Dynamic Searchable Symmetric Encryption with Minimal Leakage and Efficient Updates on Commodity Hardware

Attila A. Yavuz[1(✉)] and Jorge Guajardo[2]

[1] The School of Electrical Engineering and Computer Science, Oregon State University, Corvallis, OR 97331, USA
attila.yavuz@oregonstate.edu

[2] Robert Bosch Research and Technology Center, Pittsburgh, PA 15203, USA
Jorge.GuajardoMerchan@us.bosch.com

Abstract. Dynamic Searchable Symmetric Encryption (DSSE) enables a client to perform keyword queries and update operations on the encrypted file collections. DSSE has several important applications such as privacy-preserving data outsourcing for computing clouds. In this paper, we developed a new DSSE scheme that achieves the highest privacy among all compared alternatives with low information leakage, efficient updates, compact client storage, low server storage for large file-keyword pairs with an easy design and implementation. Our scheme achieves these desirable properties with a very simple data structure (i.e., a bit matrix supported with two hash tables) that enables efficient yet secure search/update operations on it. We prove that our scheme is secure and showed that it is practical with large number of file-keyword pairs even with an implementation on simple hardware configurations.

Keywords: Dynamic Searchable Symmetric Encryption · Privacy enhancing technologies · Secure data outsourcing · Secure computing clouds

1 Introduction

Searchable Symmetric Encryption (SSE) [8] enables a client to encrypt data in such a way that she can later perform keyword searches on it via "search tokens" [19]. A prominent application of SSE is to enable privacy-preserving keyword searches on cloud-based systems (e.g., Amazon S3). A client can store a collection of encrypted files at the cloud and yet perform keyword searches without revealing the file or query contents [13]. Desirable properties of a SSE scheme are as follows:

- *Dynamism*: It should permit adding or removing new files/keywords from the encrypted file collection securely after the system set-up.
- *Efficiency and Parallelization*: It should offer fast search/updates, which are parallelizable across multiple processors.

© Springer International Publishing Switzerland 2016
O. Dunkelman and L. Keliher (Eds.): SAC 2015, LNCS 9566, pp. 241–259, 2016.
DOI: 10.1007/978-3-319-31301-6_15

- *Storage Efficiency*: The SSE storage overhead of the server depends on the encrypted data structure (i.e., encrypted index) that enables keyword searches. The number of bits required to represent a file-keyword pair in the encrypted index should be small. The size of encrypted index should not grow with the number of operations. The persistent storage at the client should be minimum.
- *Communication Efficiency*: Non-interactive search/update a with minimum data transmission should be possible to avoid the delays.
- *Security*: The information leakage must be precisely quantified based on formal SSE security notions (e.g., dynamic CKA2 [12]).

Our Contributions. The preliminary SSEs (e.g.,[8,18]) operate on only static data, which strictly limits their applicability. Later, Dynamic Searchable Symmetric Encryption (DSSE) schemes (e.g., [4,13]), which can handle dynamic file collections, have been proposed. To date, there is no single DSSE scheme that outperforms all other alternatives for *all* metrics: privacy (e.g., info leak), performance (e.g., search, update times) and functionality. Having this in mind, we develop a DSSE scheme that achieves the highest privacy among all compared alternatives with low information leakage, non-interactive and efficient updates (compared to [12]), compact client storage (compared to [19]), low server storage for large file-keyword pairs (compared to [4,12,19]) and conceptually simple and easy to implement (compared to [12,13,19]). Table 1 compares our scheme with existing DSSE schemes for various metrics. We outline the desirable properties of our scheme as follows:

- *High Security*: Our scheme achieves a high-level of update security (i.e., *Level-1*), forward-privacy, backward-privacy and size pattern privacy simultaneously (see Sect. 5 for the details). We quantify the information leakage via leakage functions and formally prove that our scheme is dynamic CKA2-secure in random oracle model [3].
- *Compact Client Storage*: Compared to some alternatives with secure updates (e.g., [19]), our scheme achieves smaller client storage (e.g., 10–15 times with similar parameters). This is an important advantage for resource constrained clients such as mobile devices.
- *Compact Server Storage with Secure Updates*: Our encrypted index size is smaller than some alternatives with secure updates (i.e., [12,19]). For instance, our scheme achieves $4 \cdot \kappa$ smaller storage overhead than that of the scheme in [12], which introduces a significant difference in practice. Asymptotically, the scheme in [19] is more server storage efficient for small/moderate number of file-keyword pairs. However, our scheme requires only two bits per file-keyword pair with the maximum number of files and keywords.
- *Constant Update Storage Overhead*: The server storage of our scheme does not grow with update operations, and therefore it does not require re-encrypting the whole encrypted index due to frequent updates. This is more efficient than some alternatives (e.g., [19]), whose server storage grows linearly with the number of file deletions.

- *Dynamic Keyword Universe*: Unlike some alternatives (e.g., [8,12,13]), our scheme does not assume a fixed keyword universe, which permits the addition of new keywords to the system after initialization. Hence, the file content is not restricted to a particular pre-defined keyword but can be any token afterwards (encodings)[1].
- *Efficient, Non-interactive and Oblivious Updates*: Our basic scheme achieves secure updates non-interactively. Even with large file-keyword pairs (e.g., $N = 10^{12}$), it incurs low communication overhead (e.g., 120 KB for $m = 10^6$ keywords and $n = 10^6$ files) by further avoiding network latencies (e.g., 25–100 ms) that affect other interactive schemes (e.g., as considered in [4,12,16,19]). One of the variants that we explore requires three rounds (as in other DSSE schemes), but it still requires low communication overhead (and less transmission than that of [12] and fewer rounds than [16]). Notice that the scheme in [16] can only add or remove a file but cannot update the keywords of a file without removing or adding it, while our scheme can achieve this functionality intrinsically with a (standard) update or delete operation. Finally, our updates take always the same amount of time, which does not leak timing information depending on the update.
- *Parallelization*: Our scheme is parallelizable for both update and search operations.
- *Efficient Forward Privacy*: Our scheme can achieve forward privacy by retrieving not the whole data structure (e.g., [19]) but only some part of it that has already been queried.

2 Related Work

SSE was introduced in [18] and it was followed by several SSE schemes (e.g., [6,8,15]). The scheme of Curtmola et al. in [8] achieves a sub-linear and optimal search time as $O(r)$, where r is the number of files that contain a keyword. It also introduced the security notion for SSE called as *adaptive security against chosen-keyword attacks (CKA2)*. However, the static nature of those schemes limited their applicability to applications with dynamic file collections. Kamara et al. developed a DSSE scheme in [13] that could handle dynamic file collections via encrypted updates. However, it leaked significant information for updates and was not parallelizable. Kamara et al. in [12] proposed a DSSE scheme, which leaked less information than that of [13] and was parallelizable. However, it incurs an impractical server storage. Recently, a series of new DSSE schemes (e.g., [4,16,17,19]) have been proposed by achieving better performance and security. While being asymptotically better, those schemes also have drawbacks. We give a comparison of these schemes (i.e., [4,16,17,19]) with our scheme in Sect. 6.

Blind Seer [17] is a private database management system, which offers private policy enforcement on semi-honest clients, while a recent version [10] can also

[1] We assume the maximum number of keywords to be used in the system is pre-defined.

Table 1. Performance Comparison of DSSE schemes.

Scheme/Property	[13] Kamara 12'	[12] Kamara 13'	[19] Stefanov	[4] $\left(\prod_{\mathsf{2lev}}^{\mathsf{dyn,ro}}\right)$	This work
Size privacy	No	No	No	No	Yes
Update privacy	*L5*	*L4*	*L3*	*L2*	*L1*
Forward privacy	No	No	Yes	Yes	Yes
Backward privacy	No	No	No	No	Yes
Dynamic keyword	No	No	Yes	Yes	Yes
Client storage	4κ	3κ	$\kappa \log(N')$	$\kappa \cdot \mathcal{O}(m')$	$\kappa \cdot \mathcal{O}(n+m)$
Index size (server)	$z \cdot \mathcal{O}(m+n)$	$\mathcal{O}((\kappa+m) \cdot n)$	$13\kappa \cdot \mathcal{O}(N')$	$c''/b \cdot \mathcal{O}(N')$	$2 \cdot \mathcal{O}(m \cdot n)$
Grow with updates	No	No	Yes	Yes	No
Search time	$\mathcal{O}((r/p) \cdot \log n)$	$\mathcal{O}((r/p)$ $\cdot \log^3(N'))$	$\mathcal{O}((r+d_w)/p)$	$1/b \cdot \mathcal{O}(r/p)$	$\mathcal{O}(\frac{m}{p \cdot b})$
# Rounds update	1	3	3	1	1
Update bandwidth	$z \cdot \mathcal{O}(m'')$	$(2z\kappa)\mathcal{O}(m \log n)$	$z \cdot \mathcal{O}(m'' \log N')$	$z \cdot \mathcal{O}(m \log n + m'')$	$b \cdot \mathcal{O}(m)$
Update time	$\mathcal{O}(m'')$	$\mathcal{O}((m/p) \cdot \log n) + t$	$\mathcal{O}((m''/p) \cdot$ $\log^2(N')) + t$	$\mathcal{O}(m''/p) + t$	$b \cdot \mathcal{O}(m/p)$
Parallelizable	No	Yes	Yes	Yes	Yes

• All compared schemes are *dynamic CKA2 secure* in Random Oracle Model (ROM) [3], and leak search and access patterns. The analysis is given for the worst-case (asymptotic) complexity.
• m and n are the maximum # of keywords and files, respectively. m' and n' are the current # of keywords and files, respectively. We denote by $N' = m' \cdot n'$ the total number of keywords and file pairs currently stored in the database. m'' is the # unique keywords included in an updated file (add or delete). r is # of files that contain a specific keyword.
• Rounds refer to the number of messages exchanged between two communicating parties. A *non-interactive* search and an *interactive* update operation require two and three messages to be exchange, respectively. Our main scheme, the scheme in [13] and some variants in [4] also achieve *non-interactive* update with only single message (i.e., an update token and an encrypted file to be added for the file addition) to be send from the client to the server. The scheme in [19] requires a transient client storage as $\mathcal{O}(N'^{\alpha})$.
• κ is the security parameter. p is the # of parallel processors. b is the block size of symmetric encryption scheme. z is the pointer size in bits. t is the network latency introduced due to the interactions. α is a parameter, $0 < \alpha < 1$.
• Update privacy levels *L1,...,L5* are described in Sect. 5. In comparison with Cash et al. [4], we took variant $\prod_{\mathsf{bas}}^{\mathsf{dyn,ro}}$ as basis and estimated the most efficient variant $\prod_{\mathsf{2lev}}^{\mathsf{dyn,ro}}$, where d_w, a_w, and c'' denote the total number of deletion operations, addition operations, the constant bit size required to store a single file-keyword pair, respectively (in the client storage, the worst case of $a_w = m$). To simplify notation, we assume that both pointers and identifiers are of size c'' and that one can fit b such identifiers/pointers per block of size b (also a simplification). The hidden constants in the asymptotic complexity of the update operation is significant as the update of [4] requires at least six PRF operations per file-keyword pair versus this work, which requires one.
*Our persistent client storage is $\kappa \cdot \mathcal{O}(m+n)$. This can become 4κ if we store this data structure on the server side, which would cost one additional round of interaction.

handle malicious clients. Blind Seer focuses on a different scenario and system model compared to traditional SSE schemes: “The SSE setting focuses on data outsourcing rather than data sharing. That is, in SSE the data owner is the client, and so no privacy against the client is required” [17]. Moreover, Blind Seer requires three parties (one of them acts as a semi-trusted party) instead of two. Our scheme focuses on only basic keyword queries but achieves the highest update privacy in the traditional SSE setting. The update functionality of Blind Seer is not oblivious (this is explicitly noted in [17] on page 8, footnote 2). The Blind Seer solves the leakage problem due to non-oblivious updates by periodically re-encrypting the entire index.

3 Preliminaries and Models

Operators $||$ and $|x|$ denote the concatenation and the bit length of variable x, respectively. $x \overset{\$}{\leftarrow} \mathcal{S}$ means variable x is randomly and uniformly selected from set $\mathcal{S}$. For any integer l, $(x_0, \ldots, x_l) \overset{\$}{\leftarrow} \mathcal{S}$ means $(x_0 \overset{\$}{\leftarrow} \mathcal{S}, \ldots, x_l \overset{\$}{\leftarrow} \mathcal{S})$. $|\mathcal{S}|$ denotes the cardinality of set $\mathcal{S}$. $\{x_i\}_{i=0}^{l}$ denotes $(x_0, \ldots, x_l)$. We denote by $\{0,1\}^*$ the set of binary strings of any finite length. $\lfloor x \rfloor$ denotes the floor of x and $\lceil x \rceil$ denotes the ceiling of x. The set of items q_i for $i = 1, \ldots, n$ is denoted by $\langle q_1, \ldots, q_n \rangle$. Integer κ denotes the security parameter. $\log x$ means $\log_2 x$. $I[*, j]$ and $I[i, *]$ mean accessing all elements in the j'th column and the i'th row of a matrix I, respectively. $I[i, *]^T$ is the transpose of the i'th row of I.

An IND-CPA secure private key encryption scheme is a triplet $\mathcal{E} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ of three algorithms as follows: $k_1 \leftarrow \mathcal{E}.\mathsf{Gen}(1^\kappa)$ is a Probabilistic Polynomial Time (PPT) algorithm that takes a security parameter κ and returns a secret key k_1; $c \leftarrow \mathcal{E}.\mathsf{Enc}_{k_1}(M)$ takes secret key k_1 and a message M, and returns a ciphertext c; $M \leftarrow \mathcal{E}.\mathsf{Dec}_{k_1}(c)$ is a deterministic algorithm that takes $k_1$and c, and returns M if k_1 was the key under which c was produced. A Pseudo Random Function (PRF) is a polynomial-time computable function, which is indistinguishable from a true random function by any PPT adversary. The function $F : \{0,1\}^\kappa \times \{0,1\}^* \rightarrow \{0,1\}^\kappa$ is a keyed PRF, denoted by $\tau \leftarrow F_{k_2}(x)$, which takes as input a secret key $k_2 \overset{\$}{\leftarrow} \{0,1\}^\kappa$ and a string x, and returns a token τ. $G : \{0,1\}^\kappa \times \{0,1\}^* \rightarrow \{0,1\}^\kappa$ is a keyed PRF denoted as $r \leftarrow G_{k_3}(x)$, which takes as input $k_3 \leftarrow \{0,1\}^\kappa$ and a string x and returns a key r. We denote by $H : \{0,1\}^{|x|} \rightarrow \{0,1\}$ a Random Oracle (RO) [3], which takes an input x and returns a bit as output.

We follow the definitions of [12,13] with some modifications: f_{id} and w denote a file with unique identifier id and a unique keyword that exists in a file, respectively. A keyword w is of length polynomial in κ, and a file f_{id} may contain any such keyword (i.e., our keyword universe is not fixed). For practical purposes, n and m denote the maximum number of files and keywords to be processed by application, respectively. $\mathbf{f} = (f_{id_1}, \ldots, f_{id_n})$ and $\mathbf{c} = (c_{id_1}, \ldots, c_{id_n})$ denote a collection of files (with unique identifiers $id_1, \ldots, id_n$) and their corresponding ciphertext computed under k_1 via Enc, respectively. Data structures δ and γ denote the index and encrypted index, respectively.

Definition 1. *A* DSSE *scheme is comprised of nine polynomial-time algorithms, which are defined as below:*

1. $\underline{K \leftarrow \mathsf{Gen}(1^\kappa)}$: *It takes as input a security parameter* κ *and outputs a secret key* K.
2. $\underline{(\gamma, \boldsymbol{c}) \leftarrow \mathsf{Enc}_K(\delta, \boldsymbol{f})}$: *It takes as input a secret key* K*, an index* δ *and files* $\boldsymbol{f}$*, from which* δ *was constructed. It outputs encrypted index* γ *and ciphertexts* $\boldsymbol{c}$*.*
3. $\underline{f_j \leftarrow \mathsf{Dec}_K(c_j)}$: *It takes as input secret key* K *and ciphertext* c_j *and outputs a file* f_j*.*
4. $\underline{\tau_w \leftarrow \mathsf{SrchToken}(K, w)}$: *It takes as input a secret key* K *and a keyword* w*. It outputs a search token* τ_w*.*
5. $\underline{\mathbf{id_w} \leftarrow \mathsf{Search}(\tau_w, \gamma)}$: *It takes as input a search token* τ_w *and an encrypted index* γ*. It outputs identifiers* $\mathbf{id_w} \subseteq \boldsymbol{c}$*.*
6. $\underline{(\tau_f, c) \leftarrow \mathsf{AddToken}(K, f_{id})}$: *It takes as input a secret key* K *and a file* f_{id} *with identifier id to be added. It outputs an addition token* τ_f *and a ciphertext* c *of* f_{id}*.*
7. $\underline{(\gamma', \boldsymbol{c}') \leftarrow \mathsf{Add}(\gamma, \boldsymbol{c}, c, \tau_f)}$: *It takes as input an encrypted index* γ*, current ciphertexts* $\boldsymbol{c}$*, ciphertext* c *to be added and an addition token* τ_f*. It outputs a new encrypted index* γ' *and new ciphertexts* $\boldsymbol{c}'$*.*
8. $\underline{\tau'_f \leftarrow \mathsf{DeleteToken}(K, f_{id})}$: *It takes as input a secret key* K *and a file* f_{id} *with identifier id to be deleted. It outputs a deletion token* τ'_f*.*
9. $\underline{(\gamma', \boldsymbol{c}') \leftarrow \mathsf{Delete}(\gamma, \boldsymbol{c}, \tau'_f)}$: *It takes as input an encrypted index* γ*, ciphertexts* $\boldsymbol{c}$*, and a deletion token* τ'_f*. It outputs a new encrypted index* γ' *and new ciphertexts* $\boldsymbol{c}'$*.*

Definition 2. *A* DSSE *scheme is correct if for all* κ*, for all keys* K *generated by* $\mathsf{Gen}(1^\kappa)$*, for all* $\boldsymbol{f}$*, for all* $(\gamma, \boldsymbol{c})$ *output by* $\mathsf{Enc}_K(\delta, \boldsymbol{f})$*, and for all sequences of add, delete or search operations on* γ*, search always returns the correct set of identifier* $\mathbf{id_w}$*.*

Most known efficient SSE schemes (e.g., [4,5,12,13,16,19]) reveal the *access and search patterns* that are defined below.

Definition 3. *Given search query* $\mathsf{Query} = w$ *at time* t*,* the search pattern $\mathcal{P}(\delta, Query, t)$ *is a binary vector of length* t *with a 1 at location* i *if the search time* $i \leq t$ *was for* w*, 0 otherwise. The search pattern indicates whether the same keyword has been searched in the past or not.*

Definition 4. *Given search query* $\mathsf{Query} = w_i$ *at time* t*,* the access pattern $\Delta(\delta, \boldsymbol{f}, w_i, t)$ *is identifiers* $\mathbf{id_w}$ *of files* $\boldsymbol{f}$*, in which* w_i *appears.*

We consider the following leakage functions, in the line of [12] that captures dynamic file addition/deletion in its security model as we do, but we leak much less information compared to [12] (see Sect. 5).

Definition 5. *Leakage functions* $(\mathcal{L}_1, \mathcal{L}_2)$ *are defined as follows:*

1. $(m, n, \mathbf{id_w}, \langle |f_{id_1}|, \ldots, |f_{id_n}| \rangle) \leftarrow \mathcal{L}_1(\delta, \boldsymbol{f})$: *Given the index* δ *and the set of files* $\boldsymbol{f}$ *(including their identifiers),* $\mathcal{L}_1$ *outputs the maximum number of keywords* m*, the maximum number of files* n*, the identifiers* $\mathbf{id_w} = (id_1, \ldots, id_n)$ *of* $\boldsymbol{f}$ *and the size of each file* $|f_{id_j}|, 1 \le j \le n$ *(which also implies the size of its corresponding ciphertext* $|c_{id_j}|$*).*
2. $(\mathcal{P}(\delta, Query, t), \Delta(\delta, \boldsymbol{f}, w_i, t)) \leftarrow \mathcal{L}_2(\delta, \boldsymbol{f}, w, t)$: *Given the index* δ*, the set of files* $\boldsymbol{f}$ *and a keyword* w *for a search operation at time* t*, it outputs the search and access patterns.*

Definition 6. *Let* $\mathcal{A}$ *be a stateful adversary and* $\mathcal{S}$ *be a stateful simulator. Consider the following probabilistic experiments:*

$\mathbf{Real}_{\mathcal{A}}(\kappa)$: *The challenger executes* $K \leftarrow \mathsf{Gen}(1^\kappa)$*.* $\mathcal{A}$ *produces* $(\delta, \boldsymbol{f})$ *and receives* $(\gamma, \boldsymbol{c}) \leftarrow \mathsf{Enc}_K(\delta, \boldsymbol{f})$ *from the challenger.* $\mathcal{A}$ *makes a polynomial number of adaptive queries* $\mathsf{Query} \in (w, f_{id}, f_{id'})$ *to the challenger. If* $\mathsf{Query} = w$ *then* $\mathcal{A}$ *receives a search token* $\tau_w \leftarrow \mathsf{SrchToken}(K, w)$ *from the challenger. If* $\mathsf{Query} = f_{id}$ *is a file addition query then* $\mathcal{A}$ *receives an addition token* $(\tau_f, c) \leftarrow \mathsf{AddToken}(K, f_{id})$ *from the challenger. If* $\mathsf{Query} = f_{id'}$ *is a file deletion query then* $\mathcal{A}$ *receives a deletion token* $\tau'_f \leftarrow \mathsf{DeleteToken}(K, f_{id'})$ *from the challenger. Eventually,* $\mathcal{A}$ *returns a bit* b *that is output by the experiment.*

$\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(\kappa)$: $\mathcal{A}$ *produces* $(\delta, \boldsymbol{f})$*. Given* $\mathcal{L}_1(\delta, \boldsymbol{f})$*,* $\mathcal{S}$ *generates and sends* $(\gamma, \mathbf{c})$ *to* $\mathcal{A}$ *.* $\mathcal{A}$ *makes a polynomial number of adaptive queries* $\mathsf{Query} \in (w, f_{id}, f_{id'})$ *to* $\mathcal{S}$ *. For each query,* $\mathcal{S}$ *is given* $\mathcal{L}_2(\delta, \boldsymbol{f}, w, t)$*. If* $\mathsf{Query} = w$ *then* $\mathcal{S}$ *returns a simulated search token* τ_w*. If* $\mathsf{Query} = f_{id}$ *or* $\mathsf{Query} = f_{id'}$*,* $\mathcal{S}$ *returns a simulated addition token* τ_f *or deletion token* τ'_f *,respectively. Eventually,* $\mathcal{A}$ *returns a bit* b *that is output by the experiment.*

A DSSE *is said* $(\mathcal{L}_1, \mathcal{L}_2)$*-secure against adaptive chosen-keyword attacks (CKA2-security) if for all* PPT *adversaries* $\mathcal{A}$ *, there exists a* PPT *simulator* $\mathcal{S}$ *such that*

$$|\Pr[\mathbf{Real}_{\mathcal{A}}(\kappa) = 1] - \Pr[\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(\kappa) = 1]| \le \mathsf{neg}(\kappa)$$

Remark 1. In Definition 6, we adapt the notion of *dynamic CKA2-security* from [12], which captures the file addition and deletion operations by simulating corresponding tokens τ_f and τ'_f, respectively (see Sect. 5).

4 Our Scheme

We first discuss the intuition and data structures of our scheme. We then outline how these data structures guarantee the correctness of our scheme. Finally, we present our main scheme in detail (an efficient variant of our main scheme is given in Sect. 6, several other variants of our main scheme are given in the full version of this paper in [20]).

Intuition and Data Structures of Our Scheme. The intuition behind our scheme is to rely on a very simple data structure that enables efficient yet secure search and update operations on it. Our data structure is a bit matrix I that is augmented by two static hash tables T_w and T_f. If $I[i,j]=1$ then it means keyword w_i is present in file f_j, else w_i is not in f_j. The data structure contains both the traditional index and the inverted index representations. We use static hash tables T_w and T_f to uniquely associate a keyword w and a file f to a row index i and a column index j, respectively. Both matrix and hash tables also maintain certain status bits and counters to ensure secure and correct encryption/decryption of the data structure, which guarantees a high level of privacy (i.e., *L1* as in Sect. 5) with dynamic CKA2-security [12]. Search and update operations are encryption/decryption operations on rows and columns of I, respectively.

As in other index-based schemes, our DSSE scheme has an index δ represented by a $m \times n$ matrix, where $\delta[i,j] \in \{0,1\}$ for $i=1,\ldots,m$ and $j=1,\ldots,n$. Initially, all elements of δ are set to 0. I is a $m \times n$ matrix, where $I[i,j] \in \{0,1\}^2$. $I[i,j].v$ stores $\delta[i,j]$ in encrypted form depending on state and counter information. $I[i,j].st$ stores a bit indicating the state of $I[i,j].v$. Initially, all elements of I are set to 0. $I[i,j].st$ is set to 1 whenever its corresponding f_j is updated, and it is set to 0 whenever its corresponding keyword w_i is searched. For the sake of brevity, we will often write $I[i,j]$ to denote $I[i,j].v$. We will always be explicit about the state bit $I[i,j].st$. The encrypted index γ corresponds to the encrypted matrix I and a hash table. We also have client state information[2] in the form of two static hash tables (defined below). We map each file f_{id} and keyword w pair to a unique set of indices (i,j) in matrices (δ, I). We use static hash tables to associate each file and keyword to its corresponding row and column index, respectively. Static hash tables also enable to access the index information in (average) $\mathcal{O}(1)$ time. T_f is a static hash table whose key-value pair is $\{s_{f_j}, \langle j, st_j\rangle\}$, where $s_{f_j} \leftarrow F_{k_2}(id_j)$ for file identifier id_j corresponding to file f_{id_j}, index $j \in \{1,\ldots,n\}$ and st is a counter value. We denote access operations by $j \leftarrow T_f(s_{f_j})$ and $st_j \leftarrow T_f[j].st$. T_w is a static hash table whose key-value pair is $\{s_{w_i}, \langle i, \overline{st}_i\rangle\}$, where token $s_{w_i} \leftarrow F_{k_2}(w_i)$, index $i \in \{1,\ldots,n\}$ and $\overline{st}$ is a counter value. We denote access operations by $i \leftarrow T_w(s_{w_i})$ and $\overline{st}_i \leftarrow T_w[i].st$. All counter values are initially set to 1.

We now outline how these data structures and variables work and ensure the correctness of our scheme.

Correctness of Our Scheme. The correctness and consistency of our scheme is achieved via state bits $I[i,j].st$, and counters $T_w[i].st$ of row i and counters $T_f[j].st$ of column j, each maintained with hash tables T_w and T_f, respectively.

The algorithms $\mathsf{SrchToken}$ and $\mathsf{AddToken}$ increase counters $T_w[i].st$ for keyword w and $T_f[j].st$ for file f_j, after each search and update operations, respectively. These counters allow the derivation of a new bit, which is used to encrypt the corresponding cell $I[i,j]$. This is done by the invocation of random oracle

[2] It is always possible to eliminate client state by encrypting and storing it on the server side. This comes at the cost of additional iteration, as the client would need to retrieve the encrypted hash tables from the server and decrypt them. Asymptotically, this does not change the complexity of the schemes proposed here.

as $H(r_i||j||st_j)$ with row key r_i, column position j and the counter of column j. The row key r_i used in $H(.)$ is re-derived based on the value of row counter $\overline{st}_i$ as $r_i \leftarrow G_{k_3}(i||\overline{st}_i)$, which is increased after each search operation. Hence, if a search is followed by an update, algorithm AddToken derives a fresh key $r_i \leftarrow G_{k_3}(i||\overline{st}_i)$, which was not released during the previous search as a token. This ensures that AddToken algorithm securely and correctly encrypts the new column of added/deleted file. Algorithm Add then replaces new column j with the old one, increments column counter and sets state bits $I[*, j]$ to 1 (indicating cells are updated) for the consistency.

The rest is to show that SrchToken and Search produce correct search results. If keyword w is searched for the first time, SrchToken derives only r_i, since there were no past search increasing the counter value. Otherwise, it derives r_i with the current counter value $\overline{st}_i$ and $\overline{r}_i$ with the previous counter value $\overline{st}_i - 1$, which will be used to decrypt recently updated and non-updated (after the last search) cells of $I[i, *]$, respectively. That is, given search token τ_w, the algorithm Search step 1 checks if τ_w includes only one key (i.e., the first search) or corresponding cell value $I[i, j]$ was updated (i.e., $I[i, j].st = 1$). If one of these conditions holds, the algorithm Search decrypts $I[i, j]$ with bit $H(r_i||j||st_j)$ that was used for encryption by algorithm Enc (i.e., the first search) or AddToken. Otherwise, it decrypts $I[i, j]$ with bit $H(\overline{r}_i||j||st_j)$. Hence, the algorithm Search produces the correct search result by properly decrypting row i. The algorithm Search also ensures the consistency by setting all state bits $I[i, *].st$ to zero (i.e., indicating cells are searched) and re-encrypting $I[i, *]$ by using the last row key r_i.

Detailed Description. We now describe our main scheme in detail.

```
$K \leftarrow \mathsf{Gen}(1^\kappa)$: The client generates $k_1 \leftarrow \mathcal{E}.\mathsf{Gen}(1^\kappa)$, $(k_2, k_3) \xleftarrow{\$} \{0,1\}^\kappa$ and $K \leftarrow (k_1, k_2, k_3)$.
$(\gamma, \mathbf{c}) \leftarrow \mathsf{Enc}_K(\delta, \mathbf{f})$: The client generates $(\gamma, \mathbf{c})$:
    1. Extract unique keywords $(w_1, \ldots, w_{m'})$ from files $\mathbf{f} = (f_{id_1}, \ldots, f_{id_{n'}})$, where $n' \leq n$ and $m' \leq m$. Initially, set all the elements of $\delta$ to 0.
    2. Construct $\delta$ for $j = 1, \ldots, n'$ and $i = 1, \ldots, m'$:
        (a) $s_{w_i} \leftarrow F_{k_2}(w_i)$, $x_i \leftarrow T_w(s_{w_i})$, $s_{f_j} \leftarrow F_{k_2}(id_j)$ and $y_j \leftarrow T_f(s_{f_j})$.
        (b) If $w_i$ appears in $f_j$ set $\delta[x_i, y_j] \leftarrow 1$.
    3. Encrypt $\delta$ for $j = 1, \ldots, n$ and $i = 1, \ldots, m$:
        (a) $T_w[i].st \leftarrow 1$, $T_f[j].st \leftarrow 1$ and $I[i, j].st \leftarrow 0$.
        (b) $r_i \leftarrow G_{k_3}(i||\overline{st}_i)$, where $\overline{st}_i \leftarrow T_w[i].st$.
        (c) $I[i, j] \leftarrow \delta[i, j] \oplus H(r_i||j||st_j)$, where $st_j \leftarrow T_f[j].st$.
    4. $c_j \leftarrow \mathcal{E}.\mathsf{Enc}_{k_1}(f_{id_j})$ for $j = 1, \ldots, n'$ and $\mathbf{c} \leftarrow \{\langle c_1, y_1 \rangle, \ldots, \langle c_{n'}, y_{n'} \rangle\}$.
    5. Output $(\gamma, \mathbf{c})$, where $\gamma \leftarrow (I, T_f)$. The client gives $(\gamma, \mathbf{c})$ to the server, and keeps $(K, T_w, T_f)$.
$f_j \leftarrow \mathsf{Dec}_K(c_j)$: The client obtains the file as $f_j \leftarrow \mathcal{E}.\mathsf{Dec}_{k_1}(c_j)$.
$\tau_w \leftarrow \mathsf{SrchToken}(K, w)$: The client generates a token $\tau_w$ for $w$:
    1. $s_{w_i} \leftarrow F_{k_2}(w)$, $i \leftarrow T_w(s_{w_i})$, $\overline{st}_i \leftarrow T_w[i].st$ and $r_i \leftarrow G_{k_3}(i||\overline{st}_i)$.
    2. If $\overline{st}_i = 1$ then $\tau_w \leftarrow (i, r_i)$ . Else (if $\overline{st}_i > 1$), $\overline{r}_i \leftarrow G_{k_3}(i||\overline{st}_i - 1)$ and $\tau_w \leftarrow (i, r_i, \overline{r}_i)$.
```

3. $T_w[i].st \leftarrow \overline{st}_i + 1$. The client outputs τ_w and sends it to the server.

$\mathbf{id_w} \leftarrow \mathsf{Search}(\tau_w, \gamma)$: The server finds indexes of ciphertexts for τ_w:

1. If $((\tau_w = (i, r_i) \vee I[i,j].st) = 1)$ hold then $I'[i,j] \leftarrow I[i,j] \oplus H(r_i||j||st_j)$, else set $I'[i,j] \leftarrow I[i,j] \oplus H(\overline{r}_i||j||st_j)$, where $st_j \leftarrow T_f[j].st$ for $j = 1, \ldots, n$.
2. Set $I[i,*].st \leftarrow 0$, $l' \leftarrow 1$ and for each j satisfies $I'[i,j] = 1$, set $y_{l'} \leftarrow j$ and $l' \leftarrow l' + 1$.
3. Output $\mathbf{id_w} \leftarrow (\mathbf{y_1}, \ldots, \mathbf{y_l})$. The server returns $(c_{y_1}, \ldots, c_{y_l})$ to the client, where $l \leftarrow l' - 1$.
4. After the search is completed, the server re-encrypts row $I'[i,*]$ with r_i as $I[i,j] \leftarrow I'[i,j] \oplus H(r_i||j||st_j)$ for $j = 1, \ldots, n$, where $st_j \leftarrow T_f[j].st$ and sets $\gamma \leftarrow (I, T_w)$.

$(\tau_f, c) \leftarrow \mathsf{AddToken}(K, f_{id_j})$: The client generates τ_f for a file f_{id_j}:

1. $s_{f_j} \leftarrow F_{k_2}(id_j)$, $j \leftarrow T_f(s_{f_j})$, $T_f[j].st \leftarrow T_f[j].st + 1$, $st_j \leftarrow T_f[j].st$.
2. $r_i \leftarrow G_{k_3}(i||\overline{st}_i)$, where $\overline{st}_i \leftarrow T_w[i].st$ for $i = 1, \ldots, m$.
3. Extract $(w_1, \ldots, w_t)$ from f_{id_j} and compute $s_{w_i} \leftarrow F_{k_2}(w_i)$ and $x_i \leftarrow T_w(s_{w_i})$ for $i = 1, \ldots, t$.
4. Set $\overline{I}[x_i] \leftarrow 1$ for $i = 1, \ldots, t$ and rest of the elements as $\{\overline{I}[i] \leftarrow 0\}_{i=1, i \notin \{x_1, \ldots, x_t\}}^{m}$. Also set $I'[i] \leftarrow \overline{I}[i] \oplus H(r_i||j||st_j)$ for $i = 1, \ldots, m$.
5. Set $c \leftarrow \mathcal{E}.\mathsf{Enc}_{k_1}(f_{id_j})$ and output $(\tau_f \leftarrow (I', j), c)$. The client sends (τ_f, c) to the server.

$(\gamma', \mathbf{c}') \leftarrow \mathsf{Add}(\gamma, \mathbf{c}, c, \tau_f)$: The server performs file addition:

1. $I[*, j] \leftarrow (I')^T$, $I[*, j].st \leftarrow 1$ and $T_f[j].st \leftarrow T_f[j].st + 1$.
2. Output $(\gamma', \mathbf{c}')$, where $\gamma' \leftarrow (I, T_f)$ and $\mathbf{c}'$ is (c, j) added to $\mathbf{c}$.

$\tau_f' \leftarrow \mathsf{DeleteToken}(K, f)$: The client generates τ_f' for f:

1. Execute steps (1–2) of AddToken algorithm, which produce (j, r_i, st_j).
2. $I'[i] \leftarrow H(r_i||j|st_j)$ for $i = 1, \ldots, m$[3].
3. Output $\tau_f' \leftarrow (I', j)$. The client sends τ_f' to the server.

$(\gamma', \mathbf{c}') \leftarrow \mathsf{Delete}(\gamma, \mathbf{c}, \tau_f')$: The server performs file deletion:

1. $I[*, j] \leftarrow (I')^T$, $I[*, j].st \leftarrow 1$ and $T_f[j].st \leftarrow T_f[j].st + 1$.
2. Output $(\gamma', \mathbf{c}')$, where $\gamma' \leftarrow (I, T_f)$, $\mathbf{c}'$ is (c, j) removed from $\mathbf{c}$.

Keyword Update for Existing Files: Some existing schemes (e.g., [16]) only permit adding or deleting a file, but do not permit updating keywords in an existing file. Our scheme enables keyword update in an existing file. To update an existing file f by adding new keywords or removing existing keywords, the client prepares a new column $\overline{I}[i] \leftarrow b_i$, $i = 1, \ldots, m$, where $b_i = 1$ if w_i is added and $b_i = 0$ otherwise (as in AddToken, step 4). The rest of the algorithm is similar to AddToken.

[3] This step is only meant to keep data structure consistency during a search operation.

5 Security Analysis

We prove that our main scheme achieves *dynamic adaptive security against chosen-keyword attacks (CKA2)* as below. It is straightforward to extend the proof for our variant schemes. Note that our scheme is secure in the Random Oracle Model (ROM) [3]. That is, $\mathcal{A}$ is given access to a random oracle $RO(.)$ from which she can request the hash of any message of her choice. In our proof, cryptographic function H used in our scheme is modeled as a random oracle via function $RO(.)$.

Theorem 1. *If* Enc *is IND-CPA secure,* (F, G) *are PRFs and* H *is a RO then our DSSE scheme is* $(\mathcal{L}_1, \mathcal{L}_2)$*-secure in ROM according to Definition 6 (CKA-2 security with update operations).*

Proof. We construct a simulator $\mathcal{S}$ that interacts with an adversary $\mathcal{A}$ in an execution of an $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(\kappa)$ experiment as described in Definition 6.

In this experiment, $\mathcal{S}$ maintains lists $\mathcal{LR}$, $\mathcal{LK}$ and $\mathcal{LH}$ to keep track the query results, states and history information, initially all lists empty. $\mathcal{LR}$ is a list of key-value pairs and is used to keep track $RO(.)$ queries. We denote $\mathsf{value} \leftarrow \mathcal{LR}(\mathsf{key})$ and $\perp \leftarrow \mathcal{LR}(\mathsf{key})$ if key does not exist in $\mathcal{LR}$. $\mathcal{LK}$ is used to keep track random values generated during the simulation and it follows the same notation that of $\mathcal{LR}$. $\mathcal{LH}$ is used to keep track search and update queries, $\mathcal{S}$'s replies to those queries and their leakage output from $(\mathcal{L}_1, \mathcal{L}_2)$.

$\mathcal{S}$ executes the simulation as follows:

I. Handle RO(.) Queries: $b \leftarrow RO(x)$ takes an input x and returns a bit b as output. Given x, if $\perp = \mathcal{LR}(x)$ set $b \xleftarrow{\$} \{0,1\}$, insert (x, b) into $\mathcal{LR}$ and return b as the output. Else, return $b \leftarrow \mathcal{LR}(x)$ as the output.

II. Simulate $(\gamma, \mathbf{c})$: Given $(m, n, \langle id_1, \ldots, id_{n'} \rangle, \langle |c_{id_1}|, \ldots, |c_{id_{n'}}| \rangle) \leftarrow \mathcal{L}_1(\delta, \mathbf{f})$, $\mathcal{S}$ simulates $(\gamma, \mathbf{c})$ as follows:

1. $s_{f_j} \xleftarrow{\$} \{0,1\}^{\kappa}$, $y_j \leftarrow T_f(s_{f_j})$, insert (id_j, s_{f_j}, y_j) into $\mathcal{LH}$ and encrypt $c_{y_j} \leftarrow \mathcal{E}.\mathsf{Enc}_k(\{0\}^{|c_{id_j}|})$, where $k \xleftarrow{\$} \{0,1\}^{\kappa}$ for $j = 1, \ldots, n'$.
2. For $j = 1, \ldots, n$ and $i = 1, \ldots, m$
   (a) $T_w[i].st \leftarrow 1$ and $T_f[j].st \leftarrow 1$.
   (b) $z_{i,j} \xleftarrow{\$} \{0,1\}^{2\kappa}$, $I[i,j] \leftarrow RO(z_{i,j})$ and $I[i,j].st \leftarrow 0$.
3. Output $(\gamma, \mathbf{c})$, where $\gamma \leftarrow (I, T_f)$ and $\mathbf{c} \leftarrow \{\langle c_1, y_1 \rangle, \ldots, \langle c_{n'}, y_{n'} \rangle\}$.

Correctness and Indistinguishability of the Simulation: $\mathbf{c}$ has the correct size and distribution, since $\mathcal{L}_1$ leaks $\langle |c_{id_1}|, \ldots, |c_{id_{n'}}| \rangle$ and Enc is a IND-CPA secure scheme, respectively. I and T_f have the correct size since $\mathcal{L}_1$ leaks (m, n). Each $I[i,j]$ for $j = 1, \ldots, n$ and $i = 1, \ldots, m$ has random uniform distribution as required, since $RO(.)$ is invoked with a separate random number $z_{i,j}$. T_f has the correct distribution, since each s_{f_j} has random uniform distribution, for $j = 1, \ldots, n'$. Hence, $\mathcal{A}$ does not abort due to $\mathcal{A}$'s simulation of $(\gamma, \mathbf{c})$. The probability that $\mathcal{A}$ queries $RO(.)$ on any $z_{i,j}$ before $\mathcal{S}$ provides I to $\mathcal{A}$ is negligible (i.e., $\frac{1}{2^{2\kappa}}$). Hence, $\mathcal{S}$ also does not abort.

III. Simulate τ_w: Simulator $\mathcal{S}$ receives a search query w on time t. $\mathcal{S}$ is given $(\mathcal{P}(\delta, \text{Query}, t), \Delta(\delta, \mathbf{f}, w_i, t)) \leftarrow \mathcal{L}_2(\delta, \mathbf{f}, w, t)$. $\mathcal{S}$ adds these to $\mathcal{LH}$. $\mathcal{S}$ then simulates τ_w and updates lists $(\mathcal{LR}, \mathcal{LK})$ as follows:

1. If w in list $\mathcal{LH}$ then fetch corresponding s_{w_i}. Else, $s_{w_i} \xleftarrow{\$} \{0,1\}^{\kappa}$, $i \leftarrow T_w(s_{w_i})$, $\overline{st}_i \leftarrow T_w[i].st$ and insert $(w, \mathcal{L}_1(\delta, \mathbf{f}), s_{w_i})$ into $\mathcal{LH}$.
2. If $\bot = \mathcal{LK}(i, \overline{st}_i)$ then $r_i \leftarrow \{0,1\}^{\kappa}$ and insert $(r_i, i, \overline{st}_i)$ into $\mathcal{LK}$. Else, $r_i \leftarrow \mathcal{LK}(i, \overline{st}_i)$.
3. If $\overline{st}_i > 1$ then $\overline{r}_i \leftarrow \mathcal{LK}(i||\overline{st}_i - 1)$, $\tau_w \leftarrow (i, r_i, \overline{r}_i)$. Else, $\tau_w \leftarrow (i, r_i)$.
4. $T_w[i].st \leftarrow \overline{st}_i + 1$.
5. Given $\mathcal{L}_2(\delta, \mathbf{f}, w, t)$, $\mathcal{S}$ knows identifiers $\mathbf{id_w} = (y_1, \ldots, y_l)$. Set $I'[i, y_j] \leftarrow 1$, $j = 1, \ldots, l$, and rest of the elements as $\{I'[i,j] \leftarrow 0\}_{j=1, j \notin \{y_1, \ldots, y_l\}}$.
6. If $((\tau_w = (i, r_i) \vee I[i,j].st) = 1)$ then $V[i,j] \leftarrow I[i,j]' \oplus I[i,j]$ and insert tuple $(r_i||j||st_j, V[i,j])$ into $\mathcal{LR}$ for $j = 1, \ldots, n$, where $st_j \leftarrow T_f[j].st$.
7. $I[i,*].st \leftarrow 0$.
8. $I[i,j] \leftarrow I'[i,j] \oplus RO(r_i||j||st_j)$, where $st_j \leftarrow T_f[j].st$ for $j = 1, \ldots, n$.
9. Output τ_w and insert (w, τ_w) into $\mathcal{LH}$.

Correctness and Indistinguishability of the Simulation: Given any $\Delta(\delta, \mathbf{f}, w_i, t)$, $\mathcal{S}$ simulates the output of $RO(.)$ such that τ_w always produces the correct search result for $\mathbf{id_w} \leftarrow \mathsf{Search}(\tau_w, \gamma)$. $\mathcal{S}$ needs to simulate the output of $RO(.)$ for two conditions (as in *III-Step 6*): (i) The first search of w_i (i.e., $\tau_w = (i, r_i)$), since $\mathcal{S}$ did not know δ during the simulation of $(\gamma, \mathbf{c})$. (ii) If any file f_{id_j} containing w_i has been updated after the last search on w_i (i.e., $I[i,j].st = 1$), since $\mathcal{S}$ does not know the content of update. $\mathcal{S}$ sets the output of $RO(.)$ for those cases by inserting tuple $(r_i||j||st_j, V[i,j])$ into $\mathcal{LR}$ (as in *III-Step 6)*. In other cases, $\mathcal{S}$ just invokes $RO(.)$ with $(r_i||j||st_j)$, which consistently returns previously inserted bit from $\mathcal{LR}$ (as in *III-Step 8*).

During the first search on w_i, each $RO(.)$ output $V[i,j] = RO(r_i||j|st_j)$ has the correct distribution, since $I[i,*]$ of γ has random uniform distribution (see *II-Correctness and Indistinguishability* argument). Let $J = (j_1, \ldots, j_l)$ be the indexes of files containing w_i, which are updated after the last search on w_i. If w_i is searched then each $RO(.)$ output $V[i,j] = RO(r_i||j|st_j)$ has the correct distribution, since $\tau_f \leftarrow (I', j)$ for indexes $j \in J$ has random uniform distribution (see *IV-Correctness and Indistinguishability* argument). Given that $\mathcal{S}$'s τ_w always produces correct $\mathbf{id_w}$ for given $\Delta(\delta, \mathbf{f}, w_i, t)$, and relevant values and $RO(.)$ outputs have the correct distribution as shown, $\mathcal{A}$ does not abort during the simulation due to $\mathcal{S}$'s search token. The probability that $\mathcal{A}$ queries $RO(.)$ on any $(r_i||j|st_j)$ before him queries $\mathcal{S}$ on τ_w is negligible (i.e., $\frac{1}{2^{\kappa}}$), and therefore $\mathcal{S}$ does not abort due to $\mathcal{A}$'s search query.

IV. Simulate (τ_f, τ'_f): $\mathcal{S}$ receives an update request $\mathsf{Query} = (\langle \mathsf{Add}, |c_{id_j}| \rangle, \mathsf{Delete})$ at time t. $\mathcal{S}$ simulates update tokens (τ_f, τ'_f) as follows:

1. If id_j in $\mathcal{LH}$ then fetch its corresponding (s_{f_j}, j) from $\mathcal{LH}$, else set $s_{f_j} \xleftarrow{\$} \{0,1\}^\kappa$, $j \leftarrow T_f(s_{f_j})$ and insert (s_{f_j}, j, f_{id_j}) into $\mathcal{LH}$.
2. $T_f[j].st \leftarrow T_f[j].st + 1$, $st_j \leftarrow T_f[j].st$.
3. If $\perp = \mathcal{LK}(i, \overline{st}_i)$ then $r_i \leftarrow \{0,1\}^\kappa$ and insert $(r_i, i, \overline{st}_i)$ into $\mathcal{LK}$, where $\overline{st}_i \leftarrow T_w[i].st$ for $i = 1, \ldots, m$.
4. $I'[i] \leftarrow RO(z_i)$, where $z_i \xleftarrow{\$} \{0,1\}^{2\kappa}$ for $i = 1, \ldots, m$.
5. $I[*, j] \leftarrow (I')^T$ and $I[*, j].st \leftarrow 1$.
6. If $\mathsf{Query} = \langle \mathsf{Add}, |c_{id_j}| \rangle$, simulate $c_j \leftarrow \mathcal{E}.\mathsf{Enc}_k(\{0\}^{|c_{id}|})$, add c_j into $\mathbf{c}$, set $\tau_f \leftarrow (I', j)$ output (τ_f, j). Else set $\tau'_f \leftarrow (I', j)$, remove c_j from $\mathbf{c}$ and output τ'_f.

Correctness and Indistinguishability of the Simulation: Given any access pattern (τ_f, τ'_f) for a file f_{id_j}, $\mathcal{A}$ checks the correctness of update by searching all keywords $W = (w_{i_1}, \ldots, w_{i_l})$ included f_{id_j}. Since $\mathcal{S}$ is given access pattern $\Delta(\delta, \mathbf{f}, w_i, t)$ for a search query (which captures the last update before the search), the search operation always produces a correct result after an update (see *III-Correctness and Indistinguishability* argument). Hence, $\mathcal{S}$'s update tokens are correct and consistent.

It remains to show that (τ_f, τ'_f) have the correct probability distribution. In real algorithm, st_j of file f_{id_j} is increased for each update as simulated in *IV-Step 2*. If f_{id_j} is updated after w_i is searched, a new r_i is generated for w_i as simulated in *IV-Step 3* (r_i remains the same for consecutive updates but st_j is increased). Hence, the real algorithm invokes $H(.)$ with a different input $(r_i||j||st_j)$ for $i = 1, \ldots, m$. $\mathcal{S}$ simulates this step by invoking $RO(.)$ with z_i and $I'[i] \leftarrow RO(z_i)$, for $i = 1, \ldots, m$. (τ_f, τ'_f) have random uniform distribution, since I' has random uniform distribution and update operations are correct and consistent as shown. c_j has the correct distribution, since Enc is an IND-CPA cipher. Hence, $\mathcal{A}$ does not abort during the simulation due to $\mathcal{S}$'s update tokens. The probability that $\mathcal{A}$ queries $RO(.)$ on any z_i before him queries $\mathcal{S}$ on (τ_f, τ'_f) is negligible (i.e., $\frac{1}{2^{2 \cdot \kappa}}$), and therefore $\mathcal{S}$ also does not abort due to $\mathcal{A}$'s update query.

V. Final Indistinguishability Argument: (s_{w_i}, s_{f_j}, r_i) for $i = 1, \ldots, m$ and $j = 1, \ldots, n$ are indistinguishable from real tokens and keys, since they are generated by PRFs that are indistinguishable from random functions. Enc is a IND-CPA scheme, the answers returned by $\mathcal{S}$ to $\mathcal{A}$ for $RO(.)$ queries are consistent and appropriately distributed, and all query replies of $\mathcal{S}$ to $\mathcal{A}$ during the simulation are correct and indistinguishable as discussed in *I-IV Correctness and Indistinguishability* arguments. Hence, for all PPT adversaries, the outputs of $\mathbf{Real}_{\mathcal{A}}(\kappa)$ and that of an $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(\kappa)$ experiment are negligibly close:

$$|\Pr[\mathbf{Real}_{\mathcal{A}}(\kappa) = 1] - \Pr[\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(\kappa) = 1]| \leq \mathsf{neg}(\kappa)$$

□

Remark 2. Extending the proof to Variant-I presented in Sect. 6 is straightforward[4]. In particular, (i) interaction is required because even if we need to update a single entry (column) corresponding to a single file, the client needs to re-encrypt the whole b-bit block in which the column resides to keep consistency. This, however, is achieved by retrieving the encrypted b-bit block from the server, decrypting on the *client* side and re-encrypting using AES-CTR mode. Given that we use ROs and a IND-CPA encryption scheme (AES in CTR mode) the security of the DSSE scheme is not affected in our model, and, in particular, there is no additional leakage. (ii) The price that is paid for this performance improvement is that we need interaction in the new variant. Since the messages (the columns/rows of our matrix) exchanged between client and server are encrypted with an IND-CPA encryption scheme there is no additional leakage either due to this operation.

Discussions on Privacy Levels. The leakage definition and formal security model described in Sect. 3 imply various levels of privacy for different DSSE schemes. We summarize some important privacy notions (based on the various leakage characteristics discussed in [4,12,16,19]) with different levels of privacy as follows:

- *Size pattern*: The number of file-keyword pairs in the system.
- *Forward privacy*: A search on a keyword w does not leak the identifiers of files matching this keyword for (pre-defined) future files.
- *Backward privacy*: A search on a keyword w does not leak the identifiers of files matching this keywords that were previously added but then deleted (leaked via additional info kept for deletion operations).
- *Update privacy*: Update operation may leak different levels of information depending on the construction:

– Level-1 (*L1*) leaks only the time t of the update operation and an index number. *L1* does not leak the type of update due to the type operations performed on encrypted index γ. Hence, it is possible to hide the type of update via batch/fake file addition/deletion[5]. However, if the update is addition and added file is sent to the server along with the update information on γ, then the type of update and the size of added file are leaked.
– Level-2 (*L2*) leaks *L1* plus the identifier of the file being updated and the number of keywords in the updated file (e.g., as in [19]).
– Level-3 (*L3*) leaks *L2* plus when/if that identifier has had the same keywords added or deleted before, and also when/if the same keyword have been searched before (e.g., as in [4][6]).

[4] This variant encrypts/decrypts b-bit blocks instead of single bits and it requires interaction for add/delete/update operations.

[5] In our scheme, the client may delete file f_{id_j} from γ but still may send a fake file f'_{id_j} to the server as a fake file addition operation.

[6] Remark that despite the scheme in [4] leaks more information than that of ours and [19] as discussed, it does not leak the (pseudonymous) index of file to be updated.

- Level-4 (*L4*) leaks *L3* plus the information whether the same keyword added or deleted from two files (e.g., as in [12]).
- Level-5 (*L5*) leaks significant information such as the pattern of all intersections of everything is added or deleted, whether or not the keywords were search-ed for (e.g., as in [13]).

Note that our scheme achieves the highest level of *L1* update privacy, forward-privacy, backward-privacy and size pattern privacy. Hence, it achieves the highest level of privacy among its counterparts.

6 Evaluation and Discussion

We have implemented our scheme in a stand-alone environment using C/C++. By stand-alone, we mean we run on a single machine, as we are only interested in the performance of the operations and not the effects of latency, which will be present (but are largely independent of the implementation[7].) For cryptographic primitives, we chose to use the libtomcrypt cryptographic toolkit version 1.17 [9] and as an API. We modified the low level routines to be able to call and take advantage of AES hardware acceleration instructions natively present in our hardware platform, using the corresponding freely available Intel reference implementations [11]. We performed all our experiments on an Intel dual core i5-3320M 64-bit CPU at 2.6 GHz running Ubuntu 3.11.0-14 generic build with 4GB of RAM. We use 128-bit CCM and AES-128 CMAC for file and data structure encryption, respectively. Key generation was implemented using the expand-then-extract key generation paradigm analyzed in [14]. However, instead of using a standard hash function, we used AES-128 CMAC for performance reasons. Notice that this key derivation function has been formally analyzed and is standardized. Our use of CMAC as the PRF for the key derivation function is also standardized [7]. Our random oracles were all implemented via 128-bit AES CMAC. For hash tables, we use Google's C++ sparse hash map implementation [2] but instead of using the standard hash function implementation, we called our CMAC-based random oracles truncated to 80 bits. Our implementation results are summarized in Table 2.

Experiments. We performed our experiments on the Enron dataset [1] as in [13]. Table 2 summarizes results for three types of experiments: (i) Large number of files and large number of keywords, (ii) large number of files but comparatively small number of keywords and (iii) large number of keywords but small number of files. In all cases, the combined number of keyword/file pairs is between 10^9 and 10^{10}, which surpass the experiments in [13] by about two orders of magnitude and are comparable to the experiments in [19]. One key observation is that in

[7] As it can be seen from Table 1, our scheme is optimal in terms of the number of rounds required to perform *any* operation. Thus, latency will not affect the performance of the implementation anymore than any other competing scheme. This replicates the methodology of Kamara et al. [13].

contrast to [19] (and especially to [4] with very high-end servers), we do *not* use server-level hardware but a rather standard commodity Intel platform with limited RAM memory. From our results, it is clear that for large databases the process of generating the encrypted representation is relatively expensive, however, this is a one-time only cost. The cost per keyword search depends linearly as $\mathcal{O}(n)/128$ on the number of files in the database and it is not cost-prohibiting (even for the large test case of 10^{10} keyword/file pairs, searching takes only a few msec). We observe that despite this linear cost, our search operation is extremely fast comparable to the work in [13]. The costs for adding and deleting files (updates) is similarly due to the obliviousness of these operations in our case. Except for the cost of creating the index data structure, all performance data extrapolates to any other type of data, as our data structure is not data dependant and it is conceptually very simple.

Table 2. Execution times of our DSSE scheme. w.: # of words, f.: # of files

Operation	Time (ms)					
	w. $2 \cdot 10^5$	f. $5 \cdot 10^4$	w. 2000	f. $2 \cdot 10^6$	w. $1 \cdot 10^6$	f. 5000
Building searchable representation (offline, one-time cost at initialization)						
Keyword-file mapping, extraction	6.03 s		52 min		352 ms	
Encrypt searchable representation	493 ms		461 ms		823 ms	
Search and Update Operations (online, after initialization)						
Search for single key word	0.3 ms		10 ms		0.02 ms	
Add file to database	472 ms		8.83 ms		2.77 s	
Delete file from database	329 ms		8.77 ms		2.36 s	

Comparison with Existing Alternatives. Compared to Kamara et al. in [13], which achieves optimal $O(r)$ search time but leaks significant information for updates, our scheme has linear search time (for # of files) but achieves completely oblivious updates. Moreover, the [13] can not be parallelized, whereas our scheme can. Kamara et al. [12] relies on red-black trees as the main data structure, achieves parallel search and oblivious updates. However, it incurs impractical server storage overhead due to its very large encrypted index size. The scheme of Stefanov et al. [19] requires high client storage (e.g., 210 MB for moderate size file-keyword pairs), where the client fetches non-negligible amount of data from the server and performs an oblivious sort on it. We only require *one* hash table and four symmetric secret keys storage. The scheme in [19] also requires significant amount of data storage (e.g., 1600 bits) for per keyword-file pair at the server side versus 2 bits per file-keyword pair in our scheme (and a hash table[8]).

[8] The size of the hash table depends on its occupancy factor, the number of entries and the size of each entry. Assuming 80-bits per entry and a 50 % occupancy factor, our scheme still requires about $2 \times 80 + 2 = 162$ bits per entry, which is about a factor 10 better than [19]. Observe that for fixed m-words, we need a hash table with approximately $2m$ entries, even if each entry was represented by 80-bits.

The scheme in [4] leaks more information compared to [19] also incurring in non-negligible server storage. The data structure in [4] grows linearly with the number of deletion operations, which requires re-encrypting the data structure eventually. Our scheme does not require re-encryption (but we assume an upper bound on the maximum number of files), and our storage is constant regardless of the number of updates. The scheme in [16] relies on a primitive called "Blind-Storage", where the server acts only as a storage entity. This scheme requires higher interaction than its counterparts, which may introduce response delays for distributed client-server architectures. This scheme leaks less information than that of [4], but only support single keyword queries. It can add/remove a file but cannot update the content of a file in contrast to our scheme.

We now present an efficient variant of our scheme:

Variant-I: Trade-off Between Computation and Interaction Overhead. In the main scheme, H is invoked for each column of I once, which requires $O(n)$ invocations in total. We propose a variant scheme that offers significant computational improvement at the cost of a plausible communication overhead.

We use counter (CTR) mode with a block size b for $\mathcal{E}$. We interpret columns of I as $d = \lceil \frac{n}{b} \rceil$ blocks with size of b bits each, and encrypt each block B_l, $l = 0, \ldots, d-1$, separately with $\mathcal{E}$ by using a unique block counter st_l. Each block counter st_l is located at its corresponding index a_l (block offset of B_l) in T_f, where $a_l \leftarrow (l \cdot b) + 1$. The uniqueness of each block counter is achieved with a global counter gc, which is initialized to 1 and incremented by 1 for each update. A state bit $T_f[a_l].b$ is stored to keep track the update status of its corresponding block. The update status is maintained only for each block but not for each bit of $I[i,j]$. Hence, I is a binary matrix (unlike the main scheme, in which $I[i,j] \in \{0,1\}^2$). $\mathsf{AddToken}$ and Add algorithms for the aforementioned variant are as follows ($\mathsf{DeleteToken}$ and Delete follow the similar principles):

$\underline{(\tau_f, c) \leftarrow \mathsf{AddToken}(K, f_{id_j})}$: The client generates τ_f for f_{id_j} as follows:

1. $s_{f_j} \leftarrow F_{k_2}(f_{id_j})$, $j \leftarrow T_f(s_{f_j})$, $l \leftarrow \lfloor \frac{j}{b} \rfloor$, $a_l \leftarrow (l \cdot b) + 1$ and $st_l \leftarrow T_f[a_l].st$. Extract $(w_1, \ldots, w_t)$ from f_{id_j} and compute $s_{w_i} \leftarrow F_{k_2}(w_i)$ and $x_i \leftarrow T_w(s_{w_i})$ for $i = 1, \ldots, t$. For $i = 1, \ldots, m$:
   (a) $r_i \leftarrow G_{k_3}(i \| \overline{st}_i)$, where $\overline{st}_i \leftarrow T_w[i].st$ [9].
   (b) The client requests l'th block, which contains index j of f_{id} from the server. The server then returns the corresponding block $(I[i, a_l], \ldots, I[i, a_{l+1} - 1])$, where $a_{l+1} \leftarrow b(l+1) + 1$.
   (c) $(\overline{I}[i, a_l], \ldots, \overline{I}[i, a_{l+1} - 1]) \leftarrow \mathcal{E}.\mathsf{Dec}_{r_i}(I[i, a_l], \ldots, I[i, a_{l+1} - 1], st_l)$.
2. Set $\overline{I}[x_i, j] \leftarrow 1$ for $i = 1, \ldots, t$ and $\{\overline{I}[i,j] \leftarrow 0\}_{i=1, i \notin \{x_1, \ldots, x_t\}}^{m}$.
3. $gc \leftarrow gc + 1$, $T_f[a_l].st \leftarrow gc$, $st_l \leftarrow T_f[a_l].st$ and $T_f[a_l].b \leftarrow 1$.
4. $(I'[i, a_l], \ldots, I'[i, a_{l+1} - 1]) \leftarrow \mathcal{E}.\mathsf{Enc}_{r_i}(\overline{I}[i, a_l], \ldots, \overline{I}[i, a_{l+1} - 1], st_l)$ for $i = 1, \ldots, m$. Finally, $c \leftarrow \mathcal{E}.\mathsf{Enc}_{k_1}(f_{id_j})$.
5. Output $\tau_f \leftarrow (I', j)$. The client sends (τ_f, c) to the server.

$\underline{(\gamma', \mathbf{c}') \leftarrow \mathsf{Add}(\gamma, \mathbf{c}, c, \tau_f)}$: The server performs file addition as follows:

[9] In this variant, G should generate a cryptographic key suitable for the underlying encryption function $\mathcal{E}$ (e.g., the output of KDF is $b = 128$ for AES with CTR mode).

1. Replace $(I[*, a_l], \ldots, I[*, a_{l+1} - 1])$ with I'.
2. $gc \leftarrow gc + 1$, $T_f[a_l].st \leftarrow gc$ and $T_f[a_l].b \leftarrow 1$.
3. Output $(\gamma', \mathbf{c}')$, where $\gamma' \leftarrow (I, T_f)$ and $\mathbf{c}'$ is (c, j) added to $\mathbf{c}$.

Gen and Dec algorithms of the variant scheme are identical to that of main scheme. The modifications of SrchToken and Search algorithms are straightforward (in the line of AddToken and Add) and therefore will not be repeated. In this variant, the search operation requires the decryption of b-bit blocks for $l = 0, \ldots, d-1$. Hence, $\mathcal{E}$ is invoked only $O(n/b)$ times during the search operation (in contrast to $O(n)$ invocation of H as in our main scheme). That is, the search operation becomes b times faster compared to our main scheme. The block size b can be selected according to the application requirements (e.g., $b = 64$, $b = 128$ or $b = 256$ based on the preferred encryption function). For instance, $b = 128$ yields highly efficient schemes if the underlying cipher is AES by taking advantage of AES specialized instructions in current PC platforms. Moreover, CTR mode can be parallelizable and therefore the search time can be reduced to $O(n/(b \cdot p))$, where p is the number of processors in the system. This variant requires transmitting $2 \cdot b \cdot O(m)$ bits for each update compared to $O(m)$ non-interactive transmission in our main scheme. However, one may notice that this approach offers a trade-off, which is useful for some practical applications. That is, the search speed is increased by a factor of b (e.g., $b = 128$) with the cost of transmitting just $2 \cdot b \cdot m$ bits (e.g., less than 2MB for $b = 128, m = 10^5$). However, a network delay t is introduced due to interaction.

Remark 3. The b-bit block is re-encrypted via an IND-CPA encryption scheme on the *client* side at the cost of one round of interaction. Hence, encrypting multiple columns does not leak additional information during updates over our main scheme.

We discuss other variants of our main scheme in the full version of this paper in [20].

References

1. The enron email dataaset. http://www.cs.cmu.edu/enron/
2. Sparsehash: an extemely memory efficient hash_map implementation, February 2012. https://code.google.com/p/sparsehash/
3. Bellare, M., Rogaway, P.: Random oracles are practical: a paradigm for designing efficient protocols. In: Proceedings of the 1st ACM Conference on Computer and Communications Security (CCS 1993), pp. 62–73. ACM, New York (1993)
4. Cash, D., Jaeger, J., Jarecki, S., Jutla, C., Krawcyk, H., Rosu, M.-C., Steiner, M.: Dynamic searchable encryption in very-large databases: data structures and implementation. In: 21th Annual Network and Distributed System Security Symposium – NDSS. The Internet Society, 23–26 February 2014
5. Cash, D., Jarecki, S., Jutla, C., Krawczyk, H., Roşu, M.-C., Steiner, M.: Highly-scalable searchable symmetric encryption with support for boolean queries. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013, Part I. LNCS, vol. 8042, pp. 353–373. Springer, Heidelberg (2013)

6. Chang, Y.-C., Mitzenmacher, M.: Privacy preserving keyword searches on remote encrypted data. In: Ioannidis, J., Keromytis, A.D., Yung, M. (eds.) ACNS 2005. LNCS, vol. 3531, pp. 442–455. Springer, Heidelberg (2005)
7. Chen, L.: NIST special publication 800–108: recomendation for key derivation using pseudorandom functions (revised). Technical report NIST-SP800-108, Computer Security Division, National Institute of Standards and Technology, October 2009. http://csrc.nist.gov/publications/nistpubs/800-108/sp800-108.pdf
8. Curtmola, R., Garay, J., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: improved definitions and efficient constructions. In: Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006, pp. 79–88. ACM, New York (2006)
9. St Denis, T.: LibTomCrypt library. http://libtom.org/?page=features&newsitems=5&whatfile=crypt. Accessed 12 May 2007
10. Fisch, B., Vo, B., Krell, F., Kumarasubramanian, A., Kolesnikov, V., Malkin, T., Bellovin, S.M.: Malicious-client security in blind seer: a scalable private DBMS. In: IEEE Symposium on Security and Privacy, SP. IEEE Computer Society, 18–20 May 2015
11. Gueron, S.: White Paper: Intel Advanced Encryption Standard (AES) New Instructions Set, Document Revision 3.01, September 2012. https://software.intel.com/sites/default/files/article/165683/aes-wp-2012-09-22-v01.pdf. Software Library https://software.intel.com/sites/default/files/article/181731/intel-aesni-sample-library-v1.2.zip
12. 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)
13. Kamara, S., Papamanthou, C., Roeder, T.: Dynamic searchable symmetric encryption. In: Proceedings of the ACM Conference on Computer and Communications Security, CCS 2012, pp. 965–976. ACM, New York (2012)
14. Krawczyk, H.: Cryptographic extraction and key derivation: the HKDF scheme. In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 631–648. Springer, Heidelberg (2010)
15. Kurosawa, K., Ohtaki, Y.: UC-secure searchable symmetric encryption. In: Keromytis, A.D. (ed.) FC 2012. LNCS, vol. 7397, pp. 285–298. Springer, Heidelberg (2012)
16. Naveed, M., Prabhakaran, M., Gunter, C.A.: Dynamic searchable encryption via blind storage. In: 35th IEEE Symposium on Security and Privacy, pp. 48–62, May 2014
17. Pappas, V., Krell, F., Vo, B., Kolesnikov, V., Malkin, T., Choi, S.G., George, W., Keromytis, A.D., Bellovin, S.: Blind seer: a scalable private DBMS. In: IEEE Symposium on Security and Privacy, SP, pp. 359–374. IEEE Computer Society, 18–21 May 2014
18. Song, D.X., Wagner, D., Perrig, A.: Practical techniques for searches on encrypted data. In: Proceedings of the IEEE Symposium on Security and Privacy, SP 2000, pp. 44–55. IEEE Computer Society, Washington, D.C. (2000)
19. Stefanov, E., Papamanthou, C., Shi, E.: Practical dynamic searchable encryption with small leakage. In: 21st Annual Network and Distributed System Security Symposium – NDSS. The Internet Society, 23–26 February 2014
20. Yavuz A.A., Guajardo, J.: Dynamic searchable symmetric encryption with minimal leakage and efficient updates on commodity hardware. IACR Cryptology ePrint Archive 107, March 2015