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
attila.yavuz@oregonstate.edu,
[2] Robert Bosch Research and Technology Center, Pittsburgh PA, 15203
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 parallelizable DSSE scheme that achieves the highest privacy among all compared alternatives with low information leakage, non-interactive and 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 static hash tables) that enables efficient yet secure search/update operations on it. We formally prove that our scheme is secure (in random oracle model) and demonstrated that it is fully practical with large number of file-keyword pairs even with an implementation on simple hardware configurations.

Keywords: Dynamic Symmetric Searchable Encryption, Privacy Enhancing Technologies, Secure Data Outsourcing, Secure Computing Clouds

1 Introduction

Searchable Symmetric Encryption (SSE) [7] enables a client to encrypt data in such a way that she can later perform keyword searches on it via appropriate "search tokens" [23]. Thanks to this ability, SSE finds several applications in different domains. For instance, a prominent application of SSE is to enable privacy-preserving keyword searches on cloud-based systems (e.g., Amazon S3 or Google drive). With a SSE scheme, a client can store a collection of encrypted files remotely at the cloud and yet perform keyword searches without revealing any information about the contents of either the files or the queries [15]. Ideally, any practical SSE scheme should aim at achieving (at a minimum) the following properties:

- *Dynamism*: It should permit securely adding new files or deleting existing files from the encrypted file collection, after the client generates the encrypted data at set-up time.
- *Computational Efficiency and Parallelization*: It should have sub-linear search and update times with respect to the total number of file-keyword pairs in the encrypted file collection. Moreover, search and update operations should be fully parallelizable across multiple processors, as typically available in cloud-computing environments.
- *Storage Efficiency*: The storage overhead of the server depends on the size of encrypted index (i.e., the encrypted data structure that enables keyword searches)[3]. To achieve scalable solutions, the number of bits that is needed to represent a file-keyword pair in the encrypted index should be small. Moreover, the size of encrypted index should not grow with the number of search or update operations (which requires re-encrypting the entire index eventually). The size of encrypted index should grow gradually with the number of file-keyword pairs. Last, the persistent storage at the client should be minimum (e.g., a small-constant set of secret keys).
- *Communication Efficiency*: A SSE scheme should support non-interactive update/search operations to avoid the network delays. Moreover, the data exchanged between the client and the server should be minimum.
- *Security*: The current standard security notion for SSE is adaptive security against chosen-keyword attacks (CKA2) [7], which captures adaptive search queries (see Section 3.2). This notion of SSE has been further refined (e.g., [15,14,23]) to capture dynamism (i.e., updates) and interaction. That is, the information leaked through arbitrary search and update operations (including interactions) must be precisely quantified.

Our Contributions. SSE was introduced in [21] and it was followed by several SSE schemes (e.g., [4,7,5,17]) that could only operate on static data collections. The static nature of those SSE schemes limited their applicability to applications that require handling dynamic file collections. Moreover, some of them (e.g., [7]) were not parallelizable. Kamara et. al. developed a Dynamic Searchable Symmetric Encryption (DSSE) scheme in [15] that could handle dynamic file collections via encrypted updates. However, it leaked significant information during updates and was not parallelizable. A line of recent work (e.g., [14,2,23,18]) followed it by providing security and performance improvements for various metrics (e.g., parallelizability, search/update times, update communication, scalability, client/server storage). This work focuses on providing efficient updates while leaking minimum information. We discuss further related work on SSE/DSSE schemes in Section 2.

To date there is no single DSSE scheme that outperforms all other alternatives for *all* metrics: privacy (e.g., information leak), performance (e.g., search, update execution and communication, storage) and functionality (e.g., boolean versus single keyword queries). Having this in mind, we develop a DSSE scheme

[3] In addition to the files stored which are to be searched.

that achieves the highest privacy among all compared alternatives with low information leakage (see formal leakage definitions in Section 3.2, Definition 5 and discussion in Section B), non-interactive and efficient updates (compared to [14]), compact client storage (compared to [23]), low server storage for large file-keyword pairs (compared to [14,23,2]) and conceptually simple and easy

Table 1. Performance Comparison of DSSE schemes. The analysis is given for the worst-case (asymptotic) complexity. All schemes leak search and access pattern.

Scheme/Property	[15]	[14]	[23]	[2] $\left(\prod_{\mathsf{2lev}}^{\mathsf{dyn,ro}}\right)$	**This Work**
Security Notion	CKA2	CKA2	CKA2	CKA2	CKA2
Size Pattern 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
Random Oracles	Yes	Yes	Yes	Yes	Yes
Dynamic Keyword	No	No	Yes	Yes	Yes
Persistent Client Storage	4κ	3κ	$\kappa \log(N')$	$\kappa \cdot \mathcal{O}(N')$	$\kappa \cdot \mathcal{O}(n+m)$
Transient Client Storage	—	—	$\mathcal{O}(N'^{\alpha})$	—	—
Index Size (Server Storage)	$z \cdot \mathcal{O}(m+n)$	$2{\cdot}\mathcal{O}((\kappa{+}m){\cdot}n)$	$13\kappa \cdot \mathcal{O}(N')$	$c''/b \cdot \mathcal{O}(N')$	$2 \cdot \mathcal{O}(m \cdot n)$
Grow with Updates	No	No	Yes	Yes	No
Num. Rounds Search	2	2	2	2	2
Search Time	$\mathcal{O}((r/p) \cdot \log n)$	$\mathcal{O}((r/p) \log^3(N'))$	$\mathcal{O}((r+d_w)/p)$	$1/b \cdot \mathcal{O}(n/p)$	
Num. Rounds Update	1	3	3	1	1
File Update Bandwidth	$z \cdot \mathcal{O}(m'')$	$2z \cdot \kappa \cdot \mathcal{O}(m \log n)$	$z \cdot \mathcal{O}(m'' \log N')$	$z \cdot \mathcal{O}(m \log n + m'')$	$b \cdot \mathcal{O}(m)$
File Update Time	$\mathcal{O}(m'')$	$\mathcal{O}((m/p) \cdot \log n) + \Delta t$	$\mathcal{O}(m''/p \log^2(N')) + \Delta t$	$\mathcal{O}(m''/p) + \Delta t$	$b \cdot \mathcal{O}(m/p)$
Parallelizable	No	Yes	Yes	Yes	Yes

* 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. This, however, comes at the cost of one additional round of interaction (See Section 6.1).

- Rounds refer to the number of messages exchanged between two communicating parties. A *non-interactive* search operation requires two messages (one from the client to the server with a search token and one message from the server to the client with the result of the search (i.e., an encrypted file)). An *interactive* update operation (i.e., file addition or deletion) requires three messages to be exchanged. Our main scheme, the scheme in [15] and some variants in [2] also achieve *non-interactive* update that requires only single message (i.e., an update token and an encrypted file to be added for the file addition operation) to be send from the client to the server.
- 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.
- 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. Δt is the network latency introduced due to the interactions. α is a parameter, $0 < \alpha < 1$.
- Update privacy levels *L1,...,L5* are described in Section B. In comparison with Cash et al. [2], 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). Observe that the hidden constants in the asymptotic complexity of the update operation is significant as the update operation of [2] requires at least six PRF operations per file-keyword pair versus this work, which only requires one.

to implement (compared to [15,14,23]). Table 1 further compares our scheme with existing DSSE schemes for various security and performance metrics.

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 described in Section B) with CKA2-security. Search and update operations are encryption/decryption operations on rows and columns of I, respectively. Those encryption operations are also simple, easy to implement, non-interactive (for the main scheme) and fully practical with large number of file-keyword pairs even with an implementation on simple hardware configurations (as opposed to high-end servers). The advantages of our scheme are summarized below:

• *High Security*: Our scheme achieves a high-level of update security (i.e., *Level-1* update security), forward-privacy, backward-privacy and size pattern privacy simultaneously (see Section 3 for security model and privacy notations). That is, we quantify the information leakage via leakage functions and formally prove that our scheme is CKA2-secure in random oracle model [1].

• *Compact Client Storage*: Compared to some alternatives with secure updates (e.g., Stefanov et. al. in [23]), our scheme achieves smaller client storage (e.g., up to 10-15 times with similar parameters). This is an important advantage for lightweight clients such as mobile devices. The schemes in [3,23] also require keeping state information at the client side or interaction with re-encryption (as in our variant scheme). The schemes presented in [15,14,23] do not keep state at the client but leak more information than ours.

• *Compact Server Storage with Secure Updates*: Our encrypted index size is smaller than some alternatives with *secure* updates (i.e., [14,23]). For instance, our scheme requires $O(n \cdot m) + \kappa \cdot O(m)$, while the scheme in [14] requires $(4 \cdot \kappa)O(n \cdot m) + (2 \cdot z)O(m)$. Notice that the $4 \cdot \kappa$ parameter can introduce a significant difference in practice. Asymptotically, the scheme in [23] is more (server storage) efficient for small/moderate number of file-keyword pairs, since its data structure grows gradually. However, our scheme requires only **two bits** per file-keyword pair with the maximum number of files and keywords. Hence, it is more storage efficient for large number of file-keyword pairs than [23] (e.g., requiring 1600 to 3200 bits for per file-keyword pair).

• *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., Stefanov et. al. in [23]) whose server storage overhead grows linearly with the number of file deletions.

• *Dynamic Keyword Universe*: Unlike some alternatives (e.g., [7,14,15]), our scheme does not assume a fixed keyword universe. That is, our scheme allows any new keyword to be added to the system after initialization. This property makes our scheme highly practical, since the content of the files is not restricted to a particular pre-defined keyword set (e.g., English words) but can be any token afterwards (encodings, identifiers, random numbers)[4].

• *Efficient and Non-interactive 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 [14,2,18,23]). 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 [14] and fewer rounds than [18]). Notice that the scheme in [18] 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.

• *Oblivious Updates*: Our update operation takes always the same amount of time, which does not leak timing information depending on the update. It also prevents side-channel attacks based on timing information.

• *Parallelization*: Our scheme is parallelizable for both update and search operations (unlike schemes that rely on linked-lists such as [15]).

• *Forward Privacy*: As in other works (e.g., [23]), we can achieve forward privacy by simply retrieving the whole data structure and re-encrypting it. We optimize this by retrieving the part of the data structure that has already been queried and only re-encrypting them.

2 Related Work

The idea of SSE was first introduced by Song et. al. in [21], which proposed two SSE schemes. The first scheme scans the whole file collection for patterns, which incurs a linear search overhead in the *collection length* and therefore is considered impractical. The second one uses an inverted index approach with a better efficiency, but its update mechanism is shown to be insecure in [10]. Goh in [10] associates with each file an encrypted data structure that can be tested for the occurrence of a given keyword, which achieves a search time linear in n. Moreover, Goh in [10] introduced the first formal security notion for SSE schemes called as security against chosen-keyword attacks (CKA1)-security, which only provides an assurance if the search queries are independent of the encrypted index, ciphertext and of previous search results. This is not a realistic security assumption for most practical applications.

[4] In [23], the term "dynamic keyword" refers to the scheme storing the info about the keywords that currently appear in the files, but does not require extra space for future keywords. In our case, it means that the keyword dictionary is not static, meaning any keyword can be dynamically added/deleted after the initialization of the system. However, our scheme assumes the maximum number of keywords to be used in the system is predefined and the size of the encrypted index is set accordingly during the initialization phase.

Curtmola et al. in [7] associate an encrypted inverted index with the entire file collection. This approach yields efficient schemes since the search time is sub-linear and optimal as $O(r)$, where r is the number of files that contain a particular keyword. Moreover, [7] also improved the SSE formal security notion by introducing the stronger *adaptive security against chosen-keyword attacks (CKA2)* security definition. Unlike non-adaptive CKA1-security [10], CKA2-security permits adaptive search queries that may depend on past queries. Despite its advantages, the approach of [7] has important limitations: (i) It is *static*, meaning that it cannot handle dynamic file collections, which is a significant drawback for applications such as cloud-computing; and (ii) it is based on an encrypted linked list, which is inherently sequential. Hence, the scheme in [7] can *not* be parallelized on multiple processors. To address this limitation, Kamara et al. in [15] developed a Dynamic Searchable Symmetric Encryption (DSSE) scheme that supports update operations (i.e., file addition and deletion). The scheme relies on an inverted index approach with multiple (encrypted) linked lists, which achieves optimal $O(r)$ search time. However, update operations leak *significant* information (e.g., which trapdoors are associated with an added or deleted file). That is, the encrypted data structures do not serve their purposes for newly added or deleted files. Moreover, the scheme can not be parallelized as in [7]. Kamara et al. in [14] proposed another DSSE scheme, which relies on red-black trees as the main data structure. This scheme leaks much less information compared to that of [15] and also achieves *parallel* search and update operations. However, unlike [15], this scheme requires interactive updates. Moreover, it incurs in significant server storage overhead due to its very large encrypted index size.

Recently, a series of new DSSE schemes (i.e., [23,2,18,19]) have been proposed by achieving better index size and update time with less information leakage compared to [14,15]. While being asymptotically better, those schemes also have drawbacks. The scheme in [23] requires high storage overhead at the client side (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. This scheme also requires significant amount of data storage (e.g., 1600 bits) for per keyword-file pair at the server side. The scheme in [2] extends the work in [3] that focuses on boolean queries. While showing asymptotically better performance compared to [14], it leaks more information compared to [23] also incurring in non-negligible server storage. Notice that the data structure in this work grows linearly with the number of deletion operations, which requires re-encrypting the data structure eventually. The scheme in [18] uses a different approach from all the aforementioned alternatives, in which the server does not perform any processing, but just acts as a storage and transmission entity. The scheme relies on a primitive called "blind-storage". While the scheme shows good performance, it requires higher interaction than its counterparts, which may introduce response delays for distributed client-server systems. The scheme leaks less information than that of [2], but only support single keyword queries. The scheme can add/remove

a file but it cannot update the content of a file. Because of all these significant differences, we have not included it in Table 1. The scheme presented in [19] supports conjunctive queries but its update functionality is not oblivious[5]. The authors solve the leakage problem due to non-oblivious updates by periodically reencrypting the entire index. A major difference, with respect to our work and other previous work, is that the scheme in [19] requires three parties (one of them acts as a semi-trusted party) instead of two. Their performance numbers are also hard to compare to ours given that [19] uses server hardware as opposed to our implementation on a commodity platform.

Alternative Approaches. Information processing on encrypted data can be achieved via general purpose solutions such as Fully-Homomorphic Encryption (FHE) [9]. However, despite several improvements (e.g., [20]), FHE remains extremely computational and storage costly, and therefore it is considered impractical today. Another general purpose solution is Oblivious RAM (ORAM) [11], which can be used as a black-box to achieve a strong level of security for encrypted searches (the server learns nothing apart from the size of the file collection). Recent work (e.g., [22]) significantly improved the performance of ORAM. However, ORAM solutions are still communication and/or storage intensive [23].

3 Preliminaries and Models

We first give basic notation, primitives and definitions used by our scheme. We then provide the leakage and security models to prove security of our scheme.

Notation. Operators $||$ and $|x|$ denote the concatenation operation and the bit length of variable x, respectively. $x \xleftarrow{\$} \mathcal{S}$ denotes that variable x is randomly and uniformly selected from set $\mathcal{S}$. For any integer l, $(x_0, \ldots, x_l) \xleftarrow{\$} \mathcal{S}$ means $(x_0 \xleftarrow{\$} \mathcal{S}, \ldots, x_l \xleftarrow{\$} \mathcal{S})$. $|\mathcal{S}|$ 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 $\langle q_1, \ldots, q_n \rangle$. Given a bit a, $\overline{a}$ means the complement of a. Variable κ is an integer and it is used to denote the security parameter. $\log x$ means $\log_2 x$. Given a matrix I, $I[*, j]$ and $I[i, *]$ mean accessing all elements in the j'th column and the i'th row of I, respectively. $I[i, *]^T$ denotes the transpose of the i'th row of I.

Basic Cryptographic Primitives. 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

[5] This is explicitly noted in [19] on page 8, footnote 2.

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 $k_2 \xleftarrow{\$} \{0,1\}^{\kappa}$ and a string x, and returns a token τ. Similarly, $G : \{0,1\}^{\kappa} \times \{0,1\}^{*} \rightarrow \{0,1\}^{\kappa}$ is a keyed PRF denoted as $r \leftarrow G_{k_3}(x)$, which takes as input a secret key $k_3 \leftarrow \{0,1\}^{\kappa}$ and a string x and returns a key r. We denote by $H : \{0,1\}^{|x|} \rightarrow \{0,1\}$ a Random Oracle (RO) [1], which takes an input x and returns a bit as output.

3.1 Definitions

We first define generic DSSE algorithms and then describe specific data structures used by our scheme. We follow the definitions of [15,14] with some modifications: f_{id} and w denote a file with unique identifier id and a unique (key)-word 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 (also called database in the literature) and encrypted index, respectively.

Definition 1. *A Dynamic Searchable Symmetric Encryption (DSSE) scheme is a tuple of nine polynomial-time algorithms* $\mathsf{DSSE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{SrchToken}, \mathsf{Search}, \mathsf{AddToken}, \mathsf{Add}, \mathsf{DeleteToken}, \mathsf{Delete})$ *such that:*

1. $K \leftarrow \mathsf{Gen}(1^{\kappa})$*: is a probabilistic algorithm that takes as input a security parameter* κ *and outputs a secret key* K*.*
2. $(\gamma, \boldsymbol{c}) \leftarrow \mathsf{Enc}_K(\delta, \boldsymbol{f})$*: is a probabilistic algorithm that takes as input a secret key* K*, an index* δ *and files* $\boldsymbol{f}$*, from which* δ *constructed. It outputs encrypted index* γ *and ciphertexts* $\boldsymbol{c}$*.*
3. $f_j \leftarrow \mathsf{Dec}_K(c_j)$*: is a deterministic algorithm that takes as input a secret key* K *and a ciphertext* c_j *and outputs a file* f_j*.*
4. $\tau_w \leftarrow \mathsf{SrchToken}(K, w)$*: is a (possibly probabilistic) algorithm that takes as input a secret key* K *and a keyword* w*. It outputs a search token* τ_w*.*
5. $\mathbf{id_w} \leftarrow \mathsf{Search}(\tau_{\mathbf{w}}, \gamma)$*: is a deterministic algorithm that takes as input a search token* τ_w *and an encrypted index* γ*. It outputs identifiers* $\mathbf{id_w} \subseteq \boldsymbol{c}$*.*
6. $\tau_f \leftarrow \mathsf{AddToken}(K, f_{id})$*: is a (possibly probabilistic) algorithm that 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*.*
7. $(\gamma', \boldsymbol{c'}) \leftarrow \mathsf{Add}(\gamma, \boldsymbol{c}, \tau_f)$*: is a deterministic algorithm that takes as input an encrypted index* γ*, ciphertexts* $\boldsymbol{c}$*, an addition token* τ_f*. It outputs a new encrypted index* γ' *and ciphertexts* $\boldsymbol{c'}$*.*

8. $\tau'_f \leftarrow \mathsf{DeleteToken}(K, f)$: *is a (possibly probabilistic) algorithm that 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. $(\gamma', \boldsymbol{c'}) \leftarrow \mathsf{Delete}(\gamma, \boldsymbol{c}, \tau'_f)$: *is a deterministic algorithm that 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. *Let* DSSE *be a dynamic SSE scheme consisting of the tuple of nine algorithm as given in Definition 1. 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 $\mathbf{id_w}$.*

3.2 Leakage and Security Models

Most known efficient SSE schemes (e.g., [10,15,3,14,23,2,18]) reveal the *access and search patterns* that are defined below.

Definition 3. Search pattern $\mathcal{P}(\delta, Query, t)$ *is defined as follows: Given search query* $\mathsf{Query} = w$ *at time t, the search pattern 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. Access pattern $\Delta(\delta, \boldsymbol{f}, w_i, t)$ *is defined as follows: Given* $\mathsf{Query} = w$ *at time t, the access pattern is identifiers $\mathbf{id_w}$ of files $\boldsymbol{f}$, in which w appears.*

To capture information leak during the execution of DSSE algorithms, we follow the approach of [15,14,23,2] by defining leakage functions that capture what is being leaked by the scheme. Specifically, we consider the following leakage functions, in the line of [14] that captures dynamic file addition and deletion in its security model as we do, but we leak less information compared to [14] as discussed in Section B.

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

1. $(m, n, \mathbf{id_w}, \langle |\mathbf{f_{id_1}}|, \ldots, |\mathbf{f_{id_n}}| \rangle) \leftarrow \mathcal{L}_\mathbf{1}(\delta, \boldsymbol{f})$: *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 $\mathbf{id_w} = (\mathbf{id_1}, \ldots, id_n)$ of $\boldsymbol{f}$ and the size of each file $|f_{id_j}|, 1 \leq j \leq n$ (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.*

Remark 1. In our definition of security model below, we adapt the notion of *dynamic CKA2-security* from [14], which captures the file addition and deletion operations by simulating corresponding tokens τ_f and τ'_f, respectively (see Theorem 1 in Appendix A.2 for further details). Hence, our security model captures possible leakage due to update operations. Note that unlike [14], our main scheme is non-interactive and leaks less information (see Section B).

Definition 6. *Let* DSSE *be a DSSE scheme consisting of the tuple of nine algorithms as defined in Definition 1. 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* $\tau_f \leftarrow \mathsf{AddToken}(K, f_{id})$ *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* $(\gamma, \boldsymbol{c})$ *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 adaptive 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*

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

4 Our Scheme

In this section, we introduce our main scheme. We first describe our new data structure and then introduce the algorithms which make up the scheme. In Section 6, we also discuss variants and optimizations of the main scheme.

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[6] 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

[6] 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.

use static hash tables to uniquely 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 describe our main scheme in detail. Our DSSE scheme is comprised of nine algorithms (see Definition 1) (Gen, Enc, Dec, $\mathsf{SrchToken}$, Search, $\mathsf{AddToken}$, Add, $\mathsf{DeleteToken}$, Delete) defined as follows:

$K \leftarrow \mathsf{Gen}(1^\kappa)$: The client generates private keys $k_1 \leftarrow \mathcal{E}.\mathsf{Gen}(1^\kappa)$, $(k_2, k_3) \xleftarrow{\$} \{0,1\}^\kappa$ and sets $K \leftarrow (k_1, k_2, k_3)$.

$(\gamma, \mathbf{c}) \leftarrow \mathsf{Enc}_K(\delta, \mathbf{f})$: The client generates $(\gamma, \mathbf{c})$ as follows:

1. Extract all 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 δ to 0.
2. Construct δ for $j = 1, \ldots, n'$ and $i = 1, \ldots, m'$:
   (a) $s_{w_i} \leftarrow F_{k_2}(w_i)$ and $x_i \leftarrow T_w(s_{w_i})$.
   (b) $s_{f_j} \leftarrow F_{k_2}(id_j)$ and $y_j \leftarrow T_f(s_{f_j})$.
   (c) If w_i appears in f_j set $\delta[x_i, y_j] \leftarrow 1$.
3. Encrypt δ 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) $r_i \leftarrow G_{k_3}(i || \overline{st}_i)$, 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 $st_j \leftarrow T_f[j].st$.
   (d) $I[i,j].st \leftarrow 0$.
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 search token τ_w for w as follows:

1. $s_{w_i} \leftarrow F_{k_2}(w)$, $i \leftarrow T_w(s_{w_i})$, $\overline{st}_i \leftarrow T_w[i].st$.
2. $r_i \leftarrow G_{k_3}(i || \overline{st}_i)$.
3. 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 $\tau_w \leftarrow (i, r_i, \overline{r}_i)$.
4. $T_w[i].st \leftarrow \overline{st}_i + 1$.
5. Output τ_w. The client sends τ_w to the server.

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

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. $I[i,*].st \leftarrow 0$.
3. Set $l' \leftarrow 1$ and for each j satisfies $I'[i,j] = 1$, set $y_{l'} \leftarrow j$ and $l' \leftarrow l' + 1$.
4. 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 $l \leftarrow l' - 1$.
5. 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 $\gamma \leftarrow (I, T_w)$ [7].

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

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 $\{\overline{I}[i] \leftarrow 0\}_{i=1, i \notin \{x_1, \ldots, x_t\}}^{m}$.
5. $I'[i] \leftarrow \overline{I}[i] \oplus H(r_i||j||st_j)$ for $i = 1, \ldots, m$.
6. $c \leftarrow \mathcal{E}.\mathsf{Enc}_{k_1}(f_{id_j})$.
7. Output $\tau_f \leftarrow (I', j)$. The client sends (τ_f, c) to the server.

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

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

$\tau'_f \leftarrow \mathsf{DeleteToken}(K, f)$: The client generates τ'_f for f as follows:

1. Execute steps (1-2) of $\mathsf{AddToken}$ algorithm, which produce (j, r_i, st_j).
2. $I'[i] \leftarrow H(r_i||j|st_j)$ for $i = 1, \ldots, m$ [8].
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 as follows:

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

Keyword update for existing files: Some existing alternatives (e.g., Naveed et. al. [18]) only permit adding a new file or deleting an old file, but do not permit updating keywords in an existing file. Our scheme enables keyword update in an existing file. That is, to update an existing file f by adding new keywords or removing existing keywords, the client just 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 $\mathsf{AddToken}$, step 4). The rest of the algorithm is similar to $\mathsf{AddToken}$ algorithm.

Variants: We developed several variants of our main scheme, which are further discussed in Section 6 and Appendix C.

[7] This provides privacy if the server is compromised by an outsider after a search operation occurs (the server deletes r_i from the memory after the step 5 is completed). It also keeps I consistent for consecutive search operations performed on the same keyword.

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

5 Security Analysis

We prove that our main scheme is secure with the following theorem (it is straightforward to extend the proof for our variant schemes):

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 give the proof of correctness of our scheme in Appendix A.1. The security proof and simulators are presented in Appendix A.2.

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[9].) For cryptographic primitives, we chose to use the libtomcrypt cryptographic toolkit version 1.17 [8] 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 [13]. 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.

Our cryptographic primitives were chosen as follows. For file encryption we chose 128-bit CCM and for encrypting our data structure we used AES-128 CMAC. Key generation was implemented using the expand-then-extract key generation paradigm analyzed in [16]. 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, its security properties are well-understood, and it is standardized. Our use of CMAC as the PRF for the key derivation function is also standardized [6]. Our random oracles were all implemented via 128-bit AES CMAC. For hash tables, we took advantage of Google's freely available C++ sparse hash map implementation [12] 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.

Performance Comparison. Our experiments were performed using the Enron database of emails as in [15]. Table 2 summarizes results for three types of experiments: (i) cases were we have large number of files and large number of keywords, (ii) cases where we have large number of files but comparatively

[9] 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. [15].

small number of keywords and (iii) cases where we have 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 [15] by about two orders of magnitude and are comparable to the experiments in [23,2]. One key observation is that in contrast to [23,2], 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 [15]. 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. We observe that we still have room for improvement since have not taken advantage of parallelization.

Table 2. Performance of our DSSE scheme operations. w.: # of words, f.: # of files

Operation	**Time (msec)**					
	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 sec		52 min.		352 msec	
Encrypt searchable representation	493 msec		461 msec		823 msec	
Search and Update Operations (online, after initialization)						
Search for single key word	0.3 msec		10 msec		0.02 msec	
Add file to database	472 msec		8.83 msec		2.77 sec	
Delete file from database	329 msec		8.77 msec		2.36 sec	

Functionality, Security, and Data Structure Comparison. Compared to Kamara et al. in [15] scheme, which relies on an inverted index approach with multiple linked lists and achieves optimal $O(r)$ search time, our scheme has linear search time, uses an inverted index approach with a simple matrix-based data structure (augmented with hash tables for fast retrieval) but in contrast we achieve completely oblivious update operations. Moreover, the [15] can not be parallelized, whereas our scheme can. Kamara et al. [14] relies on red-black trees as the main data structure, achieves *parallel* search and oblivious update operations. However, it requires interactive updates and incurs in significant server storage overhead due to its very large encrypted in-

dex size. The scheme of Stefenavo et al. [23] requires high storage overhead at the client side (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. [23] 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[10]). The scheme in [2] it leaks more information compared to [23] also incurring in non-negligible server storage. The data structure in [2] 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 additions, deletions, or updates. The scheme in [18] requires higher interaction than its counterparts, which may introduce response delays for distributed client-server architectures, it leaks less information than that of [2], but only support single keyword queries. The scheme can add/remove a file but it cannot update the content of a file in contrast to our scheme.

6.1 Variants and Optimizations

We discuss some variants and trade-offs in our scheme, which can result in significant performance improvements.

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 of operation for the encryption function $\mathcal{E}$. Assume that the block size of $\mathcal{E}$ is b. We interpret columns of I as $d = \lceil \frac{n}{b} \rceil$ blocks with size of b bits each. We 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 operation. A state bit $T_f[a_l].b$ is stored to keep track the update status of its corresponding block. Notice that the update status is maintained only for each block but not for each bit of $I[i,j]$. Hence, in this variant, the matrix I is a just binary matrix (unlike the main scheme, in which $I[i,j] \in \{0,1\}^2$). AddToken and Add algorithms for the aforementioned variant are as follows (DeleteToken and Delete follow the similar principles):

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

[10] 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 [23]. Observe that for fixed m-words, we need a hash table with approximately $2m$ entries, even if each entry was represented by 80-bits.

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 $st_l \leftarrow T_f[a_l].st$.
2. 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 $\overline{st}_i \leftarrow T_w[i].st$ [11].
   (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) $(\overline{I}[i, a_l], \ldots, \overline{I}[i, a_{l+1} - 1]) \leftarrow \mathcal{E}.\mathsf{Dec}_{r_i}(I[i, a_l], \ldots, I[i, a_{l+1} - 1], st_l)$.
3. Set $\overline{I}[x_i, j] \leftarrow 1$ for $i = 1, \ldots, t$ and $\{\overline{I}[i, j] \leftarrow 0\}_{i=1, i \notin \{x_1, \ldots, x_t\}}^{m}$.
4. $gc \leftarrow gc + 1$, $T_f[a_l].st \leftarrow gc$, $st_l \leftarrow T_f[a_l].st$ and $T_f[a_l].b \leftarrow 1$.
5. $(I'[i, a_l], \ldots, I'[i, a_{l+1} - 1]) \leftarrow \mathcal{E}.\mathsf{Enc}_{r_i}(\overline{I}[i, a_l], \ldots, \overline{I}[i, a_{l+1} - 1], st_l)$ for $i = 1, \ldots, m$.
6. $c \leftarrow \mathcal{E}.\mathsf{Enc}_{k_1}(f_{id_j})$.
7. Output $\tau_f \leftarrow (I', j)$. The client sends (τ_f, c) to the server.

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

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 $\gamma' \leftarrow (I, T_f)$, $\mathbf{c}'$ is obtained by adding (c, j) 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 Δt is introduced due to interaction. We discuss three other variants of our main scheme in the appendix.

In Appendix A.2, we prove security of the main scheme and extend the proof to this variant. We observe that encrypting multiple columns using an

[11] 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).

IND-CPA encryption scheme does not leak additional information during updates. In particular, the re-encryption of the b-bit block happens on the *client* side at the cost of one additional and crucial round of interaction.

6.2 Limitations

The search time of our scheme is $O(\frac{m}{p \cdot b})$ and is practical as shown by the estimated execution times even for very large N(e.g., $N = 10^{11}$ for $m = 10^5$, $n = 10^6$). However, it is asymptotically less efficient than that of [14,23,2] schemes. Observe that we gain the highest level of privacy, low client storage, dynamic keyword universe with secure updates and bandwidth efficiency in exchange. Observe that our scheme preserves its practicality in terms of search execution time as discussed (being comparably efficient to [15,14,23,2]), while gaining all these advantages. Another limitation is that our scheme assumes a conservative upper bound on the maximum number of keywords and files to be used in the system (as in [14] but unlike [23,2]). However, we gain low server storage overhead for large number of file-keyword pairs (i.e., two-bits storage overhead for per file-keyword pair) in exchange. Moreover, our scheme focuses on single keyword search as in [15,14,23,18] (but unlike [2] for boolean queries, with an exchange of more information leak). In practice the requirement of having to specify the upper bound in the number of files can be work around in several ways. The most straightforward way is to define a second index data structure once the upper limit is reached in the first one. This would not be unlike requesting an additional x GB of storage in quota-based systems. An alternative is to simply retrieve the index data structure stored in the server and re-encrypt it. This would have the added advantage of "erasing" the server history and any leakage associated with it. This approach is used in ORAM to make the system oblivious and it has been proposed in other DSSE schemes as a way to reduce leakage or to re-claim storage space [2].

References

1. M. Bellare and P. Rogaway, "Random oracles are practical: A paradigm for designing efficient protocols," in *Proceedings of the 1st ACM conference on Computer and Communications Security (CCS '93)*. NY, USA: ACM, 1993, pp. 62–73.
2. D. Cash, J. Jaeger, S. Jarecki, C. Jutla, H. Krawcyk, M.-C. Rosu, and M. Steiner, "Dynamic searchable encryption in very-large databases: Data structures and implementation," in *21th Annual Network and Distributed System Security Symposium — NDSS 2014*. The Internet Society, February 23-26, 2014.
3. D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner, "Highly-scalable searchable symmetric encryption with support for boolean queries," in *Advances in Cryptology, CRYPTO 2013*, ser. Lecture Notes in Computer Science, vol. 8042. Springer Berlin Heidelberg, 2013, pp. 353–373.
4. Y.-C. Chang and M. Mitzenmacher, "Privacy preserving keyword searches on remote encrypted data," in *Proceedings of the Third International Conference on Applied Cryptography and Network Security (ACNS)*, ser. Lecture Notes in Computer Science, vol. 3531. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 442–455.

5. M. Chase and S. Kamara, "Structured encryption and controlled disclosure," in *Advances in Cryptology - ASIACRYPT 2010*, ser. Lecture Notes in Computer Science, vol. 6477, 2010, pp. 577–594.
6. L. Chen, "Nist special publicatin 800-108: Recomendation for key derivation using pseudorandom functions (revised)," National Institute of Standards and Technology. Computer Security Division, Tech. Rep. NIST-SP800-108, October 2009, available at http://csrc.nist.gov/publications/nistpubs/800-108/sp800-108.pdf.
7. R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: improved definitions and efficient constructions," in *Proceedings of the 13th ACM conference on Computer and communications security*, ser. CCS '06. New York, NY, USA: ACM, 2006, pp. 79–88.
8. T. S. Denis, "LibTomCrypt library," Available at http://libtom.org/?page=features&newsitems=5&whatfile=crypt, Released May 12th, 2007.
9. C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the 41st annual ACM symposium on Theory of computing*, ser. STOC '09. New York, NY, USA: ACM, 2009, pp. 169–178.
10. E.-J. Goh, "Secure indexes," Cryptology ePrint Archive, Report 2003/216, 2003, http://eprint.iacr.org/.
11. O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," *J. ACM*, vol. 43, no. 3, pp. 431–473, 1996.
12. google sparsehash@googlegroups.com, "sparsehash: An extemely memory efficient hash_map implementation," Available at https://code.google.com/p/sparsehash/, February 2012.
13. S. Gueron, "White Paper: Intel Advanced Encryption Standard (AES) New Instructions Set," Available at https://software.intel.com/sites/default/files/article/165683/aes-wp-2012-09-22-v01.pdf. Software Library available at https://software.intel.com/sites/default/files/article/181731/intel-aesni-sample-library-v1.2.zip, Document Revision 3.01, September 2012.
14. S. Kamara and C. Papamanthou, "Parallel and dynamic searchable symmetric encryption," in *Financial Cryptography and Data Security (FC)*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, vol. 7859, pp. 258–274.
15. S. Kamara, C. Papamanthou, and T. Roeder, "Dynamic searchable symmetric encryption," in *Proceedings of the 2012 ACM conference on Computer and communications security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 965–976.
16. H. Krawczyk, "Cryptographic extraction and key derivation: The HKDF scheme," in *Advances in Cryptology - CRYPTO 2010*, ser. LNCS, T. Rabin, Ed., vol. 6223. Springer, August 15-19, 2010, pp. 631–648.
17. K. Kurosawa and Y. Ohtaki, "UC-secure searchable symmetric encryption," in *Financial Cryptography and Data Security (FC)*, ser. Lecture Notes in Computer Science, vol. 7397. Springer Berlin Heidelberg, 2012, pp. 285–298.
18. M. Naveed, M. Prabhakaran, and C. A. Gunter, "Dynamic searchable encryption via blind storage," in *35th IEEE Symposium on Security and Privacy*, May 2014, pp. 48–62.
19. V. Pappas, F. Krell, B. Vo, V. Kolesnikov, T. Malkin, S. G. Choi, W. George, A. D. Keromytis, and S. Bellovin, "Blind seer: A scalable private DBMS," in *2014 IEEE Symposium on Security and Privacy, SP 2014*. IEEE Computer Society, May 18-21, 2014, pp. 359–374.
20. N. P. Smart and F. Vercauteren, "Fully homomorphic simd operations," *Des. Codes Cryptography*, vol. 71, no. 1, pp. 57–81, 2014.

21. D. X. Song, D. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, ser. SP '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 44–55.
22. E. Stefanov and E. Shi, "Oblivistore: High performance oblivious cloud storage," in *Security and Privacy (SP), 2013 IEEE Symposium on*, May 2013, pp. 253–267.
23. E. Stefanov, C. Papamanthou, and E. Shi, "Practical dynamic searchable encryption with small leakage," in *21st Annual Network and Distributed System Security Symposium — NDSS 2014*. The Internet Society, February 23-26, 2014.

A Proofs

We first provide the correctness argument for our main scheme followed by several variants of it. We then provide the formal proof of our main scheme.

A.1 Proof of Correctness of the DSSE Scheme

The correctness argument for our main scheme is as follows:

Lemma 1. *(Correctness) The DSSE scheme presented above is correct according to Definition 2.*

Proof: The correctness and consistency of the above scheme is guaranteed 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 SrchToken and AddToken increase the 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 as $H(r_i||j||st_j)$ with row key r_i, column position j and the counter of column j. Note that 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 operation is followed by an update operation, 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 all state bits $I[*,j]$ to 1 (indicating cells are updated) for the consistency.

The rest is to show that algorithms SrchToken and Search produce correct search results. If keyword w is searched for the first time, the algorithm 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 (i.e., step 3). 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 (i.e., update). 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 (i.e., step 5). □

A.2 Proof of Security

We prove that our main scheme achieves *adaptive security against chosen-keyword attacks (CKA2)* (with secure update operations as defined in Definition 6) as below. Note that our scheme is secure in the Random Oracle Model (ROM) [1]. 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 value $\leftarrow \mathcal{LR}$(key) and $\perp \leftarrow \mathcal{LR}$(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*: Function $b \leftarrow RO(x)$ takes an input x and returns a bit b as output. Given input x, if $\perp = \mathcal{LR}(x)$ then 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 $(\gamma, \boldsymbol{c})$: 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. $s_{f_j} \xleftarrow{\$} \{0,1\}^\kappa$, $y_j \leftarrow T_f(s_{f_j})$ and insert (id_j, s_{f_j}, y_j) into $\mathcal{LH}$, for $j = 1,\ldots,n'$.
2. $c_{y_j} \leftarrow \mathcal{E}.\mathsf{Enc}_k(\{0\}^{|c_{id_j}|})$, where $k \xleftarrow{\$} \{0,1\}^\kappa$ for $j = 1,\ldots,n'$.
3. 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 $I[i,j].st \leftarrow 0$.

4. 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: Assume that 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 information 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 $\perp = \mathcal{LK}(i, \overline{st}_i)$ then $r_i \leftarrow \{0,1\}^\kappa$ and insert $(r_i, i, \overline{st}_i)$ into $\mathcal{LK}$. Else, $r_i \leftarrow \mathcal{LK}(i, \overline{st}_i)$.
3. If $\overline{st}_i > 1$ then $\overline{r}_i \leftarrow \mathcal{LK}(i||\overline{st}_i - 1)$ and $\tau_w \leftarrow (i, r_i, \overline{r}_i)$. 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 $\mathbf{id_w} = (\mathbf{y_1}, \ldots, y_l)$. 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 $\mathbf{id_w} \leftarrow \mathsf{Search}(\tau_\mathbf{w}, \gamma)$. $\mathcal{S}$ 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): Assume that $\mathcal{S}$ 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:

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

□

Remark 2. Extending the proof to variant I presented in Section 6.1 is straightforward[12]. 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.

B Discussion on Privacy Levels

The leakage definition and formal security model described in Section 3 imply various levels of privacy for different DSSE schemes. We summarize some important privacy notions (based on the various leakage characteristics discussed in [14,23,2,18]) with different levels of privacy as follows:

• *Size pattern*: It refers to the current number of file-keyword pairs stored in the system.

• *Forward privacy*: It refers that a search on a keyword w does not leak the identifiers of files matching this keyword for (pre-defined) future files.

• *Backward privacy*: It refers that a search on a keyword w does not leak the identifiers of files matching this keywords that were previously added but then deleted (leaked though additional information 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[13]. 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.

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

[13] 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.

- 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 [23]).
- 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 [2]).
- Level-4 (*L4*) leaks *L3* plus the information whether the same keyword added or deleted from two files (e.g., as in [14]).
- 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 [15]).

The work of Naveed et al. [18] leaks the number of keywords that are common to all the files in a given subset. The search reveals "removed" versions of the files in the search results. However, it does not reveal actual file identifier during the search. Observe that, in addition to achieving CKA2-security (defined below), our construction 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.

C Additional Variants of the Main Scheme

We discuss further variants of our main scheme as below.

Variant-II: Trade-off between storage and communication overhead. The storage overhead of T_w and T_f are practically small, and therefore in our main scheme, we assume that the client stores them to maximize the update and search efficiency for an exchange of a small storage overhead. For instance, $(N = 10^{11}, m = 10^5, n = 10^6)$, the approximate storage overhead is around 6MB [14].

It is easy to avoid storing T_f at the client, which requires three message exchanges, only for update operations, with a very small communication overhead: In AddToken, given s_{f_j}, the server returns (j, st_j) to the client and the client follows the AddToken algorithm as described in the main scheme. This twist requires transmitting $\log_2(n) + |st_j|$ bits (e.g., approximately 52 bits for $n = 10^6$ and 32-bit counter) from server to the client, but reduces the client storage up to only *1MB* (which is the overhead of T_w for $m = 10^5$). Notice that this storage requirement is plausible even for resource-constrained devices such as mobile phones[15]. It is also possible to avoid storing T_w at the client by accepting further communication overhead. That is, the client encrypts T_w and stores it at the server. Whenever a search or update is required, the client retrieve T_w, decrypts it and then follows the main scheme.

[14] The client can truncate s_{f_j} and s_{w_i} (e.g., 40 bits) to further reduce the storage overhead (but with a security trade-off).

[15] The client may synchronize the encrypted version of T_w with the server from time to time to ensure that it is backed up regularly (this can be done rarely so that its communication overhead will be negligible). In any case, the client can always recover T_w from I (which is always stored at the server), since the client knows the private key.

Variant-III: Alternative Deletion Algorithm. File deletion can be simplified by adding just another bit to each element of T_f, which is denoted as $T_f[j].d, j = 1, \ldots, n$. Assume that file f_{id} with corresponding index j is needed to be deleted. The client sets $T_f[j].d \leftarrow 0$ and then sends index j to the server. The server also sets $T_f[j].d \leftarrow 0$. During a search operation, the server simply omits the column values whose deletion bits are zero. If file f_{id} is added in the future, we follow Add and AddToken algorithms by also setting $T_f[j].d \leftarrow 1$. Notice that in this variant, the update operations directly leak the type of update operation (see Section 3.2).

Variant-IV: Reducing Row Encryption Cost. In all variants, the last step of Search algorithm re-encrypts row i, which was decrypted during the search operation. As discussed, this step protects I against server breaches (e.g., if the server is compromised at time t, the attacker cannot learn past search results conducted before t)[16]. It is possible to reduce the computational overhead of re-encryption: Given row i, only the matrix cells that were encrypted with row key $\overline{r}_i$ are re-encrypted. Hence, the server can simply keep a copy of row i before the decryption operation and re-use matrix cells that are encrypted with row key r_i (and delete the copy).

Support for Parallelization: The main scheme and all its variants can be easily parallelized, since both search and update operations involve bit operations between independent vector positions. That is, (F, G, H) and $\oplus$ operations can be paralelly executed by p different processors.

[16] The overhead of re-encryption has not been included in the performance comparison, since all the compared schemes need to re-encrypt their data structures against such server breaches.