Dynamic Searchable Symmetric Encryption

Seny Kamara * Charalampos Papamanthou † Tom Roeder ‡

Abstract

Searchable symmetric encryption (SSE) allows a client to encrypt its data in such a way that this data can still be searched. The most immediate application of SSE is to cloud storage, where it enables a client to securely outsource its data to an untrusted cloud provider without sacrificing the ability to search over it.

SSE has been the focus of active research and a multitude of schemes that achieve various levels of security and efficiency have been proposed. Any *practical* SSE scheme, however, should (at a minimum) satisfy the following properties: sublinear search time, security against adaptive chosen-keyword attacks, compact indexes and the ability to add and delete files efficiently. Unfortunately, none of the previously-known SSE constructions achieve all these properties at the same time. This severely limits the practical value of SSE and decreases its chance of deployment in real-world cloud storage systems.

To address this, we propose the first SSE scheme to satisfy all the properties outlined above. Our construction extends the inverted index approach (Curtmola et al., *CCS 2006*) in several non-trivial ways and introduces new techniques for the design of SSE. In addition, we implement our scheme and conduct a performance evaluation, showing that our approach is *highly* efficient and ready for deployment.

1 Introduction

Searchable symmetric encryption (SSE) allows a client to encrypt data in such a way that it can later generate search tokens to send as queries to a storage server. Given a token, the server can search over the encrypted data and return the appropriate encrypted files. Informally, a SSE scheme is secure if: (1) the ciphertext alone reveals no information about the data; (2) the ciphertext together with a search token reveals at most the result of the search; (3) search tokens can only be generated using the secret key.

The most immediate application of SSE is to the design of searchable cryptographic cloud storage systems (see [19] for a discussion) which can provide end-to-end security for cloud storage systems without sacrificing utility. Other applications include the design of graph encryption schemes and controlled disclosure mechanisms [6].

In an *index-based* SSE scheme [15, 5, 8, 23, 6] the encryption algorithm takes as input an index δ and a sequence of n files $\mathbf{f} = (f_1, \ldots, f_n)$ and outputs an encrypted index γ and a sequence of n ciphertexts $\mathbf{c} = (c_1, \ldots, c_n)$. All known constructions [15, 5, 8, 23, 6] can encrypt the files $\mathbf{f}$ using any symmetric encryption scheme, i.e., the file encryption does not depend on any unusual properties of the encryption scheme.

To search for a keyword w, the client generates a search token τ_w and given τ_w, γ and $\mathbf{c}$, the server can find the identifiers $\mathbf{I}_w$ of the files that contain w. From these identifiers it can recover

*Microsoft Research. senyk@microsoft.com

†UC Berkeley. cpap@cs.berkeley.edu. Work done at Microsoft Research.

‡Microsoft Research. throeder@microsoft.com

Scheme	Dynamism	Security	Search time	Index size
SWP00 [22]	static	CPA	$O(\|\mathbf{f}\|)$	N/A
Z-IDX [15]	dynamic	CKA1	$O(\#\mathbf{f})$	$O(\#\mathbf{f})$
CM05 [5]	static	CKA1	$O(\#\mathbf{f})$	$O(\#\mathbf{f} \cdot \#W)$
SSE-1 [8]	static	CKA1	$O(\#\mathbf{f}_w)$	$O(\sum_w \#\mathbf{f}_w + \#W)$
SSE-2 [8]	static	CKA2	$O(\#\mathbf{f})$	$O(\#\mathbf{f} \cdot \#W)$
vLSDHJ10 [23]	dynamic	CKA2	$O(\log \#W)$	$O(\#W \cdot m_f)$
CK10 [6]	static	CKA2	$O(\#\mathbf{f}_w)$	$O(\#W \cdot m_f)$
KO12 [21]	static	UC	$O(\#\mathbf{f})$	$O(\#W \cdot \#\mathbf{f})$
this paper	dynamic	CKA2	$O(\#\mathbf{f}_w)$	$O(\sum_w \#\mathbf{f}_w + \#W)$

Table 1: Comparison of several SSE schemes. Search time is per keyword w and update time is per file f. $\mathbf{f}$ is the file collection, $|\mathbf{f}|$ is its bit length, $\#\mathbf{f}$ is the number of files in $\mathbf{f}$, $\#\mathbf{f}_w$ is the number of files that contain the keyword w, $\#W$ is the size of the keyword space and m_f is the maximum (over keywords) number of files in which a keyword appears.

the appropriate ciphertexts $\mathbf{c}_w$.[1] Notice that the provider learns some limited information about the client's query. In particular, it knows that whatever keyword is being searched for is contained in the files encrypted as $\mathbf{c}_w$. There are ways to hide even this information, most notably using the work of Goldreich and Ostrovsky [16] on oblivious RAMs, but such an approach leads to inefficient schemes.

Previous work. The problem of searching on symmetrically encrypted data can be solved in its full generality using the work of Goldreich and Ostrovsky [16] on oblivious RAMs. Unfortunately, this approach requires interaction and has a high overhead. Searchable encryption was first considered explicitly by Song, Wagner and Perrig in [22], where they give a non-interactive solution that achieves search time that is linear in the length of the file collection.

Formal security notions for SSE have evolved over time. The first notion, *security against chosen-keyword attacks* (CKA1) [15, 5, 8], guarantees that: (1) the encrypted index γ and the ciphertexts $\mathbf{c}$ do not reveal any information about $\mathbf{f}$ other than the number of files n and their length; and (2) the encrypted index γ and a token τ_w reveal at most the outcome of the search $\mathbf{I}_w$. It was observed in [8], however, that CKA1-security only provides security if the search queries are independent of $(\gamma, \mathbf{c})$ and of previous search results. To address this, the stronger notion of adaptive security against chosen-keyword attacks (CKA2) was proposed [8]. Recently, Kurosawa and Ohtaki put forth the even stronger notion of universally composable (UC) SSE [21] that, roughly speaking, guarantees security even when the scheme is used in arbitrary environments (e.g., when composed with itself and/or other cryptographic protocols and primitives).

While there are several CKA2-secure SSE schemes [8, 23, 6, 21], they all have limitations from a practical point of view. In particular, the constructions of [8] and [21] require $O(\#\mathbf{f})$ time to search, where $\#\mathbf{f}$ denotes the number of files in the collection. While the construction from [6] is asymptotically optimal and efficient in practice, the encrypted index can be very large. In addition, none of these three schemes are explicitly *dynamic*; that is, one cannot add or remove files without either re-indexing the entire data collection or making use of generic and relatively expensive dynamization techniques like the one used in [8]. As far as we know, the only SSE construction that is CKA2-secure and (explicitly) dynamic was proposed by van Liesdonk, Sedghi, Doumen, Hartel and Jonker [23]. In their scheme, search is logarithmic in the number of keywords which, for practical purposes, is likely to be efficient

[1] This is the structure-only formulation of SSE. We refer the reader to [6] for a discussion of other SSE formulations in the more general setting of structured encryption.

enough. The main limitations of the scheme, however, are that the size of the encrypted index is relatively large (roughly the same as the scheme from [6]).

Another line of work uses deterministic encryption [2, 1] to enable search on encrypted data with existing database and search techniques. This approach differs from SSE as it only provides security for data and queries that have high entropy. Starting with the work of Boneh, Di Crescenzo, Ostrovsky and Persiano [4], searchable encryption has also been considered in the public-key setting.

Table 1 summarizes the differences between our scheme and others that have appeared in the literature.

Our contributions. In this work, we focus on the problem of constructing *practical* SSE schemes for the purpose of designing practical searchable cryptographic cloud storage systems [19]. We make the following contributions:

1. We present a formal security definition for *dynamic* SSE. In particular, our definition captures a strong notion of security for SSE, which is *adaptive* security against chosen-keyword attacks (CKA2) [8].

2. We construct the first SSE scheme that is dynamic, CKA2-secure and achieves *optimal* search time. We note that, unlike previously known schemes [22, 15, 5, 8, 6], our construction is secure in the random oracle model.

3. We describe the first implementation and evaluation of an SSE scheme based on the *inverted index* approach of [8]. Our implementation shows that this type of SSE scheme can be extremely efficient.

4. We conduct a performance evaluation of our scheme that shows the incremental cost of adding confidentiality to a (searchable) cloud storage system.

2 Preliminaries and Notation

The set of all binary strings of length n is denoted as $\{0,1\}^n$, and the set of all finite binary strings as $\{0,1\}^*$. The notation $[n]$ represents the set of integers $\{1,\ldots,n\}$. We write $x \leftarrow \chi$ to represent an element x being sampled from a distribution χ, and $x \xleftarrow{\$} X$ to represent an element x being sampled uniformly at random from a set X. The output x of a probabilistic algorithm $\mathcal{A}$ is denoted by $x \leftarrow \mathcal{A}$ and that of a deterministic algorithm $\mathcal{B}$ by $x := \mathcal{B}$. Given a sequence of elements $\mathbf{v}$ we refer to its i^{th} element either as v_i or $\mathbf{v}[i]$ and to its total number of elements as $\#\mathbf{v}$. If S is a set then $\#S$ refers to its cardinality. W denotes the universe of words. If $f = (w_1,\ldots,w_m) \in W^m$ is a file, then $\#f$ denotes its total number of words and $|f|$ is its bit length. Also, $\bar{f}$ is the file that results from removing all duplicates from f (i.e., $\bar{f}$ contains only the unique words in f sequenced according to the order in which they first appear in f). If s is a string then $|s|$ refers to its bit length. We denote the concatenation of n strings $s_1,\ldots,s_n$ by $\langle s_1,\ldots,s_n\rangle$.

We use various data structures including linked lists, arrays and dictionaries. If $\mathtt{L}$ is a list then $\#\mathtt{L}$ denotes its total number of nodes. If $\mathtt{A}$ is an array then $\#\mathtt{A}$ is its total number of cells, $\mathtt{A}[i]$ is the value stored at location $i \in [\#\mathtt{A}]$ and $\mathtt{A}[i] := v$ denotes the operation that stores v at location i in $\mathtt{A}$. A dictionary (also known as a key-value store or associative array) is a data structure that stores key-value pairs (s,v). If the pair (s,v) is in $\mathtt{T}$, then $\mathtt{T}[s]$ is the value v associated with s. $\mathtt{T}[s] := v$ denotes the operation that stores the value v under search key s in $\mathtt{T}$ and $\#\mathtt{T}$ is the number of pairs in $\mathtt{T}$. We sometimes write $s \in \mathtt{T}$ to mean that there exists some pair in $\mathtt{T}$ with search key s.

Throughout, $k \in \mathbb{N}$ will denote the security parameter and we will assume all algorithms take k implicitly as input. A function $\nu : \mathbb{N} \to \mathbb{N}$ is negligible in k if for every positive polynomial $p(\cdot)$ and all sufficiently large k, $\nu(k) < 1/p(k)$. We write $f(k) = \mathsf{poly}(k)$ to mean that there exists a polynomial $p(\cdot)$ such that $f(k) \leq p(k)$ for all sufficiently large $k \in \mathbb{N}$; and we similarly write $f(k) = \mathsf{negl}(k)$ to mean that there exists a negligible function $\nu(\cdot)$ such that $f(k) \leq \nu(k)$ for all sufficiently large k. Two distribution ensembles χ and χ' are computationally indistinguishable if for all probabilistic polynomial-time (PPT) distinguishers $\mathcal{D}$, $|\Pr[\mathcal{D}(\chi) = 1] - \Pr[\mathcal{D}(\chi') = 1]| \leq \mathsf{negl}(k)$.

Basic cryptographic primitives. A private-key encryption scheme is a set of three polynomial-time algorithms $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ such that Gen is a probabilistic algorithm that takes a security parameter k and returns a secret key K; Enc is a probabilistic algorithm takes a key K and a message m and returns a ciphertext c; Dec is a deterministic algorithm that takes a key K and a ciphertext c and returns m if K was the key under which c was produced. Informally, a private-key encryption scheme is CPA-secure if the ciphertexts it outputs do not reveal any partial information about the plaintext even to an adversary that can adaptively query an encryption oracle.

In addition to encryption schemes, we also make use of pseudo-random functions (PRF) and permutations (PRP), which are polynomial-time computable functions that cannot be distinguished from random functions by any probabilistic polynomial-time adversary. We refer the reader to [20] for formal definitions of CPA-security, PRFs and PRPs.

3 Definitions

Recall from §1 that searchable encryption allows a client to encrypt data in such a way that it can later generate search tokens to send as queries to a storage server. Given a search token, the server can search over the encrypted data and return the appropriate encrypted files.

The data can be viewed as a sequence of n files $\mathbf{f} = (f_1, \ldots, f_n)$, where file f_i is a sequence of words $(w_1, \ldots, w_m)$ from a universe W. We assume that each file has a unique identifier $\mathsf{id}(f_i)$. The data is dynamic, so at any time a file may be added or removed. We note that the files do not have to be text files but can be any type of data as long as there exists an efficient algorithm that maps each document to a file of keywords from W. Given a keyword w we denote by $\mathbf{f}_w$ the set of files in $\mathbf{f}$ that contain w. If $\mathbf{c} = (c_1, \ldots, c_n)$ is a set of encryptions of the files in $\mathbf{f}$, then $\mathbf{c}_w$ refers to the ciphertexts that are encryptions of the files in $\mathbf{f}_w$.

A limitation of all known SSE constructions (including ours) is that the tokens they generate are deterministic, in the sense that the same token will always be generated for the same keyword. This means that searches leak statistical information about the user's search pattern. Currently, it is not known how to design efficient SSE schemes with probabilistic trapdoors.

Recall that we consider *dynamic* SSE so the scheme must allow for the addition and removal of files. Both of these operations are handled using tokens. To add a file f, the client generates an add token τ_a and given τ_a and γ, the provider can update the encrypted index γ. Similarly, to delete a file f, the client generates a delete token τ_d, which the provider uses to update γ.

Definition 3.1 (Dynamic SSE)**.** *A dynamic index-based SSE scheme is a tuple of nine polynomial-time algorithms* $\mathsf{SSE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{SrchToken}, \mathsf{AddToken}, \mathsf{DelToken}, \mathsf{Search}, \mathsf{Add}, \mathsf{Del}, \mathsf{Dec})$ *such that:*

> $K \leftarrow \mathsf{Gen}(1^k)$*: is a probabilistic algorithm that takes as input a security parameter k and outputs a secret key K.*
>
> $(\gamma, \mathbf{c}) \leftarrow \mathsf{Enc}(K, \mathbf{f})$*: is a probabilistic algorithm that takes as input a secret key K and a sequence of files $\mathbf{f}$. It outputs an encrypted index γ, and a sequence of ciphertexts $\mathbf{c}$.*

$\tau_s \leftarrow \mathsf{SrchToken}(K, w)$*: is a (possibly probabilistic) algorithm that takes as input a secret key* K *and a keyword* w*. It outputs a search token* τ_s*.*

$(\tau_a, c_f) \leftarrow \mathsf{AddToken}(K, f)$*: is a (possibly probabilistic) algorithm that takes as input a secret key* K *and a file* f*. It outputs an add token* τ_a *and a ciphertext* c_f*.*

$\tau_d \leftarrow \mathsf{DelToken}(K, f)$*: is a (possibly probabilistic) algorithm that takes as input a secret key* K *and a file* f*. It outputs a delete token* τ_d*.*

$\mathbf{I}_w := \mathsf{Search}(\gamma, \mathbf{c}, \tau_s)$*: is a deterministic algorithm that takes as input an encrypted index* γ*, a sequence of ciphertexts* $\mathbf{c}$ *and a search token* τ_s*. It outputs a sequence of identifiers* $\mathbf{I}_w \subseteq \mathbf{c}$*.*

$(\gamma', \mathbf{c}') := \mathsf{Add}(\gamma, \mathbf{c}, \tau_a, c)$*: is a deterministic algorithm that takes as input an encrypted index* γ*, a sequence of ciphertexts* $\mathbf{c}$*, an add token* τ_a *and a ciphertext* c*. It outputs a new encrypted index* γ' *and sequence of ciphertexts* $\mathbf{c}'$*.*

$(\gamma', \mathbf{c}') := \mathsf{Del}(\gamma, \mathbf{c}, \tau_d)$*: is a deterministic algorithm that takes as input an encrypted index* γ*, a sequence of ciphertexts* $\mathbf{c}$*, and a delete token* τ_d*. It outputs a new encrypted index* γ' *and new sequence of ciphertexts* $\mathbf{c}'$*.*

$f := \mathsf{Dec}(K, c)$*: is a deterministic algorithm that takes as input a secret key* K *and a ciphertext* c *and outputs a file* f*.*

A dynamic SSE scheme is correct if for all $k \in \mathbb{N}$, for all keys K generated by $\mathsf{Gen}(1^k)$, for all $\mathbf{f}$, for all $(\gamma, \mathbf{c})$ output by $\mathsf{Enc}(K, \mathbf{f})$, and for all sequences of add, delete or search operations on γ, search always returns the correct set of indices.

Intuitively, the security guarantee we require from a dynamic SSE scheme is that (1) given an encrypted index γ and a sequence of ciphertexts $\mathbf{c}$, no adversary can learn any partial information about the files $\mathbf{f}$; and that (2) given, in addition, a sequence of tokens $\boldsymbol{\tau} = (\tau_1, \ldots, \tau_n)$ for an adaptively generated sequence of queries $\mathbf{q} = (q_1, \ldots, q_n)$ (which can be for the search, add or delete operations), no adversary can learn any partial information about either $\mathbf{f}$ or $\mathbf{q}$.

This exact intuition can be difficult to achieve and most known efficient and non-interactive SSE schemes [15, 5, 8] reveal the access and search patterns.[2] We therefore need to weaken the definition appropriately by allowing some limited information about the messages and the queries to be revealed to the adversary. To capture this, we follow the approach of [8] and [6] and parameterize our definition with a set of *leakage functions* that capture precisely what is being leaked by the ciphertext and the tokens.

As observed in [8], another issue with respect to SSE security is whether the scheme is secure against *adaptive* chosen-keyword attacks (CKA2) or only against *non-adaptive* chosen keyword attacks (CKA1). The former guarantees security even when the client's queries are based on the encrypted index and the results of previous queries. The latter only only guarantees security if the client's queries are independent of the index and of previous results.

In the following definition, we extend the notion of CKA2-security from [8] to the dynamic setting.

Definition 3.2 (Dynamic CKA2-security)**.** *Let* $\mathsf{SSE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{SrchToken}, \mathsf{AddToken}, \mathsf{DelToken}, \mathsf{Search}, \mathsf{Add}, \mathsf{Del}, \mathsf{Dec})$ *be a dynamic index-based SSE scheme and consider the following probabilistic experiments, where* $\mathcal{A}$ *is a stateful adversary,* $\mathcal{S}$ *is a stateful simulator and* $\mathcal{L}_1$*,* $\mathcal{L}_2$*,* $\mathcal{L}_3$ *and* $\mathcal{L}_4$ *are stateful leakage algorithms:*

[2] Two exceptions are the work of Goldreich and Ostrovsky [16] which does not leak any information at all, and the SSE construction described in [6] which leaks only the access and the intersection patterns.

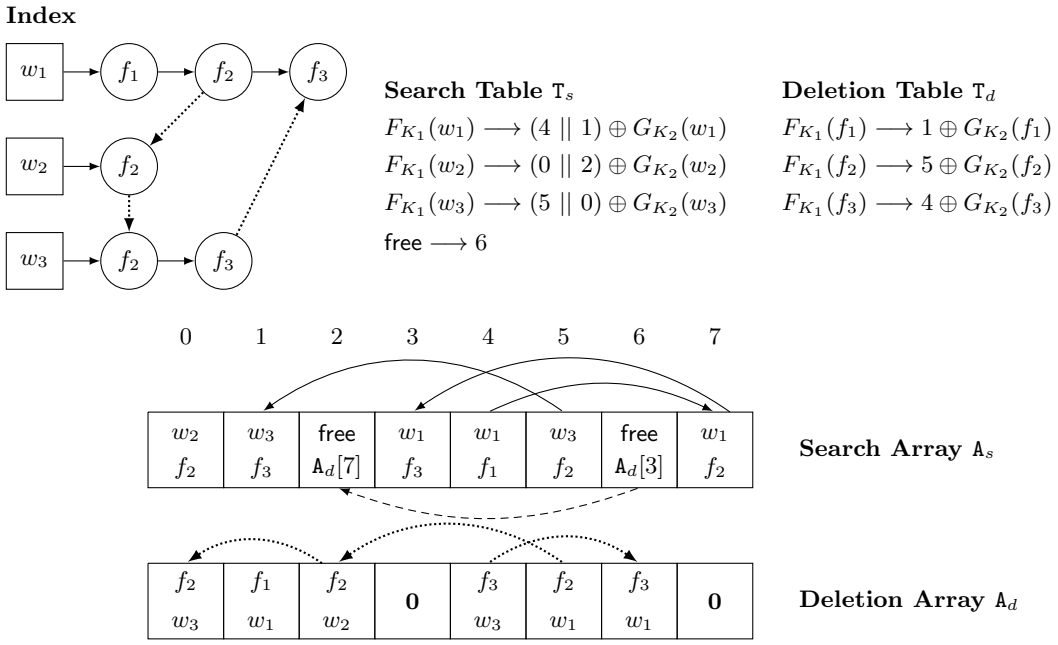

Figure 1: A small example of a dynamic encrypted index.

Real$_{\mathcal{A}}(k)$*: the challenger runs* $\mathsf{Gen}(1^k)$ *to generate a key* K*.* $\mathcal{A}$ *outputs* $\mathbf{f}$ *and receives* $(\gamma, \mathbf{c})$ *such that* $(\gamma, \mathbf{c}) \leftarrow \mathsf{Enc}_K(\mathbf{f})$ *from the challenger. The adversary makes a polynomial number of adaptive queries* $\{w, f_1, f_2\}$ *and, for each query* q*, receives from the challenger either a search token* τ_s *such that* $\tau_s \leftarrow \mathsf{SrchToken}_K(w)$*, an add token and ciphertext pair* (τ_a, c_{f_1}) *such that* $(\tau_a, c_{f_1}) \leftarrow \mathsf{AddToken}_K(f_1)$ *or a delete token* τ_d *such that* $\tau_d \leftarrow \mathsf{DelToken}_K(f_2)$*. Finally,* $\mathcal{A}$ *returns a bit* b *that is output by the experiment.*

Ideal$_{\mathcal{A},\mathcal{S}}(k)$*:* $\mathcal{A}$ *outputs* $\mathbf{f}$*. Given* $\mathcal{L}_1(\mathbf{f})$*,* $\mathcal{S}$ *generates and sends a pair* $(\gamma, \mathbf{c})$ *to* $\mathcal{A}$*. The adversary makes a polynomial number of adaptive queries* $q \in \{w, f_1, f_2\}$ *and, for each query* q*, the simulator is given either* $\mathcal{L}_2(\mathbf{f}, w)$*,* $\mathcal{L}_3(\mathbf{f}, f_1)$ *or* $\mathcal{L}_4(\mathbf{f}, f_2)$*. The simulator returns an appropriate token* τ *and, in the case of an add operation, a ciphertext* c*. Finally,* $\mathcal{A}$ *returns a bit* b *that is output by the experiment.*

We say that SSE *is* $(\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3, \mathcal{L}_4)$*-secure against adaptive dynamic chosen-keyword attacks if for all* PPT *adversaries* $\mathcal{A}$*, there exists a* PPT *simulator* $\mathcal{S}$ *such that*

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

Note that in addition to our inclusion of dynamic operations the differences between our definitions and the definitions of [8] are stylistic: we employ leakage functions in the style of [6] rather than the history in the style of [8].

4 Our Construction

Our scheme is an extension of the SSE-1 construction from [8, 9] which is based on the inverted index data structure. Though SSE-1 is practical (it is asymptotically optimal with small constants), it does

Let $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a private-key encryption scheme and $F : \{0,1\}^k \times \{0,1\}^* \to \{0,1\}^k$, $G : \{0,1\}^k \times \{0,1\}^* \to \{0,1\}^*$, and $P : \{0,1\}^k \times \{0,1\}^* \to \{0,1\}^k$ be pseudo-random functions. Let $H_1 : \{0,1\}^* \to \{0,1\}^*$ and $H_2 : \{0,1\}^* \to \{0,1\}^*$ be random oracles. Let $z \in \mathbb{N}$ be the initial size of the free list. Construct a dynamic SSE scheme $\mathsf{SSE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{SrchToken}, \mathsf{AddToken}, \mathsf{DelToken}, \mathsf{Search}, \mathsf{Add}, \mathsf{Del}, \mathsf{Dec})$ as follows:

- $\mathsf{Gen}(1^k)$: sample three k-bit strings K_1, K_2, K_3 uniformly at random and generate $K_4 \leftarrow \mathsf{SSE.Gen}(1^k)$. Output $K = (K_1, K_2, K_3, K_4)$.
- $\mathsf{Enc}(K, \mathbf{f})$:
  1. let $\mathtt{A}_s$ and $\mathtt{A}_d$ be arrays of size $|\mathbf{c}|/8 + z$ and let $\mathtt{T}_s$ and $\mathtt{T}_d$ be dictionary of size $\#W$ and $\#\mathbf{f}$, respectively. We assume $\mathbf{0}$ is a $(\log \#\mathtt{A}_s)$-length string of 0's and that free is a word not in W.
  2. for each word $w \in W$,[a]
     (a) create a list $\mathtt{L}_w$ of $\#\mathbf{f}_w$ nodes $(\mathtt{N}_1, \ldots, \mathtt{N}_{\#\mathbf{f}_w})$ stored at random locations in the search array $\mathtt{A}_s$ and defined as:
     $$\mathtt{N}_i := (\langle \mathsf{id}_i, \mathsf{addr}_s(\mathtt{N}_{i+1}) \rangle \oplus H_1(K_w, r_i), r_i)$$
     where id_i is the ID of the ith file in $\mathbf{f}_w$, r_i is a k-bit string generated uniformly at random, $K_w := P_{K_3}(w)$ and $\mathsf{addr}_s(\mathtt{N}_{\#\mathbf{f}_w+1}) = \mathbf{0}$
     (b) store a pointer to the first node of $\mathtt{L}_w$ in the search table by setting
     $$\mathtt{T}_s[F_{K_1}(w)] := \langle \mathsf{addr}_s(\mathtt{N}_1), \mathsf{addr}_d(\mathtt{N}_1^\star) \rangle \oplus G_{K_2}(w),$$
     where $\mathtt{N}^\star$ is the dual of $\mathtt{N}$, i.e., the node in $\mathtt{A}_d$ whose fourth entry points to $\mathtt{N}_1$ in $\mathtt{A}_s$.
  3. for each file f in $\mathbf{f}$,
     (a) create a list $\mathtt{L}_f$ of $\#\bar{f}$ dual nodes $(\mathtt{D}_1, \ldots, \mathtt{D}_{\#\bar{f}})$ stored at random locations in the deletion array $\mathtt{A}_d$ and defined as follows: each entry $\mathtt{D}_i$ is associated with a word w, and hence a node $\mathtt{N}$ in $\mathtt{L}_w$. Let $\mathtt{N}_{+1}$ be the node following $\mathtt{N}$ in $\mathtt{L}_w$, and $\mathtt{N}_{-1}$ the node previous to $\mathtt{N}$ in $\mathtt{L}_w$. Then, define $\mathtt{D}_i$ as follows:
     $$\mathtt{D}_i := \left(\left\langle \mathsf{addr}_d(\mathtt{D}_{i+1}), \mathsf{addr}_d(\mathtt{N}^\star_{-1}), \mathsf{addr}_d(\mathtt{N}^\star_{+1}), \mathsf{addr}_s(\mathtt{N}), \mathsf{addr}_s(\mathtt{N}_{-1}), \mathsf{addr}_s(\mathtt{N}_{+1}), F_{K_1}(w) \right\rangle \oplus H_2(K_f, r'_i), r'_i\right)$$
     where r'_i is a k-bit string generated uniformly at random, $K_f := P_{K_3}(f)$, and $\mathsf{addr}_d(\mathtt{D}_{\#\bar{f}+1}) = \mathbf{0}$.
     (b) store a pointer to the first node of $\mathtt{L}_f$ in the deletion table by setting:
     $$\mathtt{T}_d[F_{K_1}(f)] := \mathsf{addr}_d(\mathtt{D}_1) \oplus G_{K_2}(f)$$
  4. create an unencrypted free list $\mathtt{L}_{\mathsf{free}}$ by choosing z unused cells at random in $\mathtt{A}_s$ and in $\mathtt{A}_d$. Let $(\mathtt{F}_1, \ldots, \mathtt{F}_z)$ and $(\mathtt{F}'_1, \ldots, \mathtt{F}'_z)$ be the free nodes in $\mathtt{A}_s$ and $\mathtt{A}_d$, respectively. Set
     $$\mathtt{T}_s[\mathsf{free}] := \langle \mathsf{addr}_s(\mathtt{F}_z), \mathbf{0}^{\log \#\mathtt{A}} \rangle$$
     and for $z \geq i \geq 1$, set
     $$\mathtt{A}_s[\mathsf{addr}_s(\mathtt{F}_i)] := \mathbf{0}^{\log \#\mathbf{f}}, \mathsf{addr}_s(\mathtt{F}_{i-1}), \mathsf{addr}_d(\mathtt{F}'_i)$$
     where $\mathsf{addr}_s(\mathtt{F}_0) = \mathbf{0}^{\log \#\mathtt{A}}$.
  5. fill the remaining entries of $\mathtt{A}_s$ and $\mathtt{A}_d$ with random strings
  6. for $1 \leq i \leq \#\mathbf{f}$, let $c_i \leftarrow \mathsf{SKE.Enc}_{K_4}(f_i)$
  7. output $(\gamma, \mathbf{c})$, where $\gamma := (\mathtt{A}_s, \mathtt{T}_s, \mathtt{A}_d, \mathtt{T}_d)$ and $\mathbf{c} = (c_1, \ldots, c_{\#\mathbf{f}})$.

[a] Steps 2 and 3 here must be performed in an interleaved fashion to set up $\mathtt{A}_s$ and $\mathtt{A}_d$ at the same time.

Figure 2: A Fully Dynamic SSE scheme (Part 1).

- $\mathsf{SrchToken}(K, w)$: compute and output $\tau_s := \big(F_{K_1}(w), G_{K_2}(w), P_{K_3}(w)\big)$
- $\mathsf{Search}(\gamma, \mathbf{c}, \tau_s)$:
  1. parse τ_s as (τ_1, τ_2, τ_3) and return an empty list if τ_1 is not present in $\mathtt{T}_s$.
  2. recover a pointer to the first node of the list by computing $(\alpha_1, \alpha_1') := \mathtt{T}_s[\tau_1] \oplus \tau_2$
  3. look up $\mathtt{N}_1 := \mathtt{A}[\alpha_1]$ and decrypt with τ_3, i.e., parse $\mathtt{N}_1$ as (ν_1, r_1) and compute $(\mathsf{id}_1, \mathsf{addr}_s(\mathtt{N}_2)) := \nu_1 \oplus H_1(\tau_3, r_1)$
  4. for $i \geq 2$, decrypt node $\mathtt{N}_i$ as above until $\alpha_{i+1} = \mathbf{0}$
  5. let $I = \{\mathsf{id}_1, \ldots, \mathsf{id}_m\}$ be the file identifiers revealed in the previous steps and output $\{c_i\}_{i \in I}$, i.e., the encryptions of the files whose identifiers were revealed.
- $\mathsf{AddToken}(K, f)$: let $(w_1, \ldots, w_{\#\bar{f}})$ be the *unique* words in f in their order of appearance in f. Compute
$$\tau_a := (F_{K_1}(f), G_{K_2}(f), \lambda_1, \ldots, \lambda_{\#\bar{f}}),$$
where for all $1 \leq i \leq \#\bar{f}$:
$$\lambda_i := \big(F_{K_1}(w_i), G_{K_2}(w_i), \langle \mathsf{id}(f), \mathbf{0}\rangle \oplus H_1(P_{K_3}(w_i), r_i), r_i, \langle \mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0}, F_{K_1}(w_i)\rangle \oplus H_2(P_{K_3}(f), r_i'), r_i'\big),$$
and r_i and r_i' are random k-bit strings. Let $c_f \leftarrow \mathsf{SKE.Enc}_{K_4}(f)$ and output (τ_a, c_f).
- $\mathsf{Add}(\gamma, \mathbf{c}, \tau_a)$:
  1. parse τ_a as $(\tau_1, \tau_2, \lambda_1, \ldots, \lambda_{\#\bar{f}}, c)$ and return $\perp$ if τ_1 is not in $\mathtt{T}_d$.
  2. for $1 \leq i \leq \#\bar{f}$,
     (a) find the last free location φ in the search array and its corresponding entry $\varphi^\star$ in the deletion array by computing $(\varphi, \mathbf{0}) := \mathtt{T}_s[\mathsf{free}]$, and $(\varphi_{-1}, \varphi^\star) := \mathtt{A}_s[\varphi]$.
     (b) update the search table to point to the second to last free entry by setting $\mathtt{T}_s[\mathsf{free}] := (\varphi_{-1}, \mathbf{0})$
     (c) recover a pointer to the first node $\mathtt{N}_1$ of the list by computing $(\alpha_1, \alpha_1^\star) := \mathtt{T}_s[\lambda_i[1]] \oplus \lambda_i[2]$
     (d) store the new node at location φ and modify its forward pointer to $\mathtt{N}_1$ by setting $\mathtt{A}_s[\varphi] := \big(\lambda_i[3] \oplus \langle \mathbf{0}, \alpha_1\rangle, \lambda_i[4]\big)$
     (e) update the search table by setting $\mathtt{T}_s[\lambda_i[1]] := (\varphi, \varphi^\star) \oplus \lambda_i[2]$
     (f) update the dual of $\mathtt{N}_1$ by setting $\mathtt{A}_d[\alpha_1^\star] := \big(\mathtt{D}_1 \oplus \langle \mathbf{0}, \varphi^\star, \mathbf{0}, \mathbf{0}, \varphi, \mathbf{0}, \mathbf{0}\rangle, r\big)$, where $(\mathtt{D}_1, r) := \mathtt{A}_d[\alpha_1^\star]$
     (g) update the dual of $\mathtt{A}_s[\varphi]$ by setting $\mathtt{A}_d[\varphi^\star] := \big(\lambda_i[5] \oplus \langle \varphi_{-1}^\star, \mathbf{0}, \alpha_1^\star, \varphi, \mathbf{0}, \alpha_1, \lambda_i[1]\rangle, \lambda_i[6]\big)$,
     (h) if $i = 1$, update the deletion table by setting $\mathtt{T}_d[\tau_1] := \langle \varphi^\star, \mathbf{0}\rangle \oplus \tau_2$.
  3. update the ciphertexts by adding c to $\mathbf{c}$

Figure 3: A Fully Dynamic SSE scheme (Part 2).

- $\mathsf{DelToken}(K, f)$: output: $\tau_d := (F_{K_1}(f),\, G_{K_2}(f),\, P_{K_3}(f),\, \mathsf{id}(f))$.
- $\mathsf{Del}(\gamma, \mathbf{c}, \tau_d)$:
  1. parse τ_d as $(\tau_1, \tau_2, \tau_3, \mathsf{id})$ and return $\perp$ if τ_1 is not in $\mathtt{T}_d$
  2. find the first node of $\mathtt{L}_f$ by computing $\alpha'_1 := \mathtt{T}_d[\tau_1] \oplus \tau_2$
  3. for $1 \leq i \leq \#\bar{f}$,
     (a) decrypt $\mathtt{D}_i$ by computing $(\alpha_1, \ldots, \alpha_6, \mu) := \mathtt{D}_i \oplus H_2(\tau_3, r)$, where $(\mathtt{D}_i, r) := \mathtt{A}_d[\alpha'_i]$
     (b) delete $\mathtt{D}_i$ by setting $\mathtt{A}_d[\alpha'_i]$ to a random $(6 \log \#\mathtt{A} + k)$-bit string
     (c) find address of last free node by computing $(\varphi, \mathbf{0}^{\log \#\mathtt{A}}) := \mathtt{T}_s[\mathsf{free}]$
     (d) make the free entry in the search table point to $\mathtt{D}_i$'s dual by setting $\mathtt{T}_s[\mathsf{free}] := \langle \alpha_4, \mathbf{0}^{\log \#\mathtt{A}} \rangle$
     (e) free location of $\mathtt{D}_i$'s dual by setting $\mathtt{A}_s[\alpha_4] := (\varphi, \alpha'_i)$
     (f) let $\mathtt{N}_{-1}$ be the node that precedes $\mathtt{D}_i$'s dual. Update $\mathtt{N}_{-1}$'s "next pointer" by setting: $\mathtt{A}_s[\alpha_5] := (\beta_1, \beta_2 \oplus \alpha_4 \oplus \alpha_6, r_{-1})$, where $(\beta_1, \beta_2, r_{-1}) := \mathtt{A}_s[\alpha_5]$. Also, update the pointers of $\mathtt{N}_{-1}$'s dual by setting
     $$\mathtt{A}_d[\alpha_2] := (\beta_1, \beta_2, \beta_3 \oplus \alpha'_i \oplus \alpha_3, \beta_4, \beta_5, \beta_6 \oplus \alpha_4 \oplus \alpha_6, \mu*, r^*_{-1}),$$
     where $(\beta_1, \ldots, \beta_6, \mu^*, r^*_{-1}) := \mathtt{A}_d[\alpha_2]$
     (g) let $\mathtt{N}_{+1}$ be the node that follows $\mathtt{D}_i$'s dual. Update $\mathtt{N}_{+1}$'s dual pointers by setting:
     $$\mathtt{A}_d[\alpha_3] := (\beta_1, \beta_2 \oplus \alpha'_i \oplus \alpha_2, \beta_3, \beta_4, \beta_5 \oplus \alpha_4 \oplus \alpha_5, \beta_6, \mu^*, r^*_{+1}),$$
     where $(\beta_1, \ldots, \beta_6, \mu^*, r^*_{+1}) := \mathtt{A}_d[\alpha_3]$
     (h) set $\alpha'_{i+1} := \alpha_1$
  4. remove the ciphertext that corresponds to id from $\mathbf{c}$
  5. remove τ_1 from $\mathtt{T}_d$
- $\mathsf{Dec}(K, c)$: output $m := \mathsf{SKE.Dec}_{K_4}(c)$.

Figure 4: A Fully Dynamic SSE scheme (Part 3).

have limitations that make it unsuitable for direct use in cryptographic cloud storage systems: (1) it is only secure against non-adaptive chosen-keyword attacks (CKA1) which, intuitively, means that it can only provide security for clients that perform searches in a batch; and (2) it is not explicitly dynamic, i.e., it can only support dynamic operations using general and inefficient techniques.

Before discussing how we address these issues, we first recall a variant of the SSE-1 construction at a high level. The construction is essentially the same as SSE-1 except that the lookup table managed by indirect addressing is replaced with a dictionary.[3] The scheme makes use of a private-key encryption scheme $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$, two pseudo-random functions F and G, an array $\mathtt{A}_s$ we refer to as the *search array* and a dictionary $\mathtt{T}_s$ we refer to as the *search table*. Here we assume SKE is anonymous in the sense that, given two ciphertexts, one cannot determine whether they were encrypted under the same key.[4]

The SSE-1 construction. To encrypt a collection of files $\mathbf{f}$, the scheme constructs for each keyword $w \in W$ a list $\mathtt{L}_w$. Each list $\mathtt{L}_w$ is composed of $\#\mathbf{f}_w$ nodes $(\mathtt{N}_1, \ldots, \mathtt{N}_{\#\mathbf{f}_w})$ that are stored at random locations in the search array $\mathtt{A}_s$. The node $\mathtt{N}_i$ is defined as $\mathtt{N}_i = \langle \mathsf{id}, \mathsf{addr}_s(\mathtt{N}_{i+1}) \rangle$, where id is the unique

[3] This was first used in [6] to avoid the use of FKS dictionaries [14].

[4] This allows us to encrypt each node of a list using a single key as opposed to using unique keys as done in [8].

file identifier of a file that contains w and $\mathsf{addr}_s(\mathtt{N})$ denotes the location of node $\mathtt{N}$ in $\mathtt{A}_s$. To prevent the size of $\mathtt{A}_s$ from revealing statistical information about $\mathbf{f}$, it is recommended that $\mathtt{A}_s$ be of size at least $|\mathbf{c}|/8$ and the unused cells be padded with random strings of appropriate length.

For each keyword w, a pointer to the head of $\mathtt{L}_w$ is then inserted into the search table $\mathtt{T}_s$ under search key $F_{K_1}(w)$, where K_1 is the key to the PRF F. Each list is then encrypted using SKE under a key generated as $G_{K_2}(w)$, where K_2 is the key to the PRF G.

To search for a keyword w, it suffices for the client to send the values $F_{K_1}(w)$ and $G_{K_2}(w)$. The server can then use $F_{K_1}(w)$ with $\mathtt{T}_s$ to recover the pointer to the head of $\mathtt{L}_w$, and use $G_{K_2}(w)$ to decrypt the list and recover the identifiers of the files that contain w. As long as $\mathtt{T}$ supports $O(1)$ lookups (which can be achieved using a hash table), the total search time for the server is linear in $\#\mathbf{f}_w$, which is optimal.

Making SSE-1 dynamic. As mentioned above, the limitations of $\mathsf{SSE\text{-}1}$ are twofold: (1) it is only CKA1-secure and (2) it is not explicitly dynamic. As observed in [6], the first limitation can be addressed relatively easily by requiring that SKE be non-committing (in fact the CKA2-secure SSE construction proposed in that work uses a simple PRF-based non-committing encryption scheme).

The second limitation, however, is less straightforward to overcome. The difficulty is that the addition, deletion or modification of a file requires the server to add, delete or modify nodes in the encrypted lists stored in $\mathtt{A}_s$. This is difficult for the server to do since: (1) upon deletion of a file f, it does not know where (in $\mathtt{A}$) the nodes corresponding to f are stored; (2) upon insertion or deletion of a node from a list, it cannot modify the pointer of the previous node since it is encrypted; and (3) upon addition of a node, it does not know which locations in $\mathtt{A}_s$ are free.

At a high level, we address these limitations as follows:

1. (file deletion) we add an extra (encrypted) data structure $\mathtt{A}_d$ called the *deletion array* that the server can query (with a token provided by the client) to recover pointers to the nodes that correspond to the file being deleted. More precisely, the deletion array stores for each file f a list $\mathtt{L}_f$ of nodes that point to the nodes in $\mathtt{A}_s$ that should be deleted if file f is ever removed. So every node in the search array has a corresponding node in the deletion array and every node in the deletion array points to a node in the search array. Throughout, we will refer to such nodes as *duals* and write $\mathtt{N}^\star$ to refer to the dual of a node $\mathtt{N}$.

2. (pointer modification) we encrypt the pointers stored in a node with a homomorphic encryption scheme. This is similar to the approach used by van Liesdonk et al in [23] to modify the encrypted search structure they construct. By providing the server with an encryption of an appropriate value, it can then modify the pointer without ever having to decrypt the node. We use the "standard" private-key encryption scheme which consists of XORing the message with the output of a PRF. This simple construction also has the advantage of being non-committing (in the private-key setting) which we make use of to achieve CKA2-security.

3. (memory management) to keep track of which locations in $\mathtt{A}_s$ are free we add and manage extra space comprising a *free list* that the server uses to add new nodes.

Our construction is described in detail in Figures 2, 3 and 4. Figure 1, which we discuss in the next Section, illustrates the dynamic SSE data structures over a toy index containing 3 files and 3 unique words.

On our use of random oracles. As observed in [8], one of the main difficulties in designing CKA2-secure SSE schemes is that the keywords can be chosen as a function of the encrypted index and of

previous search results. This makes proving security difficult because the simulator has to be able to simulate an encrypted index *before* it receives any search results. [8] showed how to overcome this obstacle and later [6] gave a more efficient approach based on a simple private-key non-committing encryption scheme. At a high level, both works construct schemes that allow for equivocation, that is, the simulator can generate a "fake" encrypted index and later, when given a search result, can generate an appropriate token (i.e., a token that when used with the fake index will yield the correct search outcome). Unfortunately, the techniques from [8] and [6] do not work in our setting. The main problem is that in the dynamic setting there are situations where the previously described level of equivocation is not enough.

In particular, consider an adversary that first searches for a keyword w, then adds a file that contains w and, finally, searches for w again. To see why the previous level of equivocation does not suffice, notice that after the first search the simulator is committed to a token for w. Now, after the adversary adds a file with w, the simulator needs to simulate an add token for that file. The simulator, however, does not know what the file is or even that it contains w so it cannot produce a token that functions properly, i.e., the add token it simulates cannot make any meaningful change to the encrypted index. The problem is that after the adversary performs the second search for w, he expects this new search to reveal at least one new result compared to the previous one. In particular, the search should now also reveal the new file's identifier. But if the add token cannot properly modify the encrypted index in the second stage and if the simulator cannot send a new token during the third stage (since it is committed) then how can the simulator guarantee that the adversary will get an updated search result?

We overcome this by constructing a scheme that allows the simulator to modify the outcome of the search during the adversary's *execution* of the search algorithm. Note that this is a departure from the approaches of [8] and [6] which manipulate the outcome of the adversary's search by creating specially designed tokens. We do this by making use of the random oracle model. At a very high level, we design our encrypted index in a way that requires the adversary to query a random oracle during various steps of the search algorithm. The simulator is then able to program the responses of the random oracle in a way that suits it and can make sure that the execution of the search yields the outcome it wants.

4.1 An Illustrative Example

In Figure 1, we show the data structures of our fully-dynamic SSE scheme for a specific index. The index is built on three documents, namely f_1, f_2, f_3 over three keywords, namely w_1, w_2, w_3. All the documents contain keyword w_1, keyword w_2 is only contained in document f_2 and w_3 is contained in documents f_2 and f_3. The respective search table $\mathtt{T}_s$, the deletion table $\mathtt{T}_d$, the search array $\mathtt{A}_s$ and the deletion array $\mathtt{A}_d$ are also shown in Figure 1. Note that in a real DSSE index, there would be padding to hide the number of file-word pairs; we omit padding for simplicity in this example.

Searching. Searching is the simplest operation in our scheme. Suppose the client wishes to search for all the documents that contain keyword w_1. He prepares the search token, which among others contains $F_{K_1}(w_1)$ and $G_{K_2}(w_1)$. The first value $F_{K_1}(w_1)$ will enable the server to locate the entry corresponding to keyword w_1 in the search table $\mathtt{T}_s$. In our example, this value is $x = (4||1) \oplus G_{K_2}(w_1)$. The server now uses the second value $G_{K_2}(w_1)$ to compute $x \oplus G_{K_2}(w_1)$. This will allow the server to locate the right entry (4 in our example) in the search array and begin "unmasking" the locations storing pointers to the documents containing w_1. This unmasking is performed by means of the third value contained in the search token.

Adding a document. Suppose now the client wishes to add a document f_4 containing keywords w_1 and w_2. Note that the search table does not change at all since f_4 is going to be the last entry in

the list of keywords w_1 and w_2 and the search table only stores the first entries. However all the other data structures must be updated in the following way. First the server uses free to quickly retrieve the indices of the "free" positions in the search array $\mathtt{A}_s$, where the new entries are going to be stored. In our example these positions are 2 and 6. The server stores in these entries the new information (w_1, f_4) and (w_2, f_4). Now the server needs to connect this new entries to the respective keywords lists: using the add token, it retrieves the indices $i = 0$ and $j = 3$ in the search array $\mathtt{A}_s$ of the elements x and y such that x and y correspond to the last entries of the keyword lists w_1 and w_2. In this way the server homomorphically sets $\mathtt{A}_s[0]$'s and $\mathtt{A}_s[3]$'s "next" pointers to point to the newly added nodes, already stored in the search array at positions 2 and 6.

Note that getting access to the free entries in the search array also provides access to the respective free positions of the deletion array $\mathtt{A}_d$. In our example, the indices of the free positions in the deletion array are 3 and 7. The server will store the new entries (f_4, w_1) and (f_4, w_2) at these positions in the deletion array and will also connect them with pointers. Finally, the server will update the deletion table by setting the entry $F_{K_1}(f_4)$ to point to position 3 in the deletion array, so that file f_4 could be easily retrieved for deletion later.

Deleting a document. Suppose now the client wants to delete a document already stored in our index, say document f_3, containing keywords w_1 and w_3. The deletion is a "dual operation" to addition. First the server uses the value $F_{K_1}(f_3)$ of the deletion token to locate the right value $4 \oplus G_{K_2}(f_3)$ in the deletion table. This will allow the server to get access to the portion of the remaining data structures that need to be updated in a similar fashion with the addition algorithm. Namely it will "free" the positions 4 and 6 in the deletion array and positions 1 and 3 in the search array. While "freeing" the positions in the search array, it will also homomorphically update the pointers of previous entries in the keyword list w_1 and w_3 to point to the new entries (in our example, to the end of the lists—generally in the next pointers of the deleted items). Note that no such an update of pointers is required for the deletion array.

5 Security

As discussed in §3, all practical SSE schemes leak some information. Unfortunately, the extent to which the practical security of SSE is affected by this leakage is not well understood and depends greatly on the setting in which SSE is used. We are aware of only one concrete attack [18] that exploits this leakage and it depends strongly on knowledge of previous queries and statistics about the file collection. We note, however, that our scheme leaks *more* than most previously-known constructions since it is dynamic and there are correlations between the information leaked by its various operations. In the following, we provide a framework for describing and comparing the leakage of SSE schemes. Based on this framework, we compare the leakage of our scheme with the leakage of SSE-1, which is static; and the leakage of the scheme proposed in [23], which is dynamic.

A framework for characterizing leakage. Our approach is to describe leakage in terms of a database containing two tables over word and file identifiers: SSE operations write anonymized rows to tables in the database, and an adversary tries to de-anonymize the resulting data. Columns in the tables contain identifiers for files and words: each file is represented by exactly one identifier, and each word is represented by exactly one identifier, but these identifiers are chosen at random with respect to files and words. For ease of exposition, we will assume that there is a function id that produce identifiers for files and words.

Our scheme writes file-word information into two tables:

1. the File-Word table (FW), in which each row associates a word identifier with a file identifier.

2. the Adjacency (Adj) table, in which each row associates a word identifier and a file identifier with a direction "next" or "prev" and one of the following values: (1) another file identifier, or (2) a value $\perp$.

Each row also contains a timestamp of the time it was written; for simplicity in notation, we do not write the timestamps in the following description. Intuitively, the FW table records associations between file and word identifiers; the Adj table records adjacency information about files in the lists for given words.

Note that our concrete construction has two different identifiers for a given file: the ciphertext for a file is stored under a file pointer that is revealed during the search operation, but the file information in the index is stored in $\mathtt{T}_d$ under the output of a pseudo-random function on the file. However, the scheme's operations immediately reveal to an adversary the correlation between these values, so in our leakage description, we do not distinguish between these two types of file identifiers. Operations in our construction write the following values:

Search takes as input the identifier for a word w and returns a set of file identifiers to the client. So, for each file f returned by search, our scheme writes the row $(\mathsf{id}(w), \mathsf{id}(f))$ to the FW table. Note that the server then writes one row for each file-word pair.

Add takes as input an add token that contains the identifier for a file f and adds word information for a set of words associated with this file. Like Search, Add writes tuples $(\mathsf{id}(w), \mathsf{id}(f))$ for each word w associated with the file in the Add Token.

Additionally, however, the Add operation reveals to the server whether or not f is the only file that contains w. If the server has previously executed an operation that revealed the file f' associated with the head of the list for w, then Add writes the tuple $(\mathsf{id}(w), \mathsf{id}(f)$, "next", $\mathsf{id}(f'))$ to the Adj table. In either case, Add writes the tuple $(\mathsf{id}(w), \mathsf{id}(f)$, "prev", $\perp)$ to the Adj table to indicate that f is the head node for the list for w. And if the word is not yet in the index, then Add writes $(\mathsf{id}(w), \mathsf{id}(f)$, "next", $\perp)$ to the Adj table.

Delete takes as input a delete token that contains an identifier for a file f; this token does not contain any word-specific information. However, in the process of executing the Delete operation, the server uncovers in the index a word-identifier (the search key for $\mathtt{T}_s$) for each word associated with the file. So, like Search and Add, Delete writes tuples $(\mathsf{id}(w), \mathsf{id}(f))$ for each word w associated with f.

As each word w is deleted for f, it reveals the location of its neighbors in the search array. And for the purposes of our proof, we say that the leakage in this case consists of the file identifiers for the previous and next nodes in the list for w. Let the files associated with these nodes be f' and f'', respectively. The server then writes $(\mathsf{id}(w), \mathsf{id}(f)$, "prev", $\mathsf{id}(f'))$ and $(\mathsf{id}(w), \mathsf{id}(f)$, "next", $\mathsf{id}(f''))$ to the Adj table (in each case, it writes $\perp$ if there is no previous or next neighbor).

We can use this framework to compare the leakage of our scheme to the leakage of previous schemes. SSE-1 does not provide Add or Delete operations, but it writes the same values as our scheme writes to the FW table in Search. The other table (and the additional writes to FW in Add and Delete) makes up the extra leakage from our scheme.

The scheme of Sedghi et al. [23] is dynamic but leaks less information than our construction. Search in their scheme reveals the word that is searched but does not reveal the identifiers of the files returned, since that information is masked in an array of bits, with one bit per file. This can be represented by writing a word identifier to the FW table with the value $\perp$ for the file identifier. Their scheme performs the same writes to the FW table for Add and Delete, since each Add or Delete operation reveals an identifier for the file and reveals the word identifiers by the locations that it modifies in their index. However, their scheme never writes adjacency information to the Adj table: it hides all

adjacency information at the cost of requiring per-word storage and communication complexity that is linear in the maximum number of files that can be stored in their index.

Theorem and proof. Before stating our security Theorem, we provide a more formal and concise description of our scheme's leakage:

- the $\mathcal{L}_1$ leakage is defined as

$$\mathcal{L}_1(\mathbf{f}) = \big(\#\mathtt{A}_s, \big[\mathsf{id}(w)\big]_{w\in W}, \big[\mathsf{id}(f)\big]_{f\in\mathbf{f}}, \big[|f|\big]_{f\in\mathbf{f}}\big),$$

  where id is the identifier function described above.

- the $\mathcal{L}_2$ leakage is defined as:

$$\mathcal{L}_2(\mathbf{f}, w) = \big(\textsc{accp}_t(w), \mathsf{id}(w)\big),$$

  where $\textsc{accp}_t(w)$ is the access pattern which itself is defined as the sequence $(\mathsf{id}_1, \ldots, \mathsf{id}_{\#\mathbf{f}_w})$.

- the $\mathcal{L}_3$ leakage is defined as:

$$\mathcal{L}_3(\mathbf{f}, f) = \big(\mathsf{id}(f), \big[\mathsf{id}(w), \mathsf{apprs}(w)\big]_{w\in\bar{f}}, |f|\big),$$

  where $\mathsf{apprs}(w_i)$ is a bit set to 1 if w is appears in at least one file in $\mathbf{f}$ and to 0 otherwise.

- the $\mathcal{L}_4$ leakage is defined as:

$$\mathcal{L}_4(\mathbf{f}, f) = \big(\mathsf{id}(f), \big[\mathsf{id}(w), \mathsf{prev}(f, w), \mathsf{next}(f, w)\big]_{w\in\bar{f}}\big),$$

  where $\mathsf{prev}(f, w)$ and $\mathsf{next}(f, w)$ are the identities of the first files before and after f (in the natural ordering of files) that contain w. If there are no files before and after f that contain the word then $\mathsf{prev}(f, w)$ and $\mathsf{next}(f, w)$ return $\perp$, respectively. Here we assume the identifier/pointer triples are ordered according to the order in which the words appear in f.

In the following Theorem we show that our construction is CKA2-secure in the random oracle model with respect to the leakage described above.

Theorem 5.1. *If* SKE *is CPA-secure and if* F*,* G *and* P *are pseudo-random, then* SSE *as described above is* $(\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3, \mathcal{L}_4)$*-secure against adaptive chosen-keyword attacks in the random oracle model.*

At a very high level, the proof of security for our construction works as follows. The simulator $\mathcal{S}$ generates a simulated encrypted index $\widetilde{\gamma}$ and a simulated sequence of ciphertexts $\widetilde{\mathbf{c}}$ using the information it receives from $\mathcal{L}_1$, which includes the number of elements in the search array, the number of files, the number of keywords and the length of each file. The simulated index $\widetilde{\gamma}$ can be constructed similarly to a real encrypted index, except that encryptions are replaced by encryptions of the zero string (of appropriate length) and the output of the PRFs are replaced by random values. The CPA-security of the encryption schemes and the pseudo-randomness of the PRFs will guarantee that the resulting $\widetilde{\gamma}$ is indistinguishable from a real encrypted index. The simulated file encryptions $\widetilde{\mathbf{c}}$ are simulated in the same manner (i.e., replacing the ciphertexts by encryptions of the all zero string) and the CPA-security of the encryption scheme guarantees indistinguishability.

Simulating search, add and delete tokens is more complex and requires the simulator to keep track of various dependencies between the information revealed by these operations. This is because the tokens the simulator creates must all be consistent with each other, otherwise the simulation may be detected by the adversary. For this, our proof utilizes a non-trivial set of techniques so that the simulator can keep track of dependencies.

Proof. We describe a polynomial-time simulator $\mathcal{S}$ such that for all probabilistic polynomial-time adversaries $\mathcal{A}$, the outputs of $\mathbf{Real}_{\mathcal{A}}(k)$ and $\mathbf{Ideal}_{\mathcal{S}}(k)$ are computationally indistinguishable. Consider the simulator $\mathcal{S}$ that adaptively simulates an encrypted index $\widetilde{\gamma} = (\widetilde{\mathtt{A}}_s, \widetilde{\mathtt{T}}_s, \widetilde{\mathtt{A}}_d, \widetilde{\mathtt{T}}_d)$, a sequence of simulated ciphertexts $\widetilde{\mathbf{c}}$ and $n \in \mathbb{N}$ simulated tokens $(\widetilde{\tau}_1, \ldots, \widetilde{\tau}_n)$ as follows:

- (Setting up internal data structures) given

$$\mathcal{L}_1(\delta, \mathbf{f}) = \Big(\#\mathtt{A}_s, \big[\mathsf{id}(w)\big]_{w \in W}, \big[\mathsf{id}(f)\big]_{f \in \mathbf{f}}, \big[|f|\big]_{f \in \mathbf{f}}\Big),$$

  it generates $K_5 \leftarrow \mathsf{SKE.Gen}(1^k)$. Let $\mathtt{iA}_s$, $\widetilde{\mathtt{A}}'_s$, $\mathtt{iA}_d$ and $\widetilde{\mathtt{A}}'_d$ be empty arrays each of size $|\mathbf{c}|/8 + z$ and let $\mathtt{iT}_s$ and $\widetilde{\mathtt{T}}'_s$ be dictionaries of size $\#W + 1$, and $\mathtt{iT}_d$ and $\widetilde{\mathtt{T}}'_d$ be dictionaries of size and $\#\mathbf{f}$. Set $\mathtt{iT}_s[\mathsf{free}] := \perp$, $\mathtt{iT}_s[\mathsf{id}(w_i)] := \perp$ for all $i \in [\#W]$. For all $j \in [\#\mathbf{f}]$, set $\mathtt{iT}_d[\mathsf{id}(f_j)] := \perp$. Let $\mathtt{RO}_1$ and $\mathtt{RO}_2$ be empty dictionaries. Let γ_s be a bijection mapping search keys in $\mathtt{iT}_s$ to search keys in $\widetilde{\mathtt{T}}'_s$ and let γ_d be a bijection mapping search keys in $\mathtt{iT}_d$ to search keys in $\widetilde{\mathtt{T}}'_d$.

  For all $i \in [\#W]$, the simulator choose a k-bit key $K_{\mathsf{id}(w_i)}$ uniformly at random that will be associated with identifier $\mathsf{id}(w_i)$. Similarly, for all $i \in [\#\mathbf{f}]$, it chooses a k-bit key $K_{\mathsf{id}(f_i)}$ uniformly at random that will be associated with $\mathsf{id}(f_i)$.

- (Simulating $\mathtt{A}_s$) it generates an array $\widetilde{\mathtt{A}}_s$ of size $|\mathbf{c}|/8 + z$ and fills $|\mathbf{c}|/8$ of these cells (chosen at random) with random strings of the form $\langle \widetilde{\mathtt{N}}, \widetilde{r} \rangle$ such that $|\widetilde{\mathtt{N}}| = \log \#\mathtt{A}_s + \log n$ and $|\widetilde{r}| = k$. It generates a copy $\widetilde{\mathtt{A}}'_s$ of $\widetilde{\mathtt{A}}_s$. It then marks all the empty cells in $\mathtt{A}_s$ as free in $\mathtt{iA}_s$. In other words, if the ℓth cell in $\mathtt{A}_s$ is empty, then the ℓth cell in $\mathtt{iA}_s$ is marked as free.

- (Simulating $\mathtt{T}_s$) it generates a dictionary $\widetilde{\mathtt{T}}_s$ of size $\#W + 1$ and for all $i \in [\#W + 1]$, stores a random $(2 \log \#\mathtt{A}_s)$-bit string $\widetilde{v}$ in $\widetilde{\mathtt{T}}_s$ under a random k-bit search key $\widetilde{\sigma}$. In addition, it generates a copy $\widetilde{\mathtt{T}}'_s$ of $\widetilde{\mathtt{T}}_s$.

- (Simulating $\mathtt{A}_d$) it generates an array $\widetilde{\mathtt{A}}_d$ of size $|\mathbf{c}|/8 + z$ and fills $|\mathbf{c}|/8$ of these cells (chosen at random) with random strings of the form $\langle \widetilde{\mathtt{D}}, \widetilde{r} \rangle$ such that $|\widetilde{\mathtt{D}}| = 6 \log \#\mathtt{A}_d + k$ and $|\widetilde{r}| = k$. In addition, it generates a copy $\widetilde{\mathtt{A}}'_d$ of $\widetilde{\mathtt{A}}_d$. As above, it then marks all the empty cells in $\mathtt{A}_d$ as free in $\mathtt{iA}_d$. It then generates a map

$$\delta : \{1, \ldots, |\mathbf{c}|/8 + z\} \cup \perp \to \{1, \ldots, |\mathbf{c}|/8 + z\} \cup \perp$$

  which is a bijection mapping locations in $\mathtt{iA}_s$ to locations in $\mathtt{iA}_d$ such that $\delta(\perp) = \perp$ and such that free cells in $\mathtt{iA}_s$ are mapped to free cells in $\mathtt{iA}_d$.

- (Simulating $\mathtt{T}_d$) it generates a dictionary $\widetilde{\mathtt{T}}_d$ of size $\#\mathbf{f}$ and, for all $i \in [n]$, stores a random $(\log \#\mathtt{A}_d)$-bit string $\widetilde{v}$ in $\widetilde{\mathtt{T}}_d$ under a random k-bit search key $\widetilde{\sigma}$. In addition, it generates a copy $\widetilde{\mathtt{T}}'_d$ of $\widetilde{\mathtt{T}}_d$.

- (Simulating ciphertexts) for all $i \in [\#\mathbf{f}]$, let

$$\widetilde{c}_i \leftarrow \mathsf{SKE.Enc}_{K_5}(0^{|f_i|}).$$

- (Simulating search tokens) given

$$\mathcal{L}_2(\delta, \mathbf{f}, w) = \Big(\textsc{accp}_t(w), \mathsf{id}(w)\Big),$$

where $\text{ACCP}_t(w) = (\mathsf{id}_1, \ldots, \mathsf{id}_{\#\mathbf{f}_w})$, the simulator works as follows.

There are two cases to consider: either (1) the word has never appeared before, in which case $\mathsf{id}(w)$ has never appeared in any previous leakage; or (2) the word has either been searched for or is contained in a file that has been added or deleted in the past, in which case $\mathsf{id}(w)$ has appeared in previous leakage. The simulator first checks if $\mathsf{id}(w)$ has appeared in previous leakage.

Case 1. If $\mathsf{id}(w)$ has not appeared, it will update its $\mathtt{iT}_s$ and $\mathtt{iA}_s$ structures to point to and hold, respectively, a new list for $\mathsf{id}(w)$. To do this, it chooses $\#\text{ACCP}_t(w)$ unused and non-free cells in $\mathtt{iA}_s$ at random and marks them with $\mathsf{id}(w)$. It then creates a list by storing in each of these cells a file identifier id_i from $\text{ACCP}_t(w)$ and a pointer to the cell that holds id_{i+1}. If $i = \#\mathbf{f}_w$, it stores the pointer $\perp$. Let α_1 be the address of the head of this list, i.e., of the cell that holds id_1. It then makes $\mathtt{iT}_s$ point to this list by setting $\mathtt{iT}_s[\mathsf{id}(w)] := \langle \alpha_1, \delta(\alpha_1) \rangle$.

If $\#\text{ACCP}_t(w) = 0$, then no list is created and α_1 is just set to $\perp$.

Case 2. If, on the other hand, $\mathsf{id}(w)$ has appeared, then $\mathtt{iT}_s[\mathsf{id}(w)]$ is either $\langle \perp, \perp \rangle$ or $\langle \alpha_1, \delta(\alpha_1) \rangle$, where α_1 is a location in $\mathtt{iA}_s$ that holds the head of a list of nodes marked with $\mathsf{id}(w)$. The first case could occur, e.g., if the word has been searched for in the past but is not in any file.

If $\mathtt{iT}_s[\mathsf{id}(w)] = \langle \alpha_1, \delta(\alpha_1) \rangle \neq \langle \perp, \perp \rangle$, the simulator searches $\mathtt{iA}_s$ for the cells marked with $\mathsf{id}(w)$. As stated above, these cells form a list. The simulator then augments this list to length $\#\text{ACCP}_t(w)$ by choosing unused and non-free cells in $\mathtt{iA}_s$ at random and, as above, storing the appropriate file identifiers and pointers. While augmenting the list, respect the existing head and tail (if any) and mark each of the new cells with $\mathsf{id}(w)$.

If, on the other hand, $\mathtt{iT}_s[\mathsf{id}(w)] = \langle \perp, \perp \rangle$, it just sets $\alpha_1 = \perp$.

It then returns the token

$$\tau_s = \left(\gamma_s(\mathsf{id}(w)), \widetilde{\mathtt{T}}'_s[\gamma_s(\mathsf{id}(w))] \oplus \langle \alpha_1, \delta(\alpha_1) \rangle, K_{\mathsf{id}(w)} \right).$$

- (Simulating add tokens) given

$$\mathcal{L}_3(\delta, \mathbf{f}, f) = \left(\mathsf{id}(f), \left[\mathsf{id}(w_i), \mathsf{apprs}(w_i) \right]_{i \in \#\bar{f}}, |f| \right),$$

the simulator works as follows.

If $\mathsf{id}(f) \in \mathtt{iT}_d$, the file has already been added in the past (and not deleted since) so it just returns the token τ_a that was previously returned.

From now on we assume the file is new. First, it chooses a random k-bit string $K_{\mathsf{id}(f)}$ which it will associate with $\mathsf{id}(f)$ (this will be used for answering random oracle queries).

Before returning a token, the simulator must check if its internal data structures are properly setup. For this, it considers the following three cases for all $i \in [\#\bar{f}]$:

1. if $\mathtt{iT}_s[\mathsf{id}(w_i)] \neq \langle \perp, \perp \rangle$, its internal data structures are correctly set up and $\mathtt{iT}_s[\mathsf{id}(w_i)]$ points to a list in $\mathtt{iA}_s$.
2. if $\mathtt{iT}_s[\mathsf{id}(w_i)] = \langle \perp, \perp \rangle$ and $\mathsf{apprs}(w_i) = 0$, its internal data structures are correctly set up since the word does not appear in the file collection and $\mathtt{iT}_s[\mathsf{id}(w_i)]$ correctly points to $\perp$.

3. if $\mathtt{iT}_s[\mathsf{id}(w_i)] = \langle \perp, \perp \rangle$ and $\mathsf{apprs}(w_i) = 1$, its internal data structures are not properly set up since the word does appear in the file collection and $\mathtt{iT}_s[\mathsf{id}(w_i)]$ points to $\perp$. To address this, the simulator sets

$$\mathtt{iT}_s[\mathsf{id}(w_i)] = \langle \alpha_1, \delta(\alpha_1) \rangle,$$

where α_1 is a randomly chosen unused and non-free cell location in $\mathtt{iA}_s$. It then stores $\mathsf{id}(f)$ in the cell at location α_1 and marks it with $\mathsf{id}(w_i)$.

It then returns the token

$$\tau_a = \Big(\gamma_d\big(\mathsf{id}(f)\big), v, \lambda_1, \ldots, \lambda_{\#f} \Big),$$

where $v := \mathtt{G}[\mathsf{id}(f)]$ if $\mathsf{id}(f) \in \mathtt{G}$ and v is a $(\log \#\mathtt{A}_d)$-bit random string if $\mathsf{id}(f) \notin \mathtt{G}$ (in which case v is stored in $\mathtt{G}$ under search key $\mathsf{id}(f)$); and where for all $i \in [\#f]$,

$$\lambda_i = \Big(\gamma_s(\mathsf{id}(w_i)), \widetilde{\mathtt{T}}'_s[\gamma_s(\mathsf{id}(w_i))] \oplus \mathtt{iT}_s[\mathsf{id}(w_i)], u_i, r_i, u'_i, r'_i \Big),$$

where u_i and u'_i are, respectively, $(\log \#\mathbf{f} + \log \#\mathtt{A}_s)$-bit and $(6 \log \#\mathtt{A}_d + k)$-bit strings chosen uniformly at random and where r_i and r'_i are random k-bit strings.

After returning the token, the simulator updates its internal structures as follows. For all $i \in [\#\bar{f}]$,

1. let α_1 be the first element of $\mathtt{iT}_s[\mathsf{id}(w_i)]$, i.e., $\mathtt{iT}_s[\mathsf{id}(w_i)] = \langle \alpha_1, \delta(\alpha_1) \rangle$. Let φ and φ_- be the locations of the last and second-to-last nodes in the free list of $\mathtt{iA}_s$,
2. it sets $\widetilde{\mathtt{A}}'_s[\varphi] := (u_i \oplus \langle \mathbf{0}, \alpha_1 \rangle, r_i)$ and

$$\widetilde{\mathtt{A}}'_d[\delta(\varphi)] := (u'_i \oplus \langle \delta(\varphi_{-1}), \mathbf{0}, \delta(\alpha_1), \varphi, \mathbf{0}, \alpha_1, \gamma_s(\mathsf{id}(w_i)) \rangle, r'_i),$$

3. it makes the node at location φ in $\mathtt{iA}_s$ point to α_1 (note that if $\alpha_1 = \perp$, then the node at φ points to $\perp$ which means it becomes the head and tail of the list),
4. it make $\mathtt{iT}_s$ point to φ by setting $\mathtt{iT}_s[\mathsf{id}(w_i)] := \langle \varphi, \delta(\varphi) \rangle$,
5. it marks the node at φ in $\mathtt{iA}_s$ with $\mathsf{id}(w_i)$.

Finally, it creates a dual list for $\mathsf{id}(f)$ in $\mathtt{iA}_d$ by: (1) finding the duals of the cells used above (this can be done using δ); and (2) storing a dual node with the appropriate information in these cells. Here, the nodes in the list are ordered according to the order of the word identifiers in the leakage. Let h be the head of this list. It sets $\widetilde{\mathtt{T}}'_d[\mathsf{id}(f)] := \mathtt{G}[\mathsf{id}(f)] \oplus h$.

- (Simulating delete tokens) given

$$\mathcal{L}_4(\delta, \mathbf{f}, f) = \Big(\mathsf{id}(f), \big[\mathsf{id}(w_i), \mathsf{prev}(f, w_i), \mathsf{next}(f, w_i)\big]_{i \in [\#\bar{f}]} \Big)$$

the simulator works as follows.

For all $i \in [\#\bar{f}]$,

1. the simulator searches in $\mathtt{iA}_s$ for a cell marked with $\mathsf{id}(w_i)$ that has a dual in $\mathtt{iA}_d$ marked with $\mathsf{id}(f)$,
2. if no such cell exists, it chooses an unused and non-free cell in $\mathtt{iA}_s$ at random, marks it with $\mathsf{id}(w_i)$ and mark its dual in $\mathtt{iA}_d$ with $\mathsf{id}(f)$.

It now creates a dual list for $\mathsf{id}(f)$ by merging the duals of the nodes found (or created) in steps 1 and 2 above into a list (in $\mathtt{iA}_d$). Here, the order of the nodes follow the order of the word identifiers provided in the leakage. In other words, the first node of the list is the dual of the node (in $\mathtt{iA}_s$) marked with $\mathsf{id}(w_1)$. The second node of the list is the dual of node (in $\mathtt{iA}_s$) marked with $\mathsf{id}(w_2)$, and so on. It then sets $\mathtt{iT}_d[\mathsf{id}(f)]$ to be the head of the dual list just created.

After updating $\mathtt{iA}_d$, it now updates $\mathtt{iA}_s$ by merging the newly created nodes into the appropriate lists. More precisely, for all $i \in [\#\bar{f}]$, it merges all the nodes in $\mathtt{iA}_s$ marked with $\mathsf{id}(w_i)$ (note that some of these nodes could have been added to $\mathtt{iA}_s$ due to previous queries) into a list, making sure to respect the adjacency information provided by the $\mathsf{next}(f, w_i)$ and $\mathsf{prev}(f, w_i)$ leakage.

It now returns the token

$$\tau_d = \left(\gamma_d(\mathsf{id}(f)), \widetilde{\mathtt{T}}'_d[\gamma_d(\mathsf{id}(f))] \oplus \mathtt{iT}_d[\mathsf{id}(f)], K_{\mathsf{id}(f)}, \mathsf{id}(f)\right).$$

and sets

$$\mathtt{G}[\mathsf{id}(f)] := \mathtt{T}_d[\gamma_d(\mathsf{id}(f))] \oplus \mathtt{iT}_d[\mathsf{id}(f)]$$

so as to remain consistent with future add token simulations.

After returning the token, the simulator updates its data structures by freeing the cells corresponding to the deleted file. More specifically, for all $i \in [\#\bar{f}]$, it frees the cell in $\mathtt{iA}_s$ marked with $\mathsf{id}(w_i)$ that has a dual marked with $\mathsf{id}(f)$. It also frees its dual. When freeing a cell, it always update its neighbors to point to each other. If the freed node was the head of a list, then it updates the relevant pointer in $\mathtt{iT}_s$ to point to that node's neighbor.

It then removes the search key $\mathsf{id}(f)$ from $\mathtt{iT}_d$ along with its value and merges the newly freed nodes in $\mathtt{iA}_s$ and $\mathtt{iA}_d$ into the free list (here the nodes in $\mathtt{iA}_s$ are added to the free list in the order that their corresponding word identifiers appeared in the leakage).

- (Answering H_1 queries) given query (K, r), the simulator checks if K has been associated with some word identifier $\mathsf{id}(w)$, i.e., if $K = K_{\mathsf{id}(w)}$ for some $\mathsf{id}(w)$. If not, it returns a random $(\log \#f + \log \#\mathtt{A})$-bit string v and sets $\mathtt{RO}_1[\langle K, r\rangle] := v$ so as to stay consistent on future queries.

  If so, it finds all entries in $\mathtt{iA}_s$ marked with $\mathsf{id}(w)$ and checks to see if any of their corresponding cells in $\widetilde{\mathtt{A}}'_s$ store the randomness r. If not, it returns and stores in $\mathtt{RO}_1$ a random value v as above.

  If such a cell does exist, the simulator returns

  $$v \oplus \mathtt{iA}_s[\ell].$$

  where ℓ is the location in $\widetilde{\mathtt{A}}'_s$ of that cell and v is such that $(v, r) := \widetilde{\mathtt{A}}'_s[\ell]$.

- (Answering H_2 queries) given query (K, r), it checks if K has been associated with some file identifier $\mathsf{id}(f)$. If so, it finds all entries in $\mathtt{iA}_d$ marked with $\mathsf{id}(f)$ and checks to see if any of their corresponding cells in $\widetilde{\mathtt{A}}'_d$ store r. If either step fails, it returns a random $(6 \log \#\mathtt{A}_d + k)$-bit value v and sets $\mathtt{RO}_2[\langle K, r\rangle] := v$ in order to stay consistent.

  If, on the other hand, such a file identifier $\mathsf{id}(f)$ and cell in $\widetilde{\mathtt{A}}'_d$ are found, then it returns

  $$v \oplus \mathtt{iA}_d[\ell].$$

  where v is such that $(v, r) := \widetilde{\mathtt{A}}'_d[\ell]$ and ℓ is the location in $\widetilde{\mathtt{A}}'_d$ of the cell with randomness r.

$\widetilde{\mathtt{A}}_s$ and $\widetilde{\mathtt{A}}_d$ are distributed identically to $\mathtt{A}_s$ and $\mathtt{A}_d$. The indistinguishability of $\widetilde{\mathtt{T}}_s$ and $\widetilde{\mathtt{T}}_d$ from $\mathtt{T}_s$ and $\mathtt{T}_d$, respectively, follows from the pseudo-randomness of G. The indistinguishability of $\widetilde{\tau}_s$ follows from the pseudo-randomness of F, G and P and that of $\widetilde{\tau}_a$ and $\widetilde{\tau}_d$ from the pseudo-randomness of F, G and P and the CPA-security of SKE. Finally, the indistinguishability of $\widetilde{\mathbf{c}}$ follows from the CPA-security of SKE. □

6 Performance

6.1 Implementation

To demonstrate the feasibility of our algorithms, we implemented SSE in C++ over the Microsoft Cryptography API: Next Generation (CNG) [7]. Our implementation uses the algorithms described in §4. The cryptographic primitives for our protocol use CNG. Encryption is the CNG implementation of 128-bit AES-CBC [13], and the hash function is the CNG implementation of SHA-256 [12]. SSE employs two random oracles, which are implemented using HMAC-SHA256 from CNG (this employs the HMAC construction first described by Bellare, Canetti, and Krawczyk [3]). The first parameter passed to the random oracle is used as a key to the HMAC, and the second parameter is used as input to the HMAC.[5]

A system that implements SSE performs two classes of time-intensive operations: cryptographic computations and systems actions (e.g., network transmission and filesystem access). To separate the costs of cryptography from the systems costs (which will vary between underlying systems), we built a test framework that performs cryptographic computations on a set of files but does not transfer these files across a network or incur the costs of storing and retrieving index information from disk; all operations are performed in memory. We also ignore the cost of producing a plain-text index for the files, since the choice and implementation of an indexing algorithm is orthogonal to SSE.

6.2 Experiments

Cryptographic operations in SSE require widely varying amounts of time to execute. So, to evaluate SSE, we performed micro-benchmarks and full performance tests on the system and broke each test out into its component algorithms. The micro-benchmarks are used to explain the performance of the full system.

These experiments were performed on an Intel Xeon CPU 2.26 GHz (L5520) running Windows Server 2008 R2. All experiments ran single-threaded on the processors. Each data point presented in the experiments is the mean of 10 executions, and error bars provide the sample standard deviation.

The unit of measurement in all of the microbenchmarks is the *file/word pair*: for a given file f the set of file/word pairs is comprised of all unique pairs (f, w) such that w is a word associated with f in the index. The set of all such tuples across all files in a file collection is exactly the set of entries in a keyword index for this collection.

We chose three sets of real-world data for our experiments. The first set was selected from the Enron emails [11]; we extracted a subset of emails and used decreasing subsets of this original subset as file collections with different numbers of file/word pairs. The second set consisted of Microsoft Office documents (using the Word, PowerPoint, and Excel file types) used by a business group in Microsoft for its internal planning and development. In a similar fashion to the emails, we chose decreasing subsets of this collection as smaller file collections. The third data set consists of media files, which

[5] Recent work by Dodis, Ristenpart, Steinberger, and Tessaro [10] shows that HMAC is indifferentiable from a random oracle when the key used has length shorter than $d - 1$, where d is the block length of the underlying hash function. Our keys are 32-bytes in length and satisfy the theorem.

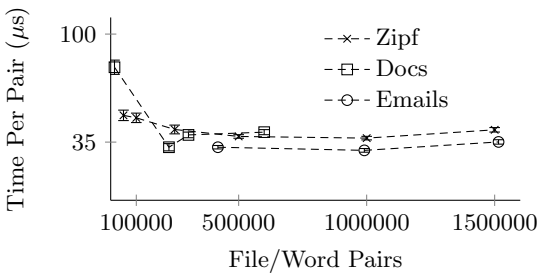

Figure 5: SSE.Enc.

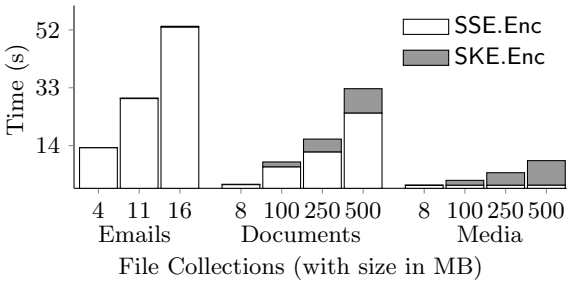

Figure 6: SSE.Enc and SKE.Enc.

have almost no indexable words but have large file size. This collection is composed of MP3, video, WMA, and JPG files that make data sets of the same sizes as the ones in the document collection. To index the emails, documents and media, we used an indexer that employs IFilter plugins in Windows to extract unique words from each file. The indexer also extracts properties of the files from the NTFS filesystem, such as the author of a Microsoft Word document, or the artist or genre of an MP3 file.

6.2.1 Micro-benchmarks

To determine the performance of SSE, we generated synthetic indexes and executed search and update operations on them. For searches, we chose the word that was present in the most files. And we deleted and added back in a file with the largest number of unique words in the index. We only compared against the email and document data sets for our micro-benchmarks, since the media data set index size was too small for useful comparisons.

We generated our synthetic indexes from a pair of Zipf distributions [24] with parameter $\alpha = 1.1$; one distribution contained randomly-generated files, and the other contained words (the words in our case were simply numbers represented as strings: "0", "1", "2", etc.). The synthetic file collection was generated as follows. First, the test code drew a file f from the Zipf file distribution (our sampling employed the algorithm ZRI from Hörman and Derflinger [17]). Second, the test code drew words from the word distribution until it found a word that was not in the index for f. It then added this word to the index information for f and drew another file to repeat the process until a given number of file/word pairs was generated. This process corresponds to writing a set of files with Zipf-distributed sizes and containing Zipf-distributed words such that the file collection as a whole contains a given number of file/word pairs.

Figure 5 shows the costs of index generation incurred by SSE, expressed as the cost per file/word pair; these are the timings for the operations that are performed after a collection of files is indexed (for the total time required to index these collections, see the results of Figure 6). The numbers of pairs range from about 14,000 to about 1,500,000 in number. The synthetic data is labeled with "Zipf", the Enron data is labeled with "Email", and the document data is labeled with "Docs". The cost per file/word pair is an amortized value: it was determined by taking the complete execution time of each experiment and dividing by the number of file/word pairs.

The cost per file/word pair in Figure 5 is small: it decreases to about 35 μs per pair. Lower numbers of pairs lead to higher per-pair costs, since there is a constant overhead for adding new words and new files to the index, and the cost is not amortized over as many pairs in this case.

The email and document data validate our synthetic model and correspond closely to this model (within 10%) for data points with approximately the same number of file/word pairs. This suggests that, at least for large numbers of pairs, the Zipf model leads to the same SSE performance as the English text as contained in the emails and documents. The synthetic data tests the sensitivity of the

operation	time	stddev
SSE.Search	7.3	0.6
SSE.AddToken	37	2
SSE.DelToken	3.0	0.2
SSE.Add	1.6	0.4
SSE.Del	24	1

Table 2: Execution time (in μs) per unit (word or file) for SSE operations.

SSE algorithms to details of the file/word distribution; experiments over the file collections are limited to always operating over the same assignment of unique words to files, but different experiments over the synthetic data contain different sets of file/word pairs, albeit drawn from the same distribution. Since our synthetic results match closely our results from real-world data sets, this sensitivity is low, as would be expected.

Micro-benchmark execution time for SSE algorithms does not depend on the number of file/word pairs in the index. And the cost per unique word is essentially independent (modulo a very small constant cost) of the total number of unique words (or files) in each operation. So, we present only the per-word (or per-file) time for these operations. Table 2 shows the costs for each operation. For ease of exposition, we show numbers only for the executions of the SSE algorithm on the document data set; the numbers for the email data set and the synthetic data are similar. Search token generation takes a constant amount of time (a mean of 35 μs), irrespective of the number of files that will be returned from the search. The results show that search and file addition and deletion on the client side are efficient and practical, even for common words, or files containing many unique words.

6.2.2 Full performance

To evaluate the performance of SSE as a whole, we ran the SSE algorithms specified in §4 on the email, document and media data sets. Note that all algorithms displayed on the graphs have non-zero cost, but in some cases, the cost is so small compared to the cost of other parts of the operation that this cost cannot be seen on the graph.

Figure 6 shows the results of the encryption operation, which takes the most time of any of the algorithms. Note that the entire encryption protocol is performed in addition to indexing that must be executed by the client before the data can be stored.

Figure 6 shows the difference between the email data and the document data. The Enron emails are a collection of plain text files, including email headers, so almost every byte of every file is part of a word that will be indexed. So, each small file contains many words, and the ratio of file/word pairs to the size of the data set is high. By contrast, Microsoft Office documents may contain significant formatting and visual components (like images) which are not indexed. So, the ratio of file/word pairs to file size is much lower. Both data sets represent a common case for office use: our results show that SSE index generation requires significantly more time for large text collections than for the common office document formats. Finally, the ratio of indexable words to file size is almost zero for the media files.

The micro-benchmark results of Figure 5 show that SSE index generation performance is linear in the number of file/word pairs for large data sets. So, for an email data set of size 16 GB (consisting entirely of text-based emails: i.e., emails containing no attachments), the initial indexing costs would be approximately 15 hours (which could be performed over the course of a day during the idle time of the computer). After this initial indexing, adding and removing emails would be fast.

To evaluate the costs of the remaining SSE algorithms, we performed experiments that gave upper

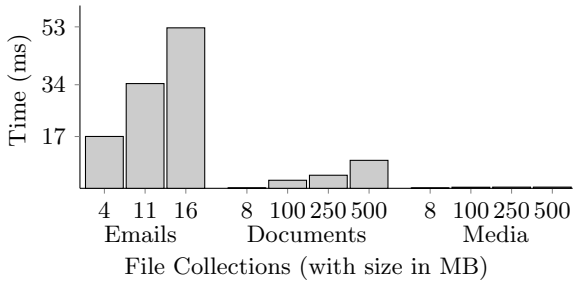

Figure 7: Execution time for SSE.Search.

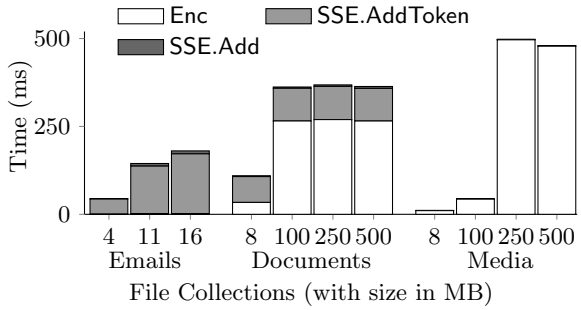

Figure 8: Execution time for adding a file.

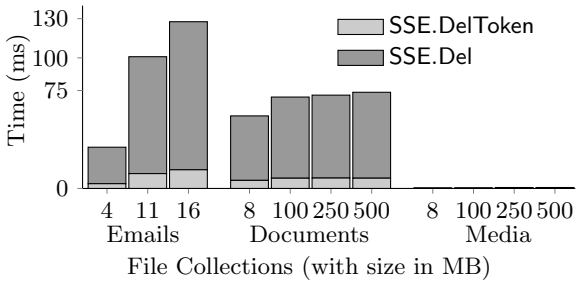

Figure 9: Execution time for deleting a file.

bounds on the cost of any operation. An upper bound for SSE.Search is a search for the word contained in the most files. Our update operations use the file with the most bytes on disk.

Since search was performed for the word that was indexed for the most files, the total time needed for the search depended on the prevalence of words in files: media files had few words, even in 500 MB of content, whereas some words occur in every email. Figure 7 gives the time needed for the server to perform a search, given a search token (we neglect the cost of generating a search token, since it is a small constant in microseconds). The SSE search costs were small, even for the email index. However, even the longest searches took only about 50 ms to complete. And for large media collections, the search time was negligibly small.[6]

Figure 8 shows the execution time for adding a file. The cost of the operation is divided into several components: "Enc" refers to the time needed to encrypt the new file, "SSE.AddToken" refers to client generation of the add token for the words being indexed in the file, and "SSE.Add" refers to the server using the add token to update the index. The costs of adding a file fall mostly on the client: the dominant costs are SSE add token generation and file encryption, both performed on the client. In a use case where add operations dominate (such as indexing encrypted emails), this allows the server to support many clients easily, since the client that performs the add also performs most of the computation.

A similar situation occurs in Figure 9 for deleting a file. The label "SSE.DelToken" refers to client generation of the delete token, and "SSE.Del" refers to the server using the delete token to update the index. As for add, the delete operation is efficient and practical; each operation on the largest files

[6]Note that in our workloads, the time to decrypt all files returned from search dominates the search costs by orders of magnitude. Higher-level protocols could mitigate this cost by using the SSE primitive in a different manner: instead of storing the files directly, it could store short, fixed-length descriptions of the files. The client could decrypt these results quickly then use their information to decide which files to download and decrypt. This would also allow clients to delete a file without downloading the file from the server.

took approximately one tenth of a second.

7 Conclusion

Searchable encryption is an important cryptographic primitive that is well motivated by the popularity of cloud storage services like Dropbox, Microsoft SkyDrive and Apple iCloud and public cloud storage infrastructures like Amazon S3 and Microsoft Azure Storage. Any practical SSE scheme, however, should satisfy certain properties such as sublinear (and preferably optimal) search, adaptive security, compactness and the ability to support addition and deletion of files.

In this work, we gave the first SSE construction to achieve all these properties. In addition, we implemented our scheme and evaluated its performance. Our experiments show that our construction is highly efficient and ready for deployment.

Acknowledgements

The authors are grateful to Jason Mackay for writing the indexer that was used in the experiments. The second author was partially supported by the Kanellakis fellowship at Brown University and by Intel's STC for Secure Computing.

References

[1] G. Amanatidis, A. Boldyreva, and A. O'Neill. Provably-secure schemes for basic query support in outsourced databases. In *Proc. Working Conference on Data and Applications Security* (DBSEC), pages 14–30, 2007.

[2] M. Bellare, A. Boldyreva, and A. O'Neill. Deterministic and efficiently searchable encryption. *Proc. Int. Cryptology Conference* (CRYPTO), pages 535–552, 2007.

[3] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. *Proc. Int. Cryptology Conference* (CRYPTO), pages 1–15, 1996.

[4] D. Boneh, G. Di Crescenzo, R. Ostrovsky, G. Persiano. Public key encryption with keyword search. *Proc. Int. Conference on the Theory and Applications of Cryptographic Techniques* (EUROCRYPT), pages 506–522, 2004.

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

[6] M. Chase and S. Kamara. Structured encryption and controlled disclosure. In *Proc. Int. Conference on the Theory and Application of Cryptology and Information Security* (ASIACRYPT), pages 577–594, 2010.

[7] Cryptography API: Next generation (Windows). http://msdn.microsoft.com/library/aa376210.aspx.

[8] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: Improved definitions and efficient constructions. In *Proc. ACM Conference on Computer and Communications Security* (CCS), pages 79–88, 2006.

[9] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: Improved definitions and efficient constructions. *Journal of Computer Security*, 19(5):895–934, 2011.

[10] Y. Dodis, T. Ristenpart, J. Steinberger, and S. Tessaro. To hash or not to hash again? (In)differentiability results for H^2 and HMAC. *Proc. Int. Cryptology Conference* (CRYPTO), 2012.

[11] Enron email dataset. http://www.cs.cmu.edu/~enron/, 2009.

[12] FIPS 180-3. Secure Hash Standard (SHS). Federal Information Processing Standard (FIPS), Publication 180-3, National Institute of Standards and Technology, Washington, DC, October 2008.

[13] FIPS 197. Advanced Encryption Standard (AES). Federal Information Processing Standard (FIPS), Publication 197, National Institute of Standards and Technology, Washington, DC, November 2001.

[14] M. Fredman, J. Komlos, and E. Szemeredi. Storing a sparse table with O(1) worst case access time. *Journal of the ACM*, 31(3):538–544, 1984.

[15] E.-J. Goh. Secure indexes. Technical Report 2003/216, IACR ePrint Cryptography Archive, 2003. http://eprint.iacr.org/2003/216.

[16] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996.

[17] W. Hörmann and G. Derflinger. Rejection-inversion to generate variates from monotone discrete distributions. *ACM Transactions on Modeling and Computer Simulation*, 6(3):169–184, 1996.

[18] M. Islam, M. Kuzu and M. Kantarcioglu. Access Pattern Disclosure on Searchable Encryption: Ramification, Attack and Mitigation. *Network and Distributed System Security Symposium (NDSS '12)*, 2012.

[19] S. Kamara and K. Lauter. Cryptographic cloud storage. In *Proc. Workshop Real-Life Cryptographic Protocols and Standardization* (RLCPS), pages 136–149, 2010.

[20] J. Katz and Y. Lindell. *Introduction to Modern Cryptography.* Chapman & Hall/CRC, Boca Raton, FL, 2008.

[21] K. Kurosawa and Y. Ohtaki. UC-secure searchable symmetric encryption. In *Proc. Financial Cryptography and Data Security* (FC), 2012.

[22] D. Song, D. Wagner, and A. Perrig. Practical techniques for searching on encrypted data. In *Proc. Symposium on Research in Security and Privacy* (SSP), pages 44–55, 2000.

[23] P. van Liesdonk, S. Sedghi, J. Doumen, P. H. Hartel, and W. Jonker. Computationally efficient searchable symmetric encryption. In *Proc. Workshop on Secure Data Management* (SDM), pages 87–100, 2010.

[24] G. K. Zipf. *Psycho-Biology of Languages.* Houghton-Mifflin, Boston, 1935.