleftarrow{\$} \{0,1\}^{2k}$;
  2. Generate a K1 ciphertext $(L_w, D_w = (D_{w,1} = (\mathbf{H}(\mathbf{F}_{k_2}(w), R) \oplus (id||P_w)), D_{w,2} = R))$, run algorithm $\mathbf{Update}(\mathcal{D}_P, (w, P_w))$, and send the ciphertext (L_w, D_w) to the server;
- Server: Take $EDB = \mathcal{D}_W$ and (L_w, D_w) as inputs, and run algorithm $\mathbf{Update}(\mathcal{D}_W, (L_w, D_w))$.

★ Protocol **Search**$((\mathcal{K}, w), EDB)$:
- Client: Take $\mathcal{K} = (k_1, k_2)$ and a keyword w as inputs, generate and send a search token $ST_w = (\mathbf{F}_{k_1}(w), \mathbf{F}_{k_2}(w))$ to the server;
- Server: Take $EDB = \mathcal{D}_W$ and $ST_w = (\mathbf{F}_{k_1}(w), \mathbf{F}_{k_2}(w))$ as inputs, initialize an empty set $\mathcal{I}$, set $L_w = \mathbf{F}_{k_1}(w)$, and do the following steps:
  1. Retrieve data $D_w \leftarrow \mathbf{Get}(\mathcal{D}_W, L_w)$ according to L_w; If $D_w = NULL$, return $\mathcal{I}$ and abort;
  2. Parse $D_w = (D_{w,1}, D_{w,2})$, and decrypt out $id||P_w = D_{w,1} \oplus \mathbf{H}(\mathbf{F}_{k_2}(w), D_{w,2})$;
  3. Add id to $\mathcal{I}$, set $L_w = P_w$, and go to step 1).

Fig. 3: Our basic DSSE scheme D-I.

protocol **Setup**. This value will be used in protocol **AddKeyword** to generate a new K1 ciphertext of keyword w. Specifically, this value is taken as the label of this new ciphertext. Hence, it can be connected to the end of the corresponding chain.

When receiving a keyword search trapdoor $ST_w = (\mathbf{F}_{k_1}(w), \mathbf{F}_{k_2}(w))$, protocol **Search** matches value $\mathbf{F}_{k_1}(w)$ with all K1 ciphertexts' labels to find out the hidden chain head of keyword w, and applies value $\mathbf{F}_{k_2}(w)$ to decrypting out a file identifier and a pointer value. The file identifier corresponds to a file containing the queried keyword, and the pointer value guides the server to find out the next matching ciphertext. So on and so forth, all file identifiers related to the queried keyword can be found.

5 Our Basic DSSE Scheme D-II

Our basic DSSE scheme D-II only consists of protocols **Setup**, **DeleteFile** and **Search**. Given a database DB, protocol **Setup** applies the same idea as the first basic DSSE scheme to generating the K1 and K2 ciphertexts. And it generates the hidden chains respectively to connect all K1 ciphertexts of the same keyword

★ Protocol **Setup**$((k, DB), NULL)$:
- Client: Take k and DB as inputs, randomly choose two k-bit secret keys $\mathcal{K} = (k_1, k_2)$, initialize two empty lists $\mathcal{T}_W$ and $\mathcal{T}_F$, and do the following steps:
  1. For each file identifier id or keyword w in DB, initialize pointer parameter $P_{id} = NULL$ or $P_w = NULL$;
  2. For each file-keyword pair $(id, w) \in DB$
     (a) If $P_w = NULL$, set label $L_w = \mathbf{F}_{k_1}(w)$, otherwise set $L_w = P_w$; Set $\{R, P_w\} \xleftarrow{\$} \{0,1\}^{2k}$;
     (b) Generate a K1 ciphertext $(L_w, D_w = (D_{w,1} = (\mathbf{H}(\mathbf{F}_{k_2}(w), R) \oplus (0||id||P_w)), D_{w,2} = R))$, and add this ciphertext into $\mathcal{T}_W$ in the lexicon order;
     (c) If $P_{id} = NULL$, set label $L_{id} = \mathbf{F}_{k_1}(id)$, otherwise set $L_{id} = P_{id}$; Set $\{R, P_{id}\} \xleftarrow{\$} \{0,1\}^{2k}$;
     (d) Generate a K2 ciphertext $(L_{id}, D_{id} = (D_{id,1} = (\mathbf{G}(\mathbf{F}_{k_2}(id), R) \oplus (L_w||P_{id})), D_{id,2} = R))$, and add this ciphertext to $\mathcal{T}_F$ in the lexicon order;
  3. Generate dictionaries $\mathcal{D}_W \leftarrow \mathbf{Creat}(\mathcal{T}_W)$ and $\mathcal{D}_F \leftarrow \mathbf{Creat}(\mathcal{T}_F)$;
  4. Keep the privacy of $\mathcal{K}$, and send the encrypted database $EDB = (\mathcal{D}_W, \mathcal{D}_F)$ to the server;
- Server: Store EDB.

★ Protocol **DeleteFile**$((\mathcal{K}, id), EDB)$:
- Client: Take $\mathcal{K} = (k_1, k_2)$ and a file identifier id as inputs, generate and send a delete token $DT_{id} = (\mathbf{F}_{k_1}(id), \mathbf{F}_{k_2}(id))$ to the server.
- Server: Take $EDB = (\mathcal{D}_W, \mathcal{D}_F)$ and $DT_{id} = (\mathbf{F}_{k_1}(id), \mathbf{F}_{k_2}(id))$ as inputs, set $L_{id} = \mathbf{F}_{k_1}(id)$, and do the following steps:
  1. Retrieve data $D_{id} \leftarrow \mathbf{Get}(\mathcal{D}_F, L_{id})$ according to L_{id}; If $D_{id} = NULL$, return $\perp$ and abort;
  2. Parse $D_{id} = (D_{id,1}, D_{id,2})$, decrypt out $L_w||P_{id} = D_{id,1} \oplus \mathbf{G}(\mathbf{F}_{k_2}(id), D_{id,2})$, and run algorithm $\mathbf{Remove}(\mathcal{D}_F, L_{id})$;
  3. Retrieve data $D_w \leftarrow \mathbf{Get}(\mathcal{D}_W, L_w)$ according to the decrypted L_w, parse $D_w = (D_{w,1}, D_{w,2})$, set the tag bit of D_w to be "1" by computing $D_{w,1} = D_{w,1} \oplus (1||0^{2k})$, run algorithms $\mathbf{Update}(\mathcal{D}_W, (L_w, D_w = (D_{w,1}, D_{w,2})))$ and $\mathbf{Remove}(\mathcal{D}_F, L_{id})$, and set $L_{id} = P_{id}$, and go to step 1).

★ Protocol **Search**$((\mathcal{K}, w), EDB)$:
- Client: Take $\mathcal{K} = (k_1, k_2)$ and a keyword w as inputs, generate and send a search token $ST_w = (\mathbf{F}_{k_1}(w), \mathbf{F}_{k_2}(w))$ to the server.
- Server: Take $EDB = (\mathcal{D}_W, \mathcal{D}_F)$ and $ST_w = (\mathbf{F}_{k_1}(w), \mathbf{F}_{k_2}(w))$ as inputs, initialize an empty set $\mathcal{I}$, a temporary label-data pair $(L_w^t = NULL, D_w^t = NULL)$ and a temporary pointer $P_w^t = NULL$, set $L_w = \mathbf{F}_{k_1}(w)$, and do the following steps:
  1. Retrieve data $D_w \leftarrow \mathbf{Get}(\mathcal{D}_W, L_w)$ according to L_w; If $D_w = NULL$, return $\mathcal{I}$ and abort;
  2. Parse $D_w = (D_{w,1}, D_{w,2})$, and decrypt out $T||id||P_w = D_{w,1} \oplus \mathbf{H}(\mathbf{F}_{k_2}(w), D_{w,2})$, where T denotes the tag bit of D_w;
  3. If $L_w^t = NULL$, set $L_w^t = L_w$, $D_w^t = D_w$, $P_w^t = P_w$ and $L_w = P_w$, and go to step 1);
  4. If $T = 1$, parse $D_w^t = (D_{w,1}^t, D_{w,2}^t)$, update $D_{w,1}^t = D_{w,1}^t \oplus (0^{k+1}||(P_w^t \oplus P_w))$, and run algorithms $\mathbf{Update}(\mathcal{D}_W, (L_w^t, D_w^t = (D_{w,1}^t, D_{w,2}^t)))$ and $\mathbf{Remove}(\mathcal{D}_W, L_w)$;
  5. If $T = 0$, add the decrypted file identifier id to $\mathcal{I}$, and set $L_w^t = L_w$, $D_w^t = D_w$ and $P_w^t = P_w$;
  6. Set $L_w = P_w$, and go to step 1).

Fig. 4: Our basic DSSE scheme D-II.

and all K2 ciphertexts of the same file identifier. It is worth noting that each K1 ciphertext in this scheme contains a tag bit with the initial value "0". When a tag bit is equal to "1", it means that the corresponding K1 ciphertext is logically deleted. To delete all searchable ciphertexts (including the K1 and K2

ciphertexts) of a file, protocol **DeleteFile** shows the physical deletion of all related K2 ciphertexts and the logical deletion of all related K1 ciphertexts. With a keyword search trapdoor, protocol **Search** not only shows how to quickly find out the related file identifiers as the first basic DSSE scheme does, but also shows how to physically delete the K1 ciphertexts that contain the queried keyword and have their tag bits equaling to "1". Let $\mathbf{F} : \{0,1\}^k \times \{0,1\}^* \rightarrow \{0,1\}^k$ be a key-based pseudo-random function. Let $\mathbf{H} : \{0,1\}^* \rightarrow \{0,1\}^{2k+1}$ and $\mathbf{G} : \{0,1\}^* \rightarrow \{0,1\}^{2k}$ be two cryptographic hash functions. The basic scheme D-II is shown in Figure 4.

In protocol **DeleteFile**, a delete token $DT_{id} = (\mathbf{F}_{k_1}(id), \mathbf{F}_{k_2}(id))$ allows the server to match value $\mathbf{F}_{k_1}(id)$ with all K2 ciphertexts' labels and find out a matching ciphertext. Then the server applies value $\mathbf{F}_{k_2}(id)$ to decrypting the matching ciphertext and gets a label L_w and a pointer P_{id}. The label L_w corresponds to a K1 ciphertext of id, and the server sets the tag bit of the K1 ciphertext to be "1". In addition, the pointer P_{id} guides the server to quickly find out the next matching K2 ciphertext. So on and so forth, all K1 and K2 ciphertexts of id can be found, and the server physically deletes the found K2 ciphertexts and logically deletes the found K1 ciphertexts. The logically deleted K1 ciphertexts will be physically deleted by protocol **Search**. When searching a keyword, protocol **Search** can quickly find out all matching K1 ciphertexts as the first basic DSSE scheme does. If the tag bit of a matching K1 ciphertext is equal to "1", the server physically deletes this ciphertext, and repairs the corresponding chain relationship among the remaining K1 ciphertexts.

6 Our Complete DSSE Scheme

The above two basic DSSE schemes respectively show our following main ideas: (1) Constructing the hidden chains to connect all searchable ciphertexts of the same keyword to accelerate the search performance; (2) Saving the storage complexity by dynamically generating chain heads; (3) Adding the new generated searchable ciphertext at the end of the corresponding chain; (4) Employing both logical and physical deletions to delete the expected searchable ciphertexts, specially, running physical deletion in due course.

Note that the above last two main ideas are used to avoid extra information leakage. In this section, we extend the above two basic schemes to construct our complete DSSE scheme. It consists of protocols **Setup**, **AddKeyword**, **DeleteFile**, **DeleteKeyword** and **Search**. Since **DeleteKeyword** is newly achieved by this scheme, it makes other protocols having some differences compared with the above two basic DSSE schemes. But this scheme has exactly the same protocol **Search** as the basic scheme D-II. Hence, this protocol will not be shown in this section. Let $\mathbf{F} : \{0,1\}^k \times \{0,1\}^* \rightarrow \{0,1\}^k$ be a key-based pseudo-random function. Let $\mathbf{H} : \{0,1\}^* \rightarrow \{0,1\}^{2k+1}$ and $\mathbf{G} : \{0,1\}^* \rightarrow \{0,1\}^{3k+1}$ be two cryptographic hash functions. Our complete DSSE scheme is shown in Figures 5 and 6.

★ Protocol **Setup**$((k, DB), NULL)$:
- Client: Take a security parameter k and a database DB as inputs, randomly choose two k-bit secret keys $\mathcal{K} = (k_1, k_2)$, initialize four empty lists $\mathcal{T}_P$, $\mathcal{T}_W$, $\mathcal{T}_F$ and $\mathcal{T}_{F,W}$, and do the following steps:
  1. For each file identifier id or keyword w in DB, initialize pointer parameter $P_{id} = NULL$ or $P_w = NULL$;
  2. For each keyword w in DB and each $id \in DB(w)$
     (a) If $P_w = NULL$, set label $L_w = \mathbf{F}_{k_1}(w)$, otherwise set $L_w = P_w$; Set $\{R, P_w\} \xleftarrow{\$} \{0,1\}^{2k}$;
     (b) Generate a K1 ciphertext $(L_w, D_w = (D_{w,1} = (\mathbf{H}(\mathbf{F}_{k_2}(w), R) \oplus (0||id||P_w)), D_{w,2} = R))$, and add this ciphertext to $\mathcal{T}_W$ in the lexicon order; Set label $L_{id,w} = \mathbf{F}_{k_1}(id, w)$;
     (c) If $P_{id} = NULL$, set label $L_{id} = \mathbf{F}_{k_1}(id)$, otherwise set $L_{id} = P_{id}$; Set $\{R, P_{id}\} \xleftarrow{\$} \{0,1\}^{2k}$;
     (d) Generate a K2 ciphertext $(L_{id}, D_{id} = (D_{id,1} = (\mathbf{G}(\mathbf{F}_{k_2}(id), R) \oplus (0||L_w||L_{id,w}||P_{id})), D_{id,2} = R))$, and add this ciphertext to $\mathcal{T}_F$ in the lexicon order; Set $R \xleftarrow{\$} \{0,1\}^k$;
     (e) Generate a K3 ciphertext $(L_{id,w}, D_{id,w} = (D_{id,w,1} = (\mathbf{H}(\mathbf{F}_{k_2}(id, w), R) \oplus (0||L_w||L_{id})), D_{id,w,2} = R))$, and add this ciphertext to $\mathcal{T}_{F,W}$ in the lexicon order;
  3. For each keyword w in DB, add tuple (w, P_w) to $\mathcal{T}_P$; For each file identifier id in DB, add tuple (id, P_{id}) to $\mathcal{T}_P$; Generate dictionaries $\mathcal{D}_P \leftarrow \mathbf{Creat}(\mathcal{T}_P)$, $\mathcal{D}_W \leftarrow \mathbf{Creat}(\mathcal{T}_W)$, $\mathcal{D}_F \leftarrow \mathbf{Creat}(\mathcal{T}_F)$ and $\mathcal{D}_{F,W} \leftarrow \mathbf{Creat}(\mathcal{T}_{F,W})$;
  4. Keep $\mathcal{K}$ and $\mathcal{D}_P$ secret, and send the encrypted database $EDB = (\mathcal{D}_W, \mathcal{D}_F, \mathcal{D}_{F,W})$ to the server;
- Server: Store EDB.

★ Protocol **AddKeyword**$((\mathcal{K}, \mathcal{D}_P, id, w), EDB)$:
- Client: Take $\mathcal{K} = (k_1, k_2)$, $\mathcal{D}_P$ and a file-keyword pair (id, w) as inputs, retrieve $P_w \leftarrow \mathbf{Get}(\mathcal{D}_P, w)$ according to w, and do the following steps:
  1. If $P_w = NULL$, set label $L_w = \mathbf{F}_{k_1}(w)$, otherwise set $L_w = P_w$; Set $\{R, P_w\} \xleftarrow{\$} \{0,1\}^{2k}$;
  2. Generate a K1 ciphertext $(L_w, D_w = (D_{w,1} = (\mathbf{H}(\mathbf{F}_{k_2}(w), R) \oplus (0||id||P_w)), D_{w,2} = R))$, and run algorithm $\mathbf{Update}(\mathcal{D}_P, (w, P_w))$; Set label $L_{id,w} = \mathbf{F}_{k_1}(id, w)$; Retrieve $P_{id} \leftarrow \mathbf{Get}(\mathcal{D}_P, id)$ according to id;
  3. If $P_{id} = NULL$, set label $L_{id} = \mathbf{F}_{k_1}(id)$, otherwise set $L_{id} = P_{id}$; Set $\{R, P_{id}\} \xleftarrow{\$} \{0,1\}^{2k}$;
  4. Generate a K2 ciphertext $(L_{id}, D_{id} = (D_{id,1} = (\mathbf{G}(\mathbf{F}_{k_2}(id), R) \oplus (0||L_w||L_{id,w}||P_{id})), D_{id,2} = R))$, and run algorithm $\mathbf{Update}(\mathcal{D}_P, (id, P_{id}))$; Set $R \xleftarrow{\$} \{0,1\}^k$;
  5. Generate a K3 ciphertext $(L_{id,w}, D_{id,w} = (D_{id,w,1} = (\mathbf{H}(\mathbf{F}_{k_2}(id, w), R) \oplus (0||L_w||L_{id})), D_{id,w,2} = R))$;
  6. Send ciphertexts $(L_w, D_w, L_{id}, D_{id}, L_{id,w}, D_{id,w})$ to the server;
- Server: Take $EDB = (\mathcal{D}_W, \mathcal{D}_F, \mathcal{D}_{F,W})$ and $(L_w, D_w, L_{id}, D_{id}, L_{id,w}, D_{id,w})$ as inputs, run algorithms $\mathbf{Update}(\mathcal{D}_W, (L_w, D_w))$, $\mathbf{Update}(\mathcal{D}_F, (L_{id}, D_{id}))$ and $\mathbf{Update}(\mathcal{D}_{F,W}, (L_{id,w}, D_{id,w}))$.

Fig. 5: Our complete DSSE scheme (Part I).

In this scheme, protocol **Setup** newly achieves generation of the K3 ciphertexts. In other words, protocol **Setup** generates three kinds of searchable ciphertexts, i.e. K1, K2 and K3, for each file-keyword pair (id, w). Figure 1 shows the generated K1, K2 and K3 ciphertexts of a file-keyword pair, and their hidden relationship.

The K3 ciphertexts are used to realize protocol **DeleteKeyword**. When deleting all searchable ciphertexts of a file-keyword pair (id, w), the generated delete token $DT_{id,w} = (\mathbf{F}_{k_1}(id, w), \mathbf{F}_{k_2}(id, w))$ of protocol **DeleteKeyword**

★ Protocol **DeleteFile**$((\mathcal{K}, id), EDB)$:
- Client: Take $\mathcal{K} = (k_1, k_2)$ and a file identifier id as inputs, generate and send a **delete** token $DT_{id} = (\mathbf{F}_{k_1}(id), \mathbf{F}_{k_2}(id))$ to the server.
- Server: Take $EDB = (\mathcal{D}_W, \mathcal{D}_F, \mathcal{D}_{F,W})$ and $DT_{id} = (\mathbf{F}_{k_1}(id), \mathbf{F}_{k_2}(id))$ as inputs, set label $L_{id} = \mathbf{F}_{k_1}(id)$, and do the following steps:
  1. Retrieve data $D_{id} \leftarrow \mathbf{Get}(\mathcal{D}_F, L_{id})$ according to L_{id}; If $D_{id} = NULL$, return $\perp$ and abort;
  2. Parse $D_{id} = (D_{id,1}, D_{id,2})$, and decrypt out $T||L_w||L_{id,w}||P_{id} = D_{id,1} \oplus \mathbf{G}(\mathbf{F}_{k_2}(id), D_{id,2})$, and run algorithm $\mathbf{Remove}(\mathcal{D}_F, L_{id})$, where T denotes the tag bit of D_{id};
  3. If $T = 0$, retrieve $D_w \leftarrow \mathbf{Get}(\mathcal{D}_W, L_w)$ according to the decrypted L_w, parse $D_w = (D_{w,1}, D_{w,2})$, set the tag bit of D_w to be "1" by computing $D_{w,1} = D_{w,1} \oplus (1||0^{2k})$, run algorithms $\mathbf{Update}(\mathcal{D}_W, (L_w, D_w = (D_{w,1}, D_{w,2})))$ and $\mathbf{Remove}(\mathcal{D}_{F,W}, L_{id,w})$; Set $L_{id} = P_{id}$, and go to step 1).

★ Protocol **DeleteKeyword**$((\mathcal{K}, id, w), EDB)$:
- Client: Take secret keys $\mathcal{K} = (k_1, k_2)$ and a file-keyword pair (id, w) as inputs, generate and send a **delete** token $DT_{id,w} = (\mathbf{F}_{k_1}(id, w), \mathbf{F}_{k_2}(id, w))$ to the server;
- Server: Take $EDB = (\mathcal{D}_W, \mathcal{D}_F, \mathcal{D}_{F,W})$ and $DT_{id,w} = (\mathbf{F}_{k_1}(id, w), \mathbf{F}_{k_2}(id, w))$ as inputs, set $L_{id,w} = \mathbf{F}_{k_1}(id, w)$ and do the following steps:
  1. Retrieve data $D_{id,w} \leftarrow \mathbf{Get}(\mathcal{D}_{F,W}, L_{id,w})$ according to $L_{id,w}$; If $D_{id,w} = NULL$, return $\perp$ and abort;
  2. Parse $D_{id,w} = (D_{id,w,1}, D_{id,w,2})$, decrypt out $T||L_w||L_{id} = D_{id,w,1} \oplus \mathbf{H}(\mathbf{F}_{k_2}(id, w), D_{id,w,2})$, and run algorithm $\mathbf{Remove}(\mathcal{D}_{F,W}, L_{id,w})$;
  3. Retrieve data $D_w \leftarrow \mathbf{Get}(\mathcal{D}_W, L_w)$ according to the decrypted label L_w, parse $D_w = (D_{w,1}, D_{w,2})$, set the tag bit of D_w to be "1" by computing $D_{w,1} = D_{w,1} \oplus 1||0^{2k}$, and run algorithm $\mathbf{Update}(\mathcal{D}_W, (L_w, D_w = (D_{w,1}, D_{w,2})))$;
  4. Retrieve data $D_{id} \leftarrow \mathbf{Get}(\mathcal{D}_F, L_{id})$ according to the decrypted label L_{id}, parse $D_{id} = (D_{id,1}, D_{id,2})$, set the tag bit of D_w to be "1" by computing $D_{id,1} = D_{id,1} \oplus 1||0^{3k}$, and run algorithm $\mathbf{Update}(\mathcal{D}_F, (L_{id}, D_{id} = (D_{id,1}, D_{id,2})))$.

★ Protocol **Search**$((\mathcal{K}, w), EDB)$ is same as the basic scheme D-II.

Fig. 6: Our complete DSSE scheme (Part II).

allows the server to find out the matching K3 ciphertext by matching value $\mathbf{F}_{k_1}(id, w)$ with all K3 ciphertexts' labels, and then the server applies value $\mathbf{F}_{k_2}(id, w)$ to decrypting the matching K3 ciphertext, and gets labels L_w and L_{id}. These two labels respectively correspond to the K1 and K2 ciphertexts of the file-keyword pair (id, w). Finally, the server physically deletes the matching K3 ciphertext, and logically deletes the corresponding K1 and K2 ciphertexts by setting their tag bits to be "1".

It is different with the second basic DSSE scheme that some K2 ciphertexts could have been logically deleted before the execution of protocol **DeleteFile**. Hence, protocol **DeleteFile** in this scheme has two ways to delete K2 ciphertexts. For example, suppose to delete the K2 ciphertext of label L_{id} in protocol **DeleteFile**, the sever decrypts this ciphertext to obtain a tag bit and two labels L_w and $L_{id,w}$. If the tag bit is equal to "0", the server finds out the corresponding K1 and K3 ciphertexts according to those two labels, then physically deletes the K2 and K3 ciphertexts and logically deletes the K1 ciphertext. Otherwise, the server only physically deletes the k2 ciphertext, since the corresponding K1 and

K3 ciphertexts have been logically or physically deleted by a previous execution of protocol **DeleteKeyword**.

In addition, protocol **AddKeyword** has the same essence with protocol **Setup** to generate searchable ciphertexts. But protocol **AddKeyword** takes only one file-keyword pair as input. Contrarily, protocol **Setup** takes a lot of file-keyword pairs as inputs.

Summarily, our complete DSSE scheme shows all of our main ideas that were introduced in Subsection 1.1. The most contributive and novel work in our scheme should be the hybrid of logical and physical deletion of searchable ciphertexts, so that the deletion function only causes small information leakage compared with schemes KPR'12, KP'13 and CJJ'14.

6.1 Provable IND-CKA2 Security

According to the IND-CKA2 security definition, the security proof of our complete DSSE scheme requires us to construct a simulator $\mathcal{S}$. This simulator only takes leakage functions as inputs, and simulates our scheme by responding the following requirements of adversary $\mathcal{A}$. When $\mathcal{A}$ chooses a database DB to engage in protocol **Setup**, $\mathcal{S}$ takes leakage function $\mathcal{L}_{Setup}$ as input, and simulates an encrypted database EDB. When $\mathcal{A}$ chooses a new file-keyword pair to engage in protocol **AddKeyword**, $\mathcal{S}$ takes leakage function $\mathcal{L}_{AddKeyword}$ as input, and simulates the corresponding searchable ciphertexts. When $\mathcal{A}$ chooses a file to engage in protocol **DeleteFile**, $\mathcal{S}$ takes leakage function $\mathcal{L}_{DeleteFile}$ as input, and simulates the corresponding delete token. When $\mathcal{A}$ chooses an old file-keyword pair to engage in protocol **DeleteKeyword**, $\mathcal{S}$ takes leakage function $\mathcal{L}_{DeleteKeyword}$ as input, and simulates the corresponding delete token. When $\mathcal{A}$ chooses a keyword to engage in protocol **Search**, $\mathcal{S}$ takes leakage function $\mathcal{L}_{Search}$ as input, and simulates the corresponding search token.

All the above simulated data will be sent to $\mathcal{A}$. Moreover they must be indistinguishable with real ones in the view of $\mathcal{A}$. To meet the above requirements, we have to assume that the hash functions and the pseudo-random function in our scheme are random oracles. $\mathcal{S}$ controls the responses of these oracles, and makes the above forgeries indistinguishable with the real ones in the view of $\mathcal{A}$.

We define the leakage functions for all proposed protocols. When defining the leakage functions, the most complex work is to define the leakage caused by the linkage of some protocols' instances. In this paper, we apply a new idea to define the leakage functions. This idea makes the definitions more clear. For all ciphertexts generated by protocol **Setup**, let $Old(id, w)$ denote the set of ciphertexts that were generated for file-keyword pair (id, w), let $Old(id)$ denote the set of ciphertexts that were generated for file id, and let $Old(w)$ denote the set of ciphertexts that were generated for keyword w. For all ciphertexts generated by protocol **AddKeyword**, let $New(id, w)$ denote the set of ciphertexts that were generated for file-keyword pair (id, w), let $New(id)$ denote the set of ciphertexts that were generated for file id, and let $New(w)$ denote the set of ciphertexts that were generated for keyword w. The leakage functions are $\mathcal{L}_{Setup} = |DB|$,

$\mathcal{L}_{AddKeyword} = New(id, w)$, $\mathcal{L}_{DeleteFile} = (Old(id), New(id))$, $\mathcal{L}_{DeleteKeyword} = (Old(id, w), New(id, w))$ and $\mathcal{L}_{Search} = (DB(w), Old(w), New(w))$ respectively.

It is clear that most of above leakage functions are defined as the set of some related ciphertexts. This method is easy to imply the leakage caused by the linkage of some protocols' instances. For example, when running protocol **AddKeyword** to add a file-keyword pair (id, w), one can decide that whether keyword w has been searched by a previous instance of protocol **Search**. This leakage is contained in our definitions. In other words, if the leakage $\mathcal{L}_{AddKeyword} = New(id, w)$ of an instance of protocol **AddKeyword** has some common ciphertexts with the leakage $\mathcal{L}_{Search} = (DB(w), Old(w), New(w))$ of an instance of protocol **Search**, it means that keyword w has been searched by the latter instance. In addition, this example also allows one to decide that whether file-keyword pair (id, w) has been deleted by a previous instance of protocol **DeleteFile** or **DeleteKeyword**. This leakage is also contained in our definitions by the similar reason. In general, given two instances of our proposed protocols, the leakage caused by their linkage is implied in our definitions. Finally, we have Theorem 1 whose proof can be found in the full version.

Theorem 1. *Suppose hash functions* $\mathbf{H}$ *and* $\mathbf{G}$ *and key-based pseudo-random function* $\mathbf{F}_{k_1}$ *are respectively modeled as three random oracles. Our complete DSSE scheme is IND-CKA2 secure with leakage functions* ($\mathcal{L}_{Setup}$, $\mathcal{L}_{AddKeyword}$, $\mathcal{L}_{DeleteFile}$, $\mathcal{L}_{DeleteKeyword}$, $\mathcal{L}_{Search}$) *in the RO model, where* $\mathcal{L}_{Setup} = |DB|$, $\mathcal{L}_{AddKeyword} = New(id, w)$, $\mathcal{L}_{DeleteFile} = (Old(id), New(id))$, $\mathcal{L}_{DeleteKeyword} = (Old(id, w), New(id, w))$ *and* $\mathcal{L}_{Search} = (DB(w), Old(w), New(w))\}$.

7 Comparisons and Experiments

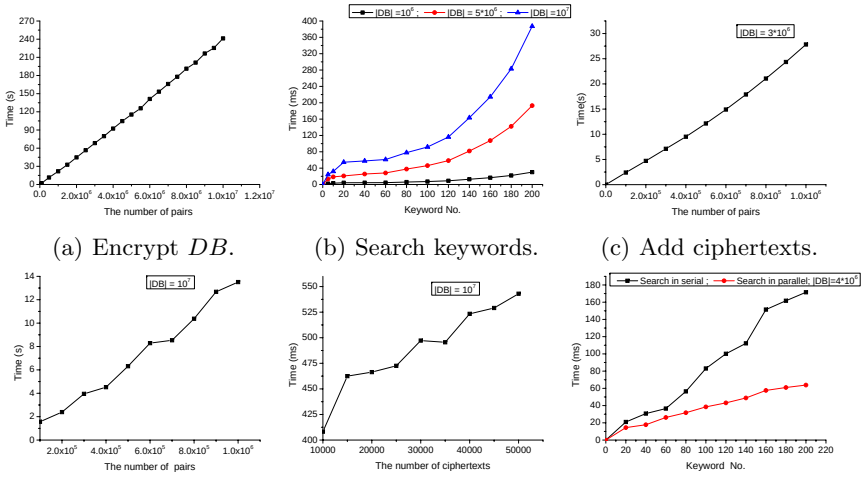
Table 2: Exact comparisons.

Scheme	**Search** Complexity	Storage Complexity	Leakage Functions				
			$\mathcal{L}_{Setup}$	$\mathcal{L}_{Search}$	$\mathcal{L}_{AddKeyword}$	$\mathcal{L}_{DeleteFile}$	$\mathcal{L}_{DeleteKeyword}$
KPR'12[1]	$O(\lvert DB(w)\rvert)$	$O(\lvert DB\rvert + \lvert\mathcal{W}\rvert)$	$\lvert\mathcal{W}\rvert$, $\lvert DB\rvert$	①	×	③	×
KP'13[2]	$O(\lvert DB(w)\rvert \cdot log\ \lvert\mathcal{ID}\rvert)$	$O(\lvert\mathcal{W}\rvert \cdot \lvert\mathcal{ID}\rvert)$	$\lvert\mathcal{W}\rvert \cdot \lvert\mathcal{ID}\rvert$	①	×	④	×
CJJ'14[3]	$O(\lvert DB(w)\rvert)$	$O(\lvert DB\rvert)$	$\lvert DB\rvert$	①	②	×	⑤ at the worst case
Ours	$O(\lvert DB(w)\rvert)$	$O(\lvert DB\rvert)$	$\lvert DB\rvert$	①	②	④	⑤

①: $DB(w)$, $Old(w)$ and $New(w)$. ②: $New(id, w)$. ③: $Old(id)$, $New(id)$ and a part of $DB(w)$ where $w \in DB(id)$. ④: $Old(id)$ and $New(id)$. ⑤: $Old(id, w)$ and $New(id, w)$. ×: means that the operation cannot be achieved. All symbols have been defined in Section 2 and Subsection 6.1.

In this section, we make thorough comparisons between our complete DSSE scheme and the related schemes KPR'12, KP'13 and CJJ'14 in Table 2. We also conduct extensive experiments to evaluate the performance of our scheme. Table 2 shows the following advantages of our scheme: (1) the lowest search complexity, which is linear with the total number of files containing the queried keyword; (2) the lowest storage complexity, which is linear with the size of database DB (or the total number of file-keyword pairs in database DB); (3) the smallest leakage

to run protocol **Setup**, which only contains the size of database DB; (4) the same leakage with schemes KPR'12, KP'13 and CJJ'14 to run protocol **Search**; (5) the same leakage with scheme CJJ'14 to run protocol **AddKeyword**; (6) the smaller leakage to run protocol **DeleteFile** than that of KPR'12; (7) the same leakage (at the worst case) with scheme CJJ'14 to run protocol **DeleteKeyword**. Hence, our DSSE scheme not only supports all functions mentioned in Table 1, but also has the lowest search and storage complexities, and the smallest leakages in most cases.



(a) Encrypt DB. (b) Search keywords. (c) Add ciphertexts.

(d) Delete ciphertexts. (e) Physical deletion. (f) Parallel search.

Fig. 7: All tests and their results.

We coded our complete DSSE scheme and tested its performance on a simulated database DB of millions file-keyword pairs. The inputted security parameter is of binary size 80 bits. All hash functions and pseudo-random function are implemented by running hash function SHA-256. The practical implementation of a DSSE scheme consists of two kinds of time-intensive operations: cryptographic computations and system actions (e.g., network transmission and file system access). To separate cryptographic costs from system costs, we built a test framework excluding network transmission and disk access. In other words, all data are stored in memory. Most of tests were implemented in a PC with Intel CPU 2.4 GHz (Core i5) and Ubuntu system by a single thread. Only the final test is a parallel search with a 4-core processor.

To simulate a database DB with millions file-keyword pairs, we chose 200 commonly used keywords from Google Search Engine. We supposed that each file has no more than 20 keywords, and each file identifier was randomly generated. We paired each keyword with some file identifiers according to this keyword's

frequency. Finally, all simulated file-keyword pairs were collected in a database DB.

Figure 7(a) shows the time cost of our complete DSSE scheme to generate an encrypted database EDB for the above simulated database DB, where $|DB|$ is between 10^6 and 10^7. This process contains generating searchable ciphertexts and constructing a self-balancing binary search tree (or AVL tree) for these ciphertexts. For example, the time cost to encrypt DB with size 10^7 is about 241 seconds. From this experiment result, we can find that the time cost per file-keyword pair is an amortized value: it was determined by taking the complete execution time of experiment and dividing by the number of file-keyword pairs.

Figure 7(b) shows the time cost of our complete DSSE scheme to search each keyword over different scaled DBs. Recall that our protocol **Search** includes the process to physically delete the K1 ciphertexts with the tag bit equaling "1". We show this process in Figure 7(e). In Figure 7(b), the line with triangles shows the time costs when $|DB| = 10^6$, the line with circles shows the time costs when $|DB| = 5 \times 10^6$, and the line with rectangles shows the time costs when $|DB| = 10^7$.

Figure 7(c) shows the time cost of our complete DSSE scheme to add the searchable ciphertexts of several file-keyword pairs. This process excludes the generation of searchable ciphertexts, since this part is similar with the process to encrypt a database. So it only contains the process to rebalance the AVL tree according to these new added ciphertexts. We have a group of experiments to add the searchable ciphertexts of the different number (between 10^5 and 10^6) of pairs to a database with the original size $|DB| = 3 \times 10^6$.

Figure 7(d) shows the time cost of our complete DSSE scheme to delete the searchable ciphertexts of several file-keyword pairs. We have a group of experiments to delete the searchable ciphertexts of different number (between 10^5 and 10^6) of pairs from a database with the original size $|DB| = 10^7$.

Figure 7(e) shows the time cost of physical deletion when one searches a keyword in our complete DSSE scheme. Recall that all K1 ciphertexts of the same keyword are applied to construction of the hidden chain relationship. Suppose some of them have the tag bit equaling "1". These ciphertexts will be physically deleted when their associated keywords are searching. This process contains to repair a broken chain and rebalance the original AVL tree of the searchable ciphertexts. For different number (between 10^4 and 5×10^4) of ciphertexts with the tag bit equaling "1", we tested the time cost to physically delete them from a database with the original size $|DB| = 10^7$.

Both in Figures 7(d) and 7(e), some singular points indicate that the time cost is not strictly linear with the number of deleted ciphertexts. These singular points are caused by the operations to rebalance the AVL tree of searchable ciphertexts. Comparatively, when different nodes are deleted in an AVL tree, the time costs to rebalance the AVL tree are also different.

Finally, Figure 7(f) shows the time cost of parallel search in our complete DSSE scheme. We simulated four databases with the same number of file-keyword pairs, and generated their encrypted databases. The total size of all databases is

$|DB| = 4 \times 10^6$. The line with circles shows the time cost to search each keyword in parallel with a 4-core processor, and the line with rectangles shows the time cost to search each keyword in the serial mode.

8 Other Related Works

SSE was first introduced by Song *et al.* in [5]. Their instantiated scheme takes search time linear with the total binary size of ciphertexts. A number of efforts [6,7,8] follow this line and refine Song *et al.*'s original work, where [8] is the first one to construct indices for ciphertexts. The SSE scheme due to Curtmola *et al.* [9] has been proven to be semantically secure against an adaptive adversary. It allows search to be processed in logarithmic time, although the keyword search token has length linear with the total number of ciphertexts in the worst case.

In addition to the above efforts devoted to either provable security or better search performance, attention has recently been paid to achieve versatile SSE schemes. Waters *et al.* [10] showed practical applications of SSE and employed it to realize secure and searchable audit logs. Several works in SSE are complex queries for conjunctive [11,12] or disjunctive [13,14] keyword combinations. The works in [9,15] extended SSE to a multi-sender scenario. Recent results [16,17] achieved efficiency improvements for these complex queries. The works in [18,19] supported fuzzy keyword searching. The schemes in [20,21,22,23] solved the problem of multi-keyword ranked search and multi-dimensional range query over encrypted cloud data. Lu [24] improved the search performance of range queries by constructing indices. Boldyreva *et al.* [25] first studied symmetric encryption primitive with order preserving and provided an instance with provable security. Chase *et al.* [26] first studied the searchable symmetric encryption of structured data.

In addition to the previously introduced DSSE schemes in [1,2,3], Naveed *et al.* [27] proposed a DSSE scheme to trade storage for performance by scattering the stored blocks using hashing instead of encrypting the indices. This work leaks keyword frequency like [1]. Emilet *et al.* [4] proposed a hierarchical index structure using oblivious random access memory (RAM) to achieve more secure and effective dynamic ciphertext updates with small leakage. This work is not efficient in practice because of a large number of communication rounds and expensive storage costs on the server side [28]. Hahn *et al.* [29] constructed visible relationships to group all searchable ciphertexts of the same keyword, when keywords are at the first time to be searched. According to those groups, the performance to repeatedly search the same keyword will be significantly improved. This method can also be applied in KPR'12, KP'13 and CJJ'14 to accelerate their performance to repeatedly search the same keyword. But the first-time search performance per keyword of [29] is linear with the total number of ciphertexts. Hence, its search complexity is the highest one comparably.

9 Conclusion

In this paper, we proposed a new DSSE scheme to simultaneously support fast keyword search, low storage complexity, versatile functions and small leakage when implementing these functions. Our scheme has the search complexity linear with the number of searchable ciphertexts containing the queried keyword, and has the storage complexity linear with the size of original database. Compared with previous works, our scheme has the most versatile functions. In addition to search a keyword, it allows a client to (1) add the searchable ciphertexts of a new file-keyword pair, (2) delete the searchable ciphertexts of a file, and (3) delete the searchable ciphertexts of a file-keyword pair. In most cases, our scheme has the smallest leakage compared with previous works. Furthermore, our scheme is proven IND-CKA2 secure, which excludes possible vulnerabilities in design. The most contributive and novel work in our scheme is to achieve physical deletion with small leakage.

Acknowledgement

The first author is partly supported by the National Natural Science Foundation of China under grant no. 61472156 and the National Program on Key Basic Research Project (973 Program) under grant no. 2014CB340600.

References

1. Kamara S., Papamanthou C., Roeder T.: Dynamic searchable symmetric encryption. In: ACM CCS 2012, pp. 965-976, ACM (2012)
2. Kamara S., Papamanthou C.: Parallel and Dynamic Searchable Symmetric Encryption. In: Sadeghi, A. R. (ed.), FC 2013, LNCS Vol. 7859, pp. 258-274. Springer, Heidelberg (2013)
3. Cash D., Jaeger J., Jarecki S., Jutla C., Krawczyk H., Ros M. C., Steiner M.: Dynamic Searchable Encryption in Very-Large Databases: Data Structures and Implementation. In: NDSS 2014.
4. Stefanov E., Papamanthou C., Shi, E.: Practical dynamic searchable encryption with small leakage. In: NDSS 2014.
5. Song D., Wagner D., Perrig A.: Practical techniques for searches on encrypted data. In: IEEE SP 2000, pp. 44-55, IEEE (2000)
6. Agrawal R., Kiernan J., Srikant R., Xu Y.: Order Preserving Encryption for Numeric Data. In: Proceedings of the 2004 ACM SIGMOD international conference on Management of data, pp. 563-574, ACM (2004)
7. Chang Y. C., Mitzenmacher M.: Privacy Preserving Keyword Searches on Remote Encrypted Data. In: Ioannidis J., Keromytis A., Yung M. (eds.), ACNS 2005, LNCS Vol. 3531, pp. 442-455. Springer, Heidelberg (2005)
8. Goh E. J.: Secure Indexes. Cryptography ePrint Archive, Report 2003/216 (2003).
9. Curtmola R., Garay J., Kamara S., Ostrovsky R.: Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions. In: ACM CCS 2006, pp. 79-88, ACM (2006)

10. Waters B. R., Balfanz D., Durfee G., Smetters D. K.: Building an Encrypted and Searchable Audit Log. In: NDSS 2004, vol. 4, pp. 5-6 (2004)
11. Golle P., Staddon J., Waters B.: Secure conjunctive keyword search over encrypted data. In: Jakobsson M., Yung M., Zhou J. (eds.), ACNS 2004, LNCS Vol. 3089, pp. 31-45, Springer, Heidelberg (2004)
12. Byun J. W., Lee D. H., Lim J. I.: Efficient conjunctive keyword search on encrypted data storage system. In: Atzeni A. S., Lioy A. (eds.), PKI 2006, LNCS Vol. 4043, pp. 184-196, Springer, Heidelberg (2006)
13. Boneh D., Waters B.: Conjunctive, subset, and range queries on encrypted data. In: Salil S. P. (ed.), TCC 2007, LNCS Vol. 4392, pp. 535-554, Springer, Heidelberg (2007)
14. Li M., Yu S., Cao N.: Authorized private keyword search over encrypted data in cloud computing. In: IEEE ISDCS 2011, pp. 383-392, IEEE (2011)
15. Jarecki S., Jutla C., Krawczyk H., Rosu M. C., Steiner M.: Outsourced symmetric private information retrieval. In: ACM CCS 2013, pp. 875-888, ACM (2013)
16. Katz J., Sahai A., Waters B.: Predicate encryption supporting disjunctions, polynomial equations, and inner products. In: Nigel S. (ed.), EUROCRYPT 2008, LNCS Vol. 4965, pp. 146-162, Springer, Heidelberg (2008)
17. Cash D., Jarecki S., Jutla C.: Highly-scalable searchable symmetric encryption with support for boolean queries. In: CRYPTO 2013, LNCS Vol. 8042, pp. 353-373, Springer, Heidelberg (2013)
18. Li J., Wang Q., Wang C., Cao N., Ren K., Lou W.: Fuzzy Keyword Search over Encrypted Data in Cloud Computing. In: IEEE INFOCOM 2010, pp. 1-5, IEEE (2010)
19. Wang B., Yu S., Lou W., Hou Y. T.: Privacy-Preserving Multi-Keyword Fuzzy Search over Encrypted Data in the Cloud. In: IEEE INFOCOM 2014, pp. 2112-2120, IEEE (2014)
20. Shi E., Bethencourt J., Chan T. H.: Multi-dimensional range query over encrypted data. In: IEEE SP 2007, pp. 350-364, IEEE (2007)
21. Wang C., Cao N., Li J., Lou,W. J.: Secure ranked keyword search over encrypted cloud data. In: IEEE ICDCS 2010, pp. 253-262, IEEE (2010)
22. Wang C., Cao N., Ren K., Lou W.: Enabling Secure and Efficient Ranked Keyword Search over Outsourced Cloud Data. IEEE Transactions on Parallel and Distributed Systems, vol. 23, no. 8, pp. 1467-1479, IEEE (2012)
23. Cao N., Wang C., Li M., Ren K., Lou W. J.: Privacy-preserving multi-keyword ranked search over encrypted cloud data. IEEE Transactions on Parallel and Distributed Systems, vol. 25, no. 1, pp. 222-233, IEEE (2014)
24. Lu Y.: Privacy-preserving logarithmic-time search on encrypted data in cloud. In: NDSS 2012.
25. Boldyreva A., Chenette N., Lee Y., O'Neill A.: Order-Preserving Symmetric Encryption. In: Joux A. (ed.), EUROCRYPT 2009, LNCS Vol. 5479, pp. 224-241, Springer, Heidelberg (2009)
26. Chase M., Kamara S.: Structured Encryption and Controlled Disclosure. In: Abe M. (ed.), ASIACRYPT 2010, LNCS Vol. 6477, pp. 577-594, Springer, Heidelberg (2010)
27. Naveed M., Prabhakaran M., Gunter C.: Dynamic Searchable Encryption via Blind Storage. In: IEEE SP 2014, pp. 639-654, IEEE (2014)
28. Bosch C., Hartel P., Jonker W., *et al.*:A survey of provably secure searchable encryption. ACM Computing Surveys, vol. 47, no. 2, Article no. 18 (2014)
29. Hahn F., Kerschbaum F.: Searchable Encryption with Secure and Efficient Updates. In: ACM CCS 2014, pp. 310-320, ACM (2014)